CHAPITRE 1 : LANGAGE C (LES ÉLÉMENTS DE BASE)

I. Présentation du langage C :

I-1. Historique :

Le C a été conçu en 1972 par deux ingénieurs des laboratoires Bell, Denis Richie et B. W. Kernighan afin d'obtenir un système d'exploitation de type UNIX, mais portable sur un grand nombre d'architectures. Il est inspiré des langages B et BCPL.

En 1978, Brian Kernighan et Dennis Richie publient la définition classique du C dans le livre « *The C Programming language* ». Le C devenant de plus en plus populaire dans les années 80, plusieurs groupes mirent sur le marché des compilateurs comportant des extensions particulières.

En 1983, l'ANSI (American National Standards Institute) décida de normaliser le langage ; ce travail s'acheva en 1989 par la définition de la norme ANSI C. Celle-ci fut reprise telle quelle par l'ISO (International Standards Organization) en 1990. C'est ce standard, ANSI C, qui est décrit dans le présent document.

I-2. Les points forts du C:

Les principaux atouts du C sont :

- *Polyvalent :* Il permet le développement de systèmes d'exploitation, de programmes scientifiques et de programmes de gestion.
- *La portabilité :* un programme C-ANSI peut être compilé sur n'importe quelle système d'exploitation disposant d'un compilateur C (Linux, *BSD, *NIX, Windows, MAC OS, z/OS, ...)
- *La rapidité :* par nature le langage laisse une grande liberté au programmeur, mais lui donne aussi beaucoup de responsabilités. La validité des accès mémoire n'est par exemple pas vérifiée. En contre partie, les programmes générés sont compacts et performants.
- il permet un contrôle, pratiquement au niveau du langage machine (assembleur), mais aussi de niveau plus élevé, par l'intermédiaire de bibliothèques additionnelles
- il est dit *faiblement typé* ; c'est à dire qu'il y a peu de types de données utilisables par le compilateur, mais ils sont très proches de leur représentation matérielle au niveau du processeur.
- *Père syntaxique* du C++, Java, PHP etc.

I-3. Les points faibles du C:

Comme nous l'avons vu plus haut, le compilateur du langage C ne prétend pas prendre la place du programmeur. Il faut donc faire preuve de rigueur, lors de la conception des programmes, pour ne pas créer d'accès mémoire illégaux, etc...

De plus, la liberté laissée par la norme permet de réaliser des programmes simples et performants, mais aussi de créer du code source mal structuré, rendant sa relecture et son débogage impossibles. C'est pourquoi, la règle d'or de tout programmeur (débutant ou non) est d'utiliser lorsque c'est possible des noms de variables explicites et significatifs et d'utiliser des commentaires, pour expliquer ce que fait telle fonction, ou telle instruction.

II. La structure d'un programme en C:

Chaque fichier source entrant dans la composition d'un programme exécutable est fait d'une succession d'un nombre quelconque d'éléments indépendants, qui sont :

- Des directives pour le préprocesseur (lignes commençant par **#**),
- Des constructions de types (***struct***, ***union***, ***enum***, ***typedef***),
- Des déclarations de variables et de fonctions externes,
- Des définitions de variables et
- Des définitions de fonctions.

Seules les expressions des deux dernières catégories font grossir le fichier objet : les définitions de fonctions laissent leur traduction en langage machine, tandis que les définitions de variables se traduisent par des réservations d'espace, éventuellement garni de valeurs initiales. Les autres directives et déclarations s'adressent au compilateur et il n'en reste pas de trace lorsque la compilation est finie

En **C** on n'a donc pas une structure syntaxique englobant tout, comme la construction « Algorithme …Début … Fin. » du langage algorithmique ; un programme n'est qu'une collection de fonctions assortie d'un ensemble de variables globales. D'ou la question : par où l'exécution doit-elle commencer ? La règle généralement suivie par l'éditeur de liens est la suivante: parmi les fonctions données il doit en exister une dont le nom est ***main***. C'est par elle que l'exécution commencera ; le lancement du programme équivaut à l'appel de cette fonction par le système d'exploitation. Notez bien que, à part cela, ***main*** est une fonction comme les autres, sans aucune autre propriété spécifique ; en particulier, les variables internes à ***main*** sont locales, tout comme celles des autres fonctions

Voici la version C du célèbre programme qui affiche bonjour. :

```
#include<stdio.h>
main() {
printf("Bonjour\n");
return 0;
      }
```

La fonction ***main*** est la fonction principale des programmes en C: Elle se trouve obligatoirement dans tous les programmes. L'exécution d'un programme entraîne automatiquement l'appel de la fonction ***main.***

Résultat de main :

- En principe tout programme devrait retourner une valeur comme code d'erreur à son environnement. Par conséquent, le type résultat de ***main*** est toujours ***int***. En général, le type de main n'est pas déclaré explicitement, puisque c'est le type par défaut. Nous allons terminer nos programmes par l'instruction:

```
return 0;
```

qui indique à l'environnement que le programme s'est terminé avec succès, sans anomalies ou erreurs fatales.

Paramètres de main :

- Si la liste des paramètres de la fonction ***main*** est vide, il est d'usage de la déclarer par **()**.
- Si nous utilisons des fonctions prédéfinies (par exemple: ***printf***), il faut faire précéder la définition de ***main*** par les instructions ***#include <stdio.h>*** correspondantes.

