Cours de Programmation :

IUT de Villetaneuse — R&T 1ère année

Laure Petrucci

8 décembre 2009

Table des matières

Chapitre 1

Principes généraux

1.1 Pourquoi la programmation ?

1.1.1 Objectifs du cours

Ce cours a pour objectif d'initier les étudiants aux principes de base de la programmation. Parmi ces principes de base, seront étudiés :

- l'écriture d'algorithmes ;
- la mise en pratique au travers du langage de programmation `C`.

Ce cours ne présentera pas les éléments avancés de programmation tels que les listes, arbres, ...

L'apprentissage de l'algorithmique et de la programmation n'est pas intrinsèquement difficile, mais nécessite, pour être efficace :

- une *rigueur* dans la conception et la présentation des programmes ;
- une *pratique régulière* de la programmation en dehors des séances de travaux pratiques.

À l'issue du module de cours, un petit projet individuel sera à réaliser. Le but du projet est que les étudiants poursuivent leur expérience pratique en réalisant un programme plus conséquent par eux-même. Ils motiveront leurs choix de programmation en s'appuyant sur les différents concepts de base abordés.

1.1.2 Les langages de programmation

Les *langages de programmation* permettent d'écrire des *programmes* de manière compréhensible par un humain, et exploitable par un ordinateur. Les programmes ainsi développés sont des *applications informatiques* réalisant l'objectif pour lequel ils ont été conçus (calculs scientifiques, gestion de données, jeux, ...)

Les langages de programmation sont de divers types, fondés sur des paradigmes différents. Par exemple :

- les *langages impératifs* sont constitués de suites d'opérations à effectuer séquentiellement. Les langages `C`, Pascal, Cobol, Fortran, Ada sont des langages impératifs ;
- les *langages à objets* adoptent une vision où chaque entité en jeu dans le programme (*objet*) a un comportement et des caractéristiques propres, et interagit avec d'autres objets. Les langages de programmation à objets les plus utilisés sont `C++`, java, python ;
- les *langages fonctionnels* tels que `ML`, lisp, haskell, exploitent l'évaluation de fonctions. Ils sont particulièrement efficaces pour manipuler des données mathématiques.

Le *choix du langage* `C` dans le cadre de ce cours réside dans la diversité des concepts qu'il met en œuvre, son exploitation dans le milieu industriel, mais aussi les facilités qu'il offre à programmer des applications d'administration du système d'exploitation et des réseaux.

1.1.3 L'algorithmique

L'écriture d'un programme commence par sa *conception algorithmique*. Un *algorithme* décrit les opérations à effectuer pour résoudre le problème auquel on s'intéresse.

Un algorithme est écrit dans un *langage de description d'algorithmes* (`LDA`), proche du langage naturel et *indépendant du langage de programmation*. Par conséquent, lors de l'écriture d'un algorithme, on s'intéresse à la *logique de fonctionnement* du programme.

Un algorithme décrit un programme à un certain *niveau d'abstraction*. On peut en effet vouloir utiliser un niveau d'abstraction élevé pour décrire les grandes lignes du fonctionnement du programme, sans rentrer dans des détails, mais pour obtenir une vue générale. Ensuite, on pourra au fur et à mesure détailler l'algorithme, en le rendant de plus en plus « concret ».

Lorsque l'algorithme décrit le programme de manière suffisamment détaillée, il peut être *implémenté*, c'est-à-dire écrit dans un langage de programmation pouvant être ensuite exploité par la machine.

L'écriture d'un algorithme comme préalable à la programmation présente plusieurs avantages :
- la *réflexion* porte sur la suite d'opérations que le programme doit réaliser et non sur des détails d'implémentation (comme des erreurs de syntaxe) ;
- l'algorithme étant écrit sous une forme proche du langage naturel, il est *facilement compréhensible*. De plus un bon niveau d'abstraction permet de *s'affranchir de détails* et contribue également à une meilleure compréhension ;
- un même algorithme peut-être *implémenté dans des langages différents* choisis pour des raisons techniques spécifiques.

1.2 Étapes de construction d'un programme

Le texte d'un programme, écrit dans le langage de programmation, est appelé *code source*. L'exécution peut se dérouler de deux manières selon le type de langage de programmation utilisé.

Un programme écrit dans un *langage interprété* est analysé au fur et à mesure — ligne à ligne — pour être exécuté. Dans ce cas, chaque une interprétation des instructions du langage a lieu à chaque exécution. C'est le cas du *shell*.

Un programme, écrit dans un langage *compilé*, doit d'abord être traduit en *langage machine* avant de pouvoir être exécuté. Le fichier obtenu, appelé « exécutable » est alors compréhensible par la machine. Il peut être exécuté plusieurs fois, sans avoir à manipuler une nouvelle fois le code source. Pour obtenir l'exécutable, le code source passe par plusieurs étapes, comme schématisé dans la figure 1.1 :
- le *préprocesseur* effectue des substitutions de texte, inclusions de fichiers (tels que les en-têtes de bibliothèques prédéfinies) dans le code source du programme. Il produit un code source modifié ;
- le *compilateur* analyse le code source produit par le préprocesseur et le transforme en code objet ;
- l'*éditeur de liens* relie entre eux les codes objets produits par le compilateur et des bibliothèques requises, pour construire le code exécutable.

Par abus de langage, on parle de « compiler » un programme pour générer l'exécutable à partir du code source.

Pour l'instant, on se contente d'une commande simple qui génère l'exécutable à partir du code source : `gcc`.

Exemple 1.1 : Soit un fichier `monprog.c` contenant le code source d'un programme `C`. La commande :

```
$ gcc monprog.c -o monprog
```

génère un exécutable appelé `monprog`. Le caractère `$` représente le *prompt* de l'interpréteur de commandes. ◇

Si l'option **`-o`** n'est pas utilisée l'exécutable est appelé par défaut `a.out`.

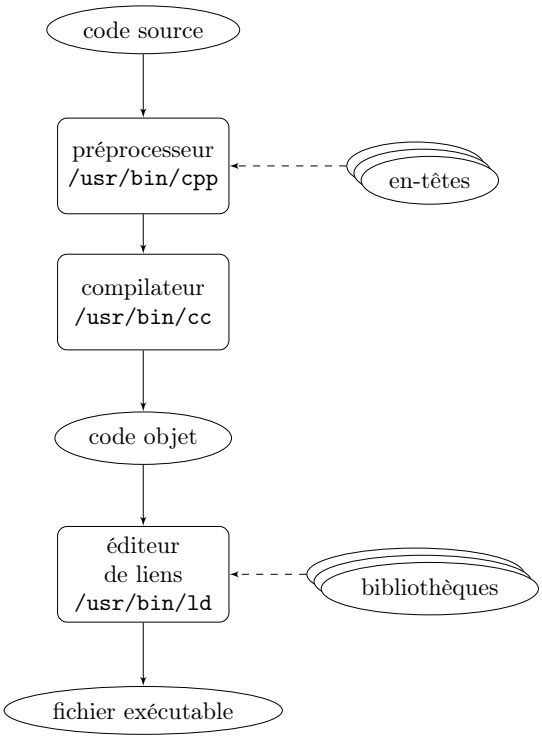


FIG. 1.1 – Transformation du code source en exécutable

1.3 Règles pour bien écrire un programme

Pour qu'un code source de grande taille puisse être facilement débogué, compris et maintenu, il est nécessaire de l'écrire « proprement ». Se conformer à quelques règles élémentaires permet de trouver les erreurs éventuelles plus facilement, rend le programme compréhensible par d'autres personnes que celui qui l'a écrit, et facilite ses évolutions futures.

1.3.1 Structure d'un programme

Un programme doit être *commenté* régulièrement. Les *commentaires* décrivent succinctement une partie du code. Dans le langage `C`, les commentaires sont du texte commençant par `/*` et finissant par `*/`. Les commentaires permettent de comprendre rapidement une partie du programme sans rentrer dans les détails du code source.

Exemple 1.2 :

```
/* ceci est un commentaire */
```

◇

Le fichier contenant le code source du programme comporte :

– un *en-tête* dans lequel se trouvent des déclarations concernant l'intégralité du programme. Il comporte des *inclusions de fichiers* tels que les en-têtes des bibliothèques prédéfinies, des définitions traitées par le préprocesseur, les déclarations des *variables globales*[1], et la

[1] Les variables globales sont connues de toutes les fonctions. Par conséquent, lorsque plusieurs fonctions modifient la valeur d'une même variable, on peut obtenir des valeurs incohérentes. L'utilisation des variables globales est à éviter absolument.

déclaration des *schémas des fonctions* du programme[2] ;
- un ensemble de *fonctions* réalisant les opérations nécessaires à résoudre le problème. Parmi ces fonctions est définie une « fonction principale » qui est le *point de départ de l'exécution* du programme. En `C` cette fonction se nomme `main`[3].

Les *inclusions de fichiers* se font à l'aide de `#include` suivi du nom de fichier à inclure, entre guillemets (`" "`) si le fichier est référencé par sa référence relative ou absolue, et entre chevrons (`< >`) si c'est un en-tête de bibliothèque. Les fichiers de déclarations ainsi inclus ont en général l'extension `.h`.

Exemple 1.3 : Les instructions :

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

déclarent l'inclusion du fichier d'en-tête de la bibliothèque `stdio` (*standard input output*, définissant les fonctions d'entrée/sortie), et du fichier `monfichier.h` du répertoire courant. ◇

1.3.2 Variables et fonctions

Un programme écrit dans un langage impératif utilise des variables permettant de stocker des valeurs en mémoire. Le langage `C` est un *langage typé*, c'est-à-dire que les *variables* sont associées au type des données qu'elles contiennent. Ce type définit non seulement quel genre de valeur elles peuvent contenir mais aussi la place mémoire utilisée pour la stocker. Les types de base seront présentés dans la section 2.1.

