ALGORITHMIQUE ET PROGRAMMATION C

CHAP 5: LES FONCTIONS

Dr. Ikbal Chammakhi Msadaa

Classes: 1 TA

Introduction

- Comme dans la plupart des langages, on peut en C découper un programme en plusieurs **fonctions**. Une seule de ces fonctions existe obligatoirement ; c'est la fonction principale appelée **main**.
- Cette fonction principale peut, éventuellement, appeler une ou plusieurs fonctions secondaires.
- De même, chaque fonction secondaire peut appeler d'autres fonctions secondaires ou s'appeler elle-même (dans ce dernier cas, on dit que la fonction est ***récursive***).

Définition d'une fonction

- La définition d'une fonction est la donnée du texte de son algorithme, qu'on appelle corps de la fonction. Elle est de la forme:

```
type  nom-fonction ( type-1 arg-1,..., type-n arg-n)
{
  [ déclarations de variables locales ]
   liste d'instructions
}
```

- Dans cet en-tête, ***type*** désigne le type de la fonction, c'est-à-dire le type de la valeur qu'elle retourne. Contrairement à d'autres langages, il n'y a pas en C de notion de procédure ou de sous-programme. Une fonction qui ne renvoie pas de valeur est une fonction dont le type est spécifié par le mot-clef **void**.

Définition d'une fonction

- Les arguments de la fonction sont appelés ***paramètres formels***, par opposition aux ***paramètres effectifs*** qui sont les paramètres avec lesquels la fonction est effectivement appelée. Les paramètres formels peuvent être de n'importe quel type. Leurs identificateurs n'ont d'importance qu'à l'intérieur de la fonction. Si la fonction ne possède pas de paramètres, on remplace la liste de paramètres formels par le mot-clef **void**.

- Le corps de la fonction débute éventuellement par des déclarations de variables, qui sont locales à cette fonction. Il se termine par l'*instruction de retour à la fonction appelante*, return, dont la syntaxe est

return(*expression*);

- La valeur de *expression* est la valeur que retourne la fonction. Son type doit être le même que celui qui a été spécifié dans l'en-tête de la fonction. Si la fonction ne retourne pas de valeur (fonction de type **void**), sa définition s'achève par

return;

- Plusieurs instructions return peuvent apparaître dans une fonction. Le retour au programme appelant sera alors provoqué par le premier return rencontré lors de l'exécution.

Exemples de Fonctions

```
int produit (int a, int b)
{
  return(a*b);
}
```

```
int puissance (int a, int n)
{
  if (n == 0)
    return(1);
  return(a * puissance(a, n-1));
}
```

```
void imprime_tab (int *tab, int nb_elements)
{
  int i;
  for (i = 0; i < nb_elements; i++)
    printf("%d \t",tab[i]);
  printf("\n");
  return;
}
```

Appel d'une fonction

- L'appel d'une fonction se fait par l'expression:

***nom-fonction*(*para-1*, *para-2*, ..., *para-n*);**

- **L'ordre et le type** des paramètres effectifs de la fonction doivent concorder avec ceux donnés dans l'en-tête de la fonction. Les paramètres effectifs peuvent être des expressions.
- Cela implique en particulier que l'ordre d'évaluation des paramètres effectifs n'est pas assuré et dépend du compilateur.
- Il est donc déconseillé, pour une fonction à plusieurs paramètres, de faire figurer des opérateurs d'incrémentation ou de décrémentation (++ ou --) dans les expressions définissant les paramètres effectifs.

Déclaration d'une fonction

- Le C n'autorise pas les fonctions imbriquées. La définition d'une fonction secondaire doit donc être placée soit avant, soit après la fonction principale main.
- Toutefois, il est indispensable que le compilateur "connaisse" la fonction au moment où celle-ci est appelée.
- Si une fonction est définie après son premier appel (en particulier si sa définition est placée après la fonction main), elle doit impérativement être déclarée au préalable.
- Une fonction secondaire est déclarée par son **prototype**, qui donne le type de la fonction et celui de ses paramètres, sous la forme :

type nom-fonction(type-1,...,type-n);

- Les fonctions secondaires peuvent être déclarées indifféremment avant ou au début de la fonction main

