Langage C

Bases du langage C

Thomas Guyet

AGROCAMPUS-OUEST, Centre de Rennes

Version 1.0, Dernière modification : 18 avril 2012

Table des matières

1 Éléments de langage C **4**

1.1 Un premier programme 4
1.2 Les variables 5
1.2.1 Déclaration et affectation d'une variable 5
1.2.2 Types de bases et leurs opérateurs usuels 7
1.3 Les tableaux 12
1.3.1 Déclaration 12
1.3.2 Désigner un emplacement du tableau 13
1.3.3 Les tableaux à multiple dimensions 14
1.3.4 Les chaînes de caractères 15
1.3.5 Les indices sont aussi des variables 16
1.3.6 Exercices 17
1.4 Les structures de contrôles 18
1.4.1 Les instructions conditionnelles `if` 18
1.4.2 Les structures itératives 20
1.4.3 Combiner les structures de contrôle 25
1.4.4 Écritures courtes 28
1.5 Les entrées/sorties 28
1.5.1 L'affichage d'un texte dans un terminal avec `printf` 29
1.5.2 Lire une valeur du clavier avec `scanf` 33
1.5.3 Repérer les erreurs de saisie de l'utilisateur 33
1.5.4 Lectures et écritures dans des fichiers : `fprintf` et `fscanf` 34
1.6 Correction des exercices 37

2 Structures de données **41**

2.1 Définir de nouveaux types avec `typedef` 41
2.2 Types énumérés 41
2.2.1 Définition d'un type énumeré 41
2.2.2 Utilisation d'un type énumeré 42
2.3 Les structures de données 43
2.3.1 Définition d'une nouvelle structure 43
2.3.2 Utilisation des variables-structure 44

2.4 Correction des exercices 46

CHAPITRE 1

Éléments de langage C

Dans cette partie du cours, je donne les bases de la syntaxe du langage C (en norme ANSI 89).
Après avoir donner, dans une première partie, le classique exemple du *Hello world*, la seconde section sera consacrée à la déclaration, à l'initialisation et la manipulation de variables simples. Dans la troisème partie, je présenterai la syntaxe des structures de contrôles usuelles en algorithmique : la condition et l'iteration. Finalement, la dernière partie sera consacrée aux entrées/sorties, c'est-à-dire à la lecture et à l'écriture de variables.
À l'issue de cette partie de cours vous serez en mesure d'écrire un programme simple en C. Mais ce n'est là que de la technique, le couteau du sculteur. Il ne sera de bon programmeur C sans une maîtrise quasi-complète de ce chapitre tout comme il ne peut y avoir d'artiste sans une maîtrise de la technique.

1.1 Un premier programme

Voici un exemple de programme C très simple, qui changera un peu du classique *HelloWorld* dans le sens où ce qu'il affiche est légèrement différent :

```
#include <stdio.h>

/*
Ceci sera mon premier programme
*/

int main(void)
{
  printf("Je teste mon premier programme !\n");
  printf("Et en plus il est en C !\n"); // Encore un commentaire
  return 0;
}
```

Listing 1.1 – Hello world nouvelle formule ...

Quelques commentaires à propos de ce programme :
- La directive `#include` permet d'importer des définitions de fonctions utiles (ici, il est nécessaire d'inclure la bibliothèque `stdio.h` pour pouvoir utiliser la fonction `printf`). C'est une directive du préprocesseur (voir le chapitre sur la compilation pour plus de détails).
- Tout ce qui est écrit entre `/* ... */` et après `//` est considéré comme un commentaire de programme. Tout ce qui est en commentaire sera totalement ignoré lors de la compilation.
- Les lignes vides ne sont pas prises en compte.
- Un programme C doit **impérativement contenir une fonction nommée** `main()`. Cette fonction sert de point d'entrée pour un programme C. On sait alors que le programme va commencer par faire l'instruction de la ligne 8.

- Les accolades ("{", "}") définissent les limites de la fonction `main()`. On parle alors de **blocs d'instructions**. Aucune instruction ne peut écrite en dehors d'un bloc d'instructions. Ici, la dernière instruction exécutée sera celle de la ligne 11.
- **Toutes** les instructions doivent se terminées par un point virgule (" ;"). Il faut alors remarquer que dans ce programme exemple, il n'y a que 3 instructions, et en particulier :
  - `int main(void)` n'est pas une instruction ! Il s'agit du profil de la fonction associée au bloc d'instructions (entre accolades) qui le suit.
  - les lignes 3 à 5 sont des commentaires, il n'est pas nécessaire d'ajouter des " ;",
  - en revanche, à la ligne 9, le point virgule doit impérativement être positionné avant les commentaires sinon il sera ignoré et il y aura une erreur de syntaxe.
- `printf` est une **fonction** : toutes les fonctions sont repérés par l'utilisation de parenthèses entre lesquels il faut placer **les paramètres de la fonction** séparés par des virgules. Ici, la fonction prend un seul paramètre (une chaîne de caractères). Cette fonction affiche la chaîne de caractère dans le terminal d'exécution.
- Les **chaînes de caractères** sont écrites entre **guillemets doubles** "`"`". Les guillemets simples servent à définir un caractère (et un seul caractère).
- L'instruction `return 0` donne fin à la fonction `main()` en "retournant" la valeur 0. Pour le système, cela signifie que tout c'est bien passé (sinon, vous pouvez faire un `return 1`). On comprendra un peu plus cette instruction lorsqu'on parlera des fonctions.
- La dernière ligne d'un programme doit être vide ! Je ne sais pas pourquoi, mais si vous ne le faites pas, le compilateur ne manquera pas de vous le signaler !

Si vous écrivez ce programme dans un éditeur avancé (tel que `gedit` ou `Code::Blocks`) qui vous fera la coloration syntaxique du programme vous pourrez mieux comprendre ce que "lit" l'ordinateur. En effet, la typographie permet de distinguer facilement les mots clés (`int`, `#include`, `void`, `return`), les instructions, les commentaires et les chaînes de caractères.

1.2 Les variables

1.2.1 Déclaration et affectation d'une variable

Definition 1 - Déclaration d'une variable

Déclarer une variable est une instruction qui indique au programme qu'on aura besoin d'une variable. Cette variable sera identifiée par un nom.
L'instruction a pour effet d'allouer un espace mémoire correspondant à la taille mémoire des valeurs que peut contenir la variable.

Pour déclarer une variable, il faut donner son type (un entier, un caractère, une structure complexe, ...). Lorsqu'on déclare une variable, le système alloue un espace mémoire dans lequel la valeur de cette variable sera retenue (sous forme binaire). Le type sert :
- (à la déclaration) à savoir quel est la taille mémoire à allouer pour contenir une valeur,
- (à l'utilisation de la variable) à savoir comment interpréter les bits contenus dans l'espace mémoire de la variable.

Une variable se déclare en donnant son type puis son nom :

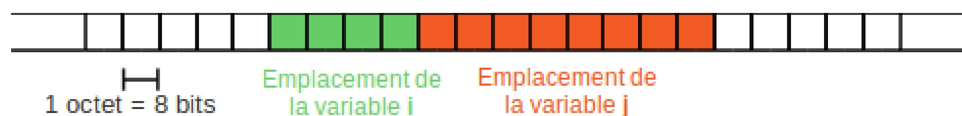
type_variable nom_variable ;

Exemple 1

Par exemple :

```
int i;
long j;
```

permet de déclarer une variable, nommée **i**, *de type* `int` *(nombre entier sur 32 bits), et une variable* **j** *de type long (nombre entier sur 64 bits). Ainsi, on aura en mémoire :*



! Attention !

Toute variable d'un programme doit être déclarée. Sinon, le programme ne sait pas de quoi vous parlez !

Definition 2 - Affectation

L'**affectation** consiste à attribuer une valeur à une variable. Une instruction d'affectation s'écrit ainsi :
On distingue généralement trois types d'affectation :
- l'affectation par valeur : on donne alors directement la valeur à attribuer à une variable

  nom_variable = valeur ;
- l'affectation par une variable : l'instruction recopie le contenue de la variable à droite du "=" dans la variable de gauche

  nom_variable1 = nom_variable2 ;
- l'affectation par le résultat d'une opération : l'instruction recopie le résultat d'une opération (par exemple une opération arthmétique)

Par exemple, l'affectation de la variable par une valeur

```
i=0;
```

L'affectation consiste à remplir les cases de l'emplacement mémoire d'une variable avec une valeur. L'illustration suivante représente le résultat de l'affectation de la variable **i** de l'exemple précédent.

Dans la mesure où la variable **j** n'a pas été initialisée, on ne sait pas ce que contient la variable **j**. En C, il est très **fortement conseillé** d'affecter une variable dès sa déclaration ! Ceci évite d'avoir des points

d'intérrogation dans les cases mémoire ... (*cf.* illutration 1.2.1) On parle alors d'**initialisation** d'une variable.
Il a déjà été indiqué que le langage C est un langage fortement typé. En particulier, l'opération d'affectation est une opération **typée**. C'est-à-dire que le type de la valeur doit être compatible avec le type de la variable à affecter. On comprend bien qu'il n'est pas possible, par exemple, de mettre une valeur flottante dans une variable qui a été déclaré comme un entier !

! Attention !

Quelques avertissements à propos de l'égalité :
- le "=" ne doit pas se lire comme une égalité !
- le signe "=" est l'indication d'une affectation qui se lit TOUJOURS de droite à gauche : la variable à gauche de l'égalité prend la valeur à droite.

L'initialisation d'une variable peut se faire en même temps que sa déclaration ainsi :

```
int i = 0;
```

On peut affecter une variable à partir d'une autre variable. Par exemple :

```
1 int i, j; // declare deux variables de type entier
2 j=10;
3 i=j;
```

On peut également affecter une variable en récupérant la valeur d'une opération :

```
1 int i, j;
2 float k; // declare trois variables de type entier
3 j=10;
4 i=2;
5 k=j+10*i;
```

1.2.2 Types de bases et leurs opérateurs usuels

Les nombres

int	Entier relatif
long	Entier relatif long
unsigned int	Entier (positif)
float	Nombre en virgule flottante sur 32 bits (architecture usuelle)
double	Nombre à virgule flottante double précision (64 bits)
long double	Nombre à virgule flottante quadruple précision (128 bits)

Les nombres disposent des opérateurs mathématiques usuels : $+$, $*$, /, –, % (reste de la division). Pour récupérer le résultat d'une opération arithmétique, vous utilisez naturellement l'instruction d'affectation :

```
1 int i =10, j=3, k;
2 k=i+j;
3 k=i/j; //k vaut 3
4 k=i%j; //k vaut 1 (le reste de la division)
```

Comme en mathématique l'usage des parenthèses permettent de modifier les ordres de priorité des opérations (ordre de priorité usuels).