Les *fonctions* constituent des suites d'instructions que l'on peut exécuter plusieurs fois dans le programme. Elles permettent également de structurer l'écriture du programme en le découpant en entités plus petites réalisant chacune un objectif plus simple. Une fonction prend en général un ou plusieurs paramètres typés, ce qui permet des exécutions avec des résultats différents, et renvoie un résultat également typé. Le *schéma d'une fonction* explicite ces différents éléments. Le schéma et la définition d'une fonction seront détaillés dans la section 2.4.

Pour améliorer la lisibilité du programme, on choisira des *noms de variables et de fonctions significatifs*. En effet, une variable `x` peut représenter n'importe quoi, alors qu'une variable `taille` sera utilisée pour contenir la taille d'un élément manipulé.

1.3.3 Blocs d'instructions et indentation

Certaines instructions, comme les instructions conditionnelles (voir section 2.5), conduisent à constituer des *blocs d'instructions* qui pourront être exécutés ou au contraire ignorés selon les valeurs de certaines variables. La séquence d'instructions définissant une fonction est également un bloc. En `C`, les blocs sont compris entre une accolade ouvrante `{` et une accolade fermante `}`.

Pour que les blocs d'instructions soient nettement identifiables, c'est-à-dire que l'on repère facilement leur début et leur fin, il sont *indentés*. L'*indentation* est un décalage du texte vers la droite. On utilise également une indentation quand l'instruction ne tient pas sur une ligne, pour montrer que les lignes suivantes constituent la suite de l'instruction. Les exemples présentés dans ce document respectent ces conseils sur le nommage et l'indentation. L'exemple 3.16 page 30 donne un aperçu de la présentation d'un programme.

[2] La déclaration des schémas de fonctions n'est pas obligatoire. Toutefois, elle permet d'une part d'éviter des erreurs de compilation lorsqu'une fonction est utilisée avant d'être définie, et d'autre part de connaître d'un seul coup d'œil l'ensemble des fonctions définies dans un fichier, ce qui est particulièrement pratique lorsque le programme est de taille conséquente.

[3] Le nom de la fonction principale peut être différent de `main`, mais cela sort du cadre de ce cours.

Chapitre 2

Mécanismes de base

Ce chapitre présente les concepts de base de l'algorithmique en langage de description d'algorithme (LDA) et leurs correspondants dans le langage `C`.

2.1 Types simples

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

2.1.1 Les types de base

Les types de base sont *prédéfinis* et permettent de construire des types plus complexes comme les tableaux ou les structures (voir les sections 3.4 et 4.2, respectivement).

Entiers

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

Alors qu'en LDA on ne se préoccupe pas de détails sur la représentation des entiers, on doit le faire en langage `C`. Pour des raison spécifiques au programme (faible utilisation mémoire ou manipulation d'entiers de grande valeur), on peut utiliser des entiers de différentes longueurs, signés ou non[2] :

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

Caractères

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

[1] Certains opérateurs sont *surchargés* : ils admettent plusieurs signatures. C'est par exemple le cas de `+` qui permet d'additionner des entiers, des réels et même des caractères.

[2] Pour les détails sur les représentations mémoire, se reporter au cours d'« architecture des ordinateurs » (module I2).

Réels

Un **réel** permet de représenter un nombre en *virgule flottante*. Comme pour les entiers, la représentation des réels en langage `C` donne le choix entre différentes tailles :
- le type **float** représente les réels en *simple précision* ;
- la *double précision* est représentée par le type **double** ;
- il est également possible d'obtenir une précision étendue avec **long double**.

2.1.2 Constantes

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

Constantes entières

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

Exemple 2.1 : La valeur entière 13 s'écrit indifféremment `13`, `015` ou `0xD`. ◇

Constantes caractères

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

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

TAB. 2.1 – Principales représentations symboliques des caractères

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

Exemple 2.2 : Les constantes `'a'`, `'\141'`, `\x61'` représentent la lettre *a*. ◇

Constantes chaînes de caractères

Une *chaîne de caractères*[3] est une suite d'un nombre quelconque de caractères. S'il n'y a pas de caractère, on parle de *chaîne vide*. Une chaîne de caractères est délimitée par des *guillements*. Cela permet de différencier une constante chaîne de caractères d'un nom de variable ou de fonction ainsi que d'un caractère, pour une chaîne réduite à un seul caractère. Tous les types de notations de caractères peuvent être utilisés dans une chaîne de caractères.

Exemple 2.3 : La chaîne de caractères `"première\nseconde"` est constituée de *première*, puis à la ligne suivante *seconde*. ◇

Constantes réelles

Les *constantes réelles* sont différenciées des constantes entières par l'écriture d'un `.` qui représente la virgule[4]. Elles peuvent également être exprimées sous *forme scientifique* en précisant la puissance de 10 par la lettre `e` ou `E`. Enfin, le format peut être exprimé explicitement avec un **f** ou **F** pour un **float**, ou un **l** ou un **L** pour un **long double**.

Exemple 2.4 : Les valeurs suivantes sont des constantes réelles : `152.`, `165.3`, `152.254e6`, `165.54e-5`, `153.5f`, `198.86543227L`. ◇

2.2 Variables

2.2.1 Déclaration de variables

Les *variables* doivent être déclarées avant de pouvoir être utilisées.
Que ce soit en LDA ou en `C`, on doit indiquer le nom de chaque variable ainsi que son type[5].

Exemple 2.5 :

Variables : compteur, nb : entier
Variables : car : caractère

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

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

On remarque, dans l'exemple 2.5, que l'ordre dans les déclarations diffère entre le LDA et le `C`. De plus, en `C`, les instructions se terminent toutes par un `;`.

2.2.2 Affectation de valeur à une variable

Souvent, avant d'utiliser une variable pour la première fois, il faut l'*initialiser*, c'est-à-dire lui donner une valeur de départ. On effectue pour cela une *affectation de variable*. La même opération est utilisée pour stocker une valeur dans une variable au cours de l'exécution du programme.

En LDA, l'affectation est représentée par ←, qui peut se lire « prend la valeur ». En `C`, l'opérateur `=` est utilisé.

[3] La manipulation de chaînes de caractères dans le langage `C` sera détaillée dans la section 3.3.
[4] En anglais, on utilise un point à la place d'une virgule.
[5] En `C`, on peut également préciser un mode de stockage de la variable, par exemple pour la forcer à être dans un registre, mais ces notions de programmation avancées dépassent le cadre de ce cours.

Exemple 2.6 :

```
compteur ← 5
car ← 'u'
```

```
compteur = 5;
car = 'u';
```

montrent des affectations de la valeur `5` à la variable `compteur` et de la valeur `u` (de type caractère) à la variable `car` de l'exemple 2.5. ◇

2.3 Opérations simples

Nous présentons maintenant les principaux opérateurs permettant d'effectuer des calculs sur les entiers ainsi que l'affichage de messages. Ceci constitue la base nécessaire pour écrire les premiers programmes.

2.3.1 Opérateurs arithmétiques

Les *opérateurs arithmétiques* du tableau 2.2 peuvent être appliqués à des opérandes de type entier, réel ou caractère. Dans ce dernier cas, l'opération travaille sur les codes ASCII et le résultat est le caractère correspondant.

TAB. 2.2 – Opérateurs arithmétiques

Symbole	Signification
`-`	moins unaire
`*`	multiplication
`/`	division
`+`	addition
`-`	soustraction
`%`	modulo (reste d'une division entière)

Exemple 2.7 : Les instructions suivantes affectent la valeur `5` à la variable `compteur` puis le résultat d'un calcul (valant `17`) à la variable `nb`.

```
compteur ← 5
nb ← compteur * 3 + 2
```

```
compteur = 5;
nb = compteur * 3 + 2;
```

◇

2.3.2 Affichage de messages

En LDA, l'*affichage d'un message* ne tient pas compte du typage des variables et consiste simplement à écrire à l'écran un texte pouvant combiner des chaînes de caractères et des valeurs de variables :

Exemple 2.8 : Les instructions suivantes

```
compteur ← 5
écrire ("La variable compteur vaut " compteur ".")
```

affichent le message :

```
La variable compteur vaut 5.
```

◇

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

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

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

où *format* est une chaîne de caractères contenant le message à afficher. Si ce message contient des variables dont il faut afficher la valeur, le format d'affichage de chaque variable est précisé, selon les spécifications décrites dans le tableau 2.3. Lors de l'exécution, les spécifications des formats de variables sont remplacées par leur valeur, et traitées dans l'ordre d'apparition dans le message : la première spécification correspond à la première variable de la liste, la seconde spécification à la seconde variable, et ainsi de suite.

TAB. 2.3 – Spécification des formats d'affichage

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

Exemple 2.9 : Les instructions suivantes affectent la valeur `15` à la variable `compteur`, `9` à la variable `nb`, puis affichent ces valeurs en décimal, octal et hexadécimal.

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

L'affichage à l'écran est alors :

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

◇

2.4 Déclaration de fonctions

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

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

Lors de l'appel d'une fonction, l'exécution en cours est transférée à la *première instruction* que la fonction appelée comporte, et se termine soit à *sa dernière instruction*, soit à l'exécution de l'instruction retournant une valeur (**retourner** en LDA, `return` en `C`). L'exécution la *fonction appelante* reprend alors, en utilisant éventuellement la valeur retournée.

L'exécution d'un programme commence par la première instruction de la *fonction principale* généralement appelée `main` dans un programme `C`. La fonction `main` retourne un entier au processus appelant.

L'écriture d'une fonction comporte son *nom*, ses *paramètres avec leur type*, ainsi que le type de la valeur retournée. La syntaxe utilisée pour la déclaration de fonctions est en LDA :

Fonction `nom_fonction`(*paramètre_1 : type_1, ...*) : type_résultat
Variables : nom_variable : type_variable...
début
| Corps de la fonction
fin

où les variables déclarées sont celles utilisées par la fonction pour réaliser l'objectif souhaité, et le *corps de la fonction* est l'ensemble des instructions à exécuter.

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

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

Le corps de la fonction est délimité par des accolades `{` et `}`. De manière générale, en `C`, les accolades délimitent des *blocs d'instructions*.

