

TP Programmation C/C++

Les fonctions

© 2017 tv <tvaira@free.fr> v.1.1

Sommaire

Les fonctions	2
Programmation procédurale	2
Fonction vs Procédure	4
Déclaration de fonction	5
Définition de fonction	6
Appel de fonction	6
Programmation modulaire	8
Passage de paramètre(s)	9
Passage par valeur	9
Passage par adresse	10
Passage par référence	11
Valeur de retour	12
Nommer une fonction	13
La fonction <code>main</code>	14
Intérêt des fonctions par l'exemple	16
Surcharge de fonctions	19
Paramètre par défaut	20
Fonctions <code>inline</code>	21
Fonctions statiques	21
Nombre variable de paramètres	22
Pointeur vers une fonction	23
Exercices : Les joies du code	23
Motif rectangulaire	23
Banquet municipal	24
Journée des cadeaux	28
Jour de marché	29
Recensement	29

Les objectifs de ce tp sont de comprendre et mettre en oeuvre la notion essentielle de fonction. Certains exercices sont extraits du site www.france-ioi.org.

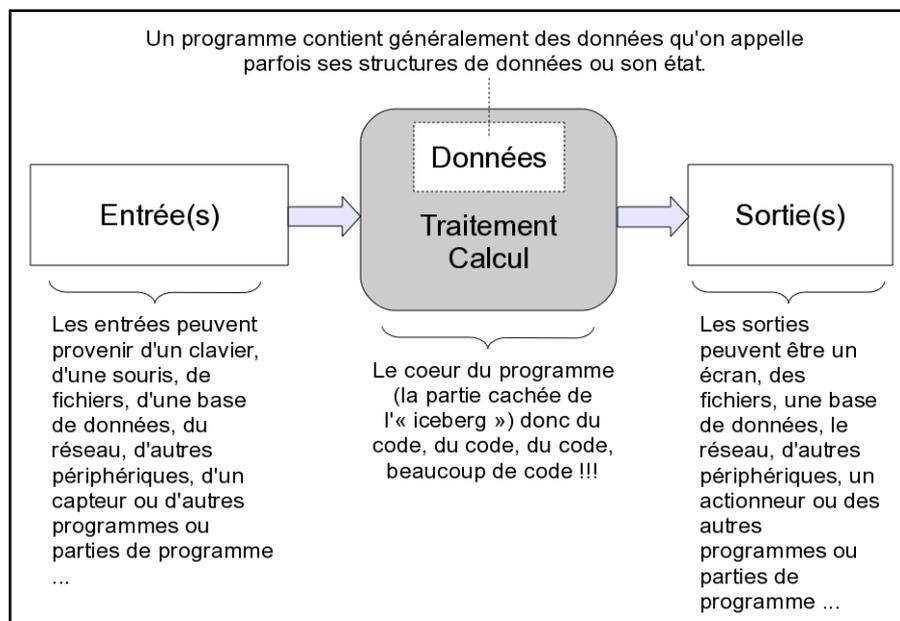
Les critères d'évaluation de ce TP sont :

- le minimum attendu : on doit pouvoir fabriquer un exécutable et le lancer !
- le programme doit énoncer ce qu'il fait et faire ce qu'il énonce !
- le respect des noms de fichiers
- le nommage des fonctions et des variables, l'utilisation de constantes
- une fonction doit énoncer ce qu'elle fait et faire seulement ce qu'elle énonce !
- les fonctions ne dépassent pas 15 lignes
- le code est fonctionnel
- la simplicité du code

Les fonctions

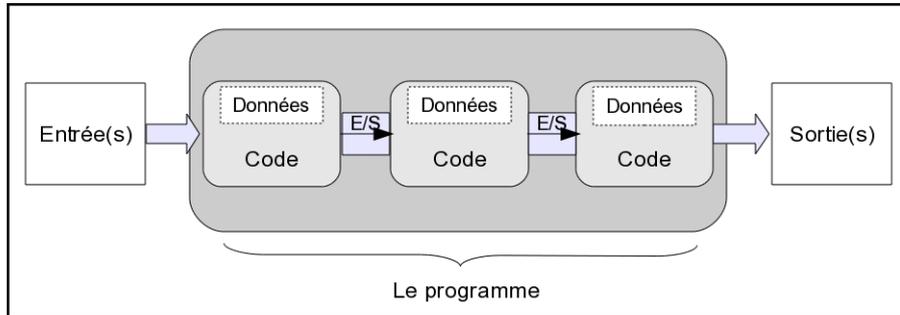
Programmation procédurale

D'un certain point de vue, un programme informatique ne fait jamais rien d'autre que **traiter des données**. Comme on l'a déjà vu, un programme accepte des entrées et produit des sorties :



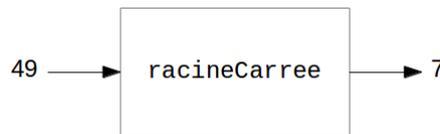
▮ *Un traitement est tout simplement l'action de produire des sorties à partir d'entrées. Les entrées dans une partie de programme sont souvent appelées des **arguments** (ou **paramètres**) et les sorties d'une partie de programme des **résultats**.*

La majeure partie du travail d'un programmeur est : comment exprimer un programme sous la forme d'un ensemble de parties (des sous-programmes) qui coopèrent et comment peuvent-elles partager et échanger des données ?



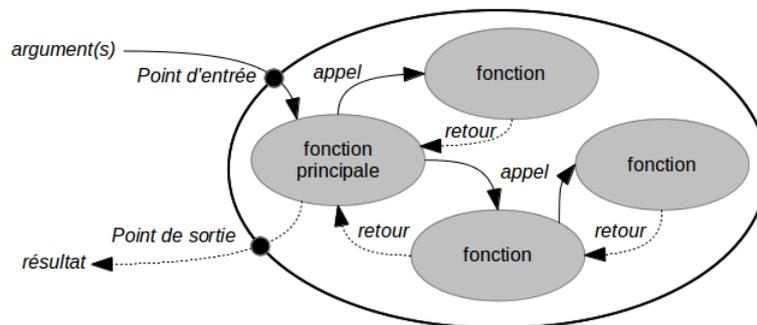
Par parties de programme (ou de code), on entend des entités (ou modules) comme une **fonction** produisant un résultat à partir d'un ensemble d'arguments en entrée.

☞ Exemple : un traitement comme produire le résultat (sortie) 7 à partir de l'argument (entrée) 49 au moyen de la fonction *racineCarree*.



En langage C, on pratique la **programmation procédurale**. La programmation procédurale se fonde sur le concept d'appel de **procédure**. Une procédure, aussi appelée **routine**, **sous-routine** ou **fonction**, contient simplement une série d'étapes à réaliser. Un **appel** de procédure (ou de fonction) déclenche l'exécution de la série d'instructions qui la compose.

En programmation procédurale, un programme n'est plus une simple séquence d'instructions mais un ensemble de sous-programmes (en C, des fonctions) s'appelant entre eux :



Le déroulement d'un programme est le suivant :

- L'exécution du programme commence toujours par l'exécution de la fonction principale (la fonction `main()` en C/C++)
- L'appel à une fonction permet de déclencher son exécution, en interrompant le déroulement séquentiel des instructions de la fonction principale
- Le déroulement des instructions de la fonction principale reprend, dès que la fonction appelée est terminée, à l'instruction qui suit l'appel

☞ En C++, on aura la possibilité de développer un programme en **Programmation Orientée Objet (POO)**.

⇒ Décomposer un programme en **fonctions** conduit souvent à diminuer la complexité d'un problème et permet de le résoudre plus facilement.

L'utilisation des fonctions permettra :

- d'éviter de répéter plusieurs fois les mêmes lignes de code : ceci simplifie le code, améliore la lisibilité et facilite la maintenance.
- de généraliser certaines parties de code : la décomposition en fonction permettra la réutilisation du code dans d'autres contextes (cf. bibliothèque logicielle).

