```c
#include <stdio.h>
int main (void)
{
  printf ("Hello world!\n");
  return 0;
}
```

Langage C

École d'ingénieurs Sup Galilée
Formation Télécom et Réseaux – 1ère année
Université Paris Nord

Table des matières

1 Introduction **7**
1.1 Objectifs du cours 7
1.2 Conseils généraux 8
1.3 Structure d'un programme 9
1.4 Bibliographie conseillée 11

2 Rappels de langage C **13**
2.1 Types, variables et opérateurs 13
2.2 Structures de contrôle 15
2.2.1 Exécution conditionnelle 16
2.2.2 Boucles 18
2.3 Tableaux et pointeurs 20
2.3.1 Déclaration d'un tableau 20
2.3.2 Notion de pointeur 21
2.3.3 Affectation d'un tableau 23
2.3.4 Chaîne de caractères 24
2.3.5 Tableau de tableaux 25
2.4 Fonctions 26
2.4.1 Déclaration d'une fonction 26
2.4.2 Définition d'une fonction 27
2.4.3 Passage des paramètres 28
2.4.4 Fonctions récursives 29
2.5 Exercices 31

3 Structures de données **33**
3.1 Déclaration d'une structure 34
3.1.1 Alias de nom de structure 35
3.1.2 Manipulation des champs 36
3.1.3 Pointeur vers une structure 37
3.2 Structure de structure 37
3.3 Tableau de structure 38
3.4 Interface d'accès à une structure 39
3.4.1 Accès aux champs 40
3.4.2 Modification des champs 40
3.4.3 Entrées / Sorties 41
3.4.4 Fonctions annexes 41
3.5 Exercices 43

4 Stockage dynamique **45**
4.1 Allocation de mémoire 46
4.1.1 Allocation simple 46
4.1.2 Allocation et initialisation 47
4.1.3 Libération de la mémoire 48
4.1.4 Ré-allocation de la mémoire 50
4.2 Tableaux dynamiques 51
4.3 Conteneurs génériques 52
4.3.1 Listes chaînées 52
4.3.2 Piles 54
4.3.3 Arbres 55
4.4 Exercices 56

5 Bibliothèque standard **57**
5.1 Afficher une variable à l'écran 57
5.2 Lire des informations au clavier 59
5.2.1 Lecture formatée 59
5.2.2 Lecture non formatée 61
5.2.3 Lecture d'une chaîne au format csv 62
5.3 Manipuler un fichier 63
5.4 La fonction *main* 65

Liste des tableaux

2.1 Les types courants de données. 14
2.2 Codes ASCII. 15
2.3 Opérateurs. 16

5.1 Codes de format de la fonction `printf`. 58

Chapitre 1

Introduction

1.1 Objectifs du cours

Ce cours a pour ambition d'introduire des concepts « avancés » de programmation, dont certains sont spécifiques au langage C et d'autres sont utilisés dans d'autres langages. L'ensemble des notions présentées ici constituent aussi un socle permettant de mieux appréhender la programmation *objet* propre aux langages plus récents comme le C++.

La notion centrale vue dans ce cours est celle de l'organisation de l'espace mémoire d'un programme, qui en langage C utilise essentiellement deux outils :

- les structures de données,
- et les pointeurs.

Nous allons donc nous concentrer sur ces deux notions à travers les exemples, contenus et exercices proposés dans le cours. Bien que très simples ces deux outils permettront de mettre en œuvre des concepts avancés qui seront abordés dans la partie 4.3 et utilisés lors des projets.

Le second objectif du cours est d'apprendre à tirer parti d'un

environnement de développement intégré (ou IDE, acronyme anglais pour *Integrated Development Environment*). Cet outil comporte (en général) :

- un éditeur de texte, avec une coloration syntaxique appropriée [1],
- des outils de correction des erreurs de programmation,
- un ensemble d'outils permettant de créer un exécutable : compilateur, éditeur de lien, ...

Pour des raisons de cohérence avec les cours de deuxième et troisième année en formation d'ingénieur en télécommunications et réseaux de sup'Galilée, l'environnement de développement que nous allons utiliser est Visual Studio 2013, développé par Microsoft. Ce logiciel existe dans une version gratuite, Visual Studio 2013 Express, que vous pouvez télécharger et installer sur votre propre ordinateur [2].

1.2 Conseils généraux

Tout ingénieur, quelque soit son domaine d'activité doit avoir aujourd'hui une bonne compréhension des outils informatiques. Cette connaissance ne peut s'acquérir que par la pratique de la programmation. C'est pourquoi vous trouverez à chaque chapitre de ce cours des exercices qu'il est impératif de faire, au fur et à mesure de la progression du cours et en complément des exercices proposés en séances de travaux pratiques.

Afin de permettre un suivi de vos progrès, vous trouverez sur l'espace numérique de travail du cours [3], des espaces ou renseigner les réponses à ces exercices et obtenir des corrections. Vous êtes fortement encouragés à utiliser ces outils mis à votre disposition pour améliorer vos compétences de programmeurs !

Enfin il faut garder en tête que l'apprentissage d'un langage de

1. voir l'exemple de programme de la page de garde...
2. En veillant à télécharger la « bonne » version, c'est à dire la *Windows Desktop*.
3. `http://ent.univ-paris13.fr`

programmation doit se faire comme l'apprentissage d'une langue étrangère ! Il y a des règles de **syntaxe** à respecter, une **grammaire** à apprendre et des *tournures à connaître ou à éviter*. En ce sens, le langage de programmation est le moyen de communiquer avec les ordinateurs, et comme pour toute langue, une pratique régulière est nécessaire afin de maintenir son niveau !

1.3 Structure d'un programme

Avant de rentrer dans le vif du sujet du cours, il est utile de se rappeler de ce que signifie exactement la notion de « programme informatique ». Pour les besoins du cours il suffit d'imaginer une machine qui lirait sur un support des instructions à effectuer, l'une après l'autre. Dans un ordinateur cette machine est le processeur et le support est la mémoire.

Le langage C étant déjà assez complexe, chacune des lignes d'instructions en C va donner lieu après compilation à une ou plusieurs instructions élémentaires [4] qui vont être exécutées par la machine. L'exemple ci-dessous donne une idée de la complexité de l'étape de compilation.

Exemple 1.1. Partant d'un programme simple en C :

```
#include <stdio.h>
int main (void)
{
  printf ("Hello world!\n");
  return 0;
}
```

le compilateur produit un exécutable binaire, dont le contenu est représenté ci-dessous :

```
Contents of section .text:
 0000 554889e5 bf000000 00e80000 0000b800  UH..............
 0010 0000005d c3                          ...].
```

4. Ces instructions sont données au processeur dans un langage *basique* appelé *assembleur*.

```
Contents of section .rodata:
 0000 48656c6c 6f20776f 726c6421 00        Hello world!.
```

Les informations contenues dans le programme binaire s'interprètent de la manière suivante :

- La première colonne indique une adresse qui permet de se repérer dans le code,
- la partie centrale (séries de chiffres en hexadécimal) contient le code binaire,
- la partie à droite donne une transcription en *ASCII* du code binaire.

On peut constater que la section `.rodata` contient la chaîne de caractères utilisée dans le programme, tandis que la section `.text` ne semble pas contenir d'information pertinente. En fait il faut décoder le contenu de cette section et une fois traduit en *assembleur* :

```
1  Disassembly of section .text:
2  0000000000000000 <main>:
3     0:   55                          push   %rbp
4     1:   48 89 e5                    mov    %rsp,%rbp
5     4:   bf 00 00 00 00              mov    $0x0,%edi
6                             5: R_X86_64_32  .rodata
7     9:   e8 00 00 00 00              callq  e <main+0xe>
8                             a: R_X86_64_PC32        puts-0x4
9     e:   b8 00 00 00 00              mov    $0x0,%eax
10   13:   5d                          pop    %rbp
11   14:   c3                          retq
```

On constate qu'elle contient le code de la fonction `main` qui va prendre la chaîne de caractères (lignes 5 et 6) et appelle une fonction d'affichage (ligne 7 et 8) avant de terminer le programme (ligne 11). Les instructions `push`, `pop` et `mov` agissent directement sur la mémoire (ici les registres du processeur).

On peut noter (ligne 8) que dans ce cas particulier le compilateur a choisi de substituer l'appel à la fonction `printf` par un appel à une fonction plus simple `puts` : il a optimisé pour nous le code !

Chaque instruction va alors lire et/ou modifier le contenu de la mémoire, ce qui, par exemple, permet d'écrire un message sur l'écran ou bien lire les touches frappées sur le clavier. C'est pour cela que nous allons insister particulièrement sur la notion de pointeur qui permet, en langage C, de manipuler directement le contenu de la mémoire de l'ordinateur et donc d'effectuer simplement beaucoup d'opérations complexes.

1.4 Bibliographie conseillée

On trouve maintenant facilement et gratuitement sur internet des ouvrages de référence qui permettent de revoir les principes de base du langage C. Parmi ceux-ci on peut conseiller la lecture de :

- `http://fr.wikibooks.org/wiki/Programmation_C` un cours relativement complet d'introduction au C,
- `http://c.learncodethehardway.org/book/` un cours d'apprentissage du C en ligne (en anglais) avec beaucoup d'exercices. Contient beaucoup d'astuces pratiques et de conseils utiles,
- `http://www.cplusplus.com/reference/clibrary/` la référence complète des bibliothèques de fonctions standards (en anglais), avec, pour chaque fonction, des exemples d'utilisation.

Par contre on déconseillera d'avoir recours au forums de discussion où, souvent, les conseils proposées sont obsolètes, incomplets voir erronés.

Chapitre 2

Rappels de langage C

Ce chapitre se compose de plusieurs rappels de la syntaxe et des pratiques du langage C. Il ne constitue pas en lui même un cours de C et ne fait que rappeler les éléments nécessaires à la compréhension des chapitres suivants. Pour des révisions plus complètes le lecteur est invité à consulter les références données dans la section 1.4.

