Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Systèmes Embarqués 1 & 2

a.06 – C – Les fonctions

Classes T-2/I-2 // 2018-2019

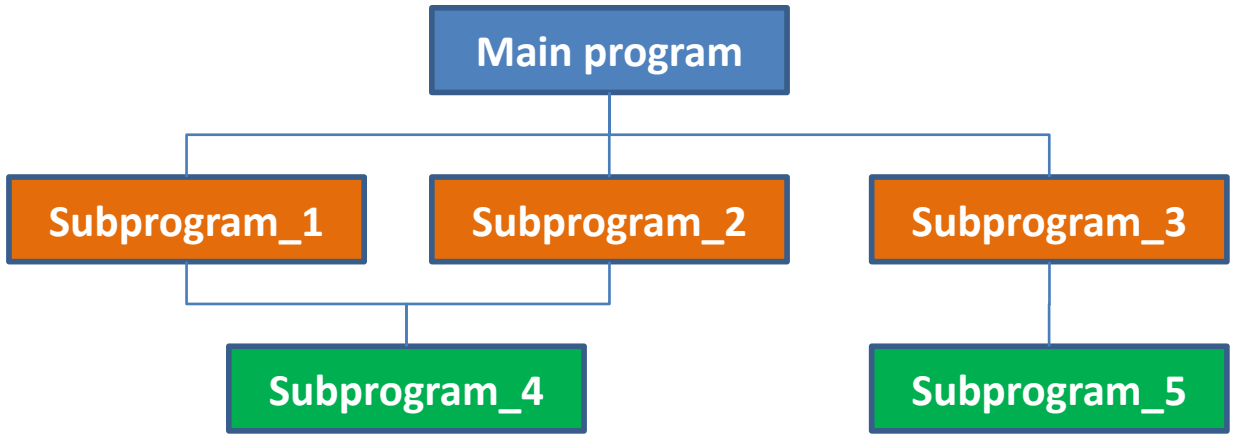
Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Daniel Gachet | HEIA-FR/TIC
a.06 | 10.09.2018

Contenu

- Introduction
- Déclaration et définition
- Arguments
- Variables
- Appels et récursivité
- Réentrance

- **Un bon programme utilise le principe de modularité, lequel consiste à fractionner le programme en plusieurs sous-programmes.**
  - Le programme principal ne contient que la structure logique de l'algorithme
  - Les sous-programmes ne s'occupent que des détails d'implémentation



- **Avantage de l'utilisation de sous-programmes**
  - Réduction de la taille du programme principal
  - Simplification lors du développement de grands programmes (bibliothèques)
  - Simplification lors de la validation de grands programmes (tests unitaires)

Déclaration

C propose, comme beaucoup d'autres langages, le concept de fonction permettant de découper des algorithmes de traitement longs et complexes en de petites procédures simples et claires.

La déclaration d'une fonction prend la forme suivante:

```
[extern] <type_name> <function_name> (argument-list);
```

La déclaration d'une fonction permet de connaître sa signature/son prototype (type et liste des arguments/paramètres) sans devoir connaître son implémentation (forward declaration). Si la fonction ne retourne aucune valeur, le type `void` doit être utilisé.

Les fonctions peuvent être naturellement regroupées en bibliothèques. Ces bibliothèques permettent aux développeurs logiciel de créer de nouvelles applications en se basant sur ce que d'autres ont déjà réalisé. On évite ainsi de recommencer de zéro à chaque fois. La déclaration de fonctions se fait alors dans des fichiers d'entête (header-files). La déclaration de la fonction peut être précédée du préfixe `extern`, signifiant que la fonction ne fait pas partie du module de compilation, mais provient d'un autre module ou d'une bibliothèque (portée par défaut).

Définition

La définition (implémentation) d'une fonction prend la forme suivante:

```
[static] <type_name> <function_name> (argument-list) {
   statements
   [return [expression];]
}
```

L'expression de retour `return` doit être du même type que la fonction (`<type_name>`).