On décomposera un programme en fonctions parce que procéder ainsi :

- rend le traitement distinct du point de vue logique
- rend le programme plus clair et plus lisible
- permet d'utiliser la fonction à plusieurs endroits dans un programme (à chaque fois qu'on en a besoin)
- facilitera les tests (on simulera des entrées et on comparera le résultat obtenu à celui attendu)

Bonnes pratiques :

- les programmes sont généralement plus facile à écrire, à comprendre et à maintenir lorsque chaque fonction réalise une SEULE ACTION logique et bien évidemment celle qui correspond à son nom ;
- on limitera la taille des fonctions à une valeur comprise entre **10 à 15 lignes maximum** ;
- les fonctions se concentrent sur le traitement qu'elles doivent réaliser et n'ont pas (la plupart du temps) à réaliser la saisie de leurs entrées et l'affichage de leur sortie. On évitera le plus possible d'utiliser des saisies et des affichages dans les fonctions pour permettre notamment leur ré-utilisation.

Fonction vs Procédure

⇒ Il existe deux types de sous-programmes :

- Les **fonctions**
 - Sous-programme qui retourne **une et une seule valeur** : permet de ne récupérer qu'**un résultat**.
 - Par convention, ce type de sous-programme ne devrait pas interagir avec l'environnement (écran, utilisateur).
- Les **procédures**
 - Sous-programme qui ne retourne **aucune valeur** : permet de produire **0 à n résultats**
 - Par convention, ce type de sous-programme peut interagir avec l'environnement (écran, utilisateur).

☞ *Cette distinction ne se retrouve pas dans tous les langages de programmation ! Le C/C++ n'admet que le concept de fonction qui servira à la fois pour écrire des fonctions et des procédures.*

Pour les fonctions en C/C++, il faut distinguer :

- la **déclaration** qui est une instruction fournissant au compilateur un certain nombre d'informations concernant une fonction (qui déclare son existence). Il existe une forme recommandée dite **prototype** :
`int plus(int, int);` ← fichier en-tête (.h)
- la **définition** qui revient à écrire le **corps** de la fonction (qui définit donc les traitements effectués dans le bloc {} de la fonction)
`int plus(int a, int b) { return a + b; }` ← fichier source (.c ou .cpp)
- l'**appel** qui est son utilisation. Il doit correspondre à la déclaration faite au compilateur qui le vérifie.
`int res = plus(2, 2);` ← fichier source (.c ou .cpp)

⚠ La définition d'une fonction tient lieu de déclaration. Mais elle doit être "connue" avant son utilisation (un appel). Sinon, le compilateur `gcc` générera un message d'avertissement (*warning*) qui indiquera qu'il a lui-même fait une déclaration implicite de cette fonction (ce n'est pas une bonne chose). Par contre, le compilateur `g++` générera une erreur : `'...' was not declared in this scope`.

☞ C/C++ supporte la **récurtivité** : c'est une technique qui permet à une fonction de s'auto-appeler. La récurtivité est une manière simple et élégante de résoudre certains problèmes algorithmiques, notamment en mathématique, mais cela ne s'improvise pas !

Déclaration de fonction

Une fonction se caractérise par :

- son **nom** (un identificateur)
- sa **liste de paramètre(s)** : le nombre et le type de paramètre(s) (la liste peut être vide)
- son **type de retour** (un seul résultat)

⚠ Comme une procédure ne retourne aucune valeur, son type de retour sera `void`.

Ces informations sont communément appelées le **prototype** de la fonction :

```
// Le prototype de la fonction calculeNombreDeSecondes :
int calculeNombreDeSecondes(int heures, int minutes, int secondes)

// Soit :
// - son nom : calculeNombreDeSecondes
// - sa liste de paramètre(s) : elle reçoit 3 int
// - son type de retour : int
```

C'est ce qu'il est nécessaire de connaître pour appeler une fonction.

Quand une fonction n'est pas encore définie, il est possible de **déclarer** son existence afin de pouvoir l'appeler. Il faut pour cela indiquer son prototype, suivi d'un point-virgule :

```
// La déclaration de la fonction calculeNombreDeSecondes :
int calculeNombreDeSecondes(int heures, int minutes, int secondes);
```

☞ Les déclarations de fonction sont placées dans des fichiers d'en-tête (*header*) d'extension `.h`. Si vous voulez utiliser (i.e. appeler) une fonction, il faudra donc inclure le fichier d'en-tête correspondant (`#include`). Un fichier d'en-tête regroupe un ensemble de déclarations.

```
#ifndef TEMPS_H /* si l'étiquette TEMPS_H n'est pas défini */
#define TEMPS_H /* alors on définit l'étiquette TEMPS_H */

int calculeNombreDeSecondes(int heures, int minutes, int secondes);

#endif /* fin si TEMPS_H */
```

Le fichier d'en-tête temps.h

⊛ Le langage C n'interdit pas l'inclusion multiple de fichiers d'en-tête mais il n'accepte pas toujours de déclarer plusieurs fois la même chose ! Par précaution, il faut donc s'assurer par des directives de pré-compilation (`#ifndef`, `#define` et `#endif`) que l'inclusion du fichier sera unique. Les directives de pré-compilation (ou préprocesseur) commencent toujours par un dièse (`#`). Ce ne sont donc pas des instructions du langage C.

Si vous voulez utiliser (i.e. appeler) une fonction, il faudra donc inclure le fichier d'en-tête correspondant :

```
#include "temps.h"

// Vous pouvez maintenant utiliser (appeler) la fonction calculeNombreDeSecondes :
int s = calculeNombreDeSecondes(1, 0, 0);
```

☞ Si vous créez vos propres fichiers d'en-tête, il est conseillé d'indiquer le nom du fichier entre guillemets ("*temps.h*") dans la directive de pré-compilation `#include`. Vous pourrez indiquer le chemin où se trouve vos propres fichiers d'en-tête avec l'option `-I` du compilateur `gcc/g++`. On n'utilisera pas les délimiteurs `<>` (`#include <stdio.h>`) qui sont réservés pour les fichiers d'en-tête systèmes. Dans ce cas, le compilateur connaît les chemins d'installation de ces fichiers.

```
$ ls
main.c temps.c include/

$ ls include/
temps.h

$ gcc -I./include -c temps.c
$ gcc -I./include -c main.c
```

Définition de fonction

La **définition** d'une fonction revient à écrire le **corps** de la fonction dans le bloc `{}`. Cela définira la suite d'instructions qui sera exécutée à chaque appel de la fonction.

```
#include "temps.h"

// La définition de la fonction calculeNombreDeSecondes :
int calculeNombreDeSecondes(int heures, int minutes, int secondes)
{
    return ((heures*3600) + (minutes*60) + secondes);
}
```

⊗ Attention à ne pas mettre de point-virgule à la fin du prototype ! Si vous déclarez votre propre fonction, il vous faudra absolument la définir si vous voulez passer l'étape de l'édition de lien (`ld`). Sinon, vous obtiendrez une erreur : `undefined reference`.

Pour que la fonction puisse effectivement retourner une valeur, il faut qu'elle contienne une instruction composée du mot-clé `return` et de la valeur que l'on veut retourner. L'instruction `return` quitte la fonction et transmet la valeur qui suit au programme appelant.

⚡ Dans le cas des fonctions dont le type de retour est `void`, il est également possible d'utiliser l'instruction `return`. Elle n'est alors suivie d'aucune valeur et a simplement pour effet de quitter la fonction immédiatement.