Remarque 2 - Division entière et division "à virgule"

L'opérateur / à deux significations en fonction des opérandes. Si les opérandes sont des nombres entiers, alors il s'agit d'une division entière : le résultat de l'opération sera alors un entier. Par exemple, la division de 3 par 2 donnera comme résultat 1.
En revanche, si les opérandes sont des nombres flottants, alors l'opérateur sera une division à virgule flottante. Le résultat de cette division sera un flottant (`float` ou `double`). Par exemple, la division de 3.0 [1] par 2.0 donnera comme résultat 1.5.
Cette erreur est courante, et parfois difficile à trouver. Dans l'exemple ci-dessous, on pourrait s'attendre à ce que le résultat affiché soit 39.75 (34,5+5.25). Mais l'opération décrite sur la ligne 3 va conduire le processeur à faire la division **entière** entre `i` et 4 qui aura pour résultat un entier (5). Le résultat affiché sera ainsi 39.5 !

```
1 double x=34.5;
2 int i=21;
3 double y = x + i/4;
4 printf("%lf\n", y);
```

Cette différence est tout sauf anodine ...

Remarque 3 - Calcul d'une puissance

Il faut faire très attention à l'opérateur ^ qui n'est pas un opérateur "puissance" !! En fait, il n'existe pas d'opérateur basique pour le calcul d'une puissance. Si vous voulez calculer le cube d'une valeur `x`, il vaut calculer `x*x*x`.

Si vous avez néanmoins des puissances à calculer, il est possible d'utiliser la fonction `double pow(double x, int i)` disponible dans la librairie de fonctions mathématiques `<math.h>`.

Pour les entiers, on dispose également d'opérateurs unaires très utiles : incrémentation (++, +=) et décrémentation (--, -=). Ces opérateurs peuvent se réécrire de la sorte :
- `i++;` est équivalent à `i=i+1;`
- `i+=5;` est équivalent à `i=i+5;`

Ces opérateurs unaires permettent d'écrire rapidement des opérations usuelles et en plus, elles sont plus efficaces que leur équivalent (ils correspondent généralement à de vrais instructions machine).

Exemple 2

```
1 int i=10, j=3;
2 i++;
3 i--;
4 i+=3;
5 j-=2;
```

À la fin de ces instructions, `i` *vaut 13 et* `j` *vaut 1.*

1. Pour indiquer qu'une constante est un nombre flottant, vous pouvez ajouter simplement un ".0".

Autres types de base

char	Caractère de texte (en fait un entier entre 0 et 255)
void	Sans type

! Attention !

Dans la norme ANSI, il n'y a pas de booléen, il s'agit simplement d'un entier qui vaut 0 pour la valeur `false` ou différent de 0 (par exemple 1) pour la valeur `true`.
Néanmoins, en utilisant la librairie <`stdlib.h`>, vous pouvez bénéficier d'un type `bool` et des valeurs `true` et `false` habituelles.

Pour affecter une valeur à un caractère, on utilise des guillemets simples (les guillemets doubles servent à définir un texte) :

```
char c ='e';
```

Opérateurs de comparaison

Les opérateurs de comparaison permettent de comparer deux nombres et retourne une valeur booléenne [2]

==	Opérateur d'égalité
<, <=	Inférieur, Inférieur ou égal
>, >=	Supérieur, supérieur ou égal
!=	Différent de ...

Exemple 3

```
1 int i = 20, j=5, k=5, r;
2 r = (i == j); // r vaut 0
3 r = (i != j); // r vaut 1 (autre chose que 0)
4 r = (i=j); // r vaut 5
5 r = (j==k); // r vaut 1
6 r = (j>k); // r vaut 0
7 r = (j>=k); // r vaut 1
```

À la ligne 2, `i` n'est pas pas égale à `j` donc `r` vaut 0, et à la ligne 3 il vaut donc 1. À la ligne 4, en revanche, il ne s'agit pas d'une comparaison mais d'une affectation (simple-égal), et ici `r` va valloir 5 (comme `i` et `j`). À la ligne 5, `j` est bien égal à `k`, donc `r` vaut 1. À la ligne 6 et 7, on voit que la comparaison avec > est une comparaison stricte.

! Attention !

Le test d'égalité est un double-égal '=='! Le '=' est l'opérateur d'affectation! Cette erreur est l'une des plus classique pour les débutants en programmation C.

2. En fait, c'est un nombre, mais vous devez le traiter comme un booléen.

Opérateurs logiques

Les opérateurs logiques permettent de combiner des valeurs booléennes :

&&	Et logique
||	Ou logique
!	Non logique

Le parenthésage suit les règles usuelles de priorité (le '&&' est prioritaire sur le '||').

Exemple 4

On donne l'exemple de code ci-dessous :

```
int i = 20, j=5, k=5, m=1, r;
r = (i<=j) || (j>k); // r vaut 0
r = ( (j==k) && m ); // r vaut 1
r = ( (i/2) == (j + k) ); // r vaut 1
```

Pour la ligne 2, `j` *n'est pas supérieur à* `k` *donc* `r` *vaut 0. Pour la ligne 3,* `j` *est bien égal à* `k` *et* `m` *(pris comme une valeur booléenne) représente la valeur* `vrai`*,* `r` *vaut donc 1. Pour la ligne 4, on compare le résultat de la division (entière) avec le résultat de l'addition de* `j` *avec* `k`*. Ces deux résultats étant égaux,* `r` *vaut 1.*

! Attention !

Les opérateurs ET et OU sont des signes doubles ('&&' ou '||') et pas des signes simples. Le '&' et le '|' ont des significations assez différentes que nous ne verrons pas : faites attention à **ne pas les utiliser** !

Conversion entre types / "cast explicite"

Dans certains cas, il peut être intéressant de demander des conversions dans les différents types de nombre. En mauvais franglais, les informaticiens parlent de "*cast explicite*" : l'opération de *cast* désigne une opération de conversion, et elle est dite explicite (le programmeur la demande), par opposition à certaines opérations de conversion qui sont implicitement réalisée par l'ordinateur. En pratique, une conversion explicite se fait de la façon suivante :

nom_variable1 = **(nouveau_type)** *nom_variable2*

```
double d;
int i=23, j=4;
d=i/j; // Effectue une division entiere
d= (double)i/(double)j; //effectue une division en virgule flottante
i=(int)d; //troncature
char c='e';
i=(int)c; //Conversion d'apres la table ascii !!
```

! Attention !

Les conversions se font sous la responsabilité du programmeur : le programme ne vérifie pas la validité de la conversion ! En dehors des cas normaux de conversion, rien n'est garantis sur le résultat.
Toutes les conversions ne sont pas possibles ! S'il est possible de transmutter des serviettes en torchons, et des bananes en oranges, il n'est pas nécessairement possible de transformer des bananes en torchons... même si on le demande.

```
1 int i = -54;
2 uint j;
3 j=(uint)i; //Le compilateur envoie un Warning
```

1.3 Les tableaux

1.3.1 Déclaration

Definition 3

Un **tableau** est une variable qui contient une succession de valeurs **du même type** et en **nombre défini et fixé**.

Pour déclarer un tableau en C, la syntaxe est la suivante :

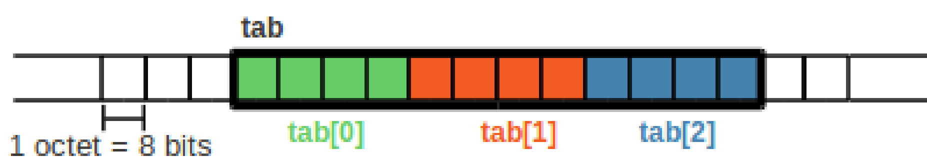
type nom_variable[taille_du_tableau]

Exemple 5

Exemple :

```
1 int tab[3];
2 char string[23];
3 double d[10];
```

Pour le tableau **tab**, *on parle de "tableau d'entiers de taille 3". La taille en mémoire d'un* int *étant de 32 bits, la taille en mémoire du tableau sera de 3*32=96bits et sera disposé ainsi :*



Remarque 4 - Taille fixe des tableaux

Les tableaux sont de taille **définie et fixée à la compilation** ! En particulier, il est **impossible** d'avoir une variable [3] comme taille de tableau.

3. Dans des cas particuliers, vous pourriez rencontrer l'utilisation de variables dans les tableaux, mais ce serait avec des variables dites "constantes". Soit qu'elles soient des variables de préprocessing, soient qu'elles soient explicitement déclarées comme constantes

L'exemple ci-dessous **ne compile pas**! Ce n'est pas correct en langage C.

```
int t;
t=34;
double tab[t];
```

Ce problème met en évidence la nécessité de disposer de structure de données plus complexes comme des listes pour gérer des collections de données dont on ne connait pas a priori la taille. Ces structures de données seront vues dans une section spécifique du cours.

1.3.2 Désigner un emplacement du tableau

Pour désigner un emplacement du tableau, il faut indiquer la position dans le tableau entre crochets :

nom_tableau[position]

! Attention !

En C, la première position est repérée par la position 0 !

Exemple 6

Par exemple, d[5] *est le sixième double dans le tableau précédent.*

En désignant un emplacement du tableau, on peut :
- utiliser la valeur que contient cet emplacement : cette valeur sera du type déclaré pour le tableau,
- modifier la valeur de l'emplacement à l'aide de l'instruction d'affectation.

