Algorithmique et structures de données

Chapitre 4 – Les fonctions

Samar MOUCHAWRAB, PhD Eng

2A Cycle Préparatoire – Semestre 1
2017/2018

Introduction

Comment réutiliser un code ou un algorithme existant sans avoir à le réécrire ?

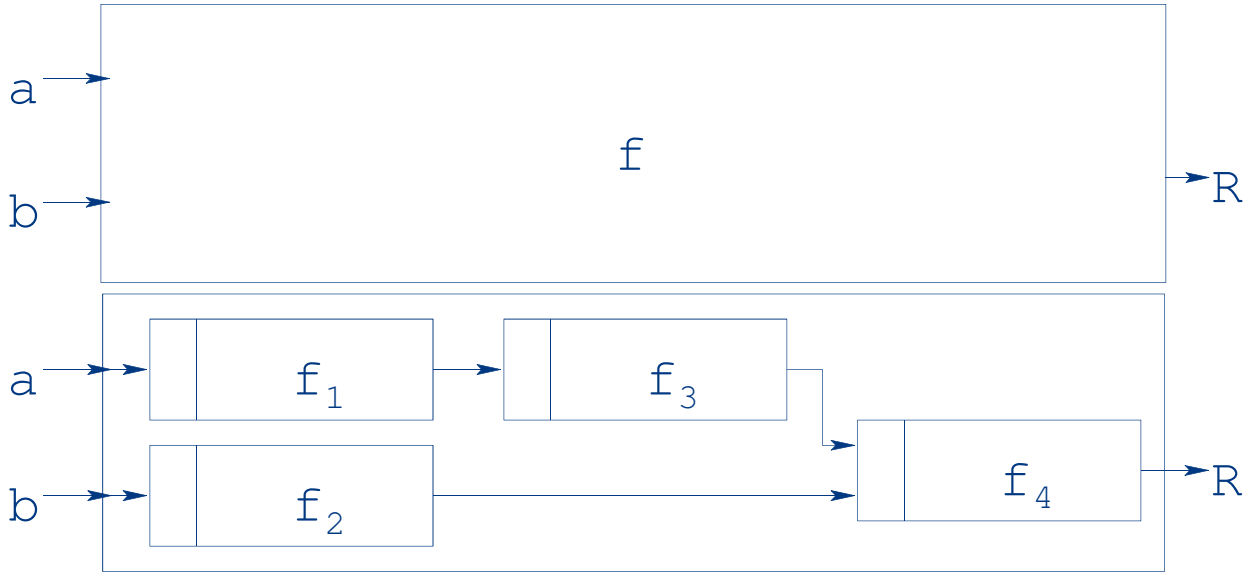
Encapsuler le code dans des fonctions ou des procédures

Comment structurer un algorithme pour le rendre plus compréhensible ?

Utiliser des fonctions ou des procédures

Diviser pour régner

Les fonctions et les procédures permettent de décomposer un programme complexe en une série de sous-programmes plus simples, lesquels peuvent à leur tour être décomposés eux-mêmes en fragments plus petits, et ainsi de suite.



Conception structurée descendante

Découper l'algorithme (action) en sous-algorithmes (sous-actions) plus simples, jusqu'à des opérations considérées primitives.

Les buts de cette conception sont:

- Simplification
- Abstraction (ignorer les détails)
- Structuration
- Réutilisation

Définition d'une fonction

Une fonction est définie comme suit :

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

La première ligne de cette définition est l'entête de la fonction. Dans cet entête, `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-clé `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.

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'entête de la fonction. Si la fonction ne retourne pas de valeur (fonction de type `void`), sa définition s'achève par `return;` ou sans `return`

Définition d'une fonction

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 :

```
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

```
nom-fonction(para-1,para-2,...,para-n);
Ident = 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'entête de la fonction. Les paramètres effectifs peuvent être des expressions.

Déclaration d'une fonction en C

Le langage 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);
```

Déclaration d'une fonction en C

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

Par exemple, on écrira

```
int puissance (int, int );
int puissance (int a, int n) {
   if (n == 0) return(1);
   return(a * puissance(a, n-1));
}
void main() {
      int a = 2, b = 5;
      printf("%d\n", puissance(a,b));
}
```

Exercice 1

Réécrire l'algorithme qui reçoit en entrée les coordonnées de 3 points et affiche le type du triangle formé (équilatéral, isocèle, rectangle ou autre) en y intégrant des fonctions.

Pour rappel:

La distance entre les points $A(x_1, y_1)$ et $B(x_2, y_2)$ est donnée par la formule suivante:
$d(A, B) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

Exercice 1

Exercice triangles avec fonctions

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

/* fonction pour la comparaison de deux floats pour limiter la
précision à 5 chiffres après la virgule */

int compare (float a, float b) {
        float c = a - b;
        if (c < 0) c = c * (-1);
        if (c < 0.00001)
                return 1;
        else
                return 0;
}

// fonction pour vérifier si triangle isocèle : deux côtés égaux
int isocele (float c1, float c2) {
        if (compare(c1,c2))
                return 1;
        else
                return 0;
}
```