Appel de fonction

Un appel à une fonction signifie qu'on lui demande d'exécuter son traitement :

```
// Appel de la fonction calculeNombreDeSecondes :
int s = calculeNombreDeSecondes(1, 0, 0);
```

Soit la fonction `carre()` ci-dessous :

```
// Déclaration :  
// Calcule et retourne le carré d'un nombre  
int carre(int); // le nom du paramètre n'est pas obligatoire  
  
// Définition :  
// Calcule et retourne le carré d'un nombre  
int carre(int x) // le nom du paramètre est obligatoire  
{  
    // le paramètre x est une variable locale à la fonction  
    return x*x;  
}
```

L'appel doit correspondre à la déclaration faite au compilateur qui le vérifie :

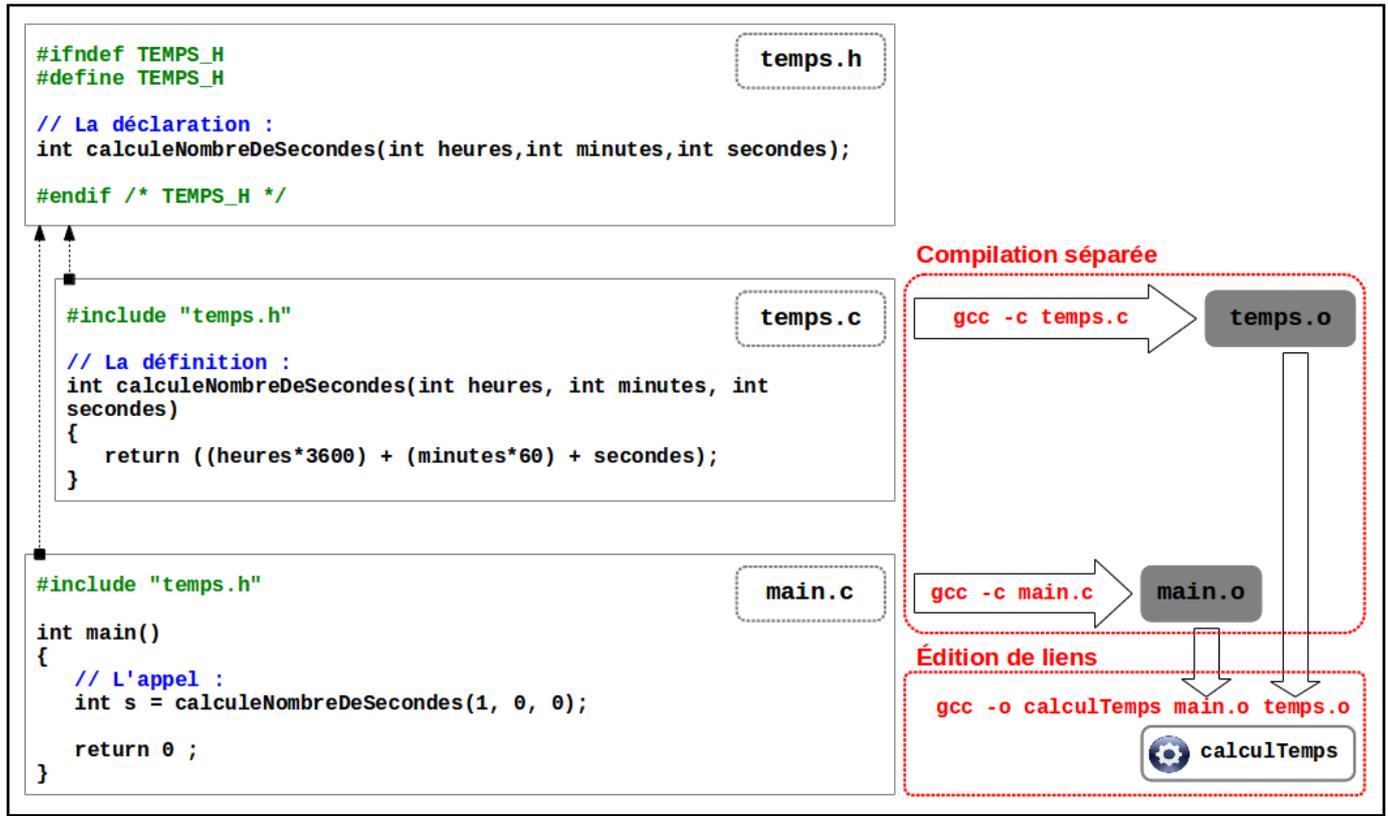
```
// Principe : on n'est pas obligé d'utiliser le résultat de retour mais on doit donner à la  
    fonction exactement les arguments qu'elle exige  
  
int c5 = carre(2); // Correct  
  
int valeur = 2;  
int c6 = carre(valeur); // Correct (valeur est copié dans x)  
  
// Mais on peut faire des erreurs :  
  
int c1 = carre(); // Erreur : pas assez d'argument  
  
int c2 = carre; // Erreur : parenthèses manquantes  
  
int c3 = carre(1, 2); // Erreur : trop d'arguments  
  
int c4 = carre("deux"); // Erreur : mauvais type d'argument car un int attendu  
  
// Attention :  
carre(2); // probablement une erreur car la valeur retour n'est pas utilisée  
  
// Soyez vigilant :  
int x = 2;  
int c7 = carre(x); // Correct (x est copié dans x)  
// après l'appel : x contient toujours 2 et c7 contient 4
```

Une fonction peut être utilisée :

- comme une **opérande** dans une expression, à partir du moment où il y a concordance de types;
- comme une **instruction**.

Programmation modulaire

Le découpage en fonctions d'un programme permet la programmation modulaire :



Par la suite, un utilitaire comme `make` permettra d'automatiser la fabrication de programme. Il utilise un fichier de configuration appelé *makefile* qui porte souvent le nom de `Makefile`. Ce fichier décrit des cibles (qui sont souvent des fichiers, mais pas toujours), de quelles autres cibles elles dépendent, et par quelles actions (des commandes) y parvenir. Au final, le fichier `Makefile` contient l'ensemble des règles de fabrication du programme :

```

CC=gcc
RM=rm

TARGET := calculTemps

all: $(TARGET)

main.o: main.c temps.h
    $(CC) -o $@ -c $<

temps.o: temps.c temps.h
    $(CC) -o $@ -c $<

$(TARGET): main.o temps.o
    $(CC) -o $(TARGET) $^

clean:
    $(RM) -f $(TARGET) *.o *~

```

Exemple de fichier Makefile

La fabrication de l'exécutable est assurée ensuite par l'utilitaire `make` :

```
$ make
gcc -o main.o -c main.c
gcc -o temps.o -c temps.c
gcc -o calculTemps main.o temps.o
```

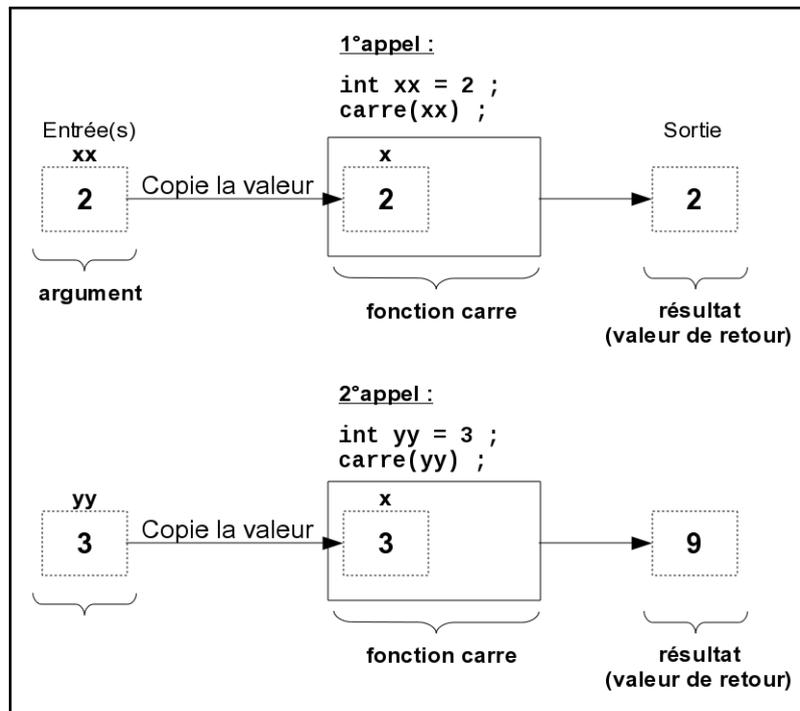
```
$ ./calculTemps
```

☞ Avec son système de dépendance, `make` ne compile que ce qui est nécessaire. Lire : <https://fr.wikipedia.org/wiki/Make>.

