C

OUTILS

Dans un environnement graphique, on opte pour un **IDE** qui compile et débogue (*Code::Blocks, Netbeans, Qt…*)

En ligne de commande, il nous faut un **éditeur de texte** (Vim, nano, emacs…), un **compilateur** (gcc, ld, make, *cf paquet build-essential*) et un **Makefile**. Pour le **débogueur**, on peut se tourner vers le programme **gdb** ou **Valgrind**.

Et on n'oublie pas de donner les droits d'exécution au programme généré par la compilation ! (**chmod +x program**)

COMMENTAIRES

On peut définir un commentaire de deux façons :

```c
// Ceci est un commentaire court C++


/* Ceci est un commentaire long
    multilignes C */
```

Il y a les commentaires **courts** qu'on introduira avec un double slash **//**.

Et il y a les commentaires **longs** qu'on encadrera d'un slash et d'une étoile **/* commentaire */**.

MAIN

Tout programme C contient une fonction **main**, située dans un fichier source qu'on appelle par convention **main.c** (*on peut tout à fait l'appeler autrement*). En effet, l'exécution du programme commence au début de **main**, c'est pour cela qu'il en faut une par projet. **main** fait appel à d'autres fonctions pour accomplir sa tâche. La syntaxe d'une fonction **main** est la suivante :

```c
int main()
{
    ...
    return (0);
}

int main(void) /* Norme ansi */
{
    ...
    return (0);
}

/* Ou encore */
int main(int argc, char *argv[])
{
    ...
    return (0);
}
```

La fonction **main** est typée **int** pour la bonne et simple raison qu'elle retourne un **code retour entier** indiquant la bonne réalisation ou non du programme principal, **0** ou la constante **EXIT_SUCCESS**, valant **0** et définie dans la bibliothèque **stdlib.h**, si tout s'est **bien passé** et tout autre **chiffre / nombre** si une **erreur** est survenue.

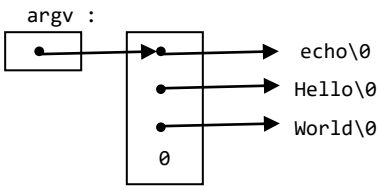
int argc est le **nombre de paramètres** (*arg count*) passés au programme et **char *argv[]** (*arg vector*) un **tableau** de pointeurs sur caractères contenant ces **paramètres**. **argv[0]** contient le path absolu de l'exécutable. *Le fonctionnement de shell est assez similaire*.

Il faut toujours s'assurer que **argc > 1** (donc que **argc** vaut au minimum **2**), sinon cela signifie qu'**aucun paramètre** n'est passé au programme exécuté. Le **1er paramètre** accessible est **argv[1]** et le dernier est **argv[argc - 1]**. On peut aussi utiliser la notation ****argv**.

Comme **argv** est passé en argument de la fonction main, alors il est **converti en pointeur** et on peut le manipuler tel quel *(voir la partie tableau pour avoir l'explication de cette conversion)*.

```c
int main(int argc, char *argv[])
{
    while (--argc > 0)
        printf((argc > 1) "%s " : "%s", *argv++);
    return (0);
}
```

La norme spécifie que **argv[argc]** doit être un **pointeur nul** :



*ⓘ Si le compilateur **gcc** a les drapeaux **-Wall -Werror -Wextra**, alors l'inutilisation de **int argc** et **char *argv[]** entraîne une erreur fatale et l'interruption de la compilation. Donc, utiliser la **première forme** de **main** pour s'éviter de telles erreurs. Tout comme de ne pas déclarer la fonction vide d'arguments (void).*

Chaque fichier C devrait se **terminer** par une **ligne vide** (le compilateur grince, sinon). *(D'où ça sort ça ? OC ?)*

INSTRUCTIONS

Toute **instruction** se **finit** par un **point-virgule**, sinon le compilateur émettra une erreur et refusera de créer l'exécutable.

```c
int mini, max;
max = 100;
printf("Hello World\n");
return (EXIT_SUCCESS);
```

PRÉPROCESSEUR

Une **directive de préprocesseur** commence par le symbole **#**.

#include est l'une d'entre elles. Il existe deux types d'**include** :

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

Et

```c
#include «stdio.h»
```

Avec les chevrons **<...>**, on indique de rechercher le fichier dans la **bibliothèque standard C** et avec les guillemets **«...»**, on indique de rechercher le fichier dans le **répertoire courant** où se situe le **code source .c**, car bien souvent il s'agit d'une bibliothèque définie par l'utilisateur.

Sous Linux, on peut retrouver ces headers dans le répertoire **/usr/include/**.

#define est une **directive de préprocesseur**, donc on la place en début de script, après les **#include**. **#define** nous permet de définir une **constante symbolique** ainsi que sa valeur de remplacement. Par convention, une **constante** est toujours écrite en **lettres capitales**. On peut aussi appeler ces directives **macros**.

```c
#define NOM texte_de_remplacement
#define MAXI 500
#define voir(expr)
printf(#expr" = %g\n", expr);
```

devant un nom de **paramètre** fait que le paramètre se **développe** en une **chaîne** entre guillemets. On peut associer cet effet à une **concaténation** (*p.89 du K&R*).

Le préprocesseur a un **opérateur** **##** qui permet de **concaténer** plusieurs arguments effectifs au moment où la **macro** est développée. Voir l'**annexe A** du **K&R** pour plus d'explications sur les règles régissant l'opérateur (*règles complexes*).

```c
#define coller(debut, fin) debut##fin
coller(nom, 1); /* Crée le lexème nom1 */
```

On peut évaluer des conditions avec les macros. **#if** évalue une **expression entière constante**. On peut l'utiliser pour s'assurer qu'un fichier d'en-tête n'est inclus qu'une fois. **defined** est une **expression** de préprocesseur. **!** représente la **négation**, comme en C.

```
#if !defined(ENTETE)
    #define ENTETE
    /* On place ici le contenu d'entete.h */
#endif
```

Ce type de construction évite d'inclure des fichiers plusieurs fois :

```
#if SYSTEME == SYSV
    #define ENTETE "sysv.h"
#elif SYSTEME == BSD
    #define ENTETE "bsd.h"
#elif SYSTÈME == MSDOS
    #define ENTETE "msdos.h"
#else
    #define ENTETE "default.h"
#endif
#include ENTETE
```

#ifdef et **#ifndef** sont des formes spécialisées de **#if** qui testent si le nom est défini. Le 1er exemple aurait pu s'écrire ainsi :

```
#ifndef ENTETE_H /* Définition de la garde d'inclusion */
#define ENTETE_H
    /* Contenu de entete.h */
    void foo(int bar, int baz);
#endif /* ENTETE_H */
```

LES VARIABLES

Une variable est une information que l'on stocke dans la mémoire vive, la RAM. À chaque **valeur** (*d'une variable*) correspond une **adresse** dans la mémoire (*la position où se situe l'information*). Les **valeurs** sont **converties** en **nombre**, car la RAM ne peut stocker que des nombres dans sa mémoire. C'est pour cela qu'on a des tables de correspondances, pour convertir une lettre en nombre.

Une variable a un nom et une valeur. Le **nom**, c'est la **traduction** en langage humain de son **adresse** dans la **mémoire**, cela évite d'avoir à retenir l'adresse par cœur. Le compilateur se charge ensuite de traduire le nom en adresse et vice versa.

Pour déclarer une variable :

```
type nom_variable;

int compteur;
unsigned int compteur_positif;
```

Les variables se déclarent au début des fonctions. Lors de la **déclaration**, la **nature** d'une variable doit être précisée et aucune place en mémoire ne lui est réservée. C'est lors de la **définition** que l'ordinateur réserve à la **variable** un **emplacement** en **mémoire**. Si on n'**initialise pas** d'emblée une variable, celle-ci a une **valeur indéfinie**, *c'est-à-dire qu'elle prend la valeur qui était présente en mémoire avant sa déclaration*. Si l'adresse mémoire a été vidée correctement ou n'a jamais été utilisée, la valeur sera 0, sinon ce sera la valeur précédente, qui peut être 512 ou 4096 ou 67 encore, n'importe quelle valeur en somme. **On ne peut pas être certain de ce qui se trouve à l'adresse mémoire que prendra notre variable à sa déclaration.** Pour pallier à ce souci, on définit notre variable à sa déclaration. On fait de même concernant une variable **scalaire** *(variable dont le type destine à contenir une valeur atomique, un int ou float est de valeur atomique les tableaux ne sont pas scalaires, on dit qu'ils sont composites. Les atomes sont des petites particules pour imager la définition)*.

```
int compteur = 0;
float eps = 1.0e-5;
```

C'est d'autant plus valable pour les variables dont la valeur ne devra jamais changer. Ces variables sont appelées **constantes**. *Si on ne définit pas leur valeur à la déclaration, on ne*

pourra pas le faire plus loin dans le code. Le compilateur affichera une erreur, si on tente de modifier la valeur d'une constante au cours du programme.

```
const int MAXIMUM = 100;
```

Par convention, le nom des variables **constantes** est écrit en lettres **capitales**. On les introduit avec le mot-clé **const**. Il faut savoir qu'en C, on peut définir des **constantes** de **3 façons** :

En passant par une macro **#define CONSTANTE valeur**. Ce n'est pas particulièrement recommandé, car de cette façon, c'est au préprocesseur qu'on s'adresse.

En passant par une **énumération**. C'est préférable à la macro **#define**, de plus que l'on peut incrémenter automatiquement les constantes d'une énumération. Le **premier nom** d'une **enum** vaut **0**, le **suivant 1** *et ainsi de suite*, à moins que l'on précise des valeurs explicites. On les appelle des **constantes énumérées**.

En passant par le mot-clé **const**.

```
#define MAXIMUM 100 /* On notera que l'instruction ne se termine pas par un point-virgule */

enum mois {JAN = 1, FEV, MAR, AVR, MAI, JUN, JUL, AOU, SEP, OCT, NOV, DEC};
/* FEV vaut 2, Mar vaut 3, etc. L'incrémentation se fait automatiquement */

const int hundred = 100;
```

Lorsqu'on a besoin d'utiliser une même variable dans plusieurs portions du code, on peut déclarer la variable comme « **extern** ».

```
extern int compteur;
extern char tableau[10];
```

La variable **externe** est une première fois déclarée **en dehors** de toute **fonction** (*main* ou autre), dans le **corps principal** du script, **après** les directives de **préprocesseur**. La variable est alors accessible de partout. Il faut également **déclarer** cette fonction dans **chaque fonction** qui l'utilise. On reproduit quasiment la même instruction, pour les tableaux on ne ré-indiquera pas leur taille :

```
extern char tableau[];
```

Puis on définit la variable externe comme on définirait n'importe quelle variable.

Dans un même script, la **double déclaration** n'est pas nécessaire, elle se fait implicitement. *C'est surtout à reproduire lorsque le script possède plusieurs fichiers sources, cela peut être source de bug sinon.* On regroupe, généralement, toutes les **variables externes** et **fonctions** dans un fichier d'**en-tête** (**header**) pour des raisons historiques.

Il n'est pas du tout recommandé d'abuser des variables externes, on peut les modifier par inadvertance ou sans s'en rendre compte.

La **déclaration statique** appliquée à une variable **externe** ou une **fonction** limite sa **portée** (*scope*) à la suite du **fichier source** en cours de compilation. *Cela permet de cacher certains objets qui doivent être externes afin d'être partagés à d'autre sans pour autant être visibles par certains autres*.

```
static int x;
```

On peut appliquer la classe **static** à des variables **automatiques internes** (*Locales* de fonction), elles auront pour différence de **toujours exister après l'appel** de la fonction où elles sont déclarées contrairement aux locales qui cessent d'exister. *C'est un moyen de stocker de façon permanente des données à usage exclusif d'une fonction*.

En l'absence d'une initialisation explicite, les variables **externes** et **statiques** sont initialisées à **0**. L'initialisateur doit être une **expression constante**, l'initialisation ne se fait qu'une fois.

Une déclaration de type **register** prévient le compilateur que la variable sera **employée abondamment**. L'idée est de placer ces variables dans des **registres** de la machine (*une sorte de mémoire*) pour **accélérer les programmes** et **réduire leur taille**. Les compilateurs ne sont pas obligés de tenir compte de ces déclarations. En effet, seulement **quelques variables** peuvent être

placées dans des **registres**, et de **certains types** (**variables automatiques**, internes aux fonctions et **paramètres formels**, paramètres de déclaration de fonction). *Les déclarations surnuméraires ou interdites ne sont pas prises en compte. Les restrictions dépendent de la machine utilisée*.

```
register int y ;

int f(register unsigned m, register long n) {...}
```

Si une variable **register** n'est **pas initialisée**, sa valeur est **indéfinie**. L'initialisation se fait à **chaque fois** que l'on entre dans le **bloc initialisateur**, une expression quelconque peut suffire à l'initialisation (*valeurs déjà définies*, *appels de fonctions*…). C'est valable aussi pour les **variables automatiques**.

Une variable définie à l'intérieur d'une structure de blocs n'existe qu'à l'intérieur de ce bloc et ne sera pas visible à l'extérieur.

```
if(n > 0)
{
    int i;
    for (i = 0; i < n; i++)
        do something
}
/* La variable i n'existe que dans la branche « vraie » du if */
```

LES TYPES

En C, **tout est typé**. Il n'y a pas tant que ça de types différents, dans la mesure où C considère que **tout est nombre**. Pour tout ce qui est caractère, il se sert de la **table ASCII** pour établir les conversions : *à chaque lettre correspond un nombre*. Les types peuvent être **signés** ou **non signés**, c'est-à-dire que les valeurs contenues dans les variables peuvent être précédées d'un signe **-**, ou non. Cela a une incidence sur les différentes valeurs que l'on peut donner à une variable typée.

Types	Description	Taille	Valeur minimum	Valeur maximum
char	caractère	1 octet	-127	127
int	entier	2 ou 4 octets	-32 767	32 767
short	entier	2 octets	-32 767	32 767
long	entier	4 octets	-2 147 483 647	2 147 483 647
float	nombre à virgule, 1 chiffre ap virgule	4 octets	1E-37	1E+37
double	nombre à virgule, 2 chiffres ap virgule	8 octets	1E-37	1E+37
unsigned char	caractère non signé	1 octet	0	255
unsigned int	entier non signé	2 ou 4 octets	0	65 535
unsigned short	entier non signé	4 octets	0	65 535
unsigned long	entier non signé	4 octets	0	4 294 967 295

Les valeurs maximales correspondent au minimum de ce que la norme impose à l'ordinateur de laisser à disposition du programmeur pour un type donné. Il se peut que l'ordinateur offre plus d'espace que ce qui est mentionné dans ce tableau. Pour les types **float** et **double**, la **virgule** est

représentée par un **point**. *Le nombre d'octets de chaque type dépend du compilateur, de ses options et de l'architecture de la machine.*

L'opérateur **sizeof** retourne la taille en **bytes** de son paramètre. Cette taille est de type **size_t**, le type **char** est l'unité de mesure.