2.1 Types, variables et opérateurs

Le langage C est un langage typé, c'est à dire que chaque variable doit être déclarée avec un type. Ce type sert ensuite à donner un sens aux opérations qui sont appliquées à la variable : par exemple l'opérateur de division n'a pas le même sens pour des entiers ou des nombres réels.

Chaque type de donnée est stocké dans un espace mémoire de taille fixée, mesurée en octet, qui dépend du système d'exploitation ou du processeur utilisé. Ces types de base peuvent être changés par un *modificateur*, qui agit soit sur la taille de stockage soit sur la manière dont le type doit être interprété. Les types de base et les types modifiés sont listés dans le tableau 2.1.

type	interprétation	taille	valeurs
`char`	caractère	1	-128 à 127
`unsigned char`		1	0 à 255
`short int`		2	-32768 à 32767
`unsigned short int`		2	0 à 65535
`int`	nombre entier	2 ou 4	
`unsigned int`		2 ou 4	
`long int`		4	-2147483648 à 2147483647
`unsigned long int`		4	0 à 4294967296
`float`	nombre réel	4	
`double`		8	
`long double`		12	

TABLE 2.1 – Les types courants de données. Les tailles sont données en octets. Elles dépendent de l'architecture du processeur et du système d'exploitation.

Lorsque l'on a besoin d'une variable dans le programme il faut la déclarer avec une instruction sous la forme :

```
nom_type nom_variable;
```

Une fois les variables déclarées elles peuvent être manipulées en utilisant des opérateurs. Les opérateurs utilisés en langage C sont listés dans le tableau 2.3. Pour éviter les ennuis il vaut mieux que les types des données manipulées avec les opérateurs soient les mêmes. Au besoin on peut utiliser un trans-typage pour changer le type d'une variable.

Exemple 2.1. Trans-typage implicite et explicite.

```
1 int x = 1;
2 double y;
3 y = x / 10;
4 y = (double)x / 10;
```

Après l'exécution de la troisième ligne la variable `y` contient la valeur `0`, car la division de l'entier `x` par la constante `10` se fait au sens des entiers. A la quatrième ligne l'entier `x` est ré-interprété comme un nombre réel et la division se fait au sens des réels, `y` contient donc la valeur `0.1`.

valeur	symbole	remarque	valeur	symbole
0	'\0'		48 à 57	0 à 9
9	'\t'	tabulation	58	' :'
10	'\n'	saut de ligne	59	';'
13	'\r'	retour chariot	60	'<'
32	' '	espace	61	'='
33	'!'		62	'>'
34	'"'		63	'?'
35	'#'		64	'@'
36	'$'		65 à 90	'A' à 'Z'
37	'%'		91	'['
38	'&'		92	'\'
39	"'"		93	']'
40	'('		94	'^'
41	')'		95	'_'
42	'*'		96	'`'
43	'+'		97 à 122	'a' à 'z'
44	','		123	'{'
45	'-'		124	'|'
46	'.'		125	'}'
47	'/'		126	'~'

TABLE 2.2 – Codes ASCII des caractères courants.

2.2 Structures de contrôle

Les structures de contrôle permettent de conditionner l'exécution d'une partie du programme : elles sont indispensables pour effectuer des taches complexes.

	Opérateurs unaires		Opérateurs binaires
`*`	déréférencement	`+ -`	Addition, soustraction
`&`	référencement	`* /`	Multiplication, division
`++`	incrémentation	`%`	modulo
`--`	décrémentation	`<< >>`	décalage de bit
`.`	membre de	`> >=`	supérieur
`->`	membre de (pointeur)	`< <=`	inférieur
`!`	négation booléenne	`== !=`	égalité/différence
`~`	négation binaire	`& | ^`	et/ou/ou exclusif binaire
`(type)`	transtypage	`&& ||`	et/ou logique
`? :`	si ... alors ... sinon ...		

TABLE 2.3 – Principaux opérateurs du langage C. Colonne de gauche les opérateurs « unaires », colonne de droite les opérateurs binaires, dernière ligne : le seul opérateur ternaire. Attention certains symboles peuvent avoir des significations différentes selon le contexte. Certains des opérateurs binaires peuvent se contracter sous la forme *opération–affectation*, par exemple : `+=`.

2.2.1 Exécution conditionnelle

L'exécution conditionnelle consiste à effectuer un test logique et suivant le résultat du test (vrai ou faux) effectuer des instructions différentes. En langage C, elle est implémentée par la syntaxe :

```
if (condition)
{
  instructions si vrai;
}
else
{
  instructions si faux;
}
```

Exemple 2.2. Dans cet exemple on teste la valeur de la variable `x` afin

d'éviter une division par zéro.

```
double divide (double y, double x)
{
  double res = 0;
  if (x != 0)
    res = y / x;
  else
  {
    fprintf (stderr, "Error: division by 0!\n");
  }
  return res;
}
```

Remarque 2.1. Dans certains cas la syntaxe peut être allégée :
- si il n'y a pas d'instructions à effectuer quand la condition est fausse on peut omettre le code `else {}`,
- lorsqu'il n'y a qu'une instruction à exécuter on peut omettre les accolades.

Dans les cas simples on peut utiliser une forme compacte d'instruction conditionnelle avec la syntaxe :

```
(condition)?(operation si vrai):(operation si faux);
```

Remarque 2.2. L'instruction conditionnelle compacte ne permet d'exécuter qu'une seule instruction par cas.

Exemple 2.3. Utilisation de l'instruction conditionnelle compacte pour calculer la valeur absolue d'un réel.

```
double x;
...
x = (x<0)?-x:x;
```

Dans les situations où un grand nombre de cas différents sont à traiter et où la condition consiste à comparer des valeurs *entières*, on peut utiliser la syntaxe :

```
switch (condition)
{
  case valeur_1:
    instructions dans le cas 1;
    break;
  ...
  case valeur_N:
    instructions dans le cas N;
    break;
  default:
    instructions par defaut;
}
```

Exemple 2.4. Utilisation de l'instruction `switch`.

```
char c;
...
switch (c)
{
  case 'a':
  case 'e':
  case 'u':
  case 'i':
  case 'o':
  case 'y':
    printf ("Voyelle !\n");
    break;
  default:
    printf ("Consonne !\n");
}
```

Ici on groupe le traitement du cas « voyelle » en plaçant plusieurs instructions `case` à la suite, sans utiliser de `break`.

2.2.2 Boucles

Les instructions de boucle permettent de répéter plusieurs fois un bloc d'instructions, autant de fois que nécessaire, c'est à dire tant qu'une condition est vraie. Dans le cas le plus simple on veut répéter une action un certain nombre de fois et on associe à la boucle un compteur que l'on incrémente à chaque itération.

Ce type de boucle s'obtient avec la syntaxe :

```
for (initialisation;test fin;incrementation)
{
  instructions a repeter;
}
```

Exemple 2.5. Utilisation d'une boucle pour le calcul du factoriel d'un entier ($n!$).

```
int factoriel(int n)
{
  int res = 1, i;
  for (i=2; i <= n; i++)
    res *= i;
  return res;
}
```

Parfois le nombre d'itérations n'est pas connu à l'avance ou bien la condition d'arrêt est non triviale : dans ce cas on préfère utiliser la forme :

```
while(test fin)
{
  instructions a repeter;
}
```

plus souple mais plus risquée à mettre en œuvre, puisque l'utilisateur est moins contraint par la syntaxe. En particulier il faut faire attention à bien choisir la condition `test fin` pour être sur que la boucle se termine à un moment donné (en général lorsque la tâche à effectuer est accomplie).

Exemple 2.6. Utilisation d'une boucle pour calculer le plus grand commun dénominateur de deux entiers.

```
int pgcd (int a, int b)
{
  while (a != b)
  {
    if (a > b)
      a = a - b;
    else
```

```
        b = b - a;
    }
    return a;
}
```

Dans cet exemple d'algorithme il faut se convaincre que le test de fin choisi est le bon, ce qui n'est pas *a priori* évident.

On trouve aussi la forme :

```
do
{
  instructions a repeter;
}
while(test fin);
```

qui garantit que le bloc d'instructions est exécuté au moins une fois.

Remarque 2.3. La syntaxe de gestion des boucles est complétée par les instructions :

- `break`; qui permet de sortir de la boucle,
- `continue`; qui permet de passer à l'itération suivante.

2.3 Tableaux et pointeurs

Les tableaux sont les espaces de stockage de données les plus simples : ils permettent d'organiser la mémoire disponible afin de l'utiliser efficacement dans un programme.

2.3.1 Déclaration d'un tableau

Lorsque l'on doit stocker un grand nombre de données de même type, plutôt que de définir explicitement une variable pour chaque donnée[1], on préfère stocker l'ensemble de ces variables dans un *espace mémoire contigu*, appelé *tableau*. Il suffit alors de spécifier l'adresse du tableau (c'est à dire l'emplacement de la première case

1. ce qui ne serait ni pratique, ni possible !

mémoire) et sa taille pour réserver l'espace mémoire correspondant. Cette déclaration de tableau s'écrit :

```
type nom_tableau[taille];
```

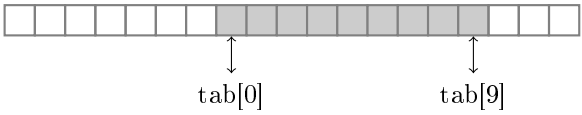

FIGURE 2.1 – Mémoire utilisée par un tableau de taille 10, suite à une allocation statique : une suite contiguë de cases mémoire est réservée pour le tableau.

Remarque 2.4. La taille du tableau doit nécessairement être une constante connue au moment de la compilation. C'est pour cela qu'on utilise rarement une variable pour spécifier la taille, si on veut lui donner un nom, on utilisera plutôt une macro (voir l'exemple ci-dessous). Pour cette raison on appelle ce type de tableau un tableau *statique*.

Exemple 2.7. Déclaration d'un tableau.

```
#define SIZE 100
int x[10];
double y[SIZE];
```

Dans les deux cas présentés ci dessus la taille du tableau est une *valeur constante*.

