Langage C

Licence 1 SRIT

KONAN HYACINTHE

2 Programme minimal

Entrons directement dans le vif du sujet avec un premier programme écrit en C, le fameux *Hello World* :

```
#include <stdio.h>
int main(void)
{
    puts("Hello World\n");
    getchar();
    return 0;
} OK
```

Pour tester ce programme, vous devez l'entrer dans l'edi, le sauvegarder (par exemple avec le nom hello.c), le compiler, faire l'édition de liens, de manière à obtenir un fichier qui se nommera probablement hello.exe. Ce dernier fichier est un programme exécutable que vous pouvez lancer depuis l'edi ou depuis le gestionnaire de fichiers en double-cliquant dessus.
Contrairement à Python, C est un langage qui nécessite une phase de compilation et d'édition de liens. Ces étapes transforment le ou les fichiers sources en programme exécutable (le .exe sous Windows). Le programme exécutable obtenu est indépendant (aux bibliothèques liées dynamiquement près) et fonctionnel. Pour s'en servir, l'utilisateur final n'a besoin que de cet exécutable (et de quelques bibliothèques éventuellement) : il n'a plus besoin du compilateur C.
Notre programme d'exemple comporte une directive à l'adresse du préprocesseur (c'est la ligne qui commence par #), et une fonction. Cette fonction est spéciale, c'est la fonction main de notre programme : son point d'entrée.
La fonction main ne comporte que 3 lignes :
— la première affiche Hello World sur l'écran ;
— la seconde marque une pause jusqu'à ce que l'utilisateur ait pressé une touche ;
— la dernière quitte en envoyant un *code de retour*, 0, qui informe le système d'exploitation que tout s'est bien déroulé.
Les fonctions puts et getchar ne sont pas des *instructions* du C, mais des *fonctions* qui appartiennent à la bibliothèque standard. Précisément ces deux fonctions font partie de la bibliothèque standard d'entrées/sorties. La première ligne indique au préprocesseur dans quel fichier trouver les *prototypes* des fonctions utilisées dans le programme. Le fichier stdio.h est un fichier d'*en-tête* (c'est le sens du .h (*header*)) qui contient diverses déclarations, dont celles des fonctions de la bibliothèque standard d'entrées-sorties dont font partie les fonctions puts et getchar.
Les programmes que nous écrirons suivront toujours le même squelette :

```
/* Directives préprocesseur */
#include ....
#define ...
/* Fonctions annexes */
/* Fonction principale */
int main(void) {
...
return 0;
}
```

Chaque fichier source comporte généralement des directives au préproceseur (commençant par #), éventuellement des déclarations globales, puis des fonctions, et (très souvent à la fin) la fonction principale main qui renvoie un entier, par convention 0 pour indiquer que tout s'est bien déroulé.

3 Variables

Comme dans la plupart des langages, le C utilise des variables. Les règles de nommage sont sans surprise (consultez la norme pour les connaître), et le C est sensible à la casse.
En C, le typage est (assez) fort et statique. Cela signifie que les conversions de types doivent être (souvent) explicites, et que le type des variables est fixé à leur déclaration. Un point important par rapport à la programmation Python est justement la déclaration préalable des variables.

Voici un exemple :

```
int a,b;
float c;
char d='E';
a=5;
b=6;
c=1.01;
```

Les trois premières lignes permettent de *déclarer* les deux variables entières (**int**) a et b, le nombre à virgule (**float**) c et le caractère (char) d. Au passage, nous voyons qu'on peut affecter une variable au moment de sa déclaration (c'est le cas pour d).

Affichage du contenu des variables
On utilise généralement la fonction printf pour afficher le contenu des variables. Cette fonction est un peu difficile à utiliser. On lui passe en paramètre le *format*, qui décrit ce qui va être affiché, et les variables, qui seront substituées aux chaînes de la forme %... dans le format.
Les formats les plus utilisés sont %c pour les caractères, %i (ou %d) pour les entiers, %f pour les nombres à virgule (on peut restreindre le nombre de décimales à afficher) et %s pour les chaînes de caractères.

```
#include <stdio.h>
int main(void) {
    int a = 5;
    float p = 3.1415926535;
    char c = 'P';
    printf("a vaut %i et p vaut %f\n", a, p);
    printf("p vaut environ %.4f\n", p);
    printf("C contient %c\n", c);
    getchar();
    return 0;
} OK
```

4 Tests, boucles et fonctions

4.1 Tests

Les tests utilisent la construction ordinaire **if**, avec un **else** facultatif. Les blocs sont matérialisés par des accolades. La condition doit être entre parenthèses.

Indentation
En Python, les blocs étaient matérialisés par l'indentation du code. Puisqu'en C, les blocs sont matérialisés par des accolades, l'indentation n'est plus nécessaire. Il est néanmoins habituel d'indenter proprement le code pour en faciliter la lecture. En particulier, il est peu probable que le chargé de tp accepte (et il aura bien raison) de relire un programme C mal indenté...

```
#include <stdio.h>
int main(void)
{
    int a=1001;
    if (a%2==0)
    {
        printf("%d est pair \n", a);
    }
    else
    {
        printf("%d est impair \n", a);
    }
    getchar();
    return 0;
} OK
```

La seconde partie de l'instruction (**else**) est facultative.

4.2 Boucles

La boucle **while** est très ordinaire (comme avec un **if**, la condition est entre parenthèses) :

```
#include <stdio.h>
int main(void)
{    int a=100;
    int c=-1;
    while (a!=c)
    {
        printf("Entrez un nb entre 0 et 99 ");
        scanf("%d",&c);
        if (c>a) printf("Votre nombre est trop grand\n");
        if (c<a) printf("Votre nombre est trop petit\n");
    }
    printf("Bravo;");
    getchar();
    return 0;
} OK
```

La ligne **scanf**("%d",&c) permet à l'utilisateur d'entrer un nombre entier, qui sera stocké dans la variable c.

4.3 Fonctions

Lors de l'écriture d'une fonction, il faut préciser le type des paramètres, et le type de valeur de retour :

```
int pgdc (int a, int b)
{
        if (a==b) return a;
        if (b>a) {return pgdc(b%a,a); }
        else {return pgdc(a%b,b); }
}
```

Chapitre II Le C, plus en détails

1 Éléments de syntaxe

1. les instructions simples se terminent par un point virgule ;
2. les conditions sont entre parenthèses (dans les **if** et **while**)
3. les blocs qui suivent **if**, **else**, switch, **while**, **for** ou la définition d'une fonction (dès qu'il est nécessaire de limiter un bloc d'instructions) sont placés entre accolades. Le bloc fait généralement partie de l'instruction, si bien qu'on ne trouve pas de ; *avant* une accolade ouvrante.

2 Types

Le C possède plusieurs types numériques, et différentes longueurs de codages. Nous allons les énumérer ici : **Les entiers** peuvent être signés ou non. Par défaut, ils sont signés. Ils existent en plusieurs longueurs de codage : short, **int**, **long** et **long long**. Chacun de ces types peut être préfixé par unsigned pour indiquer que les nombres ne sont pas signés. La longueur du codage dépend du type de machine utilisé. Le programme suivant utilise l'opérateur sizeof pour indiquer la taille en octet de ces types :

```
#include<stdio.h>
int main(void)
{
        printf("%d %d %d %d\n", sizeof(short), sizeof(int), sizeof(long), sizeof(long long));
        return 0;
} OK
```

Sur la machine de test, ce programme affiche :
2 4 4 8

Les constantes entières peuvent être données en décimal (43), en hexadécimal (0x2B) ou en binaire (0b101011).
Les nombres à virgule sont représentés par un codage en virgule flottante (norme ieee 754) en simple (**float**) ou double (**double**) précision. Les constantes virgule flottante peuvent être représentées en notation scientifique (2.54e-5)
ou non (0.00023) mais doivent toujours comporter un point décimal (**42** est un entier alors que **42.0** ou même **42.** Et **.37** sont des nombres à virgule flottante).
Les caractères (type char), sont codés sur un octet (par défaut signé, mais qui peut être précédé de unsigned.
Il y a équivalence entre un caractère et son code ascii. Selon qu'on l'affiche (avec printf par exemple) comme un nombre ou comme un caractère, la même variable de type char donnera le code ascii (un nombre) ou le caractère.
Les constantes caractères sont notées entre guillemets simples '. Les caractères spéciaux peuvent être codés par des \ :
'**\n**' pour un retour chariot par exemple.
Quelques caractères spéciaux sont mentionnés dans la table ci-dessous. Les caractères peuvent aussi être donnés sous la forme de leur code ascii en décimal :
'\65' ou en hexadécimal :
'\x41' pour 'A' par exemple.