Déclaration d'une fonction

Exemple:

```
int puissance (int, int );

main()
{
  int a = 2, b = 5;
  printf("%d\n", puissance(a,b));
}

int puissance (int a, int n)
{
  if (n == 0)
    return(1);
  return(a * puissance(a, n-1));
}
```

- Même si la déclaration est parfois facultative (par exemple quand les fonctions sont définies avant la fonction main et dans le bon ordre), elle seule permet au compilateur de vérifier que le nombre et le type des paramètres utilisés dans la définition concordent bien avec le prototype.
- De plus, la présence d'une déclaration permet au compilateur de mettre en place d'éventuelles conversions des paramètres effectifs, lorsque la fonction est appelée avec des paramètres dont les types ne correspondent pas aux types indiqués dans le prototype.

Durée de vie des variables

- Les variables manipulées dans un programme C ne sont pas toutes traitées de la même manière. En particulier, elles n'ont pas toutes la même *durée de vie*. On distingue deux catégories de variables:
  - **Les variables permanentes (ou statiques):** Une variable permanente occupe un emplacement en mémoire qui reste le même durant toute l'exécution du programme. Cet emplacement est alloué une fois pour toutes lors de la compilation. Par défaut, les variables permanentes sont initialisées à zéro par le compilateur. Elles sont caractérisées par le mot-clef **static**.
  - **Les variables temporaires:** Les variables temporaires se voient allouer un emplacement en mémoire de façon dynamique lors de l'exécution du programme. Elles ne sont pas initialisées par défaut. Leur emplacement en mémoire est libéré par exemple à la fin de l'exécution d'une fonction secondaire.
  - La durée de vie des variables est liée à leur **portée**, c'est-à-dire à la portion du programme dans laquelle elles sont définies.

Les variables globales

- On appelle *variable globale* une variable déclarée en dehors de toute fonction.
- Une variable globale est connue du compilateur dans toute la portion de code qui suit sa déclaration.
- Les variables globales sont systématiquement permanentes. Dans le programme suivant, n est une variable globale :

```
int n;
void fonction();

void fonction()
{
  n++;
  printf("appel numero %d\n",n);
  return;
}

main()
{
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}
```

Les variables locales

- On appelle variable locale une variable déclarée à l'intérieur d'une fonction (ou d'un bloc d'instructions) du programme.
- Par défaut, les variables locales sont temporaires. Quand une fonction est appelée, elle place ses variables locales dans la pile. A la sortie de la fonction, les variables locales sont dépilées et donc perdues.
- Les variables locales n'ont en particulier aucun lien avec des variables globales de même nom. Par exemple, le programme suivant:

```
int n = 10;
void fonction();

void fonction()
{
  int n = 0;
  n++;
  printf("appel numero %d\n",n);
  return;
}

main()
{
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}
```

Les variables locales

```
int n = 10;
void fonction();

void fonction()
{
  static int n;
  n++;
  printf("appel numero %d\n",n);
  return;
}

main()
{
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}
```

On voit que la variable locale n est de classe statique (elle est initialisée à zéro, et sa valeur est conservée d'un appel à l'autre de la fonction). Par contre, il s'agit bien d'une variable locale, qui n'a aucun lien avec la variable globale du même nom.

- Les variables locales à une fonction ont une durée de vie limitée à une seule exécution de cette fonction. Leurs valeurs ne sont pas conservées d'un appel au suivant.
- Il est toutefois possible de créer une variable locale de classe statique en faisant précéder sa déclaration du mot-clef static :

  static type nom-de-variable;

  Une telle variable reste locale à la fonction dans laquelle elle est déclarée, mais sa valeur est conservée d'un appel au suivant. Elle est également initialisée à zéro à la compilation.
- Par exemple, dans le programme suivant, n est une variable locale à la fonction secondaire fonction, mais de classe statique.

Transmission des paramètres d'une fonction

- Les paramètres d'une fonction sont traités de la même manière que les variables locales de classe automatique : lors de l'appel de la fonction, les paramètres effectifs sont copiés dans le segment de pile. La fonction travaille alors uniquement sur cette copie.

