IN 102 - Cours 2

ENSTA ParisTech

1 Les fonctions

Abstraction d'un calcul vis-à-vis de paramètres (variables).

Une fonction est définie sur un domaine A et à valeurs dans B :
⇒ elle prend un ou plusieurs arguments,
⇒ elle retourne UNE valeur.

Quatre éléments de finition, dans l'ordre :

1. le type de la valeur qu'elle retourne,
2. son nom,
3. les types des paramètres qu'elle attend,
4. et son corps.

```c
#include <stdio.h>

int add (int i, int j)
{
  return (i + j) ;   // Corps: retourne bien un int.
}

int main ()
{
  int n = 5 ;

  printf ("%d    %d\n", add (n, 1), add (2, add (n, 3))) ;
  return (0) ;
}
```

Paramètre formel : nom donné dans la définition de la fonction.

Paramètre effectif : expression donnée à un paramètre formel lors de l'appel.

2 Appel de fonction

Écrire une fonction ne l'appelle pas ! Pour qu'elle « *travaille* » . . . il faut l'appeler.

La fonction `main` est la seule exception : c'est le point d'entrée du programme et elle est « *automagiquement* » appelée.

Principe d'un appel de fonction, dans l'ordre :

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.

3 Déclaration/prototype

Même principe que pour les variables : les fonctions doivent être déclarées.

Primordial si votre fonction est appelée dans un autre fichier source.

Prototype : type de retour + nom + types des arguments + **;**
int add (**int**, **int**) ;

Un prototype de fonction se met dans un fichier `.h` qui est **#include**-é par les fichiers source utilisant (appelant) cette fonction.

4 Déroulement d'appel

La transmission des arguments et le retour de valeur se font via la pile d'exécution.

Pour chaque paramètre, on transmet toujours, lors de l'appel, une copie de la valeur calculée.

Lors de l'appel, les variables locales d'une fonction sont créées. Elles disparaissent à la fin de l'appel.

Si une fonction est déclarée comme retournant une valeur d'un type t, elle doit obligatoirement finir toutes ses exécutions par un **return** d'une valeur de type t.

```
float add (float i, float j)
{
  return (i + j) ;    // Correct: retourne bien un float.
}
```

```
int is_zero (int i)
{
  if (i == 0) return (1) ;
  // Incorrect: si i different de 0, ne retourne de defini.
}
```

```
float one_instead_of_zero (int i)
{
  if (i == 0) return (1.0) ;
  else return (i) // Incorrect: i est un int, pas un float.
}
```

Certaines fonctions « *n'ont rien (de pertinent) à retourner* » : elles sont de type de retour **void**. Leur **return** est optionnel en fin de corps.

```
void print_hello ()
{
  printf ("Hello\n") ;
  return ;  // Optionnel car on est a la fin du corps.
}
```

5 Fonction récursive

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

L'idée est de diviser un problème P en sous-problèmes (certains de la même forme), de résoudre ces sous-problèmes de manière directe si c'est possible sinon de la même manière que P (le problème initial), puis de combiner le résultat de ces sous-problèmes pour résoudre le problème initial P.

Autrement dit, on traite le problème P en le ramenant à un problème de la même forme mais plus petit ...

Il faut nécessairement un (des) cas d'arrêt sinon la fonction ne terminerait pas.

Une fonction peut avoir plusieurs appels récursifs.

Le déroulement des appel se passe exactement comme pour les fonctions non récursives.

On empile les arguments, l'adresse de retour, etc . . . On appelle la fonction. Lorsqu'elle a terminé d'exécuter son corps (qui procédera de la même façon pour les appels récursifs) elle transmet son résultat et on revient dans la fonction appelante.

5.1 La somme des n premiers entiers

```
unsigned int sum (unsigned int n)
{
  if (n == 0) return (0) ;
  else return (n + sum (n - 1)) ;
}
```

La somme des entiers de n à 0 est égale à $n+$ la somme des entiers de $n-1$ à 0. Donc on divise le problème en « *la valeur* n », un sous-problème de même forme sur $n-1$, on combine les solutions par une addition et on s'arrête si $n = 0$.

5.2 Fonctions mutuellement récursives

Des fonctions peuvent s'appeler mutuellement : elles sont mutuellement récursives.

```
#include <stdbool.h>

bool natural_even (unsigned int n)
{
  if (n == 0) return (true) ;
  else return (natural_odd (n - 1)) ;
}

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

Un entier n est pair s'il est égal à 0 ou si $n-1$ est impair. Si un entier n est égal à 0 alors il n'est pas impair, sinon il est impair si $n-1$ est pair. Dans les deux cas, on s'arrête si $n = 0$. Ici l'opération de combinaison est triviale (rien à faire) puisque le résultat de l'appel récursif est directement le résultat du problème.

6 Entrée au clavier

La fonction `scanf` (duale de `printf`) permet de lire une chaîne au clavier, de l'interpréter en fonction de "%♪" et de stocker la valeur découlant dans une variable.

```
#include <stdio.h>

int main ()
{
  int i ;
  float j ;
  scanf ("%d %f", &i, &j) ;
  printf ("i = %d, j = %f\n", i, j) ;
  return (0) ;
}
```

À chaque "%♪" doit correspondre un argument passé à `scanf` : en première approximation, un nom de variable précédé par &.

Nombreux "%♪" dont : %d pour lire une *valeur entière signée*, %f pour une valeur flottante.

Nécessite l'utilisation de `stdio.h` : **#include** <stdio.h>