Extrait du référentiel : BTS Systèmes Numériques option A (Informatique et Réseaux)		Niveau(x)
S4. Développement logiciel S4.4. Programmation procédurale	Manipulations de données (« quoi ») en pseudo-langage et/ou en langage C Transcription d'algorithmes (« comment ») en pseudo-langage et/ou en langage C Développement de programmes « console » avec gestion des arguments de la ligne de commande	4 4 3

Objectifs du cours :

- Les fonctions :
  - déclaration et définition d'une fonction
  - le type de retour
  - les paramètres
  - les arguments
  - les prototypes
  - les classes de stockage
- Exercices

Beaucoup de langages distinguent deux sortes de sous-programmes : les **fonctions** et les **procédures**. L'appel d'une fonction est une expression, tandis que l'appel d'une procédure est une instruction. Ou, si on préfère, l'appel d'une fonction renvoie un résultat, alors que l'appel d'une procédure ne renvoie rien.

Le concept de fonction ne vous est pas inconnu : **`printf()`**, **`scanf()`**, et **`main()`** sont des fonctions.

LES FONCTIONS

Une fonction est :

- une suite d'instructions ;
- identifiée à l'aide d'un nom ;
- a vocation à être exécutée à plusieurs reprises ;
- rassemble des instructions qui permettent d'effectuer une tâche précise (comme afficher du texte à l'écran, calculer la racine carrée d'un nombre, …).

Exemple :

```
#include <stdio.h>

int main(void)
{
    int a;
    int b;

    printf("Entrez deux nombres : ");
    scanf("%d %d", &a, &b);

    int min = (a < b) ? a : b;

    for (int i = 2; i <= min; ++i)
        if (a % i == 0 && b % i == 0)
        {
            printf("Le plus petit diviseur de %d et %d est %d\n", a, b, i);
            break;
        }

    return 0;
}
```

Ce code permet de calculer le plus petit commun diviseur de deux nombres donnés. Imaginons à présent que nous souhaitions faire la même chose, mais avec deux paires de nombres. Le code ressemblerait alors à ceci.

```
#include <stdio.h>

int main(void)
{
    int a;
    int b;

    printf("Entrez deux nombres : ");
    scanf("%d %d", &a, &b);

    int min = (a < b) ? a : b;

    for (int i = 2; i <= min; ++i)
        if (a % i == 0 && b % i == 0)
        {
            printf("Le plus petit diviseur de %d et %d est %d\n", a, b, i);
            break;
        }
```

```
    printf("Entrez deux autres nombres : ");
    scanf("%d %d", &a, &b);
    min = (a < b) ? a : b;

    for (int i = 2; i <= min; ++i)
        if (a % i == 0 && b % i == 0)
        {
            printf("Le plus petit diviseur de %d et %d est %d\n", a, b, i);
            break;
        }

    return 0;
}
```

Comme vous pouvez le voir, nous devons recopier les instructions de calcul deux fois, ce qui est assez dommage et qui plus est source d'erreurs. C'est ici que les fonctions entre en jeu en nous permettant par exemple de rassembler les instructions dédiées au calcul du plus petit diviseur commun en un seul point que nous solliciterons autant de fois que nécessaire.

Il est aussi possible ici d'utiliser une boucle pour éviter la répétition, mais l'exemple aurait été moins parlant.

DÉCLARATION ET DÉFINITION D'UNE FONCTION

Toute fonction est typée et doit l'être.
Toute fonction retourne donc un résultat et nécessite lors de son appel une « left_value ».
Le type de la fonction correspond au type du résultat retourné par cette fonction.
Une fonction en langage C ne peut retourner qu'un seul type et un seul résultat.
Le type du paramètre de retour d'une fonction ne peut être qu'un type simple (int, float, double, etc...) ou de type pointeur.
Une fonction peut recevoir ou non des arguments en paramètres, c'est-à-dire des données essentielles pour la réalisation de cette dernière.

Une fonction n'est pas obligée de recevoir des arguments en paramètres. Dans ce cas, lors de la déclaration de cette fonction, la liste des types des arguments attendus sera vide ou remplacée par le mot clé **void**.
Une fonction peut être appelée et utilisée autant de fois qu'il est nécessaire.

Une fonction qui ne retourne pas de résultat est appelée **procédure**, ce n'est plus une fonction. Dans ce cas, cette fonction appelée procédure n'est pas typée. On lui indique le mot clé **void** lors de sa déclaration. Mais elle est en mesure d'accepter des arguments.

