C

Ch 2 – Caractéristiques de base

I. INTRODUCTION 2

A. STRUCTURE GENERALE 2
B. UN PREMIER EXEMPLE DE FICHIER 3
C. LES COMMENTAIRES EN C 4
D. IDENTIFIER LE PROGRAMME 4
E. LES DIRECTIVES DU PREPROCESSEUR 4
1. Directives d'inclusion : 4
2. Macros 5
F. POINT D'ENTREE DU PROGRAMME : FONCTION « MAIN » 5
G. DECLARATION DES DONNEES – CONSTANTES ET VARIABLES 5
H. INSTRUCTIONS ET BLOCS 5

II. DECLARATION DES DONNEES : VARIABLES ET CONSTANTES 6

A. TYPES DE DONNEES DE BASE 6
1. Les nombres entiers : char, short, int, long, long long 6
2. Les nombres réels : float, double, long double 7
3. Les caractères : char 7
4. Le type logique : _Bool, bool 8
5. Complément : l'opérateur « sizeof » 8
B. IDENTIFICATEURS 8
C. DECLARATION DES VARIABLES 8
D. CONSTANTES 9
E. ENUMERATIONS 9

III. INSTRUCTION D'AFFECTATION, OPERATEURS ET EXPRESSIONS 10

A. VALEURS LITTERALES, OU LITTERAUX 10
B. EXPRESSIONS ET OPERATEURS 10
1. Expressions arithmétiques 10
2. Expressions logiques et opérateurs relationnels 11
3. Expressions logiques et opérateurs logiques 11
C. L'AFFECTATION : OPERATEUR = ... 12
D. LE REFERENCEMENT/ADRESSE : OPERATEUR & 12
E. LE DEREFERENCEMENT/INDIRECTION : OPERATEUR * 13
F. AFFECTATION ET OPERATEURS ARITHMETIQUES 13
1. Pré- et post- incrémentations 13
2. Affectations composées (ou étendues) 13
G. CONSEIL AU SUJET DES VARIABLES 14
H. CONSEIL AU SUJET DES EXPRESSIONS 14
I. TRANSTYPAGE, CHANGEMENT DE TYPE 15

IV. INSTRUCTIONS D'ENTREE-SORTIE : LES FLUX 15

A. ECRIRE DES INFORMATIONS : FLUX DE SORTIE 15
1. Librairie C stdio.h : puts, printf 15
B. LIRE DES INFORMATIONS : FLUX D'ENTREE 16
1. librairie C <stdio.h> : scanf, fscanf, gets, fgets, getchar 16

C. CODES FORMAT PRINTF ET SCANF 18
D. CONSEIL POUR LA SAISIE DES NOMBRES ENTIERS 19
V. CHAINES DE CARACTERES C 19
A. DECLARATION 19
B. LES FONCTIONS DE LA LIBRAIRIE STRING.H 19
VI. ANNEXES 20
A. MACROS : RISQUES 20

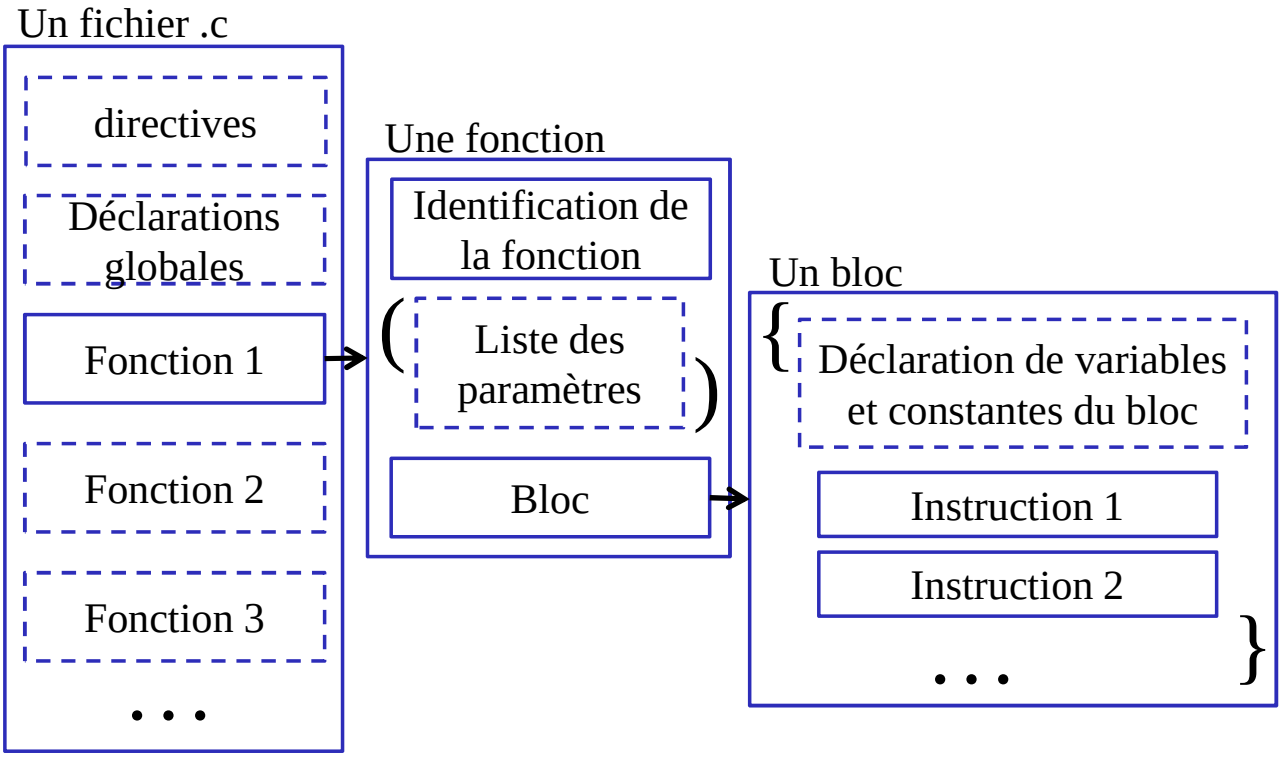
I. Introduction

A. Structure générale

Un programme source C est généralement constitué d'un ensemble de fonctions[1] dont une et une seule porte le nom de *main* (la fonction principale du programme).