Une règle importante, qui rentre en ligne de compte lors des conversions de type implicites :

sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) [<= sizeof(long long)].

Le type **long long** existe dans le **C99**, le compilateur peut émettre une erreur si on lui précise que le **C** est **ANSI**. Ces différents types existent dans le but de **mieux gérer les allocations mémoire**. C'est pour cela qu'il y a plusieurs types d'entiers : ils n'occupent pas tous le même espace mémoire selon leur valeur.

La valeur minimale et maximale des types est définie dans les fichiers d'en-tête **<limits.h>** et **<float.h>**.

Concernant le type **char**, il est préférable de lui attribuer **signed** ou **unsigned** selon l'utilité que l'on veut en faire. *En effet, selon la machine qu'on utilise, le type peut être signé ou non.*

LES OPÉRATEURS ARITHMÉTIQUES

Un ordinateur n'est en fait qu'une calculatrice géante. Les opérations qui lui sont connues sont très basiques : l'addition **+**, la soustraction **-**, la multiplication *****, la division **/** et le modulo **%**. *Le modulo est l'opération consistant à récupérer le reste d'une division euclidienne (de deux entiers), ainsi le modulo n'est pas applicable sur les types float et double.*

Il est possible d'effectuer des calculs entre les **variables**. Il faudra bien faire attention aux **types** des variables, en effet, une opération entre deux types de variables différents aura pour conséquence une **conversion implicite** des données vers le **type** le plus grand. Additionner un *int* avec un *long* convertit l'**opérande int** en **long**. Les conversions s'effectuent aussi lors des **affectations** : la valeur de la **partie droite** de l'affectation est **convertie** dans le **type** de la valeur de la **partie gauche**. *Les float ne sont pas automatiquement convertis en double.*

L'**incrémentation** est l'opération qui consiste à ajouter 1 à une variable. On écrit **variable++**. La **décrémentation** est l'opération contraire, on enlève 1 à une variable : **variable--**. On parle de **post-incrémentation**. La **pré-incrémentation** est l'opération consistant à placer l'**opérateur d'incrémentation avant** son **opérande --variable**. Dans le 1er cas, l'incrémentation se produit **après** (*post*) le traitement de la variable (*après que la variable ait pris sa valeur*), dans le 2nd, l'incrémentation se produit **avant** (*pré*) que *la variable ait pris sa valeur*. Il existe également des raccourcis pour effectuer des opérations sur un opérande tout en lui affectant le résultat :

```
int nombre = 2;

nombre += 4; // nombre + 4 = nombre
nombre -= 3; // nombre - 3 = nombre
nombre *= 5; // nombre * 5 = nombre
nombre /= 3; // nombre / 3 = nombre
nombre %= 3; // nombre % 3 = nombre
```

Pour avoir accès à des opérations plus élaborées, il suffit de charger en mémoire la bibliothèque **<math.h>**. Cette bibliothèque contient les fonctions suivantes : **fabs** (**abs** est accessible depuis la bibliothèque **<stdlib.h>**, **ceil**, **floor**, **pow**, **sqrt**, **sin**, **cos**, **tan**, **asin**, **acos**, **atan**, **exp**, **log** et **log10** entre autres.

Voici le tableau de priorité des opérateurs :

OPÉRATEURS	ASSOCIATIVITÉ
() [] -> . *(-> et . servent à accéder aux membres d'une structure)*	de gauche à droite
! ~ ++ -- +*(unaire)* **-***(unaire)* ******(unaire)* **&***(unaire)* **(type)** *(cast)* **sizeof**	de droite à gauche
******(binaire)* **/ %**	de gauche à droite
+*(binaire)* **-***(binaire)*	de gauche à droite
<< >>	de gauche à droite

< <= > >=	de gauche à droite
== !=	de gauche à droite
&*(binaire, bit)*	de gauche à droite
^	de gauche à droite
|	de gauche à droite
&&	de gauche à droite
||	de gauche à droite
?:*(ternaire)*	de droite à gauche
= += -= *= /= %= &= ^= |= <<= >>=	de droite à gauche
,*(ex : boucle for avec plusieurs expr)*	de gauche à droite

LES OPÉRATEURS BINAIRES

Un nombre **binaire** en C s'écrit sous ces formes :

```
a = 1; /* Décimal */
a = 01; /* Octal */
a = \x1 /* Hexadécimal K&R, ou encore */
a = 0x1 /* Hexadécimal partout ailleurs*/
/* Il n'existe pas de représentation binaire en C, à moins de passer par une structure,
chaque élément étant la position d'un bit dans un octet. Position de 0 à 7 */
```

Soient **a** et **b** deux **entiers** (*signés ou non*) qui valent en **hexadécimal** et en **binaire** :

```
a = 0x03EF = 0000.0011.1110.1111
b = 0x8049 = 1000.0000.0100.1001
```

& ET bit à bit (mise à 0)

```
a & b = 0x0049 = 0000.0000.0100.1001
```

Table de vérité &

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

| OU inclusif bit à bit (mise à 1

```
a | b = 0x83EF = 1000.0011.1110.1111
```

Table de vérité | (OR)

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

^ OU exclusif bit à bit

```
a ^ b = 0x83A6 = 1000.0011.1010.0110
```

Table de vérité ^ (XOR)

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	1

<< décalage à gauche

```
a << 4 /* Décalage à gauche de 4 bits */
       = 0x3EF0 = 0011.1110.1111.0000
```

Les bits décalés à gauche sont perdus et des 0 apparaissent à droite (*cela revient à une multiplication par 4*).

>> décalage à droite

```
a >> 3 /* Décalage à droite de 3 bits */
       = 0x007D = 0000.0000.0111.1101
```

Les bits décalés à droite sont perdus et des 0 (*si bit non signé*) ou des bits de signes comme 1 apparaissent à gauche (**décalage arithmétique**, si **décalage logique** alors les 1 sont remplacés par des 0, cela dépend de la machine utilisée).

~ complément à un (opérateur unaire), NOT bit à bit

```
~a = 0xFC10 = 1111.1100.0001.0000
```

On se sert de **~** pour établir le **masque** d'un nombre binaire, mais pas que. On peut aussi créer un masque par décalage de bits s'il ne concerne qu'un seul bit.

```
1 << 0 = 0000.0000.0000.0001
1 << 5 = 0000.0000.0010.0000

~(1 << 0) = 1111.1111.1111.1110
~(1 << 5) = 1111.1111.1101.1111
```

LES CARACTÈRES D'ÉCHAPPEMENT

\a	produit une alerte sonore
\b	retour en arrière *(retour d'un caractère)*
\f	saut de page
\n	fin de ligne
\r	retour chariot
\t	tabulation horizontale
\v	tabulation verticale
\\	backslash
\?	point d'interrogation
\'	apostrophe
\''	guillemet
\ooo	nombre octal *(1e lettre = o, les autres sont des chiffres de 0 à 7)*
\xhh	nombre hexadécimal *(1e lettre = x, les autres sont des chiffres de 0 à 9 et des lettres de A à F)*

LES CONDITIONS

Les conditions sont à la base de tous les programmes. C'est un moyen pour l'ordinateur de prendre une décision en fonction de la valeur d'une variable.

```c
if (condition)
{
    // Instructions si condition vraie
    if (condition)
    {
        /* Même s'il n'y a qu'une seule instruction pour ce if, il faut mettre des accolades
        pour dissiper toute ambiguïté  */
    }
}
else if (autre condition)
{
    // Instruction si condition fausse mais autre condition vraie
}
else
{
    // Instruction si toutes les conditions sont fausses
}
```

Lorsqu'il y a une **seule instruction**, on peut **omettre** les accolades **{}**. *On évite de tout placer sur une même ligne pour une question de lisibilité.* Si on **imbrique** une condition **if** dans une autre condition **if**, alors, *et même s'il n'y a qu'une instruction par if*, il faut **mettre** les **accolades**. Sans elle, le **compilateur**, selon sa configuration, émettra soit une erreur avec les flags `-Wall -Wextra -Werror -ansi -pedantic`, ou **associera** la condition **if** imbriquée au **prochain else** qu'il rencontrera dans le bloc initial, *et ce même s'il ne concerne pas le if imbriqué*. Les **accolades dissipent** toute **ambiguïté**.

Il est courant de combiner plusieurs conditions grâce aux **opérateurs de comparaison**.

Symbole	Signification
==	est égal à
>	supérieur à
<	inférieur à
>=	supérieur ou égal à
<=	inférieur ou égal à
!=	n'est pas égal à
!	négation
&&	ET logique
||	OU logique

Les **opérateurs de comparaison** sont **moins prioritaires** que les opérateurs **arithmétiques**. C'est-à-dire qu'une **multiplication** ou une **addition** s'effectuera **avant** n'importe quelle **comparaison**, si elles sont décrites sur la même ligne, sauf si la comparaison est placée entre parenthèses. Les opérateurs `>`, `>=`, `<` et `<=` sont **prioritaires** sur les opérateurs `==` et `!=`. Les opérateurs logiques `&&` (ET LOGIQUE) et `||` (OU LOGIQUE) sont **moins prioritaires** que les **opérateurs** de **comparaison**. `&&` est **prioritaire** sur `||`. Les opérateurs de **comparaison** ont la **priorité** sur les opérateurs d'**affectation**. Ainsi, si on veut qu'une affectation, dans un test, se produise en premier, il faut écrire :

```c
(c = getchar() ) != '\n'
```

Un **opérateur** arithmétique, de comparaison ou logique **vaut 1** si la **relation** est **vraie** et **0** si elle est **fausse**.

Un **booléen** est une variable qui peut avoir deux états :

vrai, équivalent au chiffre **1**;

faux, équivalent au chiffre **0**. Toute valeur différente de **0** est en fait considérée comme vraie. On utilise des int pour stocker des booléens, car ce ne sont rien d'autre que des nombres.

```c
if (1)
{
    printf("C'est vrai");
}
else
{
    printf("C'est faux");
}
```

Lorsque l'on fait un test comme `if (age >= 18)`, l'ordinateur **remplace** la **condition** par **1** si elle est **vérifiée**. C'est comme si l'on écrivait `if (1)`. Si elle est **fausse**, l'ordinateur **remplacera** la **condition** par **0** et lira les instructions données dans la partie **else** du test. C'est comme si, on écrivait `if (0)`, la conséquence étant la lecture du **else**, ou la **sortie** de la condition si pas de partie else. Pour s'en rendre compte, on peut faire :

```c
int majeur = 0;

majeur = age >= 18;
printf("Majeur vaut :%d\n", majeur);
```

La variable `majeur` aura pris la valeur de **1**. Pour obtenir une valeur **0** dans cette variable, on peut faire :

```c
majeur = age == 10;
```

On dit que la variable **majeur** est un **booléen**. *En C, il n'existe pas de type booléen, comme dans d'autres langages. On passe donc par des int pour simuler cet état. Cela permet de s'épargner une longueur de frappe.*

Ainsi :

```c
if (variable == 1)

/* ou */
if (variable != 0)
```

est **équivalent** à

```c
if(variable)
```

On aura tendance à privilégier la dernière syntaxe.

Le `switch` est une **alternative** au `if {...} else if {...} else` quand il s'agit d'analyser la valeur d'une variable. Il permet de rendre un code source plus lisible lorsque l'on a besoin de tester plusieurs cas de figure. Si on utilise beaucoup de **else if**, c'est souvent le **signe** qu'une condition **switch** serait **plus adaptée** pour rendre le code source plus clair.

```c
switch (condition)
{
    case valeur1:
        // Instructions
        break;

    case valeur2:
        // Instructions
        break;

    case valeur3:
        // Instructions
        break;

    ...

    default:
        // Instructions si aucune valeur ne correspond
        break;
}
```

Si l'instruction **break** est omise, l'ordinateur lira les instructions suivantes sans **s'arrêter** et sortir de la condition. Le cas **default** correspond au **else** d'une condition. On utilise souvent le **switch** pour générer un **menu** de **choix**.

Les **ternaires** sont des conditions très **concises**, *un peu comme les conditions SI utilisées dans Excel*, permettant d'affecter rapidement une valeur à une variable en fonction du résultat d'un test. On les utilise avec parcimonie car le code source a tendance à devenir moins lisible avec elles. Le raisonnement est le même que pour une condition **si...else**, sauf que **tout tient sur une ligne** :

```
condition ? valeur si vraie : valeur si faux ;
```

LES BOUCLES

While

```
while (condition ou 1)
{
    ...
}
```

Tant que la **condition** est **évaluée** à **1** ou vrai (*cf chapitre conditions pour explication*), alors la **boucle** est **exécutée**. *En fait c'est tant que la condition ne vaut pas 0 que la boucle est lancée. Toute autre valeur que 0 est considérée comme « vraie » (cf K&R*).

For

```
for (initialisation; définition; incrémentation)
for (fahr = 0; fahr <= 300; fahr = fahr + 20)
{
    ... /* Instructions */
}
```

La boucle **for** est un cas particulier de la boucle **while**. Il est possible d'ajouter **plusieurs expressions** en les séparant par l'opérateur **,**. On n'est pas obligé de se servir de tous les blocs d'expressions, il suffit de les laisser vides en les séparant par **;**. L'incrémentation n'est pas la seule action que l'on peut mener, voir l'exemple.

```
for (i = 0, y = 0; i < n; i++, y++)
for (;;) /* Boucle infinie */
for (i = 0; isdigit(x); x = getchar())
```

Do...while

```
do
{
    instructions
}
while (expression);
```

L'**instruction** s'**exécute**, **puis** l'**expression** est **évaluée**, avec une boucle **do {...} while()**. Avec cette boucle l'instruction est toujours au moins exécutée une fois, contrairement à **while** où l'instruction n'est *exécutée* uniquement *si l'expression ne vaut pas 0*. C'est une forme de boucle qu'on utilise moins couramment.

