Bases de la programmation :
Cours de C

1ère année

IUT de Villetaneuse.

Hanène Azzag, Frédérique Bassino, Bouchaïb Khafif, François Lévy,Mustapha Lebbah

28 février 2012

Table des matières

Cours 1

Présentation du C

1.1 Mon premier programme C

Python que nous avons d'abord appris est un langage **interprété**. Cela veut dire que quand on a écrit le source, on le passe au logiciel python qui le lit et l'exécute directement – que ce soit en utilisant "run script" avec Eric ou en tapant python monfic.py dans un terminal, c'est la même chose. Le C a un cycle d'utilisation différent : c'est un langage **compilé**. Il n'y a pas de logiciel capable de l'exécuter directement ; il faut d'abord le traduire dans le langage du processeur de la machine sur laquelle vous êtes, et ensuite le processeur pourra exécuter la traduction. Pour traduire, on utilise un programme (une commande Unix) qui s'appelle **gcc**. Un exemple sera plus facile à comprendre :

- Pour créer le programme source, se placer dans un répertoire TD1 et créer un nouveau fichier bonjour.c avec un éditeur de texte (gedit ou eric4 peuvent convenir)
- Tapez dans ce fichier le code suivant (les numéros de ligne ne font pas partie du code, bien sûr) :

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

  N'oubliez pas de sauver le fichier. Laissez l'éditeur de texte ouvert.
- Ouvrez un terminal, et changez de répertoire pour aller dans TD1. (tapez ls pour vérifier : vous devez voir bonjour.c)
- Dans le terminal, tapez

  gcc bonjour.c -o bonjour

  Vous venez de lancer gcc en lui demandant de lire bonjour.c, de le traduire en instructions pour le processeur et de mettre le résultat dans un fichier bonjour. C'est cela qu'on appelle compiler. Si vous tapez à nouveau ls, vous verrez un fichier bonjour qui n'existait pas avant.
- pour exécuter votre programme, restez dans le terminal et tapez

  ./bonjour

1.2 Quelques notions de C

En ce qui concerne les instructions, le langage C a beaucoup de ressemblances avec Python - vous verrez un peu plus de détails au prochain chapitre. Quelques différences :

- l'exécution du code ne commence pas nécessairement au début. Elle commence par une fonction qui s'appelle main() — quel que soit l'endroit où cette fonction est définie dans le code.

– les blocs de code ne sont plus repérés par l'indentation, mais par une accolade **{** au début et une autre **}** à la fin. C'est ainsi que le corps de la fonction main() se trouve entre accolades :

```
int main(void){
   printf ("bonjour") ;
    return 0 ;
}
```

– les instructions sont obligatoirement terminées par des " **;**". Les changements de ligne peuvent figurer au milieu d'une instruction – en particulier quand elle est trop longue.
– Contrairement à Python, le compilateur C ne devine pas le type des variables ; il faut le lui indiquer explicitement avant leur utilisation. Pour les entiers et les réels, on doit donc écrire

```
   int x, y, z ;
   float u, v, w ;
```

avant d'utiliser x, y, z, u, v, w. De la même façon, la fonction main() comporte le type de ses arguments int main(void) veut dire que dans ce programme on n'attend pas d'argument et que main() renvoie un entier ;
– Il y a une autre version de main qui s'écrit int main(int argc, char* argv[]) ce qui veut dire que le programme attend un entier et un tableau de chaînes (vous aurez bientôt l'explication du *argv[]). Quand on utilise cette seconde version, argv[1] est le premier mot qui suit le nom du programme à l'appel, argv[2] le deuxième mot qui suit ce nom, etc.
– en C, l'opération + sur les chaînes n'existe pas. Au lieu de cela, printf utilise une *chaîne de format* : dans le premier argument, on peut mettre des *marqueurs de place* qui seront remplacés par les arguments qui suivent. par exemple :

```
   int x ;
   x = 3 ;
    printf ("Il y a %d candidats", x) ;
```

écrit "il y a 3 candidats". Les principaux marqueurs de place sont %d (entiers), %f (floats), %s (chaînes). Voici un exemple plus développé :

```
   int nb = 25 ; float moy = 10.25 ; char sec[] = "S3 groupe 2'' ;
   printf("Il y a %d étudiants en %s.  Leur moyenne est %f", nb, sec, moy) ;
```

écrit "Il y a 25 étudiants en S3 groupe 2. Leur moyenne est 10.25". Remarquez bien que les mrqueurs de place sont remplis dans l'ordre par les arguments qui suivent, et que les types doivent correspondre.

On continue à utiliser les changements de ligne et l'indentation pour la lisibilité, et c'est fortement recommandé. Mais pour le compilateur, il n'y a pas de différence entre blanc, tabulation ou passage à la ligne : cela ne sert qu'à séparer des mots [1].

Le C peut comme tous les langages utiliser des fonctions toutes faites s'il a accès à une bibliothèque. Une bibliothèque C, ce sont des fonctions qu'on a conservées dans un format spécial, proche du langage du processeur, pour s'en resservir dans d'autres programmes. Pour afficher, nous utiliserons la fonction printf() qui se trouve dans la bibliothèque standard – vous venez de voir un exemple. En TD, nous verrons une autre fonction d'affichage, puts(). Dans les prochains cours, nous introduirons quelques fonctions qui nous permettront de faire des programmes plus intéressants, avant d'apprendre à les écrire. Le `#include` sert à incorporer des fichiers de description des bibliothèques : printf () est dans la bibliothèque standard, donc on incorpore le fichier stdio.h qui décrit cette bibliothèque (attention, stdio.h est un fichier source, alors que la bibliothèque standard est déja en langage machine).

1. mais il ne peut y avoir de passage à la ligne au milieu d'une chaîne (c'est \n qui indique qu'on veut afficher un changement de ligne dans la chaine).

Nous utilisons aussi des remarques : tout ce qui est après // et sur la même ligne n'est pas lu par le compilateur. De même pour tout le texte entre /* et */. Cela sert seulement à aider le programmeur qui relit – et c'est déjà beaucoup, c'est même indispensable. Par exemple :

```
1  #include <stdio.h> /* on utilise la bibliothèque d'entrées-sorties standard,
2    il faut donc inclure son fichier de description */
3  int main(void) {
4     printf ("Bonjour !\n");
5    return 0 ; // Pour Unix, 0 veut dire que tout s'est bien passé
6  }//fin du main
```

1.3 Trouver les erreurs

Il faut distinguer deux sortes d'erreurs.

1.3.1 Erreurs de syntaxe

Les premières sont celles qui se produisent à la compilation (quand vous passez le source à gcc). Le compilateur signale une erreur *parce qu'il n'arrive pas à traduire*. On appelle cela une erreur de syntaxe. Il faut **lire le message** pour savoir à quel endroit il n'arrive pas à traduire. Si par exemple vous compilez

```
1  #include <stdio.h>
2  int main(void) {
3     pintf ("Bonjour !\n") ;
4    return 0 ;
5  }//fin du main
```

vous obtenez

```
Undefined symbols:
  "_pintf", referenced from:
      _main in ccTs8cj8.o
```

qui vous dit bien que pintf n'existe pas (il manque le r, mais le compilateur ne peut pas savoir à quoi vous pensiez). Si vous essayez à la place

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

vous obtenez

```
essai.c: In function 'main':
essai.c:3: error: syntax error before ';' token
```

Autrement dit, à la ligne 3, quand le compilateur arrive au " ;" il ne s'en sort plus. C'est donc qu'il y a une erreur avant – effectivement, il manque la parenthèse fermante.

1.3.2 Erreurs à l'exécution

Quand le source se laisse compiler sans problème, on n'en a pas fini pour autant avec les erreurs. Prenons un programme correct pour afficher un nombre (le %d est un code spécial pour dire d'afficher le nombre qui est dans l'argument suivant) :

```
  #include <stdio.h>
int main(void){
        printf("%d\n", 3/2);
    return 0 ;
}
```

Il compile sans problème, et à l'exécution il affiche 1. Si par contre vous écrivez

```
#include <stdio.h>
int main(void){
       int x ;
       x = 0 ;
        printf("%d\n", 3/x);
       return(0);
}
```

vous pourrez compiler sans problème. Mais à l'exécution, vous obtiendrez :

```
Floating point exception
```

Autrement dit le processeur n'a pas pu faire une opération — en fait la division par 0. Parmi les erreurs à l'exécution, on trouve aussi les boucles infinies, la lecture de fichiers qui n'existent pas, etc.

1.3.3 Erreurs de conception

Dans d'autres cas, l'exécution ne produit pas de message d'erreur, mais le résutat n'est pas ce qu'on attendait. Par exemple :

```
  #include <stdio.h>
int main(void){
       int x ;
       x = 0 ;
        printf("%d\n", "3");
       return(0);
}
```

On compile ce programme, il s'exécute et il affiche 8185 ! ! ! (sur ma machine). On a fourni au %d de printf() la **chaîne** "3" au lieu de **l'entier** 3.

Il y a de multiples erreurs de conception : une variable mise pour une autre, une boucle dont le test d'arrêt est mal formulé, un test mal rédigé, etc. Quand on n'a pas de chance, l'exécution se déroule sans message d'erreur, mais le résultat n'est pas ce qu'on voulait. Quand on a fini un programme, **il faut toujours le tester** avant de dire qu'il est bon.

1.4 Compléments

1.4.1 Plusieurs fichiers sources

Quand on commence à vraiment programmer, on ne peut plus mettre tout le code dans un seul fichier. Il faut alors savoir que la traduction comporte en fait trois étapes, que l'on peut réaliser l'une après l'autre :

gcc -S bonjour.c compilation proprement dite. On voit apparaître un fichier bonjour.s qui est du langage assembleur (lisible, même s'il n'est pas très compréhensible).

gcc -c bonjour.s assemblage. On voit apparaître le fichier bonjour.o qui est un fichier objet, c'est à dire du langage machine pas tout à fait fini. Ca n'est plus du tout lisible.

gcc bonjour.o -o bonjour edition de lien. Permet de rassembler plusieurs fichiers en un seul programme, ce qui est indispensable quand on écrit de gros programmes. On voit apparaître le fichier bonjour qui est un fichier exécutable (du langage machine fini).

./bonjour exécution du programme obtenu, pour vérifier qu'il fait bien ce qu'on voulait.

1.4.2 Références

Pour terminer, voici quelques bonnes références sur le web pour le langage C.

- `http://www-clips.imag.fr/commun/bernard.cassagne/Introduction_ANSI_C/hyperdoc.html`
- `http://www-ipst.u-strasbg.fr/pat/program/tpc.htm`
- `http://www-rocq.inria.fr/who/Anne.Canteaut/COURS_C/` (très clair, et sans doute trop complet pour tout lire).
- et d'autres propositions sur `http://cours.comyr.com/`, cliquez "études" puis "informatique", "langage C" et, en fin de liste, "cours en langage C"

TP1 : Mon premier programme et ses variations

Avant toute chose, organisez vos données sur le disque : créez un répertoire C. Tout ce que nous ferons devra y être rangé, et rien d'autre. Dans C, créez le sous-répertoire TD1 pour le travail d'aujourd'hui.

Exercice 1 :

Ecrire le premier programme du cours dans un fichier bonjour.c placé dans le répertoire **C/TD1/source**.

```
1  #include <stdio.h> /* on utilise la bibliothèque standard */
2
3  int main(void) {
4    printf ("Bonjour !\n");
5          return 0 ; /* tout s'est bien passé */
6  }//fin du main
```

Sauver le fichier bonjour.c et laisser sa fenêtre d'édition ouverte. Ouvrir un terminal à coté. Se placer dans C/TD1/source.

Compilez le programme en utilisant la commande du début du cours (section 1.1), puis exécutez-le.

Exercice 2 :

Dans le terminal, effacez l'exécutable./bonjour. Faites une compilation par étape en regardant ce qui se passe (prenez des notes) :

> ls
> ./bonjour
> gcc -S bonjour.c
> ls
> gcc -c bonjour.s
> ls
> gcc bonjour.o -o bonjour
> ls
> ./bonjour

Exercice 3 :

Maintenant que vous avez vu, vous allez utiliser la version compacte de la commande pour éviter les fichiers intermédiaires et apprendre à ranger vos fichiers. Avec le terminal dans C/TD1/source, effacez tout sauf bonjour.c. Toujours dans le terminal, remontez dans C/TD1 et créez un répertoire exe (on va séparer les fichiers sources et les exécutables). Placez-vous dans ce répertoire. Exécutez la commande

> gcc ../source/bonjour.c -o bonjour
> ls
> ./bonjour

Par la suite,

- **vérifiez que les sources sont bien tous dans le sous-répertoire source et les exécutables dans exe**,
- **Ne rouvrez jamais le même fichier source dans une seconde fenêtre**

Exercice 4 :

Créez bonjour2.c en remplaçant printf par puts. Comparez ce que font les deux programmes.

Exercice 5 :

Copiez le programme bonjour.c dans bonjour3.c. Essayez une des modifications de la liste, notez à chaque fois le message d'erreur et expliquez le (revenez au programme d'origine avant d'essayer une autre modification) :

- supprimez le point-virgule à la fin de la ligne 4
- supprimez l'étoile à la fin de la ligne 5
- remplacez le "//fin du main" ligne 6 par "/* fin du main */" et refaites la même modification ligne 5
- supprimez le premier des deux / ligne 5
- supprimez la première ligne

Exercice 6 :

Question 6.1 : Tapez un programme C qui contient l'instruction `printf("%d%d%d%d", 1, 2, 3, 4)`.

Question 6.2 : Tapez un programme C qui contient l'instruction `printf("%s%s%s'', ''per'', ''sua'', ''der'');` .

Question 6.3 : Taper un programme C qui contient des variablesde tous types, et utilisez printf() pour afficher toutes ces variables

Exercice 7 :

Dans cet exercice, nous allons utiliser la fonction main() sous la forme `int main(int argc, char *argv[])`. Vous apprendrez toutes les subtilités nécessaires un peu plus tard, nous allons seulement utiliser le fait que argv[1] est la chaîne de caractères qui suit l'appel de fonction (on dit *le premier argument de la ligne de commande*).

Question 7.1 : Pour bien comprendre, tapez le programme monecho.c ci-dessous :

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[]){
4    printf (argv[1]) ;
5    return 0 ; /* tout s'est bien passe */
6  }// fin du main
```

Compilez-le et exécutez monecho Durand. Faites d'autres essais avec d'autres arguments à la place de Durand.

Question 7.2 : Ecrivez maintenant monbonjour.c pour que, quand on exécute monbonjour Durand cela écrive "Salut Durand" (et bien sûr monbonjour Dupont écrit "Salut Dupont"). On utilisera un %s comme dans l'exemple de la question précédente.

Exercice 8 :

La fonction atoi() convertit une chaîne numérique en entier. Sa description est dans stdlib.h. On peut par exemple avoir dans un programme :

```
1    #include <stdio.h>
2    #include <stdlib.h>
3   int main(void) {
4     char nom[] = "32" ; int val = atoi(nom) ;
5     printf ("le nombre %d s'écrit %s", val, nom) ;
6   }// fin du main
```

Ecrire un programme qui lit un nombre entier sur la ligne de commande et affiche son double.

Exercice 9 :

Le deuxième argument de la ligne de commande est argv[2]. Soit par exemple le programme moninv.c :