- Cette copie disparaît lors du retour au programme appelant. Cela implique en particulier que, si la fonction modifie la valeur d'un de ses paramètres, seule la copie sera modifiée ; la variable du programme appelant, elle, ne sera pas modifiée. On dit que les paramètres d'une fonction sont *transmis par valeurs*. Par exemple, le programme suivant:

Transmission des paramètres d'une fonction

```
void echange (int, int );

void echange (int a, int b)
{
  int t;
  printf("debut fonction :\n a = %d \t b = %d\n",a,b);
  t = a;
  a = b;
  b = t;
  printf("fin fonction :\n a = %d \t b = %d\n",a,b);
  return;
}

main()
{
  int a = 2, b = 5;
  printf("debut programme principal :\n a = %d \t b = %d\n",a,b);
  echange(a,b);
  printf("fin programme principal :\n a = %d \t b = %d\n",a,b);
}
```

Transmission des paramètres d'une fonction

- Pour qu'une fonction modifie la valeur d'un de ses arguments, il faut qu'elle ait pour paramètre l'adresse de cet objet et non sa valeur. Par exemple, pour échanger les valeurs de deux variables, il faut écrire :

```
void echange (int *, int *);

void echange (int *adr_a, int *adr_b)
{
  int t;
  t = *adr_a;
  *adr_a = *adr_b;
  *adr_b = t;
  return;
}

main()
{
  int a = 2, b = 5;
  printf("debut programme principal :\n a = %d \t b = %d\n",a,b);
  echange(&a,&b);
  printf("fin programme principal :\n a = %d \t b = %d\n",a,b);
}
```

La fonction main

- La fonction principale **main** est une fonction comme les autres. Nous avons jusqu'à présent considéré qu'elle était de type void, ce qui est toléré par le compilateur.
- En fait, la fonction **main** **est de type** **int**. Elle doit retourner un entier dont la valeur est transmise à l'environnement d'exécution. Cet entier indique si le programme s'est ou non déroulé sans erreur.
- La valeur de retour 0 correspond à une **terminaison correcte**, toute valeur de retour non nulle correspond à une **terminaison sur une erreur**.
- On peut utiliser comme valeur de retour les deux constantes symboliques définies dans **stdlib.h**:
  - **EXIT_SUCCESS** (égale à 0) et
  - **EXIT_FAILURE** (égale à 1)
- Lorsqu'elle est utilisée sans arguments, la fonction main a donc pour prototype:

```
int main(void);
```

La fonction main

- la fonction **main** possède **deux paramètres formels**, appelés par convention **argc** (argument count) et **argv** (argument vector).
  - **argc** est une variable de type **int** dont la valeur est égale au nombre de mots composant la ligne de commande (y compris le nom de l'exécutable). Elle est donc égale au nombre de paramètres effectifs de la fonction + 1.
  - **argv** est **un tableau de chaînes de caractères** correspondant chacune à un mot de la ligne de commande. Le premier élément argv[0] contient donc le **nom de la commande** (du fichier exécutable), le second argv[1] contient **le premier paramètre**. . . .
- Le second prototype valide de la fonction main est donc:

int main (int argc, char *argv[]);

La fonction main

- La ligne de commande qui sert à lancer le programme est, dans ce cas, composée du nom du fichier exécutable suivi par des paramètres: dans cet exemple: **a.out 12 8**
- Ici, **argv** sera un tableau de 3 chaînes de caractères **argv[0], argv[1] et argv[2]** qui, dans notre exemple, valent respectivement **"a.out", "12" et "8"**.
- Enfin, la fonction de la librairie standard atoi(), déclarée dans stdlib.h, prend en argument une chaîne de caractères et retourne l'entier dont elle est l'écriture décimale.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int a, b;