Remarque : toute instruction se termine par **;**

III. Types, constantes et variables :

III-1. Types de données:

Lorsqu'on souhaite utiliser une donnée en C, il est nécessaire de lui donner un type, qui permet au compilateur de connaître l'occupation mémoire de la donnée, ainsi que son mode de représentation binaire (entier signé ou non, nombre à virgule flottante ont un codage propre).
Il existe deux types de données en C-ANSI, les données scalaires et les pointeurs.
Voici la liste exhaustive des différentes données scalaires en C :

Type	Signification	Taille	Plage de variation
char	caractère	1	-2^7 à 2^7-1
unsigned char	caractère non signé	1	0 à 2^8-1
short [int]	entier court	2	-2^8 à 2^8-1
unsigned short [int]	entier court non signé	2	0 à 2^{16}
int	entier	4	-2^{31} à $2^{31}-1$
unsigned [int]	entier non signé	4	0 à $2^{32}-1$
long [int]	entier long	4	-2^{-31} à $2^{31}-1$
unsigned long [int]	entier long non signé	4	0 à 2^{32}
float	réel	4	3.4×10^{-38} à 3.4×10^{38}
double	réel double précision	8	1.7×10^{-308} à 1.7×10^{308}
long double	réel long double précision	10	3.4×10^{-4932} à 3.4×10^{4932}

a. Les entiers :

Les entiers connaissent plusieurs modes de représentation en C. Les trois déclarations suivantes sont équivalentes :

- la représentation décimale habituelle est bien sûr autorisée : **int** a=30;
- en hexadécimal en faisant précéder le nombre de **0x** : int a=**0x**1E;
- en octal en faisant commencer le nombre par **0** : int a=**0**46;

b. Les réels :

Les réels sont représentés sous deux formes principalement :

- la forme pleine habituelle **float** b=123456.789;
- la forme exponentielle **float** b=12.3456789e4;

c. Les caractères :

Le type **char** est très proche des types entier, puisqu'en fait on y stocke le code ASCII du caractère que l'on souhaite enregistrer. Par exemple, les deux lignes suivantes sont équivalentes :

- **char** caractere = 'B';
- **char** caractere = 66;

Les chaînes de caractères n'existent pas comme type prédéfini en C. On peut néanmoins créer des chaînes en réalisant des tableaux de caractères se terminant par le caractère nul **'\0'**.

III-2. Les constantes :

Une constante est une valeur qui apparaît littéralement dans le code source d'un programme, le type de la constante étant déterminé par la façon dont la constante est écrite. Les constantes peuvent être de 4 types : entier, flottant (nombre réel), caractère, énumération. Ces constantes vont être utilisées, par exemple, pour initialiser une variable.

a. Les constantes entières :

Syntaxe :

On dispose de 3 notations pour les constantes entières : décimale, octale et hexadécimale. Les constantes décimales s'écrivent de la manière usuelle (ex : 372). Les constantes octales doivent commencer par un zéro et ne comporter que des chiffres octaux (ex : 0477). Les constantes hexadécimales doivent commencer par **0x** ou **0X** et être composées des chiffres de 0 à 9, ainsi que des lettres de **a** à **f** sous leur forme majuscule ou minuscule (ex : **0x**5a2b, **0X**5a2b, **0x**5A2B).

Une constante entière peut être suffixée par la lettre **u** ou **U** pour indiquer qu'elle doit être interprétée comme étant non signée. Elle peut également être suffixée par la lettre **l** ou **L** pour lui donner l'attribut de précision long.

Exemple :

constante	type
1234	int
02322	int /* octal */
0x4D2	int /* hexadécimal */
123456789L	long
1234U	unsigned int
123456789UL	unsigned long int

b. Les constantes réelles :

Syntaxe :

Les constantes réelles sont représentées par la notation classique par mantisse et exposant. L'exposant est introduit par la lettre e ou E ; il s'agit d'un nombre décimal éventuellement signé. Par défaut, une constante réelle est représentée avec le format du type double. On peut cependant influer sur la représentation interne de la constante en lui ajoutant un des suffixes **f** (indifféremment **F**) ou **l** (indifféremment **L**). Les suffixes **f** et **F** forcent la représentation de la constante sous forme d'un **float**, les suffixes **l** et **L** forcent la représentation sous forme d'un long double.

Exemple :

constante	type
12.34	double
12.3e-4	double
12.34F	float
12.34L	long double

c. Les constantes caractères :

Syntaxe :

Pour désigner un caractère imprimable, il suffit de le mettre entre apostrophes (par ex. 'A' ou '$'). Les seuls caractères imprimables qu'on ne peut pas représenter de cette façon sont l'antislash et l'apostrophe, qui sont respectivement désignés par \\ et \'. Le point d'interrogation et les guillemets peuvent aussi être désignés par les notations \? et \". Les caractères non imprimables peuvent être désignés par '\code-octal' où code-octal est le code en octal du caractère. On peut aussi écrire '\xcode-hexa' où code-hexa est le code en hexadécimal du caractère. Par exemple, '\33' et '\x1b' désignent le caractère escape. Toutefois, les caractères non-imprimables les plus fréquents disposent aussi d'une notation plus simple :

\n	nouvelle ligne	\r	retour chariot
\t	tabulation horizontale	\f	saut de page
\v	tabulation verticale	\a	signal d'alerte
\b	retour arrière		

d. Les constantes chaînes de caractères :

Une chaîne de caractères est une suite de caractères entourés par des guillemets. Par exemple :
"Ceci est une chaîne de caractères"

Une chaîne de caractères peut contenir des caractères non imprimables, désignés par les représentations vues précédemment. Par exemple :
"ligne 1 \n ligne 2"

A l'intérieur d'une chaîne de caractères, le caractère " doit être désigné par \". Enfin, le caractère \ suivi d'un passage à la ligne est ignoré. Cela permet de faire tenir de longues chaînes de caractères sur plusieurs lignes. Par exemple :
*"ceci est une longue longue longue longue longue longue longue longue *
chaîne de caractères"

III-3. Déclaration de variables :

Un programme doit pouvoir mémoriser et manipuler les données utiles au déroulement de ses traitements (entrées, sorties et données intermédiaires).

La notion de ***variable*** consiste à réserver un emplacement de la mémoire destiné à recevoir une donnée et à lui donner un nom. Cet emplacement mémoire peut alors recevoir une donnée.

Nous avons vu que toute information doit être codée et que plusieurs codages sont utilisés dans un ordinateur : Il sera donc nécessaire de préciser la nature de chaque variable en lui associant un certain ***type***.

Le type d'une variable permet de définir le codage interne utilisé pour représenter la valeur contenu dans cette variable mais aussi la taille de l'emplacement mémoire associé à cette variable et les opérations que l'on pourra réaliser sur ces variables.