2.3.2 Notion de pointeur

La variable `nom_tableau` contient l'adresse d'un tableau : c'est à dire un nombre entier de type `unsigned long int`, donnant l'adresse (en octet, à partir du début) d'une « case » de mémoire physique. Ce type de variable est appelé *pointeur*. Dans le cas d'une déclaration

de tableau ce pointeur ne peut pas être modifié (il est constant), car l'adresse physique d'un tableau statique ne peut être changée au cours de l'exécution du programme.

La notion de pointeur n'est cependant pas restreinte aux tableaux : on peut définir un pointeur vers n'importe quel type de variable, puisque toute variable possède une adresse ! Cette déclaration s'effectue ainsi :

```
type * nom_du_pointeur;
```

Dans ce cas la variable `nom_du_pointeur` contient l'adresse d'une variable de type `type`.

Remarque 2.5. Il est nécessaire de spécifier le type de la variable vers laquelle le pointeur pointe, puisqu'une variable n'occupe pas nécessairement une seule case de mémoire : le type du pointeur indique donc le nombre de cases à lire lorsqu'on accède à l'adresse. Cependant tous les pointeurs sont « compatibles » entre eux puisqu'ils ne contiennent qu'un chiffre (une adresse mémoire). C'est pour cela qu'il existe un type de pointeur spécifique, `void *`, qui est un pointeur vers un type non spécifié et donc de facto compatible avec n'importe quel type de données. C'est à l'utilisateur de spécifier le type en temps voulu lors de l'accès au contenu de l'adresse.

Pour faciliter l'utilisation des pointeurs il existe deux opérateurs spécifiques :

- l'opérateur d'*indirection* `&(nom_variable)` qui calcule l'adresse d'une variable ;
- l'opérateur *déréférencement* `*(nom_pointeur)` qui donne accès au contenu pointé par le pointeur.

Remarque 2.6. L'opération de déréférencement suppose de connaître le type de donnée pointée et donc ne peut pas s'appliquer sur un pointeur de type `void *` sans trans-typage explicite préalable.

Exemple 2.8. Utilisation de pointeurs.

```
1 int x = 10;
2 int *px = &x;
```

```
3 void *py = &x;
4 x = x + *px + *((int *)py);
```

Dans cet exemple on affecte à `px` et à `py` l'adresse de la variable `x`. A la dernière ligne `*px` vaut `x` et il faut spécifier un type de pointeur avant de déréférencer `py`.

Puisqu'un pointeur contient une adresse il est possible de l'utiliser pour *calculer* l'adresse de variables, en particulier lorsqu'elles sont stockées dans un tableau. Pour cela on utilise des opérations d'arithmétique de pointeur : augmenter la valeur d'un pointeur de une unité consiste à pointer vers l'adresse suivant la variable actuelle. Puisque les variables ont des tailles mémoire différentes cette opération dépend du type du pointeur.

Exemple 2.9. Arithmétique de pointeur.

```
int x[10];
int *px = x;
px += 5;
```

A la fin de cette séquence d'instructions `px` contient l'adresse de la cinquième case du tableau.

Remarque 2.7. La manipulation d'adresse doit se faire avec précautions : en effet lors de l'exécution du programme le système d'exploitation lui alloue une certaine plage mémoire, correspondant à une certaine plage d'adresses. Essayer de lire ou écrire à une adresse en dehors de cette plage va provoquer une erreur d'accès mémoire, appelée erreur de segmentation, et souvent l'arrêt inopiné du programme.

2.3.3 Affectation d'un tableau

Jusqu'à présent nous avons vu comment déclarer un tableau (réserver l'espace mémoire associé) mais pas comment l'initialiser (remplir le contenu de chaque case du tableau). Cette initialisation peut se faire soit au moment de la déclaration (et uniquement à ce moment là), en affectant aux cases du tableau des valeurs constantes :

```
type nom_tableau[taille] = {val_case0, val_case1, ...,
                            val_case_fin};
```

soit à l'aide d'une boucle, en accédant individuellement à chaque case du tableau à l'aide d'un déréférencement ou de l'opérateur d'accès `[]` :

```
type nom_tableau[taille];
int n;
for (n = 0; n < taille; n++)
  nom_tableau[n] = valeur_par_defaut;
```

Remarque 2.8. L'opérateur d'accès est simplement une notation qui simplifie le calcul de l'adresse des cases d'un tableau : `nom_tableau[n]` est strictement équivalent à `*(nom_tableau+n)` qui combine un calcul d'adresse et un déréférencement. Avec cette remarque il est facile de comprendre pourquoi les cases d'un tableau sont numérotées en C de `0` à `taille-1`. Nous constatons de plus que le caractère contigu du stockage en mémoire est absolument essentiel pour la manipulation de tableaux !

Remarque 2.9. Il n'est pas prévu en C de mécanisme direct permettant de manipuler les tableaux dans leur ensemble : pour copier le contenu d'un tableau dans un autre il faut copier le contenu de chaque case du tableau !

Lorsqu'on effectue l'affectation d'un tableau on peut omettre de donner la taille qui est alors calculée automatiquement d'après le nombre de valeurs données.

Exemple 2.10. Affectation de tableau avec une taille calculée automatiquement.

```
int tab[] = {0, 1, 5, -7, 8, 10};
```

La variable `tab` représente ici un tableau de six entiers.

2.3.4 Chaîne de caractères

Un texte n'étant qu'une suite de caractères, il est naturel de stocker le contenu d'un texte dans un tableau de caractères, appelé

chaîne de caractère. En pratique l'utilisation d'une chaîne de caractère suppose que le dernier élément du tableau contient le caractère null (c'est à dire l'entier `0`), afin de pouvoir détecter la fin de la chaîne de caractères. Cette convention facilite l'écriture de fonctions manipulant les chaînes de caractère et simplifie leur utilisation. Cependant elle introduit aussi une faille potentielle dans les programmes, lorsque la convention n'est pas respectée, qui peut induire des accès à la mémoire illégaux (erreurs de segmentation) et donc des « crashs » du programme.

Un cas particulier très utile de chaînes de caractères est la *chaîne de caractères constante*, matérialisée par l'utilisation des guillemets : `"texte"`, qui permettent de définir facilement des morceaux de texte dans les programmes, par exemple pour de l'affichage.

Exemple 2.11. Utilisation de chaînes de caractères.

```
char msg[] = "This is a string!\n";
const char *name = "text.txt";
```

La première ligne déclare un tableau de 19 caractères (en comptant les espaces, la ponctuation et le caractère terminal) auquel est affecté le contenu de la chaîne de caractère. La deuxième ligne déclare un pointeur constant vers une chaîne de caractère statique (qui ne peut être modifiée).

2.3.5 Tableau de tableaux

La notion de tableau se généralise sans problème à des tableaux multi-dimensionnels en indiquant que l'on déclare un tableau... de tableaux :

```
type nom_tableau[taille1][taille2];
```

Ici il faut comprendre que la deuxième indication de taille (`taille2`) s'applique à un objet qui est déjà un tableau, de taille `taille1`.

La manipulation des éléments de ce tableau se fait alors en utilisant un *double* opérateur d'accès `[ ][ ]` :

```
type nom_tableau[taille1][taille2];
int n;
```

```
for (n = 0; n < taille1; n++)
{
  int m;
  for (m = 0; m < taille2; m++)
    nom_tableau[n][m] = valeur_par_defaut;
}
```

2.4 Fonctions

Les fonctions permettent d'isoler un ensemble d'instructions à exécuter en vue d'obtenir un certain résultat. Elles permettent :

- de structurer le code,
- de le rendre plus lisible,
- d'augmenter son utilité : une fonction peut même être réutilisée par un autre programme ![2]

En contrepartie elles alourdissent un peu le fonctionnement du programme à l'exécution. Cependant sur les processeurs modernes ce facteur est rarement limitant.

2.4.1 Déclaration d'une fonction

En principe une fonction doit être déclarée avant d'être définie ou appelée. Cette déclaration consiste à donner l'identificateur de la fonction (son nom unique), sa valeur de retour et la liste de ses paramètres, incluant leur type :

```
type_retour nom_fonction (type_par1 nom_par1, ...,
                          type_parN nom_parN);
```

Cette définition s'appelle aussi la *signature* de la fonction.

Lors de l'appel à une fonction le programme va directement à la définition de la fonction et exécute le code (en utilisant les bons paramètres) puis retourne au point d'appel et remplace l'appel à la

2. Elles sont alors regroupées en *bibliothèques*, par exemple les fameux fichiers `.dll` (pour *dynamically linked library*) dans les environnements de travail Windows.

fonction par la valeur de retour, qui peut être alors utilisée. C'est pour cette raison que la fonction doit posséder un type de retour. Si elle ne retourne rien, le type de retour doit être `void`.

Remarque 2.10. Souvent pour les fonctions complexes on utilise comme valeur de retour un entier qui signale si la fonction s'est bien terminée ou si un problème est survenu lors de son exécution. C'est en particulier pour cela que la fonction `main` retourne en général un entier : cela permet de communiquer au système d'exploitation le résultat de l'exécution du programme.

Exemple 2.12. Déclaration d'une fonction.

```
int sum (int *tab, int n);
```

2.4.2 Définition d'une fonction

Une fonction préalablement déclarée doit être ensuite définie, c'est à dire qu'il faut écrire le code qui va être exécuté lors de l'appel à la fonction. Cet ensemble d'instructions constitue le corps de la fonction.

```
type_retour nom_fonction (type_par1 nom_par1, ...,
                          type_parN nom_parN)
{
  /* instructions */
  return valeur_de_retour;
}
```

Un exemple plus concret de fonction est donné un peu plus loin.

Il est bien entendu possible de déclarer dans la fonction des variables temporaires de portée locale. Ces variables sont crées lors de l'appel à la fonction et détruites lorsque la fonction se termine : elle n'existent donc que lors de l'appel à la fonction. Il faut bien comprendre que le mécanisme d'appel à des fonctions compartimente le programme : des variables de fonctions différentes peuvent avoir le même nom, ce n'est pas un problème car elles ne seront jamais utilisées simultanément.

