Programmation en Langage C

Alexandre Meslé

3 février 2020

Table des matières

Chapitre 1

Notes de cours

1.1 Introduction

1.1.1 Définitions et terminologie

Un programme **exécutable** est une suite d'instructions exécutée par le processeur. Ces instructions sont très difficile à comprendre, si par exemple, vous ouvrez avec un éditeur de texte (notepad, emacs, etc) un fichier exécutable, vous verrez s'afficher un charabia incompréhensible :

```
00000000
0000001f
0000003e
0000005d
0000007c
0000009b
000000ba
000000d9
000000f8
00000117
00000136
00000155
00000174
00000193
000001b2
000001d1
000001f0
0000020f
0000022e
0000024d
0000026c
0000028b
000002aa
000002c9
000002e8
00000307
...
```

Il n'est pas envisageable de créer des programmes en écrivant des suites de chiffres et de lettres. Nous allons donc utiliser des langages de programmation pour ce faire.

Un programme C est un **ensemble d'instructions** qui se saisit dans un fichier `.c` à l'aide d'un éditeur (ex. : Notepad), ce type de fichier s'appelle une **source**. Les instructions qui y sont écrites s'appellent du **code** ou encore le

code source. Un compilateur est un logiciel qui lit le code source et le convertit en un **code exécutable**, c'est-à-dire un ensemble d'instructions compréhensible par le processeur. La compilation d'une source se fait en deux étapes : la compilation proprement dite et le **linkage**. On utilise en général le terme compilation en englobant les deux étapes précédemment citées.

— La compilation à proprement parler produit un fichier objet, dont l'extension est `.obj`.
— Les sources peuvent être réparties dans plusieurs fichiers, ceux-ci doivent tous être compilés séparément. Ensuite, le linkage des différents fichiers objets est assemblage produisant un exécutable `.exe`.

Certains environnement de développement servent d'éditeur, et prennent en charge la compilation et le linkage (eclipse, dev-C++).

1.1.2 Hello World !

Allez sur `http://www.01net.com/telecharger/windows/Programmation/langage/fiches/2327.html`, cliquez sur télécharger. Installez Dev-C++. Ouvrez un nouveau fichier source. Copiez-collez le code ci-dessous dans l'éditeur :

```
#include<stdio.h>

int main()
{
  printf("Hello World!\n");
  return 0;
}
```

Sauvegardez ce fichier sous le nom `helloWorld.c`. Dans le menu Exécuter, cliquez sur Compiler. Remarquez que la compilation a produit un fichier `helloWorld.exe`. Vous avez deux façons de l'exécuter :

— double-cliquez directement sur `helloWorld.exe`
— Dans le menu Exécuter, cliquez sur exécuter.

1.1.3 Structure d'un programme C

Importation des bibliothèques

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

Selon ce que l'on souhaite faire dans notre programme, on peut avoir besoin de différentes fonctions. Celles-ci sont disponibles dans des bibliothèques. Par exemple, `stdio.h` propose des fonctions de saisie et d'affichage.

Corps du programme

```
int main()
{
        printf("Hello world !");
        getch();
        return 0;
}
```

On place dans les accolades du main les instructions que l'on souhaite voir s'exécuter :

```
int main()
{
        <instructionsAExecuter>
        return 0;
}
```

Remarquez que chaque ligne se termine par un points-virgule. Pour afficher une variable de type aphanumérique en C, on utilise `printf(<valeur àafficher>)`;. Les littéraux de type alphanumériques en C s'écrivent entre doubles quotes.

1.1.4 Commentaires

Un commentaire est une séquence de caractères ignorée par le compilateur, on s'en sert pour expliquer des portions de code. Alors ayez une pensée pour les pauvres programmeurs qui vont reprendre votre code après vous, le pauvre correcteur qui va essayer de comprendre votre pensée profonde, ou bien plus simplement à vous-même au bout de six mois, quand vous aurez complètement oublié ce que vous aviez dans la tête en codant. On délimite un commentaire par /* et */. Par exemple,

```
int main()
{
        /*
        Ceci est un commentaire :
                L'instruction ci-dessous affiche ''Hello morld!''
        Ces phrases sont ignorées par le compilateur.
        */
        printf("Hello world!\n");
        return 0;
}
```

1.2 Variables

Une variable est un emplacement de la mémoire dans lequel est stockée une valeur. Chaque variable porte une nom et c'est ce nom qui sert à identifier l'emplacement de la mémoire représenté par cette variable. Pour utiliser une variable, la première étape est la déclaration.

1.2.1 Déclaration

Déclarer une variable, c'est prévenir le compilateur qu'un nom va être utilisé pour désigner un emplacement de la mémoire. En C, on déclare les variables juste après l'accolade suivant main() (pour le moment). On place les instructions à exécuter à la suite de la déclaration de variables.

```
main()
{
        <declaration de variables>
        <instructions a executer>
}
```

Nous ne travaillerons pour le moment que sur les variables de type numérique entier. Le type qui y correspond, en C, est **int**. On déclare les variables entières de la manière suivante :

```
int <var_1>, <var_2>, ..., <var_n>;
```

Cette instruction déclare les variables `var_1`, `var_2`, ..., `var_n` de type entier. Par exemple,

```
#include<stdio.h>

int main()
{
        int variable1, variable2;
        int autrevariable1, autrevariable2;
        return 0;
}
```

On appelle bloc le contenu des accolades du main.

1.2.2 Affectation

Si on souhaite affecter à la variable `v` une valeur, on utilise l'opérateur =. Par exemple,

```
int main()
{
        int v;
        v = 5;
        return 0;
}
```

Ce programme déclare une variable de type entier que l'on appelle `v`, puis lui affecte la valeur 5. Comme vous venez de le voir, il est possible d'écrire directement dans le code une valeur que l'on donne à une variable. Notez l'abscence de `include`, ce programme n'effectue aucune entrée-sortie, il n'est donc pas nécessaire d'importer les bibliothèques d'entrées/sorties.

Les opérations arihmétiques disponibles sont l'addition (+), la soustraction (−), la multiplication (∗), la division entière dans l'ensemble des entiers relatifs (quotient : /, reste : %).

```
int main()
{
        int v, w, z;
        v = 5;
        w = v + 1;
```

```
        z = v + w / 2;
        v = z % 3;
        v = v * 2;
        return 0;
}
```

1.2.3 Saisie

Traduisons en C l'instruction `Saisir <variable>` que nous avons vu en algorithmique. Pour récupérer la saisie d'un utilisateur et la placer dans une variable `<variable>`, on utilise l'instruction suivante :

```
scanf("%d", &<variable>);
```

`scanf` suspend l'exécution du programme jusqu'à ce que l'utilisateur ait saisi une valeur et pressé la touche **return**. La valeur saisie est alors affetée à la variable `<variable>`. **N'oubliez surtout pas le &**, sinon des choses absurdes pourraient se produire !

1.2.4 Affichage

Traduisons maintenant l'instruction `Afficher variable` en C. Cette instruction permet d'afficher la valeur d'une variable.

```
printf("%d", <variable>);
```

Cette instruction affiche la valeur contenue dans la variable `variable` (ne vous laissez pas impressionner par l'apparente complexité de cette instruction !). Nous avons étendu, en algorithmique, l'instruction `Afficher` en intercalant des valeurs de variables entre les messages affichés. Il est possible de faire de même en C :

```
printf("la valeur de la variable v est %d", v);
```

Cette instruction substitue à `%d` la valeur de la variable v. Il est possible de l'étendre à volonté :

```
printf("les valeurs des variables x, y et z sont %d, %d et %d", x, y, z);
```

Cette instruction substitue à chaque `%d` les valeurs des trois variables `x`, `y` et `z`. Par exemple, le programme suivant

```
#include<stdio.h>

int main()
{
        int a, b, c;
        a = 1;
        b = 2;
        c = 3
        printf("La valeur de a est %d, celle de b est %d, et celle de"
                                "c est %d.", a, b, c);
        return 0;
}
```

affiche

```
La valeur de a est 1, celle de b est 2, et celle de c est 3.
```

1.2.5 Entiers

Trois types de base servent à représenter les entiers :

nom	taille (t)	nombre de valeurs (2^{8t})	chaîne de format
`short`	1 octet	2^8 valeurs	`%hd`
`int`	2 octets	2^{16} valeurs	`%d`
`long`	4 octets	2^{32} valeurs	`%ld`

Entiers non signés

Par défaut, les entiers permettent de stocker des valeurs de signe quelconque. Si on préfixe un type entier par **unsigned**, on le restreint à des valeurs uniquement positives. Dans ce cas, on a

nom	taille (t)	nombre de valeurs (2^{8t})	valeur min	valeur max	format
`unsigned short`	1 octet	2^8 valeurs	0	2^8-1	`%hu`
`unsigned int`	2 octets	2^{16} valeurs	0	$2^{16}-1$	`%u`
`unsigned long`	4 octets	2^{32} valeurs	0	$2^{32}-1$	`%lu`

La plage de valeur d'un **unsigned short**, encodée en binaire, est

$$\{0000\ 0000, 0000\ 0001, 0000\ 0010, 0000\ 0011, \ldots, 1111\ 1110, 1111\ 1111\}$$

Entiers signés

Si par contre les données sont signées, on a comme plage de valeurs

nom	taille (t)	nombre de valeurs (2^{8t})	plus petite valeur	plus grande valeur
`short`	1 octet	2^8 valeurs	-2^7	2^7-1
`int`	2 octets	2^{16} valeurs	-2^{15}	$2^{15}-1$
`long`	4 octets	2^{32} valeurs	-2^{31}	$2^{31}-1$

La plage de valeur d'un **short**, encodée en binaire, est aussi

$$\{0000\ 0000, 0000\ 0001, 0000\ 0010, 0000\ 0011, \ldots, 1111\ 1110, 1111\ 1111\}$$

Même si les codages sont les mêmes, la signification ne l'est pas, les nombres entiers positifs sont codés sur l'intervalle :

$$\{0000\ 0000, 0000\ 0001, 0000\ 0010, 0000\ 0011, \ldots, 0111\ 1100, 0111\ 1101, 0111\ 1110, 0111\ 1111\}$$

Et les nombres négatifs sur l'intervalle

$$\{1000\ 0000, 1000\ 0001, 1000\ 0010, 1000\ 0011, \ldots, 1111\ 1100, 1111\ 1101, 1111\ 1110, 1111\ 1111\}$$

Les nombres négatifs sont disposés du plus grand jusqu'au plus petit, l'intervalle précédent code les valeurs :

$$\{-2^7, -(2^7-1), -(2^7-2), -(2^7-3), \ldots, -4, -3, -2, -1\}$$

Les opérations arithmétiques sont exécutées assez bêtement, si vous calculez $0111\ 1111 + 0000\ 0001$, ce qui correspond à $(2^7-1)+1$, le résultat mathématique est 2^7, ce qui se code 1000 0000, ce qui est le codage de -2^7. Soyez donc attentifs, en cas de dépassement de capacité d'un nombre entier, vous vous retrouverez avec des nombres qui ne veulent rien dire. Si vous souhaitez faire des calculs sur des réels, un type flottant sera davantage adapté.

1.2.6 Flottants

Les flottants servent à représenter les réels. Leur nom vient du fait qu'on les représente de façon scientifique : un nombre décimal (à virgule) muni d'un exposant (un décalage de la virgule). Trois types de base servent à représenter les flottants :

nom	taille	chaîne de format
`float`	4 octet	`%f`
`double`	8 octets	`%f`
`long double`	10 octets	`%Lf`

Il est conventionel d'écrire des littéraux flottants avec un point, par exemple l'approximation à 10^{-2} près de π s'ecrit 3.14. Il est aussi possible d'utiliser la notation scientifique, par exemple le décimal 1 000 s'écrit `1e3`, à savoir 1.10^3.Nous nous limiterons à la description du type **float**, les autres types obéissent à des règles similaires.

Représentation en mémoire d'un float

Le codage d'un nombre de type **float** (32 bits) est découpé en trois parties :

partie	taille
le bit de signe	1 bit
l'exposant	8 bits
la mantisse	23 bits

Le nombre est positif si le bit de signe est à 0, négatif si le bit de signe est à 1. La mantisse et l'exposant sont codés en binaire, la valeur absolue d'un flottant de mantisse m et d'exposant e est $\frac{m}{2^{23}}.2^e$. Le plus grand entier qu'il est possible de coder sur 23 octets est $\frac{(2^{23}-1)}{2^{23}}$, le plus grand exposant est 2^7 , donc le plus grand flottant est $\frac{(2^{23}-1)}{2^{23}}.2^{(2^7)}$, donc le plus petit est $-\frac{(2^{23}-1)}{2^{23}}.2^{(2^7)}$.

1.2.7 Caractères

Un **char** sert à représenter le code `ASCII` d'un caractère, il est donc codé sur 1 octet. Il est possible d'affecter à une telle variable toute valeur du code `ASCII` entourée de simples quotes. Par exemple, l'affectation suivante place dans `a` le code `ASCII` du caractère `'B'`.

```
char a;
a = 'B';
```

De la même façon que l'on code des nombres entiers sur 1 octet avec un **short**, un **char** contient avant tout un nombre codé sur 1 octet. Pour afficher le caractère corrspondant au code `ASCII` stocké dans un **char**, on utilise la chaîne de format `%c`, et pour afficher le code `ASCII` lui-même on utilise la chaîne de format `%d`. Si on souhaite manipuler des **char** comme des nombres, le type **unsigned char** permet de coder des valeurs positives sur 1 octet, alors que **char** code aussi bien des nombres positifs que négatifs. Par conséquent, les caractères de code `ASCII` 255 et 128 correspondent aux **unsigned char** de valeurs 255 et 128 et aux **char** de valeurs -1 et -128.

1.2.8 Constantes

Une constante est une valeur portant un nom, contrairement aux variables, elles ne sont pas modifiables. Les constantes en C sont non typées, on les définit dans l'entête de la source, juste en dessous des **#include**. La syntaxe est **#define** <NOM_CONSTANTE> <valeurConstante>, par exemple **#define** N 25 définit une constante N qui a la valeur 25. Au début de la compilation, un programme appelé **préprocesseur** effectue un rechercher/remplacer et substitue à toutes les occurences de <`NOM_CONSTANTE`> la valeur <`valeurConstante`>. Faites donc très attention au fait que dans l'exemple présent, le préprocesseur va remplacer tous les N par des 25. Donc si une variable s'appelle N, le préprocesseur va le remplacer par un 25 ! Si par exemple vous écrivez

```
#include<stdio.h>

#define N 25

int main()
{
  int N;
  N = 12;
  printf("N = %d", N);
  return 0;
}
```

Le préprocesseur va le transformer en

```
#include<stdio.h>

#define N 25
```

```c
int main()
{
  int 25;
  25 = 12;
  printf("N = %d", 25);
  return 0;
}
```

Vous remarquez que les seules exceptions sont les valeurs délimitées par des ".

1.3 Opérateurs

1.3.1 Généralités

Opérandes et arité

Lorsque vous effectuez une opération, par exemple 3 + 4, le + est un opérateur, 3 et 4 sont des opérandes. Si l'opérateur s'applique à 2 opérandes, on dit qu'il s'agit d'un opérateur **binaire**, ou bien d'**arité** 2. Un opérateur d'arité 1, dit aussi **unaire**, s'applique à un seul opérande, par exemple `-x`, le `x` est l'opérande et le − unaire est l'opérateur qui, appliqué à `x`, nous donne l'opposé de celui-ci, c'est-à-dire le nombre qu'il faut additionner à `x` pour obtenir 0. Il ne faut pas le confondre avec le − binaire, qui appliqué à x et y, additionne à x l'opposé de y.
En C, les opérateurs sont unaires ou binaires, et il existe un seul opérateur ternaire.

Associativité

Si vous écrivez une expression de la forme `a + b + c`, où `a`, `b` et `c` sont des variables entières, vous appliquez deux fois l'opérateur binaire + pour calculer la somme de 3 nombres `a`, `b` et `c`. Dans quel ordre ces opérations sont-elles effectuées ? Est-ce que l'on a `(a + b) + c` ou `a + (b + c)` ? Cela importe peu, car le + **entier** est **associatif**, ce qui signifie qu'il est possible de modifier le parenthèsage d'une somme d'entiers sans en changer le résultat. Attention : l'associativité est une rareté ! Peu d'opérateurs sont associatifs, une bonne connaissance des règles sur les priorités et le parenthèsage par défaut est donc requise.

Formes préfixes, postfixes, infixes

Un opérateur unaire est **préfixe** s'il se place avant son opérande, **postfixe** s'il se place après. Un opérateur binaire est **infixe** s'il se place entre ses deux opérandes (`a + b`), **préfixe** s'il se place avant (`+ a b`), **postfixe** s'il se place après (`a b +`). Vous rencontrez en `C` des opérateurs unaires préfixes et d'autres postfixes (on imagine difficilement un opérateur unaire infixe), par contre tous les opérateurs binaires seront infixe.

Priorités

Les règles des priorités en C sont nombreuses et complexes, nous ne ferons ici que les esquisser. Nous appelerons **parenthèsage implicite** le parenthèsage adopté par défaut par le C, c'est à dire l'ordre dans lequel il effectue les opérations. La première règle à retenir est qu'un opérateur unaire est **toujours prioritaire** sur un opérateur binaire.

1.3.2 Les opérateurs unaires

Négation arithmétique

La négation arithmétique est l'opérateur − qui à une opérande x, associe l'opposé de x, c'est-à-dire le nombre qu'il faut additionner à x pour obtenir 0.

Négation binaire

La négation binaire `~` agit directement sur les bits de son opérande, tous les bits à 0 deviennent 1, et vice-versa. Par exemple, `~127` (tous les bits à 1 sauf le premier) est égal à 128 (le premier bit à 1 et tous les autres à 0).

Priorités

Tous les opérateurs unaires sont de priorité équivalente, le parenthèsage implicite est fait le plus à droite possible, on dit que ces opérateurs sont **associatifs à droite**. Par exemple, le parenthèsage implicite de l'expression `~-~i` est `~(-(~i))`. C'est plutôt logique : si vous parvenez à placer les parenthèses différement, prevenez-moi parce que je ne vois pas comment faire...

1.3.3 Les opérateurs binaires

Opérations de décalages de bits

L'opération `a` $>>$ 1 effectue un décalage des bits de la représentation binaire de `a` vers la droite. Tous les bits sont décalés d'un cran vers la droite, le dernier bit disparaît, le premier prend la valeur 0. L'opération `a` $<<$ 1 effectue un décalage des bits de la représentation binaire de `a` vers la gauche. Tous les bits sont décalés d'un cran vers la gauche, le premier bit disparaît et le dernier devient 0. Par exemple, 32 $<<$ 2 associe à `0010 0000 « 2` la valeur dont la représentation binaire `1000 0000` et `32 » 3` associe à `0010 0000 » 3` la valeur dont la représentation binaire `0000 0100`.

Opérations logiques sur la représentation binaire

L'opérateur & associe à deux opérandes le ET logique de leurs représentations binaires par exemple 60 & 15 donne `12`, autrement formulé `0011 1100 ET 0000 1111 = 0000 1100`. L'opérateur | associe à deux opérandes le OU logique de leurs représentations binaires par exemple `60 | 15` donne `63`, autrement formulé `0011 1100 OU 0000 1111 = 0011 1111`. L'opérateur ^ associe à deux opérandes le OU **exclusif** logique de leurs représentations binaires par exemple 60 ^ 15 donne `51`, autrement formulé `0011 1100 OU EXCLUSIF 0000 1111 = 0011 0011`. Deux ^ successifs s'annulent, en d'autres termes `a^b^b`=`a`.

Affectation

Ne vous en déplaise, le = est bien un opérateur binaire. Celui-ci affecte à l'opérande de gauche (appelée `Lvalue` par le compilateur), qui doit être une variable, une valeur calculée à l'aide d'une expression, qui est l'opérande de droite. Attention, il est possible d'effectuer une affectation pendant l'évaluation d'une expression. Par exemple,

```
a = b + (c = 3);
```

Cette expression affecte à `c` la valeur 3, puis affecte à `a` la valeur `b` + `c`.

Priorités

Tous les opérateurs binaires ne sont pas de priorités équivalentes. Ceux de priorité la plus forte sont les opérateurs arithmétiques ($*$, /, &, $+$, $-$), puis les opérateus de décalage de bit ($<<$, $>>$), les opérateurs de bit (&, ^, |), et enfin l'affectation =. Représentons dans une tableau les opérateurs en fonction de leur priorité, plaçons les plus prioritaire en haut et les moins prioritaires en bas. Parmi les opérateurs arithmétiques, les multiplications et divisions sont prioritaires sur les sommes et différence :

noms	opérateurs
produit	`*`, `/`, `%`
sommes	`+`, `-`

Les deux opérateurs de décalage sont de priorité équivalente :

noms	opérateurs
décalage binaire	`»`, `«`

L'opérateur & est assimilé à un produit, | à une somme. Donc , & est prioritaire sur |. Comme ^ se trouve entre les deux, on a

noms	opérateurs
ET binaire	`&`
OU Exlusif binaire	`^`
OU binaire	`|`

Il ne nous reste plus qu'à assembler les tableaux :

noms	opérateurs
produit	*, /, %
somme	+, -
décalage binaire	», «
ET binaire	&
OU Exlusif binaire	^
OU binaire	|
affectation	=

Quand deux opérateurs sont de même priorité le parenthèsage implicite est fait le plus à **gauche possible**, on dit que ces opérateurs sont **associatifs à gauche**. Par exemple, le parenthèsage implicite de l'expression `a - b - c` est

```
(a - b) - c
```

et **certainement pas** `a - (b - c)`. Ayez donc cela en tête lorsque vous manipulez des opérateurs non associatifs ! La seule exception est le =, qui est associatif à droite. Par exemple,

```
a = b = c;
```

se décompose en `b = c` suivi de `a = b`.

1.3.4 Formes contractées

Le C étant un langage de paresseux, tout à été fait pour que les programmeurs aient le moins de caractères possible à saisir. Je vous préviens : j'ai placé ce chapitre pour que soyez capable de décrypter la bouillie que pondent certains programmeurs, pas pour que vous les imitiez ! Alors vous allez me faire le plaisir de faire usage des formes contractées avec parcimonie, n'oubliez pas qu'il est très important que votre code soit **lisible**.

Unaires

Il est possible d'**incrémenter** (augmenter de 1) la valeur d'une variable `i` en écrivant `i++`, ou bien `++i`. De la même façon on peut **décrémenter** (diminuer de 1) `i` en écrivant `i--` (forme postfixe), ou bien `--i` (forme préfixe). Vous pouvez décider d'incrémenter (ou de décrémenter) la valeur d'une variable pendant un calcul, par exemple,

```
a = 1;
b = (a++) + a;
```

évalue successivement les deux opérandes `a++` et `a`, puis affecte leur somme à `b`. L'opérande `a++` est évaluée à 1, puis est incrémentée, donc lorsque la deuxième opérande `a` est évaluée, sa valeur est 2. Donc la valeur de b après l'incrémentation est 3. L'incrémentation contractée sous forme postfixe s'appelle une **post-incrémentation**. Si l'on écrit,

```
a = 1;
b = (++a) + a;
```

On opère une **pré-incrémentation**, `++a` donne lieu à une incrémentation avant l'évaluation de `a`, donc la valeur 4 est affectée à `b`. On peut de façon analogue effectuer une **pré-decrémentation** ou a **post-decrémentation**. Soyez très attentifs au fait que ce code n'est pas portable, il existe des compilateurs qui évaluent les opérandes dans le désordre ou diffèrent les incrémentations, donnant ainsi des résultats autres que les résultats théoriques exposés précédement. Vous n'utiliserez donc les **incrémentation** et **decrémentation** contractées que lorsque vous serez certain que l'ordre d'évaluation des opérandes ne pourra pas influer sur le résultat.

Binaires

Toutes les affectations de la forme `variable = variable operateurBinaire expression` peuvent être contractées sous la forme `variable operateurBinaire= expression`. Par exemple,

avant	après
`a = a + b`	`a += b`
`a = a - b`	`a -= b`
`a = a * b`	`a *= b`
`a = a / b`	`a /= b`
`a = a % b`	`a %= b`
`a = a » i`	`a »= i`
`a = a « i`	`a «= i`
`a = a & b`	`a &= b`
`a = a ^ b`	`a ^= b`
`a = a | b`	`a |= b`

Vous vous douterez que l'égalité ne peut pas être contractée...

1.3.5 Opérations hétérogènes

Le fonctionnement par défaut

Nous ordonnons de façon grossière les types de la façon suivante : **long double** > **double** > **float** > **unsigned long** > **long** > **unsigned int** > **int** > **unsigned short** > **short** > **char**. Dans un calcul où les opérandes sont de types hétérogènes, l'opérande dont le type T est de niveau le plus élévé (conformément à l'ordre énoncé ci-avant) est sélectionné et l'autre est converti dans le type T.

Le problème

Il se peut cependant que dans un calcul, cela ne convienne pas. Si par exemple, vous souhaitez calculer l'inverse $\frac{1}{x}$ d'un nombre entier x, et que vous codez

```
int i = 4;
printf("L'inverse de %d est %d", i, 1/i);
```

Vous constaterez que résultat est inintéressant au possible. En effet, comme i et 1 sont tout deux de type entier, c'est la division entière est effectuée, et de toute évidence le résultat est 0. Changer la chaîne de format

```
int i = 4;
printf("L'inverse de %d est %f\n", i, 1/i);
```

se révèlera aussi d'une inefficacité notoire : non seulement vous vous taperez un warning, mais en plus `printf` lira un entier en croyant que c'est un flottant. Alors comment on se sort de là ? Ici la bidouille est simple, il suffit d'écrire le 1 avec un point :

```
int i = 4;
printf("L'inverse de %d est %f\n", i, 1./i);
```

Le compilateur, voyant un opérande de type flottant, convertit lors du calcul l'autre opérande, `i`, en flottant. De ce fait, c'est une division flottante et non entière qui est effectuée. Allons plus loin : comment faire pour appliquer une division flottante à deux entiers ? Par exemple :

```
int i = 4, j= 5;
printf("Le quotient de %d et %d est %f\n", i, j, i/j);
```

Cette fois-ci c'est inextricable, vous pouvez placer des points où vous voudrez, vous n'arriverez pas à vous débarasser du warning et ce programme persistera à vous dire que ce quotient est `-0.000000` ! Une solution particulièrement crade serait de recopier `i` et `j` dans des variables flottantes avant de faire la division, une autre méthode de bourrin est de calculer `(i + 0.)/j`. Mais j'espère que vous réalisez que seuls les boeufs procèdent de la sorte.

Le `cast`

Le seul moyen de vous sortir de là est d'effectuer un `cast`, c'est à dire une conversion de type sur commande. On caste en plaçant entre parenthèse le type dans lequel on veut convertir juste avant l'opérande que l'on veut convertir. Par exemple,

```
int i = 4, j= 5;
printf("Le quotient de %d et %d est %f\n", i, j, (float)i/j);
```

Et là, ça fonctionne. La variable valeur contenue dans `i` est convertie en **float** et de ce fait, l'autre opérande, `j`, est aussi convertie en **float**. La division est donc une division flottante. Notez bien que le `cast` est un opérateur unaire, donc prioritaire sur la division qui est un opérateur binaire, c'est pour ça que la conversion de `i` a lieu avant la division. Mais si jamais il vous vient l'idée saugrenue d'écrire

```
int i = 4, j= 5;
printf("Le quotient de %d et %d est %f\n", i, j, (float)(i/j));
```

Vous constaterez très rapidement que c'est une alternative peu intelligente. En effet, le résulat est flottant, mais comme la division a lieu avant toute conversion, c'est le résultat d'une division entière qui est converti en flottant, vous avez donc le même résultat que si vous n'aviez pas du tout casté.

1.3.6 Les priorités

Ajoutons le `cast` au tableau des priorités de nos opérateurs :

noms	opérateurs
opérateurs unaires	`cast`, `-`, , `++`, `-`
produit	`*`, `/`, `%`
somme	`+`, `-`
décalage binaire	`»`, `«`
ET binaire	`&`
OU Exlusif binaire	`^`
OU binaire	`|`
affectation	`=`

1.4 Traitements conditionnels

On appelle traitement conditionnel une portion de code qui n'est pas exécutée systématiquement, c'est à dire des instructions dont l'exécution est conditionnée par le succès d'un test.

1.4.1 Si ... Alors

Principe

En algorithmique un traitement conditionnel se rédige de la sorte :

Si *condition* **alors**
| instructions
Fin si

Si la condition est vérifiée, alors les instructions sont exécutées, sinon, elles ne sont pas exécutées. L'exécution de l'algorithme se poursuit alors en ignorant les instructions se trouvant entre le `alors` et le `finSi`. Un traitement conditionnel se code de la sorte :

```
if (<condition>)
{
        <instructions>
}
```

Notez bien qu'il n'y a pas de point-virgule après la parenthèse du **if**.