Définir une variable consiste donc à lui donner un nom et un type. Le compilateur se charge alors de réserver un emplacement de la mémoire et de lui associer le nom. Il est alors possible d'utiliser cet emplacement en précisant simplement son nom.

Syntaxe :

```
Type_variable   Nom_variable =valeur_initiale;
```

Exemple :

- **int** nombre = 12;
- **char** caractere ;
- **float** somme;
- **float** x=3.17;

Il est possible de déclarer plusieurs variables de même type sur une seule ligne, tout en initialisant certaines d'entre elles. La déclaration suivante définit les variables i, j, k comme des flottantes double précision, et fixe i à la valeur 0.0123456, k à la valeur −123.456789 et laisse j sans valeur initiale :

- **double float** i=0.0123456, j, k=-123.456789;

Remarque importante :

Le langage C étant sensible à la casse **:** Dans le langage C les minuscules sont différenciées des majuscules. Par exemple **int** ne veut pas dire la même chose que **Int** ou **INT**.

III-4. Choix du nom des variables :

Les noms de variables sont des identificateurs qui doivent répondre à quelques critères :

- Commencer par une lettre, ou par _,
- Ne comporter que des lettres, des chiffres, ou le caractère _,
- Ne pas être un nom réservé (Les mots-clefs).
- Le premier caractère d'un identificateur de ne peut pas être un chiffre. Par exemple, var1, tab_23 ou _deb sont des identificateurs valides ; par contre, 1i et i:j ne le sont pas. Il est cependant déconseillé d'utiliser _ comme premier caractère d'un identificateur car il est souvent employé pour définir les variables globales de l'environnement C.
- Les majuscules et minuscules sont différenciées.
- Le compilateur peut tronquer les identificateurs au-delà d'une certaine longueur. Cette limite dépend des implémentations, mais elle est toujours supérieure à 31 caractères. (Le standard dit que les identificateurs externes, c'est-à-dire ceux qui sont exportés à l'édition de lien, peuvent être tronqués à 6 caractères, mais tous les compilateurs modernes distinguent au moins 31 caractères).

III-5. Les mots-clefs :

Un certain nombre de mots, appelés mots-clefs, sont réservés pour le langage lui-même et ne peuvent pas être utilisés comme identificateurs. L'ANSI C compte 32 mots clefs :

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

Que l'on peut ranger en catégories :

- *type des données :*
  `char const double float int long short signed unsigned void volatile`
- *classes d'allocation :*
  `auto extern register static`
- *constructeurs :*
  `enum struct typedef union`

instructions de boucle :
`do for while`

- *sélections :*
  `case default else if switch`
- *ruptures de séquence :*
  `break continue goto return`
- *divers :*
  `asm entry fortran sizeof`

III-6. Initialisation des variables :

L'initialisation se définit comme l'affectation d'une valeur lors de la définition de la variable. Toute modification de valeur d'une variable postérieure à sa définition n'est pas une initialisation mais une affectation.

Il est possible de réaliser des initialisations selon trois techniques suivant le type de la variable :

- dans le cas d'une variable simple, il suffit de faire suivre la définition du signe égal (=) et de la valeur que l'on veut voir attribuée à la variable.
- dans le cas de tableaux ou structures, il faut faire suivre la définition du signe égal suivi d'une accolade ouvrante et de la série de valeurs terminée par une accolade fermante. Les valeurs sont séparées par des virgules. Elles doivent correspondre aux éléments du tableau ou de la structure. La norme préconise que lorsque le nombre de valeurs d'initialisation est inférieur au nombre d'éléments à initialiser, les derniers éléments soient initialisés avec la valeur nulle correspondant à leur type.
- les tableaux de caractères peuvent être initialisés à partir d'une chaîne de caractères. Les caractères de la chaîne sont considérés comme constants.

Exemple :

`int i = 10 ;`	Entier i initialisé à 10
`int j = 12, k = 3, l ;`	entiers j initialisé à 12 k initialisé à 3 et l non initialisé.
`char d = '\n' ;`	Caractère initialisé à la valeur du retour chariot.

III-7. Déclaration de constantes :

Syntaxe :

const Type_constante Nom_constante =valeur;

Exemple :

- **const** float pi = 3.14;

IV. Les opérateurs les plus usuels :

IV-1. L'affectation :

En C, l'affectation est un opérateur à part entière. Elle est symbolisée par le signe =. Sa syntaxe est la suivante :

Nom_variable = expression

Le terme de gauche de l'affectation peut être une variable simple, un élément de tableau mais pas une constante. Cette expression a pour effet d'évaluer *expression* et d'affecter la valeur obtenue à *Nom_variable*. De plus, cette expression possède une valeur, qui est celle *expression*.

Exemple :

A = 10 ;
RES = 2*A+10 ;
La première instruction demande de placer la valeur 10 dans la variable A (dans l'emplacement mémoire ayant pour nom A)
La seconde demande de calculer l'expression 2* A+10 et de placer le résultat dans la variable RES

L'affectation effectue une conversion de type implicite : la valeur de ***l'expression*** (terme de droite) est convertie dans le type du terme de gauche. Par exemple, le programme suivant

```
main()
{
int i, j = 2;
float x = 2.5;
i = j + x;
x = x + i;
printf("\n %f \n",x);
}
```

imprime pour x la valeur 6.5 (et non 7), car dans l'instruction i = j + x;, l'expression j + x a été convertie en entier.

IV-2. Conversions de type :

La grande souplesse du langage C permet de mélanger des données de différents types dans une expression. Avant de pouvoir calculer, les données doivent être converties dans un même type. La plupart de ces conversions se passent automatiquement, sans l'intervention du programmeur, qui doit quand même prévoir leur effet. Parfois il est nécessaire de convertir une donnée dans un type différent de celui que choisirait la conversion automatique; dans ce cas, nous devons forcer la conversion à l'aide d'un opérateur spécial ("***cast***").

a. Les conversions de type automatiques :

Si un opérateur a des opérandes de différents types, les valeurs des opérandes sont converties automatiquement dans un type commun.