Quelques caractères spéciaux en C	
\a	bip
\b	backspace
\n	saut de ligne
\t	tabulation
\\	antislash
\’	guillemet simple

Booléens et complexes La norme C99 a ajouté les deux types **complex** (avec les variantes **float** et double) et **bool**. Les seules constantes de type **bool** étant true et false.

2.1 Conversions implicites et *cast*

Les opérations arithmétiques fonctionnent sur des opérandes de même type (**int** avec **int** ou **float** avec **float** par exemple).
Il est pourtant possible d’additionner un **int** et un **float**. Pour cela, le compilateur réalise des conversions de types implicites, uniquement lorsque c’est nécessaire. Les données sont converties de telle manière que la valeur soit *dégradée le moins possible* (voir plus loin pour un exemple où la donnée est dégradée). Par exemple, convertir un **float** en **int** entraîne des modifications de valeur (généralement) plus gênante que convertir un **int** en **float**.
La hiérarchie de conversion est la suivante :
int → long → float → double
Une simple affectation peut en outre mener à une conversion, systématiquement vers le type de la variable à affecter :

```
int a;
a=4.8;
```

Ce qui précède est valide. La valeur 4.8 est convertie en **int** avant d’être affectée à la variable a (qui contiendra donc 4).
Enfin, un opérateur de conversion explicite est disponible (opérateur de *cast*). La priorité de cet opérateur est plus élevée que celle des opérations arithmétique. En ajoutant à ceci les conversions implicites, nous pouvons étudier le bout de code suivant :

```
int n=5,d=9;
float f1,f2,f3,f4;
f1=n/d;
f2=(float) n/d;
f3=(float) (n/d);
f4=(float) n / (float) d;
printf("%f %f %f %f\n", f1, f2, f3, f4);
```

— Lors du calcul de f1, on divise deux **int**, le résultat est donc un **int** (0), converti en **float** pour l'affectation : 0.0
— Lors du calcul de f2, n est converti en **float** (présence du *cast*). Nous avons une opération entre un **float** et un **int**. L'entier est donc converti en **float** avant de donner un résultat de type **float**, affecté à f2 : 0.555...
— À cause des parenthèses, le *cast* est appliqué au résultat qui, nous l'avons vu pour f1 est nul. Le 0 est donc converti en 0.0 puis est affecté à f3.
— Dans le dernier cas, les deux opérandes sont converties en **float** par le *cast*. Le résultat est donc 0.5555...
Le programme précédent affiche donc : **0.0 0.5555 0.0 0.5555**.

Opérateurs arithmétiques en C	
Opérations	**Signification**
+	addition
-	soustraction
*	multiplication
/	division
%	reste de la division entière

Dans le calcul qui précède, les règles de priorité des opérateurs arithmétiques impliquent que * est la première opération.
c étant un **int** et 1.8 un **float**, c est converti en **float** et le résultat de la multiplication est un **float** : 82.8. Puis, il y a une addition entre un **int** (32) et un **float**. La valeur 32 est donc convertie en **float** et le résultat vaut 114.8.
Ce résultat étant affecté à un **int**, une nouvelle conversion est réalisée, et f contiendra finalement 114.

2.2 Occupation mémoire

L'opérateur sizeof permet de connaître l'occupation mémoire d'une variable ou d'un type :
— si n est déclaré ainsi : **int** n; alors sizeof(n) vaudra 4, la taille en octet d'un entier de type **int** ;
— sizeof(**float**) vaut 4, puisque les flottants simple précision sont codés sur 4 octets.

3 Opérateurs et expressions

3.1 Opérateurs de base

Le C dispose d'opérateurs arithmétiques (table II.2), de comparaisons (table II.3), logiques (table II.4) et de manipulations de bits (table II.5).
Le résultat d'un opérateur de comparaison est généralement vrai ou faux. Le C (avant la norme C99) ne disposait pas de type booléen, et ce sont donc des nombres qui sont utilisés avec la convention suivante : 0 vaut faux et toute valeur non nulle vaut vrai. Le résultat d'un opérateur de comparaison sera 0 pour faux et 1 pour vrai. Ainsi : 3<6 vaut 1.
De même, le connecteurs logiques manipulent aussi des nombres avec les mêmes conventions. Ainsi, a<b && b<c vaudra 1 si a, b, c sont 3 nombres strictement croissants et 0 sinon.
Tous ces opérateurs peuvent être mêlés dans des instructions et des expressions. S'il est préférable, dans le doute, de parenthéser, on peut se reporter aux règles de priorité (règles de précédence) données table II.6.
Ainsi cette expression : a=5*3+1>6+2&&5&2 se lit : a = (((5*3)+1)> (6+2)) && (5&2) ce qui n'est pas forcément très lisible non plus... et vaut 0.

Table II.3 – Opérateurs de comparaison en C	
Op	**Signification**
>	strictement supérieur à
<	strictement inférieur à
>=	supérieur ou égal à
<=	inférieur ou égal à
==	égal à
!=	différent de

Table II.4 – Connecteurs logiques en C	
Op	**Signification**
&&	et
||	ou
!	non

Table II.5 – Manipulations de bits en C	
Op	**Signification**
&	et
|	ou inclusif
^	ou exclusif
<<	décalage à gauche
>>	décalage à droite
~	complément à un

Table II.6 – Priorité de quelques opérateurs (décroissante de haut en bas)
++ -- ! ~
* / %
+ -
<< >>
< > <= >=
== !=
&^|
&&
||
? : (opérateur ternaire)
= += -= *= /= (toutes les affectation)
,

Exercice : Arithmétique, comparaison, et connecteurs logiques
Écrivez une expression booléenne indiquant si *n* est une année bissextile. Les années bissextiles sont les années multiples de 4, excepté si elles sont aussi multiples de 100 sans l'être de 400.
Question subsidiaire : avec ce calcul, quelle est la durée moyenne d'une année civile ?

Exercice : Manipulations de bits
Calculez :
— 186 & 203
— 186 | 203
— 186 ^ 203
— 186 << 2
— 203 >> 3

3.2 Autres opérateurs

Le C possède un opérateur conditionnel ternaire, utilisant les symboles ? et : . Cet opérateur permet d'écrire des expressions dont le calcul est différent selon qu'une condition est remplie ou non : condition ? expr1 : expr2
L'expression qui précède aura la valeur de expr1 si condition est vraie et la valeur de expr2 sinon.
Par exemple : a > b ? a : b vaut nécessairement le plus grand des deux nombres a et b.

4 Entrées sorties

Dès les premiers programmes nous aurons besoin, soit d'avoir un retour sur le terminal d'une valeur calculée, soit d'utiliser une valeur entrée par l'utilisateur.
Les affichages sont réalisés par la fonction printf qui prend en premier argument le *format* (une chaîne de caractères),
et en arguments supplémentaires, des expressions. Le format permet de préciser le type d'affichage que l'on souhaite pour chacune des expressions : nombre entier, caractère, nombre à virgule etc... :

```
#include<stdio.h>
int main(void)
{
    float a;
    char c;
    int i;
    printf("Bonjour a tous\n");
    printf("Bonjour, vive %i\n",42);
    a=42.;
    i=42;
    printf("Attention, %i n est pas %f\n", i, a);
    printf("Un petit 42 scientifique : %e\n",a);
    c='\x42';
    printf("Mais 42, est aussi (en hexa) %x, ou %c\n",c ,c);
    return 0;
} OK
```

Table II.8 – Quelques exemples de formats	
Format	**Signification**
%d	entier décimal
%ld	entier long décimal
%09d	entier décimal sur 9 chiffres complété par des 0 à gauche
%f	nombre à virgule flottante
%.3f	nombre à virgule avec 3 chiffres après la virgule
%e	nombre à virgule flottante en notation scientifique
%s	chaîne de caractères (délimitée par un espace)
%c	caractère
%x	nombre hexadécimal

