INITIATION AU LANGAGE C

Table des matières

1 Introduction **5**
1.1 Algorithmique et Programmation . . . 5
1.2 Présentation du langage . . . 6
1.2.1 Un peu d'histoire . . . 6
1.2.2 Représentation de l'information en mémoire centrale . . . 6
1.2.3 Structure d'un programme . . . 6

2 Variables, Types de base et Opérateurs **9**
2.1 Les types de base . . . 9
2.1.1 Nombres entiers . . . 9
2.1.2 Nombres flottants . . . 9
2.1.3 Les caractères . . . 10
2.2 Les variables . . . 10
2.2.1 Déclaration . . . 10
2.2.2 Identificateur . . . 10
2.2.3 Initialisation . . . 11
2.3 Les Opérateurs . . . 11
2.3.1 L'affectation . . . 11
2.3.2 Les opérateurs arithmétiques . . . 11
2.3.3 Incrémentation avec ++. . . . 12
2.3.4 Décrémentation avec −−. . . . 12
2.3.5 Les affectations généralisées . . . 12
2.3.6 Les opérateurs logiques . . . 13
2.3.7 Les opérateurs de comparaison . . . 13
2.3.8 L'opérateur & . . . 13
2.3.9 L'opérateur sizeof . . . 13
2.3.10 Les parenthèses . . . 13
2.4 Priorité des opérateurs . . . 14
2.5 Définition constructive des expressions . . . 14

3 Les Entrées-Sorties **15**
3.1 printf . . . 15
3.2 scanf . . . 15

4 Les Instructions **17**
4.1 Les instructions simples . . . 17
4.2 Les instructions conditionnelles . . . 18
4.2.1 L'instruction-if . . . 18
4.2.2 L'instruction-switch . . . 18
4.2.3 L'instruction-ou-case . . . 18

4.3 Les boucles . . . 19
4.3.1 L'instruction-while . . . 19
4.3.2 L'instruction-dowhile . . . 19
4.3.3 L'instruction-for . . . 20
4.3.4 Les instructions *break* et *continue* . . . 20

5 Fonctions **21**
5.1 Analyse descendante . . . 21
5.2 Définition de fonction . . . 21
5.3 Appel de fonction . . . 22
5.4 Déclaration de fonction . . . 22
5.5 Les variables : 2ème partie . . . 22
5.5.1 Visibilité des variables . . . 23
5.5.2 Passage de paramètres . . . 23

6 Les Tableaux **25**
6.1 Les tableaux à une dimension . . . 25
6.2 Les tableaux multidimensionnels . . . 26

7 Les Pointeurs **27**
7.1 Définition . . . 27
7.2 Déclaration et Initialisation . . . 27
7.3 Les Opérateurs & et * . . . 28
7.4 Arithmétique des adresses . . . 28
7.5 Passage de paramètres et Pointeurs . . . 29
7.6 Tableaux et Pointeurs . . . 29
7.6.1 Tableaux multidimensionnels et Pointeurs . . . 29
7.7 Conversion de type : opérateur de "cast" . . . 30

8 Chaînes de caractères **31**
8.1 Les variables chaînes de caractères . . . 31
8.2 Manipulation . . . 31

9 Gestion dynamique de la mémoire **33**
9.1 Principales fonctions de gestion de la mémoire . . . 33
9.1.1 Allocation . . . 33
9.1.2 Libération . . . 34
9.1.3 Les tableaux dynamiques . . . 34
9.2 Organisation du programme en mémoire . . . 36

10 Les structures **37**
10.1 Typedef : définition de nouveaux types . . . 38

11 Quelques éléments supplémentaires **39**
11.1 Variables, fonctions et compilation séparée . . . 39
11.1.1 Identificateurs publics et privés . . . 39
11.1.2 Déclaration d'objets externes . . . 39
11.1.3 Variables locales statiques . . . 39
11.1.4 Variables critiques . . . 40
11.1.5 Variables constantes et volatiles . . . 40