Tous les éléments d'un programme C sont mémorisés dans des fichiers au format texte (à l'extension .c pour le programme source, ou .h pour déclarer les entêtes de fonctions ou définir des directives communes) :

- Un *fichier* comporte des directives de compilation, des déclarations de constantes, variables et fonctions, et des définitions de fonctions.
- Une *fonction* comporte un type, un nom, une liste de paramètres, et un bloc.
- Un *bloc* est délimité par des accolades et comporte des déclarations locales à la fonction : constantes ou variables du bloc, des instructions ou d'autres blocs (imbrication de blocs).



Des commentaires peuvent être ajoutés à un fichier, une fonction ou un bloc.

[1] Au moins une fonction

B. Un premier exemple de fichier

La structure générale d'un programme est la suivante :

- Des commentaires d'entête (optionnels, mais fortement conseillés) :

```
/* But du programme : à partir de la saisie de votre âge, ce programme détermine si vous êtes majeur
     Nom : Moi
     Date de création : 04/09/2012
     DONE (REALISE) : date et mise à jour effectuée. . .
     TODO (A FAIRE) : mise à jour à effectuer . . .
*/
```

- Des directives de compilation (librairies à inclure, définition de « macros »)

```
#include <stdio.h>
```

- *Les prototypes de fonctions*

```
(absent ici)
```

- *Les déclarations de constantes et variables globales (possible, mais à éviter)*

```
(absent ici)
```

- La définition de la fonction principale (= traduction de l'algorithme principal)

```
int main()
```

- Début du bloc de la fonction *main*

```
{
```

- Déclarations de constantes et variables

```
 const int AGE_MAJORITE = 18; // une constante
  int age ; // une variable
```

- La suite d'instructions à exécuter pour obtenir le résultat :

```
    printf("donnez votre âge :");
   scanf("%d",&age);
    if (age >= AGE_MAJORITE)
       {
          printf("majeur");
       }
       else
       {
          printf("mineur");
       }
   return 0 ; // 0 = tout s'est bien passé
```

- Fin du bloc de la fonction *main*

```
} // fin main
```

- *La définition d'autres fonctions*

```
  (absent ici)
```

C. Les commentaires en C

Les commentaires sont ajoutées au programme source afin d'apporter des précisions sur certaines parties complexes ou donner des informations générales sur l'objet du programme ou d'une fonction[2].

Syntaxe1 : (multi-lignes)

```
/* . . . un commentaire
 sur plusieurs
 lignes
 . . . */
```

Ce commentaire délimité par les caractères /* et */ peut s'étendre sur plusieurs lignes.

Syntaxe2 : (en ligne) (avec C99)

```
// . . . un commentaire sur une ligne . . .
instruction ; // un commentaire en bout de ligne
```

Ce commentaire s'étend jusqu'au bout de la ligne : le retour à la ligne met fin au commentaire. Il peut être placé n'importe où sur la ligne

Des commentaires situés dans l'entête incluent en général :

- le but du programme,
- Le nom de l'auteur,
- La date de création,
- La journalisation des modifications réalisées et un rappel de celles à apporter.

LES COMMENTAIRES PARTICIPENT A LA QUALITE D'UN PROGRAMME (MAINTENABILITE). IL FAUT CEPENDANT QUE CES COMMENTAIRES SOIENT UTILES !

D. Identifier le programme

Le langage C ne prévoit pas d'identificateur de programme dans le code source. C'est à l'issue de la phase de compilation qu'un nom sera attribué au programme exécutable.

E. Les directives du préprocesseur

Les directives du préprocesseur sont des instructions particulières permettant, entre autre, l'inclusion de fichiers de déclaration de fonctions existantes ou de définitions de macros dans le programme source juste avant la compilation[3].

1. Directives d'inclusion :

Syntaxe :

```
#include <nom_fichier_entete>
#include "nom_fichier_entete"
```

À cet endroit du fichier source, sera inséré (inclus)

[2] Les commentaires sont indispensables lorsqu'on revient plusieurs mois après sur un programme ou bien lorsqu'un autre développeur doit prendre en charge la modification d'un programme.

[3] Les définitions de fonctions existantes permettent au compilateur de valider l'appel de ces fonctions dans le code source du programme.

- le fichier dont le nom est balisé par « < » et « > » ; ce fichier se trouve alors dans une liste de répertoires système ;
- le fichier dont le nom est entre guillemets ; ce fichier se trouve alors dans le même répertoire que le fichier en cours de compilation

Ces fichiers d'entêtes (.h, pour header) contiennent généralement des prototypes[4] de fonctions de bibliothèque standard du C ou de bibliothèques d'extension[5]. Le développeur pourra créer ses propres fichiers d'entêtes pour les fonctions qu'il aura développées.

2. Macros

Les macros permettent de définir des substitutions de valeurs dans le code source avant la phase de compilation.

Syntaxe :

```
#define valeur1 valeur2
```

Toutes les occurrences de « valeur1 » dans le fichier seront remplacées par « valeur2 ».