LES FONCTIONS

En C, la procédure de déclaration d'une fonction est la suivante :

On déclare le **prototype avant** de **définir** et d'**appeler** la fonction. Bien souvent, le **prototype** et la **définition** de la fonction, ne sont pas placés dans le même fichier : on place le **prototype** dans un fichier **header** d'extension **.h**. Le **prototype** est placé en début de script, après les directives de préprocesseur (fichier header ou source). Le **prototype** se **termine** toujours par un **point-virgule**.

```c
/* Prototype de la fonction puiss*/
int puiss(int m, int n);
```

Dans le fichier où est placée la **définition**, le fichier **source**, d'extension **.c**, on **inclut** le **header** avec une directive de préprocesseur en début de script **#include "header.h"**. Puis on **définit** la fonction. Si la définition se passe dans le fichier source main.c, on peut placer sa définition à l'endroit où on le souhaite, avant ou après main. Enfin on **appelle** la fonction dans le **corps** d'une autre **fonction** : soit une *fonction lambda*, soit la *fonction main*.

```c
/* puiss.c */
#include "puiss.h"

/* Déclaration de la fonction */
int puiss(int base, int n)
{
    int i, int p ;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return (p);
}
```

Si la **définition** est **située** à un **autre endroit** que son **appel**, alors il faut **inclure** le **header** contenant le **prototype** de la fonction également là où elle est **appelée**.

```c
/* main.c */
#include <stdio.h>
#include <stdlib.h>
#include "puiss.h"

int main(void)
{
    int i;

    /* Appel de la fonction puiss */
    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, puiss(2, i), puiss(-3, i));
    return (EXIT_SUCCESS);

}
```

Une fonction est toujours **typée** et **retourne une valeur du même type** au programme l'appelant, **sauf** si son type est **void** alors elle ne renvoie rien. Par convention, une valeur de retour égale à **0** signifie que le programme s'est terminé normalement. Typer la fonction ainsi que ses paramètres permet au compilateur de vérifier que la fonction est appelée correctement.

Il y a **erreur** si la **définition** d'une fonction ou l'un de ses **appels** n'est **pas conforme** à son **prototype**.

Il n'est pas nécessaire que les **noms** des **paramètres** soient les **mêmes** du **prototype** à la **définition**. En fait, ils sont **facultatifs** dans le **prototype**, seul le **type** des paramètres est **obligatoire**, mais on s'abstiendra de procéder de la sorte. *En effet, un choix judicieux de noms de paramètre permet de constituer une bonne documentation.* On différencie le **paramètre** de l'**argument**. Un **paramètre** est une variable dont le nom est donné au niveau d'une **définition** de fonction quand l'**argument** est la variable employée lors de l'**appel** de la-dite fonction. Parfois, on parle d'*argument formel* et d'*argument effectif*.

Tous les **arguments** des fonctions se passent **par valeur**. On transmet à la fonction appelée les **valeurs** de ses **arguments** dans une **variable temporaire**, et non dans celles d'origine. Cela signifie que le langage C **ne permet pas** à la fonction appelée de **modifier** directement **une variable** de la fonction appelante, la **modification** ne peut que s'opérer au travers d'une **copie temporaire** propre à la fonction appelée, comme si les *paramètres* étaient des *variables locales*.

```c
/* Déclaration de la fonction puiss utilisant un paramètre comme variable locale sans jamais modifier l'originale */
int puiss(int base, int n)
{
    int p;
```

```
    for (p = 1; n > 0; --n)
        p = p * base;
    return (p);
}
/* Le paramètre n est décrémenté, mais sa valeur originale n'est jamais modifiée, seulement
la copie de sa valeur, cela permet de se passer de la variable i */
```

En revanche, si un **argument** est passé par **adresse**, la valeur de la **variable originale** est **modifiée**.

Pour **modifier** une variable en C, la fonction **appelante** doit fournir l'**adresse mémoire** de la variable à manipuler, autrement dit : un **pointeur sur** cette **variable**. La fonction **appelée** doit déclarer le **paramètre** correspondant comme un **pointeur** ; *elle accèdera indirectement à la variable via ce pointeur*.

Lorsqu'on donne un nom de **tableau** comme **argument**, la valeur transmise est l'**adresse** du **début** du **tableau**, les **éléments** du tableau ne sont **pas copiés**. En **indexant** cette **valeur**, la **fonction** peut **accéder** à tous les **éléments** du **tableau** et les **modifier**.

On utilise beaucoup la **récursivité** dans les fonctions : la fonction s'appelle elle-même au sein de sa déclaration.

```
/* affd : affiche n en décimal */
void affd(int n) {
    if (n < 0){
        putchar('-');
        n = -n;
    }
    if (n / 10)
        affd( n / 10);
}

/* appel de la fonction */
affd(123);
```

Si l'identifiant prédéfini `__func__` est employé au sein d'une fonction, *par exemple avec printf*, alors c'est le nom de la fonction qui est utilisé.

```
void my_function(void)
{
    ...
    printf("%s\n", __func__);
    ...
}

my_function;
/* A chaque appel de la fonction, s'affichera sur la sortie le nom de la fonction soit
« my_function » */
```

Pour la gestion d'une liste variable d'arguments, voir plus bas, au niveau du chapitre concernant la gestion des arguments facultatifs. (*p.29*)

LES TABLEAUX

Il n'existe pas de type **array** comme en PHP. Un tableau se déclare comme une **variable** : on la **type** et on lui donne des **valeurs** pour chaque **indice** du tableau. Pour cela, il faut indiquer une taille limite au tableau, cela permet de lui réserver un espace suffisant en mémoire. Les indices débutent à **0**.

```
int nchiffres[10] ; /* Le tableau possède 10 indices */
nchiffres[5] = 6 /* Définition du 6e indice du tableau */
```

On peut aussi **initialiser** un tableau en faisant suivre sa déclaration d'une **liste d'initialisateurs**. Si la **taille** du tableau n'est **pas précisée**, le **compilateur** la **calcule** en comptant les initialisateurs. *Il faut initialiser le tableau lors de la déclaration pour que cela fonctionne*.

```
int jours[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Si le **nombre d'initialisateurs** < à la **taille** déclarée alors les autres éléments **non initialisés** valent **0** si les éléments sont **externes** ou **statiques** ; la valeur est **indéfinie** s'il s'agit de variables **automatiques**.

S'il y a trop d'initialisateurs, alors une erreur est déclenchée. *Il n'est pas possible de répéter un initialisateur ni d'initialiser un élément du milieu du tableau sans définir les éléments précédents*.

nchiffres donne l'**adresse mémoire** du 1er **indice** du tableau, les adresses suivantes étant consécutives (*cf pointeurs*) à l'**indice 0**. On passe un argument **tableau de char** de cette façon dans une fonction, de sorte à ce que l'indice 0 soit traité en premier, *à moins de vouloir qu'un indice en particulier ne soit pris en compte*.

Les tableaux de caractères :

Une **chaîne de caractères** (*string*) est en fait un tableau contenant lesdits caractères. Pour indiquer qu'une chaîne de caractères est **finie**, on ajoute le caractère **\0** (le **caractère nul**) à la **fin** du **tableau**. *En effet, il ne peut exister de lignes de longueur nulle, chacune contenant au minimum le caractère \n ou EOF, on peut donc se servir du caractère \0 pour signifier cet état de fin de chaîne*. C'est une convention utilisée par le langage C.

```
/* lire une ligne dans s et retourne sa longueur*/
int get_line(char s[], int lim)
{
    int c, i;

    for (i = 0; i < lim - 1 && (c = getchar()) != EOF && c != '\n'; ++i)
        s[i] = c;
    if (c == '\n')
    {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return i;
}

/* copie 'from' vers 'to', suppose que 'to' est assez longue */
void copy(char to[], char from[])
{
    int i;

    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}

/* Les définitions ci-dessous sont équivalentes en tant que paramètres formels d'une
** fonction. En dehors de ce cas de figure, les deux déclarations sont strictement
** différentes ! Un tableau n'est pas un pointeur !
*/
char *s;
char s[];
```

Les tableaux de caractères sont un cas particulier d'initialisation : **soit on donne une chaîne constante, soit on définit entre accolades**.

```
char modele[] = "ous";
char modele[] = {'o', 'u', 's', '\0'};
```

La 1ère version est une abréviation de la 2nde. La 2nde est plus longue que la 1ère du fait du caractère \0 supplémentaire. Il est inclus implicitement dans la 1ère version.

Une **chaîne littérale** comme *« ous »* est considérée comme une constante, il vaut mieux la déclarer telle quelle :

```
const char modele[] = "ous";
```

Les tableaux multidimensionnels

Un tableau **multidimensionnel** est en fait un **tableau** à **une dimension** dont **chaque élément** est un **tableau**. C'est un **tableau de tableaux**.

```
/* Déclaration et définition d'un tableau multidimensionnel */
static char tabjour[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};
```

C'est un genre de tableau que l'on utilise moins souvent que les tableaux de pointeurs. Dans l'exemple ci-dessus, il s'agit d'un tableau stockant le nombre de jours par mois, selon que l'année est bissextile ou non, les indices **lignes** représentent les **années** tandis que les indices **colonnes** représentent les **mois**, d'où la déclaration **[2][13]**. On a ajouté un **0** pour chaque indice **0** de chaque ligne afin de faire correspondre l'indice au numéro du mois.

Les **indices lignes** correspondent aux **indices** du **tableau initial** et les **indices colonnes** sont les **indices** des **tableaux éléments**. Comme on accède **d'abord** aux **indices** du tableau **initial** puis aux **indices** des tableaux **éléments**, on appelle un tel tableau de la façon suivante :

```
tabjour[ligne][colonne];
```

Quand on passe un tableau **multidimensionnel** à une **fonction**, il faut préciser le **nombre** de **colonnes** lors de la **déclaration** du paramètre. Le **nombre** de **lignes** est **sans importance** car on passe un **pointeur** sur un tableau de lignes, tableau dans lequel chaque ligne est un tableau de 13 ints (donc on passe l'adresse de l'élément initial de chaque tableau).

Pour déclarer une telle fonction, on peut écrire indifféremment :

```
f(int tabjour[2][13]) {...}

f(int tabjour[][13]) {...}

f(int (*tabjour)[13]) {...} /* Ici on met des parenthèses car les crochets sont prioritaires sur *, sans parenthèses *tabjour[13] désigne un tableau constitué de 13 pointeurs sur entiers */
```

Seule la première dimension (indice) d'un tableau est libre ; on doit spécifier toutes les autres.

Tableaux et conversion :

*Sauf quand elle est l'opérande de l'opérateur **sizeof**, de l'opérateur **_Alignof** ou de l'opérateur unaire **&**, ou est une **chaine de caractères littérale utilisée pour initialiser un tableau**, une **expression** de type « **tableau de type** » est **convertie** en une **expression** de type « **pointeur de type** » qui pointe sur l'élément initial de l'objet tableau et n'est pas une lvalue. Si le tableau est d'une classe de stockage registre, le comportement est indéterminé. C99 traduit*
http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf

En fait, il n'existe que **quatre cas** où une **expression** de type **tableau** se **comporte** clairement **comme** un **tableau**. Ils sont soulignés dans l'extrait de la norme ci-dessus (**sizeof**, **_Alignof**, **&** et **« chaine littérale constante de caractères » initialisant un array**). *C'est pour cela qu'un tableau passé en argument d'une fonction peut se manipuler comme un pointeur : il est converti en pointeur*.

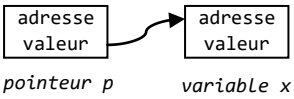
Un **tableau** ne peut **pas** être **affecté**, autrement dit, il ne peut **pas** être la **valeur à gauche** (*left value*) d'une affectation. Et bien sûr, il ne peut **pas** non plus être **incrémenté** ni changé d'une quelconque manière. Pour être tout à fait rigoureux, à la base, un **tableau** est une **lvalue non modifiable**. *Et la conversion qui donne un pointeur sur l'élément initial produit une simple valeur, qui n'est pas une lvalue*. Un tableau ne peut **pas** être **initialisé** avec une valeur **scalaire** (*il ne peut pas être initialisé avec un pointeur par exemple, qui est une valeur scalaire – atomique, indénombrable-*).

LES POINTEURS

En C, les arguments sont passés par **valeur** : c'est une copie de l'argument qui est passé à la fonction et qui manipulé. *Si on modifie l'argument, c'est sa copie qui est impactée et non l'argument lui-même*.

Pour modifier la version **originale** de l'argument, il faut passer cet argument par **adresse** : on a besoin d'accéder à son **adresse mémoire** et c'est là qu'entre en jeu les **pointeurs**.

Un **pointeur** est une **variable** contenant l'**adresse mémoire** d'une **autre** variable et **accédant** à la **valeur** de cette **variable**. On dit que le **pointeur pointe sur** cette **variable**. *Les liens symboliques d'Unix fonctionnent sur le même principe : le lien symbolique pointe sur le nom du fichier qui lui-même pointe sur un inode (contenu)*.



L'opérateur **unaire** **&** donne l'**adresse** d'un **objet**. L'opérateur **unaire** ***** donne **accès** à l'objet **pointé** quand il est appliqué sur un pointeur. * est l'**opérateur** d'indirection ou de déréférence. & ne peut **pas** s'appliquer à des **expressions**, des **constantes** ou des variables de types **register** ; il s'applique **uniquement** aux objets en mémoire : les **variables** et les éléments de **tableau**.

```c
int x = 1, y = 2, z[10];
int *pi; /* pi est un pointeur sur int */

pi = &x; /* pi pointe sur x : la valeur de pi est l'adresse mémoire de x */
y = *pi; /* y vaut la valeur de x, soit 1, on accède à la valeur de x par *pi */
*pi = 0; /* x vaut désormais 0 */
pi = &z[0]; /* *pi pointe désormais sur z[0], la valeur de pi est l'adresse mémoire de z[0]
*/

++pi; /* pré-incrémente pi, incrémente l'adresse mémoire inscrite dans p de 1 unité */
++*pi; /* pré-incrémente l'objet pointé par p */
*pi++; /* post-incrémente pi, l'adresse mémoire contenue dans p */
(*pi)++ /* post-incrémente l'objet pointé par p, les parenthèses sont très importantes dans
ce cas précis */
```

Pointeurs et arguments de fonction

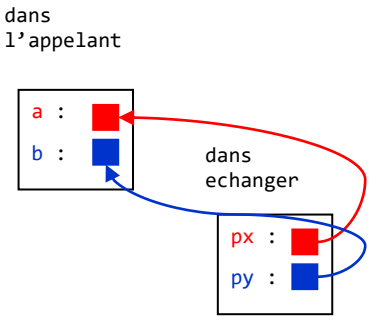
Comme on l'a vu, les **arguments** d'une fonction sont passés par valeur. Pour les **passer** par **adresse**, afin de pouvoir les **modifier**, on passe leurs **adresses**. Étant donné que l'opérateur **&** **donne** l'**adresse** d'une variable, **&a** est un **pointeur sur a**.

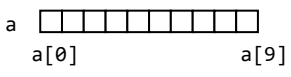
Les **paramètres** sont quant à eux manipulés tels des **pointeurs**, et on accède indirectement aux opérandes par leur intermédiaire.