Exercice 1

```
//fonction pour vérifier si triangle equilateral: trois côtés égaux

int equilateral (float c1, float c2, float c3) {
        if (compare(c1,c2) && compare(c1,c3))
                return 1;
        else
                return 0;
}

// fonction pour vérifier si triangle rectangle

int rectangle (float c1, float c2, float c3) {
        if (compare(pow(c1,2), pow(c2,2)+pow(c3,2)))
                return 1;
        else
                return 0;
}
```

Exercice 1

```
int main() {

        // Définition des variables
        float xA, yA, xB, yB, xC, yC; // coordonnées des trois points
        float AB, AC, BC; //Les distances des côtés

        //Lecture des coordonnées
        printf("Entrer l'abscisse du point A : ");
        scanf("%f", &xA);
        printf("Entrer l'ordonnee du point A : ");
        scanf("%f", &yA);
        printf("Entrer l'abscisse du point B : ");
        scanf("%f", &xB);
        printf("Entrer l'ordonnee du point B : ");
        scanf("%f", &yB);
        printf("Entrer l'abscisse du point C : ");
        scanf("%f", &xC);
        printf("Entrer l'ordonnee du point C : ");
        scanf("%f", &yC);
        //Calcul des distances
        AB = sqrt(pow((xB - xA),2) + pow((yB - yA),2));
        AC = sqrt(pow((xC - xA),2) + pow((yC - yA),2));
        BC = sqrt(pow((xC - xB),2) + pow((yC - yB),2));
```

Exercice 1

```
printf("AB = %f\n", AB);
printf("AC = %f\n", AC);
printf("BC = %f\n", BC);

//Tests pour identification de type de triangle
if (compare(AB + AC , BC) || compare(AC + BC , AB) ||
        compare(BC + AB , AC)) {
        printf("ABC n\'est pas un triangle.");
} else {
        if (equilateral(AB, AC, BC)) {
                printf("Le triangle ABC est equilateral");
        } else if (isocele (AB, AC)) {
                if (rectangle(BC,AC,AB))
                        printf("Le triangle ABC est rectangle et
                        isocele en A");
                else
                        printf("Le triangle ABC est isocele en A");
        } else if (isocele(AC,BC)) {
                if (rectangle(AB,AC,BC))
                        printf("Le triangle ABC est rectangle et
                        isocele en C");
                else
                        printf("Le triangle ABC est isocele en C");
```

Exercice 1

```
                } else if (isocele(AB,BC)) {
                        if (rectangle(AC,AB,BC))
                                printf("Le triangle ABC est rectangle et
isocele en B");
                        else
                                printf("Le triangle ABC est isocele en B");
                } else if (rectangle(AB,AC,BC)) {
                        printf("Le triangle ABC est rectangle en C");
                } else if (rectangle(AC,AB,BC)) {
                        printf("Le triangle ABC est rectangle en B");
                } else if (rectangle(BC,AC,AB)) {
                        printf("Le triangle ABC est rectangle en A");
                } else {
                        printf("Le triangle ABC n\'est ni isocele ni
equilateral ni rectangle.");
                }
        }
        return 0;
}
```

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.

Dans le programme suivant, n est une variable globale :

```
int n = 0;
void fonction();
void fonction() {
   n++;
   printf("Appel numero  % d\n",n); return;
}
void main() {
   int i;
   for (i = 0; i < 5; i++)
     fonction();
}
```

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. Exemple :

```
int n  = 10;
void fonction() {
   int n = 0; n++;
   printf("Appel numero %d\n",n);
}
main(){
   int i;
   for (i = 0; i < 5; i++)
      fonction();
}
```

Variables locales

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-clé 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.