- Substitution simple :

```
#define PI 3.1415927
```

Toutes les occurrences de « PI » seront remplacées par « 3.1415927 »,

- Substitution avec arguments :

```
#define carre(x) (x*x)
```

Toutes les occurrences de « carre(x) » seront remplacées par « (x*x) » avec substitution des arguments : ainsi, carre(2) sera remplacé par (2*2). (ATTENTION : cf. Annexe 1)

(Plus d'informations sur http://gcc.gnu.org/onlinedocs/gcc-3.4.6/cpp/index.html)

F. Point d'entrée du programme : fonction « main »

L'exécution d'un programme doit débuter à une adresse précise : c'est celle de la fonction « *main* » (« principal » en anglais), le point d'entrée d'un programme C.

Au terme de son exécution, cette fonction renvoie un code numérique (nombre entier) qui signale à l'environnement d'exécution (le système d'exploitation, par exemple) que le traitement s'est bien déroulé (cas où la valeur renvoyée est 0) ou non (cas où la valeur renvoyée est différente de 0). C'est l'objet de l'instruction « return » qui termine l'exécution normale.

G. Déclaration des données – constantes et variables

La déclaration des données peut se trouver à n'importe quel endroit d'un fichier. Cependant, les données doivent être déclarées avant toute utilisation, c'est-à-dire « au dessus » des instructions qui les utilisent. Il est recommandé de déclarer les constantes et variables dans un bloc (tel que défini plus haut). Les constantes partagées entre plusieurs fonctions peuvent éventuellement être déclarées hors de la fonction, au même niveau que les

H. Instructions et blocs

Les instructions sont écrites au sein de blocs et peuvent être :

- Une instruction vide : « ; »

[4] Un prototype d'une fonction reprend les éléments principaux de l'identification d'une fonction ainsi que la liste des paramètres qu'elle attend.

[5] Par exemple la bibliothèque SDL pour utiliser des fonctions graphiques

- Une instruction contenant une action (affectation, appel de fonction) et se terminer par « ; »
- L'instruction d'une structure de contrôle

Des blocs d'instructions peuvent être ouverts n'importe où à l'intérieur du bloc principal, pour isoler des portions de code avec leurs données propres.

II. Déclaration des données : variables et constantes

Le langage C est typé : chaque objet déclaré (variable, constante) doit être associé à un type de donnée afin que le compilateur puisse vérifier la validité des opérations effectuées sur celui-ci.

A. Types de données de base

1. Les nombres entiers : char, short, int, long, long long

Le langage C dispose de plusieurs types d'entiers, chacun utilisant une zone mémoire de taille différente (en nombre d'octets) et permettant de stocker des plages de nombres différentes :

			Taille	Plage de valeurs minimales[6]
signed (défaut)		**char**	1 octet	[-127, +127]
	short	int	2 octets *(au moins 16 bits)*	[-32767,+32767]
		int	4 octets *(au moins 16 bits)*	-2.147.483.647, +2.147.483.647
	long	int	4 octets *(au moins 32 bits)*	Idem.
	long long (C99)	int	8 octets *(au moins 64 bits)*	-9.223.372.036.854.775.807 à +9.223.372.036.854.775.807
unsigned				
		char	1 octet	[0, +255]
	short	int	2 octets *(au moins 16 bits)*	[0,+65535]
		int	4 octets *(au moins 16 bits)*	[0,4.294.967.295]
	long	int	4 octets *(au moins 32 bits)*	Idem.
	long long (C99)	int	8 octets *(au moins 64 bits)*	+18.446.744.073.709.551.615

L'inclusion des fichiers d'entête « limits.h » et «float.h » donne accès à des constantes systèmes donnant les valeurs limites de chacun des types en fonction de l'architecture utilisée :

- CHAR_MIN, CHAR_MAX et UCHAR_MAX pour le type char
- SHRT_MIN, SHRT_MAX et USHRT_MAX pour le type short (abbrev. de short int)
- FLT_MIN, FLT_MAX pour le type float
- Etc.

[6] Les plages de valeurs pour les entiers relatifs dépendent de la manière de gérer le signe : bit de signe et valeur absolue, ou complément à 1 ou complément à 2.

L'espace occupé par les entiers dépend de l'architecture de l'ordinateur, mais respecte toujours la règle : short <= int <= long <= long long, avec au moins :16 bits <= 16 bits <= 32 bits <= 64 bits. L'opérateur « sizeof » permet de connaître la taille correspondant à un type de données :

```
long nb ;
printf("%d",sizeof nb); // affiche la longueur d'un nombre de type long
```

2. Les nombres réels : float, double, long double

Le langage C dispose de 3 types de nombres réels, chacun utilisant une zone mémoire de taille différente (en nombre d'octets) et permettant de stocker des plages de nombres différentes.

Les nombres sont dits en virgule flottante (signe, mantisse et exposant) :

	float	4 octets	32 bits : 1 bit de signe, 8 bits d'exposant, 23 bits de mantisse [1.175494e-038, 3.402823e+038] Simple précision
	double	8 octets	64 bits : 1 bit de signe, 11 bits d'exposant, 52 bits de mantisse [2.225074e-308, 1.797693e+308] Double précision
long	**double**	16 octets	80 bits : 1 bit de signe, 15 bits d'exposant, 64 bits de mantisse Quadruple précision

(La norme IEEE-754 défini la structure des nombres réels en informatique : http://grouper.ieee.org/groups/754/). La capacité des valeurs, la précision et les arrondis peuvent être différentes en fonction de l'association compilateur/architecture matérielle)

LES NOMBRES EN VIRGULE FLOTTANTE PERMETTENT LA REALISATION DE CALCULS SUR DES GRANDS NOMBRES, MAIS AVEC UNE PRECISION LIMITEE.