Passage de paramètre(s)

Passage par valeur

Lorsque l'on passe une valeur en paramètre à une fonction, cette valeur est copiée dans une variable locale de la fonction correspondant à ce paramètre. Cela s'appelle un **passage par valeur**.



Passage par valeur

⊛ Dans un passage par valeur, il est impossible pour une fonction de modifier les paramètres qu'elle reçoit. Essayons de permuter deux variables :

```
// Tente de permuter deux entiers
void permute(int a, int b)
{
    printf("avant [permute] : a=%d et b=%d\n", a, b); // affiche : a=5 et b=3
    int temp = a;
    a = b;
    b = temp;
    printf("après [permute] : a=%d et b=%d\n", a, b); // affiche : a=3 et b=5
}
```

```

int main()
{
    int a = 5, b = 3;

    printf("avant [main] : a=%d et b=%d\n", a, b); // affiche : a=5 et b=3
    permute(a, b);
    printf("après [main] : a=%d et b=%d\n", a, b); // affiche : a=5 et b=3

    return 0;
}

```

Passage par valeur

☞ Les variables *a* et *b* sont locales au bloc (`{}`) où elles sont définies. Des variables définies dans des blocs différents peuvent porter le même nom.

Passage par adresse

Pour permettre à une fonction de modifier les paramètres qu'elle reçoit, il faudra passer les adresses des variables comme paramètres. On utilise dans ce cas des pointeurs. Cela s'appelle un **passage par adresse** :

```

// Permute deux entiers
void permute(int *pa, int *pb)
{
    printf("avant [permute] : a=%d et b=%d\n", *pa, *pb); // affiche : a=5 et b=3
    int temp = *pa;
    *pa = *pb;
    *pb = temp;
    printf("après [permute] : a=%d et b=%d\n", *pa, *pb); // affiche : a=3 et b=5
}

int main()
{
    int a = 5, b = 3;

    printf("avant [main] : a=%d et b=%d\n", a, b); // affiche : a=5 et b=3
    permute(&a, &b); // passe l'adresse des variables a et b
    printf("après [main] : a=%d et b=%d\n", a, b); // affiche : a=3 et b=5

    return 0;
}

```

Passage par adresse

Le passage d'un **tableau** en paramètre d'une fonction est évidemment possible :

```

#include <stdio.h>

void raz(int t[], int n) /* équivalent à : void raz(int *t, int n) */
{
    int i;

    for(i=0;i<n;i++)

```

```

    t[i] = 0;
}

int main()
{
    int t[2] = { 2, 3 }; // 2 éléments

    printf("avant [main] : t[0] = %d et t[1] = %d\n", t[0], t[1]); // affiche t[0] = 2 et t
    [1] = 3
    raz(t, 2); // équivalent à : raz(&t[0], 2);
    printf("après [main] : t[0] = %d et t[1] = %d\n", t[0], t[1]); // affiche t[0] = 0 et t
    [1] = 0

    return 0;
}

```

Passage d'un tableau en paramètre d'une fonction

Le tableau passé en paramètre a bien été modifié car la fonction `raz()` à travailler avec l'adresse du tableau. Les cases du tableau n'ont pas été recopiées et la fonction accède au tableau original. On rappelle qu'il n'existe pas de variable désignant un tableau comme un tout. Quand on déclare `int t[2]`, `t` ne désigne pas l'ensemble du tableau mais l'adresse de la première case. `t` est une **constante de type pointeur** vers un `int` dont la valeur est `&t[0]`, l'adresse du premier élément du tableau.

☞ *C'est une très bonne chose en fait car dans le cas d'un "gros tableau", on évite ainsi de recopier toutes les cases. Le passage par adresse sera beaucoup plus efficace et rapide.*

Passage par référence

En C++, il existe aussi un **passage par référence** :

```

// Permute deux entiers
void permute(int &ra, int &rb)
{
    printf("avant [permute] : a=%d et b=%d\n", ra, rb); // affiche : a=5 et b=3
    int temp = ra;
    ra = rb;
    rb = temp;
    printf("après [permute] : a=%d et b=%d\n", ra, rb); // affiche : a=3 et b=5
}

int main()
{
    int a = 5, b = 3;

    printf("avant [main] : a=%d et b=%d\n", a, b); // affiche : a=5 et b=3
    permute(a, b);
    printf("après [main] : a=%d et b=%d\n", a, b); // affiche : a=3 et b=5

    return 0;
}

```

Passage par référence (C++)

☞ Intérêt : lorsqu'on passe des variables en paramètre de fonctions et que le coût d'une recopie par valeur est trop important, on choisira un passage par référence. Si le paramètre ne doit pas être modifié, on utilisera alors un passage par référence sur une variable constante :

```
void foo(const long double &a) // la référence est déclarée const
{
    printf("J'ai un accès en lecture : a = %.1Lf\n", a);
    printf("mais pas en écriture !\n");
    //a = 3.5; // interdit car déclarée const ! erreur: assignment of read-only reference 'a'
}

int main()
{
    long double a = 2.0;

    printf("Je suis une grosse variable de taille %d octets\n", sizeof(a)); // affiche Je
        suis une grosse variable de taille 12 octets
    printf("Je suis a et j'ai pour valeur : a = %.1Lf\n", a); // affiche Je suis a et j'ai
        pour valeur : a = 2.0

    foo(a);

    return 0;
}
```

Passage par référence constante (C++)

Valeur de retour

Par définition, une fonction fournit un **résultat**. Pour cela, on lui déclare un **type de retour** et on renvoie une valeur (du type déclaré) avec l'instruction **return**. Cette instruction provoque évidemment la sortie de la fonction appelée. La valeur de retour peut servir à **renvoyer un résultat** ou **un état sur l'exécution** de la fonction appelée.

Certains programmeurs utilisent donc la valeur de retour pour indiquer si la fonction a réalisé son traitement avec succès ou si elle a rencontré une erreur. C'est le cas de beaucoup de fonctions systèmes. On distingue différentes pratiques pour la convention du type de retour :

- valeur de retour de type booléen (**bool**) : **true** (succès) ou **false** (erreur)
- valeur de retour de type entière (**int**) : 0 (succès) ou !=0 (un code d'erreur)
- valeur de retour de type entière (**int**) : 1 (succès) ou <0 (un code d'erreur)

☹ Il ne faut jamais négliger ce type de code de retour lorsqu'on utilise une fonction. Prenons un exemple : vous appelez une fonction pour créer un répertoire (**man 2 mkdir**) puis une fonction pour copier un fichier dans ce répertoire. Si la création du répertoire échoue, cela ne sert à rien d'essayer de copier ensuite le fichier. La réussite (ou l'échec) de la première action conditionne l'exécution de la deuxième action. Si vous ne testez jamais les codes de retour de fonction, vous allez provoquer des cascades d'erreurs.

