TD 5 — Fonctions et programmation modulaire

Comme dans la plupart des langages, on peut en C découper un programme en plusieurs fonctions. Une seule de ces fonctions est obligatoire : 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 ou s'appeler elle-même (dans ce dernier cas, on dit que la fonction est récursive).

Les deux premières parties de cet énoncé contiennent des rappels et explications sur les fonctions. La plupart des exercices sont regroupés dans la partie 3.

1 Rappels sur les fonctions

1.1 Fonctions sans valeur de retour

Certaines fonctions se contentent d'exécuter des instructions, mais ne renvoient pas de résultat. On parle alors de *procédures*.

1.1.1 Procédure sans paramètre

Voici un premier exemple :

```c
/* Prodédure affiche_bonjour */
void affiche_bonjour() {
    printf("Bonjour tout le monde !\n");
}

int main(void) {
    affiche_bonjour_anonyme(); // Appel de la procédure affiche_bonjour

    return EXIT_SUCCESS;
}
```

Le code commence par la fonction `main(void)`, qui contient un appel à la procédure `affiche_bonjour()`. Ce code est alors exécuté, puis on revient dans la fonction `main(void)`.
Dans l'exemple précédent, la procédure `affiche_bonjour()` ne reçoit aucun paramètre entre parenthèses : elle fait toujours la même chose. C'est simplement une façon d'organiser le code source différemment, et de séparer les choses pour mieux s'organiser.

1.1.2 Procédure avec paramètre

Il est possible de passer des valeurs en paramètres dans les procédures, comme pour les fonctions mathématiques. Par exemple :

```c
/* Prodédure void affiche_bonjour_personnalise */
void affiche_bonjour_personnalise(char* prenom) {
    printf("Bonjour %s !\n", prenom);
}

int main(void) {
    /* Appel de la procédure affiche_bonjour_personnalise, avec
       en argument la chaine de caractères "Luc" */
    affiche_bonjour_personnalise("Luc");

    return EXIT_SUCCESS;
}
```

Ici, la chaine `"Luc"` est passée à la procédure, et sera copiée dans la variable `char* prenom` (qui est un pointeur vers un `char`, donc en fait un tableau de `char` : c'est une chaine de caractères).

Abus de langage

Par abus de langage, on utilise parfois le terme « fonction » à la place du terme « procédure ».

1.2 Fonction avec valeur de retour

Les fonctions peuvent, comme en mathématiques, *renvoyer* un résultat. Par exemple :

```c
/* Fonction qui calcule et renvoie la moitié du flottant x */
float moitie(float x) {
    y = x/2;

    return y;
}

int main(void) {
    float a = 7.5;
    float b = -8.4;

    printf("La moitié de %f est %f.\n", a, moitie(a));
    printf("La moitié de %f est %f.\n", b, moitie(b));
    return EXIT_SUCCESS;
}
```

À la ligne 12, la `moitie(a)` appelle la fonction `flot moitié(float x)`. La valeur de la variable `a` est alors *copiée* dans l'argument `float x`. La fonction calcule `x/2` et renvoie le résultat avec l'instruction `return y;` à la ligne 5.

Type d'une fonction

En langage C, les fonctions ont un type, comme les variables. Le type d'une fonction est le *type de la valeur retournée*. C'est pourquoi la fonction `float moitie(float x)` est de type `float`.

Une procédure (qui ne renvoie rien) est de type `void`.

1.3 Exercice d'application

Exercice 1. **1)** Écrire une procédure qui reçoit en paramètre une variable de type `char` et qui affiche à l'écran s'il s'agit d'une majuscule, d'une minuscule ou d'un chiffre.

2) Modifier votre procédure pour qu'elle renvoie une variable de type `int`, qui vaut 1 si l'on a une minuscule, 2 si c'est une majuscule et 3 si c'est un chiffre.

Ordre des procédures et des fonctions

Pour le moment, les procédures et fonctions sont définies dans le fichier `main.c`. Leur code doit *obligatoirement* être placé avant le code de la fonction `main`, car le compilateur doit connaitre le code avant que la fonction ne soit appelée.

