INITIATION AU LANGAGE C

Table des matières

Chapitre 1

Introduction

1.1 Références

Les livres et supports de cours sur le Langage C de **Henri Garreta** et **Claude Delannoy** sont des références de premier choix pour apprendre les bases du langage. Ils sont facilement accessibles sur internet. Ils sont bien plus détaillés que le présent support et vont souvent au-delà de l'objet de ce cours. Toutefois, il plus que souhaitable de les consulter pour approfondir les notions évoquées ici.

1.2 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.

1.3 Présentation du langage

1.3.1 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 }
```

1.3.2 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...

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 }
```

1.3.3 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 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

— Très petite taille : **(unsigned) char**
Les entiers représentés sur au moins 1 octet (8 bits) : de -128 à 127 (de 0 à 255 pour les unsigned).
— Petite taille : **(unsigned) short**
Les entiers représentés sur au moins 2 octets (16 bits) : de -32.768 à 32.767 (de 0 à 65.535 pour les unsigned)
— Taille moyenne : **(unsigned) int**
Les entiers représentés sur au moins 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 au moins 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 au moins 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 adapté pour représenter sa valeur parmi les types int, long et long 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 au moins 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 au moins 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 au moins 8 octets (64 bits) : de -0.90E308 à -0.56E-308 et de 0.56E-308 à 0.90E308.

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 : < $\textbf{expression}_1$ > **OP** < $\textbf{expression}_2$ >

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 ++.

Souvent, le but de l'utilisation de cet opérateur (c'est également le cas pour l'opérateur de décrémentation) est d'ajouter 1 à la valeur courante d'une variable. Dès lors, comprendre la différence entre une post-incrémentation et une pré-incrémentation (idem pour la décrémentation) n'est utile que si on envisage d'utiliser la valeur de l'expression associée à la combinaison variable/opérateur.

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

À l'origine, et cela demeure pour beaucoup de compilateurs, les valeurs booléennes Vrai et Faux étaient représentées par des valeurs numériques. La valeur Faux utilisant la valeur numérique 0, La valeur Vrai (la négation de Faux) pouvait utiliser n'importe quelle valeur numérique différente de 0. Souvent, on réduit les possibilités numériques de la valeur Vrai à la seule valeur numérique 1, mais il faut garder à l'esprit que toutes valeurs numériques différentes de 0 renvoient à la valeur booléenne Vrai.

Les opérateurs logiques ci-dessous utilisent cette correspondance pour implémenter les opérations classiques en logique booléenne dans le sens originel. Dès lors, raisonner avec les valeurs booléennes ou les valeurs numériques correspondantes conduit au même résultat.

- **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.
  - **Ou :** $< expression_1 >$ && $< expression_2 >$ a pour valeur Faux si la valeur d'une des deux expressions est à Faux et Vrai 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.
  - **Ou :** $< expression_1 >$ || $< expression_2 >$ a pour valeur Faux si les valeurs des deux expressions sont à Faux et Vrai sinon.

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

2.3.7 Les opérateurs de comparaison

Les opérateurs classiques de comparaison sont implémentés dans le langage. La valeur des expressions correspondantes est de type numérique (0 ou 1) et repose sur la correspondance entre Faux et 0, Vrai et toute valeur différente de 0 (et bien évidemment 1).

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

Syntaxe : < **expression$_1$** > **OP** < **expression$_2$** > < $expression_1$ > OP < $expression_2$ > a pour valeur 1 si la comparaison est à Vrai et 0 sinon.
< $expression_1$ > et < $expression_2$ > doivent être de type simple (entiers, flottants ou pointeurs).

2.3.8 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).

2.4 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**("$\mathbf{texte_1}\%\mathbf{x_1}\mathbf{texte_2}\%\mathbf{x_2}$...", < $\mathbf{expression_1}$ >, < $\mathbf{expression_2}$ >, ...)

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.
— appel d'une fonction : **Syntaxe** : < **fonction**(...) >;
Opération : *fonction*(...) > exécute les instructions de la fonction.
— 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.

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 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   }
```

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, $< expression_3\text{-}opt >$ 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  :entrées x,n : int
4  :pré-cond : n ≥ 0
5  :sortie puiss : int
6  :post-cond : puiss = x^n
7  */
8  {
9  int puiss = 1, i ;
10 for(i = 1 ; i <= n ; i++) puiss* = x ;
11 return puiss ;
12 }
```

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** > (< $\mathbf{argument_1}$ >, ..., < $\mathbf{argument_p}$ >)
Syntaxe : < **variable** >=< **identificateur** > (< $\mathbf{argument_1}$ >, ..., < $\mathbf{argument_p}$ >)

< *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.

$< \textbf{type} >< \textbf{identificateur} > (< \textbf{declaration} - \textbf{ident} >, ..., < \textbf{declaration} - \textbf{ident} >);$

5.5 Les variables : 2ème partie

Il existe trois types de variables :
— 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.

5.5.3 Tests unitaires en boîte noire

Après la définition d'une fonction, il faudra s'assurer qu'elle réponde correctement au problème posé. Une vérification rigoureuse d'une fonction passe par une preuve de correction, complétude et finitude de l'algorithme sous-jacent. Toutefois, on se contente souvent d'une validation expérimentale au travers de la vérification sur jeu d'essais/de tests du comportement de la fonction. L'approche la plus simple pour une validation expérimentale est celle des tests unitaires en "boîte noire". Cette notion renvoie au fait qu'aucune analyse du code n'est réalisée : on suppose même que le code est inaccessible (dans une boîte noire). De ce fait, au travers de la spécification formelle de la fonction, il faut construire un jeu de tests permettant de révéler le plus grand nombre de bugs possible. Le fait que la fonction ait un comportement correct sur ce jeu de tests n'offre pas de garantie absolue sur sa correction, mais si le jeu de tests reflète fidèlement les conditions d'utilisation, on peut espérer qu'aucun bug ne se révèle lors de son utilisation.

