Langage de programmation

**HAUTE ÉCOLE DE NAMUR-LIÈGE-LUXEMBOURG**

**Bloc 1**

Atelier 5 – Fonctions

Objectifs

* Définir et appeler une fonction
* Distinguer paramètres d’entrée et paramètres de sortie
* Définir la portée des identifiants
* Distinguer fonction et procédure

[Introduction 1](#_Toc138425201)

[A. Pourquoi des fonctions ? 2](#_Toc138425202)

[B. Qu’est-ce qu’une fonction 6](#_Toc138425203)

[C. Première fonction… ou plutôt première procédure 9](#_Toc138425204)

[D. Avec des paramètres 12](#_Toc138425205)

[E. Premières fonctions ! 15](#_Toc138425206)

[F. Fonctions sans paramètre d’entrée 18](#_Toc138425207)

[G. Tableaux et fonctions 19](#_Toc138425208)

[H. Portée des identifiants (variables et fonctions) 25](#_Toc138425209)

# Introduction

Dans cet atelier, vous allez apprendre à découper votre code en fonctions. En effet, dans les deux modules précédents, vous avez modifié le **flux des instructions** grâces aux alternatives et aux répétitives. Une autre façon de modifier ce flux est de définir une fonction et d’y faire appel. Mais pourquoi définir une fonction ?

Dans ce document, plusieurs conventions sont utilisées :

* les mots gras désignent des termes de vocabulaire liés à l’**informatique en général**.
* les mots soulignés et gras désignent des termes de vocabulaire directement liés aux cours de **programmation**.
* le logo signifie que vous avez quelque chose à réaliser.
* le logo est associé aux cadres présentant certaines conventions.
* le logo est associé aux cadres présentant les éléments liés à la propreté/lisibilité du code (*clean code*).

# Pourquoi des fonctions ?

Un problème complexe nécessite souvent un algorithme qui l’est tout autant. C’est pourquoi une stratégie de **modularité** est souvent mise en place lors de l’écriture d’un programme : les instructions sont rassemblées en différentes parties distinctes et indépendantes. Cette stratégie fournit, entre autres, les avantages suivants : éviter les répétitions, favoriser la réutilisation, simplifier le code... Ceux-ci sont développés par la suite.

Il existe de nombreux termes et définitions permettant de décrire les entités à la base de cette modularité : fonctions, procédures, routines... Ils traduisent tous le concept de **sous-programmes**, des séries d’instructions qui peuvent être exécutées à partir d’un programme parent. Dans le cours de « Principes de programmation », ce sont les **modules** qui permettent de répondre à ce besoin.

Jusqu’à maintenant, vous avez eu l’occasion d’utiliser diverses fonctions telles que printf(), scanf\_s(), getchar()… Ces fonctions sont des fonctions natives, c’est-à-dire qu’elles font partie de la bibliothèque standard du C. D’autres fonctions sont également disponibles en important des bibliothèques déjà écrites par d’autres programmeurs.

À quoi ça sert de découper un programme en petits morceaux de code alors qu’on pourrait simplement tout écrire « en une fois » ?

La première façon de présenter ce concept est complètement indépendant de la programmation. La seconde, en revanche, y est directement liée.

## Dans la vie réelle

Dans la gestion de l’organisation de votre journée, vous avez la possibilité de faire diverses activités :

* vous préparer le matin,
* aller au cours,
* revoir vos cours,
* aller à la salle de sport,
* …

Chacune de ces activités est elle-même constituée de tâches plus précises. Par exemple, pour l’activité vous préparer le matin, vous devez réaliser les tâches suivantes :

* vous lever,
* aller aux toilettes,
* préparer vos vêtements,
* prendre votre douche,
* prendre votre petit déjeuner,
* vous laver les dents.

Chaque tâche peut encore une fois être divisée en actions. Par exemple, lorsque vous prenez votre douche, vous devez

* entrer dans la douche,
* ouvrir le robinet,
* vous laver le corps,
* vous laver les cheveux,
* vous rincer,
* sortir de la douche,
* vous essuyer.

Dans le cas où vous allez à la salle de sport, la dernière tâche que vous allez réaliser est probablement de prendre votre douche. Il s’agit de la même tâche que celle réalisée lorsque vous vous préparez le matin.

Au lieu de faire la liste des actions associées au fait de prendre votre douche, vous dites juste « je vais prendre ma douche ». Il semble donc logique de regrouper les actions liées à une tâche sous un même nom afin d’y faire référence au moyen de ce nom, plutôt que de lister toutes les actions à chaque fois.

De plus, pensez au cas où des actions sont ajoutées à la tâche prendre sa douche, comme par exemple prendre le savon et/ou prendre le shampoing. Il faut ajouter ces actions à chaque fois qu’on rencontre la tâche prendre sa douche… Quid si on ne les modifie pas toutes, par distraction ?

## En programmation

Comme dans l’exemple ci-dessus, un programme, surtout s'il est long, réalise souvent des traitements similaires, voire identiques, à plusieurs moments de son exécution. Par exemple, la validation d’une adresse mail, le calcul d’une moyenne, l’affichage d’un titre… pourraient être répétés à des moments différents d’un même programme.

Dans le cas d’un programme permettant de soumettre un questionnaire à l’utilisateur, on pose des questions en espérant une réponse au format 'o' pour « oui » et 'n' pour « non ». Le programme doit donc à chaque fois valider la réponse, c’est-à-dire s’assurer que la réponse est soit 'o', soit 'n'. Le bout de code qui correspond à ce traitement est le suivant.

char reponse;

do {

printf("Réponse (o/n) : ");

scanf\_s("%c", &reponse, 1);

getchar();

} while(reponse != 'o' && reponse != 'n');

Comme il s’agit d’un questionnaire, cette démarche doit être répétée autant de fois qu’il y a de questions. La manière la plus « évidente », mais aussi la moins habile, de le faire, est bien entendu de répéter le code autant de fois que nécessaire, c'est-à-dire en recopiant presque mot à mot les lignes de codes ou en faisant CTRL+C et CTRL+V autant de fois que nécessaire. Par exemple,

void main(void) {

char reponseQ1, reponseQ2, reponseQ3;

printf("1. Aimez-vous la Chimay bleue ?\n");

do {

printf("Réponse (o/n) : ");

scanf\_s("%c", &reponseQ1, 1);

getchar();