Comparaisons

La formulation d'une condition se fait souvent à l'aide des opérateurs de comparaison. Les opérateurs de comparaison disponibles sont :

- == : égalité
- != : non-égalité
- <, <= : inférieur à, respectivement strict et large
- >, >= : supérieur à, respectivement strict et large

Par exemple, la condition `a == b` est vérifiée si et seulement si `a` et `b` ont la même valeur au moment où le test est évalué. Par exemple,

```
#include<stdio.h>

int main()
{
        int i;
        printf("Saisissez une valeur : ");
        scanf("%d", &i);
        if (i == 0)
        {
                printf("Vous avez saisi une valeur nulle\n.");
        }
        printf("Adios !");
        return 0;
}
```

Si au moment où le test `i == 0` est évalué, la valeur de `i` est bien 0, alors le test sera vérifié et l'instruction `printf("Vous avez saisi une valeur nulle.")` sera bien exécutée. Si le test n'est pas vérifié, les instructions du bloc délimité par des accolades suivant le **if** sont ignorées.

Si ... Alors ... Sinon

Il existe une forme étendue de traitement conditionnel, on la note en algorithmique de la façon suivante :

Si *condition* **alors**
| instructions
Sinon
| autresinstructions
Fin si

Les instructions délimitées par `alors` et `sinon` sont exécutées si le test est vérifié, et les instructions délimitées par `sinon` et `finSi` sont exécutées si le test n'est pas vérifié. On traduit le traitement conditionnel étendu de la sorte :

```
if (<condition>)
{
        <instructions1>
}
else
{
        <instructions2>
}
```

Par exemple,

```
#include<stdio.h>

int main()
{
        int i;
        printf("Saisissez une valeur : ");
        scanf("%d", &i);
        if (i == 0)
        {
                printf("Vous avez saisi une valeur nulle\n.");
        }
        else
        {
                printf("La valeur que vous saisi, "
                "a savoir %d, n'est pas nulle.\n", i);
        }
        return 0;
}
```

Notez la présence de l'opérateur de comparaison ==. **N'utilisez jamais = pour comparer deux valeurs !**.

Connecteurs logiques

On formule des conditions davantage élaborées en utilisant des connecteurs `et` et `ou`. La condition `A et B` est vérifiée si les deux conditions `A` et `B` sont vérifiées simultanément. La condition `A ou B` est vérifiée si au moins une des deux conditions `A` et `B` est vérifiée. Le `et` s'écrit && et le `ou` s'écrit ||. Par exemple, voici un programme C qui nous donne le signe de $i \times j$ sans les multiplier.

```
#include<stdio.h>

int main()
{
 int i, j;
 printf("Saisissez deux valeurs : ");
 scanf("%d %d", &i, &j);
 printf("Le produit de ces deux valeurs est ");
 if ((i < 0 && j < 0) || (i >= 0 &&  j>= 0))
 {
  printf("positif\n.");
 }
```

```
 else
 {
  printf("negatif");
 }
 printf(".\n");
 return 0;
}
```

Accolades superflues

Lorsqu'une seule instruction d'un bloc **if** doit être exécutée, les accolades ne sont plus nécessaires. Il est possible par exemple de reformuler le programme précédent de la sorte :

```
#include<stdio.h>

int main()
{
 int i, j;
 printf("Saisissez deux valeurs : ");
 scanf("%d %d", &i, &j);
 printf("Le produit de ces deux valeurs est ");
 if ((i < 0 && j < 0) || (i >= 0 &&  j>= 0))
  printf("positif\n.");
 else
  printf("negatif");
 printf(".\n");
 return 0;
}
```

Opérateur ternaire

En plaçant l'instruction suivante à droite d'une affectation,

```
<variable> = (<condition>) ? <valeur> : <autrevaleur> ;
```

on place `valeur` dans `variable` si `condition` est vérifié, `autrevaleur` sinon. Par exemple,

```
max = (i>j) ? i : j ;
```

place la plus grande des deux valeurs `i` et `j` dans `max`. Plus généralement on peut utiliser le si ternaire dans n'importe quel calcul, par exemple

```
  printf("%d\n", (i> (l = (j>k) ? j : k)) ? i : l);
```

`j = (j>k)? j : k)` place dans `l` la plus grande des valeurs `j` et `k`, donc `(i > (l = (j > k)? j : k)) ? i : l` est la plus grande des valeurs `i`, `j` et `k`. La plus grande de ces trois valeurs est donc affichée par cette instruction.

1.4.2 Switch

Le **switch** en C s'écrit, avec la syntaxe suivante :

```
switch(<nomvariable>)
{
  case <valeur_1> : <instructions_1> ; break ;
  case <valeur_2> : <instructions_2> ; break ;
  /* ... */
  case <valeur_n> : <instructions_n> ; break ;
  default : /* instructions */
}
```

N'oubliez surtout pas les **break** ! Si par exemple, nous voulons afficher le nom d'un mois en fonction de son numéro, on écrit

```
switch(numeroMois)
{
  case 1 : printf("janvier") ; break ;
  case 2 : printf("fevrier") ; break ;
  case 3 : printf("mars") ; break ;
  case 4 : printf("avril") ; break ;
  case 5 : printf("mai") ; break ;
  case 6 : printf("juin") ; break ;
  case 7 : printf("juillet") ; break ;
  case 8 : printf("aout") ; break ;
  case 9 : printf("septembre") ; break ;
  case 10 : printf("octobre") ; break ;
  case 11 : printf("novembre") ; break ;
  case 12 : printf("decembre") ; break ;
  default : printf("Je connais pas ce mois...");
}
```

1.4.3 Booléens

Une variable booléenne ne peut prendre que deux valeurs : *vrai* et *faux*. Il n'existe pas de type booléen à proprement parler en C. On utilise des **int** pour simuler le comportement des booléens. On représente la valeur booléenne *faux* avec la valeur entière 0, toutes les autres valeurs entières servent à représenter *vrai*.

Utilisation dans des if

Lorsqu'une condition est évaluée, par exemple lors d'un test, cette condition prend à ce moment la valeur *vrai* si le test est vérifié, *faux* dans le cas contraire. La valeur entière 0 représente la constante *faux* et toutes les autres sont des constantes *vrai*. Observons le test suivant :

```
if (8)
{
  //...
}
```

1 est un littéral de type entier dont le valeur est non nulle, donc il représente le booléen *vrai*. De ce fait, le test **if** (8) est toujours vérifié. Par contre le test **if** (0) n'est jamais vérifié.

La valeur *vrai*

Même si tout entier non nul a la valeur vrai, on prend par défaut la valeur 1. Lorsqu'une condition est évaluée, elle prend la valeur 1 si elle est vérifié, 0 dans le cas contraire. par exemple,

```
x = (3>2);
```

On remarque que (3>2) est une condition. Pour décider quelle valeur doit être affectée à `x`, cette condition est évaluée. Comme dans l'exemple ci-dessus la condition est vérifiée, alors elle prend la valeur 1, et cette valeur est affectée à `x`.

Connecteurs logiques binaires

Les connecteurs || et && peuvent s'appliquer à des valeurs (ou variables) entières. Observons l'exemple suivant :

```
x = (3 && 0) || (1);
```

Il s'agit de l'affectation à `x` de l'évaluation de la condition (3 && 0)|| (1). Comme 3 est vrai et 0 est faux, alors leur conjonction est fausse. Donc (3 && 0) a pour valeur 0. La condition 0 || 1 est ensuite évaluée et prend la valeur vrai. Donc la valeur 1 est affectée à `x`.

Opérateur de négation

Parmi les connecteurs logiques se trouve !, dit opérateur de négation. La négation d'une expression est vraie si l'expression est fausse, fausse si l'expression est vraie. Par exemple,

```
x = !(3==2);
```

Comme 3 == 2 est faux, alors sa négation !(3 == 2) est vraie. Donc la valeur 1 est affectée à `x`.

1.4.4 Les priorités

Complètons notre tableau des priorités en y adjoignant les connecteurs logiques et les opérateurs de comparaison :

noms	opérateurs
opérateurs unaires	`cast`, `-`, `~`, `!`, `++`, `-`
produit	`*`, `/`, `%`
somme	`+`, `-`
décalage binaire	`»`, `«`
comparaison	`>`, `<`, `>=`, `<=`
égalité	`==`, `!=`
ET binaire	`&`
OU Exlusif binaire	`^`
OU binaire	`|`
connecteurs logiques	`&&`, `||`
if ternaire	`()?:`
affectations	`=`, `+=`, `-=`, ...

1.4.5 Préprocesseur

Macro-instructions

Le C nous permet de placer à peu près n'importe quoi dans les constantes. Si par exemple, on est lassé d'écrire des instructions de retour à la ligne, il suffit de définir

```
#define RC printf("\n")
```

Dans ce cas, toutes les occurrences de `RC` dans le code seront remplacées par des `printf("$\setminus$n")`. Par exemple

```
#include<stdio.h>

#define BEGIN {
#define END }

#define RC printf("\n")
#define AFFICHE_DEBUT printf("Debut")
#define AFFICHE_FIN printf("Fin")
#define AFFICHE_MLILIEU printf("Milieu")
#define RETOURNE_0 return 0


int main()
BEGIN
  AFFICHE_DEBUT;
  RC;
  AFFICHE_MLILIEU;
  RC;
  AFFICHE_FIN;
  RC;
  RETOURNE_0;
END
```

affiche

```
Debut
Milieu
Fin
```

Macros paramétrées

Il est possible de paramétrer les instructions du préprocesseur. Par exemple,

```
#define TROIS_FOIS_N(n) (3 * (n))
```

va remplacer toutes les occurences de `TROIS\_FOIS\_N(...)` par (3 * (...)) en substituant aux points de suspension la valeur se trouvant entre les parenthèses de la constante. Par exemple

```
#include<stdio.h>

#define RC printf("\n")
#define DECLARER_INT(i) int i
#define AFFICHER_INT(i) printf("%d", i)
#define SAISIR_INT(i) scanf("%d", &i)

int main()
{
  DECLARER_INT(k);
  printf("Saisissez un entier : ");
  SAISIR_INT(k);
  printf("Vous avez saisi ");
  AFFICHER_INT(k);
  RC;
  return 0;
}
```

affiche

```
Saisissez un entier : 4
Vous avez saisi 4
```

1.5 Boucles

Nous souhaitons créer un programme qui nous affiche tous les nombres de 1 à 10, donc dont l'exécution serait la suivante :

```
1 2 3 4 5 6 7 8 9 10
```

Une façon particulièrement vilaine de procéder serait d'ecrire 10 `printf` successifs, avec la lourdeur des copier/coller que cela impliquerait. Nous allons étudier un moyen de coder ce type de programme avec un peu plus d'élégance.

1.5.1 Définitions et terminologie

Une boucle permet d'exécuter plusieurs fois de suite une même séquence d'instructions. Cette ensemble d'instructions s'appelle le **corps** de la boucle. Chaque exécution du corps d'une boucle s'appelle une **itération**, ou plus informellement un **passage** dans la boucle. Lorsque l'on s'apprête à exécuter la première itération, on dit que l'on **rentre** dans la boucle, lorsque la dernière itération est terminée, on dit qu'on **sort** de la boucle. Il existe trois types de boucle :

- **while**
- **do** ... **while**
- **for**

Chacune de ces boucles a ses avantages et ses inconvénients. Nous les passerons en revue ultérieurement.

1.5.2 while

En C, la boucle `tant que` se code de la façon suivante :

```
while(<condition>)
{
  <instructions>
}
```

Les instructions du corps de la boucle sont délimitées par des accolades. La condition est évaluée **avant** chaque passage dans la boucle, à chaque fois qu'elle est vérifiée, on exécute les instructions de la boucle. Un fois que la condition n'est plus vérifiée, l'exécution se poursuit après l'accolade fermante. Affichons par exemple tous les nombres de 1 à 5 dans l'ordre croissant,

```
#include<stdio.h>

int main()
{
  int i = 1;
  while(i <= 5)
    {
      printf("%d ", i);
      i++;
    }
  printf("\n");
  return 0;
}
```

Ce programme **initialise** i **à** 1 et tant que la valeur de i n'éxcède pas 5, cette valeur est affichée puis incrémentée. Les instructions se trouvant dans le corps de la boucle sont donc exécutées 5 fois de suite. La variable i s'appelle un **compteur**, on gère la boucle par incrémentations successives de i et on sort de la boucle une fois que i a atteint une certaine valeur. **L'initialisation du compteur est très importante** ! Si vous n'initialisez pas i explicitement, alors cette variable contiendra n'importe quelle valeur et votre programme ne se comportera pas du tout comme prévu. Notez bien par ailleurs qu'il n'y a **pas de point-virgule après le while !**

1.5.3 do ... while

Voici la syntaxe de cette boucle :

```
do
{
  <instructions>
}
while(<condition>);
```

La fonctionnement est analogue à celui de la boucle `tant que` à quelques détails près :

- la condition est évaluée **après** chaque passage dans la boucle.
- On exécute le corps de la boucle **tant que** la condition est vérifée.

En C, la boucle `répéter ... jusqu'à` est en fait une boucle `répéter ... tant que`, c'est-à-dire une boucle `tant que` dans laquelle la condition est évaluée **à la fin**. Une boucle **do ... while** est donc exécutée donc **au moins une fois**. Reprennons l'exemple précédent avec une boucle **do ... while** :

```
#include<stdio.h>

int main()
{
  int i = 1;
  do
    {
      printf("%d ", i);
      i++;
    }
  while(i <= 5);
  printf("\n");
  return 0;
}
```

De la même façon que pour la boucle **while**, le compteur est initialisé avant le premier passage dans la boucle. Un des usages les plus courant de la boucle **do ... while** est le contrôle de saisie :

```
#include<stdio.h>

int main()
{
  int i;
  do
    {
      printf("Saisissez un entier positif ou nul : ");
      scanf("%d", &i);
      if (i < 0)
        printf("J'ai dit positif ou nul ! `\n");
    }
  while(i < 0);
  return 0;
}
```

1.5.4 for

Cette boucle est quelque peu délicate. Commençons par donner sa syntaxe :

```
for(<initialisation> ; <condition> ; <pas>)
{
  <instructions>
}
```

L'<initialisation> est une instruction exécutée avant le premier passage dans la boucle. La <condition> est évaluée **avant** chaque passage dans la boucle, si elle n'est pas vérifiée, on ne passe pas dans la boucle et l'exécution de la boucle pour est terminée. La <pas> est une instruction exécutée **après** chaque passage dans la boucle. On peut convertir une boucle **for** en boucle **while** en procédant de la sorte :

```
<initialisation>
while(<condition>)
{
  <instructions>
  <pas>
}
```

On re-écrit l'affiche des 5 premiers entiers de la sorte en utilisant le fait que <initialisation> = i = 1, <condition> = i <= 5 et <pas> = i++. On obtient :

```
#include<stdio.h>

int main()
{
  int i;
  for( i = 1 ; i <= 5 ; i++)
      printf("%d ", i);
  printf("\n");
  return 0;
}
```

On utilise une boucle **for** lorsque l'on connait en entrant dans la boucle combien d'itérations devront être faites. Par exemple, n'utilisez pas une boucle `pour` pour contrôler une saisie !

1.5.5 Accolades superflues

De la même façon qu'il est possibl de supprimer des accolades autour d'une instruction d'un bloc **if**, on peut supprimer les accolades autour du corps d'une boucle si elle ne contient qu'une seule instruction.

1.6 Tableaux

Considérons un programme dont l'exécution donne :

```
Saisissez dix valeurs :
1 : 4
2 : 7
3 : 34
4 : 1
5 : 88
6 : 22
7 : 74
8 : 19
9 : 3
10 : 51
Saissez une valeur
22
22 est la 6-eme valeur saisie
```

Comment programmer cela sans utiliser 10 variables pour stocker les dix premières valeurs saisies ?

1.6.1 Définitions

Une tableau est un regroupement de variables de même type, il est identifié par un nom. Chacune des variables du tableau est numérotée, ce numéro s'appelle un **indice**. Chaque variable du tableau est donc caractérisée par le nom du tableau et son indice.

Si par exemple, T est un tableau de 10 variables, alors chacune d'elles sera numérotée et il sera possible de la retrouver en utilisant simultanément le nom du tableau et l'indice de la variable. Les différentes variables de T porteront des numéros de 0 à 9, et nous appellerons chacune de ces variables un **élément** de T.

Une variable n'étant pas un tableau est appelée variable **scalaire**, un tableau par opposition à une variable scalaire est une variable **non scalaire**.

1.6.2 Déclaration

Comme les variables d'un tableau doivent être de même type, il convient de préciser ce type au moment de la délaration du tableau. De même, on précise lors de la déclaration du tableau le nombre de variables qu'il contient. La syntaxe est :

```
<type> <nomdutableau>[<taille>];
```

Par exemple,

```
int T[4];
```

déclare un tableau `T` contenant 4 variables de type **int**.

1.6.3 Initialisation

Il est possible d'initialiser les éléments d'un tableau à la déclaration, on fait cela comme pour des variables scalaires :

```
<type> <nom>[<taille>] = <valeur d initialisation>;
```

La seule chose qui change est la façon d'écrire la valeur d'initialisation, on ecrit entre accolades tous les éléments du tableau, on les dispose par ordre d'indice croissant en les séparant par des virgules. La syntaxe générale de la valeur d'initialisation est donc :

```
<type> <nom>[<taille>] = {<valeur_0>, <valeur_1>, ... , <valeur_n-1>};
```

Par exemple, on crée un tableau contenant les 5 premiers nombres impairs de la sorte :

```
int T[5] = {1, 3, 5, 7, 9};
```

1.6.4 Accès aux éléments

Les éléments d'un tableau à n éléments sont indicés de 0 à $n-1$. On note `T[i]` l'élément d'indice i du tableau `T`. Les cinq éléments du tableau de l'exemple ci-avant sont donc notés `T[0]`, `T[1]`, `T[2]`, `T[3]` et `T[4]`.

1.6.5 Exemple

Nous pouvons maintenant mettre en place le programme du début du cours. Il est nécessaire de stocker 10 valeurs de type entier, nous allons donc déclarer un tableau `E` de la sorte :

```
int E[10];
```

La déclaration ci-dessus est celle d'un tableau de 10 **int** appelé `E`. Il convient ensuite d'effectuer les saisies des 10 valeurs. On peut par exemple procéder de la sorte :

```
printf("Saisissez dix valeurs : \n");
printf("1");
scanf(%d", &E[0]);
printf("2");
scanf(%d", &E[1]);
printf("3");
scanf(%d", &E[2]);
printf("4");
scanf(%d", &E[3]);
printf("5");
scanf(%d", &E[4]);
printf("6");
scanf(%d", &E[5]);
printf("7");
scanf(%d", &E[6]);
printf("8");
scanf(%d", &E[7]);
printf("9");
scanf(%d", &E[8]);
printf("10");
scanf(%d", &E[9]);
```

Les divers copier/coller necéssaires pour rédiger un tel code sont d'une laideur à proscrire. Nous procéderons plus élégament en faisant une boucle :

```
printf("Saisissez dix valeurs : \n");
for(i = 0 ; i < 10 ; i++)
{
  printf("%d", i+1);
  scanf("%d", &E[i]);
}
```

Ce type de boucle s'appelle un **parcours** de tableau. En règle générale on utilise des boucles pour manier les tableaux, celles-ci permettent d'effectuer un traitement sur chaque élément d'un tableau. Ensuite, il faut saisir une valeur à rechercher dans le tableau :

```
printf("Saissez une valeur \n");
scanf("%d", &t);
```

Nous allons maintenant rechercher la valeur `t` dans le tableau `E`. Considérons pour ce faire la boucle suivante :

```
while (E[i] != t)
      i++;
```

Cette boucle parcourt le tableau jusqu'à trouver un élément de `E` qui ait la même valeur que `t`. Le problème qui pourrait se poser est que si `t` ne se trouve pas dans le tableau `E`, alors la boucle pourrait ne pas s'arrêter. Si `i` prend des valeurs strictement plus grandes que 9, alors il se produira ce que l'on appelle un **débordement d'indice**. Vous devez toujours veiller à ce qu'il ne se produise pas de débordement d'indice ! Nous allons donc faire en sorte que la boucle s'arrête si `i` prend des valeurs strictement supérieures à 9.

```
while (i < 10 && E[i] != t)
      i++;
```

Il existe donc deux façons de sortir de la boucle :

- En cas de débordement d'indice, la condition `i` < 10 ne sera pas vérifiée. Une fois sorti de la boucle, `i` aura la valeur 10.
- Dans le cas où `t` se trouve dans le tableau à l'indice `i`, alors la condition `E[i] != t` ne sera pas vérifiée et on sortira de la boucle. Un fois sorti de la boucle, `i` aura comme valeur l'indice de l'élément de `E` qui est égal à `t`, donc une valeur comprise entre 0 et 9.

On identifie donc la façon dont on est sorti de la boucle en testant la valeur de `i` :

```
if (i == 10)
  printf("%d ne fait pas partie des dix valeurs saisies", t);
else
  printf("%d est la %d-eme valeur saisie", t, i+1);
```

Si (`i` == 10), alors nous sommes sorti de la boucle parce que l'élément saisi par l'utilisateur ne trouve pas dans le tableau. Dans le cas contraire, `t` est la `i`+1-ème valeur saisie par l'utilisateur. On additionne 1 à l'indice parce que l'utilisateur ne sait pas que dans le tableau les éléments sont indicés à partir de 0. Récapitulons :

```
#include<stdio.h>

#define N 10

int main()
{
  int E[N], i, t;
  printf("Saisissez dix valeurs : \n");
  for (i = 0 ; i < N ; i++)
    {
      printf("%d : ", i+1);
      scanf("%d", &E[i]);
    }
  printf("Saissez une valeur \n");
  scanf("%d", &t);
  i = 0;
  while (i < N && E[i] != t)
    {printf("%d %d\n", i, E[i]);i++;}
    if (i == N)
      printf("%d ne fait pas partie des dix valeurs saisies", t);
    else
      printf("%d est la %d-eme valeur saisie", t, i+1);
  printf("\n");
  return 0;
}
```

1.7 Chaînes de caractères

1.7.1 Exemple

Etant donné le programme dont l'exécution est tracée ci dessous :

```
Saisissez une phrase :
Les framboises sont perchees sur le tabouret de mon grand-pere.
Vous avez saisi :
Les framboises sont perchees sur le tabouret de mon grand-pere.
Cette phrase commence par une majuscule.
Cette phrase se termine par un point.
```

Comment faire pour saisir et manier des séquences de caractères de la sorte ?

1.7.2 Définition

Une **chaîne de caractères** est un **tableau** de **char** contenant un **caractère nul**. Le caractère nul a **0** pour code **ASCII** et s'écrit `'\0'`. Les valeurs significatives de la chaîne de caractères sont toutes celles placées avant le caractère nul. On remarque donc que si le caractère nul est en première position, on a une chaine de caractères vide.

Par exemple, la phrase "Toto" sera codée de la sorte :

'T'	'o'	't'	'o'	0	...

Prenez bien note du fait que le dernier caractère de la chaine est suivi d'un caractère nul.

1.7.3 Déclaration

Comme une chaîne de caractères est un tableau de **char**, on le déclare :

```
char <nom_chaine>[<taille_chaine>];
```

Par exemple, on déclare une chaine `c` de 200 caractères de la sorte :

```
char c[200];
```

Attention ! Le nombre maximal de lettres qu'il sera possible de placer dans `c` ne sera certainement pas 200 mais 199, car il faut placer après le dernier caractère de la chaîne un caractère nul !

1.7.4 Initialisation

On initialise une chaîne à la déclaration, et **seulement** à la déclaration de la sorte :

```
char <nom_chaine>[<taille_chaine>] = <valeur_initialisation>;
```

Où la valeur d'initiallisation contient la juxtaposition de caractères formant la chaîne entourée de guillemets (double quotes). Par exemple,

```
char c[50] = "Toto";
```

Cette instruction déclare une chaîne de caratères `c` initialisée à `"Toto"`. Les 5 premiers éléments du tableau seront occupés par les 4 caractères de la chaîne ainsi que par le caratère nul, les autres contiendront des valeurs non significatives. Observez bien l'exemple suivant :

```
char c[4] = "Toto";
```

Cette déclaration engendrera un `warning` à la compilation et probablement une erreur à l'exécution car l'affectation du caractère nul à la 5-ème position du tableau donnera lieu à un débordement d'indice.

1.7.5 Accès aux éléments

Du fait qu'une chaine de caractère est un tableau, il est aisé d'en isoler un élément. Ainsi `c[i]` est le `i`+1-ème élément de la chaîne `c`. On teste donc si le premier caratère de `c` est une majuscule de la sorte :

```
if (c[0] >= 'A' && c[0] <= 'Z')
  printf("Cette phrase commence par une majuscule.\n");
else
  printf("Cette phrase ne commence pas par une majuscule.\n");
```

Cette propriété permet aussi d'afficher une chaîne caractère par caractère :

```
int i = 0;
while(c[i] != 0)
  printf("%c", c[i++]);
```

Notez que le corps de la boucle **while** est itéré jusqu'à ce que le caractère nul soit rencontré. **Il est donc impératif que votre chaîne se termine par le caractère nul** et que **le caractère nul se trouve dans la plage d'indices du tableau**. Est-ce que le code suivant est correct ?

```
char c[26];
int i;
for(i = 0 ; i < 26, i++)
  c[i] = 'a' + i;
```

Si la question vous est posée, vous pouvez présumer que ce code n'est pas correct. Le fait que chaque élément du tableau contienne un caractère non nul ne peut que corroborer cette présomption... Si l'on souhaite placer l'alphabet dans une chaîne, on procède de la sorte :

```
char c[27];
int i;
for(i = 0 ; i < 26, i++)
  c[i] = 'a' + i;
c[26] = 0;
```

Notez bien que le tableau contient 27 éléments si l'on compte le caractère nul, et que celui-ci est placé à la fin du tableau juste après la boucle.