```
#include <stdio.h>
int main(int argc, char * argv[]) {
   printf ("%s, %s", argv[2], argv[1]) ;
  return(0)
}
```

Quand on exécute `moninv Durand Dupont` ce programme affiche `Dupont, Durand`.

Ecrire un programme sommeint.c qui lit deux nombres entiers sur la ligne de commande et affiche leur somme.

Exercice 10 :

La fonction atof() convertit une chaîne numérique en réel. Sa description est dans stdlib.h. Ecrire un programme quotientfloat.c qui lit deux nombres réels sur la ligne de commande et affiche leur quotient.

Exercice 11 :

Testez les deux programmes précédents en fournissant à l'exécution un mot au lieu d'un des nombres. Que se passe-t-il ? De même, que se passe-t-il si l'on fournit un seul argument ?

Cours 2

Types, adresses, entrées/sorties

2.1 Types simples

Le langage `C` est un *langage typé*, c'est-à-dire que les éléments manipulés ont un type défini. Les opérations et fonctions manipulent des opérandes de types fixés et renvoient un résultat également typé. Ceci constitue la *signature* de l'opération. Par exemple l'opérateur d'addition `+` permet d'ajouter des entiers et renvoie un résultat de type entier [1].

2.1.1 Les types de base

Les types de base sont *prédéfinis* et permettent de construire des types plus complexes comme les tableaux ou les structures.

Entiers

Le type **entier** permet de représenter des nombres entiers relatifs (c'est-à-dire dans $\mathbb{Z}$).

Pour des raison spécifiques au programme (faible utilisation mémoire ou manipulation d'entiers de grande valeur), on peut utiliser des entiers de différentes longueurs, signés ou non [2] :

- le type **int** (*integer*) permet de représenter un entier correspondant à un *mot machine*. Sa taille dépend donc du processeur utilisé, généralement 16 ou 32 bits ;
- un entier peut être représenté sur un nombre de bits plus faible (type **short**) ou au contraire plus long (type **long**) ;
- la plage de valeurs que peut prendre un entier dépend à la fois de sa longueur mais aussi de l'utilisation ou non d'un *bit de signe*, pour ne représenter que des valeurs positives ou des valeurs pouvant être soit positives soit négatives. On indique alors si l'entier est *signé* avec **signed** (valeur par défaut si rien n'est précisé) ou *non signé* avec **unsigned**.

Caractères

Le type **caractère** représente un caractère sur *un octet*, sous la forme de son *code ASCII*. En langage `C`, ce type est dénoté **char** (*character*).

Réels

Un **réel** permet de représenter un nombre en *virgule flottante*. Comme pour les entiers, la représentation des réels en langage `C` donne le choix entre différentes tailles :

- le type **float** représente les réels en *simple précision* ;

1. Certains opérateurs sont *surchargés* : ils admettent plusieurs signatures. C'est par exemple le cas de `+` qui permet d'additionner des entiers, des réels et même des caractères.
2. Pour les détails sur les représentations mémoire, se reporter au cours d'« architecture des ordinateurs » (module I2).

– la *double précision* est représentée par le type **double** ;
– il est également possible d'obtenir une précision étendue avec **long double**.

2.1.2 Constantes

L'écriture de *constantes* suit des règles fixant non seulement leur *valeur* mais aussi leur *type*.

Constantes entières

Une *constante entière* est un nombre dont la valeur, selon son écriture, est donnée sous forme :
– *décimale* : le nombre est écrit normalement ;
– *octale* : le premier chiffre est un `0` ;
– *hexadécimale* : le nombre est précédé de `0x` ou `0X`.

Par exemple, la valeur entière 13 s'écrit indifféremment `13`, `015` ou `0xD`.

Constantes caractères

Une *constante caractère* est entourée d'*apostrophes*. Cela permet de la différencier d'une variable ou d'une fonction dont le nom ne comporte qu'un seul caractère.

Le caractère peut être indiqué sous différentes formes :
– *normale* : le caractère lui-même ;
– *octale* : le code ASCII octal du caractère précédé de `\0` ;
– *hexadécimale* : le code ASCII hexadécimal du caractère précédé de `\x` ;
– *symbolique* : certains caractères ASCII ne sont pas imprimables. Ils sont représentés par un \ et un caractère. Les principales représentations symboliques de caractères sont données dans la table 2.1.

Table 2.1 – Principales représentations symboliques des caractères

Symbole	Code ASCII	Acronyme	Nom	Signification
`\0`	0	NUL	Null	Caractère nul
`\a`	7	BEL	Bell	
`\b`	8	BS	Backspace	Suppression du caractère précédent
`\n`	10	NL	New Line	Passage à une nouvelle ligne
`\r`	13	CR	Carriage Return	Retour à la ligne
`\t`	9	HT	Horizontal Tabulation	Tabulation horizontale
`\"`	34		Double Quote	Caractère guillemet
`\\`	92		Backslash	Caractère \

Par exemples, les constantes `'a'`, `'\041'`, `\x61'` représentent la lettre *a*.

Constantes chaînes de caractères

Une *chaîne de caractères* est une suite d'un nombre quelconque de caractères. S'il n'y a pas de caractère, on parle de *chaîne vide*. Une chaîne de caractères est délimitée par des *guillements*. Cela permet de différencier une constante chaîne de caractères d'un nom de variable ou de fonction ainsi que d'un caractère, pour une chaîne réduite à un seul caractère. Tous les types de notations de caractères peuvent être utilisés dans une chaîne de caractères. Par exemple, la chaîne de caractères `"première\nseconde"` est constituée de *première*, puis à la ligne suivante *seconde*.

Constantes réelles

Les *constantes réelles* sont différenciées des constantes entières par l'écriture d'un `.` qui représente la virgule [3]. Elles peuvent également être exprimées sous *forme scientifique* en précisant la puissance de 10 par la lettre `e` ou `E`. Enfin, le format peut être exprimé explicitement avec un **f** ou **F** pour un **float**, ou un **l** ou un **L** pour un **long double**. Par exemple, les valeurs suivantes sont des constantes réelles : `152.`, `165.3`, `152.254e6`, `165.54e-5`, `153.5f`, `198.86543227L`.

2.1.3 Déclaration de variables

Les *variables* doivent être déclarées avant de pouvoir être utilisées.

En `C`, on doit indiquer le nom de chaque variable ainsi que son type [4]. Par exemple

```
1  int compteur, nb;
2  char car;
```

montrent les déclarations de deux variables entières `compteur` et `nb`, et d'une variable caractère `car`, en `C`.

2.2 Les pointeurs

2.2.1 Qu'est-ce qu'un pointeur ?

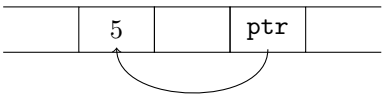
Un *pointeur* est une *adresse*.

Un pointeur fait donc référence à une adresse en mémoire, permettant ainsi de manipuler la mémoire et son contenu.

En `C`, une variable de type pointeur est déclarée sous la forme suivante :

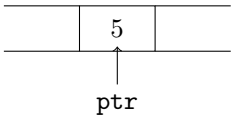
type_pointé `*`*nom_pointeur*`;`

```
1  int *ptr;
```



déclare une variable `ptr` de type pointeur sur un entier. Par conséquent, `ptr` contient l'adresse d'une zone en mémoire où l'on peut ranger un entier. Le schéma représente la variable `ptr` pointant sur une zone mémoire contenant l'entier `5`.

Pour simplifier le schéma, on préfère souvent utiliser la représentation suivante :



2.2.2 Opérateurs sur les pointeurs

La manipulation des pointeurs utilise deux opérateurs principaux : l'opérateur d'*adresse* et l'opérateur d'*indirection* (ou de *contenu*).

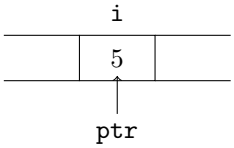
3. En anglais, on utilise un point à la place d'une virgule.

4. En `C`, on peut également préciser un mode de stockage de la variable, par exemple pour la forcer à être dans un registre, mais ces notions de programmation avancées dépassent le cadre de ce cours.

Adresse

L'*opérateur d'adresse* `&` permet d'obtenir l'adresse en mémoire d'une variable. Cette adresse peut en particulier être utilisée lors d'une affectation de pointeur.

```
1 int i;
2 int *ptr;
3
4 i = 5;
5 ptr = &i;
```

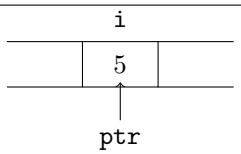


Dans ce programme, on déclare deux variables : `i` de type *entier* et `ptr` de type *pointeur sur un entier*. Ligne 4 la valeur `5` est affectée à la variable `i`, et ligne 5 l'*adresse* de la variable `i` est affectée à la variable `ptr`. Ce pointeur pointe donc sur `i` qui contient la valeur `5`.

Indirection (contenu)

L'opérateur d'*indirection* `*` permet d'obtenir le *contenu* de la mémoire à l'adresse pointée.

```
1 int i;
2 int *ptr;
3
4 i = 5;
5 ptr = &i;
6 printf ("Contenu à l'adresse pointée par ptr = %d\n",*ptr);
```



L'affichage ligne 6 permet de récupérer le contenu de la mémoire à l'adresse pointée par `ptr`, et va donc afficher : `Contenu à l'adresse pointée par ptr = 5`.

Pointeur nul

Un pointeur dont la valeur n'a pas été affectée est *nul*. Sa valeur est alors la constante `NULL`. Aucune opération n'est autorisée sur un pointeur nul, sauf bien sûr l'affectation.

2.2.3 Affectation de pointeur et typage

Un pointeur sur un type donné ne peut pointer que sur une variable de ce type. Par exemple, un pointeur sur un entier ne peut pas pointer sur une variable de type caractère. En effet, le type de la variable permet de déterminer la taille de la zone mémoire à prendre en compte. Lorsque des différences de type sont nécessaires, ce problème est contourné en utilisant à bon escient la conversion explicite de types (*cast*, voir Section 2.4.2) .

▶ Sur ce thème : **Exercice 1**

2.3 Entrées-Sorties

2.3.1 Affichage-`printf`

Dans le langage `C`, l'affichage de messages à l'écran se fait en utilisant la fonction `printf` de la bibliothèque d'entrées/sorties `stdio.h` (*standard input/output*), incluse par :

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

La fonction `printf` est utilisée sous la forme :
`printf(`*format, liste de variables*`);`

où *format* est une chaîne de caractères contenant le message à afficher. Si ce message contient des variables dont il faut afficher la valeur, le format d'affichage de chaque variable est précisé, selon les spécifications décrites dans le tableau 2.2. Lors de l'exécution, les spécifications des formats de variables sont remplacées par leur valeur, et traitées dans l'ordre d'apparition dans le message : la première spécification correspond à la première variable de la liste, la seconde spécification à la seconde variable, et ainsi de suite.

TABLE 2.2 – Spécification des formats d'affichage

Format	Signification
`%d`	entier décimal
`%f`	réel
`%x`	hexadécimal
`%o`	octal
`%c`	caractère
`%s`	chaîne de caractères

Par exemple, les instructions suivantes affectent la valeur `15` à la variable `compteur`, `9` à la variable `nb`, puis affichent ces valeurs en décimal, octal et hexadécimal.

```
1    compteur = 15;
2    nb = 9;
3     printf ("En décimal, compteur = %d, nb = %d\n",compteur,nb);
4     printf ("En octal, compteur = %o, nb = %o\n",compteur,nb);
5     printf ("En hexadécimal, compteur = %x, nb = %x\n",compteur,nb);
```

L'affichage à l'écran est alors :

```
En décimal, compteur= 15, nb = 9
En octal, compteur= 17, nb = 11
En hexadécimal, compteur= f, nb = 9
```

2.3.2 Saisie-`scanf`

Dans le langage `C`, la saisie de donnée par l'utilisateur se fait en utilisant la fonction `scanf` de la bibliothèque d'entrées/sorties `stdio.h` (*standard input/output*), incluse par :

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

La fonction `scanf` permet de saisir desdonnées au clavier et de les stocker aux adresses spécifiées dans les arguments de la fonction :
`scanf(`*format, liste d'adresses de variables*`);`

où *format* est une chaîne de caractères indiquant le format dans lequel les données lues sont converties, elle ne contient pas d'autres caractères. Comme pour `printf`, les conversions de format sont spécifiées par un caractère précédé du signe `%`, selon les spécifications décrites dans le tableau 2.2.

Les données à entrer au clavier doivent être séparées par des blancs ou des retours-chariot sauf s'il s'agit de caractères. On peut toutefois fixer le nombre de caractères à lire. Par exemple, `%3s` pour une chaîne de 3 caractères, `%10d` pour un entier sur 10 chiffres signe inclus

Par exemple, les instructions suivantes affectent la valeur `15` à la variable `compteur`, `9` à la variable `nb`, puis affichent ces valeurs en décimal, octal et hexadécimal.

```
1     int i, *j;
2     printf ("Entrer un premier entier");
3     scanf(\%d,&i);
4     printf ("Entrer un autre entier");
5     scanf(\%d,j);
6     printf ("i vaut \%d et l'entier à l'adresse j vaut \%d",i,*j);
```

A la ligne 3, la valeur entière saisie par l'utilisateur est affectée à l'adresse de l'entier î, à la ligne 5, la valeur entière saisie par l'utilisateur est affectée à l'adresse j correspondant à un espace où stocker une variable entière,

Si l'utilsateur répond 3 et 5, l'affichage à l'écran est alors :

```
Entrer un premier entier
3
Entrer un autre entier
5
i vaut 3 et l'entier à l'adresse j vaut 5
```

- ▶ Sur ce thème : **Exercice 2**
- ▶ Sur ce thème : **Exercice 3**
- ▶ Sur ce thème : **Exercice 4**
- ▶ Sur ce thème : **Exercice 5**
- ▶ Sur ce thème : **Exercice 6**
- ▶ Sur ce thème : **Exercice 7**

2.4 Le typage

Le *typage* permet d'associer un type à une variable. Les opérations relatives aux type peuvent alors être appliquées à la variable.

2.4.1 Taille d'un type

Lorsqu'une variable est déclarée, la zone mémoire nécessaire au stockage de cette variable est réservée. La *taille* de cette zone mémoire dépend du type de la variable. En effet, un caractère n'occupe pas la même quantité de mémoire qu'un entier long. Cette taille dépend aussi du système et de la machine sur laquelle le programme est exécuté.

L'opérateur `sizeof` permet d'obtenir la taille d'un type donné.

Par exemple,

```
1 printf ("taille d'un caractère : %d\n",sizeof(char));
2 printf ("taille d'un entier : %d\n",sizeof(int));
```

affiche la taille d'un caractère puis celle d'un entier.

2.4.2 Changement de type (*cast*)

La *conversion de type* (encore appelée *cast*) permet de convertir une valeur d'un type en sa représentation dans un autre type. En C, le typage est *explicite* et la conversion de type l'est également. Elle s'effectue en faisant précéder la valeur (ou variable) à convertir par le type cible entre parenthèses.

Par exemple,

```
1 #include <stdio.h>
2
3 int main(){
4       char c;
```

```
5         int i;
6
7         c = 'a';
8         i = (int) c;
9          printf ("c = %c\n",c);
10         printf ("i = %d\n",i);
11 }
```

Ce programme commence par déclarer deux variables : `c` de type caractère (ligne 4) et `i` de type entier (ligne 5). La variable `c` est initialisée au caractère `a` (ligne 7) puis elle est convertie en entier par un *cast*, ligne 8, pour être stockée dans la variable `i`. Enfin, les valeurs des deux variables sont affichées. Le résultat est alors :

```
c = a
i = 97
```

On remarque que 97 est le code ASCII du caractère `a`.

Il est possible que lors d'un calcul les opérandes n'aient pas le même type ou que le type du résultat de l'opération ne soit pas celui des opérandes. Il faut alors expliciter le type désiré par la syntaxe suivante : `(type) expression`.

Par exemple, dans le cas de la division entre deux entiers, en C le résutat est le quotient euclidien (le résultat est arrondi à l'entier inférieur).

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

va afficher :

```
un = 1.25
```

```
1  #include <stdio.h>
2  int main(void)
3  {
4   float un;
5  nt n=2;
6
7   un = 1;
8          un= un + 1/(n*n);
9    printf ("\n un = %f", un);
10  return(1);
11 }
```

va afficher :

```
un = 1
```

TP2 : Types, adresses, entrées-sorties

Exercice 1 : Mémoire-adresses

"Dessiner la mémoire" du programme suivant et expliquer les affichages qui ont lieu

```
int main()
{
 int n=5, m;
 float s, r=0.3 ;
 int *pp ;
 float *pr ;
 pp=&m ;
 *pp=n ;
  printf ("%d -", m) ;
 pr=&r ;
 s=*pr ;
  printf ("%f -", s) ;
}
```

Exercice 2 : Essais

1. Compiler et exécuter le programme suivant que vous aurez stocker dans un fichier `essai.c` :

```
#include <stdio.h>

int main(void)
{
     int   a  =  5   ;
     float b = 6.3 ;
      printf ("\n debut du programme" ) ;
      printf ("\n a vaut : %d", a) ;
      printf ("\n b vaut : %f", b) ;
     return (0) ;
}
```

2. Les variables `a` et `b` ont été initialisées dès leur déclaration. Modifier le programme de façon à déclarer ces variables sans les initialiser : `int a ; float b` ; Compiler et exécuter. Que peut-on en conclure ?
3. Modifier le fichier `essai.c` pour obtenir, compiler et exécuter le programme suivant. Qu'en déduisez-vous ?