} while(reponseQ1 != 'o' && reponseQ1 != 'n');

printf("2. Aimez-vous la Blanche ?\n");

do {

printf("Réponse (o/n) : ");

scanf\_s("%c", &reponseQ2, 1);

getchar();

} while(reponseQ2 != 'o' && reponseQ2 != 'n');

printf("3. Aimez-vous le fromage d’abbaye ?\n");

do {

printf("Réponse (o/n) : ");

scanf\_s("%c", &reponseQ3, 1);

getchar();

} while(reponseQ3 != 'o' && reponseQ3 != 'n');

if(reponseQ1 == 'o' && reponseQ2 == 'o' && reponseQ3 == 'o') {

printf("Vous avez toutes vos chances en C...") ;

} else {

printf("Vous allez devoir étudier à fond !");

}

}

La rédaction d'un tel programme parait simple puisque ce ne sont que des copier/coller avec quelques modifications. Cependant, le code est inutilement long et il contient des répétitions.

En procédant de cette manière, vous prenez des risques, aussi bien en termes de **lisibilité** qu’en terme de **maintenabilité**.

En effet, toute modification du code répété, par exemple ajouter un choix de réponse, impose la modification de chacune des conditions et de la chaîne de caractères proposant ces choix. Par exemple,

do {

printf("Réponse (o/O/n/N) : ");

scanf\_s("%c", &reponseQ3, 1);

getchar();

} while(reponseQ3 != 'o' && reponseQ3 != 'O' &&   
 reponseQ3 != 'n' && reponseQ3 != 'N');

Dans ce cas-ci il ne s’agit que d’un questionnaire sans grande importance, mais qu’en est-il si le bout de code répété contient des instructions influençant le résultat du programme ? En cas d’oubli d’une modification, le résultat de celui-ci peut devenir incertain.

Or, le fait d’être lisible et facilement modifiable est un critère essentiel pour l’élaboration d’un programme informatique. Cela devient même une nécessité incontournable lorsqu’on travaille en équipe et que le programme doit être mis à jour par des personnes qui n’en sont pas les concepteurs.

Il faut donc opter pour une autre stratégie, qui consiste à

* séparer ces instructions du corps de la fonction principale,
* leur donner un nom (ici reponseValide) ,
* faire appel à ces instructions (qui ne figurent donc plus qu’en un seul exemplaire) à chaque fois qu’on en a besoin via le nom qu’on leur a donné.

Sans vous demander de comprendre les nouveaux éléments syntaxiques (void, return…) qui sont expliqués ci-après, voici ce que ça donne en passant par une fonction.

#include <stdio.h>

char reponseValide(void) {

char reponse;

do {

printf("Reponse (o/n) : ");

scanf\_s("%c", &reponse, 1);

getchar();

} while(reponse != 'o' && reponse != 'n');

return reponse;

}

void main(void) {

char reponseQ1, reponseQ2, reponseQ3;

printf("1. Aimez-vous la Chimay bleue ?\n");

reponseQ1 = reponseValide();

printf("2. Aimez-vous la Blanche ?\n");

reponseQ2 = reponseValide();

printf("3. Aimez-vous le fromage d'abbaye ?\n");

reponseQ3 = reponseValide();

if(reponseQ1 == 'o' && reponseQ2 == 'o' && reponseQ3 == 'o') {

printf("Vous avez toutes vos chances en C...") ;

} else {

printf("Vous allez devoir étudier à fond !");

}

}

Créez un nouveau fichier source, nommé soudoyer.c et copiez-y le code source proposé ci-dessus en ajoutant ce qui est nécessaire. Exécutez-le afin de voir le résultat.

Grâce à cette façon de faire, la lisibilité est assurée et il suffit de faire une seule modification au bon endroit, pour que cette modification prenne effet dans la totalité du programme. Il s’agit du concept de **point de modification unique**.

Vous allez ainsi apprendre à décomposer votre programme en une série de **sous-programmes**, appelés fonctions. Chaque fonction doit correspondre à une tâche algorithmique simple/élémentaire/**unitaire**.

Quelques questions à se poser afin de s'assurer que c'est le cas :

* Le sous-programme permet-il de réaliser **une seule** **tâche** ?
* Puis-je lui associer un nom **court** et **pertinent** ?
* Puis-je décrire cette tâche en une **phrase simple** ?
* Est-ce que sa définition favorise la **réutilisabilité** ?
* Reste-il, dans le code, des tâches qui pourraient être extraites sous la forme de sous-programmes (soit parce qu'elles sont répétées et contreviennent au concept de **point de modification unique**, soit parce qu'il s'agit d'une tâche dite **unitaire**) ?

Attention à ne pas exagérer dans la découpe en fonctions au risque d’obtenir un code illisible et des fonctions d’une ligne ou deux qui ne sont appelées qu’une seule fois… Le programme qui en résulte doit être plus lisible, plus facilement modifiable et permettre d’avoir une meilleure vue d’ensemble sur le programme.

# Qu’est-ce qu’une fonction

Une fonction est la traduction en C de la notion de module vue au cours de « Principes de programmation ». C’est une séquence d’instructions pouvant être exécutée indépendamment. La fonction peut être exécutée à n’importe quel moment dans un programme comme s’il s’agissait d’une seule instruction.

L’utilisation des fonctions nécessite donc deux étapes :

* la définition de la fonction, et
* l’appel de cette fonction.

La **définition d’une fonction** commence par l’entête de la fonction et se poursuit par le corps de la fonction placé entre accolades {}.

type\_de\_retour **nom** **(**paramètres\_formels**) {** // entête\_de\_la\_fonction

corps\_de\_la\_fonction

**}**

L’**entête de la fonction**, plus souvent appelée **signature de la fonction**, est composé

* du type de la valeur de retour,
* du nom de la fonction, et
* des paramètres d’entrée nécessaires à la fonction.

Le **corps de la fonction** est constitué d’une séquence d’instructions. Parmi celles-ci peuvent se trouver des instructions commençant par le mot réservé return, qui sert à renvoyer une valeur à la fonction appelante.