Exemple 7

Exemple de modification du contenu du tableau :

```
tab[0] = 1;
tab[1] = 243;
tab[2] = 98;
```

Exemple d'utilisation de la valeur d'un emplacement du tableau :

```
int a=tab[1]; // Utilisation d'une valeur du tableau
tab[3] = a + tab[2]; //Modification d'une valeur du tableau
```

! Attention !

Le compilateur ne vérifie jamais que vous accéder à des positions valides d'un tableau !!
Une erreur très classique est de lire ou d'écrire en dehors d'un tableau.

! Attention !

La ligne 2 du dernier exemple va provoquer une erreur grave au moment de l'exécution du programme !

Par leur organisation en mémoire, les tableaux sont très contraints :
- il est impossible de modifier la taille d'un tableau. Il faut donc initialement prévoir quelle doit être la bonne taille du tableau pour faire la tâche assignée au programme,
- l'initialisation du tableau doit se faire manuellement, emplacement par emplacement (il n'est pas possible d'écrire quelque chose comme `int tab[5]=0` : ça n'a pas de sens de mettre un nombre dans un tableau de nombre !!),
- tous les éléments d'un tableau doivent être du même type (ce qui est différent des structures de liste de R ou python).
- il n'existe pas d'opérateurs prédéfinis sur les tableaux (addition, produit , ...) [4]

1.3.3 Les tableaux à multiple dimensions

Il est tout naturellement possible d'utiliser des tableaux à dimensions multiple. Un tableau à dimensions multiple se déclare ainsi : *type nom_tableau**[taille_dim1][taille_dim2][taille_dim3]...***

Exemple 8

Exemple de tableau en dimension 2 :

```
int tab [3][2];
```

Pour désigner un emplacement d'un tableau multiple il faut utiliser une notation similaire à la notation de la déclaration (utiliser plusieurs crochets et non des virgules entre les dimensions) :
`tab [1][0]` désigne un entier du tableau à deux dimensions. Cet emplacement peut être utilisé pour des affectations ou des calculs :

```
tab [1][0]  = 1;
int x=1 + tab[1][0];
```

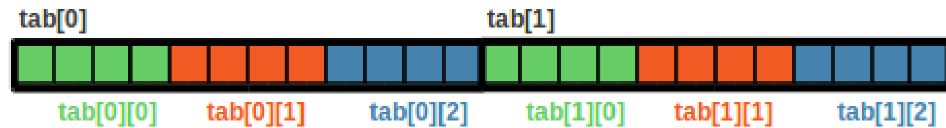
En fait, un tableau à dimension multiple est un tableau de tableaux de tableaux de ... De cette remarque, on déduit l'organisation en mémoire de ces données.

Exemple 9

Sur l'exemple `tab` est un tableau d'entiers à deux dimensions. En fait, c'est un tableau de comprenant 2 tableaux de taille 3 (lecture de droite à gauche). Sa représentation en mémoire est la suivante :

Il vient également des possibilités syntaxiques dangereuses en pratique. Sur l'exemple, vous pouvez écrire `tab[1]` sans problème, mais cet emplacement désigne une variable de type `int [3]` , et donc, ce n'est pas une variable sur laquelle on sait faire des opérations.
En particulier, les instructions suivantes ne sont pas possibles :

4. Ça serait utile pour des opérations matricielles : addition de vecteurs, produit scalaire, etc.



– **tab**[1]=2 : pas possible de mettre un nombre dans un tableau
– **tab**[0]=**tab**[1] : C'est moins débile, mais dans ce cas, le C n'est pas très aidant puisque l'exemple compile bien, mais c'est, a priori, une grosse bétise si vous souhaitiez recopier un sous-tableau : pour recopier deux tableaux, il faut tout faire soit même, case par case !
– **tab**[0] + **tab**[1] : A fortiori, il faut également faire les opérations entre tableaux à la main. De nouveau, cet exemple doit quand même compiler, mais son exécution risque d'être catastrophique ...

Autre conséquence, les tableaux à dimension multiples répondent aux mêmes contraintes que les tableaux en simple dimension : tous les éléments sont du même type et sa taille ne peut pas varier, l'initialisation est manuelle, il n'y a pas d'opérateurs pour manipuler les valeurs du tableau dans leur ensemble, ...

1.3.4 Les chaînes de caractères

Definition 4 - Chaînes de caractères

Une chaîne de caractères est un tableau de caractères dont l'emplacement indiquant la fin de chaîne contient la valeur nulle ("0" écrit en nombre et noté également `'\0'`, le caractère nul). La taille du tableau fixe la taille limite de la chaîne.

Pour déclarer une chaîne de caractères :

char ma_chaine[**taille_chaine**]

Exemple 10

L'instruction suivante déclare une chaîne de caractères de longueur maximale 299 :

```
char s[300];
```

Du fait de la nature tabulaire d'une chaîne de caractères, il est **impossible** d'affecter une valeur à la variable de la sorte :

```
s="coucou";
```

En revanche, il est tout à fait possible d'écrire :

```
s[0]='c';
s[1]='o';
s[2]='u';
s[3]='c';
s[4]='o';
s[5]='u';
s[6]='\0';
```

Pour chaque emplacement, de type char, on assigne ainsi une valeur de cet emplacement mémoire à partir d'une valeur de type char également.
On ajoute finalement la valeur \0 à l'emplacement 6 du tableau. Ainsi, par convention, on sait que c'est la fin de la chaîne de caractères et que les 293 autres emplacements mémoires ne sont pas utilisé (mais ils pourront l'être si on décide de modifier le contenu du tableau ...).

! Attention !

On voit ici la différence entre guillemets simples et guillemets doubles. Les guillemets simples indiquent des caractères, les guillemets doubles indiquent des chaînes de caractères.

On verra lors d'un TD les fonctions de la librairie <string.h> qui permettent de manipuler les chaînes de caractères sans passer systématiquement par les caractères.
En attendant, une solution simple a été prévue pour déclarer rapidement de longues chaînes de caractères :

```
char string [] = "ceci est un texte";
```

Lors de cette initialisation, la taille du tableau est automatiquement définie. Ici, c'est un tableau de taille **18** : 17 caractères + 1 caractère de fin \0.

1.3.5 Les indices sont aussi des variables ...

L'indice utilisé pour désigner une case d'un tableau est une valeur entière. Cette valeur entière peut être un nombre constant **ou une variable**.

Exemple 11

Dans le programme ci-dessous, on utilise une chaîne de caractères comme un tableau de caractères. L'instruction string[i] *permet de désigner la i-ème case du tableau* string*. À la ligne 3, cela permet d'affecter la variable* c *avec le contenu de la i-ème case du tableau. Comme* i *vaut 5, il s'agit en fait de la 6-eme case... puisque les indices commencent à 0.*
Lorsque la valeur de i *change, la case désignée par* string[i] *change également. À la ligne 5, on désigne la 11ème case du tableau.*
Finalement, on illustre un problème classique du C, le débordement de tableau. La dernière ligne de l'exemple fonctionne, mais affiche n'importe quoi. Il est possible de demander d'accéder à la case numéro 45, même si elle n'existe pas.

```
1 char string [] = "ceci est un texte";
2 int i=5;
3 char c= string[i]; // Utilisation de la variable i comme indice
4 printf("caractre la position %d : %c\n", i+1, c);
5 i=10;
6 printf("caractre la position %d : %c\n", i+1, string[i] );
7 i=45;
8 printf("caractre la position %d : %c\n", i+1, string[i] );
```

Le programme affichera :

```
caractère à la position 6 : e
caractère à la position 11 : e
caractère à la position 46 : ■
```

Remarque 5 - Connaître la taille d'un tableau ?

Supposons qu'un tableau ait été déclaré dans une partie du programme et que vous cherchiez à l'utiliser dans une autre partie, mais sans savoir qui et comment il a été déclaré. Vous ne connaissez pas a priori sa taille ... Le problème est que, en C, vous n'aurez AUCUN moyen de retrouver la taille du tableau. Il sera donc nécessaire de toujours conserver une trace de la taille du tableau.
Ceci sera particulièrement utile (et sensé) lorsque vous travaillerez avec des fonctions. En paramètre d'une fonction qui traite un tableau (en paramètre), il est souvent indispensable de passer également un paramètre contenant la taille de ce tableau.

Opérations sur les indices

Si les indices peuvent être des valeurs ou des variables, la logique des choses veux qu'ils puissent également être le résultat d'opérations.

```
char string [] = "ceci est un texte";
int i=5;
char c= string[i];

if( c=='e' ) {
  c = string[i+1]; // dsigne le caractre suivant de celui de la case i
  printf(" caractre aprs la position d'un e : %c\n", c);
}
```

1.3.6 Exercices

Exercice 1 (Instructions sur tableau) *On utilise les déclarations suivantes :*

```
const int TAILLE = 10;
int tab[TAILLE] = {1, 2, 3, 0, 5, -1, -2, -3, -4, -5};//< initialisation du tableau tab avec les
    valeurs entre crochets
int indice, ind1, ind2;
int t[7] = {1, 2 ,4, 8, 16, 32, 64}; //< initialisation du tableau t avec les valeurs entre crochets
```

Question a) Indiquer le contenu du tableau tab après la séquence d'instruction suivante (les instructions menant à des erreurs seront signalées et ignorées) :

```
tab[7]=0;
tab[0]=7;
tab[6]+tab[2]=3;
tab[6]=tab[2]+3;
tab[tab[4]]=tab[8];
tab[tab[9]]=4;
tab[tab[2]] = tab[tab[3]];
```

Question b) Faire un tableau d'évolution des variables pour le programme ci-dessous. Affichage ?

```
indice = 0;
while(indice < TAILLE && tab[indice] >= 0) {
  indice ++;
}
printf("premier nombre negatif : %d\n", tab[indice]);
```

Question c) Faire un tableau d'évolution des variables pour le programme ci-dessous. Contenu de `tab` à la fin ?

```
for (ind1=0; ind1 <= TAILLE/2; ind1++) {
  tab[2*ind1] = ind1;
}
```

Question d) Faire un tableau d'évolution des variables pour le programme ci-dessous. Affichage ? Contenu de `tab` à la fin ?

```
ind1 = ind2 = 0;
while(ind1<TAILLE) {
  if (tab[ind1] > tab[ind2])
    ind2 = ind2 + 2;
  else
    ind1 = ind1 + 2;
  printf("tab[%d] = %d\n", ind1, tab[ind1]);
}
```

1.4 Les structures de contrôles

Les structures de contrôles permettent de définir dans quel ordre doivent être réalisées les instructions.