3. Les caractères : char

Le type « **char** » est stocké sur 1 octet : il est interprété comme caractère en utilisant la table de codification ASCII[7]. Dans sa forme interne, il reste toujours un nombre.

```
char c = 'A' ;
c = (c + 1);
printf("%c",c); // affiche la lettre B à l'écran
```

Un certain nombre de caractères de contrôles sont définis en C sur 2 positions de caractères (le premier étant « \ » , antislash, AltGr-8), mais occupent 1 seul octet en mémoire :

\t	tabulation	
\n	Retour à la ligne	
\0	Caractère nul	Termine les tableaux de caractères
\a	sonnerie	

[7] American Standard Code for Information Interchange : standard de codage de caractères en informatique associant la valeur numérique d'un octet à un caractère. Initialement, l'ASCII définit 128 caractères de base (0 à 127, sur 7 bits, le 1er bit est à 0)

4. Le type logique : _Bool, bool

Le type logique n'était pas à l'origine prévu comme type de donnée ; plusieurs manières de définir ce type absent ont été mises en œuvre par les développeurs :

- un nombre entier : si sa valeur est 0, «false » sinon (à 1 ou différent de 0), comme « true ».
- un type de donnée énumérée, déclarant 2 valeurs, « false » et « true » aux positions 0 et 1.

La norme C99 introduit un type booléen (« **_Bool** »). Un nouveau fichier d'entête « stdbool.h » définit les macros « bool », « true » et « false ».

```
bool estGrand;
estGrand = (taille > 220) ; // affectation
if (estGrand == true) printf("grand personnage !"); // test
if (estGrand) printf("grand personnage !"); // test équivalent
```

5. Complément : l'opérateur « sizeof »

L'opérateur « **sizeof** » permet de connaître le nombre d'octets réservés par un type de données de base *(utilisation de la notation transtypée)* :

```
printf("%d",sizeof (char)); // affiche : 1
printf("%d",sizeof (short)); // affiche : 2
printf("%d",sizeof (int)); // affiche : 4
```

B. Identificateurs

Un identificateur permet l'identification d'un objet unique (fonction, variable, constante) dans un fichier. C'est une suite de caractères de l'ensemble : {a, …,z},{A, …,Z},{0, …,9},{_}. La première lettre ne doit pas être un chiffre.

Un identificateur ne doit pas faire partie de la liste des mots réservés du C (c99) :

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

C. Déclaration des variables

Syntaxe :

```
type_de_donnée ident1 ;
type_de_donnée ident2, ident3, . . . ;
type_de donnée ident4 = valeur_init4;
```

- **type_de_donnée** est l'un des types défini plus haut
- **ident1, ident2, ident3, ident4 :** identificateurs
- **valeur_init4** : valeur initiale de la variable

Exemples :

```
int age ; // entiers
float prime, primeEnFrancs, salaireBase ; // reels (cf. remarque)
int valeurDebut = 100 ; // entier initialisé
char lettre ; // caractère - entier
char prenom[20] ; // chaine, tableau de caractères
```

REMARQUE : IL EST RECOMMANDE DE DECLARER UNE VARIABLE PAR LIGNE DE CODE SOURCE.

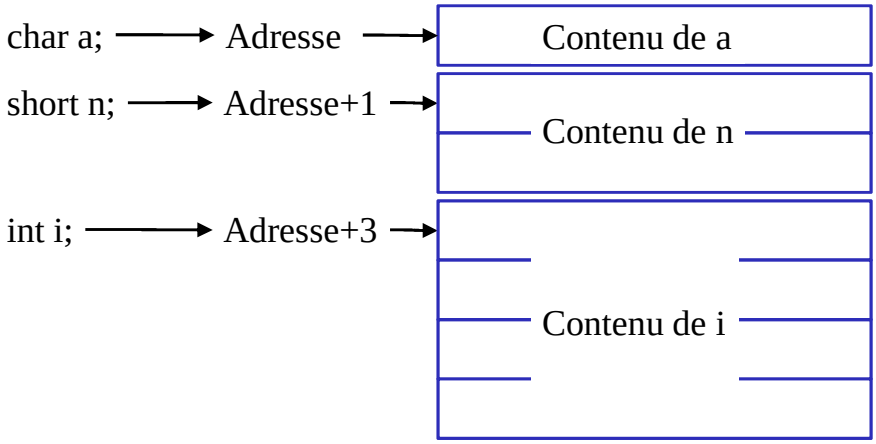


Figure 1 : la déclaration (identificateur et type) réserve un emplacement mémoire (adresse mémoire) qui permettra le stockage du contenu lors de l'exécution

D. Constantes

Le qualificateur 'const' modifie l'accessibilité d'une variable pour ne plus la rendre accessible qu'en lecture seule (*le compilateur vérifie qu'aucune affectation n'est réalisée après la déclaration*)

Syntaxe :

```
const type_de_donnée IDENTIFICATEUR = valeur;
```

- **type_de_donnée** est l'un des types défini ci-dessus
- **identificateur**
  - généralement en lettres MAJUSCULEs (c'est UNE convention)
- **valeur:**
  - valeur que prendra cette constante tout au long de l'exécution

Exemples :

```
const double PI = 3.14 ;
const double TAUX_EURO = 6.55957 ;
const int NOMBRE_DEBUT = 1, NOMBRE_FIN = 10 ;
const char DEBUT_ALPHABET  = 'a' ;
```

E. Enumérations

L'énumération permet la constitution d'une liste de noms associés à des valeurs constantes entières.

