



**UNIVERSITÉ  
DE LORRAINE**



nancy

**Charlemagne**

Département Informatique

# **Architecture et programmation des mécanismes de base d'un système informatique**

**DUT Informatique**

**Auteurs : Ph. Dosch, B. Mangeol**

**Date : 2017/2018**

UNIVERSITÉ DE LORRAINE  
INSTITUT UNIVERSITAIRE DE TECHNOLOGIE  
2 ter boulevard Charlemagne  
CS 5227  
54052 • NANCY cedex

Tél : 03.83.91.31.31

Fax : 03.83.28.13.33

<http://iut-charlemagne.univ-nancy2.fr/>



# Sommaire

<b>1</b>	<b>Introduction</b>	<b>5</b>
1	Généralités	5
2	Historique et différentes normes	5
3	Avantages	6
4	Inconvénients	7
5	Conventions	7
6	Ressources	8
<b>2</b>	<b>Du Java au C</b>	<b>9</b>
1	Premier programme	9
2	Principales différences entre Java et C	10
3	Types de base	12
4	Définitions, déclarations	12
1	Les variables	12
2	Les types de données	12
3	Les fonctions	13
5	Portée	14
6	Tableaux	14
7	Chaînes de caractères	16
8	Fonctions	17
1	Définition, déclaration	17
2	Appels	18
3	Cas des tableaux	19
4	Cas des chaînes de caractères	20
9	Adresses mémoire	21
10	printf() et scanf()	22
1	L'affichage : printf()	22
2	La saisie : scanf()	23
11	Synopsis de la construction d'un programme C	26
1	Introduction	26
2	Le préprocesseur	27
3	Compilation, fichiers objets	27
4	Édition de liens	28
12	Construction et ligne de commande	28
1	Généralités	28
2	Options fréquentes	29
3	Options relatives au respect des normes	29
4	Recommandations	29
13	Un débogueur : gdb	30
14	Un profiler : valgrind	32
15	Un exemple	32

<b>3</b>	<b>Modularité et bibliothèques</b>	<b>37</b>
1	Modularité et abstraction	37
2	Bibliothèque	37
1	Fichier d'en-tête d'une bibliothèque	37
2	Implantation des bibliothèques	39
3	Directives du préprocesseur	39
1	#define	39
2	#include	40
3	Compilation conditionnelle	41
4	Exemple de compilation conditionnelle	41
<b>4</b>	<b>Make</b>	<b>47</b>
1	Introduction	47
2	Historique de GNU Make	47
3	Les deux étapes de la création d'un programme	48
4	Exemple de construction de Makefile	48
5	Définition des Makefile	52
<b>5</b>	<b>Les pointeurs</b>	<b>53</b>
1	Une explication (*, & et NULL)	53
2	Arithmétique des pointeurs	54
3	Les pointeurs et les tableaux	55
4	Le passage par adresse	57
1	Principe	57
2	Fonctions renvoyant plusieurs résultats	59
5	Gestion dynamique de la mémoire	61
1	Allocation : la fonction malloc, l'opérateur sizeof	61
2	Allocation dynamique de structures	63
3	Libération : la fonction free	64
6	Les périls de la programmation avec pointeurs	65
7	valgrind	65
<b>6</b>	<b>Représentation des listes par une liste chaînée</b>	<b>71</b>
1	Type	71
2	Exemple d'application : une bibliothèque Liste	72
<b>7</b>	<b>Les signaux</b>	<b>75</b>
1	Introduction	75
2	Liste des signaux d'un système	75
3	Traitement des signaux	76
1	La fonction signal()	76
2	La fonction sigaction()	78
4	La fonction pause()	80
5	Émission d'un signal	80
1	La fonction kill()	80
2	La fonction raise()	80
6	Alarmes	81
	<b>Index</b>	<b>83</b>

# Introduction

## SECTION

## 1

## Généralités

Le langage C est un langage de programmation procédurale dont la première version date de 1972. C'est un langage compilé permettant la génération de programmes exécutables contenant des instructions assembleur spécifiques à une architecture donnée <sup>❶</sup> (sur ce point, c'est une approche radicalement différente de celle suivie par Java, générant du *byte-code* exploitable — portable — sur toute architecture). Toutefois, le langage C est normalisé, ce qui autorise la compilation (en majeure partie) d'un programme C donné sur des architectures différentes.

De nombreux langages créés par la suite se sont inspirés du langage C. Tout d'abord le C++, créé au début des années 80, qui est le premier langage à objets notoire étendant le langage C. En effet, le C++, dans ses spécifications, présente la particularité de rester assez proche du C dans le sens où « *tout programme C valide doit être un programme C++ valide* » <sup>❷</sup>. Si l'apport de la couche objet dans C++ est un plus indéniable, la contrainte énoncée ci-dessus en a également fait un langage permissif vis-à-vis du paradigme objet, en limitant ainsi parfois l'intérêt...

Partant sur des bases conceptuelles radicalement différentes, Sun Microsystems a publié le langage Java en 1995. Ce langage a également une syntaxe très largement inspirée de celle de C, mais n'est plus contraint d'assurer une compatibilité ascendante à l'instar du C++. La couche objet est du coup beaucoup plus homogène et cohérente que dans C++. Il n'en reste pas moins que ces deux langages à objets directement inspirés de C présentent des spécificités (sur la vitesse, la portabilité, etc.) leur permettant d'être l'un et l'autre largement utilisés, dans des contextes différents.

Enfin, la syntaxe du langage C a également influencé de nombreux autres langages, tels que JavaScript, PHP, Perl...

## SECTION

## 2

## Historique et différentes normes

Le langage C est apparu au cours de l'année 1972 dans les laboratoires Bell (téléphonie au USA). Il a été développé en même temps que le système Unix (et pour la création de ce dernier) par Dennis Ritchie et Ken Thompson. Ken Thompson avait développé un prédécesseur de C, le langage B. Dennis Ritchie a

<sup>❶</sup> On qualifie ici d'*architecture* un couple processeur / système d'exploitation.

<sup>❷</sup> Ceci pour faciliter le portage des logiciels écrits de C en C++...

fait évoluer le langage B dans une nouvelle version suffisamment différente pour qu'elle soit appelée C. En 1983, l'ANSI (institut de normalisation américain) a formé un comité de normalisation du langage qui a abouti en 1989 à la norme dite ANSI C ou C89, également adoptée par l'ISO en 1990.

En 1999, une nouvelle évolution du langage a été normalisée par l'ISO : la « C99 ». Voici, à des fins d'illustration, quelques-unes des nouveautés apportées par cette norme :

- tableau à longueur variable,
- support des nombres complexes grâce à `complex.h`,
- types `long long int` et `unsigned long long int` d'au moins 64 bits,
- famille de fonctions `vscanf()`,
- les fameux commentaires à la C++ `//`,
- les familles de fonctions `snprintf()`,
- un type « booléen »...

Enfin, le langage reste actif puisqu'une norme supplémentaire est parue en date du 8 décembre 2011 (soit quelque temps après la mort de Dennis Ritchie, qui s'est éteint le 12 octobre 2011). Cette nouvelle norme, estampillée « C11 », intègre quelques nouveautés, dont :

- la gestion du multithreading : intégration de nouveaux mots-clés et d'une API pour créer et gérer les threads / mutex / conditions,
- la gestion des chaînes de caractères unicode, par l'introduction de nouveaux types et préfixes pour la définition de chaînes de caractères constantes,
- le remplacement de certaines fonctions à l'origine de *buffer overflows* (`get_s` remplace ainsi `gets`),
- ainsi que la possibilité de ne pas supporter certaines des fonctionnalités introduites par la norme C99, qui avaient représenté un frein à son adoption (l'objectif est donc de faciliter la conformité des compilateurs vis-à-vis de la norme).

La norme C99 a eu en effet beaucoup de mal à s'imposer auprès de certains éditeurs de compilateurs. Microsoft, en particulier, a même refusé de l'implémenter... Difficile dans ces conditions de savoir si la toute dernière norme a des chances de « percer » dans les années à venir, autant au niveau de sa disponibilité dans les compilateurs que dans les usages des développeurs.

Dans l'état actuel des choses, tout ce dont on peut être sûr, c'est que n'importe quel bon compilateur supporte au moins la norme C90. C'est cette norme qui est traitée en priorité dans le cadre de ce cours.

#### SECTION

## 3

## Avantages

Le langage C reste un des langages les plus utilisés car :

- Il existe depuis le début des années 1970, il est basé sur un standard ouvert, de nombreux informaticiens le connaissent et des compilateurs et bibliothèques logicielles existent sur la plupart des architectures.
- Sa syntaxe est claire et efficace. Elle a d'ailleurs influencé de nombreux autres langages dont C++, Java, JavaScript, Perl et PHP...
- Il met en œuvre un nombre restreint de concepts, ce qui facilite sa maîtrise et l'écriture de compilateurs simples et rapides.
- Il permet l'écriture de logiciels qui n'ont besoin d'aucun support à l'exécution (ni bibliothèque logicielle ni machine virtuelle), au comportement prédictible en temps d'exécution comme en consommation de mémoire vive, comme des noyaux de systèmes d'exploitation et des logiciels embarqués.

## Inconvénients

Il présente également certains inconvénients, dont :

- Par rapport à d'autres langages, le C fait particulièrement peu de vérifications lors de la compilation, et il n'offre aucune vérification pendant l'exécution, ce qui fait que des erreurs qui pourraient être automatiquement détectées lors du développement ne le sont que plus tard, souvent au prix d'un plantage du logiciel. Cet inconvénient est également un avantage lors de l'écriture de système d'exploitation ou de pilotes, car les développeurs ont un accès très fin aux ressources utilisées et très peu de contraintes...
- Il n'offre pas de support direct à des concepts informatiques plus modernes comme la programmation orientée objet ou la gestion d'exceptions. D'autres langages, qui lui ont succédé, ont permis par la suite de combler ces manques.
- Il est parfois difficile d'écrire des programmes portables car le comportement exact des exécutables dépend de l'ordinateur cible. Les différentes normalisations ont permis de combler un peu ces problèmes de portabilité, mais certains aspects inhérents au langage rendent impossible une portabilité complète.
- Le support de l'allocation de mémoire et des chaînes de caractères est minimaliste, ce qui oblige les programmeurs à s'occuper de ces détails fastidieux. Cet aspect est souvent sources de bogues (comme vous pourrez vous en rendre compte).
- D'autres limitations inhérentes au langage sont sources de bogues, comme le débordement de tampons (*buffers*). Ces limitations constituent des failles de sécurité informatique et peuvent être exploitées par des logiciels malveillants comme des vers informatiques.