Par abus de langage, les procédures sont appelées également fonctions par les programmeurs.
La déclaration d'une fonction ou d'une procédure consiste à donner le type de la fonction, son nom, et le type des arguments reçus en paramètres, tout cela en dehors de la fonction **main( )**.

```
< type de la fonction > < nom de la fonction > (type des arguments)
```

Exemples :

```
float calcul(int); /* fonction calcul récupérant un entier en argument et
retournant un réel */
```

```
void affiche(void); /* Aucun argument reçu en paramètre, cest une procédure */
```

Le compilateur prend en compte les fonctions dans l'ordre où elles apparaissent.

Par conséquent, si une fonction est utilisée avant d'être déclarée le compilateur la considère comme une fonction retournant un **int** valeur par défaut. Il est donc nécessaire de déclarer ou de prototyper les fonctions avant leur définition et leur utilisation.

Le prototypage des fonctions doit être en accord avec leur définition. C'est-à-dire que les types des paramètres définis lors de la définition de la fonction doivent correspondre à la liste des types déclarés dans le prototype de la fonction.

La **définition d'une fonction** (son corps) consiste tout simplement à écrire son contenu, son code source constitué de variables et d'instructions. Toute fonction a forcément un rôle bien établi (exemple : calcul d'une superficie).

C'est la fonction **return(…)** qui permet le retour de la valeur résultat de la fonction. Cette valeur peut être le contenu d'une variable où le résultat d'une expression.
La fonction **return(…)** peut se trouver n'importe où dans la fonction, et pas forcément à la fin de son code.

Exemple :

```
int somme(int x, int y)
{
int z;
z = x + y;
return(z);
}
```

Exemple : création d'une fonction « **bonjour()** »

```
#include <stdio.h>

void bonjour(void)
{
    printf("Bonjour !\n");
}

int main(void)
{
    bonjour();
    return 0;
}
```

Comme nous pouvons le voir, la fonction se nomme « **bonjour »** et est composée d'un appel à **printf()**. Reste les deux mots-clés **void** :

- dans le cas du type de retour, il spécifie que la fonction ne retourne rien ;
- dans le cas des paramètres, il spécifie que la fonction n'en reçoit aucun (cela se manifeste lors de l'appel : il n'y a rien entre les parenthèses).

LE TYPE DE RETOUR

Le type de retour permet d'indiquer deux choses : si la fonction retourne une valeur et le type de cette valeur.

Exemple :

```
#include <stdio.h>

int trois(void)
{
    return 3;
}

int main(void)
{
    printf("Retour : %d\n", trois());
    return 0;
}
```

Résultat : **Retour : 3**

Dans l'exemple ci-dessus, la fonction **trois()** est définie comme retournant une valeur de type **int**. **return** arrête l'exécution de la fonction courante et provoque un retour (techniquement, un saut) vers l'appel à cette fonction qui se voit alors attribuer la valeur de retour (s'il y en a une). Autrement dit, dans notre exemple, l'instruction **return 3** stoppe l'exécution de la fonction **trois()** et ramène l'exécution du programme à l'appel qui vaut désormais 3, ce qui donne finalement **printf ("Retour : %d\n", 3)**.

LES PARAMÈTRES

Un paramètre sert à fournir des informations à la fonction lors de son exécution. La fonction **printf()** par exemple récupère ce qu'elle doit afficher dans la console à l'aide de paramètres. Ceux-ci sont définis de la même manière que les variables si ce n'est que les définitions sont séparées par des virgules.

```
type nom(type paramètre1, type paramètre2, ...)
{
    /* Corps de la fonction */
}
```

Il est possible d'utiliser un maximum de 127 paramètres, toutefois il est conseillé de limiter à le nombre à 5 afin de conserver un code concis et lisible.

LES PARAMÈTRES ET LES ARGUMENTS

Il est important de préciser qu'un paramètre est propre à une fonction, il n'est pas utilisable en dehors de celle-ci.

Voici un exemple explicite à ce sujet.

Exemple :

```
#include <stdio.h>

void fonction(int nombre)
{
    ++nombre;
    printf("Variable nombre dans 'fonction' : %d\n", nombre);
}

int main(void)
{
    int nombre = 5;

    fonction(nombre);
    printf("Variable nombre dans 'main' : %d\n", nombre);
    return 0;
}
```

Résultat :

```
Variable nombre dans 'fonction' : 6
Variable nombre dans 'main' : 5
```

Comme vous le voyez, les deux variables nombre sont bel et bien distinctes. En fait, lors d'un appel de fonction, vous spécifiez des **arguments** à la fonction appelée. Ces arguments ne sont rien d'autres que des expressions dont les résultats seront ensuite affectés aux différents **paramètres** de la fonction.