L’**appel de la fonction** est écrit comme suit.

***nom*(***paramètres\_effectifs***);**

Ces différentes notions vont être détaillées dans la suite de cet atelier !

## Spécification

Une **spécification** est une façon de préciser ce qu’un morceau de programme fait.

En fait, il s’agit de déterminer ce qui doit être produit par le bout de code en question, mais sans préciser la façon dont il doit le produire.

Pour permettre au programmeur de savoir si son code répond aux attentes initiales, on lui fournit ce qu’on appelle une spécification, c’est-à-dire une **précondition** et une **postcondition**. Une précondition est une condition à vérifier **avant** l’exécution du bout de code. Une postcondition est une condition qui doit être remplie **après** l'exécution du bout de code.

Par exemple, dans la bibliothèque standard math.h, la fonction sqrt permet de déterminer la racine carrée d’un nombre.

Pour vérifier ce qui se passe réellement, créez un nouveau fichier source racineCarree.c et ajoutez-y le code ci-dessous.

#include <stdio.h>

#include <math.h>

void main(void) {

double nbLu;

printf("Nombre : ");

scanf\_s("%lf", &nbLu);

printf("%lf", sqrt(nbLu));

}

Compilez et exécutez le programme. Pour voir ce qu’il fait, entrez 9 comme valeur. Le résultat affiché vous semble-t-il correspondre à ce que vous attendez ? Exécutez à nouveau le programme, mais cette fois-ci, entrez -9 comme valeur. Que se passe-t-il ? À votre avis, pourquoi ?

En fait, selon la définition mathématique de la racine carrée dans , elle ne peut être calculée que pour des nombres positifs ou nuls.

Dans la norme du C, la dernière spécification de la fonction sqrt se trouve dans le document officiel <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1778.pdf>. La spécification en question est écrite comme montré à la Figure 1.

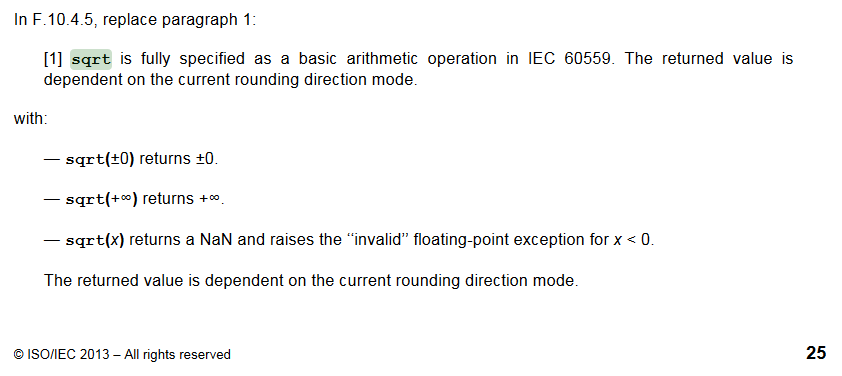


Figure - spécification de sqrt dans la Norme C11

NaN signifie « Not a Number » et est utilisé dans de nombreux langages pour préciser que la valeur en question est le résultat d'une opération arithmétique invalide. Vous en saurez un peu plus sur sa représentation en mémoire dans le cours d’« Architecture des ordinateurs ». En effet, il s’agit d’un nombre représenté en virgule flottante suivant le standard IEEE 754.

Deux leçons sont à tirer de ce petit exercice :

* il est nécessaire de lire les spécifications des fonctions existantes avant de les utiliser afin de vérifier que vos programmes ne provoquent pas d’erreurs inattendues.
* il est nécessaire de proposer une spécification claire des fonctions que vous écrivez de façon à permettre leur utilisation sans risque d’erreur.

Une spécification de la fonction sqrt permet de la comprendre à sa simple lecture. Une façon de proposer une spécification pour cette fonction pourrait être la suivante :

**Précondition :** x est un nombre ≥ 0

**Postcondition :** résultat² = x et résultat ≥ 0

Dans cet exemple, grâce à une spécification bien écrite, le programmeur sait qu’il n’est pas question de fournir un nombre négatif à cette fonction sous peine de provoquer une erreur ou d’avoir un résultat indéterminé, selon le langage utilisé.

Il sait également que, s’il respecte la précondition en fournissant un nombre positif à cette fonction, celle-ci renvoie toujours le résultat attendu, c’est-à-dire la racine carrée du nombre fourni.

La fonction ainsi spécifiée, il ne reste qu’à vérifier les valeurs avant de les transmettre à la fonction, afin qu’elles respectent la précondition !

## Fonction ou procédure ?

En algorithmique, on distingue la notion de fonction de celle de procédure :

* une **fonction** joue le rôle d’une **expression** évaluable,
* une **procédure** joue le rôle d’une **instruction**.

Le langage C ne fait pas la distinction, à strictement parler, entre le concept de procédure et celui de fonction. Cependant, le programmeur peut réaliser une procédure à l'aide d'une fonction qui ne renvoie aucune valeur.

Nom de fonction et de procédure

Qu’il s’agisse d’une procédure ou d’une fonction, le nom de celle-ci est important car il permet de savoir ce que le bout de code associé fait.

Dans le cas d’une **fonction**, comme elle renvoie un résultat, on utilise un **nom** qui décrit la/les valeur(s) renvoyée(s).

Dans le cas d’une **procédure**, on utilise un **verbe** qui décrit l’action accomplie.

En C, le terme générique est donc **fonction**…

# Première fonction… ou plutôt première procédure

Comme vous l’avez vu en « Principes de programmation », certains modules n’ont ni paramètre d’entrée, ni paramètre de sortie (voir Figure 2).