```c
void echanger(int *px, int *py); /* Déclaration */

void echanger(int *px, int *py) { /* Définition */
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}

echanger(&a, &b); /* Appel */
```

Pointeurs et tableaux

Toute opération que l'on peut effectuer par indexation dans un tableau peut être réalisée à l'aide de pointeurs. La version avec **pointeurs** sera plus **rapide**.





Soit **a** un tableau :

```c
int a[10]; /* tableau de taille 10 -> bloc de 10 objets consécutfs */
int *pa; /* pointeur sur entier */
int x; /* variable x banale */

pa = &a[0]; /* contient l'adresse mémoire de a[0] */
x = *pa; /* contient la valeur de a[0] */
```

Si `pa` pointe sur un élément alors, `pa + 1` pointe sur l'élément suivant, `pa + i` pointe sur le i-ème élément après pa et `pa - i` sur le i-ème élément avant pa. Ils contiennent les adresses mémoire des objets pointés.

Par conséquent, `*(pa + 1)` représente le contenu de `a[1]` ; `pa + i` est l'adresse mémoire de `a[i]` et `*(pa + i)` est le contenu de `a[i]`.

La valeur d'une variable, ou d'une expression, de type tableau est l'adresse de l'élément 0 du tableau. Après l'affectation
`pa = &a[0]`, `pa` et `a` ont la même valeur, donc `pa = a`.

> Si `pa = a`, alors `a = &a[0]`, `pa + i = (a + i) = &a[i] = &pa[i]`.
>
> Par conséquent, `*pa = *a` et `*(pa + i) = *(a + i) = a[i] = pa[i]`.

Un pointeur est une variable, mais le nom d'un tableau, non. Donc `pa = a` et `pa++` sont correctes, mais pas `a = pa` et `a++`.

Quand on passe un nom de tableau en argument à une fonction, c'est l'adresse de son élément initial qui est transmise. Cet argument est une variable locale au sein de cette fonction, donc un nom de tableau en paramètre est un pointeur. C'est de cette propriété dont on se servira au sein des fonctions. En tant que paramètres formels dans une définition de fonction `char s[]` et `char *s` sont strictement équivalents.

```c
/* strlen : return the length of a string */
int strlen(char *s) {
    int n;

    for (n = 0; *s != '\0'; s++)
        n++;
    return (n);
}
```

Une fonction qui retourne un pointeur se déclare ainsi (*p.99 K&R*):

```c
char *allouer(int n); /* Retourne un pointeur sur char */
```

Calculs d'adresse

Les calculs sur pointeurs permettent uniquement, soit d'incrémenter / décrémenter une adresse mémoire soit de comparer les positions de 2 pointeurs pointant sur un même tableau. On peut aussi utiliser dans les calculs l'adresse de l'élément qui suit immédiatement la fin d'un tableau.

Donc, si p et tampalloue pointent sur des éléments d'un même tableau alors `p < tampalloue` compare la position des éléments pointés, compare si l'adresse mémoire de p est inférieure à celle de tampalloue, comme les adresses du tableau sont consécutives. *Le résultat est indéfini si on compare 2 pointeurs pointant sur 2 tableaux différents*.

On sait que `p + n` représente l'adresse du n-ième élément objet après celui pointé par p ; n est mis à l'échelle en fonction de la taille (en octets) des objets pointés, taille déterminée par la déclaration de p. *Si p est un pointeur sur int (4 octets), alors n est multiplié par 4*.

Pour calculer la longueur d'une chaîne, on peut se servir de la différence d'adresse entre deux pointeurs. Mais le nombre de caractères peut être plus grand que celui qu'on peut stocker dans int. Le type `ptrdiff_t` défini dans `<stddef.h>` peut être utilisé pour stocker cette différence, car il est assez grand, mais pour plus de prudence, on préférera le type `size_t`, qui est le type entier non-signé retourné par l'opérateur `sizeof`.

On peut aussi comparer un pointeur à 0, ou à la constante symbolique NULL de `<stdio.h>`. *Toute autre opération sur des pointeurs (addition de 2 pointeurs, multiplication, division, décalage de bits, cf p.101 K&R) est interdite*.

Pointeurs de caractères et fonctions

Une constante de type chaîne est un tableau de caractères dont le dernier indice est '\0' : il occupe en mémoire une unité de plus que le nombre de caractères entre guillemets.

Pour y accéder dans un programme, on utilise un pointeur sur caractères.

```
char *p = "nous partîmes cinq cents"; /* Affecte à p un pointeur sur tableau de caractères, il ne s'agit pas d'une copie, seuls les pointeurs interviennent */
```

Le C ne fournit aucun opérateur pour traiter une chaîne comme un tout. Il y a une différence importante entre :

```
char message[] = "nous partîmes cinq cents";
/* et */
char *p = "nous partîmes cinq cents";
```

message est un tableau de taille suffisante pour recevoir la chaîne + le caractère '\0'. On peut changer les caractères individuellement mais message représentera toujours le même espace mémoire.

p est un pointeur sur char qui pointe sur une constante de type chaîne, on peut le modifier pour qu'il pointe ailleurs mais le résultat est indéterminé si on modifie le contenu de la chaîne.

```
while ((*s++ = *t++) != '\0')
```

La valeur de *t++ est celle du caractère pointé AVANT incrémentation : on affecte à l'objet pointé par s la valeur de l'objet pointé par t puis on avance d'une case mémoire (pour s et t) jusqu'à rencontrer '\0'. Le '\0' est copié. *La comparaison est redondante avec '\0' puisque la question consiste à savoir si l'expression vaut 0.* Or while ne s'exécute que tant que l'expression est différente de 0. On peut donc s'en passer et écrire :

```
while (*s++ = *t++)
```

C'est un concept fréquent en C.

```
/* strcmp : compare s et t ; retourne < 0, 0 ou > 0 selon que s est de façon lexicographique < t, = t ou > t. On obtient cette valeur en soustrayant la valeur des caractères s et t */
int strcmp (char *s, char *t) {
    for (; *s == *t; s++, t++)
        if (*s == '\0')
            return (0);
        return (*s - *t);
}
*--p; /* décrémente p avant d'aller chercher la valeur de l'objet sur lequel pointe p */
*p++ = val; /* met val sur la pile */
val = *--p; /* extrait val du sommet de la pile */
```

Tableaux de pointeurs : les pointeurs de pointeurs

On ne peut pas trier les chaînes comme on trierait un int : il faut plus d'actions pour garder en mémoire les caractères + le symbole `'\0'`. Pour cela, on se sert d'un tableau dans lequel on rangera les lignes les unes après les autres. Si l'on stocke les lignes à ranger les unes à la suite des autres, il est possible d'accéder à chaque ligne à l'aide d'un pointeur sur son premier caractère. On peut alors stocker les pointeurs eux-mêmes dans un tableau. On peut alors comparer deux lignes en passant leurs pointeurs à la fonction `strcmp`, vu ci-dessus.

Pour trier les lignes, c'est encore plus simple : on triera alors les pointeurs en les changeant de place et les lignes ne bougeront pas. *On gagne en temps et en mémoire*. Un tel tableau se déclare de la sorte :

```
char *ptrlg[MAXLIGNES];
```

ptrlg[i] est un pointeur de caractères (il faudra ne pas oublier d'assigner un pointeur à chaque indice) :

```
char *p;
int nlig;

ptrlg[nlig++] = p;
```

*ptrlg[i] est le caractère pointé, le premier de la ligne sauvegardée d'indice i.

Puisque ptrlg est un tableau passé en argument de fonction (p. *107 du K&R*), on peut le traiter comme un pointeur :

```
while (nlig-- > 0)
    printf("%s\n", *ptrlg++);
```

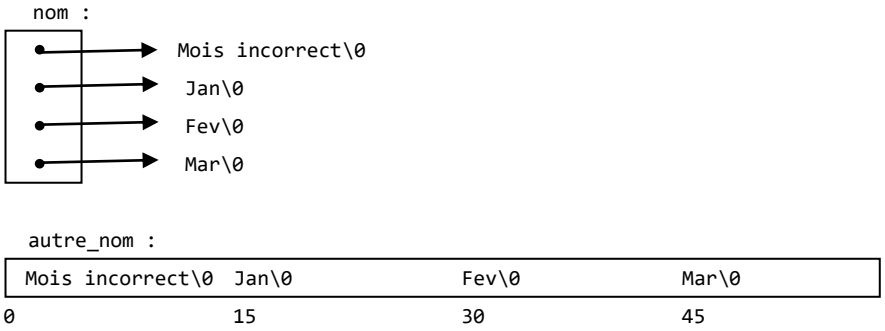
Au départ *ptrlg pointe sur la 1ère ligne, à chaque incrémentation, on passe à la ligne suivante.

- `int *p[]` est un tableau de pointeurs sur int
- `int **p` est un pointeur sur pointeur sur int. La nuance est subtile.
- Tout comme `int (*p)[]` se lit : p est un pointeur sur tableau d'int.

Comparaison pointeurs de tableaux et tableaux multi-dimensionnels

On utilise surtout les tableaux de pointeurs pour stocker les chaîne de char de longueurs diverses.

```
char *nom[] = {"Mois incorrect", "Jan", "Fev", "Mar"};
char *autre_nom[][15] = {"Mois incorrect", "Jan", "Fev", "Mar"};
```



Les pointeurs de fonctions

```
int (*pfonction) (int, int);
```

Un pointeur de fonction contient l'adresse du début du code binaire de la fonction sur laquelle il pointe. C'est utile pour passer une fonction en paramètre d'une autre (*puisque le passage se fait par valeur -copie- dans une fonction, il faut alors la passer par adresse si on veut que le résultat perdure au-delà du scope*).

```
qsort(..., (int (*) (void *, void *)) (numerique ? numcmp : strcmp));
```

(p.117 K&R).

Comme on ne sait pas à l'avance si on n'aura besoin de retourner le résultat de numcmp ou de strcmp, *donc si les arguments seront de type int ou char*, il faut caster le type du pointeur qu'on va utiliser. On caste les paramètres, à la déclaration, à void *, cela permet la surcharge de type et d'éviter que le compilateur beugle : **tout pointeur peut être converti en void * et être reconverti en son type d'origine sans perte d'informations**.

On fait ce cast avec `(int (*) (void *, void *))`, et avec `(numerique ? numcmp : strcmp)` on donne l'adresse mémoire de la fonction à utiliser en fonction de la valeur de la variable numérique : si elle vaut 1 c'est numcmp qui est passée sinon c'est strcmp. Comme ce sont des pointeurs, il n'y a pas besoin de mettre & devant pour obtenir l'adresse. On en déduit que *strcmp et *numcmp sont les fonctions, quand strcmp et numcp sont les adresses.

TECHNIQUE DE LECTURE DU VOCABULAIRE C : CLOCKWISE / SPIRAL RULE

On commence avec l'**identifiant** et on tourne dans le **sens** des **aiguilles** d'une **montre** : on fait une **spirale**. Quand on rencontre les éléments suivants on les **remplace** par les propositions correspondantes :

[x] ou []
Tableau de taille X , ou tableau de taille indéfinie…
Array X size of, or Array undefined size of…

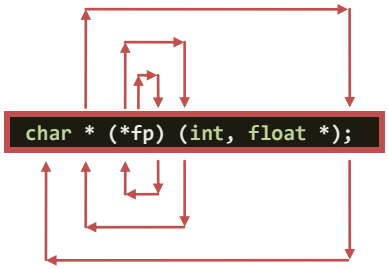
(type 1, type 2)
Fonction passant les arguments de type 1 et type 2 et retournant…
Function passing type 1 and type 2, returning…

*****
Pointeur(s) sur …
Pointer(s) to…

On continue la spirale jusqu'à ce que tous les **jetons** / **tokens** aient été **déchiffrés**.

On résout **toujours** ce qui est en **parenthèses** en **premier**

Exemple :



Qu'est-ce que **fp** ? *What is fp ?*

fp est un **pointeur sur** …
fp is a pointer to…

On est maintenant en dehors de la parenthèse et on tombe sur **(**, donc c'est une **fonction** :

fp est un **pointeur sur** une **fonction passant** pour arguments un **int** et un **pointeur sur float retournant**…
fp is a pointer to a function passing an int and a pointer to float returning…

En continuant la spirale, on tombe sur ***** :

fp est un **pointeur** sur une **fonction passant** pour arguments un **int** et un **pointeur sur float retournant** un **pointeur sur**…
fp is a pointer to a function passing an int and a pointer to float returning a pointer to…

Toujours en spirale, on rencontre ; mais on n'a pas encore lu tous les **tokens**, donc on continue et on tombe sur **char** :

fp est un **pointeur sur** une **fonction passant** pour arguments un **int** et un **pointeur sur float retournant** un **pointeur sur char**.
fp is a pointer to a function passing an int and a pointer to float returning a pointer to char.

LES STRUCTURES DE DONNÉES

Une **structure** rassemble **une** ou **plusieurs variables**, pouvant être de **types différents**. On les **regroupe** sous une seule **étiquette** (*nom de la structure*) pour les **manipuler** plus facilement (*objets, POO*). L'étiquette sert de **notation abrégée** pour représenter la partie de la **déclaration** entre **accolades**. Une **structure** peut être **anonyme** (*sans étiquette*). Les variables des structures sont appelées **membres**. Une déclaration **struct** définit un **type**, l'accolade **fermante** de la structure peut être **suivie** de **variables**, comme n'importe quel type de base. Une **déclaration** de structure **non suivie** d'une liste de **variables** ne réserve **pas d'espace** en **mémoire** (*mais suivie d'une liste, oui*). Elle **décrit** simplement le **modèle** ou la forme d'une **structure**. **Si** la structure

a une étiquette alors on pourra l'utiliser ultérieurement dans des définitions d'objets du type de cette structure.

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

struct point pt = {320, 200}; /* définit une variable pt qui est une struct de type struct point*/
```

On peut initialiser une structure en faisant suivre sa définition par une liste de valeurs initiales, chacune étant une expression constante correspondant à chaque membre. On peut également initialiser une structure automatique par affectation ou appel à une fonction retournant une structure de type approprié.

L'opérateur « . » associe le nom de la structure et le nom du membre. On peut imbriquer les structures.

```
double distance;
distance = sqrt ((double)pt.x * pt.x + (double)pt.y * pt.y);

struct rect {
    struct point pt1;
    struct point p2;
};

/* Si on déclare ecran ainsi : */
struct rect ecran;
/* alors */
ecran.pt.x;
/* désigne la coordonnée x du membre pt1 de ecran */
```

Les opérations autorisées sur structures :

- copie ou affectation en la considérant comme un tout ;
- récupération de son adresse avec l'opérateur & ;
- accès à ses membres.