## Conventions

Nous utiliserons dans ce cours quelques conventions dont le but est de rendre plus lisible un programme :

- *Constantes* : les constantes sont écrites en majuscules : MAX, LONG\_CHAINE
- *Variables* : les variables sont écrites en minuscules : i, j, taille
- *Pointeurs* : les pointeurs peuvent commencer par le préfixe ptr\_ comme dans ptr\_i, ptr\_taille. Par exemple ptr\_i ou ptr\_taille désignent des pointeurs sur i et taille respectivement.
- *Types* : les types définis commencent par une majuscule : Personne, Liste, Pile.
- *Fonctions* : les noms de fonctions ont généralement comme base des verbes (à l'infinitif, au présent, via un substantif associé...) et commencent toujours par une minuscule : afficher, empile, addition. Les noms composés se construisent grâce à « \_ », comme pour personne\_afficher par exemple.
- *Bibliothèques* : les identificateurs d'une bibliothèque, déclarés dans le fichier d'en tête, doivent comporter, en préfixe ou en suffixe, le nom de la bibliothèque.

Par ailleurs, les programmes en C ou les extraits de programmes sont en police non proportionnelle. Les lignes de commandes, à saisir dans une console Unix, le sont aussi, mais **entourées dans une boîte**.

## Ressources

Il existe de nombreux livres relatifs au langage C. L'ouvrage de référence reste celui des auteurs du langage, à savoir « *Kernighan B.W. & Ritchie D.M. (1997), Le langage C Norme ANSI, 2ième édition, Dunod, Paris.* ». Cependant, compte tenu des évolutions du langage, il peut être complété par d'autres ouvrages plus récents et plus en phase avec ses dernières caractéristiques. Par ailleurs, de nombreuses documentations en ligne sont également disponibles. Outre celles que vous trouverez par vous-mêmes, vous pouvez consulter :

- Le C en 20 heures, chez Framabook  
<http://www.framabook.org/c20h.html>
- Un site simple et qui reprend vraiment tout de zéro comme son nom l'indique :  
<http://www.siteduzero.com/tutoriel-3-14189-apprenez-a-programmer-en-c.html> (notamment si ce polycopié vous semble trop compliqué)
- <http://picolibre.int-evry.fr/projects/cvs/coursc/>
- <ftp://ftp.laas.fr/pub/ii/matthieu/c-superflu/c-superflu.pdf>



# Du Java au C

Le but de ce chapitre est de présenter très brièvement les caractéristiques du langage C. Les aspects présentés ne prétendent donc pas à l'exhaustivité et nous nous attachons essentiellement à présenter les caractéristiques les plus remarquables du C **pour un programmeur ayant déjà de bonnes bases de Java**. Certaines caractéristiques de ces deux langages étant très proches, voire parfois totalement similaires (Java étant inspiré de C), nous nous efforcerons de faire ressortir les différences.

## SECTION

## 1

## Premier programme

Le premier programme que tout un chacun écrit lorsqu'il apprend un nouveau langage répond à la question suivante : comment écrire un programme principal qui imprime « Bonjour » ?

Voici ce programme écrit en Java :

```
1 class Bonjour {
2     public static void main(String args[]) {
3         System.out.println("Bonjour !");
4     }
5 }
```

Voici ce même programme écrit en C :

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Bonjour !\n");
6     return 0;
7 }
```

Il est constitué de la manière suivante :

- La ligne n° 1 est une directive pour le préprocesseur, lui demandant d'inclure un fichier de déclarations de fonctions (représentant l'interface standard des entrées-sorties). Cette ligne doit être présente dans tout module faisant appel à des fonctions d'entrées sorties et en particulier aux fonctions `printf()` et `scanf()`.
- Il n'y a pas de variables.

- Le programme contient une seule fonction `main()` qui joue le rôle de programme principal. Cette fonction est utilisée sans argument.
- La définition de la fonction `main()` commence comme pour toute fonction par une accolade ouvrante.
- L’instruction principale (ligne n° 5) correspond à un appel de fonction qui provoque l’écriture sur le terminal. Cette instruction est une demande d’écriture d’une chaîne de caractères.
- L’instruction `return` (ligne n° 6) permet à la fonction principale `main()`, et ainsi au programme en lui-même, de renvoyer un code indiquant la façon dont l’exécution s’est déroulée (avec ou sans erreur).
- La fonction `main()` est terminée par une accolade fermante.

On sauvegardera ce programme sous le nom `bonjour.c`. Pour compiler le programme, on tape la commande suivante : `gcc bonjour.c -o bonjour` <sup>①</sup>. Si la compilation s’est déroulée correctement, on doit obtenir le fichier `bonjour`. Pour exécuter le programme, il suffit de taper : `./bonjour`.

## SECTION

# 2

## Principales différences entre Java et C

Le tableau (Tab. 2.1) résume les différences principales entre Java et C. Le langage C n’étant pas un langage objet, il n’est pas possible d’y représenter des concepts tels que les classes existant en Java par exemple. En revanche, il est tout de même possible de regrouper des données au sein de *structures*, qui sont des ensembles de données non homogènes. Ces données peuvent donc avoir des types différents. Les structures sont définies selon le modèle :

```
struct nom_facultatif {
    type_1 liste_d'attributs_1;
    type_2 liste_d'attributs_2;
    ...
    type_n liste_d'attributs_n;
} liste_de_variables;
```

Prenons pour exemple la définition de structure en C suivante :

```
1 struct st1 {
2     int a1;
3     float b1, b2;
4     long c1;
5 } objst1;
```

Dans cet exemple : `st1` est un nom de modèle de structure et `objst1` est une variable de type struct `st1` <sup>②</sup>. Les différents attributs de la structure `objst1` sont accessibles par `objst1.a1`, `objst1.b1` et `objst1.c1`.

**Exercice 1** Répondre aux questions suivantes :

1. Afficher 50 fois « Hello world ».
2. Écrire la table de multiplication du 8, sachant que `printf("%d\n", i);` permet d’afficher le contenu de la variable entière `i`.

<sup>①</sup> Sous Unix, le fait qu’un fichier soit identifié comme un programme exécutable n’est pas lié à l’extension (`.exe` par exemple) de ce fichier, mais à certains attributs associés à ce fichier. Lors d’une compilation, ces attributs sont automatiquement positionnés.

<sup>②</sup> « struct st1 » est le nom complet du type introduit. Il n’est pas possible d’y faire référence juste par « st1 ».

	Java	C
1.	Classes, objets	—
2.	Méthodes	fonctions
3.	Attributs	types composites (structures)
4.	Types fondamentaux : int, double, boolean	(unsigned) int, int, float, double
5.	Déclaration des variables : n'importe où	au début de chaque fonction (ANSI C)
6.	Opérations mathématiques : +, -, /, *	les mêmes
7.	Opérations logiques : &&,   , ==, !=	les mêmes
8.	opérations de comparaison : >, <, ==, !=, <=, >=	les mêmes
9.	Sortie standard : System.out.println ()	printf()
10.	Lecture des données : Scanner.nextType() (par exemple, Scanner.nextInt())	scanf()
11.	Conditionnelles : if (condition) ... else ... condition : un booléen	les mêmes condition : un entier
12.	Boucles : for () ... while (condition) ... condition : un booléen	le même le même condition : un entier
13.	Programme principal : class Test { public static void main(String [] args)	int main()
14.	Tableau (indexation commence par 0) : classes d'objets type [] nom_tableau; nom_tableau = new type[taille]; nom_tableau[0]=valeur; (il y a tab.length)	pointeurs type nom_tableau[taille]; — nom_tableau[0]=valeur; (il n'y a pas de tab.length)
15.	Type pointeur : implicite	explicite
16.	Pas de préprocesseur (pas de lignes avec #define )	Présence d'un préprocesseur (les lignes commençant par #)
17.	La mémoire est gérée par l'interpréteur Java (« garbage collector »)	malloc, free

**Table 2.1** : Principales différences entre le langage Java et le langage C.

## Types de base

Il existe 5 types prédéfinis en C :

- `int` : type entier, pouvant être décliné avec des qualificatifs permettant de préciser la taille (long ou short) ou le domaine des entiers (`unsigned` pour les positifs, *rien* ou `signed` pour les entiers naturels),
- `float` : pour les réels,
- `double` : pour les réels avec une précision et un domaine de représentation plus grand,
- `char` : pour les caractères,
- `void` : type vide. Il est utilisé pour les fonctions ne renvoyant aucune valeur (procédure) et joue un rôle particulier dans l'utilisation des pointeurs (page 53).

À noter : le langage C ne dispose ni d'un type représentant les booléens (généralement représentés par des entiers), ni d'un type représentant les chaînes de caractères. Ces dernières sont conventionnellement représentées sous forme de tableaux de caractères (page 16).

Contrairement à Java, la taille en octets associée à chacun de ces types peut varier d'une machine à une autre. L'opérateur `sizeof()`, applicable sur une variable ou un type de données, retourne dynamiquement la taille en octets occupée par son paramètre.

## Définitions, déclarations

Comme dans beaucoup de langages informatiques, il est nécessaire de *définir* en C certaines choses avant de pouvoir les utiliser. C'est en particulier vrai pour les variables, les types de données, les fonctions...

### 1 Les variables

Elles se définissent comme en Java, en précisant d'abord le type de données et ensuite une liste de variables à créer. Les variables peuvent être initialisées au moment de leur définition. Elles contiennent sinon une **valeur indéfinie**<sup>③</sup>. Quelques exemples de définition :

- `int a;`
- `float x, y;`
- `char c = 'A';`

### 2 Les types de données

Il est possible de définir de nouveaux types de données en C à partir des types existants. Ces définitions se font grâce au mot-clé `typedef`, selon la syntaxe : `typedef type_existant nouveau_type`. On peut intuitivement assimiler le fonctionnement de `typedef` à celui de la définition d'alias. Quelques exemples :

- `typedef int Couleur;` : définit un nouveau type de données appelé `Couleur` correspondant exactement à un entier.
- `typedef float Note;` : définit un nouveau type de données appelé `Note` correspondant exactement à un réel.

<sup>③</sup> Il n'y a donc aucune initialisation effectuée. La valeur initiale d'une variable correspond à la valeur résiduelle se trouvant dans les cases mémoire réservées pour cette variable au moment de sa définition.

- Il est également intéressant d'utiliser `typedef` lors de la définition de structures, comme c'est le cas pour les deux exemples ci-dessous. À noter : dans le deuxième exemple, le nom de la structure est omis étant donné qu'il y a une définition de type en même temps que la définition de la structure. Les futures variables créées à partir de cette structure / de ce type le seront par l'intermédiaire de définitions du genre : `Compte c1, c2;`.