Chapitre 1

Introduction

1.1 Algorithmique et Programmation

La programmation informatique est l'ensemble des activités qui permettent l'écriture des programmes informatiques. L'algorithmique désigne l'ensemble des activités logiques qui relèvent des algorithmes. Ce mot vient du nom d'un mathématicien perse Abou Jafar Muhammad Ibn Al-Khuwarizm qui au neuvième siècle, écrivit le premier ouvrage systématique sur la résolution des équations linéaires et quadratiques. Un algorithme est une séquence d'actions (plus ou moins simples) qui permet d'aller d'un état initial à un état final, le but. Un algorithme décrit une démarche sous la forme d'une suite d'opérations, pour résoudre un problème donné. L'écriture de cette démarche dans un langage de programmation constitue la brique élémentaire de la programmation informatique. Les termes implémentation et codage sont souvent utilisés par les informaticiens pour y faire référence. Il existe des milliers de langages de programmation informatique : Pascal, Java, Fortran, C, C++, etc.
Ce sont des langages formels avec une syntaxe stricte qui permet d'éviter les ambiguïtés contrairement au langage naturel. Ils peuvent être plus ou moins évolués. Le langage machine est le plus basique. Le code d'un algorithme en langage machine n'est pas très lisible (compréhensible), il traduit les opérations à effectuer au niveau de l'ordinateur. C'est une suite de 0 et 1 associée aux instructions élémentaires exécutables par le microprocesseur. Le défaut majeur de ce type de langage pour un informaticien, est sa non lisibilité qui nuit nécessairement à la compréhension et la modification du code. Heureusement, il existe des langages plus évolués tels que l'Assembleur qui est un peu plus lisible et assez souvent utilisé pour coder les instructions élémentaires pour le microprocesseur. D'autres langages de niveau encore plus élevé sont largement utilisés. Ils donnent un niveau de compréhension très élevé à toute personne maîtrisant leur syntaxe. Cependant, ces langages évolués ne sont pas adaptés pour une exécution directe du microprocesseur des instructions élémentaires sous-jacentes. Une traduction en langage machine est donc nécessaire. Cette opération dans le cas du langage C, se fait en deux étapes : la compilation et l'édition de liens. La compilation traduit le code source (écrit en C) en langage machine dans un fichier dit objet. Ce fichier peut contenir des trous qui sont des appels à d'autres programmes déjà existants. L'éditeur de liens associe le fichier objet de notre programme avec celui des programmes appelés pour construire un unique fichier exécutable.

L'objet de ce cours est de s'initier à la syntaxe du langage C qui est un langage de programmation impérative : les instructions sont exécutées pour transformer l'état actuel du programme.

1.2 Présentation du langage

1.2.1 Un peu d'histoire

Le langage C a été mis au point par Ritchie et Kernighan au début des années 70. Leur but était de développer un langage qui permettrait d'obtenir un système d'exploitation de type UNIX portable.

Les opérations possibles sont limitées : les assignations, les branchements conditionnels, les branchements inconditionnels, les bouclages.

La règle veut que tout apprentissage d'un langage informatique commence par le codage de l'algorithme suivant :

Algorithme 1 : Affiche_bonjour()

```
1 Entrées :
2 Sorties :
3 Afficher "Bonjour !" à l'écran.
```

Algorithme 2 : Programme C : Afficher bonjour

```
1 #include < stdio.h >
2 int main()
3 {
4 printf("Bonjour ! \n") ;
5 return 0 ;
6 }
```

1.2.2 Représentation de l'information en mémoire centrale

La mémoire centrale peut être vue comme un tableau à une dimension dont le contenu des cases (appelées bits) est une des deux valeurs 0 ou 1. Dans ce tableau, un indice (numéro) est attribué à chaque paquet de 8 bits contigus (appelé octet). Cet indice représente l'adresse de cet octet dans la mémoire et permet de le repérer. De ce fait, une donnée stockée en mémoire est représentée sur au moins un octet.

Cependant, le nombre de données différentes qu'il est possible de représenter en binaire (suite de 0 et 1) sur un octet est limité à $2^8 = 256$. Aussi, une information peut être représentée sur plusieurs octets.

Donc, pour repérer une information en mémoire centrale, il faut connaître l'adresse de son premier octet et sa taille (i.e. le nombre d'octets sur lequel elle est représentée). La taille d'une information est donnée par son type (sa nature). Ce dernier permet également de décoder correctement sa représentation binaire en mémoire. En effet, une même suite de 0 et 1 sera interprétée différemment selon qu'elle représente un entier naturel, un entier relatif, un caractère...

1.2.3 Structure d'un programme

Dans le cas général, un programme C se présente sous la forme de plusieurs fichiers sources (avec l'extension .c) et fichiers d'en-tête (avec l'extension .h). L'intérêt de ce découpage du code est tout d'abord de regrouper des parties du programme en fonction de leur finalité permettant un maintien plus aisé. Mais plus encore, cela permet la compilation séparée des différentes composantes et donc de ne pas reprendre totalement cette phase après chaque modification. Par

Algorithme 3 : Programme C : Structure d'un programme

```
1 # directives adressées au préprocesseur
2 déclarations (des objets qui seront manipulés)
3 int main()
4 {
5 déclarations (des objets qui seront manipulés)
6 instructions (les opérations à exécuter)
7 }
```

soucis de simplicité, nous nous plaçons ici dans le cas simple où tout le programme se trouve dans un unique fichier source.