2 La programmation modulaire

Dès que l'on écrit un programme de taille importante ou destiné à être utilisé et maintenu par d'autres personnes, il est indispensable de se fixer un certain nombre de règles d'écriture. En particulier, il est nécessaire de fractionner le programme en plusieurs fichiers sources, que l'on compile séparément. Ces règles d'écriture ont pour objectifs de rendre un programme lisible, portable, réutilisable, facile à maintenir et à modifier.

2.1 Prototype d'une fonction

Le prototype d'une fonction est le résumé des informations importantes qui la concernent : son nom, le type de la valeur de retour et le type de ses éventuels arguments. Ainsi, les fonctions données en exemple dans la partie précédente on les prototypes suivants :

```c
void affiche_bonjour(); // Notez la présence du point-virgule à la fin
void affiche_bonjour_personnalise(char*);
float moitie(float x); /* Il est possible, mais pas obligatoire, de préciser
                          les noms des arguments en plus de leur type */
```

En utilisant les prototypes, il devient possible d'écrire les fonctions *dans n'importe quel ordre*, car le compilateur disposera alors des informations nécessaires. Cela permet de programmer sans se demander dans quel ordre les fonctions doivent être écrites. À la page 8 en annexe, vous trouverez un exemple de fichier `main.c` avec les prototypes des fonctions.

2.2 Fichiers d'en-tête

Ce découpage avec des fonctions et les prototypes facilite la programmation. Nous allons maintenant voir comment découper le code source d'un programme en *plusieurs fichiers indépendants*.

Par exemple, pour écrire un programme qui saisit deux entiers au clavier et affiche leur produit, on place le code de la fonction `int produit(int, int)` dans un fichier `produit.c`, qui sera ensuite inclus dans le fichier `main.c` avec la directive `#include`. De plus, on crée également un fichier nommé `produit.h`, qui contiendra *uniquement les prototypes* et les `#include` utilisés dans le fichier

`produit.c`.

À la fin, on obtient donc *trois fichiers* : `main.c`, `produit.c` et `produit.h`.

produit.h

```c
extern int produit(int, int); // Calcule et retourne le produit de deux entiers
```

produit.c

```c
#include produit.h // Inclusion du fichier produit.h avec les prototypes

int produit(int a, int b) { // Calcule et retourne le produit de deux entiers
    return a*b;
}
```

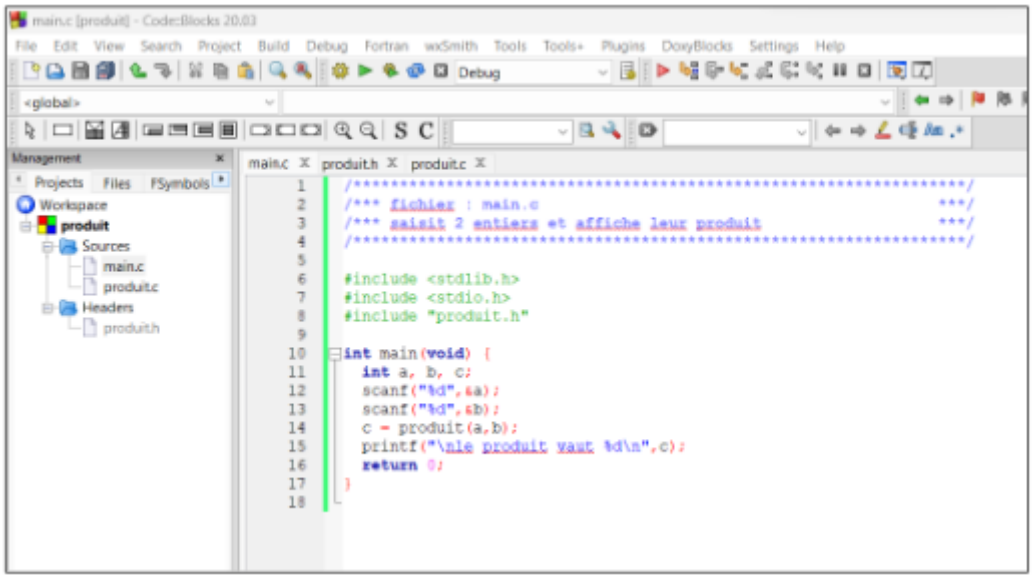
main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "produit.h" // Inclusion de produit.h pour la fonction int produit(int, int)