Notez bien cette différence car elle est très importante : **un argument est une expression alors qu'un paramètre est une variable**.

Autrement dit, la valeur de la variable nombre de la fonction **`main()`** est passée en argument à la fonction **`fonction()`** et est ensuite affectée au paramètre nombre. La variable nombre de la fonction **`main()`** n'est donc en rien modifiée.

LES PROTOTYPES

Jusqu'à présent, nous avons toujours défini nos fonctions avant la fonction **`main()`**. Cela paraît de prime abord logique (nous définissons la fonction avant de l'utiliser), cependant cela est surtout indispensable. En effet, si nous déplaçons la définition après la fonction **`main()`**, le compilateur se retrouve dans une situation délicate : il est face à un appel de fonction dont il ne sait rien (nombres d'arguments, type des arguments et type de retour).
Le compilateur va considérer que la fonction retourne une valeur de type **`int`** et qu'elle reçoit un nombre indéterminé d'arguments.

Toutefois, si cette décision à l'avantage d'éviter un arrêt de la compilation, elle peut en revanche conduire à des problèmes lors de l'exécution si cette supposition du compilateur s'avère inadéquate. Il serait pratique de pouvoir définir les fonctions dans l'ordre que nous souhaitons sans se soucier de qui doit être défini avant qui.

Pour résoudre ce problème, il est possible de déclarer une fonction à l'aide d'un **prototype**. Celui-ci permet de spécifier le type de retour de la fonction, son nombre d'arguments et leur type, mais ne comporte pas le corps de cette fonction. La syntaxe d'un prototype est la suivante.

```
type nom(paramètres);
```

Exemple :

```
#include <stdio.h>

void bonjour(void);

int main(void)
{
    bonjour();
    return 0;
}

void bonjour(void)
{
    printf("Bonjour !\n");
}
```

N'oubliez pas le point virgule qui est obligatoire à la fin du prototype !

Étant donné qu'un prototype ne comprend pas le corps de la fonction qu'il déclare, il n'est pas obligatoire de préciser le nom des paramètres de celles-ci.

LES VARIABLES GLOBALES

Il arrive parfois que l'utilisation de paramètres ne soit pas adaptée et que des fonctions soient amenées à travailler sur des données qui doivent leur être communes. Prenons un exemple simple : vous souhaitez compter le nombre d'appels de fonction réalisé durant l'exécution de votre programme. Ceci est impossible à réaliser, sauf à définir une variable dans la fonction **`main()`**, la passer en argument de chaque fonction et de faire en sorte que chaque fonction retourne sa valeur augmentée de un, ce qui est très peu pratique.

À la place, il est possible de définir une variable dite « **globale** » qui sera utilisable par toutes les fonctions. Pour définir une variable globale, il vous suffit de définir une variable en dehors de tout bloc, autrement dit en dehors de toute fonction.

Exemple :

```
#include <stdio.h>

void fonction(void);

int appels = 0;
void fonction(void)
```

```
{
    ++appels;
}

int main(void)
{
    fonction();
    fonction();
    printf("Ce programme a realise %d appel(s) de fonction\n", appels);
    return 0;
}
```

Résultat :

```
Ce programme a realise 2 appel(s) de fonction
```

Comme vous le voyez, nous avons simplement placé la définition de la variable **appels** en dehors de toute fonction et avant toute définition de fonction de sorte qu'elle soit partagée entre elles.

Le terme « global » est en fait un peu trompeur étant donné que la variable n'est pas globale au programme, mais tout simplement disponible pour toutes les fonctions du fichier dans lequel elle est située. Ce terme est utilisé en opposition aux paramètres et variables des fonctions qui sont dites « **locales** ».

N'utilisez les variables globales que lorsque cela vous paraît vraiment nécessaire. Ces dernières étant utilisables dans un fichier entier (voire dans plusieurs, nous le verrons un peu plus tard), elles ont tendances à rendre la lecture du code plus difficile.

LES CLASSES DE STOCKAGE

Les variables locales et les variables globales ont une autre différence de taille : leur **classe de stockage**. La classe de stockage détermine (entre autre) la **durée de vie** d'un « objet », c'est-à-dire le temps durant lequel celui-ci existera en mémoire.

Classe de stockage automatique
Les **variables locales** sont par défaut de classe de stockage **automatique**. Cela signifie qu'elles sont allouées automatiquement à chaque fois que le bloc auquel elles appartiennent est exécuté et qu'elles sont détruites une fois son exécution terminée.