Exemple 2.10 : Le programme suivant illustre un programme complet en LDA et en `C`, avec une fonction principale et une fonction appelée, `carre`. Cette fonction calcule le carré d'un entier `nb` passé en paramètre et le retourne à la fonction appelante. Dans la fonction principale, `carre` est appelée deux fois, avec les valeurs `2` et `5` comme paramètre.

Fonction `carre`(*nb : entier*) : entier
Variables : resultat : entier
début
| `/* calcul du carré de nb */`
| resultat ← nb * nb
| **retourner** (resultat)
fin

Fonction `principale`

Variables : res : entier
début
  `/* fonction principale */`
  res ← carre(2)
  `écrire` ("Le carré de 2 est " res ".\n")
  `écrire` ("Le carré de 5 est " carre(5) ".\n")
fin

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

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

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

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

/* fonction principale */
int main(){
    int res;

    res = carre(2);
    printf("Le carré de 2 est %d.\n",res);
    printf("Le carré de 5 est %d.\n",carre(5));
}
```

Le résultat de l'éxécution de ce programme est :

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

◇

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

2.5 Instructions conditionnelles

Les *instructions conditionnelles* permettent de conditionner l'exécution de certaines instructions *selon le résultat d'un test*. Elles se répartissent en un opérateur de test et des opérateurs de boucle permettant la répétition d'un bloc d'instructions.

2.5.1 Opérateurs booléens et de comparaison

Les *opérateurs booléens* et les *opérateurs de comparaison* permettent de tester si une expression est vraie ou fausse. En LDA, on peut utiliser le type `booléen`. Par contre, il n'existe pas en `C`. Les conventions suivantes sont alors utilisées :

- une expression non nulle (c'est-à-dire de valeur différente de 0) est toujours vraie ;
- une expression nulle est toujours fausse.

Les opérateurs booléens et les opérateurs de comparaison sont présentés dans le tableau 2.4. Il faut bien distinguer = qui réalise une affectation et == qui constitue le test d'égalité. En effet, une affectation est une opération qui renvoie systématiquement la valeur affectée.

TAB. 2.4 – Opérateurs booléens et de comparaison

LDA	C	Signification
non	`!`	négation
et	`&&`	et booléen
ou	`||`	ou booléen
$<$	`<`	inférieur
$\leq$	`<=`	inférieur ou égal
$>$	`>`	supérieur
$\geq$	`>=`	supérieur ou égal
$=$	`==`	égal
$\neq$	`!=`	différent

2.5.2 Tests

Les *instructions de test* permettent d'exécuter des instructions différentes selon les valeurs d'une expression conditionnelle.

Le test « si... alors... sinon... »

Cette instruction de test simple suit la structure :

si *condition* **alors**
| instructions_1
sinon
| instructions_2
finsi

```c
if (condition){
    instructions_1
} else {
    instructions_2
}
```

Si la condition est vraie, le bloc d'instructions `instructions_1` est exécuté. Si au contraire elle est fausse, c'est le bloc `instructions_2` qui est exécuté. La partie « **sinon** » est optionnelle. Si elle n'est pas précisée, dans le cas où la condition est fausse, l'exécution de cette structure se termine immédiatement. De plus, en `C`, les accolades peuvent être omises lorsque le bloc d'instructions est réduit à un seul élément[6]. Par contre les parenthèses `(` et `)` entourant la condition sont obligatoires.

Exemple 2.11 : Soient les instructions suivantes :

si *nb* $<$ *10* **alors**
| `écrire` ("nb est un chiffre")
sinon
| `écrire` ("nb est un nombre")
finsi

```c
if (nb < 10){
    printf("nb est un chiffre");
} else {
    printf("nb est un nombre");
}
```

Si la valeur de la variable `nb` est plus petite que `10`, le message « `nb est un chiffre` » est affiché. Dans le cas contraire, le message est « `nb est un nombre` ». ◇

[6] Cette remarque s'applique de manière générale à la plupart des blocs d'instructions.

L'énumération de « cas »

Lorsqu'une expression peut prendre plusieurs valeurs, chacune conduisant à l'exécution d'instructions différentes, on utilise une structure « **cas** » plutôt qu'une série de « **si**... **alors**... **sinon**... » imbriqués.

```
suivant expression faire
    cas valeur_1
     | instructions_1
    fincas
    cas valeur_2
     | instructions_2
    fincas
    ...
    autres cas
     | instructions_autres
    fincas
fincas
```

```
switch (expression){
    case valeur_1 :
        instructions_1
    case valeur_2 :
        instructions_2
    ...
    default : instructions_autres
}
```

Tout d'abord, l'`expression` est évaluée. Si sa valeur est `valeur_1`, le bloc d'instructions `instructions_1` est exécuté. Sinon, si sa valeur est `valeur_2`, le bloc d'instructions `instructions_2` est exécuté. Et ainsi de suite. Enfin, si aucune valeur ne correspond, les instructions par défaut `instructions_autres` sont exécutées. Cette clause (**autres cas**) est optionnelle.

En `C`, les instructions sont exécutées séquentiellement jusqu'à ce que l'on rencontre une instruction **break** d'*échappement d'un bloc*.

Exemple 2.12 : Le programme suivant affiche des messages différents selon la valeur de la variable `nb` :

```
suivant nb faire
    cas 1
     | écrire ("nb vaut 1")
    fincas
    cas 2
     | écrire ("nb vaut 2")
    fincas
    autres cas
     | écrire ("nb ne vaut ni 1 ni 2")
    fincas
fincas
```

```
switch (nb){
    case 1 :
        printf("nb vaut 1\n");
        break;
    case 2 :
        printf("nb vaut 1\n");
        break;
    default :
        printf("nb ne vaut ni 1 ni 2\n");
}
```

◇

2.5.3 Boucles

Les *boucles* sont utilisées pour répéter l'exécution les mêmes instructions avec des valeurs différentes de variables.

La boucle « tant que »

La structure de *boucle* « **tant que** » répète un bloc d'instructions tant qu'une condition est satisfaite. Elle s'écrit de la manière suivante :

tant que *condition* **faire**
| instructions
fintq

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

L'exécution des `instructions` est effectuée tant que la `condition` est vraie.

Exemple 2.13 : Le programme suivant affiche la parité des entiers de `1` à `5` :

Fonction `parite`(*nb : entier*)
début
| **si** *nb % 2 = 0* **alors**
| | écrire (nb " est pair")
| **sinon**
| | écrire (nb " est impair")
| **finsi**
fin

Fonction `principale`
Variables : i : entier
début
| i ← 1
| **tant que** $i \leq 5$ **faire**
| | parite(i)
| | i ← i+1
| **fintq**
fin

```
#include <stdio.h>

/* schémas des fonctions */
void parite(int nb);

/* parité d'un nombre */
void parite(int nb){
    if (nb % 2 == 0)
        printf("%d est pair\n",nb);
    else printf("%d est impair\n",nb);
}

int main(){
    int i;

    i = 1;
    while (i <= 5){
        parite(i);
        i = i + 1;
    }
}
```

Le résultat de l'exécution est alors :

```
1 est impair
2 est pair
3 est impair
4 est pair
5 est impair
```

◇

La boucle « pour »

La structure de boucle « **pour** » est utilisée pour faire prendre à une variable successivement toutes les valeurs d'un ensemble. Cet ensemble peut être décrit en LDA sous la forme d'une variable ou d'un intervalle de valeurs.

pour *variable* **dans** *ensemble* **faire**
| instructions
finpour

Les instructions sont exécutées pour toutes les valeurs de la `variable` dans l'`ensemble`.

pour *variable* **de** *min* **à** *max* **faire**
| instructions
finpour

Les instructions sont exécutées pour toutes les valeurs de la `variable` dans l'intervalle allant de la valeur `min` à la valeur `max`.

En `C`, la boucle « **pour** » est réalisée par l'instruction **for**, selon la structure :

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

L'`initialisation` permet d'affecter des valeurs initiales à des variables la première fois que l'on entre dans la boucle. Le corps de la boucle, c'est-à-dire les `instructions`, est exécuté tant que la `condition` est satisfaite. Enfin le `pas` indique les nouvelles affectations de variables à réaliser après l'exécution des `instructions` et avant de tester une nouvelle fois la `condition`.

Exemple 2.14 : La fonction principale de l'exemple 2.13 peut être écrite avec une boucle « **pour** » :

Fonction `principale`
Variables : i : entier
début
| **pour** *i* **de** *1* **à** *5* **faire**
| | parite(i)
| **finpour**
fin

```
int main(){
    int i;

    for (i = 1; i <= 5; i = i + 1){
        parite(i);
    }
}
```

◇

2.6 Codage binaire et opérateurs associés

Les différentes variables sont stockées en mémoire comme un *ensemble de bits* qui sont directement manipulables. Par exemple, un caractère est généralement représenté par un octet, soit 8 bits.

2.6.1 Opérations bit à bit

Les *opérations bit à bit* permettent d'effectuer ce genre de manipulation. Elles sont résumées dans le tableau 2.5.

Tab. 2.5 – Opérations bit à bit

LDA	`C`	Signification
¬	`~`	négation
∧	`&`	*et* bit à bit
∨	`|`	*ou* bit à bit

L'opérateur de *négation* inverse les bits (les `0` sont transformés en `1` et vice-versa). Les opérateurs *et* et *ou* travaillent sur les bits de même rang des deux opérandes.

Exemple 2.15 : Soit le programme suivant :

Fonction `principale`

Variables : nb1, nb2 : entier
début
  nb1 ← 3
  nb2 ← 9
  `écrire` (nb1 " et " nb2 " = " (nb1 ∧ nb2))
  `écrire` (nb1 " ou " nb2 " = " (nb1 ∨ nb2))
fin

```c
#include <stdio.h>

int main(){
    int nb1, nb2;

    nb1 = 3;
    nb2 = 9;
    printf("%d et %d = %d\n",nb1,nb2,nb1 & nb2);
    printf("%d ou %d = %d\n",nb1,nb2,nb1 | nb2);
}
```

Le résultat est :