1.7.6 Affichage

Nous avons vu qu'il était possible d'utiliser le fait qu'une chaîne est un tableau pour l'afficher. Il existe une méthode plus simple, en utilisant `printf` avec la chaîne de format `"%s"`. Par contre soyez attentifs au fait que si votre chaîne ne contient pas de caractère nul ou que le caractère nul se trouve en dehors de la plage d'indice de votre chaîne, il faudra vous attendre aux pires horreurs à l'exécution ! Dans le code donné en exemple nous pouvons donc écrire l'instruction d'affichage de la chaîne saisie par l'utilisateur :

```
printf("Vous avez saisi :\n%s", c);
```

1.7.7 Saisie

C'est maintenant que les choses se corsent, il faut être très attentif lors des saisies : tout **débordement d'indice** et/ou **absence de caratère nul** peut donner lieu à des bugs très difficiles à trouver ! La plus grande vigilance est donc de mise. Beaucoup d'amateurs utilisent des fonctions comme `gets`. Par exemple,

```
#include<stdio.h>

#define N 20

int main()
{
```

```
  char chaine[N];
  int i;
  printf("Saisissez une phrase :\n");
//  gets(chaine);
  for (i = 0 ; chaine[i] != 0 ; i++)
    printf("chaine[%d] = %c (code ASCII : %d)\n", i, chaine[i], chaine[i]);
  printf("chaine[%d] = %c (code ASCII : %d)\n", i, chaine[i], chaine[i]);
  return 0;
}
```

Tout d'abord compilons ce programme :

```
[klaus@Isengard chaines]$ gcc -Wall mauvaiseSaisie.c -o mauvaiseSaisie.c
/home/klaus/tmp/ccyKd0hf.o: In function `main':
mauvaiseSaisie.c:(.text+0x24): warning: the `gets' function is
dangerous and should not be used.
```

La première réaction du compilateur est une insulte. A-t-il raison ? Testons ce programme :

```
[klaus@Isengard chaines]$ ./mauvaiseSaisie
Saisissez une phrase :
Les framboises sont perchees sur le tabouret de mon grand pere.
chaine[0] = L (code ASCII : 76)
chaine[1] = e (code ASCII : 101)
chaine[2] = s (code ASCII : 115)
chaine[3] =  (code ASCII : 32)
chaine[4] = f (code ASCII : 102)
chaine[5] = r (code ASCII : 114)
chaine[6] = a (code ASCII : 97)
chaine[7] = m (code ASCII : 109)
chaine[8] = b (code ASCII : 98)
chaine[9] = o (code ASCII : 111)
chaine[10] = i (code ASCII : 105)
chaine[11] = s (code ASCII : 115)
chaine[12] = e (code ASCII : 101)
chaine[13] = s (code ASCII : 115)
chaine[14] =  (code ASCII : 32)
chaine[15] = s (code ASCII : 115)
chaine[16] = o (code ASCII : 111)
chaine[17] = n (code ASCII : 110)
chaine[18] = t (code ASCII : 116)
chaine[19] =  (code ASCII : 32)
chaine[20] =  (code ASCII : 20)
chaine[21] =  (code ASCII : 0)
Erreur de segmentation
```

Que se passe-t-il ? **Des horreurs !** La fonction `gets` est la pire des choses qui puisse arriver à un programme C ! Ne l'utilisez sous aucun prétexte ! Maintenant, nous allons envisager une façon propre de saisir une chaîne de caractère : `fgets`. La syntaxe est la suivante :

```
  fgets(<chaine>, <taille>, stdin);
```

La taille de la chaîne saisie est limitée par <`taille`>, caractère nul compris. Le résultat est placé dans <`chaine`>. Tous les caractères supplémentaires saisis par l'utilisateur ne sont pas placés dans <`chaine`>, seuls les *taille* − 1 premiers caractères sont récupérés par `fgets`. Nous saisirons donc la phrase de notre programme de la sorte :

```
  fgets(c, 200, stdin);
```

1.7.8 Problèmes liés à la saisie bufferisée

Lorsque vous saisissez des caractères avec `fgets`, les caractères ignorés ne sont pas éliminés du buffer. Cela signifie qu'ils iront parasiter la prochaine saisie. Il convient donc tout d'abord de vérifier s'il reste des caractères dans buffer et de les éliminer si nécessaire. Commençons par le premier point : comment s'assurer que tous les caractères saisis ont été lus ? La réponse est très simple : grâce au caractère d'échappement `'\n'`. La saisie est validée par le caratère d'échappement, il s'agit donc toujours du dernier caractère saisi. Le buffer a donc été entirèrement lu si caractère d'échappement a été lu. Si ce caractère a été lu, alors il figure juste avant le caractère nul dans la chaîne. On le repère donc de la sorte :

```
while(c[i] != 0)
  i++;
if (c[i-1] != '\n')
  printf("La saisie est incomplete");
else
  printf("La saisie est complete");
```

Si la chaîne `c` a été saisie correctement, aucun risque de débordement d'indice ou de bouclage infini ne peut se présenter. La boucle au début de l'extrait recherche le caractère nul dans la chaîne. On sort de la boucle quand `c[i]` $== 0$, donc l'indice du caratère nul est `i`. Si l'avant dernier caractère de la chaîne, donc celui d'indice $(i-1)$ est le caractère d'échappement, alors la saisie a été complète et le buffer est vide. Sinon, c'est que des caractères ont été laissés dans le buffer par `fgets`. Il convient donc, si l'on souhaite effectuer d'autres saisies par la suite, de vider le buffer. `getchar` est une instruction retournant un caractère lu dans le buffer. Si le buffer est vide, `getchar` attend une saisie de l'utilisateur **non validée par le caractère d'échappement !** On vide donc le buffer de la sorte :

```
while(getchar() != '\n');
```

Le point-virgule à la fin de l'instruction indique au compilateur que la boucle n'a pas de corps. Cette boucle lit le buffer jusqu'à ce que le caractère d'échappement ait été lu. Pour davantage de lisibilité, on définit une macro-instruction vidant le buffer :

```
#define CLEAR_BUFFER while(getchar() != '\n')
```

Vérifiez bien avant de lancer cette instruction que le buffer n'est pas vide, sinon le programme bouclera jusqu'à ce que l'utilisateur ait saisi un caractère d'échappement.

1.7.9 La bibliothèque `string.h`

Cette bibliothèque propose des fonctions de maniement de chaînes de caractères, à savoir :

- `strcmp` : comparer deux chaînes.
- `strlen` : longueur d'une chaîne de caractère
- `strsubs` : rechercher une sous-chaîne
- `strcat` : concaténer deux chaînes
- `strcpy` : copier une chaîne

Il vous est conseillé d'examiner de quelle façon fonctionnent ces fonctions, et comment elles gèrent le caractère nul.

1.7.10 Exemple

```
#include<stdio.h>

#define N 10
#define CLEAR_BUFFER while(getchar() != '\n')

int main()
{
  char chaine[N];
  int i;
  printf("Saisissez une phrase :\n");
```

```
  fgets(chaine, N, stdin);
  i = 0;
  while(chaine[i] != 0)
    i++;
  if (i > 0 && chaine[i-1] != '\n')
    CLEAR_BUFFER;
  else
    chaine[i-1] = 0;
  printf("Vous avez saisi :\n%s\n", chaine);
  if (chaine[0] >= 'A' && chaine[0] <= 'Z')
    printf("Cette phrase commence par une majuscule.\n");
  else
    printf("Cette phrase ne commence pas par une majuscule.\n");
  i = 0;
  while(chaine[i] != 0 && chaine[i] != '.')
    i++;
  if (chaine[i] == '.')
    if (chaine[i+1] == 0)
      printf("Cette phrase se termine par un point.\n");
    else
      printf("Vous avez saisi plus d'une phrase.\n");
  else
    printf("Cette phrase ne se termine pas par un point.\n");
  return 0;
}
```

1.8 Fonctions

1.8.1 Les procédures

Une **procédure** est un **ensemble d'instructions** portant un **nom**. Pour définir une procédure, on utilise la syntaxe :

```
void nomprocedure()
{
  /*
        instructions
  */
}
```

Une procédure est une nouvelle instruction, il suffit pour l'exécuter d'utiliser son nom. Par exemple, pour **exécuter** (on dit aussi **appeler** ou **invoquer**) une procédure appelée *pr*, il suffit d'écrire *pr*();. Les deux programmes suivants font la même chose. Le premier est écrit sans procédure :

```
#include<stdio.h>

int main()
{
  printf("Bonjour\n");
  return 0;
}
```

Et dans le deuxième, le `printf` est placé dans la procédure `afficheBonjour` et cette procédure est appelée depuis le `main`.

```
#include<stdio.h>

void afficheBonjour()
{
  printf("Bonjour\n");
}

int main()
{
  afficheBonjour();
  return 0;
}
```

Vous pouvez définir autant de procédures que vous le voulez et vous pouvez appeler des procédures depuis des procédures :

```
#include<stdio.h>

void afficheBonjour()
{
  printf("Bonjour\n");
}

void afficheUn()
{
  printf("1\n");
}

void afficheDeux()
{
  printf("2\n");
}
```

```
void afficheUnEtDeux()
{
  afficheUn();
  afficheDeux();
}

void afficheAuRevoir()
{
  printf("Au revoir\n");
}

int main()
{
  afficheBonjour();
  afficheUnEtDeux();
  afficheAuRevoir();
  return 0;
}
```

Ce programme affiche :

```
Bonjour
1
2
Au revoir
```

Regardez bien le programme suivant et essayez de déterminer ce qu'il affiche.

```
#include<stdio.h>

void procedure1()
{
  printf("debut procedure 1\n");
  printf("fin procedure 1\n");
}

void procedure2()
{
  printf("debut procedure 2\n");
  procedure1();
  printf("fin procedure 2\n");
}

void procedure3()
{
  printf("debut procedure 3\n");
  procedure1();
  procedure2();
  printf("fin procedure 3\n");
}

int main()
{
  printf("debut main\n");
  procedure2();
  procedure3();
  printf("fin main\n");
  return 0;
}
```

La réponse est

```
debut main
debut procedure 2
debut procedure 1
fin procedure 1
fin procedure 2
debut procedure 3
debut procedure 1
fin procedure 1
debut procedure 2
debut procedure 1
fin procedure 1
fin procedure 2
fin procedure 3
fin main
```

Vous remarquez au passage que `main` est aussi une procédure. `main` est exécutée automatiquement au lancement du programme.

1.8.2 Variables locales

Une procédure est un bloc d'instructions et est sujette aux mêmes règles que `main`. Il donc possible de déclarer des variables :

```c
void nomprocedure()
{
  /*
        declaration de variables
  */
  /*
        instructions
  */
}
```

Attention, ces variables ne sont accessibles que dans le corps de la procédure, cela signifie qu'elle naissent au moment de leur déclaration et qu'elles sont détruites une fois la dernière instruction de la procédure exécutée. C'est pour cela qu'on les apelle des **variables locales**. Une variable locale n'est **visible** qu'entre sa déclaration et l'accolade fermant la procédure. Par exemple, ce code est illégal :

```c
#include<stdio.h>

/*
void maProcedure()
{
  char a = b;
}


int main()
{
  char b = 'k';
  printf("%c, %c\n", a, b);
  return 0;
}
*/
```

En effet, la variable b est déclarée dans la procédure `main`, et n'est donc visible que dans cette même procédure. Son utilisation dans `maProcedure` est donc impossible. De même, la variable a est déclarée dans `maProcedure`, elle n'est pas visible dans le `main`. Voici un exemple d'utilisation de variables locales :

```c
#include<stdio.h>

void unADix()
{
  int i;
  for(i = 1 ; i <= 10 ; i++ )
    printf("%d\n", i);
}


int main()
{
  unADix();
  return 0;
}
```

1.8.3 Passage de paramètres

Il est possible que la valeur d'une variable locale d'une procédure ne soit connue qu'au moment de l'appel de la procédure. Considérons le programme suivant :

```c
int main()
{
  int i;
  printf("Veuillez saisir un entier : ");
  scanf(''%d'', &i);
  /*
  Appel d'une procedure affichant la valeur de i.
  */
  return 0;
}
```

Comment définir et invoquer une procédure `afficheInt` permettant d'afficher cet entier saisi par l'utilisateur ? Vous conviendrez que la procédure suivante ne passera pas la compilation

```c
void afficheInt()
{
  printf(''%d'', i);
}
```

En effet, la variable i est déclarée dans le `main`, elle n'est donc pas visible dans `afficheInt`. Pour y remédier, on définit `afficheInt` de la sorte :

```c
void afficheInt(int i)
{
  printf(''%d'', i);
}
```

i est alors appelé un **paramètre**, il s'agit d'une variable dont la valeur sera précisée lors de l'appel de la procédure. On peut aussi considérer que i est une valeur inconnue, et qu'elle est **initialisée** lors de l'invocation de la procédure. Pour initialiser la valeur d'un paramètre, on place cette valeur entre les parenthèses lors de l'appel de la procédure, par exemple : `afficheInt`(4) lance l'exécution de la procédure `afficheInt` en initialisant la valeur de i à 4. On dit aussi que l'on **passe en paramètre** la valeur 4. La version correcte de notre programme est :

```c
#include<stdio.h>

void afficheInt(int i)
{
  printf("%d", i);
```

```
}

int main()
{
  int i;
  printf("Veuillez saisir un entier : ");
  scanf("%d", &i);
  afficheInt(i);
  printf("\n");
  return 0;
}
```

Attention, notez bien que le i de `afficheInt` et le i du `main` sont **deux variables différentes**, la seule chose qui les lie vient du fait que l'instruction `afficheInt(i)` initialise le i de `afficheInt` à la valeur du i du `main`. Il serait tout à fait possible d'écrire :

```
#include<stdio.h>

void afficheInt(int j)
{
  printf("%d", j);
}

int main()
{
  int i;
  printf("Veuillez saisir un entier : ");
  scanf("%d", &i);
  afficheInt(i);
  printf("\n");
  return 0;
}
```

Dans cette nouvelle version, l'instruction `afficheInt(i)` initialise le j de `afficheInt` à la valeur du i du `main`.

Il est possible de passer plusieurs valeurs en paramètre. Par exemple, la procédure suivante affiche la somme des deux valeurs passées en paramètre :

```
void afficheSomme(int a, int b)
{
  printf(''%d'', a + b);
}
```

L'invocation d'une telle procédure se fait en initialisant les paramètres dans le même ordre et en séparant les valeurs par des virgules, par exemple `afficheSomme`(3, 4) invoque `afficheSomme` en initialisant a à 3 et b à 4. Vous devez intialiser tous les paramètres et vous devez placer les valeurs dans l'ordre. Récapitulons :

```
#include<stdio.h>

void afficheSomme(int a, int b)
{
  printf("%d", a + b);
}

int main()
{
  int i, j;
  printf("Veuillez saisir deux entiers :\na = ");
  scanf("%d", &i);
  printf("b = ");
```

```
  scanf("%d", &j);
  printf("a + b = ");
  afficheSomme(i, j);
  printf("\n");
  return 0;
}
```

La procédure ci-avant s'exécute de la façon suivante :

```
Veuillez saisir deux entiers :
a = 3
b = 5
a + b = 8
```

Lors de l'appel `afficheInt(r)` de la procédure `void afficheInt(int i)`, r est le **paramètre effectif** et i le **paramètre formel**. Notez bien que i et r sont deux variables distinctes. Par exemple, qu'affiche le programme suivant ?

```
#include<stdio.h>

void incr(int v)
{
  v++;
}

int main()
{
  int i;
  i = 6;
  incr(i);
  printf("%d\n", i);
  return 0;
}
```

La variable v est initialisée à la valeur de i, mais i et v sont deux variables différentes. Modifier l'une n'affecte pas l'autre.

1.8.4 Les fonctions

Le principe

Nous avons vu qu'un sous-programme appelant peut communiquer des valeurs au sous-programme appelé. Mais est-il possible pour un sous-programme appelé de communiquer une valeur au sous-programme appelant ? La réponse est oui. Une **fonction** est un sous-programme qui communique une valeur au sous-programme appelant. Cette valeur s'appelle **valeur de retour**, ou **valeur retournée**.

Invocation

La syntaxe pour appeler une fonction est :

```
v = nomfonction(parametres);
```

L'instruction ci-dessus place dans la variable v la valeur retournée par la fonction `nomfonction` quand lui passe les paramètres `parametres`. Nous allons plus loin dans ce cours définir une fonction `carre` qui retourne le carré de valeur qui lui est passée en paramètre, alors l'instruction

```
v = carre(2);
```

placera dans v le carré de 2, à savoir 4. On définira aussi une fonction `somme` qui retourne la somme de ses paramètres, on placera donc la valeur $2 + 3$ dans v avec l'instruction

```
v = somme(2, 3);
```

Définition

On définit une fonction avec la syntaxe suivante :

```
typeValeurDeRetour nomFonction(listeParametres)
{
}
```

La fonction `carre` sera donc définie comme suit :

```
int carre(int i)
{
  /*
        instructions
  */
}
```

Une fonction ressemble beaucoup à une procédure. Vous remarquez que **void** est remplacé par **int**, **void** signifie aucune type de retour, une procédure est donc une fonction qui ne retourne rien. Un **int** est adapté pour représenter le carré d'un autre **int**, j'ai donc choisi comme **type de retour** le type **int**. Nous définirons la fonction `somme` comme suit :

```
int somme(int a, int b)
{
  /*
        instructions
  */
}
```

L'instruction servant à retourner une valeur est **return**. Cette instruction interrompt l'exécution de la fonction et retourne la valeur placée immédiatement après. Par exemple, la fonction suivante retourne toujours la valeur 1.

```
int un()
{
  return 1;
}
```

Lorsque l'on invoque cette fonction, par exemple

```
v = un();
```

La valeur 1, qui est retournée par `un` est affectée à v. On définit une fonction qui retourne le successeur de son paramètre :

```
int successeur(int i)
{
  return i + 1;
}
```

Cette fonction, si on lui passe la valeur 5 en paramètre, retourne 6. Par exemple, l'instruction

```
v = successeur(5);
```

affecte à v la valeur 6. Construisons maintenant nos deux fonctions :

```
int carre(int i)
{
  return i * i ;
}

int somme(int a, int b)
{
  return a + b ;
}
```

Vous noterez qu'une fonction ne peut pas retourner un tableau, **une fonction ne peut retourner que des valeurs scalaires**. Vous comprendrez pourquoi en étudiant les pointeurs.

1.8.5 Passages de paramètre par référence

En C, lorsque vous invoquez une fonction, toutes les valeurs des paramètres effectifs sont recopiés dans les paramètres formels. On dit dans ce cas que le passage de paramètre se fait **par valeur**. Vous ne pouvez donc, *a priori*, communiquer qu'une seule valeur au programme appelant. Effectivement :

— seule la valeur de retour vous permettra de commnuniquer une valeur au programme appelant.
— une fonction ne peut retourner que des valeurs scalaires.

Lorsque vous passez un tableau en paramètre, la valeur qui est recopiée dans le paramètre formel est l'adresse de ce tableau (l'adresse est une valeur scalaire). Par conséquent toute modification effectuée sur les éléments d'un tableau dont l'adresse est passée en paramètre par valeur sera repercutée sur le paramètre effectif (i.e. le tableau d'origine). Lorsque les modifications faites sur un paramètre formel dans un sous-programme sont repércutées sur le paramètre effectif, on a alors un **passage de paramètre par référence**. Nous retiendrons donc les trois règles d'or suivantes :

— Les variables scalaires se passent en paramètre par valeur
— Les variables non scalaires se passent en paramètre par référence
— Une fonction ne peut retourner que des valeurs scalaires

1.9 Structures

1.9.1 Définition

Nous avons utilisé des tableaux pour désigner, avec un nom unique, un ensemble de variables. Par exemple, un tableau T à n éléments est un ensemble n de variables désignées par la lettre T. Dans un tableau, les variables doivent être de type homogène, cela signifiant qu'il n'est pas possible de juxtaposer des *char* et des *int* dans un tableau. Lorsque l'on souhaite faire cohabiter dans une variable non scalaire des types hétérogènes, on utilise des structures.
Une **structure**, appelé enregistrement dans d'autres langages, est une variable contenant plusieurs variables, appelées **champs**. Si une structure t contient un *char* et un *int*, chacun de ces champs portera un nom, par exemple i et c. Dans ce cas, $t.c$ désignera le *char* de t et $t.i$ l'int de t.

1.9.2 Déclaration

Pour créer un type structuré, on utilise la syntaxe suivante :

```
struct nomdutype
{
typechamp_1 nomchamp_1;
typechamp_2 nomchamp_2;
...
typechamp_n nomchamp_n;
};
```

On précise donc le nom de ce type structuré, et entre les accolades, la liste des champs avec pour chacun d'eux son type. Vous remarquez que le nombre de champs est fixé d'avance. On n'accède pas à un champ avec un indice mais avec un nom. Considérons par exemple le type structuré suivant :

```
struct point
{
  double abs;
  double ord;
};
```

Ce type permet de représenter un point dans R^2, avec respectivement uns abscisse et une ordonnée. *struct point* est maintenant un type, il devient possible de déclarer un point p comme tout autre variable :

```
struct point p;
```

1.9.3 Accès aux champs

On accède aux champs d'une variable structurée à l'aide de la notation pointée

```
nomvariable.nomduchamp
```

Ainsi, le champ *ord* de notre variable p sera accessible avec $p.ord$ et le champ *abs* de notre variable p sera accessible avec $p.abs$. Voici un exemple de programme illustrant ce principe :

```
#include<stdio.h>

struct point
{
  double abs;
  double ord;
};


main()
```

```
{
  struct point p;
  p.ord = 2;
  p.abs = p.ord + 1;
  printf("p = (%f, %f)\n", p.abs, p.ord);
}
```

Ce programme affiche :

```
p = (3.000000, 2.000000)
```

Attention, l'opérateur d'accès au champ . est prioritaire sur tous les autres opérateurs unaires, binaires et ternaires ! Il faudra vous en rappeler quand on étudiera les listes chaînées.

1.9.4 Typedef

On se débarasse du mot clé *struct* en renommant le type, on utilisera par exemple la syntaxe suivante :

```
#include<stdio.h>

typedef struct point
{
  double abs;
  double ord;
}point;


main()
{
  point p;
  p.ord = 2;
  p.abs = p.ord + 1;
  printf("p = (%f, %f)\n", p.abs, p.ord);
}
```

point est donc le nom de ce type structuré.

1.9.5 Tableaux de structures

Rien de nous empêche de créer des tableaux de structures, par exemple :

```
#include<stdio.h>

#define N 10

typedef struct point
{
  double abs;
  double ord;
}point;


main()
{
  point p[N];
  int i;
  p[0].ord = 0;
  p[0].abs = 1;
  for(i = 1 ; i < N ; i++)
    {
```

```
      p[i].ord = p[i - 1].ord + 1.;
      p[i].abs = p[i - 1].abs + 2.;
    }
  for(i = 0 ; i < N ; i++)
    {
      printf("p[%d] = (%f, %f)\n", i, p[i].abs, p[i].ord);
    }
}
```

Ce programme affiche :

```
p[0] = (1.000000, 0.000000)
p[1] = (3.000000, 1.000000)
p[2] = (5.000000, 2.000000)
p[3] = (7.000000, 3.000000)
p[4] = (9.000000, 4.000000)
p[5] = (11.000000, 5.000000)
p[6] = (13.000000, 6.000000)
p[7] = (15.000000, 7.000000)
p[8] = (17.000000, 8.000000)
p[9] = (19.000000, 9.000000)
```

1.9.6 Structures et fonctions

Lorsqu'on les passe en paramètre, les structures se comportent comme des variables scalaires, cela signifie qu'on ne peut les passer en paramètre que par valeur. Par contre, un tableau de structures est nécessairement passé en paramètre par référence. Réecrivons le programme précédent avec des sous-programmes :

```
#include<stdio.h>

#define N 10

typedef struct point
{
  double abs;
  double ord;
}point;

void initTableauPoints(point p[], int n)
{
  int i;
  p[0].ord = 0;
  p[0].abs = 1;
  for(i = 1 ; i < n ; i++)
    {
      p[i].ord = p[i - 1].ord + 1.;
      p[i].abs = p[i - 1].abs + 2.;
    }
}

void affichePoint(point p)
{
  printf("(%f, %f)", p.abs, p.ord);
}

void afficheTableauPoints(point p[], int n)
{
  int i;
 for(i = 0 ; i < n ; i++)
```

```
    {
      printf("p[%d] = ", i);
      affichePoint(p[i]);
      printf("\n");
    }
}

main()
{
  point p[N];
  initTableauPoints(p, N);
  afficheTableauPoints(p, N);
}
```

Comme une structure se comporte comme une variable scalaire, il est possible de retourner une structure dans une fonction, il est donc possible de modifier le programme ci-avant de la sorte :

```
#include<stdio.h>

#define N 10

typedef struct point
{
  double abs;
  double ord;
}point;

point nextPoint(point previous)
{
  point result;
  result.ord = previous.ord + 1.;
  result.abs = previous.abs + 2.;
  return result;
}

void initTableauPoints(point p[], int n)
{
  int i;
  p[0].ord = 0;
  p[0].abs = 1;
  for(i = 1 ; i < n ; i++)
    p[i] = nextPoint(p[i - 1]);
}

void affichePoint(point p)
{
  printf("(%f, %f)", p.abs, p.ord);
}

void afficheTableauPoints(point p[], int n)
{
  int i;
 for(i = 0 ; i < n ; i++)
    {
      printf("p[%d] = ", i);
      affichePoint(p[i]);
      printf("\n");
    }
}
```

```
main()
{
  point p[N];
  initTableauPoints(p, N);
  afficheTableauPoints(p, N);
}
```

1.10 Pointeurs

La notion de pointeur est très importante en C, elle va vous permettre de comprendre le fonctionnement d'un programme, de programmer de façon davantage propre et performante, et surtout de concevoir des programmes que vous ne pourriez pas mettre au point sans cela.

Une des premières choses à comprendre quand on programme, c'est qu'une variable est un emplacement de la mémoire dans lequel vous allez placer une valeur. En programmant, vous utilisez son **nom** pour y lire ou écrire une valeur. Mais ce qui se passe au coeur de la machine est quelque peu plus complexe, les noms que l'on donne aux variables servent à masquer cette complexité.

Pour le compilateur, une variable est un emplacement dans la mémoire, cet emplacement est identifié par une **adresse mémoire**. Une adresse est aussi une valeur, mais cette valeur sert seulement à spécifier un emplacement dans la mémoire. Lorsque vous utilisez le nom d'une variable, le compilateur le remplace par une adresse, et manipule la variable en utilisant son adresse.

Ce système est ouvert, dans le sens où vous pouvez décider d'utiliser l'adresse d'une variable au lieu d'utiliser son nom. Pour ce faire, on utilise des pointeurs.

1.10.1 Introduction

Un **pointeur** est une variable qui contient l'**adresse mémoire** d'une autre variable.

Déclaration

$T*$ est le type d'une variable contenant l'adresse mémoire d'une variable de type T. Si une variable p de type $T*$ contient l'adresse mémoire d'une variable x de type T, on dit alors que p **pointe vers** x (ou bien **sur** x). $\&x$ est l'adresse mémoire de la variable x. Exposons cela dans un exemple,

```
#include<stdio.h>

int main()
{
  int x = 3;
  int* p;
  p = &x;
  return 0;
}
```

x est de type *int*. p est de type *int*$*$, c'est à dire de type **pointeur de** *int*, p est donc faite pour contenir l'adresse mémoire d'un *int*. $\&x$ est l'adresse mémoire de la variable x, et l'affectation $p = \&x$ place l'adresse mémoire de x dans le pointeur p. A partir de cette affectation, p pointe sur x.

Affichage

La chaîne de format d'une adresse mémoire est "%X". On affiche donc une adresse mémoire (très utile pour débugger :-) comme dans l'exemple ci-dessous,

```
#include<stdio.h>

int main()
{
  int x = 3;
  int* p;
  p = &x;
  printf("p contient la valeur %X, qui n'est autre que l'adresse %X de x\n",
         p, &x);
  return 0;
```

```
}
```

Accès à la variable pointée

Le lecteur impatient se demande probablement à quoi peuvent servir toutes ces étoiles ? Quel peut bien être l'intérêt des pointeurs ?

Si p pointe sur x, alors il est possible d'accéder à la valeur de x en passant par p. Pour le compilateur, $*p$ est la variable pointée par p, cela signifie que l'on peut, pour le moment du moins, utiliser indifférement $*p$ ou x. Ce sont deux façons de se référer à la même variable, on appelle cela de l'**aliasing**. Explicitons cela sur un exemple,

```
#include<stdio.h>

int main()
{
  int x = 3;
  int* p;
  p = &x;
  printf("x = %d\n", x);
  *p = 4;
  printf("x = %d\n", x);
  return 0;
}
```

L'affectation $p = \&x$ fait pointer p sur x. A partir de ce moment, $*p$ peut être utilisé pour désigner la variable x. De ce fait, l'affectation $x = 4$ peut aussi être écrite $*p = 4$. Toutes les modifications opérées sur $*p$ seront répercutées sur la variable pointée par p. Donc ce programme affiche

```
x = 3
x = 4
```

Récapitulons

Qu'affiche, à votre avis, le programe suivant ?

```
#include<stdio.h>

int main()
{
  int x = 3;
  int y = 5;
  int* p;
  p = &x;
  printf("x = %d\n", x);
  *p = 4;
  printf("x = %d\n", x);
  p = &y;
  printf("*p = %d\n", *p);
  *p = *p + 1;
  printf("y = %d\n", y);
  return 0;
}
```

x est initialisé à 3 et y est initialisé à 5. L'affectation $p = \&x$ fait pointer p sur x, donc $*p$ et x sont deux écritures différentes de la même variable. Le premier *printf* affiche la valeur de x : plus précisément `x` $= 3$. Ensuite, l'affectation $*p = 4$ place dans la valeur pointée par p, à savoir x, la valeur 4. Donc le deuxième *printf* affiche `x` $= 4$. L'affectation $p = \&y$ fait maintenant pointer p sur y, donc la valeur de $*p$ est la valeur de la variable pointée y. le troisième *printf* affiche donc `*p` $= 5$. N'oubliez pas que comme p pointe sur y, alors $*p$ et y sont deux alias pour la même variable, de

ce fait, l'instruction $*p = *p + 1$ peut tout à fait s'écrire $y = y + 1$. Cette instruction place donc dans y la valeur 6, le quatrième *printf* affiche donc `y = 6`. Ce programme affiche donc :

```
x = 3
x = 4
*p = 5
y = 6
```

1.10.2 Tableaux

Il est possible d'aller plus loin dans l'utilisation des pointeurs en les employant pour manipuler des tableaux. Tout d'abord, éclaircissons quelques points.

Démystification (et démythification) du tableau en C

Les éléments d'un tableau sont juxtaposés dans la mémoire. Autrement dit, ils sont placés les uns à coté des autres. Sur le plan de l'adressage, cela a une conséquence fort intuitive. Sachant qu'un *int* occupe 2 octets en mémoire et que $\&T[0]$ est l'adresse mémoire du premier élément du tableau T, quelle est l'adresse mémoire de $T[1]$? La réponse est la plus simple : $\&T[0] + 2$ (ne l'écrivez jamais ainsi dans un programme, vous verrez pourquoi plus loin dans le cours...). Cela signifie que si l'on connait l'adresse d'un élément d'un tableau (le premier en l'occurrence), il devient possible de retrouver les adresses de tous les autres éléments de ce tableau.