Syntaxe :

```
enum ident_enumération{ NOM [= valeur], ...};
```

- **ident_énumération :** Optionnel, identificateur donné à l'énumération
- **NOM** : identifiant unique de la constante
- **valeur:** valeur entière que prendra ce nom ;
  - Par défaut de valeurs, les constantes se voient attribuées une valeur croissante de 1 en 1 à partir de 0.

Exemples :

```
enum logique {FAUX, VRAI} ; // FAUX vaut 0, VRAI vaut 1
enum sexe {HOMME = 1, FEMME = 2} ; // HOMME vaut 1, FEMME vaut 2
// exemple d'utilisation :
```

```
enum logique test ; // déclaration de la variable test
if (test == VRAI) { . . . ;} ;
```

III.Instruction d'affectation, opérateurs et expressions

A. Valeurs littérales, ou littéraux

Une **VALEUR LITTERALE**, ou un **LITTERAL**, désigne une valeur fixe, non identifiée (nombre, caractère, texte, etc.).

Exemple de littéraux : -100, 1, 3.14, 'z', "bonjour"

Un signe – (moins) précède une valeur numérique pour indiquer qu'elle est négative.
Un signe + (plus) peut précéder une valeur numérique pur indiquer qu'elle est positive.

En C, des valeurs littérales peuvent être définies

- **En héxadécimal** : **0x**2AF (débute par 0x, zéro et x)
- **En octal** : **0**127 (débute par 0, zéro)

B. Expressions et opérateurs

Une **EXPRESSION** désigne un calcul associant des valeurs (littéraux, constantes et variables, etc.) et des opérateurs.
L'évaluation de cette expression fournit un résultat d'un certain type de données qui dépend des valeurs et des opérateurs utilisés.

Les principaux opérateurs utilisés sont les suivants :

- Les opérateurs arithmétiques,
- Les opérateurs relationnels et les opérateurs logiques.

LES VALEURS RESULTANT DE L'EVALUATION D'UNE AFFECTATION SONT PERDUES, A MOINS D'ETRE CONSERVEES : C'EST L'OBJET DE L'INSTRUCTION D'AFFECTATION.

1. Expressions arithmétiques

Une **EXPRESSION ARITHMETIQUE** utilise des valeurs numériques (type ENTIER ou REEL) et des opérateurs arithmétiques

Opérateur arithmétique	Signification	Exemple avec des valeurs littérales	Résultat de l'évaluation	
+	Addition	2 + 3	5	
-	Soustraction	2 – 3	-1	
*****	Multiplication	3 * 2	6	
/	Division	3 / 2	1.5	
/	Division entière	3 / 2	1	
%	Modulo, reste de	3 % 2	1	

	la division			

Dans une expression arithmétique, les calculs intermédiaires sont effectués selon la priorité des opérateurs : multiplication, division et modulo, puis addition et soustraction.

Par exemple : 2 + 3 * 5 ➔ effectuer (3 * 5), soit 15, puis ajouter 2 ➔ 17

L'USAGE DES PARENTHESES EST FORTEMENT CONSEILLE POUR DEFINIR PRECISEMENT UN CALCUL.

Par exemple : (2 + 3) * 5 ➔ effectuer (2 + 3), soit 5, puis 5 * 5 ➔ 25

2. Expressions logiques et opérateurs relationnels

Un **EXPRESSION LOGIQUE** est une expression dont le résultat est une valeur logique, soit VRAI ou FAUX.

Elle résulte de la combinaison de valeurs et d'opérateurs de comparaison

Opérateurs de comparaison	Signification	Exemple	Résultat de l'évaluation	
==	Egal	2 == 3	**false**	
!=	Différent	2 != 3	**true**	
<	Inférieur	2 < 3	**true**	
<=	Inférieur ou égal	2 <= 3	**true**	
>	Supérieur	2 > 3	**false**	
>=	Supérieur ou égal	2 >= 3	**false**	

Exemple :

```
(age >= 18) // VRAI si age est >= 18, FAUX sinon
(1 == 1) // toujours VRAI
(1 == 2) // toujours FAUX
```

3. Expressions logiques et opérateurs logiques

Un **EXPRESSION LOGIQUE** peut aussi résulter de la combinaison d'expressions logiques et d'opérateurs logiques.

Les opérateurs logiques **&&** (and), || (or) et **!** (not) permettent la construction d'expressions logiques plus élaborées : les tables de vérités représentent sous forme d'un tableau les valeurs logiques résultant de la combinaison des valeurs des variables d'entrée (A et B, ici)

Expression logique A	Expression logique B	A **ET** B **&&** (and)	A **OU** B || (or)	**NON** A **!** (not)
F	F	F	F	V
F	V	F	V	V
V	F	F	V	F
V	V	V	V	F

Exemples pour A valant (age>=18) et B valant (note >10) :

```
((age >= 18) && (note > 10)) // VRAI si age >= 18 ET note > 10
((age >= 18) || (note > 10)) // VRAI si age >= 18 OU note > 10
```

Attention : cf. Conseil au sujet des expressions

L'algèbre de Boole définit un ensemble d'outils mathématiques pour la simplification des expressions logiques.

C. L'affectation : opérateur =

Le **AFFECTATION** est l'opération qui donne une nouvelle valeur à une variable. La valeur située à droite du « = » (*rvalue, right value*) est placée dans la zone mémoire référencée par le nom de la variable (à gauche du « = ») (*lvalue, left value*)

La nouvelle valeur doit correspondre au type de données de la variable. L'ancienne valeur est bien entendue « écrasée » par la nouvelle.