```
1 typedef struct point {
2     int x, y;
3 } Point;
4
5 typedef struct {
6     int numero_compte;
7     float solde;
8 } Compte;
```

### 3 Les fonctions

Pour pouvoir utiliser une fonction dans un module C (dans un fichier source à extension `.c`), il est nécessaire que cette fonction ait été *définie* ou *déclarée* au préalable dans ce même module. La *définition* d'une fonction consiste à fournir, comme dans la plupart des autres langages :

- son nom,
- son type de retour,
- la liste des paramètres attendus,
- le corps de la fonction, c'est-à-dire les instructions que cette fonction exécutera lorsqu'elle sera appelée.

Cependant, il n'est pas toujours possible ou souhaitable de définir toutes les fonctions d'un programme donné à l'intérieur d'un unique fichier. Par ailleurs, il n'est pas possible de définir plusieurs fois une même fonction au sein d'un programme donné. Comment faire alors pour utiliser une fonction donnée dans des fichiers sources `.c` différents d'un même programme ?

Il faut dans ce cas simplement *déclarer* cette fonction dans les modules où elle est utilisée. La *déclaration* d'une fonction est assez similaire à sa définition, mais le corps de la fonction n'est pas fourni et il est remplacé par un « ; ». On dit alors qu'on fournit l'*en-tête* de la fonction, ou encore son *prototype* ou sa *signature* (ces termes sont équivalents dans ce contexte). Les en-têtes standards, comme ceux vu dans l'exemple présenté (page 9), regroupe des *définitions* de types et des *déclarations* de fonctions standards. Après inclusion par l'intermédiaire d'une directive `#include`, ils permettent ainsi à un fichier source `.c` de disposer de la déclaration (à défaut de la définition) des fonctions qu'ils contiennent. L'exemple ci-dessous présente la définition d'une fonction, ainsi que sa déclaration juste en dessous.

```
1 int addition(int a, int b)
2 {
3     int res;
4
5     res = a + b;
6     return res;
7 }
8
9 int addition(int a, int b);
```

Attention : en C, il est impossible de *définir* plusieurs fois une fonction donnée dans un programme, il est en revanche possible de la *déclarer* plusieurs fois (les prototypes doivent évidemment être les mêmes à chaque fois). D'autres caractéristiques des fonctions sont présentées (page 17).

**Exercice 2** Gestion d'étudiants (cet énoncé se continue dans les exercices suivants) :

1. Définir un type *Etudiant*, permettant de représenter son numéro (entier) et trois notes (réelles, choisir le type *float*).
2. Définir une fonction *moyenne()* renvoyant la moyenne des 3 notes d'un étudiant passé en paramètre (donc d'une variable de type *Etudiant*).
3. Faire un programme principal permettant de saisir un étudiant et d'en afficher la moyenne. On donne les précisions suivantes :
  - `scanf("%d", &i);` permet de lire un entier et de le ranger dans la variable *i*,
  - `scanf("%f", &r);` permet de lire un réel (type *float*) et de le ranger dans la variable *r*,
  - `printf("%f\n", r);` permet d'afficher le contenu d'un réel *r* (type *float*).

SECTION

5

## Portée

Un identificateur est le nom d'une variable, d'une constante, d'un type, d'un paramètre d'une fonction, ou encore le nom d'une fonction. La *portée* d'un identificateur est la partie du programme dans laquelle l'identificateur est connu et où nous pouvons donc y faire référence.

- La portée d'une constante commence à sa déclaration et se termine à la fin du fichier.
- La portée d'une fonction commence à la déclaration de son prototype et se termine à la fin du fichier.
- La portée d'une variable locale ou d'un paramètre formel commence à la déclaration de l'identificateur et s'étend jusqu'à l'accolade qui ferme la fonction.
- Si un type est déclaré à l'extérieur de toute fonction, alors sa portée commence à sa déclaration et s'étend jusqu'à la fin de fichier complet.
- Si le type est déclaré dans une fonction alors sa portée est restreinte de sa déclaration à l'accolade fermante de la fonction, comme pour les variables locales.

SECTION

6

## Tableaux

Les tableaux en C, comme en Java, représentent des collections de variables. La syntaxe de définition d'un tableau en C est assez proche de celle existant en Java :

- `int tab1[50];` : définition d'un tableau de 50 entiers,
- `float tab2[20];` : définition d'un tableau de 20 réels,
- `int tab3[10][50];` : définition d'un tableau à deux dimensions d'entiers.

Si la syntaxe de définition d'un tableau en C est proche de celle de Java, il y a cependant une différence **fondamentale** entre les deux langages au niveau de la gestion de la mémoire. En effet, un tableau en Java est avant tout un objet. Ainsi, si on considère les instructions Java suivantes :

```
1 int tab[];  
2  
3 tab = new int[20];
```

- la variable `tab`, située dans la pile, est une référence vers le tableau,
- le tableau en lui-même, instancié par l'opérateur `new`, est situé dans le tas (Fig. 2.1).

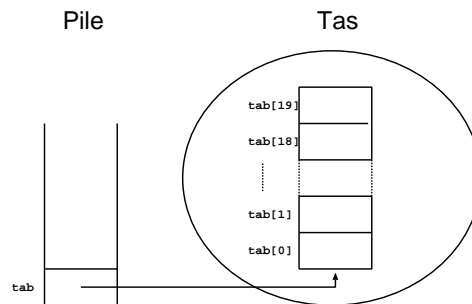


Figure 2.1 : Représentation mémoire d'un tableau en Java.

En C, une définition du type `int tab[20]` crée immédiatement un tableau de 20 entiers **dans la pile** (Fig. 2.2). Ces tableaux peuvent être qualifiés de *tableaux statiques*<sup>4</sup>. Le fait que ces tableaux ne soient pas des objets en C a du coup plusieurs incidences :

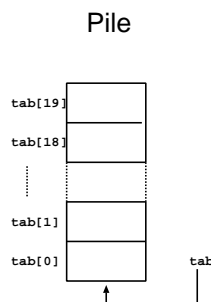


Figure 2.2 : Représentation mémoire d'un tableau en C.

- *Aucun tableau C (statique ou dynamique) ne contient sa longueur en interne*, comme cela peut l'être en Java au moyen de l'attribut `length`. Le programmeur doit donc avoir conscience de la taille des tableaux qu'il manipule et définir, au besoin, une variable associée à chaque tableau pour en mémoriser la taille.
- *C ne vérifie pas les indices utilisés lors de l'accès aux éléments d'un tableau (statique ou dynamique)*. Une instruction essayant d'accéder au 20<sup>e</sup> élément d'un tableau de 10 éléments ne provoquera donc pas d'exception. Dans le meilleur des cas, cette instruction invalide provoquera une erreur d'exécution si elle a modifié une partie critique du programme (ou essayé de modifier une case mémoire située en dehors de l'espace mémoire alloué au programme par le système d'exploitation). Dans le pire des cas, l'instruction ne provoquera aucune erreur d'exécution. Cette situation est généralement très embêtante, car le fait d'accéder à un emplacement mémoire non conforme aux prévisions du développeur va provoquer des dysfonctionnements. En l'absence d'erreur explicite de la part du programme, le développeur va généralement avoir beaucoup de mal à comprendre pourquoi son programme ne lui fournit pas les résultats escomptés...
- *Ces tableaux — statiques — étant représentés dans la pile et non pas dans le tas*, leur durée de vie correspond à la durée de vie de la variable correspondante. Dès que la variable devient inaccessible (hors portée), le tableau correspondant est automatiquement détruit. Pour mémoire, en Java, les objets (et ainsi les tableaux, puisqu'ils sont avant tout des objets) demeurent accessibles tant qu'il existe encore au moins une référence les adressant ; leur durée de vie n'est donc pas conditionnée par la *portée* (voir (page 14)) des variables qui ont permis leur création.

**Remarque :** comme on le voit (Fig. 2.2), les différents éléments du tableau sont accessibles *via* la syntaxe `tab[0]`, `tab[1]`, etc. Il est également possible d'utiliser `tab` tout seul : **l'identificateur d'un tableau utilisé seul référence le début du tableau en mémoire**. Cette syntaxe est utilisée très fréquemment, notamment pour la manipulation des chaînes de caractères (page 16), pour les passages de tableaux comme

<sup>4</sup> Par opposition aux *tableaux dynamiques* qui seront étudiés (page 61).

paramètres de fonctions (page 17), etc. Des explications supplémentaires sur cette syntaxe et sa signification technique sont données (page 55) dans le chapitre traitant les pointeurs.

**Exercice 3** Reprendre l'exercice précédent sur la gestion des étudiants. Modifier le programme principal pour qu'il :

1. demande le nombre d'étudiants à gérer (on supposera que ce nombre est obligatoirement inférieur à la capacité du tableau qui sera utilisé pour leur stockage),
2. permette la saisie de plusieurs étudiants, ces étudiants devant être rangés dans un tableau (surdimensionné, nous n'avons pas encore les connaissances permettant de définir une taille dynamique pour ce tableau),
3. affiche la moyenne de chacun des étudiants présents dans ce tableau d'étudiants<sup>⑤</sup>,
4. affiche la moyenne des moyennes des étudiants.

## SECTION

# 7

## Chaînes de caractères

Comme précisé (page 12), il n'existe pas de type de données correspondant aux chaînes de caractères en C. Le seul moyen de les manipuler est de définir des tableaux de caractères. Ce mode de représentation est **standard** en C et il est **normalisé**. En effet, la définition `char chaîne[50];`, correspondant à un tableau de caractères à 50 emplacements, devrait permettre potentiellement de représenter toute chaîne de caractères ayant une longueur comprise entre 0 et 50 caractères. Cependant, comme nous l'avons vu (page 12), les variables ne sont pas initialisées en C. Les tableaux ne dérogent pas à la règle et leurs éléments sont donc eux aussi non initialisés (et peuvent contenir n'importe quoi, y compris des caractères...) Incidemment, si une chaîne de caractères de longueur 15 est stockée dans un tableau de capacité 50, comment distinguer par la suite les caractères pertinents de ceux qui ne le sont pas ?

Conventionnellement, pour éviter ce genre de problèmes, les chaînes de caractères, stockées dans les tableaux de caractères, sont suffixées par un caractère spécial, `'\0'`, dont le code ASCII est égal à 0. Ainsi, si la variable `chaîne` définie ci-dessus contient la chaîne de caractères "bonjour", la représentation en mémoire sera celle présentée (Fig. 2.3).

b	o	n	j	o	u	r	\0	?	?	.....
---	---	---	---	---	---	---	----	---	---	-------

Figure 2.3 : Représentation d'une chaîne de caractères en C.

Cette convention est utilisée dans tous les programmes C et en premier lieu dans les bibliothèques standards C. **Il est donc obligatoire de la suivre...** Il en résulte un certain nombre de conséquences :

- Le caractère `'\0'` occupant une case mémoire, il n'est du coup possible que de stocker une chaîne d'au maximum  $n - 1$  caractères dans un tableau de  $n$  caractères.
- Le type « chaîne de caractères » n'existant pas en C, il n'y a donc **aucun** opérateur permettant de les manipuler (donc même pas l'opérateur d'affectation `=`). Toutes les manipulations de chaînes de caractères doivent être réalisées au moyen de fonctions dédiées, qui se basent justement sur la convention exposée ci-dessus.
- Dans les bibliothèques standards C, les déclarations de ces fonctions sont regroupées dans le fichier d'en-tête `string.h`. On y trouve en particulier :
  - `strcpy` : permettant de copier une chaîne de caractères d'un emplacement mémoire à un autre (attention, le premier paramètre de cette fonction est la chaîne destination et le second est la chaîne source),

<sup>⑤</sup> L'appel `printf("%f\n", r);` permet d'afficher le contenu de la variable réelle `r`.



- `strcmp` : permettant de comparer deux chaînes de caractères,
  - `strlen` : permettant de calculer la longueur d'une chaîne de caractères.
- Toutes ces fonctions s'attendent donc à ce que les chaînes de caractères soient terminées par un `'\0'` et l'ajoutent automatiquement lors des manipulations qu'elles sont amenées à faire. Pour plus d'information sur ces fonctions, et les autres, utiliser le manuel en ligne sous Linux (ex : `man strcpy`, `man string.h`).
- De même, les constantes chaînes de caractères du type `"bonjour"` incluent automatiquement ce marqueur de fin de chaîne.

Quelques exemples de manipulation de chaînes de caractères en C :