```
3 et 9 = 1
3 ou 9 = 11
```

En effet :

$$\begin{array}{rr} & 0011 \\ \wedge & 1001 \\ \hline & 0001 \end{array} \qquad \begin{array}{rr} & 0011 \\ \vee & 1001 \\ \hline & 1011 \end{array}$$

◇

2.6.2 Décalages

Il est également possible de *décaler les bits* d'une zone mémoire. L'effet dépend bien sûr de la taille de la zone traitée.

L'opérateur `décaler_gauche (`x `de` n `bits)` décale la valeur binaire de la variable x de n bits vers la gauche. Les bits sortant de la zone mémoire sont perdus, et des `0` sont ajoutés à droite. En `C`, ceci s'écrit : `x << n`.

Exemple 2.16 : Supposons que les entiers sont non signés et codés sur 8 bits.

nb1 ← 5
`décaler_gauche` (nb1 de 4 bits)
nb2 ← 235
`décaler_gauche` (nb2 de 2 bits)

```c
nb1 = 5;
nb1 = nb1 << 4;
nb2 = 235;
nb2 = nb2 << 2;
```

Les entiers `nb1`et `nb2` sont représentés en mémoire, avant le décalage par :

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
`nb1`	0	0	0	0	0	1	0	1

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
`nb2`	1	1	1	0	1	0	1	1

Après les instructions de décalage, ils sont représentés par :

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
`nb1`	0	1	0	1	0	0	0	0

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
`nb2`	1	0	1	0	1	1	0	0

Après ces instructions, `nb1` vaut 80 et `nb2` vaut 172. ◇

De manière similaire, l'opérateur `décaler_droite (`x `de` n `bits)` décale la valeur binaire de la variable x de n bits vers la droite[7]. Les bits sortant de la zone mémoire sont perdus, et des `0` sont ajoutés à gauche. En `C`, ceci s'écrit : `x >> n`.

Exemple 2.17 : Supposons que les entiers sont non signés et codés sur 8 bits.

nb1 ← 5
`décaler_droite` (nb1 de 4 bits)
nb2 ← 235
`décaler_droite` (nb2 de 2 bits)

```
nb1 = 5;
nb1 = nb1 >> 4;
nb2 = 235;
nb2 = nb2 >> 2;
```

Les entiers `nb1`et `nb2` sont représentés en mémoire, avant le décalage par :

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
`nb1`	0	0	0	0	0	1	0	1

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
`nb2`	1	1	1	0	1	0	1	1

Après les instructions de décalage, ils sont représentés par :

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
`nb1`	0	0	0	0	0	0	0	0

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
`nb2`	0	0	1	1	1	0	1	0

Après ces instructions, `nb1` vaut 0 et `nb2` vaut 58. ◇

2.7 Le typage

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

2.7.1 Taille d'un type

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

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

Exemple 2.18 :

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

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

[7] Lorsque l'on manipule des valeurs signées, le décalage à droite est arithmétique.

2.7.2 Changement de type (*cast*)

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

Exemple 2.19 :

```
1  #include <stdio.h>
2
3  int main(){
4      char c;
5      int i;
6
7      c = 'a';
8      i = (int) c;
9      printf("c = %c\n",c);
10     printf("i = %d\n",i);
11 }
```

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

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

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

Chapitre 3

Pointeurs, chaînes de caractères et tableaux

3.1 Les pointeurs

3.1.1 Qu'est-ce qu'un pointeur ?

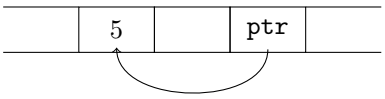
Un *pointeur* est une *adresse*.

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

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

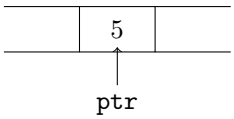
Exemple 3.1 :

```
int *ptr;
```



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

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



◇

3.1.2 Opérateurs sur les pointeurs

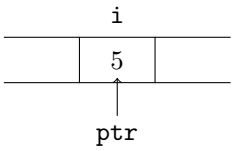
La manipulation des pointeurs utilise deux opérateurs principaux : l'opérateur d'*adresse* et l'opérateur d'*indirection* (ou de *contenu*).

Adresse

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

Exemple 3.2 :

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



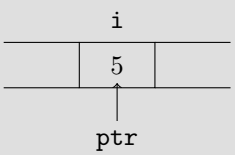
Dans ce programme, on déclare deux variables : `i` de type *entier* et `ptr` de type *pointeur sur un entier*. Ligne 4, la valeur `5` est affectée à la variable `i`, et ligne 5, l'*adresse* de la variable `i` est affectée à la variable `ptr`. Ce pointeur pointe donc sur `i` qui contient la valeur `5`.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

Indirection (contenu)

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

Exemple 3.3 :

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



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

Pointeur nul

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

3.1.3 Affectation de pointeur et typage

Un pointeur sur un type donné ne peut pointer que sur une variable de ce type. Par exemple, un pointeur sur un entier ne peut pas pointer sur une variable de type caractère. En effet, le type de la variable permet de déterminer la taille de la zone mémoire à prendre en compte. Lorsque des différences de type sont nécessaires, ce problème est contourné en utilisant à bon escient la conversion explicite de types (*cast*, voir section 2.7.2).

3.1.4 Allocation et libération de mémoire

Un pointeur peut soit pointer sur une variable déjà déclarée par ailleurs, et par conséquent allouée, soit sur une nouvelle zone mémoire. Il faut alors réserver cette zone mémoire pour éviter qu'elle soit utilisée par d'autres variables et donc modifiée de manière inattendue.

Les fonctions `malloc` d'*allocation de mémoire* et `free` de *libération de mémoire* sont définies dans la bibliothèque `stdlib.h` qui est inclue par :

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

Allocation de mémoire

Pour allouer une zone mémoire, on utilise la fonction `malloc` :

```
void *malloc(int taille);
```

où *taille* est la taille de la zone mémoire à allouer. Elle dépend du nombre et du type d'éléments que l'on veut ranger à cet emplacement. Il faut donc prévoir une taille suffisante en fonction des besoins d'utilisation. Le pointeur renvoyé est l'adresse de la zone mémoire réservée par la fonction `malloc`.

Exemple 3.4 :

```
int *ptr;

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

déclare un pointeur `ptr` sur des entiers, puis alloue une zone de la taille de 4 entiers. La fonction `malloc` retournant un pointeur non typé, on utilise une conversion explicite de type (*cast*, voir section 2.7.2) pour assigner le bon type à la variable `ptr`. ◇

Libération de mémoire

Lorsqu'une zone mémoire devient inutilisée, il faut la libérer par exemple pour un usage ultérieur. Pour cela, on utilise `free`(*pointeur*).

Exemple 3.5 :

```
int *ptr;

ptr = (int *) malloc(4*sizeof(int));
...
free(ptr);
```

Lorsque l'on a complètement fini d'utiliser la zone mémoire pointée par `ptr`, on la libère. ◇

3.1.5 Arithmétique des pointeurs

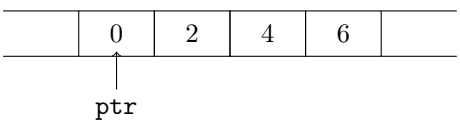
Diverses opérations arithmétiques permettent de manipuler les pointeurs.

Incrémentation et décrémentation de pointeur

Les *addition* et *soustraction* d'une valeur entière à un pointeur permettent de se déplacer dans la mémoire. La valeur entière indique le nombre d'éléments (du type pointé) constituant le déplacement.

Exemple 3.6 :

```
1 int i;
2 int *ptr;
3
4 ptr = (char *) malloc(4*sizeof(int));
5 for (i=0;i<4;i++)
6     *(ptr+i)=2*i;
```



Tout d'abord, une zone mémoire pouvant contenir 4 entiers est réservée ligne 4, et son adresse affectée au pointeur `ptr`. Ensuite, la boucle `for` permet d'affecter 4 valeurs, une par une, dans cet espace mémoire. L'affectation est réalisée ligne 6. On considère l'adresse mémoire à l'endroit pointé par `ptr` décalé de `i` éléments (`ptr + i`). Son contenu (utilisation de l'opérateur `*`) prend comme valeur le double de `i`. On obtient alors un contenu de mémoire comme représenté sur le schéma. ◇

Comparaison de pointeurs

La comparaison de pointeurs consiste en la comparaison des adresses, en particulier pour vérifier si deux pointeurs pointent sur le même objet ou si un pointeur est nul.