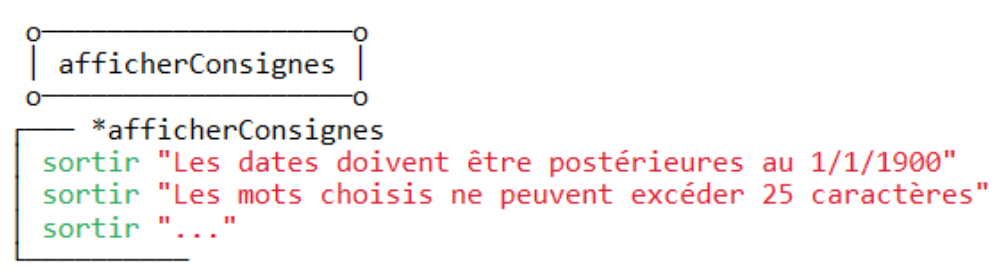


Figure - Module sans entrée et sans sortie

**Précondition :** /

**Postcondition :** les consignes sont affichées

Dans un nouveau fichier source consignes.c, définissez la fonction principale sans rien prévoir comme corps. Écrivez, sous la fonction principale, la procédure qui correspond au module ci-dessus. Comme le module ne renvoie rien et qu’il ne reçoit rien, en C, la procédure a void comme type de retour et void comme paramètre d’entrée.

void signifie « vide » et est utilisé dans les signatures des procédures/fonctions pour préciser que soit elle n’a besoin d’aucun paramètre en entrée, soit elle n’a aucun paramètre en sortie, soit les deux.

La signature de cette procédure est la suivante.

void afficherConsignes(void) {

...

}

Dans la fonction principale que vous avez prévue, écrivez l’appel de la procédure.

Comme celle-ci n’a aucun argument, il suffit d’écrire son nom suivi de parenthèses ().

afficherConsignes();

Les parenthèses permettent de distinguer les appels de procédure/fonction de tout autre instruction, elles sont donc importantes ! N’oubliez pas le ; à la fin de l’instruction.

Vous devriez avoir le programme suivant.

#include <stdio.h>

void main(void) {

afficherConsignes();

}

void afficherConsignes(void) {

printf("Les dates doiventt etre superieures a 1/1/1900");

printf("Les mots choisis ne peuvent exceder 25 caracteres");

printf("...");

}

Compilez le programme. Que se passe-t-il ? À votre avis, pourquoi ?

Voici une piste de réponse : le compilateur C analyse les instructions les unes à la suite des autres.

En fait, lors de la compilation, le fichier est analysé ligne par ligne. Or, lors de l’appel de la procédure afficherConsignes dans la fonction principale, aucune définition n’a encore été proposée pour la procédure puisque, dans le fichier source, elle se trouve après la fonction principale.

Le compilateur fait alors l’hypothèse que le paramètre de sortie est de type int. Au moment où il arrive sur la définition de la procédure, il se trouve face à un conflit puisqu’il a implicitement prévu une définition pour celle-ci… La procédure est donc définie deux fois (une fois implicitement et une fois via la définition que vous avez écrite après la fonction principale) et cela n’est pas admis dans la syntaxe du C.

Une procédure/fonction, ne peut être définie qu’une seule fois dans un programme !

Que faut-il faire pour résoudre ce souci ?

Une solution est de placer l’appel dans les lignes qui suivent sa définition. Dans ce cas-ci, il suffit de définir la procédure afficherConsignes avant la fonction principale. Une autre solution sera proposée plus tard, appelée le prototype de la fonction.

Inverser la définition de la fonction principale avec celle de la procédure et compilez à nouveau le programme. Exécutez-le.

Le programme devient le suivant.

#include <stdio.h>

void afficherConsignes(void) {

printf("Les dates doiventt etre superieures a 1/1/1900");

printf("Les mots choisis ne peuvent exceder 25 caracteres");

printf("...");

}

void main(void) {

afficherConsignes();

}

En résumé, voici ce qui s’est produit :

* lorsque vous écrivez la fonction principale **avant** la définition de la procédure, l’appel de la procédure se trouve avant sa définition. Au moment où il arrive sur l’instruction d’appel, le compilateur n’a pas encore pris connaissance de la définition de la procédure et suppose que le type de retour est int. Quand il arrive sur la signature de la fonction, il se retrouve face à une redéfinition de fonction, ce qui provoque une erreur.
* lorsque vous placez la fonction principale **après** la définition de la procédure, l’appel de la procédure ne pose plus de soucis au compilateur puisqu’il connaît la procédure au moment de l’appel et il peut exécuter le code qui lui est associé.

Maintenant que vous avez écrit votre première procédure et que vous savez y faire appel, modifiez la fonction principale afin de bien visualiser le flux des instructions :

* Ajoutez l’instruction suivante : printf("Avant l'appel de la fonction.");.
* Faites un appel à la procédure.
* Ajoutez l’instruction suivante : printf("Après l'appel de la fonction.");.

Le programme devient le suivant.

#include <stdio.h>

void afficherConsignes(void) {

printf("Les dates doiventt etre superieures a 1/1/1900");

printf("Les mots choisis ne peuvent exceder 25 caracteres");

printf("...");

}

void main(void) {

printf("Avant l'appel de la fonction.");

afficherConsignes();

printf("Après l'appel de la fonction.");

}

Comme vous l’avez vu dans l’atelier précédent, le flux de l’exécution des instructions est directement lié à l’ordre de ces instructions. Ce flux peut être modifié grâce à certaines instructions de contrôle de flux.

Avez-vous compris qu’un appel de procédure/fonction modifie le flux des instructions, au même titre que les alternatives et les répétitives ? Assurez-vous que c’est le cas !

# Avec des paramètres

Dans certains cas, la procédure a besoin d’informations provenant d’une autre procédure/fonction (voir Figure 3).

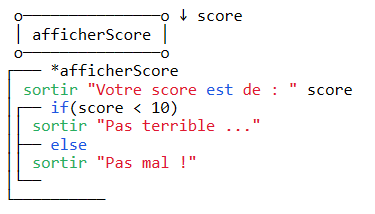


Figure - Module avec entrée et sans sortie

Dans un nouveau fichier source score.c, définissez la fonction principale sans rien prévoir comme corps. Écrivez ensuite le code de la procédure qui correspond au module ci-dessus, au bon endroit (vous avez normalement compris où placer les définitions de procédure/fonction par rapport à leur appel).

Comme le module ne renvoie rien, en C, la procédure a void comme type de retour.

Comme le module reçoit le score en entrée, il faut préciser, dans la signature de la procédure, qu’elle doit recevoir une valeur correspondant au score.