La spécification formelle d'une fonction précise le type des entrées et des sorties, les préconditions sur les entrées, les liens entre les entrées et les sorties et de ce fait le but de la fonction. À partir de ces informations, il faut regrouper les entrées en classes d'équivalence dont les éléments devraient avoir un comportement similaire. Ensuite, on choisit quelques représentants dans chaque classe sur lesquels on exécute la fonction pour s'assurer de son bon fonctionnement. Dans le choix des représentants, on veillera à intégrer les extrema (valeurs limites) des classes car ils sont souvent sources d'erreurs dans les programmes.

On peut utiliser la fonction **void assert (int expression)** (en fait, il s'agit d'une macro se trouvant dans la bibliothèque assert.h) pour effectuer nos tests unitaires durant le développement d'un programme. **assert** évalue la valeur de "expression". Si elle est évaluée à vrai (non nulle), alors l'exécution du programme se poursuit normalement. Sinon, un message d'erreur est affiché sur la console précisant le fichier et la ligne du programme contenant l'expression, puis celui-ci s'arrête. Cela permet d'identifier précisément une ligne problématique et de corriger l'erreur plus

aisément.
Une fois la correction du programme terminée, il est important de supprimer les appels à **assert** car ils peuvent ralentir de manière considérable son exécution. La neutralisation de tous ces appels peut se faire avec la directive adressée au préprocesseur **#define NDEBUG 1**.

Algorithme 16 : Programme C : Fonction et tests

```
1  #include < stdio.h >
2  #include < assert.h >
3  int puissance(int x, int n)
4  /*
5  :entrées x,n : int
6  :pré-cond : n ≥ 0, x ≠ 0
7  :sortie puiss : int
8  :post-cond : puiss = x^n
9  */
10 {
11 int puiss = 1, i ;
12 for(i = 1 ; i <= n ; i++) puiss* = x ;
13 return puiss ;
14 }
15 int main()
16 {
17 assert(puissance(2,3)==8) ;
18 assert(puissance(0,5)==0) ;
19 assert(puissance(1,2)==1) ;
20 assert(puissance(3,0)==1) ;
21 assert(puissance(-1,3)==-1) ;
22 assert(puissance(-3,2)==9) ;
23 return 0 ;
24 }
```

Chapitre 6

Les Tableaux

6.1 Les tableaux à une dimension

Un tableau est un objet structuré. Il contient un certain nombre d'éléments de type identique.

Déclaration :
$<$ **type** $>$$<$ **identificateur** $>$ [$<$ **expression** $-$ **constante** $>$];
$<$ **type** $>$$<$ **identificateur** $>$ [$<$ **expression** $-$ **constante** $-$ **opt** $>$] =
{$<$ **expression**$_1$ $>$, ..., $<$ **expression**$_n$ $>$};

A la déclaration d'un tableau, le programme procède à l'allocation d'un espace mémoire suffisant pour contenir le tableau. Les éléments du tableau sont donc dans des cases mémoires contiguës.

Le premier élément est à la position d'indice 0. Pour un tableau *tab* déclaré de taille N, les éléments ont pour indices : 0, 1,..., $N-1$. La syntaxe pour accéder à ces éléments est *tab*[i], $i = 0, 1, ..., N-1$. Chaque élément d'un tableau a un comportement similaire à celui d'une variable. Ce qui n'est pas le cas du tableau dans sa globalité. En effet, il n'est pas possible d'affecter un tableau à un autre tableau. Pour cela, il faut procéder à l'affectation élément par élément.

Exemple : Écrire les fonctions permettant la saisie dans un tableau et l'affichage de notes.

6.2 Les tableaux multidimensionnels

Il n'existe pas en C un type dédié pour les tableaux à plusieurs dimensions. Cependant, les éléments d'un tableau unidimensionnel peuvent être à leur tour des tableaux. Cela rend possible la définition de tableaux multidimensionnels.

Déclaration :
$<$ **type** $>$$<$ **identificateur** $>$ [$<$ **expression**$_1$ $-$ **constante** $>$]...[$<$ **expression**$_p$ $-$ **constante** $>$];

Exemple 6.2.1 *int matrice[10][10] ;*
matrice[i] est un tableau de 10 entiers et matrice[i][j] est un élément (entier) du tableau matrice[i] (si $0 <= i, j < 10$*).*

Algorithme 17 : Programme C : Tableaux et fonctions

```
1  #include < stdio.h >
2  int saisie_notes(int tab[100])
3  /*
4  :entrée/sortie tab : tableau de 100 int
5  :pré-cond : ∅
6  :sortie i : int
7  :post-cond : remplit les i premières cases de tab avec des entiers saisis au clavier et
   renvoie le nombre i de valeurs saisies
8  */
9   {
10      int note, i = 0;
11      do
12      {
13          printf("Donner la note suivante ou taper -1 si terminé : ");
14          scanf("%d", &note);
15          if(note >= 0)
16          {
17              tab[i] = note;
18              i++;
19          }
20      }
21      while(note!=-1 && i < 100);
22      return i;
23  }
```

Algorithme 18 : Programme C : Tableaux et fonctions

```
1  #include < stdio.h >
2  void affichage_notes(int tab[100], int n)
3  /*
4  :entrée tab : tableau de 100 int
5  :entrée n : int
6  :pré-cond : n < 100
7  :post-cond : affiche les n premières valeurs de tab
8  */
9  {
10 int i;
11 for(i=0; i<n; i++) printf("%d \n", tab[i]);
12 }
```

Chapitre 7

Les Pointeurs

7.1 Définition

Un pointeur est une variable dont la valeur est l'adresse d'une cellule de la mémoire. Les pointeurs sont utilisés énormément en C. En effet, ils permettent de construire des programmes où le nombre et la taille des informations à gérer ne sont pas fixés à la compilation mais ajustés dynamiquement en cours d'exécution.