Exemple 3.7 :

```
1 int i;
2 int *ptr;
3
4 ptr = (char *) malloc(4*sizeof(int));
5 if (ptr == NULL)
6     return();
7 for (i=0;i<4;i++)
8     *(ptr+i)=2*i;
```

Ce programme complète celui de l'exemple 3.6 en testant, à la ligne 5, si le pointeur est nul. Ce cas se produit lorsque la fonction `malloc` a retourné un pointeur nul parce qu'elle n'a pas pu trouver une zone mémoire libre de taille suffisante. La fonction en cours est alors arrêtée par un retour à la fonction appelante (ligne 6). Sinon, quand l'allocation s'est bien passée, l'exécution se poursuit avec la boucle `for`. ◇

3.2 Passage de paramètres

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

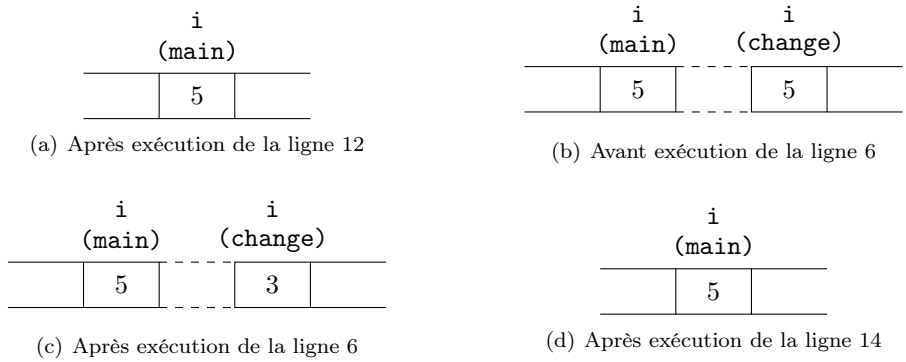
3.2.1 Passage de paramètres par valeur

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

Exemple 3.8 : Soit le programme suivant :

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

```
10 int main(){
11     int i;
12     i = 5;
13     printf("i=%d\n",i);
14     change(i);
15     printf("i=%d\n",i);
16 }
```



(a) Après exécution de la ligne 12

(b) Avant exécution de la ligne 6

(c) Après exécution de la ligne 6

(d) Après exécution de la ligne 14

FIG. 3.1 – Passage de paramètre par valeur

Dans la fonction principale, `main`, est déclarée une variable `i`, initialisée ligne 12. L'état de la mémoire après l'exécution de cette instruction est décrit dans la figure 3.1(a). Seule la variable `i` de la fonction `main` est présente en mémoire et sa valeur est `5`. L'instruction ligne 13 affirche `i=5`. La fonction `change` est ensuite appelée avec passage de la variable `i` par valeur. Une copie de la variable est donc créée en mémoire et affectée au paramètre de la fonction `change` (qui s'appelle également `i`). L'état de la mémoire est alors comme indiqué dans la figure 3.1(b). L'instruction ligne 5 affiche donc `i=5`. Puis, la valeur de `i` est changée ligne 6. Une seule variable `i` est connue de la fonction `change`, et c'est par conséquent celle-là qui est modifiée (voir figure 3.1(c)). La ligne 7 affiche alors `i=3`. À ce moment-là, l'exécution de `change` est terminée et les variables associées libérées. Alors, après l'exécution de la ligne 14, la mémoire est dans le même état qu'avant cet appel, comme décrit par la figure 3.1(d). ◇

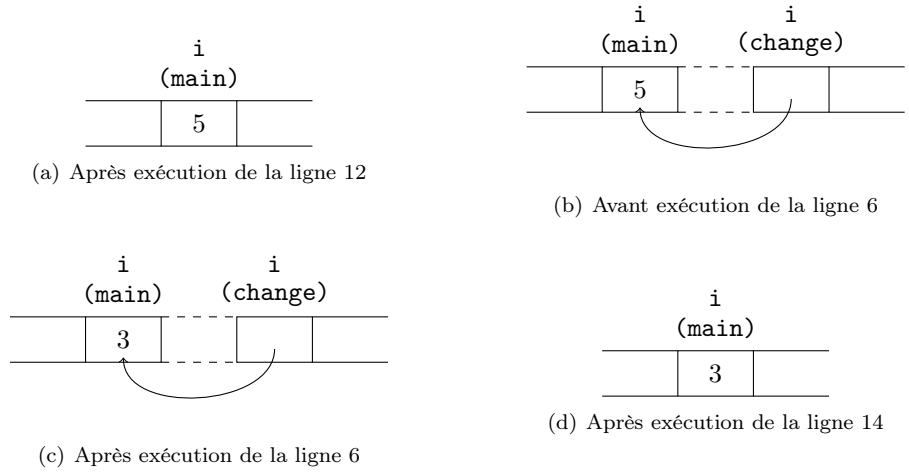
3.2.2 Passage de paramètres par adresse

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

Exemple 3.9 : Modifions le programme de l'exemple 3.8 pour passer le paramètre `i` par adresse :

```
1 #include <stdio.h>
2 void change(int *i);
3
4 void change(int *i) {
5     printf("i=%d\n",*i);
6     *i = 3;
7     printf("i=%d\n",*i);
8 }
```

```
9
10 int main(){
11     int i;
12     i = 5;
13     printf("i=%d\n",i);
14     change(&i);
15     printf("i=%d\n",i);
16 }
```



(a) Après exécution de la ligne 12

(b) Avant exécution de la ligne 6

(c) Après exécution de la ligne 6

(d) Après exécution de la ligne 14

FIG. 3.2 – Passage de paramètre par adresse

Dans la fonction principale, `main`, est déclarée une variable `i`, initialisée ligne 12. L'état de la mémoire après l'exécution de cette instruction est décrit dans la figure 3.2(a). Seule la variable `i` de la fonction `main` est présente en mémoire et sa valeur est `5`. L'instruction ligne 13 affirche `i=5`. La fonction `change` est ensuite appelée avec passage de la variable `i` par adresse. Une variable est donc créée en mémoire et affectée au paramètre de la fonction `change`, qui s'appelle également `i`, mais qui est un pointeur sur l'adresse de la variable `i` de la fonction `main`. L'état de la mémoire est alors comme indiqué dans la figure 3.2(b). L'instruction ligne 5 affiche donc `i=5`. Puis, la valeur de l'adresse pointée par `i` dans la fonction `change` est modifiée ligne 6 (voir figure 3.2(c)). La ligne 7 affiche alors `i=3`. À ce moment-là, l'exécution de `change` est terminée et les variables associées libérées. Alors, après l'exécution de la ligne 14, la mémoire est dans l'état décrit par la figure 3.2(d). ◇

3.3 Chaînes de caractères

La manipulation des chaînes de caractères est basée sur l'utilisation de pointeurs.

3.3.1 Déclaration et contenu d'une chaîne de caractères

Une *chaîne de caractères* est désignée en `C` par un *pointeur sur son premier caractère*. Elle est par conséquent déclarée par :

```
char *chaine;
```

Pour pouvoir l'utiliser, il faut qu'une zone mémoire de taille suffisante ait été réservée au préalable, soit en utilisant la fonction `malloc` (voir section 3.1.4), soit en initialisant la variable *au moment de sa déclaration*.

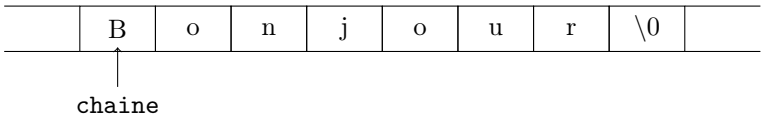
Exemple 3.10 :

```
char *chaine="Bonjour";
```

déclare une chaîne de caractères `chaîne` dans une zone mémoire de la taille pouvant contenir `Bonjour`. ◇

Si le pointeur permet de déterminer l'adresse du premier caractère, il faut également qu'un mécanisme permettant de détecter la *fin de chaîne* soit mis en œuvre. On utilise pour cela un *caractère nul* (de code ACSII 0, écrit `\0`). La *taille de la zone mémoire* doit donc correspondre à la *taille maximale de la chaîne plus 1* (pour stocker le caractère nul).

Exemple 3.11 : La représentation en mémoire de l'exemple 3.10 est :



◇

3.3.2 Fonctions manipulant les chaînes de caractères

Plusieurs fonctions prédéfinies permettent de manipuler des chaînes de caractères. Elles sont définies dans la bibliothèque `string.h` qu'il faut inclure :

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

Attention : ces fonctions supposent que les zones mémoire utilisées par les chaînes de caractères manipulées ont été correctement allouées.

Copie d'une chaîne de caractères

La fonction `strcpy` (*string copy*) permet de copier une chaîne de caractères dans une autre. Son schéma est :

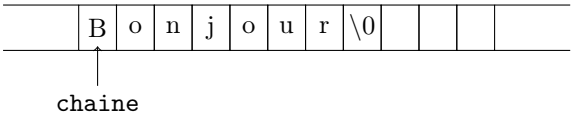
```
char *strcpy(char *chaine1, char *chaine2);
```

Cette fonction copie `chaine2` dans `chaine1`, y compris le caractère nul de fin de chaîne. Elle renvoie `chaine1`. Cette fonction peut être utilisée avec `chaine2` qui est une variable ou une constante de type chaîne de caractères (entre guillemets).

Exemple 3.12 :

```
char *chaine;

chaine = (char *) malloc(10);
strcpy(chaine,"Bonjour");
```



déclare une chaîne de caractères `chaine`. Ensuite, une zone mémoire pouvant contenir 10 caractères est allouée. Enfin, la chaîne de caractères `Bonjour` est copiée, d'où le résultat indiqué sur le schéma. ◇

La fonction `strncpy` :

```
char *strncpy(char *chaine1, char *chaine2, int n);
```

est similaire à `strcpy` mais ne copie qu'au plus `n` caractères. Si la chaîne à copier est plus courte, `chaine1` est complétée avec des caractères nuls, si elle est plus longue, seuls `n` caractères sont copiés, et le caractère nul n'est pas positionné à la fin de `chaine1`.

Concaténation de chaînes de caractères

La fonction `strcat` (*string concatenate*) permet de concaténer deux chaînes de caractères, c'est-à-dire mettre une chaîne à la suite de l'autre.

```
char *strcat(char *chaine1, char *chaine2);
```

copie `chaine2` à la fin de `chaine1`. Par conséquent, le caractère nul terminant `chaine1` est supprimé, les caractères de `chaine2` sont copiés dans `chaine1` y compris le caractère nul, qui termine ainsi la nouvelle chaîne de caractères `chaine1`.

De façon similaire, une fonction `strncat` copie au plus `n` caractères de `chaine2` et termine la nouvelle `chaine1` par un caractère nul :

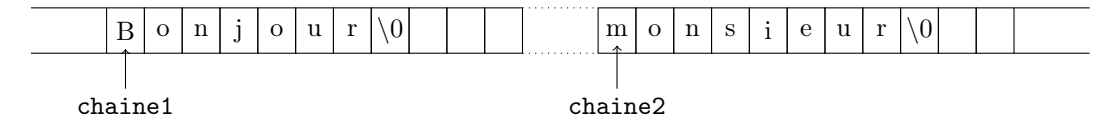
```
char *strncat(char *chaine1, char *chaine2, int n);
```

Ces deux fonctions renvoient le pointeur `chaine1`.