J'ai par ailleurs, une assez mauvaise surprise pour vous. Vous utilisez sans le savoir des pointeurs depuis que vous utilisez des tableaux. Etant donné la déclaration *int* $T[50]$, vous conviendrez que les désignations $\{T[0], \ldots, T[49]\}$ permettent de se référer aux 50 éléments du tableau T. Mais vous est-il déjà arrivé, en passant un tableau en paramètre, d'écrire T sans écrire d'indice entre crochets ? Par exemple,

```c
#include<stdio.h>

void initTab(int K[], int n)
{
  int i;
  for(i = 0 ; i < n ; i++)
    K[i] = i + 1;
}

void afficheTab(int K[], int n)
{
  int i;
  for(i = 0 ; i < n ; i++)
    printf("%d\n", K[i]);
}

int main()
{
  int T[50];
  initTab(T, 50);
  afficheTab(T, 50);
  return 0;
}
```

Vous remarquez que lorsque l'on passe un tableau en paramètre à un sous-programme, on mentionne seulement son nom. En fait, le nom d'un tableau, T dans l'exemple ci-avant, est **l'adresse mémoire du premier élément de ce tableau**. Donc, T est une variable contenant une adresse mémoire, T est par conséquent un pointeur. Lorsque dans un sous-programme auquel on a passé un tableau en paramètre, on mentionne un indice, par exemple $K[i]$, le compilateur **calcule** l'adresse du i-ème élément de K pour lire ou écrire à cette adresse.

Utilisation des pointeurs

Commençons par observer l'exemple suivant :

```
#include<stdio.h>

int main()
{
  char t[10];
  char* p;
  t[0] = 'a';
  p = t;
  printf("le premier element du tableau est %c.\n", *p);
  return 0;
}
```

La variable t contient l'adresse mémoire du premier élément du tableau t. p est un pointeur de char, donc l'affectation $p = t$ place dans p l'adresse mémoire du premier élément du tableau t. Comme p pointe vers $t[0]$, on peut indifféremment utiliser $*p$ ou $t[0]$. Donc ce programme affiche

```
le premier element du tableau est a.
```

Calcul des adresses mémoire

Tout d'abord, je tiens à rappeler qu'une variable de type *char* occupe un octet en mémoire. Considérons maintenant les déclarations

$$char\ t[10]\ ;$$

et

$$char*\ p = t\ ;$$

Nous savons que si p pointe vers $t[0]$, il est donc aisé d'accéder au premier élément de t en utilisant le pointeur p. Mais comment accéder aux autres éléments de t ? Par exemple $T[1]$? Souvenez-vous que les éléments d'un tableau sont juxtaposés, dans l'ordre, dans la mémoire. Par conséquent, si p est l'adresse mémoire du premier élément du tableau t, alors $(p+1)$ est l'adresse mémoire du deuxième élément de ce tableau. Vous êtes conviendrez que $*p$ est la variable dont l'adresse mémoire est contenue dans p. Il est possible, plus généralement, d'écrire $*(p+1)$ pour désigner la variable dont l'adresse mémoire est $(p+1)$, c'est à dire $(la\ valeur\ contenue\ dans\ p)+1$. Illustrons cela dans un exemple, le programme

```
#include<stdio.h>

int main()
{
  char t[10];
  char* p;
  t[1] = 'b';
  p = t;
  printf("le deuxieme element du tableau est %c.\n", *(p+1));
  return  0;
}
```

affiche

```
le deuxième element du tableau est b.
```

En effet, $p+1$ est l'adresse mémoire du deuxième élément de t, il est donc possible d'utiliser indifféremment $*(p+1)$ et $t[1]$. Plus généralement, on peut utiliser $*(p+i)$ à la place de $t[i]$. En effet, $(p+i)$ est l'adresse du i-ème élément de t, et $*(p+i)$ est la variable dont l'adresse mémoire est $(p+i)$. Par exemple,

```c
#include<stdio.h>
#define N 26

int main()
{
  char t[N];
  char v = 'A';
  char* p;
  int i;
  p = t;

  /* initialisation du tableau*/
  for (i = 0 ; i < N ; i++)
    *(p + i) = v++;

  /* affichage du tableau*/
  for(i = 0 ; i < N ; i++)
    printf("%c ", *(p + i));

  printf("\n");
  return 0;
}
```

Ou encore, en utilisant des sous-programmes,

```c
#include<stdio.h>
#define N 26

void initTab(char* k, int n)
{
  int i;
  int v = 'A';
  for(i = 0 ; i < n ; i++, k++, v++)
     *k = v;
}

void afficheTab(char* k, int n)
{
  int i;
  for(i = 0 ; i < n ; i++, k++)
    printf("%c ", *k);
  printf("\n");
}

int main()
{
  char t[N];
  initTab(t, N);
  afficheTab(t, N);
  return 0;
}
```

Ces deux sous-programmes affichent

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Arithmétique des pointeurs

Supposons que le tableau t contienne des *int*, sachant qu'un *int* occupe 2 octets en mémoire. Est-ce que $(t + 1)$ est l'adresse du deuxième élément de t ?

Mathématiquement, la réponse est non, l'adresse du deuxième élément est $t + (la\ taille\ d'un\ int) = (t + 2)$. Etant donné un tableau p d'éléments occupant chacun n octets en mémoire, l'adresse du i-ème élément est alors $p + i \times n$.

Cependant, la pondération systématique de l'indice par la taille occupée en mémoire par chaque élément est d'une part une lourdeur dont on se passerait volontier, et d'autre part une source d'erreurs et de bugs. Pour y remédier, le compilateur prend cette partie du travail en charge,**on ne pondérera donc pas les indices** ! Cela signifie, plus explicitement, que quel que soit le type des éléments du tableau p, l'adresse mémoire du i-ème élément de p est $p + i$. On le vérifie expérimentalement en exécutant le programme suivant :

```
#include<stdio.h>
#define N 30

void initTab(int* k, int n)
{
  int i;
  *k = 1;
  for(i = 1, k++ ; i < n ; i++, k++)
     *k = *(k - 1) + 1;
}

void afficheTab(int* k, int n)
{
  int i;
  for(i = 0 ; i < n ; i++, k++)
    printf("%d ", *k);
  printf("\n");
}

int main()
{
  int t[N];
  initTab(t, N);
  afficheTab(t, N);
  return 0;
}
```

1.10.3 Allocation dynamique de la mémoire

La lecture du chapitre précédent, si vous y avez survécu, vous a probablement mené à une question que les élèves posent souvent : "Monsieur pourquoi on fait ça ?". C'est vrai ! Pourquoi on manipulerait les tableaux avec des pointeurs alors que dans les exemples que je vous ai donné, on peut le faire sans les pointeurs ? Dans la mesure où un tableau est un pointeur, on peut, même à l'intérieur d'un sous-programme auquel un tableau a été passé en paramètre, manipuler ce tableau avec des crochets. Alors dans quel cas utiliserons-nous des pointeurs pour parcourir les tableaux ?

De la même façon qu'il existe des cas dans lesquels on connait l'adresse d'une variable scalaire mais pas son nom, il existe des tableaux dont on connait l'adresse mais pas le nom.

Un problème de taille

Lorsque l'on déclare un tableau, il est obligatoire de préciser sa taille. Cela signifie que la **taille d'un tableau doit être connue à la compilation**. Alors que faire si on ne connait pas cette taille ? La seule solution qui se présente pour le moment est le surdimensionnement, on donne au tableau une taille très (trop) élevée de sorte qu'aucun débordement ne se produise.

Nous aimerions procéder autrement, c'est à dire **préciser la dimension du tableau au moment de l'exécution**. Nous allons pour cela rappeler quelques principes. Lors de la déclaration d'un tableau t, un espace mémoire

alloué au stockage de la variable t, c'est à dire la variable qui contient l'adresse mémoire du premier élément du tableau. Et un autre espace mémoire est alloué au stockage des éléments du tableau. Il y a donc deux zones mémoires utilisées.

La fonction *malloc*

Lorsque vous déclarez un pointeur p, vous allouez un espace mémoire pour y stocker l'adresse mémoire d'un entier. Et p, jusqu'à ce qu'on l'initialise, contient n'importe quoi. Vous pouvez ensuite faire pointer p sur l'adresse mémoire que vous voulez (choisissez de préférence une zone contenant un *int*...). Soit cette adresse est celle d'une variable qui existe déjà. Soit cette adresse est celle d'un espace mémoire créée spécialement pour l'occasion.

Vous pouvez demander au système d'exploitation de l'espace mémoire pour y stocker des valeurs. La fonction qui permet de réserver n octets est $malloc(n)$. Si vous écrivez $malloc(10)$, l'OS réserve 10 octets, cela s'appelle une **allocation dynamique**, c'est à dire une allocation de la mémoire au cours de l'exécution. Si vous voulez réserver de l'espace mémoire pour stocker un *int* par exemple, il suffit d'appeler $malloc(2)$, car un *int* occupe 2 octets en mémoire.

En même temps, c'est bien joli de réserver de l'espace mémoire, mais ça ne sert pas à grand chose si on ne sait pas où il se trouve ! C'est pour ça que *malloc* est une fonction. *malloc* retourne l'adresse mémoire du premier octet de la zone réservée. Par conséquent, si vous voulez créer un *int*, il convient d'exécuter l'instruction : $p = malloc(2)$ où p est de type $int*$. Le *malloc* réserve deux octets, et retourne l'adresse mémoire de la zone allouée. Cette affectation place donc dans p l'adresse mémoire du *int* nouvellement créé.

Cependant, l'instruction $p = malloc(2)$ ne peut pas passer la compilation. Le compilateur vous dira que les types $void*$ et $int*$ sont incompatibles (`incompatible types in assignement`). Pour votre culture générale, $void*$ est le type "adresse mémoire" en C. Alors que $int*$ est le type "adresse mémoire d'un *int*". Il faut donc dire au compilateur que vous savez ce que vous faites, et que vous êtes sûr que c'est bien un *int* que vous allez mettre dans la variable pointée. Pour ce faire, il convient d'effctuer ce que l'on appelle un **cast**, en ajoutant, juste après l'opérateur d'affectation, le type de la variable se situant à gauche de l'affectation entre parenthèses. Dans l'exemple ci-avant, cela donne : $int * p = (int*)malloc(2)$.

Voici un exemple illustrant l'utilisation de *malloc*.

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

int main()
{
  int* p;
  p = (int*)malloc(sizeof(int));
  *p = 28;
  printf("%d\n", *p);
  return 0;
}
```

Vous remarquez que nous sommes bien dans un cas où l'on connait l'adresse d'une variable mais pas son nom. Le seul moyen de manier la variable allouée dynamiquement est d'utiliser un pointeur.

La fonction *free*

Lorsque que l'on effectue une allocation dynamique, l'espace réservé ne peut pas être alloué pour une autre variable. Une fois que vous n'en avez plus besoin, vous devez le libérer explicitement si vous souhaitez qu'une autre variable puisse y être stockée. La fonction de libération de la mémoire est $free$. $free(v)$ où v est une variable contenant l'adresse mémoire de la zone à libérer. A chaque fois que vous allouez une zone mémoire, vous devez la libérer ! Un exemple classique d'utilisation est :

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

```c
int main()
{
  int* p;
  p = (int*)malloc(sizeof(int));
  *p = 28;
  printf("%d\n", *p);
  free(p);
  return 0;
}
```

Notez bien que la variable p, qui a été allouée au début du *main*, a été libéré par le $free(p)$.

La valeur $NULL$

La pointeur p qui ne pointe aucune adresse a la valeur $NULL$. Attention, il n'est pas nécessairement initialisé à $NULL$, $NULL$ est la valeur que, conventionnellement, on décide de donner à p s'il ne pointe sur aucune zone mémoire valide. Par exemple, la fonction *malloc* retourne $NULL$ si aucune zone mémoire adéquate n'est trouvée. Il convient, **à chaque** *malloc*, de vérifier si la valeur retournée par *malloc* est différente de $NULL$. Par exemple,

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

int main()
{
  int* p;
  p = (int*)malloc(sizeof(int));
  if(p == NULL)
    exit(0);
  *p = 28;
  printf("%d\n", *p);
  free(p);
  return 0;
}
```

Vous remarquez que le test de non nullité de la valeur retournée par *malloc* est effectué immédiatement après l'allocation dynamique. Vous ne devez jamais utiliser un pointeur sans avoir vérifié sa validité, autrement dit, sa non-nullité. Un pointeur contenant une adresse non valide est appelé un **pointeur fou**. Vous devrez, dans votre vie de programmeur, les traquer avec hargne !

L'allocation dynamique d'un tableau

Lors de l'allocation dynamique d'un tableau, il est nécessaire de déterminer la taille mémoire de la zone de la zone à occuper. Par exemple, si vous souhaitez allouer dynamiquement un tableau de 10 variables de type *char*. Il suffit d'exécuter l'instruction $malloc(10)$, car un tableau de 10 *char* occupe 10 octets en mémoire. Si par contre, vous souhaitez allouer dynamiquement un tableau de 10 *int*, il conviendra d'exécuter $malloc(20)$, car chaque *int* occupe 2 octets en mémoire.

Pour se simplifier la vie, le compilateur met à notre disposition la fonction *sizeof*, qui nous permet de calculer la place prise en mémoire par la variable d'un type donné. Par exemple, soit T un type, la valeur $sizeof(T)$ est la taille prise en mémoire par une variable de type T. Si par exemple on souhaite allouer dynamiquement un *int*, il convient d'exécuter l'instruction $malloc(sizeof(int))$. Attention, *sizeof* prend en paramètre un **type** !

Si on souhaite allouer dynamiquement un tableau de n variables de type T, on exécute l'instruction $malloc(n * sizeof(T))$. Par exemple, pour allouer un tableau de 10 *int*, on exécute $malloc(10 * sizeof(int))$. Voici une variante du programme d'un programme précédent :

```c
#include<stdio.h>
#include<stdlib.h>
#define N 26
```

```
char* initTab(int n)
{
  char* k;
  int i;
  char value = 'a';
  k = (char*)malloc(n*sizeof(char));
  if (k == NULL)
    return NULL;
  for(i = 0 ; i < n ; i++, value++)
     *(k + i) = value;
  return k;
}

void afficheTab(char* k, int n)
{
  int i;
  for(i = 0 ; i < n ; i++, k++)
    printf("%c ", *k);
  printf("\n");
}

int main()
{
  char* p = initTab(N);
  if (p == NULL)
    return -1;
  afficheTab(p, N);
  free(p);
  return 0;
}
```

1.10.4 Passage de paramètres par référence

J'ai dit tout à l'heure : "Pour le compilateur, $*p$ est la variable pointée par p, cela signifie que l'on peut, pour le moment du moins, utiliser indifférement $*p$ ou x". En précisant "pour le moment du moins", j'avais déjà l'intention de vous montrer des cas dans lesquels ce n'était pas possible. C'est à dire des cas dans lesquels on connait l'adresse d'une variable mais pas son nom.

Permutation de deux variables

A titre de rappel, observez attentivement le programme suivant :

```
#include<stdio.h>

void echange(int* x, int* y)
{
  int t = *x;
  *x = *y;
  *y = t;
}

int main()
{
  int a = 1;
  int b = 2;
  printf("a = %d, b = %d\n", a, b);
  echange(&a, &b);
  printf("a = %d, b = %d\n", a, b);
```

```
  return 0;
}
```

A votre avis, affiche-t-il

```
a = 1, b = 2
a = 2, b = 1
```

ou bien

```
a = 1, b = 2
a = 1, b = 2
```

Méditons quelque peu : la question que l'on se pose est "Est-ce que le sous-programme *echange* échange bien les valeurs des deux variables a et b" ? Il va de soi qu'il échange bien les valeurs des deux variables x et y, mais comme ces deux variables ne sont que des **copies** de a et b, cette permutation n'a aucun effet sur a et b. Cela signifie que la fonction *echange* ne fait rien, on aurait pu écrire à la place un sous-programme ne contenant aucune instruction, l'effet aurait été le même. Ce programme affiche donc

```
a = 1, b = 2
a = 1, b = 2
```

Remarques

Ceux dont la mémoire n'a pas été réinitialisé pendant les vacances se souviennent certainement du fait qu'il était impossible de passer en paramètre des variables scalaires par référence. J'ai menti, il existe un moyen de passer des paramètres par référence, et vous aviez des indices vous permettant de vous en douter ! Par exemple, l'instruction $scanf("\%d", \&x)$ permet de placer une valeur saisie par l'utilisateur dans x, et $scanf$ est un sous-programme... Vous conviendrez donc que **la variable x a été passée en paramètre par référence**. Autrement dit, que la valeur de x est modifiée dans le sous-programme $scanf$, donc que la variable permettant de désigner x dans le corps de ce sous-programme n'est pas une copie de x, mais la variable x elle-même, ou plutôt un **alias de la variable x**.

Vous pouvez d'ores et déjà retenir que

$$nomsousprogramme(\ldots, \&x, \ldots)$$

sert à passer en paramètre la variable x par référence. Et finalement, c'est plutôt logique, l'instruction

$$nomsousprogramme(\ldots, x, \ldots)$$

passe en paramètre la **valeur** de x, alors que

$$nomsousprogramme(\ldots, \&x, \ldots)$$

passe en paramètre l'**adresse** de x, c'est-à-dire un moyen de retrouver la variable x depuis le sous-programme et de modifier sa valeur.

Cependant, si vous écrivez $echange(\&a, \&b)$, le programme ne compilera pas... En effet, le sous-programme *echange* prend en paramètre des *int* et si vous lui envoyez des adresses mémoire à la place, le compilateur ne peut pas "comprendre" ce que vous voulez faire... Vous allez donc devoir modifier le sous-programme *echange* si vous voulez lui passer des adresses mémoire en paramètre.

Utilisation de pointeurs

On arrive à la question suivante : dans quel type de variable puis-je mettre l'adresse mémoire d'une variable de type entier ? La réponse est $int*$, un pointeur sur *int*. Observons le sous-programme suivant,

```
void echange(int* x, int* y)
{
  int t = *x;
  *x = *y;
  *y = t;
}
```

x et y ne sont pas des *int*, mais des pointeurs sur *int*. De ce fait le passage en paramètre des deux adresses $\&a$ et $\&b$ fait pointer x sur a et y sur b. Donc $*x$ est un alias de a et $*y$ est un alias de b. Nous sommes, comme décrit dans l'introduction de ce chapitre dans un cas dans lequel on connait l'adresse d'une variable, mais pas son nom : dans le sous-programme *echange*, la variable a est inconnue (si vous l'écrivez, ça ne compilera pas...), seul le pointeur $*x$ permet d'accéder à la variable a.

Il suffit donc, pour écrire un sous-programme prenant en paramètre des variables passées par référence, de les déclarer comme des pointeurs, et d'ajouter une $*$ devant à chaque utilisation.

1.10.5 Pointeurs sur fonction

1.11 Fichiers

1.11.1 Définitions

Supposons que l'on dispose d'un programme permettant saisir 10 entiers dans un tableau.

```c
#include<stdio.h>

int main()
{
  int t[10], i;
  for(i = 0; i < 10 ; i++)
    {
      printf("Saisir un entier : ");
      scanf("%d", t + i);
    }
  for(i = 0 ; i < 10 ; i++)
    printf("%d ", t[i]);
  printf("\n");
  return 0;
}
```

Le problème qui se pose est que lorsque l'on sort de ce programme, les données saisies sont perdues. Si l'on souhaite les avoir à disposition pour d'une exécution ultérieure, il convient d'utiler une mémoire persistante, c'est-à-dire qui conserve les données entre deux exécutions. On utilise pour ce faire un **fichier**. Un fichier est une mémoire stockée de façon permanente sur un disque et à laquelle on accède avec un **nom**. Les données dans un fichier se présentent de façon séquentielle, et donc se lisent ou s'écrivent du début vers la fin.

Nous survolerons dans ce cours un ensemble de fonctions de `stdio.h` permettant de manier des fichiers. Pour plus détails, il est très hautement recommandé de se reporter à la documentation.

1.11.2 Ouverture et fermeture

Pour accéder au contenu d'un fichier en lecture ou en écriture, on utilise les deux fonctions `fopen` et `fclose`.

La fonction `fopen`

`fopen` permet, comme son nom l'indique, d'ouvrir un fichier. Son prototype est `FILE *fopen(const char *path, const char *mode)` :

- `path` est une chaîne de caractère contenant le chemin (relatif ou absolu) et le nom du fichier. Si le fichier est dans le répertoire dans lequel s'exécute le programme, alors le nom du fichier suffit.
- `mode` est une chaîne de caractère contenant `"r"` pour ouvrir le fichier en mode lecture, `"w"` en mode écriture, etc.
- `FILE*` est un type permettant de référencer un fichier ouvert, `fopen` retourne `NULL` s'il est impossible d'ouvrir le fichier (par exemple si le nom est incorrect). La valeur retournée devra être placée dans une variable de type `FILE*`, c'est cette valeur qui permettra par la suite d'accéder au contenu du fichier.

La fonction `fclose`

`fclose` sert à fermer un fichier. Son prototype est `int fclose(FILE *fp)` :

- `fp` est la variable de type `FILE*` permettant de référencer le fichier à fermer.
- Cette fonction retourne 0 si la fermeture s'est bien passée. Dans le cas contraire, des indications sur l'erreur survenue sont accessibles dans des variables globales.

Utilisation

Si l'on souhaite par exemple lire dans un fichier s'appelant `"toto.txt"` :

```c
#include<stdio.h>

int main()
```

```
{
  FILE* f;
  f = fopen("toto.txt", "r");
  if (f == NULL)
    {
      printf("Erreur lors de l'ouverture du fichier toto.txt\n");
      return -1;
    }
  /*
    Lecture dans le fichier
    ...
   */
  if (fclose(f))
    {
      printf("Erreur lors de la fermeture du fichier toto.txt\n");
      return -1;
    }
  return 0;
}
```

1.11.3 Lecture et écriture

Il existe plusieurs façons de lire dans un fichier : caractère par caractère, ligne par ligne, par paquets de caractères, etc. Chaque lecture se faisant à l'aide d'une fonction appropriée. Lors d'un traitement se faisant à partir d'une lecture dans un fichier, on appelle de façon itérée une fonction de lecture faisant avancer un curseur dans un fichier jusqu'à ce que la fin du fichier soit atteinte.

L'écriture fonctionne de façon analogue, à un détail près : il est inutile d'écrire le caractère de fin de fichier, il est ajouté automatiquement lors du `fclose`.

Caractère par caractère

La fonction `int fgetc(FILE* stream)` retourne un caractère lu dans le fichier `f`. Bien que le caractère lu soit un octet, il est retourné dans un `int`. Le caractère `EOF` indique que la fin du fichier a été atteinte. Par exemple,

```
#include<stdio.h>

int main()
{
  FILE* f;
  char c;
  f = fopen("toto.txt", "r");
  if (f == NULL)
    {
      printf("Erreur lors de l'ouverture du fichier toto.txt\n");
      return -1;
    }
  while((c = fgetc(f)) != EOF)
    printf("caractere lu : %c\n", c);
  if (fclose(f))
    {
      printf("Erreur lors de la fermeture du fichier toto.txt\n");
      return -1;
    }
  return 0;
}
```

On écrit un caractère dans un fichier à l'aide de la fonction `int fputc(int c, FILE* stream)`.

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

```c
int main()
{
  FILE* f;
  char c[8] = "Toto !\n";
  int i;
  f = fopen("toto.txt", "w");
  if (f == NULL)
    {
      printf("Erreur lors de l'ouverture du fichier toto.txt\n");
      return -1;
    }
  for(i = 0 ; i < 7 ; i++)
    fputc(c[i], f);
  if (fclose(f))
    {
      printf("Erreur lors de la fermeture du fichier toto.txt\n");
      return -2;
    }
  return 0;
}
```

Par chaînes de caractères

Les deux fonctions `char *fgets(char *s, int size, FILE *stream)` et `int fputs(const char *s, FILE *stream)` permettent de lire et d'écrire des chaînes de caractères dans des fichiers, voir la documentation pour plus de détails.

Par paquets

Les deux fonctions `size\_t fread(void *ptr, size\_t size, size\_t nmemb, FILE *stream)` et `size\_t fwrite(const void *ptr, size\_t size, size\_t nmemb, FILE *stream)` sont très utiles lorsque l'on veut sauvegarder un tableau dans un fichier, ou recopier un fichier dans un tableau (voir la documentation pour plus de détails). Voici tout de même deux exemples :

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

struct personne
{
  char nom[30];
  int age;
};

int main(int argv, char** argc)
{
        FILE* f;
        struct personne repertoire[4] =
          {{"tata", 2}, {"toto", 8}, {"titi", -1}, {"tutu", 9}};
        f = fopen("toto.txt", "w");
        if (f == NULL)
          {
            printf("Impossible d'ouvrir le fichier toto.txt");
            return -1;
          }
        fwrite(repertoire, 4, sizeof(struct personne), f);
        if (fclose(f))
          {
            printf("Impossible de fermer le fichier toto.txt");
            return -2;
```

```
        }
    return 0;
}
```

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

struct personne
{
  char nom[30];
  int age;
};

int main(int argv, char** argc)
{
    FILE* f;
    int i, n = 0;
    struct personne repertoire[4];
    f = fopen("toto.txt", "r");
    if (f == NULL)
      {
        printf("Impossible d'ouvrir le fichier toto.txt.\n");
        return -1;
      }
    while(fread(repertoire + n, 1, sizeof(struct personne), f))
      n++;
    for (i = 0 ; i < n ; i++)
      printf("%s %d\n", repertoire[i].nom, repertoire[i].age);
    if (fclose(f))
      {
        printf("Impossible de fermer le fichier toto.txt.\n");
        return -2;
      }
    return 0;
}
```

1.12 Listes Chaînées

1.12.1 Le problème

Plusieurs problèmes surviennent lorsque l'on utilise des tableaux pour stocker des valeurs :

- En cas de redimensionnement, il faut refaire une allocation dynamique et recopier tout le tableau.
- En cas d'insertion ou de suppression d'un élément, il faut décaler vers la gauche ou vers la droite tous les successeurs.
- Concaténer ou fusionner des tableaux est une opération lourde au niveau mémoire.
- Tous les éléments doivent être stockés dans des zones contigües, l'allocation est donc un lourd travail lorsque la mémoire est fragmentée.
- Bref, de nombreuses opérations sont lourdes en temps d'exécution.

Nous définirons dans ce cours un autre moyen de stocker les données en formant un ensemble ordonné, le listes chaînées.

1.12.2 Pointeurs et structures

Soit T un type structuré défini comme suit :

```
typedef struct T
{
        int i;
        char c;
}T;
```

Si p est un pointeur de type $T*$, alors p contient l'adresse mémoire d'un élément de type T. Soit t une variable de type T, et soit l'affectation `p` = &`t`. Alors, p pointe sur t. De ce fait $*p$ et t sont des alias, et nous pourrons indifférement utiliser $t.i$ (resp. $t.c$) et $(*p).i$ (resp $(*p).c$). Par exemple, réecrivons le programme de l'exemple du cours sur les structures :

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

#define N 10

/*************************************************************/

typedef struct point
{
  double abs;
  double ord;
}point;

/*************************************************************/

point nextPoint(point* previous)
{
  point result;
  result.ord = (*previous).ord + 1.;
  result.abs = (*previous).abs + 2.;
  return result;
}

/*************************************************************/

void initTableauPoints(point* p, int n)
{
  int i;
  (*p).ord = 0;
  (*p).abs = 1;
  for(i = 1 ; i < n ; i++)
```

```
    *(p + i) = nextPoint(p + i - 1);
}

/*************************************************************/

void affichePoint(point* p)
{
  printf("(%f, %f)", (*p).abs, (*p).ord);
}

/*************************************************************/

void afficheTableauPoints(point* p, int n)
{
  int i;
 for(i = 1 ; i < n ; i++)
    {
      printf("p[%d] = ", i);
      affichePoint(p + i);
      printf("\n");
    }
}

/*************************************************************/

int main()
{
  point* p;
  p = (point*)malloc(N * sizeof(point));
  if (p == NULL)
    return -1;
  initTableauPoints(p, N);
  afficheTableauPoints(p, N);
  free(p);
  return 0;
}
```

L'écriture $(*p).i$ permet de désigner le champ i de la variable pointée par p. Cette écriture, peu commode, peut être remplacée par $p->i$, par exemple :

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

#define N 10

/*************************************************************/

typedef struct point
{
  double abs;
  double ord;
}point;

/*************************************************************/

point nextPoint(point* previous)
{
  point result;
  result.ord = previous->ord + 1.;
  result.abs = previous->abs + 2.;
```

```
  return result;
}

/*************************************************************/

void initTableauPoints(point* p, int n)
{
  int i;
  p->ord = 0;
  p->abs = 1;
  for(i = 1 ; i < n ; i++)
    *(p + i) = nextPoint(p + i - 1);
}

/*************************************************************/

void affichePoint(point* p)
{
  printf("(%f, %f)", p->abs, p->ord);
}

/*************************************************************/

void afficheTableauPoints(point* p, int n)
{
  int i;
 for(i = 1 ; i < n ; i++)
    {
      printf("p[%d] = ", i);
      affichePoint(p + i);
      printf("\n");
    }
}

/*************************************************************/

main()
{
  point* p;
  p = (point*)malloc(N * sizeof(point));
  if (p == NULL)
    return -1;
  initTableauPoints(p, N);
  afficheTableauPoints(p, N);
  free(p);
}
```

Attention ! Les opérateurs d'accès aux champs . et −> ont une priorité supérieure à celles de tous les autres opérateurs du langage ! Si vous manipulez un tableau de structures `t` et que vous souhaitez accéder au champ `t` du `i`-ème élément, est-il intelligent d'écrire `*(i + t).c` ? Absolument pas ! . est prioritaire sur ∗, donc le parenthèsage implicite est `*(( i + t).c)`, essayez de vous représenter ce que cela fait, et vous comprendrez pourquoi votre programme plante ! En écrivant `(i + t)->c`, vous obtenez une expression équivalente à `(*(i + t)).c`, qui est déjà bien plus proche de ce que l'on souhaite faire.