1.4.1 Les instructions conditionnelles `if`

Syntaxe

```
if (condition)
{
  ... // instructions a realiser si la condition est vraie (!=0)
} else {
  ... // instruction a realiser si la condition est fausse (=0)
}
```

! Attention !

Pour des questions de lisibilité, il faut indenter le code !!!

Les conditions sont construites à partir des opérateurs de comparaison et des opérateurs binaires :

Exemple 12

```
if ( a==1 && b<=20 ) {
    printf("je suis ici");
} else {
    printf("je suis la");
}
```

Variantes des syntaxes conditionnelles

Sans sinon :

```
if (condition)
{
    ...
}
```

Conditions sinon si :

```
if (condition1) {
    ...
} else if (condition2) {
    ...
} else if (condition3) {
    ...
} else {
    ...
}
```

Exemple 13

```
int i=2;
if (i>0) {
    printf("i (%d) est strictement positif \n",i);
} else if (i<0) {
    printf("i (%d) est strictement negatif \n",i);
} else {
    printf("null");
}
```

Exercice 2 () *Écrire en C une suite d'instructions permettant de faire ceci :*

```
soit c un caractère,
si c est la lettre 'h' ou 'a', alors afficher ''Aide'',
si c est la lettre 'q', alors afficher ''quitter'',
dans tout les autres cas, afficher ''autre''.
```

Opérateur triadique

Pour les conditions simples, du type "si vrai telle valeur, sinon une autre valeur", alors il est possible d'utiliser un opérateur particulier (que je nomme "triadique" parce que je me souviens plus de son vrai nom dans la norme ANSI ...). Sa syntaxe est la suivante :

(*condition* **?** *valeur_si_vrai* **:** *valeur_sinon*)

Par exemple, par calculer le maximum de deux valeurs x et y vous pouvez écrire :

```
double max = (x>y ? y : x );
```

1.4.2 Les structures itératives

Les boucles `while`

Une boucle permet de répéter des instructions similaires tant qu'une condition est vraie :

```
while( condition )
{
  ... // instructions
}
```

Exemple 14

```
int i, j;
i=0;
j=6;
while( i!=j ) {
  i++;
  j--;
}
printf("%d %d\n", i,j);
```

Le tableau suivant illustre l'évolution des variables au cours des boucles :

Ligne courante	i	j	test i!=j
1	?	?	
2	0	?	
3	0	6	
4	0	6	VRAI
5	1	6	
6	1	5	
4	1	5	VRAI
5	2	5	
6	2	4	
4	2	4	VRAI
5	3	4	
6	3	3	
4	3	3	FAUX
8	3	3	

Exercice 3 () *Cet exemple utilisant la fonction* `sqrt` *nécessite l'utilisation de la librairie mathématique : ajouter* `#include <math.h>` *en haut de votre programme.*

```
float f=81.0;
int nbTours=0;
while( f>5 ) {
  f = sqrt(f);
  nbTours++;
}
```

Question a) Faire un tableau qui illustre l'évolution des variables au cours des boucles !

Question b) Quel est en résultat ?

Question c) Même question avec la condition : `f<5`

Exercice 4 () *Écrire une suite d'instructions en C réalisant l'algorithme suivant : compter le nombre de caractères que contient une chaîne de caractères* `s` *(rappel : le dernière caractère est nécessairement le caractère \0)*

La syntaxe suivante est également possible pour une boucle `while`. Dans cette syntaxe la condition se place en fin de boucle. Ceci peut être pratique lorsqu'il faut effectivement réaliser une première iteration avant un test.

```
do{
  ...
} while(condition)
```

Boucles *for*

Une autre manière d'écrire une boucle est d'écrire un `for`. Le principe d'un `for` est de parcourir un ensemble de valeurs.

```
for( initialisation ; condition; incrementation) {
  ... // instructions
}
```

Pour comprendre ce que fait cette structure de contrôle, il faut voir que celle-ci peut être réécrite sous la forme d'un `while` de la manière suivante :

```
 initialisation ;
while(condition) {
  ... // instructions
  incrementation;
}
```

Exemple 15

Voici un exemple très usuel pour traiter les cas pour `i` *allant de 0 à 25 :*

```
int i;
for(i=0; i<25; i++) {
  printf("%d",i);
}
```

L'initialisation, la condition et l'incrémentation peuvent contenir des "instructions" complexes :
– Il est possible de déclarer des variables dans l'initialisation (*cf.* problèmes de portée des variables), par exemple :

```
for(int i=0; i<25; i++) {
  printf("%d",i);
}
```

Cette écriture se croise assez souvent et est plutôt pratique. Néanmoins, cette notation n'est pas toujours acceptée par les compilateurs.

- La condition peut contenir une combinaison de conditions (combinées avec des && et des ||). Par exemple :

```
for(int i=0; i<25 || c='\0'; i++) {
  printf("%d",i);
}
```

- L'initialisation et l'incrémentation peuvent contenir l'initialisation et l'incrémentation de plusieurs variables (séparées par des virgules), par exemple :

```
int i,j;
for(i=0, j=10; i<25; i++, j--) {
  printf("%d %d",i, j);
}
```

Exemple 16

Exemple d'incrémentation non linéaire :

```
int j=100;
double d= 13;
for(j=16; j >= 1 ; j=j/2 ) {
  d=d/2;
}
printf("result~: d=%f\n",d);
```

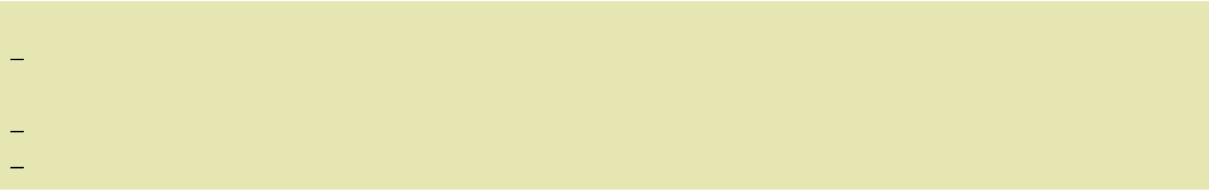
Exemple 17 - Calcul de π

Dans cet exemple, on illustre les boucles sur un problème du calcul des décimales de π.
Tout d'abord, quelques rappels de mathématiques et plus précisément sur les séries numériques. On peut montrer que :

$$\frac{\pi}{6} = \sum_{k=1}^{\infty} \frac{1}{k^2}$$

C'est à dire que si on somme $\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \ldots$ alors on s'approche progressivement de la valeurs $\frac{\pi}{6}$. Pour approcher la valeur de π, une méthode consiste à calculer cette somme avec le plus de termes possibles.
Le programme ci-dessous réalise le calcul de cette somme.

```
int nbtermes=10000, i;
double somme = 0;
for(i=1; i<nbtermes ; i++ ) {
  somme=somme + 1/((double)(i*i));
}
printf("pi~=%lf\n",6*somme);
```



Exercice 5 (Approximation polynomiale (⋆)) *Le but de l'exercice est de construire un programme qui fait l'approximation de la valeur $\frac{1}{1-x}$ pour tout x tel que $|x| < 1$. Pour cela, on utilise les développements limités*[5] *qui permet une approximation polynomiale d'ordre n avec la formule suivante :*

$$\forall x, |x| < 1, \ \frac{1}{1-x} \approx 1 + x + x^2 + \ldots + x^{n-1}$$

L'approximation qui est faite par l'équation ci-dessus est faite à x^n. Avec $|x| < 1$, plus n est grand, plus on s'approche de la valeur réelle de $\frac{1}{1-x}$.

Question a) Déduire un algorithme pour calculer l'approximation de $\frac{1}{1-x}$ à l'ordre n, où x et n seront des variables du programme.

Question b) Traduire l'algorithme en C.

Question c) Déduire un algorithme pour calculer l'approximation de $\frac{1}{1-x}$ avec une précision ϵ, où x et ϵ seront des variables du programme.

Question d) Traduire l'algorithme en C.

Parcours d'un tableau

L'utilisation des tableaux est intéressant pour faire des opérations similaires sur tous les éléments du tableau. Pour abstraire des traitements, il est donc utile de faire des parcours de ce tableau grâce à des boucles.
Voici quelques exemples classiques qu'il faut avoir bien compris :
- Notez bien les limites de parcours donnés dans la boucle : commencez avec une valeur d'indice de 0 et terminer en testant
- la variable `i` sert d'indice de la case du tableau : à chaque tour de boucle on traite la `i`-eme case, mais la valeur de `i` change

Exemple 18 - Initialisation des valeurs d'un tableau

L'exemple ci-dessous initialise un tableau de taille 23 avec les valeurs 1, 2, 3, ... 23.

```
double tab[23];
int i;
for( i=0; i<23; i++) {
  tab[i]=i+1;
}
```

Exemple 19 - Traitement des valeurs d'un tableau

À la suite de l'exemple précédent, voici un exemple qui utilise les valeurs successives du tableau pour réaliser une tâche (sommer les 23 premiers entiers).

```
int i;
int somme=0;
for( i=0; i<23; i++) {
  somme = somme + tab[i];
}
printf("somme : %lf\n", somme);
```

Équivalent avec un `while` *:*

```
int i=0; //attention bien initialiser l'iterateur !!
int somme=0;
while(i<23) {
  somme = somme + tab[i];
  i++; //attention ne pas oublier d'incrmenter votre iterateur
}
printf("somme : %lf\n", somme);
```

Exemple 20

L'exemple ci-dessous affiche tous les caractères de la chaîne, un à un. Ici, la condition d'arrêt de la boucle ne se fait pas sur la longueur de la chaîne, mais lorsqu'on tombe sur un caractère nul.