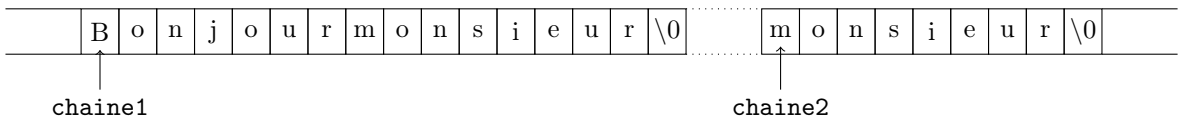
Exemple 3.13 : Soit le programme suivant :

```
1 char *chaine1,*chaine2;
2
3 chaine1 = (char *) malloc(20);
4 chaine2 = (char *) malloc(10);
5 strcpy(chaine1,"Bonjour");
6 strcpy(chaine2,"monsieur");
7 strcat(chaine1,chaine2);
```

Les zones mémoire pour 2 chaînes de caractères `chaine1` et `chaine2` est allouée aux lignes 3 et 4. En ligne 5, `Bonjour` est copié dans`chaine1` et `monsieur` dans `chaine2`, ligne 6. La description de la mémoire à ce moment-là est donnée dans la figure3.3(a). Une concaténation est effectuée ligne 7. Les caractères de `chaine2` sont copiés dans `chaine1` à partir de l'endroit où se trouvait auparavant le caractère nul, comme indiqué dans la figure 3.3(b).



(a) Avant exécution de la ligne 7



(b) Après exécution de la ligne 7

FIG. 3.3 – Concaténation de chaînes de caractères

◇

Longueur d'une chaîne de caractères

La fonction `strlen` (*string length*) retourne la longueur d'une chaîne de caractères :

```
int strlen(char *chaine);
```

Le nombre de caractères considéré n'inclut pas le caractère nul, car celui-ci n'est qu'un artifice permettant de trouver la fin de la chaîne.

Exemple 3.14 : La chaîne de caractères `chaine1` de l'exemple 3.13 représentée dans la figure 3.3(a) est de longueur 7. ◇

Comparaison de chaînes de caractères

La fonction `strcmp` compare deux chaînes de caractères :

```
int strcmp(char *chaine1, char *chaine2);
```

Cette comparaison est effectuée en utilisant l'ordre lexicographique (*grosso modo* l'ordre du dictionnaire). Les codes ASCII des deux premières lettres sont comparés, s'il sont égaux, on compare ceux des secondes lettres, et ainsi de suite jusqu'à la fin de la plus courte des deux chaînes de caractères. L'entier retourné est :

- > 0 si `chaine1` $>$ `chaine2`
- < 0 si `chaine1` $<$ `chaine2`
- $= 0$ si `chaine1` $=$ `chaine2`

Exemple 3.15 : Soit le programme suivant :

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

int main(){
    char *chaine1, *chaine2;

    /* initialisation des deux chaînes de caractères */
    chaine1 = (char *) malloc(strlen("bonjour") + 1);
    strcpy(chaine1,"bonjour");
    chaine2 = (char *) malloc(strlen("bonsoir") + 1);
    strcpy(chaine2,"bonsoir");
    /* comparaison des deux chaînes de caractères */
    printf("strcmp(chaine1,chaine2)=%d\n",strcmp(chaine1,chaine2));
    printf("strcmp(chaine2,chaine1)=%d\n",strcmp(chaine2,chaine1));
    printf("strcmp(chaine1,chaine1)=%d\n",strcmp(chaine1,chaine1));
    /* libération de la mémoire */
    free(chaine1);
    free(chaine2);
}
```

Les bibliothèques nécessaires sont incluses au début du programme : `stdio.h` pour utiliser `printf`, `stdlib.h` pour `malloc` et `free`, `string.h` pour `strlen`, `strcpy` et `strcmp`.

La mémoire nécessaire pour stocker les chaînes de caractères est réservée par `malloc`. On note que la taille réservée est calculée pour être exactement ce dont on a besoin. Pour ce faire, la longueur de la chaîne de caractères à stocker est calculée, à laquelle on ajoute 1 pour pouvoir écrire le caractère nul de fin de chaîne.

L'exécution de ce programme affiche :

```
strcmp(chaine1,chaine2)=-9
strcmp(chaine2,chaine1)=9
strcmp(chaine1,chaine1)=0
```

La chaîne de caractères `chaine1` est plus petite que `chaine2`. En effet, les trois premiers caractères sont identiques, le quatrième est `j` dans `chaine1` qui est plus petit que le quatrième caractère de `chaine2`, `s`. Lorsque l'on compare deux chaînes de caractères identiques, comme c'est le cas pour le troisième affichage, le résultat de la comparaison est nul. ◇

3.3.3 Traitement de chaînes avec l'arithmétique des pointeurs

Le contenu d'une chaîne de caractères peut être directement modifié ou manipulé en utilisant l'arithmétique des pointeurs. Pour cela, on accède aux adresses mémoire des caractères concernés, à l'aide du pointeur de chaîne de caractères et de la position du caractère par rapport au début de chaîne.

Exemple 3.16 : Soit le programme suivant qui affiche un par un les mots d'une phrase :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(){
6      char *chaine, *mot1, *mot2;
7      int i;
8
9      /* chaîne de caractères à afficher mot à mot */
10     chaine = (char *) malloc(100);
11     strcpy(chaine,"Un programme trouvant les mots d'une phrase");
12     /* variables repérant le début d'un mot et du mot suivant */
13     mot1 = chaine;
14     mot2 = chaine;
15     /* découpage de la chaîne de caractères */
16     while (*mot2 != '\0') {
17         /* la phrase n'est pas terminée */
18         /* positionnement au début du nouveau mot */
19         mot1 = mot2;
20         /* repérage de la fin du mot */
21         for (i = 0;
22             *(mot1 + i) != ' ' && *(mot1 + i) != '\0';
23             i++);
24         /* découpage du mot */
25         if (*(mot1 + i) != '\0') {
26             /* la phrase n'est pas terminée */
27             *(mot1 + i) = '\0';
28             mot2 = mot1 + i + 1;
29             printf("%s\n",mot1);
30             *(mot1 + i) = ' ';
31         } else {/* la fin de la phrase est atteinte */
32             printf("%s\n",mot1);
33             mot2 = mot1+i;
34         }
```

```
35      }
36      free(chaine);
37  }
```

Tout d'abord, la zone mémoire suffisamment grande pour la chaîne de caractères à manipuler est allouée en ligne 10. Puis, ligne 11, la chaîne de caractères est initialisée avec une phrase. Les deux pointeurs `mot1` et `mot2` pointent sur la même adresse que `chaine`. Il ne faut pas faire d'allocation pour ces pointeurs car ils contiennent une adresse dans une zone déjà allouée.

Le pointeur `mot1` est utilisé pour repérer le début d'un mot, et le pointeur `mot2` pointera sur le mot suivant ou sur le caractère nul lorsque la fin de la phrase est atteinte. La boucle `while` (lignes 16–35) est donc exécutée jusqu'à ce que la fin de la phrase soit atteinte.

Ligne 19, le pointeur `mot1` prend l'adresse du début du mot suivant. Puis ce nouveau mot est parcouru par la boucle `for` (lignes 21–23) depuis sa première lettre jusqu'à la dernière. La dernière lettre est atteinte lorsque le caractère que l'on examine est un espace (auquel cas, il y a séparation d'avec le mot suivant) ou un caractère nul (et la phrase est terminée).

Si la sortie de boucle a eu lieu sur lecture d'un espace, on remplace celui-ci par un caractère nul, ligne 27. Le pointeur sur le mot suivant, `mot2` est alors positionné pour pointer sur le caractère d'après (ligne 28). Le mot lu peut être affiché : il commence à l'adresse `mot1` et termine au caractère nul qui a remplacé le caractère espace lu. Ligne 30, l'espace est remis dans la chaîne de caractères.

Si la sortie de boucle a eu lieu à la lecture d'un caractère nul, le mot peut être directement affiché (ligne 32). En effet, il commence à l'adresse pointée par `mot1` et se termine au caractère nul de fin de phrase. ◇

3.4 Tableaux

Les tableaux permettent de regrouper des éléments d'un même type en une seule structure. Un tableau peut avoir une ou plusieurs dimensions.

La représentation en mémoire d'un tableau est un suite séquentielle d'éléments. En particulier, une chaîne de caractères peut être manipulée avec des pointeurs ou comme un tableau.

3.4.1 Déclaration d'un tableau

La déclaration d'un *tableau* à une dimension s'écrit :
type_éléments nom_tableau`[`*nombre_éléments*`] ;`
La déclaration d'un tableau à deux dimensions s'écrit :
type_éléments nom_tableau`[`*nombre_éléments1*`][`*nombre_éléments2*`] ;`
Pour utiliser des tableaux de plus grande dimension, on utilise une déclaration similaire avec autant de *nombre_éléments* que nécessaire.

Exemple 3.17 :

```
char tab_carac[20];
```

déclare un tableau `tab_carac` de 20 caractères. Cette déclaration est équivalent à celle d'une chaîne de caractères de même taille :

```
char *tab_carac;

tab_carac = (char *) malloc(20);
```

La principale différence entre ces deux déclarations réside dans le mode d'allocation mémoire utilisé. En effet, lors de la déclaration d'un tableau, on effectue une *allocation statique* car le nombre d'éléments est fixé lors de l'écriture du programme et donc connu au moment de la compilation, tandis qu'en utilisant des pointeurs, l'allocation est *dynamique*. Cette dernière est faite par l'instruction `malloc`, lors de l'exécution du programme. La taille à allouer peut alors dépendre de valeurs de variables, du résultat d'un calcul, ...

```
int matrice[5][8];
```

déclare un tableau `matrice` à deux dimensions, par exemple pour stocker une matrice. Le nombre d'éléments pour chaque dimension est précisé : il s'agit d'une matrice à 5 lignes et 8 colonnes. ◇

3.4.2 Accès aux éléments d'un tableau

Les éléments d'un tableau peuvent être référencés soit par un indice dans le tableau, soit par un pointeur.