Conception/Analyse

Une question importante est de savoir sous quelle forme le score est exprimé : il peut s’agir soit d’un entier, soit d’un réel, soit d’une lettre... Il est donc nécessaire de faire un choix selon le contexte d’utilisation du programme.

Ce rôle revient à l’analyste de l’application (qui peut aussi être le programmeur). Il est important de vous rendre compte que c’est avant de rédiger le code que la **précondition** et la **postcondition** doivent être fixées. En effet, le format, et donc le type, du score doit être choisi avant d’arriver à la phase de codage.

Dans le cas où vous n’avez pas les informations nécessaires, le mieux est de retourner vers l’analyste et de lui demander ce qu’il en est. L’analyste peut, vu qu’il connait le contexte d’utilisation du programme, préciser le format de l’information reçue.

Dans le cadre de ce cours, ce sont soit les énoncés des exercices qui l’imposent, soit vous qui devez faire le bon choix…

Concernant cet exercice, les scores sont considérés comme étant des entiers positifs. La validation est donc supposée avoir été réalisée lors de l’obtention ou du calcul du score.

**Précondition :** score est un nombre entier ≥ 0

**Postcondition :** un message différent est affiché selon que le score est inférieur à 10 ou pas

La signature de cette procédure est la suivante.

void afficherScore(int score) {

...

}

Complétez le corps de la procédure avec les instructions C correspondant au DA ci-dessus. Dans la fonction principale que vous avez prévue, écrivez l’appel de la procédure. Compilez le code.

Si vous avez procédé comme pour la procédure précédente, vous devriez avoir un message d’erreur précisant qu’il manque des arguments lors de l’appel.

En effet, comme la définition de la procédure le précise, elle a besoin d’un score comme paramètre d’entrée. Il faut écrire son nom suivi des parenthèses () dans lesquelles vous devez préciser la valeur du score.

Plusieurs expressions peuvent être utilisées comme valeur à fournir à la procédure, mais quelles qu’elles soient, elles doivent avoir un résultat entier.

Testez les différents appels ci-dessous et observez le résultat.

afficherScore(7);

afficherScore(5 + 9);

// déclarez score dans la fonction principale et initialisez-la à 13

afficherScore(score);

// déclarez points dans la fonction principale et initialisez-la à 5

afficherScore(points);

...

La procédure est ainsi devenue paramétrable, c’est-à-dire que vous pouvez l’appeler avec différentes valeurs de score. Ainsi, le score est un paramètre de la procédure, plus précisément un **paramètre d’entrée** puisqu’il est fourni à la procédure. Ce paramètre peut alors être utilisé dans le **corps de la procédure**, et uniquement dans ce bloc (voir Portée des identifiants) !

Paramètres formels et paramètres effectifs

Lorsque vous définissez une fonction, vous devez lister les paramètres d’entrée. Il s’agit d’associer un nom de variable et un type à chacune des valeurs que la fonction est supposée recevoir lors de son appel. Chacune de ces variables est appelée **paramètre formel**, car sa valeur n’est pas connue au moment de la **définition** de la fonction. Dans l’exemple ci-dessous, les paramètres formels sont degats et modificateur.

void afficherDegatsModifies(**int** **degats**, **double modificateur**) {

printf("%.2f",degats \* modificateur);

}

Lors de son **appel**, chaque valeur fournie à la fonction est un **paramètre effectif**. Les paramètres effectifs sont également appelés **arguments**. Ci-dessous, voici un exemple d’appel de la fonction avec deux valeurs qui sont les paramètres effectifs de la fonction dans le cadre de cet appel.

afficherDegatsModifies(28, 0.8)

Le compilateur C a notamment pour tâche de vérifier la cohérence entre le type des paramètres effectifs et celui des paramètres formels. En effet, si la valeur fournie à la fonction lors de l’appel n’est pas du type imposé lors de la définition de la fonction, cela provoque une erreur à la compilation !

Maintenant que vous avez compris la notion de paramètre, modifiez la procédure afin qu’elle reçoive le score ainsi que le score maximum et affichez le pourcentage correspondant au lieu du score. Adaptez la condition de l’alternative pour que les pourcentages inférieurs à 50 soient considérés comme « Pas terrible... ».

Par exemple, si le score est de 18 et que le score maximum est de 30, le message affiché est « Votre score est de : 60.0 %. Pas mal ! ».

**Précondition :** score est un nombre entier ≥ 0   
maximum est un nombre > 0

**Postcondition :** un message différent est affiché selon que le rapport en % entre score et maximum est inférieur à 50 ou pas

Faites appel à la procédure en lui fournissant le score et le score maximum de votre choix. Le résultat est-il celui attendu ?

Faites à nouveau appel à la procédure, mais cette fois-ci, inversez les deux arguments. Que se passe-t-il ? À votre avis, pourquoi ? Qu’en concluez-vous ?

# Premières fonctions !

Une fonction se définit exactement de la même façon qu’une procédure à cela près qu’elle renvoie un résultat… La fonction a besoin d’informations provenant d’une autre procédure/fonction et renvoie un résultat (voir Figure 4).

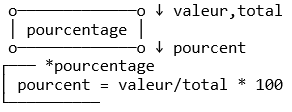


Figure - Module avec entrées et sortie

Dans un nouveau fichier source pourcentage.c, définissez la fonction principale sans rien prévoir comme corps.

Vous allez maintenant écrire la fonction qui correspond au module ci-dessus.

En plus de préciser le nom et le type de chacun des paramètres formels, il est nécessaire de préciser le type de la valeur renvoyée.

Comme le module renvoie un pourcentage, en C, la procédure a double comme type de retour.

Comme le module reçoit la valeur et le total en entrées, il faut préciser, dans la signature de la fonction, qu’elle doit recevoir les deux valeurs…

Concernant cet exercice, la valeur et le total sont considérés comme étant des entiers positifs. La validation est donc supposée avoir été réalisée avant l’appel de la fonction.

**Précondition :** valeur est un nombre entier ≥ 0

total est un nombre > 0

**Postcondition :** renvoie le rapport en % entre valeur et total

La signature de cette fonction est la suivante.

double pourcentage(int valeur, int total) {

...

}

Complétez le corps de la fonction avec les instructions C correspondant au DA ci-dessus.