- Le fichier source contient des directives au préprocesseur (un programme qui procède à la transformation du code avant la compilation) : souvent l'objectif est d'inclure le contenu d'autres fichiers dans le fichier courant.
- Le fichier source contient des déclarations et des définitions de fonctions qui sont des sous-programmes accomplissant une tâche déterminée qui rentre dans la résolution globale du problème posé. En outre, le programme doit contenir une et une seule fonction main : c'est le point d'entrée du programme.
- Le fichier source contient des déclarations des objets qui sont manipulés. Ces derniers, de même que les données constantes, peuvent être de différents types : entiers, flottants, caractères, chaînes de caractères...
- Les opérations arithmétiques usuelles sont définies : +, *, -, /...
- Ils existent des mots réservés qui ne peuvent être utilisés pour nommer nos objets ou fonctions : if, else, while, for, do, char, short, int, long, float, double, signed, unsigned, switch, case, break, goto, default, continue, void, return, sizeof, struct, typedef, const, static, union, enum, extern, auto, register, volatile.
- Les blancs (espaces, tabulations, fins de lignes) sont ignorés.
- Les commentaires /* Ceci est un commentaire */ sont également ignorés.

Le langage C n'étant pas directement exécutable pour le processeur, une phase de traduction en langage machine, la compilation, est nécessaire. La compilation du fichier source génère un nouveau fichier dit objet (avec l'extension .o). L'édition de liens rajoute le code objet des sous-programmes pré-définis utilisés dans notre programme et crée un fichier exécutable autonome (avec l'extension .exe). Les sous-programmes pré-définis sont des fonctions dont le code est déjà rédigé et stocké dans des bibliothèques dans le but de faciliter la création d'un programme C. Par exemple, l'affichage d'une chaîne de caractères à l'écran peut se faire de manière très simple en utilisant une fonction déjà existante, printf qui se trouve dans la bibliothèque stdio (standard input/output). Pour ce faire, il est nécessaire de rédiger la directive au préprocesseur #include < stdio.h > qui permet d'inclure le fichier stdio.h contenant les en-têtes des fonctions de cette bibliothèque.

Chapitre 2

Variables, Types de base et Opérateurs

2.1 Les types de base

En programmation, les nombres et les textes sont les données fondamentales ; le langage propose comme types de base les entiers, les réels et les caractères.

2.1.1 Nombres entiers

- Petite taille : **(unsigned) char**
  Les entiers représentés sur 1 octet (8 bits) : de -128 à 127 (de 0 à 255 pour les unsigned).
- Taille moyenne : **(unsigned) short**
  Les entiers représentés sur 2 octets (16 bits) : de -32.768 à 32.767 (de 0 à 65.535 pour les unsigned)
- Grande taille : **(unsigned) long**
  Les entiers représentés sur 4 octets (32 bits) : de -2.147.483.648 à 2.147.483.647 (de 0 à 4.294.967.296 pour les unsigned)
- Très grande taille (Iso C99) : **(unsigned) long long**
  Les entiers représentés sur 8 octets (64 bits) : de -9.223.372.036.854.775 808 à 9.223.372.036.854.775.807 (de 0 à 18.446.744.073.709.551.615 pour les unsigned)

Le type **int** est le plus adapté parmi les types précédents sur la machine utilisée.
Ces différents types laissent la possibilité au programmeur de choisir le plus approprié sachant que la solution simpliste consistant à définir uniquement des objets de grande taille peut créer des problèmes de surcharge de la mémoire.

Une constante littérale entière a pour type le plus petit suffisant pour représenter sa valeur parmi les types int, long et unsigned long.

2.1.2 Nombres flottants

Un nombre flottant (123.456E-78) est composé :
- d'une partie entière (123)
- d'un point qui fait office de virgule
- d'une partie fractionnaire (456)
- d'une des deux lettres E ou e
- d'un signe éventuel + ou -
- d'un exposant (78)

On peut omettre :

– la partie entière ou la partie fractionnaire, mais pas les deux
– le point ou l'exposant, mais pas les deux

A l'image des entiers, il existe plusieurs types de flottants avec des précisions et des tailles différentes.

– Simple précision : **float**
  Les flottants représentés sur 4 octets (32 bits) : de -1.70E38 à -0.29E-38 et de 0.29E-38 à 1.70E38.
– Grande précision : **double**
  Les flottants représentés sur 8 octets (64 bits) : de -0.90E308 à -0.56E-308 et de 0.56E-308 à 0.90E308.
– Très grande précision : **long double**
  Les flottants de très grande précision sur 10 octets (80 bits) : de -3.4E4932 à -3.4E-4932 et de 3.4E-4932 à 3.4E4932.
  Il faut noter qu'ils sont souvent stockés sur 12 octets pour maintenir la structure d'alignement des données.

Une constante flottante est considérée par défaut de type double.

2.1.3 Les caractères

Il n'existe pas de type dédié à la représentation des caractères. Mais le codage des caractères dans l'ordinateur par des entiers (leur code ASCII), fait que le type **char** peut être utilisé pour représenter tous les caractères présents sur le clavier, chiffres compris. Les entiers de ce type sont suffisants pour représenter tous les caractères possibles : unsigned char (de 0 à 255).

