iut Brest
INSTITUT
UNIVERSITAIRE
DE TECHNOLOGIE
Morlaix

M105 : Programmation

IUT Brest Morlaix

2015

www.iut-brest.fr

Cours

Enseignants : Vincent Choqueuse

contact : vincent.choqueuse@univ-brest.fr

UBO
université de bretagne occidentale

Table des matières

Table des figures

Liste des programmes

Introduction

À qui s'adresse ce cours ?

Ce cours s'adresse à des débutants en programmation. Le background nécessaire pour l'appréhender se limite à des bases de mathématique et d'informatique.

Déroulement de l'enseignement

Ce cours est une initiation au langage de programmation C. Le langage C est avant tout un langage. Tout comme l'anglais, l'arabe, les hiéroglyphes, le langage C possède son propre vocabulaire et sa propre syntaxe. Contrairement aux langages précédents qui permettent aux hommes de communiquer entre eux, le langage C permet à l'homme de communiquer avec la machine (l'ordinateur, les smartphones, ...).

L'enseignement est composé de 20H de Cours-TD-TP. L'acquisition des connaissances sera évaluée au moyen de :

— Deux devoirs sur table.
— Quatres devoirs de Travaux Pratiques sous Code : :Block.

Pour en savoir plus

— OpenClassrooms [4] : Ce site propose des cours d'une excellente qualité pédagogique. Après chaque leçon, il est possible de tester ses connaissances en effectuant des quizz notés.
— C Programming Language [3] : Livre de référence en anglais sur la programmation en C écrit par les développeurs du langage.

Chapitre 1

Débuter en programmation

En l'espace d'une 20aine d'année, le champs d'application des technologies numériques s'est développé très rapidement. Par rapport à l'analogique, la technologie numérique possède de nombreux avantages (prix, facilité de copie de l'information, ...) qui ont permis son ascension rapide.

Le langage binaire (suite de 0 et de 1) est la base de la technologie numérique. Malgré sa simplicité apparente, ce langage est difficilement compréhensible par l'homme sous sa forme initiale. Pour dialoguer plus facilement avec les machines, les informaticiens ont mis en place des langages de programmation. Ces langages sont situés à mi-chemin entre le langage de la machine et le langage de l'homme supposé anglophone [1]. Chacun pouvant créer son propre langage, de nombreux langages de programmation ont vu le jour [6]. L'ensemble de ces langages peut se diviser en plusieurs catégories suivant le paradigme de programmation utilisé :

- les langages impératifs : C, Pascal, Fortran, Python
- les langages orientés objets : Java, C++, C#, Objective C, Python, Smalltalk
- les langages logiques : Prolog, ...
- ...

Le choix d'un langage particulier dépend de l'application visée. Dans le contexte des jeux vidéos, les langages C et C++ sont des références incontournables. Concernant les applications pour téléphones portables, l'Iphone d'Apple utilise un langage orienté objet dérivé du C (l'Objective C) alors que les téléphones sous Android utilise le langage orienté objet Java. Quant à lui, le monde d'internet est le berceau du PHP ou du Python du côté serveur, et du combo HTML5/ CSS3/ Javascript du côté client. Quel que soit le langage utilisé, tous reposent sur une méthodologie et une rigueur qu'il vous faudra acquérir pour être capable de vous adapter à n'importe quelle situation.

Dans ce cours d'initiation à la programmation, nous allons nous intéresser à un langage impératif : le langage C. Ce choix est motivé par plusieurs raisons. D'une part, les langages impératifs sont souvent plus simples à appréhender et de ce fait représentent un point de départ incontournable pour débuter en programmation. D'autre part, parmi les langages impératifs, le langage C reste un des langages les plus utilisés au monde malgré son âge (voir table 1.1). En effet, le C possède l'avantage d'être rapide, portable et d'être à la base d'autres langages plus évolués comme le C++, le C# et l'Objective C.

1. Pas de soucis ici si l'anglais n'est pas votre "cup of tea" ! Les langages de programmation ne comportent qu'une minuscule partie du dictionnaire anglais

Position	Langage	Type	Utilisation
1	Java	Objet	19.565%
2	C	Impératif	15.621%
3	C++	Objet	6.782%
4	C#	Objet	4.909%
5	Python	Impératif-Objet	3.664%

Tableau 1.1 – Popularité des langages de programmation (indicateur Septembre 2015 [5])

1.1 Environnement de développement

Pour réaliser un programme, il est nécessaire d'utiliser un environnement de développement. L'environnement de développement permet d'écrire des listes d'instructions, d'identifier d'éventuelles erreurs de programmation, de transcrire le programme en binaire, etc. Il existe différents environnements de développement permettant de programmer en langage C. Dans ce cours, nous allons utiliser Code : :Blocks [2]. Cet environnement possède l'avantage d'être gratuit et relativement complet. L'installation de Code : :Blocks s'obtient en suivant les étapes suivantes :

1. Aller à l'adresse suivante : `www.codeblocks.org/`.
2. Aller dans le menu vertical `Download`, puis aller dans la section `Download the binary release`.
3. Télécharger la dernière version de Code : :Blocks.
4. Lancer l'installation de Code : :Blocks.

Attention ! Assurez-vous de bien installer la version intégrant le compilateur GCC (fichier `codeblocks-13.12mingw-setup.exe`).

1.2 Mon premier programme

Dans cette section nous allons créer notre premier programme en langage C sous Code : :Blocks. Ce premier programme permettra de vérifier que l'installation de l'environnement s'est réalisée correctement. Les étapes suivantes décrivent la marche à suivre :

1. Lancer Code : :Blocks
2. Aller dans `create a new project`.
3. Choisir `console application` pour créer un projet en mode console.
4. Appuyer sur `next`, choisir le langage `C`, puis appuyer sur `next`.
5. Donner le titre `mon_premier_programme` à votre projet, puis appuyer sur `next`.
6. Appuyer sur `finish`.

Lorsque le projet est créé, Code : :Block stocke le code source du programme dans le fichier `main.c`. Pour y accéder rapidement, il suffit d'ouvrir le contenu du dossier `sources` situé à gauche de l'écran. Si tout se passe correctement, l'écran affiche une fenêtre similaire à celle de la figure 1.1. Le contenu du fichier `main.c` est présenté dans le programme 1.1.

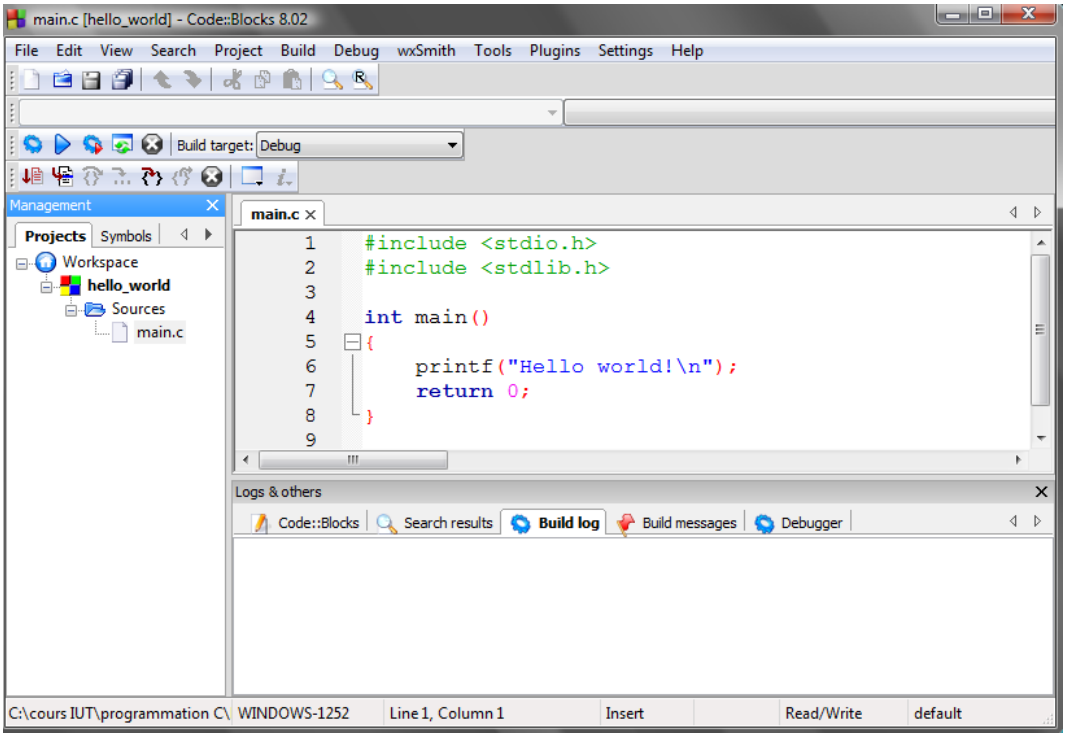

Figure 1.1 – Mon premier programme sous l'environnement Code : :Blocks

```
1: #include <stdio.h>
2: #include <stdlib.h>
3:
4: // debut du programme
5: int main()
6: {
7:     printf("Hello world!\n");
8:     return 0;
9: }
```

Programme 1.1 – Mon premier programme

Ce premier programme est écrit en langage C. Pour l'exécuter, il est nécessaire de convertir le fichier `main.c` en un fichier exécutable d'extension `.exe`. Cette conversion est réalisée en deux temps (voir figure 1.2) :

- une phase de compilation. La compilation vérifie la syntaxe du fichier et génère le code machine correspondant aux instructions. S'il n'y a pas d'erreur de syntaxe, la compilation génère un fichier objet d'extension `.o`.
- une phase d'édition des liens. L'édition des liens importe les librairies nécessaires au programme puis génère un fichier exécutable `.exe`.

Sous l'environnement Code : :Blocks, ces deux phases sont regroupées en une seule étape. Pour la lancer, il suffit d'aller dans le menu `Build` puis de sélectionner `build`. Ensuite pour

exécuter le programme, il suffit d'aller dans le menu `Build` puis de sélectionner `run`. Si tout s'est bien passé, l'exécution lance une console où est affichée le message `Hello world!`.