Quelques exemples de formats sont donnés dans la table II.8.
La fonction scanf marque une pause et *lit des valeurs formatées* sur l'entrée standard. Voici quelques exemples :

```
#include<stdio.h>
int main(void)
{
    int n;
    float v;
    printf("Entrez un nombre entier puis un nombre a virgule");
    scanf("%i%f", &n, &v);
    printf("Vous avez entre %i et %f\n", n, v);
    return 0;
} OK
```

Le fait que les variables doivent être préfixées par & sera expliqué dans le chapitre sur les pointeurs. Lors qu'on souhaite saisir des chaînes de caractères, la fonction scanf n'est pas toujours pratique, car elle découpe l'entrée en mots (en fait la saisie s'arrête sur tout type de *blanc* : espace, tabulation, retour chariot....). Il est alors conseillé d'utiliser la fonction fgets, à laquelle on fournit un endroit où stocker la chaîne de caractères (voir section 11 sur les chaînes), la taille maximum de la saisie et le flot d'entrée (voir section 8). Pour une saisie au clavier, le flot d'entrée sera stdin :

```
#include<stdio.h>
int main(void)
{
    char texte[256];
    printf("Entrez une phrase (255 caractères max)\n");
    fgets(texte, 256, stdin);
    printf("Merci d'avoir dit :\n%s\n", texte);
    return 0;
} OK
```

5 Instructions et expressions

Dans de nombreux langages, on ne confond pas ces deux notions. En C, **la plupart des instructions sont aussi des expressions** et peuvent à ce titre être utilisées dans un calcul.
Une simple affection i=5; est une instruction. Mais c'est aussi une expression, qui vaut la valeur affectée (donc 5).
Nous pouvons donc écrire j=(i=5). Le symbole d'affectation étant associatif à droite (Associatif à droite signifie que a=b=c équivaut à a=(b=c) et non pas à (a=b)=c (associatif à gauche).), on omet généralement les parenthèses pour écrire : j=i=5;.
Les opérateurs d'affectation élargie, comme i++ sont des instructions, mais aussi des expressions. Ainsi, dans le programme suivant :

```
#include<stdio.h>
int main(void)
{
        int i, j;
        i=5;
        j=i++;
        printf("%d %d\n", i, j);
        return 0;
} OK
```

la seconde ligne ajoute 1 dans i et réalise une affectation. Mais dans quel ordre ? La différence entre i++ et ++i réside justement dans cet ordre. Si nous écrivons i++, c'est la valeur de i initiale qui est utilisée dans l'expression, *puis* i est augmenté de 1. Si au contraire nous écrivons ++i, alors la valeur de i est d'abord augmentée, puis cette nouvelle valeur est utilisée dans l'expression.
Le programme précédent affiche donc 6 5. Si nous avions écrit ++i, il aurait affiché 6 6.
Une instruction simple (qui se termine par un ;) peut être composée de plusieurs expressions, séparées par des virgule.
Auquel cas, les expressions sont évaluées de manière séquentielles, de gauche à droite, et l'instruction aura pour valeur celle de la dernière expression évaluée. Ainsi le programme suivant affichera 5 10 4.

```
#include<stdio.h>
int main(void)
{
   int i=4,b,a;
   b=(a=i++,2*i);
   printf("%d %d %d\n",i,b,a);
   return 0;
} OK
```

En effet, lors de l'évaluation de la ligne : b=(a=i++,2*i); on commence par évaluer la parenthèse (Les parenthèses sont ici nécessaires, car en leur absence, l'instruction serait équivalente à (b=a=i++),2*i; car la précédence de = est supérieurs à celle de ,.). Il y a deux expressions séquentielles. La première a=i++ copie i dans a (qui vaut maintenant 4) et augmente i de 1 (qui vaut maintenant 5). Puis, la seconde expression, 2*i est évaluée, et vaut 10. La valeur des deux expressions séquentielles étant celle de la dernière, la parenthèse vaut 10. Cette valeur est affectée à b qui vaut maintenant 10.
Naturellement, cet exemple est volontairement tordu et permet d'illustrer les possibilités et les pièges. Un programmeur n'a jamais intérêt (sauf dans le cadre d'un jeu) à écrire un programme difficile à comprendre, pour lui ou pour un autre. Hormis dans le cadre d'un exercice de style particulier, c'est un défaut d'écrire des programmes qui sont difficiles à comprendre, et non pas une preuve des grandes capacités du programmeur.

6 Instructions et structures de contrôle

On distingue en C trois types d'instructions :
1. Les instructions simples, comme a=3; ou i=collatz(27);. Elles se terminent nécessairement pas un ; (Une instruction simple peut être composée, et les morceaux qui la composent sont séparés par des , comme dans : i=4,b=3*i+1;)
2. Les instructions structurées (tests, boucles...), qui contiennent souvent un ou plusieurs blocs et que nous allons détailler dans la suite.
3. Les blocs, qui sont délimités par des accolades, et qui peuvent contenir des instructions simples, des blocs et des instructions structurées (il peut aussi être vide...).
L'acquisition de ce vocabulaire est nécessaire à la compréhension de la syntaxe (Faut-il un ; à tel endroit ? Faut-il des accolades ?...)

6.1 Tests

L'instruction **if** existe sous deux formes, selon qu'elle est accompagnée de **else** ou non.
La syntaxe générale (Gardez à l'esprit qu'une «instruction» peut être : une instruction simple, un bloc, ou une instruction structurée...) est :

```
if (condition)
    instruction
```

```
if (condition)
        instruction 1
else
    instruction 2
```

Exercice

Parmi les portions de code suivantes, relevez celles qui sont valides et celles qui ne le sont pas. Donnez les raisons de leur invalidité.

1) **if** (a==3) printf("Quel beau a...\n");

2)
```
if a==b
{
        printf("a et b...\n");
        printf("...sont égaux...\n")
}
```

3)
```
if (a==b)
        printf("a et b...\n");
        printf("...sont égaux...\n");
else
        printf("a et b diffèrent\n");
```

4)
```
if (a==b)
        printf("a et b...\n");
        printf("...sont égaux...\n");
```

5)
```
if (a<=b)
        if (b<=c)
                printf("Triés\n");
        else
                printf("Non Triés\n");
else printf("Non Triés\n");
```

L'instruction switch est une instruction de branchement, qui rend certaines portions de code plus lisibles. Voici sa syntaxe générale :

```
switch (expression)
{
        case constante_0 : instructions...
        case constante_1 : instructions...
        ...
        default : instructions...
}
```

La partie default est facultative. Le déroulement est le suivant : l'expression est évaluée (par exemple à 42), puis la série de case est examinée. Le branchement est réalisé au **premier** case qui convient (celui pour lequel la constante vaut 42). Les instructions qui suivent ce case sont exécutées. Attention, il s'agit d'une série d'instructions, et non d'un bloc.

Penser au break

Attention, une fois le branchement effectué, la série d'instruction est exécutée, jusqu'à la fin du bloc (tous les case sont exécutés à partir du premier convenable). La suite d'instruction se termine donc généralement par l'instruction **break**, qui permet de ressortir du bloc switch.

Voici un exemple d'utilisation :

```
#include<stdio.h>
int main(void)
{
	int a;
	printf("saisir un entier A");
	scanf("%d",&a);
	switch(a%3)
	{
		case 0 : printf("A est un multiple de 3\n");
		break;
		case 1 : printf("A est congru à 1 modulo 3\n");
		break;
		default : printf("A n'est pas congru à 1 ou 0 modulo 3\n");
		break;
	}
 return 0;
} OK
```

La boucle **while** s'écrit :

```
while (expression) instruction
```

Ici aussi, l'instruction peut être simple, structurée, ou peut être un bloc.
Voici un exemple d'utilisation :

```
#include<stdio.h>
int main(void)
{
	int n;
	printf("saisir un entier N");
	scanf("%d",&n);
	while(n%7!=0)
	{
		printf("%d n est pas un multiple de 7...\n",n);
		n=n+1;
	}
	printf("Le premier multiple de 7 trouve est %d\n",n);
	return 0;
} OK
```

Il existe un second type de boucle, pour lequel la condition est évaluée en *fin* de boucle :

```
do instruction
while (expression);
```

Voici un exemple d'utilisation :

```
#include<stdio.h>
int main(void)
{
	int a, p;
	a=100;
	do
	{
		printf("Entrez un nombre P");
		scanf("%d",&p);
		if (p>a) printf("Trop grand...\n");
		if (p<a) printf("Trop petit...\n");
	} while (p!=a);
	printf("Bien joue...\n");
	return 0;
} OK
```

Le langage C dispose aussi d'une instruction **for** :

```
for (expression1 ; expression2 ; expression3)
	instruction
```

Comme précédemment, instruction peut être simple, structurée ou bloc. Les trois expressions, qui figurent dans les parenthèses sont évaluées à différents moments :
— expression1 est évaluée une seule fois, avant d'entrer dans le boucle.
— expression2 est évaluée avant chaque tour de boucle (y compris le premier). Le tour est effectuée si expression2 est vraie. Sinon, on passe à la suite.
— expression3 est évaluée à chaque tour, après l'exécution de instruction.