1.12.3 Un premier exemple

Considérons le type suivant :

```
typedef struct maillon
```

```
{
        int data;
        struct maillon* next;
}maillon;
```

On appelle cette forme de type un type récursif, c'est-à-dire qui contient un pointeur vers un élément du même type. Observons l'exemple suivant :

```
#include<stdio.h>

/*************************************************************/

typedef struct maillon
{
        int data;
        struct maillon* next;
}maillon;

/*************************************************************/

int main()
{
  maillon m, p;
  maillon* ptr;
  m.data = 1;
  m.next = &p;
  p.data = 2;
  p.next = NULL;
  for (ptr = &m ; ptr != NULL ; ptr = ptr->next)
    printf("data = %d\n", ptr->data);
  return 0;
}
```

`m` et `p` sont deux maillons, et `m->next` pointe vers `p`, cela signifie que lorsque l'on initialise `ptr` à `&m`, alors `ptr` pointe sur `m`. Donc, lors de la première itération de la boucle **for**, la valeur `prt->data` est `m.data`, à savoir 1. Lorsque le pas de la boucle est exécuté, `ptr` reçoit la valeur `ptr->next`, qui n'est autre que `m->next`, ou encore `&p`. Donc `ptr` pointe maintenant sur `p`. Dans la deuxième itération de la boucle, `ptr->data` est la valeur `p.data`, à savoir 2. Ensuite le pas de la boucle est exécuté, et `ptr` prend la valeur `ptr->next`, à savoir `p->next`, ou encore `NULL`. Comme `ptr == NULL`, alors la boucle s'arrête. Ce programme affiche donc :

```
data = 1
data = 2
```

1.12.4 Le chaînage

La façon de renseigner les valeurs des pointeurs `next` observée dans l'exemple précédent s'appelle le **chaînage**. Une liste de structures telle que chaque variable structurée contienne un pointeur vers vers une autre variable du même type s'appelle une **liste chaînée**. Utilisons un tableau pour stocker les éléments d'une liste chaînée à 10 éléments.

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

#define N 10

/*************************************************************/

typedef struct maillon
{
        int data;
        struct maillon* next;
```

```
}maillon;

/*************************************************************/

void printData(maillon* ptr)
{
  for( ; ptr != NULL ; ptr = ptr->next)
    printf("data = %d\n", ptr->data);
}

/*************************************************************/

int main()
{
  maillon* l;
  int i;
  l = (maillon*)malloc(N * sizeof(maillon));
  if (l == NULL)
    return -1;
  l->data = 0;
  for(i = 1 ; i < N ; i++)
    {
      (l + i)->data = i;
      (l + i - 1)->next = l + i;
    }
  (l + N - 1)->next = NULL;
  printData(l);
  free(l);
  return 0;
}
```

Les 10 maillons de la liste chaînée ont été placés dans le tableau l, la première boucle dispose le chaînage des éléments dans le même ordre que dans le tableau, l'affichage de la liste est fait dans le sous-programme `printfData`, et seul le chaînage y est utilisé. Comme le champ `data` du i-ème élément de la liste contient la valeur i, alors ce programme affiche :

```
data = 0
data = 1
data = 2
data = 3
data = 4
data = 5
data = 6
data = 7
data = 8
data = 9
```

Pour modifier l'ordre de parcours des maillons, il suffit de modifier le chaînage, par exemple,

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

#define N 10

/*************************************************************/

typedef struct maillon
{
        int data;
```

```
        struct maillon* next;
}maillon;

/*************************************************************/

void printData(maillon* ptr)
{
  for( ; ptr != NULL ; ptr = ptr->next)
    printf("data = %d\n", ptr->data);
}

/*************************************************************/

int main()
{
  maillon* l;
  int i;
  l = (maillon*)malloc(N * sizeof(maillon));
  if (l == NULL)
    {
      printf("Plus de mémoire");
      return 1;
    }
  l->data = 0;
  (l + 1)->data = 1;
  (l + N - 2)->next = l + 1;
  (l + N - 1)->next = NULL;
  for(i = 2 ; i < N ; i+=1)
    {
      (l + i)->data = i;
      (l + i - 2)->next = l + i;
    }
  printData(l);
  free(l);
  return 0;
}
```

Ce programme affiche

```
data = 0
data = 2
data = 4
data = 6
data = 8
data = 1
data = 3
data = 5
data = 7
data = 9
```

1.12.5 Utilisation de *malloc*

Pour le moment, nous avons stocké les éléments dans un tableau, les maillons étaient donc regroupés dans des zones mémoire contigües. Il est possible de stocker les maillons dans les zones non contigües, tous peuvent être retrouvés à l'aide du chaînage. Par exemple,

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

```c
#define N 10

/*************************************************************/

typedef struct maillon
{
        int data;
        struct maillon* next;
}maillon;

void printData(maillon* ptr)
{
  for( ; ptr != NULL ; ptr = ptr->next)
    printf("data = %d\n", ptr->data);
}

/*************************************************************/

void freeLL(maillon* l)
{
  maillon* n;
  while(l != NULL)
    {
      n = l->next;
      free(l);
      l = n;
    }
}

/*************************************************************/

int main()
{
  maillon* l;
  maillon* current;
  maillon* previous;
  int i;
  l = (maillon*)malloc(sizeof(maillon));
  if(l == NULL)
    return -1;
  l->data = 0;
  previous = l;
  for(i = 1 ; i < N ; i++)
    {
      current = (maillon*)malloc(sizeof(maillon));
      if(current == NULL)
        exit(0);
      current->data = i;
      previous->next = current;
      previous = current;
    }
  current->next = NULL;
  printData(l);
  freeLL(l);
  return 0;
}
```

Ce programme affiche

```
data = 0
```

```
data = 1
data = 2
data = 3
data = 4
data = 5
data = 6
data = 7
data = 8
data = 9
```

Pour plus de clarté, on placera l'initialisation de la liste dans une fonction :

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

#define N 10

/*************************************************************/

typedef struct maillon
{
        int data;
        struct maillon* next;
}maillon;

/*************************************************************/

void printLL(maillon* ptr)
{
  for( ; ptr != NULL ; ptr = ptr->next)
    printf("data = %d\n", ptr->data);
}

/*************************************************************/

maillon* initLL(int n)
{
  maillon* first;
  maillon* current;
  maillon* previous;
  int i;
  first = (maillon*)malloc(sizeof(maillon));
  if(first == NULL)
    return NULL;
  first->data = 0;
  previous = first;
  for(i = 1 ; i < n ; i++)
    {
      current = (maillon*)malloc(sizeof(maillon));
      if(current == NULL)
        exit(0);
      current->data = i;
      previous->next = current;
      previous = current;
    }
  current->next = NULL;
  return first;
}

/*************************************************************/
```

```
void freeLL(maillon* l)
{
  maillon* n;
  while(l != NULL)
    {
      n = l->next;
      free(l);
      l = n;
    }
}

/*************************************************************/

int main()
{
  maillon* l;
  l = initLL(N);
  if (l == NULL)
    return -1;
  printLL(l);
  freeLL(l);
  return 0;
}
```

1.12.6 Opérations

Observons de quelle façon il est possible d'effectuer certaines opérations sur une liste chaînée. Les sous-programmes ci-dessous ont été pensé et combinés pour être les plus simples possibles...

Création d'un maillon

```
maillon* creeMaillon(int n)
{
  maillon* l;
  l = (maillon*)malloc(sizeof(maillon));
  if(l == NULL)
    exit(0);
  l->data = n;
  l->next = NULL;
  return l;
}
```

Insertion au début d'une liste chaînée

```
maillon* insereDebutLL(maillon* l, int n)
{
  maillon* first = creeMaillon(n);
  first->next = l;
  return first;
}
```

Création d'une liste chaînée $\{0, \ldots, n-1\}$

```
 maillon* initLL(int n)
{
  maillon* l = NULL;
  int i;
  for(i = n - 1 ; i >= 0 ; i--)
    l = insereDebut(l, i);
  return l;
}
```

1.12.7 Listes doublement chaînées

Un maillon d'une liste doublement chaînée contient deux pointeurs, un vers le maillon précédent, et un vers le maillon suivant.

```
typedef struct dmaillon
{
  int data;
  struct dmaillon* previous;
  struct dmaillon* next;
}dmaillon;
```

Aussi paradoxal que cela puisse paraître, bon nombre d'opérations sont davantage aisées sur des listes doublement chaînées. Pour se faciliter la tâche, nous manipulerons les listes chaînées à l'aide de deux pointeurs, un vers son premier élément, et un vers son dernier :

```
typedef struct dLinkedList
{
  struct dmaillon* first;
  struct dmaillon* last;
}dLinkedList;
```

Le soin d'ecrire les opérations sur ces listes vous est laissé dans le TP...

1.13 Conseils techniques

1.13.1 Le problème

Dès que l'on code un programme d'une certaine envergure, bon nombre de problèmes se posent.

1. Le débogage peut devenir un cauchemar sans fin.
2. Ensuite le code écrit peut être une bouillie lisible que par son auteur.
3. Le code peut contenir des redondances qui n'en facilite ni la lecture, ni le débogage.

Je passe les nombreuses difficultés et laideurs pouvant résulter d'un développement hâtif. Retenez que sans méthode, vous ne parviendrez pas à développer un projet important, tout simplement parce que le temps de débogage sera prohibitif, et parce que votre code, après les multiples bidouillages auxquels vous soumettrez pour déboguer sera un gros pavé bordélique et illisible.

1.13.2 Les règles d'or

La solution au problème mentionné ci-avant réside dans l'application des règles d'or suivantes :

Généralités

Avant toute chose, rappelons quelques évidences :

1. **indentez** votre code, on doit pouvoir trouver, pour chaque accolade ouvrante, où se trouve l'accolade fermante qui lui correspond en moins d'un quart de seconde.
2. utilisez des **noms de variable explicites**, le nom de la variable doit être suffisant pour pouvoir comprendre à quoi elle sert.

Fonctions

Commencez par découper votre très gros problème en plein de petits problèmes que vous traiterez individuellement avec des fonctions. Chaque fonction devra :

1. **porter un nom explicite** : vous avez droit à 256 caractères...
2. **être précédée d'un commentaire** décrivant clairement et sans paraphraser le code ce que fait la fonction.
3. **tenir sur une page** : il faut que vous puissiez voir toute la fonction sans avoir à utiliser l'ascenseur, et comprendre aisément ce qu'elle fait.
4. **contenir au maximum trois niveaux d'imbrication** : si vous avez plus de trois blocs (boucles, conditions, etc.) imbriqués, placez tout ce qui déborde dans une autre fonction.
5. **ne jamais utiliser de variables globales** : l'utilisation des variables globales est réservée à certains cas très précis. Dans tous les autres cas, vos fonctions doivent utiliser les passages de paramètres (par adresse si nécessaire) et les valeurs de retour.
6. **être précédée de toutes les fonctions qu'elle appelle** : d'une part un lecteur parcourant votre code le comprendra plus aisément si pour chaque appel de fonction, il a déjà vu la définition de cette fonction. Ainsi on pourra lire votre code dans l'ordre et éviter de slalomer entre les fonctions. D'autre part, cela vous évitera d'avoir à écrire les prototypes des fonctions en début de fichier. Vous réserverez aux cas où il et impossible (ou laid) de faire autrement les fonctions qui s'invoquent mutuellement.

Compilation séparée

Un fichier contenant plusieurs centaines de fonctions est impossible à lire, en plus d'être d'une laideur accablante. Vous prendrez donc soin de regrouper vos fonctions dans des fichiers, de les compiler séparément et de les linker avec un makefile.

1.13.3 Débogage

Vous êtes probablement déjà conscients du fait que le débogage occupe une partie très significative du temps de développement. Aussi est-il appréciable de la diminuer autant que possible.

1. D'une part parce que lorsque ce temps devient prohibitif, il constitue une perte de temps fort malvenue.
2. D'autre part parce que les multiples bidouilles opérées pour corriger les erreurs ne font souvent que nuire à la lisibilité du code
3. Et enfin parce qu'un temps anormalement élevé est en général une conséquence d'une analyse et d'un codage hâtifs.

Aussi est-il généralement plus efficace de passer un peu plus de temps à coder, et beaucoup moins de temps à déboguer. Pour ce faire :

1. Notez bien que tous les conseils énoncés précédemment sont encore d'actualité. En particulier ceux sur les fonctions.
2. Construisez les fonctions dans l'ordre inverse de l'ordre d'invocation, et testez-les une par une. Les bugs sont beaucoup plus facile à trouver quand on les cherche dans une dizaine de lignes que dans une dizaine de pages. Et en assemblant des fonctions qui marchent correctement, vous aurez plus de chances de rédiger un programme correct.
3. N'hésitez pas à utiliser des logiciels comme `valgrind`. `valgrind` examine l'exécution de votre code et vous rapporte bon nombre d'utilisation irrégulière de la mémoire (pointeurs fous, zones non libérées, etc.)

1.13.4 Durée de vie du code

Si vous ne voulez pas qu'un jour un développeur fasse un `sélectionner`/`supprimer` sur votre code, vous devez avoir en tête l'idée que quand quelqu'un reprend ou utilise votre code vous devez réduire au minimum à la fois le temps qu'il mettra à le comprendre et le nombre de modifications qu'il devra faire.

Le code doit être réutilisable

Cela signifie qu'un autre programmeur doit pouvoir poursuivre votre projet en faisant un minimum de modifications. Autrement dit, un autre programmeur doit pouvoir appeler vos fonctions aveuglément et sans même regarder ce qu'il y a dedans. Les noms des fonctions et les commentaires décrivant leurs comportement doivent être clairs, précis, et bien évidemment exempt de bugs. Sinon, un `sélectionner`/`supprimer` mettra fin à la courte vie de votre code. Notez bien par ailleurs que si vous avez réparti votre code dans des fichiers séparés de façon intelligente, votre code sera bien plus simple à réutiliser.

Le code doit être adaptable

Supposons que pour les besoins d'un projet, un autre développeur veuille partir de votre code, par exemple utiliser des listes chaînées au lieu de tableau, ou encore ajouter une sauvegarde sur fichier, etc. Si votre code est bien découpé, il n'aura que quelques fonctions à modifier. Si par contre, vous avez mélangé affichage, saisies, calculs, sauvegardes, etc. Notre développeur devra passer un temps considérable à bidouiller votre code, et cela sans aucune certitude quand à la qualité du résultat. Il fera donc un `sélectionner`/`supprimer` et recodera tout lui-même.

1.13.5 Exemple : le carnet de contacts

Je me suis efforcé, autant que possible, de suivre mes propres conseils et de vous donner un exemple. Je vous laisse à la fois observer mes recommandations dans le code qui suit, et traquer les éventuelles effractions que j'aurais pu commettre.

util.h

```c
#ifndef UTIL_H
#define UTIL_H

/*
  Saisit une chaine de caracteres de longueur sizeMax a l'adresse
  adr, elimine le caractere de retour a la ligne et vide si necessaire
  tous les caracteres supplementaires du tampon de saisie.
 */

void getString(char* adr,  int sizeMax);

/*****************************************************/

/*
  Saisit un entier.
 */

int getInt();

/*********************************************************/

/*
  Echange les chaines de caracteres s1 et s2, de tailles
  maximales sizeMax.
*/

void swapStrings(char* s1, char* s2, int sizeMax);

#endif
```

util.c

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

void getString(char* adr,  int sizeMax)
{
  int len;
  fgets(adr, sizeMax, stdin);
  len = strlen(adr);
  if (*(adr + len - 1) == '\n')
      *(adr + len - 1) = 0;
  else
    while(getchar() != '\n');
}

/*****************************************************/

int getInt()
{
  char tab[10];
  getString(tab, 10);
  return atoi(tab);
}

/*********************************************************/
```

```
void swapStrings(char* s1, char* s2, int sizeMax)
{
  char temp[sizeMax];
  strncpy(temp, s1, sizeMax);
  strncpy(s1, s2, sizeMax);
  strncpy(s2, temp, sizeMax);
  *(s1 + sizeMax - 1) = 0;
  *(s2 + sizeMax - 1) = 0;
}
```

tableau.h

```
#ifndef TABLEAU_H
#define TABLEAU_H

#define SIZE_MAIL 30
#define NB_MAILS 6

/*
  Implemente un carnet de contacts a l'aide d'un tableau.
  Une case inoccupee est representee par une chaine vide,
  toutes les adresses sont disposees par ordre alphabetique
  au debut du tableau.
 */

/*****************************************************/

/*
  Affiche le carnet de contacts.
*/

void afficheMails(char* mails);

/*****************************************************/

/*
  Retourne l'adresse du i-eme mail.
*/

char* getMail(char* mails, int i);

/*****************************************************/

/*
  Retourne le nombre de contacts.
*/

int nombreMails(char* mails);

/*****************************************************/

/*
  Creee un tableau de d'e-mails et le retourne. Ce tableau contient
  NB_MAILS chaines de capacites longueur SIZE_MAIL initialises
  avec des chaines vides.
*/

char* creerMails();
```

```
/*************************************************/

/*
  Libere la memoire
*/

void detruitMails(char* mails);

/*************************************************/

/*
  Supprime l'adresse dont l'indice est passe en parametre.
*/

void supprimeMail(char* mails, int indice);

/*************************************************/

/*
  Ajoute le mail mail dans le tableau mails.
*/

void ajouteMail(char* mails, char* mail);

/*************************************************/

/*
  Remplace le indice-eme mail du tableau mails par mail. L'indice
  est suppose valide.
*/

void changeMail(char* mails, char* mail, int indice);

/*************************************************/

/*
  Ecrit tous les contacts de mails dans le fichier nomFichier.
*/

void sauvegardeMails(char* mails, char* nomFichier);

/*************************************************/

/*
  Lit tous les contacts de mails dans le fichier nomFichier.
*/

void restaureMails(char* mails, char* nomFichier);

#endif
```

tableau.c

```
#include "../../src/methodo/tableau.h"

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

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

#include "util.h"

/*******************************************************/

void afficheMails(char* mails)
{
  int indice = 0;
  printf("Liste des contacts : \n");
  while(indice < NB_MAILS && *getMail(mails, indice))
    {
      printf("%d : %s\n", indice + 1, getMail(mails, indice));
      indice++;
    }
}

/*******************************************************/

char* getMail(char* mails, int i)
{
  return mails + i * SIZE_MAIL;
}

/*******************************************************/

int nombreMails(char* mails)
{
  int indice = 0;
  while(indice < NB_MAILS && *getMail(mails, indice))
      indice++;
  return indice;
}

/*******************************************************/

char* creerMails()
{
  char* adr = (char*)malloc(sizeof(char) * SIZE_MAIL * NB_MAILS);
  int i;
  if (adr == NULL)
    {
      printf("Heap overflow");
      exit(0);
    }
  for(i = 0 ; i < NB_MAILS ; i++)
      *getMail(adr, i) = 0;
  return adr;
}

/*******************************************************/

void detruitMails(char* mails)
{
  free(mails);
}

/*******************************************************/
```

```
void supprimeMail(char* mails, int indice)
{
  while(indice < NB_MAILS && *getMail(mails, indice + 1))
    {
      strncpy(getMail(mails, indice),
              getMail(mails, indice + 1),
              SIZE_MAIL);
      indice++;
    }
  if(indice < NB_MAILS)
    *getMail(mails, indice) = 0;
}

/*********************************************************/

/*
  Retourne l'indice du premier emplacement libre dans
  le tableau mails contenant nbMax adresses. On suppose que le tableau
  n'est pas plein.
*/

int indicePremiereChaineVide(char* mails, int indiceMax)
{
  int milieu;
  if (indiceMax == 0)
    return 0;
  milieu = indiceMax / 2;
  if (!*getMail(mails, milieu))
    return indicePremiereChaineVide(mails, milieu);
  else
    return milieu + 1 +
      indicePremiereChaineVide(getMail(mails, milieu + 1),
                               indiceMax - (milieu + 1));
}

/*********************************************************/

/*
  Trie le tableau mails contenant (indice + 1) elements,
  ne fonctionne que si tous les autres elements
  sont tries.
*/

void placeMail(char* mails, int indice)
{
  if (indice > 0 && indice < NB_MAILS &&
      strncmp(getMail(mails, indice),
              getMail(mails, indice - 1),
              SIZE_MAIL) < 0)
    {
      swapStrings(getMail(mails, indice),
                  getMail(mails, indice - 1),
                  SIZE_MAIL);
      placeMail(mails, indice - 1);
    }
  else
    if (indice >= 0 && indice < NB_MAILS - 1 &&
        *getMail(mails, indice + 1) &&
```

```
        strncmp(getMail(mails, indice),
                getMail(mails, indice + 1),
                SIZE_MAIL) > 0)
      {
        swapStrings(getMail(mails, indice),
                    getMail(mails, indice + 1),
                    SIZE_MAIL);
        placeMail(mails, indice + 1);
      }
}

/*******************************************************/

void ajouteMail(char* mails, char* mail)
{
  int indice;
  if (*getMail(mails, NB_MAILS - 1))
    {
      printf("Carnet de contact plein.\n");
    }
  else
    {
      indice = indicePremiereChaineVide(mails, NB_MAILS - 1);
      strncpy(getMail(mails, indice), mail, SIZE_MAIL);
      *(getMail(mails, indice) + SIZE_MAIL - 1) = 0;
      placeMail(mails, indice);
    }
}

/*******************************************************/

void changeMail(char* mails, char* mail, int indice)
{
  strncpy(getMail(mails, indice), mail, SIZE_MAIL);
  *(getMail(mails, indice) + SIZE_MAIL - 1) = 0;
  placeMail(mails, indice);
}

/*******************************************************/

void sauvegardeMails(char* mails, char* nomFichier)
{
  FILE* f = fopen(nomFichier, "w");
  int i;
  if (f == NULL)
    printf("Impossible d'ouvrir le fichier %s", nomFichier);
  else
    for(i = 0 ; i < NB_MAILS && *getMail(mails, i) ; i++)
      fwrite(getMail(mails, i), sizeof(char), SIZE_MAIL, f);
  fclose(f);
}

/*******************************************************/

void restaureMails(char* mails, char* nomFichier)
{
  FILE* f = fopen(nomFichier, "r");
  int i, ret = 1;
  if (f == NULL)
```

```
    printf("Impossible d'ouvrir le fichier %s", nomFichier);
  else
    for(i = 0 ; i < NB_MAILS && ret ; i++)
      ret = fread(getMail(mails, i), sizeof(char), SIZE_MAIL, f);
  fclose(f);
}
```

eMails.c

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

#include "tableau.h"
#include "util.h"

#define AFFICHER_OPTION 1
#define SUPPRIMER_OPTION 2
#define MODIFIER_OPTION 3
#define AJOUTER_OPTION 4
#define QUITTER_OPTION 5
#define NB_OPTIONS 5

#define F_NAME ".adressesMails.txt"

/*********************************************************/

/*
  Affiche le menu principal.
*/

void afficheMenu()
{
  printf("\nOptions disponibles :\n"
         "%d - afficher les contacts\n"
         "%d - supprimer un contact\n"
         "%d - modifier un contact\n"
         "%d - ajouter un contact\n"
         "%d - quitter\n",
         AFFICHER_OPTION,
         SUPPRIMER_OPTION,
         MODIFIER_OPTION,
         AJOUTER_OPTION,
         QUITTER_OPTION);
}

/*********************************************************/

/*
  Affiche le menu principal, retourne la valeur saisie par
  l'utilisateur.
*/

int choisitOptionMenu()
{
  int option;
  do
    {
      afficheMenu();
```

```
      printf("Choisissez une option en saisissant son numero : ");
      option = getInt();
      if (option <= 0 && option > NB_OPTIONS)
        printf("option invalide\n");
    }
  while(option <= 0 && option > NB_OPTIONS);
  return option;
}

/*******************************************************/

/*
  Demande a l'utilisateur de saisir un mail, le place
  a l'adresse adr.
*/

void saisitMail(char* adr)
{
  printf("Veuillez saisir l'adresse e-mail de votre contact : ");
  do
    {
      getString(adr, SIZE_MAIL);
      if (!*adr)
        printf("Vous devez saisir une adresse");
    }
  while(!*adr);
}

/*******************************************************/

/*
  Affiche la liste de mails, saisit et retourne le numero de
  l'un d'eux.
*/

int choisitMail(char* mails)
{
  int i, nbMails;
  nbMails = nombreMails(mails);
  afficheMails(mails);
  do
    {
      printf("Choisissez un mail en saisissant son numero : ");
      i = getInt();
      if (i <= 0 && i > nbMails)
        printf("Cet indice n'existe pas ! \n");
    }
  while(i <= 0 && i > nbMails);
  return i - 1;
}

/*******************************************************/

/*
  Saisit un mail m et un indice i, puis remplace le i-eme mail
  de mails par m.
*/

void modifierOption(char* mails)
```

```
{
  int i;
  char m[SIZE_MAIL];
  printf("Modification d'un contact : \n");
  i = choisitMail(mails);
  saisitMail(m);
  changeMail(mails, m, i);
}

/*******************************************************/

/*
  Saisit un mail m et un indice i, puis remplace le i-eme mail
  de mails par m.
*/

void ajouterOption(char* mails)
{
  char m[SIZE_MAIL];
  printf("Ajout d'un contact : \n");
  saisitMail(m);
  ajouteMail(mails, m);
}

/*******************************************************/

/*
  Saisit un indice i, puis supprime le i-eme mail dans mails.
*/

void supprimerOption(char* mails)
{
  int i;
  printf("Suppression d'un contact : \n");
  i = choisitMail(mails);
  supprimeMail(mails, i);
}

/*******************************************************/

/*
  Sauve les mails dans le fichier F_NAME et affiche un message
  d'adieu.
*/

void quitterOption(char* mails)
{
  sauvegardeMails(mails, F_NAME);
  printf("Au revoir !\n");
}

/*******************************************************/

/*
  Affiche le menu principal, saisit une option, et effectue le
  traitement necessaire.
*/

void executeMenu(char* mails)
```

```
{
  int option;
  do
    {
      option = choisitOptionMenu();
      switch(option)
        {
        case AFFICHER_OPTION :
          afficheMails(mails);
          break;
        case AJOUTER_OPTION :
          ajouterOption(mails);
          break;
        case MODIFIER_OPTION :
          modifierOption(mails);
          break;
        case SUPPRIMER_OPTION :
          supprimerOption(mails);
          break;
        case QUITTER_OPTION :
          quitterOption(mails);
          break;
        default:
          break;
        }
    }
  while(option != QUITTER_OPTION);
}

/*******************************************************/

int main()
{
  char* mails = creerMails();
  restaureMails(mails, F_NAME);
  executeMenu(mails);
  detruitMails(mails);
  return 0;
}
```

makefile