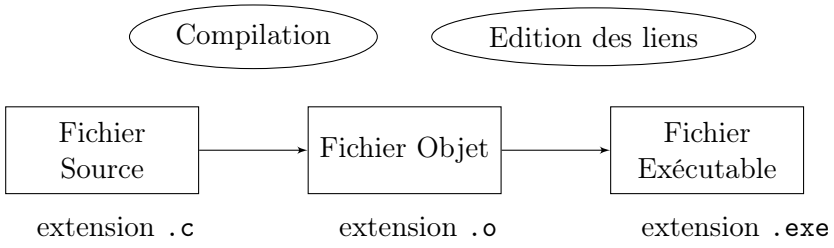

Figure 1.2 – Création d'un fichier exécutable

Attention ! Après chaque modification du code source (`.c`), il est nécessaire de relancer une compilation avant d'exécuter le programme sinon le fichier exécutable `.exe` ne sera pas mis à jour.

1.3 Un peu de vocabulaire

Avant de continuer, nous allons nous arrêter ici pour définir un peu de vocabulaire.

Définition. Le programme principal correspond à l'ensemble des lignes débutant par `int main()` et se terminant par `return 0;}`. Dans le programme 1.1, le programme principal débute en ligne 4 et se termine en ligne 8.

Définition. Une instruction correspond à une (ou plusieurs) ligne(s) de code. Une instruction correspond soit à une opération, soit à l'appel d'une fonction, soit à une structure de contrôle, etc. Par exemple, dans le programme 1.1, l'instruction à la ligne 6 permet d'appeler la fonction `printf( )`.

Attention ! À l'exception des structures de contrôle (voir chapitre 5), toutes les instructions se terminent par un **point virgule**.

Définition. Une fonction est un sous-programme appelé par le programme principal pour réaliser des tâches spécifiques et répétitives. Une fonction peut être programmée directement dans le même fichier que le programme principal (`main.c`) ou dans des librairies externes. Par exemple dans le programme 1.1, la fonction `printf( )` permet d'afficher à l'écran "Hello world !". Cette fonction est programmé dans la librairie externe `stdio` se trouvant le fichier `stdio.h`.

Définition. Une librairie (ou bibliothéque) est un recueil de sous-programmes (appelés fonctions). Pour utiliser une librairie, il est nécessaire de l'importer en début de programme. L'importation d'une librairie s'obtient en utilisant l'instruction `#include`. Dans le programme 1.1, la ligne 1 correspond à l'importation des fonctions de lecture au clavier et d'affichage à l'écran.

Définition. Un commentaire correspond à une information non interprétée par le compilateur. L'utilisation de commentaires permet d'apporter une meilleure lisibilité au programme pour le programmeur. Lorsque le commentaire tient sur une ligne, il est précédé par le délimiteur `//`. Si le commentaire tient sur plusieurs lignes, il sera délimité par le marqueur de début `/*` et de fin `*/`. Par exemple, dans le programme 1.1, l'instruction à la ligne 3 permet d'écrire un commentaire.

Attention ! Lors de l'écriture d'un programme, il ne faut pas négliger les commentaires. Ces commentaires sont d'une grande utilité lorsqu'il s'agit de travailler à plusieurs et/ou de reprendre d'anciens projets.

Définition. La compilation est l'étape qui permet la conversion d'un programme en langage C vers le langage binaire.

Définition. L'**exécution** est l'étape qui permet de lancer le programme à partir du programme compilé. Pour se lancer, ce programme devra être préalablement compilé. En fonction du type de programme créé, le programme peut se lancer dans une console de texte, dans une fenêtre windows etc.

Dans les chapitre suivants, nous allons détailler les différentes notions permettant de créer nos propres programmes.

Chapitre 2

Les variables

Dans ce chapitre, nous allons introduire la notion de variable. Les variables en programmation sont indispensables car elles permettent de stocker de l'information non-connue à l'avance (informations entrées par l'utilisateur lors de l'exécution du programme, résultats de calculs, etc). Par analogie avec les mathématiques, une variable correspond à une entité pouvant prendre différentes valeurs (la variable x dans une fonction $x \to f(x)$ par exemple). De manière plus formelle, en programmation nous adopterons la définition suivante :

Définition (variable)**.** Une variable est un symbole, le plus souvent un nom, qui renvoie à un emplacement en mémoire dont le contenu peut prendre successivement différentes valeurs.

Les valeurs que peut prendre une variable dépendent de son type (nombre entier, nombre réel, caractère, etc). Le langage C est un langage dit à typage statique c'est-à-dire que le type d'une variable doit être spécifié lors de sa création. Le typage statique présente plusieurs avantages : il permet notamment au compilateur d'optimiser certaines parties du code et de détecter d'éventuelles erreurs avant l'exécution du programme. Dans les sections suivantes, nous montrons comment déclarer (créer) et utiliser une variable.

Attention ! Une variable doit être préalablement déclarée avant d'être utilisée.

Attention ! La déclaration d'une variable doit être réalisée uniquement en début de programme (ou de fonction).

2.1 Déclaration simple

Lors de la déclaration d'une variable, un programme alloue automatiquement un espace en mémoire à une variable. La taille de cet espace dépend du type de la variable. Ainsi, un nombre comportant un grand nombre de chiffres après la virgule nécessite logiquement plus d'espace qu'un nombre entier comportant peu de décimales.

Syntaxe. La déclaration d'une variable s'obtient en utilisant la ligne de commande suivante :

```
1: type nom_var;          /* declaration */
```

où :

— `type` correspond au type de la variable. Le C intègre 3 types de base. Ces 3 types sont décrits dans le tableau 2.1. Il existe également d'autres types dérivés permettant de stocker des nombres plus ou moins grands, avec ou sans signe etc (voir tableau C.1 en annexe) .

type	description du type	exemples
int	Nombre entier	`1, -5, 19, 5787`
float	Nombre réel	`2.32, 3.14, -19.2, 543.23`
char	Caractère	`'5', 'o', 'k', '-', '\0'`

Tableau 2.1 – Les types de bases en langage C

— `nom_var` correspond au nom de la variable. Tous les noms sont admis sauf : les noms débutant par un chiffre, les noms comportant des espaces, accents et ou caractères spéciaux et les noms réservés du langage C (voir annexe B).

Attention ! Le langage C dissocie les minuscules et les majuscules. Ainsi, les noms `index` et `INDEX` ne désignent par la même variable.

Attention ! Il est fortement recommandé de donner des noms de variables explicites, c'est à dire directement en rapport avec leur contenu.

Attention ! Pour les variables de type `char`, le programme travaille en réalité sur des nombres entiers compris entre 0 et 255. Le tableau de correspondance ASCII [1] permet de faire le lien entre un caractère et un entier entre 0 et 255 .

Remarquons que lorsque plusieurs variables possèdent le même type, il est possible de les déclarer sur la même ligne. Le programme 2.1 illustre cette remarque.

```
1: int main(void)
2: {
3:     char consonne,voyelle;
4:     return 0;
5: }
```

Programme 2.1 – Déclaration de deux caractères

2.2 Déclaration avec initialisation

Dans certains cas, il est utile d'affecter à une variable une valeur par défaut juste après sa déclaration. En programmation, cette phase porte le nom d'initialisation. Par exemple, si nous voulons réaliser un programme pour compter le nombre de fois où l'utilisateur a appuyé sur

la touche *, la variable servant de compteur devra être initialisée à 0 en début de programme. Généralement, l'initialisation permet d'éviter de manipuler de l'information dont le contenu n'est pas maitrisé par l'utilisateur.

Syntaxe. La déclaration avec initialisation s'obtient en utilisant la ligne de commande suivante :

```
1: type nom_var=valeur;  // declaration et initialisation
```

où :

- `valeur` est une valeur fixe (pas de nom de variable)
- le signe = correspond à une opération d'affectation (voir chapitre 4)

Le programme 2.2 réalise simultanément une phase de déclaration et d'initialisation d'une variable réelle.

```
1: int main(void)
2: {
3:     float g=9.81;
4:     return 0;
5: }
```

Programme 2.2 – Déclaration et initialisation d'un réel

2.3 Utilisation

Après la phase de déclaration, le contenu d'une variable peut être modifié en utilisant des opérations d'affectation.

Syntaxe. Une affectation s'obtient en utilisant la ligne de commande suivante :

```
1: nom_var=valeur;
```

où :

- `nom_var` correspond au nom d'une variable préalablement déclarée.
- `valeur` est une valeur (ou une expression qui retourne une valeur) du même type que `nom_var`.

Attention ! Lors d'une affectation, le contenu anciennement stocké à l'emplacement de la variable `nom_var` est écrasé.

Attention ! Le signe = n'a pas la même signification en programmation et en mathématique. En programmation si x contient initialement la valeur 1, l'instruction $x = 2 * x + 1$ aura pour effet de remplacer la valeur de x par 3 (alors qu'en mathématique cela reviendrait à dire que x est égal à -1).

```
1:  #include <stdio.h>
2:  #include <stdlib.h>
3:
4:  int main(void)
5:  {
6:      int x=2;
7:      int y;
8:
9:      y=x;
10:     x=x+3;
11:     y=5;
12:     y=y+x;
13:
14:     printf("%d\n",x);
15:     printf("%d",y);
16:
17:     return 0;
18: }
```

Programme 2.3 – Affectation du contenu de deux variables

ligne	Contenu de `x`	Contenu de `y`
Déclaration	2	\
6	2	2
7	5	2
8	5	5
9	5	10

Tableau 2.2 – Evolution du contenu des variables `x` et `y` après l'exécution de chaque instruction du programme 2.3. Le caractère \ signifie que le contenu de la variable n'est pas maîtrisé

Lorsque l'on réalise une affectation, la valeur située à droite du signe égal est placée dans l'emplacement mémoire de la variable `nom_var`. Si une variable est présente des 2 côtés du signe =, sa valeur avant exécution de l'instruction est utilisée pour effectuer les opérations décrites à droite du signe =, puis sa valeur est finalement remplacée par le résultat.

Afin d'illustrer ce fonctionnement, le programme 2.3 réalise des opérations d'affectation sur deux variables `x` et `y`. Lors de la phase de déclaration (ligne 3 et 4), deux variables `x` et `y` sont déclarées en tant qu'entier. Le tableau 2.2 présente ensuite l'évolution du contenu de ces deux variables. À la fin du programme, `x=5` et `y=10`.