La copie et l'affectation permettent de passer les structures en arguments, et aussi de les utiliser comme valeurs de retour des fonctions. On ne peut pas comparer des structures.

3 approches possibles pour fonctions manipulant des structures :

- passer en argument les composantes séparément ;
- passer la structure entière ;
- passer un pointeur sur cette structure.

```
/* fabpoint : fabrique 1 point à partir de ses composantes x et  y */
struct point fabpoint(int x, int y)
{
    struct point temp;

    temp.x = x;
    temp.y = y;
    return (temp);
}

struct rect ecran;
struct point milieu;
struct point fabpoint(int, int);

ecran.pt1 = fabpoint(0, 0);
ecran.pt2 = fabpoint(XMAX, YMAX);
milieu = fabpoint((ecran.pt1.x + ecran.pt2.x) / 2, (ecran.pt1.y + ecrant.pt2.y) / 2);
```

L'étape suivant consiste à réaliser un ensemble de fonctions effectuant des calculs sur les points.

```c
/* addpoint : additionne deux points */
struct point addpoint (struct point p1, struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return (p1);
}
```

Ici les arguments et la valeur de retour sont des structures. Au lieu de se servir d'une variable temporaire, on a incrémenté les composantes de `p1`, afin de mettre en évidence le fait que les structures sont passées en **arguments** comme les variables d'autres types.

```c
/* pt_dans_rect : retourne 1 si p est dans r, 0 sinon */
int pt_dans_rect(struct point p, struct rect r)
{
    return (p.x >= r.pt1.x && p.x < r.pt2.x && p.y >= r.pt1.y && p.y < r.pt2.y);
}
```

On suppose `r.pt1 < r.pt2`. La fonction suivante met un rectangle sous cette forme **canonique**

```c
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))

/* canonrect : met les coordonnées d'un rectangle sous forme canonique */
struct rect canonrect(struct rect r)
{
    struct rect temp;

    temp.pt1.x = min(r.pt1.x, r.pt2.x);
    temp.pt1.y = min(r.pt1.y, r.pt2.y);
    temp.pt2.x = max(r.pt1.x, r.pt2.x);
    temp.pt2.y = max (r.pt1.y, r.pt2.y);
    return (temp);
}
```

Si on doit **passer** une **structure** de **taille importante** en argument à une fonction, il est généralement **plus efficace** de lui passer un **pointeur** plutôt que de copier la structure entière :

```c
struct point origine, *pp;

pp = &origine ;
printf("L'origine est (%d, %d)\n", (*pp).x, (*pp.y));
```

- Cette déclaration signifie que `pp` est un **pointeur** sur **structure** de type `struct point`.
  Si `pp` **pointe** sur une **structure point**, `*pp` désigne cette **structure** et `(*pp).x` et `(*pp).y` ses **membres**.
- Les **parenthèses** sont nécessaires dans `(*pp).x` parce que la **priorité** de l'opérateur « . » est **supérieure** à celle de « * ».
- L'expression `*pp.x` est **équivalente** à `*(pp.x)`, ce qui n'est **pas correct** ici, car x n'est **pas** un **pointeur**.
- Les pointeurs de structures sont si fréquemment utilisés qu'il existe une **notation abrégée**.
- Si `p` est un **pointeur** sur une **structure** alors `p->membre` désigne le **membre** en question. Donc l'expression suivante est équivalente à celle-ci-dessus :

```c
printf("L'origine est (%d, %d)\n", pp->x, pp->y);
```

Les opérateurs « `.` » et « `->` » s'évaluent de **gauche** à **droite**, donc les 4 expressions suivantes sont équivalentes :

```c
struct rect r, *pr = &r;
```

```
r.pt1.x;
pr->pt1.x;
(r.pt1).x;
(pr_>pt1).x;
```

Les **opérateurs** sur les structures « **.** » et « **->** » ainsi que les « **()** » des appels de fonctions et les « **[]** » des indices sont les opérateurs les plus **prioritaires** et créent des **liens** très **serrés**. Par exemple :

```
struct {
    int lgr;
    char *ch;
} *p;
```

Alors **++p->lgr** **incrémente lgr**, mais **pas p**, parce que les **parenthèses implicites** se placent ainsi **++(pt->lgr)**.

On utilise des parenthèses pour forcer le niveau de priorité :

- **(++p)->lgr** **incrémente p avant** d'**accéder** à **lgr**
- **(p++)->lgr** l'**incrémente après** (les parenthèses ne sont pas nécessaires dans cette illustration).

De la même façon :

- ***p->ch** permet d'**accéder** à l'**objet pointé** par **ch** quel qu'il soit ;
- ***p->ch++** **post-incrémente ch** ;
- **(*p->ch)++** **incrémente** l'**objet pointé** par **ch** ;
- enfin ***p++->ch** **incrémente p après** avoir accédé à l'**objet pointé** par **ch**.

Les tableaux de structures

Si une situation nécessite que l'on **manipule 2 tableaux** en **parallèle**, alors un **tableau** de **structures** est tout **indiqué**.

```
/* Compte le nb de mots-clés du langage C */
struct cle {
    char *mot;
    int cpt;
} tabcle[NCLES];

/* ou */
struct cle {
    char *mot;
    int cpt;
};
struct cle tabcle[NCLES];
```

tabcle est un **tableau** de clés de type **struct cle** (*la 1ère forme définit et réserve une place en mémoire*). **Chaque élément** du tableau est une **structure**. Puisque **tabcle** contient un ensemble constant de nom, il est plus facile d'en faire une **variable externe** et de l'**initialiser** une bonne fois pour toutes **lors** de sa **définition** :

```
struct cle {
    char *mot;
    int cpt;
} tabcle[] = {
    {"auto", 0},
    {"break", 0},
    {"case", 0},
    /* et plein d'autres ... */
    {"while", 0}
};
```

Les **valeurs** initiales sont présentées par **paires** correspondant aux **membres** de ses structures. On peut également fournir une **structure littérale** plutôt que de l'affecter à une variable

```
struct cle *key = &(struct cle) {.mot = "return", .cpt = 0};
key->cpt++; /* key->cpt = 1 */
```

À l'inverse des chaînes de caractères littérales, les **structures littérales peuvent** parfaitement **être modifiées**. De cette façon, on n'est pas obligé d'initialiser les membres dans l'ordre, ou en totalité, on aurait pu affecter une valeur à **.mot** sans toucher à **.cpt**, qui aurait été mis à une valeur 0 (comme pour les tableaux).

On peut trouver la quantité **NCLES** de la sorte :

```
#define sizeof(tabcle) / sizeof(struct cle) /* Retourne un entier (%ld) égal à la taille en octet */
```

Le type retourné par **sizeof** est **size_t**, type défini dans le header **<stddef.h>**.

```
#define sizeof(tabcle) / sizeof(tabcle[0]) /* Quotient taille du tableau par taille d'une cellule */
```

Cette dernière forme a l'avantage de ne rien avoir à changer si le type du tableau change ; on ne peut **pas utiliser sizeof** dans une directive **#if** parce que le préprocesseur n'analyse pas les noms de types, par contre l'expression dans **#define** n'est pas évaluée par le préprocesseur, donc le **code** est **correct**.

Les pointeurs de structures

On se ressert de la définition de **tabcle** vue ci-dessus :

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXMOT 100

int liremot(char *, int);
struct cle *dicho(char *, struct cle *, int);

/* Compte les mots-clés du C */
int main(void)
{
    char mot[MAXMOT];
    struct cle *p;

    while (liremot(mot, MAXMOT) != EOF)
    {
        if (isalpha(mot[0]))
            if ((p = dicho(mot, tabcle, NCLES)) != NULL)
                  p->cpt++; /* incrémente le compteur */
    for (p = tabcle; p < tabcle + NCLES; p++)
        if (p->cpt > 0)
            printf("%4d %s\n", p->cpt, p->mot);
    return (0);
}

/* dicho : cherche un mot dans tab[0]...tab[n-1] */
struct cle *dicho(char *mot, struct cle *tab, int n)
{
    int cond;
    struct cle *bas = &tab[0];
    struct cle *haut = &tab[n];
    struct cle *milieu;

    while (bas < haut)
    {
        milieu = bas + (haut - bas) / 2;
        if ((cond = strcmp(mot, milieu->mot)) < 0)
            haut = milieu;
        else if (cond > 0)
            bas = milieu + 1;
        else
            return (milieu);
```

```
    }
    return (NULL);
}

/* liremot : lit le mot ou car suivant input */
int liremot(char *mot, int lim)
{
    int c, lirecar(void);
    void remettrecar(int);
    char *m = mot;

    while (isspace(c = lirecar()))
        ; /* Ne rien faire si car d'espacement */
    if (c != EOF)
        *m++ = c;
    if (!isalpha(c))
    {
        *m = '\0';
        return (c);
    }
    for ( ;--lim > 0; m++)
        if (!isalnum(*m = lirecar()))
        {
            remettrecar(*m);
            break;
        }
    *m = '\0';
    return (mot[0]);
}

#define TAILLETAMP 100
char tamp[TAILLETAMP];
int ptamp;

/* lit un car en input */
int lirecar(void)
{
    return ((ptamp > 0) ? tamp[--ptamp] : getchar());
}

/* remet c sur l'input */
void remettrecar(int c)
{
    if (ptamp >= TAILLETAMP)
        printf("remettrecar : trop de caractères.\n");
    else
        tamp[ptamp++] = c;
}
```

1. la déclaration de **dicho** doit indiquer que cette fonction **retourne** un **pointeur** de type **struct cle**. Si **dicho trouve** un mot, elle **retourne** le **pointeur** sur lui **ou NULL**.
2. On **accède** aux **éléments** de **tabcle** via des **pointeurs**.

Les **valeurs initiales** de **bas** et **haut** sont des **pointeurs** sur le **début** et sur l'**endroit** situé juste **après** la **fin** du tableau. Comme il est **interdit** d'**additionner** deux **pointeurs**, pour trouver le **milieu**, on se sert de la **différence** entre **haut** et **bas divisée** par **deux**, qu'on **ajoute** à **bas** (= *addition adresse mémoire du pointeur à un int*).

On doit s'assurer qu'un **pointeur correct** est **généré** et qu'on ne tente pas d'**accéder** en **dehors** du tableau. **&tab[-1]** et **&tab[n]** sont **deux adresses** situées en **dehors**. La 1ère écriture est tout à fait **incorrecte** et on ne peut **pas utiliser** l'**indirection** sur la 2e. Cependant, la **définition** du langage **garantit** qu'un **calcul** sur des **pointeurs** mettant en jeu le 1er **élément APRÈS** la **fin** du **tableau** (**&tab[n]**) fonctionnera **correctement**.

Si **p** est une **structure**, tout **calcul** sur p prend en **compte** la **taille** de cette **structure** ; ainsi **p++ ajoute** à **p** le **nombre** ad hoc pour le faire **pointer** sur l'élément **suivant** du tableau de structures et le test arrête la boucle au moment opportun (**for (p = tabcle; p < tabcle + NCLES; p++)**). Toutefois, il ne faut **pas présumer** que la **taille** d'une **structure** est **égale** à la **somme** des

tailles de ses membres. Du fait des besoins d'alignements (_Alignof C11 ; macrofonction offsetof <stddef.h>) des différents objets, il peut y avoir des « trous » non référencés dans une structure. Si la taille d'un char est 1 octet et celle d'un int de 4 octets, une structure avec deux membres de ces types pourrait bien demander 8 octets et non 5. L'opérateur sizeof retourne la taille correcte (*=> taille alignée*).

Les structures auto-référentielles

Il s'agit de structures faisant référence à elles-mêmes directement ou indirectement (*référence, par opposition au passage par valeur*).

Il est interdit de définir une structure contenant une instance d'elle-même. En revanche, cette structure peut contenir un pointeur sur une instance d'elle-même.

Une structure auto-référentielle est particulièrement adaptée pour la gestion d'un arbre « binaire » (*arbre généalogique, tableau d'un tournoi etc.*). En informatique, un arbre binaire est formé d'éléments appelés nœuds et organisés de façon hiérarchique. L'arborescence a un facteur de branchement égal à 2 (*d'où binaire*). Le nœud au sommet est appelé racine. Les nœuds situés immédiatement sous un nœud sont les fils de ce nœud, eux-mêmes pouvant avoir leurs propres fils. À l'exception de la racine, chaque nœud a exactement un père, qui est le nœud situé juste au-dessus de lui. Une branche est une suite de nœuds partant de la racine et se terminant sur une feuille, qui est un nœud situé sur le dernier rang de l'arbre et qui ne possède aucun fils.

Chaque nœud d'un arbre binaire est formé de 3 parties :

- Un champ de données (*pouvant être composées de plusieurs variables*)
- Deux pointeurs appelés gauche et droite.

```
struct nœud
{
    char *mot;
    int compteur;
    struct nœud *fils_gauche;
    struct nœud *fils_droit;
}
```

Si le nœud n'a aucun ou qu'un seul fils, les pointeurs inutilisés du nœud seront mis à NULL.

Une façon de procéder est de placer les nœuds de telle façon que le sous-arbre gauche d'un nœud ne contienne que des données de valeur (*lexicographique ou numérique*) inférieure à la donnée du nœud ; le sous-arbre de droite ne contient que les valeurs supérieures. On commence depuis la racine (*cf K&R p.137-140 et Algorithmes en C p.178*).

De temps en temps, on a besoin que deux structures se réfèrent mutuellement. Il faudra impérativement déclarer l'une des structures avant sa définition sinon la structure sera inutilisable et le problème restera insoluble. Une déclaration de structure crée un type dit incomplet. Dès lors, il ne peut pas être utilisé pour définir une variable (*puisque les membres qui composent la structure sont inconnus*). Ceci n'est utilisable que pour définir des pointeurs !

```
struct s ;

struct t
{
    ...
    struct s *p; /* pointe sur une struct s */
};

struct s
{
    ...
    struct t *q; /* pointe sur une struct t */
};
```

Énormément de problématiques trouvent solution via les structures de données en C.

Problème concernant les programmes d'allocation mémoire :

Il est souhaitable qu'il n'y ait qu'un seul allocateur de mémoire par programme même s'il alloue des types différents d'objets. Néanmoins, deux questions se posent :

- Comment faire pour satisfaire les exigences des machines concernant les règles d'alignement que doivent respecter les objets de certains types (comme avec les int qui doivent souvent être placés sur des adresses mémoire paires) ?
- Quelles déclarations peuvent prendre en compte le fait qu'un allocateur doit obligatoirement pouvoir retourner différents types de pointeurs ?

On résout facilement les exigences d'alignement, quitte à gaspiller de l'espace mémoire, en s'assurant que l'allocateur retourne toujours un pointeur satisfaisant aux conditions d'alignement. La fonction malloc de la bibliothèque standard, garantit les alignements. En C, la méthode correcte consiste à déclarer que malloc retourne un pointeur de type void, puis de « caster » le type de ce pointeur.