Ces manipulations implicites convertissent en général des types plus 'petits' en des types plus 'larges'; de cette façon on ne perd pas en précision.

Lors d'une affectation, la donnée à droite du signe d'égalité est convertie dans le type à gauche du signe d'égalité. Dans ce cas, il peut y avoir perte de précision si le type de la destination est plus faible que celui de la source.

Règles de conversion automatique :

Conversions automatiques lors d'une opération avec :

- deux entiers:
  D'abord, les types char et short sont convertis en int. Ensuite, l'ordinateur choisit le plus large des deux types dans l'échelle suivante: int, unsigned int, long, unsigned long
- un entier et un rationnel:
  Le type entier est converti dans le type du rationnel.
- deux rationnels:
  L'ordinateur choisit le plus large des deux types selon l'échelle suivante:
  float, double, long double
- affectations et opérateurs d'affectation:
  Lors d'une affectation, le résultat est toujours converti dans le type de la destination. Si ce type est plus faible, il peut y avoir une perte de précision.

Exemple :

Observons les conversions nécessaires lors d'une simple division:
int X;
float A=12.48;
char B=4;
X=A/B;
B est converti en **float** (règle 2). Le résultat de la division est du type **float** (valeur 3.12) et sera converti en **int** avant d'être affecté à *X* (règle 4), ce qui conduit au résultat *X=3*

b. Les conversions de type forcées (casting) :

Il est possible de convertir explicitement une valeur en un type quelconque en forçant la transformation à l'aide de la syntaxe:

(type) objet

Exemple 1:

```
main()
{
int i = 3 , j = 2;
printf("%f \n", (float) i/j);
}
Renvoie la valeur 1.5
```

Exemple 2:

char A=3;
int B=4;
float C;
C = (float)A/B;
La valeur de *A* est explicitement convertie en **float**. La valeur de *B* est automatiquement convertie en **float** (règle 2). Le résultat de la division (type rationnel, valeur 0.75) est affecté à *C*. Résultat: *C=0.75*

IV-3. Les opérateurs arithmétiques :

Les opérateurs arithmétiques classiques sont l'opérateur unaire - (changement de signe) ainsi que les opérateurs binaires

- + addition
- - soustraction
- * multiplication
- / division
- % reste de la division (modulo)

Ces opérateurs agissent de la façon attendue sur les entiers comme sur les flottants. Leurs seules spécificités sont les suivantes :

- L'opérateur **%** ne s'applique qu'à des opérandes de type entier. Si l'un des deux opérandes est négatif, le signe du reste dépend de l'implémentation, mais il est en général le même que celui du dividende.
- Il n'y a pas en C d'opérateur effectuant l'élévation à la puissance. De façon générale, il faut utiliser la fonction **pow(x,y)** de la librairie **math.h** pour calculer x^y.

IV-4. Les opérateurs de comparaisons:

- > strictement supérieur
- >= supérieur ou égal
- < Strictement inférieur
- <= inférieur ou égal
- == égal
- != différent

Leur syntaxe est :

expression-1 op expression-2

Les deux expressions sont évaluées puis comparées. La valeur rendue est de type int (il n'y a pas de type booléen en C); elle vaut 1 si la condition est vraie, et 0 sinon.

Rq : à ne pas confondre l'opérateur de test d'égalité == avec l'opérateur d'affection =

Exemple 1:

```
main()
{
int a = 0;
int b = 1;
if (a = b)
printf("\n a et b sont egaux \n");
else
printf("\n a et b sont differents \n");
}
Affiche à l'écran a et b sont egaux
```

Exemple 2:

```
main()
{
int a = 0;
int b = 1;
if (a == b)
printf("\n a et b sont egaux \n");
else
printf("\n a et b sont differents \n");
}
affiche à l'écran a et b sont differents
```

IV-5. Les opérateurs logiques booléens :

- || est un OU logique,
- && est un ET logique,
- ! est le NON logique,

Comme pour les opérateurs de comparaison, la valeur retournée par ces opérateurs est un int qui vaut 1 si la condition est vraie et 0 sinon.

Dans une expression de type

expression-1 op-1 expression-2 op-2 ...expression-n

L'évaluation se fait de gauche à droite et s'arrête dès que le résultat final est déterminé.

Exemple:

```
int i;
int p;
if ((i >= 0) && (i <= 9) && !(p== 0))
...
La dernière clause ne sera pas évaluée si i n'est pas entre 0 et 9.
```

IV-6. Les opérateurs d'affectation composée :

Les opérateurs d'affectation composée sont :

+= -= *= /= %=

Pour tout opérateur **op**, l'expression :

expression-1 op= expression-2

est équivalente à :

expression-1 = expression-1 op expression-2

Toutefois, avec l'affection composée, expression-1 n'est évaluée qu'une seule fois.

Exemple:

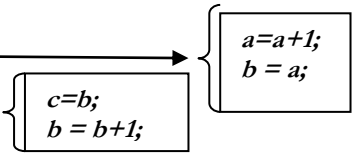
X+=3 équivaut à **X=X+3**

IV-7. Les opérateurs d'incrémentation et de décrémentation :

Les opérateurs d'incrémentation **++** et de décrémentation **--** s'utilisent aussi bien en suffixe (**i++**) qu'en préfixe (**++i**). Dans les deux cas la variable i sera incrémentée, toutefois dans la notation suffixe la valeur retournée sera l'ancienne valeur de i alors que dans la notation préfixe se sera la nouvelle. Par

Exemple:

```
int a = 3, b, c;
b = ++a; /* a et b valent 4 */
c = b++; /* c vaut 4 et b vaut 5 */
```



IV-8. L'opérateur virgule :

Une expression peut être constituée d'une suite d'expressions séparées par des virgules :

expression-1, expression-2, ... , expression-n

Cette expression est alors évaluée de gauche à droite. Sa valeur sera la valeur de l'expression de droite.

Exemple:

```
main()
{
int a, b;
b = ((a = 3), (a + 2));
printf("\n b = %d \n",b);
}
affiche à l'écran b = 5.
```

Rq : La virgule séparant les arguments d'une fonction ou les déclarations de variables n'est pas l'opérateur virgule.