Chapitre 3

Les entrées-sorties

Le chapitre précédent nous a montré comment déclarer, initialiser et modifier des variables. Pour rendre un programme plus vivant, l'étape suivante consiste à interagir avec leur contenu au moyen de périphériques externes (écran, clavier). C'est le rôle de l'interface homme-machine (IHM). Plus précisément, L'IHM permet à l'utilisateur :

— de spécifier le contenu des variables à la machine (par exemple via un clavier ou une souris)

— d'afficher le contenu des variables (par exemple sur un écran)

Dans ce chapitre, nous allons nous limiter à la lecture d'informations au clavier et à l'affichage d'informations à l'écran en mode console. En langage C, la lecture et l'affichage sont réalisées en appelant des fonctions d'entrées/sorties. Bien que le chapitre sur les fonctions ne soit abordé que plus tard dans ce document (voir chapitre 7), nous présentons ici les notions de bases permettant l'appel des fonctions d'entrées-sorties.

3.1 Affichage à l'écran : `printf`

L'écran permet d'afficher des informations visuelles à l'utilisateur. Ces informations peuvent être de natures diverses : résultats, message à l'utilisateur, etc.

3.1.1 Affichage statique

Un affichage statique permet d'afficher une information fixe à l'écran. En langage C, l'affichage d'une information s'effectue en utilisant la fonction `printf`.

Syntaxe. La fonction `printf` nécessite d'inclure au préalable la librairie `<stdio.h>`. Cette fonction s'utilise de la manière suivante :

```
1: printf("message");       // affichage
```

où

— `message` correspond au texte à afficher à l'écran.

L'affichage de caractères spéciaux, tels que les retours à la ligne ou les tabulations, s'obtient en utilisant les commandes du tableau 3.1. Un exemple d'affichage fixe est donné par le programme 1.1. À la ligne 6, le programme permet d'afficher à l'écran le message `hello world !`.

Commande	Caractère	Exemple
\n	passage à la ligne	`printf("message\n");`
\t	tabulation horizontale	`printf("\t message");`

Tableau 3.1 – Utilisation de la fonction `printf`

3.1.2 Affichage de variable

En pratique, l'affichage d'une information fixe à l'écran présente peu d'intérêt. Le plus souvent, la fonction `printf` est associée à une variable pour afficher du contenu dynamique.

Syntaxe. La fonction `printf` nécessite d'inclure au préalable la librairie `<stdio.h>`. L'affichage du contenu d'une variable de type `int` s'obtient en utilisant la ligne suivante :

```
1: printf("%d",variable);
```

où

— `%d` signifie que la variable doit être affichée sous la forme d'un entier. Le tableau 3.2 indique la lettre à utiliser pour les différents formats d'affichage.

indicateur	Format	Exemples
%d	int	`printf("%d",variable);`
%f	float	`printf("%f",variable);`
%c	char	`printf("%c",variable);`
%s	char *(chaîne de caractères)	`printf("%s",variable);`

Tableau 3.2 – Indicateur du format pour l'affichage et l'écriture

Pour afficher le contenu de plusieurs variables, il suffit d'ajouter plusieurs pourcentages et de séparer les noms de variables par des virgules. Il est également possible d'ajouter de l'information statique comme le montre le programme 3.1.

```
1: #include <stdio.h>
2:
3: int main(void)
4: {
5:     int nombre1=2,nombre2=3;
6:     printf("Les nombres sont %d et %d",nombre1,nombre2);
7:     return 0;
8: }
```

Programme 3.1 – Affichage du contenu de deux variables

Attention ! Quel que soit le nombre de variables à afficher, la fonction `printf` doit comporter autant de caractères `%` que de virgules `,`.

3.2 Lecture au clavier

La lecture permet à l'utilisateur de saisir le contenu de variables au clavier. La fonction utilisée pour la lecture clavier dépend du type de la variable.

3.2.1 Lecture multitype : `scanf`

La fonction `scanf` permet de lire une (ou plusieurs) variable(s) au clavier, quelque soit son type. Les informations sont transmises au fur et à mesure dans une mémoire tampon jusqu'à ce que l'utilisateur appuie sur la touche entrée.

Syntaxe. La fonction `scanf` nécessite d'inclure au préalable la librairie `<stdio.h>`. Cette fonction s'appelle de la manière suivante :

```
1: scanf("%d",&variable);  // lecture d'un entier
```

où :

— `%d` signifie que la variable saisie doit être stockée dans un variable de type `int` (voir tableau 3.2).

Attention ! Il ne faut pas oublier le caractère `&`. En effet, la fonction `scanf` attend l'adresse d'une variable spécifiée au moyen d'un `&` (voir section **??**).

Il est possible de combiner la lecture de plusieurs variables en utilisant un seul `scanf`. Par exemple, le programme 3.2 montre comment lire 2 entiers au clavier puis les afficher à l'écran.

```
1:  #include <stdio.h>
2:
3:  int main(void)
4:  {
5:      int nombre1,nombre2;
6:      printf("Entrer 2 nombres entiers separes par un espace : ");
7:      scanf("%d %d",&nombre1,&nombre2);
8:      printf("Les nombres sont: %d et %d",nombre1,nombre2);
9:      return 0;
10: }
```

Programme 3.2 – Lecture des deux variables au clavier

3.2.2 Lecture d'un caractère : `getch`/`getche`

Les fonction `getch` permet la lecture d'une variable de type caractère.

Syntaxe. La fonction `getch` nécessite d'inclure au préalable la librairie non-standard `<conio.h>`. Cette fonction s'appelle de la manière suivante :

```
1: variable=getch();        // lecture d'un caractere
```

où :

— `variable` désigne un nom de variable de type `char`.

Par rapport à la fonction `scanf`, la fonction `getch` effectue la lecture sans attendre la touche entrée. Le programme 3.3 illustre une utilisation possible de cette fonction. À noter que la fonction `getch` n'affiche pas de caractères à l'écran. L'affichage de l'écho s'obtient en appelant la fonction `getche`.

```
1: #include <conio.h>
2:
3: int main(void)
4: {
5:     char lettre;
6:     lettre=getch();
7:     printf("Le caractere est: %c",lettre);
8:     return 0;
9: }
```

Programme 3.3 – Lecture d'un caractère avec getch

3.2.3 Lecture d'une chaîne de caractères : `gets`

Les chaînes de caractères sont des tableaux comportant plusieurs caractères à la suite. L'utilisation des tableaux et des chaînes de caractères sera présentée dans le chapitre 6.1. Pour lire une chaîne de caractères, il est conseillé d'utiliser la fonction `gets`. Par rapport au `scanf`, le `gets` permet de lire des caractères spéciaux comme les espaces, les accents, etc.

Syntaxe. La fonction `gets` nécessite d'inclure au préalable la librairie `<stdio.h>`. Cette fonction s'utilise de la manière suivante :

```
1: gets(variable);          // lecture d'une chaine
```

où :

— `variable` désigne le nom d'une chaine de caractères (voir chapitre 6.1).

3.3 Un exemple d'IHM

L'exemple 3.4 présente un programme permettant d'entrer deux nombres entiers et un nombre réel au clavier. Un message d'invite est affiché à l'écran avant chaque saisie au clavier.

```
1:  #include <stdio.h>
2:
3:  int main(void)
4:  {
5:      int nb1,nb2;
6:      float reel;
7:      printf("Bonjour\n");
8:      printf("Veuillez entrer deux nombres entiers :");
9:      scanf("%d %d",&nb1,&nb2);
10:     printf("et un reel");
11:     scanf("%f",&reel);
12:     printf("Les nombres sont %d %d et %f",nb1,nb2,reel);
13:     return 0;
14: }
```

Programme 3.4 – Exemple d'IHM

Chapitre 4

Les opérations de base

Ce chapitre présente les opérations de base du langage C. Une opération permet de manipuler le contenu d'une ou de plusieurs variables lors de l'exécution du programme (par exemple l'addition est une opération).

Syntaxe. Une opération s'exprime sous la forme générale suivante :

```
1: nom_var1 operation nom_var2;
```

où

- `nom_var1` et `nom_var2` désignent deux variables
- `operation` désigne, comme son nom l'indique, une opération.

L'ensemble des opérations disponibles en langage C peut se classer en 4 catégories : les opérations d'affectation (voir tableau 4.1), les opérations mathématiques (voir tableau 4.2), les opérations de comparaison (voir tableau 4.3) et les opérations binaires (voir tableau 4.4).

4.1 Les opérations d'affectation

Les opérations d'affectation permettent de stocker une valeur dans une variable. Lors d'une affectation, la valeur initialement stockée dans la variable est remplacée par sa nouvelle valeur. Les opérations d'affectation disponibles en C sont décrites dans la table 4.1. Le programme 4.1 présente un exemple d'affectation. À la fin du programme, les variables `nombre1` et de `nombre2` seront toutes les deux égales à 3. Le programme 4.2 illustre l'utilisation de l'opérateur incrémentation `++`. Lors de la déclaration, `nombre1` est initialisé à 2. Ensuite à la ligne 5, `nombre1` est incrémenté de 1 et sa valeur passe donc à 3.