Dans certaines situations extrêmes, vous risquer d'avoir besoin de sortir immédiatement du programme quelque soit l'endroit où vous êtes. Vous pouvez utiliser la fonction **exit()** qui termine normalement un programme en indiquant une valeur de retour. Évidemment, la fonction **exit()** ne revient jamais.

```
#include <stdio.h>
#include <stdlib.h> /* pour exit() */
#include <sys/stat.h> /* pour mkdir() */
```

```
#include <sys/types.h>

int main()
{
    mode_t mode = 0750;
    int retour;

    // Création d'un répertoire
    retour = mkdir("./dossier", mode);
    // Une erreur s'est-elle produite ?
    if(retour == -1)
    {
        // La fonction perror() affiche un message d'erreur décrivant la dernière erreur
        // rencontrée durant un appel système ou une fonction de bibliothèque
        perror("mkdir");
        // Cela ne sert à rien de continuer, on quitte le programme en indiquant qu'on a
        // rencontré une erreur
        exit(EXIT_FAILURE);
    }

    // ...

    return EXIT_SUCCESS; // sortir de la fonction main() revient à quitter le programme
}
```

Quitter un programme

Cela donne :

```
$ gcc mkrep.c
```

```
$ ./a.out
```

```
$ ls -l
```

```
-rwxrwxr-x 1 tv tv 8,3K août 9 10:47 a.out*
drwxr-x--- 2 tv tv 4,0K août 9 10:47 dossier/
-rw-rw-r-- 1 tv tv 329 août 9 10:47 mkrep.c
```

```
$ ./a.out
```

```
mkdir: File exists
```

Nommer une fonction

Un nom de fonction est construit à l'aide d'un **verbe** (surtout pas un nom), et éventuellement d'éléments supplémentaires comme :

- une quantité
- un complément d'objet
- un adjectif représentatif d'un état

☞ *Une fonction est composée d'une série d'instructions qui effectue un traitement. Il faut donc utiliser un verbe pour caractériser l'action réalisée par la fonction.*

On utilisera la convention suivante : **un nom de fonction commence par une lettre minuscule puis les différents mots sont repérés en mettant en majuscule la première lettre d'un nouveau mot.**

Le verbe peut être au présent de l'indicatif ou à l'infinitif. L'adjectif représentatif d'un état concerne surtout les fonctions booléennes. La quantité peut, le cas échéant, enrichir le sens du complément.

Exemples : `void ajouter()`, `void sauverValeur()`, `estPresent()`, `estVide()`, `videAMoitieLeReservoir()`, ...

⊗ *Les mots clés du langage sont interdits comme noms.*

Les noms des paramètres d'une fonction sont construits comme les noms de variables : ils commencent, notamment par une minuscule. L'ordre de définition des paramètres doit respecter la règle suivante :

`nomFonction(parametrePrincipal, listeParametres)`

où `parametrePrincipal` est la donnée principale sur laquelle porte la fonction, la `listeParametres` ne comportant que des données secondaires, nécessaires à la réalisation du traitement réalisé par la fonction.

Exemple : `ajouter(float mesures[], float uneMesure)`

Le rôle de cette fonction est d'ajouter une `mesure` à un ensemble de `mesures` (qui est la donnée principale) et non, d'ajouter un ensemble de `mesures` à une `mesure`.

⚡ *Bonne pratique : L'objectif de la programmation procédurale est de décomposer un traitement de grande taille en plusieurs parties plus petites jusqu'à obtenir quelque chose de suffisamment simple à comprendre et à résoudre (cf. Descartes et son discours de la méthode). Pour cela, on va **s'obliger à limiter la taille de nos fonctions à une valeur comprise entre 10 à 15 lignes maximum** (accolades exclues).*

La fonction main

Tout programme C/C++ doit posséder une **fonction** nommée `main` (dite fonction principale) pour indiquer où commencer l'exécution. La fonction `main()` représente le point d'entrée (et de sortie) d'un programme exécuté par le "système" (c'est-à-dire le système d'exploitation).

```
// Forme simplifiée :
int main()
{

    return 0; // on doit retourner une valeur entière
}

// Forme normalisée :
int main(int argc, char **argv)
{

    return 0; // on doit retourner une valeur entière
}

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

    return 0; // on doit retourner une valeur entière
}
```

La fonction main()

⇒ **Paramètres d'entrée** : la fonction `main()` reçoit deux paramètres (ils peuvent être ignorés) : un entier (`int`) qui précisera le nombre d'arguments passés au programme et un tableau de chaînes de caractères (`char **` ou `char *[]`) qui contiendra la liste de ses arguments.

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int i;

    printf("nb d'arguments = %d\n", argc);

    for(i=0;i<argc;i++)
        printf("argv[%d] = %s\n", i, argv[i]);

    //...

    return 0;
}
```

Les arguments d'un programme

Ce qui donne :

```
$ ./a.out les parametres du programme
nb d'arguments = 5
argv[0] = ./a.out
argv[1] = les
argv[2] = parametres
argv[3] = du
argv[4] = programme
```

```
$ ./a.out
nb d'arguments = 1
argv[0] = ./a.out
```

☞ Il existe une fonction `getopt()` qui permet d'analyser plus facilement les options d'une ligne de commande (*man 3 getopt*).

➡ **Valeur de retour** : la fonction `main()` doit retourner une valeur entière (`int`). Sur certains systèmes (Unix/Linux), elle peut servir à vérifier si le programme s'est exécuté correctement. Un zéro (0) indique alors que le programme s'est terminé avec succès (c'est une convention). Évidemment, une valeur différente de 0 indiquera que le programme a rencontré une erreur. Et sa valeur précisera alors le type de l'erreur. Sur un système Unix/Linux, la dernière valeur de retour reçue est stockée dans une variable `$?` du *shell*.

```
#include <stdio.h>

int main(int argc, char **argv)
{
    // Exemple : on peut vérifier que l'on reçu le nombre suffisant d'arguments
    if (argc < 2)
        return 1;

    //...

    return 0;
}
```

La valeur de retour d'un programme

Ce qui donne :

```
$ ./a.out les parametres du programme

// Affichage de la valeur de retour
$ echo $?
0
// 0=succès

$ ./a.out

// Affichage de la valeur de retour
$ echo $?
1
// 1=erreur
```

Intérêt des fonctions par l'exemple

Pour présenter l'intérêt d'utiliser des fonctions, voici un exemple de programme en pseudo-code :

```
Variable prix : RÉEL
Variable quantité : ENTIER
Variable total_ttc : RÉEL = 0.00
Constante tva : RÉEL = 19.60

Début
  -- début des achats
  Ecrire "*****"

  -- achat de 3 articles à 10,50 euros HT chacun
  prix <- 10.50
  quantité <- 3
  total_ttc <- total_ttc + tva * ( prix * quantité )
  Ecrire "Sous-total TTC : " & total_ttc

  -- achat de 2 articles à 21,30 euros HT chacun
  prix <- 21.30
  quantité <- 2
  total_ttc <- total_ttc + tva * ( prix * quantité )
  Ecrire "Sous-total TTC : " & total_ttc

  -- achat d'1 article à 2,40 euros HT
  prix <- 2.40
  quantité <- 1
  total_ttc <- total_ttc + tva * ( prix * quantité )
  Ecrire "Sous-total TTC : " & total_ttc

  -- fin des achats
  Ecrire "*****"

  -- total des achats TTC
  Ecrire "#####"
  Ecrire "Total TTC : " & total_ttc
  Ecrire "#####"

Fin
```

➔ Pour les 3 achats de l'exemple, le calcul est répété à chaque fois. Ce qui n'est pas pratique pour la maintenance du code : si la formule n'est pas la bonne (une erreur est toujours possible) ou si la formule doit être modifiée (autre mode de calcul), il faudra changer le code en plusieurs endroits pour un même type de calcul.

On va donc créer une fonction qui calcule le prix TTC pour l'achat d'une certaine quantité d'articles, et le retourne au programme appelant :

```
const double tva = 19.6;

double calculePrixTTC(double prix, int quantite)
{
    return tva * (prix * quantite);
}
```

Celui-ci utilise le prix TTC retourné pour l'ajouter au total :

```
int main()
{
    double total_ttc = 0.;

    // ...
    total_ttc += calculePrixTTC(10.50, 3); // achat de 3 articles à 10,50 euros HT chacun
    // ...