Cependant, il faut bien noter qu'un pointeur demeure une variable comme une autre et qu'une adresse n'est qu'un nombre entier qui donne l'indice d'un élément du tableau qu'est la mémoire de l'ordinateur. De ce fait, un pointeur doit pouvoir supporter toutes les opérations réalisables sur un nombre si tant est qu'elles aient un sens quand ce nombre exprime un rang. L'apport d'un pointeur réside dans la possibilité de manipuler la variable dont la valeur du pointeur est l'adresse.

7.2 Déclaration et Initialisation

Déclaration : $<$ **type** $>*<$ **identificateur** $><=$ **initialisation** $-$ **opt** $>$;
Cette déclaration entraîne la création d'une variable pointeur qui ne peut contenir qu'une adresse contenant ou susceptible de contenir une donnée dont le type est $<$ **type** $>$. Cette variable peut être initialisée à la déclaration : cela est optionnel.

Si le pointeur est typé c'est-à-dire qu'il connaît le type de l'objet pointé, il permet la manipulation de ce dernier. Sinon, il est atypique (il contient une simple adresse) et permet de repérer en mémoire n'importe quel objet mais pas de le manipuler.

Déclaration pointeur atypique : void$*<$ **identificateur** $><=$ **initialisation** $-$ **opt** $>$;

Algorithme 19 : Programme C : Déclaration de Pointeurs

```
1 #include < stdio.h >
2 int main( )
3 {
4 int variable = 1 ;
5 int * pt = NULL ;
6 int ** pt_pt = NULL ;
7 int **** pt4 ;
8 }
```

Par convention, on attribue la valeur **NULL** à un pointeur qui ne pointe sur aucun objet. A la déclaration d'une variable pointeur, il est possible qu'elle contienne une valeur différente de **NULL**. Cette valeur sera interprétée comme une adresse si cette variable est utilisée sans initialisation ou modification, et cela peut détruire des données et causer des incohérences dans le programme.

La taille d'une variable pointeur ne dépend pas de l'objet pointé, mais de la machine utilisée : c'est le nombre d'octets nécessaire pour coder une adresse sur cette machine.

7.3 Les Opérateurs & et *

L'opérateur **&** permet de connaître l'adresse d'une variable. Notons qu'appliquer cet opérateur à une constante n'a pas de sens.

Syntaxe : & < **variable** >

Algorithme 20 : Programme C : Initialisation de Pointeurs

```
1 #include < stdio.h >
2 int main( )
3 {
4 int variable = 1 ;
5 int * pt_variable = &variable ; /* Commentaire : pt_variable pointe vers variable */
6 int ** pt_pt = &pt_variable ; /* Commentaire : pt_pt pointe vers pt_variable */
7 }
```

L'opérateur d'indirection ***** est l'inverse de **&** : il signifie "objet pointé par". Il permet de manipuler l'objet repéré par un pointeur typé.

Syntaxe : ∗ < **expression** >
où < **expression** > est de type pointeur typé.

Algorithme 21 : Programme C : Initialisation de Pointeurs

```
1 #include < stdio.h >
2 int main( )
3 {
4 int variable = 1 ;
5 int * pt_variable = &variable ;
6 /* Commentaire : pt_variable est un pointeur, ∗pt_variable est l'objet pointé et donc
  ∗pt_variable vaut 1 */
7 }
```

7.4 Arithmétique des adresses

Certaines opérations définies sur les nombres gardent un sens lorsque les opérandes sont des adresses. Si **exp** est l'adresse d'un objet $\mathbf{O_1}$, $\mathbf{exp + 1}$ exprime l'adresse de l'objet de même type que $\mathbf{O_1}$ et qui se trouverait dans la mémoire immédiatement après $\mathbf{O_1}$ et $\mathbf{exp - 1}$ l'adresse de celui qui se trouverait juste avant.

Ainsi $\mathbf{exp} + \mathbf{n}$ correspond à $(\mathbf{Type}*)(\mathbf{exp} + \mathbf{n} * \mathbf{sizeof}(\mathbf{Type}))$ (on rajoute n fois la taille de l'objet pointé). De même $\mathbf{exp_2} - \mathbf{exp_1}$ correspond à $(\mathbf{exp_2} - \mathbf{exp_1})/\mathbf{sizeof}(\mathbf{Type})$ dans le cas où $\mathbf{exp_2}$ et $\mathbf{exp_1}$ sont de types tout à fait identiques.

L'indexation permet également d'avoir une notation différente pour certaines de ces opérations. En effet, $*(\mathbf{exp} + \mathbf{1})$ et $\mathbf{exp}[\mathbf{1}]$ sont équivalentes, de même que $*(\mathbf{exp} + \mathbf{n})$ et $\mathbf{exp}[\mathbf{n}]$.