Conséquence directe : 'A' étant à la fois un caractère et un entier, l'opération 'A'+1 est tout à fait valide en C.

2.2 Les variables

On appelle variable, un emplacement mémoire dans lequel est codé une information que l'on peut modifier et utiliser grâce à un identificateur. Les variables sont les objets manipulés dans le programme. Avant l'utilisation d'une variable, il faut donner ses spécifications d'abord. Cette étape est appelée déclaration.

2.2.1 Déclaration

Syntaxe : $<$ **type de base** $><$ **identificateur** $<=$ **initialisation** $-$ **opt** $>>, ...;$

Opération : la déclaration précise donc les caractéristiques d'une variable. Le type permet de réserver un emplacement mémoire de taille suffisante pour stocker sa valeur, et par la suite de connaître la taille de cet emplacement pour pouvoir modifier et extraire cette valeur.

Toute variable doit être déclarée avant toute utilisation.

2.2.2 Identificateur

L'identificateur est le nom qu'on attribue à une variable. C'est une suite contiguë composée de lettres et de chiffres et du caractère _ (blanc souligné) et qui commence obligatoirement par une lettre. Les mots réservés du langage ne peuvent être des identificateurs. Par ailleurs, les majuscules et les minuscules ne sont pas équivalentes.
L'identificateur constitue un lien symbolique avec l'adresse de la variable (l'adresse du premier octet de l'emplacement mémoire où est stockée sa valeur). De ce fait, il permet de manipuler la variable : trouver son adresse dans la mémoire, trouver sa valeur, modifier cette dernière.

2.2.3 Initialisation

L'initialisation est l'attribution d'une valeur à la variable dès sa création.

Algorithme 4 : Programme C : Initialisation

```
1 int main()
2 {
3 int somme = 0, moyenne, pgcd, i ;
4 }
```

2.3 Les Opérateurs

A ce stade de nos connaissances, on peut définir une expression comme une entité syntaxiquement correcte qui a une valeur d'un des types de base. Nous aurons une définition plus précise avant la fin du chapitre.

2.3.1 L'affectation

Syntaxe : < **variable** >=< **expression** >

Opération : La valeur de l'expression est évaluée et devient la nouvelle valeur de la variable (enregistrée à l'emplacement de la variable).

La valeur de l'expression est convertie au type de la variable si cela a un sens.

Algorithme 5 : Programme C : Affectation

```
1 int main()
2 {
3 int var = 0 ;
4 var = var + 10 * (1 + 2 + 3 + 4 + 5) ;
5 return 0 ;
6 }
```

2.3.2 Les opérateurs arithmétiques

- Addition : +
- Soustraction : −
- Multiplication : ∗
- Division : /
- Modulo (reste division entière : les deux arguments sont des entiers) : %

Syntaxe : < **expression$_1$** > **OP** < **expression$_2$** >

Le type du résultat est le type nécessitant le plus grand nombre de bits parmi celles des deux expressions concernées.

Exemple 2.3.1 *La somme d'un int et d'un double est de type double.*
*La division de deux entiers est un entier : par conséquence, c'est une division entière (*1/2 *a pour résultat* 0*).*

2.3.3 Incrémentation avec ++.

2.3.3.1 Post-incrémentation.

Syntaxe : < **variable** > ++

Opération : < *variable* > ++ a le même type que < *variable* > (entier, flottant ou pointeur), la même valeur que < *variable* > avant l'évaluation de < *variable* > ++, un effet de bord équivalent à celui de < *variable* >=< *variable* > +1.

2.3.3.2 Pré-incrémentation.

Syntaxe : ++ < **variable** >

Opération : ++ < *variable* > a le même type que < *variable* > (entier, flottant ou pointeur), la même valeur que < *variable* > après l'évaluation de ++ < *variable* >, un effet de bord équivalent à celui de < *variable* >=< *variable* > +1.

Algorithme 6 : Programme C : Incrémentation

```
1 # include < stdio.h >
2 int main()
3 {
4 int i = 1, j, k ;
5 j = i + + ; /* équivaut à j = i ; i = i + 1 ; */
6 i = 1 ;
7 k = + + i ; /* équivaut à i = i + 1 ; k = i ; */
8 return 0 ;
9 }
```

2.3.4 Décrémentation avec −−.

2.3.4.1 Post-décrémentation.

Syntaxe : < **variable** > −−

Opération : < *variable* > −− a le même type que < *variable* > (entier, flottant ou pointeur), la même valeur que < *variable* > avant l'évaluation de < *variable* > −−, un effet de bord équivalent à celui de < *variable* >=< *variable* > −1.

2.3.4.2 Pré-décrémentation.

Syntaxe : −− < **variable** >

Opération : −− < *variable* > a le même type que < *variable* > (entier, flottant ou pointeur), la même valeur que < *variable* > après l'évaluation de −− < *variable* >, un effet de bord équivalent à celui de < *variable* >=< *variable* > −1.