```
all : eMails eMails.tgz

util.o: util.c util.h
        gcc -Wall -c util.c

tableau.o: tableau.c tableau.h util.h
        gcc -Wall -c tableau.c

eMails.o: eMails.c tableau.h util.h
        gcc -Wall -c eMails.c

eMails: util.o tableau.o eMails.o
        gcc -o eMails -Wall util.o tableau.o eMails.o

eMails.tgz: util.h util.c tableau.h tableau.c eMails.c makefile
        tar cvfz eMails.tgz util.h util.c tableau.h tableau.c eMails.c makefile
```

Chapitre 2

Exercices

2.1 Variables et opérateurs

f

2.1.1 Entiers

Exercice 1 Saisie et affichage

Saisir une variable entière, afficher sa valeur.

Exercice 2 Permutation de 2 variables

Saisir deux variables et les permuter avant de les afficher.

Exercice 3 Permutation de 4 valeurs

Ecrire un programme demandant à l'utilisateur de saisir 4 valeurs A, B, C, D et qui permute les variables de la façon suivante :

noms des variables	A	B	C	D
valeurs avant la permutation	1	2	3	4
valeurs après la permutation	3	4	1	2

Exercice 4 Permutation de 5 valeurs

On considère la permutation qui modifie cinq valeurs de la façon suivante :

noms des variables	A	B	C	D	E
valeurs avant la permutation	1	2	3	4	5
valeurs après la permutation	4	3	5	1	2

Ecrire un programme demandant à l'utilisateur de saisir 5 valeurs que vous placerez dans des variables appelées A, B, C, D et E. Vous les permuterez ensuite de la façon décrite ci-dessus.

2.1.2 Flottants

Exercice 5 Saisie et affichage

Saisir une variable de type **float**, afficher sa valeur.

Exercice 6 Moyenne arithmétique

Saisir 3 valeurs, afficher leur moyenne.

Exercice 7 Surface du rectangle

Demander à l'utilisateur de saisir les longueurs et largeurs d'un rectangle, afficher sa surface.

Exercice 8 Moyennes arithmétique et géométrique

Demander à l'utilisateur de saisir deux valeurs a et b, afficher ensuite la différence entre la moyenne arithmétique $\frac{(a+b)}{2}$ et la moyenne géométrique $\sqrt{ab}$. Pour indication, `sqrt(f)` est la racine carrée du flottant `f`, cette fonction est disponible en important \#`include<math.h>`.

Exercice 9 Cartons et camions

Nous souhaitons ranger des cartons pesant chacun k kilos dans un camion pouvant transporter M kilos de marchandises. Ecrire un programme C demandant à l'utilisateur de saisir M et k, que vous représenterez avec des nombres flottants, et affichant le nombre (entier) de cartons qu'il est possible de placer dans le camion. N'oubliez pas que lorsque l'on affecte une valeur flottante à une variable entière, les décimales sont tronquées.

2.1.3 Caractères

Exercice 10 Prise en main

Affectez le caractère `'a'` à une variable de type **char**, affichez ce caractère ainsi que son code `ASCII`.

Exercice 11 - Successeur

Ecrivez un programme qui saisit un caractère et qui affiche son successeur dans la table des codes ASCII.

Exercice 11 - Casse

Ecrivez un programme qui saisit un caractère miniscule et qui l'affiche en majuscule.

Exercice 11 Codes `ASCII`

Quels sont les codes `ASCII` des caractères `'0'`, `'1'`, ..., `'9'`

2.1.4 Opérations sur les bits (difficiles)

Exercice 12 Codage d'adresses IP

Une adresse `IP` est constituée de 4 valeurs de 0 à 255 séparées par des points, par exemple 192.168.0.1, chacun de ces nombres peut se coder sur 1 octet. Comme une variable de type **long** occupe 4 octets en mémoire, il est possible de s'en servir pour stocker une adresse `IP` entière. Ecrivez un programme qui saisit dans des **unsigned short** les 4 valeurs d'une adresse `IP`, et place chacune d'elle sur un octet d'une variable de type **long**. Ensuite vous extrairez de cet entier les 4 nombres de l'adresse `IP` et les afficherez en les séparant par des points.

Exercice 13 Permutation circulaire des bits

Effectuez une permutation circulaire vers la droite des bits d'une variable `b` de type **unsigned short**, faites de même vers la droite.

Exercice 14 Permutation de 2 octets

Permutez les deux octets d'une variable de type **unsigned short** saisie par l'utilisateur.

Exercice 15 Inversion de l'ordre de 4 octets

Inversez l'ordre des 4 octets d'une variable de type **long** saisie par l'utilisateur. Utilisez le code du programme sur les adresses `IP` pour tester votre programme.

2.1.5 Morceaux choisis (difficiles)

Exercice 16 Pièces de monnaie

Nous disposons d'un nombre illimité de pièces de 0.5, 0.2, 0.1, 0.05, 0.02 et 0.01 euros. Nous souhaitons, étant donné une somme S, savoir avec quelles pièces la payer de sorte que le nombre de pièces utilisée soit minimal. Par exemple, la somme de 0.96 euros se paie avec une pièce de 0.5 euros, deux pièces de 0.2 euros, une pièce de 0.05 euros et une pièce de 0.01 euros.

1. Le fait que la solution donnée pour l'exemple est minimal est justifié par une idée plutôt intuitive. Expliquez ce principe sans excès de formalisme.
2. Ecrire un programme demandant à l'utilisateur de saisir une valeur comprise entre 0 et 0.99. Ensuite, affichez le détail des pièces à utiliser pour constituer la somme saisie avec un nombre minimal de pièces.

Exercice 17 Modification du dernier bit

Modifiez le dernier bit d'une variable `a` saisie par l'utilisateur.

Exercice 18 Associativité de l'addition flottante

L'ensemble des flottants n'est pas associatif, cela signifie qu'il existe trois flottants a, b et c, tels que $(a+b)+c \neq a+(b+c)$. Trouvez de tels flottants et vérifiez-le dans un programme.

Exercice 19 Permutation sans variable temporaire

Permutez deux variables `a` et `b` sans utiliser de variables temporaires.

2.2 Traitements conditionnels

2.2.1 Prise en main

Exercice 1 Majorité

Saisir l'âge de l'utilisateur et lui dire s'il est majeur.

Exercice 2 Valeur Absolue

Saisir une valeur, afficher sa valeur absolue.

Exercice 3 Admissions

Saisir une note, afficher "ajourné" si la note est inférieure à 8, "rattrapage" entre 8 et 10, "admis" dessus de 10.

Exercice 4 Assurances

Une compagnie d'assurance effectue des remboursements sur lesquels est ponctionnée une franchise correspondant à 10% du montant à rembourser. Cependant, cette franchise ne doit pas excéder 4000 euros. Demander à l'utilisateur de saisir le montant des dommages, afficher ensuite le montant qui sera remboursé ainsi que la franchise.

Exercice 5 Valeurs distinctes parmi 2

Afficher sur deux valeurs saisies le nombre de valeurs disctinctes.

Exercice 6 Plus petite valeur parmi 3

Afficher sur trois valeurs saisies la plus petite.

Exercice 7 Recherche de doublons

Ecrire un algorithme qui demande à l'utilisateur de saisir trois valeurs et qui lui dit s'il s'y trouve un doublon.

Exercice 8 Valeurs distinctes parmi 3

Afficher sur trois valeurs saisies le nombre de valeurs disctinctes.

Exercice 9 $ax + b = 0$

Saisir les coefficients a et b et afficher la solution de l'équation $ax + b = 0$.

Exercice 10 $ax^2 + bx + c = 0$

Saisir les coefficients a, b et c, afficher la solution de l'équation $ax^2 + bx + c = 0$.

2.2.2 Switch

Dans le problème suivant, vous serez confronté au problème dit de *saisie bufferisée* (ce néologisme est très vilain). Explication : lorsque vous saisissez des valeurs au clavier, elles sont placées dans le tampon de saisie (buffer). Une fois que vous avez saisi <entrée>, (le caractère `'$\setminus$n'`), les données sont lues directement dans le buffer par `scanf`. Mais cette fonction ne lira que le caractère qui a été saisi, et le `'$\setminus$n'` qui a validé la saisie restera dans le buffer jusqu'à la prochaine saisie d'un caractère. L'instruction `getchar`(); lit un octet dans le buffer. Donc, à chaque fois que vous avez placé un `'$\setminus$n'` dans le buffer, pensez à le vider avec un `getchar`();.

Exercice 11 Calculatrice

Ecrire un programme demandant à l'utilisateur de saisir
- deux valeurs a et b, de type int ;
- un opérateur op de type $char$, vérifiez qu'il s'agit del'une des valeurs suivantes : $+, -, *, /$.

Puis affichez le résultat de l'opération a op b.

2.2.3 L'échiquier

On indice les cases d'un échiquer avec deux indices i et j variant tous deux de 1 à 8. La case (i, j) est sur la ligne i et la colonne j. Par convention, la case $(1, 1)$ est noire.

Exercice 12 Couleurs

Ecrire un programme demandant à l'utilisateur de saisir les deux coordonnées i et j d'une case, et lui disant s'il s'agit d'une case blanche ou noire.

Exercice 13 Cavaliers

Ecrire un programme demandant à l'utilisateur de saisir les coordonnées (i, j) d'une première case et les coordonnées (i', j') d'une deuxième case. Dites-lui ensuite s'il est possible de déplacer un cavalier de (i, j) à (i', j').

Exercice 14 Autres pièces

Même exercice avec la tour, le fou, la dame et le roi. Utilisez un switch et présentez le programme de la sorte :

```
Quelle piece souhaitez-vous deplacer ?
0 = Cavalier
1 = Tour
2 = Fou
3 = Dame
4 = Roi
3
Saisissez les coordonnees de la case de depart :
A = 2
A = 1
Saisissez les coordonnees de la case d'arrivee :
A = 6
A = 5
Le mouvement (2, 1) -> (6, 5) est valide.
```

2.2.4 Heures et dates

Exercice 15 Opérations sur les heures

Ecrire un programme qui demande à l'utilisateur de saisir une heure de début (heures + minutes) et une heure de fin (heures + minutes aussi). Ce programme doit ensuite calculer en heures + minutes le temps écoulé entre l'heure de début et l'heure de fin. Si l'utilisateur saisit $10h30$ et $12h15$, le programme doit lui afficher que le temps écoulé entre l'heure de début et celle de fin est $1h45$. On suppose que les deux heures se trouvent dans la même journée, si celle de début se trouve après celle de fin, un message d'erreur doit s'afficher. Lors la saisie des heures, séparez les heures des minutes en demandant à l'utilisateur de saisir :
- heures de début
- minutes de début
- heures de fin
- minutes de fin

Exercice 16 Le jour d'après

Ecrire un programme permettant de saisir une date (jour, mois, année), et affichant la date du lendemain. Saisissez les trois données séparément (comme dans l'exercice précédent). Prenez garde aux nombre de jours que comporte chaque mois, et au fait que le mois de février comporte 29 jours les années bissextiles. Allez sur `http://fr.wikipedia.org/wiki/Ann\%C3\%A9e_bissextilepourconnaÃőtrelesrÃÍglesexactes`, je vous avais dit que les années étaient bissextiles si et seulement si elles étaient divisible par 4, après vérification, j'ai constaté que c'était légèrement plus complexe. Je vous laisse vous documenter et retranscrire ces règles de la façon la plus simple possible.

2.2.5 Intervalles et rectangles

Exercice 17 Intervalles bien formés

Demandez à l'utilisateur de saisir les deux bornes a et b d'un intervalle $[a, b]$. Contrôler les valeurs saisies.

Exercice 18 Appartenance

Demandez-lui ensuite de saisir une valeur x, dites-lui si $x \in [a, b]$

Exercice 19 Intersections

Demandez-lui ensuite de saisir les bornes d'un autre intervalle $[a', b']$. Contrôlez la saisie. Dites-lui ensuite si
- $[a, b] \subset [a', b']$
- $[a', b'] \subset [a, b]$
- $[a', b'] \cap [a, b] = \emptyset$.

Exercice 20 Rectangles

Nous représenterons un rectangle R aux cotés parallèles aux axes des abscisses et ordonnées à l'aide des coordonnées de deux points diamétralement opposés, le point en haut à gauche, de coordonnées $(xHautGauche, yHautGauche)$, et le point en bas à droite, de coordonnées $(xBasDroite, yBasDroite)$. Demander à l'utilisateur de saisir ces 4 valeurs, contrôlez la saisie.

Exercice 21 Appartenance

Demandez à l'utilisateur de saisir les 2 coordonnées d'un point (x, y) et dites à l'utilisateur si ce point se trouve dans le rectangle R.

Exercice 22 Intersection

Demandez à l'utilisateur de saisir les 4 valeurs

$$(xHautGauche', yHautGauche', xBasDroite', yBasDroite')$$

permettant de spécifier un deuxième rectangle R'. Précisez ensuite si
- $R \subset R'$
- $R' \subset R$
- $R \cap R' = \emptyset$

2.2.6 Préprocesseur

Exercice 23 L'échiquer

Même exercice mais en utilisant le plus possible les spécifications suivantes :

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

/*-------------------------------------------------------------*/
```

```
#define CAVALIER 0
#define TOUR 1
#define FOU 2
#define DAME 3
#define ROI 4

/*-------------------------------------------------------*/

#define MAIN int main() \
                {

/*-------------------------------------------------------*/

#define FINMAIN return 0;\
                }
/*-------------------------------------------------------*/

#define SI if(

/*-------------------------------------------------------*/

#define ALORS ){

/*-------------------------------------------------------*/

#define SINON } else {

/*-------------------------------------------------------*/

#define FINSI }

/*-------------------------------------------------------*/

#define SUIVANT(var) switch(var) {

/*-------------------------------------------------------*/

#define FINSUIVANT(def) default: def }

/*-------------------------------------------------------*/

#define CAS(valeur, instructions) case valeur : instructions \
                                  break;

/*-------------------------------------------------------*/

#define PIECES_LIST    \
printf("%hu = Cavalier\n", CAVALIER); \
printf("%hu = Tour\n", TOUR); \
printf("%hu = Fou\n", FOU); \
printf("%hu = Dame\n", DAME); \
printf("%hu = Roi\n", ROI)

/*-------------------------------------------------------*/

#define CHECK_COORD(i) \
SI i<1 || i>8 ALORS \
        printf("coordonnee invalide\n"); \
```

```
        return 1; \
FINSI

/*-------------------------------------------------------*/

#define GET_VAR(A, B) \
                printf("A = "); \
                scanf("%hu", &B); \
                CHECK_COORD(B);

/*-------------------------------------------------------*/

#define PRINT_VALID printf("valide")

/*-------------------------------------------------------*/

#define PRINT_INVALID printf("invalide")

/*-------------------------------------------------------*/

#define FATAL_ERROR printf("System Error. It is recommended that " \
                "you format your hard disk."); \
                return 1;

/*-------------------------------------------------------*/

#define CONDITIONAL_PRINT(cond) \
SI cond ALORS \
  PRINT_VALID; \
SINON \
  PRINT_INVALID; \
FINSI
```

2.2.7 Nombres et lettres

Exercice 24 Conversion numérique → français (très difficile)

Ecrire un programme saisissant un **unsigned long** et affichant sa valeur en toutes lettres. Rappelons que 20 et 100 s'accordent en nombre s'ils ne sont pas suivis d'un autre mot (ex. : quatre-vingt**s**, quatre-vingt-un). Mille est invariable (ex. : dix mille) mais pas million (ex. : deux millions) et milliard (ex. : deux milliards). Depuis 1990, tous les mots sont séparés de traits d'union (ex. : quatre-vingt-quatre), sauf autour des mots mille, millions et milliard (ex. : deux mille deux-cent-quarante-quatre, deux millions quatre-cent mille deux-cents). (source : `http://www.leconjugueur.com/frlesnombres.php`). Par exemple,

```
Saisissez un nombre = 1034002054
1034002054 : un milliard trente-quatre millions deux mille cinquante-quatre
```

Vous prendrez soin de respecter les espacements, les tirets, et de ne pas faire de fautes d'orthographe. Vous userez et abuserez de macros-instructions pour simplifier le plus possible votre code.

2.3 Boucles

2.3.1 Compréhension

Exercice 1 Programme mystère 1

Qu'affiche le programme suivant ?

```c
#include<stdio.h>
#define N 5

int main()
{
  int a = 1, b = 0;
  while(a <= N)
    b += a++;
  printf("%d, %d\n", a, b);
  return 0;
}
```

Exercice 2 Programme mystère 2

Qu'affiche le programme suivant ?

```c
#include<stdio.h>
#define M 3
#define N 4

int main()
{
  int a, b, c = 0, d;
  for (a = 0 ; a < M ; a++)
    {
      d = 0;
      for(b = 0 ; b < N ; b++)
          d+=b;
      c += d;
    }
  printf("a = %d, b = %d, c = %d, d = %d", a, b, c, d);
  return 0;
}
```

Exercice 3 Programme mystère 3

Qu'affiche le programme suivant ?

```c
#include<stdio.h>

int main()
{
  int a, b, c, d;
  a = 1; b = 2;
  c = a/b;
  d = (a==b)?3:4;
  printf("c = %d, d = %d\n", c, d);
  a = ++b;
  b %= 3;
  printf("a = %d, b = %d\n", a, b);
  b = 1;
```

```
  for(a = 0 ; a <= 10 ; a++)
    c = ++b;
  printf("a = %d, b = %d, c = %d, d = %d\n", a, b, c, d);
  return 0;
}
```

2.3.2 Utilisation de toutes les boucles

Les exercices suivants seront rédigés avec les trois types de boucle : tant que, répéter jusqu'à et pour.

Exercice 4 Compte à rebours

Ecrire un programme demandant à l'utilisateur de saisir une valeur numérique positive n et affichant toutes les valeurs $n, n-1, \ldots, 2, 1, 0$.

Exercice 5 Factorielle

Ecrire un programme calculant la factorielle (factorielle $n = n! = 1 \times 2 \times \ldots \times n$ et $0! = 1$) d'un nombre saisi par l'utilisateur.

2.3.3 Choix de la boucle la plus appropriée

Pour les exercices suivants, vous choisirez la boucle la plus simple et la plus lisible.

Exercice 7 Table de multiplication

Ecrire un programme affichant la table de multiplication d'un nombre saisi par l'utilisateur.

Exercice 8 Tables de multiplications

Ecrire un programme affichant les tables de multiplications des nombres de 1 à 10 dans un tableau à deux entrées.

Exercice 9 Puissance

Ecrire un programme demandant à l'utilisateur de saisir deux valeurs numériques b et n (vérifier que n est positif) et affichant la valeur b^n.

2.3.4 Morceaux choisis

Exercice 11 Approximation de 2 par une série

On approche le nombre 2 à l'aide de la série $\sum_{i=0}^{+\infty} \frac{1}{2^i}$. Effectuer cette approximation en calculant un grand nombre de terme de cette série. L'approximation est-elle de bonne qualité ?

Exercice 12 Approximation de e par une série

Mêmes questions qu'à l'exercice précédent en e à l'aide de la série $\sum_{i=0}^{+\infty} \frac{1}{i!}$.

Exercice 13 Approximation de e^x par une série

Calculer une approximation de e^x à l'aide de la série $e^x = \sum_{i=0}^{+\infty} \frac{x^i}{i!}$.

Exercice 14 Conversion d'entiers en binaire

Ecrire un programme qui affiche un **unsigned short** en binaire. Vous utiliserez l'instruction **sizeof(unsigned short)**, qui donne en octets la taille de la représentation en mémoire d'un **unsigned short**.

Exercice 15 Conversion de décimales en binaire

Ecrire un programme qui affiche les décimales d'un **double** en binaire.

Exercice 16 Inversion de l'ordre des bits

Ecrire un programme qui saisit une valeur de type **unsigned short** et qui inverse l'ordre des bits. Vous testerez ce programme en utilisant le précédent.

Exercice 17 Joli carré

Ecrire un programme qui saisit une valeur n et qui affiche le carré suivant ($n = 5$ dans l'exemple) :

```
n = 5
X  X  X  X  X

X  X  X  X  X

X  X  X  X  X

X  X  X  X  X

X  X  X  X  X
```

Exercice 18 Racine carrée par dichotomie

Ecrire un algorithme demandant à l'utilisateur de saisir deux valeurs numériques x et p et affichant $\sqrt{x}$ avec une précision p. On utilisera une méthode par dichotomie : à la k-ème itération, on cherche x dans l'intervalle $[min, sup]$, on calcule le milieu m de cet intervalle (à vous de trouver comment le calculer). Si cet intervalle est suffisament petit (à vous de trouver quel critère utiliser), afficher m. Sinon, vérifiez si $\sqrt{x}$ se trouve dans $[inf, m]$ ou dans $[m, sup]$, et modifiez les variables inf et sup en conséquence. Par exemple, calculons la racine carrée de 10 avec une précision 0.5,

- Commençons par la chercher dans $[0, 10]$, on a $m = 5$, comme $5^2 > 10$, alors $5 > \sqrt{10}$, donc $\sqrt{10}$ se trouve dans l'intervalle $[0, 5]$.
- On recommence, $m = 2.5$, comme $\frac{5}{2}^2 = \frac{25}{4} < 10$, alors $\frac{5}{2} < \sqrt{10}$, on poursuit la recherche dans $[\frac{5}{2}, 5]$
- On a $m = 3.75$, comme $3.75^2 > 10$, alors $3.75 > \sqrt{10}$ et $\sqrt{10} \in [2.5, 3.75]$
- On a $m = 3.125$, comme $3.125^2 < 10$, alors $3.125 < \sqrt{10}$ et $\sqrt{10} \in [3.125, 3.75]$
- Comme l'étendue de l'intervalle $[3.125, 3.75]$ est inférieure 2×0.5, alors $m = 3.4375$ est une approximattion à 0.5 près de $\sqrt{10}$.

2.3.5 Extension de la calculatrice

Une calculatrice de poche prend de façon alternée la saisie d'un opérateur et d'une opérande. Si l'utilisateur saisit 3, + et 2, cette calculatrice affiche 5, l'utilisateur a ensuite la possibilité de se servir de 5 comme d'une opérande gauche dans un calcul ultérieur. Si l'utilisateur saisit par la suite $*$ et 4, la calculatrice affiche 20. La saisie de la touche = met fin au calcul et affiche un résultat final.

Exercice 19 Calculatrice de poche

Implémentez le comportement décrit ci-dessus.

Exercice 20 Puissance

Ajoutez l'opérateur $ qui calcule a^b, vous vous restreindrez à des valeurs de b entières et positives.

Exercice 21 Opérations unaires

Ajoutez les opérations unaires racine carrée et factorielle.

2.4 Tableaux

2.4.1 Exercices de compréhension

Qu'affichent les programmes suivants ?

Exercice 1 Programme mystère

```
char C[4];
int k;
C[0] = 'a';
C[3] = 'J';
C[2] = 'k';
C[1] = 'R';
for(k = 0 ; k < 4 ; k++)
        printf("%c\n", C[k]);
for(k = 0 ; k < 4 ; k++)
        C[k]++;
for(k = 0 ; k < 4 ; k++)
        printf("%c\n", C[k]);
```

Exercice 2 Programme mystère

```
int K[10], i, j;
K[0] = 1;
for(i = 1 ; i < 10 ; i++)
        K[i] = 0;
for(j = 1 ; j <= 3 ; j++)
        for(i = 1 ; i < 10 ; i++)
                K[i] += K[i - 1];
for(i = 0 ; i < 10 ; i++)
        printf("%d\n", K[i]);
```

Exercice 3 Programme mystère

```
int K[10], i, j;
K[0] = 1;
K[1] = 1;
for(i = 2 ; i < 10 ; i++)
        K[i] = 0;
for(j = 1 ; j <= 3 ; j++)
        for(i = 1 ; i < 10 ; i++)
                K[i] += K[i - 1];
for(i = 0 ; i < 10 ; i++)
        printf("%d\n", K[i]);
```

2.4.2 Prise en main

Exercice 4 Initialisation et affichage

Ecrire un programme plaçant dans un tableau `int T[10];` les valeurs $1, 2, \ldots, 10$, puis affichant ce tableau. Vous initialiserez le tableau à la déclaration.

Exercice 5 Initialisation avec une boucle

Même exercice en initialisant le tableau avec une boucle.

Exercice 6 somme

Affichez la somme des n éléments du tableau T.

Exercice 7 recherche

Demandez à l'utilisateur de saisir un *int* et dites-lui si ce nombre se trouve dans T.

2.4.3 Indices

Exercice 8 permutation circulaire

Effectuez une permutation circulaire vers la droite des éléments de T en utilisant un deuxième tableau.

Exercice 9 permutation circulaire sans deuxième tableau

Même exercice mais sans utiliser de deuxième tableau.

Exercice 10 miroir

Inversez l'ordre des éléments de T sans utiliser de deuxième tableau.

2.4.4 Recherche séquentielle

Exercice 11 modification du tableau

Etendez le tableau `T` à 20 éléments. Placez dans `T[i]` le reste modulo 17 de i^2.

Exercice 12 min/max

Affichez les valeurs du plus petit et du plus grand élément de `T`.

Exercice 13 recherche séquentielle

Demandez à l'utilisateur de saisir une valeur `x` et donnez-lui la liste des indices `i` tels que `T[i]` a la valeur `x`.

Exercice 14 recherche séquentielle avec stockage des indices

Même exercice que précédemment, mais vous en affichant `La valeur ... se trouve aux indices suivants : ...` si `x` se trouve dans `T`, et `La valeur ... n'a pas ététrouvée` si `x` ne se trouve pas dans `T`.

2.4.5 Morceaux choisis

Exercice 15 pièces de monnaie

Reprennez l'exercice sur les pièces de monnaie en utilisant deux tableaux, un pour stocker les valeurs des pièces dans l'ordre décroissant, l'autre pour stocker le nombre de chaque pièce.

Exercice 16 recherche de la tranche minimale en $\mathcal{O}(n^3)$

Une tranche est délimitée par deux indices i et j tels que $i \leq j$, la valeur d'une tranche est $t_i + \ldots + t_j$. Ecrire un programme de recherche de la plus petite tranche d'un tableau, vous utiliserez trois boucles imbriquées. Vous testerez votre algorithme sur un tableau T à 20 éléments aléatoires (utilisez la fonction `random` de `stdlib.h`) de signes quelconques.

Exercice 17 recherche de la tranche minimale en $\mathcal{O}(n^2)$

Même exercice mais en utilisant deux boucles imbriquées.

Exercice 18 recherche de la tranche minimale en $\mathcal{O}(n)$ (difficile)

Même exercice mais en utilisant une seule boucle.

2.5 Chaînes de caractères

2.5.1 Prise en main

Exercice 1 Affichage

Créer une chaîne de caractères contenant la valeur "Les framboises sont perchees sur le tabouret de mon grand-pere." et affichez-la avec `%s`. Vous donnerez au tableau la plus petite taille possible.

Exercice 2 Affichage sans `%s`

Même exercice mais sans utiliser `%s`.

Exercice 3 Longueur

Ecrire un programme saisissant proprement une chaîne de caractère (sans débordement d'indice, avec le caractère nul et en faisant le ménage dans le buffer) et calculant sans `strlen` la taille de chaîne (nombre de caractères sans compter le caractère nul).

Exercice 4 Longueur sans retour chariot

Même exercice mais en supprimant de la chaîne l'éventuel caractère de validation de la saisie (retour à la ligne).

Exercice 5 Extraction

Ecrire un programme saisissant proprement une chaîne de caractère t, deux indices i et j et recopiant dans une deuxième chaîne t' la tranche $[t_i, \ldots, t_j]$.

Exercice 6 Substitution

Ecrire un programme saisissant proprement une chaîne de caractère t, deux caractères a et b et substituant des a à toutes les occurrences de b.

2.5.2 Les fonctions de `string.h`

Pour chacun des exercices suivants, vous vous documenterez sur les fonctions de `string.h` utiles et vous vous en servirez de façon convenable. Et ne faites pas de saletés !

Exercice 7 Comparaisons

Saisissez deux chaînes de caractères, déterminez la plus grande selon l'ordre lexicographique.

Exercice 8 Longueur

Saisissez deux chaînes de caractères, déterminez la plus longue des deux..

Exercice 9 Copie

Saisissez une chaîne de caractères, copiez-là dans une deuxième chaîne.

Exercice 10 Concaténation

Saisissez deux chaînes de caractères, affichez la concaténation de la première à la suite de la deuxième.

2.5.3 Morceaux choisis

Exercice 11 Extensions

Ecrire un programme saisissant un nom de fichier et affichant séparément le nom du fichier et l'extension. Vous prévoirez le cas où plusieurs extensions sont concaténées, par exemple : `langageCCC.tar.gz`.

Exercice 12 Expressions arithmétiques

Ecrire un programme saisissant une expression arithmétique totalement parenthésée, (par exemple $3 + 4$, $((3 - 2) + (7/3))$) et disant à l'utilisateur si l'expression est correctement parenthésée.