7.5 Passage de paramètres et Pointeurs

Le passage de paramètres en C se fait uniquement par valeur. Cependant, il est parfois nécessaire (voire souvent) qu'une fonction modifie la valeur d'une variable définie dans le programme appelant, celle-ci étant passée en argument. Prenons l'exemple de la fonction **echange** qui doit échanger la valeur de deux variables passées en argument. Nous savons que donner directement en paramètre ces deux variables à notre fonction ne permet pas de résoudre le problème étant donné que le passage de paramètres se fait par valeurs (la fonction travaille sur des copies). La solution consiste à donner comme arguments à **echange** des pointeurs sur les variables en question.

Algorithme 22 : Programme C : Échange des valeurs de deux variables

```
1  #include < stdio.h >
2  void echange (int * p, int * q)
3  {
4  int x = *p;
5  *p = *q;
6  *q = x;
7  }
8  int main( )
9  {
10 int a = 2, b = 3;
11 echange(&a, &b);
12 }
```

7.6 Tableaux et Pointeurs

L'identificateur d'un tableau en C est un pointeur constant vers le premier élément du tableau.

Considérons la déclaration **int tab**[**10**];, **tab** est équivalent à &**tab**[**0**]. Cela explique pourquoi le passage d'un tableau en paramètre d'une fonction en permet la modification. En outre, on peut en déduire l'équivalence de $*\mathbf{tab}$ et $\mathbf{tab}[\mathbf{0}]$, de même que celle de $*(\mathbf{tab} + \mathbf{1})$ et $\mathbf{tab}[\mathbf{1}]$, $*(\mathbf{tab} + \mathbf{2})$ et $\mathbf{tab}[\mathbf{2}]$... Ce constat est valable aussi pour une variable pointeur quelconque : il est possible d'utiliser l'opérateur d'indexation.

On observe qu'au final un tableau et un pointeur sont des concepts très voisins. Cependant, une différence notable réside dans le fait que l'identificateur d'un tableau est une constante contrairement à une variable de type pointeur. Il est donc interdit de modifier l'adresse contenue dans l'identificateur d'un tableau.

7.6.1 Tableaux multidimensionnels et Pointeurs

L'identificateur d'un tableau multidimensionnel demeure un pointeur vers le premier élément du tableau. Si, on prend l'exemple d'un tableau à deux dimensions **int matrice**[**10**][**15**], les

éléments sont rangés de manière linéaire dans des cases contigües de la mémoire (les éléments de la première ligne étant suivis par ceux de la deuxième...). **matrice** est un pointeur de type **int**$*$ qui contient l'adresse du premier élément du tableau. Ainsi, l'utilisation de l'indirection pour répérer les éléments de notre tableau est beaucoup plus complexe que dans le cas des tableaux unidimensionnels. En effet, l'expression **matrice**[**i**][**j**] est équivalente à
$*(\mathbf{matrice} + (\mathbf{i} * \mathbf{15} + \mathbf{j}) * \mathbf{sizeof}(\mathbf{int}))$.

7.7 Conversion de type : opérateur de "cast"

La définition d'un pointeur générique permet d'avoir une variable compatible avec tout type d'adresse. Par contre l'utilisation et la modification de l'objet pointé nécessitent de savoir son type. De ce fait, il peut être nécessaire de repasser à un pointeur typé pour intervenir sur l'objet. La conversion de type (cast) permet de modifier le type d'un objet : il suffit de rajouter le nouveau type entre parenthèses juste devant l'expression.

Syntaxe : (**nouveau_type**)**expression**
Nous listons ci-dessous les conversions dites légitimes :
- entier vers entier plus long : le codage de **expression** est étendu de sorte que sa valeur soit inchangée.
- entier vers entier plus court : si la valeur de **expression** est assez petite pour tenir dans le nouveau type, la valeur est inchangée. Sinon, elle est tronquée (sans intérêt).
- entier signé vers non signé et vice versa : l'interprétation du compilateur change alors que le codage reste inchangé.
- flottant vers entier : la partie fractionnaire est supprimée.
- entier vers flottant : le flottant obtenu est celui qui approche le mieux l'entier sauf en cas de débordement (imprévisible).
- adresse d'un objet de type $\mathbf{Type_1}$ vers adresse d'un objet de type $\mathbf{Type_2}$: l'interprétation change alors que le codage demeure inchangé.
- entier vers adresse d'un objet de type **Type** : l'interprétation change alors que le codage reste inchangé.

De nombreux dangers sont présents dans ces conversions de type. L'opération de "cast" est rarement nécessaire car l'objectif principal est de faire taire le compilateur.

Chapitre 8

Chaînes de caractères

Une chaîne de caractères est un tableau de caractères qui se termine par le caractère **NULL** ('\0').
Une constante chaîne de caractères est constituée d'une suite de caractères délimitée par **" "**. Le compilateur ajoute automatiquement le caractère **NULL** et range la suite dans la zone des données statiques et la considère comme un tableau de caractères sans nom. Dans un programme, les chaînes identiques ne sont pas nécessairement stockées à des endroits distincts ; aussi, il est fortement déconseillé de modifier le contenu d'une chaîne constante.

8.1 Les variables chaînes de caractères

On peut manipuler les chaînes de caractères à l'aide de tableaux ou de pointeurs.