4.2 Les opérations mathématiques

Ces opérations permettent de réaliser des calculs mathématiques basés sur le contenu des variables `nom_var1` et `nom_var2`. Les opérations mathématiques disponibles en langage C sont décrites dans la table 4.2. Il est recommandé de faire très attention avec la division pour deux raisons. D'une part, la division par zéro entraînera une erreur lors de l'exécution du programme. D'autre part, si `nombre1` et `nombre2` sont des entiers, le résultat de leur division sera par défaut de type entier. Les programmes 4.3 et 4.4 illustrent ce problème. Avant correction, le programme affiche un message d'avertissement à la compilation et affiche `resultat=0.000000`

```
1: #include <stdio.h>
2:
3: int main(void)
4: {
5:     int nombre1=2,nombre2=3;
6:     nombre1=nombre2;
7:     printf("nombre1=%d nombre2=%d",nombre1,nombre2);
8:     return 0;
9: }
```

Programme 4.1 – Opération d'affectation

```
1: #include <stdio.h>
2:
3: int main(void)
4: {
5:     int nombre1=2;
6:     nombre1++;
7:     printf("nombre1=%d",nombre1);
8:     return 0;
9: }
```

Programme 4.2 – Incrémentation d'un nombre

à l'exécution. Une manière de contourner le problème consiste forcer le type du résultat en `float` en multipliant le résultat de la division par le réel `1.0`. Après correction, le programme affiche bien le résultat attendu c-a-d `resultat=0.666667`.

```
1: #include <stdio.h>
2:
3: int main(void)
4: {
5:     int nombre1=2,nombre2=3;
6:     printf("resultat=%f",nombre1/nombre2);
7:     return 0;
8: }
```

Programme 4.3 – Division de deux entiers (avant correction)

4.3 Les opérations de comparaison

Ces opérations permettent de réaliser des tests basés sur la valeur de deux variables. Par convention, le résultat d'un test est égal à 0 si celui-ci est faux et est égal à 1 si celui-ci est vrai. Les opérations de comparaison seront majoritairement utilisées avec les instructions `if` et `while` (voir chapitre 5) pour aiguiller le programme en fonction du résultat d'un test.

```
1: #include <stdio.h>
2:
3: int main(void)
4: {
5:     int nombre1=2,nombre2=3;
6:     printf("resultat=%f",1.0*nombre1/nombre2);
7:     return 0;
8: }
```

Programme 4.4 – Division de deux entiers (après correction)

Les opérations de comparaison disponibles en langage C sont décrites dans la table 4.3. Le programme 4.5 présente un exemple de test d'égalité. À l'exécution, le programme affiche `resultat=0`.

```
1: #include <stdio.h>
2:
3: int main(void)
4: {
5:     int nombre1=2,nombre2=3;
6:     printf("resultat=%d",nombre1==nombre2);
7:     return 0;
8: }
```

Programme 4.5 – Comparaison de deux entiers

Attention ! Il ne faut surtout pas confondre l'opérateur `==`, qui effectue un test d'égalité, avec l'opérateur `=`, qui lui effectue une affectation.

4.4 Les opérations binaires

Ces opérations sont essentiellement utilisées pour lier le résultat de plusieurs comparaisons et pour réaliser des masques binaires. Les opérations binaires disponibles en langage C sont décrites dans la table 4.4. L'exemple 4.6 présente un programme réalisant un ET sur deux entiers (non signés). Les deux entiers sont tout d'abord convertis en binaire ($13 \rightarrow (0..01101)_2$ et $6 \rightarrow (0...00110)_2$, puis l'opération ET est appliquée. Le résultat est ensuite converti en décimal. Á l'exécution, le programme affiche `a ET b =4`. Le programme 4.7 montre comment réaliser un masque binaire sur la valeur entière $125 \rightarrow (0..01111101)_2$. La valeur du masque est spécifiée sous forme hexadécimale et est égale à $F0 \rightarrow (11110000)_2$. En utilisant l'opération `&`, ce masque permet de récupérer les 4 bits de poids fort de 125 c-a-d $(0...01110000)_2 \rightarrow 112$.

```
1: #include <stdio.h>
2:
3: int main(void)
4: {
5:     unsigned int a=13,b=6;
6:     printf("a ET b = %u\n",a&b);
7:     return 0;
8: }
```

Programme 4.6 – Opérations binaires

```
1: #include <stdio.h>
2:
3: int main(void)
4: {
5:     printf("%u",125&(0xF0));
6:     return 0;
7: }
```

Programme 4.7 – Masque binaire

opération	Description de l'opération
`=`	Le contenu de `nom_var2` est stocké dans `nom_var1`
`++`	Le contenu de `nom_var1` est incrémenté de 1 (`nom_var2` non nécessaire).
`--`	Le contenu de `nom_var1` est décrémenté de 1 (`nom_var2` non nécessaire).

Tableau 4.1 – Les opérations d'affectation

opération	Description de l'opération
`+`	Addition de deux variables
`-`	Soustraction de deux variables
`*`	Multiplication de deux variables
`/`	Division de deux variables (attention au type du résultat)
`%`	Reste de la division entière (modulo)

Tableau 4.2 – Les opérations mathématiques en langage C

opération	description de l'opération
`==`	test d'égalité
`!=`	test de différence
`>`	test de supériorité (stricte)
`>=`	test de supériorité
`<`	test d'infériorité (stricte)
`<=`	test d'infériorité

Tableau 4.3 – Les opérations de comparaison en langage C

opération	Description de l'opération
`&&`	opérateur ET logique
`||`	opérateur OU logique
`!`	opérateur NON
`&`	opérateur ET bit à bit
`|`	opérateur OU bit à bit
`^`	opérateur OU exclusif bit à bit
`<<`	`nom_var1` est décalé de `nom_var2` bits vers la gauche
`>>`	`nom_var1` est décalé de `nom_var2` bits vers la droite

Tableau 4.4 – Les opérations binaires en langage C

Chapitre 5

Les structures de contrôle

Jusqu'à présent, nos premiers programmes se lisaient de haut en bas. Dans ce chapitre, nous présentons des structures de contrôle qui permettent de modifier le sens de lecture d'un programme. Nous allons utiliser des organigrammes pour décrire les sens de lecture. Ces organigrammes sont composés de plusieurs éléments :

— un point de départ et d'arrivée : représentés par des ellipses.
— des séquences d'instructions : représentées par des blocs.
— des tests : représentés par des losanges.
— des retours en arrière : représentés par des flèches orientées vers le haut

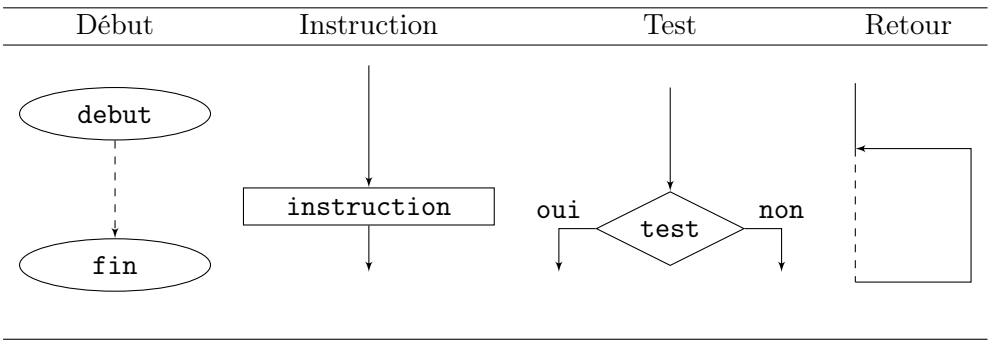

Tableau 5.1 – Eléments d'un organigramme

Le langage C intègre 4 structures de contrôle :

— deux branchements conditionnels : le `if` et le `switch`. Les branchements conditionnels aiguillent le programme vers un bloc d'instructions particulier en fonction d'une condition.
— deux boucles : le `for` et le `while`. Les boucles permettent de répéter un bloc d'instructions un certain nombre de fois.

D'apparence très simples, ces 4 structures de contrôle permettent de réaliser des programmes très sophistiqués lorsqu'elles sont utilisées conjointement. Toute la difficulté en programmation consistera à utiliser efficacement ces 4 structures de contrôle.

5.1 L'instruction `if`

Le branchement conditionnel `if` permet d'aiguiller le programme en fonction d'une condition binaire. Si la condition est différente de 0, une liste particulière d'instructions est réalisée. Dans le cas contraire, le programme est aiguillé vers une autre liste d'instructions.

Organigramme. L'organigramme de l'instruction `if` est donné par la figure suivante :

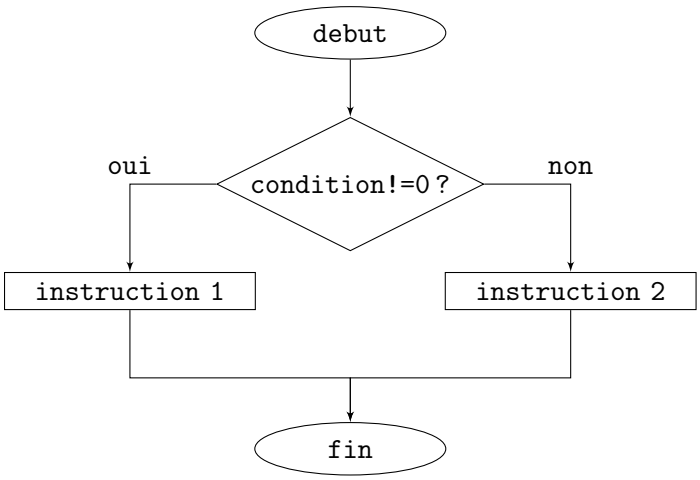

Figure 5.1 – L'instruction `if`

Syntaxe. Le test binaire s'utilise de la manière suivante :

```
1: if (condition)
2:      {
3:      /* liste d'instructions */
4:      }
5: else
6:      {
7:      /* liste d'instructions */
8:      }
```

où :

- `condition` correspond à une valeur binaire.
- le bloc d'instructions relatif au `if` est réalisé lorsque la valeur de `condition` est différente de 0.
- le bloc d'instructions relatif au `else` est réalisé lorsque le contenu de `condition` est égal à 0.

Le plus souvent, le contenu de `condition` s'obtiendra via des opérateurs de comparaison et/ou des opérateurs binaires (voir sections 4.3 et 4.4). À noter qu'il est possible d'omettre l'instruction `else` si le bloc d'instructions dans le `else` est vide.

Attention ! Bien qu'il soit possible d'omettre les accolades lorsque la liste d'instructions ne

comporte qu'une seule instruction, cette pratique reste fortement déconseillée.

Les programmes 5.1 et 5.2 présentent deux exemples d'utilisation de l'instruction `if`.

— Le premier programme affiche un message si l'utilisateur appuie sur la lettre `'a'`.