IV-9. L'opérateur conditionnel ternaire :

L'opérateur conditionnel **?** est un opérateur ternaire. Sa syntaxe est la suivante :

condition ? expression-1 : expression-2

Cette expression est égale à expression-1 si condition est satisfaite, et à expression-2 sinon.

Exemple:

L'expression
x >= 0 ? x : -x :
correspond à la valeur absolue d'un nombre. De même l'instruction
m = ((a > b) ? a : b);
affecte à m le maximum de a et de b.

IV-10. L'opérateur adresse :

L'opérateur d'adresse **&** appliqué à une variable retourne l'adresse-mémoire de cette variable. La syntaxe est

&objet

IV-11. Règles de priorité des opérateurs :

Le tableau suivant classe les opérateurs par ordres de priorité décroissants. Les opérateurs placés sur une même ligne ont même priorité. Si dans une expression figurent plusieurs opérateurs de même priorité, l'ordre d'évaluation est défini par la flèche de la seconde colonne du tableau. On préfèrera toutefois mettre des parenthèses en cas de doute...

opérateurs	
() [] -> .	→
! ~ ++ -- -(unaire) (*type*) *(indirection) &(adresse) sizeof	←
* / %	→
+ -(binaire)	→
<< >>	→
< <= > >=	→
== !=	→
&(et bit-à-bit)	→
^	→
|	→
&&	→
||	→
? :	←
= += -= *= /= %= &= ^= |= <<= >>=	←
,	→

V. Les instructions de lecture et d'écriture :

En général pour qu'un programme présente un intérêt pratique, il doit pouvoir produire des informations, en particulier vers le périphérique de sortie que constitue un écran. De même il doit pouvoir recevoir des données, en particulier à partir du périphérique appelé clavier. Les fonctions de lecture et d'écriture se trouvent dans la librairie standard **stdio.h**

Sur certains compilateurs, l'appel à la librairie stdio.h par la directive au préprocesseur **#include <stdio.h>**

V-1. Sorties vers l'écran : l'instruction printf :

La fonction **printf** est une fonction d'impression formatée, ce qui signifie que les données sont converties selon le format particulier choisi. Sa syntaxe est :

printf (<*format*>, [*expression1* , *expression2*, ..., *expressionN*]) ;

Cette instruction affiche la valeur résultant des expressions mentionnées en tenant compte des indications précisées dans le format.

Le format est constitué de caractères qui sont affichés tel quels et de codes de format qui précisent chacun la manière d'écrire une des valeurs spécifiées dans la liste qui suit (le premier code pour la première expression, le deuxième pour la deuxième, etc.). Tous les codes format commencent par **%**.

Exemple:

```
printf( "Bonjour.") ;
```

Affiche : **Bonjour**
Cette instruction ne contient qu'un format simple

```
int RES ;
RES=123 ;
printf( "Le resultat est : %d . Bonjour.", RES ) ;
```

Affiche : **Le resultat est 123 .Bonjour**
%d est le format permettant d'afficher une valeur entière dans un format décimal

```
int i =2, j=3 ;
printf( "%d + %d = %d",  i  , j , i + j );
```

Affiche : **2 + 3 = 5**

Les principaux formats sont :

format	conversion en	écriture
%d	int	décimale signée
%ld	long int	décimale signée
%u	unsigned int	décimale non signée
%lu	unsigned long int	décimale non signée
%o	unsigned int	octale non signée
%lo	unsigned long int	octale non signée
%x	unsigned int	hexadécimale non signée
%lx	unsigned long int	hexadécimale non signée
%f	double	décimale virgule fixe
%lf	long double	décimale virgule fixe
%e	double	décimale notation exponentielle
%le	long double	décimale notation exponentielle
%g	double	décimale, représentation la plus courte parmi %f et %e
%lg	long double	décimale, représentation la plus courte parmi %lf et %le
%c	unsigned char	caractère
%s	char*	chaîne de caractères

Remarque Importante :

printf n'est pas réellement une instruction. Il s'agit en fait d'une fonction, c'est à dire d'un morceau de programme réutilisable (nous en reparlerons plus tard). Cela explique l'utilisation des parenthèses. Cela implique de dire au programme où trouver cette fonction (dans quel fichier). Dans le cas de **printf** cela se fait en ajoutant la déclaration suivante au début du programme : ***#include*** <***stdio.h***>

Ce qui veut dire "inclure les fonctions déclarées dans le fichier stdio.h".

a. Largeur minimale pour les entiers :

Pour les entiers, nous pouvons indiquer la largeur minimale de la valeur à afficher. Dans le champ ainsi réservé, les nombres sont justifiés à droite.

Exemple:

instruction	Affichage
printf("%4d", 123);	_123
printf("%4d", 1234);	1234
printf("%4d", 12345);	12345
printf("%4u", 0);	___0
printf("%4X", 123);	__7B
printf("%4x", 123);	__7b

b. Largeur minimale et précision pour les rationnels :

Pour les rationnels, nous pouvons indiquer la largeur minimale de la valeur à afficher et la précision du nombre à afficher. La précision par défaut est fixée à six décimales. Les positions décimales sont arrondies à la valeur la plus proche.

Exemple:

instruction	Affichage
printf("%f", 100.123);	100.123000
printf("%12f", 100.123);	__100.123000
printf("%.2f", 100.123);	100.12
printf("%5.0f", 100.123);	__100
printf("%10.3f", 100.123);	___100.123
printf("%.4f", 1.23456);	1.2346

V-2. La fonction de saisie scanf :

Syntaxe : **scanf (** <format>, **&***variable1* [, **&***variable2*, ..., **&***variableN*] **) ;**

Cette instruction permet de ranger dans une ou plusieurs variables des valeurs entrées au clavier. Lorsqu'elle est exécutée, l'ordinateur attend que des données soient entrées au clavier et que la touche [Entrée] soit pressée. Le format permet de préciser comment les caractères entrés au clavier doivent être interprétés avant d'être affectés aux variables. Les noms de variables doivent être précédés du caractère **&** qui signifie "adresse de".