On peut construire une boucle équivalente avec un **while** :

```
expression1
while (expression2)
{
	instruction
	expression3
}
```

Notons enfin que chacune des expressions dans la parenthèse du **if** peut être vide.
Nous avons vu l'opérateur virgule (,) qui permet de réunir plusieurs expression en une seule. Il est rarement utilisé, mais a de l'intérêt dans le cas des boucles **for** et permet par exemple d'écrire :

```
for (i=0,k=1; i<10 ; i++,k*=2)
        printf("2**%d=%d\n",i,k);
```

6.3 Branchements

Le langage C contient trois instructions de branchement : **break**, **continue**, goto. Elles sont à éviter, exception faite de **break** qui est presque indispensable si on utilise l'instruction switch.
L'instruction **break** permet de sortir prématurément de la boucle (ou du switch) la plus petite qui le contient.
L'instruction **continue**, utilisée dans une boucle permet d'ignorer la fin du corps de boucle et de passer directement au tour suivant. On peut utiliser ces instructions, mais uniquement si elles simplifient grandement l'écriture. Autrement, mieux vaut les éviter car elles nuisent la plupart du temps à la lisibilité.
L'instruction goto permet de se brancher sur une instruction de son choix, repérée par un label :

```
#include<stdio.h>
int main(void)
{
   printf("La ligne avant le Goto\n");
   goto fin;
   printf("Personne ne veut m'exécuter\n");
   fin : printf("C'est fini\n");
   return 0;
} OK
```

Sauf cas exceptionnel (programme très court, nécessité d'optimisation...), cette instruction n'est jamais utilisée.

7 Fonctions

En C, on ne fait pas la distinction entre procédures et fonctions. Il n'y a que des fonctions (qui peuvent ne rien renvoyer). Contrairement à Python, les paramètres d'une fonction doivent être typés. De même le type de retour est fixe.
Voici un exemple de fonction :

```
double exposant(double x, int n)
{
        double res;
        if (n==0)
                return 1;
        if (n%2==0)
                res = exposant(x*x, n/2);
        else
                res=exposant(x*x, n/2)*x;
        return res;
}
```

La première ligne comporte les informations suivantes :
— le type de retour est double (premier mot de la ligne)
— le nom de la fonction est exposant
— la fonction prend deux paramètres : le premier est un double et le second est un **int**.
Voici comment utiliser cette fonction dans un programme :

```
#include <stdio.h>
double exposant(double, int);
int main(void)
{
        int a=13;
        double val, x=3;
        val=exposant(x,a);
        printf("%f EXPOSANT %d=%f\n", x, a, val);
        return 0;
} OK
```

Notez la ligne : double exposant(double,**int**); qui permet au compilateur de connaître la signature de la fonction.
Cette ligne de «déclaration», qu'on appelle aussi *prototype de la fonction* pourrait aussi figurer *dans* la fonction main. On peut s'en passer si la fonction elle même est définie avant son utilisation dans le code. Néanmoins, écrire systématiquement la ligne de déclaration des fonctions est une bonne habitude.

Paramètres formels
Les noms des paramètres utilisés dans la fonction (x et n dans l'exemple) ne servent qu'à décrire ce que fait la fonction en interne : ce sont des paramètres formels. Modifier le nom de ces paramètres (modifier x en y par exemple) implique que l'on modifie le *corps* de la fonction (on remplace x par y partout), mais pas qu'on modifie quoi que ce soit *en dehors* de la fonction.
Ce principe est similaire à celui de la définition d'une fonction mathématique : $f(x) = 3x + 2$.
On pourrait écrire de manière équivalents $f(y) = ...$, à condition de remplacer x par y dans l'expression : $f(y) = 3x + 2$. Mais cette modification ne change pas la fonction (ni ce qu'elle calcule, ni la manière de l'utiliser) : elle s'appelle toujours *f*, et associe toujours le même nombre au même argument.

L'instruction return interrompt la fonction
En plus de permettre de préciser la valeur qui sera retournée, **return** stoppe l'exécution de la fonction, ce qui est parfois pratique pour éviter trop de **if** ou **else** imbriqués. Dans l'exemple qui précède, si n vaut 0, la fonction retourne 1 et stoppe son exécution au niveau du **return** 1, si bien que le reste de la fonction n'est même pas considéré.

Les fonctions qui ne renvoient rien (et qui sont donc des procédures) doivent annoncer void comme type de retour. De même, si une fonction ne prend pas de paramètre, la signature doit indiquer **int** mafonction(void) (les compilateurs acceptent généralement la syntaxe C++ **int** mafonction()).

Type de retour par défaut
Attention, si on ne précise pas le type de retour, il est pris par défaut égal à **int** et non pas à void....

7.1 Portée des déclarations

La portée d'une variable s'étend de sa déclaration jusqu'à la fin du bloc qui contenait cette déclaration. Si la variable n'est pas déclarée à l'intérieur d'un bloc, sa portée s'étend jusqu'à la fin du fichier source.

```
#include <stdio.h>
        int n;
        void affn(int a)
        {
                int b=5;
                printf("%d %d %d\n", a, b, n);
        }
        int m;
        void affm(void)
        {
                printf("%d\n", m);
        }
int main(void)
{
        int c=3;
        n=10;
        m=6;
        affn(c);
        affm();
        return 0;
} OK
```

Dans l'exemple qui précède la portée de la variable globale n est l'ensemble du fichier. La porté de m est en revanche limitée à affm et main. La portée de a et b est la fonction affn uniquement. La portée de c est la fonction main.
Il est possible de partager une variable entre plusieurs fichiers en utilisant le mot-clé extern.

mot clé static

Le mot clé static, lorsqu'il accompagne la déclaration d'une variable locale à une fonction, permet à cette dernière de conserver sa valeur entre deux appels. Si la variable est initialisée sur la ligne de déclaration, l'initialisation n'est réalisée qu'au premier appel de la fonction.

8 Fichiers

L'utilisation de fichiers se fait généralement en trois temps :
1. ouverture du fichier (en lecture ou écriture)
2. lecture ou écriture dans le fichier
3. fermeture du fichier

8.1 Ouverture et fermeture d'un fichier

La fonction fopen permet d'ouvrir les fichiers en lecture ou en écriture.

```
#include <stdio.h>
FILE * f;
f=fopen("toto.txt", "rw");
```

La fonction renvoie un pointeur vers un descripteur de fichier (FILE*). En cas d'échec, fopen renvoie NULL.
Les principaux modes d'ouverture des fichiers sont :
— r : lecture
— w : écriture
— a : écriture en fin de fichier (append)
On dispose aussi de :
— r+ (rw) : lecture/écriture (le fichier doit exister)
— w+ : lecture/écriture (le fichier est créé)
— a+ : écriture en fin de fichier et lecture
— b : mode binaire (à ajouter au mode d'ouverture)
La fonction fclose, qui prend comme unique paramètre un pointeur vers un descripteur de fichier, permet de refermer le fichier après utilisation.