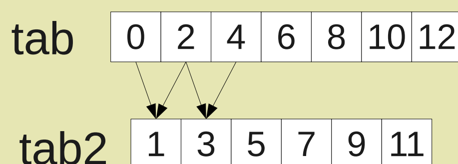
```
char string[] = "ceci est un texte";
int i;
for( i=0; string[i]!='\0'; i++) {
  char c= string[i] // Utilisation de la variable i comme indice
  printf("caractre la position %d : %c\n", i, c);
}
```

Opérations sur les indices

C'est là que l'algorithmique se complexifie un peu. Mais, bien évidemment, l'algorithmique commence à être intéressante quand on traite ce genre de problème.

Exemple 21

Voici un exemple plutôt simple. On commence par initialiser une tableau **tab**, *puis on construit un second tableau* **tab2**. *La ligne intéressante est la ligne 9 : sur cette ligne, on indique que dans la ième case du tableau* **tab2**, *on met la moyenne de la case de droite et de la case de gauche du tableau* **tab**.



```
double tab[7], tab2[5];
int i;
// Initialisation du tableau tab avec des valeurs de 0   12
for( i=0; i<7; i++) {
  tab[i]=2*i;
}

//Construction de tab2
for( i=0; i<5; i++) {
  tab2[i]=(tab[i-1]+tab[i+1])/2; // <- utilisation d' opration sur les indices
}
```

Exercice 6 (Traitement d'une chaîne de caractères) *On donne le programme ci-dessous :*

```
char text[] = "voici un texte";
char c;
for(int i=0; c != '\0'; i+=2) {
  c=text[i];
   printf("%c?", c);
}
printf("\n");
```

Question a) Que fait cet exemple ?

Question b) Réécrire cet exemple sous la forme d'une boucle `while`

1.4.3 Combiner les structures de contrôle

Imbrication de structures de contrôle

Les structures de contrôle peuvent s'imbriquer à l'infinie, par exemple :

```
for (... ; ... ; ...) {
  if (...) {
    //...
  } else {
    while( ...) {
      //..
    }
  }
  while (...) {
    //...
  }
}
```

! Attention !

L'indentation est nécessaire pour que le code soit lisible !!

Exercice 7 (Double boucles) *Exemple de double boucles :*

```c
for (int i=0; i< 4; i++) {
  for (int j=0; j <4; j++) {
    // faire quelque chose avec i et j
  }
}
```

Question a) Faire un tableau d'évolution des variables au cours de l'exécution du programme

```c
int s=0;
for (int i=0; i< 10; i++) {
  s-=i;
  for (int j=0; j <10; j++) {
    s++;
  }
}
```

Question b) Faire un tableau d'évolution des variables pour le début de l'exécution du programme

Question c) À partir du fonctionnement compris des boucles, indiquer quel sera la valeur finale de la variable s ?

`break`, `continue` : modification du déroulement d'une boucle

L'instruction break permet au programme de sortir directement de la boucle **la plus proche**.

```c
for (... ; ... ; ...) {
  ... // instructions
  if ( condition ) {
    break;
  }
}
```

L'instruction continue permet de passer directement à la fin de la boucle et de passer au tour suivant :
- pour les `for`, l'incrémentation est réalisée,
- les conditions sont évaluées avant de rentrer dans le tour de boucle suivant.

```c
while (...) {
  ... // instructions
  if ( condition ) {
    continue;
  }
  ... // instructions 2
}
```

Exemple 22 - Recherche avec arrêt

L'exemple suivant permet d'illustrer un ***algorithme de recherche avec arrêt****.*
Le programme suivant recherche une occurrence du caractère 'e' dans une chaîne de caractère et affiché "présent" ou "non-présent". Dès qu'on a trouvé la première occurrence de 'e' alors, ca ne sert à rien de continuer, alors on sort de la boucle. L'algorithme ressemble alors à :

```
char *str="chaine de test";
int i=0;
while( str[i]!= '\0' ) //test si c'est la fin de la chaine
{
  if( str[i] == 'e' ) {
    //ICI, on c'est qu'il existe un 'e'
    break;
  }
  i++;
}
// ICI : on ne sait rien !!!!
```

On pourrait afficher "présent" lorsqu'on est à la ligne 10, mais à quelle ligne peut-on afficher "non-présent" ? Car à la ligne 10, on ne sait pas comment on est arrivé là. On peut être arrivé là après un break, où bien simplement à la fin de la boucle, auquel cas, on n'a pas trouvé de 'e' !!

La seule solution est de retenir dans une variable qu'on a bien trouvé un 'e' dans la chaîne de caractères, et le programme résultant est celui-ci :

```
char *str="chaine de test";
int i=0;
int found=0;
while( str[i]!= '\0' ) //test si c'est la fin de la chaine
{
  if( str[i] == 'e' ) {
    //ICI, on c'est qu'il existe un 'e'
    found=1;
    break;
  }
  i++;
}
// ICI : on sait que si found est faux, alors on n'a rien trouve
if( found) {
  //ICI on sait que un 'e' a ete trouve
  printf("present\n");
} else {
  //ICI on sait qu'aucun 'e' n'a ete trouve
  printf("non-present\n");
}
```

Exercice 8 ()

```
for(int i=0; i< 4; i++) {
  for(int j=0; j <4; j++) {
    if( j>=i ) {
      break;
    }
  }
}
```

Question a) Faire un tableau d'évolution des variables au cours de l'exécution du programme.

Question b) Modifier le programme pour supprimer le `break` et utiliser un `while`.

Question c) Répondre de nouveau aux deux questions précédentes en remplacant le `break` par un `continue`.

1.4.4 Écritures courtes

Lorsque le code qui est entre crochet ne comporte qu'une seule instruction, il est possible de supprimer les accolades. Vous pouvez donc rencontrer ces écritures, mais il est déconseillé de les utiliser. Les codes suivants sont équivalents :

```
for(int i=0; i<max; i++) {
  printf("%d\n",i);
}

for(int i=0; i<max; i++)
  printf("%d\n",i);

for(int i=0; i<max; i++) printf("%d'\n",i);
```

Les codes suivants sont également équivalents :

```
if (...) {
  printf("vrai\n");
} else {
  printf("faux\n");
}

if (...)
  printf("vrai\n");
else
  printf("faux\n");

if (...) printf("vrai\n"); else printf("faux\n");
```

! Attention !

L'imbrication des structures de contrôle (en particulier les structures conditionnelles) peuvent être ambiguë à lire ! Par exemple, le programme suivant peut être lu de plusieurs manières

```
if(a==1)
  printf("A\n");
else if(a==2)
  printf("B\n");
else
  printf("C\n");
```

Mieux vaut mettre des accolades systématiquement sur les blocs d'instructions

1.5 Les entrées/sorties

Les instructions d'entrée/sortie (*in/out*) désignent les instructions qui permettent à un programme d'échanger des informations avec un utilisateur. Par défaut, le C (norme ANSI 89) ne propose pas d'instructions simples pour les entrées/sorties. Néanmoins, toutes les distributions usuelles de C disposent de la librairie standard <stdio.h> [6]. Cette librairie définie des instructions d'entrée/sortie que nous allons présenter dans cette partie du cours.

6. Détails de la librairies <stdio.h> : www.cplusplus.com/reference/clibrary/cstdio/

De part la proximité du C avec les Unix et son bas niveau, les entrées/sorties possibles sont simplement possible sous la forme de texte (pour dessiner sur un écran, il faut utiliser des librairies très évoluées comme la Xlib ou les librairies Motif[7], gtk[8] ou ncurses[9]).
Un programme peut écrire ou lire du texte :
- à partir d'un terminal d'exécution,
- à partir de fichier.

Pour l'affichage dans un terminal, on parle de "sortie standard" (nommée stdout), mais vous pouvez également affichée sur la sortie "erreurs" (nommée **stderr**). L'"entrée standard" (nommée **stdin**) désigne le canal par lequel les caractères tapées au clavier peuvent être récupérés. C'est concepts sont très liés aux systèmes Linux. Ils ne vous sont donc certainement pas très familiers.

1.5.1 L'affichage d'un texte dans un terminal avec `printf`

L'instruction **printf** permet de faire des affichages (de texte) dans la sortie standard du terminal. Sa syntaxe est la suivante :

int printf(const char *format [, arg [, arg]...]) ;

Contrairement à la plupart des fonctions C, la fonction **printf** n'a pas un nombre fixé de paramètres : elle en a au moins un (*la chaîne de format*) ou plus (les arguments). De ce point de vue, c'est une des fonctions C les plus complexes que vous devriez rencontrer.
Dans la suite, on présente donc cette fonction par des exemples pratiques usuels.

Affichage d'un texte simple (sans variable)

Le premier argument de la fonction doit toujours être une chaîne de caractères. Ce sera la chaîne de caractères qui indique ce qui va être écris dans le terminal.
Cette chaîne de caractères peut être directement écrite entre les parenthèses de la fonction, ou bien dans une variable (de type `char *` ou `const char *`[10]).
Il est préférable de systématiquement terminer la chaîne de caractères que vous souhaitez afficher par le caractère spécial \n qui indique qu'il faut faire un saut de ligne. Si vous ne le faites pas, d'une part, ce sera illisible et, d'autre part, vous risquez de ne voir apparaître vos textes qu'une fois le programme terminer et de croire que c'est un problème de programmation ! alors que (pour une fois ...) non !

Exemple 23

```
1 printf("le texte que je souhaite afficher\n");
```

ou

```
1 char texte[]="le texte que je souhaite afficher";
2 printf( texte );
```

Les caractères spéciaux utiles :

7. Motif : www.openmotif.org
8. GTK : www.gtk.org
9. NCurses : www.gnu.org/software/ncurses/ncurses.html
10. Le "const" signifie que le contenu de l'emplacement mémoire de la variable ne doit pas être modifié, c'est une manière de programmer en "protégeant" son code.

\n	Fin de ligne
\t	Tabulation

Remarque 6

L'affichage est, généralement, mis à jour uniquement lorsqu'on demande d'afficher un retour à la ligne !

Remarque 7

Vous devez éviter les accents dans les chaînes de caractères que vous voulez afficher. Ils risquent de ne pas être bien retranscrits.