```
#include <stdio.h>

int main(void)
{
     int   a  =  5   ;
     float b = 6.3 ;

      printf ("\n entrez une valeur entiere : ") ;
     scanf("%d", &a) ;
      printf ("\n entrez une valeur reelle : ") ;
     scanf("%f", &b) ;

      printf ("\n a vaut : %d", a);
```

```
14        printf ("\n  b vaut : %f", b);
15        printf ("\n a vaut: %d, b vaut %f, le double de b vaut: %f", a, b, 2*b);
16
17       return(0);
18  }
```

Exercice 3 : Syntaxe en C et comportement des variables

```
1   #include<stdio.h>
2
3   int main(void)
4   {
5   /* variables d'entrée */
6       float  intensite  ;
7       float tension ;
8
9
10  /* saisie des variables d'entrée */
11       printf ("\n Quelle est l'intensité ?" );
12       scanf("%f", intensite );
13       printf ("\n Quelle est la tension ? ");
14       scanf("%d",&tension);
15
16  /* traitement */
17       puissance  =  intensite  * tension
18       printf ("\n La puissance est : %.3f \n, puissance) ;
19      tension=tension+5;
20      intensite=2*intensite;
21      printf("\n La puissance est : %.3f \n", puissance) ;
22      puissance = intensite * tension ;
23      return(0);
24  }
```

1. Le programme suivant comporte 5 erreurs de syntaxe. Identifier ces incorrections et modifier le programme de façon à ce qu'il puisse être compilé convenablement
2. Quelles sont les instructions qui modifient les variables ? Quelles sont les valeurs prises par les différentes variables au cours de l'exécution du programme si les valeurs saisies par l'utilisateur sont $10,1$ pour l'intensité et 15 pour la tension. Quels sont les messages qui s'affichent à l'écran ?
3. Que fait le programme corrigé ?

Exercice 4 : Un calcul simple

1. Ecrire un programme qui calcule l'expression $y = ax + b$ où x est égal à 5, a à 18 et b à 7 ; $x; a$ et b seront déclarés comme des variables entières. Le programme affichera

```
1   y=97
```

2. Modifier le programme de sorte que x soit demandé à l'utilisateur au cours de l'exécution du programme

Exercice 5 : Peinture

On veut peindre des pièces rectangulaires (excepté le plafond) dont les dimensions (longueur, largeur et hauteur) sont indiquées par le commanditaire et varient selon les pièces. Ces pièces ont

une fenêtre carrée de $1,2$m de côté, l'épaisseur entre le cadre de la fenêtre (à peindre également) et la vitre étant de 15cm.

On utilise une peinture dont le pouvoir couvrant est de $3m^2/litre$. Cette peinture est vendue par pot de 2 litres. Sachant que les murs et le cadre de la fenêtre doivent être peints, donner un programme qui calcule et affiche la quantité de peinture ainsi que le nombre de pots nécessaires pour peindre une pièce en fonction de ses dimmensions.

On prendra soi d'analyser le problème avant de commencer à programmer la solution.

Exercice 6 : La tortue : un carré et ses diagonales

On dispose d'une tortue qui peut dessiner des traits sur l'écran de votre ordinateur. Les instructions doivent être formulées à l'aide des deux instructions suivantes :

- `avance(longueur)` qui a pour résultat le tracé d'un segment de droite d'une certaine longueur (en points) à partir de l'endroit où se trouve la tortue
- et `tourne(angle)` qui fait tourner la tortue d'un certain angle (la tortue tourne à droite si l'angle est positif et à gauche si l'angle est négatif).

On suppose que la tortue est initialement postée dans le coin supérieur gauche et qu'elle regarde vers le haut.

NOTE : Pour l'utilisation de la tortue, vous devez :

- Ajouter la directive d'inclusion :

```
1 #include <Tortue.h>
```

- Séparer les étapes de compilation et d'édition des liens en tapant :
  1. compilation : `gcc -c essai.c -o essai.o`
  2. édition des liens : `gcc -o essai essai.o -lTortue`

Exemple : un programme essaiTortue.c utilisant la tortue

```
1  #include <stdio.h>
2  #include <Tortue.h>
3
4  int main(void)
5  {
6   printf ("\n debut du programme essaiTortue" ) ;
7   printf ("\n Observez ce que sait faire la tortue");
8  ouvre();        /* ouvre une fenêtre graphique, jamais de printf ou de scanf
9                          après cette instruction */
10 avance(100);
11 tourne(90);
12 avance(100);
13 ferme(); /* ferme le fenêtre graphique, scanf et printf
14                           à nouveau autorisés */
15  printf ("\n fin du programme essaiTortue" ) ;
16 return (0) ;
17 }
```

1. Donner un algorithme qui permet à la tortue de dessiner la figure ci-dessous (un carré dont la longueur entière du coté est saisie par l'utilisateur) sans jamais repasser sur un segment déjà tracé.

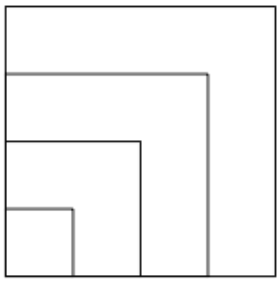
2. Mëme question pour la figure suivante

Exercice 7 :

La tortue est positionnée au coin inférieur gauche et regarde vers le haut.

Donner un algorithme qui fera dessiner à la tortue la figure suivante : où la longueur du côté du carré le plus petit est demandé à l'utilisateur ainsi que la différence entre les longueurs des cotés des différents carrés.

Qu'en concluez vous ?



Cours 3

Les alternatives et les boucles en C

3.1 Introduction : Le C est un outil logiciel pour coder un algorithme...

Comme vous l'avez vu dans les cours précédents, le langage C est un outil de programmation permettant de coder ces algorithmes afin de pouvoir les exécuter sur une plateforme informatique (système d'exploitation + architecture matérielle). Il en existe plusieurs autres : Ada, Python, Pascal.... Nous avons décidé d'utiliser le C pour mettre en application les algorithmes ; c'est à dire d'implémenter l'algorithme en respectant les contraintes du langage C :

- Le langage C dispose de sa propre syntaxe, c'est à dire qu'il possède des mots clefs réservés (`int`, `float`, `if`, `for`, `sizeof` etc...) permettant d'écrire les instructions,
- les instructions se terminent par un point virgule `;`,
- il nécessite la déclaration explicite des variables avant leurs utilisations,
- la zone de déclaration des variables doit se faire en **début de bloc** (notion de bloc, voir item suivant),
- sa propre façon de délimiter des blocs d'instructions ; souvenez-vous, en algorithmique vous avez utilisé l'*indentation* pour indiquer que des instructions appartiennent à un même bloc (séquence d'instruction(s)) ; en C nous utiliserons des accolades,
- par des librairies de fonctions d'affichages, d'accès aux fichiers, etc.

Exemple : En algorithmique vous écriviez ...

```
...     # points de suspensions pour dire qu'il y a d'autres instructions
I1
I2
I3
...     # points de suspensions pour dire qu'il y a d'autres instructions
        # (if, while ou autres)
        IB1             # indentation des instructions IB1 et IB2 pour marquer
        IB2             # qu'elles appartiennent à un bloc
...
I6
I7
```

En C ces instructions se coderont :

```
...             /* points de suspensions pour dire qu'il y a d'autres instructions */
I1;             /* N'oubliez pas le point virgule à la fin de l'instruction ! */
I2;
I3;
...             /* points de suspensions pour dire qu'il y a d'autres instructions
                   if(), while() ou autres */
{               /* Accolade ouvrante pour indiquer le début du bloc */
IB1;
IB2;
}               /* Accolade fermante pour indiquer la fin du bloc */
...      /* points de suspensions pour dire qu'il y a d'autres instructions */
I6;
I7;
```

Pour des questions de `lisibilité`, on peut utiliser une indentation (**qui n'a pas de signification en C** contrairement à l'algorithme qu'on écrit en Python).

```
I1;             /* N'oubliez pas le point virgule à la fin de l'instruction ! */
I2;
I3;
...             /* points de suspensions pour dire qu'il y a d'autres instructions
                   if(), while() ou autres */
{               /* Accolade ouvrante pour indiquer le début du bloc */
        IB1;            /* Indentation, juste pour la lisibilité */
        IB2;
}               /* Accolade fermante pour indiquer la fin du bloc */
...             /* points de suspensions pour dire qu'il y a d'autres instructions */
I6;
I7;
```

3.2 Le type booléen n'existe pas en C !

3.2.1 Rappels ...

Nous avons vu en algorithmique qu'une variable de type *booléen* peut prendre deux valeurs possibles :

- `True` pour indiquer un état *vrai* c'est à dire avéré,
- `False` pour indiquer un état *faux* c'est à dire non avéré.

Exemple d'algorithme pour vous rafraîchir la mémoire...

```
a=True              # a est une variable booléenne qui contient la valeur True (vrai)
i = 4                 # i est un entier qui contient la valeur 4
j = 10               # j est un entier qui contient la valeur 10

z = i>j             # z est un booleen qui contient la valeur False
```

3.2.2 Comment faire en C pour "simuler" le type booléen

On utilise un entier (`int`) ! (ou tout autre type compatible avec un entier : `char`, `short`, `long` etc.)

Pour coder les valeurs `True` ou `False`, on considère que :

- la **valeur 0** correspond à la valeur *FAUX*,
- **toute autre valeur différente de 0** (**donc pas forcément 1 !**) est *VRAI*.

3.2.3 Opérateurs relationnels et logiques en C

– Opérateurs relationnels :

Opérateur en C	<	<=	>	>=	==	!=
Equivalent algorithmique	<	<=	>	>=	==	!= ou <>

– Opérateurs conditionnels :

Opérateur en C	&&	||	!
Equivalent algorithmique	*and*	*or*	*not*

Opérations booléennes composites

Comme tous les langages de programmation, c'est une opération booléenne composée de plusieurs opérations booléennes simples. L'évaluation se fait en deux étapes :

1. chaque opération booléenne simple est évaluée,
2. l'expression booléenne composite résultante est évaluée suivant les tables de vérité suivantes :

exp1	exp2	!(exp2)	exp1 && exp2	exp1 || exp2
True	True	False	True	True
True	False	True	False	True
False	True		False	True
False	False		False	False

FIGURE 3.1 – Principaux opérateurs logiques

Exemple :

```
int main(void) {
                              /* On doit déclarer TOUTES les variables en début de programme */
  int c ;                     /* c est un entier non encore initialisé */
  int d = 14;          /* d est un entier qui contient 14 */
  int e = 1;                  /* e est un entier qui contient la valeur 1 */
  int f;                              /* f est un entier non encore initialisé */


  c = 2 > 4;                  /* c contient la valeur 0 pour dire False, car en C la valeur 0
                                  signifie False */

  f = d*2 >= e+1;      /* f contient une valeur différente de 0 pour signifier True, cela peut être
                                  1 ou -10, ou 100 peut importe pourvue que la valeur
                                  soit différente de 0 */

  printf ("%d %d", c,f);      /* affiche 0 25 */
                                      /* on a décidé que c'est la valeur 25 qui a été choisie pour
                                      signifier True */

 return 0;
}
```

Avez-vous compris ?

- ▶ Sur ce thème : **TD3, EXERCICE 1**
- ▶ Sur ce thème : **TD3, EXERCICE 2**

3.3 Les structures de contrôle conditionnelles en C

Le mot-clé **if** permet d'effectuer une instruction (simple ou composée) de manière conditionnelle. Il peut être suivi du mot-clé **else** pour spécifier l'instruction à effectuer lorsque la condition est fausse.

3.3.1 Première structure de contrôle conditionnelle, `if (condition) { bloc; }`

Codage en C :

```
if ( condition )
{       I1;
        I2;
        I3;
        I4;
}
```

Lecture de cette instruction :
Si la condition *condition* est vraie (*c'est à dire l'évaluation de cette expression (entre parenthèse) donne une valeur entière différente de 0*) alors j'exécute le **bloc d'instruction(s) I1 ; I2 ; I3 ; I4 ;** entre accolades.

Avez-vous compris ?

3.3.2 Deuxième structure de contrôle conditionnelle, `if (condition) { bloc1; } else { bloc2;}`

```

if (condition)
{       I1 ;
        I2 ;
        I3 ;
 }
else
{       I4 ;
        I5 ;
        I6 ;
        }
```

Si la condition *condition* est vraie (*c'est à dire l'évaluation de cette condition donne une valeur entière différente de 0*) alors j'exécute le **bloc d'instruction(s) I1 ; I2 ; I3 ; entre accolades sinon** (*c'est à dire si la condition est fausse*), j'exécute le bloc d'instruction *I4 ; I5 ; I6 ;*

Lorsque le bloc conditionnelle ne contient **qu'une instruction** alors l'utilisation des accolades pour délimiter ce code est facultative.

Les codes suivants sont équivalents :

– Pour la première structure de contrôle conditionnelle :

```

if (condition)
{       I1 ;
}

```

```
/* équivalent à */

if (condition)
        I1 ;
```

– Pour la deuxième structure de contrôle conditionnelle :

```
if (condition)
{       I1 ;
}
else
{
        I2;
}

/* équivalent à */

if (condition)
        I1 ;
else
        I2;
```

Note : Il existe d'autres possibilités selon que le premier bloc (bloc du `if()`) et/ou le deuxième bloc (bloc du `else`) ne contienne(nt) ou pas qu'une instruction ?

Avez-vous compris ?

▶ Sur ce thème : **TD3, Exercice 3**
▶ Sur ce thème : **TD3, Exercice 4**
▶ Sur ce thème : **TD3, Exercice 5**
▶ Sur ce thème : **TD3, Exercice 6**
▶ Sur ce thème : **TD3, Exercice 7**

3.4 Les structures itératives en C

Les structures itératives permettent de répéter un bloc d'instructions tant qu'une condition est vérifiée. Elles sont au nombre de 3.

– `while(condition){bloc;}`,
– `do { bloc; } while (condition)`,
– `for(I1; condition; I2) { bloc}`.

La première utilisation évalue une condition puis exécute l'instruction (simple ou composée) qui suit si la condition est vraie. Cette boucle continue tant que la condition reste vraie. La seconde utilisation exécute tout d'abord l'instruction (simple ou composée). Puis la condition est évaluée et, tant qu'elle reste vraie, l'instruction est exécutée à nouveau. La troisième utilisation permet elle aussi d'effectuer des boucles. Nous allons les décrire une par une.

3.4.1 La boucle while(condition){ bloc ;}

```
while (condition)
{       I1 ;
        I2;
        I3;
}
```

En premier lieu, la condition *condition* est évaluée (**il faut donc veiller à sa valeur lors de l'entrée dans la boucle**) : si sa valeur est vrai (*c'est à dire correspond à un entier différent de*

0), le **bloc d'instruction(s) *I1 ; I2 ; I3 ;*** est exécuté puis la condition *condition* est réévaluée (**il faut donc qu'elle puisse changer de valeur pour sortir de la boucle**) et si elle a la valeur faux (*c'est à dire correspond à un entier égal à 0*), on exécute l'instruction qui suit le `while()`.

Le corps de la boucle peut ne pas être exécuté, dans le cas où la condition est fausse dès le départ.
Avez-vous compris ?

▶ Sur ce thème : **TD3, Exercice 8**
▶ Sur ce thème : **TD3, Exercice 9**

3.4.2 La boucle do { bloc ;} while(condition) ;

```
do
{       I1 ;
        I2;
        I3;
} while (condition);
```

La condition *condition* est évaluée **après l'exécution du corps de la boucle** ; si sa valeur est vrai (*c'est à dire correspond à un entier différent de 0*) , le corps de la boucle est exécuté à nouveau puis la condition est réévaluée (**il faut donc qu'elle puisse changer de valeur pour sortir de la boucle**) et si elle a la valeur faux (*c'est à dire correspond à un entier égal à 0*) , on exécute l'instruction qui suit le `do while();`.

Le corps de la boucle est exécuté au moins une fois.
Avez-vous compris ?

▶ Sur ce thème : **TD3, Exercice 10**

3.4.3 La boucle for(I1 ; condition ; I2) { bloc}

```
for(I1; condition; I2)
{       I1 ;
        I2;
        I3;
}
```

Cette itérative fonctionne en trois temps :

1. `I1` est **exécutée une seule fois !**, à l'entrée de la boucle,
2. la condition *condition* est évaluée (**il faut donc veiller à sa valeur lors de l'entrée dans la boucle**) : si sa valeur est vrai (*c'est à dire correspond à un entier différent de 0*), le **bloc d'instruction(s) *I1 ; I2 ; I3 ;*** est exécuté, si elle a la valeur faux (*c'est à dire correspond à un entier égal à 0*) , on exécute l'instruction qui suit le `for()`
3. puis `I2` est exécutée, et on reprend en 2)

Plusieurs remarques s'imposent :
- L'instruction `I1` sert souvent à l'initialisation d'un compteur qui évoluera au fur et à mesure des itérations jusqu'à rendre la condition fausse,
- le corps de la boucle peut ne pas être exécuté dans le cas où la condition est fausse dès le départ,
- vous pouvez trouver facilement l'équivalent de la boucle `for()` en utilisant une boucle **while()**, mais l'intérêt d'utiliser la boucle `for()` réside, entre-autre, dans le fait qu'elle regroupe dans une même instruction les initialisations, la condition, et le corps de la boucle,

– on utilise cette boucle lorsque le nombre d'itération est connu à l'avance, c'est à dire qu'on sait à l'avance combien on va faire de répétitions.

Avez-vous compris ?

▶ Sur ce thème : **TD3, Exercice 11**

3.4.4 Il est temps de rassembler toutes vos connaissances...

▶ Sur ce thème : **TD3, Exercice 12**
▶ Sur ce thème : **TD3, Exercice 13**
▶ Sur ce thème : **TD3, Exercice 14**
▶ Sur ce thème : **TD3, Exercice 15**

TD3 : Alternatives et boucles en C

Note importante : Dans tout ce td, il s'agit de comprendre et interpréter **rigoureusement** et **méthodiquement** sur papier ce que le programme réalise et les particularités du langage C. En effet, beaucoup d'exercices évoquent des erreurs courantes qui font perdre beaucoup de temps au programmeur dans la phase de *débogage* d'un programme.

Exercice 1 :

Prévoir sur le papier et interprétez les résultats du programme suivant (**vous devrez le compléter éventuellement par des instructions d'affichages**).

```
/* Environnement */
int x =12 ;
bool test ;
/* Corps des instructions */
(1) test = x>12 ;
(2) test = x<11 || (x>8 && x<10) ;
(3) test = x !=9 ;
(4) test = x>10 && x<=12 ;
```

Exercice 2 :

Prévoir sur le papier et interprétez les résultats du programme suivant (**vous devrez le compléter éventuellement par des instructions d'affichages**).