    // ...
    total_ttc += calculePrixTTC(21.30, 2); // achat de 2 articles à 21,30 euros HT chacun
    // ...

    // ...
    total_ttc += calculePrixTTC(2.40, 1); // achat d'1 article à 2,40 euros HT
    // ...
}
```

➔ L'affichage des '*' et des dièses '#' pourrait lui aussi être factorisé en fonction. De manière naïve, on pourrait faire :

```
void afficher40Etoiles()
{
    for (i = 0; i < 40; i++)
    {
        printf("*");
    }
    printf("\n");
}

void afficher30Dieses()
{
    for (i = 0; i < 30; i++)
    {
        printf("#");
    }
    printf("\n");
}
```

On pourrait surtout améliorer le pseudo-code comme ceci :

```

...
Début
    ...
    Afficher 40 "*"
    Ecrire "Sous-total TTC : " & total_ttc
    ...
    Afficher 40 "*"
    ...
    Afficher 30 "#"
    Ecrire "Total TTC : " & total_ttc
    Afficher 30 "#"
Fin

```

On va donc créer une fonction paramétrable avec 2 arguments : le `caractere` à afficher et `nb` fois qu'il faut l'afficher :

```

void afficherCaracteres(char caractere, int nb)
{
    for (i = 0; i < nb; i++)
    {
        printf("%c", caractere);
    }
    printf("\n");
}

// Pour afficher les 40 étoiles :
afficherCaracteres('*', 40);

// Pour afficher les 30 dièses :
afficherCaracteres('#', 30);

```

Au final, on obtient le code source suivant :

```

#include <stdio.h>

const double tva = 19.6;

double calculePrixTTC(double prix, int quantite)
{
    return (1. + tva/100.) * (prix * (double)quantite);
}

void afficherCaracteres(char caractere, int nb)
{
    int i;

    for (i = 0; i < nb; i++)
    {
        printf("%c", caractere);
    }
    printf("\n");
}

void afficherTotalTTC(double total_ttc)
{

```

```

    afficherCaracteres('#', 30);
    printf("Total TTC : %.3f euros\n", total_ttc);
    afficherCaracteres('#', 30);
}

int main()
{
    double total_ttc = 0.;

    // début des achats
    afficherCaracteres('* ', 40);

    // achat de 3 articles à 10,50 euros HT chacun
    total_ttc += calculePrixTTC(10.50, 3);
    printf("Sous-total TTC : %.2f euros\n", total_ttc);

    // achat de 2 articles à 21,30 euros HT chacun
    total_ttc += calculePrixTTC(21.30, 2);
    printf("Sous-total TTC : %.2f euros\n", total_ttc);

    // achat d'1 article à 2,40 euros HT
    total_ttc += calculePrixTTC(2.40, 1);
    printf("Sous-total TTC : %.2f euros\n", total_ttc);

    // fin des achats
    afficherCaracteres('* ', 40);

    // total des achats TTC
    afficherTotalTTC(total_ttc);
}

```

⇒ Le programme est maintenant plus simple à lire et à maintenir !

Surcharge de fonctions

Il est interdit en C de définir plusieurs fonctions qui portent le même nom. Mais c'est autorisé en C++ si le compilateur peut différencier deux fonctions en analysant les paramètres qu'elle reçoit.

La **surcharge** (*overloading*) est une technique permettant de définir plusieurs fonctions qui portent le même nom si celles-ci ont des **signatures différentes**. La signature d'une fonction correspond au prototype **sans le type de retour**. Une surcharge de fonction est donc possible si les paramètres (leurs nombre et/ou leur type) sont différents :

```

// Retourne le minimum entre 2 entiers
int min(int val1, int val2)
{
    if(val1 <= val2)
        return val1;
    return val2;
}

// Surcharge : les types des paramètres sont différents
// Retourne le minimum entre 2 flottants
float min(float val1, float val2)

```

```

{
    if(val1 <= val2)
        return val1;
    return val2;
}

// Surcharge : le nombre de paramètres est différent
// Retourne le minimum entre 3 entiers
int min(int val1, int val2, int val3)
{
    int val = min(val1, val2);
    val = min(val, val3);
    return val;
}

```

Surcharge de fonctions (C++)

⊗ La surcharge n'est pas possible en langage C. Elle est utilisable seulement en C++.

Paramètre par défaut

Le C++, mais pas le C, offre la possibilité d'affecter des **valeurs par défaut** aux paramètres d'une fonction. La syntaxe de la liste de paramètres sera alors la suivante :

```
type variable [= valeur] [, type variable [= valeur] [...]]
```

où **type** est le type du paramètre variable qui le suit et **valeur** sa valeur par défaut. La valeur par défaut d'un paramètre est la valeur que ce paramètre prend si aucune valeur ne lui est attribuée lors de l'appel de la fonction.

```

// Retourne la somme entre 2 entiers
int somme(int val1=0, int val2=0)
{
    return val1 + val2;
}

int main()
{
    int a = 5, b = 10;

    printf("%d\n", somme(a, b)); // affiche 15 car 10 + 5
    printf("%d\n", somme(b, a)); // affiche 15 car 5 + 10

    printf("%d\n", somme(a)); // affiche 5 car 5 + 0
    printf("%d\n", somme(b)); // affiche 10 car 10 + 0

    printf("%d\n", somme()); // affiche 0 car 0 + 0

    return 0;
}

```

Paramètre par défaut (C++)

⊗ Les paramètres par défaut doivent être déclarés successivement de la droite vers la gauche. La fonction « `int somme(int val1=0, int val2) {...}` » est invalide, car si on ne passe pas deux paramètres, `val2` ne sera pas initialisé. Les paramètres par défaut n'existent pas en langage C, seulement en C++.

Fonctions inline

Le C++ dispose du mot clé `inline`, qui permet de modifier la méthode d'implémentation des fonctions. Placé devant la déclaration d'une fonction, il propose au compilateur de ne pas instancier cette fonction (notion de **macro**). Cela signifie que l'on désire que le compilateur remplace l'appel de la fonction par le code correspondant.

☞ Si la fonction est “grosse” ou si elle est appelée souvent, le programme devient **plus volumineux**, puisque la fonction est réécrite à chaque fois qu'elle est appelée. En revanche, il devient nettement **plus rapide**, puisque les mécanismes d'appel de fonctions, de passage des paramètres et de la valeur de retour sont ainsi évités.

🔗 *En pratique, on réservera cette technique pour les “petites” fonctions appelées dans du code devant être rapide. Cela peut s'apparenter à de l'optimisation.*

Fonctions statiques

Par défaut, lorsqu'une fonction est définie dans un fichier C/C++, elle peut être utilisée dans tout autre fichier pourvu qu'elle soit déclarée avant son utilisation. Dans ce cas, la fonction est dite **externe** (mot-clé `extern`).

Il peut cependant être intéressant de définir des fonctions locales à un fichier, soit afin de résoudre des conflits de noms (entre deux fonctions de même nom et de même signature mais dans deux fichiers différents), soit parce que la fonction est uniquement d'intérêt local. Le C/C++ fournit donc le mot clé `static` qui, une fois placé devant la définition et les éventuelles déclarations d'une fonction, la rend unique et utilisable uniquement dans ce fichier. À part ce détail, les fonctions statiques s'utilisent exactement comme des fonctions classiques.

➡ Cas des variables :

Le mot clé `static`, placé devant le nom d'une **variable globale** (une variable déclarée en dehors de tout bloc `{}`), a le même effet que pour les fonctions statiques. On parle alors de variable globale cachée.

🕒 Le mot clé `static`, placé devant le nom d'une **variable locale** (une variable déclarée dans un bloc `{}`), rend la variable persistante (la variable conserve sa valeur) entre chaque entrée dans le bloc.