Algorithme 23 : Programme C : Déclaration et Initialisation de chaînes de caractères

```
1  #include < stdio.h >
2  int main( )
3  {
4  char * message ="coucou le monde" ;
5  /* Commentaire : "coucou le monde" est une constante dont l'adresse du premier
   caractère est affectée à la variable message */
6  char tab[ ] ="coucou le monde" ;
7  /* Commentaire : l'identificateur tab est une constante mais les éléments du tableau sont
   modifiables */
8  char tab2[ ] = {'c','o','u','c','o','u',' ','l','e',' ','m','o','n','d','e','\0'} ;
9  /* Commentaire : les déclarations et initialisations des deux tableaux sont équivalentes */
10 }
```

8.2 Manipulation

La fonction **scanf** permet la saisie de chaînes de caractères grâce au spécificateur de type **%s**. Toutefois, elle ne permet pas de gérer les espaces dans les chaînes de caractères : ils sont traités comme des symboles de fin de chaînes.

La bibliothèque **stdio.h** contient d'autres fonctions, comme **gets** et **fgets**, permettant la saisie des chaînes de caractères contenant des espaces. Sachant que la fonction **gets** ne permet de borner le nombre de caractères lus, il est donc préférable d'utiliser **fgets**.

Pour lire au clavier une chaîne de caractères avec la fonction **char *fgets(char *s, int n, FILE *flot) ;**, il suffit de lui donner comme argument troisième argument **stdin** qui est l'entrée standard (clavier). Ainsi, elle va stocker dans **s** tous les caractères saisis jusqu'au premier caractère de fin de ligne en veillant à ne pas lire plus de **n - 1** caractères. Elle sauvegardera dans la chaîne le caractère de fin de ligne et y ajoutera automatiquement le caractère de fin de chaîne.

La fonction **int sprintf(char *destination, const char *format, ...)** procède de manière identique à printf sauf que le résultat n'est pas affiché sur la console, mais stockée dans la chaîne **destination**. De même, la fonction **int sscanf(const char *source, const char *format, ...)** fait le même traitement que scanf, mais en remplaçant la lecture au clavier par celle dans la chaîne **source**.

Enfin, la bibliothèque **string.h** contient également des fonctions facilitant la manipulation des chaînes de caractères telles que : **strlen** (calcule la longueur d'une chaîne), **strcpy** (copie une chaîne dans une autre), **strcat** (concatène deux chaînes), **strcmp** (compare deux chaînes), **strchr** (recherche d'un caractère dans une chaîne)....

Algorithme 24 : Programme C : Saisie de chaînes de caractères

```
1 #include < stdio.h >
2 int main( )
3 {
4 char chaine[50] ;
5 fgets(chaine, 50, stdin) ;
6 printf("chaine = %s \n", chaine) ;
7 return 0 ;
8 }
```

Chapitre 9

Gestion dynamique de la mémoire

L'attribution d'une adresse valide à un pointeur peut être faite de manière simple en lui affectant l'adresse d'une variable existante (déclarée au préalable). Une autre manière de procéder consiste à demander au système l'allocation d'un nouvel espace mémoire, et cela grâce à des fonctions de la bibliothèque **stdlib.h**. Cette approche permet une gestion dynamique de la mémoire en réservant des zones, non plus durant toute l'exécution du programme, mais seulement pendant le laps de temps où cela est nécessaire.

9.1 Principales fonctions de gestion de la mémoire

Voici les prototypes de ces fonctions (**size_t** est un entier non signé déclaré dans **stdlib.h**) :
- **void * calloc (size_t nb, size_t taille) ;**
- **void free (void * pointeur) ;**
- **void * malloc (size_t nb_octets) ;**
- **void * realloc (void * pointeur, size_t nb_octets) ;**

9.1.1 Allocation

- **malloc (nb_octets) ;**
  La fonction **malloc** fournit un pointeur de type **void*** sur une zone de la mémoire dont la taille est **nb_octets** ou **NULL** en cas d'échec.
- **calloc(nb, taille) ;**
  La fonction **calloc** fournit un pointeur de type **void *** sur une zone de la mémoire permettant de ranger **nb** objets de grandeur **taille** ou **NULL** en cas d'échec.

- **realloc(pointeur, nb_octets) ;**
  La fonction **realloc** permet d'agrandir une zone précédemment allouée et repérée par le pointeur **pointeur** à la taille **nb_octets**, sans perte d'informations. En cas d'échec, **realloc** retourne le pointeur **NULL** et le bloc pointé par **pointeur** peut être détruit.

9.1.2 Libération

- **free(pointeur) ;**
  La fonction **free** libère l'emplacement mémoire précédemment alloué par l'une des fonctions précédentes.

Algorithme 25 : Programme C : malloc

```
1  #include < stdio.h >
2  #include < stdlib.h >
3  int main( )
4   {
5       int * pt = NULL ;
6       pt = malloc(sizeof(int)) ;
7       if(pt == NULL)
8       {
9           printf("Echec allocation \n") ;
10          return 1 ;
11      }
12      return 0 ;
13  }
```

Algorithme 26 : Programme C : calloc

```
1  #include < stdio.h >
2  #include < stdlib.h >
3  int main( )
4   {
5       int * pt = NULL ;
6       pt = calloc(5, sizeof(int)) ;
7       if(pt == NULL)
8       {
9           printf("Echec allocation \n") ;
10          return 1 ;
11      }
12      return 0 ;
13  }
```

Algorithme 27 : Programme C : realloc

```
1  #include < stdio.h >
2  #include < stdlib.h >
3  int main( )
4  {
5      int * pt1 = NULL, * pt2 = NULL;
6      pt1 = calloc(5, sizeof(int));
7      if(pt1 == NULL)
8      {
9          printf("Echec allocation \n");
10         return 1;
11     }
12     pt2 = realloc(pt1, 10*sizeof(int));
13     if(pt2 == NULL)
14     {
15         printf("Echec reallocation \n");
16         return 2;
17     }
18     return 0;
19 }
```

9.1.3 Les tableaux dynamiques

L'utilisation de tableaux dynamiques permet de ne pas encombrer la mémoire avec des cases inutiles dans le cas où, par exemple, c'est à l'utilisateur de fixer le nombre d'éléments. En effet, à la déclaration d'un tableau fixe, la taille est forcément fournie par une expression constante. Dans le cas des tableaux statiques à taille variable introduits par la norme C99, leur stockage se fait dans la pile d'exécution en limitant ainsi la taille. En outre, il s'agit de variables automatiques dont la durée d'existence est limitée au bloc les contenant.