  if (argc != 3)
    {
      printf("\nErreur : nombre invalide d'arguments");
      printf("\nUsage: %s int int\n",argv[0]);
      return(EXIT_FAILURE);
    }
  a = atoi(argv[1]);
  b = atoi(argv[2]);
  printf("\nLe produit de %d par %d vaut : %d\n", a, b, a * b);
  return(EXIT_SUCCESS);
}
```

La compilation séparée et ses conséquences

- Le langage C est un langage que l'on peut qualifier d'opérationnel, en partie grâce à ses possibilités dites de **compilation séparée**.
- En C, en effet, il est possible de compiler séparément plusieurs programmes (fichiers) source et de rassembler les modules objet correspondants au moment de l'édition de liens.
- D'ailleurs, dans certains environnements de programmation, la notion de **projet** permet de gérer la multiplicité des fichiers (source et modules objet) pouvant intervenir dans la création d'un programme exécutable.
- Les possibilités de compilation séparée ont toutefois une **incidence importante au niveau de la portée des variables globales**.

La compilation séparée et ses conséquences

- **La portée d'une variable globale - la déclaration extern**
  - A priori, la portée d'une variable globale semble limitée au fichier source dans lequel elle a été définie. Ainsi, supposez que l'on compile séparément ces deux fichiers source:
  - Il ne semble pas possible, dans les fonctions fct2 et fct3 de faire référence à la variable globale x déclarée dans le premier fichier source (alors qu'aucun problème ne se poserait si l'on réunissait ces deux fichiers source en un seul.

```
source 1
 int x ;
 main()
 {
    .....
 }
 fct1()
 {
    .....
 }
```

```
source 2
   fct2()
   {
      .....
   }
   fct3()
   {
      .....
   }
```

La compilation séparée et ses conséquences

- En fait, le langage C prévoit une déclaration permettant de spécifier qu'une variable globale a déjà été définie dans un autre fichier source. Celle-ci se fait à l'aide du mot-clé **extern**.
- Ainsi, en faisant précéder notre second fichier source de la déclaration :

  extern int x ;
- il devient possible de mentionner la variable globale x (déclarée dans le premier fichier source) dans les fonctions fct2 et fct3.

La compilation séparée et ses conséquences

- **Les variables globales cachées - la déclaration static**
  - Il est possible de « cacher » une variable globale, c'est-à-dire de la rendre inaccessible à l'extérieur du fichier source où elle a été définie (on dit aussi « rendre confidentielle » au lieu de « cacher» ; on parle alors de « variables globales confidentielles »). Il suffit pour cela d'utiliser la déclaration **static** comme dans cet exemple :

```
static int a ;
main()
{
    .....
}
fct()
{
    .....
}
```

La compilation séparée et ses conséquences

- Sans la déclaration **static**, **a** serait une variable globale ordinaire. Par contre, cette déclaration demande qu'aucune trace de a ne subsiste dans le module objet résultant de ce fichier source. Il sera donc impossible d'y faire référence depuis une autre source par **extern**.
- Si une autre variable globale apparaît dans un autre fichier source, elle sera acceptée à l'édition de liens puisqu'elle ne pourra pas interférer avec celle du premier fichier source.
- Cette possibilité de cacher des variables globales peut s'avérer précieuse lorsque vous êtes amené à développer un ensemble de fonctions d'intérêt général qui doivent partager des variables globales. En effet, elle permet à l'utilisateur éventuel de ces fonctions de ne pas avoir à se soucier des noms de ces variables globales puisqu'il ne risque pas alors d'y accéder par mégarde.
- Cela généralise en quelque sorte à tout un fichier source la notion de variable locale telle qu'elle était définie pour les fonctions.

variables locales à un bloc

- On peut classer les variables en quatre catégories en fonction de leur portée (ou espace de validité). Nous avons déjà rencontré les trois premières que sont : les variables globales, les variables globales cachées et les variables locales à une fonction. En outre, il est possible de définir des **variables locales à un bloc**. Elles se déclarent en début d'un bloc de la même façon qu'en début d'une fonction. Dans ce cas, la portée de telles variables est limitée au bloc en question. Leur usage est, en pratique, peu répandu.

Tableau Récapitulatif

Type, portée et classe d'allocation des variables

Type de variable	Déclaration	Portée	Classe d'allocation
Globale	en dehors de toute fonction	• la partie du fichier source suivant sa déclaration, • n'importe quel fichier source, avec `extern`.	**Statique**
Globale cachée	en dehors de toute fonction, avec l'attribut `static`	uniquement la partie du fichier source suivant sa déclaration	
Locale « rémanente »	au début d'une fonction, avec l'attribut `static`	la fonction	
Locale à une fonction	au début d'une fonction	la fonction	**Automatique**
Locale à un bloc	au début d'un bloc	le bloc	