8.2 Lecture et écriture

Les deux fonctions printf et scanf possèdent leur pendant pour les fichiers : fprintf et fscanf. Ces deux dernières fonctions prennent en premier argument supplémentaire un pointeur vers le descripteur de fichier.
L'exemple qui suit ouvre le fichier data.txt (contenant les nombres 10 et 90, et se trouvant sur le disque d:), calcule la somme des nombres à virgule qui s'y trouvent, puis ajoute ce total à la fin du fichier somme.txt (contenant le nombre 99, et se trouvant sur le disque d:).

```
#include <stdio.h>
int main(void)
{
	FILE * in, * out;
	float sum=0,val=0;
	in=fopen("d:data.txt","r");
	if (in==NULL)
	{
		printf("L'ouverture du fichier d'entrée a échoué\n");
		return -1;
	}
	while(fscanf(in,"%f",&val)!=EOF)
	{
		sum+=val;
	}
	fclose(in)
	out=fopen("d:somme.txt","a");
	if (out==NULL)
	{
		printf("L'ouverture du fichier de sortie a échoué\n");
		return -1;
	}
	fprintf(out,"%f",sum);
	fclose(out);
} OK
```

8.3 Fichiers binaires

Les deux fonctions fread et fwrite permettent de lire et écrire des octets dans un fichier.
La fonction fread prend en premier paramètre un pointeur vers la zone mémoire où inscrire les données lues, en second paramètre le nombre de données à lire, en troisième paramètre la taille d'une donnée en octets, et en dernier paramètre, un pointeur vers le descripteur de fichiers.
La fonction fwrite prend en premier paramètre un pointeur vers la zone mémoire où prendre les données à écrire dans le fichier, en second paramètre le nombre de données à écrire, en troisième paramètre la taille d'une donnée en octets, et en dernier paramètre, un pointeur vers le descripteur de fichiers.

9 Pointeurs

Les pointeurs sont très largement utilisés en C, mais sont parfois délicats à manipuler.
Nous avons vu la notion de variable qui peut contenir une valeur d'un certain type. En mémoire, cette valeur est inscrite à une certaine adresse. On peut donc parler d'adresse d'une variable. Cette adresse est constante, pendant toute la durée de vie de la variable, et si on n'utilise pas les pointeurs, deux variables ne peuvent normalement pas avoir la même adresse.
Il est possible de connaître l'adresse d'une variable avec l'opérateur & :

```
#include <stdio.h>
int main(void)
{
	int a=3;
	printf("A EST STOCKEE A L'ADRESSE %p\n", &a)
} OK
```

Nous voyons au passage qu'une adresse peut être formatée en hexadécimal, grâce au format %p. Cette adresse est elle même une valeur... et nous pouvons donc la stocker dans une variable. Le type de cette nouvelle variable sera *pointeur*. Si l'adresse est censée contenir un entier, le type de pointeur sera *pointeur vers un entier* :

```
int main(void)
{
	int a=3;
	int *t;
	t = &a;
	printf("A EST STOCKEE A L'ADRESSE %p \n", t);
} OK
```

L'intérêt des pointeurs consiste à donner la possibilité de modifier une variable (par exemple a), à partir de son adresse, ou d'un pointeur qui pointe vers a (par exemple t). Faire cette opération met en jeu un nouvel opérateur, l'opérateur d'*indirection* qui sert en quelque sorte à désigner *le contenu de la variable dont l'adresse est dans le pointeur* :

```
int main(void)
{
	int a=3;
	int *t;
	t=&a;
	*t = *t + 1;
	printf(" A CONTIENT : %i\n", a);
} OK
```

La ligne *t=*t+1 ajoute 1, non pas dans t (Pour ajouter 1 dans t, il faut écrire simplement t=t+1... En fait, ce n'est pas tout à fait exact, car t=t+1 ajoute réellement à t la quantité nécessaire pour «avancer» d'un entier. La valeur ajoutée vaut donc probablement 4 (un entier est codé sur 4 octets).), mais dans la variable pointée par t, c'est à dire dans a.

scanf et &

Le sens du & présent devant les variables avec la fonction scanf doit maintenant être plus clair. Avec scanf, nous ne devons pas fournir le contenu de la variable (qui ne contient rien d'intéressant avant l'appel à scanf), mais bien l'adresse de la zone mémoire dans laquelle scanf doit écrire les données saisies par l'utilisateur. Nous reviendrons sur ceci section 12.

scanf et chaînes

C'est pour la même raison que lors d'une saisie de chaîne avec scanf (format %s), le & n'est pas nécessaire puisque, comme nous allons le voir section 11, la chaîne est *déjà* désignée par un pointeur....

Pointeurs génériques

Le type de pointeur void* est dit générique. L'objet désigné par ce type de pointeur peut être de n'importe quel type.

10 Tableaux

Les tableaux sont des collections ordonnées d'éléments homogènes (contrairement aux listes de Python, dont les éléments peuvent être hétérogènes).
En C (avant la norme C99), la taille des tableaux doit être connus *à la compilation* 8. De plus, la taille des tableaux doit rester fixe (pas de fonction append ou extend comme en Python).
L'indice des tableaux commence classiquement à 0, et tout comme les autres variables, les tableaux doivent être déclarés :

```
int main(void)
{
	int mtab[20];
	int i;
	mtab[0] = 0;
	mtab[1] = 1;
	for (i=2; i < 20 ; i++)
	{
		mtab[i] = 3*mtab[i-1] - 5*mtab[i-2];
	}
	for (i=0;i<20;i++)
		printf("%d\n", mtab[i]);
} OK
```

Les tableaux peuvent avoir plusieurs dimensions (plusieurs indices), auquel cas, il suffit de préciser la taille sur chaque dimension à la déclaration :

```
int t[4][3];
```

En mémoire, les éléments sont rangés de manière séquentielle. Pour connaître l'ordre de rangement des éléments des tableaux multidimensionnels, on fait varier le dernier indice en premier : t[0][0], t[0][1], t[0][2], t[1][0],...
Attention à ne pas confondre la *dimension* d'un tableau, qui correspond au nombre de paires de crochet, et la taille, qui correspond au nombre d'éléments (en tout ou le long d'une dimension).
Les tableaux peuvent être initialisés sur la ligne de déclaration, en énumérant les valeurs entre accolades. Dans ce cas, il n'est pas forcément utile de préciser la taille du tableau si elle peut être déduite du nombre d'éléments utilisés dans l'initialisation :

```
int t1[]={4,5,6,7,8};
int t2[3]={10,11,10};
int t2[2][3]={{2,3,4},{4,5,6}};
```

11 Chaînes de caractères

11.1 Déclaration et allocation

Le C ne dispose pas d'un véritable *type* chaîne. Les chaînes seront représentées par de simples suites d'octets (la plupart du temps un tableau de char). Pour des raisons pratiques, la chaîne se terminera par l'octet de valeur 0. Cette convention permettra, bien que le tableau accueillant la chaîne soit de longueur fixe, de connaître la longueur *réelle* de la chaîne qu'il contient (en repérant la position du 0 final).
Voici plusieurs façons de déclarer et d'initialiser une chaîne de caractères. Certaines font appel aux pointeurs.

```
char chaine1[]="Programmer en C";
char chaine2[50]="Programmer en C";
char chaine3[]={'P','r','o','g','a','m','m','e','r',' ','e','n',' ','C','\0'};
char * chaine4 = "Programmer en C";
```

L'utilisation des constantes entre doubles guillemets nous évite de devoir penser au 0 final (qui est précisé pour chaine3). Si la taille du tableau n'est pas précisée, elle est calculée au plus juste : 15 caractères + le 0 final c'est à dire 16 caractères.

11.2 Fonctions sur les chaînes

La bibliothèque standard permet de réaliser des opérations sur les chaînes de caractères. Le fichier d'en-tête contenant les déclarations de ces fonctions est string.h.

Voici quelques fonctions parmi les plus utiles :

```
// Recopie une chaîne, la destination doit être allouée
char * strcpy(char *dest, const char *src);

// Recopie une chaîne au bout d'une autre
char * strcat(char *dest, const char *src);

// Donne la longueur d'une chaîne (sans compter le 0 final)
size_t strlen(const char *s);

// Compare deux chaînes (donne un classement type dictionnaire)
int strcmp(const char *s1, const char *s2);

// Recherche une chaîne dans une autre
char *strstr(const char *meule_de_foin, const char *aiguille);
```

Voici des portions de code utilisant ces fonctions :

```
char str1[]="Il est morne ";
char str2[]=" taciturne";
char str4[255];
char *str5;
strncpy(str4,str1,6);
strcat(str4,str2);
printf("%s\n",str1);
printf("%s\n",str4);
/* Attention !!! */
strcpy(str5,str1);
strcpy(str2,str3);
```

Le code précédent afficherait :
Il est morne
Il est taciturne
Erreur de segmentation (core dumped)

Les deux dernières lignes ne peuvent pas fonctionner car les chaînes de destination ne sont pas allouées du tout ou avec un espace trop faible.