On peut procéder de la même manière pour définir des tableaux multidimensionnels dynamiques.

9.2 Organisation du programme en mémoire

En mémoire, le stockage des informations associées à un programme est fait dans l'ordre suivant :

- Le code du programme (les instructions)
- La zone des données statiques (variables globales, les variables locales statiques, les constantes)
- Le tas qui contient les données gérées de manière dynamique (allocation/libération de mémoire)
- La pile qui contient les variables automatiques (variables locales et paramètres)

Algorithme 28 : Programme C : Tableaux dynamiques

```
1  #include < stdio.h >
2  #include < stdlib.h >
3  int main( )
4   {
5       int n, ∗tab;
6       printf("Donner la taille du tableau \n") ;
7       scanf("%d", &n) ;
8       tab =calloc(n,sizeof(int)) ;
9       if(tab == NULL)
10      {
11          printf("Echec allocation \n") ;
12          return 1 ;
13      }
14      initialiser(tab,n) ;
15      afficher(tab,n) ;
16      free(tab) ;
17      return 0 ;
18  }
```

Algorithme 29 : Programme C : Tableaux multidimensionnels dynamiques

```
1  #include < stdio.h >
2  #include < stdlib.h >
3  int main( )
4   {
5      int n, m, i, j, ∗matrice;
6      printf("Donner le nombre de lignes : ") ;
7      scanf("%d", &n) ;
8      printf("Donner le nombre de colonnes : ") ;
9      scanf("%d", &m) ;
10     matrice =calloc(n,sizeof(int*)) ;
11     if(matrice == NULL)
12     {
13         printf("Echec allocation \n") ;
14         return 1 ;
15     }
16     else
17     {
18         for(i = 0; i < n; i + +)
19         {
20             matrice[i] =calloc(m, sizeof(int)) ;
21             if(matrice[i] == NULL)
22             {
23                 for(j = 0; j < i; j + +) free(matrice[j]) ;
24                 free(matrice) ;
25                 printf("Echec allocation \n") ;
26                 return 1 ;
27             }
28         }
29         initialiser2(matrice, n, m) ;
30         afficher2(matrice, n, m) ;
31         for(i = 0; i < n; i + +) free(matrice[i]) ;
32         free(matrice) ;
33     }
34     return 0 ;
35 }
```

Chapitre 10

Les structures

Les structures sont des variables composées de champs de types différents, chaque champ étant repéré par un nom.

Syntaxe :
< **struct** >< **identificateur** − **opt** >
{
< **declaration** >;
...
< **declaration** >;
} < **identificateur** − **opt** ><= {**initialisation**, ..., **initialisation**} − **opt** >, ...,
< **identificateur** − **opt** ><= {**initialisation**, ..., **initialisation**} − **opt** >;

Ceci déclare une structure dont les différents champs sont déclarés entre accolades, mais aussi des variables qui sont des structures de ce type. L'utilisation de l'identificateur permet par la suite de déclarer d'autres variables de ce type en faisant précéder uniquement struct identificateur comme indicateur de type.

L'accès à un champ d'une structure est effectué grâce à l'opérateur "**.**". Cependant si la variable est un pointeur vers une structure, on accède à un champ par l'opérateur "− >".

Algorithme 30 : Programme C : Déclaration de Structure

```
1  #include < stdio.h >
2  struct etudiant
3    {
4       char nom[30], prenom[30] ;
5       int groupe ;
6       float moyenne ;
7    }
8  int main( )
9    {
10      struct etudiant x = {"ndiaye", "samba", 1} ;
11      /* Le champ moyenne va être initialisé à 0 */
12      struct etudiant *y = &x;
13      x.moyenne = 10. ;
14      y− > moyenne = 15.5 ;
15      return 0 ;
16   }
```

Manipulations des structures :
— On peut affecter une expression d'un type structure à une variable de type structure pourvu que ces deux types soient identiques.
— Les structures sont passées par valeur lors de l'appel des fonctions.
— Le résultat d'une fonction peut être une structure.

10.1 Typedef : définition de nouveaux types

Cet opérateur permet de définir de nouveaux types. Cela permet d'alléger par la suite les expressions.

Syntaxe : **typedef** < **declaration** >
l'identificateur de la déclaration est le nom du nouveau type.

Algorithme 31 : Programme C : Définition de nouveaux types

```
1  #include < stdio.h >
2  typedef struct etudiant
3   {
4      char nom[30], prenom[30] ;
5      int groupe ;
6      float moyenne ;
7   } ETUDIANT ;
8  int main( )
9   {
10     ETUDIANT x = {"ndiaye", "samba", 1} ;
11     /* Le champ moyenne va être initialisé à 0 */
12     ETUDIANT *y = &x;
13     x.moyenne = 10. ;
14     y- > moyenne = 15.5 ;
15     return 0 ;
16  }
```

Chapitre 11

Fichiers : lecture et écriture

Les systèmes informatiques offrent des espaces de stockage permettant de sauvegarder des données sous-forme de fichiers. La représentation interne de ces données est faite au format binaire peu importe leur nature (texte, audio, vidéo, image,exécutable...). Les fonctions standard de manipulation des fichiers du langage C permettent de les voir comme un flot, une suite d'octets traduisant leur représentation interne. Toutefois, dans le cas d'un fichier texte, il est possible d'obtenir une interprétation des données en suite de caractères afin de travailler directement sur le texte. Évidemment, le traitement d'un fichier au format binaire est plus rapide que celui d'un fichier texte car il ne nécessite aucune interprétation des données.