2.6 Fonctions

2.6.1 Géométrie

Ecrire un programme demandant à l'utilisateur de saisir une valeur numérique positive n et affichant un carré, une croix et un losange, tous de coté n. Par exemple, si $n = 10$, l'exécution donne

```
Saisissez la taille des figures
10
* * * * * * * * * *
*                 *
*                 *
*                 *
*                 *
*                 *
*                 *
*                 *
*                 *
* * * * * * * * * *
                  *
                *   *
              *       *
            *           *
          *               *
        *                   *
      *                       *
    *                           *
  *                               *
*                                   *
  *                               *
    *                           *
      *                       *
        *                   *
          *               *
            *           *
              *       *
                *   *
                  *
*                                   *
  *                               *
    *                           *
      *                       *
        *                   *
          *               *
            *           *
              *       *
                *   *
                  *
                *   *
              *       *
            *           *
          *               *
        *                   *
      *                       *
    *                           *
```

```
  *                                   *
*                                       *
```

Vous définirez des sous-programmes de quelques lignes et au plus deux niveaux d'imbrication. Vous ferez attention à ne jamais écrire deux fois les mêmes instructions. Pour ce faire, complétez le code source suivant :

```c
#include<stdio.h>

/*
  Affiche le caractere c
 */

void afficheCaractere(char c)
{
}

/*****************************************/

/*
  affiche n fois le caractere c, ne revient pas a la ligne
  apres le dernier caractere.
*/

void ligneSansReturn(int n, char c)
{
}

/*****************************************/

/*
  affiche n fois le caractere c, revient a la ligne apres
  le dernier caractere.
*/

void ligneAvecReturn(int n, char c)
{
}

/*****************************************/

/*
  Affiche n espaces.
*/

void espaces(int n)
{
}

/*****************************************/

/*
  Affiche le caractere c a la colonne i,
  ne revient pas a la ligne apres.
*/

void unCaractereSansReturn(int i, char c)
{
}

/*****************************************/
```

```c
/*
  Affiche le caractere c a la colonne i,
  revient a la ligne apres.
*/

void unCaractereAvecReturn(int i, char c)
{
}

/***************************************/

/*
  Affiche le caractere c aux colonnes i et j,
  revient a la ligne apres.
*/

void deuxCaracteres(int i, int j, char c)
{
}

/***************************************/

/*
  Affiche un carre de cote n.
*/

void carre(int n)
{
}

/***************************************/

/*
  Affiche un chapeau dont la pointe - non affichee - est
  sur la colonne centre, avec les caracteres c.
*/

void chapeau(int centre, char c)
{
}

/***************************************/

/*
  Affiche un chapeau a l'envers avec des caracteres c,
  la pointe - non affichee - est a la colonne centre
*/

void chapeauInverse(int centre, char c)
{
}

/***************************************/

/*
  Affiche un losange de cote n.
*/
```

```c
void losange(int n)
{
}

/*****************************************/

/*
   Affiche une croix de cote n
*/

void croix(int n)
{
}

/*****************************************/

main()
{
      int taille;
      printf("Saisissez la taille des figures\n");
      scanf("%d", &taille);
      carre(taille);
      losange(taille);
      croix(taille);
}
```

2.6.2 Arithmétique

Exercice 1 chiffres et nombres

Rapellons que `a \% b` est le reste de la division entière de a par b.

1. Ecrire la fonction `int unites(int n)` retournant le chiffre des unités du nombre n.
2. Ecrire la fonction `int dizaines(int n)` retournant le chiffre des dizaines du nombre n.
3. Ecrire la fonction `int extrait(int n, int p)` retourant le p-ème chiffre de représentation décimale de n en partant des unités.
4. Ecrire la fonction `int nbChiffres(int n)` retournant le nombre de chiffres que comporte la représentation décimale de n.
5. Ecrire la fonction `int sommeChiffres(int n)` retournant la somme des chiffres de n.

Exercice 2 Nombres amis

Soient a et b deux entiers strictement positifs. a est un diviseur strict de b si a divise b et $a \neq b$. Par exemple, 3 est un diviseur strict de 6. Mais 6 n'est pas un diviseur strict 6. a et b sont des nombres amis si la somme des diviseurs stricts de a est b et si la somme des diviseurs de b est a. Le plus petit couple de nombres amis connu est 220 et 284.

1. Ecrire une fonction `int sommeDiviseursStricts(int n)`, elle doit renvoyer la somme des diviseurs stricts de n.
2. Ecrire une fonction `int sontAmis(int a, int b)`, elle doit renvoyer 1 si a et b sont amis, 0 sinon.

Exercice 3 Nombres parfaits

Un nombre parfait est un nombre égal à la somme de ses diviseurs stricts. Par exemple, 6 a pour diviseurs stricts 1, 2 et 3, comme $1 + 2 + 3 = 6$, alors 6 est parfait.

1. Est-ce que 18 est parfait ?
2. Est-ce que 28 est parfait ?
3. Que dire d'un nombre ami avec lui-même ?
4. Ecrire la fonction `int estParfait(int n)`, elle doit retourner 1 si n est un nombre parfait, 0 sinon.

Exercice 4 Nombres de Kaprekar

Un nombre n est un nombre de Kaprekar en base 10, si la représentation décimale de n^2 peut être séparée en une partie gauche u et une partie droite v tel que $u + v = n$. $45^2 = 2025$, comme $20 + 25 = 45$, 45 est aussi un nombre de Kaprekar. $4879^2 = 23804641$, comme $238 + 04641 = 4879$ (le 0 de 046641 est inutile, je l'ai juste placé pour éviter toute confusion), alors 4879 est encore un nombre de Kaprekar.

1. Est-ce que 9 est un nombre de Kaprekar ?
2. Ecrire la fonction `int sommeParties(int n, int p)` qui découpe n est deux nombres dont le deuxième comporte p chiffres, et aui retourne leur somme. Par exemple,

$$sommeParties(12540, 2) = 125 + 40 = 165$$

3. Ecrire la fonction `int estKaprekar(int n)`

2.6.3 Passage de tableaux en paramètre

Exercice 5 Somme

Ecrire une fonction `int somme(int T[], int n)` retournant la somme des n éléments du tableau T.

Exercice 6 Minimum

Ecrire une fonction `int min(int T[], int n)` retournant la valeur du plus petit élément du tableau T.

Exercice 7 Recherche

Ecrire une fonction `int existe(int T[], int n, int k)` retournant 1 si k est un des n éléments du tableau T, 0 sinon.

Exercice 8 Somme des éléments pairs

Ecrivez le corps de la fonction `int sommePairs(int T[], int n)`, $sommePairs(T, n)$ retourne la somme des éléments pairs du tableau T à n éléments. N'oubliez pas que $a\%b$ est le reste de la division entière de a par b, et que vous êtes tenu d'utiliser au mieux les booléens.

Exercice 9 Vérification

Ecrivez le corps de la fonction `int estTrie(int T[], int n)`, $estTrie(T, n)$ retourne vrai si et seulement si le tableau T, à n éléments, est trié dans l'ordre croissant.

Exercice 10 Permutation circulaire

Ecrire une fonction `void permutation(int T[], int n)` effectuant une permutation circulaire vers la droite des éléments de T.

Exercice 11 Miroir

Ecrire une fonction `void miroir(int T[], int n)` inversant l'ordre des éléments de T.

2.6.4 Décomposition en facteurs premiers

On pose ici $N = 25$, vous utiliserez un `\#define N 25` dans toute cette partie. On rappelle qu'un nombre est premier s'il n'est divisible que par 1 et par lui-même. Par convention, 1 n'est pas premier.

1. Ecrivez une fonction `int estPremier(int n, int T[], int k)` retournant 1 si n est premier, 0 sinon. Vous vérifierez la primarité de n en examinant les restes des divisions de n par les k premiers éléments de T. On suppose que k est toujours supérieur ou égal à 1.
2. Modifiez la fonction précédente en tenant compte du fait que si aucun diviseur premier de n inférieur à $\sqrt{n}$ n'a été trouvé, alors n est premier

3. Ecrivez une fonction **void** `trouvePremiers`(**int** `T`[], **int** `n`) plaçant dans le tableau T les n premiers nombres premiers.
4. Ecrivez une fonction **void** `décompose`(**int** `n`, **int** `T`[], **int** `K`[]) plaçant dans le tableau K la décomposition en facteurs premiers du nombre n, sachant que T contient les N premiers nombres premiers. Par exemple, si $n = 108108$, alors on décompose n en produit de facteurs premiers de la sorte

$$108108 = 2 * 2 * 3 * 3 * 3 * 7 * 11 * 13 = 2^2 * 3^3 * 5^0 * 7^1 * 11^1 * 13^1 * 17^0 * 19^0 * \ldots * Z^0$$

(où Z est le N-ième nombre premier). On représente donc n de façon unique par le tableau K à N éléments

$$\{2, 3, 0, 1, 1, 1, 0, 0, 0, \ldots, 0\}$$

5. Ecrivez une fonction **int** `recompose`(**int** `T`[], **int** `K`[]) effectuant l'opération réciproque de celle décrite ci-dessus.
6. Ecrivez une fonction **void** `pgcd`(**int** `T`[], **int** `K`[], **int** `P`[]) prenant en paramètre les décompositions en facteurs premiers T et K de deux nombres, et plaçant dans P la décomposition en facteurs premiers du plus grand commun diviseur de ces deux nombres.
7. Ecrivez une fonction **int** `pgcd`(**int** `i`, **int** `j`) prenant en paramètres deux nombres i et j, et combinant les fonctions précédentes pour retourner le *pgcd* de i et j.

2.6.5 Statistiques

Nous souhaitons faire des statistiques sur les connexions des clients d'un site Web. Le tableau C, à n éléments, contient les identifiants des clients qui se sont connectés. Ainsi $C[i]$ contient l'identifiant du i-ème client à s'être connecté, notez bien que si un client se connecte plusieurs fois, son identifiant apparaîtra plusieurs fois dans le tableau C. Le tableau D contient les durées de connexion. Ainsi $D[i]$ est le temps de connexion de la i-ème connexion. Le but est de déterminer, pour chaque, client, son temps total de connexion.

1. Ecrire le corps du sous-programme **void** `decalageGauche`(**int** `T`[], **int** `a`, **int** `b`), ce sous-programme décale la tranche

$T[a]$	$T[a+1]$	$\ldots$	$T[b]$

d'une case vers la gauche.
2. Ecrivez le corps de la fonction **int** `calculeTempsTotalConnexionClient`(**int** `C`[], **int** `D`[], **int** `n`, **int** `i`). `calculeTempsTotalConnex` (`C`, `D`, `n`, `i`) retourne le temps total de connexion du client d'identifiant $C[i]$, on suppose que i est l'indice de la première occurence de $C[i]$ dans C.
3. Ecrivez le corps de la fonction **int** `supprimeDoublons`(**int** `C`[], **int** `D`[], **int** `n`, **int** `i`). `supprimeDoublons`(`C`, `D`, `n`, `i`) supprime toutes les occurences (d'indices strictement supérieurs à i) du client d'identifiant $C[i]$ dans C et D et retourne le nombre d'éléments supprimés. Vous devrez utiliser `decalageGauche`.
4. Ecrivez le corps de la fonction **int** `tempsTotalDeConnexion`(**int** `C`[], **int** `D`[], **int** `n`, **int** `i`). `tempsTotalDeConnexion`(`C`, `D`, `n`,`i`) place dans $D[i]$ le temps total de connexion du client $C[i]$ et élimine de C et de D toutes les autres occurences de ce client. Cette fonction doit retourner le nombre d'éléments supprimés. Vous devrez utiliser `calculeTempsTotalConnexionClient` et `supprimeDoublons`.
5. Ecrire le corps du sous-programme `inttempsTotauxDeConnexion`(**int** `C`[], **int** `D`[], **int** `n`). `tempsTotauxDeConnexion`(`C` , `D`, `n`) additionne les temps de connexions de chaque visiteur. Vous devrez modifier C et D de sorte que chaque client n'apparaîsse qu'une fois dans C et que D contienne pour chaque client son temps total. Cette fonction retourne le nombre d'éléments significatifs dans C et D. Vous devrez utiliser `tempsTotalDeConnexion`.

2.6.6 Chaînes de caractères

Exercice 12 affichage

Ecrivez une fonction **void** `afficheChaine`(**char** `s`[]) qui affiche la chaîne de caractère s sans utiliser de $\%s$.

Exercice 13 longueur

Ecrivez une fonction **int** `longueur`(**char** `s`[]) qui retourne la longueur de la chaîne de caractère s sans utiliser `strlen`.

Exercice 14 extraction

Ecrivez une fonction **char** `extrait`(**char** `s`[], **int** `n`) qui retourne le n-ème caractère de s, vous considérerez que les indices commencent à 1.

Exercice 15 substitution

Ecrivez une fonction **void** `subs`(**char** `s`[], **int** `n`, **char** `a`) qui remplace le n-ème caractère de s par a, vous considérerez que les indices commencent à 1.

2.6.7 Programmation d'un Pendu

Cette section a pour but la programmation d'un pendu, vous êtes invités à utiliser les fonctions définies ci-dessus.

Exercice 16 initialisation

Ecrire une fonction **void** `initialise`(**char** `t`[], **int** `n`) qui place dans t n caractères `'\_'`.

Exercice 17 vérification

Ecrire une fonction **void** `verifie`(**char** `t`[], **char** `k`[], **char** `c`) qui recherche toutes les occurences du caractère c dans t, soit i l'indice d'une occurrence de c dans t, alors cette fonction remplace le i-ème caractère de k par c. Par exemple, si on invoque `verifie`("`bonjour", ''b\_\_j\_\_r"`, `'o'`), alors k prendra la valeur `"bo\_jo\_r''`.

Exercice 18 Le jeu

Programmez un pendu, faites le plus simplement possible, et utilisez les fonctions ci-dessus..

2.6.8 Tris

Exercice 19 Tri à bulle

1. Ecrivez le corps de la fonction **void** `echange`(**int** `T`[], **int** `a`, **int** `b`), $echange(T, a, b)$ échange les éléments $T[a]$ et $T[b]$.
2. Ecrivez le corps de la fonction **void** `ordonne`(**int** `T`[], **int** `a`, **int** `b`), $ordonne(T, a, b)$ échange les éléments $T[a]$ et $T[b]$ si $T[a] > T[b]$. Vous utliserez le sous-programme *echange*.
3. Ecrivez le corps de la fonction **void** `bulle`(**int** `T`[], **int** `a`, **int** `b`), $bulle(T, a, b)$ place le plus grand élément de la tranche $\boxed{T[a] \mid T[a+1] \mid \ldots \mid T[b]}$ dans $T[b]$. Vous utiliserez le sous-programme *ordonne*.
4. Ecrivez le corps de la fonction **void** `triBulle`(**int** `T`[], **int** `n`), $triBulle(T, n)$ tri le tableau T à n éléments. Vous utiliserez le sous-programme *bulle*.

Exercice 20 Tri par sélection

1. Implémentez la fonction **int** `indiceDuMin`(**int** `T`[], **int** `i`, **int** `j`) retournant l'indice du plus petit élément de `T(i)` , ..., `T(j)`, c'est-à-dire de tous les éléments du tableau dont l'indice est compris entre i et j.
2. Implémentez le sous-programme **void** `placeMin`(**int** `T`[], **int** `i`, **int** `j`, **int** `k`) échangeant avec `T(k)` le plus petit élément de `T` dont l'indice est compris entre i et j.
3. Le tri par sélection est une méthode consistant à rechercher dans un tableau T à n éléments le plus petit élément du tableau et à l'échanger avec le $T(1)$. Puis à chercher dans $T(2), \ldots, T(N)$ le deuxième plus petit élément et à l'échanger avec $T(2)$, etc. Une fois un tri par sélection achevé, les éléments du tableau doivent être disposés par ordre croissant. Ecrivez le sous-programme **void** `triParSelection`(**int** `T`[], **int** `N`).

2.7 Structures

2.7.1 Prise en main

Exercice 1

Créez un type structuré *st* contenant un *char* appelé *c* et un *int* appelé *i*.

Exercice 1

Créez deux variables k et l de type *st*. Affectez-leur les valeurs $('a', 1)$ et $('b', 2)$.

Exercice 1

Affichez les valeurs de tous les champs de ces deux variables.

2.7.2 Heures de la journée

Nous utiliserons pour représenter des heures de la journée au format $hh : mm$ le type

```
typedef struct
{
  int heure;
  int minute;
}heure_t;
```

Le champ heure devra contenir une valeur de $\{0, 1, \ldots, 11\}$ et le champ minute une valeur de $\{0, 1, 2, \ldots, 59\}$. Complétez le code source suivant :

```
#include<stdio.h>

typedef struct
{
  int heure;
  int minute;
}heure_t;

/*************************************************/

/*
  Retourne uen structure initialisee avec les
  valeurs heures et minutes.
*/

heure_t creerHeure(int heures, int minutes)
{
  heure_t result = {0, 0};
  return result;
}

/*************************************************/

/*
  Convertit t en minutes.
*/

int enMinutes(heure_t t)
{
  return 0;
}
```

```
/************************************************/

/*
  Convertit la duree t en heure_t.
*/

heure_t enHeures(int t)
{
  heure_t result = {0, 0};
  return result;
}

/************************************************/

/*
  Affiche x au format hh:mm
*/

void afficheHeure(heure_t x)
{
}

/************************************************/

/*
  Additionne a et b.
*/

heure_t additionneHeures(heure_t a, heure_t b)
{
  heure_t result = {0, 0};
  return result;
}

/************************************************/

/*
  Retourne la valeur a ajouter a x pour obtenir 00:00.
*/

heure_t inverseHeure(heure_t x)
{
  heure_t result = {0, 0};
  return result;
}

/************************************************/

/*
  Soustrait b a a.
*/

heure_t soustraitHeures(heure_t a, heure_t b)
{
  heure_t result = {0, 0};
  return result;
}
```

```c
/*************************************************/

/*
  Retourne 1 si a > b, -1 si a < b, 0 si a = b.
 */

int compareHeures(heure_t a, heure_t b)
{
  return 0;
}

/*************************************************/

/*
  Retourne la plus petite des heures a et b.
*/

heure_t minHeure(heure_t a, heure_t b)
{
  heure_t result = {0, 0};
  return result;
}


/*************************************************/

/*
  Pour tester les fonctions...
*/

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

2.7.3 Répertoire téléphonique

Nous considérons les déclarations suivantes :

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

#define TAILLE_NOM 50
#define TAILLE_TEL 10
#define NB_MAX_NOMS 500

struct personne
{
        char nom[TAILLE_NOM+1];
        char tel[TAILLE_TEL+1];
};
```

Il vous est demandé de programmer un répertoire télphonique en stockant les entrées du répertoire dans un tableau de *structpersonne*. Les entrées devront être triées, il devra être possible de rechercher des entrées avec le préfixe du nom, insérer et supprimer des entrées. Prévoyez aussi la possibilité d'afficher toutes les entrées du répertoire les unes à la suite des autres. Bon courage.

2.8 Pointeurs

2.8.1 Aliasing

Exercice 1

Ecrivez un programme déclarant une variable i de type int et une variable p de type pointeur sur int. Affichez les dix premiers nombres entiers en :
— n'incrémentant que i
— n'affichant que $*p$

Exercice 2

Même exercice en
— n'incrémentant que $*p$
— n'affichant que i

Exercice 3

Déterminez ce qu'affiche ce programme, exécutez-ensuite pour vérifier.

```
#include<stdio.h>

main()
{
  int i = 4;
  int j = 10;
  int* p;
  int* q;
  p = &i;
  q = &j;
  printf("i = %d, j = %d, p = %d, q = %d\n", i, j, *p, *q);
  *p = *p + *q;
  printf("i = %d, j = %d, p = %d, q = %d\n", i, j, *p, *q);
  p = &j;
  printf("i = %d, j = %d, p = %d, q = %d\n", i, j, *p, *q);
  *q = *q + *p;
  printf("i = %d, j = %d, p = %d, q = %d\n", i, j, *p, *q);
  q = &i;
  printf("i = %d, j = %d, p = %d, q = %d\n", i, j, *p, *q);
  i = 4;
  printf("i = %d, j = %d, p = %d, q = %d\n", i, j, *p, *q);
  *q = *q + 1;
  printf("i = %d, j = %d, p = %d, q = %d\n", i, j, *p, *q);
}
```

2.8.2 Tableaux

Exercice 4 prise en main

Ecrire un programme qui place dans un tableau t les N premiers nombres impairs, puis qui affiche le tableau. Vous accéderez à l'élément d'indice i de t avec l'expression $*(t+i)$.

Exercice 5 Tri à bulle

Ecrire un sous-programme triant un tableau t avec la méthode du tri à bulle. Vous accéderez à l'élément d'indice i de t avec l'expression $*(t+i)$.

2.8.3 Exercices sans sous-programmes

Exercice 6 Tableaux de carrés

Ecrire un programme qui demande à l'utilisateur de saisir un nombre n, et qui place les n premiers nombres impairs dans un tableau t. Utilisez ensuite le fait que k^2 est la somme des k premiers nombres impairs pour calculer puis afficher les n premiers nombres carrés.

Exercice 7 Matrices et pointeurs de pointeurs

Ecrivez un programme qui demande à l'utilisateur de saisir un nombre n et qui crée une matrice T de dimensions $n \times n$ avec un tableau de n tableaux de chacun n éléments. Nous noterons t_{ij} le j-ème élément du i-ème tableau. Vous initialiserez T de la sorte : pour tous i, j, $t_{ij} = 1$ si $i = j$ (les éléments de la diagonale) et $t_{ij} = 0$ si $i \neq j$ (les autres éléments). Puis vous afficherez T.

Exercice 8 Copie de chaînes de caractères

Ecrivez un programme qui saisit proprement une chaîne de caractère s et qui la recopie dans un tableau crée avec `malloc` et de contenance correspondant exactement à la longueur de s. Vous n'utiliserez ni `strlen`, ni `strcpy`.

Exercice 9 Tableau de chaînes

Reprennez le programme précédent en saisissant successivement 4 chaînes de caractères et en les stockant dans un tableau de chaînes de caractères.

2.8.4 Allocation dynamique

Exercice 10 Prise en main

Créez dynamiquement un *int*, affectez-y la valeur 5, affichez-le, libérez la mémoire.

Exercice 11 Tableaux sur mesure

Demandez à l'utilisateur la taille du tableau qu'il souhaite créer, allouez dynamiquement l'espace nécessaire et placez-y les valeurs $\{0, \ldots n-1\}$, affichez-le et libérez la mémoire.

2.8.5 Pointeurs et pointeurs de pointeurs

Exercice 12 Tableaux sur mesure

Qu'affiche le programme suivant :

```
#include<stdio.h>

int main()
{
  long A, B, C, D;
  long *p1, *p2, *p3;
  long **pp;
  A = 10 ;
  B = 20 ;
  C = 30 ;
  D = 40;
  p1 = &A;
  p2 = &B;
  p3 = &B;
  *p1 = (*p1 + 1);
  *p3 = (*p2 + D);
  p3 = &C;
  *p3 = (*p2 + D);
```

```
  printf("A = %ld, B = %ld, C = %ld, D = %ld\n", A, B, C, D);
  pp = &p3;
  printf("%ld\n", **pp);
  return 0;
}
```

2.8.6 Passages de paramètres par référence

Exercice 13

Déterminez ce qu'affiche ce programme, exécutez-ensuite pour vérifier.

```
#include<stdio.h>

void affiche2int(int a, int b)
{
  printf("%d, %d\n", a, b);
}

void incr1(int x)
{
  x = x + 1;
}

void incr2(int* x)
{
  *x = *x + 1 ;
}

void decr1(int* x)
{
  x = x - 1;
}

void decr2(int* x)
{
  *x = *x - 1;
}

int main()
{
  int i = 1;
  int j = 1;
  affiche2int(i, j);
  incr2(&i);
  affiche2int(i, j);
  decr1(&j);
  affiche2int(i, j);
  decr2(&j);
  affiche2int(i, j);
  while(i != j)
    {
      incr1(j);
      decr2(&i);
    }
  affiche2int(i, j);
  return 0;
}
```

Exercice 14

Ecrivez le corps du sous-programme `additionne`, celui-ci doit placer dans *res* la valeur $i + j$.

```c
#include<stdio.h>

void additionne(int a, int b, int* res)
{
  /* Ecrivez le corps du sous-programme ici */


}

int main()
{
  int i = 2;
  int j = 3;
  int k;
  additionne(i, j, &k);
  printf("k = %d\n", k); // doit afficher "k = 5"
  return 0;
}
```

Exercice 15

Ecrivez le sous-programme `void puissance(int b, int n, int* res)`, qui place dans la variable pointée par *res* le résultat de b^n.

Exercice 16

Ecrivez le sous-programme `void tri(int* a, int* b, int*\ c)` qui permute les valeurs de $*a$, $*b$ et $*c$ de sorte que $*a \leq *b \leq *c$. Vous utiliserez le sous-programme `void echange(int* x, int* y)`.

2.8.7 Les pointeurs sans étoile

Le but de cet exercice est d'encapsuler les accès à la mémoire par le biais des pointeurs dans des fonctions. Ecrivez le corps des fonctions suivantes :

1. `int getIntVal(int* p)` retourne la valeur se trouvant à l'adresse p.
2. `void setIntVal(int* p, int val)` affecte la valeur *val* à la variable pointée par p.
3. `int* getTiAdr(int* t, int i)` retourne l'adresse de l'élément $T[i]$
4. `int getTiVal(int* t, int i)` retourne la valeur de l'élément $T[i]$, vous utiliserez `getTiAdr` et `getIntVal`
5. `void setTiVal(int* t, int i, int val)` affecte à $T[i]$ la valeur *val*, vous utiliserez `getTiAdr` et `setIntVal`.
6. `void swapInt(int* a, int* b)` échange les valeurs des variables pointées par a et b. Vous utiliserez `getIntVal` et `setIntVal`.
7. `void swapTij(int* t, int i, int j)` échange les valeurs $T[i]$ et $T[j]$, vous utiliserez `swapInt` et `getTiAdr`.

Ecrivez le corps de la fonction de tri `void sort(int* t, int n)` de votre choix en utilisant au mieux les fonctions ci-dessus.

2.8.8 Tableau de tableaux

Le but de cet exercice est de créer n tableaux de chacun m éléments, de placer les n pointeurs ainsi obtenus dans un tableau de pointeurs T de type $int**$ et de manier T comme un tableau à deux indices. Ecrivez les corps des fonctions suivantes :

1. `int** getTi_Adr(int** T, int i)` retourne l'adresse du pointeur vers le i-ème tableau d'*int* de T.
2. `int* getTi_(int** T, int i)` retourne le i-ème tableau d'*int* de T, vous utiliserez `getTi_Adr`.

3. **void** `setT__Adr`(**int**∗∗ `T`, **int**∗ `p`) place dans la variable pointée par T le pointeur p.
4. **void** `setTi_`(**int**∗∗ `T`, **int** `i`, **int**∗ `p`) fait de p le i-ème tableau de T, vous utiliserez `getT__Adr` et `setTi_Adr`
5. **void** `createT__`(**int**∗∗ `T`, **int** `n`, **int** `m`) fait pointer chacun des n éléments de T vers un tableau à m éléments. Vous utiliserez `setTi_Adr`.

Nous noterons T_{ij} le j-ème élément du tableau $T[i]$. Pour tous $(i, j) \in \{1, \ldots, 10\}^2$, le but est de placer dans t_{ij} la valeur $10i + j$. Vous utiliserez les fonctions **int** `getIntVal`(**int**∗ `p`), **void** `setIntVal`(**int**∗ `p`, **int** `val`), **int**∗ `getTiAdr`(**int**∗ `t`, **int** `i`), **int** `getTiVal`(**int**∗ `t`, **int** `i`) et **void** `setTiVal`(**int**∗ `t`, **int** `i`, **int** `val`) pour écrire les corps des sous-programmes ci-dessous :

1. **int**∗ `getTijAdr`(**int**∗∗ `t`, **int** `i`, **int** `j`) retourne l'adresse de T_{ij}
2. **int** `getTijVal`(**int**∗∗ `t`, **int** `i`, **int** `j`) retourne la valeur de T_{ij}
3. **void** `setTijVal`(**int**∗∗ `t`, **int** `i`, **int** `j`, **int** `val`) affecte à T_{ij} la valeur val.

2.8.9 Triangle de Pascal

En utilisant les sous-programmes ci-dessus, ecrire un programme qui demande à l'utilisateur un nombre n, puis qui crée un triangle de Pascal à $n + 1$ lignes, et qui pour finir affiche ce triangle de Pascal. Vous utiliserez les sous-programmes définis dans les questions précédentes et rédigerez des sous-programmes les plus simples possibles (courts, le moins d'étoiles possible).

2.8.10 Pointeurs et récursivité

Vous rédigerez toutes les fonctions suivantes sous forme itérative, puis récursive.