```
 int a =-40 ;
  int d = 12;
  int e = 40;
  int f;


  f = a && e>d;
```

Exercice 3 : Les alternatives if() et if() else

Saisir les programmes suivants et exécutez-les et interprétez **rigoureusement** les résultats obtenus. **Sont-ils conformes à ceux attendus ?**

– Programme 1

```
#include <stdio.h>
int main(void) {
 int i=14;
 int z=17;


 if ( i== 15 && z>4) {
        printf ("C'est vrai");
 } else {
        printf ("C'est faux");
}

 return 0;
}
```

– Programme 2

```c
#include <stdio.h>
int main(void) {
 int i=14;
 int z=17;


 if ( i= 15 && z>4) {
        printf ("C'est vrai");
 } else {
        printf ("C'est faux");
}

 return 0;
}
```

Exercice 4 : Les pièges des opérateurs de pré(incrémentation/décrémentation) et de post(incrémentation/décrémentation)

Note : Pour réaliser l'opération : `i=i+1`, plusieurs solutions s'offrent au programmeur :

1. Ecrire `i=i+1` ; // Trivial
2. utilisez la **postincrémentation**, `i++` ; dans ce cas l'incrémentation se fait **APRES** l'évaluation de la variable `i`,
3. utilisez la **préincrémentation**, `++i` ; dans ce cas l'incrémentation se fait **AVANT** l'évaluation de la variable `i`.

Remarque : Même raisonnement pour `i=i-1`.

Saisir les programmes suivants et exécutez-les et interprétez **rigoureusement** les résultats obtenus. **Sont-ils conformes à ceux attendus ?**

– Programme 1

```c
#include <stdio.h>
int main(void) {
 int i=1;


 if (i--) {
        printf ("Mystere\n");
 } else {
        printf ("C'est du tout cuit\n");
}

  printf ("La valeur de i est %d\n",i);

 return 0;
}
```

– Programme 2

```c
#include <stdio.h>
int main(void) {
 int i=1;


 if (--i) {
        printf ("Mystere\n");
 }
```

```
else {
        printf ("C'est du tout cuit\n");
}
 printf ("La valeur de i est %d\n",i);

 return 0;
}
```

Exercice 5 : Calcul des racines réelles d'un polynôme de seconde degrés : a*x*x + b*x +c = 0

Ecrire un programme en C qui à partir des coefficients réels `a`, `b`, `c` **saisis par l'utilisateur**, calcule et affiche les racines réelles du plolynôme de second degrés correspondant.

Exercice 6 :

Ecrire un programme en C qui vérifie qu'une suite de 3 caractères **saisis par l'utilisateur** est dans l'ordre alphabétique.

Exercice 7 : Il court, il court le temps...

Ecrire un programme en C qui calcule à partir d'une heure fournis par l'utilisateur (sous forme de 3 variables entières correspondant aux *heure*, *minute* et *seconde*), calcule l'heure qu'il sera après une seconde.

L'affichage à obtenir est le suivant :

```
   Heure courante : HH:MM:SS
Une seconde après il sera : HH:MM:SS
```

Dans tous les cas l'affichage des heures, minutes et secondes se fera sur deux chiffres.

Exercice 8 : La boucle while()

Saisir les programmes suivants, analyser l'affichage obtenu et expliquez se qui s'est passé.

– Programme 1

```
#include <stdio.h>

int main(void){

        int i=10;

        while(i--){
                printf ("%d\t",i);      // \t permet d'inserer une tabulation
        }
return 0;
}
```

– Programme 2

```

#include <stdio.h>

int main(void){

        int i=0;

        while(++i){
                printf ("%d\t",i);
```

```
10        }
11 return 0;
12 }
```

– Programme 3

```
1  #include <stdio.h>
2
3  int main(void){
4
5         int i=1;
6         int test=1;
7
8
9         while(i<10 && test==1){
10                printf ("%d\t",i);
11                if (i%7==0) test=0;     /* permet de calculer le reste de la division
12                                                   euclidienne */
13                i++;
14        }
15 return 0;
16 }
```

– Programme 4

```
1  #include <stdio.h>
2
3  int main(void){
4
5         int i=1;
6         int test=1;
7
8
9         while(i<10 && (test=1)){
10                printf ("%d\t",i);
11                if (i%7==0) test=0;     /* permet de calculer le reste de la division
12                                                   euclidienne */
13                i++;
14        }
15 return 0;
16 }
```

Exercice 9 :

1-Tapez le programme suivant puis exécutez le :

```
1  # include <stdio.h>
2  int main() {
3    int n = 0, i;
4
5    while(n <= 0) {
6      printf ("Entrez un nombre > 0 : ") ;
7      scanf("%d",&n) ;
8    }
9    for(i=n ; i>=0 ; i--)
10     printf ("Compte : %d\n",i) ;
11
12    printf ("Depart\n") ;
13
```

```
14  return 0 ;
```

2-Modifiez ce programme pour qu'il affiche les nombres dans l'ordre croissant.

3-Modifiez ce programme pour qu'il affiche à la fin la somme des carrés ainsi que la somme des nombres.

Exercice 10 : La boucle do while()

Saisir les programmes suivants, analyser l'affichage obtenu et expliquez se qui s'est passé.

– Programme 1

```
1  #include <stdio.h>
2
3  int main(void){
4
5          int i=1000;
6
7
8
9          do{
10                 printf ("%d\t",i);
11                 i--;
12         }while(i<10);
13 return 0;
14 }
```

– Programme 2

```
1  #include <stdio.h>
2
3  int main(void){
4
5          int i=1;
6
7
8
9          do{
10                 printf ("%d\t",i);
11                 i--;
12         }while(i<10);
13 return 0;
14 }
```

– Programme 3

```
1  #include <stdio.h>
2  \begin{ lstlisting }
3  int main(void){
4
5          int i=1;
6
7
8
9          do{
10                 printf ("%d\t",i);
11                 i++;
12         }while(i<10);
13 return 0;
14 }
```

Exercice 11 : La boucle for()

Saisir les programmes suivants, analyser l'affichage obtenu et expliquez se qui s'est passé.

– Programme 1

```
#include <stdio.h>

int main(void){

        int i;

        for(i=0; i<10;i++){
                printf ("%d\t",i);
        };
return 0;
}
```

– Programme 2

```
#include <stdio.h>

int main(void){

        int i;

        for(i=0; i==10;i++){
                printf ("%d\t",i);
        };
return 0;
}
```

– Programme 3

```
#include <stdio.h>

int main(void){

        int i;

        for(i=0; i=10;i++){
                printf ("%d\t",i);
        };
return 0;
}
```

– Programme 4

```
#include <stdio.h>

int main(void){

        int i;
        int j;

        for(i=0, j=10; i!=j;i++, j--){
                printf ("i=%d\tj=%d\n",i,j);
        };
return 0;
}
```

Exercice 12 :

Ecrivez un programme qui affiche tous les couples `(x,y)`, où `x` est un entier compris entre 1 et `p` et `y` un entier compris entre 1 et `q`; `p` et `q` sont deux entiers lus au clavier. L'affichage doit se faire comme sur l'exemple suivant, qui correspond à `p` = 3 et `q` = 5 :
(1 , 1) (1 , 2) (1 , 3) (1 , 4) (1 , 5)
(2 , 1) (2 , 2) (2 , 3) (2 , 4) (2 , 5)
(3 , 1) (3 , 2) (3 , 3) (3 , 4) (3 , 5)

Exercice 13 : Calcul du modulo sans l'opérateur module %

Nous souhaitons écrire un programme en C qui permet de savoir si un nombre entier est divisible par un autre (quelque soit l'ordre de saisi des arguments ce sera toujours de savoir si le grand nombre est divisible par le plus petit). Les deux nombres seront donnés par l'utilisateur.
Vous ne devrez pas utiliser l'opérateur modulo (%), mais la méthode des soustractions successives.
Exemple 1 :
si `a = 6` et `b = 2`, ou `a=2` et `b=6`
On appliquera la méthode :
`6 -2 = 4`
`4 - 2 = 2`
`2 - 2 = 0` (c'est fini car je ne peux plus soustraire 2 à 0)
Donc 6 est divisible par 2 (car la dernière soustraction permet d'obtenir un 0)
Exemple 2 :
si `a = 7` et `b = 3`, ou `a=3` et `b=7`
On appliquera la méthode :
`7 - 3 = 4`
`4 - 3 = 1` (c'est fini car je ne peux plus soustraire 3 à 1)
Donc 7 n'est pas divisible par 3 (car la dernière soustraction ne permet pas d'obtenir un 0)

Exercice 14 : Contrôle de saisie (filtrage des valeurs saisies)

On suppose une variable entière `n` déclarée et non initialisée.
On veut saisir une valeur pour n :

1. S'assurer que la valeur saisie est strictement positive,
2. filtrer uniquement les valeurs entre 0 inclus et 9 inclus.

Exercice 15 : Calcul d'une moyenne d'un ensemble de valeurs

Nous désirons saisir 30 valeurs réelles comprises entre 0 et 20 (bornes comprises). Ces valeurs n'auront pas besoin d'être mémorisé dans un tableau. La moyenne sera calculée (à la volée c'est-à-dire lors de la saisie).Un affichage des résultats sera effectué.

Exercice 16 : Calcul du PGCD de deux nombres entiers

L'algorithme d'Euclide calculant le Plus Grand Commun Diviseur (PGCD) peut être décrit comme suit :

"tant que les 2 nombres sont différents soustraire le plus petit du plus grand. Une fois égaux, afficher l'un d'eux : c'est le pgcd."

1. Faire tourner cet algorithme sur papier avec les entiers 18 et 14,
2. écrire le programme en C qui permet de saisir deux valeurs entières `val1` et `val2` et qui calcule et affiche le pgcd de ces deux nombres.

Cours 4

Fonctions

4.1 Déclaration de fonctions

L'écriture de *fonctions* permet de réaliser des *programmes structurés* en les découpant en entités plus petites, donc plus faciles à comprendre, réaliser et vérifier, selon le paradigme « diviser pour régner ».

Une fonction réalise un objectif particulier, en général dépendant de paramètres et retournant une valeur. Elle peut être exécutée plusieurs fois dans un même programme, pour des valeurs de paramètres différentes (et fournit alors les résultats correspondant à ces différentes exécutions). Le codage par fonction permet une simplicité du code et fournit une taille de programme minimale.

L'écriture d'une fonction comporte son *nom*, ses *paramètres avec leur type*, ainsi que le type de la valeur retournée. Attention, Contrairement au langage *Python*, en C il n'y a qu'une seule valeur de retour.

En `C`, on déclare dans l'en-tête du programme le *déclaration de la fonction* qui renseigne le compilateur (et le programmeur) sur la manière de l'utiliser. Ensuite, dans le corps du programme, la fonction est explicitée en précisant également les instructions qu'elle exécute :

```
/* en-tête : déclaration de la fonction */
type_résultat nom_fonction(paramètre_1: type_1, ...);
.....
/* définition de la fonction */
type_résultat nom_fonction(paramètre_1: type_1, ...) {
type_variable nom_variable;
....
/* Corps de la fonction */
}
```

Le corps de la fonction est délimité par des accolades `{` et `}`.

```
/* inclusion des bibliothèques */
#include <stdio.h>

/* déclaration des fonctions */
int carre(int nb);

/* calcul du carré de nb */
int carre(int nb){
        int resultat;

        resultat = nb * nb;
        return(resultat);
```

```
13 }
14
15 /* fonction principale */
16 int main(){
17         int res;
18
19         res = carre(2);
20          printf ("Le carré de 2 est %d.\n",res);
21          printf ("Le carré de 5 est %d.\n",carre(5));
22 }
```

Le résultat de l'éxécution de ce programme est :

```
Le carré de 2 est 4.
Le carré de 5 est 25.
```

Lorsqu'une fonction ne renvoie pas de résultat, son type est **void** (*vide*).

4.2 Remarque : Visibilité des variables

Lorsque l'on déclare une variable à l'intérieur d'une fonction (entre des accolades), sa portée se confine à l'intérieur de la fonction dans laquelle elle est déclarée. Il faut également faire attention à la valeur passée en paramètre de la fonction, elle ne doit pas être initialisé avant d'être utilisée dans le corps de la fonction sous risque de masquer la valeur passée en paramètre (voir exemple ci-dessous).

```
1  #include<stdio.h>
2  void bidon(int i)
3  {
4          int j=20;
5          printf ("dans bidon %d,%d\n", i, j);
6  }
7  void rebidon(int i)
8  {
9          i=1000;
10         printf ("dans rebidon %d\n", i);
11 }
12 int main (void)
13 {
14         int i =100;
15         int j = 10;
16         bidon(i);
17         printf ("%d,%d\n",i, j);
18         rebidon(i);
19         printf ("%d\n",i);
20 }
```

Le résultat de l'exécution de ce programme est :

```
dans bidon 100,20
100,10
dans rebidon 1000
100
```

4.3 Passage de paramètres

Lors de l'appel d'une fonction, une zone mémoire est réservée pour chaque paramètre de la fonction, et remplie avec les valeurs d'appel.

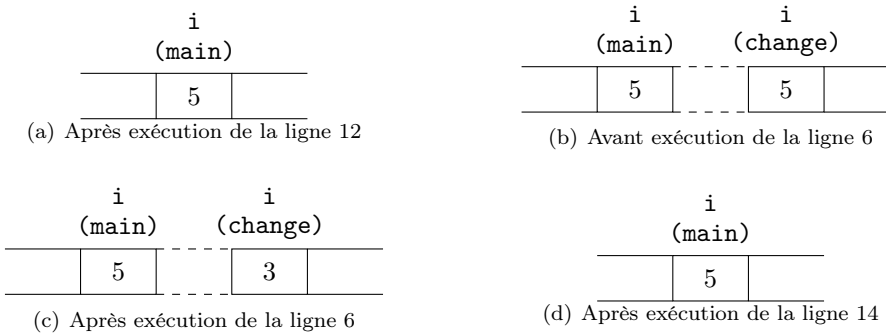
4.3.1 Passage de paramètres par valeur

Lors de l'appel, lorsqu'une variable est passée en paramètre, une copie est créée, qui correspond au paramètre. Le nom de la variable est local à la fonction à laquelle il appartient, par conséquent une variable `x` de la fonction appelante n'est pas la même (zone mémoire) que la variable `x` de la fonction appelée. L'utilisation de noms identiques, dans ce cas, permet seulement au programmeur d'utiliser des variables externes. Si une telle variable est utilisée, toute modification ne porte que sur la copie.

example

Soit le programme suivant :

```
1  #include <stdio.h>
2  void change(int i);
3
4  void change(int i) {
5          printf ("i=%d\n",i);
6          i = 3;
7          printf ("i=%d\n",i);
8  }
9
10 int main(){
11         int i;
12         i = 5;
13         printf ("i=%d\n",i);
14         change(i);
15         printf ("i=%d\n",i);
16 }
```



(a) Après exécution de la ligne 12

(b) Avant exécution de la ligne 6

(c) Après exécution de la ligne 6

(d) Après exécution de la ligne 14

FIGURE 4.1 – Passage de paramètre par valeur

Dans la fonction principale, `main`, est déclarée une variable `i`, initialisée ligne 12. L'état de la mémoire après l'exécution de cette instruction est décrit dans la figure 4.1(a). Seule la variable `i` de la fonction `main` est présente en mémoire et sa valeur est `5`. L'instruction ligne 13 affiche `i=5`. La fonction `change` est ensuite appelée avec passage de la variable `i` par valeur. Une copie de la variable est donc créée en mémoire et affectée au paramètre de la fonction `change` (qui s'appelle également `i`). L'état de la mémoire est alors comme indiqué dans la figure 4.1(b). L'instruction ligne 5 affiche donc `i=5`. Puis, la valeur de `i` est changée ligne 6. Une seule variable `i` est connue de la fonction `change`, et c'est par conséquent celle-là qui est modifiée (voir figure 4.1(c)). La ligne 7 affiche alors `i=3`. À ce moment-là, l'exécution de `change` est terminée et les variables associées libérées. Alors, après l'exécution de la ligne 14, la mémoire est dans le même état qu'avant cet appel, comme décrit par la figure 4.1(d).

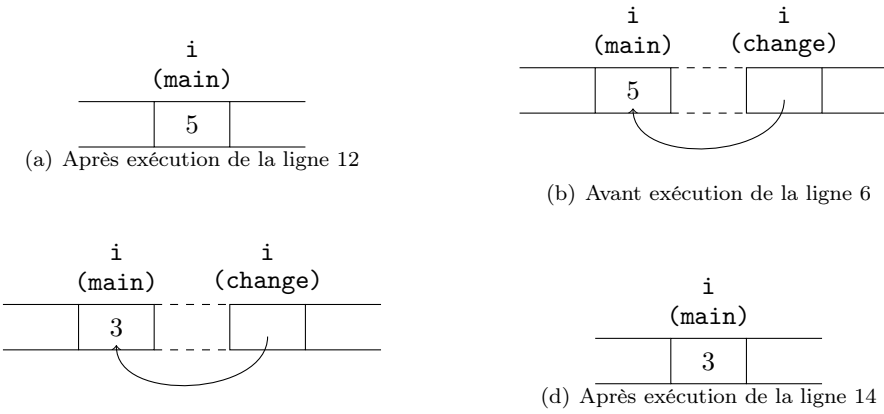
4.3.2 Passage de paramètres par adresse

Lorsque l'on veut pouvoir modifier le contenu d'une variable dans la fonction appelée, on effectue un *passage par adresse*. Il s'agit non plus de transmettre la valeur de la variable, mais son adresse. Le mécanisme en jeu est le même que précédemment, en ce sens que l'adresse ne sera pas changée. Par contre le contenu de la mémoire à cette adresse peut être modifié.

example

Modifions le programme de l'exemple 4.3.1 pour passer le paramètre `i` par adresse :

```
1  #include <stdio.h>
2  void change(int *i);
3
4  void change(int *i) {
5          printf ("i=%d\n",*i);
6          *i = 3;
7          printf ("i=%d\n",*i);
8  }
9
10 int main(){
11         int i;
12         i = 5;
13         printf ("i=%d\n",i);
14         change(&i);
15         printf ("i=%d\n",i);
16 }
```



(a) Après exécution de la ligne 12

(b) Avant exécution de la ligne 6

(c) Après exécution de la ligne 6

(d) Après exécution de la ligne 14

FIGURE 4.2 – Passage de paramètre par adresse

Dans la fonction principale, `main`, est déclarée une variable `i`, initialisée ligne 12. L'état de la mémoire après l'exécution de cette instruction est décrit dans la figure 4.2(a). Seule la variable `i` de la fonction `main` est présente en mémoire et sa valeur est `5`. L'instruction ligne 12 affiche `i=5`. La fonction `change` est ensuite appelée avec passage de la variable `i` par adresse. Une variable est donc créée en mémoire et affectée au paramètre de la fonction `change`, qui s'appelle également `i`, mais qui est un pointeur sur l'adresse de la variable `i` de la fonction `main`. L'état de la mémoire est alors comme indiqué dans la figure 4.2(b). L'instruction ligne 5 affiche donc `i=5`. Puis, la valeur de l'adresse pointée par `i` dans la fonction `change` est modifiée ligne 6 (voir figure 4.2(c)). La ligne 7 affiche alors `i=3`. À ce moment-là, l'exécution de `change` est terminée et les variables

associées libérées. Alors, après l'exécution de la ligne 14, la mémoire est dans l'état décrit par la figure 4.2(d).

4.4 Une fonction particulière : main

Contient le programme principal, elle est automatiquement appelée par le système d'exploitation au chargement du programme, sa valeur de retour est une code de retour d'état d'exécution du programme (Valeur =0 exécution correcte, Valeur != 0 erreur du programme).