— Le second programme permet de tester la parité d'un nombre. Si le nombre est divisible par 2, un message est affiché à l'écran. Pour tester la parité, le programme évalue le reste de la division entière par 2 puis applique l'opérateur non (`!`).

```
1:  #include <stdio.h>
2:  #include <conio.h>
3:
4:  int main(void)
5:  {
6:      if (getch()=='a')
7:      {
8:          printf("Ceci est la premiere lettre de l'alphabet");
9:      }
10:     return 0;
11: }
```

Programme 5.1 – Recherche du caractère a

```
1:  #include <stdio.h>
2:
3:  int main(void)
4:  {
5:      int nombre;
6:
7:      printf("Veuillez entrer un nombre: ");
8:      scanf("%d",&nombre);
9:      printf("Le nombre %d est",nombre);
10:     if(!(nombre%2))
11:     {
12:         printf("pair");
13:     }
14:     else
15:     {
16:         printf("impair");
17:     }
18:     return 0;
19: }
```

Programme 5.2 – Test de parité avec l'instruction `if`

5.2 L'instruction `switch`

Le `switch` permet de réaliser plusieurs tests successifs sur la valeur d'une variable entière.

Organigramme. L'organigramme de l'instruction `switch` est donné par la figure suivante :

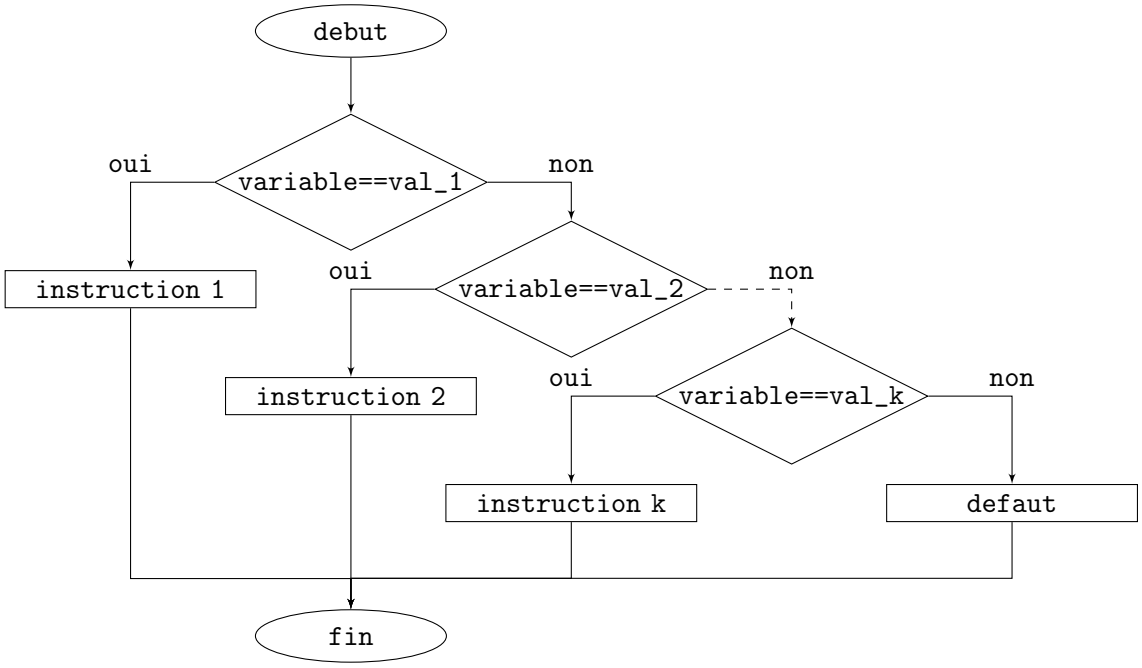

Figure 5.2 – L'instruction `switch`.

Le `switch` peut s'écrire sous la forme de plusieurs `if` imbriqués. Toutefois pour la réalisation de tests multiples, l'instruction `switch` sera préférée car son implémentation est optimisée.

Syntaxe. Le test multiple s'utilise de la manière suivante :

```
1:  switch (variable)
2:      {
3:      case val_1:  {
4:                   /*liste d'instructions*/
5:                   } break;
6:      case val_2:  {
7:                   /*liste d'instructions*/
8:                   } break;
9:      ...
10:     case val_k:  {
11:                  /*liste d'instructions*/
12:                  } break;
13:     default:     {
14:                  /*liste d'instructions*/
15:                  }
16:     }
```

où :

- `variable` correspond au nom de la variable testée.
- `val_1`, `val_2`,... `val_k` correspondent à des valeurs spécifiques de la variable testée.
- `default` correspond au bloc d'instructions réalisé lorsque le contenu de la variable ne correspond à aucune des valeurs listées.

Le programme 5.3 illustre une utilisation de l'instruction `switch`. Ce programme permet l'indexation d'une lettre de l'alphabet. Si la lettre correspond à une voyelle, le programme affiche `"voyelle"` suivi du nom de la voyelle. Si la lettre tapée est une consonne, le programme affiche `"la lettre est une consonne"`.

```
1:  #include <stdio.h>
2:  #include <conio.h>
3:
4:  int main(void)
5:  {
6:      char lettre;
7:      lettre=getche();
8:      switch (lettre)
9:      {
10:         case 'a':   {
11:             printf("voyelle a");
12:         } break;
13:         case 'e':   {
14:             printf("voyelle e");
15:         } break;
16:         case 'i':   {
17:             printf("voyelle i");
18:         } break;
19:         case 'o':   {
20:             printf("voyelle o");
21:         } break;
22:         case 'u':   {
23:             printf("voyelle u");
24:         } break;
25:         case 'y':   {
26:             printf("voyelle y");
27:         } break;
28:         default:    {
29:             printf("la lettre est une consonne");
30:         }
31:     }
32:     return 0;
33: }
```

Programme 5.3 – Indexation des lettres de l'alphabet

5.3 L'instruction `while`

Le `while` effectue en boucle une liste d'instructions tant qu'une condition est vérifiée. L'instruction `while` se décline sous deux formes.

Organigramme. Les organigrammes des deux déclinaisons du `while` sont donnés par les figures suivantes :

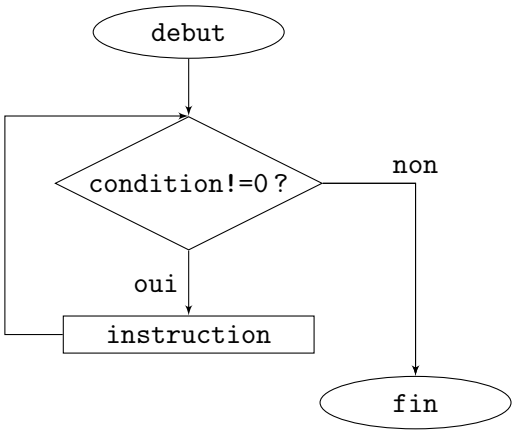

Figure 5.3 – L'instruction `while`

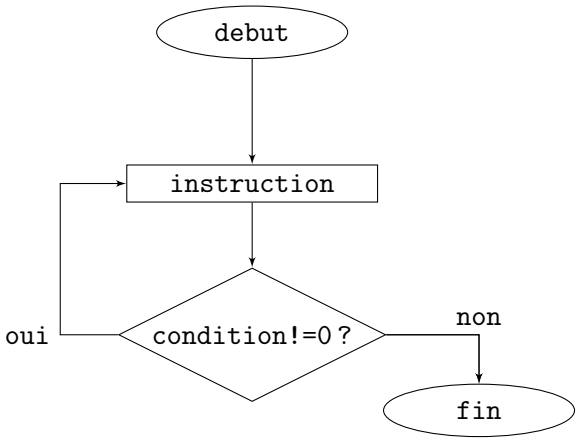

Figure 5.4 – L'instruction `do...while`

La différence majeure entre ces deux déclinaisons réside dans leur comportement lors de la première itération. La première version du `while` évalue la `condition` avant d'exécuter (ou pas) le bloc d'instructions entre accolades. À l'inverse, la seconde version exécute le bloc d'instructions avant d'évaluer la `condition`. Dans cette deuxième configuration, le bloc d'instructions sera réalisé au minimum une fois.

Syntaxe. L'instruction `while` s'utilise de la manière suivante :

```
1: while (condition)
2:     {
3:     /* liste d'instructions */
4:     }
```

où :

— `condition` est une valeur binaire.

Syntaxe. L'instruction `do...while` s'utilise de la manière suivante :

```
1: do
2:     {
3:     /* liste d'instructions */
4:     }
5: while(condition);
```

où :

— `condition` est une valeur binaire.

Attention ! Il faudra bien veiller à ce que la condition passe à 0 pour éviter les boucles

infinies. Cela implique nécessairement que le contenu de `condition` évolue dans la boucle.
Les programmes 5.4 et 5.5 présentent des exemples d'utilisation des instructions `while` et `do...while`.

— Le premier programme permet d'afficher la liste des puissances de x inférieures à 1000. Pour calculer chaque puissance, un nombre est tout d'abord initialisé à 1 puis multiplié par x à chaque itération. Le programme sort de la boucle lorsque le résultat de la multiplication est supérieur à 1000.

— Le second programme lit en boucle un caractère jusqu'à ce que l'utilisateur appuie sur la lettre `q`.

```
1:  #include <stdio.h>
2:
3:  int main(void)
4:  {
5:      int nombre;
6:      int puissance=1;
7:
8:      scanf("%d",&nombre);
9:      while(puissance<=1000)
10:     {
11:         printf("nombre=%d\n",puissance);
12:         puissance=puissance*nombre;
13:     }
14:     return 0;
15: }
```

Programme 5.4 – Affichage des puissances de x inférieures à 1000

```
1:  #include <stdio.h>
2:  #include <conio.h>
3:
4:  int main(void)
5:  {
6:      char lettre;
7:
8:      do
9:      {
10:         printf("\nAppuyer sur q pour quitter: ");
11:     }
12:     while(getch()!='q');
13:     return 0;
14: }
```

Programme 5.5 – Attente d'un caractère

5.4 L'instruction `for`

L'instruction `for` effectue un bloc d'instructions en boucle.