```
#include <stdio.h>

void compte()
{
    static int compteur = 0; // la variable compteur conservera sa valeur entre chaque appel

    ++compteur;
    printf("La fonction a été appelée : %d fois\n", compteur);
}

int main()
{
    compte();
    compte();
    compte();

    return 0;
}
```

Variable locale statique

On obtient :

```
$ ./a.out
La fonction a été appelée : 1 fois
La fonction a été appelée : 2 fois
La fonction a été appelée : 3 fois
```

Nombre variable de paramètres

En général, les fonctions ont un nombre constant de paramètres. Pour les fonctions qui ont des paramètres par défaut en C++, le nombre de paramètres peut apparaître variable à l'appel de la fonction, mais en réalité, la fonction utilise toujours le même nombre de paramètres.

Le C et le C++ disposent toutefois d'un mécanisme qui permet au programmeur de réaliser des fonctions dont le nombre et le type des paramètres sont variables. C'est le cas par exemple des fonctions `printf()` et `scanf()`.

Pour indiquer au compilateur qu'une fonction peut accepter une liste de paramètres variable, il faut simplement utiliser des points de suspensions dans la liste des paramètres dans les déclarations et la définition de la fonction :

```
type identificateur(paramètres, ...)
```

Dans tous les cas, il est nécessaire que la fonction ait au moins un paramètre classique. Les paramètres classiques doivent impérativement être avant les points de suspensions.

On utilisera le type `va_list` et les expressions `va_start`, `va_arg` et `va_end` pour récupérer les arguments de la liste de paramètres variable, un à un. Pour cela, il faudra inclure le fichier d'en-tête `stdarg.h`.

```
#include <stdarg.h>

/* effectue la somme de compte flottants (float ou double) et la renvoie dans un double */
double somme(int compte, ...)
{
    double resultat = 0; /* Variable stockant la somme */
    va_list varg; /* Variable identifiant le prochain paramètre */

    va_start(varg, compte); /* Initialisation de la liste */

    /* Parcours de la liste */
    while (compte != 0)
    {
        resultat = resultat + va_arg(varg, double); /* récupère le prochain paramètre dans
            la liste */
        compte = compte-1;
    }

    va_end(varg);

    return resultat;
}

int main()
{
    double total = 0;
```

```
total = somme(3, 1.5, 2.5, 5.); // 4 paramètres
printf("%.1f\n", total); // affiche 9.0

total = somme(2, 1.5, 2.5); // et maintenant 3 paramètres
printf("%.1f\n", total); // affiche 4.0

return 0;
}
```

Nombre variable de paramètres

Pointeur vers une fonction

Le **nom** d'une fonction est **une constante de type pointeur** :

```
// une fonction
int f(int x, int y)
{
    return x+y;
}

// un pointeur sur une fonction :
int (*pf)(int, int); // pf est un pointeur vers une fonction admettant 2 entiers en
    paramètres et retournant un entier

pf = f;           // pf pointe vers la fonction f

// appel :
printf("%d\n", (*pf)(3, 5)); // affiche 8
```

Pointeur sur une fonction

Exercices : Les joies du code

C'est le soir et la bibliothèque se retrouve fermée. Alors que vous prenez votre repas, vous êtes abordé par des adolescents qui sont intéressés par votre robot. Amateurs de jeux, de codes et d'énigmes, ils souhaiteraient vous voir réaliser quelques programmes. Vous profitez de l'occasion pour avancer dans votre manuel.

☞ Objectif : vous allez découvrir comment la notion de fonction permet de découper le code en différentes parties et de structurer un long programme de manière très efficace et agréable :

- [Motif rectangulaire](#) : page 23
- [Banquet municipal](#) : page 24
- [Journée des cadeaux](#) : page 28
- [Jour de marché](#) : page 29
- [Recensement](#) : page 29

Motif rectangulaire

À présent, vos spectateurs ont envie que votre robot imprime un motif sur une feuille rectangulaire (dont la taille est de 80 colonnes et 43 lignes maximum), car cela leur sert pour des jeux de géométrie.

☞ Ce que doit faire votre programme :

Votre programme doit lire le nombre de lignes et de colonnes de la feuille, puis le motif à afficher sous la forme d'un caractère. Il doit alors afficher le motif de sorte qu'il remplisse chaque cellule de la feuille. Vous allez créer et utiliser une fonction `afficheMotif` pour cela.

☞ Exemples :

```
$ ./motif-rectangulaire
4
9
F
```

```
FFFFFFFFF
FFFFFFFFF
FFFFFFFFF
FFFFFFFFF
```

```
$ ./motif-rectangulaire
8
3
P
```

```
PPP
PPP
PPP
PPP
PPP
PPP
PPP
PPP
PPP
```

Question 1. Écrire le programme `motif-rectangulaire.c`. Tester et valider.

Banquet municipal

Vous commencez à être connu pour vos talents de programmeur, aussi lors de votre passage dans la ville d'Incerto, le maire décide de vous inviter au grand banquet qu'il organise. Le maire se charge lui-même de faire le plan de table mais il change toujours d'avis et les serveurs doivent constamment changer de place les petites affiches sur lesquelles sont indiqués les noms des personnes.

Afin de l'aider, vous lui proposez d'utiliser votre robot pour déterminer la position de chaque personne après tous les changements décidés par le maire. Afin de simplifier le problème, on suppose qu'il n'y a qu'une seule très grande table circulaire.

☞ Ce que doit faire votre programme :

Votre programme lit tout d'abord le nombre total de positions sur la table (au maximum 1000). Il lit ensuite, pour chaque place, le nom de la personne et la position (numérotée à partir de 1) à laquelle elle doit s'installer. Si la position indiquée est déjà occupée, il faudra lui faire une place en décalant les personnes d'un rang. Une fois toutes les personnes placées, on affiche pour chaque position le nom de la personne qui s'y trouve.

Orthonart, l'un des élèves de la classe, vous a demandé de lui apprendre à utiliser votre petit robot. Après lui avoir expliqué les bases de la programmation, vous lui prêtez le robot afin qu'il améliore le

programme que vous avez déjà écrit en le décomposant en fonctions. On limitera la taille de des fonctions à 10 lignes maximum (accolades exclues).

```
// Le programme lit tout d'abord le nombre total de positions sur la table (au maximum 1000)
. Il lit ensuite, pour chaque place, le nom de la personne et la position (numérotés à
partir de 1) à laquelle elle doit s'installer. Si la position indiquée est déjà occupée
par une personne, il faudra décaler les personnes d'une place. On affiche, pour chaque
position, le nom de la personne qui s'y trouve une fois tous les changements faits.