Une fonction doit finir par l'instruction `return valeur_de_retour;`. Si le type de la fonction est `void` on écrira `return;` ou on pourra simplement omettre cette instruction.

Pour communiquer avec le reste du programme une fonction peut utiliser sa valeur de retour et ses paramètres. A ce titre, les paramètres doivent être considérés comme des variables locales initialisées avec la valeur des paramètres lors de l'appel de la fonction.

Exemple 2.13. Définition de la fonction `sum`.

```
int sum (int *tab, int n)
{
  int res = 0, i;
  for (i = 0;i < n; i++)
    res += tab[i];
  return res;
}
```

Dans cet exemple, la fonction est définie par :

- son type de retour : `int`,
- son nom : `sum`,
- le type et le nom du premier paramètre : `int *` et `tab`,
- le type et le nom du second paramètre : `int` et `n`.

Les variables `res` et `i` sont des variables locales qui sont définies uniquement lors de l'appel à la fonction.

2.4.3 Passage des paramètres

En langage C le mécanisme de passage des paramètres à une fonction se fait par valeur, c'est à dire que la fonction reçoit une copie de la valeur du paramètre dans une variable locale. Si cette variable est modifiée à l'intérieur de la fonction, cette modification n'est pas répercutée sur le paramètre (puisqu'on manipule une copie).

Exemple 2.14. Définition de la fonction `sum` en économisant une variable locale.

```
int sum (int *tab, int n)
{
  int res = 0;
```

```
  while (n>0)
  {
    n--;
    res += tab[n];
  }
  return res;
}
```

La variable `n` est initialisée avec la taille du tableau et est décrémentée dans la boucle afin de parcourir le tableau de la dernière à la première case pour calculer la somme. Cela n'a pas d'incidence sur le reste du programme puisque c'est seulement une copie de la valeur de la taille du tableau qui change !

Dans certains cas il est nécessaire de modifier la valeur d'un paramètre passé en argument d'une fonction. Cela semble contradictoire avec le principe de passage par valeur. La solution consiste alors à utiliser comme paramètre non pas la variable mais l'adresse de la variable en utilisant un pointeur [3] : la fonction reçoit alors une copie du pointeur, contenant l'adresse de la variable et on peut alors modifier le contenu de la variable en utilisant un déréférencement.

Exemple 2.15. Passage par adresse : la fonction suivante calcule le carré de l'entier pointé par son argument et stocke le résultat au même endroit.

```
void square (int *x)
{
  *x *= *x;
}
```

2.4.4 Fonctions récursives

Les fonctions récursives ont la propriété de s'appeler elles mêmes. Elles sont particulièrement adaptée pour implémenter certains algorithmes mais nécessitent certaines précautions : en particulier il est de la responsabilité du programmeur de s'assurer que la récursion se termine, pour éviter une pile d'appel infinie.

3. on désigne parfois cette technique comme un passage par adresse.

Exemple 2.16. Calcul du factoriel d'un nombre en utilisant une fonction récursive :

```
int factoriel (int x)
{
  if (x == 0)
    return 1;
  return x * factoriel (x - 1);
}
```

La récursion s'arrête lorsque `x` vaut `0`. L'implémentation de cette fonction résulte de la définition du factoriel d'un nombre comme : $x! = x * (x-1)!$ pour $x \in \mathbb{N}^*$ et avec la convention $0! = 1$.

Remarque 2.11. Les fonctions récursives permettent d'implémenter intuitivement certains algorithmes. Cependant il faut bien garder en tête que leur utilisation est extrêmement coûteuse puisque un appel à ce type de fonctions entraîne un grand nombre d'appels à la fonction, avec à chaque fois la gestion du passage des arguments.

2.5 Exercices

Exercice 2.1. Écrire une fonction : `double solution (double *x1, double *x2, double a, double b, double c)` qui calcule les solutions de l'équation $aX^2 + bX + c = 0$. Cette fonction retourne la valeur du discriminant $\Delta = \sqrt{b^2 - 4ac}$, et suivant les cas, place dans les variables `x1` et `x2` :

- si $\Delta > 0$: $x_1 = (-b - \sqrt{\Delta})/(2a)$ et $x_1 = (-b + \sqrt{\Delta})/(2a)$,
- si $\Delta = 0$: $x_1 = x_2 = -b/(2a)$,
- si $\Delta < 0$: $x_1 = -b/(2a)$ et $x_2 = \sqrt{-\Delta}/(2a)$.

Exercice 2.2. Écrire une fonction `void toUpper (char *str)` qui transforme le contenu de la chaîne de caractères `str` en lettres majuscules. Par exemple la chaîne `"Ceci est un test!"` doit devenir `"CECI EST UN TEST!"`. On pourra pour cela s'aider du contenu de la table 2.2.

Exercice 2.3. Écrire une fonction qui calcule le produit scalaire entre deux vecteurs, stockés dans des tableaux de taille `n` :
`double dot (double x[], double y[], int n)`.

Exercice 2.4. Écrire une fonction :
`void reduce (unsigned int n)`
qui affiche la décomposition en facteurs premiers de l'entier non signé `n`. Par exemple `reduce(65)` affichera `65 = 1 * 5 * 13`.

Chapitre 3

Structures de données

La notion de structure de donnée vient répondre à la question de la représentation dans la mémoire de l'ordinateur de quelque chose de plus compliqué que simplement un ou plusieurs nombres, entiers, réels ou représentant des caractères. Ces représentations sont nécessaires dans des programmes « réalistes » pour manipuler des données réelles.

L'exemple typique d'un tel besoin est donné par le calcul de nombres complexes : un nombre complexe $z = x + \imath y$ résulte de la combinaison de deux nombre réels x et y (la partie réelle et la partie imaginaire) qui n'ont de sens que considérés ensemble. Il est donc souhaitable de représenter ces nombres comme un seul objet dans la mémoire de l'ordinateur. Si certains langages fournissent un tel mécanisme pour manipuler les nombres complexes, celui-ci n'existe pas en C [1]. Nous allons voir que la notion de structure de donnée permet de répondre à ce besoin.

L'autre exemple, un peu moins trivial, que nous utiliserons dans ce chapitre, est celui du carnet d'adresse. Imaginons que l'on souhaite réaliser un programme capable de maintenir un carnet d'adresse,

1. du moins dans la version ANSI C90.

c'est à dire un ensemble de fiches comportant un certain nombre de données sur des personnes, comme par exemple leurs nom, prénom, numéros de téléphone, adresse et e-mail. Comment représenter un tel ensemble de données dans le programme ? Une solution, aussi inefficace que inélégante, consisterait à stocker l'ensemble de ces données comme des chaînes de caractères, ce qui n'est pas forcément approprié. La « bonne » solution consiste à utiliser des *structures*.

3.1 Déclaration d'une structure

Le langage C permet au programmeur de définir un nouveau type de donnée en utilisant la syntaxe :

```
struct nom_de_structure
{
  type1 champ1;
  ...
  typeN champN;
};
```

Le type ainsi défini résulte de l'agrégation en mémoire de l'ensemble des champs définis dans la structure. La taille d'une variable de type `struct nom_de_structure` est donc égal au moins à la somme des tailles de ces constituants.

Remarque 3.1. Afin de ne pas confondre les noms de variables, les types standards du C et les noms de structure, nous utiliserons comme convention pour les noms de structures un symbole « _ » suivi d'un nom en majuscule.

Une fois la structure déclarée nous pouvons déclarer des variables possédant ce type et éventuellement leur affecter une valeur par défaut :

```
struct nom_de_structure nom_de_variable = {
  valeur_champ1, ..., valeur_champN
};
```

Il est important de noter que le type de la structure est donné par l'ensemble `struct nom_de_structure`, composé du mot clé `struct` et du nom choisi `nom_de_structure`. De plus le type ainsi obtenu possède toutes les propriétés des types standards du langage C. En particulier il peut être utilisé comme type pour un paramètre de fonction ou bien pour une valeur de retour.

Exemple 3.1. Déclaration et utilisation d'un type complexe.

```
struct _COMPLEX
{
  double re, im;
};
struct _COMPLEX z = {1.0, 0.0};
```

Cette structure contient deux nombres réels stockés consécutivement en mémoire qui représentent la partie réelle et imaginaire d'un nombre complexe. Lorsqu'on affecte deux valeurs réelles à une variable de type `struct _COMPLEX` la première est affectée au champ `re`, la seconde au champ `im`.

3.1.1 Alias de nom de structure

Si l'utilisation des structures permet d'introduire une grande flexibilité dans le code, leur manipulation est susceptible de l'alourdir considérablement. Pour éviter d'avoir à employer le mot clé `struct` constamment, il est courant de définir un alias pour le nom de la structure en utilisant la syntaxe :

```
typedef struct nom_de_structure nom_d_alias;
```

Remarque 3.2. Nous utiliserons comme convention pour les alias de nom de structure un mot écrit en majuscules.

Exemple 3.2. Définition d'un alias.

```
typedef struct _COMPLEX COMPLEX;
COMPLEX z = {0.0, 1.0};
```

Bien sur le type `struct _COMPLEX` doit être défini précédemment (ici à la pre-

mière ligne). Souvent la définition de l'alias est intégrée à la déclaration de la structure :

```
typedef struct _COMPLEX
{
  double re, im;
} COMPLEX;
```

Dans ce cas on peut même utiliser une structure *anonyme* en omettant le nom `_COMPLEX`, puisque *a priori* nous n'utiliserons que l'alias dans la suite du programme.