```c
#include <stdlib.h>

/* allouer_noeud : créer un nœud */
struct nœud *allouer_noeud(void)
{
    return ((struct nœud *) malloc(sizeof(struct nœud)));
}
```

malloc retourne NULL s'il n'y a pas d'espace mémoire disponible.

Les unions

Une union est une variable pouvant contenir des objets de tailles différentes, d'où son lien avec les structures. On la déclare de la même façon. La différence, c'est qu'une union ne peut contenir qu'un seul de ses membres à la fois. Les unions permettent de manipuler différents types de données dans un même espace de données. La variable de type union sera suffisamment grande pour contenir le plus grand des types de ses membres.

```c
union etiq_u
{
    int val_i;
    float val_f;
    char *val_ch;
} u;
```

On accède aux membres d'une union par nom-union.membre ou pointeur-union->membre.

```c
struct nombre
{
    unsigned entier : 1; /* champs de bits */
    unsigned flottant : 1;
    union
    {
        int e;
        int f;
    } u;
};
```

Pour accéder aux membres :

```c
struct nombre a = {0};
a.entier = 1;
a.u.e = 10;
```

On peut placer les unions à l'intérieur de structures et vice-versa. Une union est une structure dont tous les membres ont un décalage nul par rapport à la base : cette structure est assez grande pour contenir le membre le plus large et l'alignement est compatible avec tous les types de l'union.

Les opérations autorisées sont les mêmes que pour les structures : affectation ou copie par bloc, récupération adresse mémoire et accès à un membre. On ne peut initialiser une union que par une valeur du type du 1er membre (*lors de la définition*).

Les champs de bits

Un champ de bits est une structure composée exclusivement de champs de type int ou unsigned int dont la taille en bits est précisée. Cette taille ne peut être supérieure à la taille en bits du type int.

On utilise ce système pour optimiser la mémoire que l'on réserve. On peut l'utiliser aussi pour lever et baisser des drapeaux.

Prenons le cas de la gestion d'une date via structure :

```
struct date
{
    unsigned char jour;
    unsigned char mois;
    unsigned short annee;
};
```

À supposer, puisque cela dépend des machines, qu'un type short fasse 2 octets et un type char 1 octet, on mobiliserait 4 octets, ou 32 bits, de la mémoire vive, alors qu'on n'aurait besoin effectivement que de 21 bits (12 bits pour l'année pour gérer jusqu'à 4096 années – $2^{12} = 4096$ -, 4 bits pour le mois -$2^4 = 16$ – et 5 bits pour les jours – $2^5 = 32$ -). En soit, ce n'est un problème que si la structure est créée de nombreuses fois.

On pourrait gérer le cas avec les opérateurs de manipulations de bits, mais on perdrait en temps de calcul. Notre structure serait alors :

```
struct date
{
    unsigned jour : 5;
    unsigned mois : 4;
    unsigned annee : 12;
};
```

Le nombre qui suit « : » est la longueur du champ en BITS. Les champs de bits ne disposent pas d'une adresse mémoire et ne peuvent en conséquence se voir appliquer l'opérateur d'adressage &.

Un champ de bits est un ensemble de bits adjacents à l'intérieur d'une même unité de stockage définie par l'implémentation que nous appelons « mot ». La syntaxe de la définition d'un champ et de son accès s'appuie sur les structures.

Un drapeau correspond à un bit qui est soit levé, soit baissé, dans l'objectif d'indiquer si une situation est vraie ou fausse. Il s'agit d'un bon cas d'utilisation des champs de bits, chacun d'entre eux représentant un drapeau.

```
struct propriete
{
    unsigned int pair : 1;
    unsigned int puissance : 1;
    unsigned int premier : 1;
};

void traitement(int nombre, struct propriete prop)
{
    if (prop.pair) /* si nombre est pair */
        ...
    if (prop.puissance) /* si nombre est puissance de 2 */
        ...
    if (prop.premier) /* si nombre est premier */
        ...
}

int main(void)
{
    int nombre = 2;
    struct propriete prop = {.pair = 1, .puissance = 1};
    traitement(nombre, prop);
    return (0);
}
```

Presque tout ce qui concerne les champs dépend de l'implémentation, c'est elle qui définit si un champ peut chevaucher ou non la limite d'un mot. On utilise des champs sans nom (seulement « : » suivi de la longueur) pour le remplissage. On peut utiliser la largeur spéciale 0 pour former l'alignement sur le début du mot suivant. *Selon les machines, les champs sont affectés de droite à gauche ou de gauche à droite.* **Les programmes qui dépendent de ça ne sont pas portables.**

TYPEDEF

Il s'agit d'une fonctionnalité servant à créer des noms de nouveaux types de données, ou plutôt à créer un alias de ces types.

```
typedef int longueur;
```

longueur devient synonyme de int et on peut l'employer dans toutes les situations interagissant avec les types (déclarations, définitions, cast…). Le type déclaré dans typedef figure à la place d'un nom de variable et non à la place du nom du type. Syntaxiquement, typedef est équivalent aux classes de stockage extern, static…

```
typedef struct s_noeud *t_ptr_arbre;

typedef struct s_noeud
{
    char *mot;
    int cpt;
    t_ptr_arbre gauche;
    t_ptr_arbre droite;
} t_noeud;
```

Deux nouveaux mots-clés sont « créés » : t_ptr_arbre et t_noeud

```
t_ptr_arbre allouer_noeud(void)
{
    return ((t_ptr_arbre) malloc(sizeof t_noeud));
}
```

En fait, une déclaration typedef équivaut à une directive #define, sauf qu'elle est interprétée par le compilateur et permet d'effectuer des substitutions de texte plus puissantes que celles du préprocesseur.

On utilise typedef pour deux raisons (en dehors des raisons esthétiques) :

- Paramétrer un programme pour faire face aux problèmes de portabilité, il suffit alors de modifier les typedefs si les types sont différents sur la machine accueillant le programme. size_t et ptrdiff_t en sont l'illustration ;
- Fournir une meilleure documentation sur un programme.

GESTION DES ARGUMENTS FACULTATIFS

... signifie que le nombre et le type d'arguments peut varier. Ne peut figurer qu'à la fin d'une déclaration :

```
int printf(char *fmt,...);
```

La partie difficile consiste à savoir comment printf parcourt la liste d'arguments alors que cette liste n'a pas de nom. Le header <stdarg.h> à un ensemble de définitions de macros qui indiquent comment parcourir une liste d'arguments.

Le type `va_list` sert à déclarer une variable qui sera associée à chaque argument à tour de rôle. Appelons cette variable pa pour pointeur d'arguments.

La macro `va_start` initialise pa de sorte qu'elle pointe sur le 1er argument non nommé. On l'appelle une seule fois avant pa. Il doit y avoir au moins un argument nommé car va_start se sert du dernier argument nommé pour l'initialisation de pa.

Chaque appel de `va_arg` retourne un argument et fait pointer pa sur le suivant. va_arg a besoin d'un nom de type pour déterminer le type de la valeur de retour et la taille du pas pour pointer sur l'argument suivant.

Enfin `va_end` réalise le nettoyage nécessaire, on doit l'appeler avant que la fonction ne rende la main. (*cf K&R p.153-154*).

L'ALLOCATION DYNAMIQUE

Il n'est pas toujours possible de savoir **quelle quantité de mémoire** sera **utilisée** par un **programme**. Par exemple, si on demande à l'utilisateur de fournir un tableau, on devra lui fixer une **limite**, ce qui pose deux problèmes :

- la **limite** en elle-même, qui ne convient peut-être pas à l'utilisateur ;
- l'**utilisation excessive** de **mémoire** du fait qu'on réserve un tableau d'une taille fixée à l'avance. Si on utilise un **tableau statique**, alors cette quantité de **mémoire superflue** sera **inutilisable** jusqu'à la **fin** de votre **programme**.

Or, un ordinateur ne dispose que d'une quantité limitée de mémoire vive, il est donc important de ne pas en réserver abusivement. L'**allocation dynamique** permet de **réserver** une partie de la **mémoire vive** inutilisée pour stocker des données et de **libérer** cette même partie une fois qu'elle n'est **plus nécessaire**.

La notion d'objet

En C, un **objet** est une **zone mémoire** pouvant **contenir** des **données** et est **composée** d'une **suite contiguë** d'un ou plusieurs **multiplets**. En fait, **tous** les **types** du langage C **manipulent** des **objets**. La **différence** entre les types tient simplement en la manière dont ils **répartissent** les **données** au sein de ces **objets**, ce qui est appelé leur **représentation**. Ainsi, la **valeur 1** n'est **pas représentée** de la **même manière** dans un objet de type **int** que dans un objet de type **double**.

Un objet étant une suite contiguë de multiplets, il est possible d'en examiner le contenu en lisant ses multiplets un à un. Ceci peut se réaliser en C à l'aide de l'**adresse** de l'objet **et** d'un **pointeur** sur **unsigned char**, *le type char du C ayant toujours la taille d'un multiplet. Il est impératif d'utiliser la version non signée du type char afin d'éviter des problèmes de conversions.*

```c
/* Affiche les multiplets composant un objet de type int et un objet de type double en hexadécimal. */

#include <stdio.h>

int main(void)

{

    int n = 1;
    double f = 1.;
    unsigned char *byte = (unsigned char *)&n;

    for (unsigned i = 0; i < sizeof n; ++i)
        printf("%x ", byte[i]);
    printf("\n");
    byte = (unsigned char *)&f;
    for (unsigned i = 0; i < sizeof f; ++i)
        printf("%x ", byte[i]);
    printf("\n");
    return 0;
}

/* Résultat */
1 0 0 0
0 0 0 0 0 0 f0 3f
```

Il se peut que le résultat obtenu ne soit pas le même, car cela dépend de la machine où est exécuté le programme. La **représentation** de la **valeur 1** n'est **pas** du tout la **même** entre le type **int** et le type **double**.

Malloc et consoeurs

La bibliothèque standard fournit **trois fonctions** permettant d'**allouer** de la **mémoire** : **malloc()**, **calloc()** et **realloc()** et **une** vous permettant de la **libérer** : **free()**. Ces quatre fonctions sont déclarées dans l'en-tête **<stdlib.h>**.

malloc

```c
void *malloc(size_t taille);
```

La fonction malloc() permet d'allouer un objet de la taille fournie en argument (*qui représente un nombre de multiplets*) et retourne l'adresse de cet objet sous la forme d'un pointeur générique. En cas d'échec de l'allocation, elle retourne un pointeur nul.

> Il faut toujours vérifier le retour d'une fonction d'allocation afin de s'assurer qu'on manipule bien un pointeur valide.

Allocation d'un objet

Dans l'exemple ci-dessous, on réserve un objet de la taille d'un int, et on y stocke ensuite la valeur « 10 » et on l'affiche. Pour cela, on utilise un pointeur sur int qui va se voir affecter l'adresse de l'objet ainsi alloué et qui va nous permettre de le manipuler comme on le ferait s'il référençait une variable de type int.

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

int main(void)
{
    int *p = malloc(sizeof(int));
    if (p == NULL)
    {
        printf("Échec de l'allocation\n");
        return EXIT_FAILURE;
    }
    *p = 10;
    printf("%d\n", *p);
    return 0;
}

/* Résultat */
10
```

Allocation d'un tableau

Pour allouer un tableau, il faut réserver un bloc mémoire de la taille d'un élément multiplié par le nombre d'éléments composant le tableau. *L'exemple suivant alloue un tableau de dix int, l'initialise et affiche son contenu.* Pour allouer dynamiquement un objet de type T, il faut créer un pointeur sur le type T qui conservera son adresse.

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

int main(void)
{
    int *p = malloc(sizeof(int) * 10);
    if (p == NULL)
    {
        printf("Échec de l'allocation\n");
        return EXIT_FAILURE;
    }
    for (unsigned i = 0; i < 10; ++i)
    {
        p[i] = i * 10;
        printf("p[%u] = %d\n", i, p[i]);
    }
    return 0;
}

/* Résultat */
p[0] = 0
p[1] = 10
p[2] = 20
p[3] = 30
p[4] = 40
p[5] = 50
p[6] = 60
p[7] = 70
```

```
p[8] = 80
p[9] = 90
```

Il est possible de simplifier l'expression fournie à malloc() en écrivant `sizeof(int[10])` qui a le mérite d'être plus concise et plus claire. La fonction malloc() n'effectue aucune initialisation, le contenu du bloc alloué est donc indéterminé.

free

```
void free(void *ptr);
```

La fonction free() libère le bloc précédemment alloué par une fonction d'allocation dont l'adresse est fournie en argument. Dans le cas où un pointeur nul lui est fourni, elle n'effectue aucune opération.

> À chaque appel à une fonction d'allocation doit correspondre un appel à la fonction free().

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

int main(void)
{
    int *p = malloc(sizeof(int));
    if (p == NULL)
    {
        printf("Échec de l'allocation\n");
        return EXIT_FAILURE;
    }
    *p = 10;
    printf("%d\n", *p);
    free(p);
    return 0;
}
```

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

int main(void)
{
    int *p = malloc(sizeof(int[10]));
    if (p == NULL)
    {
        printf("Échec de l'allocation\n");
        return EXIT_FAILURE;
    }
    for (unsigned i = 0; i < 10; ++i)
    {
        p[i] = i * 10;
        printf("p[%u] = %d\n", i, p[i]);
    }
    free(p);
    return 0;
}
```

Même si le deuxième exemple alloue un tableau, il n'y a bien eu qu'une seule allocation. Un seul appel à la fonction free() est donc nécessaire.

calloc

```
void *calloc(size_t nombre, size_t taille);
```

La fonction calloc() attend deux arguments : le nombre d'éléments à allouer et la taille de chacun de ces éléments. Techniquement, elle revient au même que d'appeler malloc() comme suit : `malloc(nombre * taille);` à un détail près : chaque multiplet de l'objet ainsi alloué est initialisé à zéro.

Cette initialisation n'est pas similaire à celle des variables de classe de stockage statique ! En effet, le fait que chaque multiplet ait pour valeur zéro ne signifie pas qu'il s'agit de la représentation du zéro dans le type donné. Si cela est par exemple vrai pour les types entiers, ce n'est pas le cas pour les pointeurs (*un pointeur nul référence une adresse invalide qui dépend de votre système*) ou pour les nombres réels. De manière générale, ne considérer cette initialisation utile que pour les entiers et les chaînes de caractères.

realloc