Affichage d'une variable

Pour afficher une variable, il faut toujours utiliser une première chaîne de caractères dans laquelle vous aller décrire comment afficher une variable, et ensuite, le second argument de la fonction indiquera le nom de la variable à afficher.
L'affichage d'une variable se fait de la sorte :

```
printf("%...", arg);
```

Exemple 24

```
int i;
printf("%d", i);
```

- % est un caractère spéciale indiquant qu'il faut écrire dans le terminal la valeur d'une variable,
- %d indique qu'il faut afficher une variable de type int (*cf.* tableau suivant pour les autres codes possibles)
- la variable affichée est celle qui est donnée en second argument de la fonction (ici i) !

C'est au programmeur d'indiquer comment afficher une variable en disant s'il s'agit d'un entier, un flottant ou un caractère. On parle de "code de format" :

%d, %i	Nombre entier (relatif ou non)
%ld, %li	Nombre entier long (relatif ou non)
%f, %g, %e	Nombre à virgule flottante
%lf, %lg, %le	Nombre à virgule flottante double précision
%c	Un caractère
%s	Une chaîne de caractères
%x	Affichage de la valeur héxadécimal

L'affichage de nombre à virgule flottante peut se paramétrer à volonté (nombre de chiffre significatifs, nombre de chiffre avant et après la virgule, notation exponentielle...). Il faut se référer à la documentation[11] de la fonction `printf` pour tous ces détails.

Exemple 25

Quelques exemples d'affichage de nombres à virgule flottante :

Avec `printf("%f", x)`
- *si x=1.2345,*
  `1.234500`
- *si x=12.3456789,*
  `12.3456789`

Avec `printf("%10f", x)`
- *si x=1.2345,*
  `__1.234500`
- *si x=12.345,*
  `_12.345000`
- *si x=12.345e5,*
  `1234500.00000`

Avec `printf("%10.3f", x)`
- *si x=1.2345,*
  `_____1.235`
- *si x=1.2345e5,*
  `__1234.500`

Avec `printf("%e", x)`
- *si x=1.2345,*
  `1.234500e+000`
- *si x=123.45,*
  `1.2345000e+002`

Avec `printf("%12.4e", x)`
- *si x=1.2345,*
  `_1.2345e+000`
- *si x=123.456789e8,*
  `_12346e+010`

On peut faire des conversions directement par l'affichage. Par exemple :

```
char a='c';
printf("%d", c);
```

affiche le code ascii du caractère.

! Attention !

11. ou de regarder sur internet : par exemple sur le site `www.cplusplus.com`.

Le compilateur de vérifie pas que ce qu'il doit vous afficher correspond aux variables.

```
double f;
printf("%s", f); //Aucun resultat garantis !!
```

Formatage d'une chaîne de caractères

On peut mélanger le texte et les variables. L'ordre des paramètres doit correspondre à l'ordre des insertion à faire dans la chaîne de caractères.

```
char texte[]="suite du texte";
int i=4;
float f=45.6;
printf("ceci est un entier %d, la un float (%f), et voici la %s.\n", i, f, texte);
```

Résultat de l'affichage :

```
> ceci est un entier 4, la un float (45.6), et voici la
suite du texte.
```

! Attention !

Le compilateur ne vérifie pas si le nombre d'arguments est correct.

Exemple 26

Exemple d'affichage de nombre sous différentes formes :

```
/* printf example */
#include <stdio.h>
int main()
{
  printf("Characters: %c %c \n", 'a', 65);
  printf("Decimals: %d %ld\n", 1977, 650000L);
  printf("Preceding with blanks: %10d \n", 1977);
  printf("Preceding with zeros: %010d \n", 1977);
  printf("Some different radixes: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);
  printf("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);
  printf("Width trick: %*d \n", 5, 10);
  printf("%s \n", "A string");
  return 0;
}
```

Le résultat de l'affichage sera alors :

```
Characters: a A
Decimals: 1977 650000
Preceding with blanks:  1977
Preceding with zeros: 0000001977
Some different radixes: 100 64 144 0x64 0144
floats: 3.14 +3e+000 3.141600E+000
Width trick:  10
A string
```

1.5.2 Lire une valeur du clavier avec `scanf`

L'instruction scanf permet de lire (du texte) à partir de l'entrée standard (clavier utilisé depuis le terminal). Sa syntaxe est la suivante :

int scanf(const char *format [, arg [, arg]...]) ;

L'instruction suivante illustre la lecture d'un entier depuis l'entrée standard :

```
int n;
scanf ("%d",&n);
```

Dans cet exemple, le chiffre qui sera donné par l'utilisateur (et validé par l'utilisation de la touche `Entrée`) permettra d'affecter une valeur à la variable **n**.
Cette fonction montre une certaine ressemblance avec **printf**, et fait appel aussi à des "codes de format". Globalement, les codes formats présentés dans la section précédente peuvent être utilisés avec la même sémantique (par exemple, **%d** parle d'un entier, **%lf** d'un long float).
En revanche, les arguments définissant les variables dont vous cherchez à modifier la valeur doivent être systématiquement précédés d'un &. On verra plus loin que cette expression désigne un pointeur sur la variable **n**.

! Attention !

Dans la fonction **scanf**, vous devez mettre un & avant le nom de la variable dont vous souhaitez modifier la valeur.

1.5.3 Repérer les erreurs de saisie de l'utilisateur

Lorsque le type de la variable ne correspond pas au "code de format", la fonction **scanf** retourne la valeur 0 pour indiquer qu'elle n'a pas reconnue le format annoncé dans l'entrée standard.
Ainsi, le code suivant permet de récupérer une erreur de saisie :

```
int main(void) {
  int n=0;
  printf("Donne moi un entier\n");
  if( !scanf("%d",&n) ) {
    //ici, l'utilisateur a fait une erreur
    printf("Il aurait fallu donner un nombre\n");
    return 1;
  }
  //Ici, l'utilisateur a bien fait son job !
  printf("bien\n");
  return 0;
}
```

Remarque 8

Aucune erreur ne peut être rattrapée si vous demandez un texte à l'utilisateur et que, par exemple, il rentre un nombre. Un nombre sera alors vu comme une suite de caractères figurant des chiffres !

Exercice 9 (Calcul IMC) *Écrire un programme demandant à l'utilisateur d'entrer son prénom, son poids et sa taille et qui affiche ensuite à l'écran le prénom, et l'IMC de la personne.*
L'IMC sera caculée par la formule ci-dessous :

$$IMC = \frac{poids}{taille^2}$$

Exercice 10 ("Le capital") *Dans le même esprit, écrire un programme qui calcule le capital A produit par x euros, placés au taux r au bout de n années, avec :*

$$A = x(1+r)^n$$

Aide : Dans la librairie <math.h>*, vous pourrez utiliser la fonction* `double pow(double a, int n)` *qui permet de calculer a à la puissance n (a^n).*

Exercice 11 (Calcul en place) *On souhaite écrire un petit programme qui calcule la moyenne et l'écart type d'une série de notes. Les notes sont saisies une à une. À chaque saisie on affiche la moyenne courante et l'écart type courant.*
Dans cet exercice, vous demande un algorithme qui n'utilise pas de tableau. Une solution "simpliste" serait effectivement de retenir toutes les valeurs saisies et de refaire à chaque fois les calculs des moyennes et écart-types. Mais il y a beaucoup plus malin que cela (on parle d'algorithme "en place"). Les questions ci-dessous vous dirigent vers la construction de l'algorithme.

Question a) Identifier et établir les formules mathématiques qui permettent de calculer ces deux grandeurs.

Question b) Exprimer ces formules au rang N en fonction du rang N-1.

Question c) Rechercher les variables nécessaires à la saisie et à mémoriser l'état précédent (rang N-1).

Question d) Décrire l'algorithme du programme qui effectue la boucle de saisie.

Question e) Indiquer comment on termine le programme.

Question f) Écrire le programme C.

Question g) Quel est l'avantage de la solution proposée par rapport à une solution utilisant un tableau

1.5.4 Lectures et écritures dans des fichiers : `fprintf` et `fscanf`

Écrire ou lire dans un fichier est très similaire à écrire ou à lire dans les sorties/entrées standard. Les fonctions `printf` et `scanf` ont leur équivalent qui fonctionne de manière très similaire. Ces fonctionnalités sont disponibles grâce à la librairie <stdio.h>.

Les deux différences sont, d'une part, la désignation d'un fichier dans lequel il faudra écrire à l'aide d'une variable de type **FILE*** et, d'autre part, l'utilisation des fonctions **fprintf** et **fscanf** à la place de **printf** et **scanf** (je vous laisse noter la différence !).
Les étapes principales pour écrire ou lire dans un fichier sont le suivantes :

1. Ouvrir un fichier (en lecture ou en écriture) (fonction **fopen**)
2. Utiliser le fichier à l'aide de **fprintf** et de **fscanf**
3. Fermer le fichier (fonction **fclose**)

Ouverture et fermeture d'un fichier

La fonction **FILE *fopen(char *nomfic, char *mode)** ouvre le fichier dont le nom est donné comme premier argument, selon le mode d'ouverture précisé (**w** = écriture, **r** = lecture, **a** = ajout en fin de fichier) et l'assigne à un flux, i.e. à une variable de type **FILE** * (c'est un nouveau type qui désigne un fichier, on parle de *pointeur de fichier*).
Dans le programme ci-dessous, **nom_fic** est une chaîne de caractères qui contient le nom du fichier à ouvrir, et **fic** est une variable de type **FILE** *.

```
1 // ouverture du fichier
2 FILE *fic = fopen(nom_fic, "r"); // ouvrir en mode lecture
3 if (fic == NULL){
4   printf("Impossible d'ouvrir le fichier %s\n", nom_fic);
5 } else {
6   printf("Ouverture du fichier %s\n", nom_fic);
7 }
```

Les différents types d'accès à un fichier sont les suivants :
- **r** :ouverture d'un fichier en lecture, le fichier doit exister, autrement la fonction retourne la valeur **NULL** ;
- **w** :création et ouverture d'un fichier en écriture, si le fichier existe, son contenu est détruit ;
- **a** : ouverture d'un fichier en écriture à la fin du fichier, si le fichier n'existe pas, il est créé ;
- **r+** : ouverture d'un fichier en lecture et écriture, le fichier doit exister, autrement la fonction **fopen** retourne la valeur **NULL** ;
- **w+** : création et ouverture d'un fichier en lecture et écriture, si le fichier existe, son contenu est détruit ;
- **a+** : ouverture d'un fichier en lecture et en écriture à la fin du fichier, si le fichier n'existe pas, il est créé.