```
1 int main(void);
2 int main(int argc, char *argv[]);
3 int main(int argc, char *argv[], char *envp[])
4 // argc : nombre de paramètres de la ligne de commande
5 /* argv : tableau de chaine de caractères contenant les arguments de la ligne de commande */
6 /* envp: tableau de chaine de caractères contenant les variables d'environnement système */
```

TP4 : Fonctions

Exercice 1 : Portée des variables en C

Qu'affiche le programme C suivant ? Expliquez en détail votre réponse.

```
1
2  #include<stdio.h>
3
4  void affiche1(int x);
5  int affiche2(int x);
6
7
8  void affiche1(int x){
9          x=x+1;
10         printf("affiche1:x=%d\n", x);
11 }
12
13 int affiche2(int x){
14         x=x+1;
15         printf("affiche2:x=%d\n", x);
16         return(x);
17 }
18
19 int main(){
20         int x;
21         x=10;
22         printf("main: x=%d\n", x);
23         affiche1(x);
24         printf("main: x=%d\n", x);
25         x=affiche2(x);
26         printf("main: x=%d\n", x);
27 }
```

Exercice 2 : Paramètres et pointeurs

Soit la fontion suivante :

```
1  void faitQuoi(int x, int y, int d)
2  {
3  x = x + d;
4  y = y + d;
5  }
```

Soit le programme principal suivant

```
1  int main()
2  {
3  int a = 8;
4  int b = 2;
5  int c = 4;
6  faitQuoi (a, b, c);
7  printf("a=%d, b=%d", a, b);
8  return(0);
9  }
```

Dérouler le programme et "dessiner la mémoire" très précisément au fur et à mesure de l'exécution du programme. Indiquer ce qui s'affiche.

Même question avec la fonction et l'appelant suivants :

```
void faitQuoiBis(int *px, int *py, int d)
{
*px = *px + d;
*py = *py + d;
}
```

Soit le programme principal suivant :

Exercice 3 : Factorielle

écrire une fonction factorielle qui calcule le factoriel d'un entier et écrire un programme de test qui l'utilise.

Exercice 4 : Somme

écrire une fonction qui saisie 30 valeurs réelles comprises entre 0 et 20 et qui affiche la moyenne, le max et le min.

Exercice 5 : Maj/Min

écrire en langage C une fonction majmin qui prend un caractère majuscule en paramètre et retourne la minuscule correspondante (on ne vérifiera pas dans la fonction que le caractere est bien une majuscule). Tester un programme principal appelant cette fonction.

Exercice 6 : Echange

1. écrire une fonction C, qui échange deux nombres réels. Tester un programme principal appelant cette fonction.

2. Utilisez la fonction 'echange' pour classer trois nombres dont les valeurs sont données sur la ligne de commandes (solution).

Exercice 7 : Est premier

écrire un programme qui permette d'afficher sur une seule ligne l'ensemble des diviseurs, sépares par des -, d'un entier donné. On programmera la fonction divise(n,m) qui retourne 1 si n est divisible par m, 0 sinon.

Exercice 8 : Calculatrice

écrire un programme C (calculSimple.c) dont les arguments constituent une expression arithmétique simple (operande1 opérateur operande2) et qui affiche l'evaluation de l'expression. On suppose que tous les arguments sont sépares par un espace.

Par exemple :

```
./calculSimple 12 + 14 va afficher: 12 + 14 = 26
```

Exercice 9 :

Ecrivez le programme, de manière à ce que le nombre de personnes soit passé sur la ligne de commandes et qu'une fonction soit utilisée pour calculer la quantité de chaque ingrédient pour le nombre de personnes donné.

```
sucre_semoule <- (300 g / 8) * nb_personnes ;
sucre_vanille <- (1 c. à soupe / 8) * nb_personnes ;
oeufs <- (4 / 8) * nb_personnes ;
farine <- (50 g / 8) * nb_personnes ;
beurre <- (250 g / 8) * nb_personnes ;
```

```
6 chocolat_noir <- (160 g / 8) * nb_personnes ;
7 noix_de_pecan <- (90 g / 8) * nb_personnes ;
8 amande_en_poudre <- (30 g / 8) * nb_personnes ;
```

Cours 5

Pointeurs, tableaux

5.1 Quest-ce qu'un pointeur

Les pointeurs sont des variables destinées à contenir des **adresses**. C'est une grande particularité du C de permettre de savoir **où une donnée est placée dans la mémoire** et de l'utiliser dans un calcul. Pour n'importe quel nom de variable d'un type de base, on a son adresse en ajoutant & devant ce nom. Voici un exemple :

```
1  #include <stdio.h>
2
3  int main(void) {
4    char c1,c2 ;
5    int i, j ;
6     printf ("Adresses : %u %u %u  %u\n", &c1, &c2, &i, &j);
7  }//fin du main
8
9  // Affiche :
10 // Adresses : 3221222488  3221222489  3221222492  3221222496
```

Les adresses ont été affichées avec "%u" qui les traite comme des nombres non signés – ce qui convient pour l'affichage (le spécifieur "%p" fait la même traduction, mais écrit en hexadécimal). On peut voir à l'exécution du programme que les adresses de c1 et c2 diffèrent de 1 alors que celles de i et j diffèrent de 4.

On peut avoir à stocker des adresses dans des variables pour s'en reservir plus tard. Il n'est pas souhaitable de les stocker dans des variables entières, car alors on ne saurait plus *de quel type de données c'est l'adresse*, ce qui rendrait difficile de s'en servir. Au lieu de cela, le C utilise des types de variables construits, les types **pointeurs** : il y a autant de types de pointeurs que de types de variable, puisque pour chaque pointeur on précise sur quel type il pointe.

On peut affecter 'd' à *cptr parce qu'on sait que *cptr est de type char. Mais comment le compilateur le saura-t-il ? Parce que le programmeur le déclare, comme pour n'importe quel type. Il suffit de connaitre la formule de déclaration, et maintenant elle est simple à comprendre :

```
1  char *cptr ;
2  int *iptr ;
3  float *fptr ;
```

*cptr est de type char comme *iptr est de type int et *fptr est de type float. Le nom choisi (cptr, etc.) n'y est pour rien, bien que ce soit très conseillé d'avoir des noms aide-mémoire comme cela. C'est simplement que, **l'objet pointé** est de type char, int ou float – donc le pointeur est de type pointeur de caractère, pointeur d'entier ou pointeur de réel. On dira aussi que cptr, iptr ou fptr sont de type char*, int* ou float*.

```
int i, j  ; // un entier
int *iptr ; //*iptr est de type int, ce qui indique que iptr est un pointeur d'entier (de type int*)
iptr = &i ; //on initialise iptr avec l'adresse d'un int (celle de i)
*iptr = 7 ; // maintenant i vaut 7
iptr = &j ; // on a changé l'adresse contenue dans iptr
*iptr = 3 ; // maintenant j vaut 3, et i vaut toujours 7.
```

Notez de suite que, comme un pointeur est une variable comme une autre, rien n'empêche d'utiliser son adresse, par exemple &fptr. Il est même souvent utile de stocker cette adresse dans une variable, qui sera donc un pointeur puisqu'elle contient une adresse. Mais alors, de quel type faut-il déclarer ce pointeur de pointeur ? C'est assez facile à comprendre :

```
int i, j  ; // i et j sont des entiers
int *iptr ; // iptr est un pointeur d'entier (de type int*)
int **ipptr ; // **ipptr est de type int, ou *ipptr est un pointeur d'entier (de type int*)
                    // ou ipptr est un pointeur de pointeur d'entier (de type int**)
iptr = &i ; //on initialise iptr avec l'adresse d'un int
ipptr = &iptr ; // on initialise ipptr avec l'adresse de iptr
**ipptr = 7 ; // maintenant i vaut 7, puisque la double déréférenciation ** indique i
*ipptr = &j ; //iptr contient maintenant l'adresse de j
**ipptr = 3 ; // j passe à 3.
```

5.2 Tableaux et pointeurs

5.2.1 Tableaux

Un tableau est une suite d'objets de même type placés les uns à coté des autres. En C, les tableaux (comme les variables) doivent obligatoirement être déclarés avant d'être utilisés. On déclare un tableau en donnant son type, son nom et, entre crochets, le nombre d'éléments qu'il contient :

```
char tabchar[43];
int tabint[750];
float tabfloat[120] ;
```

On peut initialiser un tableau par une liste de valeurs constantes séparées par des virgules :

```
float tabfloat[6] = {4, 332, 96, 7, 57, 67}
```

Le nombre d'éléments est obligatoirement une constante (explication : si on mettait une variable, le compilateur ne pourrait pas utiliser son contenu puisque celui-ci dépend de l'exécution). On accède à un élément du tableau par son indice, comme en Python, et l'indice peut être une variable. Les parcours de tableau utilisent constamment cette possibilité. Par contre, le compilateur ne contrôle pas que l'indice respecte la taille du tableau. Si l'indice utilisé dépasse cette taille, le programme ira lire ou écrire à l'adresse calculée, ce qui peut être catastrophique.

```
int i ; float tabfloat[120] ;
// initialisation du tableau
for (i = 0 ; i < 120 ; i++) {
  tabfloat[i] = 1. + sqrt(i) ; // ne pas oublier #include <math.h> et compiler avec -lm
} // fin du for. Le tableau est initialisé
printf("Valeur de tabint[17] : %f", tabint[17]) ; // lecture d'un élément
i = 200 ; // pas de problème
tabint[i] = 3.55 ; /* le compilateur ne voit pas de problème. A l'exécution, le programme fera
                      ce que le compilateur a codé, et le résultat est imprévisible.    */
```

En C, c'est au programmeur de s'assurer que les indices ne sortent pas du tableau. Souvent, on prévoit pour chaque tableau une constante ou une variable indiquant le nombre d'éléments : on l'utilisera pour les parcours et les contrôles :

```
int i, taille = 120 ; float tabfloat[120] ;
// initialisation du tableau
for (i = 0 ; i < taille ; i++) {
  tabfloat [i] = 1. + sqrt(i) ; // ne pas oublier #include <math.h> et compiler avec -lm
} // fin du for. Le tableau est initialisé
printf ("Valeur de tabint[17] : %f", tabint[17]) ; // lecture d'un élément
i = 200 ; // pas de problème
if (i < taille ) {tabint[i] = 3.55 ;} // on est dans le tableau
else { // A vous de voir. Si on passe là, il faut modifier le programme}
```

Pour le cas particulier des chaînes de caractère, qui sont en C des tableaux de caractères terminés par un caractère '\0' (le caractère de code ascii 0), on se passe en général de mémoriser le nombre d'éléments parce que la présence d'un code de fin de chaîne permet de calculer facilement ce nombre. Nous verrons dans peu de temps les techniques spécifiques aux chaînes.

5.2.2 Pointeurs et indices

Si l'on déclare le tableau `int tabint[500]` et qu'on déclare un pointeur d'entier `int *iptr`, l'instruction `iptr = &tabint[132]` fera pointer `iptr` sur le 133ème élément du tableau. Mais alors, *parce que iptr est déclaré du type pointeur d'entier*, `iptr = iptr + 1` le déplacera sur l'entier voisin, donc sur l'élément suivant du tableau (ici, le 134ème). La tournure compacte `iptr++` est aussi acceptée.

C'est tellement utile que le nom du tableau est en C une constante synonyme de son adresse : `ptr = tabint` est la même chose que `iptr = &tabint[0]`. En fait, quand on déclare un tableau, le compilateur C fait deux choses :

- il lui attribue un emplacement de la taille nécessaire dans l'espace mémoire de l'exécutable, et
- il conserve dans ses propres structures de données l'adresse de début de cet emplacement (celle du premier élément) et le type du tableau. Le nom du tableau lui sert en fait à désigner l'adresse de début.

C'est comme cela que le compilateur calcule où est tabint[132] (il ajoute 132 fois la taille d'un entier à tabint) et peut soit y accéder, soit fournir son adresse.

Attention toutefois, **le nom du tableau n'est pas un pointeur**, c'est seulement une constante connue du compilateur. Ecrire tabint = iptr n'a pas plus de sens qu'écrire 4 = iptr .... Par contre, puisqu'un pointeur contient une adresse du même type que le nom du tableau, on peut utiliser un pointeur avec la notation des tableaux. Voici un exemple ou l'on utilise un pointeur pour créer un adressage auxiliaire à partir du 10eme élément du tableau :

```
int tabint[50] ;
int *iptr ;
iptr = tabint ;
iptr = iptr + 10 ;
printf ("%d ", intptr[3]) ; /* affiche tabint[13]
```

On peut ainsi travailler commodément sur un sous-tableau d'un tableau - par exemple sur une ligne d'un tableau à deux dimensions. En fait, que l'on écrive iptr + 3 ou iptr[3], le compilateur produit la même traduction.

Dernier point à remarquer, tous les types de donnée peuvent être répétés, donc on peut faire des tableaux de n'importe quel type de donnée, pas seulement les types primitifs. Pour lire ces déclarations, la règle est simple : 1/ les * sont avant le nom de la variable, les [ ] après. 2/ en négligean le nom de la variable, ce qui est avant le [ ] est le type du tableau. Par exemple, `int * tabptr[12]` déclare un tableau dont chaque élement pourrait être déclaré par `int * nomvar` (donc est du type `int *`). Autrement dit, c'est un tableau de 12 pointeurs sur des entiers. Autre

exemple, `int tabtab[23][37]` déclare un tableau dont chaque élément pourrait être déclaré par `int nomvar[23]`, autrement dit un tableau de 37 tableaux (chacun de taille 23).

On peut aussi utiliser des tableaux de pointeurs ; par exemple, `int *tabintptr[30]` déclare un tableau de 30 pointeurs d'entiers (`*tabintptr[0]`, `*tabintptr[1]`, ...sont des entiers). De même, quand on utilise `char *argv[]` dans le main(), chaque argv[i] est un pointeur sur une chaîne de caractères : l'argument i auquel le système a ajouté un '\0' final.

5.3 Pointeurs, tableaux et fonctions

Nous avons vu qu'une fonction travaille toujours sur une copie de ses arguments, ce qui fait qu'elle ne peut pas les modifier. Nous avons aussi vu comment écrire une fonction qui modifie la variable qu'on lui indique à l'appel, en passant en argument **l'adresse** de la variable (le paramètre formel est donc un pointeur). Comme un tableau est en fait repéré par une adresse, on peut de la même façon modifier un tableau. Il faut pour cela faire attention à un point : pour écrire la fonction, on a besoin de connaitre l'adresse de début du tableau **et** sa taille. En général, on veut pouvoir travailler avec des tableaux de taille différentes. Le plus classique est de passer leur taille. Voici un exemple. C'est une fonction qui affiche n'importe quel tableau d'entier, pourvu qu'on le passe correctement :