```

1  /* Définition de variables */
2
3  char ch1[50];
4  char ch2[30];
5
6  /* Affectations de chaînes de caractères */
7
8  strcpy(ch1, "bonjour");
9  strcpy(ch2, ch1);
10 strcpy(ch1, "autre chaîne");
11
12 /* Comparaison de chaînes */
13
14 if (strcmp(ch1, ch2) == 0)
15     printf("Les deux chaînes sont identiques...\n");

```

**Exercice 4** Suite de l'exercice sur les étudiants...

- Redéfinir le type `Etudiant` en remplaçant le numéro des étudiants par leur nom (dont la longueur est au maximum de 50 caractères).
- Faire du refactoring sur le reste du programme pour qu'il tienne compte de cette modification.

## SECTION

# 8

## Fonctions

### 1 Définition, déclaration

Contrairement à Java, C ne supporte aucune caractéristique du paradigme objet. De ce fait, entre autres, il existe un certain nombre de différences dans la définition et la manipulation des fonctions en C par rapport aux méthodes en Java :

- Il n'y a aucune méthode en C, il n'y a que des fonctions...
- La surcharge n'existe pas : il n'est donc pas possible comme en Java de définir plusieurs fonctions ayant le même nom mais ayant une liste de paramètres différente.
- Il ne peut pas y avoir deux fonctions ayant le même nom dans un *programme* C (et donc pas seulement dans un *module*). Là où il était possible d'avoir une méthode `afficher()` dans une classe `Client` et une autre méthode `afficher()` dans une classe `Produit` en Java, il faudra définir les deux fonctions `client_afficher()` et `produit_afficher()` en C.

Les autres aspects liés à la définition et à la déclaration des fonctions en C ont déjà été traités (page 12).

## 2 Appels

Les fonctions, contrairement aux méthodes, ne peuvent pas être invoquées sur des objets, puisque ceux-ci n'existent pas... Par analogie, pour afficher un client dans un programme, on aura en Java et en C (la fonction `printf()` est présentée plus en détail ([page 22](#))) :

### Classe client (Client.java) en Java

```
1 class Client {
2     String nom;
3     int age;
4
5     Client(String n, int a)
6     {
7         nom = n;
8         age = a;
9     }
10
11     void afficher()
12     {
13         System.out.println("Nom : " + nom + ", âge : " + age);
14     }
15 }
```

### Affichage d'un client en Java

```
1 Client c;
2
3 c = new Client("Durand", 35);
4 c.afficher();
```

### Affichage d'un client en C

```
1 #include <stdio.h>
2 #include <string.h>
3
4 typedef struct {
5     char nom[50];
6     int age;
7 } Client;
8
9 void client_afficher(Client c)
10 {
11     printf("Nom : %s, âge : %d\n", c.nom, c.age);
12 }
13
14 int main()
15 {
16     Client c;
```

```

17
18     strcpy(c.nom, "Durand");
19     c.age = 35;
20
21     client_afficher(c);
22     return 0;
23 }

```

En C, comme dans la majeure partie des langages de programmation, les passages de paramètres sont faits par *valeur*. Ainsi, lors d'un appel, la valeur des paramètres effectifs est *recopiée* dans les paramètres formels.

### 3 Cas des tableaux

Les tableaux peuvent contenir de nombreux éléments, ils sont justement utilisés pour cela... Lors d'un passage de tableau comme paramètre d'une fonction, les éléments du tableau ne sont du coup pas copiés : **on ne fait que passer une référence vers le tableau comme paramètre**, ce qui occupe moins de place en mémoire. Une référence vers un tableau est facilement accessible : il suffit d'utiliser l'identificateur du tableau seul, comme cela est expliqué (page 14). Par ailleurs, comme les tableaux ne disposent pas intrinsèquement du nombre d'éléments qu'ils contiennent (comme cela peut être le cas en Java au moyen de l'attribut `length`), on ajoute généralement un paramètre pour chaque tableau passé comme paramètre d'une fonction, permettant de connaître la longueur du tableau correspondant.

```

1  #include <stdio.h>
2
3  void affiche_tableau(int tab[], int taille)
4  {
5      int i;
6
7      for (i = 0; i < taille; i++)
8          printf("Élément no %d : %d\n", i + 1, tab[i]);
9  }
10
11 int main()
12 {
13     int tab1[20];
14     int tab2[30];
15
16     /* Initialisation des tableaux... */
17
18     /* Affichage de chaque tableau */
19
20     affiche_tableau(tab1, 20);
21     affiche_tableau(tab2, 30);
22
23     return 0;
24 }

```

On constate en ligne n° 3 que le paramètre `tab` est bien défini comme un tableau, mais que la taille n'est pas précisée (il n'y a aucun indice entre les crochets []). Cette syntaxe permet de spécifier que la fonction recevra une référence sur un tableau plutôt qu'un tableau en lui-même. Cela ne change pas la

syntaxe d'accès aux éléments de ce tableau par la suite, comme cela est visible ligne n° 8. Par ailleurs, cette syntaxe présente l'avantage de pouvoir fournir des tableaux de taille différente comme paramètres de cette fonction (voir les appels de fonction en lignes n° 20 et n° 21).

Ce mode de transmission a plusieurs incidences :

- Une fonction peut modifier les éléments d'un tableau défini en dehors de cette fonction, si le tableau lui est passé comme paramètre. Les tableaux qui sont passés comme paramètres de fonction se comportent donc automatiquement comme les paramètres InOut vus en algorithmique.
- Le développeur doit être particulièrement vigilant lors des appels de fonctions acceptant des tableaux comme paramètres. Il faut en effet que le paramètre `taille` fourni concorde bien avec la taille effective du tableau associé... <sup>⑥</sup>

## ④ Cas des chaînes de caractères

En C, les chaînes de caractères sont de simples tableaux de caractères ([page 16](#)). Le passage de chaînes de caractères comme paramètres d'une fonction suit donc les mêmes règles que celles relatives aux tableaux, présentées dans le paragraphe précédent. Il y a toutefois une différence : les chaînes de caractères incluent automatiquement le marqueur de fin de chaîne `'\0'` et elles sont principalement manipulées par l'intermédiaire des fonctions standards déclarées dans `string.h`. Par conséquent, il est inutile d'ajouter un paramètre indiquant la taille des chaînes lorsqu'elles sont paramètres de fonction.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void affiche_chaine(char ch[])
5  {
6      printf("Chaîne %s de longueur %d\n", ch, strlen(ch));
7  }
8
9  int main()
10 {
11     char ch1[20];
12     char ch2[30];
13
14     /* Initialisation des chaînes... */
15
16     strcpy(ch1, "Bonjour...");
17     strcpy(ch2, "...tout le monde !");
18
19     /* Affichage de chaque chaîne */
20
21     affiche_chaine(ch1);
22     affiche_chaine(ch2);
23
24     return 0;
25 }
```

**Exercice 5** *Toujours en considérant l'exercice sur les étudiants :*

---

<sup>⑥</sup> En Java, ces deux paramètres n'en font qu'un, étant donné que les tableaux incluent automatiquement un attribut `length`, ce qui empêche ce type d'inconsistance.

1. Définir une fonction `affiche_resultat()` pour qu'elle affiche le nom de l'étudiant passé en paramètre ainsi que sa moyenne<sup>7</sup>. Utiliser naturellement la fonction `moyenne()` lors de cette définition...
2. Refactoring
  - Définir une fonction `affiche_moyennes()`, acceptant un tableau d'étudiants en paramètre, affichant la moyenne de chaque étudiant<sup>8</sup>.
  - Définir une fonction `moyenne_generale()`, acceptant un tableau d'étudiants en paramètre, renvoyant la moyenne des moyennes des étudiants.
  - Le programme principal précédent intégrant déjà ces fonctionnalités, le modifier pour qu'il fasse maintenant appel à ces fonctions.
3. Définir une fonction comptant le nombre de caractères d'une chaîne de caractères, passée en paramètre, sans vous servir des fonctions standards de manipulation des chaînes. Déclarer cette même fonction.