int main(void) {
    int a, b, c;
    scanf("%d",&a);
    scanf("%d",&b);
    c = produit(a,b); // Appel à la fonction produit dans le fichier produit.c
    printf("\nle produit vaut %d\n", c);
    return EXIT_SUCCESS;
}
```

2.3 Les projets sous `Code::Blocks`

L'IDE `Code::Blocks` facilite la gestion des codes sources séparés en plusieurs fichiers, avec la fonctionnalité « Projets ». Ainsi, le projet nommé `produits` sur `Code::Blocks` se présente comme suit :



Une dernière règle consiste à éviter les possibilités de double inclusion de fichiers en-tête. Pour cela, il est recommandé de définir une constante symbolique, habituellement appelée `NOM_H`, au début du fichier `nom.h` dont l'existence est précédemment testée. Si cette constante est définie, c'est que le

fichier `nom.h` a déjà été inclus. Dans ce cas, le préprocesseur ne le prend pas en compte. Sinon, on définit la constante et on prend en compte le contenu de `nom.h`.

En appliquant cette règle, le fichier `produit.h` de l'exemple précédent devient :

produit.h

```
#ifndef PRODUIT_H
#define PRODUIT_H

extern int produit(int, int); // Calcule et retourne le produit de deux entiers

#endif /* PRODUIT_H */
```

Programmation modulaire et taille du code source

Les exemples donnés ici sont très petits, avec des fichiers minuscules. L'intérêt de la programmation modulaire dans cette situation peut sembler limitée.

Mais en pratique, les projets contiennent souvent plusieurs milliers, et même dizaines de milliers, de lignes de codes. Il devient alors indispensable d'avoir convenablement découpé et organisé son code en modules et en fonctions.

3 Exercices de mise en pratique

Travail à rendre

Vous devrez déposer sur EPREL une archive `zip` nommée `R101_TD5_<NOMDEFAMILLE>.zip`, contenant un `Code::Blocks` contenant les réponse aux exercices. Ce projet devra contenir les différents fichiers `*.c` et `*.h` que vous aurez écrits.

Commencer par créer un projet `r101_td5.cbp` contenant un ficher `main.c`. Ce fichier ne devra contenir que la fonction `int main(void)`. Votre fonction `main` devra appeler les fonctions auxiliaires que vous allez programmer pour les tester.

Exercice 2 (Conversions de températures)**.** On rappelle la formule qui permet d'obtenir une tempéture en °C à partir d'une température e °F :

$$T_C = \frac{5}{9}(T_F - 32)$$

1) Créez les fichiers `temperatures.h` et `temperatures.c` et ajoutez-les à votre projet.

2) Quel est le prototype de la fonction qui convertit une température en °F vers une température en °C ? Ajoutez ce prototype à au fichier `temperatures.h` et son code dans le fichier `temperatures.c`.

3) Dans le fichier `main.c`, effectuez des appels à la fonction que vous venez de créer pour réaliser des tests.

4) De la même manière, écrivez une fonction qui effectue la conversion dans l'autre sens : d'une température en °C vers une température en °F.

Exercice 3 (Un peu de mathématiques)**.** Ajoutez les fichiers `calculs.h` et `calcul.c` à votre projet.

1) Écrivez une procédure qui reçoit un nombre entier positif en argument et affiche la liste de ses diviseurs. Vous placerez les prototypes et le code aux bons endroits.

2) Écrire une première fonction `carre` qui permet de calculer et retourner le carré d'une valeur flottante passée en argument.

3) Écrire une seconde fonction `cube` qui permet de calculer et retourner le cube d'une valeur flottante passée en argument. Cette fonction fera appel à la fonction `carre` pour produire le résultat.

Avant d'étudier l'exercice 4, nous avons besoin d'une notion qui relie pointeurs et fonctions. Pour ce faire, étudions un exemple. Nous souhaitons créer une fonction nommée `minMax` qui, étant donné un tableau de nombres passé en paramètre, renvoie *deux* résultats : le minimum et le maximum. En langage C, ce n'est pas possible car une fonction renvoie toujours *un seul résultat*. Une solution pour résoudre ce problème est de rajouter deux arguments à la fonction, qui sont des pointeurs vers deux variables, une contenant le minimum et l'autre le maximum que l'on calcule. Ainsi, en accédant à l'adresse de la variable, on peut la modifier. Ce mécanisme s'appelle le *passage par référence*.

ptrFct.c