Le mot clé return permet de renvoyer le résultat de l’évaluation de l’expression qui le suit.

return *expression*;

L’instruction suivante permet de renvoyer le produit des valeurs des deux variables citées.

return degats \* modificateur;

Une solution pour le pourcentage est de renvoyer directement le résultat de l’expression.

double pourcentage(int valeur, int total) {

return (double)valeur / total \* 100;

}

Comme dit plus haut, plusieurs expressions peuvent être utilisées comme arguments de la fonction, mais quelles qu’elles soient, elles doivent chacune avoir le même type que le paramètre formel associé.

Dans la fonction principale que vous avez prévue, écrivez l’appel de la fonction.

Testez l’appel ci-dessous et observez le résultat.

pourcentage (11, 15);

Si vous avez suivi les étapes précédentes, vous ne devriez pas avoir de résultat visible… C’est frustrant !?

À votre avis, comment résoudre ce problème ? (Il y a plusieurs façons de faire…)

Vous pourriez simplement afficher directement le résultat produit en fournissant l’appel de la fonction à la fonction printf(). Mais si vous voulez utiliser le résultat produit dans la suite du programme, vous n’avez toujours pas récupéré le résultat…

En effet, lorsque vous appelez la fonction, vous lui demandez de calculer un pourcentage, et à la fin de la fonction, elle renvoie en effet le résultat du calcul, mais vous n’en faites rien.

Un indice : « affectation » …

Modifiez l’appel de la fonction de façon à affecter ce résultat à une variable qui a été déclarée dans la fonction principale. Le code de la fonction principale devient semblable au suivant.

void main(void) {

double resultat;

resultat = pourcentage(15, 52);

printf("%.1f", resultat);

}

Après avoir compilé ce programme, testez-le et vous pourrez enfin observer le résultat affiché !

Dans un nouveau fichier source cube.c, écrivez une fonction qui reçoit un entier en argument et détermine son cube (renvoyez directement le résultat sans passer par une variable intermédiaire).

Faites ensuite appel à la fonction en lui fournissant un nombre entier. N’oubliez pas de faire en sorte de récupérer le résultat et affichez celui-ci.

Maintenant que vous avez écrit cette fonction et que vous savez y faire appel et récupérer le résultat produit, organisez votre fonction principale comme proposé ci-dessous :

* Ajoutez l’instruction suivante : printf("Avant l’appel de la fonction.");.
* Faites un appel à la fonction et récupérez le résultat.
* Affichez le résultat récupéré.
* Ajoutez l’instruction suivante : printf("Après l’appel de la fonction.");.

À nouveau, assurez-vous d’avoir compris qu’un appel de fonction modifie le flux des instructions, au même titre que les alternatives, les répétitives et les procédures.

Dans un nouveau fichier source pair.c, écrivez une fonction qui reçoit un entier en argument et détermine s'il est pair ou non (évitez d’utiliser une alternative et renvoyez directement le résultat de l’évaluation de la condition). Dans la fonction principale, ajoutez les instructions nécessaires pour afficher un message adéquat en fonction de la valeur renvoyée par la fonction.

Effet de bord (bis)

En toute généralité, un **effet de bord** (en anglais, *side effect*) est le fait de provoquer quelque chose (un effet, un changement…) en plus de l'effet "principal"… C'est un effet secondaire.

Une fonction qui calcule quelque chose et qui, en plus de cela, affiche un message à l’écran, demande une information à l'utilisateur, joue un son, sauvegarde le résultat dans un fichier, modifie une variable globale… est une fonction à effet de bord.

int somme2Entiers(int x, int y) {

int somme = x + y;

printf("Somme : %d", somme); // effet de bord !

return somme;

}

On essaie d'éviter au maximum ces effets de bord, car ils compliquent la lisibilité des programmes et/ou nuisent à leur réutilisabilité.

Bien sûr, certaines fonctions/procédures ont pour seul objectif d’interagir avec l'utilisateur (via des printf et des scanf\_s). Celles-ci provoquent bien sûr des effets de bord, mais qui, dans ce cas-ci, se justifient.

# Fonctions sans paramètre d’entrée

Certains modules n’ont besoin d’aucun argument, mais ils renvoient une ou plusieurs valeur(s). C’est le cas des modules d’obtention.

En C, les fonctions ne peuvent renvoyer qu’une seule valeur via l’instruction return. Des astuces existent pour contrer cette limite, mais elles seront abordées dans le cours de « Langage de programmation avancé ».

Le module suivant peut être traduit en C directement puisqu’il ne renvoie qu’une seule valeur (voir Figure 5).

Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement

Figure - Module sans entrée et avec sortie

Après les 3 premières versions, vous ne devriez plus avoir de problème à traduire ce module en C.

Dans un nouveau fichier source entierLu.c, écrivez une fonction qui permet d’obtenir un entier valide et de le renvoyer à la fonction principale. Dans la fonction principale, ajoutez les instructions nécessaires pour afficher cet entier, après l’avoir obtenu.

# Tableaux et fonctions

Il existe deux façons de passer des paramètres à une fonction. Selon que vous passez une variable de type scalaire ou de type tableau, le passage de paramètres ne fonctionne pas de la même manière.

Dans le cas du passage d’un type scalaire, le passage est un « passage par valeur/copie ».

Dans le cas du passage d’un tableau en paramètre d’une fonction, le passage est un « passage par adresse ».

Cette partie va tenter de vous expliquer ce dont il s’agit.

## Le passage par valeur/copie

Les arguments de type scalaire passent par **valeur/copie**. Ainsi, une fonction ne reçoit pas l’original mais reçoit une copie de la valeur de chaque argument.

De ce fait, aucune modificationde la valeur d’un paramètre de la fonction, effectuée à l'intérieur de celle-ci, n'est prise en comptedans la fonction appelante.

Pour récupérer la valeur modifiée, il faut soit **renvoyer** la valeurmodifiée à la fonction appelante, soit utiliser un **pointeur**. Les pointeurs et l’allocation dynamique de mémoire seront vus dans le cours de « Langage de programmation avancé ».