Le format est codé en utilisant les mêmes symboles que pour l'instruction **printf().**

Il est fortement conseillé d'utiliser un code format unique dans une instruction **scanf()**. Et de toujours faire précéder cette instruction par une instruction **printf()** pour indiquer à l'utilisateur la nature de la donnée que le programme attend. Il n'y a rien de pire qu'un programme qui se met à attendre une donnée sans avoir rien demandé....

Exemple:

```
int age ;
printf("Entrez votre age") ;
scanf("%d", &age) ;
printf("Votre age est %d", age) ;
```

Si l'utilisateur entre au clavier la valeur **18**, le programme affiche Votre age est 18

VI. Un premier programme complet :

Il est possible (et nécessaire) de donner des explications sur le programme en utilisant des commentaires. Il suffit de placer le texte explicatif entre les symboles **/*** et ***/**.

```
/* Exemple de programme calculant le carré d'un nombre fournit par l'utilisateur : */

#include <stdio.h>
#include <conio.h>

main()
{
        float     nb ;
        float     carre ;

        printf( "Veuillez  donner  votre nombre : ") ;
        scanf( "%f", &nb);

        carre = nb*  nb ;  /* Calcul le carre de nb */

        printf( "Son carre est : %f\n", carre) ;
        getch() ;
        return 0 ;

}
```

Remarque Importante *:*

- Il est impératif de toujours bien aérer et documenter un programme pour qu'il soit agréable et donc facile à lire. En particulier en utilisant les tabulations et de sauts à la ligne pour bien mettre en relief les articulations du programme.
- De même il est important de toujours donner des noms explicatifs aux variables (Le nom "Resultat" est plus parlant que "x")

Voici le même programme :

```
#include <stdio.h>
#include <conio.h>
main() {float titi ;float      tata ;
printf( "Veuillez  fournir votre nombre : ") ;
scanf( "%f", &tata);titi=tata*tata;printf( "Son carre est : %f\n", tata) ;getch() ;return 0 ;}
```

VII. Schémas de contrôle :

VII-1. Définition

Un schéma de contrôle, appelé parfois structure de contrôle, est une instruction qui définit la manière dont les instructions qu'elle contient doivent être exécutées : dans quel ordre, combien de fois, à quelles conditions...

Un schéma de contrôle est donc une règle de composition des instructions : toute instruction d'un programme se trouve toujours sous le contrôle d'une telle structure.

Un schéma de contrôle (et toutes les instructions qui en dépendent) est lui-même une instruction: il s'utilise donc comme toute instruction sous le contrôle d'un autre schéma.

Seul le schéma de plus haut niveau n'est pas sous contrôle (il faut s'arrêter un jour !) : il s'agit du programme lui-même.

VII-2. Schéma séquentiel (instruction composée)

C'est le schéma de base de tout programme : dans ce schéma les instructions sont exécutées dans l'ordre dans lequel elles ont été écrites.

En C, le schéma séquentiel est délimité par les symboles { et } :

Syntaxe :

```
{
        instruction1 ;
        instruction2 ;
        ...
        instructionN ;
}
```

Les instructions sont exécutées séquentiellement dans l'ordre d'écriture. Les premières instructions peuvent être des instructions de déclaration.

Exemple:

```
{
        int x, y, z;
        printf( "donnez X = " ) ;
        scanf ( "%d", &x ) ;
        printf( "donnez Y = " ) ;
        scanf ( "%d", &y ) ;
        z = x ;
        x = y ;
        y = z ;
        printf( " X = %d, Y= %d\n", x, y ) ;
}
```

VII-3. Schéma parallèle

C'est un schéma utilisé dans certains langages dédiés aux machines multiprocesseurs : dans ce schéma toutes les instructions sont exécutées simultanément. Ce schéma n'existe pas dans C.

VII-4. Les conditions en C (et les nombres booléens)

Les deux schémas de contrôle qui suivent nécessitent de pouvoir tester des conditions (est-ce qu'une certaine condition est remplie oui ou non ?). Il est donc nécessaire de pouvoir disposer d'un type de variable pouvant représenter les valeurs Vraie et Faux (c'est à dire un nombre booléen). En C il n'y a pas de type spécifique car il est possible d'utiliser n'importe quel type numérique pour représenter un tel nombre.

La règle est simple :

Faux est représenté par la **valeur 0**
Vrai est représenté par une **valeur différente de 0**

Pour des raisons d'économie de place, c'est le type char qui est généralement utilisé pour représenter un booléen. (Ce type a donc trois utilisations)

Remarque : Une condition est souvent appelée expression booléenne.

Il existe en C différents opérateurs de comparaison qui permettent de comparer le résultat de deux expressions (et donc en particulier le contenu de deux variables), et ainsi de retourner un résultat de type booléen.

Exemple 1:

```
char b1, b2, b3 ;
int A,B;

b1 = 3 > 5 ;

A= 5 ;
B = 1 ;
b2 = A >= B+2

A la fin :
        b1 est égal à zéro ( Faux)
        b2 est différent de zéro (Vrai)
```

Il est possible de composer les expressions booléennes avec des opérateurs logiques.
Soit B1 et B2 deux expressions rendant un résultat de type booléen (n'importe quel type numérique), les principales opérations de composition possibles sont :

- Négation (le NON logique) : **!** B1 ==> Boolean
- Conjonction (le ET logique) : B1 **&&** B2 ==> Boolean
- Disjonction (le OU logique) : B1 || B2 ==> Boolean

B1	B2	! B1	B1 && B2	B1 || B2
Vrai	Vrai	False	Vrai	Vrai
Vrai	Faux	Faux	Faux	Vrai
Faux	Vrai	Vrai	Faux	Vrai
Faux	Faux	Vrai	Faux	Faux

Il est bien entendu possible d'utiliser des parenthèses pour regrouper ces opérations