Accès par indice

Un tableau peut être considéré comme un ensemble de cases dans lesquelles les éléments sont rangés. L'accès à ces éléments se fait directement en utilisant l'indice de la case dans le tableau. *Le premier indice est 0.*

Exemple 3.18 :

```
int tableau[4], i;

for (i=0; i<4; i++)
    tableau[i] = 2*i;
```

déclare un tableau de quatre entiers et une variable entière `i`. La boucle `for` initialise les quatre éléments du tableau au double de l'indice où ils se trouvent. Ce programme a le même effet que celui de l'exemple 3.6. ◇

Accès par pointeur

Les éléments d'un tableau étant rangés de manière séquentielle en mémoire, la variable représentant le tableau peut être considérée comme un pointeur sur le premier élément. Par conséquent, les opérations de manipulations de pointeurs s'appliquent également aux tableaux.

Exemple 3.19 :

```
int tableau[4], i;

for (i=0; i<4; i++)
    *(tableau+i) = 2*i;
```

Ce programme est similaire à celui des exemples 3.6 (avec une allocation statique au lieu d'une allocation dynamique) et 3.18. ◇

3.5 Les arguments de `main`

Le passage de paramètres à la fonction `main` permet d'utiliser le programme avec des arguments lors de l'appel. Ceci ressemble au passage de paramètres à un *script shell*. La déclaration de la fonction `main` a alors la forme :

```
int main(int argc, char **argv){
...
}
```

où les variables `argc` et `argv` sont transmises au programme en fonction des arguments fournis par l'utilisateur. L'entier `argc` (*argument count*) contient le nombre d'arguments fournis plus un (qui correspond au nom du programme appelé), et `argv` (*argument value*) est un tableau de chaînes de caractères contenant les arguments. On note que le principe retenu est similaire à celui du *shell* avec les variables `$0`, `$1`, ...

Exemple 3.20 : Soit le programme suivant contenu dans un fichier `monprog.c` :

```
#include <stdio.h>

int main(int argc, char **argv){
    int i;

    printf("argc = %d\n",argc);
    printf("Programme appelé = argv[0] = %s\n",argv[0]);
    for (i=1; i<argc; i++)
        printf("Argument %d = argv[i] = %s\n",i,argv[i]);
}
```

Compilons et exécutons ce programme en lui passant 3 arguments, puis analysons l'affichage :

```
$ gcc monprog.c -o monprog
$ ./monprog param1 param2 param3
argc = 4
Programme appelé = argv[0] = ./monprog
Argument 1 = argv[i] = param1
Argument 2 = argv[i] = param2
Argument 3 = argv[i] = param3
```

Le compteur `argc` contient la valeur 4 : le programme et les trois arguments. Le premier élément du tableau `argv` est la référence du programme exécuté, alors que les trois éléments suivants sont les arguments passés par l'utilisateur. ◇

Chapitre 4

Structures

4.1 Concept de structure

Les *structures* permettent de regrouper des informations concernant un même objet. Une seule variable, concernant l'objet est alors nécessaire. De plus, cela fait ressortir la logique de regroupement entre les différents éléments de la variable.

Exemple 4.1 : Une date est composée de trois éléments : le jour, le mois et l'année. Plutôt que d'utiliser trois variables *a priori* indépendantes, on préférera définir une structure `date` regroupant un *entier* `jour`, une *chaîne de caractères* `mois` et un *entier* `année`.

De même, pour collecter des informations sur une personne, on est amené à définir une structure `personne` comprenant une *chaîne de caractères* contenant son `nom`, une autre *chaîne de caractères* pour son `prenom` et une *structure date* permettant de stocker sa date de naissance, `date_naissance`. ◇

4.2 Définition d'une structure

En `C`, la définition d'une structure utilise le mot-clé `struct`. Même si l'on peut déclarer directement une variable avec un type structure, il est souvent nécessaire d'utiliser cette structure plusieurs fois au sein d'un même programme. Par conséquent, il est préférable de déclarer un nouveau type, avec `typedef`, identifiant une structure particulière. La forme générale de la déclaration est alors :

```
typedef struct nom_structure{
    type_1 champ_1;
    type_2 champ_2;
    ...
} nom_type_structure;
```

On constate que deux noms sont précisés dans cette déclaration :
- `nom_structure` permet d'identifier la structure. Il doit être utilisé avec le mot-clé `struct` ;
- `nom_type_structure` est le type de structure défini par `typedef`. Il constitue un raccourci pour `struct nom_structure`.

Les différents éléments composant une structure sont appelés des *champs*.

Exemple 4.2 : Déclarons les types des structures présentées dans l'exemple 4.1. Pour cela, nous définissons une structure de nom `DATE`, associée au nouveau type `date` et une structure de nom `PERSONNE` associée au nouveau type `personne`. On note que la structure `PERSONNE` comporte un champ de type `date`.

```
typedef struct DATE{
    int     jour;
    char    mois[10];
    int     annee;
} date;

typedef struct PERSONNE{
    char    nom[30];
    char    prenom[30];
    date    date_naissance;
} personne;
```
◇

De manière générale, l'instruction `typedef` permet de définir un nouveau type, et pas seulement une structure.

4.3 Utilisation des structures

Une fois une structure définie, elle peut être utilisée pour typer des variables (voir section 4.3) et pour accéder à ses champs (voir section 4.3.2).

4.3.1 Déclaration de variables

La déclaration d'une variable de type structure est similaire à celle d'une variable d'un type de base. Des pointeurs sur des structures peuvent également être déclarés.

Exemple 4.3 : Les lignes suivantes déclarent une variable `quelquun` de type `personne` et un pointeur `quelquunptr` sur une structure de type `personne`.

```
personne quelquun;
personne *quelquunptr;
```
◇

4.3.2 Accès aux champs d'une structure

Il est nécessaire de pouvoir *accéder aux champs d'une structure* pour leur attribuer une valeur ou tester leur valeur. On accède à un champ `nom_champ` d'une structure `nom_structure` par :

```
nom_structure.nom_champ
```

Lorsque l'on manipule un pointeur sur une structure, `nom_structure_ptr`, il est également possible d'accéder à un champ `nom_champ` à l'aide de l'une des deux méthodes d'adressage suivantes (la seconde est un raccourci équivalent à la première) :

```
(*nom_structure_ptr).nom_champ
nom_structureptr->nom_champ
```

Exemple 4.4 : Le programme suivant complète les déclarations de structures et de variables des exemples 4.2 et 4.3 :

```
1 #include <stdio.h>
2 #include <string.h>
3
```

```
4  typedef struct DATE{
5      int      jour;
6      char     mois[10];
7      int      annee;
8  } date;
9
10 typedef struct PERSONNE{
11     char     nom[30];
12     char     prenom[30];
13     date     date_naissance;
14 } personne;
15
16 int main(){
17     personne quelquun;
18     personne *quelquunptr;
19
20     strcpy(quelquun.nom,"Dupont");
21     strcpy(quelquun.prenom,"Jean");
22     quelquun.date_naissance.jour = 12;
23     strcpy(quelquun.date_naissance.mois,"janvier");
24     quelquun.date_naissance.annee = 1995;
25     quelquunptr = &quelquun;
26     printf("%s %s est né le %d %s %d.\n",
27            quelquunptr->prenom,
28            quelquunptr->nom,
29            (quelquunptr->date_naissance).jour,
30            (quelquunptr->date_naissance).mois,
31            (quelquunptr->date_naissance).annee);
32 }
```

Tout d'abord, lignes 1–2, sont incluses les bibliothèques permettant d'utiliser les fonctions `printf` et `strcpy`. Les types définissant les structures `date` et `personne` sont définis aux lignes 4–8 et 10–14, respectivement.

Dans la fonction principale `main` sont d'abord déclarées deux variables, l'une étant une structure de type `personne` (ligne 17), et l'autre (ligne 18) un pointeur sur une structure de ce type. Les champs de la variable `quelquun` sont peuplés par des valeurs aux lignes 20–24. On remarque l'utilisation des structures imbriquées pour la date de naissance : `quelquun.date_naissance.annee` représente le champ `annee` du champ `date_naissance` de la variable `quuelquun`.

En ligne 25, on établit le lien entre les deux variables : `quelquunptr` pointe sur `quelquun`. L'affichage réalisé aux lignes 26–31 illustre l'utilisation combinée de pointeurs et de champs de structures.

Lorsque l'on exécute ce programme, on obtient l'affichage suivant :

```
Jean Dupont est né le 12 janvier 1995.
```

◇

4.4 Listes chaînées

L'utilisation de structures permet de construire des *listes chaînées* d'éléments d'un même type, par exemple pour représenter un ensemble. Pour cela, la structure comporte non seulement les champs détaillant ses différents composants, mais aussi un champ contenant un pointeur sur une structure identique, qui permet de référencer l'élément suivant dans la liste. Un pointeur sur le

premier élément, appelé *tête de liste*, permet de référencer le début de la liste. Le dernier élément de la liste n'ayant pas d'élément suivant, le champ référençant ce dernier a la valeur `NULL`.