Pour reprendre un exemple simple, voici un programme qui est supposé mettre en majuscule un caractère reçu en paramètre. Les fonctions getchar et putchar sont utilisées pour récupérer et afficher un seul caractère. Elles sont expliquées ci-après.

char toupper(char c);

void main (void) {

char caract = 'a';

toupper(caract);

putchar(caract); // affiche un caractère

}

char toupper(char c) {

// voir cours d’Architecture des systèmes

// ou dans la bibliothèque ctype.h

return c;

}

putchar et getchar

En C, il existe plusieurs façons d’obtenir et d’afficher un caractère. La première façon qui a été vue est l’utilisation des fonctions scanf\_s et printf avec le format de lecture/affichage %c.

char caractere;

scanf\_s("%c",&caractere, 1);

...

printf("%c", caractere);

Vous avez déjà compris et utilisé la fonction getchar dans les ateliers et exercices précédents, mais sans mémoriser le caractère obtenu.

Pour mémoriser la valeur obtenue, il faut l’affecter à une variable.

<var\_name> = getchar();

Il faut y remplacer <var\_name> par le nom de la variable qui permet de mémoriser cette valeur.

Une autre façon d’afficher un caractère est d’utiliser la petite sœur de getchar, c’est-à-dire putchar.

putchar(<var\_name>);

Il faut y remplacer <var\_name> par le nom de la variable dont on désire afficher la valeur.

Les instructions proposées ci-dessus deviennent…

char caractere;

caractere = getchar();

...

putchar(caractere);

Ainsi, lors de l’exécution du programme, une partie de la mémoire est allouée à la fonction principale. Dans cet espace, on associe 8 bits (la taille d’un caractère) à la variable locale caract pour y mémoriser la valeur initiale 'a', comme montré à la   
Figure 1.

Une image contenant texte, capture d’écran, ligne, Rectangle

Description générée automatiquement

Figure 1 - Pile d'appels : main

Lors de l’appel de la fonction toupper, une partie de la mémoire lui est allouée avec à nouveau 8 bits associé à la variable locale c, comme montré sur la Figure 2.

Une image contenant texte, capture d’écran, ligne, nombre

Description générée automatiquement

Figure 2 - Pile d'appels : toupper

La fonction toupper modifie la valeur de c dans la partie de la mémoire qui lui est associée, en local, comme montré à la Figure 3.

Une image contenant texte, capture d’écran, ligne, nombre

Description générée automatiquement

Figure 3 - Pile d'appel : toupper

Ensuite la fonction toupper se termine et la partie de mémoire qui lui est allouée est libérée. Le résultat de cette étape est le même que celui de la Figure 1. En effet, la variable caract n’a pas été modifiée puisqu’il s’agit d’un passage par valeur/copie. La modification effectuée en local de la fonction toupper est perdue…

Il est nécessaire de renvoyer la valeur modifiée de la variable c à la fin du traitement de la fonction toupper, et de l’affecter à la variable caract dès son « retour » dans la fonction main.

Pour tenir compte de la modification effectuée par la fonction toupper, le programme doit être écrit comme ci-dessous.

char toupper(char c);

void main (void) {

char caract = 'a';

caract = toupper(caract);

putchar(caract);

}

char toupper(char c) {

// voir cours d’Architecture des systèmes

// ou dans la bibliothèque ctype.h

return c;

}

## Le passage par adresse

Les arguments de type tableau passent par **adresse**. Ainsi, une fonction reçoit l’adresse du tableau (via son nom) et travaille sur le tableau original !

De ce fait, toute modification, effectuée à l'intérieur de la fonction appelée, d’un tableau reçu en argument est prise en comptedans la fonction principale (appelante).

Attention, en C, une fonction ne peut jamais renvoyer un tableau au moyen de l’instruction return ! En effet, comme ils sont passés par adresse, il est inutile de les renvoyer.

Voici un autre exemple qui propose une fonction d’obtention chaineLue pour garnir une chaîne de caractères, utilisée dans ce cas pour obtenir un nom.

void chaineLue(char chaine[]);

void main (void) {

char nom[TAILLE\_CHAINE];

chaineLue(nom);

puts(nom);

}

void chaineLue(char chaine[]) {

…

gets(chaine);

}

Comme dans l’exemple précédent, lors de l’exécution du programme, une partie de la mémoire est allouée à la fonction principale. Dans cet espace, on associe   
TAILLE\_CHAINE \* 8 bits (la taille du tableau \* la taille d’un caractère) à la variable locale nom sans l’initialiser, comme montré à la Figure 4.

Une image contenant texte, capture d’écran, ligne, diagramme

Description générée automatiquement

Figure 4 - Pile d'appels : main

Lors de l’appel de la fonction chaineLue, une partie de la mémoire lui est allouée mais cette fois avec la place nécessaire pour mémoriser l’adresse de tableau (et pas TAILLE\_CHAINE \* 8 bits) dans la variable locale chaine, comme montré sur la Figure 5.

Une image contenant texte, capture d’écran, ligne, diagramme

Description générée automatiquement

Figure 5 - Pile d'appel : chaineLue

Comme le nom du tableau correspond à son adresse, lors de l’appel de gets(chaine), c’est l’adresse mémorisée dans chaine qui est utilisée comme cible pour mémoriser la chaîne de caractères lue. Ainsi, la valeur est mémorisée à l’adresse en question, qui est celle de la variable nom. La fonction chaineLue modifie donc directement la valeur de nom dans la partie de la mémoire associée à la fonction principale, comme montré à la   
Figure 6.

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

Figure 6 - Pile d'appels : main

Il est à noter que, comme le tableau est passé par adresse, il n’est pas nécessaire de préciser sa taille dans le prototype et l’entête de la fonction.

Dans un nouveau fichier source minimum.c, écrivez un programme qui affiche la plus petite valeur d’un tableau. Le minimum doit être recherché dans une fonction.

Dans un premier temps, initialisez le tableau lors de sa déclaration dans la fonction principale.

Modifiez ensuite le programme en ajoutant une fonction qui permet de garnir le tableau en demandant chacune des valeurs à l’utilisateur.

Vous devriez arriver à un résultat semblable à celui proposé ci-dessous.