```
1  #include <stdio.h>
2
3  /* La fonction affiche reçoit l'adresse du tableau et sa taille */
4  void affiche_tableau(int *adrtab, int taille){
5    int i ;
6    for(i=0; i< taille ; i++)  {
7      printf ("%d --",adrtab[i]); // on pourrait écrire *adrtab + i)
8    }//fin du for
9  }//fin de affiche_tableau
10
11 int main(void) {
12   int tabint1[] = {3, 7, 2, 6, 1, 45, 23} ;
13   int tabint2[] = {23, 5, 40, 8, -2, 39, -31,25, 62, 12 };
14   int longtab1 = 7 ; // longueur de tabint1
15   int longtab2 = 10 ; // longueur de tabint2
16
17    affiche_tableau (tabint1, longtab1) ;
18    printf ("\n");
19    affiche_tableau (tabint2, longtab2) ;
20    printf ("\n");
21 }//fin du main
```

TP5 : Pointeurs

Exercice 1 :

Question 1.1 : Déclarer un tableau `tab_alph` de 26 caractères et l'initialiser avec l'alphabet minuscule dans le désordre.

Question 1.2 : Afficher ce tableau à l'endroit, puis à l'envers.

Exercice 2 :

Question 2.1 : Faire un programme C qui déclare un tableau d'entiers et l'initialise (par ex. les 15 premiers nombres dans l'ordre), puis déclare un pointeur d'entiers et l'utilise pour parcourir le tableau et afficher son contenu (3 solutions différentes pour parcourir le tableau).

Question 2.2 : Modifier le programme précédent pour n'afficher qu'un nombre sur deux, d'abord ceux de rang pair, puis une autre version avec ceux de rang impair.

Exercice 3 :

On définit deux variables chaines `char prem[] = ''Zoe''; char deus[] = ''Aurelie'';`

Question 3.1 : Calculer leur longueur sans utiliser les fonctions décrites dans string.h

Question 3.2 : Comment peut-on les remettre dans l'ordre alphabétique (échanger leurs valeurs) ?

Question 3.3 : Avez-vous une meilleure idée pour écrire ce code ?

Exercice 4 :

Déclarez un tableau de caractères de dimension 2, de taille 4 * 6 (4 lignes de 6 colonnes). Initialisez le.

Question 4.1 : Affichez-le par lignes (4 mots séparés par un blanc).

Question 4.2 : Affichez-le par colonnes (6 mots séparés par un blanc).

Exercice 5 :

Ecrivez une fonction qui calcule la somme des éléments d'un tableau d'entiers quelconque.

Exercice 6 :

Ecrivez une fonction qui calcule la moyenne d'un tableau de flottants.

Exercice 7 :

Ecrire une fonction qui calcule le nombre de voyelles dans un tableau de caractères

Exercice 8 :

Question 8.1 : Ecrire un programme qui déclare un tableau de 25 entiers et les intialise en demandant à l'utilisateur de les taper au clavier. (astuce pour ne pas retaper les nombres à chaque fois qu'on fait un test : tapez les nombres dans un fichier `nombres.txt`, et, si votre executable s'appelle `bouclescan`, lancez-le par `bouclescan < nombres.txt`. Ce sera comme si vous tapiez à chaque fois les mêmes nombres au clavier.)

Question 8.2 : Utilisez une fonction pour afficher tous ces nombres

Question 8.3 : Ecrire une fonction qui reçoit un tableau d'entiers et sa taille, et qui renvoie vrai ou faux selon que le tableau est trié dans l'ordre croissant ou pas.

Question 8.4 : Testez cette fonction avec le tableau précédent.

Question 8.5 : Ecrire une fonction inserer() qui reçoit un tableau d'entiers et sa taille. Le tableau est trié dans l'ordre croissant, sauf le dernier élément qui peut être n'importe quoi. La fonction met le dernier élément à sa place par rapport au reste sans rien perdre, de façon que le tableau soit trié (cela s'appelle insérer le dernier élément).

Question 8.6 : Testez la fonction précédente sur quelques exemples

Question 8.7 : Que se passe-t-il si on utilise la fonction insérer() plusieurs fois sur le même tableau (la même adresse de tableau) avec une taille plus grande de 1 à chaque fois ?

Question 8.8 : Utiliser le résultat de la question précédente pour trier le tableau des entiers fournis par l'utilisateur.

Question 8.9 : Ecrire un programme qui calcule si l'utilisateur a rentré deux fois le même nombre.

Cours 6

Les structures de données en C

6.1 Introduction sur les structures de données

Jusqu'à présent la seule structure d'objet complexe qu'on a vu est la structure tableau à une ou plusieurs dimensions. C'est une structure d'objet complexe, dans le sens composé d'objets plus élémentaires. Dans le cas du tableau ces objets élémentaires sont des objets de même type. C'est pour cela qu'on peut les ranger dans des cases consécutives de même taille et donc numérotées.

6.1.1 Les structures de données pourquoi faire ?

Les types simples (`char`,`int`,`float` etc..) sont suffisants pour de petits problèmes mais en règle générale il est souvent utile de manipuler des types structurés.

– **Exemple 1** : Vous devez écrire un programme qui manipule deux points dans un plan :

```
1 int p1x;        /* l'abscisse du premier point */
2 int p2x;       /* l'abscisse du second point */
3 int p1y;       /* l'ordonnée du premier point */
4 int p2y;       /* l'ordonnée du second point */
```

Problèmes :
– La déclaration est **fastidieuse**,
– **Peu de lisibilité** : rien n'indique que `p1x`, `p1y` correspondent au premier point.

Solution : Avoir la possibilité de déclarer des variables de type **point** :

```
1 point p1;
2 point p2;
```

Mais le type point n'existe pas !!!!!!!!!

– **Exemple 2** : Le problème est encore plus criant dans l'exemple suivant :
Vous devez écrire un programme qui va travailler sur des étudiants (maximum 200). Chaque étudiant est défini par un *nom*, un *prénom*, une *date de naissance* :

```
1 char nomEt1[20];      /* nom du premier étudiant */
2 char nomEt2[20];      /* nom du deuxième étudiant */
3 ....
4 char prenomEt1[20];          /* prénom du premier étudiant */
5 char prenomEt2[200];  /* prénom du deuxième étudiant */
6 ....
```

```
int jj1Et1;              /* jour de naissance du premier étudiant */
int jjEt2;               /* jour de naissance du deuxième étudiant */
.....
```

Un peu mieux serait de définir des tableaux :

```
char nomEt[200] [20];           /* le tableaux des noms des étudiants */
char prenomEtu[200] [20];    /* le tableau des prénoms des étudiants */
int jjEt[200];       /* le tableau des jours de naissances des étudiants */
......
```

On sent bien quand même que cela pose un problème de lisibilité et d'utilisation !!!!!
Solution :
Avoir la possibilité de déclarer un tableau d'étudiants : `etudiant tabEtu[200];`
Mais le type etudiant n'existe pas !!!!!!!!!

6.2 Les structures de données en C

6.2.1 Définir une structure de données avec le mot clef struct

Une autre structure d'objet complexe est celle d'enregistrement. C'est une structure composée d'une juxtaposition d'objets quelconques appelés champs.
En C une structure est déclarée par la forme générale :

```
    struct [<nom>] { <ListeDeclarations> };
```

Plusieurs éléments interviennent dans la définition d'une structure de données :
- Le nom <nom> est celui de la structure et il va jouer un rôle de type pour déclarer des variables de cette structure. On parlera alors de **variables structurées**.
- *<ListeDeclarations>* donne les déclarations des différents champs de la structure. Chaque champ de la structure peut-être de n'importe quel type, c'est à dire de type `int`, `float`, `char`, etc... Mais aussi le champ peut-être lui aussi de type structuré ; nous verrons ce cas de figure plus loin dans ce cours.

6.2.2 Exemple

- **Exemple 1** : En ce qui concerne la représentation d'un *point du plan* :
  1. une abscisse x de type `int`,
  2. une ordonnnée y de type `int`.

  On pourra définir le type structuré `struct point` comme suit :

```
struct point
      {
         int x;
         int y;
      };
```

- **Exemple 2** : Supposons que nous voulons définir un type permettant de définir une *date* :
  1. un *jour* de type `int`,
  2. un *mois* de type chaîne de caractères,
  3. une *annee* de type `int`.

  Nous définirons en langage C le type `struct date` comme suit :

```
struct date {
 int jour;      /* champs représentant le jour */
```

```
3  char mois[10];   /* mois est une chaine représentant le mois */
4  int annee; /* le champ année qui représente l'année */
5  };
```

Attention !
- **struct date** et **struct point** deviennent ainsi des **nouveaux types**.
- Les champs d'un type structuré peuvent être **de n'importe quel type** (de bases, des tableaux, types structurés).
- Les structures se déclarent **en dehors du main()** avant les fonctions et les procédures.

- **Exemple 3** : Supposons que nous voulons définir un type *etudiant* dont les caractéristiques sont les suivantes :
  1. un *nom* de type chaîne de caractères,
  2. un *prenom* de type chaîne de caractères,
  3. une *date de naissance* de **struct date**. Souvenez-vous, **struct date** est un type structuré. Donc ce champ sera structuré et "encapsulera" le *jour*, le *mois* et l'*année* de naissance.

Nous définirons en langage C le type `struct etudiant` comme suit :

```
1  struct  etudiant
2        {
3           char nom[20];
4           char prenom[20];
5           struct date datNaiss;        /* Ce champ est structuré et il se réfère
6                                                au type struct date définit plus haut. */
7        };
```

Attention ! : La définition d'un type structuré **NE RESERVE PAS** d'espace mémoire. Elle déclare un nouveau type.

▶ Sur ce thème : **TD6, Exercice 1**

6.2.3 Déclarer des variables de types structurés

Maintenant, pour déclarer des variables de types structurés on peut écrire :

```
1  struct point p1;
2  struct date d;
3  struct etudiant et1;
```

où :
- `p1` est une variable structurée composée de deux champs : `x` et `y`,
- `d` est une variable structurée composée de trois champs : `jour`, `mois` et `annee`.
- `et1` est une variable structurée composée de trois champs : `nom`, `prenom` et `date` (qui est lui-même un type structuré composée de trois champs : `jour`, `mois` et `annee`.) .

C'est toujours l'écriture que vous connaissez déjà pour déclarer une variable :

```
1  <type> < variable >;
```

6.2.4 Utilisation des variables de types structurés

1. **Par affectation des champs individuellement** : On se réfère aux champs individuellement d'une variable structurée par l'opérateur **point** (`.`) ; Chaque champs se manipule comme une variable classique :
   - En utilisant l'opérateur `=` d'affectation :

```
struct point p1;
struct etudiant etud1;

p1.x = 20;
p1.y = 30;
 printf ("Valeur du champ x de p1:\%d\n", p1.x);
 printf '"Valeur du champ y de p1:\%d\n", p1.y);


strcpy(etud1.nom,"Dupond");
strcpy(etud1.nom,"Jacques");
etud1.dateNaiss.jour=1;
strcpy(etud1.dateNaiss.mois,"Decembre");
etud1. dateNaiss.mois.annee=1970;

printf("Valeur du champ nom de etud1:\%s\n", etud1.nom);
printf("Valeur du champ prenom de etud1:\%s\n", etud1.prenom);
printf("Valeur du champ jour de naissance de etud1:\%d\n", etud1.date.jour);
printf("Valeur du champ mois de naissance de etud1:\%s\n", etud1.date.mois);
printf("Valeur du champ annee de naissance de etud1:\%d\n", etud1.date.annee);
```

Observations :

- Comme dit plus haut, l'accès à un champ d'une variable structurée se fait par l'opérateur `.`, par exemple `p1.x=20;`,
- si un champ est lui même structuré, comme le champ `dateNaiss` du type `struct etudiant`, alors il faut utiliser l'opérateur `.` pour indiquer quel est le champ cible, par exemple `etud1.dateNaiss.jour=1;`,
- la fonction `strcpy()` permet d'affecter une chaine de caractère(s) à un tableau de caractère(s) qui représentera à partir de là une chaîne de caractère(s) ; l'opérateur `=` étant interdit dans ce cas, **voir "A propos des chaînes de caractères en C"**,
- pour afficher une variable structurée, il faut le faire **champs par champs**, par exemple

```
.....
 printf ("Valeur du champ x de p1:\%d\n", p1.x);
 printf '"Valeur du champ y de p1:\%d\n", p1.y);
....
```

2. **Par saisie au clavier** en utilisant la fonction `scanf()` :

```
struct etudiant etu;
 printf ("Donnez moi le nom de l'étudiant: ");
scanf("%s", etu.nom);
 printf ("\nDonnez moi le prénom de l'étudiant: ");
scanf("%s", etu.prenom);
 printf ("\nDonnez moi son jour de naissance:");
scanf("%d", &etu.datNaiss.jour);
 printf ("\nDonnez moi son mois de naissance:");
scanf("%s", etu.datNaiss.mois);
 printf ("\nDonnez moi son année de naissance:");
scanf("%d", &etu.datNaiss.annee);
```

Observations :

- Notez l'utilisation de l'opérateur `&` (qui fournit l'adresse de la variable concernée) pour la saisie des champs de types scalaires, par exemple `scanf("%d", &etu.datNaiss.annee);` car `etu.datNaiss.annee` est de type `int`,
- que l'opérateur `&` n'a pas été utilisé pour la saisie de champs de type *tableau de caractère(s)*, par exemple `scanf("%s", etu.datNais.mois);` car `etu.datNaiss.mois` représente déjà une adresse, celle du tableau `char mois[20]` de la structure `struct date`.

▶ Sur ce thème : **TD6, Exercice 2**

Initialisation à la déclaration : et à la **déclaration seulement**, en utilisant les accolades ouvrantes { et fermantes },comme pour les tableaux.

```
1       struct point p1 = {20,45};
2       struct date d1 = {27, "octobre", 2004};
3       struct etudiant etu = {"Dupond", "Benoit", {20, "avril", 85}};
4  /* Notez les accolades imbriquées pour l'initialisation du champ etu.dateNaiss */
```

▶ Sur ce thème : **TD6, Exercice 3**

3. **Par l'affetcation par bloc** d'une structure à une autre de **type !** :

```
1  struct point p1 = {12,20};
2  struct point p2;
3  p2 = p1;       /* Tous les champs de p1 sont affectés au champs de p2 */
```

▶ Sur ce thème : **TD6, Exercice 4**

6.3 Occupation mémoire des variables structurées de données en C

La déclaration d'une variable structurée réserve la place mémoire nécessaire pour la structure :

```
1     struct date d1;
2     struct etudiant tabEtu[200];  /* déclare un tableau de 200 éléments de type struct etudiant */
```

Observations :

- Pour la variable `d1` ce sont 14 octets qui sont réservés ; 2 pour le jour, 10 pour le mois et 2 pour l'annee,
- pour le tableau d'étudiants `tabEtu` ce sont 200*(20 (pour le nom) + 20 (pour le prénom) + 14 (pour la date) qui sont réservés.

Rappel : Vous pouvez utiliser l'opérateur `sizeof()` pour calculer la taille d'un type.

▶ Sur ce thème : **TD6, Exercice 5**

6.4 Les tableaux de structures

On peut déclarer des tableaux de structures :

```
1
2  <type> < variable > [<dimension>] ...
```

Par exemple, pour avoir un tableau (variable `etudiant`) constitué de 27 étudiants (structure `struct etudiant`).

```
1    struct etudiant s1groupeB2[27];
```

La dimension [27] suit donc le nom du tableau comme vous avez l'habitude de le faire avec les type simple, par exemple `int x[100];`
On utilise cette variable simplement comme tout autre tableau avec le *point* (.) pour chaque accéder à chaque champs.

```
1       int jour;
2       char c;
3       struct etudiant s1groupeB2[27];
4
5       ....
```

```
6
7       jour = s1groupeB2[i].dateNaiss.jour;    /* jour récupère le jour de naissance
8                                   de l'étudiant d'indice i */
9
10      c = employes[i].nom[0] /* récupère le premier caractère du nom de
11                              l'étudiant qui à pour indice 0 */
```

Mais il faut faire attention à l'affectation du genre : `etudiant[2].nom = etudiant[1].nom;` qui copie le nom du deuxième étudiant sur le troisième est une **instruction qui n'est pas correcte**. Il faut se rappeler que **les tableaux ne sont pas affectables** (il faut utiliser, soit un pointeur `char* nom` lors de la déclaration du champ `nom` dans la définition du type `struct etudiant`, soit utiliser la fonction `strcpy()` en gardant la définition actuelle) ; **voir "A propos des chaînes de caractères en C"**.
Par contre les structures, elles, sont affectables même au niveau sous-structures. Tout l'objet est copié.
Par exemple : `etudiant[2] = etudiant[1];` est correcte c'est une affectation en bloc.

A propos des chaînes de caractères en C ; le type chaîne de caractère(s) n'existe pas en C ! ! ! ! :

1. En fait le type chaîne de caractère(s) n'existe pas en C, un peu comme ce que vous aviez découvert pour le type *booléen* ! Que propose le C pour pallier à ce manque ?
2. Une chaîne de caractères n'est qu'une succession de caractères rangés dans une variable de type ... tableau de caractères ! Mais ...
3. C'est le caractère spécial **'\0'** de code ASCII 0 qui est la marque obligatoire de fin de chaîne (dernier caractère de votre tableau de caractère(s)). **Il faut donc prévoir une place pour ce caractère de fin de chaîne !** L'incidence c'est que pour stocker une chaine de n caractère(s), il faut prévoir un tableau ne $n+1$ caractère(s).
4. *""* est donc une chaîne vide qui contient un caractère, le caractère **'\0'** ,
5. la chaîne littérale *"bonjour"* représente une adresse de type `char *` qui est constante,
6. il existe une batterie de fonctions dont `strcpy()` qui permettent de traiter les chaînes de caractères... Ces fonctions font parties de la bibliothèque `<string>` ; il faudra, donc, rajouter dans votre source C la directive d'inclusion : `#include<string.h>`,
7. A quoi sert cette fonction ? L'affectation en bloc d'une chaine **n'est pas possible** que si la variable affectée est de type pointeur (de type `char *`) ; si c'est un tableau de caractère(s) qui est destiné a être affecté, l'affectation en bloc est interdite car le nom d'un tableau (quelque soit son type) est un pointeur **constant !**). Pour cela, la fonction `strcpy()` est là pour ça ; elle recopie caractère par caractère la chaîne source vers la chaîne destination.
8. une exception cependant, lors de la déclaration d'un tableau de caractères avec affectation immédiate d'une chaîne de caractère(s).

Voyons tout cela par des exemples ; vous devrez les "exécuter sur papier" et comprendre ce qui se passe ; votre enseignant vous aidera en cas de difficultés :

```
1
2   #include <stdio.h>
3   #include <string.h> /* Pour utiliser les fonctions relatives aux chaînes
4                                   de caractères */
5
6   int main(void){
7
8   /* Déclaration des variables */
9   char ch[3];
10  /* Correct ! C'est un tableau de caractères pouvant contenir jusqu'à 3 caractères */
11
```