Le préfixe `static` permet de privatiser une fonction, c.à.d. de ne la rendre visible qu'à l'intérieur du module de compilation.

p. ex.

```
int square (int n) {
   return n*n;
}

static void do_nothing () {
}
```

Attention:

- **Il n'est pas possible d'imbriquer la définition de fonctions.**
- **Le nom d'une fonction doit être unique à l'intérieur d'un module de compilation ou d'une application si la fonction est globale.**
- **Il est de bon conseil de n'utiliser l'instruction «*return*» qu'à la fin de la fonction.**

Arguments

- **Deux techniques sont utilisées lorsqu'un programme passe les paramètres:**
  - Passage par valeur
  - Passage par référence (adresse)

- **La technique de passage par valeur (call by value) place une copie de la donnée actuelle sur la pile:**
  - Le sous-programme reçoit une copie de la donnée
  - Il peut la lire et la modifier sans que la donnée originale ne soit altérée
  - Il existe deux positions mémoires distinctes
  - La modification de l'une des positions n'a aucune conséquence sur l'autre position

  Exemple : `int fnct1 (int a, char c);`

- **La technique de passage par référence (call by reference) place l'adresse de la donnée sur la pile:**
  - Le sous-programme reçoit l'adresse de la donnée
  - Cette donnée est ainsi partagée entre la fonction appelante et la fonction appelée
  - La donnée ne se situe que dans une seule position mémoire
  - La modification du paramètre par la fonction appelée est visible par la fonction appelante

  Exemple: `int fnct2 (char* s, int* b);`

Arguments – par référence / adresse constante

Pour éviter que des paramètres passés par adresse soient modifiés par la fonction, ceux-ci peuvent être passés comme valeur constante `const`.

p.ex.

1. `int fnct3 (const char* s, int* b);`
   le contenu de 's' ne peut pas être modifié

2. `int fnct4 (const char* const s, int* b);`
   ni le contenu, ni l'adresse ne peuvent être modifiés

Attention:

Il est important de savoir que tous les paramètres d'une fonction sont copiés sur la pile. Si des arguments sont passés par valeur et qu'ils sont de grande taille, le programme perd énormément en performance et des dépassements de la pile peuvent survenir («*stack overflow*»). Pour éviter ce genre de désagrément, il est recommandé de passer de tels paramètres par référence (adresse constante).

Arguments – ellipsis operator

Si le nombre d'arguments ne peut pas être défini lors de la déclaration d'une fonction, C permet de définir la signature de la fonction à l'aide de l'opérateur `"..."` (ellipsis operator).

p.ex.

```
extern void printargs (int argc, ...);
```

La bibliothèque `<stdarg.h>` permet ensuite d'itérer sur la liste d'arguments, p.ex.

```
void printargs (int argc, ...)
{
  va_list ap;                          → pointeur sur les arguments non-déclarés
  va_start(ap, argc);                  → fait que 'ap' pointe sur le 1er argument non-déclaré
  while (argc-- > 0) {
    int i = va_arg(ap, int);           → converti l'argument dans le type souhaité
    printf("%d ", i);
  }
  va_end(ap);                          → restaure l'état après usage (vide la liste)
  printf ("\n");
}
```

- **Conventions C pour les processeurs ARM**
  - Les 4 premiers paramètres sont placés dans les registres R0 à R3, les autres paramètres sont placés sur la pile de droite à gauche, c.à.d. le paramètre le plus à droite est placé en premier sur la pile et le paramètre le plus à gauche en dernier.
  - Sur un processeur 32 bits, les caractères, les entiers et les types énumérés sont étendus à 32 bits avant d'être placés sur la pile.
  - L'utilisation des registres est défini dans *04_ARM_Architecture_Procedure_Call_Standard.pdf* comme suit :

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Table 2, Core registers and AAPCS usage

Variables locales

Les fonctions peuvent définir des variables locales, variables visibles qu'à l'intérieur du corps de la fonction ou d'un bloc d'une fonction.