Exemple 2:

```
main() {
        int I1,I2,I3,I4;
        char B2;


        I1 = 1 ;
        I2 = 2 ;
        I3 = 3 ,
        I4 = 4 ;
        B1 = ! ( (I1 == I2) || (I3 < I4) ) ;    { expression booléenne => Non ( Faux Ou Vrai ) =>
Faux}
        B1 = B1 && (I3 < I4) ) ;  { expression booléenne => Faux Et Vrai => Faux}

}
```

VII-5. Schémas de choix (instructions conditionnelles) :

Le schéma de choix permet de choisir d'exécuter, en fonction d'une condition, une (ou aucune) des instructions qu'il contrôle.

a. Choix alternatif (choix entre une ou deux instructions) :

En C le schéma de choix alternatif est représenté à l'aide des mots clés **if** et **else** :

Syntaxe :

```
if ( expression booléenne )
      instruction1 ;
 else
      instruction2 ;
```

Si l'expression est évaluée à Vrai alors instruction1 est exécutée, sinon c'est instruction2 qui est exécutée (à condition que la clause **else** soit présente).

```
/* Programme permettant de trouver le plus grand de deux nombres */
#include <stdio.h>
main()
{
        int x, y, max;

        printf ("Donner X : " ) ; scanf("%d", &x) ;
        printf ("Donner Y : " ) ; scanf("%d",  &y) ;

        if  (x >y)          max = x ;
        else       max = y ;

        printf ( "Le maximum est %d\n" , max) ;
}
```

Autre solution :

```
/*  AUTRE SOLUTION : Programme permettant de trouver le plus grand de deux nombres */

        .......
        max = y ;
        if  (x >y)          max = x ;
        .......
```

Remarque : les deux instructions d'un schéma de choix alternatif sont souvent des instructions composées, ce qui permet de mettre plusieurs instructions dans chaque alternative :

```
if ( x>y)
  {
        max = x ;
        min = y ;
  }
else
  {
        max = y ;
        min = x ;
  }
```

Application : Résolution de l'équation AX + B = 0

Question : Ecrire un programme permettant de résoudre l'équation AX+B=0.

Cahier des charges :

Données :
A : de type réel.
B : de type réel.

Résultat :
X : de type réel, représentant l'inconnu.

Relation :
si A<> 0 alors X=-B/A
si A=0 et B=0 alors une infinité de solution
si A=0 et B<>0 alors pas de solution

Programme C

```
void main()
{
        float a,b,x ;

        printf ("Donner a : " ) ; scanf("%f", &a) ;
        printf ("Donner b : " ) ; scanf("%f",  &b) ;

        if ( a==0 )
           {
                if (b==0)           printf( "Il y a une infinité de solutions") ;
                else       printf( "Impossible"') ;
           }
        else
           {
                x  = -b/a ;
                printf( "La solution est : %f", x)  ;
           }
}
```

b. Choix multiple (choix entre plus de deux instructions) et instruction break

En C le schéma de choix multiple est représenté à l'aide des mots clés **switch**, **case** et **default** :

Syntaxe :

```
switch ( expression de type entier ou caractère )
    {
      case choix1 : liste_d'instructions 1 ;
      case choix2 : liste_d'instructions 2 ;
      ...
      case choixN : liste_d'instructions N ;
       default liste_d'instructions ;
    }
```

L'expression est évaluée et son résultat est comparé aux valeurs incluses dans chaque choix. Si la même valeur est trouvée dans le ième choix, alors la ième liste d'instructions et toutes les suivantes sont exécutée, sinon et si la clause default est présente, c'est la liste d'instructions présente dans cette clause qui est exécutée.

```
main ( )
{
        int n ;
        printf ("Donner un entier : " ) ; scanf("%d", &n) ;

        switch ( n )
           {
                case 0 : printf ("zero\n") ;
                case 1 : printf ("un\n") ;
                case 2 : printf ("deux\n") ;
           }
        printf( "FINI\n" ) ;
}

Donner un entier : 1
un
deux
FINI
```

```
main ( )
{
        int n ;
        printf ("Donner un entier : " ) ; scanf("%d", &n) ;

        switch ( n )
           {
                case 0 : printf ("zero\n") ;
                case 1 : printf ("un\n") ;
                case 2 : printf ("deux\n") ;
                default : printf ("autre\n") ;
           }

        printf( "FINI\n" ) ;
}

Donner un entier : 1
un
deux
FINI

Donner un entier : 34
autre
FINI
```

Il est évidemment indispensable de pouvoir n'exécuter qu'une des listes d'instructions. Il suffit pour cela d'utiliser l'instruction **break** là où cela est nécessaire. Cette instruction provoque la sortie immédiate de l'instruction **switch** :

```
main ( )
{
        int n ;
        printf ("Donner un entier : " ) ; scanf("%d", &n) ;

        switch ( n )
           {
                case 0 :  printf ("zero\n") ;
                          break ;
                case 1 :  printf ("un\n") ;
                          break ;
                case 2 :  printf ("deux\n") ;
                          break ;
                default :          printf ("autre\n") ;
           }
        printf( "FINI\n" ) ;
}

Donner un entier : 1
un
FINI
```

VII-6. Schémas itératifs (instructions répétitives)

Les schémas de contrôle de type itératifs permettent de répéter plusieurs fois la même suite d'instructions. Cette répétition ne s'arrête que lorsqu'une certaine condition est remplie ou au bout d'un certain nombre de fois.

a. tant que ...faire

Syntaxe :

```
while ( expression_booléenne ) instruction ;
```

Tant que le résultat de l'expression booléenne est vrai on répète l'exécution de l'instruction. Si l'expression est fausse dès le départ, l'instruction ne sera jamais exécutée.

Ce schéma est le plus général : il permet de réaliser tous les types d'itérations et peut donc remplacer tout autre schéma itératif.

Pour utiliser correctement cette instruction il est nécessaire de la faire précéder par des instructions d'initialisation de l'itération (Initialisation des variables permettant de calculer l'expression booléenne, et de celles utiles lors de l'exécution de l'instruction).

Le schéma complet d'utilisation de cette instruction est donc :

```
Instructions d'initialisation ;
while( condition_de_poursuite ) instruction_à_répéter ;
```

Question : Calculer la somme des carrés des N premiers entiers.