```
void *realloc(void *p, size_t taille);
```

La fonction realloc() libère un bloc de mémoire précédemment alloué, en réserve un nouveau de la taille demandée et copie le contenu de l'ancien objet dans le nouveau. Dans le cas où la taille demandée est inférieure à celle du bloc d'origine, le contenu de celui-ci sera copié à hauteur de la nouvelle taille. À l'inverse, si la nouvelle taille est supérieure à l'ancienne, l'excédent n'est pas initialisé. La fonction attend deux arguments : l'adresse d'un bloc précédemment alloué à l'aide d'une fonction d'allocation et la taille du nouveau bloc à allouer. Elle retourne l'adresse du nouveau bloc ou un pointeur nul en cas d'erreur.

L'exemple ci-dessous alloue un tableau de dix int et utilise realloc() pour agrandir celui-ci à vingt int.

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

int main(void)
{
    int *p = malloc(sizeof(int[10]));
    if (p == NULL)
    {
        printf("Échec de l'allocation\n");
        return EXIT_FAILURE;
    }
    for (unsigned i = 0; i < 10; ++i)
        p[i] = i * 10;
    int *tmp = realloc(p, sizeof(int[20]));
    if (tmp == NULL)
    {
        free(p);
        printf("Échec de l'allocation\n");
        return EXIT_FAILURE;
    }
    p = tmp;
    for (unsigned i = 10; i < 20; ++i)
        p[i] = i * 10;
    for (unsigned i = 0; i < 20; ++i)
        printf("p[%u] = %d\n", i, p[i]);
    free(p);
    return 0;
}

/* Résultat */
p[0] = 0
p[1] = 10
p[2] = 20
p[3] = 30
p[4] = 40
p[5] = 50
p[6] = 60
p[7] = 70
p[8] = 80
p[9] = 90
p[10] = 100
p[11] = 110
p[12] = 120
p[13] = 130
```

```
p[14] = 140
p[15] = 150
p[16] = 160
p[17] = 170
p[18] = 180
p[19] = 190
```

On a utilisé une autre variable, **tmp**, pour **vérifier** le **retour** de la fonction **realloc()**. En effet, si on avait procédé comme ceci :

```
p = realloc(p, sizeof(int[20]));
```

Il aurait été **impossible** de **libérer** le **bloc** mémoire **référencé** par **p** en cas d'**erreur** puisque celui-ci serait **devenu** un **pointeur nul**. Il est donc **impératif** d'**utiliser** une **seconde variable** afin d'éviter des **fuites** de **mémoire**.

Les tableaux multidimensionnels

L'**allocation** de **tableaux multidimensionnels** est un petit peu **plus complexe** que celles des autres objets. Techniquement, il existe **deux solutions** : l'**allocation** d'**un seul bloc** de mémoire (*comme pour les tableaux simples*) et l'**allocation** de **plusieurs tableaux** eux-mêmes **référencés** par les **éléments** d'un **autre tableau**.

Allocation en un bloc

Comme pour un **tableau simple**, il est **possible** d'**allouer** un **bloc** de mémoire dont la **taille** correspond à la **multiplication** des **longueurs** de **chaque dimension**, elle-même **multipliée** par la **taille** d'**un élément**. Toutefois, cette solution contraint à **effectuer** une partie du **calcul d'adresse** soi-même puisque on alloue en fait un seul tableau. *L'exemple ci-dessous illustre ce qui vient d'être dit en allouant un tableau à deux dimensions de trois fois trois int.*

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

int main(void)
{
    int *p = malloc(sizeof(int[3][3]));

    if (p == NULL)
    {
        printf("Échec de l'allocation\n");
        return EXIT_FAILURE;
    }
    for (unsigned i = 0; i < 3; ++i)
        for (unsigned j = 0; j < 3; ++j)
        {
            p[(i * 3) + j] = (i * 3) + j;
            printf("p[%u][%u] = %d\n", i, j, p[(i * 3) + j]);
        }
    free(p);
    return 0;
}

/* Résultat */
p[0][0] = 0
p[0][1] = 1
p[0][2] = 2
p[1][0] = 3
p[1][1] = 4
p[1][2] = 5
p[2][0] = 6
p[2][1] = 7
p[2][2] = 8
```

Une **partie** du **calcul** d'adresse doit être **effectuée** en **multipliant** le **premier indice** par la **longueur** de la **première dimension**, ce qui permet d'arriver à la **bonne** « **ligne** ». Ensuite, il ne reste plus qu'à **sélectionner** le **bon élément** de la « **colonne** » à l'aide du **second indice**. Bien qu'un petit peu plus complexe quant à l'accès aux éléments, cette **solution** à l'**avantage** de n'**effectuer** qu'**une seule allocation** de mémoire. Dans ce cas ci, la **taille** du **tableau** étant **connue**

à l'avance, on aurait pu définir le pointeur p comme un pointeur sur un tableau de 3 int et ainsi s'affranchir du calcul d'adresse.

```
int (*p)[3] = malloc(sizeof(int[3][3]));
```

Allocation de plusieurs tableaux

La seconde solution consiste à allouer plusieurs tableaux plus un autre qui les référencera. Dans le cas d'un tableau à deux dimensions, cela signifie allouer un tableau de pointeurs dont chaque élément se verra affecter l'adresse d'un tableau également alloué dynamiquement. Cette technique permet d'accéder aux éléments des différents tableaux de la même manière que pour un tableau multidimensionnel puisque on utilise cette fois plusieurs tableaux. L'exemple ci-dessous revient au même que le précédent, mais utilise le procédé qui vient d'être décrit. Puisqu'on réserve un tableau de pointeurs sur int, l'adresse de celui-ci doit être stockée dans un pointeur de pointeur sur int.

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

int main(void)
{
    int **p = malloc(sizeof(int[3]));

    if (p == NULL)
    {
        printf("Échec de l'allocation\n");
        return EXIT_FAILURE;
    }
    for (unsigned i = 0; i < 3; ++i)
    {
        p[i] = malloc(sizeof(int[3]));
        if (p[i] == NULL)
        {
            printf("Échec de l'allocation\n");
            return EXIT_FAILURE;
        }
    }
    for (unsigned i = 0; i < 3; ++i)
        for (unsigned j = 0; j < 3; ++j)
        {
            p[i][j] = (i * 3) + j;
            printf("p[%u][%u] = %d\n", i, j, p[i][j]);
        }
    for (unsigned i = 0; i < 3; ++i)
        free(p[i]);
    free(p);
    return 0;
}
```

Si cette solution permet de faciliter l'accès aux différents éléments, elle présente toutefois l'inconvénient de réaliser plusieurs allocations et donc de nécessiter plusieurs appels à la fonction free().

Les tableaux de longueur variable

Il existe une autre solution pour allouer dynamiquement de la mémoire : les tableaux de longueur variable (ou variable length arrays en anglais, souvent abrégé en VLA).

Définition

En fait, jusqu'à présent, on avait toujours défini des tableaux dont la taille était soit déterminée, soit déterminable lors de la compilation.

```
int t1[3]; /* Taille déterminée */
int t2[] = { 1, 2, 3 }; /* Taille déterminable */
```

Toutefois, la taille d'un tableau étant spécifiée par une expression entière, il est parfaitement possible d'employer une variable à la place d'une constante entière. Ainsi, le code suivant est parfaitement correct.

```c
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    size_t n;

    printf("Entrez la taille du tableau : ");
    if (scanf("%zu", &n) != 1)
    {
        printf("Erreur lors de la saisie\n");
        return EXIT_FAILURE;
    }
    int tab[n];
    for (size_t i = 0; i < n; ++i)
    {
        tab[i] = i * 10;
        printf("tab[%zu] = %d\n", i, tab[i]);
    }
    return 0;
}

/* Résultat */
Entrez la taille du tableau : 10
tab[0] = 0
tab[1] = 10
tab[2] = 20
tab[3] = 30
tab[4] = 40
tab[5] = 50
tab[6] = 60
tab[7] = 70
tab[8] = 80
tab[9] = 90
```

Dans le cas où la taille du tableau ne peut pas être déterminée à la compilation, le compilateur va lui-même ajouter des instructions en vue d'allouer et de libérer de la mémoire dynamiquement.

Un tableau de longueur variable ne peut pas être initialisé lors de sa définition. Ceci tient au fait que le compilateur ne peut effectuer aucune vérification quant à la taille de la liste d'initialisation étant donné que la taille du tableau ne sera connue qu'à l'exécution du programme.

```c
int tab[n] = { 1, 2, 3 }; /* Incorrect */
```

> Une structure ne peut pas contenir de tableaux de longueur variable.

Utilisation

Un tableau de longueur variable s'utilise de la même manière qu'un tableau, avec toutefois quelques ajustements. En effet, s'agissant d'un tableau, s'il est employé dans une expression, il sera converti en un pointeur sur son premier élément. Toutefois, se pose alors la question suivante : Comment peut-on préciser le type du paramètre tab étant donné que la taille du tableau ne sera connue qu'à l'exécution ?

```c
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

void affiche_tableau(int (*tab)[/* Heu ... ? */], unsigned n, unsigned m)
{
    for (unsigned i = 0; i < n; ++i)
        for (unsigned j = 0; j < m; ++j)
            printf("tab[%u][%u] = %d\n", i, j, tab[i][j]);
}
```

```c
int main(void)
{
    size_t n, m;

    printf("Entrez la longueur et la largeur du tableau : ");
    if (scanf("%zu %zu", &n, &m) != 2)
    {
        printf("Erreur lors de la saisie\n");
        return EXIT_FAILURE;
    }
    int tab[n][m];
    affiche_tableau(tab, n, m);
    return 0;
}
```

Dans un tel cas, il est **nécessaire** d'**inverser** l'**ordre** des **paramètres** afin d'utiliser ceux-ci **pour préciser** au **compilateur** que le **type** ne sera **déterminé** qu'à l'**exécution**. *La définition de la fonction affiche_tableau() devient celle-ci.*

```c
void affiche_tableau(size_t n, size_t m, int (*tab)[m]);
{
    for (unsigned i = 0; i < n; ++i)
        for (unsigned j = 0; j < m; ++j)
            printf("tab[%u][%u] = %d\n", i, j, tab[i][j]);
}
```

Ainsi, on précise au **compilateur** que le **paramètre tab** sera un **pointeur** sur un **tableau** de **m int**, **m** n'étant **connu** qu'à l'**exécution**.

Application en dehors des tableaux de longueur variable

Cette **syntaxe** n'est **pas réservée** aux **tableaux** de **longueur variable**. En effet, si elle est **obligatoire dans** le cas de **leur utilisation**, elle peut être **employée** dans d'**autres situations**. *En reprenant l'exemple d'allocation d'un tableau multidimensionnel en un bloc et en le modifiant de sorte que l'utilisateur puisse spécifier la taille de ses dimensions, il est possible d'écrire ceci et dès lors éviter le calcul d'adresse manuel.*

```c
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    size_t n, m;

    printf("Entrez la longueur et la largeur du tableau : ");
    if (scanf("%zu %zu", &n, &m) != 2)
    {
        printf("Erreur lors de la saisie\n");
        return EXIT_FAILURE;
    }
    int (*p)[m] = malloc(sizeof(int[n][m]));
    if (p == NULL)
    {
        printf("Échec de l'allocation\n");
        return EXIT_FAILURE;
    }
    for (size_t i = 0; i < n; ++i)
        for (size_t j = 0; j < m; ++j)
        {
            p[i][j] = (i * n) + j;
            printf("p[%zu][%zu] = %d\n", i, j, p[i][j]);
        }
    free(p);
    return 0;
}
```

```
/* Résultat */
Entrez la longueur et la largeur du tableau : 3 3
p[0][0] = 0
p[0][1] = 1
p[0][2] = 2
p[1][0] = 3
p[1][1] = 4
p[1][2] = 5
p[2][0] = 6
p[2][1] = 7
p[2][2] = 8
```

Limitations

Pourquoi dans ce cas s'ennuyer avec l'allocation dynamique manuelle ? Parce que si cette méthode peut s'avérer salvatrice dans certains cas, elle souffre de **limitations** par rapport à l'**allocation dynamique manuelle**.

Durée de vie

La **première limitation** est que la **classe** de **stockage** des **tableaux** de **longueur variable** est **automatique**. Finalement, c'est assez logique, puisque le compilateur se charge de tout, il faut bien fixer le moment où la mémoire sera libérée. Du coup, **comme pour** n'importe quelle **variable automatique**, il est **incorrect** de **retourner** l'**adresse** d'un **tableau** de longueur variable puisque celui-ci **n'existera plus** une fois la **fonction terminée**.

```c
int *ptr(size_t n)
{
    int tab[n];

    return tab; /* Incorrect */
}
```

Goto

La **seconde limitation** tient au fait qu'une **allocation dynamique** est **réalisée** lors de la **définition** du **tableau** de longueur variable. En effet, finalement, la définition d'un tableau de longueur variable peut être vue comme suit.

```c
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    size_t n;

    printf("Entrez la taille du tableau : ");
    if (scanf("%zu", &n) != 1)
    {
        printf("Erreur lors de la saisie\n");
        return EXIT_FAILURE;
    }
    int tab[n]; /* int *tab = malloc(sizeof(int) * n); */
    /* free(tab); */
    return 0;
}
```

Or, assez logiquement, si l'**allocation** est **passée**, typiquement avec une instruction de saut comme **goto**, des **problèmes** risquent d'être **rencontrés** … Dès lors, le code suivant est incorrect :

```c
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    size_t n;
```

```c
    printf("Entrez la taille du tableau : ");
    if (scanf("%zu", &n) != 1)
    {
        printf("Erreur lors de la saisie\n");
        return EXIT_FAILURE;
    }
    goto boucle; /* Faux, car on saute l'allocation */
    if (n < 1000)
    {
        int tab[n]; /* int *tab = malloc(sizeof(int) * n); */
    boucle:
        for (size_t i = 0; i < n; ++i)
            tab[i] = i * 10;
        /* free(tab); */
    }
    return 0;
}
```

Il n'est pas permis de sauter après la définition d'un tableau de longueur variable si le saut est effectué en dehors de sa portée.

Taille des objets alloués

Une autre limitation vient du fait que la quantité de mémoire pouvant être allouée de cette manière est plus retreinte qu'avec malloc() et consœurs. Ceci tient à la manière dont la mémoire est allouée par le compilateur dans le cas des tableaux de longueur variable, qui diffère de celle employée par malloc(). Globalement et de manière générale, il faut éviter d'allouer un tableau de longueur variable si on a besoin de plus de 4 194 304 multiplets (*environ 4 Mo*).