Organigramme. L'organigramme de l'instruction `for` est donné par la figure suivante :

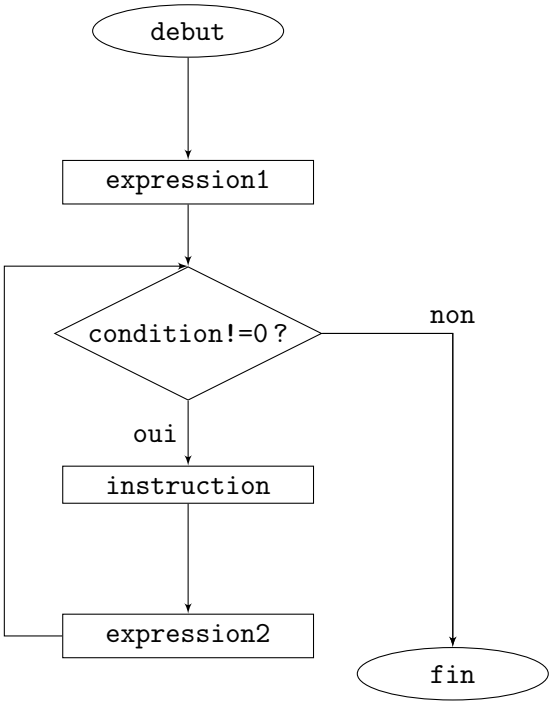

Figure 5.5 – La boucle `for`

Comme le montre l'organigramme 5.5, la boucle `for` peut s'exprimer sous la forme d'une boucle `while`. Toutefois pour les boucles définies (nombre d'itérations connu à l'avance), l'instruction `for` sera préférée car son implémentation est optimisée et son utilisation évitera bien des erreurs de programmation !

Syntaxe. La boucle `for` s'utilise de la manière suivante :

```
1: for (expression1;condition;expression2)
2:     {
3:     /* liste d'instructions */
4:     }
```

où :

- `expression1` est une instruction réalisée avant d'entrer dans la boucle.
- `condition` correspond à la condition nécessaire pour continuer à rester dans la boucle.
- `expression2` est une instruction réalisée après chacune des itérations de la boucle.

Le plus souvent, la boucle `for` sera utilisée pour réaliser un nombre N d'itérations. Dans cette configuration, `expression1` sert à initialiser une variable compteur (`indice=0`) , `expression2` sert à incrémenter le compteur (`indice++`) et `condition` permet de spécifier le nombre d'itérations de la boucle (`indice<10`) .

Les programmes 5.6 et 5.7 présentent deux exemples d'utilisation de l'instruction `for`.

— Le premier programme affiche l'ensemble des caractères ASCII. Un En langage C, un caractère est identifié par son code ASCII (une valeur entière comprise entre 0 et 255). Pour afficher l'ensemble des caractères ASCII, le programme 5.6 utilise une boucle `for` contenant 256 itérations. Cette boucle initialise une variable `indice` à 0 puis l'incrémente jusqu'à la valeur 255. À chaque itération, le caractère ASCII correspondant à `indice` est affiché au moyen de la fonction `printf`.

— Le second programme affiche le résultat de la somme arithmétique $S = 1 + \cdots + N$. Le programme demande initialement la valeur de N à l'utilisateur puis la somme est calculée par une boucle contenant N itérations [1].

```
1:  #include <stdio.h>
2:
3:  int main(void)
4:  {
5:      int indice;
6:
7:      for (indice=0;indice<=255;indice++)
8:      {
9:          printf("Valeur %d, ASCII %c\n",indice,indice);
10:     }
11:     return 0;
12: }
```

Programme 5.6 – Affichage des caractères ASCII

```
1:  #include <stdio.h>
2:
3:  int main(void)
4:  {
5:      int S=0;
6:      int indice,N;
7:      printf("Valeur de N ?");
8:      scanf("%d",&N);
9:      for(indice=1;indice<=N;indice++)
10:     {
11:         S=S+indice;
12:     }
13:     printf("S=%d",S);
14:     return 0;
15: }
```

Programme 5.7 – Calcul de la somme arithmétique

1. Remarquons que ce programme n'est pas très performant puisque mathématiquement le résultat de cette somme est connu et s'obtient directement via la formule $S = \frac{N(N+1)}{2}$.

Chapitre 6

Les variables composées

Dans le chapitre 2, nous avons appris à déclarer et à utiliser les 3 types de variables de bases : `int`, `float` et `char`. Dans certaines situations, le nombre de variables à déclarer peut être très grand et leur utilisation difficile à gérer. Imaginons le cas d'un enseignant souhaitant réaliser un programme pour entrer les notes de ses (super bons) élèves et calculer la moyenne de sa classe. Si seuls 3 élèves survivent à son cours et passent l'examen, l'enseignant pourra simplement déclarer dans son programme 3 variables de type `float` pour stocker la note de chaque étudiant. Maintenant si son cours, de par sa qualité, a attiré un nombre d'élèves important, l'enseignant devra déclarer un grand nombre de variables pour stocker les notes de l'ensemble de la classe. Bien que possible, la déclaration et l'utilisation de k variables (avec k élevé) sera difficile à mettre en place. C'est pour résoudre ce genre de problèmes que le C intègre des variables dites composées.

Nous pouvons distinguer deux types de variables composées :
— les tableaux, permettant de stocker plusieurs variables de même type.
— les structures, permettant de stocker plusieurs variables indépendamment de leur type.
Dans ce chapitre nous montrons comment déclarer, utiliser et initialiser ces variables composées.

6.1 Les tableaux

Un tableau dimensionnel permet de stocker plusieurs variables du même type. Il se présente sous la forme d'un vecteur (voir figure 6.1). L'emplacement de chaque élément est spécifié au moyen d'un index.

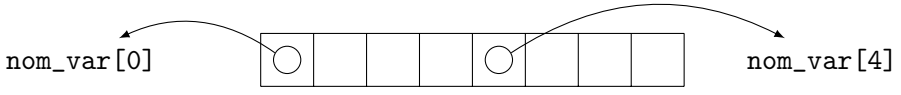

Figure 6.1 – Tableau à 1 dimension nommé `nom_var`

Attention ! En langage C, le premier élément d'un tableau est stocké à l'index 0. Le k^{e} élément est donc stocké à l'index $k-1$.

6.1.1 Déclaration

La déclaration d'un tableau est très similaire à une déclaration classique. La seule différence réside dans l'ajout d'un nombre entre crochets. Ce nombre spécifie la taille du tableau.

Syntaxe. La déclaration d'un tableau s'obtient en utilisant la ligne de commande suivante :

```
1: type nom_var[k];          /* declaration */
```

où :

- `type` correspond à un type de variable.
- `nom_var` correspond au nom du tableau.
- `k` correspond au nombre maximum d'éléments que peut contenir le tableau (par exemple le nombre de notes). Ce nombre est une valeur entière fixée à l'avance (pas de variable !)

Attention ! Lorsque le tableau contient des éléments de type `char`, le tableau est une chaîne de caractères. Le dernier élément d'une chaine de caractères sera systématiquement le caractère `\0`.

Pour déclarer un tableau, il est nécessaire de fixer au préalable sa taille k. La plupart du temps, la valeur k nous sera directement spécifiée. Si ce choix est libre, il faudra prendre en compte deux contraintes. En choisissant un k trop petit, le programme risque de ne pas disposer d'assez d'éléments. Au contraire, en choisissant un k trop grand, le programme monopolisera inutilement trop d'espace mémoire. Il faudra donc trouver un compromis entre ces deux contraintes.

6.1.2 Utilisation

Pour accéder à un élément du tableau, il faut spécifier son index. Par convention, le premier élément du tableau est stocké à l'index 0 et l'élément $k+1$ à l'index k [1].

Syntaxe. L'accès à l'élément `k+1` d'un tableau s'obtient en utilisant la ligne de commande suivante :

```
1: valeur=nom_var[k];      /* acces au (k+1)ieme element */
```

Pour affecter une valeur à un élément du tableau, il faut également spécifier son index entre crochets.

Syntaxe. L'affectation d'une valeur au `k+1`$^{\text{e}}$ élément du tableau s'obtient en utilisant la ligne de commande suivante :

```
1: nom_var[k]=valeur;      /* aff. du (k+1)ieme element */
```

Attention ! Pour un tableau de taille k, seuls les éléments indexés par un nombre appartenant à l'intervalle $[0, \texttt{k-1}]$ peuvent être accessibles ou modifiables.

1. Cette convention n'est pas adoptée par tous les langages. Par exemple sous Matlab©, le premier élément d'un tableau est stocké à l'index numéro 1.

6.1.3 Initialisation

L'initialisation d'un tableau est une phase importante à ne pas négliger. L'initialisation peut se faire de deux faÃ§ons : soit lors de la phase de déclaration, soit à la suite de la déclaration.

Initialisation à la déclaration

Pour initialiser un tableau lors de sa déclaration, il suffit de spécifier ses différents éléments entre accolades.

Syntaxe. Lors de la déclaration d'un tableau, l'initialisation des éléments s'obtient en utilisant la ligne de commande suivante :

```
1: type nom_var[l]={val_1,val_2,...,val_l};
```

où :

— `val_1`, `val_2`, ..., `val_l` sont des valeurs du même type que le tableau. Lorsque $l < k$, les $k - l$ derniers éléments sont initialisés à 0.

```
1: #include <stdio.h>
2:
3: int main(void)
4: {
5:     int tableau[10]={0};
6:     return 0;
7: }
```

Programme 6.1 – Initialisation d'un tableau à 0 (lors de la déclaration)

Initialisation après la déclaration

Après la déclaration, il n'existe pas de syntaxe équivalente permettant l'initialisation d'un tableau. La seule possibilité consiste à affecter les différentes éléments un à un. Le programme 6.2 illustre cette technique.

6.1.4 Exemple

Revenons ici au problème de notre enseignant. Notre enseignant désire créer un programme pour stocker les notes de ses étudiants à un examen. Imaginons que sa classe comporte 20 élèves. Bien que possible, il serait trop long et surtout totalement inefficace de déclarer 20 variables de type `float` pour stocker les notes. À la place, nous allons déclarer un tableau de 20 `float`. Le programme 6.3 illustre cette solution. Remarquons qu'à ce stade, le programme n'a aucun intérêt pratique car à la fermeture du programme... l'ensemble des notes sera supprimé !

```
1:  #include <stdio.h>
2:
3:  int main(void)
4:  {
5:      int indice,tableau[10];
6:      for (indice=0;indice<10;indice++)
7:      {
8:          tableau[indice]=0;
9:      }
10:     return 0;
11: }
```

Programme 6.2 – Initialisation d'un tableau à 0 (post-déclaration)

```
1:  #include <stdio.h>
2:
3:  int main(void)
4:  {
5:      float tableau[10]={0};
6:      int indice;
7:
8:      for (indice=0;indice<10;indice++)
9:      {
10:         printf("Veuillez entre la %d ieme note: ",indice+1);
11:         scanf("%f",&tableau[indice]);
12:     }
13:     return 0;
14: }
```

Programme 6.3 – Mémorisation de notes dans un tableau

6.1.5 Les tableaux multidimensionnels

Dans certaines situations, l'utilisation de tableaux multidimensionnels devient inévitable. Prenons par exemple un jeu de dames. Pour stocker le contenu des cases du damier il sera nettement plus simple de manipuler un tableau à 2 dimensions de taille 8×8 plutôt que 8 tableaux de taille 8 à une dimension. Alors que les tableaux à une dimension se présentent sous la forme de vecteurs, les tableaux multidimensionnels se présentent sous la forme de matrices à plusieurs dimensions (voir figures 6.2 et 6.3) [2].