3.1.2 Manipulation des champs

Puisque les structure définies à l'aide du mot clé `struct` sont *a priori* arbitraires il n'est pas possible de les manipuler directement à l'aide des opérateurs standards [2]. En effet, si l'addition de deux nombres complexes possède un sens bien défini en mathématique, l'addition de deux structures complexes en C n'est pas pré-définie. En réalité ce qu'il faut réaliser c'est d'une part l'addition des champs « partie réelle » et d'autre part l'addition des champs « partie imaginaire ». Il faut donc pouvoir accéder directement à ces champs. Cette opération s'effectue avec l'opérateur *membre de* « . » :

```
nom_de_variable.nom_de_champ
```

Remarque 3.3. Cette expression se lit de droite à gauche « champ nommé `nom_de_champ` membre de la variable nommée `nom_de_variable` ». Bien entendu il faut que la variable en question représente un objet possédant un tel champ !

Exemple 3.3. Accès direct au champ d'une structure.

```
COMPLEX z = {0,0};
z.re = 1;
z.im = -2 * z.re;
```

Les deux dernières instructions modifient les champs `re` et `im` respectivement

2. hormis avec l'opérateur d'assignation qui copie le contenu d'une structure dans une autre.

de la variable `z` de type `COMPLEX`.

3.1.3 Pointeur vers une structure

Comme nous l'avons déjà dit les structures déclarées peuvent être considérées comme des types standard du langage C. En particulier il est autorisé, et souvent utile, de définir un pointeur vers une variable contenant une structure :

```
struct nom_de_structure * nom_de_pointeur;
```

La seule subtilité concerne alors l'opérateur *membre de* qui devient « -> », lorsque l'on accède au champ d'une structure à travers un pointeur :

```
nom_de_pointeur->nom_de_champ
```

Exemple 3.4. Utilisation d'un pointeur vers une structure.

```
COMPLEX z = {1, 0};
COMPLEX *ptr = &z;
ptr->im = -1;
```

Après cette suite d'instructions la partie imaginaire de `z` vaut `-1`.

3.2 Structure de structure

Puisque les structures sont des types comme les autres, on peut bien entendu utiliser une structure comme champ à l'intérieur d'une autre. Si nous revenons à l'exemple du carnet d'adresse, on peut définir une structure qui va contenir les champs d'une fiche. Dans ce cas particulier il paraît logique d'utiliser des chaînes de caractères pour noter le nom, les prénoms et l'e-mail et des tableaux de nombres pour représenter un numéro de téléphone[3]. Le cas de l'adresse pose lui même question puisque elle contient elle même plusieurs types de données différents.

3. en se limitant aux numéros français par exemple, avec dix chiffres.

Exemple 3.5. Une implémentation possible d'une fiche de carnet d'adresse.

```
#define BUFFERSIZE 80
typedef char STRING[BUFFERSIZE];
typedef struct
{
  int number;
  STRING name;
  int code;
  STRING city;
} ADDRESS;
typedef struct
{
  STRING name;
  STRING surname[10];
  ADDRESS address;
  int phone[5];
  int fax[5];
  STRING email;
} FICHE;
FICHE fiche;
```

Dans ce cas particulier, comment modifier le numéro de rue de l'adresse contenue dans une fiche? Il faut utiliser deux opérateurs *membre de* successifs, d'abord pour accéder au champ `address` de la fiche puis au champ `number` de l'adresse : `fiche.address.number = 10;`.

3.3 Tableau de structure

Finalement une fois les structures définies, on peut créer des tableaux de variables de type structure, comme des tableaux standard et on manipule le contenu des cases du tableau avec l'opérateur *membre de*.

Exemple 3.6. Tableau de structure.

```
COMPLEX tab[10];
int i;
for (i = 0; i < 10; i++)
  tab[i].re = tab[i].im = 0.0;
```

Les tableaux de structures permettent d'organiser la mémoire (pour l'instant de manière statique) en vue de gérer de grandes quantités de données.

3.4 Interface d'accès à une structure

Un point essentiel pour la bonne utilisation des structures est de bien identifier les opérations courantes pour lesquelles elles vont être utilisées. Ainsi on peut associer à ces opérations des fonctions d'interface qui permettent d'éviter de ré-écrire ce code à chaque fois [4]. Cela permet aussi d'avoir un code lisible et facilement portable.

En général on défini à minima les fonctions permettant de :

- accéder aux champs,
- modifier le contenu des champs,
- sauver une instance de la structure dans un fichier au format ASCII,
- lire une instance de la structure depuis une chaîne au format ASCII,
- afficher une instance de la structure à l'écran de manière explicite.

Exemple 3.7. Suggestion d'interface de programmation de la classe complexe.

```
typedef struct { double re, im; } COMPLEX;
/* Access to the class members */
double getRe (COMPLEX z);
double getIm (COMPLEX z);
/* Modification of the class members */
void setRe (COMPLEX *z, double x);
void setIm (COMPLEX *z, double y);
/* Input/Output */
void printToFile (FILE *file, COMPLEX z);
COMPLEX loadFromString (const char *str);
```

Ces fonctions « de base » possèdent des caractéristiques génériques que nous allons maintenant détailler.

4. On évite ainsi des erreurs!

3.4.1 Accès aux champs

Les fonctions d'accès aux champs d'une structure sont en général déclarées de la manière suivante :

```
type getChamp1 (struct nom_de_struct var);
```

Elles ne font rien d'autre que retourner la valeur du champ `type champ1` et en conséquence leur implémentation est souvent « triviale » :

```
type getChamp1 (struct nom_de_struct var)
{
  return var.champ1;
}
```

3.4.2 Modification des champs

Les fonctions de modification des champs d'une structure sont en général déclarées de la manière suivante :

```
void setChamp1 (struct nom_struct *var, type value);
```

Dans ce cas la fonction utilise un pointeur vers la structure car celle-ci va être modifiée (voir la section 2.4.3) et le second paramètre contient la nouvelle valeur à actualiser.

Typiquement l'implémentation de ces fonctions aura la structure suivante :

```
void setChamp1 (struct nom_struct *var, type value)
{
  if (var != NULL)
    var->champ1 = value;
}
```

Dans des cas plus complexes, par exemple pour la modification d'une chaîne de caractères, l'opération d'assignation peut être plus complexe.

Remarque 3.4. Parfois il est utile d'effectuer des tests complémentaires avant l'assignation et éventuellement retourner un code d'erreur en cas de problème ou afficher un message d'erreur.

3.4.3 Entrées / Sorties

Les fonctions d'interfaces avec les entrées et les sorties du programme sont en général beaucoup plus complexes et dépendent très fortement du type de données stockées dans la structure. Il est néanmoins toujours possible de convertir [5] le *contenu* d'une structure en une chaîne de caractères, où les valeurs des champs sont séparées par un symbole (appelé séparateur), souvent une virgule ou un point virgule. On peut alors stocker le contenu des structures dans des fichiers, où chaque ligne contient une structure. Le format de ce type de fichier est appelé (pour des raisons évidentes) format csv, pour *comma separated values* (valeurs séparées par des virgules).

Dans la pratique on définira donc une fonction de *sortie* permettant d'écrire le contenu de la structure sur une ligne dans un fichier (sur le disque dur ou l'écran) :

```
void printToFile (FILE *file, struct nom_struct var);
```

et une fonction d'entrée permettant de lire le contenu depuis une chaîne de caractère (correspondant par exemple à une des lignes d'un fichier csv) :

```
void printToFile (struct nom_struct *var, char *str);
```

3.4.4 Fonctions annexes

Cette interface minimale peut être complétée par les fonctions les plus utiles, c'est à dire celles qui sont susceptibles d'être le plus souvent utilisées. Bien entendu ces fonctions vont dépendre fortement du type de données stockées dans la structure !

5. à l'aide par exemple de la fonction `fprintf` de la bibliothèque `stdio.h`

Exemple 3.8. Une extension possible pour l'interface de programmation de la classe complexe.

```
COMPLEX complex (double re, double im);
COMPLEX product (double lambda, COMPLEX z);
COMPLEX addition (COMPLEX z1, COMPLEX z2);
COMPLEX subtraction (COMPLEX z1, COMPLEX z2);
COMPLEX multiplication (COMPLEX z1, COMPLEX z2);
COMPLEX division (COMPLEX z1, COMPLEX z2);
```

Remarque 3.5. Cette manière de structurer les programmes se rapproche de la notion de *classe* en langage C++.

3.5 Exercices

Exercice 3.1. Écrire (et tester) une implémentation des fonctions d'interface définies dans l'exemple 3.7. On sauvera un nombre complexe dans un fichier sous la forme : `(re,im)`, et on interprétera une chaîne de caractères `(x,y)`, comme le nombre complexe $z = x + \imath y$.

Exercice 3.2. Écrire (et tester) une implémentation des fonctions d'interface définies dans l'exemple 3.8.

Exercice 3.3. Modifier la fonction de lecture d'un nombre complexe depuis une chaîne de caractère pour reconnaître aussi les nombres complexes sous la forme `"2.53445+7.189814I"`, où le caractère `'I'` est utilisé pour signaler la partie imaginaire.

Exercice 3.4. Modifier la même fonction pour reconnaître aussi les possibilités suivantes : `"2.4534647"` (nombre réel pur), `"1.432525I"` (nombre imaginaire pur) et les stocker dans le nombre complexe approprié.

Chapitre 4

Stockage dynamique

Le stockage dynamique correspond à une utilisation *dynamique* de la mémoire, par opposition à l'approche *statique* jusqu'ici utilisée. Cette notion apporte une solution pratique à la question suivante : comment réserver une quantité de mémoire suffisante quand on ne connaît pas à l'avance [1] la taille des tableaux que l'on va manipuler. Par exemple dans toute application qui utilise une base de données avec des entrées qui peuvent être ajoutées ou supprimées, on ne sait pas à l'avance le nombre d'entrées, qui dépend des choix de l'utilisateur.

Nous verrons aussi que le concept d'allocation dynamique permet de construire des conteneurs de données plus souples, pour certaines applications, que les tableaux. Les fonctions utilisées dans ce chapitre sont déclarées dans la bibliothèque standard `stdlib.h`, qu'il faut donc inclure dans les programmes utilisant de l'allocation dynamique.

1. c'est à dire au moment de l'écriture du programme.