2.3.5 Les affectations généralisées

Ce sont des combinaisons entre l'opérateur d'affectation et les opérateurs arithmétiques : + =, − =, ∗ =, / =, % =.

Syntaxe : < **variable** > **OP** =< **expression** >

Opération : la valeur de < *variable* > *OP* < *expression* > est évaluée et affectée à < *variable* >. C'est équivalent à < *variable* >=< *variable* > *OP* < *expression* >.

2.3.6 Les opérateurs logiques

- **Conjonction :** &&
  - **Syntaxe** : < **expression$_1$** > && < **expression$_2$** >
  - **Opération :** < $expression_1$ > && < $expression_2$ > a pour valeur 0 si la valeur d'une des deux expressions est nulle et est différente de 0 sinon.

- **Disjonction :** ||
  - **Syntaxe** : < **expression$_1$** > || < **expression$_2$** >
  - **Opération :** < $expression_1$ > || < $expression_2$ > a pour valeur 0 si les valeurs des deux expressions sont nulles et est différente de 0 sinon.

- **Négation :** !
  - **Syntaxe** : ! < **expression** >
  - **Opération :** ! < *expression* > a pour valeur 1 si la valeur de l'expression est nulle et 0 sinon.

2.3.7 Les opérateurs de comparaison

- égalité : ==
- Supériorité stricte : >
- Supériorité non stricte : >=
- Infériorité stricte : <
- Infériorité non stricte : <=
- Différence : ! =

Syntaxe : < **expression$_1$** > **OP** < **expression$_2$** >

< $expression_1$ > et < $expression_2$ > doivent être de type simple (entiers, flottants ou pointeurs).

2.3.8 L'opérateur &

Syntaxe : & < **variable** >

Opération : l'opérateur & permet de connaître l'adresse (l'emplacement mémoire) d'une variable.

2.3.9 L'opérateur sizeof

Syntaxe : **sizeof**(< **variable** >)

Syntaxe : **sizeof**(< **type** >)

Opération : l'opérateur **sizeof** renvoie un entier (en fait un **size_t** défini dans **stddef.h**) qui indique la taille en octets de la variable (ou d'une variable du < *type* > donné).

2.3.10 Les parenthèses

Syntaxe : < **expression$_0$** > (< **expression$_1$** >, ..., < **expression$_n$** >)

Opération : Les parenthèses permettent l'application d'une fonction à une liste de valeurs appelées arguments. Si < $expression_0$ > est une fonction qui renvoie une valeur d'un type T : < $expression_0$ > (< $expression_1$ >, ..., < $expression_n$ >) est de type T.

Exemple 2.3.2 $y = sqrt(2 * x) + 3$ *; /* sqrt renvoie un double */*

2.4 Priorité des opérateurs

15	( )	[ ]	.	$->$		
14	!	++	−−	$-_{un}$	$*_{un}$	&	sizeof()
13	*	/	%			
12	+	−				
10	<	<=	>	>=		
9	==	!=				
5	&&					
4	||					
2	=	*=	/=	%=	+=	−=
1	,					

Le tableau ci-dessus donne la priorité d'évaluation des opérateurs. Ainsi, les affectations avec une priorité de 2 sont évaluées bien après les opérateurs arithmétiques tels que $*$ et $+$. L'utilisation des parenthèses (priorité maximale) permet de sortir des ambiguïtés.

2.5 Définition constructive des expressions

La définition d'expression est récursive :
une constante littérale est une expression (1, 2.34E-56, 'A', "bonjour") ;
une variable est une expression ;
une expression correcte formée par l'application d'un opérateur à une (pour les opérateurs unaires) ou deux (pour les opérateurs binaires) expressions respectant les pré-conditions (le format) de l'opérateur est une expression (1+(-1.56), 'A'*3, "bonjour"+10, x=y+360*.56).

La définition d'expression constante est également récursive :
une constante littérale est une expression constante (1, 2.34e-56, 'A', "bonjour") ;
une expression correcte formée par l'application d'un opérateur à une (pour les opérateurs unaires) ou deux (pour les opérateurs binaires) expressions constantes est une expression constante (1+(-1.56), 'A'*3, "bonjour"+10).

Chapitre 3

Les Entrées-Sorties

Ce sont des fonctions (des sous-programmes) définies dans une bibliothèque du langage (stdio : Standard Input/Output Library) qui permettent de communiquer avec un programme à travers, entre autres, l'écran et le clavier. Avant toute utilisation de ces fonctions, il faut inclure dans le fichier courant la bibliothèque stdio à travers la directive au pré-processeur : # include < stdio.h >.

3.1 printf

Syntaxe : **printf**("**texte**")

Opération : affichage de *texte* à l'écran sans aucune modification.