Variables locales

Par exemple, dans le programme suivant, n est une variable locale à la fonction secondaire fonction, mais de classe statique.

```
int n  = 10;

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

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

Transmission des paramètres d'une fonction

Les paramètres d'une fonction sont traités de la même manière que les variables locales : 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.

Transmission des paramètres d'une fonction

Par exemple, le programme suivant

```
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

Sortie du programme :

```
debut programme principal : a = 2     b = 5
debut fonction : a = 2   b = 5
fin  fonction :a = 5     b = 2
fin programme principal : a = 2       b = 5
```

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);
}
```

Transmission des paramètres d'une fonction

Rappelons qu'un tableau est un pointeur sur le premier élément du tableau. Lorsqu'un tableau est transmis comme paramètre à une fonction secondaire, ses éléments sont donc modifiés par la fonction. Par exemple, le programme initialise les éléments du tableau tab:

```
#include <stdlib.h>
void init (int *, int );
void init (int *tab, int n){
    int i;
    for (i = 0; i < n; i++)
        tab[i] = i;
    return;
}
main(){
    int i, n = 5;
    int *tab;
    tab = (int*)malloc(n * sizeof(int));
    init(tab,n);
}
```

Transmission des paramètres d'une fonction

malloc réserve dans la mémoire l'espace disque qui sera occupé par la variable. Elle est définie dans la librairie stdlib

Sans malloc, le code peut être réécrit:

```
#include <stdio.h>

void init (int *tab, int n){
        int i;
        for (i = 0; i < n; i++) tab[i] = i;
}
void main(){
        int i, n = 5;
        int tab[n];
        init(tab,n);
        for (i=0; i<n; i++) printf("%d\n", tab[i]);
}
```

A noter que la déclaration du type de paramètre tableau dans la fonction peut aussi se faire avec les []:

```
void init (int tab[], int n);
```

Exercice 2

Ecrire une fonction qui calcule la factorielle d'un nombre entier avec une structure itérative (pour ou tant que).

Tester cette fonction avec appel de la fonction main d'un programme en C

Rappel:

$0! = 1$

Pour tout entier $n > 0$, $n! = (n - 1)! \times n$.

Exercice 2

```
#include <stdio.h>
int factorielle (int n);
int main() {
        int a, f;
        printf("Entrer un nombre entier positif : ");
        scanf("%d", &a);
        f = factorielle (a);
        if (f == -1)
                printf("La factorielle n'est pas definie
        pour les entiers negatifs.");
        else
                printf("La factorielle de %d est %d", a, f);
        return 0;
}
int factorielle (int n) {
        int i, result = 1;
        if (n < 0)
                return -1;
        else
                for (i=1; i <= n; i++) {
                        result = result * i;
                }
        return result;
}
```

Exercice 3

Un nombre premier est un entier naturel qui admet exactement deux diviseurs distincts entiers et positifs (qui sont alors 1 et lui-même). Ainsi, 1 n'est pas premier car il n'a qu'un seul diviseur entier positif ; 0 non plus car il est divisible par tous les entiers positifs. Par opposition, un nombre non nul produit de deux nombres entiers différents de 1 est dit composé. Par exemple 6 = 2 × 3 est composé, tout comme 21 = 3 × 7 ou 7 × 3, mais 11 est premier car 1 et 11 sont les seuls diviseurs de 11.

Les nombres 0 et 1 ne sont ni premiers ni composés.

Les vingt-cinq nombres premiers inférieurs à 100 sont :

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89 et 97.

Ecrire une fonction qui vérifie la primalité d'un nombre entier avec une structure itérative. Tester avec appel de la fonction main

Exercice 3

```
#include <stdio.h>
int premier (int i) {
        int j, nb = 0;
        //nb nombre de diviseurs autres que 1 et nombre lui même
        if (i == 0 || i == 1) return 0;
        else {
                for (j = 2; j<=i ; j++) {
                        if (i % j == 0)
                                nb++;
                }
                if (nb == 1) return 1;
// nb = 1 veut dire que le nombre est divisible par lui même
                else return 0;
/* si nb different de 1 ça veut dire qu'il y a autres diviseurs donc
nombre non premier */
        }
}
void main() {
        int n;
        printf("Entrer un nombre: ");
        scanf("%d", &n);
        if (premier(n)== 1) printf ("%d est premier", n);
        else printf("%d n\'est pas premier", n);
}
```

Exercice 3 – Deuxième solution

```
#include <stdio.h>

int premier (int i) {
        int j;
        if (i == 0 || i == 1) return 0;
        else {
                for (j = 2; j<i ; j++) {
                        if (i % j == 0) {
                           return 0;// un diviseur existe donc non premier
                        }
                }
                return 1; // pas de diviseurs donc premier
        }
}

void main() {
        int n;
        printf("Entrer un nombre: ");
        scanf("%d", &n);
        if (n<0) {printf("Erreur."); return;}
        if (premier(n)== 1) printf ("%d est premier", n);
        else printf("%d n\'est pas premier", n);
}
```

La récursivité

La récursivité est une autre façon que les boucles pour itérer.

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.

Exemple : n!= $n \times (n - 1)!$ ... Et on arrête le calcul à 0! = 1

Fonction récursive

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

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

Il faut nécessairement un ou plusieurs cas d'arrêt, sinon la fonction ne se terminerait pas.