Autre exemple :

```
int main(void)
{
	char str1[]="Il preside aux choses du temps";
	char str2[]="Il porte un joli nom : Saturne";
	char * str3;
	int l,c1,c2;
	l=strlen(str1);
	printf("%d\n",l);
	str3=strstr(str1,"choses");
	printf("%s\n",str3);
	c1=strcmp(str1,str2);
	str1[3]= '\0';
	str2[3]= '\0';
	c2=strcmp(str1,str2);
	printf("%d %d\n",c1,c2);
} OK
```

L'exécution de ce programme afficherait :
>
30
choses du temps
3 0

De même que les fonctions printf et scanf permettent d'écrire sur la sortie standard et de lire sur l'entrée standard, les fonctions fscanf et fprintf permettent de lire et écrire dans un fichier.
Toujours de la même manière, les fonctions sscanf et sprintf permettent de lire et écrire dans une chaîne de caractères.
Un pointeur vers la chaîne en question est alors donné en premier argument :

```
int main(void)
{
	char st[256];
	int h=4, m=45, s=56;
	int i ;
	sprintf(st, "%02d HEURES %02d MINUTES %02d SECONDES", h, m, s);
		for (i = 0; i < 33 ; i++)
		printf("%c \n", st[i]);
}
```

Après exécution, la chaîne st contiendra : 04 HEURES 45 MINUTES 56 SECONDES

11.3 Pointeurs et chaînes

Revenons sur cette déclaration de chaîne :

char * chaine = "Programmer en C";
La représentation en mémoire de la chaîne est créée (16 octets), puis on affecte au pointeur chaine l'adresse de début de cette zone.

scanf et chaînes

Lors d'une saisie de chaîne avec scanf (format %s), le & n'est pas nécessaire puisque la chaîne est *déjà* désignée par une adresse (un char * ou un tableau de char)...
Il est courant de construire un tableau de chaînes de caractères sur le même modèle :

```
char * magie[]={"am", "stram", "gram"};
```

Dans le cas qui précède, magie est un tableau de 3 éléments. Chacun de ces éléments est un char* qui pointe vers le début d'une chaîne de caractères. Ces chaînes occupent respectivement 3, 6 et 5 octets en mémoire, et elles ne sont pas nécessairement contiguës.

11.4 Conversions

Les fonctions sprintf et sscanf, évoquées plus haut permettent de réaliser des conversions entre valeurs numériques et chaînes. Signalons aussi les fonctions : atoi (ascii to **int**), atof (ascii to **float**).

12 Retour sur les fonctions

12.1 Passage par valeur et par adresse

En C, les variables sont passées pas valeur.
Si l'on n'utilise pas de pointeur, et que le paramètre n'est pas un tableau, c'est une copie de la valeur qui parvient à la fonction (et modifier cette copie dans la fonction ne modifie par la variable d'origine).

```
#include <stdio.h>
	void fonc(int a)
	{
		printf("A(FONCTION) RECU : %d\n",a);
		a=a+1;
		printf("A(FONCTION) MODIFIE : %d\n",a);
	}
int main(void)
{
	int a=4;
	printf("A(MAIN) AVANT L'APPEL : %d\n",a);
	fonc(a);
	printf("A(MAIN) APRES L'APPEL : %d\n",a);
	return 0;
} OK
```

Exercice
Que va afficher le code précédent ?

Lorsqu'on passe un tableau à une fonction, c'est un pointeur qui est donné en paramètre, et c'est donc l'adresse du tableau, et non son contenu, qui est communiqué à la fonction. Modifier le tableau dans la fonction revient donc à modifier le tableau original.
Si un tableau est passé en paramètre, c'est bien le tableau d'origine qui est manipulé dans la fonction. Modifier ce tableau dans la fonction revient à modifier le tableau d'origine.

```
#include <stdio.h>
	void affiche(int t[])
	{
		int i;
		for(i=0;i<5;i++)
			printf("%d ",t[i]);
		printf("\n");
	}

	void fonc(int t[])
	{
		int i;
		printf("T(FONCTION) RECU : ");
		affiche(t);
		for(i=0;i<5;i++) t[i]=t[i]*t[i];
			printf("T(FONCTION) MODIFIE : ");
		affiche(t);
	}
int main(void)
{
	int t[]={1,2,3,4,5};
	printf("T(MAIN) AVANT L'APPEL : ");
	affiche(t);
	fonc(t);
	printf("T(MAIN) AVANT L'APPEL : ");
	affiche(t);
	return 0;
}
```

Exercice
Que va afficher le code précédent ?

Une variable qui n'est pas de type tableau peut tout de même être passée par adresse. Pour cela on utilise explicitement les pointeurs pour communiquer à la fonction l'adresse de la variable plutôt que son contenu.

```
#include <stdio.h>
	void fonc(int *pa, int a)
	{
		printf("*PA(FONCTION), A(FONCTION) : %d %d\n", *pa, a);
		a = a*2;
		*pa = *pa*3;
		printf("*PA(FONCTION), A(FONCTION) MODIFIE : %d %d\n",*pa, a);
	}
int main(void)
{
	int a = 5;
	printf("A(MAIN) AVANT L'APPEL : %d\n",a);
	fonc(&a,a);
	printf("A(MAIN) APRES L'APPEL : %d\n",a);
	return 0;
}
```

Exercice
Que va afficher le code précédent ?

12.2 Passage des tableaux en paramètres (reloaded)

Lorsqu'on passe un tableau en paramètre d'une fonction, on souhaite pouvoir utiliser le formalisme des tableaux (les crochets []) *dans* la fonction.
Dans le cas des tableaux à une seule dimension, cela ne pose pas de problème (on rajoute souvent la taille en paramètre).
Dans le cas des tableaux à plusieurs dimensions, il faut préciser, par une constante, la taille selon toutes les dimensions, sauf éventuellement la première :

```
#define N1 3
#define N2 4
#define N3 5
	void fonc(int t[][N2][N3])
	{
		.... t[i][j][k]...
	}
int main(void)
{
	int t[N1][N2][N3]={...};
	fonc(t);
	...
}
```

12.3 Pointeurs vers des fonctions

De même qu'il existe des pointeurs vers des données, nous pouvons utiliser et manipuler des pointeurs vers des fonctions. Ceci peut être utile, par exemple, pour l'écriture de code générique, ou en programmation événementielle (enregistrement des *callbacks*).
Voici comment déclarer un pointeur vers une fonction :

```
type_retour (* nom_var) (signature);
```

Dans l'exemple ci-dessous, pfonc sera un pointeur vers une fonction prenant en argument deux entiers, en retournant un entier :

```
int (*pfonc) (int, int);
```

Voici un exemple d'utilisation (sans intérêt, mais qui illustre le principe) :

```
#include <stdio.h>
        int mul(int a, int b) {return a*b;}
        int add(int a, int b) {return a+b;}

 int main(void)
{
        int a = 5, b = 6, c , d;
        int (*pfonc) (int, int);
        pfonc = mul;
        c = (*pfonc)(a, b);
        pfonc = add;
        d = (*pfonc)(a,b);
        printf("%d %d\n", c, d);
        return 0;
} OK
```

Exercice
Qu'affichera l'exécution du programme précédent ?

13 Stuctures

Les structures permettent de créer un nouveau type de variable dont la valeur est un *agrégat* de plusieurs éléments, nommés, de types éventuellement différents.
Si par exemple nous voulons créer un type fiche capable de stocker l'identité, l'âge et la taille de quelqu'un, nous pouvons écrire :

```
struct  fiche
        {
                char nom[255];
                int age;
                float taille;
        };
```

Puis nous déclarons une variable ayant ce type et la renseignons :

```
struct enreg fiche1;
strcpy(fiche1.nom,"Dupond");
fiche1.age = 35;
fiche1.taille = 1.72;
```

Les champs de la structure sont utilisables de la même façon que des variables ordinaires.