## SECTION

# 9

## Adresses mémoire

Le langage C a été conçu en partie pour écrire le système Unix (page 5). Incidemment, même si C est un langage de haut-niveau, il permet également d'accéder très finement aux ressources disponibles sur une machine. L'un des passages obligés pour permettre ce type d'accès est de permettre la manipulation des adresses mémoire, associées aux variables, aux fonctions...

Nous nous intéresserons uniquement ici aux adresses mémoire associées aux variables. C dispose d'un opérateur, « & », permettant de renvoyer l'adresse de n'importe quelle variable. Ainsi, si `v` est une variable, `&v` correspond à l'adresse de cette variable. De même, si on considère la variable `tab[2]` (`tab` étant un tableau), l'expression `&tab[2]` renvoie l'adresse du troisième élément du tableau `tab`<sup>9</sup>. Les adresses peuvent ensuite être utilisées partout où est cela est nécessaire, par exemple pour fournir des arguments à la fonction `scanf()` (page 22).

Remarque : si la variable considérée est un tableau, comme dans l'exemple `int tab[20]`, nous savons déjà que l'identificateur du tableau utilisé seul (`tab`) référence le début du tableau en mémoire (page 14). Il se trouve que l'adresse du début d'un tableau en mémoire correspond à l'adresse du premier de ses éléments, soit `&tab[0]` sur l'exemple que nous traitons. On peut donc reformuler la propriété vue en énonçant que l'identificateur d'un tableau utilisé seul correspond à l'adresse du premier de ses éléments, soit encore que `tab ≡ &tab[0]` pour n'importe quel tableau `tab` donné<sup>10</sup>.

### Exercice 6 Quelques questions applicatives :

1. Définir un entier et un réel. Comment obtenir leur adresse ?
2. Définir un tableau de 20 entiers. Quelle est l'adresse du cinquième élément de ce tableau ?
3. Selon vous, est-ce que l'écriture `&&a` a un sens, sachant que la variable `a` est définie comme `int a;` ?

<sup>7</sup> L'appel `printf("%s\n", ch);` permet d'afficher le contenu de la chaîne de caractères stockée dans le tableau de caractères `ch` (possédant par exemple une définition du genre `char ch[30];`).

<sup>8</sup> L'appel `printf("%f\n", r);` permet d'afficher le contenu de la variable réelle `r`.

<sup>9</sup> Attention à ne pas oublier le décalage qui existe entre les indices d'un tableau et les numéros d'éléments correspondant...

<sup>10</sup>  $\equiv$  est le symbole de l'équivalence.

# 10 printf() et scanf()

Les fonctions `printf()` et `scanf()` sont des fonctions permettant de réaliser des entrées-sorties C au même titre que les méthodes `System.out.println()` et `Scanner.nextXxx()` en Java (Tab. 2.1). Ce sont des fonctions acceptant un nombre variable de paramètres <sup>11</sup>.

## 1 L'affichage : printf()

Comme son nom l'indique, cette fonction réalise des écritures formatées. Dans sa forme la plus simple, cette fonction prend un unique paramètre, la chaîne de caractères qui doit être affichée :

- `printf("Hello");`
- `printf("Bonjour à tous!\n");`

Dans l'exemple ci-dessus, l'usage de `\n` permet, comme en Java, de revenir à la ligne. Il est également possible d'afficher le contenu de variables grâce à `printf()`. Il faut dans ce cas utiliser des *spécificateurs de format* dans la chaîne que l'on souhaite imprimer, en ajoutant autant de paramètres à `printf()` qu'il y a de spécificateurs de format, en prenant soin de faire concorder l'ordre des paramètres par rapport à l'ordre des spécificateurs de format. Les *spécificateurs de format* sont des séquences du type `%x`, où `x` permet d'indiquer à `printf()` le format d'impression souhaité. Les principaux spécificateurs de format sont présentés (Tab. 2.2). La liste est non exhaustive... faire un `man printf` ou `man scanf` pour plus de détails.

Formateur	Type associé
<code>%d</code>	entier
<code>%f</code>	réel
<code>%c</code>	caractère (1 seul)
<code>%s</code>	chaîne de caractères

Table 2.2 : Principaux spécificateurs de format en C.

Le développeur doit être **particulièrement vigilant** à bien faire concorder les spécificateurs de format par rapport aux variables qu'il souhaite afficher :

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main()
5  {
6      int a = 8;
7      float b = 9.4;
8      char c = 'e';
9      char chaine[50];
10
11     /* Affectation d'une chaîne */
12
13     strcpy(chaine, "Bonjour...");
14

```

<sup>11</sup> Il existe une syntaxe en C vous permettant de définir vos propres fonctions acceptant un nombre variable de paramètres. Cet aspect n'est pas traité dans ce cours, il en dépasse les objectifs.

```

15  /* Affichages corrects d'une valeur à la fois */
16
17  printf("%d\n", 4);
18  printf("%d\n", a);
19  printf("%f\n", b);
20  printf("%c\n", c);
21  printf("%s\n", chaine);
22  printf("%s\n", "hello");
23
24  /* Affichages corrects d'une variable à la fois et de texte */
25
26  printf("Valeur de a : %d\n", a);
27  printf("Valeur de b : %f\n", b);
28  printf("%s les gens !\n", chaine);
29
30  /* Autres affichages corrects */
31
32  printf("L'entier défini dans ce programme vaut %d, le réel vaut %f\n",
33        a, b);
34  printf("%s %s %s\n", chaine, chaine, chaine);
35
36  /* Affichages *invalides*, qui compileront cependant... */
37
38  printf("%s\n", a);
39  printf("%d\n", chaine);
40
41  return 0;
42 }

```

Comme on peut le constater sur l'exemple précédent, les paramètres à fournir à `printf()` sont, suivant les cas :

- Pour les variables *entières*, *réelles* ou *caractères* : la variable en elle-même. Comme les passages de paramètres se font par valeur en C (page 17), le fait d'utiliser la variable en elle-même permet bien de fournir directement les *valeurs* que l'on souhaite afficher.
- Pour les variables contenant des *chaînes de caractères*, c'est-à-dire les tableaux de caractères (page 16), il faut fournir l'*adresse* de début de chaîne, directement accessible *via* l'identificateur du tableau utilisé seul (page 14). En interne, en recevant ce type de paramètre, la fonction `printf()` :
  1. teste si le caractère se trouvant à l'adresse mémoire reçue n'est pas le caractère `'\0'` (sinon elle considère que l'affichage de la chaîne de caractères courante est terminé),
  2. affiche le caractère se trouvant à l'adresse mémoire,
  3. incrémente l'adresse mémoire pour se déplacer sur le caractère suivant,
  4. retourne à l'étape n° 1.

Au final, cet algorithme permet bien d'afficher l'intégralité d'une chaîne de caractères dont l'adresse de début a été fournie, à la condition que cette chaîne se termine par le caractère `'\0'`, ce qui est systématique en C...

## 2 La saisie : `scanf()`

La fonction `scanf()` fonctionne selon un principe analogue à celui de la fonction `printf()`, mais permet de réaliser des saisies au clavier. Comme `printf()`, `scanf()` a comme premier paramètre une

chaîne de caractères incluant des spécificateurs de format, auquel il faut ajouter un paramètre supplémentaire par spécificateur défini. Les spécificateurs utilisables sont les mêmes que ceux existants pour `printf()` (Tab. 2.2).

Il y a par contre une **différence fondamentale** entre ces deux fonctions au niveau de la nature des paramètres supplémentaires à leur fournir. Là où `printf()` attend en général des *valeurs* (c'est toujours le cas, sauf pour les chaînes de caractères), `scanf()` attend **systématiquement des adresses**. C'est d'ailleurs tout à fait logique ! Le rôle de la fonction `scanf()` est de réaliser des saisies (au clavier) et de stocker les valeurs lues dans des variables. Considérons, par exemple, que l'on souhaite lire un entier dans une variable `int a`; Si on écrit l'instruction `scanf("%d", a);`, la fonction `scanf()` va recevoir en paramètre la *valeur* contenue dans la variable `a`. Cette valeur ne lui sera d'aucune utilité pour savoir *où* stocker la valeur qu'elle doit lire au clavier... En revanche, si on écrit l'instruction `scanf("%d", &a);`, la fonction `scanf()` reçoit l'*adresse* de la variable `a` et va donc pouvoir stocker la valeur lue au clavier à cette adresse...

Comme pour la fonction `printf()`, le développeur doit être **particulièrement vigilant** à bien faire concorder les spécificateurs de format par rapport aux variables qu'il souhaite lire :

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int a;
6      float b;
7      char c;
8      char chaine[50];
9
10     /* Lecture d'une valeur à la fois */
11
12     scanf("%d", &a);
13     scanf("%f", &b);
14     scanf("%c", &c);
15     scanf("%s", chaine);
16
17     /* Affichage des valeurs lues, pour vérification */
18
19     printf("%d\n", a);
20     printf("%f\n", b);
21     printf("%c\n", c);
22     printf("%s\n", chaine);
23
24     /* Lecture de plusieurs variables à la fois */
25
26     scanf("%d %f", &a, &b);
27     scanf("%s,%d", chaine, &a);
28
29     /* Lectures *invalides*, qui compileront cependant... */
30
31     scanf("%d", a); /* C'est l'adresse de "a" qui est attendu, pas sa
        valeur */
32     scanf("%d", &b); /* Spécificateur de format et type de variable
        discordants */
33
34     return 0;
```



Attention : le format de la première chaîne de caractères, celle incluant les spécificateurs de format, doit être **scrupuleusement respecté** lors de la saisie. Si ce format n'est pas respecté, l'appel à la fonction `scanf()` ne se termine pas, la fonction restant en attente des informations de ce format.

- Cette remarque est sans incidence sur des « lectures simples », telles que celles des lignes n° 12 à 15 par exemple. En effet, dans ces cas, la seule information attendue par `scanf()` est la valeur d'une variable (il convient toutefois de saisir au clavier le bon type de données naturellement).
- En revanche :
  - pour la ligne n° 26, où deux variables sont lues : ces deux variables doivent **obligatoirement être séparées** par une espace <sup>12</sup> ;
  - pour la ligne n° 27, où deux variables sont également lues : ces deux variables doivent **obligatoirement être séparées** par une virgule.