2. Un tableau à 1 dimension peut se représenter par un vecteur. Un tableau à 2 dimensions peut se représenter par une matrice. Plus difficile, à la dimension 3, un tableau peut se représenter par un hyperrectangle. Les tableaux de dimensions supérieures sont beaucoup plus difficiles à concevoir pour notre cerveau.

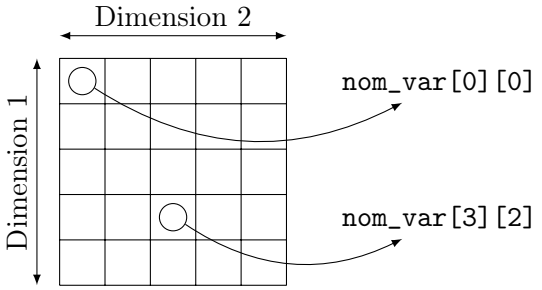

Figure 6.2 – Tableau à 2 dimensions

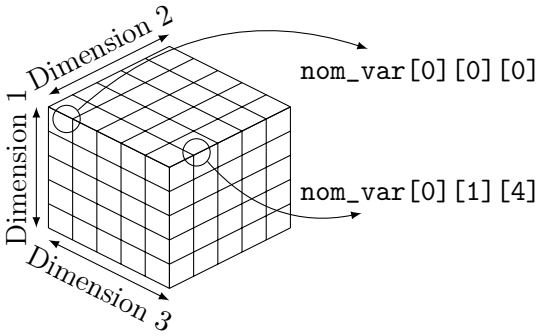

Figure 6.3 – Tableau à 3 dimensions

La déclaration d'un tableau multidimensionnel est très similaire à la déclaration d'un tableau mono-dimensionnel. La seule différence réside dans l'ajout de plusieurs nombres entre crochets. Ces nombres spécifient la taille des différentes dimensions.

Syntaxe. La déclaration d'un tableau multidimensionnel s'obtient en utilisant la ligne de commande suivante :

```
1: type nom_var[k_1][k_2][...][k_n];          /* declaration */
```

où :

— `k_1`,`k_2`,...,`k_n` correspondent respectivement à la taille des dimensions 1,2 et n.

Concernant l'utilisation des tableaux multidimensionnels, la syntaxe est quasi-identique au cas mono-dimensionnel. L'emplacement d'un élément particulier est spécifié en précisant entre crochets son index pour chaque dimension (voir figures 6.2 et 6.3).

6.2 Les structures

Les variables stockées dans un tableau possèdent la contrainte d'être de même type. Dans certaines situations, le stockage de variables de types différents au sein d'une même variable permet de faciliter considérablement la mise en place d'un programme. Ces variables composées sont appelées structures. Pour illustrer leur intérêt, revenons au cas de notre enseignant. Afin de rendre le programme plus convivial, l'enseignant souhaite renseigner pour chaque élève plusieurs informations : le nom, le prénom, la note au premier examen et la note au deuxième examen. Pour réaliser ce programme, une solution élégante et pratique consiste à créer une structure contenant plusieurs champs : deux chaînes de caractères (le nom et le prénom) et deux réels (les deux notes). Ainsi, l'ensemble des informations relatives à un élève sera stocké dans une même et unique variable contenant plusieurs champs. L'enseignant pourra ensuite créer un tableau de structures pour stocker les informations relatives à la classe (voir figure 6.4).

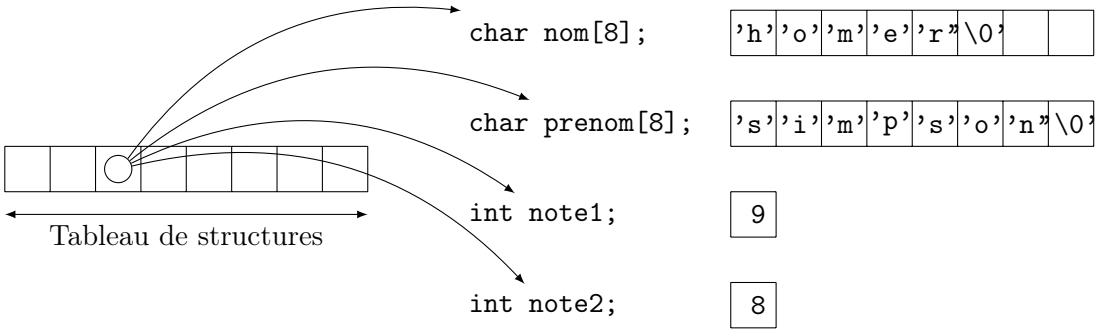

Figure 6.4 – Tableau de structures permettant de stocker les informations de plusieurs élèves

6.2.1 Définition

La définition d'une structure permet de lister l'ensemble de ses champs. Le plus souvent la définition est accompagnée d'une définition de type. La définition de type s'obtient au moyen de la commande `typedef`.

Syntaxe. La définition d'une structure s'obtient dans ce cas en utilisant la ligne :

```
1: typedef struct {
2:         type_1 champ_1;
3:         type_2 champ_2;
4:         ...
5:         type_k champ_k;
6: }nom_structure;
```

où :

- `champ_1`, ...,`champ_k` correspondent aux noms des différents champs
- `type_1`, ...,`type_k` correspondent aux types des différents champs. Le type d'un champ peut être soit un `int`, un `float`, un `char`, un tableau ou une autre structure définie.

Attention ! La définition d'une structure se place après les `#include` et avant le `main`.

6.2.2 Déclaration

Lorsqu'une structure est définie à l'aide de `typedef`, la déclaration d'une variable structurée s'obtient de la même faÃ§on que pour les variables de type `int`, `float` ou `char`.

Syntaxe. La déclaration d'une variable de type `nom_structure` s'obtient en utilisant la ligne de commande :

```
1: nom_structure nom_var;
```

Attention ! Il ne faut pas confondre `nom_structure` et `nom_var`. En effet, `nom_structure` est un type, tout comme les types `int`, `float` et `char`, alors que `nom_var` correspond au nom d'une variable.

6.2.3 Utilisation

L'accès à un champ particulier s'obtient en ajoutant à la suite de la variable le caractère `.` précédé du nom du champ.

Syntaxe. L'accès au champ `champ_1` d'une variable `nom_var` de type `nom_structure` s'obtient en utilisant la ligne de commande :

```
1: valeur=nom_var.champ_1;
```

Pour affecter une valeur, la syntaxe est similaire.

Syntaxe. L'affectation d'une valeur au champ `champ_1` d'une variable `nom_var` s'obtient en utilisant la ligne de commande :

```
1: nom_var.champ_1=valeur;
```

6.2.4 Initialisation

L'initialisation d'une structure s'obtient de la même faÃ§on qu'une initialisation de tableau. Cette initialisation peut se faire, soit lors de la déclaration, soit à la suite du programme.

Syntaxe. Lors de la déclaration d'une variable structurée, l'initialisation des valeurs s'obtient en utilisant la ligne de commande suivante :

```
1: nom_structure nom_var={val_1,val_2,...,val_k};
```

où

— les valeurs `val_1`, `val_2`, `val_k` sont stockées respectivement dans le 1^{er}, 2^{e} et k^{e} champ de la structure.

Après la déclaration, il n'existe pas de syntaxe équivalente permettant l'initialisation de la structure. La seule solution consiste à affecter les champs un à un.

6.2.5 Exemple

Le programme 6.4 illustre une utilisation possible des structures. Ce programme permet à un enseignant de rentrer les noms, prénoms et notes d'une classe composée de 5 étudiants. Lors de l'exécution, le programme demande à l'enseignant d'entrer l'ensemble des informations, puis le programme affiche les données à l'écran. Les informations relatives à un élève sont stockées dans une structure `eleve` composée de 4 champs : `nom`, `prenom`, `note1` et `note2`. Un tableau de type `eleve` est utilisé pour stocker les informations des 5 étudiants.

```
1:  #include <stdio.h>
2:
3:  typedef struct{
4:      char nom[20];
5:      char prenom[20];
6:      float note1;
7:      float note2;
8:  } eleve;
9:
10: int main(void)
11: {
12:     eleve liste[5];
13:     int indice;
14:
15:     for(indice=0;indice<5;indice++)
16:     {
17:         printf("Nom de l'eleve ?");
18:         gets(liste[indice].nom);
19:         printf("Prenom de l'eleve ?");
20:         gets(liste[indice].prenom);
21:         printf("Premiere note ?");
22:         scanf("%f",&liste[indice].note1);
23:         printf("Seconde note ?");
24:         scanf("%f",&liste[indice].note2);
25:         //pour purger le retour a la ligne
26:         scanf("%*[^\n]");
27:     }
28:
29:     for(indice=0;indice<5;indice++)
30:     {
31:         printf("%s %s:",liste[indice].nom,liste[indice].prenom);
32:         printf("%f %f\n",liste[indice].note1,liste[indice].note2);
33:     }
34:
35:     return 0;
36: }
```

Programme 6.4 – Stockage des informations relatives à une classe

Chapitre 7

Les fonctions

Lors de la réalisation de certains programmes, il n'est pas rare de retrouver plusieurs fois le même enchainement d'instructions. Pour éviter de surcharger inutilement le programme, le langage C permet la création de sous-programmes, nommés fonctions, recevant des entrées et renvoyant des sorties. Ces sous-programmes sont appelés par le programme principal (ou par un autre sous-programme) pour sous-traiter certaines tâches. Par exemple, les fonctions `printf` et `scanf` permettent respectivement de sous-traiter la gestion de l'affichage et de la lecture clavier. Précédemment, nous avons appris comment appeler ces sous-programmes. Dans ce chapitre, nous allons aller plus loin en apprenant comment développer et appeler nos propres fonctions.

