Programmation Procédurale – Fonctions – Variables

Polytech'Nice Sophia Antipolis

Erick Gallesio

2015 – 2016

Définition de fonction K&R (1 / 2)

En C traditionnel, les fonctions sont déclarées avec la forme suivante:

```
type_résultat nom(parametres)
     types des paramètres
{
    corps de la fonction
}
```

Exemple:

```
int max(a, b)
    int a, b;
{
    return (a>b) ? a : b;
}
```

Si les paramètres sont omis dans la liste, ils sont considérés comme de type `int`

Définition de fonction K&R (2 / 2)

Lorsqu'une fonction est déclarée sous la forme K&R

- Pas de vérification du type des paramètres !!!
- Pas de vérification du nombre de paramètres !!!

```
    #include <stdio.h>
    int max(a, b)
         int a, b;
    {
        return (a>b) ? a : b;
    }

    int main() {
      printf("max(3,6) = %d\n", max(3., 6));    /* un "." qui traîne */
      printf("max(5,1) = %d\n", max(5.1));      /* un "." au lieu d'une "," */
      return 0;
    }
==> max(3,6) = 1074266112
    max(5,1) = 1717986918
```

Pas d'erreur ni de warning $\Rightarrow$ **Ne jamais utiliser la forme K&R!!!**

Définition de fonction ANSI (1 / 2)

La norme ANSI redéfinit la façon de définir une fonction avec des *prototypes*:

```
type_résultat nom(liste typée de parametres)
{
    corps de la fonction
}
```

Exemple:

```
int max(int a, int b)
{
    return (a>b) ? a : b;
}
```

- Pas de type implicite

Définition de fonction ANSI (2 / 2)

- **Type du résultat**
  - `void` (pas de type ⇒ procédure)
  - n'importe quel type scalaire (entier, réel, pointeur, enum)
  - structure et union
  - **Attention**: pas tableau
- **Type des paramètres**
  - n'importe quel type scalaire (entier, réel, pointeur, enum)
  - structure et union
  - tableau
    - paramètre formel et paramètre effectif doivent être compatibles (cf pointeurs)
    - La taille peut être spécifiée (ou non)

```
int strlen(char str[MAX]):
int strlen(char str[]);
```

Appel de fonction: K&R vs ANSI

```
void f(a, b)                    /* Version K&R */
    int a; double b;
{
    ...
}

void g(int a, double b) { /* Version ANSI */
    ...
}

f();                            /* pas d'erreur */
f(1, 2, 3, 4);                  /* pas d'erreur */
f("false", "even more");        /* pas d'erreur */
f(1, 2);                        /* problème potentiel */

g();                            /* erreur détectée */
g(1, 2, 3, 4);                  /* erreur détectée */
g("false", "even more");        /* erreur détectée */
g1, 2);                         /* 2 converti en double */
```

Résultat de fonction

- La valeur de la fonction est donnée par l'énoncé **return**
- Type de résultat: celui donné à la définition de la fonction
- Conversion éventuelle de la valeur du **return** vers le type de la fonction
- Si la fonction est de type **void**, pas de valeur après le **return**

Passage de paramètre

- Un seul mode: **passage par valeur**
- Pour les tableaux, on passe un pointeur sur le début du tableau (par valeur)

```
    void f(int x) {
        x = x + 1;
        printf("In function f: %d\n", x);
    }

    void main(void) {
        int a = 1;
        f(a);
        printf("After call to f: %d\n", a);
    }
==> In function f: 2
    After call to f: 1
```

- L'ordre d'évaluation n'est pas garanti

```
    i=5
    f(i++, t[i])          /* t[5] ou t[6] ?????*/
```

Déclaration de fonction (1 / 2)

- Déclarer une fonction = donner son type avec un *header* ou *prototype*
- C'est utile:
  - si la fonction est utilise "en avant"
  - si la fonction est définie dans un autre fichier
- Définition K&R

```
double cos();          /* paramètres absents (inutiles) */
```

- Définition ANSI

```
double cos(double x); /* en-tête complet */
```

- Si utilisation de la fonction sans *prototype*
  - *auto-déclaration* de la fonction par le compilateur
  - résultat de type entier
  - pas de contôle du nombre et du type des paramètres

Déclaration de fonction (2 / 2)

Même en ANSI C, la rétro compatibilité avec le C de K&R peut être source d'erreurs

```
void f2(int a, int b);  /* forward declaration */

void f1(int a, int b)
{
    int x, y, z;
    x = f2(b, a);       /* error detected   */
    y = f3(b, a);       /* auto declaration */
    z = f4(b, a);       /* auto declaration */
}

void f2(float a, float b)   /* error detected */
{ ... }

int f3(void)        /* conform to autodeclaration !!!! */
{ ... }

double f4(int a, int b) /* error detected, even in K&R */
{ ... }
```

Fonctions à arité variable (1 / 2)

C'est une extension ANSI

- La liste de paramètres variable est dénotée par '...' après le dernier paramètre fixe
- Il doit y avoir au moins un paramètre fixe
- Le fichier `<stdarg.h>` définit les macros suivantes:
  - `va_start(va_list ap, last_fixed_parameter)`
  - `va_arg(va_list ap, type)`
  - `va_end(va_list ap)`

Fonctions à arité variable (2 / 2)

Exemple

```c
#include <stdarg.h>

int max(int first, ...)  {/* Liste terminée par un nombre < 0 */
    va_list ap;
    int M = 0;

    va_start(ap, first);
    while (first > 0) {
        if (first > M) M = first;
        first = va_arg(ap, int);
    }
    va_end(ap);
    return M;
}

void main(void) {
    int x = max(12, 18, 17, 20, 1, 34, 5, -1);
    ...
}
```

Variables (1 / 3)

On ne considère ici que les programmes mono fichier

Variable globale

- *Définition*: en dehors d'un bloc
- *Durée de vie*: tout le programme
- *Visibilité*: depuis son point de définition jusqu'à la fin de fichier (avec possibilité de masquage dans un bloc)

Variable locale

- *Définition*: dans un bloc
- *Durée de vie*: la durée de vie du bloc
- *Visibilité*: restreinte au bloc de définition

Variables (2 / 3)

```c
int counter = 0;

int f(void)
{
    counter += 1;          /* incrementing the global variable */
    x += 1;                /* error: x unknown */
}

int x;                     /* now, x is known */

int g(void)
{
    int counter = 0;       /* mask global variable */
    counter += 1;          /* incrementing the local variable */
    x += 1;
}
```

Variables (3 / 3)

Toujours dans le cas de programmes mono-fichiers

Variable statique

- *Définition*: dans le bloc (doit être pré-fixée par **static**)
- *Durée de vie*: tout le programme (comme une globale)
- *Visibilité*: restreinte au bloc de définition (comme une locale)

```
    void f1(void) {
        static int counter = 0;
        printf("f1 was called %d times\n", ++counter);
    }
    void f2(void) {
        static int counter = 0;
        printf("f2 was called %d times\n", ++counter);
    }
    void main(void) {
        f1(); f2(); f1();
    }
==>     f1 was called 1 times
        f2 was called 1 times
        f1 was called 2 times
```