La fermeture d'un fichier utilise la fonction **void fclose(FILE *)**.

```
1 // fermeture du fichier
2 if ( fclose(fic)==EOF ){
3   printf("Probleme de fermeture du fichier %s", nom_fic);
4 } else {
5   printf("Fermeture du fichier %s\n", nom_fic);
6 }
```

Ici, la constante **EOF**, définie dans la bibliothèque <**stdio.h**>, caractérise la fin d'un fichier. En particulier, cette valeur est retournée par la fonction **fclose** lorsque la fermeture ne s'est pas bien passée.
La fonction **feof** de la bibliothèque <**stdio.h**> retourne une valeur non nulle (vrai) lorsque la fin d'un fichier est atteinte ou qu'une erreur de lecture est survenue. Cette deuxième possibilité est très souvent oubliée, et cela conduit à l'écriture de programmes faux.

Écriture et lecture d'un fichier

Des fonctions similaires à **printf** et à **scanf** existent pour écrire ou lire dans des fichiers. Leurs syntaxes sont les suivantes :

void fprintf(FILE *, const char *format [, arg [, arg]...]) ;
int fscanf(FILE *, const char *format [, arg [, arg]...]) ;

En ce qui concerne l'usage des chaînes de caractères formatés, le fonctionnement est identique aux fonctions **printf** et **scanf**. La seule différence est que ces fonctions doivent impérativement avoir comme premier paramètre un pointeur de fichier (**FILE** $*$) qui aura été préalablement construit grâce à l'instruction **fopen**.

Remarque 9

La fonction **fscanf** ne peut être utilisé que si un fichier qui est ouvert en écriture, et **fprintf** ne peut être utilisé que sur un fichier ouvert en lecture.

Un exercice "pratique"

Voici un exemple dans lequel on recopie (quasiment) un fichier bancaire. Ce fichier contient les noms des clients de la banque avec leur capital actuel. Ce fichier ressemble à cela :
`fichier.txt` :

```
Alice Machine 500
Albert Einstein 2300
Tryphon Tournesol 10
Thomas Guyet 300
Truc Much 450
```

```
1  #include <stdio.h>
2  int main()
3  {
4    char strnom [80], strprenom [80];
5    int valeur;
6    FILE *fin, *fout;
7    fin = fopen("fichier.txt","r");
8    fout = fopen("copie.txt","w+");
9    fscanf( fin ,"%s %s %d", strprenom, strnom, &valeur);
10   if( !strcmp(strnom, "Guyet") && !strcmp(strprenom, "Thomas") ) {
11     valeur += 10;
12   }
13   fprintf (fout, "%s %s %d", strprenom, strnom, valeur);
14   fclose (fin);
15   fclose (fout);
16   return 0;
17 }
```

Exercice 12 (Banco !) *Écrire le fichier* `copie.txt` *après l'exécution du programme. Que fait ce programme ?*

1.6 Correction des exercices

Solution à l'exercice 1

Question a) *contenu de tab*

- tab = [1, 2, 3, 0, 5, -1, -2, 0, -4, -5]
- tab = [7, 2, 3, 0, 5, -1, -2, 0, -4, -5]
- Erreur ! On ne peut pas affecter la partie gauche !
- tab = [7, 2, 3, 0, 5, -1, 6, 0, -4, -5]
- tab = [7, 2, 3, 0, 5, -4, 6, 0, -4, -5]
- Erreur ! tab[9]=-5 : hors des limites du tableau
- tab = [7, 2, 0, 7, 5, -4, 6, 0, -4, -5]

Solution à l'exercice 2 Dans un main :

```
void main() {
  char c;
  //...
  if (c=='h' || c=='a') {
    printf("Aide\n");
  } else if (c=='q') {
    printf("Quitter\n");
  } else {
    printf("Autre\n");
  }
}
```

Solution à l'exercice 3

Question a) *Tableau d'évolution*

Ligne courante	**f**	**nbTours**	test f>5
1	81	?	
2	81	0	
3	81	0	FAUX
4	9	0	
5	9	1	
3	9	1	FAUX
4	3	1	
5	3	2	
3	3	2	VRAI
6	3	2	

Question b) *Le résultat est donc* $f = 3$

Question c) *Avec* `f<5` *on ne rentre pas du tout dans la boucle et ni* `f` *ni* `nbTours` *n'est modifié*

Solution à l'exercice 4 Comptage du nombre de caractère d'une chaîne de caractères :

```
char S[200]; // Texte de taille maximum 200
int i=0;
while( S[i] != 0 ) {
  i++;
}
printf("Nombre de caractres : %d\n", i+1 );
```

Solution à l'exercice 6

Question a) *Le programme affiche le texte en masquant un caractère sur deux. Le texte affiché sera ainsi :* `v?i?i ?n ?e?t?`

Question b) *Récriture avec un while*

```
char text[] = "voici un texte";
char c;
int i=0;
while( c != '\0' )) {
  c=text[i];
  printf("%c?", c);
  i+=2;
}
printf("\n");
```

Solution à l'exercice 7

Question a)

Ligne courante	i	j
3	1	1
3	1	2
3	1	3
3	2	1
3	2	2
3	2	3
3	3	1
3	3	2
3	3	3

Solution à l'exercice 9

Programme attendu :

```
#include <stdio.h>
void main() {
  char prenom[10];
  int age;
  printf("Entrez votre prenom :");
  scanf("%s",&prenom);
  printf("\nEntrez votre age :");
  scanf("%d",&age);
  printf("\nBonjour %s, vous avez %d ans et vous avez vecu au moins %d jours\n", prenom, age);
}
```

Solution à l'exercice 10 Programme attendu :

```
void main() {
  float x, r, A;
  int n;
  printf("Taux d'interet ?");
  scanf("%f",&r);
  printf("Capital initial ?");
  scanf("%f",&x);
  printf("Nombre d'annees ?");
  scanf("%d",&n);
  A = x*pow(1+r,n);
  printf("Capital final = %f\n", A);
}
```

Solution à l'exercice 11

Question a) *Identifier et établir les formules mathématiques qui permettent de calculer ces deux grandeurs.*

$$\mu = n = \frac{(t_1 + t_2 + ... + t_n)}{n} \quad \sigma_n = \sqrt{\frac{((t_1 - \overline{t_n})^2 + (t_2 - \overline{t_n})^2 + ... + (t_n - \overline{t_n})^2)}{n}}$$

Question b) *Exprimer ces formules au rang N en fonction du rang N-1.*

$$\mu_n = \frac{(\mu_{n-1}(n-1) + t_n)}{n} \quad {\sigma_n}^2 = (\frac{n-1}{n})^2 {\sigma_{n-1}}^2 + \frac{n-1}{n^2}(\mu_{n-1} - t_n)^2$$

Question c) *Rechercher les variables nécessaires à la saisie et à mémoriser l'état précédent (rang N-1).*

- la note courante
- la moyenne du rang courant
- le sigma du rang courant

Question d) *Décrire l'algorithme du programme qui effectue la boucle de saisie.*

```
Tant que pas fini {
    Saisir nouvelle note
    Calculer nouvelle moyenne
    Calculer nouvel écart type
    Afficher moyenne
    Afficher ecart type
}
```

Question e) *Indiquer comment on termine le programme.* On peut faire la moyenne uniquement sur des saisies de nombres positifs, et arrêter lorsque la saisie est négative (par exemple)

Question f) *Écrire le programme C.*

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

int main(int argc, char *argv[])
{
    float moyenne;
```

```
8      float sigma;
9      float variance;
10
11     int note;
12     int i;
13
14     printf("Calcul de moyenne et de sigma.\nEntrez une note : ");
15     scanf("%d", &note);
16     i = 0;
17
18     while ((note >= 0) && (note <= 20)) {
19         if (i == 0) {
20             moyenne = note;
21             sigma = 0;
22         }
23         else {
24             variance = sigma * sigma;
25             variance = ((float)i / ((float)i + 1))
26                 * ((float)i / ((float)i + 1)) * variance
27                 + ((float)i / (((float)i + 1) * ((float)i + 1)) )
28                 * (moyenne - note) * (moyenne - note) ;
29             sigma = sqrt(variance);
30             moyenne = (moyenne * (float)i + note ) / ((float)i + 1);
31         }
32
33     /* impression */
34     printf("note : %d\n", note);
35     printf("moyenne : %f\n", moyenne);
36     printf("sigma : %f\n", sigma);
37
38     /* instructions de boucle */
39     i++;
40     scanf("%d", &note);
41     }
42   printf("au revoir.\n");
43   return 0;
44 }
```

Question g) *Quel est l'avantage de la solution proposée par rapport à une solution utilisant un tableau*
L'avantage de la solution proposée (sans tableau) est de ne pas dépendre du nombre de saisies. Une solution avec tableau impose de limiter le nombre de saisies en fixant la taille du tableau. Admettons qu'il faille 1 milliard de saisie, il faut un taille de 1 milliard ... c'est pas pratique (ni même possible). Alors qu'avec notre algorithme, quelque soit le nombre de saisie, l'espace mémoire (le nombre de variable) nécessaire sera toujours le même. On parle alors de calcul **en place**.

Solution à l'exercice 12 Alors voilà enfin un programme utile : le programme recopie la liste du fichier bancaire, mais ajoute 10 pour le compte de "Thomas Guyet" ...

CHAPITRE

2 Structures de données

Dans cette partie du cours, j'introduis la notion de structure de données (mots clés **struct**, **union**, **enum** et **typedef**) pour définir de nouveaux types à partir des types de base.
Dans la vie réelle, les données qui sont traitées dans les programmes ne peuvent pas toujours se réduire à des nombres ou des caractères. Les données sont très souvent structurées et organisées. Par exemple, si les numéros de facture d'une entreprise sont de la forme 021100576 avec comme les deux premiers chiffres qui indiquent le mois de l'années de la facture puis deux chiffres pour l'année, ensuite trois chiffres pour le numéro de la facture dans le mois et enfin deux chiffres pour coder le nom de l'employé qui a édité la facture. On peut choisir de représenter ce numéro de facture sous la forme d'un nombre entier. Dans ce cas, à chaque fois que l'ordinateur voudra connaître le mois d'édition de cette facture, il devra réaliser des opérations arithmétiques. Il semble plus raisonnable de structurer ces informations dans un seul bloc et avec les données de mois, années, numéro et employé clairement séparées, facilement accessibles sans calcul complexe.
L'objectif des structures de données est de faciliter la vie du programmeur en lui offrant la possibilité de définir ses propres structures de données composites.

2.1 Définir de nouveaux types avec `typedef`

L'instruction **typedef** permet de définir de nouveau type de données. Sa syntaxe est la suivante :

typedef *description_du_type nom_du_type ;*

Vous pouvez ainsi définir le type **bool** à partir des entiers ainsi :