7.1 Corps de la fonction

Syntaxe. Le corps d'une fonction est donné par la syntaxe suivante :

```
1: type_sortie nom_fonction(type_1 entree_1, type_2 entree_2,...)
2: {
3: /* declaration des variables de la fonction */
4:
5: /* liste d'instructions */
6:
7: return (variable);
8: }
```

où :

- la première ligne correspond au prototype de la fonction.
  - `nom_fonction` correspond au nom de la fonction.
  - `type_k` correspond au type de la k^{e} entrée (`int`, `float`, `char`,...).
  - `entree_k` correspond au nom de la variable où la k^{e} entrée va être copiée.
  - `type_sortie` correspond au type de la valeur passée en sortie (`int`, `float`, `char`,...). Si la fonction ne renvoie pas de valeur en sortie, `type_sortie` est fixé à `void`.
- la variable à renvoyer en sortie est spécifiée via l'instruction `return`. Cette variable doit être du même type que `type_sortie`. Si aucune variable n'est renvoyée en sortie, l'instruction `return` n'est pas nécessaire.

Attention ! Un prototype mal choisi entraînera nécessairement des complications dans la réalisation d'un programme.

Attention ! Seul le contenu de la variable passée au `return` est renvoyé en sortie de fonction. En particulier les variables déclarées à l'intérieur d'une fonction sont supprimées une fois la fonction terminée.

Le corps d'une fonction peut se placer à deux endroits dans le programme :
— Soit après l'importation des librairies et avant le main (voir programme 7.1).
— Soit après le main. Dans ce cas, le prototype de la fonction, suivi d'un point virgule, doit être ajouté avant le main. Cela permet au compilateur de vérifier l'intégrité des différents appels (voir programme 7.2).

```
1:  #include ...
2:
3:  type fonction(...)
4:  {
5:  ...
6:  return();
7:  }
8:
9:
10: int main(void) {
11: ...
12: return 0;
13: }
```

Programme 7.1 – Fonction placée entre les librairies et le main

```
1:  #include ...
2:  type fonction(...);
3:
4:  int main(void) {
5:  ...
6:  return 0;
7:  }
8:
9:  type fonction(...)
10: {
11: ...
12: return();
13: }
```

Programme 7.2 – Fonction placée après le main

7.1.1 Transmission des entrées

Il existe deux mécanismes de transmission des entrées
— le passage par valeur des entrées. Ces fonctions travaillent sur une recopie des valeurs passées en entrée. En utilisant une recopie, les modifications des entrées opérées à l'intérieur de la fonction ne sont pas repercutées à l'exterieur de la fonction.

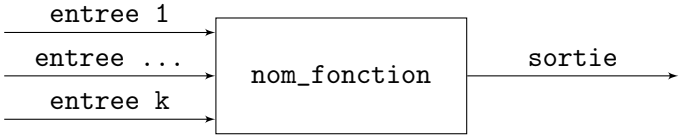

Figure 7.1 – Schéma bloc d'une fonction à k entrées et 1 sortie.

— le passage par adresse. Ces fonctions travaillent directement à partir des adresses physiques des variables passées en entrées. Les modifications des entrées opérées à l'intérieur de la fonction sont également répercutées à l'exterieur de la fonction.

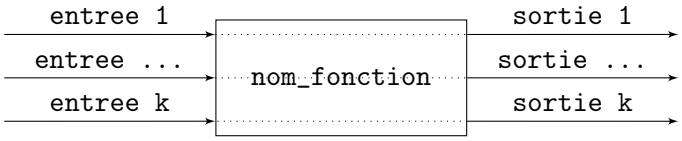

Figure 7.2 – Schéma bloc d'une fonction à k entrées et k sorties.

Pour des fonctions simples, un passage par valeur sera la plupart du temps utilisé. Pour des fonctions plus compliquées (envoie d'un tableau en entrée, renvoie de plusieurs variable), un passage par adresse sera inévitable. Le passage par adresse repose sur deux opérateurs :

— l'opérateur adresse `&`. Suivi d'un nom de variable, cet opérateur permet d'extraire l'adresse physique d'une variable (exemple : `adresse=&variable;`).
— l'opérateur contenu `*`. Suivi d'un nom de variable, cet opérateur permet d'extraire le contenu d'une adresse (exemple : `variable=*adresse;`).

Par défaut, une fonction utilise un passage par valeur des entrées. Pour signaler à la fonction que nous souhaitons passer des entrées par adresse, il faut ajouter devant les noms des entrées concernées le caractère $*$ dans le prototype de la fonction. Par exemple, une entrée désignée dans le protoype par `int nb1` utilisera un passage par valeur, alors que l'entrée désignée par `int *nb2` utilisera un passage par adresse.

7.2 Appel de la fonction

Une fonction est lancée via un mécanisme d'appel à partir du programme principal (ou d'une autre fonction).

Syntaxe. L'appel d'une fonction s'obtient via l'instruction

```
1: sortie=nom_fonction(var_1,var_2,...);
```

où :

— `var_k` correspond à la k^e valeur passée en entrée.
— `sortie` correspond à la variable stockant la sortie de la fonction. Si la fonction ne retourne pas de valeur, l'appel est réalisé via la syntaxe `nom_fonction(var_1,var_2,...);`.

Attention ! Le type des entrées et de la sortie doit correspondre aux types définis par le prototype de la fonction. Ainsi `var_k` a pour type `type_k` et `sortie` à pour type `type_sortie`.

Attention ! Si la fonction retourne une valeur, il ne faut surtout pas oublier d'affecter la sortie à une variable (`sortie=...`) sinon la fonction est appelée pour rien !

Attention ! Si une entrée utilise un passage par adresse, il faut transmettre son adresse (`&var_1`) et non son contenu (`var_1`).

7.2.1 Exemples

Le programme 7.3 contient une fonction permettant de convertir des degrés en radians. Cette fonction utilise un passage par valeur de l'angle. Le programme 7.4 réalise une permutation de deux nombres. Cette fonction utilise un passage par adresse des entrées.

```
1:  #include <stdio.h>
2:
3:  float deg_2_rad(float deg)
4:  {
5:      float rad;
6:      rad=deg*3.14/180;
7:      return(rad);
8:  }
9:
10: int main(void)
11: {
12:     float v_deg,v_rad;
13:     printf("Veuillez entrer un angle en degree:");
14:     scanf("%f",&v_deg);
15:     v_rad=deg_2_rad(v_deg);
16:     printf("%f deg -> %f rad",v_deg,v_rad);
17:     return 0;
18: }
```

Programme 7.3 – Conversion d'angles

```
1:  #include <stdio.h>
2:
3:  void permutation(int *a,int *b)
4:  {
5:      int temp;
6:      temp=*a;
7:      *a=*b;
8:      *b=temp;
9:  }
10:
11: int main(void)
12: {
13:     int nb1,nb2;
14:     printf("Veuillez entrer deux entiers:");
15:     scanf("%d %d",&nb1,&nb2);
16:     permutation(&nb1,&nb2);
17:     printf("permutation -> %d %d",nb1,nb2);
18:     return 0;
19: }
```

Programme 7.4 – Permutation de deux nombres

Annexe A

Les librairies standards du C

En début de programme, il est possible d'importer des librairies. Ces librairies comportent des fonctions dont le but est de vous faciliter la vie. L'importation des librairies standards s'obtient en utilisant la ligne `#include`. Une partie des librairies standards du langage C est donnée dans le tableau A.1.

librairies	fonctionnalités
`stdlib.h`	Conversion, génération de nombres pseudo-aléatoires, ...
`stdio.h`	écriture au clavier et affichage à l'écran,...
`string.h`	manipulation des chaînes de caractères.
`time.h`	conversion entre différents formats de date et d'heure,...
`math.h`	Calcul des fonctions mathématiques courantes,...
`complex.h`	Manipulation des nombres complexes,...
`ctype.h`	classification des caractères, conversion entre majuscules et minuscules,...

Tableau A.1 – Libairies standards en langage C.

La liste des fonctions disponibles dans chacune de ces librairies est accessible à l'adresse http://www.utas.edu.au/infosys/info/documentation/C/CStdLib.html.

Annexe B

Les mots réservés du C

Le langage C comporte plusieurs mots réservés. Ces mots ont une signification particulière et ne pourront pas être utilisés comme nom de variable ou de fonction. La liste des mots réservés est donnée dans le tableau B.1.

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

Tableau B.1 – Les mots réservés du langage C

Annexe C

Les différents types de variables

type	description	Octets	valeurs
`char`	caractère	1	-128 à 127
`short`	entier court	2	-32 768 à 32 767
`int`	entier	4	-2 147 483 648 à 2 147 483 648
`long`	entier long	4	-2 147 483 648 à 2 147 483 648
`unsigned char`	caractère non signé	1	0 à 255
`unsigned short`	entier court non signé	2	0 à 65 535
`unsigned int`	entier non signé	4	0 à 4 294 967 295
`unsigned long`	entier long non signé	4	0 à 4 294 967 295
`float`	simple précision	4	$1.2e^{-38}$ à $3.4e^{-38}$
`double`	double précision	8	$2.2e^{-38}$ à $1.8e^{-38}$

Tableau C.1 – Les types de variables en langage C

Bibliographie

[1] ASCIItable. Tableau de correspondance ascii. http://www.asciitable.com/.

[2] Site du logiciel Code : :Blocks. Code : :blocks. http://www.codeblocks.org/.

[3] Brian W. Kernighan and Dennis M. Ritchie. C Programming Language (2nd Edition). Prentice Hall, 1988.

[4] OpenClassrooms. Apprenez a programmer en c. https://openclassrooms.com/courses/apprenez-a-programmer-en-c.

[5] TIOBE. Programming community index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[6] Wikipedia. Chronologie des langages de programmation. http://fr.wikipedia.org/wiki/Chronologie_des_langages_de_programmation.