Affectation de structures

Lors d'une affectation de type :
fiche2=fiche1;
tous les champs sont recopiés. En fait, c'est toute la zone mémoire contenue dans la structure qui est recopiée, même si celle-ci contient des chaînes de caractères ou des tableaux (qui d'habitude ne sont pas recopiés par une simple affectation).

```
#include <stdio.h>

struct  fiche
        {
                char nom[255];
                int age;
                float taille;
        };


 int main(void)
{
        struct fiche fiche1;
        strcpy(fiche1.nom,"Dupond");
        fiche1.age = 35;
        fiche1.taille = 1.72;
        printf("%d %f \n", fiche1.age, fiche1.taille);
        return 0;

}
```

Utilisation de typedef

La répétition du mot clé struct lors de la déclaration des variable (ou des prototypes des fonctions) est fastidieuse.
Aussi, on utilise souvent conjointement avec les structures la possibilité, en C, de donner de nouveaux noms aux types.
Par exemple, typedef **int** entier; permettra d'utiliser le mot clé **entier** à la place de **int**.
De même :

```
struct  fiche
        {
                char nom[255];
                int age;
                float taille;
        };
```

```
typedef struct fiche fiche;
```

ou encore :

```
typedef         struct
                {
                        char nom[255];
                        int age;
                        float taille;
                }fiche ;
```

permettra d'utiliser le mot **fiche** à la place de **struct fiche**.

```
#include <stdio.h>

typedef         struct
                {
                        char nom[255];
                        int age;
                        float taille;
                }fiche ;


 int main(void)
{
        fiche fiche1;
        strcpy(fiche1.nom,"Dupond");
        fiche1.age = 35;
        fiche1.taille = 1.72;
        printf("%d %f \n", fiche1.age, fiche1.taille);
        return 0;

}
```

Pointeurs et structures

Nous pouvons bien entendu définir un pointeur vers une structure :

```
struct fiche * pf;
```

L'accès aux champs peut alors se faire ainsi :

```
(*pf).age=25;
```

ou ainsi :

```
pf->taille=1.69;
```

14 Énumérations

Les énumérations permettent de définir un nouveau type discret, contenant des valeurs telles que des couleurs, des états,...
L'exemple le plus courant est sans doute :

```
// Définitiion du type enum couleur
enum couleur {rouge,orange,jaune,vert,bleu,violet,indigo};
// Utilisation de variables de ce type
enum couleur c1,c2;
c1 = orange;
c2 = vert;
```

La réalité est que le compilateur associe un entier, en partant de 0, à chaque valeur possible pour l'énumération.
L'utilisateur manipule de son côté des symboles plus évocateurs que des nombres Il est toutefois possible de contrôler les entiers utilisés :

```
enum jour {lundi=1,mardi,mercredi,jeudi,vendredi,samedi,dimanche};
```

Dans le cas qui précède, lundi sera associé à 1 (plutôt que 0 par défaut), et les autres valeurs suivront. Chaque valeur peut néanmoins être spécifiée :

```
enum etat {sain=0, feu=2, cendre=4};
```

15 Champs de bits

Dans certains cas, il est nécessaire de pouvoir contrôler le nombre de bits exacts sur lesquels une information est stockée (plutôt que de s'accommoder des 8 bits d'un unsigned char par exemple). La déclaration d'un champ de bits ressemble à celle d'une structure. Pour chaque membre de la structure, qui peut être signé ou non signé, on indique combien de bits il faudra utiliser. Ainsi, si on veut diviser un mot de 16 bits en 2 groupes de 6 bits, et un groupe de 4, on peut écrire :

```
struct state {
        int grp1 : 6;
        unsigned int grp2 : 6;
        int val : 4;
};
```

Les trois champs se nommeront alors grp1, grp2 et val. Enfin, grp1 pourra prendre des valeurs entre -32 et +31, grp2 entre 0 et 63, et val entre 0 et 15.
On accède aux données comme on le ferait avec une structure :

```
struct state var;
var.grp2=50;
...
```

16 Unions

Les unions permettent de créer un type de données qui pourra être interprété de différentes manières, c'est-à-dire décodé selon les principes d'un certain type ou d'un autre.
Voici par exemple un type de donné, nommé tabchar qui peut être interprété comme un entier codé sur 4 octets (**int**) ou bien comme un tableau de 4 octets :

```
union  tabchar{
                        int val;
                        unsigned char tab[4];
        };
```

On peut ensuite interpréter la donnée de ce type comme un entier ou comme un tableau de char :

```
union  tabchar v;
int i;
v.val=65538;
for (i=0;i<4;i++) {
printf("%d ",v.tab[i]);
}
printf("%d\n",v.val);
```