Syntaxe 1 :

```
ident1 = nouvelle_valeur ;
```

Syntaxe 2 :

```
ident2 = ident3 = . . . = nouvelle_valeur ;
```

- **ident1, ident2, ident3**
  - identificateurs de variables préalablement déclarées
- **nouvelle_valeur**
  - valeur à affecter à la variable : valeur littérale, variable ou constante, expression
  - dans la syntaxe 2, la propagation de l'affectation va de la droite vers la gauche : **une opération d'affectation est ainsi également une expression dont la valeur résulte de la valeur affectée**.

Exemples avec des valeurs littérales :

```
age = 20 ;
prime = 120.5 ;
estMajeur = true ;
lettre = 'a' ;
i = j = k = l = 0 ; // initialisation de plusieurs variables à 0
```

Exemples avec des expressions :

```
age = (20 + 5) ;
prime = (120.5 * (1 + 0.50)) ;
estMajeur = (age > 17) ;
```

D. Le référencement/adresse : opérateur &

L'OPERATEUR DE REFERENCEMENT (**&**) d'une variable retourne l'adresse mémoire de la variable qui suit.

Syntaxe :

```
&ident1
```

```
int age;
printf("l'adresse de age est %p",&age) ;
```

E. Le déréférencement/indirection : opérateur *

L'OPERATEUR DE DEREFERENCEMENT (*) d'une variable de type pointeur retourne la valeur située à l'adresse mémoire pointée.

Syntaxe :

```
*ptr1
```

```
int age = 18;
int * ptr1 ; // déclaration d'une variable pointeur vers int
ptr1 = &age ; // ptr1 recoit l'adresse de age, il pointe vers age
printf("age vaut %d ou %d",age, *ptr1) ;
```

F. Affectation et opérateurs arithmétiques

Le langage C propose des opérateurs permettant de simplifier l'écriture de certaines opérations.

1. Pré- et post- incrémentations

Simplifier l'écriture des opérations d'incrémentation et de décrémentation :

Opérateur arithmétique	Signification	Exemple avec des valeurs littérales	Equivalent à	
++	Incrémentation	i++	i = i + 1	Post incrémentation
		++i	i = i + 1	pré incrémentation
--	Décrémentation	i--	i = i - 1	Post décrémentation
		--i	i = i – 1	pré décrémentation

L'utilisation de ces opérateurs est très intéressante :

```
for (i=1 ;i<=10 ;i++) {;} // boucle 10 fois . . . pour rien . . .
```

ATTENTION : L'UTILISATION DANS DES EXPRESSIONS EST DELICATE (ET DECONSEILLEE) !!!!

EN EFFET

- **EN POST-, LA VALEUR EST AFFECTEE PUIS L'OPERATION ARITHMETIQUE EST REALISEE**
- **EN PRE-, L'OPERATION ARITHMETIQUE EST REALISEE PUIS LA VALEUR EST AFFECTEE**

Ainsi,

```
int i = 10 ;
- printf("%d",i++) ; // affichera 10, puis i sera incrémenté de 1
- printf("%d",++i) ; // i sera incrémenté de 1, puis affichera 1
```

2. Affectations composées (ou étendues)

Combiner l'affectation avec d'autres opérateurs :

Opérateur arithmétique	Signification	Exemple avec des valeurs littérales	Equivalent à	
+=	Addition et affectation	i+=5	i = i + 5	

-=	soustraction et affectation	i-=5	i = i - 5	
***=**	multiplication et affectation	i*=5	i= i + 5	
/=	Division et affectation	i/=3	i=i/3	

G. Conseil au sujet des variables

L'INITIALISATION D'UNE VARIABLE consiste à lui affecter une valeur initiale en fonction de son type de données.

TOUTE VARIABLE DOIT SUBIR UNE AFFECTATION AVANT D'ÊTRE UTILISEE SINON SON CONTENU EST INDETERMINE.

H. Conseil au sujet des expressions

UTILISEZ DES PARENTHESES POUR DEFINIR CLAIREMENT LA PRIORITE DES OPERATIONS A EFFECTUER.

EVITER D'UTILISER LE CARACTERES ASSOCIATIF DES OPERATEURS DE COMPARAISON

Exemple à rejeter :

```
int a = 2;
int b = 2;
int c = 2;
if ( a < b < c ) ... ;// retourne VRAI, FAUX est plutôt attendu
if ( a == b == c ) ... ;// retourne FAUX, VRAI est attendu
```

Exemple à utiliser :

```
int a = 2;
int b = 2;
int c = 2;
if ( a < b && b < c ) ... ;  // retourne FAUX : OK
if ( a == b && b == c ) ... ; // retourne VRAI : OK
```

EVITER LES OPERATEURS POUVANT PRODUIRE DES EFFETS DE BORD DANS LES EXPRESSIONS DE COMPARAISON APRES LE PREMIER AND ou OR (évalué seulement si la première condition est suffisante pour évaluer l'expression)

Exemple à rejeter :

```
int i;
int max;
// ...
if ( (i >= 0 && (++i) <= max) ) {
// ...
}
```

Exemple à utiliser :

```
int i;
int max;
// ...
if ( (i >= 0 && (i + 1) <= max) ) {
  i++;
// ...
}
```

I. Transtypage, changement de type

Le **TRANSTYPAGE** consiste à demander explicitement la conversion de la valeur d'une expression en un certain type donné après son évaluation.