Exemple :

```
int fonction(int a, int b)
{
    int min = (a < b) ? a : b;

    for (int i = 2; i <= min; ++i)
        if (a % i == 0 && b % i == 0)
            return i;

    return 0;
}
```

À chaque fois que la fonction **`fonction()`** est appelée, les variables **`a`**, **`b`**, **`min`** sont allouées en mémoire vive et détruites à la fin de l'exécution de la fonction. La variable **`i`** quant à elle est allouée au début de la boucle for et détruite à la fin de cette dernière.

Classe de stockage statique

Les **variables globales** sont toujours de classe de stockage **statique**. Ceci signifie qu'elles sont allouées au début de l'exécution du programme et sont détruites à la fin de l'exécution de celui-ci. En conséquence, elles conservent leur valeur tout au long de l'exécution du programme. Également, à l'inverse des autres variables, celles-ci sont initialisées à zéro si elles ne font pas l'objet d'une initialisation. L'exemple ci-dessous est donc correct et utilise deux variables valant zéro.

Exemple :

```
#include <stdio.h>

int a;
double b;

int main(void)
{
    printf("%d, %f\n", a, b);
    return 0;
}
```

Résultat :

```
0, 0.000000
```

Modification de la classe de stockage

Il est possible de modifier la classe de stockage d'une variable automatique en précédant sa définition du mot-clé **`static`** afin d'en faire une variable statique.

```
#include <stdio.h>

int compteur(void)
{
    static int n;

    return ++n;
}

int main(void)
{
    compteur();
    printf("n = %d\n", compteur());
    return 0;
}
```

Résultat :

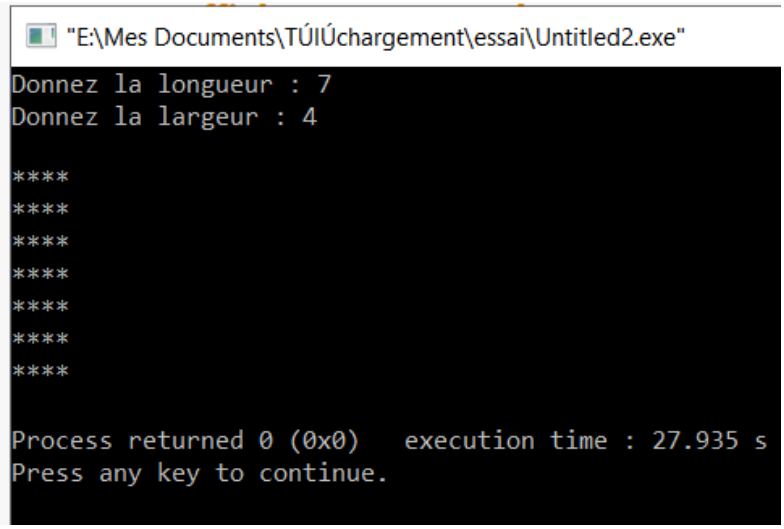
```
n = 2
```

EXERCICES

EXERCICE N°1

Question

Réalisez un programme permettant d'afficher un rectangle dans la console. Voici ce que devra donner l'exécution de votre programme.



Affichage de la console

EXERCICE N°2

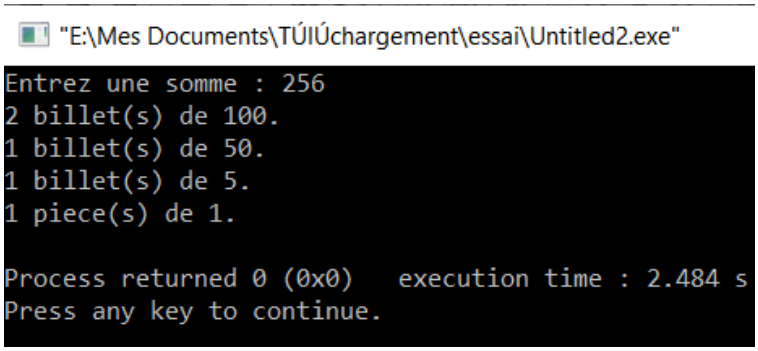
Question

Réalisez un programme qui reçoit en entrée une somme d'argent et donne en sortie la plus petite quantité de coupures nécessaires pour reconstituer cette somme.

Pour cet exercice, vous utiliserez les coupures suivantes :

- des billets de 100€ ;
- des billets de 50€ ;
- des billets de 20€ ;
- des billets de 10€ ;
- des billets de 5€ ;
- des pièces de 2€ ;
- des pièces de 1€ ;

Ci dessous un exemple de ce que devra donner votre programme une fois terminé.



Affichage de la console