Ces variables sont initialisées lorsque l'exécution du programme arrive à leur définition. Cette initialisation est effectuée à chaque appel de la fonction.

Les variables sont créées à chaque invocation de la fonction (une copie par appel), p.ex.

```
int fnct (int a) {
  int b=8;                    // initialized at each call
  return a * b;
}
```

Attention:

Les variables locales sont créées sur la pile («*stack*») et peuvent engendrer des dépassements de la pile («*stack overflow*»).

Variables rémanentes

Si une fonction a besoin de conserver la valeur d'une variable entre deux appels, elle peut le faire en déclarant la variable `static`. Celle-ci sera initialisée qu'une seule fois, lors du lancement de l'application.

p.ex.

```
int fnct (int a) {
  int b=8;                  // initialized at each call
  static int count = 0;     // initialized once (0, 1, 2 … n calls)
  count++;
  return a * b + count;
}
```

Attention:

L'utilisation de variables statiques rend le code non-réentrant et peut poser d'énormes difficultés dans des applications utilisant des threads.

Variables globales

Les fonctions peuvent également accéder des variables globales, variables définies à l'extérieur de la fonction. Ces variables sont naturellement partagées par toutes les fonctions d'un module/programme.

p.ex.

```
int g_var = 10;  //initialized once, while launching the program

int fnct (int a) {
  int b=8;                  // initialized at each call
  static int count = 0;     // initialized once (0, 1, 2 … n calls)
  count++;
  return (a * b + count) * g_var;
}
```

L'accès à ces variables globales par des fonctions est connu sous le nom d'effet de bord (side effect). Son usage n'est pas recommandé. Il est préférable de passer leur référence/adresse à la fonction lors de son appel.

Attention:

L'utilisation de variables rend le code non-réentrant et peut poser d'énormes difficultés dans des applications faisant appel à des threads.

Appel

L'appel d'une fonction peut s'effectuer de deux manières différentes, soit comme une instruction, soit comme une opérande d'une expression.

p.ex.

1. Instruction `fnct (10);`
2. Opérande `int i = fnct (20);`

L'appel d'une fonction est réalisé en 4 étapes:

1. Les paramètres d'appel sont évalués
2. Pour chaque paramètre, le résultat de son évaluation est placé dans un registre ou sur la pile
3. Appel de la fonction
4. Restauration de la pile

Attention:

L'ordre d'évaluation des paramètres n'est défini et peut avoir des comportements différents entre les compilateurs ou leur version.

```
extern int fnct (int a, int b, int c);
int i=0; int j=3; int k=0;
k = fnct (i, ++i, j);          → a peut contenir aussi bien 0 que 1!!!
k = fnct (i, i+1, j); i++;     → a le bon comportement.
```

C implémente le concept de récursivité, c.à.d. l'appel d'une fonction par elle-même, permettant ainsi au développeur logiciel de programmer des algorithmes récursifs.

p.ex.

```
int power (int n, int exp) {
  if (exp > 0) return n*power(n, exp-1);
  return 1;
}
```

Attention:

L'utilisation d'algorithmes récursifs peut avoir un impact majeur sur les performances de l'application ainsi que sur l'utilisation de la pile.

Code réentrant - Librairies

- **En informatique, la réentrance est la propriété pour une fonction d'être utilisable par plusieurs tâches simultanément. La réentrance permet d'éviter la duplication en mémoire vive de fonctions utilisées simultanément par plusieurs applications.**

- **Les conditions pour qu'une fonction soit réentrante sont:**
  - Ne doit pas utiliser de données statiques (globales) non-constantes
  - Ne doit retourner l'adresse de données statiques (globales) non-constantes
  - Ne doit traiter que des données fournies par le programme appelant
  - Ne doit pas modifier son propre code
  - Ne doit pas appeler des sous-programmes non-réentrants