Syntaxe (en C):

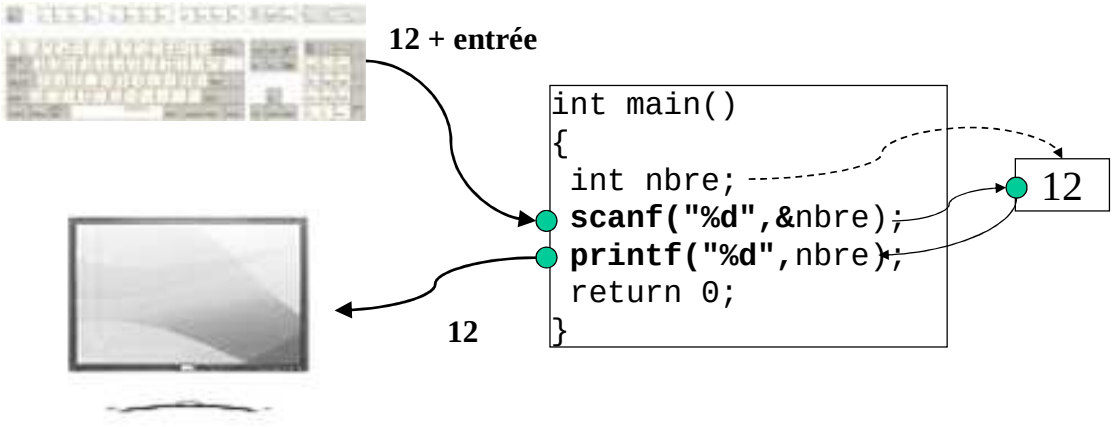
```
int dividende ;
int diviseur ;
double resultat ;
resultat = ((double)dividende) / diviseur ;
```

Des changements de type interviennent automatiquement lors des opérations de calcul (on parle de « promotion de type » afin que les calculs intermédiaires puissent être exécutés sans perte de précision). Il est parfois nécessaire de définir explicitement le transtypage afin d'éviter certaines anomalies de calcul (pertes de précision).

IV. Instructions d'Entrée-sortie : les flux

Un flux représente un flot de données

- en provenance d'une source de données (clavier, par exemple),
- ou à destination d'une autre partie du système (écran, par exemple) .



A. Ecrire des informations : flux de sortie

1. Librairie C stdio.h : puts, printf

Prototypes des fonctions :

- int puts(const char *astring);

- int fputs (const char * str, FILE * stream);

puts(chaine) ; // écrit la chaine (sans \0) et un CRLF sur **stdout**
fputs(chaine, **stdout**) ; // écrit la chaine (sans \0) et un CRLF sur **stdout**

- **valeur1, valeur2, valeur3** représentent
  - les valeurs à envoyer sur le flux de sortie : littéraux (textes et nombres), variables, constantes

Prototypes des fonctions :

- int printf(const char* format, ...);
- int fprintf (FILE * stream, const char * format, ...);

printf(chaine de formatage, variable1, variable2, variable3, . . .) ;
fprintf(**stdout**, format, variable1, variable2, variable3, . . .) ;

- **la chaine de formatage** représente le texte à envoyer vers le flux de sortie
  - une valeur littérale chaine de caractères
  - des marques de formatage indiquant le type et la position des valeurs à fusionner avant l'envoi
- **valeur1, valeur2, valeur3** représentent
  - les valeurs à fusionner avec la chaine de format sur le flux de sortie : littéraux (textes et nombres), variables, constantes

	Type de données à afficher	**format**
Nombres	Entier signé	%d %i
	Entier non signé	%u
	Entier long signé et non signé	%ld , %lu
	Réels	%f, %4.6f, %lf %e, %g, %E, %G
Caractère	Isolé	%c
	Chaînes de caractères	%s

Cf. tableau des codes formats : ci-dessous

Exemple : affichage d'un nombre et d'une chaine de caractères

```
printf("la valeur du nombre entier est %d", nombre) ;
printf("\nVous vous appelez %s", "tom") ;
```

B. Lire des informations : flux d'entrée

1. librairie C <stdio.h> : scanf, fscanf, gets, fgets, getchar

Prototypes des fonctions :

- int scanf (const char * format, ...);
- int fscanf (FILE * stream, const char * format, ...);

scanf(format, ref_variable1, ref_variable2, ref_variable3, . . .) ;
fscanf(**stdin**, format, ref_variable1, ref_variable2, ref_variable3, . . .) ;

- **ref_variable1, ref_variable2, ref_variable3** représentent
  - des références vers les variables variable1, variable2, variable3

- **format** représente la chaîne de format, c'est-à-dire les caractères de formatage des variables à saisir (%d pour un entier, %u pour un entier non signé, %f pour un réel)

Exemple :

```
scanf("%d", &nombre) ;
char nom[12] ;
scanf("%s", nom) ; // nom est un tableau : l'adresse pointe au début
```

Attention :

- la lecture s'arrête au premier caractère espace (à prendre en compte surtout pour les chaines de caractères)
- La **lecture d'une valeur supérieure à la taille du tableau cause une erreur de débordement** (buffer overflow)

http://www.cplusplus.com/reference/clibrary/cstdio/scanf/

Prototypes des fonctions :

- char * gets (char * str);
- char * fgets (char * str, int num, FILE * stream);

gets(ref_variable1) ; // lecture à partir de stdin
fgets (ref_variable1, longueur, STDIN) ;

- **ref_variable1** : référence vers la variable réceptrice
- **longueur :** longueur de la chaine (récupération d'un caractère en moins, pour la réservation du caractère de fin de chaine \0)
- **flux : stdin :** représente le flux d'entrée clavier

Exemple : déclaration d'une chaine, lecture et affectation des 12 premiers caractères

```
char nom[12] ;
fgets(nom, 12, stdin) ;
```

Attention :