Exemple 4.5 : Le programme suivant construit une liste de trois éléments de type `personne`, puis affiche leurs caractéristiques.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  /* définition des structures */
6  typedef struct DATE{
7      int      jour;
8      char     mois[10];
9      int      annee;
10 } date;
11
12 typedef struct PERSONNE{
13     char     nom[30];
14     char     prenom[30];
15     date     date_naissance;
16     struct PERSONNE *suivant;
17 } personne, *personneptr;
18
19 /* schémas des fonctions */
20 personneptr peuple_personne(char *nom,char *prenom,
21                     int jour,char *mois,int annee);
22 void affiche_personne(personneptr quelquunptr);
23 void liberer_liste(personneptr tete);
24
25 /*  remplissage d'une structure personne */
26 personneptr peuple_personne(char *nom,char *prenom,
27                     int jour,char *mois,int annee){
28     personneptr quelquunptr;
29
30     /* réservation de la place mémoire pour la structure */
31     quelquunptr = (personneptr) malloc(sizeof(personne));
32     /* copie du nom et du prénom */
33     strcpy(quelquunptr->nom,nom);
34     strcpy(quelquunptr->prenom,prenom);
35     /* remplissage de la date de naissance */
36     (quelquunptr->date_naissance).jour = jour;
37     strcpy((quelquunptr->date_naissance).mois,mois);
38     (quelquunptr->date_naissance).annee = annee;
39     /* pour l'instant on ne connait pas le suivant */
40     quelquunptr->suivant = NULL;
41     /* on renvoie la structure remplie */
42     return(quelquunptr);
43 }
44
45 /* affichage d'une structure personne */
46 void affiche_personne(personneptr quelquunptr){
47     printf("%s %s est né le %d %s %d.\n",
48             quelquunptr->prenom,
```

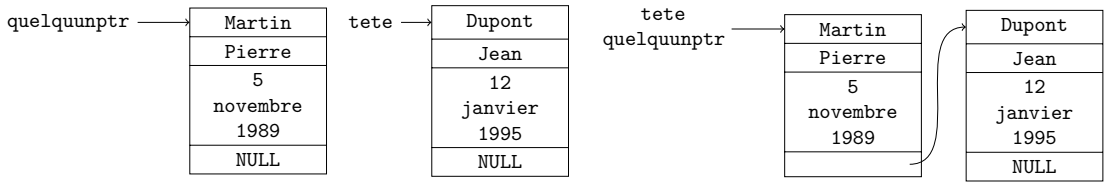
```
49              quelquunptr->nom,
50              (quelquunptr->date_naissance).jour,
51              (quelquunptr->date_naissance).mois,
52              (quelquunptr->date_naissance).annee);
53  }
54
55  /* libération de la mémoire utilisée par une liste de personnes */
56  void liberer_liste(personneptr tete){
57      personneptr quelquunptr;
58
59      while (tete != NULL){
60          quelquunptr = tete;
61          /* la tête sera bientôt l'élément suivant */
62          tete = tete->suivant;
63          /* on libère l'élément courant */
64          free(quelquunptr);
65      }
66  }
67
68  /* fonction principale */
69  int main(){
70      personneptr quelquunptr, tete;
71
72      /* création d'un premier élément de la liste */
73      tete = peuple_personne("Dupont","Jean",12,"janvier",1995);
74      /* création d'un second élément */
75      quelquunptr = peuple_personne("Martin","Pierre",5,"novembre",1989);
76      /* enchaînement des 2 éléments avec insertion en tête de liste */
77      quelquunptr->suivant = tete;
78      tete = quelquunptr;
79      /* création d'un troisième élément */
80      quelquunptr = peuple_personne("Dubois","Paul",22,"août",1991);
81      /* enchaînement avec insertion en tête de liste */
82      quelquunptr->suivant = tete;
83      tete = quelquunptr;
84      /* affichage des éléments de la liste, à partir de la tête */
85      for (quelquunptr = tete ;
86           quelquunptr != NULL ;
87           quelquunptr = quelquunptr->suivant)
88          affiche_personne(quelquunptr);
89      /* libération de la liste */
90      liberer_liste(tete);
91  }
```

Dans ce programme, on inclut également la bibliothèque `stdlib.h` qui permet d'utiliser la fonction `malloc`. La structure `personne` de l'exemple 4.4 est complétée à la ligne 16 en ajoutant un pointeur sur l'élément suivant dans la liste. Le nouveau type `personneptr` permet d'utiliser directement un pointeur sur une structure `personne`.

Décrivons le fonctionnement de ce programme. Deux pointeurs sur des structures de type `personne` sont utilisés : `tete` qui représente la tête de liste, et `quelquunptr` qui permet de référencer une structure en cours de traitement. La fonction `peuple_personne` crée une structure puis peuple ses champs avec les données passées en paramètre. Elle reprend les mêmes principes de renseignement des champs que dans l'exemple 4.4. On remarque qu'il est nécessaire de réserver l'emplacement mémoire pour stocker la structure (ligne 31). De plus, lorsque l'on crée une nouvelle structure, on ne connait pas l'élément suivant dans la liste, par conséquent c'est le pointeur nul (ligne 40). La fonction revoie, ligne 42, un pointeur sur la structure allouée et remplie.

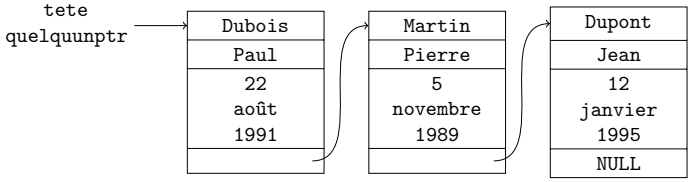
La fonction principale crée, lignes 73 et 75, deux structures de type `personne`, la première pointée par `tete` et la seconde par `quelquunptr`. La représentation de la mémoire est indiquée dans la figure 4.1(a). L'enchaînement entre les éléments de la liste est ensuite effectué en insérant la structure pointée par `quelquunptr` en tête de liste (lignes 77–78), correspondant au schéma de la figure 4.1(b). Les mêmes opérations sont effectuées pour insérer une troisième personne dans la liste (lignes 80–83). On obtient alors le schéma complet de la mémoire présenté dans la figure 4.1(c).

La boucle `for` aux lignes 85–88 affiche les éléments de la liste un par un, en commençant par l'élément de tête. Enfin, la fonction `liberer_liste` permet de libérer la mémoire occupée par les structures qui avaient été allouées ligne 31.



(a) Après exécution de la ligne 75

(b) Après exécution de la ligne 78



(c) La liste complête

FIG. 4.1 – Construction d'une liste chaînée

L'exécution de ce programme affiche :

```
Paul Dubois est né le 22 août 1991.
Pierre Martin est né le 5 novembre 1989.
Jean Dupont est né le 12 janvier 1995.
```

◇

Chapitre 5

Manipulation de fichiers

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

5.1 Ouverture et fermeture d'un fichier

5.1.1 Descripteur de fichier

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

5.1.2 Fichiers particuliers

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

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

5.1.3 Ouverture d'un fichier

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

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

Les paramètres de la fonction sont le nom du fichier sur disque, `nom_fichier`, avec éventuellement son chemin d'accès, ainsi qu'un `mode` d'ouverture. Les différents modes sont indiqués dans le tableau 5.1.

Tab. 5.1 – Modes d'ouverture d'un fichier

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

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

5.1.4 Fermeture d'un fichier

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

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

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

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

Exemple 5.1 : Le programme suivant ouvre un fichier de nom `toto` en lecture, teste le résultat de la fonction d'ouverture du fichier et affiche le message idoine, puis ferme le fichier.

```
#include <stdio.h>

int main(){
    FILE *fich;

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

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

5.2 Lecture/écriture binaire

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

5.2.1 Écriture binaire

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

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

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

Exemple 5.2 : Le programme suivant ouvre un fichier `toto` en écriture et y écrit 5 caractères contenus dans la chaîne de caractères `Bonjour`.

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

int main(){
    FILE *fich;
```

```
    int nb;
    char buf[20];

    fich = fopen("toto","w");
    if (fich == NULL){
        printf("Erreur de fopen\n");
        return(1);
    }
    /* L'ouverture s'est bien passée */
    /* mise en place dans buf de la chaîne à écrire */
    strcpy(buf,"Bonjour");
    /* écriture de 5 caractères de buf */
    nb = fwrite(buf,sizeof(char),5,fich);
    printf("Nb de caractères écrits : %d\n",nb);
    fclose(fich);
}
```

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

```
Nb de caractères écrits : 5
```

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

```
Bonjo
```

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

5.2.2 Lecture binaire

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

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

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

Exemple 5.3 : Le programme suivant ouvre un fichier `toto` en lecture et y écrit au plus 10 caractères qu'elle place dans une zone pointée par `buf`.

```
#include <stdio.h>

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

    fich = fopen("toto","r");
    if (fich == NULL){
        printf("Erreur de fopen\n");
        return(1);
    }
    /* L'ouverture s'est bien passée */
    /* lecture de 10 caractères dans buf */
    nb = fread(buf,sizeof(char),10,fich);
    printf("Nb de caractères lus : %d\n",nb);
    printf("Chaîne lue : %s\n",buf);
```

```
    fclose(fich);
}
```

Si le fichier de nom `toto` est celui écrit par l'exemple 5.2, l'exécution de ce programme affiche :

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

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

5.2.3 Fin de fichier

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

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

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

Exemple 5.4 : Le programme suivant est similaire à celui de l'exemple 5.3, mais lit les caractères deux par deux.

```
#include <stdio.h>

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

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

Si le fichier de nom `toto` est celui écrit par l'exemple 5.2, l'exécution de ce programme affiche :

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

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

5.3 Lecture/écriture d'une chaîne de caractères

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

5.3.1 Lecture d'une chaîne de caractères

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

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

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

5.3.2 Écriture d'une chaîne de caractères

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

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

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

Exemple 5.5 : Le programme suivant affiche la chaîne de caractères `Coucou`.

```
#include <stdio.h>

int main(){
    fputs("Coucou\n",stdout);
}
```

◇

5.4 Lecture/écriture formatée

Les lectures et écritures formatées travaillent selon un format identique à celui utilisé par `printf` (voir section 2.3.2).

5.4.1 Écriture formatée

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

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

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

5.4.2 Lecture formatée

L'écriture formatée est similaire :

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

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

Exemple 5.6 : Le programme suivant demande à l'utilisateur d'entrer un entier et une chaîne de caractères, puis les affiche.

```
#include <stdio.h>

int main(){
    int i;
    char buf[20];

    printf("Entrer un entier puis un mot : ");
    scanf("%d %s",&i,buf);
    printf("Vous avez entré : %d et %s\n",i,buf);
}
```

◇

Index