```c
void minMax(int* tab, int taille, int* min, int* max) {
    *min = tab[0], *max = tab[0];

    for(int i=0 ; i<taille ; i++) {
        if(tab[i]<*min) *min = tab[i];
        if(tab[i]>*max) *max = tab[i];
    }
}
```

On rappelle que si `int* p_variable` est un pointeur vers une variable de type `int`, i.e. `p_variable` contient l'adresse d'une variable de type `int`, alors on accède à la *valeur pointée* via l'opération `*p_variable`.

Exercice 4 (Pointeurs et fonctions)**.** Ajoutez à votre projet les fichiers `ptrFct.c` et `ptrFct.h`, avec les inclusions placées aux bons endroits.

1) **a.** Intégrez le prototype de la fonction `minMax` au fichier `ptrFct.h`. Puis copiez son code dans le fichier `ptrFct.c`. Pourquoi cette fonction est-elle de type `void` ?

   b. Dans le fichier `main.c`, créez un tableau et les variables nécessaires pour tester la fonction précédente.

2) Créez dans le même fichier une fonction qui échange les valeurs de deux flottants reçus en paramètres. A-t-on besoin d'utiliser un passage par référence ? Pourquoi ?

3) Écrire une fonction qui recherche dans un tableau `char tab[]` de taille `int n` un caractère `char c`. Son prototype doit être :

   ptrFct.h

   ```c
   void chercherVal(char tab[], int n, char c, int *pos, int *nbOcc);
   ```

   Dans `int *pos`, la fonction sauvegarde l'indice de la dernière apparition du caractère `char c` et `-1` si le caractère n'a pas été trouvé. Dans `int nbOcc`, elle sauvegarde le nombre d'occurrences de `char c` dans le tableau.

Exercice 5 (Fonctions récursives — Factorielle)**.** On rappelle que si $n \in \mathbb{N}^*$, la factorielle de n, notée $n!$, est définie par :

$$n! = \prod_{k=1}^{n} k,$$

avec la convention $0! = 1$.

1) Ajoutez à votre projet les fichiers `fctRecursives.h` et `fctRecursives.c`.

2) Écrivez une fonction `int factorielleIterative(int n);` qui reçoit un entier positif en paramètre et renvoie sa factorielle au moyen d'une boucle itérative.

3) Écrire une version récursive de votre fonction, qui effectue le même calcule, mais de façon récursive. *On rappelle qu'une fonction est dite* récursive *lorsqu'elle s'appelle elle-même.*
On notera que pour tout $n \in \mathbb{N}^*$, $n! = n(n-1)!$.

Exercice 6 (Fonctions récursives — Suite de Fibonacci)**.** On appelle *suite de Fibonacci* la suite $(u_n)_{n\in\mathbb{N}}$ définie par :

$$u_0 = u_1 = 1 \quad \text{et} \quad \forall n \geq 2, u_n = u_{n-1} + u_{n+2}$$

1) En restant dans les fichiers `fctRecursives.h` et `fctRecursives.c`, proposez une fonction itérative qui reçoit un entier n en paramètre et renvoie u_n.

2) Proposez une version récursive de cette fonction.

3) Comparez les temps d'exécution des deux algorithmes pour de grandes valeurs de n. Que remarquez-vous ?

Annexe

main.c

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

/* Prototypes des fonctions */
void affiche_bonjour(); //
void affiche_bonjour_personnalise(char* prenom);
float moitie(float x);

int main(void) {
    float a = 7.5;
    float b = -8.4;

    affiche_bonjour("Luc");
    affiche_bonjour_personnalise("Luc");

    printf("La moitié de %f est %f.\n", a, moitie(a));
    printf("La moitié de %f est %f.\n", b, moitie(b));
    return EXIT_SUCCESS;
}

/* Le code de la procédure affiche_bonjour est après la fonction main,
   ce qui est rendu possible par la présence des prototypes */
void affiche_bonjour() {
    printf("Bonjour tout le monde !\n");
}

void affiche_bonjour_personnalise(char* prenom) {
    printf("Bonjour %s !\n", prenom);
}

/* Fonction qui calcule et renvoie la moitié du flottant x */
float moitie(float x) {
    y = x/2;

    return y;
}
```