Les interactions avec un fichier se font souvent au travers d'une mémoire tampon pour accélérer leur exécution. En effet, les opérations de lecture/écriture sur un disque sont bien plus lentes que sur la mémoire interne de l'ordinateur. De ce fait, un espace mémoire (dit mémoire tampon) est réservé dans la mémoire interne à l'ouverture d'un fichier. La mémoire tampon est de taille réduite du fait de la petite taille de la mémoire interne comparée aux espaces de stockages. Elle permet de charger en mémoire une partie du fichier pour accélérer sa lecture ou d'accumuler des données à écrire dans le fichier afin de les écrire en une seule fois. Il est donc possible que des données écrites dans un fichier ne soit pas immédiatement visibles. Pour cela, il faut attendre que le contenu de la mémoire tampon soit écrite dans le fichier.

L'accès aux données d'un fichier est fait de manière séquentielle par défaut. Toute opération de lecture place l'indicateur de position (curseur) à la suite des données lues afin de lire la suite à la prochaine opération de lecture. De même, toute opération d'écriture place le curseur à la suite des données écrites afin d'écrire à la suite de celles-ci. Il existe des fonctions de la bibliothèque standard qui permettent un accès direct aux données en déplaçant le curseur à la position souhaitée dans le fichier.

11.1 Principales fonctions de gestion de fichiers

Ces fonctions sont incluses dans la bibliothèque stdio et font référence au type structure FILE, également défini dans cette bibliothèque. Cette structure permet de représenter un flot et d'intérargir avec le fichier associé.

11.1.1 Fonctions générales

— **FILE *fopen(const char *nom_fic, const char *mode)** : elle permet d'ouvrir un flot lié au fichier dont le chemin d'accès est contenu dans la chaîne **nom_fic**. Cette étape est nécessaire avant de pouvoir lire ou modifier le contenu d'un fichier. Il faut également préciser le mode d'ouverture du fichier dans la chaîne **mode**. La fonction renvoie un pointeur sur une structure de type FILE ou NULL en cas d'échec. Si on souhaite utiliser

le fichier au format binaire, on peut le préciser en ajoutant la lettre **b** au mode. Les modes d'ouverture possibles pour un fichier sont :
— "r" : lecture, le fichier doit exister et le curseur est placé au début du fichier
— "w" : écriture, si le fichier existe son contenu est effacé, sinon il est créé
— "a" : ajout, si le fichier n'existe pas, il est créé, le curseur est placé à la fin du fichier
— "r+" ou "w+" ou "a+" : lecture et écriture
— "rb", "wb", "ab", "r+b", "w+b", "a+b" : ... sur un fichier binaire.
Il existe trois flots ouverts automatiquement au lancement d'un programme et fermé automatiquement à sa fin : **stdin** (entrée standard : par défaut, le clavier), **stdout** (sortie standard : par défaut, la console), **stderr** (sortie erreur : par défaut, la console).

— **int fclose(FILE *flot)** : elle ferme **flot** et libère les espaces mémoire alloués. Il faut fermer les flots ouverts avant la fin du programme afin de préserver la cohérence des fichiers concernés avec notamment l'écriture des données de la mémoire tampon.
— **int feof(FILE *flot)** : si **flot** est en mode lecture, elle permet de détecter la fin du fichier. Elle renvoie vrai (une valeur non nulle) s'il y a eu une tentative de lecture au-delà de la fin du fichier. Il faut donc au moins une tentative de lecture pour qu'elle puisse passer à vrai.
— **int fflush(FILE *flot)** : elle permet d'écrire immédiatement le contenu de la mémoire tampon vers **flot**.

11.1.2 Fichiers binaires

— **size_t fread(void *destination, size_t taille, size_t nombre, FILE *flot)** : elle copie depuis **flot nombre** objets, chaque objet étant constitué de **taille** octets, vers l'espace mémoire pointé par **destination**. Elle renvoie le nombre d'objets effectivement copiés.
— **size_t fwrite(const void *source, size_t taille, size_t nombre, FILE *flot)** : elle copie depuis **source nombre** objets, chaque objet étant constitué de **taille** octets, vers **flot**. Elle renvoie le nombre d'objets effectivement copiés.
— **int fseek(FILE *flot, long deplacement, int origine)** : elle permet de déplacer le curseur (indicateur de position) dans **flot** afin que la prochaine opération de lecture/écriture se fasse à partir de cette position. La nouvelle position du curseur est déterminée à partir de la position **origine** à laquelle on ajoute la valeur **deplacement**. On peut utiliser les constantes suivantes définies dans stdio.h pour fixer la valeur de **origine** :
  — **SEEK_SET** : début du flot
  — **SEEK_CUR** : la position courante du curseur
  — **SEEK_END** : la fin du flot
— **long ftell(FILE *flot)** : elle renvoie la position courante du curseur dans **flot**. On peut l'utiliser notamment pour déterminer la fin d'un fichier et ainsi éviter les tentatives de positionnement du curseur en dehors de celui-ci.
— **void fgetpos(FILE *flot, fpos t *ptr)** : elle stocke dans **ptr** la position courante du curseur dans **flot** afin de pouvoir l'utiliser ultérieurement dans la fonction **fsetpos**
— **void fsetpos(FILE *flot, const fpos t *ptr)** : elle positionne le curseur de **flot** à la position **ptr** obtenue préalablement grâce à la fonction **fgetpos**
— **void rewind(FILE *flot)** : elle positionne le curseur au début de **flot**