Syntaxe : **printf**("$\mathbf{texte_1}\%\mathbf{x_1}\mathbf{texte_2}\%\mathbf{x_2}$...", < **expression$_1$** >, < **expression$_2$** >, ...)

Opération : affichage à l'écran de $texte_1$ suivi de la valeur de $< expression_1 >$ dont le type est donné par le caractère x_1, puis $texte_2$ suivi de la valeur de $< expression_2 >$ dont le type est donné par le caractère x_2...

Le type de l'expression à afficher	le spécificateur
int (décimale)	%d ou %i
int (octale)	%o
int (hexadécimale)	%x
unsigned int	%u
short	%hd ou %ho ou %hx ou %hu
long	%ld ou %lo ou %lx ou %lu
long long	%lld ou %llo ou %llx ou %llu
char (décimale)	%d ou %o ou %x ou %u
char (caractère)	%c
float ou double (décimale)	%f
long double (décimale)	%Lf
float ou double (exponentielle)	%E ou %e
long double (exponentielle)	%LE ou %Le
chaîne de caractères	%s

Algorithme 7 : Programme C : printf

```
1  #include < stdio.h >
2  int main()
3  {
4  int var = 2, i ;
5  double j = 3. ;
6  printf("Exemple \n") ;
7  printf("La valeur de l'entier var est : %d \n", var) ;
8  printf("La valeur de l'entier i est %d et celle du flottant j est %f \n", i, j) ;
9  return 0 ;
10 }
```

3.2 scanf

Syntaxe : **scanf**("%$\mathbf{x_1}$%$\mathbf{x_2}$...", & < **variable$_1$** >, & < **variable$_2$** >)

Opération : permet la saisie d'une valeur dont le type est donné par le caractère x_1, puis l'affecte à la variable $< variable_1 >$ qui est de même type et la saisie d'une nouvelle valeur dont le type est donné par le caractère x_2, puis l'affecte à la variable $< variable_2 >$ qui est de même type.

Le type de l'expression à saisir	le spécificateur
int (décimale)	%d
int (décimale, octale ou hexadécimale)	%i
int (octale)	%o
int (hexadécimale)	%x
unsigned int	%u
short	%hd ou %ho ou %hx ou %hu
long	%ld ou %lo ou %lx ou %lu
long long	%lld ou %llo ou %llx ou %llu
char (décimale)	%d ou %o ou %x ou %u
char (caractères)	%c
float (décimale)	%f
double (décimale)	%lf
long double (décimale)	%Lf
float (exponentielle)	%e
double (exponentielle)	%le
long double (exponentielle)	%Le
chaîne de caractère	%s

Algorithme 8 : Programme C : scanf

```
1  #include < stdio.h >
2  int main()
3  {
4  int var_1, i ;
5  char var_2 ;
6  double j ;
7  scanf("%d", &var_1) ;
8  scanf("%c", &var_2) ;
9  scanf("%d %lf", &i, &j) ;
10 return 0 ;
11 }
```

Chapitre 4

Les Instructions

4.1 Les instructions simples

Une instruction a une définition récursive.

- instruction vide :
  Syntaxe : ;
  Opération : ne fait rien.
- instruction-expression : **Syntaxe** : < **expression** >;
  Opération : < *expression* > est évaluée.
- instruction-bloc :
  Syntaxe :
  {
  declarations
  instructions
  }
  Opération : les déclarations entraînent la création de variables, puis la série d'instructions est exécutée et pour finir les variables déclarées sont détruites.
  Un bloc est une suite de déclarations et d'instructions encadrée par { et }. Il se comporte comme une unique instruction et peut ainsi figurer à tout endroit où une instruction simple est permise.

Algorithme 9 : Programme C : Instructions simples

```
1  #include < stdio.h >
2  int main()
3  {
4  int var = 0 ;
5  var = var + 10 * (1 + 2 + 3 + 4 + 5) ;
6  {
7  float var ;
8  var = 1.23 ;
9  printf("%f \n",var) ;
10 }
11 printf("%d \n",var) ;
12 return 0 ;
13 }
```

4.2 Les instructions conditionnelles

4.2.1 L'instruction-if

Syntaxe : **if**(< **expression** >) < **instruction$_1$** > **else** < **instruction$_2$** >

Opération : si < *expression* > est vraie, alors < $instruction_1$ > est exécutée, sinon < $instruction_2$ > est exécutée.

Syntaxe : **if**(< **expression** >) < **instruction** >

Opération : si < *expression* > est vraie, alors < *instruction* > est exécutée, sinon on ne fait rien.

A la place de < *instruction* >, < $instruction_1$ >, < $instruction_2$ >, on peut avoir un bloc.