```
12
13
14  char tab1[7] = "bonjour";
15  /*  Incorrect ! Le nombre de caractères nécessaires est
16  de  7 + 1 (pour le '\0') ; donc pas assez de place*/
17
18  char tab2[8] = "bonjour";
19  /* Correct car le nombre de caractères nécessaires est de 8 et le tableau dispose
20  de 8 cases!  Le marqueur de fin de chaîne '\0' est ajouté automatiquement.
21  Par contre impossible d'y mettre une chaîne plus grande */
22
23
24  char tab3[100]="bonjour, ca va";
25   /* Correct ! car 8 cases nécessaires, et nous en avons 100 disponibles.
26  Seules les 8 premières sont réquisitionnées. Par contre on pourra mettre
27  plus tard une chaîne d'au maximum 99 caractères car il faut prévoir
28  un caractère pour le caractère de fin de chaîne  '\0' .
29
30  char s[10] = {'b','o','n','j','o','u','r','\0'};
31  /* Correct ! Tableau initialisé case par case, avec la place nécessaire,
32  et le caractère '\0', mis explicitement ! En effet le marqueur de fin de chaîne
33  n'est pas mis automatiquement.*/
34
35  char s1[10] = {'b','o','n','j','o','u','r'};
36  /* Ce n'est pas une chaîne de caractères mais un tableau de caractères
37  car le marqueur de fin de chaîne n'a pas été affecté au tableau ! */
38
39
40  char *ptrChaine;
41  /* Correct ! On déclare un pointeur sur caractère; vous connaissez
42  les pointeurs maintenant :0) */
43
44  /* Début des instructions */
45
46  tab3 = "Le langage C ";
47  /* Incorrect, car tab3 est une adresse constante et représente
48   l'adresse de la première case du tableau tab3, donc la valeur de
49  tab3 ne peut pas être changée ! */
50
51  strcpy(tab3, "Le langage C ");
52  /* Correct ! tab3 a assez de place pour contenir la chaîne "bonjour",
53  tab3 contiendra la chaîne "bonjour" */
54
55  strcpy(ch, tab3);
56  /* Incorrect ! car ch n'a pas assez de place pour contenir la chaine contenue dans
57  tab3 */
58
59  ptrChaine = tab2;
60  /* Correct ! car ptrChaine est un pointeur "variable", ici il pointe sur la première
61  case du tableau tab2 */
62
63   printf ("%s\n", tab3);                  /* Affiche : Le langage C */
64   printf ("%s\n", ptrChaine);             /* Affiche : bonjour */
65
66  *ptrChaine='t';
67  /* On affecte le caractère 't' à la case pointée par ptrChaine, c'est à dire
68  la première case du tableau tab2 */
69
```

```
70  printf ("%s\n", tab2);                  /* Affiche : tonjour */
71
72  printf ("%s\n", ptrChaine);             /* Affiche : tonjour */
73
74 ptrChaine = ptrChaine + 1;
75
76  printf ("%s\n", ptrChaine);             /* Affiche : onjour */
77
78 return 0;
79 }
```

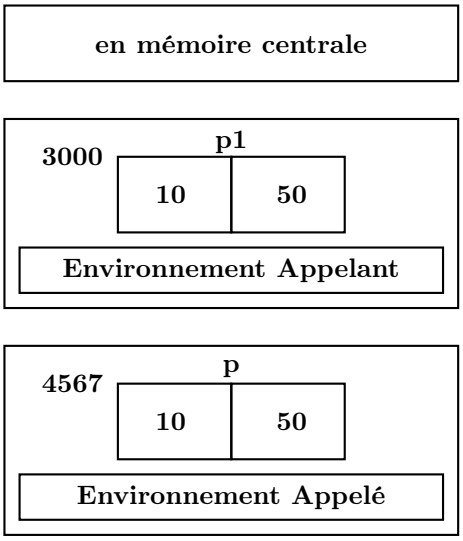
▶ Sur ce thème : **TD6, Exercice 4**

6.5 Comment utiliser les structures de données avec les fonctions et les procédures

6.5.1 types structurés en paramètres en entrées

Considérons l'exemple suivant :

```
1  struct point
2    {
3       int x;
4       int y;
5    };
6  void  afficherPoint  (struct point p)
7   {
8     printf ("Point de coordonnées (%d;%d) \n", p.x, p.y);
9   }
10 int main (void)
11  {
12    struct point p1 = {10,50};
13     afficherPoint (p1);
14
15 return 0;
16  }
```



Il y a passage par valeur, les champs des paramètres effectifs sont copiés dans les champs correspondants des paramètres formels. C'est à dire, dans notre exemple, la variable structurée `p1` est affecté en bloc à la variable `p`. **Toutes modifications de la variable structurée `p` dans la procédure `afficherPoint()` n'entraîne pas la modification de la variable `p1`**. C'est ce qu'on appelle un **transfert par valeur**.

6.5.2 types structurés en paramètres en valeurs retournées

Dans le cas ou la fonction a besoin de communiquer qu'**un seul résultat** de type structuré ; pour cela il est plus simple d'utiliser une fonction qui retourne une variable structurée :

```
1  struct point
2    {
3       int x;
4       int y;
5    };
6  struct point saisirPoint ()
7   {
8     struct point pTemp; /* variable locale à la fonction */
9
10     printf ("Donnez l'abscisse :");
11    scanf("%d",&pTemp.x);
12     printf ("Donnez l'ordonnee :");
13    scanf("%d",&pTemp.y);
14
15    return pTemp;              /* On retourne la variable structurée pTemps
16                                        au programme appelant */
17
18   }
19 int main (void)
20   {
21     struct point p1;
22     p1 = saisirPoint ();       /* p1 est affecté par bloc par la valeur retournée
23                                        par la fonction saisirPoint() */
24     afficherPoint (p1);
25
26     return 0;
27   }
```

Observations :

- `pTemp` est une variable locale à la fonction `saisirPoint()`, sa durée de vie et sa visibilité sont limité à cette fonction,
- c'est au moment de l'instruction `return pTemp;` que les champs de la structure sont renvoyés au programme appelant,
- c'est du ressort du programme appelant de récupérer la valeur retournée, par affectation dans notre exemple.

▶ Sur ce thème : **TD6, Exercice 6**
▶ Sur ce thème : **TD6, Exercice 7**

6.5.3 types structurés en paramètres en sorties ou en entrées/sorties

Dans le cas ou la procédure a besoin de communiquer **plusieurs résultats** dont certains sont de types structurés ; nous allons utiliser des pointeurs sur des variables structurées, comme vous l'avez fait pour les variables simples (`float`, `int`, etc.) :

1. Pour commencer, parlons des pointeurs vers les structures :
   (a) Pour avoir un pointeur vers une structure il suffit de le déclarer.

```
struct etudiant *pp;
```

déclare pp comme pointeur vers une structure `struct`.
On peut écrire pour accéder au champs `jour` du champ `dateNaiss` de la structure de donnée pointée par pp :

```
(*pp).dateNaiss.jour
```

L'opérateur `.` étant prioritaire sur l'indirection `*`, il faut parenthéser celle-ci. Mais il y a une autre notation d'utilisation. Au lieu de `(*pp).dateNaiss.jour`, on utilise l'opérateur `->` pour écrire plus simplement :

```
pp->dateNaiss.jour
```

pour accéder au champs `jour` du champ `dateNaiss` de la structure de données pointée par pp.
Le signe `->` est là pour le refléter.

Auparavant on aura initialisé (ne pas l'oublier) pp, par un étudiant qui existe déjà ; par exemple :

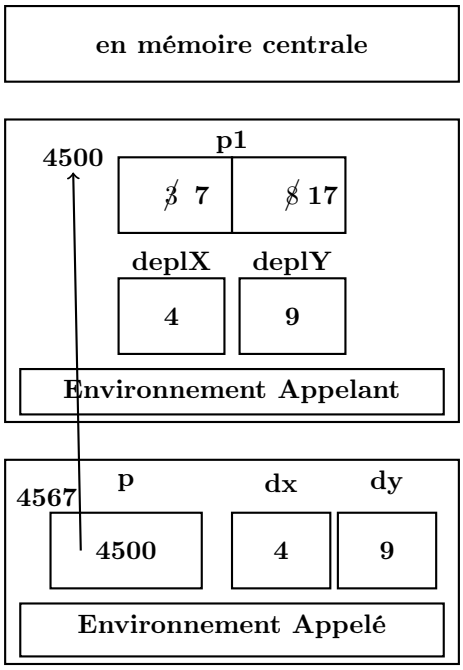
```
struct etudiant s1groupeB2[27];
struct etudiant *pp;

    /* initialisation du tableau d'étudiant(s) */
 ......

     pp = &s1groupeB2[20]; /* pp reçoit l'adresse du 21ième étudiant */
```

(b) Une variable pointeur vers une structure peut aussi subir de l'arithmétique. Comme les autres pointeurs. L'incrémenter par exemple, la fait pointer une structure plus loin.

```
struct etudiant s1groupeB2[27];
struct etudiant *pp;

/* initialisation du tableau d'étudiant(s) */
......

pp = &s1groupeB2[20]; /* pp reçoit l'adresse du 21ième étudiant */

 printf ("Nom : %s", pp->nom); /* Affiche le nom du 21ième étudiant */
pp=pp+1;                      /* pp pointe le 22ième étudiant */

 printf ("Nom : %s", pp->nom); /* Affiche le nom du 22ième étudiant */

pp = s1groupeB2;       /* pp recoit l'adresse de la première case
                                    du tableau étudiant*/

 printf ("Nom : %s", pp->nom); /* Affiche le nom du 1er étudiant */

pp = pp + 3;           /* pp reçoit l'adresse du 4ième étudiant */

 printf ("Nom : %s", pp->nom); /* Affiche le nom du 4ième étudiant */
```

2. Appliquons ce que nous savons sur les pointeurs sur les structures pour notre problème :

```
struct point
  {
```




```
3       int x;
4       int y;
5     };
6
7   /*
8           p       est un pointeur sur le type structuré struct point, ici c'est un paramètre
9                   en entrée sortie.
10          dx et dy sont transmis par valeur, ce sont des paramètres en entrées
11  */
12  void deplacerPoint (struct point* p, int dx, int dy)
13   {
14    (*p).x = (*p).x + dx;
15    (*p).y = (*p).y + dy;
16   }
17  void main (void)
18   {
19     struct point p1 = {3,8};
20     deplX = 4;
21     deplY = 9;
22     deplacerPoint(&p1, deplX, deplY);
23   }
```

Observations :

(a) Le programme appelant passe trois paramètres à la fonction `deplacerPoint()` :
- `&p1`, c'est à dire l'adresse de la variable structurée `p1`, dans l'exemple cette adresse vaut : 4500,
- `deplX` et `deplY` sont deux entiers transmis par valeur qui valent respectivement 4 et 9.

(b) `(*p).x` permet d'accéder au champs `x` de la variable structurée du programme appelant, c'est à dire la variable dont l'adresse est 4500.
Ainsi `(*p).x = (*p).x + dx;` permet d'ajouter `deplX` au champs `x` de la variable `p1` du programme appelant,
Ainsi `(*p).y = (*p).y + dy;` permet d'ajouter `deplY` au champs `y` de la variable `p1`

du programme appelant,

(c) Finalement la transmission par valeur de l'adresse de la structure de données `p1` a permis à la procédure de modifier indirectement la variable `p1` du programme appelant.

▶ Sur ce thème : **TD6, Exercice 8**

▶ Sur ce thème : **TD6, Exercice 9**

TD6 : Les types structurés en C

Exercice 1 : Les planètes d'un système planétaire

Les exercices suivants vont vous permettre d'écrire une application relative au traitement des données d'un système planétaire.
On caractérise chaque planète d'un système planétaire par :

- Son *nom* (20 caractères au maximum),
- sa *densité* (`float`)
- la *distance moyenne* de l'étoile autour de laquelle elle gravite (`float`) en 10E6 km,
- son *nombre de satellites* (`int`)

Définir un type structuré permettant de représenter une planète.

Exercice 2 : Compréhension des variables structurées

1. Déclarez une planète *p1*, et demandez à l'utilisateur de remplir les différents champs de cette planète, et enfin vous l'afficherez.
2. De quel type est la variable : `p1.nom[1]` ; essayez de changer cette valeur par une autre et ré-affichez p1 ; que remarquez-vous ?

Exercice 3 : Comprendre l'initialisation à la déclaration

1. Déclarez et initialisez à la déclaration les planètes suivantes :

nom	densité	distance moyenne	nombre de satellites
Mercure	5.42	58	0
Venus	5.25	108.2	0
Terre	5.52	149.6	1

2. Affichez ces planètes.

Exercice 4 : Comprendre l'affectation par bloc

Déclarez trois variables structurées de type `struct planete` et leur affecter les planètes précédentes. Affichez toutes les planètes.

Exercice 5 : Calcul de la taille mémoire occupée par les structures de données

1. Calculez sur papier la taille du type `struct planete` et le vérifier avec l'opérateur `sizeof`,
2. déclarez un tableau de 20 planètes et répondre aux mêmes questions que pour le point précédent.

Exercice 6 : Comprendre les paramètres en entrée et les valeurs retournées de types structurés

Vous remarquez que le nombre de lignes de code augmente pour chaque saisie et chaque affichage de planète. Comme vous l'avez appris, nous allons écrire des fonctions permettant de structurer notre application :

1. Ecrire une fonction en C, `struct planete creerPlanete()` qui invite à la saisie une planète au clavier et la retourne (par l'utilisation de `return`) ; mais avant **analysez** bien le prototype de cette fonction,

2. Ecrire une fonction en C, `void afficherPlanete()`, qui affiche **clairement** les caractéristiques de la planète reçue en paramètre ; mais avant, **analysez** bien le prototype de cette fonction,
3. Ecrire une fonction en C, `int egales(struct planete p1, struct planete p2)` qui compare les caractéristiques des planètes `p1` et `p2`. La fonction retourne 0 (signifie *false* en C) si les planètes ont des caractéristiques différentes, et une valeur différente de 0 (signifie *true* en C) si les deux planètes ont les mêmes caractéristiques.
4. finalement vous devriez pouvoir exécuter le programme suivant :

```
#include<stdio.h>
/* Fichier testPlanete.c */

/* planete.h : Contient la définition des structures de données,
les prototypes de vos procédures et fonctions,
le code de celles-ci pourra se trouver dans un fichier
 de nom planete.c */


#include"planete.h"


/* Debut du programme */
int main(void}{
struct planete p1,p2;

p1 = creerPlanete();
p2 = creerPlanete();

/* Affichage des caractéristiques de p1 devra apparaître à l'écran */
 afficherPlanete (p1);
/* Affichage des caractéristiques de p2 devra apparaître à l'écran */
 afficherPlanete (p2);

if (egales(p1,p2)==0) {
        printf ("p1 n'a pas les même caractéristiques que p2");
} else {
        printf ("p1 a les mêmes caractéristiques que p2?);
}
return 0;
```

Pour pouvoir exécuter ce programme vous devriez tapez en ligne de commande :

```
gcc -c planete.c -o planete.o
gcc -c testPlanete.c -o testPlanete.o
gcc testPlanete.o planete.o -o testPlanete
../testPlanete
```

Vous devrez tester plusieurs cas de figure pour valider le bon fonctionnement de vos fonctions et procédures.

Exercice 7 : Laissons les planètes un instant...

1. Ecrire une fonction `saisirDate()` qui demande la saisie au clavier d'une date et la retourne à l'appelant,
2. écrire une fonction `saisirEtudiant()` qui demande la saisie au clavier d'un étudiant et la retourne à l'appelant,
3. écrire une procédure `afficherEtudiant()` qui permet d'afficher un étudiant,

4. écrire un programme principal qui déclare un tableau de 5 étudiants et le remplit "au clavier",
5. et qui les affichent.

Exercice 8 : Modification des caractéristiques d'une planète

1. Ecrire une procédure `void modifiePlanete(...)` qui modifie les caractéristiques d'une planète donnée :
   – sa densité doit être multipliée par un facteur *dens*,
   – on a découvert *nSat* supplémentaires,
   – *dens* et *nSat* feront partie des paramètres.
2. Complétez votre application pour modifier la planète Mercure en multipliant sa densité par 1.2 et en lui rajoutant 3 satellites.
3. Affichez la planète Mercure pour vérifier que les modifications ont bien été faites.

Exercice 9 : Fonctions et procédures permettant de traiter un système de planètes

1. Ecrire une procédure `...afficherPlanetes(...)` qui affiche les `n` planètes contenues dans un tableau de planètes,
2. écrire une procédure `...initSysteme(...)` qui initialise un système planétaire comportant `n` planètes,
3. écrire une fonction `... nbMoy(..)` qui renvoie le nombre moyen de satellites par planète ainsi que la densité moyenne des planètes d'un système de `n` planètes,
4. écrire une fonction `... modifieSysteme(..)` qui modifie toutes les planètes du système palnétaire de `n` planètes en multipliant leur densité par un facteur *dens* et rajoutant un nombre supplémentaire de satellite(s) découvert(s) *nPSup* ; ces données seront reçues en paramètres.
5. écrire un programme principal qui :
   (a) Déclare un système planétaire de 4 planètes,
   (b) initialise au clavier ce système,
   (c) affiche ce système,
   (d) calcule et affiche le nombre moyen de satellite(s) des planètes du système planétaire,
   (e) multiplie la densité de toutes les planètes par un coefficient demandé à l'utilisateur, de même , rajoute un nombre de planète(s) décidé par l'utilisateur,
   (f) affiche le système modifié.

TD6 : ANNEXE

Voici quelques planètes avec leurs caractéristiques :

nom	densité	distance moyenne	nombre de satellites
Mercure	5.42	58	0
Venus	5.25	108.2	0
Terre	5.52	149.6	1
Mars	3.94	227.9	2
Jupiter	1.314	778.3	16
Saturne	0.69	1427	17
Uranus	1.19	2869	15
Neptune	1.6	4496	2
Pluton	1.8	5900	1

Cours 7

Fichiers

Ce chapitre est consacré aux principales fonctions d'entrées/sorties que l'on peut utiliser dans un programme `C`. Celles-ci font partie de la bibliothèque `stdio.h`.

7.1 Ouverture et fermeture d'un fichier

7.1.1 Descripteur de fichier

Dans un programme `C`, on accède à un fichier via son *descripteur*. C'est une structure contenant tous les renseignements nécessaires pour gérer l'accès au fichier par le programme. Un descripteur de fichier a le type `FILE`, défini dans la bibliothèque `stdio.h`. Les fonctions de manipulation des fichiers n'utilisent pas directement cette structure, mais un pointeur sur une telle structure.

7.1.2 Fichiers particuliers

Les *entrées/sorties* standards sont gérées comme des fichiers. Les descripteurs qui leur sont associés sont :

- entrée standard : `stdin` (*standard input*)
- sortie standard : `stdout` (*standard output*)
- sortie erreur standard : `stderr` (*standard error*)

7.1.3 Ouverture d'un fichier

Avant de pouvoir travailler sur un fichier, il faut l'*ouvrir* avec la fonction`fopen`. Cette opération crée et remplit la structure descripteur de fichier. Sa syntaxe est la suivante :