4.1 Allocation de mémoire

Avant de discuter des mécanismes d'allocation de mémoire il est important de souligner que l'allocation de mémoire est implicitement présente dans beaucoup d'opérations en C. Considérons par exemple l'instruction suivante :

```
int a = 5;
```

Cette instruction déclare une variable nommée `a`, c'est à dire qu'elle réserve un espace mémoire[2] lié à cette variable et lui affecte la valeur entière `5`. Cet espace mémoire ne peut être utilisé pour stocker la valeur d'une autre variable tant que la variable `a` existe.

Pour ce convaincre que la variable `a` est bien stockée en mémoire on peut récupérer son adresse :

```
int *pa = &a;
```

le pointeur `pa` contient alors un chiffre qui représente le numéro de la case mémoire dans laquelle se trouve la variable `a`. On sait de plus (voir la section 2.3.2) que le pointeur peut permettre de modifier la valeur de la variable, en utilisant l'opérateur de déréférencement.

4.1.1 Allocation simple

Peut-on alors se passer de la déclaration de la variable et utiliser seulement des pointeurs ? La réponse est oui, à condition de réserver explicitement un espace mémoire vers lequel le pointeur va pointer. Cette opération fait appel à une fonction :

```
void * malloc (size_t size);
```

qui, lorsque elle est appelée réserve un espace mémoire de taille `size` (octets) et retourne son adresse.

2. correspondant à quatre « cases » élémentaires de un octet.

Remarque 4.1. Le type `size_t` est en général un alias de `unsigned long int`. On calculera en général la taille en utilisant l'opérateur `sizeof`.

Remarque 4.2. La fonction `malloc` retourne un pointeur de type `void *` afin d'être compatible avec tous les types de données.

Remarque 4.3. La fonction `malloc` n'initialise pas le contenu de la mémoire allouée, qui contient donc une valeur inconnue. C'est à l'utilisateur de faire l'initialisation après l'allocation.

Exemple 4.1. Utilisation de l'allocation dynamique pour manipuler un entier.

```
int *p = malloc (sizeof (int));
*p = 4;
```

Après la première ligne le pointeur pointe vers un espace mémoire capable de contenir un entier. Cet espace est initialisé à la valeur `4` à la deuxième ligne.

4.1.2 Allocation et initialisation

L'allocation de mémoire peut aussi se faire en utilisant la fonction :

```
void * calloc (size_t number, size_t size);
```

qui alloue un espace mémoire de `number` cases de taille `size` (octets). A la différence de `malloc`, la fonction `calloc` *initialise* la mémoire allouée en écrivant la valeur `0` sur tous les bits réservés.

Il est donc conseillé d'utiliser `calloc` plutôt que `malloc` afin d'éviter les erreurs dues à une mauvaise initialisation, surtout pour des structures définies par le programmeur.

Exemple 4.2. Utilisation de `calloc`.

```
int *p = calloc (1, sizeof (int));
```

Cette instruction déclare un pointeur et lui associe un espace mémoire d'un entier, initialisé à zéro.

4.1.3 Libération de la mémoire

Il est important de comprendre que l'utilisation des fonctions d'allocation dynamique offre des avantages (qui deviendront évidents dans la suite de ce chapitre), mais implique aussi la responsabilité du programmeur : en particulier toute la gestion « cachée » des allocations statiques de mémoire est maintenant à sa charge, pour la mémoire allouée dynamiquement. Illustrons ceci par l'exemple suivant [3].

Exemple 4.3. Allocation multiple de mémoire statique.

```
int i, n = 0;
for (i = 0; i < 100; i++)
{
  int j = i * i;
  n += j;
}
```

Dans cet example les variables `i` et `n` sont dites *globales* : elles sont définies à la première ligne et « vivent » jusqu'à la dernière ligne du code de cet exemple. La variable `j` est une variable *locale* : à chaque passage dans la boucle un entier est déclaré, alloué et initialisé à la valeur `i*i`. A la cinquième ligne il est implicitement détruit (la mémoire est libérée et prête à être à nouveau utilisée). Donc, à la fin de ce code, un espace mémoire correspondant à deux entiers est encore alloué : pour la variable `i` un espace contenant la valeur entière 100, pour la variable `n` un espace contenant la somme des carrés des entiers de 0 à 99 (soit 328350).

Que se passe-t-il maintenant si la variable locale `j` est allouée dynamiquement ?

Exemple 4.4. Allocation multiple de mémoire dynamique.

3. Cet exemple est ici utilisé à des fins pédagogiques, un compilateur moderne optimisera certainement la boucle en faisant disparaître la variable intermédiaire `j`.

```
int i, n = 0;
for (i = 0; i < 100; i++)
{
  int *p = malloc (sizeof (int));
  *p = i * i;
  n += *p;
}
```

Dans cette version, c'est le pointeur `p` qui est une variable locale, allouée et libérée à chaque itération. Par contre l'espace mémoire associé par la fonction `malloc` n'est lui jamais libéré ! A la fin de ce code un espace mémoire correspondant à 102 entiers est donc encore alloué, la plupart étant inutile car ne contenant qu'un intermédiaire de calcul.

Il faut donc à chaque fois qu'un espace mémoire a été alloué dynamiquement le libérer explicitement dès qu'il n'est plus utile, en utilisant la fonction :

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

Cette fonction libère l'espace mémoire situé à l'adresse contenue dans le pointeur `ptr`. Cette mémoire doit avoir été allouée dynamiquement.

Remarque 4.4. Le paramètre de la fonction `free` est un pointeur de type `void *` afin d'être compatible avec tout type de données.

Remarque 4.5. La fonction `free` n'efface pas le contenu de la mémoire libérée.

L'erreur qui consiste à oublier de libérer la mémoire utilisée rend les programmes moins fiables et sécurisés. Il existe des outils qui permettent de détecter ces erreurs dans les programmes [4].

Exemple 4.5. Allocation multiple de mémoire dynamique, avec libération explicite.

```
int i, n = 0;
for (i = 0; i < 100; i++)
{
```

4. Sur les systèmes unix on peut citer l'excellent (et gratuit !) outil `valgrind` (`http://valgrind.org`).

```
    int *p = malloc (sizeof (int));
    *p = i * i;
    n += *p;
    free (p);
}
```

4.1.4 Ré-allocation de la mémoire

On trouve dans la bibliothèque standard une dernière fonction de gestion dynamique de la mémoire :

```
void *realloc (void *ptr, size_t size);
```

Cette fonction permet de réajuster la taille d'un espace mémoire réservé dynamiquement. Son utilisation est un petit peu plus délicate et dépend de la valeur des paramètres :

- si le paramètre `ptr` est `NULL`, elle se comporte comme un appel à la fonction `malloc(size)` et retourne l'adresse de la zone mémoire allouée.
- si le paramètre `size` vaut zéro, elle se comporte comme un appel à la fonction `free(ptr)` et retourne le pointeur `NULL`,
- sinon elle change la taille de la zone mémoire en la déplaçant éventuellement à une adresse où cela est possible.

Remarque 4.6. Si la zone mémoire est agrandie, les nouvelles cases ne sont pas initialisées, la fonction retourne l'adresse de la nouvelle zone et l'ancienne est libérée par un appel à la fonction `free`.

En pratique on n'utilise pas l'allocation dynamique pour une tache aussi simple que dans les exemples proposés, où l'allocation de mémoire statique est parfaitement adaptée. La fin de ce chapitre est consacré à l'étude de cas plus pertinents.

Remarque 4.7. Les fonctions `malloc`, `calloc` et `realloc` retournent un pointeur `NULL` si l'allocation de mémoire a échoué. Il est donc recommandé de tester la valeur du pointeur obtenu avant de l'utiliser pour éviter les problèmes d'accès en mémoire.

4.2 Tableaux dynamiques

La première utilisation de l'allocation dynamique consiste à allouer non pas une mais plusieurs cases mémoires afin d'obtenir ainsi un tableau. Afin de matérialiser ce processus, la figure 4.1 illustre l'utilisation de la mémoire.

Exemple 4.6. Allocation dynamique d'un tableau.

```
int *tab = NULL;
int n;
n = getUserInput ();
tab = malloc ((size_t)(n * sizeof (int)));
if (tab == NULL)
  fprintf (stderr, "Error: memory allocation failed!\n");
...
free (tab);
```

Ici on suppose que la fonction `getUserInput` demande à l'utilisateur de saisir un nombre au clavier et retourne l'entier ainsi obtenu : on ne connaît ainsi pas à l'avance la taille du tableau à utiliser.

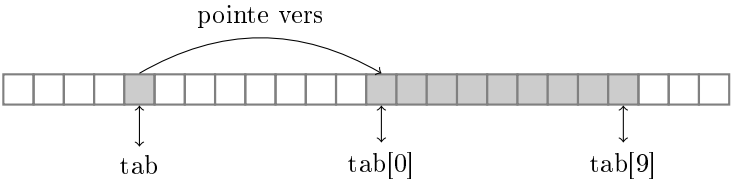

FIGURE 4.1 – Mémoire utilisée par un tableau, suite à une allocation dynamique. Si on ré-alloue le tableau, l'adresse de la première case risque de changer, car la valeur contenue dans `tab` peut changer.

Bien entendu le tableau ainsi obtenu s'utilise de la même manière que les tableaux statiques vus à la section 2.3. Cependant il ne faut pas oublier de libérer la mémoire allouée en appelant la fonction `free`.

4.3 Conteneurs génériques

Si les tableaux statiques et dynamiques peuvent servir à stocker efficacement des données en mémoire, ils sont particulièrement inefficaces dès lors que l'on ajouter ou supprime régulièrement des données. En effet pour la suppression, si la mémoire concernée se trouve au milieu du tableau il faut déplacer toutes les cases suivantes avant de réduire la taille du tableau d'une unité, ce qui fait beaucoup d'opérations. Pour ces cas particuliers d'autres types de conteneurs de données sont plus efficaces, bien que plus complexes à mettre en œuvre. Nous alons en présenter quelques un dans cette partie.