**Exercice 7** Définir une fonction qui prenne un groupe d'étudiants en paramètre et qui affiche, pour chaque étudiant, son nom et sa moyenne (en considérant ses 3 notes).

#### Débordement mémoire

Un des dangers d'utilisation d'une fonction comme `scanf()` réside dans les possibilités de débordement mémoire qu'elle permet dans certaines situations. En effet, que se passe-t-il si on prévoit la lecture d'une chaîne de caractères en réservant  $n$  caractères pour cette chaîne et qu'une saisie dépasse cette capacité ? Considérons par exemple le programme suivant :

```

1 #include <stdio.h>
2
3 int main()
4 {
5     char chaine[4];
6
7     printf("Entrez une chaîne :\n");
8     scanf("%s", chaine);
9     printf("Vous avez tapé : %s\n", chaine);
10    return 0;
11 }
```

En entrant une chaîne d'une longueur supérieure à la prévision, on obtient :

```

1 Entrez une chaîne :
2 Salut !
3 Vous avez tapé : Salut
4 *** stack smashing detected ***: ./testscanf terminated
5 zsh: abort (core dumped) ./testscanf
```

Ce comportement est heureux ! Nous avons droit à une erreur franche, ce qui va permettre une correction rapide. Mais il est aussi dû au fait que ce programme a été compilé et exécuté dans un environnement récent. Sur des environnements plus anciens, pas de message d'erreur, mais une erreur (aléatoire) tout de même et une porte ouverte aux pirates capables d'exploiter ce genre de failles...

<sup>12</sup> On rappelle que le nom « espace » est féminin lorsqu'il désigne le caractère de typographie.

Quelles sont les solutions permettant de résoudre ce problème ? Tour d'abord, une remarque : la fonction `scanf()` n'est pas vraiment sensée être utilisée dans un dialogue avec un utilisateur. Comme son nom l'indique, elle est plutôt destinée aux saisies *formatées*, qu'on réalise généralement dans des fichiers générés par des programmes. Cependant, même dans ce contexte, il est nécessaire de se prémunir contre ce genre de failles. Donc, quelques solutions :

- Utiliser un spécificateur de format décrivant la longueur maximale à saisir. Dans l'exemple précédent, ce serait ainsi : `scanf("%3s", chaîne);`. La saisie s'arrête ainsi automatiquement au bout du troisième caractère, évitant ainsi les débordements. Cependant, cette solution ne peut pas être utilisée dans les cas où la longueur n'est pas connue à la compilation (ici, le « 3 » ne peut pas être paramétré).
- Réaliser la saisie au moyen d'une fonction comme `fgets()` qui permet d'exprimer directement dans les paramètres qu'elle attend la longueur maximale attendue. Cette fonction va lire une chaîne tenant sur une ligne — c'est le retour chariot qui stoppe la saisie —, à partir de laquelle on peut ensuite extraire des informations formatées au moyen d'une fonction comme `sscanf()`.

## SECTION

# 11

## Synopsis de la construction d'un programme C

### 1 Introduction

Comme cela est présenté (Fig. 2.4), un programme C est constitué d'un ensemble de fichiers sources destinés à être compilés séparément et à subir une édition de liens commune. Ces fichiers sources sont aussi appelés modules et ce type de programmation est appelé programmation modulaire.

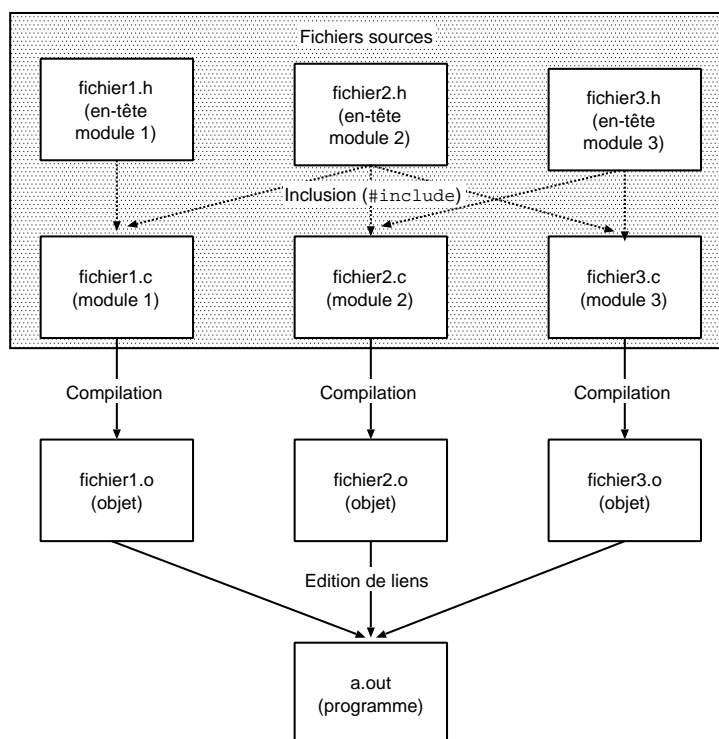


Figure 2.4 : Schéma de la construction d'un programme C.

Le fait de pouvoir compiler chaque fichier source de manière autonome amène à concevoir des programmes de manière modulaire en regroupant, dans chaque fichier source, des fonctions qui manipulent

les mêmes variables ou qui participent aux mêmes algorithmes. Chaque module est réparti dans deux fichiers distincts :

- un fichier à extension `.h` (c'est le fichier d'en-tête ou *header* en anglais) contient les prototypes des fonctions,
- un fichier à extension `.c` qui contient l'implémentation des fonctions.

## 2 Le préprocesseur

Les inclusions de fichiers d'en-têtes sont réalisées grâce au préprocesseur, au moyen de la directive `#include`. Chaque module inclut obligatoirement le fichier d'en-tête qui lui est associé<sup>13</sup> et inclue également les fichiers d'en-tête des modules qu'il utilise (voir un exemple d'inclusions possibles sur la (Fig. 2.4)).

Le préprocesseur ou pré-compilateur est un utilitaire qui traite automatiquement et implicitement le fichier source avant le compilateur. C'est un manipulateur de chaînes de caractères. Il retire les parties de commentaires, qui sont comprises entre `/*` et `*/`. Il prend aussi en compte les lignes du texte source ayant un `#` en première colonne pour créer le texte que le compilateur analysera. Ses possibilités sont de trois ordres :

- inclusion de fichiers,
- définition d'alias et de macro-expressions,
- sélection de parties de texte.

## 3 Compilation, fichiers objets

Le fichier résultat produit par le préprocesseur est ensuite *compilé*. Cette compilation correspond à un certain nombre d'étapes menant à la génération d'un fichier *objet*<sup>14</sup> :

- une *analyse lexicale*, correspondant à la transformation du code source en une liste de *lexèmes* (les identificateurs, les opérateurs, les constantes, etc.),
- une *analyse syntaxique*, permettant de vérifier que les instructions présentes dans le source ont une syntaxe conforme au langage (par analogie avec le français, cela revient par exemple à vérifier que les phrases ont bien une construction du type « sujet-verbe-complément »),
- une *analyse sémantique*, permettant de vérifier si les instructions ont du sens (qu'on affecte pas une chaîne à un entier par exemple),
- la génération d'un *fichier objet*, contenant les instructions assembleurs correspondant aux instructions C contenues dans le source initial.

Les fichiers objets contiennent des instructions assembleurs, mais également des informations permettant de savoir quelles fonctions ils utilisent et quelles fonctions ils définissent. Par exemple, pour le fichier source suivant :

```
1 #include <stdio.h>
2
3 int min(int a, int b)
4 {
5     return (a < b)? a : b;
6 }
7
8 int max(int a, int b)
9 {
10    return (a > b)? a : b;
```

<sup>13</sup> Ce qui permet de vérifier à la compilation la cohérence entre les déclarations du module — présentes dans le `.h` — et les définitions du module — présentes dans le `.c`.

<sup>14</sup> Attention, dans ce contexte, ce terme n'a évidemment rien à voir avec la conception objet.

```

11 }
12
13 int main()
14 {
15     int x = 8;
16     int y = 9;
17
18     printf("Le minimum est %d\n", min(x, y));
19     printf("Le maximum est %d\n", max(x, y));
20
21     return 0;
22 }

```

Il est possible de savoir quelles fonctions sont définies et quelles fonctions sont utilisées. La commande **nm**<sup>15</sup> permet d'avoir accès à tous les symboles présents dans un fichier objet :

```

1 royal 54 : nm exobj.o
2 0000004a T main
3 00000025 T max
4 00000000 T min
5          U printf

```

Les résultats fournis par **nm** correspondent à :

- avec un préfixe « T » : à un symbole défini dans le fichier objet (l'adresse à laquelle sont disponibles les instructions assembleurs correspondantes est alors en préfixe),
- avec un préfixe « U » : à un symbole utilisé dans le fichier objet, mais non défini dans ce même objet.

## 4 Édition de liens

La phase d'édition de liens consiste à assembler tous les fichiers objets entrant dans la composition d'un programme afin de générer un exécutable. Lors de cette phase, il est également vérifié que toutes les fonctions utilisées dans des modules, mais non définies dans ceux-ci (donc les fonctions préfixées par « U » avec la commande **nm**), sont bien présentes dans un des fichiers objets à considérer. Si des fonctions manquent à l'appel, l'édition de liens ne peut avoir lieu et la liste des fonctions manquantes est alors affichée.

### SECTION

## 12 Construction et ligne de commande

### 1 Généralités

Les travaux pratiques réalisés lors de ce cours seront effectués sous Linux, *via* une connexion SSH. Le compilateur C disponible sous cet environnement, utilisé pour construire les programmes à réaliser, est **gcc**<sup>16</sup>. Invoqué sans option, **gcc** tente de construire un programme exécutable intégrant tous les fichiers qui lui sont fournis en paramètre. Si aucun nom de programme résultat n'est fourni, **gcc** le nomme **a.out**. Exemples :

<sup>15</sup> Il n'est normalement pas nécessaire d'utiliser cette commande lors d'un développement. Celle-ci est présentée dans ce cours uniquement à des fins pédagogiques et illustratives.

<sup>16</sup> GNU C Compiler.