11.1.3 Fichiers texte

— **int fgetc(FILE *flot)** : elle renvoie le prochain caractère dans **flot** ou **EOF** si la fin du flot est atteinte.
— **char *fgets(char *s, int n, FILE *flot)** : elle copie dans **s** la prochaine ligne de **flot** en veillant en ne pas dépasser **n - 1** caractères. Elle sauvegarde dans **s** le caractère de fin

de ligne et ajoute automatiquement le caractère de fin de chaîne. Elle renvoie la chaîne **s** en cas de succès et **NULL** sinon.
- **int fputc(int caractere, FILE *flot)** : elle écrit **caractere** dans **flot**. Elle renvoie le caractère écrit en cas de succès, **EOF** sinon.
- **int fputs(const char *s, FILE *flot)** : elle écrit **s** dans **flot**. Elle renvoie une valeur ≥ 0 en cas de succès, **EOF** sinon.
- **int fprintf(FILE *flot, const char *format, ...)** : elle procède comme printf en remplaçant la sortie standard (stdout) par **flot**.
- **int fscanf(FILE *flot, const char *format, ...)** : elle procède comme scanf en remplaçant l'entrée standard (stdin) par **flot**.

En cas d'erreur détectée dans un programme et nécessitant l'arrêt de celui-ci, on peut utiliser la fonction **void exit(int code)** définie dans la bibliothèque stdlib qui permet de fermer tous les fichiers ouverts au préalable. Le code retour donné en argument peut être choisi parmi les constantes **EXIT_SUCCESS** et **EXIT_FAILURE** traduisant sur le système sous-jacent que le programme a réussi ou échoué dans l'accomplissement de son objectif.

Exemple : Afficher le contenu d'un fichier texte à l'écran puis ajouter à la fin la chaîne "DONE".

Algorithme 32 : Programme C : E/S fichiers

```
1  #include < stdio.h >
2  #include < stdlib.h >
3  #define nom_fic "../exemples/fic.txt"
4  int main( )
5   {
6      char chaine[100] ;
7      FILE * fic = fopen(nom_fic,"r+") ;
8      if(fic == NULL)
9      {
10         printf("Echec ouverture fichier \n") ;
11         exit(EXIT_FAILURE) ;
12     }
13     do
14     {
15         fgets(chaine,100,fic) ;
16         printf("%s",chaine) ;
17     }
18     while( ! feof(fic)) ;
19     if(fprintf(fic,"DONE \n") < 0)
20     {
21         printf("Echec écriture \n") ;
22         exit(EXIT_FAILURE) ;
23     }
24     fclose(fic) ;
25     return 0 ;
26  }
```

Chapitre 12

Quelques éléments supplémentaires

12.1 Variables, fonctions et compilation séparée

12.1.1 Identificateurs publics et privés

Cette partie donne les règles qui régissent la visibilité inter-fichiers des identificateurs. La question concerne uniquement les noms des variables et des fonctions. Le problème ne se pose pas pour les variables locales dont la visibilité est restreinte à l'étendue du bloc ou de la fonction contenant leur définition. Il sera donc uniquement question des noms des variables globales et des fonctions.

Un nom de variable ou de fonction définit dans un fichier source et pouvant être utilisé dans d'autres fichiers sources est dit public. Un identificateur qui n'est pas public est dit privé.

— sauf indication contraire, tout identificateur global est public ;
— le qualifieur **static**, précédent la déclaration d'un identificateur global, rend celui-ci privé

12.1.2 Déclaration d'objets externes

Nous ne considérons désormais que les noms publics. Un identificateur référencé dans un fichier alors qu'il est défini dans un autre fichier est appelé externe. En général, les noms externes doivent faire l'objet d'une déclaration car le compilateur ne traite qu'un fichier à la fois.

— Toute variable doit être définie (déclaration normale) ou déclarée externe avant son utilisation
— Une fonction peut être référence sans aucune définition ou déclaration externe au préalable ; elle est alors supposée externe, de type int et sans prototype. Il est donc préférable de procéder à sa définition ou déclaration avant toute utilisation.

Syntaxe declaration externe variable : **extern** < **declaration** >

Syntaxe declaration externe fonction : < **extern** − **opt** >< **prototype** >

12.1.3 Variables locales statiques

Le qualifieur **static**, placé devant la déclaration d'une variable locale, produit une variable locale pour sa visibilité et statique pour sa durée de vie (permanente).

12.1.4 Variables critiques

Le qualifieur **register** précédant une déclaration de variable informe le compilateur que la variable en question est très fréquemment accédée pendant l'exécution du programme et qu'il y a donc lieu de prendre toutes les dispositions pour en accélérer l'accès.

Les variables ainsi déclarées doivent être locales et d'un type simple (nombre, pointeur).

12.1.5 Variables constantes et volatiles

Le qualifieur **const** placé devant une variable ou un argument formel informe le compilateur que la variable ou l'argument ne changera pas de valeur tout au long de l'exécution du programme ou de l'activation de la fonction. Ce renseignement permet au compilateur d'en optimiser la gestion.

Le qualifieur **volatile** informe le compilateur que la valeur de la variable en question peut changer mystérieusement, y compris dans une section du programme qui ne comporte aucune référence à cette variable.