Le programme affichera : 2 0 1 0 65538

Exercice
Expliquez l'affichage précédent

17 Allocation dynamique de mémoire

L'allocation dynamique de mémoire est généralement nécessaire lors de l'utilisation de structures dont la taille n'est pas connue à la compilation : tableau de taille *a priori* inconnue (par exemple lecture en mémoire de l'intégralité d'un fichier dont on ne connaît pas la taille), utilisation de données de type listes, arbres, graphes...
La fonction d'allocation malloc permet d'allouer lors de l'exécution (et non pas lors de la compilation) un nombre donné d'octets. La fonction renvoie un pointeur (de type void *) vers la zone allouée. Il est habituel de stocker la valeur renvoyée dans un pointeur typé (par exemple **unsigned int** * si on sait que les données seront des entiers non signés), pour pouvoir utiliser l'arithmétique sur les pointeurs (ajouter 1 au pointeur passera bien à l'entier non signé suivant et pas à l'octet suivant).
La mémoire allouée peut être libérée par la fonction free à laquelle on communique un pointeur vers le début de la zone allouée.
La fonction realloc prend en paramètre un pointeur vers une zone déjà allouée, une taille en octets, et renvoie un pointeur vers la zone agrandie (qui peut être la même zone, réellement agrandie, ou une nouvelle zone plus grande dans laquelle les données d'origines auront été copiées).
L'exemple suivant alloue une zone mémoire par blocs de 512 octets, et y stocke l'intégralité d'un fichier texte.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
	unsigned char * txt;
	size_t taillealloc = 512;
	size_t taillereelle = 0;
	FILE *f;
	int c;
	txt = malloc(taillealloc*sizeof(char));
	if (txt == NULL) return -1;
	f = fopen("data.txt","r");
	while( (c = fgetc(f)) != EOF )
	{
		if (taillereelle >= taillealloc)
		{
			taillealloc += 512;
			txt = realloc(txt, taillealloc*sizeof(char));
			if (txt == NULL) {fclose(f); return -1;}
		}
		txt[taillereelle] = c;
		taillereelle++;
		//printf("%c",c);
	}
	fclose(f);
	printf("Taille alloc : %ld\n", taillealloc);
	printf("Taille réelle : %ld\n", taillereelle);
	free(txt);
	return 0;
}
```

Chapitre V
Exercices

Exercice 1 : Nombres entiers
Écrivez en base 2, 10 et 16 les nombres suivantes, exprimés en base 2, 10 ou 16 :
127|10, 15|10, 170|10, 10|10, 10011010|2, 1001|2, 10|2, *E*|16, *B*6|16, *FF*|16, 10|16

Exercice 2 : Complément à 2
Écrivez en codage en complément à 2 sur 8 chiffres, les nombres suivants : -15, 127, -1

Exercice 3 : Virgule fixe
Écrivez en binaire les nombres suivants : 6.25 3.3125 1.2

Exercice 4 : Limites virgule flottante
En simple précision, calculez (dans les nombres psitifs) le plus petit et le plus grand nombre dénormalisé, ainsi que le plus petit et le plus grand nombre normalisé.

Exercice 5 : Répétition virgule flottante
Supposons qu'on utilise un codage en virgule flottante sur 5 bits avec 1 bit de signe et 2 bits d'exposant (le décalage d'exposant vaut alors 1). Écrivez tous les codes possibles et le nombre qu'ils représentent sur l'axe réel. Effectuez les opération suivantes en virgule flottante : (3+0.25)+0.25 et 3+(0.25+0.25). Quelle règle utiliser pour choisir l'ordre des additions ?

Exercice 6 : Commandes de Leds
Nous écrivons un programme en C qui permet de piloter une série de 16 leds, par le biais d'une variable nommée OUTLEDS. Chacune des 16 leds correspond à un bit de la variable. La led 0 au bit 0 (bit de poids faible) etc... Si le bit est à 1, la led est allumée. Écrivez les instructions qui permettent de réaliser les opérations suivantes :
1. éteindre toute les leds ;
2. allumer toutes les leds ;
3. allumer la led 4 et éteindre les autres ;
4. allumer la led 5 sans toucher aux autres ;
5. éteindre la led 5 sans touche aux autres ;
6. allumer les leds de rang pair (0,2,4,..) sans toucher aux autres ;
7. provoquer le clignotement : leds paires/leds impairess
8. provoquer le clignotement des leds 0 à 7, indépendamment de leur état de départ, sans modifier l'état des autres leds.

<u>Sur les entrées/sorties</u>

Exercice 7 : Cercle
Écrivez un programme qui demande à l'utilisateur le rayon d'un cercle, puis affiche son périmètre et sa surface, avec deux chiffres après la virgule.

Sur les instructions et les structures de contrôle

Exercice 8 : Extraction de chiffres
Écrivez un programme qui demande un nombre entier à l'utilisateur, puis affiche successivement ses chiffres et en fait la somme.

Exercice 9 : Le plus grand des 3
Écrivez un programme qui demande à l'utilisateur d'entrer trois nombres à virgule, puis affiche le plus grand des trois.

Exercice 10 : Degrés
Écrivez un programme qui affiche un table de conversion de degrés Celcius (toutes les valeurs de -200 à 200 par pas de 20) vers dégrés Farhenheit et Kelvin. La table devra être correctement formatée et les nombres alignés.
$TK = TC + 273.15$ et $TF = 95$
$TC + 32$

Exercice 11 : Encadrement
Écrivez une fonction qui encadre au plus juste un nombre choisi par l'utilisateur entre des puissances successives d'un nombre de son choix.
Exemple :
Entrez un nombre : 345
Entrez un (petit) entier : 3
345 et compris entre 3^5(243) et 3^6(729)

Sur les fichiers

Exercice 12 : Somme et produit
Écrivez un programme qui ouvre un fichier contenant des nombres, les lit, et affiche leur somme et leur produit à l'écran.

Sur les tableaux

Exercice 13 : Tableaux
Écrivez un programme qui demande des nombres à l'utilisateur (50 maximum) et les stocke dans un tableau, puis affiche le maximum, le minimum et la somme alternée de ces nombres (le premier moins le second, plus le troisième moins le quatrième etc..).

Exercice 14 : Conversion en binaire
Écrivez une fonction qui prend en paramètres un nombre et un tableau et place dans le tableau les chiffres binaires représentant le nombre. Le bit de poids faible sera placé au début du tableau (case 0).

Exercice 15 : Insertion
Écrivez une fonction qui prend en paramètres un tableau, sa taille, un nombre *n* et un indice *i* et place le nombre *n* en position *i* dans le tableau. Toutes les cases situées après *i* seront décalées et la dernière case sera supprimée.

Exercice 16 : Histogramme
Écrivez un programme qui lit un fichier contenant des relevés de taille de personnes. Produisez un tableau de nombre
indiquant en case *k* combien de personnes mesurent entre $k \times 5$ et $(k + 1) \times 5$ centimètres.

Exercice 17 : Calcul de moyenne
Complétez le programme suivant pour qu'il calcule la moyenne des éléments sur le tableau :

```
#include <stdio.h>
int main(void)
{
	short vals[]={9876,8723,6918,3542,9767,5687, 5485,9847,2934,1029};
	short m;
	short i;
	for (i=0;i<10;i++) {
		...
	}
	...
	printf("La moyenne est : %d\n",m);
	return 0;
}
```

Exercice 18 : Carré démoniaque
Écrivez une procédure qui prend un mot (par exemple PROGRAMME) en paramètre et l'affiche sous forme d'un carré :

PROGRAMME
ROGRAMMEP
OGRAMMEPR
GRAMMEPRO
RAMMEPROG
AMMEPROGR
MMEPROGRA
MEPROGRAM
EPROGRAMM

Exercice 19 : Mots
Un fichier contient des mots de moins de 8 lettres. Écrivez un programme pour stocker ces mots en mémoire centrale, qui recherchera ensuite les mots correspondant à un motif particulier. Les motifs seront donnés sous forme d'une chaîne contenant des lettres ou des points. Les points pourront représenter n'importe quelle lettre. Par exemple, le motif .vi.. correspond à avion et ovins, mais pas à avions, ovnis ou avant.

Exercice 20 : Chiffre de César
Écrivez un programme qui demande une phrase à l'utilisateur, convertit les minuscules en majuscules, puis affiche chaque caractère décalé de 3 rangs dans l'alphabet.
Par exemple, si l'utilisateur entre :
Enfants, voici des boeufs qui passent, cachez vos rouges tabliers.
le programme devra répondre :
HQIDQWV, YRLFL GHV ERHXIV TXL SDVVHQW, FDFKHC YRV URXJHV WDEOLHUV.

Exercice 21 : Échange
Écrivez une fonction qui échange le contenu de deux variables (dont l'adresse sera passée en paramètres).

Exercice 22 : Retours multiples avec les pointeurs
Écrivez une fonction qui prend un tableau d'entiers en paramètres, et renvoie les valeurs min et max de ce tableau
(puisqu'on ne peut pas (simplement) retourner deux entiers en C, on utilisera les pointeurs).

Exercice 23 : Analyse fréquentielle
Écrivez une fonction qui comptabilise le nombre d'apparitions de la lettre e dans une chaîne de caractères.
Écrivez une fonction qui prend en paramètres une chaîne et un tableau de 256 cases et stocke dans la case *i* du tableau le nombre d'apparitions dans la chaîne du caractère de code *i*.

Exercice 24 : Chiffre de Vigénère
Le chiffre de Vigénère permet de chiffrer des caractères avec un mot de passe (voir les détails sur Ars-cryptographica ou Wikipedia, ou écouter en td... :)).
Par exemple, la phrase :
Il porte un joli nom : Saturne. Mais c'est un Dieu, fort inquiétant.
chiffrée avec le mot clé *Georges* donnera :
OPDFX XWARX FRMFU QGRZY JTIAR OWUKW HLTHA KYTFX XATUI ZZEFZ
Écrivez des fonctions pour chiffrer et déchiffrer des messages utilisant cette méthode.

Exercice 25 : Produit matriciel
Écrivez une fonction qui calcule le produit de deux matrices.

Exercice 26 : Fonction Reduce
On souhaite simuler la fonction **reduce** de Python. Pour cela, on voudrait pouvoir écrire :

```
#include <stdio.h>
        int mul(int a, int b) {return a*b;}
        int add(int a, int b) {return a+b;}
        int reduce(...)

int main(void)
{
        int t[]={1,3,4,6,7};
        int s, p;
        s=reduce(t,5,add);
        p=reduce(t,5,mul);
        printf("Somme : %d \nProduit : %d\n",s,p);
        return 0;
}
```

Le programme précédent affichera :
Somme : 21
Produit : 504
Écrivez la fonction **reduce**

Exercice 27 : Nombres rationnels
Décrivez une structure pour représenter les nombres rationnels sous forme de fractions. Écrivez une fonction qui simplifie au maximum une fraction, qui additionne deux fractions, et qui multiplie deux fractions.

Exercice 28 : Chargement en mémoire centrale
Un fichier de données contient en premier enregistrement le nombre de valeurs (**float**) du fichier. Écrivez un programme
qui alloue la mémoire nécessaire et stocke les données en mémoire centrale.

Exercice 29 : Allocation dynamique
Écrivez un programme qui ouvre un fichier de données contenant des entiers, alloue une zone mémoire pour les stocker et y stocke les entiers. On fera en sorte que la zone allouée s'agrandisse au fur et à mesure de la nécessité. On écrira ensuite une fonction qui calcule le nombre de valeurs du fichier contenue dans une plage de plus ou moins 10% autour de la moyenne. Enfin, on fera une fonction de désallocation propre.