Gestion d'erreur

La dernière limitation tient à l'impossibilité de mettre en place une gestion d'erreur. En effet, il n'est pas possible de savoir si l'allocation a réussi ou échoué et le programme est dès lors condamné à s'arrêter brutalement si cette dernière échoue.

En résumé

*Un **objet** est une **zone mémoire** pouvant **contenir** des **données** composée d'une **suite contiguë** d'un ou plusieurs **multiplets** ;*

*La **représentation** d'un **type** détermine de quelle **manière** les **données** sont **réparties** au sein d'un objet de ce type ;*

*Il est **possible** de **lire** le **contenu** d'un **objet multiplet par multiplet** à l'aide d'un **pointeur sur unsigned char** ;*

*La fonction **malloc()** permet d'**allouer dynamiquement** un **objet** d'une **taille donnée** ;*

*La fonction **calloc()** fait de même en **initialisant** chaque **multiplet** à **zéro** ; la **valeur zéro** n'est **pas forcément représentée** en **initialisant** tous les **multiplets** d'un objet à **zéro**, cela **dépend** des **types** ;*

*La fonction **realloc()** permet d'**augmenter** ou de **diminuer** la **taille** d'un **bloc** précédemment **alloué** avec **malloc()** ou **calloc()** ;*

*Il est **possible** de **préciser** la **taille** d'un **tableau** à l'aide d'une **variable** auquel cas le **compilateur** se **charge** pour nous d'**allouer** et de **libérer** de la **mémoire** pour ce tableau ;*

*L'**utilisation** de tels tableaux souffre de **limitations** par rapport à l'allocation dynamique manuelle.*

https://zestedesavoir.com/tutoriels/755/le-langage-c-1/1043_aggregats-memoire-et-fichiers/4285_lallocation-dynamique/

INTERFACE AVEC LE SYSTÈME UNIX

Les descripteurs de fichiers

Dans un **OS UNIX**, toutes les **entrées-sorties** sont réalisées en **lisant/écrivant** dans des **fichiers**. Il s'agit d'une interface unique et homogène gérant toutes les communications entre un programme et les périphériques.

Cas général : **avant** de **lire/écrire** dans un fichier, il faut **indiquer** au système son **intention** d'ouvrir un fichier. Si on a besoin d'écrire dans un fichier, il faudra peut être le créer ou écraser son contenu. Le **système vérifie** si on a les **droits** pour et dans l'**affirmative**, il **renvoie** au programme un **entier** nul ou positif de petite valeur appelé **descripteur de fichier**.

À **chaque entrée/sortie** sur un fichier, on utilise le **DF** à la **place** du **nom** du **fichier** pour le **référencer** (un DF ressemble à un pointeur sur fichier FILE *fp de la librairie standard – descripteur de fichier pour MS-DOS -). Toutes les **informations** concernant les **fichiers ouverts** sont tenues **à jour** par le **système**, le **programme** utilisateur faut **référence** au **fichier** uniquement à l'aide du **DF**.

Dispositions particulières pour les entrées/sorties clavier/écran pour les rendre plus pratiques :

Quand le **shell** lance un **programme**, **3 fichiers** sont **ouverts** auxquels on associe les **DFs 0**, **1** et **2** (*input std, output std et output erreur*). Un programme **lit** via le **DF 0** et **écrit** dans les fichiers descripteurs **1** et **2**, il peut alors réaliser des I/O sans se soucier d'ouvrir des fichiers. On peut **rediriger** les **flux** avec **<**, **<<** et **>**, **>>**. Le **shell** change alors l'**affectation implicite** des **DFs 0** et **1** en les affectant aux fichiers indiqués. Normalement le **DF 2** reste **attaché** à l'**écran**. C'est pareil en ce qui concerne les **I/O** via **|**. C'est le **shell** qui se **charge** de ces **changements**, **pas** le **programme** exécuté. **Tant que le programme utilise les DFs 0, 1 et 2, il n'a pas besoin de savoir d'où viennent les I/O.**

Entrées/sorties de bas niveau – read et write

Les **I/O** sont réalisées par les appels système **read** et **write**, qui sont accessibles via les programmes C par 2 fonctions appelées **read** et **write**. Pour les 2 fonctions, le **1er argument** est un **DF**, le **2e** est un **tableau** de **caractères** du programme, où les données doivent être reçues ou lues, le **3e argument** est le **nombre** d'**octets à transférer**.

```
int n_lus = read(int fd, char *tamp, int n);
int n_ecrits = write(int fd, char *tamp, int n);
```

Chaque **appel renvoie** le nombre d'**octets transférés**. En **lecture**, le **nombre** d'**octets** retournés peut être **inférieur** au nombre demandé en **argument**. Si **0** est **retourné**, alors on a atteint la **fin** du **fichier**, si c'est **-1** alors il y a une **erreur** quelconque. En **écriture**, c'est le **nombre** d'**octets écrits** qui est **retourné** et **-1** si le **nombre** d'octets est **différent** de l'**argument** passé.

On peut **lire/écrire** un **nombre quelconque** d'**octets** en utilisant un **seul appel**. Les valeurs les plus courantes sont **1**, **pour 1 caractère à la fois** (*sans mise en mémoire tampon*). Un nombre tel que **1024** ou **4096** indique la **taille** d'un **bloc physique** pour un **périphérique**. **Des tailles plus grandes augmentent l'efficacité car il y a moins d'appels système**.

Cf bibilothèque **<unistd.h>** pour utiliser **read** et **write**. La taille du tableau de caractères est supposée être une bonne taille de tampon pour le système sur lequel on travaille.

Si la taille d'un fichier n'est pas un multiple de **BUFSIZ** (*BUFSIZ = 8192 <stdio.h>*), certains appel à **read** retournent un **nombre inférieur** à l'**argument**, l'appel **suivant** à **read** retournera **0** (*fin de fichier*).

Appels système open, creat, close et unlink

En dehors des **DFs 0**, **1** et **2**, ouverts par défaut, on doit **explicitement ouvrir** les **fichiers** sur lesquels on doit **lire/écrire**. Pour cela, il y a 2 appels système à utiliser : **open** et **creat** ; **open** ressemble à **fopen** sauf que le **retour** est un **DF** au lieu d'un pointeur sur fichier. S'il y a une **erreur**, **-1** est retourné.

```
int fd;
int open(char *name, int flags, int perms);

fd = open(nom, drapeaux, perms);
```

Le **nom** correspond au nom du **fichier** *(ou son chemin absolu ou relatif)* à ouvrir, les **flags** indiquent **comment ouvrir** le fichier : **O_RDONLY** (*lecture*), **O_WRONLY** (*écriture*) et **O_RDWR** (*lecture/écriture*), ces **constantes** sont disponibles dans la bibliothèque **<fcntl.h>** pour **UNIX** et **<sys/file.h>** pour **BSD**. Les perms seront expliquées plus bas, en attendant, dans ces exemples elles sont passées à 0.

Une **erreur** se produit si on **tente** d'**ouvrir** un fichier **inexistant**. Dans ce cas-là c'est **creat** qu'il faut utiliser : cela **crée** un **nouveau** fichier ou **réécrit** des **anciens**.

```
int creat(char *name, int perms);

fd = creat(nom, perms);
```

Un **DF** est **retourné** s'il est **possible** de **créer** le **fichier**, sinon **-1** si **erreur**. Si le **fichier** existait **déjà**, **creat** le **réduit** à une **longueur nulle**, effaçant le contenu précédent (ce n'est pas une erreur). Les **perms** sont les **droits** d'**accès** au fichier. Sur **UNIX**, il y a **9 bits** de **droits** d'**accès** associés par fichier. On peut utiliser un **nombre** à **3 chiffres** en **base 8** pour donner les droits d'accès : **0755** (*lect/écrit/exéc pour le propriétaire, lect/écrit pour le groupe et les autres utilisateurs*).

Avec l'appel système **stat**, on peut **déterminer** le **mode** d'un fichier existant et **donner** le **même mode** au fichier copié. L'appel système stat prend un nom de fichier et retourne toutes les informations contenues dans l'inode correspondant ou -1 en cas d'erreur. http://manpagesfr.free.fr/man/man2/stat.2.html , *la structure stat y est développée, sinon p178 du K&R et <sys/stat.h>*.

```
int stat(const char *path, struct stat *buf);
```

Le **nombre** de **fichiers ouverts** en même temps est **limité** (env. **20**). Donc tout programme désirant manipuler beaucoup de fichiers doit être prêt à se servir/réutiliser les DFs. La fonction **close(int fd)** **rompt** la **liaison** entre un **DF** et un **fichier** ouvert, et **libère** le **DF** pour une utilisation sur un autre fichier : correspond à **fclose** de la **lib std**, sauf qu'il n'y a pas de tampon dont il faut forcer l'écriture.

La **fin** d'un **programme** par **exit** ou **return** **ferme tous** les **fichiers ouverts**. La fonction **unlink(char *name)** **détruit** le **fichier name** du système de fichiers, correspond à **remove** de la **lib std** (ou *rm* de *bash*).

L'accès sélectif – lseek

Les **I/O** sont normalement **séquentielles** : chaque **lecture/écriture** a lieu dans le fichier à la **position** juste derrière de celle où a eu **lieu** la **précédente** opération. Toutefois, en cas de nécessité, un fichier peut être **lu/écrit** dans un **ordre arbitraire**. L'appel système **lseek** permet de se **déplacer** dans un fichier sans lire/écrire :

```
long lseek(int fd, long offset, int origin);
```

On fixe la **position courante** à **offset** dans le fichier dont le **DF** est **fd** ; **offset** est une **position relative** à celle donnée par **origin**. La **lecture/écriture** suivante se fera à partir de cette **nouvelle position**. Les **valeurs** prises par **origin** peuvent être **0**, **1** ou **2** et permettent d'indiquer s'il faut calculer offset : **0** = depuis le **début**, **1** = depuis la **position courante**, **2** = depuis la **fin** du fichier.

Par exemple, pour **ajouter** à la **suite** :

```
lseek(fd, 0L, 2);
```

Pour **retourner** au **début** (rembobinage) :

```
lseek(fd, 0L, 0);
```

Avec **lseek**, on peut traiter des **fichiers** comme des grands **tableaux**, mais le **temps** d'accès est plus **long**.

```
/* lire : Lit n octets à partir de la position pos */
int lire(int fd, long pos, char *tamp, int n);
{
    if (lseek(fd, pos, 0) >= 0) /* se place à pos */
```

```
        return read(fd, tamp, n);
    else
        return -1
}
/* Cette fonction lit un nombre d'octets quelconque à un endroit arbitraire dans un fichier
et retourne le nombre d'octets lus ou -1 si erreur.*/
```

lseek retourne une **longueur** qui donne la **nouvelle position** dans le fichier ou **-1** si **erreur**. La fonction **fseek** de la **lib std** est **équivalente** sauf que le 1er argument est de type **FILE *** (*pointeur de fichier*) et que la valeur de retour est différente de 0 si une erreur se produit.

Notons qu'un **pointeur** de **fichier** (*FILE **) est un **pointeur** sur une **structure** contenant quelques informations sur le fichier : un **pointeur** sur un **tampon** permettant au fichier d'être lu par gros morceaux, un **compteur** du **nombre** de **caractères** restant dans le **tampon**, le **descripteur** de **fichier** et des **drapeaux** indiquant le mode de lecture/écriture, la description d'erreurs, etc. La **structure** de données qui décrit un **fichier** est contenue dans **<stdio.h>**. *Les noms des bibliothèques commençant par _ signifient que ces noms ne sont destinés qu'à être employés par la bibliothèque les décrivant (= private)*.

QUELQUES FONCTIONS UTILES

```
int printf(const char *format,...)

printf("%d\t%d\n", fahr, celsius);
```

fahr et **celsius** sont deux **variables**, et on demande à les afficher respectivement comme des **int** grâce à aux **arguments %d**. Les chaînes de caractères sont séparées par une tabulation, puis après **celsius**, on déclare une fin de ligne. On peut préciser une longueur d'affichage, en ajoutant une taille numérique devant les arguments. On peut tout à fait remplacer une variable par un calcul.

```
printf("%3d\t%6d\n", fahr, 5 * (fahr-32) / 9);
printf("%10.0f\t%7.2f\n", fahr, 5 * (fahr-32) / 9);
```

%f affiche un nombre **flottant** (un nombre à virgule), le point **.** précise le **nombre** de chiffre après la **virgule**. Pour 0 chiffre après la virgule, on écrira « **.0** ».

```
getchar(lire_caractère) // lit depuis l'entrée standard en général
c = getchar();
```

Pour **rediriger** l'**entrée** du **clavier** vers un **fichier** à l'appel de **getchar** via un programme appelé **prog** :

```
prog <fichier_entree
```

<fichier_entree ne sera **pas pris** en **compte** par **argv**. C'est **invisible pour** le programme **prog**. On peut également **utiliser** le pipe **|** (tube), le **>** de redirection de sortie (*voir plutôt putchar pour sortie*). **getchar** et **putchar** sont des **macros** définies dans **<stdio.h>** -> **/usr/include**.

```
putchar(écrire_caractère) // écrit sur la sortie standard
putchar(c);
```

Tous les **arguments** dans lesquels **scanf** stockera les données utilisateurs doivent être des **pointeurs** ! (*appel par valeur*)

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

Index des fonctions trouvées dans le K&R

NOM DES FONCTIONS	PAGES				
_fillbuff	176				
addpoint	128				
aff_arbre	139				
affd (atod) afficher n en décimal	85				
affiche 10 éléments d'un tableau par	52				

stat	179				
strcat	48				
strcmp	104				
strlen	39	97	101		
system (exécute cmd dans un programme)	164				
taillefichier	179				
tasser (supprimer tous les "c" dans "s")	47				
tri de Shell	62				
trim (supprime caractères d'espacements fin de chaîne)	64				
unlink	172				
write	168				

LIENS UTILES

http://www.gecif.net/articles/linux/gcc.html
https://tdinfo.phelma.grenoble-inp.fr/2Aproj/fiches/prototype_main.pdf
https://fr.wikiversity.org/wiki/Introduction_au_langage_C
https://zestedesavoir.com/tutoriels/755/le-langage-c-1/
https://zestedesavoir.com/tutoriels/2787/la-verite-sur-les-tableaux-et-pointeurs-en-c/
www.c-faq.com/decl/spiral.anderson.html
http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf

Le langage C – Norme ANSI – 2e édition – *Brian W. Kernighan & Dennis M. Ritchie*.

NOTES