Cahier des charges :
Données :
N : nombre de type entier.
SOMME : nombre de type entier.
Relation :
$SOMME = 1^2+2^2+ ...+N^2$

C'est à dire :
initialiser i avec 1 et SOMME avec 0
puis tant que i est inférieure ou égale à N faire :
ajouter i*i à SOMME
ajouter 1 à i.

```
/* debut programme */
main ()
{
        int N ;             /* Données */
        int SOMME ;         /* Résultat */
        int i ;             /* Variable de travail */

        printf( "Donner le nombre de carrés à additionner : " );
        scanf( "%d",  &N ) ;

        /* Initialisation */
        i = 1 ;
        SOMME = 0 ;

        /* Itération */
        while ( i <= N )
          {
                SOMME = SOMME + i*i ;
                i = i + 1 ;
          }

        printf ( "La somme des carrés des %d  premiers entiers est %d", N, SOMME )
}
```

Observez bien dans cet exemple l'utilisation de la variable i qui est initialisée avec une certaine valeur, puis incrémentée de 1 à chaque itération et dont la valeur est utilisée comme une condition d'arrêt (ici on s'arrête dès que i devient égal à N). Une telle variable est appelée "compteur de boucle".

Conseil : Vérifiez toujours soigneusement "à la main" le nombre d'itération que le programme réalisera, et le calcul en résultant : dans l'exemple précédent le dernier carré additionné est bien N^2 (et pas $(N+1)^2$, comme on pourrait le craindre).

A la fin de l'étape d'	N	I	SOMME	I<=N
Initialisation	3	1	0	Vrai
Itération 1	3	2	1	Vrai
Itération 2	3	3	1+4=5	Vrai
Itération 3	3	4	1+4+9=14	False
Edition	3	4	14	False

b. répéter ...tant que

Syntaxe :

```
do Instruction ;
while ( Expression Booléenne );
```

On répète l'exécution de l'instruction jusqu'à ce que l'expression soit fausse. L'instruction sera toujours exécutée au moins une fois, même si l'expression est fausse dès le départ.

Ce schéma est donc plus restrictif que le précédent, puisqu'il ne s'applique pas aux cas ou zéro itération doit être réalisée. Il est cependant parfois plus facile à utiliser.

Pour utiliser correctement cette instruction il est nécessaire de la faire précéder par des instructions d'initialisation de l'itération (initialisation des variables permettant de calculer l'expression booléenne, et de celles utiles lors de l'exécution de l'instruction).

Le schéma complet d'utilisation de cette instruction est donc :

```
Instructions_d'initialisation ;
do Instruction_à_répéter ;
while ( condition );
```

```
/* debut programme */
void main ()
{
        int N ;             /* Données */
        int SOMME ;         /* Résultat */
        int I ;             /* Variable de travail */

        printf( "Donner le nombre de carrés à additionner : " );
        scanf( "%d",  &N ) ;

        /* Initialisation */
        i = 1 ;
        SOMME = 0 ;

        /* Itération */
        do
          {
                SOMME = SOMME + i*i ;
                i = i + 1 ;
          }
        while (  i <= N ) ;

        printf ( "La somme des carrés des %d  premiers entiers est %d", N, SOMME )
}
```

Remarque : Ce programme ne marche plus pour N=0

c. pour… faire

Syntaxe :

```
for ( instruction1 ; condition ; instruction2 ) instruction3 ;
```

L'instruction 1 est exécutée une fois et permet d'initialiser la boucle. Puis tant que la condition est vraie, l'instruction 3 est exécutée, suivie de l'instruction 2 (appelée post-instruction).

Ce schéma est idéal lorsque le nombre de répétitions (d'itérations) est connu lors de l'écriture du programme. On se sert alors d'un compteur de boucle (souvent appelé i ou j) pour compter le nombre de répétition. Ainsi, pour répéter N fois une même instruction on utilise souvent le schéma suivant :

```
for ( i=0 ; i<10 ; i=i+1 ) instruction_à_répéter ;
```

```
/* debut programme */
void main ()
{
        int N ;             /* Données */
        int SOMME ;         /* Résultat */
        int I ;             /* Variable de travail */

        printf( "Donner le nombre de carrés à additionner : " );
        scanf( "%d",  &N ) ;

        /* Initialisation */
        SOMME = 0 ;

        /* Itération */
        for( i=1 ; i<=N  ; i=i+1 )  SOMME = SOMME + i*i ;

        printf ( "La somme des carrés des %d  premiers entiers est %d", N, SOMME )
}
```

Autre possibilité :

```
        /* Initialisation */
        SOMME = 0 ;

        /* Itération */
        for( i=N ; i>0  ; i=i+1 )  SOMME = SOMME + i*i ;
```

Remarque : il est possible dans un **for** de mettre plusieurs instructions d'initialisation en les séparant par des virgules. Cela est aussi vrai pour la post-instruction. Le programme peut donc devenir :

```
        for( i=N,SOMME=0 ; i>0  ; i=i+1 )  SOMME = SOMME + i*i ;
```

VII-7. Les instructions break et continue :

L'instruction **break** peut être utilisée dans une boucle (**for** ou **while**) pour sortir immédiatement de la boucle et passer à l'instruction qui suit cette boucle (de manière similaire à ce qui se passe avec une instruction **switch**).

```
...
for ( i=1 ; i<7 ; i++)
   {
          printf("Debut %d\n", i ) ;
          if ( i >= 3 && i < 6) break ;
          printf("Fin %d\n", i ) ;
   }
...

Affiche :
Debut 1
Fin 1
Debut 2
Fin 2
Debut 3
```

L'instruction **continue** permet de passer directement à l'itération suivante sans exécuter les instructions suivantes de la boucle.

```
...
for ( i=1 ; i<7 ; i++)
   {
          printf("Debut %d\n", i ) ;
          if ( i >= 3 && i < 6) continue ;
          printf("Fin %d\n", i ) ;
   }
...

Affiche :
Debut 1
Fin 1
Debut 2
Fin 2
Debut 3
Debut 4
Debut 5
Debut 6
Fin 6
```