4.3.1 Listes chaînées

Une liste chaînée est un conteneur de données qui permet de gérer simplement l'ajout et la suppression de données en représentant l'ensemble des données sous la forme d'une liste, c'est à dire d'éléments liés entre eux. Cette liaison se fait naturellement en C en utilisant les pointeurs : chaque nœud de la liste contient une donnée et l'adresse de la donnée suivante.

Remarque 4.8. On adopte la convention que le dernier élément de la liste contient l'adresse `NULL` pour signifier la fin de la liste.

Déclaration

Pour définir une liste on utilise en général deux structures : une pour représenter les nœuds et une autre pour représenter la liste. Bien entendu, les données seront elles-mêmes stockées dans une structure appropriée. Il est donc naturel de définir un nœud de la manière suivante :

```
typedef struct _NODE
{
  type data;
```

```
  struct _NODE *next;
} NODE;
```

Remarque 4.9. Ce type de déclaration de structure est « récursif », puisqu'on utilise un pointeur vers une structure du type déclaré à l'intérieur de la structure elle-même. C'est un cas particulier où on ne peux pas utiliser l'alias `NODE` (à la ligne 4), puisqu'il n'est pas encore déclaré !

En utilisant cette structure `NODE` on déclare ensuite la liste elle même comme une nouvelle structure :

```
typedef struct
{
  int size;
  NODE *first;
} LIST;
```

Remarque 4.10. La structure `LIST` doit contenir l'adresse du premier élément de la liste. Souvent on ajoute d'autres propriétés de la liste, comme sa taille (ou longueur), plus par commodité que par nécessité.

Remarque 4.11. Dans l'implémentation proposée, une liste vide pointe vers l'adresse `NULL`.

Remarque 4.12. Il peut être utile de chercher à représenter graphiquement le contenu d'une liste, en utilisant des flèches pour symboliser les pointeurs, comme représenté dans la figure 4.2.

Interface

De manière à simplifier l'utilisation d'une telle liste, on défini les fonctions permettant d'effectuer les opérations courantes sur une liste, ajout et suppression d'éléments, éventuellement calcul de la taille, affichage, etc...

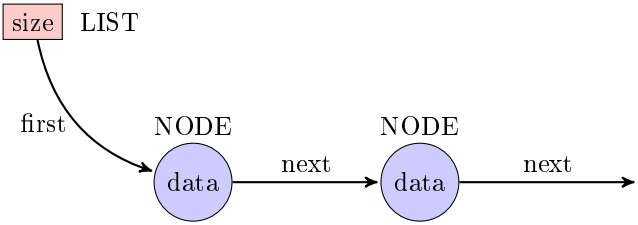

FIGURE 4.2 – Représentation visuelle d'une liste chaînée comportant deux éléments. Les flèches symbolisent les pointeurs.

```
void prepend (LIST *list, type data);
void append (LIST *list, type data);
void insertAt (LIST *list, type data, int i);
NODE *remove (LIST *list, int i);
NODE *get (LIST *list, int i);
void print (LIST *list);
```

4.3.2 Piles

Une pile est un type particulier de liste, adapté à un usage précis, et avec un fonctionnement simplifié. En général on utilise des piles « FIFO » (*first in first out*) ou « LIFO » (*last in first out*), avec une interface simplifiée :

```
void push (LIST *list, type data);
NODE *pop (LIST *list);
```

où la fonction `push` ajoute un élément (en tête ou en queue) à la liste et la fonction `pop` retire un élément (en tête ou en queue).

Remarque 4.13. Les piles sont très utiles dans certains contextes particuliers :

- la notation polonaise inversée pour faire des calculs mathématiques

utilise une pile,
- les processeurs utilisent des piles pour gérer l'appel des fonctions dans les programmes,
- la fonction « annuler » des éditeurs de texte utilise une pile qui garde l'historique des modifications.

4.3.3 Arbres

Les arbres sont une sorte de généralisation de la notion de liste à des objets plus complexes où chaque élément possède deux « suivants », traditionnellement appelés *droit* (*right*) et *gauche* (*left*). Les éléments d'un arbre sont alors appelés *feuilles* (*leaf*) et le premier élément est la *racine* (*root*). Les arbres permettent d'organiser des données en imposant une relation d'ordre entre les éléments, ce qui permet d'effectuer des recherches rapidement dans l'ensemble des données. Par analogie avec les listes chaînées il est naturel de définir les feuilles comme :

```
typedef struct _LEAF
{
  type data;
  struct _LEAF *left;
  struct _LEAF *right;
} LEAF;
```

et de déclarer la structure d'arbre ainsi :

```
typedef struct
{
  LEAF *root;
} TREE;
```

Il est par contre plus délicat de définir une interface de manipulation de l'arbre car elle va fortement dépendre de ce que l'on cherche à faire avec l'arbre. Il faudra donc définir des fonction d'insertion, de recherche et de destruction spécifiques à un problème particulier.

4.4 Exercices

Exercice 4.1. Écrire une fonction `getLine(FILE *file)` qui lit une chaîne de caractère de longueur arbitraire dans un fichier (jusqu'à rencontrer une fin de ligne ou une fin de fichier), réserve l'espace mémoire correspondant, y copie le contenu de la ligne et retourne l'adresse de la mémoire.
Indication : on pourra utiliser la fonction `fgets` de la bibliothèque standard qui permet de lire dans un fichier une chaîne de caractère de taille connue.

Exercice 4.2. Implémenter une liste chaînée d'entiers, c'est à dire les fonctions d'interface suggérées dans la partie 4.3.1.

Exercice 4.3. Implémenter une liste doublement chaînée d'entiers, ou chaque nœud possède un élément précédent et un suivant.

Exercice 4.4. Utiliser une liste chaînée d'entiers pour réaliser un tri par insertion, permettant de classer un tableau d'entiers du plus petit au plus grand.
Indication : il faut utiliser une liste et insérer chaque nouvel élément à « sa » place, c'est à dire de telle sorte qu'il soit plus grand que son précédent et plus petit que son suivant.

Chapitre 5

Bibliothèque standard

Ce dernier court chapitre propose une introduction à la bibliothèque standard du langage C, c'est à dire l'ensemble des fonctions pré-existantes qui permettent d'effectuer certaines tâches. Plutôt que de passer en revue l'ensemble des fonctions de la bibliothèque standard nous avons choisi de présenter quelques exemples d'utilisation répondant à des questions pratiques. En particulier on va étudier ici les fonctions déclarées dans le fichier `stdio.h`. Vous êtes libres de vous en inspirer pour créer vos propres programmes !

On rappelle qu'une documentation plus détaillée de ces fonctions est accessible à cette adresse : `http://www.cplusplus.com/reference/clibrary/`.

5.1 Afficher une variable à l'écran

Afficher le contenu d'une ou plusieurs variables à l'écran est un élément essentiel de tout programme en particulier pour indiquer le résultat d'un calcul. Pour ce faire on utilise la fonction :

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

Son fonctionnement fait appel à un mécanisme un peu compliqué permettant de gérer un nombre d'arguments variables. Il est suffisant de retenir que la chaîne de caractères `format` doit contenir ce que l'on veut afficher, en utilisant des *codes* spéciaux pour afficher le contenu des variables (voir le tableau 5.1).

code	type
`"%c"`	`char`
`"%d"`, `"%ld"`	`int`, `long int`
`"%g"`, `"%lg"`	`float`, `double`
`"%s"`	`char *`

TABLE 5.1 – Liste des principaux codes de format de la fonction `printf`. Ces codes peuvent être complétés par des modificateur qui permettent d'optimiser l'affichage. Il serait trop long de les présenter ici.

Exemple 5.1. Exemple d'utilisation de la fonction `printf` pour afficher à l'écran un `char`, un `int`, un `double` et une chaîne de caractères `char*`.

```
#include <stdio.h>
int main (void)
{
  char c = 'r';
  int i = 25;
  double x = -3.1415;
  char msg[] = "This is a string!\n";
  printf ("Character : %c\n", c);
  printf ("Integer   : %d\n", i);
  printf ("Double    : %g\n", x);
  printf ("String    : %s\n", msg);
  printf ("All: %c / %d / %g / %s\n", c, i, x, msg);
  return 0;
}
```

La dernière utilisation de `printf` dans cet example montre comment afficher plusieurs variables avec un seul appel.

Remarque 5.1. A la place de `printf` on peut aussi utiliser la fonction :

```
int fprintf (FILE *file, const char *format, ...);
```

qui permet *d'afficher* le contenu des variables dans un fichier quelconque. En particulier si on utilise le fichier `stdout` ou `stderr` on retrouve l'affichage à l'écran.

Remarque 5.2. La fonction `printf` ne vérifie pas que le code de format utilisé correspond bien au type de la variable : si ce n'est pas le cas l'affichage sera probablement mauvais, sans pour autant provoquer l'arrêt du programme. Normalement si le compilateur est bien configuré, les erreurs de format les plus fréquentes sont indiquées comme des *warnings* lors de la compilation. Il est recommandé d'y prêter une attention particulière.

5.2 Lire des informations au clavier

En général on attend d'un programme informatique une certaine inter-activité : l'utilisateur peut alors changer l'exécution du programme en entrant des informations au clavier. Dans un programme en C, le clavier est un fichier spécial `stdin` dans lequel vont aller « s'imprimer » les touches frappées sur le clavier.

5.2.1 Lecture formatée

Pour récupérer et convertir les données ainsi saisies on peut utiliser la fonction :

```
int scanf (const char * format, ...);
```

Cette fonction peut être vue comme « l'inverse » de la fonction `printf` et fonctionne en utilisant les mêmes codes de format. Cependant comme on cherche à récupérer des informations on a besoin de *modifier* le contenu des variables passées en argument : il faut donc impérativement utiliser **l'adresse** de ces variables.