- `gcc toto.c` : compile le fichier `toto.c` (générant ainsi `toto.o`) et réalise ensuite une édition de liens à partir de `toto.o` pour créer un exécutable nommé `a.out`.
- `gcc toto.c tata.c` : compile **séparément** les fichiers `toto.c` et `tata.c` (générant ainsi `toto.o` et `tata.o`) et réalise ensuite une édition de liens à partir de `toto.o` et `tata.o` pour créer un exécutable nommé `a.out`.
- `gcc toto.c tata.c tutu.o` : compile **séparément** les fichiers `toto.c` et `tata.c` (générant ainsi `toto.o` et `tata.o`) et réalise ensuite une édition de liens à partir de `toto.o`, `tata.o` et `tutu.o` pour créer un exécutable nommé `a.out`.
- `gcc toto.o tata.o tutu.o` : réalise immédiatement une édition de liens (sans aucune compilation) à partir de `toto.o`, `tata.o` et `tutu.o` pour créer un exécutable nommé `a.out`.

## 2 Options fréquentes

Voici quelques autres options très pratiques :

- `-c` : provoque la génération d'un module objet non exécutable et s'arrête avant l'édition de liens.  
`gcc -c toto.c` → produit le fichier `toto.o`.
- `-E` : lance le préprocesseur seul, qui écrit sur la sortie standard.  
`gcc -E toto.c` → affiche le résultat sur la sortie standard
- `-S` : génère le fichier assembleur correspondant à un source C après passage du préprocesseur et du compilateur. Le fichier généré est suffixé par `.s`.  
`gcc -S toto.c` → produit le fichier `toto.s`.
- `-O` : optimise le code généré (à utiliser lors de la compilation, ne sert à rien s'il n'y a qu'une édition de liens). Cette option est incompatible avec l'option `-g`.  
`gcc -O toto.c`
- `-o nom` : donne un nom au fichier résultat différent du nom par défaut (`a.out` dans le cas de la création d'un exécutable par exemple).  
`gcc -o toto toto.c` → produit le fichier exécutable `toto` après avoir compilé `toto.c`.
- `-g` : ajoute des informations de débogage lors de la compilation d'un module. Cette option est à utiliser lors d'une compilation et est nécessaire pour pouvoir effectuer un débogage dans de bonnes conditions avec `gdb` typiquement (page 30).  
`gcc -g -c toto.c`

## 3 Options relatives au respect des normes

Certaines options sont par ailleurs utiles pour forcer la vérification d'une syntaxe correspondant à la norme ANSI (garantissant ainsi une meilleure portabilité) ainsi que pour produire des avertissements passés sous silence sinon par le compilateur :



- `-ansi` : avec cette option, le compilateur se comporte comme un compilateur de langage C ANSI (version C89) sans le support des extensions spécifiques au compilateur utilisé.
- `-pedantic` : cette option demande au compilateur de refuser la compilation de programme non ANSI. Certains avertissements (*warnings*) sont promus comme erreurs (*errors*) à cette fin.
- `-Wall` : cette option augmente le nombre de messages d'alertes (avertissements) générés par le compilateur lorsqu'il rencontre des constructions dangereuses.

## 4 Recommandations

L'utilisation des options présentées ci-dessus est recommandée lors de l'apprentissage du langage C en utilisant le compilateur `gcc`. Incidemment, il est recommandé d'utiliser des lignes de commandes proches de celles présentées dans les exemples ci-dessous dans le cadre de ce cours :

- `gcc -ansi -pedantic -Wall -c toto.c`  
Compilation (uniquement) du fichier `toto.c`, produisant l'objet `toto.o`.
- `gcc -g -ansi -pedantic -Wall -c toto.c`  
Compilation (uniquement) du fichier `toto.c`, produisant l'objet `toto.o`, en incluant des directives de débogage.
- `gcc -ansi -pedantic -Wall toto.c -o toto`  
Compilation du fichier `toto.c`, produisant l'objet `toto.o`, et édition de liens afin de construire un programme exécutable nommé `toto`.
- `gcc toto.o tata.o tutu.o -o yopla`  
Édition de liens incluant les fichiers objets `toto.o`, `tata.o` et `tutu.o` afin de construire un programme exécutable nommé `yopla` (les options `-ansi`, `-pedantic` et `-Wall` sont dans ce cas inutiles puisqu'elles ne sont actives que lors d'une compilation).

Naturellement, l'usage des ces différentes options n'a de sens que si **tous** les messages émanant de `gcc` sont corrigés. Cela sera forcément le cas pour les messages d'erreur, interrompant automatiquement le processus de construction de programme. Mais cela doit également être le cas pour les avertissements. Même si ceux-ci n'empêchent pas la construction du programme, ils indiquent généralement des erreurs de conception qui doivent être corrigées au même titre que les autres erreurs.

Les lignes de commande pouvant être assez longues à saisir, il est fortement conseillé de bien se familiariser avec l'environnement fourni par les *shell* sous Linux. Les touches  et  permettent, en particulier, de naviguer dans l'historique des commandes saisies... Par ailleurs, le chapitre traitant de l'utilitaire `make` (page 47) présente un outil permettant d'automatiser et d'optimiser la construction de programmes C, et ainsi d'échapper en grande partie à ces lignes de commande fastidieuses...

## SECTION

# 13 Un débogueur : gdb

`gdb` est un utilitaire qui permet de déboguer un programme en cours d'exécution (en le déroulant instruction par instruction ou en examinant et modifiant ses données). Il permet également un débogage post-mortem, en analysant un fichier `core` qui représente le contenu d'un programme terminé anormalement.

L'interface de `gdb` est une simple ligne de commande, mais il existe des applications frontales qui lui offrent une interface graphique beaucoup plus conviviale. C'est le cas en particulier de l'utilitaire `ddd`, permettant par exemple de cliquer sur une ligne de code directement dans le listing pour y placer un point d'arrêt alors que `gdb` utilisé seul nécessite la saisie du numéro de ligne.

Les différentes manières de lancer `gdb` sont présentées (Tab. 2.3). Pour que `gdb` puisse afficher les symboles de votre programme (les identificateurs comme les noms de variables et de fonctions en particulier), il est nécessaire que les sources soient compilés avec l'option `-g` de `gcc` (page 29).

Méthode	Commande
Pour déboguer un exécutable	<code>gdb programme</code>
Pour déboguer un exécutable et un fichier <code>core</code>	<code>gdb programme core</code>
Pour déboguer un processus en cours (pid : 1234)	<code>gdb programme 1234</code>

Table 2.3 : Exemples de commandes permettant de lancer `gdb`.

Une fois `gdb` lancé, il est possible d'utiliser les commandes présentées (Tab. 2.4). Lors de l'analyse du crash d'un programme en particulier, la commande `bt` est particulièrement pratique pour comprendre dans quel fichier et à quel numéro de ligne l'erreur s'est produite. Cela fournit généralement un point d'entrée suffisant pour découvrir la source du problème. Par exemple, sur l'extrait suivant :

Commande	Description
run [ <i>liste_arguments</i> ]	lance le programme avec <i>liste_arguments</i>
bt	backtrace : édite la pile du programme (appels imbriqués des fonctions)
print <i>expr</i>	donne la valeur d'une expression <i>expr</i>
c	continue le programme après un arrêt
next	exécute la prochaine ligne de programme
step	exécute la ligne de programme suivante
help <i>nom</i>	informations sur la commande gdb <i>nom</i>
quit	sortir de gdb

**Table 2.4 :** Commandes gdb courantes.

```

1 royal tmp 98 : ./labyrinthe tests/labfich2.txt
2 Segmentation fault (core dumped)
3 royal tmp 99 : gdb labyrinthe core
4 GNU gdb 6.8-debian
5 Copyright (C) 2008 Free Software Foundation, Inc.
6 License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.
  html>
7 This is free software: you are free to change and redistribute it.
8 There is NO WARRANTY, to the extent permitted by law. Type "show copying
  "
9 and "show warranty" for details.
10 This GDB was configured as "i486-linux-gnu"...
11
12 warning: Can't read pathname for load map: Input/output error.
13 Reading symbols from /lib/tls/i686/cmov/libc.so.6...done.
14 Loaded symbols for /lib/tls/i686/cmov/libc.so.6
15 Reading symbols from /lib/ld-linux.so.2...done.
16 Loaded symbols for /lib/ld-linux.so.2
17 Core was generated by './labyrinthe tests/labfich2.txt'.
18 Program terminated with signal 11, Segmentation fault.
19 [New process 9612]
20 #0  0x08048fcd in minimum (a=2147483647, b=2147483647) at outils.c:11
21 11      *pi = 6;
22 (gdb) bt
23 #0  0x08048fcd in minimum (a=2147483647, b=2147483647) at outils.c:11
24 #1  0x08048bd0 in Labyrinthe_Cherche_Sortie (lab=0x9dc7170) at labyrinthe
  .c:258
25 #2  0x08048f61 in main (argc=2, argv=0xbf88c4f4) at main.c:52
26 (gdb) quit

```

- Le programme labyrinthe a rencontré une erreur d'exécution et un fichier core, représentant l'image mémoire du programme au moment de l'erreur, a été généré (ligne n° 2).
- La ligne n° 20 indique dans quel source C l'erreur s'est produit (outils.c) et à quelle numéro de ligne (11). On apprend également que cette ligne est une instruction de la fonction minimum qui a été appelée avec les paramètres a, de valeur 2147483647, et b, de valeur 2147483647.
- La ligne n° 21 affiche l'instruction se trouvant en ligne 11 du fichier outils.c.
- Enfin, la commande bt (ligne n° 22) permet d'afficher l'empilement des fonctions ayant mené à cette instruction, ainsi que les paramètres reçus par ces fonctions...

Pour plus d'informations sur gdb, faire `man gdb` dans votre environnement Linux.

## 14 Un profiler : valgrind

Les problèmes les plus embêtants rencontrés lors du développement d'un programme en C sont liés à des fautes de gestion de la mémoire (accès à une variable non initialisée, utilisation d'un mauvais indice dans un tableau...) Dans le meilleur des cas, le programme sera tué par le système (si la case mémoire en question n'appartient pas au programme), mais bien souvent rien ne sera visible durant l'exécution... hormis un résultat faux à la fin ou un comportement « bizarre ». Seule une bonne expertise permet alors de comprendre la provenance de l'erreur...