#define NB\_NOMBRES 10

void nombresLus(int nombres[], int taille) {

for (int i = 0; i < taille; i++) {

printf("Nombre %d : ", i + 1);

scanf("%d", &nombres[i]);

}

}

int minimum(int nombres[], int taille) {

int min = nombres[0];

for (int i = 1; i < taille; i++) {

if (nombres[i] < min) {

min = nombres[i];

}

}

return min;

}

void main(void) {

int min;

int nombres[NB\_NOMBRES];

nombresLus(nombres, NB\_NOMBRES);

min = minimum(nombres, NB\_NOMBRES);

printf("%d", min);

}

# Portée des identifiants (variables et fonctions)

Pour mieux comprendre le concept de portée en C, il faut d’abord comprendre ce qu’est un identifiant.

Un **identifiant** est un nom qui sert à désigner de façon unique les éléments du programme : variable, fonction, structure, énumération… Dans le cadre de cet atelier, seules les variables et les fonctions sont abordées.

Attention, les explications qui suivent ciblent le C et peuvent différer des autres langages tels que Javascript ou Python.

La portée d’un identifiant est définie par la portion du code source dans laquelle cet identifiant est accessible. Sa portée dépend donc de l’endroit où il est déclaré.

Toute variable ou fonction doit être déclarée avant d’être utilisée.

Il y a deux niveaux de portée : une portée globale et une portée locale.

Un identifiant à **portée globale** est visible dans l'ensemble du code source qui suit sa déclaration.

Un identifiant à **portée locale** n'est visible qu'au sein du bloc ou de la fonction dans lequel il est déclaré.

Une fonction est utilisable à partir du moment où elle est définie. Toutes les fonctions ont donc une portée globale au sein d’un même fichier source, mais seulement à partir de leur définition. Dans le cas où le programme devient plus long et où l’organisation des fonctions est plus complexe, l’utilisation de **prototypes** est nécessaire.

Une variable peut être déclarée à trois endroits du code source :

* à l’intérieur d’une fonction ou d’un bloc : ce sont des **variables locales**.
* en dehors de toutes les fonctions : ce sont des **variables globales**.
* dans la définition des paramètres de fonction : ce sont les **paramètres formels**.

Voici quelques informations supplémentaires sur ces diverses notions.

## Prototypes de fonction

Dans cet atelier, vous avez appris que lorsque la définition de la fonction se trouve après son appel, cela pouvait engendrer soit des erreurs à la compilation, soit un résultat indéterminé.

Or, dans certains cas, on peut vouloir définir une fonction après la fonction principale ou toute autre fonction qui y fait appel.

De plus, si les fonctions sont nombreuses et que plusieurs fichiers sources sont nécessaires, il devient compliqué de gérer l’ordre dans lequel les fonctions sont définies, et certaines de celles-ci peuvent être nécessaires dans d’autres fichiers que dans le fichier où elles sont définies. Vous en apprendrez plus sur la programmation multi-fichiers dans le cours de « Langage de programmation avancé ».

Pour en revenir à ce que vous connaissez déjà, reprenez le programme pourcentage.c que vous avez fait au point E de cet atelier. Placez la définition de la fonction pourcentage après celle de la fonction principale.

Compilez le programme. Vous avez à nouveau le problème rencontré précédemment.

Le compilateur a besoin de connaître la signature de la fonction lors de l'appel à cette fonction. Il faut donc déclarer la fonction avant son appel. Pour ce faire, on utilise un prototype.

Un prototype est la façon de déclarer une fonction. Il précise

* le nom de la fonction,
* le type de la valeur de retour,
* le nombre des paramètres, et
* le type de chacun des paramètres.

Repartez à nouveau de pourcentage.c. Ajoutez le prototype de la fonction en début de fichier, entre les directives préprocesseur et la fonction principale. Le prototype correspond à la signature de la fonction suivi d’un ;. Vous devriez avoir le code suivant.

double pourcentage(int valeur, int total);

void main(void) {

double resultat;

resultat = pourcentage(15, 52);

printf("%.1f", resultat);

}

double pourcentage(int valeur, int total) {

return (double)valeur / total \* 100;

}

Selon la syntaxe du langage, on peut se limiter à écrire le type de chacun des paramètres formels. Cependant, pour une question de lisibilité, il est préférable de préciser également le nom de chacun d’eux !

Le prototype fournit donc les informations nécessaires pour que le compilateur connaisse les fonctions avant leur appel. Ainsi vous allez pouvoir définir vos fonctions n'importe où dans notre code source.

Il faut donc toujours mettre le prototype au-dessus du premier appel. Un prototype n’alourdit en aucun cas l'exécutable final ; il est ignoré à la fin de la compilation.

## Variables locales

Vous avez déjà abordé cette notion dans l’atelier précédent.

Les variables déclarées dans une fonction ou dans un bloc sont appelées variables locales. Pour rappel, elles ne peuvent être utilisées que par des instructions qui se trouvent à l’intérieur de cette fonction ou de ce bloc.

## Variables globales

**Aucune variable globale n’est tolérée au bloc 1.**

Les variables globales sont définies en dehors d’une fonction, généralement en début de fichier source, entre les directives préprocesseur et les prototypes.

Elles gardent leur valeur tout au long de la durée de vie de votre programme et peuvent être consultées dans n’importe quelle fonction du fichier source.

Un programme peut avoir le même nom pour les variables locales et globales, mais la variable locale est prioritaire sur la variable globale dans une fonction.

## Paramètres formels

Les paramètres formels sont traités comme des variables locales dans une fonction et ils ont priorité sur les variables globales.

## Initialisation des variables locales et globales

Lorsqu’une variable locale est déclarée, elle n’est pas initialisée par le système, vous devez l’initialiser vous-même.

Lorsque vous les déclarez, les variables globales sont initialisées automatiquement par le système comme suit

|  |  |
| --- | --- |
| **Type de données** | **Valeur initiale par défaut** |
| int | 0 |
| char | '' |
| double | 0 |

Il est donc important de penser à initialiser les variables lorsque c’est nécessaire. Si une variable n’est pas initialisée, votre programme peut produire des résultats inattendus. En effet, les variables locales non initialisées prennent la valeur déjà présente à l’adresse mémoire qui leur est associée lors de la déclaration.