Algorithme 10 : Programme C : Instruction if

```
1  #include < stdio.h >
2  int main()
3  {
4  int i, j ;
5  scanf("%d %d", &i, &j) ;
6  if(i < j) printf("%d est plus petit que %d \n", i, j) ;
7  else
8  {
9  if(i > j) printf("%d est plus grand que %d \n", i, j) ;
10 else printf("%d est egal a %d \n", i, j) ;
11 }
12 return 0 ;
13 }
```

4.2.2 L'instruction-switch

Syntaxe : **switch**(< **expression** >){< **instructions − ou − case** >}

Opération : < *expression* > est évaluée, puis :
s'il existe un *case* tel que la valeur de < *expression-constante* > est égale à la valeur de < *expression* > alors < *instruction-opt* > est exécutée, de même que toutes les instructions suivantes jusqu'à la fin du bloc ou la rencontre d'une instruction break qui nous fait sortir du bloc.
S'il n'existe pas un tel *case*, mais qu'il existe un $default$, alors l'instruction à la suite du $default$ est exécutée.
S'il n'existe pas de $default$, on sort du bloc.

4.2.3 L'instruction-ou-case

Syntaxe :
case < **expression − constante** > : < **instruction − opt** >
default : < **instruction** >

Algorithme 11 : Programme C : Instruction switch

```
1  #include < stdio.h >
2  int main()
3  {
4  int i;
5  scanf("%d", &i);
6  switch(i)
7  {
8  case 1 : printf("i a pour valeur 1 \n");
9  break;
10 case 2 : printf("i a pour valeur 2 \n");
11 break;
12 case 3 : printf("i a pour valeur 3 \n");
13 break;
14 default : printf("i a une valeur differente de 1, 2 et 3 \n");
15 }
16 return 0;
17 }
```

4.3 Les boucles

4.3.1 L'instruction-while

Syntaxe : **while**(< **expression** >) < **instruction** >
Opération : Tant que < *expression* > est vraie exécuter < *instruction* >.

< *expression* > est évaluée d'abord, si elle est vraie alors < *instruction* > est exécutée.

Algorithme 12 : Programme C : Instruction while

```
1  #include < stdio.h >
2  int main()
3  {
4  int i = 10;
5  while(i <= 20)
6  {
7  printf("i = %d \n", i);
8  i++;
9  }
10 return 0;
11 }
```

4.3.2 L'instruction-dowhile

Syntaxe : **do** < **instruction** > **while**(< **expression** >);
Opération : Exécuter < *instruction* > tant que < *expression* > est vraie .

< *instruction* > est exécutée une première fois, puis < *expression* > est évaluée. Si elle est vraie alors < *instruction* > est exécutée à nouveau.

Algorithme 13 : Programme C : Instruction dowhile

```
1  #include < stdio.h >
2  int main()
3  {
4  int i = 10 ;
5  do
6  {
7  printf("i = %d \n", i) ;
8  i++ ;
9  }
10 while(i <= 20) ;
11 return 0 ;
12 }
```

4.3.3 L'instruction-for

Syntaxe :
for(< **expression$_1$ − opt** >; < **expression$_2$ − opt** >; < **expression$_3$ − opt** >) < **instruction** >
Opération : équivalent à
< *expression$_1$-opt* > ;
while (< *expression$_2$-opt* >)
{
< *instruction* >
< *expression$_3$-opt* > ;
}

< *expression$_1$-opt* > effectue les initialisations avant le début de la boucle, < *expression$_2$-opt* > est la condition de continuation de la boucle, qui elle est évaluée avant l'exécution du corps de la boucle, < *expression$_3$-opt* > est une expression évaluée à la fin du corps de la boucle.

Algorithme 14 : Programme C : Instruction for

```
1  #include < stdio.h >
2  int main()
3  {
4  int i ;
5  for(i = 10 ; i <= 20 ; i++) printf("i = %d \n", i) ;
6  return 0 ;
7  }
```

4.3.4 Les instructions *break* et *continue*

L'instruction *continue* ne peut être utilisée que dans une boucle (*while*, *dowhile*, *for*). Son effet est d'interrompre l'exécution des instructions de la boucle et de relancer l'évaluation de la condition.

L'instruction *break* ne peut être utilisée que dans le corps d'une boucle ou d'un *switch* et provoque la fin de ces instructions.

Chapitre 5

Fonctions

5.1 Analyse descendante

Une fonction est un sous-programme qui effectue un certain traitement puis renvoie ou non un résultat (à l'image du main()). Elle permet de regrouper des instructions et de lancer leur exécution grâce à un identificateur.

Cette notion est incontournable dans le cadre de l'analyse descendante (décomposition par cas) d'un problème. L'analyse descendante consiste à décomposer le problème de départ en plusieurs sous-problèmes qui peuvent à leur tour être décomposés en sous-sous-problèmes. Cela jusqu'à obtenir des sous-problèmes simples (qu'on peut résoudre facilement).

Un sous-programme (appelé fonction) dédié à la résolution de chaque sous-problème est donc défini. La combinaison de ces sous-programmes donne un programme, solution du problème de départ.