1. Ecrire une fonction **void** `initTab`(**int**∗ `t`, **int** `n`) plaçant dans le tableau t les éléments $1, 2, \ldots, n$.
2. Ecrire une fonction **void** `printTab`(**int**∗ `t`, **int** `n`) affichant les n éléments du tableau t.
3. Ecrire une fonction **int** `sommeTab`(**int**∗ `t`, **int** `n`) retournant la somme des n éléments du tableau t.
4. Ecrire une fonction (ni récursive, ni itérative) **void** `swap`(**int**∗ `t`, **int** `i`, **int** `j`) échangeant les éléments d'indices i et j du tableau t.
5. Ecrire une fonction **void** `mirrorTab`(**int**∗ `t`, **int** `n`) inversant l'ordre des éléments du tableau t.
6. Ecrire une fonction **int** `find`(**int**∗ `t`, **int** `n`, **int** `x`) retournant 1 si x se trouve dans le tableau t à n éléments, 0 sinon.
7. Ecrire une fonction **int** `distinctValues`(**int**∗ `t`, **int** `n`) retournant le nombre de valeurs distinctes du tableau t à n éléments. Le nombre de valeurs distinctes s'obtient en comptant une seule fois chaque valeur du tableau si elle apparaît plusieurs fois. Par exemple, les valeurs distinctes de $\{1, 2, 2, 3, 9, 4, 3, 7\}$ sont $\{1, 2, 3, 4, 7, 9\}$ et il y en a 6.

En utilisant les exercices sur les heures dans la section structures, vous rédigerez les fonctions suivantes sous forme récursive.

1. Ecrire une fonction **void** `afficheTabHeures`(`heure_t`∗ `t`, **int** `n`) affichant les n heures de t.
2. Ecrire une fonction **void** `initTabHeures`(`heure_t`∗ `t`, **int** `n`, `heure_t depart`, `heure_t pas`) initialisant le tableau t à n éléments comme suit : la première heure est *depart* et chaque heure s'obtient à additionnant *pas* à la précédente.
3. Ecrire une fonction `heure_t sommeTabHeures`(`heure_t`∗ `t`, **int** `n`) retournant la somme des n heures du tableau t.
4. Ecrire une fonction `heure_t minTabHeure`(`heure_t`∗ `t`, **int** `n`) retournant la plus petite des n heures du tableau t.

2.8.11 Tri fusion

Complétez le code suivant :

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

#define MOD 10000

/*****************************************/

/*
  Affiche les n elements du tableau t.
*/

void printTab(int* t, int n)
{
  if (n > 0)
    {
      printf("%d ", *t);
      printTab(t + 1, n - 1);
    }
  else
    printf("\n");
}

/*****************************************/

/*
  Place n elements aleatoires de valeurs maximales
  MOD - 1 dans le tableau t.
*/

void initTab(int* t, int n)
{
  if (n > 0)
    {
      *t = rand()%MOD;
      initTab(t + 1, n - 1);
    }
}

/*****************************************/

/*
  Retourne un tableau de n
  elements alloue dynamiquement.
*/

int* createTab(int n)
{
  int* t = (int*)malloc(sizeof(int)*n);
  if (t == NULL)
    {
      printf("no memory avalaible\n");
      exit(0);
    }
  return t;
}

/*****************************************/
```

```c
/*
  Libere la zone memoire pointee par *t
  et met ce pointeur a NULL.
*/

void destroyTab(int** t)
{
  free(*t);
  *t = NULL;
}

/*****************************************/

/*
  (Recursive)
  Retourne l'indice du plus petit
  element du tableau t a n elements.
  Affiche une erreur si le tableau est vide.
*/

int indexOfMin(int* t, int n)
{
  return 0;
}

/*****************************************/

/*
  Echange les elements *x et *y.
*/

void swap(int* x, int* y)
{
  int temp = *x;
  *x = *y;
  *y = temp;
}

/*****************************************/

/*
  Echange le plus petit element du tableau
  t a n elements avec le premier.
*/

void swapMin(int* t, int n)
{
}

/*****************************************/

/*
  (Recursive)
  Trie le tableau t a n elements avec la
  methode du tri par selection.
*/

void selectionSort(int* t, int n)
{
```

```
}

/*****************************************/

/*
  (Recursive)
  Recopie les n elements du tableau source
  a l'adresse dest.
 */

void copyTab(int* source, int* dest, int n)
{
}

/*****************************************/

/*
  (Recursive)
  Interclasse les n1 elements de source1 avec les n2 elements de source2.
  source1 et source2 sont supposes tries. L'interclassement se fait
  en disposant ces elements dans l'ordre dans le tableau dest.
*/

void shuffleTab(int* source1, int* source2, int* dest,
                int n1, int n2)
{
}

/*****************************************/

/*
  Trie les n elements de t avec la methode du tri fusion.
*/

void fusionSort(int* t, int n)
{
}

/*****************************************/

/*
  Compare les performances en temps de calcul
  des tris par selection et par fusion.
*/

int compareSorts(int firstValue, int lastValue, int step)
{
  int i;
  int start, stop;
  int *t, *q;
  srand(time(NULL));
  for(i = firstValue ; i <= lastValue ; i += step)
    {
      printf("with %d elements : \n", i);
      t = createTab(i);
      q = createTab(i);
      initTab(t, i);
      copyTab(t, q, i);
      start = time(NULL);
```

```
      selectionSort(t, i);
      stop = time(NULL);
      printf("* selection sort : %d\n", stop - start);
      destroyTab(&t);
      start = time(NULL);
      fusionSort(q, i);
      stop = time(NULL);
      printf("* fusion sort : %d\n", stop - start);
      destroyTab(&q);
    }
  return 0;
}

/*****************************************/

/*
  Pour tester les fonctions au fur et a mesure qu'elles sont
  ecrites...
*/

int main()
{
  compareSorts(10000, 500000, 1000);
  return 0;
}
```

2.9 Fichiers

2.9.1 Ouverture et fermeture

Exercice 1 `touch`

La commande shell `touch` permet de créer un fichier vide. Ecrire un programme prenant le nom d'un fichier en ligne de commande et le créant.

2.9.2 Lecture

Exercice 2 `more`

La commande shell `more` permet d'afficher le contenu d'un fichier. Ecrire un programme prenant le nom d'un fichier en ligne de commande et affichant son contenu.

2.9.3 Ecriture

Exercice 3 Alphabet

Ecrire un programme C `alphabet` permettant de créer un fichier. Le nom du fichier crée sera passé en ligne de commande et le fichier contiendra l'alphabet.

Exercice 4 Initialiser un fichier

Ecrire un programme C `createFile` permettant de créer un fichier. Le nom du fichier crée sera le premier des arguments, tous les arguments suivants seront écrits dans le fichier. Par exemple, la commande `./createFile toto.txt ceci est le contenu de mon fichier` doit créer un fichier appelé `toto.txt` et contenant le texte

```
ceci est le contenu de mon fichier
```

2.9.4 Lecture et écriture

Exercice 5 `cp`

La commande shell `cp` permet d'afficher le contenu d'un fichier. Ecrire un programme prenant en ligne de commande le nom d'un fichier source et celui d'un fichier de destination, et recopiant le contenu du fichier source dans celui de destination.

Exercice 6 Liste de noms

Ecrire deux programmes `storeNames` et `getNames`. `storeNames` demande à l'utilisateur de saisir une liste de noms et les enregistre au fur et à mesure qu'ils sont saisis dans un fichier dont le nom a été passé en ligne de commande au lancement du programme. On arrête le programme en saisissant -1. `getNames` affiche la liste des noms stockée dans le fichier dont le nom est passé en ligne de commande.

2.9.5 Enigma

Compléter le fichier source suivant :

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

#define NB_ROTORS 5
#define NB_LETTERS 26

#define FORWARD 1
#define BACKWARD 2
```

```
/*********************************************/

typedef struct rotor
{
  char permutation[NB_LETTERS];
  char permutationInverse[NB_LETTERS];
  char position;
}rotor;

/*********************************************/

typedef struct enigma
{
  rotor rotors[NB_ROTORS];
  char mirror[NB_LETTERS];
}enigma;

/*********************************************/

/*
  Dispose tous les rotors dans leur position de depart.
*/

void initialiseRotors(enigma* e)
{
}

/*********************************************/

/*
  Libere la memoire occupee par e.
*/

void enigmaDestroy(enigma* e)
{
}

/*********************************************/

/*
  Alloue la memoire et initialise les champs.
*/

enigma* enigmaCreate()
{
  return NULL;
}

/*********************************************/

/*
  Retourne le rang de letter dans l'alphabet, en indicant
  a partir de 0.
*/

char indexOfLetter(char letter)
{
  return 0;
}
```

```
/*******************************************/

/*
  Retourne la lettre d'indice index dans l'alphabet,
  'a' est d'indice 0.
 */

char letterOfIndex(char index)
{
  return 0;
}

/*******************************************/

/*
  Fait de la lettre cipherLetter l'image de la lettre
  d'indice clearIndex par un passage dans le rotor d'indice rotorIndex
  de e.
*/

void setImage(enigma* e, int rotorIndex, int clearIndex, char cipherLetter)
{
}

/*******************************************/

/*
  Fait de firstLetter le reflet de secondLetter.
 */

void setMirror(enigma* e, char firstLetter, char secondLetter)
{
}

/*******************************************/

/*
  Retourne vrai si et seulement si letter est une minuscule
  de l'alphabet.
 */

int isEncryptable(char letter)
{
  return 0;
}

/*******************************************/

/*
  Affiche les rotors et le miroir de e.
*/

void enigmaPrint(enigma* e)
{
}

/*******************************************/
```

```c
/*
  Fait pivoter le rotor d'indice indexOfRotor de e
  en modifiant sa position de depart. Retourne
  vrai ssi le rotor est revenu dans sa position
  initiale.
*/

int rotateRotor(enigma* e, int indexOfRotor)
{
  return 0;
}

/*********************************************/

/*
  Fait pivoter le jeu de rotors de e d'une position.
*/

void rotateRotors(enigma* e)
{
}

/*********************************************/

/*
  Indice d'entree de la lettre d'indice indexOfLetter
  dans le rotor r, en tenant compte de la position de
  ce rotor.
*/

int inputIndex(rotor* r, int indexOfLetter)
{
  return 0;
}

/*********************************************/

/*
  Indice de la lettre sortie a l'indice indexOfLetter
  du rotor r, en tenant compte de la position de
  ce rotor.
*/

int outputIndex(rotor* r, int indexOfLetter)
{
  return 0;
}

/*********************************************/

/*
  Fait passer la lettre d'indice indexOfLetter dans r
  dans la direction direction (FORWARD ou BACKWARD).
 */

int rotorEncrypt(rotor* r, int indexOfLetter, char direction)
{
  return 0;
}
```

```c
/*********************************************/

/*
  Fait passer la lettre d'indice indexOfLetter dans
  le miroir de e.
*/

int mirrorEncrypt(enigma* e, int indexOfLetter)
{
  return 0;
}

/*********************************************/

/*
  Chiffre letter avec e, fait ensuite pivoter les rotors
  de e..
*/

char enigmaEncrypt(enigma* e, char letter)
{
  return 0;
}

/*********************************************/

/*
  Chiffre le fichier clearFName avec e, ecrit le resultat
  dans cipherFName.
*/

void encryptFile(enigma* e, char* clearFName, char* cipherFName)
{
}

/*********************************************/

/*
  Initilialise les NB_ROTORS rotors de e avec deux
  ecrits dans le fichier rotors.
 */

void loadRotors(enigma* e, FILE* rotors)
{
}

/*********************************************/

/*
  Initilialise le miroir de e avec une ligne du fichier rotors.
 */

void loadMirror(enigma* e, FILE* rotors)
{
}

/*********************************************/
```

```c
/*
  Cree une machine enigma initialisee avec le contenu du fichier
  rotorFileName.
*/

enigma* loadFile(char* rotorFileName)
{
  return NULL;
}

/*********************************************/

/*
  Chiffre le fichier clear avec la machine enigma
  decrite dans rotors, ecrit le resultat dans cipher.
*/

void enigmaRun(char* clear, char* cipher, char* rotors)
{
  enigma* e = loadFile(rotors);
  encryptFile(e, clear, cipher);
  enigmaDestroy(e);
}

/*********************************************/

int main(int argc, char* argv[])
{
  if (argc == 4)
    enigmaRun(argv[1], argv[2], argv[3]);
  else
    printf("usage : ./enigma source cipher rotorfile\n ");
  return 0;
}
```

```
klmnopqvwxyzgabcdefhijrstu
uvwxystzabcdejklmnopqrfghi
klmnopqabcdvwxyzgstuefhijr
zgabcdefklmnopqvhijrstuwxy
tuklmnopqvwxyzgabcdefhijrs
wzgabchijrmdefynopqvstuklx
```

Que contient le message suivant ? (attention, les accents s'affichent mal...)

```
Bobuq, uynl myck ipyvp zémjbé lr lpkn zvw eywbovtssi !
```

2.10 Matrices

Exercice 1 identite

Ecrire une fonction initialisant une matrice identite d'ordre n.

Exercice 2 somme

Ecrire une fonction calculant la somme de deux matrices.

Exercice 3 transposition

Ecrire une fonction calculant la matrice transposée d'une matrice $N \times M$ passée en paramètre.

Exercice 4 produit

Ecrire une fonction calculant le produit de deux matrices $N \times M$ et $M \times P$ passées en paramètre.

Exercice 5 triangle de Pascal

Un triangle de Pascal peut être placé dans une matrice contenant des 1 sur la première colonne et la diagonale, et tel que chaque élément $m[i][j]$ de la matrice soit la somme des éléments $m[i-1][j-1]$ et $m[i-1][j]$. Seule la partie triangulaire inférieure d'un triangle de Pascal contient des valeurs significatives. Ecrire une fonction initialisant un triangle de Pascal à n lignes.

Exercice 6 Matrice marrante

Ecrire une fonction plaçant dans chaque emplacement d'indices (i, j) d'une matrice la valeur $(i+1)^j$. Vous utiliserez le fait que

$$(i+1)^j = P(0,j)i^0 + P(1,j)i^1 + P(1,j)i^1 + \ldots + P(k,j)i^k + \ldots + P(j,j)i^j$$

où $P(a,b)$ est l'élément se trouvant à la colonne a et à la ligne b du triangle de Pascal.

2.11 Initiation à la récursivité

2.11.1 Sous-programmes récursifs

Tous les sous-programmes dans les exercices ci-dessous doivent être rédigés de façon récursive, ils ne doivent pas contenir de boucle.

Exercice 1 Compte à rebours

Écrire une procédure affichant un compte à rebours en partant du nombre passé en paramètre.

Exercice 2 Somme

Écrire une fonction retournant la somme de deux nombres entiers positifs passés en paramètre sans les additionner. Vous n'utiliserez que des incrémentations et des décrémentations.

Exercice 3 Factorielle

Écrire une fonction retournant la factorielle du nombre passé en paramètre.

Exercice 4 Produit

Écrire une fonction retournant le produit de deux nombres entiers passés en paramètre sans les multiplier.

Exercice 5 Série arithmétique

Écrire une fonction retournant la somme des n premiers entiers (où n est passé en paramètre).

Exercice 6 Puissance

Écrire une fonction retournant b^n (où b et n sont tous deux passés en paramètre).

Exercice 7 Nombre de chiffres

Écrire une fonction retournant le nombre de chiffres d'un entier n passé en paramètre. *Ne pas convertir le nombre en chaîne de caractères !*

Exercice 8 p-ème chiffre

Écrire une fonction retournant le p-ième chiffre d'un entier n en partant de la droite. Si par exemple $n = 13489765123$ et $p = 4$, alors la fonction retourne 5 car il s'agit du 4 chiffre de n en partant des unités.

Exercice 9 Somme des chiffres

Écrire une fonction retournant la somme des chiffres d'un entier n passé en paramètre.

2.11.2 Morceaux choisis

Exercice 10 Recherche par dichotomie

Écrire une fonction $recherche(t[], x, i, j : entier) : booleen$ retournant vrai si et seulement si il existe un élément du tableau t qui soit égal à x et dont l'indice se trouve entre i et j. On suppose que le tableau t est trié.

Exercice 11 Suite de Fibonacci

Écrire une fonction permettant de calculer le n-ème terme de la suite de Fibonacci définie par $u_0 = 0$, $u_1 = 1$ et $u_n = u_{n-1} + n_{n-2}$. Est-ce efficace pour de grandes valeurs de n ? Est-il possible de rédiger un algorithme itératif ayant de meilleures performances ?

Exercice 12 Tri fusion

Écrire une procédure appliquant la méthode du tri fusion à un tableau passé en paramètre.

Exercice 13 Belle marquise...

Écrire une procédure affichant toutes les permutations de *Belle marquise, vos beaux yeux me font mourir d'amour.*

2.12 Listes Chaînées

2.12.1 Pointeurs et structures

Exercice 1

Créer un type `st` contenant un champ `i` de type **int** et un champ `c` de type **char**. Déclarez une variable `p` de type `st*`, allouez dynamiquement une variable de type `st` et faites pointer `p` dessus. Affectez à cette variable les valeurs 5 et `'a'`. Affichez-les, libérez la mémoire.

2.12.2 Maniement du chaînage

Exercice 1 prise en main

Etant donné le programme

```
#include<stdio.h>

/*************************************************************/

typedef struct maillon
{
        int data;
        struct maillon* next;
}maillon;

/*************************************************************/

int main()
{
  maillon m, p;
  maillon* ptr;
  m.data = 1;
  m.next = &p;
  p.data = 2;
  p.next = NULL;
  for (ptr = &m ; ptr != NULL ; ptr = ptr->next)
    printf("data = %d\n", ptr->data);
  return 0;
}
```

Reprennez ce programme : créez deux maillons q et r en renseignant les champ *data* aux valeurs respectives 3 et 4, renseignez les valeurs *next* des maillons de sorte que la boucle *for* affiche

```
data = 1
data = 2
data = 3
data = 4
```

Exercice 2 tableaux

Reprennez le programme

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

#define N 10

/*************************************************************/
```

```c
typedef struct maillon
{
        int data;
        struct maillon* next;
}maillon;

/*************************************************************/

void printData(maillon* ptr)
{
  for( ; ptr != NULL ; ptr = ptr->next)
    printf("data = %d\n", ptr->data);
}

/*************************************************************/

int main()
{
  maillon* l;
  int i;
  l = (maillon*)malloc(N * sizeof(maillon));
  if (l == NULL)
    return -1;
  l->data = 0;
  for(i = 1 ; i < N ; i++)
    {
      (l + i)->data = i;
      (l + i - 1)->next = l + i;
    }
  (l + N - 1)->next = NULL;
  printData(l);
  free(l);
  return 0;
}
```

Modifiez le chaînage de sorte que les maillons soient disposés dans l'ordre inverse et que, par conséquent, ce programme affiche :

```
data = 9
data = 8
data = 7
data = 6
data = 5
data = 3
data = 4
data = 2
data = 1
data = 0
```

2.12.3 Opérations sur les listes chaînées

Pour les exercices suivants, vous utiliserez ce fichier source :

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

#define N 10

/*************************************************************/
```

```c
typedef struct maillon
{
  int data;
  struct maillon* next;
}maillon;

/*************************************************************/

void printLL(maillon* ptr)
{
  for( ; ptr != NULL ; ptr = ptr->next)
    printf("data = %d\n", ptr->data);
}

/*************************************************************/

maillon* creeMaillon(int n)
{
  maillon* l;
  l = (maillon*)malloc(sizeof(maillon));
  if(l == NULL)
    exit(0);
  l->data = n;
  l->next = NULL;
  return l;
}

/*************************************************************/

maillon* insereFinLL(maillon* l, int n)
{
  return NULL;
}

/*************************************************************/

maillon* copyLL(maillon* source)
{
  return NULL;
}

/*************************************************************/

maillon* reverseLL(maillon* l)
{
  return NULL;
}

/*************************************************************/

maillon* insereDebutLL(maillon* l, int n)
{
  maillon* first = creeMaillon(n);
  first->next = l;
  return first;
}

/*************************************************************/
```

```
maillon* initLL(int n)
{
  maillon* l = NULL;
  int i;
  for(i = n - 1 ; i >= 0 ; i--)
    l = insereDebutLL(l, i);
  return l;
}

/*************************************************************/

void freeLL(maillon* l)
{
  maillon* n;
  while(l != NULL)
    {
      n = l->next;
      free(l);
      l = n;
    }
}

/*************************************************************/

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

Exercice 3 ajout d'un élément à la fin

Erire le corps de la fonction suivante :

```
maillon* insereFinLL(maillon* l, int n)
```

Vous insérerez un maillon contenant la valeur n à la fin de la liste dont le premier élément est pointé par l. Vous retournerez un pointeur vers le premier élément de la liste.

Exercice 4 copie d'une liste chaînée

Erire le corps de la fonction suivante :

```
maillon* copyLL(maillon* source)
```

Vous copierez la liste l et retournerez un pointeur vers le premier élément de la copie. Vous avez le droit d'utiliser `insereFinLL`.

Exercice 5 inversion de l'ordre des éléments

Ecrivez un sous-programme qui inverse l'ordre des éléments d'une liste chaînée, et qui retourne un pointeur sur le premier élément de la liste nouvellement formée.

2.12.4 Listes doublement chaînées

Pour les exercices suivants, vous pouvez utiliser ce fichier source :

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

```c
#define N 10

/*************************************************************/
typedef struct dmaillon
{
  int data;
  struct dmaillon* previous;
  struct dmaillon* next;
}dmaillon;

/*************************************************************/

typedef struct dLinkedList
{
  struct dmaillon* first;
  struct dmaillon* last;
}dLinkedList;

/*************************************************************/

/*
  Affiche les elements d'une liste chainee.
 */

void printDLL(dLinkedList* dl)
{
}

/*************************************************************/

/*
  Libere tous les maillons, puis libere dl.
 */

void freeDLL(dLinkedList* dl)
{
}

/*************************************************************/

/*
  Alloue la memoire pour une dLinkedList,
  initialise les pointeurs a NULL
*/

dLinkedList* makeDLL()
{
  return NULL;
}

/*************************************************************/

/*
  Cree un maillon contenant la valeur n.
 */

dmaillon* makeDMaillon(int n)
{
```

```
  return NULL;

}

/*************************************************************/

/*
  Accroche le maillon m a la fin de la liste chainee dl
 */

void appendDMaillonDLL(dLinkedList* dl, dmaillon* m)
{
}

/*************************************************************/

/*
  Accroche le maillon m au debut de la liste chainee dl
 */

void pushDMaillonDLL(dLinkedList* dl, dmaillon* m)
{
}

/*************************************************************/

/*
  Ajoute a la fin de dl un maillon contenant la valeur n.
 */

void appendIntDLL(dLinkedList* dl, int n)
{
}

/*************************************************************/

/*
  Ajoute au debut de dl un maillon contenant la valeur n.
 */

void pushIntDLL(dLinkedList* dl, int n)
{
}

/*************************************************************/

/*
  Place dans la liste doublement chainee
  les valeurs {0, ..., n}
 */

void initDLL(dLinkedList* dl, int n)
{
}

/*************************************************************/

/*
  Inverse l'ordre des elements de dl.
```

```
 */

void reverseDLL(dLinkedList* dl)
{
}

/*************************************************************/

/*
  Retourne une copie de source.
 */

dLinkedList* copyDLL(dLinkedList* source)
{
  return NULL;
}

/*************************************************************/

/*
  Concatene fin a la suite de debut, vide la liste fin.
 */

void concatDLL(dLinkedList* debut, dLinkedList* fin)
{
}

/*************************************************************/

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

Nous définissons comme suit une liste doublement chaînée

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

#define N 10

/*************************************************************/

typedef struct dmaillon
{
  int data;
  struct dmaillon* previous;
  struct dmaillon* next;
}dmaillon;

/*************************************************************/

typedef struct dLinkedList
{
  struct dmaillon* first;
  struct dmaillon* last;
}dLinkedList;
```

Implémentez les fonctions suivantes :

1. **void** `printDLL(dLinkedList* dl)` affiche les éléments d'une liste chaînée.
2. **void** `freeDLL(dLinkedList* dl)` libère tous les maillons, puis libère `dl`.
3. `dLinkedList* makeDLL()` alloue la mémoire pour une `dLinkedList`, initialise les pointeurs à `NULL`
4. `dmaillon* makeDMaillon(int n)` crée un maillon contenant la valeur `n`.
5. **void** `appendDMaillonDLL(dLinkedList* dl, dmaillon* m)` accroche le maillon `m` à la fin de la liste chaînée `dl`
6. **void** `pushDMaillonDLL(dLinkedList* dl, dmaillon* m)` accroche le maillon `m` au début de la liste chaînée `dl`
7. **void** `appendIntDLL(dLinkedList* dl, int n)` ajoute à la fin de `dl` un maillon contenant la valeur `n`.
8. **void** `pushIntDLL(dLinkedList* dl, int n)` ajoute au début de `dl` un maillon contenant la valeur `n`.
9. **void** `initDLL(dLinkedList* dl, int n)` place dans la liste doublement chaînée les valeurs $\{0, ..., n-1\}$
10. **void** `reverseDLL(dLinkedList* dl)` inverse l'ordre des éléments de `dl`.
11. `dLinkedList* copyDLL(dLinkedList* source)` retourne une copie de `source`.
12. **void** `concatDLL(dLinkedList* debut, dLinkedList* fin)` concatène `fin` à la suite de `debut`, vide la liste `fin` .

2.12.5 Fonctions récursives et listes chaînées

Complétez le code sources suivant. Les boucles sont interdites !

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

/*
  Dans toutes les fonctions à partir d'insere, il est interdit de
  creer des maillons, toutes ces operations doivent se faire
  par modification du chainage et non par recopie.
*/

typedef struct lls
{
  int data;
  struct lls* next;
}ll;

/*****************************************************/

/*
  Alloue dynamiquement et initialise un maillon avec les
  valeurs data et next, retourne son adresse.
*/

ll* cree(int data, ll* next)
{
  ll* maillon = (ll*)malloc(sizeof(ll));
  if (maillon == NULL)
  {
    printf("Plus de mémoire");
    exit(1);
  }
  maillon->data = data;
  maillon->next = next;
  return maillon;
}

/*****************************************************/

/*
  Affiche le maillon l
```

```
*/

void affiche(ll* l)
{
  printf("%d -> ", l->data);
}

/*****************************************************/

/*
  Affiche, dans l'ordre, tous les maillons de la liste l.
*/

void afficheTout(ll* l)
{
  if (l != NULL)
    {
      affiche(l);
      afficheTout(l->next);
    }
  else
    printf("\n");
}

/*****************************************************/

/*
  Affiche en partant de la fin tous les maillons
  de la liste l.
*/

void afficheALEnvers(ll* l)
{
  if (l != NULL)
    {
      afficheALEnvers(l->next);
      affiche(l);
    }
}

/*****************************************************/

/*
  Detruit tous les maillons de la liste *l, met ce pointeur
  a NULL.
*/

void detruit(ll* l)
{
  if (l != NULL)
    {
      detruit(l->next);
      free(l);
    }
}

/*****************************************************/

/*
```

```
  Retourne la liste n -> n-1 -> ... -> 2 -> 1
*/

ll* entiersALEnvers(int n)
{
  if (n < 1)
    return NULL;
  return cree(n, entiersALEnvers(n - 1));
}

/*****************************************************/

/*
  Retourne la liste 1 -> 2 -> ... -> n
*/

ll* entiers(int n)
{
  return NULL;
}

/*****************************************************/

/*
  Insere le maillon x dans la liste l, supposee triee.
*/

ll* insere(ll* l, ll* x)
{
  return NULL;
}

/*****************************************************/

/*
  Tri la liste l avec la methode du tri par insertion, retourne
  l'adresse du premier element de la liste triee.
*/

ll* triInsertion(ll* l)
{
  return NULL;
}

/*****************************************************/

/*
  Repartit les elements de l entre les listes l1 et l2.
  ex : l = 1 -> 2 -> 3 -> 4 -> 5 nous donne
  l1 = 5 -> 3 -> 1et l2 = 4 -> 2
*/

void split(ll* l, ll** l1, ll** l2)
{
}

/*****************************************************/

/*
```

```
  Retourne l'interclassement des listes l1 et l2, supposees triees.
*/

ll* interclasse(ll* l1, ll* l2)
{
  return NULL;
}

/*******************************************************/

/*
  Trie l avec la methode du tri fusion, retorune l'adresse
  du premier element de la liste triee.
*/

ll* triFusion(ll* l)
{
  return NULL;
}

/*******************************************************/

/*
  Pour tester les fonctions...
*/

int main()
{
  ll* maillon = entiersALEnvers(10);
  afficheTout(maillon);
  afficheALEnvers(maillon);
  detruit(maillon);
  return 0;
}
```