- la lecture conserve le caractère « entrée » (\n) qu'il faudra éliminer du texte (localiser par 'strchr' et remplacer par \0)

```
/* exemple de code pour nettoyer ce qui a été lu au clavier …*/
void viderBuffer()
{
    int c = 0;
    while (c != '\n' && c != EOF)
    {
        c = getchar();
    }
}

int lire(char *chaine, int longueur)
{
    char *positionEntree = NULL;

    if (fgets(chaine, longueur, stdin) != NULL)
    {
        positionEntree = strchr(chaine, '\n');
        if (positionEntree != NULL)
```

```
        {
            *positionEntree = '\0';
        }
        else
        {
            viderBuffer();
        }
        return 1;
    }
    else
    {
        viderBuffer();
        return 0;
    }
}
```

Syntaxe getchar :renvoie le caractère lu au clavier

getchar() ;

Exemple : attend la lecture de la lettre ‘a’ (demande la saisie d’un caractère tant qu’il est différent de ‘a’)

```
char ch;
do
{
  ch = getchar();
} while (ch!='a');
```

C. Codes format printf et scanf

`%[flag][largeur][.precision][modificateur]type = affichage`
`%[largeur][modificateur]type = saisie`

	flag	largeur	*Précision*	modificateur	type
entier	- + *espace*	nombre minimal de caractères à écrire * = variable, arg1	*nombre minimal de chiffres* ** = variable, arg2*	h :short l : long	d u
réel	- + *espace*	nombre minimal de caractères à écrire * = variable, arg1	*nombre de chiffres après la virgule (réels)* ** = variable, arg2*	l : double L : long double	f e
caractère					c
chaîne de caractères		Nombre de caractères maxi (longueur – 1)			s

D. Conseil pour la saisie des nombres entiers

TOUTE DONNEE PROVENANT D'UNE SOURCE EXTERIEURE (SAISIE, FICHIER, ETC.) DOIT ETRE CONTROLEE AFIN DE LIMITER LES POSSIBLES EFFETS DE BORD

1. **IDENTIFIER TOUTES LES SOURCES POSSIBLES D'ACQUISITION DE DONNEES ;**
2. **IDENTIFIER LES POINTS D'ACQUISITION DE DONNEES DANS LE CODE SOURCE ;**
3. **DEFINIR LES CRITERES POUR LES DONNEES SOIENT VALIDES (NOMBRES, CHAINES DE CARACTERES, DATES, ETC.) ;**
4. **DEFINIR LE COMPORTEMENT DU PROGRAMME SI UNE DONNEE ERRONEE EST REÇUE ;**
5. **METTRE EN ŒUVRE LE CODE SOURCE POUR CONTROLER LES DONNEES.**

La récupération d'informations d'un flux dans le but de récupérer une valeur entière présente un risque : si la valeur saisie n'est pas entière ? Cela produit une erreur à l'exécution.

Exemple :

```
int age ;
scanf("%d",& age); // erreur si la saisie n'est pas un entier
```

La solution consiste en la récupération d'une chaîne de caractère et sa conversion en un entier après avoir effectué quelques vérifications (cela peut faire l'objet d'une fonction cf. chapitre 5).

V. Chaînes de caractères C

La forme C standard de traitement des chaines de caractères est le tableau de caractères (cf. chapitre 4).

Une chaîne de caractères est une suite de caractères contigus en mémoire et dont le dernier caractère marque la fin : c'est le caractère NUL, '\0'

L'utilisation de fonctions de manipulation des chaines de caractères nécessite l'inclusion d'un fichier d'entêtes de fonctions :

```
#include <string.h>
```

A. Déclaration

Plusieurs formes de déclaration sont possibles :

```
char nom[24] ; // nom : longueur de 24 caractères (23 + \0)
char nom[] = "Jean Paul" ; // nom :  10 octets (9 + '\0')
char nom[] = {'J','e','a','n',' ','P','a','u','l'} ; // idem.
```

B. Les fonctions de la librairie string.h

- la fonction **strcat**(Vstr1, Vstr2) : ajoute une chaine Vstr1 au bout de la chaine Vstr1 (attention au débordement : cf. chapitre 4 sur les tableaux)

```
printf("avant = %s",str1);
strcat(str1, str2) ; // ajoute Vstr2 à la fin de Vstr1
printf("apres = %s",str1);
```

- la fonction **strcmp**() : permet la comparaison de 2 chaînes

```
char ch1[] = "Pierre", ch2[] = "Paul" ;
if (strcmp(ch1, ch2) != 0) printf("différent");
```

```
if (strcmp(str1, str2)==0) printf("egal\n");
```

```
            else printf("différent\n");
```

- la fonction **strcpy**() : AFFECTATION : permet la copie d'une chaîne vers une autre

```
printf("avant = %s", str1);
strcpy(str1, str2) ; // copie str2 dans str1
printf("apres = %s", str1) ;
strcpy(str1, "bonjour") ; // affecte (copie) 'bonjour' à str1
```

VI. Annexes

A. Macros : risques

Les macros sont d'un usage très délicat, il est indispensable de s'assurer des effets de bord pouvant être produit par leur définition. Ceux-ci sont parfois difficiles à trouver…

Ainsi la simple macro « carre » :

```
#define carre(x)     x*x
```

2/carre(10) → 2/10*10 → 2 alors qu'on attend 0.02

```
#define carre(x)     (x*x)
```

carre(1+1) → (1+1*1+1) → 3 alors qu'on attend 4

```
#define carre (x)   ((x)*(x))
```

mais pour x valant 2 :

carre(++x) → ((++x)*(++x)) → 16, alors qu'on attend 9

Sur ce dernier exemple, se souvenir également que l'utilisation des opérateurs d'incrémentation (++ et --) sont à proscrire dans des expressions (calculs, comparaisons, appels de fonctions, etc.)