```
1 FILE *fopen(char *nom_fichier, char *mode);
```

Les paramètres de la fonction sont le nom du fichier sur disque, `nom_fichier`, avec éventuellement son chemin d'accès, ainsi qu'un `mode` d'ouverture. Les différents modes sont indiqués dans le tableau 7.1.

Table 7.1 – Modes d'ouverture d'un fichier

Mode	Description
`r`	Ouverture en lecture d'un fichier existant
`w`	Ouverture en écriture d'un fichier. S'il existe, il est détruit.
`a`	Ouverture pour écriture *en fin de fichier*. S'il n'existe pas, il est créé.

Si l'ouverture du fichier réussit, la fonction `fopen` renvoie un pointeur sur la structure `FILE` associée. Dans le cas contraire, elle renvoie un pointeur nul.

7.1.4 Fermeture d'un fichier

Lorsque l'on a fini d'utiliser un fichier, on le *ferme* à l'aide de la fonction `fclose` :

```
int fclose(FILE *fichier);
```

La fonction `fclose` renvoie `0` en cas de succès de la fermeture et `-1` en cas d'erreur.

Même si la fin d'un processus ferme tous les fichiers ouverts, il vaut mieux les fermer explicitement dès qu'on n'en a plus besoin, pour éviter des modifications malencontreuses.

Le programme suivant ouvre un fichier de nom `toto` en lecture, teste le résultat de la fonction d'ouverture du fichier et affiche un message, puis ferme le fichier.

```
#include <stdio.h>

int main(){
        FILE *fich;

        fich = fopen("toto","r");
        if (fich == NULL){
                printf("Erreur de fopen\n");
               return(1);
        }
        printf("L'ouverture s'est bien passée\n");
        fclose(fich);
}
```

Si le fichier n'existe pas, l'ouverture en lecture ne peut avoir lieu, et le message `Erreur de fopen` est donc affiché.

7.2 Lecture/écriture binaire

Les lecture et écriture dans un fichier peuvent s'effectuer soit sur un *nombre de données* précisé, soit selon un format. Dans le premier cas, on parle de *lecture/écriture binaire*.

7.2.1 Écriture binaire

L'*écriture binaire* s'effectue à l'aide de la fonction `fwrite` :

```
size_t fwrite(void *ptr,size_t taille, size_t nb_elements, FILE *fichier);
```

Cette fonction écrit dans le `fichier` au plus `nb_elements` de longueur `taille` octets contenus dans la zone mémoire pointée par `ptr`. Elle retourne le *nombre d'éléments effectivement écrits*.

Le programme suivant ouvre un fichier `toto` en écriture et y écrit 5 caractères contenus dans la chaîne de caractères `Bonjour`.

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

int main(){
        FILE *fich;
        int nb;
        char buf[20];

        fich = fopen("toto","w");
        if (fich == NULL){
```

```
11                 printf ("Erreur de fopen\n");
12                 return(1);
13         }
14         /* L'ouverture s'est bien passée */
15         /* mise en place dans buf de la chaîne à écrire */
16         strcpy(buf,"Bonjour");
17         /* écriture de 5 caractères de buf */
18         nb = fwrite(buf,sizeof(char),5,fich);
19         printf ("Nb de caractères écrits : %d\n",nb);
20         fclose (fich);
21 }
```

L'exécution de ce programme conduit d'une part à l'affichage de :

```
Nb de caractères écrits : 5
```

et d'autre part, à l'écriture du fichier `toto` qui contient alors :

```
Bonjo
```

c'est-à-dire les cinq premiers caractères de la chaîne `Bonjour`.

7.2.2 Lecture binaire

La *lecture binaire* est similaire à l'écriture binaire. Elle utilise la fonction `fread` :

```
1 size_t fread(void *ptr,size_t taille, size_t nb_elements, FILE *fichier);
```

Cette fonction lit dans le `fichier` au plus `nb_elements` de longueur `taille` octets et les place dans la zone mémoire pointée par `ptr`. Elle retourne le *nombre d'éléments effectivement lus*.

Le programme suivant ouvre un fichier `toto` en lecture et y écrit au plus 10 caractères qu'elle place dans une zone pointée par `buf`.

```
1  #include <stdio.h>
2
3  int main(){
4          FILE *fich;
5          int nb;
6          char buf[20];
7
8          fich = fopen("toto","r");
9          if (fich == NULL){
10                 printf ("Erreur de fopen\n");
11                 return(1);
12         }
13         /* L'ouverture s'est bien passée */
14         /* lecture de 10 caractères dans buf */
15         nb = fread(buf,sizeof(char),10,fich);
16         printf ("Nb de caractères lus : %d\n",nb);
17         printf ("Chaîne lue : %s\n",buf);
18         fclose (fich);
19 }
```

Si le fichier de nom `toto` est celui écrit par l'exemple 7.2.1, l'exécution de ce programme affiche :

```
Nb de caractères lus : 5
Chaîne lue : Bonjo
```

Par conséquent, bien que l'instruction `fread` doive lire 10 caractères, elle n'a pu en lire que 5.

7.2.3 Fin de fichier

La fonction `feof` (*end of file*) permet de tester si la fin du fichier est atteinte, par exemple suite à plusieurs lectures :

```
int feof(FILE *fichier);
```

Elle renvoie un *entier positif* si la fin de fichier est atteinte ;

Le programme suivant est similaire à celui de l'exemple 7.2.2, mais lit les caractères deux par deux.

```
#include <stdio.h>

int main(){
        FILE *fich;
        int nb;
        char buf[20];

        fich = fopen("toto","r");
        if (fich == NULL){
                printf("Erreur de fopen\n");
                return(1);
        }
        /* L'ouverture s'est bien passée */
        while (!feof(fich)){
                /* la fin du fichier n'est pas atteinte */
                /* lecture de 2 caractères dans buf */
                nb = fread(buf,sizeof(char),2,fich);
                printf("Nb de caractères lus : %d\n",nb);
                printf("Chaîne lue : %s\n",buf);
        }
        printf("Fin du fichier.\n");
        fclose(fich);
}
```

Si le fichier de nom `toto` est celui écrit par l'exemple 7.2.1, l'exécution de ce programme affiche :

```
Nb de caractères lus : 2
Chaîne lue : Bo
Nb de caractères lus : 2
Chaîne lue : nj
Nb de caractères lus : 1
Chaîne lue : oj
Fin du fichier.
```

On remarque que la dernière lecture n'a lu qu'un seul caractère, mais que la chaîne `oj` est affichée. En effet, le second caractère dans `buf` était `j`, et il n'a pas été écrasé par un caractère de fin de chaîne.

7.3 Lecture/écriture d'une chaîne de caractères

Il est possible d'accéder directement à la lecture ou l'écriture d'une chaîne de caractères.

7.3.1 Lecture d'une chaîne de caractères

La fonction `fgets` permet de lire une chaîne de caractères dans un fichier :

```
char * fgets(char *ptr, int n, FILE *fichier);
```

Elle lit depuis le `fichier` une chaîne d'au plus `n` caractères et la place dans la zone mémoire pointée par `ptr`. La lecture s'arrête lorsque la fin du fichier est atteinte où que l'on rencontre un retour à la ligne `\n`. La valeur retournée est la chaîne elle-même ou `NULL` en cas de fin de fichier ou d'erreur.

7.3.2 Écriture d'une chaîne de caractères

La fonction `fputs` permet d'écrire une chaîne de caractères dans un fichier :

```
1 char * fputs(char *ptr, FILE *fichier);
```

Elle écrit dans le `fichier` la chaîne de caractères pointée par `ptr`. La valeur retournée est le dernier caractère écrit ou `NULL` en cas d'erreur.

Le programme suivant affiche la chaîne de caractères `Coucou`.

```
1 #include <stdio.h>
2
3 int main(){
4         fputs("Coucou\n",stdout);
5 }
```

7.4 Lecture/écriture formatée

Les lectures et écritures formatées travaillent selon un format identique à celui utilisé par `printf`.

7.4.1 Écriture formatée

L'*écriture formatée* est réalisée par la fonction `fprintf` :

```
1 int fprintf(FILE *fichier,char *format, liste_variables);
```

Elle se comporte de manière identique à `printf`, et écrit dans le `fichier` passé en paramètre.

7.4.2 Lecture formatée

L'écriture formatée est similaire au `scanf` :

```
1 int fscanf(FILE *fichier,char *format, liste_adresses_variables);
```

On remarque que les variables *doivent être passées par adresse*. La fonction `scanf` lit sur l'entrée standard.

Le programme suivant demande à l'utilisateur d'entrer un entier et une chaîne de caractères, puis les affiche.

```
1  #include <stdio.h>
2
3  int main(){
4          int i;
5          char buf[20];
6
7          printf("Entrer un entier puis un mot : ");
8          scanf("%d %s",&i,buf);
9          printf("Vous avez entré : %d et %s\n",i,buf);
10 }
```

TP7 : fichier

Exercice 1 : Caractères

ecrivez une fonction c qui prend en argument le nom d'un fichier et affiche le nombre de caractères de ce fichier.

Exercice 2 : Écriture d'un fichier texte

ecrivez un programme qui affiche le contenu d'un fichier texte en faisant précéder chaque ligne par son numéro.

Exercice 3 : lecture d'un fichier

ecrivez un programme qui affiche le nombre de ligne contenue d'un fichier texte.

Exercice 4 : caractère majuscules

ecrivez un programme qui affiche le contenu d'un fichier texte, en passant tous les caractères en majuscules.

Exercice 5 : Écriture d'un fichier texte

ecrivez un programme qui crée un Fichier texte nommé "tableX.txt" qui contienne la table de multiplication pour le nombre X (le fchier "table7.txt" contiendra la table de 7), présentée sous la forme suivante :

```
1 x 7 = 7
2 x 7 = 14
etc...
9 x 7 = 63
```

Exercice 6 : Lectures et écritures de chaînes

1. Écrire un programme en C qui lit 10 chaînes de caractères tapées par l'utilisateur (d'une longueur maximale de 100 caractères) et qui les écrit dans un fichier dont le nom sera demandé à l'utilisateur.
2. Ecrire une fonction en C `lireChaine` qui ouvre un fichier texte en lecture et qui affiche à l'écran tous les enregistrements (d'une longueur maximale de 100 caractères) lus dans le fichier. Le nom du fichier est en paramètre. Ecrire un programme principal qui appelle cette fonction avec le fichier créé dans la question précédente).
   Rappel : en fin de fichier, `fgets]` positionne un indicateur qu'on l'on teste avec la fonction `feof`.

Exercice 7 : Lectures et écritures formatées

1. Ecrire une fonction en C `sauverEntText` qui sauvegarde tous les nombres compris entre 100 et 110 (à raison d'un par ligne) dans un fichier texte dont le nom est passé en paramètre. Ecrire un programme principal qui appelle cette fonction.
2. Ecrire une fonction en C `lireEnt` qui lit et affiche à l'écran les enregistrements d'un fichier texte dont le nom est en paramètre. Ces enregistrements sont des entiers. Ecrire également un programme principal qui appelle cette fonction avec le fichier créé à la question 1).
   Rappel : en fin de fichier, `fscanf` renvoie `EOF` et positionne un indicateur qu'on l'on teste avec la fonction `feof`.

Cours 8

Révisions et compléments

Exercice 1 : La représentation des chaînes

Question 1.1 : Quelle est la taille du tableau nécessaire pour stocker la chaîne de caractères "bonjour tout le monde" ? Déclarer et initialiser ce tableau.

Question 1.2 : Écrire un programme en C qui affiche la chaîne précédente en une seule fois puis caractère par caractère.

Question 1.3 : Quel est l'affichage produit par l'exécution du code ci-dessous :

```
1       char ch[]="hurlement";
2       char *pt1, *pt2;
3       pt1=&(ch[3]);
4       pt2=&(ch[5]);
5       pt1[2]='v';
6        printf ("%s:%s:%s",pt2,pt1,ch);
```

Exercice 2 : Arguments des fonctions

Que fait le programme suivant ?

```
1  float somme1(float x, float y){
2     return(x+y) ;
3  }
4
5  float somme2(float *x, float *y){
6     return(*x+*y) ;
7  }
8
9  float somme3(float x, float y){
10   if(x > y) {
11     float tmp ; tmp = x ;
12     x= y ; y = tmp ;
13   }
14    return(x+y) ;
15 }
16
17 float somme4(float *x, float* y){
18   if(*x > *y) {
19     float tmp ; tmp = *x ;
20     *x= *y ; *y = tmp ;
21   }
22    return(*x+*y) ;
```

```
}

void somme5(float *x, float* y, float *z)){
  *z += *x + *y ;
}

void somme6(float *x, float* y, float *z){
  if(*x < *y) {
    float tmp ; tmp = *x ;
    *x= *y ; *y = tmp ;
  }
  *z += *x + *y ;
}

int main(void) {
  float prem = 12.5, sec = 5.2 ;
  float res, *fptr1, *fptr2 ;
  res = somme1(prem, sec) ://1
  somme5(&prem, &sec, &res) ;//2
  res = somme2(&prem, &sec) ;//3
  res = somme3(prem, sec) ://4
  res = 0 ;//5
  res = somme4(&prem, &sec) ;//6
  res = somme4(&prem, &sec) ;//7
  res = 0 ;//8
  somme6(&prem, &sec, &res) ;//9
  somme6(&prem, &sec, &res) ;//10
```

Exercice 3 : Algorithmes sur les chaînes

Dans la bibliothèque `string.h`, la fonction `strlen` calcule la longueur d'une chaîne de caractères (sans tenir compte du caractère de fin de chaîne) ;

Question 3.1 : Écrire une fonction en C qui transforme une chaîne en l'inversant. Par exemple, si on lui passe la chaîne

```
bonjour
```

l'algorithme retourne la chaîne

```
ruojnob.
```

.

Question 3.2 : Écrire une fonction qui indique si une chaîne de caractères représentant une expression parenthésée est syntaxiquement correcte du point de vue des parenthèses ou non. La fonction renvoie -1 si l'expression est correcte et la position du caractère où la première erreur a été détectée si l'expression est incorrecte.

exemple :

- pour "(a.(b))" elle retourne -1 ;
- pour "(()())" elle retourne -1 ;
- pour "a.(b))(" elle retourne 5 ;
- pour "(()(()a.(b))" elle retourne 12 ;
- pour ")(" elle retourne 0 ;

Exercice 4 : Nombres d'entiers pairs et d'entiers impairs

Question 4.1 : Écrire en C une fonction qui calcule le nombre d'entiers pairs, le nombre d'entiers impairs et le nombre d'entiers multiples de 3 d'un tableau d'entiers, de façon que le code appelant puisse utiliser ces résultats.

Question 4.2 : Écrire un programme principal en C appelant cette fonction

Exercice 5 : Copie d'une sélection d'éléments d'un tableau

Question 5.1 : Écrire en C une fonction `rangePositif` qui permet de ranger dans un tableau `tRes` les entiers positifs ou nuls contenus dans un tableau `t` de `nb` éléments effectifs. Bien réfléchir aux paramètres utiles.

Question 5.2 : Écrire un programme principal en C permettant de tester cette fonction.

Exercice 6 : Suppressions d'éléments dans un tableau

On veut écrire une fonction `enleveValeurs` en C qui modifie le tableau `t` en enlevant toutes ses valeurs égales à `r`.

Par exemple, si le tableau `t` contient les 7 éléments effectifs suivants :

3 5 6 3 8 4 3

Après l'appel à la fonction qui enlève la valeur 3, le tableau ne contiendra plus que les 4 éléments suivants :

5 6 8 4

En revanche, après l'appel à la fonction qui enlève la valeur 9, le tableau n'est pas modifié, il contient le même nombre d'élément qu'avant l'appel de la fonction.

Question 6.1 : Donner le prototype de la fonction `enleveValeurs`. Quels sont les paramètres de la fonction.

Question 6.2 : Écrire en langage C la fonction `enleveValeurs`.

Question 6.3 : Écrire un programme principal en C qui corresponde au premier exemple donné.

Exercice 7 : Retour aux chaînes

Question 7.1 : Écrire une fonction en C qui insère des tirets bas ('_ ') entre chaque lettre d'une chaîne. Par exemple, si on lui passe la chaîne

```
1 bonjour
```

l'algorithme retourne la chaîne

```
1 b_o_n_j_o_u_r
```

. On suppose le tableau qui contient la chaine suffisamment grand pour ajouter les tirets.