Exemple 5.2. Exemple d'utilisation de la fonction `scanf`, pour lire des informations au clavier.

```
#include <stdio.h>
int main (void)
{
  int age;
  double height;
  char name[80];
  printf ("Enter your name   : ");
  scanf ("%79s", name);
  printf ("Enter your age    : ");
  scanf ("%d", &age);
  printf ("Enter your height: ");
  scanf ("%lg", &height);
  printf ("Your name is %s\n"\
          "You are %d years old\n"\
          "Your height is %lg\n",
          name, age, height);
  return 0;
}
```

Il est important de remarquer que l'on utilise bien des pointeurs lors d'un appel à la fonction `scanf`.

Remarque 5.3. Ici on a utilisé des modificateurs de formats qui sont essentiels pour que la fonction `scanf` fonctionne correctement :

- `"%lg"` signifie que la variable attendue est de type `double` (`%g` correspond en fait à `float`),
- `"%79s"` indique que la chaîne de caractère peut contenir au plus 79 caractères (on garde un emplacement pour le caractère null de fin de chaîne).

Remarque 5.4. On peut combiner la lecture de plusieurs variables en une seule instruction en utilisant plusieurs formats successifs. Cette pratique est cependant déconseillée car elle peut entraîner facilement des erreurs.

Remarque 5.5. On peut utiliser la valeur de retour de la fonction `scanf` pour vérifier que la saisie s'est déroulée sans erreurs : par exemple que l'utilisateur à bien entré seulement des chiffres lorsqu'on demande un entier.

Remarque 5.6. La fonction `scanf` lit la chaîne de caractère entrée jusqu'à rencontrer un caractère espace, un retour à la ligne ou une tabulation : elle ne peut donc être utilisée que pour lire des mots, pas des phrases entières.

5.2.2 Lecture non formatée

Lorsque l'on souhaite lire une chaîne de caractères arbitraire en s'affranchissant des contraintes de la fonction `scanf` il est recommandé d'utiliser la fonction :

```
char *fgets (char *str, int n, FILE *stream);
```

Cette fonction lit les $n-1$ premiers caractères présents dans le fichier `stream` et les place au fur et à mesure dans la chaîne `str`. Si elle rencontre un caractère saut de ligne ou fin de fichier elle s'arrête immédiatement et retourne la chaîne contenant les caractères lus. En cas d'erreur de lecture cette fonction retourne un pointeur `NULL`, sinon elle retourne `str`.

Remarque 5.7. En général le paramètre `n` est la taille du tableau de caractères vers lequel point `str`.

Remarque 5.8. Lorsque la lecture s'arrête sur un caractère retour à la ligne celui-ci est inclus dans la chaîne `str`.

Exemple 5.3. Utilisation de la fonction `fgets`, pour lire un chaîne de caractères au clavier, avec contrôle des erreurs.

```
#include <stdio.h>
#define SIZE 80
int main (void)
{
  char str[SIZE];
  printf ("Enter a string: ");
  if (fgets (str, SIZE, stdin) == NULL)
    printf ("Read error!\n");
  else
    printf ("You entered: %s", str);
```

```
    return 0;
}
```

5.2.3 Lecture d'une chaîne au format csv

On sera souvent amené à manipuler des chaînes contenant des données stockées en format *csv* (voir par exemple la section 3.4.3), par exemple la chaîne suivante :

```
char str[] = "3.141,2.726,1.00,-56.42,125.9";
```

qui contient des nombres réels que l'on souhaiterait convertir au format `double`. Les nombre réels étant séparés ici par des virgules il faudrait arriver à découper la chaîne au niveau de ces virgules et convertir les sous chaînes ainsi obtenues en nombre réels. Cette opération assez compliquée peut être réalisée en utilisant la fonction :

```
char *strtok (char * str, const char *lim);
```

où `lim` contient les délimiteurs sur lesquels le découpage va être fait. Il serait un peu trop long de détailler le fonctionnement complet de cette fonction aussi nous donnons ci-dessous un exemple de son utilisation.

Exemple 5.4. Utilisation de la fonction `strtok`, pour convertir une chaîne de caractères contenant des champs séparés par un délimiteur.

```
#include <stdio.h>    /* for printf() */
#include <stdlib.h>   /* for atof()   */
#include <string.h>   /* for strtok() */
int main (void)
{
  char str[] = "3.141,2.726,1.00,-56.42,125.9";
  char *ptr;
  ptr = strtok (str,",");     //initialize ptr
  while (ptr != NULL)
  {
    double x = atof (ptr);    //convert token
    print ("Number: %g\n", x);
    ptr = strtok (NULL,","); //get next token
  }
  return 0;
```

```
    || }
```

On a utilisé la fonction `atof` pour convertir une chaîne de caractères en `double`.

On peut facilement généraliser cette exemple à la conversion d'une chaîne contenant des types mixtes de données dans un ordre prédéfini, comme par exemple le contenu d'une structure enregistrée en format ASCII.

Remarque 5.9. On trouve dans la bibliothèque `string.h` de nombreuses fonctions de manipulation de chaînes qui permettent de simplifier la gestion des chaînes de caractère.

5.3 Manipuler un fichier

Si l'inter-activité d'un programme avec un utilisateur est un aspect important il est souvent indispensable de pouvoir faire en sorte que plusieurs programmes s'échangent des informations ou bien sauvegardent des valeurs pour les utiliser plus tard. C'est pour ce type d'opérations que l'on a besoin de manipuler des fichiers. Par commodité on utilise le plus souvent des fichiers *ASCII* c'est à dire qui contiennent le contenu des variables en écriture « normale », par opposition à un fichier binaire qui contient directement les valeurs binaires stockées en mémoire.

Pour manipuler les fichiers on utilise en général les fonctions :

```
FILE *fopen (const char *name,
             const char *mode);
int fclose (FILE *stream);
int feof (FILE *stream);
```

La fonction `fopen` permet d'ouvrir un fichier dont le nom est stocké dans la chaîne de caractère `name`. Si on veut lire le contenu du fichier il faut utiliser le mode `"r"` (pour *read*, lecture) et si on veut modifier le contenu du fichier il faut utiliser le mode `"w"` (pour *write*, écriture). Il existe d'autres modes d'ouverture de fichier adaptés à des usages spécifiques et qu'on déconseille en général d'utiliser. Si la

fonction `fopen` réussi à ouvrir le fichier elle renvoie un pointeur vers une structure `FILE` permettant de manipuler le fichier et sinon, en cas d'échec d'ouverture [1], renvoie un pointeur `NULL`.

La fonction `fclose` permet de fermer un fichier préalablement ouvert avec `fopen`.

Remarque 5.10. Il faut penser à utiliser `fclose` après `fopen`, comme on doit utiliser `free` après `malloc`.

La fonction `feof` permet de tester si l'on est à la fin d'un fichier : elle est donc utile pour parcourir le contenu d'un fichier et déterminer lorsque l'on doit s'arrêter. Cette fonction renvoie la valeur 0 tant que l'on est pas à la fin du fichier et une valeur non nulle si on est à la fin du fichier.

Exemple 5.5. Manipulation de fichier avec contrôle des erreurs.

```
#include <stdio.h>
#define SIZE 80
int main (void)
{
  char name[] = "test.csv";
  FILE *file = fopen (name, "r");
  if (file == NULL)
    printf ("Error: could not open %s\n", name);
  else
  {
    char line[SIZE];
    while (!feof (file))
    {
      fgets (line, SIZE, file);
      printf ("%s", line);
    }
    fclose (file);
  }
  return 0;
}
```

Ce programme ouvre le fichier `test.csv` en mode lecture, lit et affiche son contenu ligne par ligne. Si le fichier n'existe pas il affiche un message d'erreur.

1. Ce qui peut arriver si le fichier n'existe pas, est déjà utilisé par un autre programme ou est un fichier système...

5.4 La fonction *main*

Nous finissons ce support de cours en revenant sur la fonction principale du programme, qui est appelée lors de l'exécution. Jusqu'à présent nous avons donné des exemples utilisant une version simplifiée de la fonction `main`, sans arguments. Or il est souvent utile d'utiliser ces arguments pour contrôler l'exécution du programme. Par exemple imaginons que nous souhaitons écrire un utilitaire permettant d'afficher le contenu d'un fichier. Il faut bien, au moment où on appelle ce programme [2] spécifier le nom du fichier ! Cela est rendu possible en utilisant les arguments de la fonction `main` :

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

Ces deux arguments sont :

- `argc` : un entier contenant le nombre d'arguments,
- `argv` : un tableau de `argc` chaînes de caractères, permettant de récupérer la valeur des arguments.

Remarque 5.11. `argc` vaut toujours au moins 1, puisque par convention le nom du programme est passé en premier argument à la fonction `main`.

Exemple 5.6. Exemple d'utilisation des arguments de la fonction `main` pour réaliser un utilitaire d'affichage du contenu d'un fichier.

```
#include <stdlib.h>//malloc(), free(), exit()
#include <stdio.h> //printf(), fopen(), fclose()
int main (int argc, char *argv[])
{
  FILE * pFile;
  char * buffer;
  size_t size;
  if (argc != 2) //check number of arguments
  {
    printf ("Usage: %s file\n", argv[0]);
    return -1;
  }
  pFile = fopen (argv[1] , "r");
  if (pFile==NULL)
```

2. On peut ici penser à l'instruction `cat` sur un système unix.

```
  {
    printf ("File error %s\n", argv[1]);
    exit (1);
  }
  fseek (pFile, 0, SEEK_END); //get file size
  size = (size_t)ftell (pFile);
  rewind (pFile);
  /* Allocate a buffer the size of the file */
  buffer = (char*) malloc (sizeof(char)*size);
  if (buffer == NULL)
  {
    printf ("Memory error\n");
    exit (2);
  }
  /* copy file into buffer */
  if (size != fread (buffer,1,size,pFile))
  {
    printf ("Reading error\n");
    exit (3);
  }
  /* print the buffer */
  printf ("%s", buffer);
  /* free memory */
  fclose (pFile);
  free (buffer);
  return 0;
}
```

Remarque 5.12. Lorsqu'une erreur est détectée on utilise la fonction `exit` qui permet de quitter un programme tout en s'assurant que les éventuels fichiers ouverts ont bien été fermés.