```
int loop (int n){
        return (loop (n));
}
```

Exécution de la fonction récursive fact

```
Appel à fact(4)
.       4*fact(3) = ?
.       Appel à fact(3)
.       .       3*fact(2) = ?
.       .       Appel à fact(2)
.       .       .       2*fact(1) = ?
.       .       .       Appel à fact(1)
.       .       .       .       1*fact(0) = ?
.       .       .       .       Appel à fact(0)
.       .       .       .       Retour de la valeur 1
.       .       .       .       1*1
.       .       .       Retour de la valeur 1
.       .       .       2*1
.       .       Retour de la valeur 2
.       .       3*2
.       Retour de la valeur 6
.       4*6
Retour de la valeur 24
```

Pile d’exécution

La Pile d’exécution du programme en cours est un emplacement mémoire destiné à mémoriser les paramètres, les variables locales ainsi que les adresses de retour des fonctions en cours d’exécution.

Elle fonctionne selon le principe LIFO (Last-In-First-Out) : dernier entré premier sorti.

Attention ! La pile a une taille fixée, une mauvaise utilisation de la récursivité peut entraîner un débordement de pile.

Point terminal

Comme dans le cas d'une boucle, il faut un **cas d'arrêt** où l'on ne fait pas d'appel récursif.

```
procédure récursive(paramètres):
        si TEST_D'ARRET:
                instructions du point d'arrêt
        sinon
                instructions
                récursive(paramètres changés);  // appel récursif
                instructions
```

Récursivité terminale

On dit qu'une fonction est récursive terminale, si tout appel récursif est de la forme *return f(...)*

Autrement dit, la valeur retournée est directement la valeur obtenue par un appel récursif, sans qu'il n'y ait aucune opération sur cette valeur. Il n'y a ainsi rien à retenir sur la pile.

Quand une fonction est récursive terminale, on peut transformer l'appelle récursif en une boucle, sans utilisation de mémoire supplémentaire.

Exemple:

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

Devient :

```
int Algo(int n, int a) {
    while (n != 0) {
        a = n * a;
        n = n - 1;
    }
    return a;
}
```

Exemple

Ecrire une fonction récursive de calcul de factorielle et appeler la de la fonction main

```
#include <stdio.h>
int factorielle (int n);
int main() {
        int a, f;
        printf("Entrer un nombre entier positif : ");
        scanf("%d", &a);
        f = factorielle (a);
        if (f == -1)
                printf("La factorielle n'est pas definie pour les
                        entiers negatifs.");
        else
                printf("La factorielle de %d est %d", a, f);
        return 0;
}
int factorielle (int n) {
        if (n < 0) return -1;
        else
                if (n == 0) return 1;
                else return n * factorielle(n - 1);
}
```

Exercice 4

Ecrire une fonction récursive qui calcule le PGDC de deux nombres entiers et appeler de la fonction main

```
#include <stdio.h>
int pgcd (int a, int b);
int main() {
        int a, b;
        printf("Entrer deux nombres : ");
        scanf("%d", &a);
        scanf("%d", &b);
        if (a == 0 && b == 0)
                printf("Non defini");
        else
                printf("PGDC (%d,%d) = %d", a, b, pgcd(a,b));
        return 0;
}

int pgcd (int a, int b) {
        if (b == 0) return a;
        else return pgcd(b, a%b);
}
```

Exercice 5

Ecrire une fonction récursive qui calcule la suite Fibonacci (les premiers 20 éléments de la suite).

La suite de Fibonacci est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent. Elle commence généralement par les termes 0 et 1 (parfois 1 et 1) et ses premiers termes sont : 0, 1, 1, 2, 3, 5, 8, 13, 21, etc

$F_0 = 0$, $F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_n = F_{n-1} + F_{n-2}$

$\mathcal{F}_0$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_3$	$\mathcal{F}_4$	$\mathcal{F}_5$	$\mathcal{F}_6$	$\mathcal{F}_7$	$\mathcal{F}_8$	$\mathcal{F}_9$	$\mathcal{F}_{10}$	$\mathcal{F}_{11}$	$\mathcal{F}_{12}$	$\mathcal{F}_{13}$	$\mathcal{F}_{14}$	$\mathcal{F}_{15}$	$\mathcal{F}_{16}$	$\mathcal{F}_{17}$
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597

Exercice 5

```
#include <stdio.h>

int fibonacci (int n);

int main() {
        int i;
        for (i = 0; i< 20; i++) {
                printf("Fibonacci %d = %d\n", i, fibonacci(i));
        }
        return 0;
}
int fibonacci (int n) {
        if (n == 0 || n == 1) return n;
        else return fibonacci(n-1)+ fibonacci(n-2);
}
```