5.2 Définition de fonction

En général, on fait la différence entre les sous-programmes qui retournent un résultat (fonctions) et ceux qui n'en retournent pas (procédures). Dans C, ils sont tous considérés comme des fonctions avec une différence qui se fait à la définition.

La définition d'une fonction est composée de deux éléments :

a) **son prototype :** il permet de préciser la classe de mémorisation, le type, l'identificateur et les paramètres de la fonction ; c'est l'entête de la fonction.

$<$ **type** $><$ **identificateur** $>$ ($<$ **declaration** $-$ **ident** $>, ..., <$ **declaration** $-$ **ident** $>$)

$<$ *declaration* $-$ *ident* $>$ est la déclaration d'un paramètre par la donnée de son type suivi de son identificateur.

b) **son corps :** c'est une instruction-bloc

En résumé :

$<$ **type** $><$ **identificateur** $>$ ($<$ **declaration** $-$ **ident** $>, ..., <$ **declaration** $-$ **ident** $>$)
instruction $-$ **bloc**

Remarques :

- Les identificateurs d'une fonction ou d'une variable obéissent aux mêmes règles de construction.
- Le type d'une fonction est déterminée par le type du résultat qu'elle renvoie : int, long, double,etc. Une fonction qui ne retourne pas de valeur au programme appelant est de type

vide (void). Le type par défaut d'une fonction est int.
- L'utilisation du prototype permet de contrôler les arguments de la fonction :
  i) A la compilation, le nombre d'arguments de chaque appel est comparé au nombre de paramètres formels du prototype. S'il n'y a pas concordance, une erreur est signalée.
  ii) A chaque appel, le type de chaque argument est converti (si nécessaire et si cela a un sens) au type du paramètre formel correspondant.
- Dans le cas des fonctions de type non vide, on retrouve une instruction "return expression ;" dans le corps, qui retourne la valeur de *expression* comme résultat de la fonction. Cette instruction interrompt l'exécution de la fonction.
- Si la fonction n'a pas de paramètres, il est aussi possible de le spécifier par "void" :
  $<$ **type** $><$ **identificateur** $>$ (**void**)
  instruction − **bloc**

Exemple : Écrire une fonction qui prend deux variables à valeurs entières et positives x et n en paramètres pour calculer x^n.

Algorithme 15 : Programme C : Définition d'une fonction

```
1 int puissance(int x, int n)
2 {
3 int puiss = 1, i ;
4 for(i = 1 ; i <= n ; i++) puiss* = x ;
5 return puiss ;
6 }
```

5.3 Appel de fonction

Toutes les fonctions sont au même niveau, le main y compris. La seule particularité de ce dernier est qu'il constitue le point de départ de l'exécution. Par défaut, toute fonction peut en appeler une autre (mis à part le main).

Appel de fonction :
Syntaxe : $<$ **identificateur** $>$ ($<$ **argument$_1$** $>, ..., <$ **argument$_p$** $>$)
Syntaxe : $<$ **variable** $>=<$ **identificateur** $>$ ($<$ **argument$_1$** $>, ..., <$ **argument$_p$** $>$)

$<$ *argument* $>$ dénote un argument effectif. Les types des arguments doivent être compatibles avec les types des paramètres formels déclarés à la définition de la fonction.

5.4 Déclaration de fonction

Le compilateur peut rencontrer une référence à une fonction dont il connaît pas encore la définition : il ignore alors son type ; il lui attribue systématiquement le type int. Il n'y aura pas plus tard de vérification de la concordance avec la réalité. Ceci peut créer quelques problèmes. Pour y remédier, on peut procéder à la déclaration de la fonction.

Une fonction est déclarée dans un fichier à partir soit de sa définition, soit de son prototype.
$<$ **type** $><$ **identificateur** $>$ ($<$ **declaration** − **ident** $>, ..., <$ **declaration** − **ident** $>$);

5.5 Les variables : 2ème partie

Il existe trois types de variable :

- Les variables de fichier (globales) : elles sont déclarées à l'extérieur de toute fonction
- Les variables de bloc : elles sont déclarées à l'intérieur d'un bloc
- Les paramètres d'une fonction

5.5.1 Visibilité des variables

Une variable de bloc est visible (utilisable) de sa déclaration à la fin du bloc contenant cette déclaration.
Les variables globales sont visibles de leur déclaration à la fin du fichier.
Les paramètres d'une fonction ne sont visibles qu'à l'intérieur du corps de la fonction.
La déclaration d'un paramètre ou d'une variable de bloc masque localement la déclaration de toute variable de même nom faite à l'extérieur du bloc.

5.5.2 Passage de paramètres

La transmission des paramètres se fait par valeur : il y a création de nouvelles variables et copie des valeurs des arguments dans ces variables. Les modifications éventuelles sur ces nouvelles variables ne sont pas répercutées sur les arguments. Ceci à l'exception des variables de type tableaux pour lesquelles le passage de paramètres se fait directement par adresse.