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

#define MAX_POSITIONS 1000
#define MAX_NOM      32

int main()
{
    int nbPositions;
    char nom[MAX_NOM+1];
    int numero;
    int position, positionLibre, positionDepart, positionDestination;
    int i, n, c;

    // saisir le nombre de positions sur la table
    do
    {
        scanf("%d", &nbPositions);
    }
    while(nbPositions > MAX_POSITIONS);

    // vider le tampon
    while ((c = getchar()) != '\n' && c != EOF);

    char *planTable[nbPositions];

    // initialiser le plan de table
    for(i=0;i<nbPositions;i++)
    {
        planTable[i] = (char *)malloc((MAX_NOM+1) * sizeof(char));
        strcpy(planTable[i], ""); // place libre
    }

    // remplir le plan de table
    for(i=0;i<nbPositions;i++)
    {
        // saisir le placement d'une personne
        scanf("%32s%d", &nom[0], &numero);

        // vider le tampon
        while ((c = getchar()) != '\n' && c != EOF);

        // la place est libre ?
        if(strlen(planTable[numero-1]) == 0)
```

```
{
    // positionner la personne
    strcpy(planTable[numero-1], nom);
}
else
{
    // rechercher la première place de libre
    positionLibre = -1;
    position = numero % nbPositions;
    do
    {
        if(strlen(planTable[position]) == 0)
        {
            positionLibre = position;
            break;
        }
        position = (position+1) % nbPositions;
    }
    while(positionLibre == -1);

    // calculer le nombre de décalage à effectuer
    int decalage = ((positionLibre+1-numero) % nbPositions);
    if(decalage < 0)
        decalage = nbPositions + decalage;

    // décaler la ou les personne(s)
    positionDestination = positionLibre;
    for(n=0;n<decalage;n++)
    {
        positionDepart = positionDestination - 1;
        if(positionDepart < 0) positionDepart = nbPositions - 1;
        strcpy(planTable[positionDestination], planTable[positionDepart]);
        positionDestination = (positionDestination-1) % nbPositions;
        if(positionDestination < 0) positionDestination = nbPositions - 1;
    }
    // positionner la personne
    strcpy(planTable[numero-1], nom);
}

// afficher le plan de table
for(i=0;i<nbPositions;i++)
{
    printf("%d %s\n", i+1, planTable[i]);
}

// libérer le plan de table
for(i=0;i<nbPositions;i++)
{
    free(planTable[i]);
}

return 0;
```

}

banquet-municipal.c

☞ Exemples :

\$./banquet-municipal

4
robert 1
roger 2
raymond 3
alphonse 41 robert
2 roger
3 raymond
4 alphonse

\$./banquet-municipal

4
robert 2
roger 2
raymond 2
alphonse 21 robert
2 alphonse
3 raymond
4 roger

\$./banquet-municipal

4
robert 2
roger 1
raymond 2
alphonse 11 alphonse
2 roger
3 raymond
4 robert

\$./banquet-municipal

4
robert 4
roger 4
raymond 1
alphonse 11 alphonse
2 raymond
3 robert
4 roger**Question 2.** Modifier le programme `banquet-municipal.c`. Tester et valider.

Question 3. Est-ce que la majorité des commentaires sont-ils toujours utiles ?

Journée des cadeaux

Dans la ville de Largition, la tradition veut que, une fois par mois, les habitants les plus riches fassent un cadeau aux habitants les moins riches. Cela permet de maintenir une bonne ambiance malgré les inévitables différences de richesses.

Un habitant est considéré comme riche si sa fortune est plus grande que celle de la moitié de la population. Calculer, chaque mois, qui est riche ou pas est un long travail, aussi lorsque le maire a entendu parlé de vos talents, il vous a demandé votre aide.

☞ Ce que doit faire votre programme :

Il devra lire un premier entier, le nombre d'habitants (au plus 1000) puis, pour chaque habitant il devra lire sa fortune, un entier. Il devra calculer puis afficher une valeur (la **médiane**) permettant de dire facilement si une personne est riche ou pas, simplement en regardant si la fortune de cette personne est plus grande ou plus petite que cette valeur.

Deux cas peuvent se présenter :

- Si le nombre d'habitants est impair, par exemple si leurs fortunes sont 10, 5, 12, 8, 3 alors la valeur recherchée est 8. En effet, il y aura alors 2 personnes "riches" (10 et 12), 2 "moins riches" (3 et 5) et 1 juste au milieu (8) qui ne donnera ni recevra de cadeau.
- Si le nombre d'habitants est pair, par exemple si leurs fortunes sont 10, 5, 12, 8, 3, 9 alors la valeur recherchée est entre 8 et 9. Il y a en effet 3 personnes "riches" (9, 10 et 12) et 3 "moins riches" (3, 5 et 8). Par convention on prendra la valeur 8.5, c'est-à-dire la moyenne de 8 et 9.

📖 La bibliothèque C standard offre une fonction de tri : `qsort()`. Elle sert à trier un tableau d'éléments à l'aide d'une fonction de comparaison à fournir. Voir le cours sur les tableaux.

☞ Exemples :

```
$ ./journee-cadeaux
```

```
5
10
5
12
8
3
```

```
8
```

```
$ ./journee-cadeaux
```

```
6
10
5
12
8
3
9
```

```
8.5
```

Question 4. Écrire le programme `journee-cadeaux.c`.

Jour de marché

Lorsqu'on organise un marché, certains emplacements sont beaucoup plus intéressants que d'autres. Afin d'éviter les tentations de « pots de vin », la ville dans laquelle vous venez d'arriver a décidé qu'on organiserait à chaque fois un tirage au sort, chacun ayant sa chance !

Chaque marchand met donc son nom dans un grand panier, et des tirages sont effectués. On tire d'abord le nom du marchand qui aura le premier emplacement, puis celui qui aura le second, et ainsi de suite. On décide alors d'afficher par ordre alphabétique, les noms des marchands avec le numéro de leur emplacement.

☞ Ce que doit faire votre programme :

Votre programme devra lire le nombre d'emplacements `nbEmplacements` (au maximum 1000), puis pour chaque emplacement à partir de 0, le nom du marchand à qui est attribué l'emplacement. Vous allouerez dynamiquement la mémoire pour stocker le nom du marchand.

Ensuite, votre programme devra afficher la liste par ordre alphabétique (cf. `qsort()`) des emplacements attribués : le nom du marchand et le numéro de l'emplacement qui lui est attribué numéroté à partir de 1.

☞ Exemples :

```
$ ./jour-marche
```

```
4
```

```
robert
```

```
roger
```

```
raymond
```

```
alphonse
```

```
alphonse 4
```

```
raymond 3
```

```
robert 1
```

```
roger 2
```

Question 5. Écrire le programme `jour-marche.c`.

Recensement

Lors de votre passage dans la ville d'Incerto, le maire décide de faire un recensement de la population. Pour cela, il faut se déplacer chez chaque habitant et lui demander son nom et son âge. Au dernier recensement, il y avait 1000 habitants mais toutes les données ont été perdues et la population a augmenté depuis. Afin de l'aider, vous lui proposez d'utiliser votre robot pour effectuer la collecte des données.

☞ Ce que doit faire votre programme :

Votre programme devra lire une suite d'entiers positifs (l'âge d'un habitant) et afficher les statistiques suivantes : le nombre habitants, la moyenne d'âge et la liste des âges par ordre décroissant (du plus âgé au moins âgé). On ne sait pas combien il y aura d'habitants, et le recensement se terminera par la saisie de la valeur -1 (qui n'est pas un âge, juste un marqueur de fin).

Afin de simplifier le problème, on suppose qu'il y avait seulement 10 habitants au dernier recensement. Vous allouerez dynamiquement la mémoire pour stocker les données de la collecte par tranche de 10 habitants.

☞ Exemples :

\$./recensement

10

51

5

66

23

27

78

1

12

38

40

-1

11

31.9

78

66

51

40

38

27

23

12

10

5

1

Question 6. Écrire le programme `recensement.c`.

Bilan

Conclusion : il est indispensable de décomposer un traitement de grande taille en plusieurs parties plus petites jusqu'à obtenir quelque chose de suffisamment simple à comprendre et à résoudre. Cette approche consistant à décomposer une tâche complexe en une suite d'étapes plus petites (et donc plus facile à gérer) ne s'applique pas uniquement à la programmation et aux ordinateurs. Elle est courante et utile dans la plupart des domaines de l'existence.

Descartes (mathématicien, physicien et philosophe français) dans le *Discours de la méthode* :

« diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourrait, et qu'il serait requis pour les mieux résoudre. »

D'un point de vue théorique, vous pouvez désormais faire tout ce qu'on peut faire avec un ordinateur, le reste n'est que détails! Cela montre tout de même la valeur des "détails" et l'importance des compétences pratiques parce qu'il est clair que vous débutez à peine votre carrière de programmeur.