```
typedef int bool;
bool b;
```

2.2 Types énumérés

2.2.1 Définition d'un type énumeré

Un type énuméré est constitué par une famille finie de nombres entiers, chacun associé à un identificateur qui en est le nom. Mis à part ce qui touche à la syntaxe de leur déclaration, il n'y a pas grand-chose à dire à leur sujet.
Les valeurs d'un type énuméré se comportent comme des constantes entières ; elles font donc double emploi avec celles qu'on définit à l'aide de **#define**. Leur unique avantage réside dans le fait que certains compilateurs détectent parfois, mais ce n'est pas exigé par la norme, les mélanges entre objets de types énumérés distincts (par exemple, ne pas mettre une valeur de type **mois_annee** dans une variable de type **jour_semaine**) ; l'utilisation de ces types permet alors d'augmenter la sécurité des programmes.
La syntaxe de la déclaration des énumérations est décrite dans les exemples suivants.

Exemple 27

La ligne ci-dessous introduit un type énuméré, appelé enum jour_semaine, *constitué par les constantes* lundi *valant 0,* mardi *valant 1,* mercredi *valant 2, etc.*

```
enum jour_semaine {lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche};
```

Ainsi, les expressions mardi+2 *et* jeudi *représentent la même valeur. Ces valeurs peuvent être directement utilisées dans le code.*
Pour définir une variable ayant ce type, il y a deux solutions :

– *sans* typedef

```
enum jour_semaine maVariableJour;
```

– *avec typedef (on définit un nouveau type)*

```
typedef enum jour_semaine {lundi, mardi, mercredi, jeudi, vendredi} type_jour_ouvre;
type_jour_ouvre maVariableJourOuvre;
```

Exemple 28 - Type booléen

```
typedef enum valeurs_booleenne {false, true} bool;
bool b=true;
```

Dans cet exemple, l'énumération permet de définir deux valeurs (conforme à l'usage usuel en C) : false *(avec la valeur 0) et* true *(avec la valeur 1), et de définir le type* bool.

Remarque 10

Lors de la déclaration des éléments énumérés on peut explicitement donner des valeurs entières à chacune des valeurs symboliques.
Par exemple :

```
enum Booleen {Vrai =1, Faux = 0};
```

2.2.2 Utilisation d'un type énumeré

Les type énumérés peuvent servir avec des switch à distinguer des comportements en fonction de la valeur d'une variable de ce type.

Exemple 29 - Type énuméré et switch

```
typedef enum {MR, MME, MLLE, DR} civilite;

void main() {
   civilite  c;

  //...   initialisation  de c


  printf("Bonjour ");

  //Suite de l'affichage en fonction de   civilit   :
  switch(c) {
    case MR:
      printf("Monsieur");
      break;
    case MLLE:
      printf("Madame");
      break;
    case MME:
      printf("Madame");
      break;
    case DR:
      printf("Docteur");
      break;
    default:
  }
  printf(", \n");
}
```

2.3 Les structures de données

Definition 5 - Structures

Les structures sont des variables **composées de champs** de types différents.

2.3.1 Définition d'une nouvelle structure

Pour définir une structure, il faut utiliser le mot clé **struct**. À la suite du mot-clé, on indique :

- facultativement, le nom que l'on souhaite donner à la structure (**Point3D** dans l'exemple). Ce nom de structure pourra être utilisé comme un nom de type.
- entre accolades : la liste des déclarations des champs de la structure séparés par des virgules. Chaque champs à un type et un nom (ça s'écrit comme la déclaration d'une variable). Chaque champ peut avoir un type quelconque (y compris un autre type structure, bien sûr).

La structure ci-dessous illustre un structure de point en 3 dimensions :

```
struct Point3D {
  double x,
  double y,
  double z
};
```

La définition d'une structure permet de définir de nouveau type en combinant le mot clé **struct** avec le mot clé **typedef** :

```
typedef struct {
  // declarations des champs
} nom_type;
```

! Attention !

Le nom du type définit par typedef se situe à la fin des accolades.

Exemple 30 - Structure Point2D

Nous allons définir d'un type de structure Point2D *et déclaration d'une variable* p *de type* Point2D.

```
typedef struct {
  double x;
  double y;
} Point2D ;

Point2D p;
```

Remarque 11 - Alignement des données

Contrairement aux tableaux, l'organisation en mémoire peut être plus complexe que la simple contiguïté des champs. Pour des raisons d'efficacité, certains compilateurs "décalent" les emplacements mémoire pour les "aligner" sur des multiples de tailles élémentaires.

2.3.2 Utilisation des variables-structure

Pour cette partie, on définit le type de structure ci-dessous :

```
typedef struct {
  int numero,
  char nom[32],
  char prenom[32],
  double salaire
} Fiche;
```

– Accès aux champs d'une structure : utilisation de la "notation pointée"

```
Fiche f1, f2;
//Mettre ici l' initialisation de f1 et f2
f1.numero = 1;
if( f1.salaire >f2.salaire ) {
  printf("%s gagne plus que %s", f1.nom, f2.nom);
} else {
  printf("%s gagne plus que %s", f2.nom, f1.nom);
}
```

– Accès aux champs d'une structure définie par un pointeur [1] : utilisation de la "notation fléchée"

```
Fiche *f1, *f2; // declaration de deux pointeurs sur des structures
//Mettre ici allocation des pointeurs f1 et f2 et initialisation du contenu des emplacements pointes
f1->numero = 1;
if( f1->salaire > f2->salaire ) {
  printf("%s gagne plus que %s",f1->nom, f2->nom);
} else {
  printf("%s gagne plus que %s", f2->nom, f1->nom);
}
```

Notez que pour l'allocation mémoire d'une structure, tout ce passe exactement pareil qu'avant. En particulier, la fonction sizeof va se charger de calculer la place nécessaire pour une instance de votre structure. Ainsi, dans ce cas, l'allocation ressemblerait à cela :

```
f1 = (Fiche *)malloc( sizeof(Fiche) );
```

– Initialisation d'une structure : l'initialisation d'une structure peut se faire manuellement (champs par champs en utilisant la notation pointée ou fléchée), ou bien à la déclaration ainsi :

```
Fiche f1 ={1, "Guyet","Thomas", 3500.0};
```

Exemple 31 - Exemple de structures complexes

– *une structure avec des champs diversifiés*

```
typedef struct {
  FILE *imagefile; // fichier image
  char *filename; // nom du fichier correspondant a l'image
  int height, width; // dimensions de l'image
  Point2D points[height*width]; // tableau de points de l'image (de taille < height*width)
} Image;
```

– *une structure d'un nœud d'arbre binaire*

```
typedef struct snode {
  snode *fd; // noeud fils droite
  snode *fg; // noeud fils gauche
} node;
```

– *une structure d'un élément d'une liste doublement chaînée de* double

```
typedef struct selem {
  selem *prec; // element precedent
  double value; // valeur contenu par l'element
  selem *succ; // element suivant
} elem;
```

Exercice 13 () *Nous voulons définir un type composé Joueur qui comprend 4 informations : le nom du joueur, le nombre de match joués, le nombre de match gagnés et le nombre de matchs restant à jouer.*

1. se référer au chapitre sur les pointeurs pour la compréhension de cette notation.

Question a) Définir la structure **Joueur**

Question b) Écrire une fonction **SaisieJoueur** pour saisir au clavier les informations d'un joueurs. Cette fonction retournera une variable de type **Joueur**

Question c) Écrire une fonction **Saisies** pour remplir un tableau de joueurs. Cette fonction prendra comme paramètre d'entrée **TabJ** : un tableau de joueur. La fonction ne retournera pas de résultat. Le nombre de joueur sera contenu dans une variable de pré-processing **NB_JOUEUR**

Question d) (⋆) Écrire **Calcul** pour calculer le nombre total de match restant à jouer en fonction du nombre de match que chaque joueur doit jouer (paramètre de la fonction) .

2.4 Correction des exercices

Solution à l'exercice 13

Question a) *Structure*

```
typedef struct {
    char nom[20];
    int nbmj;
    int nbmg;
    int nbmrj;
} Joueur;
```

Question b) *SaisieJoueur*

```
Joueur SaisieJoueur(){
    Joueur j;
    printf("Nom?\n");
    scanf("%s",&j.nom);
    printf("Nombre de matchs jous?\n");
    scanf("%d",&j.nbmj);
    printf("Nombre de matchs gagns?\n");
    scanf("%d",&j.nbmg);
    printf("Nombre de matchs restants?\n");
    scanf("%d",&j.nbmrj);
}
```

Question c) *Saisies*

```
#define NB_JOUEUR 3
void saisies(Joueur J[NB_JOUEUR]){
    int i;
    for(i=0; i<NB_JOUEUR; i++) {
      printf("JOUEUR %d\n", i+1);
      Joueur j = SaisieJoueur();
      J[i]=j;
    }
}
```

Question d) *Calcul*

```
int Calcul(Joueur J[NB_JOUEUR], int nb) {
    int i, deuxtotaljoue=0;

    int nbmatchs = NB_JOUEUR * nb;

    for(i=0; i<NB_JOUEUR; i++) {
      deuxtotaljoue+=J[i].nbmrj;
    }

    return nbmatchs-deuxtotaljoue/2;
}
```