Dans ce contexte, l'outil `valgrind` permet la détection et le diagnostic des erreurs de gestion de mémoire. Lorsqu'un programme est exécuté sous la supervision de `valgrind`<sup>17</sup>, toutes les lectures et écritures réalisées en mémoire sont tracées.

Pour utiliser cet outil, il faut :

1. Compiler le programme à tester avec l'option `'-g'`, permettant de générer des informations de débogage (et surtout de faire le lien des erreurs avec la ligne correspondante dans le fichier source) (page 28).
2. Encapsuler l'exécution du programme dans `valgrind` au moyen de la ligne de commande :  
`valgrind ./executable options de l'exécutable`.

`valgrind` est présenté de façon plus complète dans le chapitre traitant des pointeurs (page 65).

## 15 Un exemple

**Exemple** : on souhaite établir des statistiques à partir de notes des étudiants d'une matière. Pour chaque étudiant sont donnés son nom et sa note. Afficher cette information. Pour simplifier notre programme (pour l'instant) nous supposons que le nombre d'étudiants est connu et est égal à 30 par exemple. Voici une version possible du programme demandé en Java :

### Classe Java Etudiant

```

1  import java.util.*;
2
3  /**
4   * Classe Etudiant, chaque objet caracterise un étudiant : son nom
5   * et sa note
6   */
7
8  public class Etudiant
9  {
10     /* Attribut nom d'étudiant */
11     private String nom;
12
13     /* Attribut note d'étudiant */

```

<sup>17</sup> Cet outil est disponible sous version pré-packagée pour la plupart des distributions Linux. Une simple invocation de l'outil de gestion des packages suffit donc bien souvent pour l'installer. Sinon, les sources sont disponibles à l'adresse <http://valgrind.kde.org/>.



```

14     private double note;
15
16     /**
17      * Méthode de saisie du nom et de la note d'un étudiant
18      */
19
20     public void saisirDonnees()
21     {
22         Scanner sc = new Scanner(System.in);
23
24         System.out.println("Le nom de l'étudiant :");
25         nom = sc.nextLine();
26         System.out.println("La note de l'étudiant :");
27         note = sc.nextDouble();
28
29         while ((note < 0) || (note > 20)) {
30             System.out.print("Erreur ! Indiquez une valeur entre 1 et 20
31                               ");
32             note = sc.nextDouble();
33         }
34
35     /**
36      * Méthode d'affichage du nom et de la note d'un étudiant
37      */
38
39     public void afficherDonnees()
40     {
41         System.out.println(" Nom : " + nom + ", note : " + note);
42
43     }
44 }

```

#### Programme principal Java

```

1  /**
2   * Classe principale
3   */
4
5  public class EssaiEtudiant
6  {
7      public static void main(String args[]) {
8          Etudiant [] t;
9          t = new Etudiant[30];
10
11         /* Lecture de données */
12
13         for (int i = 0; i < t.length; i++) {
14             System.out.println("Etudiant " + (i + 1));
15             t[i] = new Etudiant();
16             t[i].saisirDonnees();

```

```

17     }
18
19     /* Affichage de données lues */
20
21     for (int i = 0; i < t.length; i++) {
22         System.out.println("Etudiant " + (i + 1));
23         t[i].afficherDonnees();
24     }
25 }
26 }

```

Et maintenant le même programme en C :

```

1  /* Déclaration de constantes */
2
3  #define TAILLE_NOM 20
4  #define TAILLE_TABLEAU_MAX 30
5
6  /* Inclusion d'une librairie standard */
7
8  #include <stdio.h>
9
10 /* Déclaration d'une structure de type Etudiant */
11
12 typedef struct Etud {
13     char nom[TAILLE_NOM];
14     float note;
15 } Etudiant;
16
17 /* Fonction de lecture de données */
18
19 void saisir_donnees(Etudiant t[], int taille)
20 {
21     int i;
22
23     for (i = 0; i < taille; i++) {
24
25         printf("Nom du %d ème étudiant :\n", i + 1);
26
27         /* Saisir une chaîne de caractères */
28         scanf("%s", t[i].nom);
29
30         printf("Note du %d ème étudiant :\n", i + 1);
31
32         /* Saisir un réel */
33         scanf("%f", &t[i].note);
34
35         while ((t[i].note < 0) || (t[i].note > 20)) {
36             printf("Erreur ! Indiquez une valeur entre 1 et 20\n");
37             scanf("%f", &t[i].note);
38         }
39     }
40 }

```

```

41
42 /* Fonction d'affichage de données lues */
43
44 void afficher_donnees(Etudiant t[], int taille)
45 {
46     int i;
47
48     /* Affichage de chaînes de caractères et de réels */
49     for (i = 0; i < taille; i++)
50         printf("Nom : %s, note : %f\n", t[i].nom, t[i].note);
51 }
52
53 /* Programme principal */
54
55 int main ()
56 {
57     Etudiant t[TAILLE_TABLEAU_MAX];
58     int taille;
59
60     printf("Nombre d'étudiants : \n");
61     scanf("%d", &taille);
62     saisir_donnees(t, taille);
63     afficher_donnees(t, taille);
64
65     return 0;
66 }

```

**Exercice 8** Compléter le programme précédent en y ajoutant les fonctionnalités suivantes :

1. une fonction permettant d'afficher un tableau d'étudiants;
2. une fonction renvoyant la moyenne d'un tableau d'étudiants;
3. une fonction affichant la note minimale d'un tableau d'étudiants et l'affichage du nom d'étudiant ayant cette note (on suppose qu'il n'y en a qu'un);
4. une fonction permettant la mise à jour de toutes les notes d'un tableau d'étudiants en les multipliant par un coefficient donné;

Le programme principal permettant de lancer chaque fonction interactivement est disponible sur l'ENT (prog\_princ\_chap2.c).



# Modularité et bibliothèques

## SECTION

## 1

## Modularité et abstraction

Nous avons programmé une fonction `Trier` qui trie un tableau. Pour utiliser cette fonction, nous n'avons pas besoin de savoir comment est réalisé ce tri. Seuls nous importe le rôle et le prototype de `Trier`. De même, nous n'avons pas besoin de savoir comment est concrètement implantée une structure de données comme une liste. Par contre, nous devons savoir quelles sont les opérations disponibles pour manipuler une structure de données comme une liste.

Nous pouvons donc séparer ce que fait une fonction de comment elle le fait. C'est le principe de modularité. De manière concrète, cela se fait grâce à l'implantation de bibliothèques de fonctions. Lorsque l'on souhaite utiliser une fonction d'une bibliothèque, le programme qui appelle la fonction de la bibliothèque sait ce que fait la fonction, mais ne sait pas comment elle le fait. L'avantage est double. Cela réduit la complexité d'un programme et améliore ainsi sa lisibilité. Cela permet également de créer du code réutilisable.

## SECTION

## 2

## Bibliothèque

Une bibliothèque se construit autour de deux éléments : un *fichier d'en-tête*, qui décrit le contenu de la bibliothèque, et l'*implantation* des fonctions déclarées dans le fichier d'en-tête. Lorsqu'un programme veut utiliser une bibliothèque, le fichier d'en-tête correspondant doit être inclus, par l'intermédiaire d'une instruction du type :

```
1 #include "nom_bibliotheque.h"
```

### ❶ Fichier d'en-tête d'une bibliothèque

Un fichier d'en-tête contient toutes les informations nécessaires à l'utilisation de la bibliothèque, à savoir : les **déclarations de constantes**, de **types** et les **prototypes des fonctions**. Il sert d'interface et

c'est lui que l'on doit consulter quand on veut utiliser une bibliothèque. L'extension d'un fichier d'en-tête est `.h` (pour *header*).

Reprenons l'exemple précédent et créons le fichier d'en-tête `personne.h` de la bibliothèque qui gère une liste de personnes :

```
1  /*****
2      Déclaration des constantes
3      *****/
4
5  #define PERSONNE_MAX_PERSONNES    30    /* Nombre de personnes */
6  #define PERSONNE_LONG_CHAINE      20    /* Taille des chaînes */
7
8  /*****
9      Déclaration des types
10     *****/
11
12  typedef struct {
13      char cnom[PERSONNE_LONG_CHAINE];
14      int  cage;
15  } Personne;
16
17  /*****
18      Prototype des fonctions
19      *****/
20
21  /* Saisir n personnes dans tab_pers */
22  void personne_saisir(Personne tab_pers[], int n);
23
24  /* Afficher n personnes de tab_pers */
25  void personne_afficher(Personne tab_pers[], int n);
26
27  /* Insérer p dans tab_pers trié, retourne la nouvelle taille du tableau
28     */
29  int personne_inserer(Personne tab_pers[], int n, Personne p);
30
31  /* Trier par nom les n premières personnes de tab_pers */
32  void personne_trier(Personne tab_pers[], int n);
33
34  /*
35     Recherche nom parmi les n premières personnes de tab_pers, qui est
36     supposé
37     trié. La fonction retourne place, si nom se trouve à la position place
38     dans tab_pers ; sinon la fonction retourne -1.
39  */
40  int personne_rechercher_dico(Personne tab_pers[], int n, char nom[]);
41
42  /*
43     Supprimer élément p dans un tableau trié tab_pers.
44     La fonction retourne la nouvelle taille du tableau.
45  */
46  int personne_supprimer(Personne tab_pers[], int n, char nom[]);
```

Par convention, les noms sont préfixés ou suffixés par le nom du module, ici `Personne`. Ainsi, le nom de l'identificateur permet de retrouver le nom de la bibliothèque auquel il appartient.

## 2 Implantation des bibliothèques

Une bibliothèque est implantée dans un fichier d'extension `.c`. Dans l'exemple ci-dessus, le nom est `personne.c`. Dans ce fichier, le programme de chaque fonction de la bibliothèque est donné. C'est à ce niveau que les choix d'implantation sont faits. Le fichier d'en-tête est inclus afin de connaître les valeurs des constantes, les types et les prototypes des fonctions. Pour ces deux dernières composantes, cela permet d'assurer la consistance entre un module et son interface (son en-tête). Il ne faudrait pas, par exemple, qu'une fonction soit définie dans un module avec un prototype différent de celui qui est déclaré dans l'en-tête de ce module.

```
1  /* Inclusion des bibliothèques standards */
2
3  #include <stdio.h>
4  #include <string.h>
5
6  /* Inclusion des bibliothèques personnelles */
7
8  #include "personne.h"
9
10 /* Définition des fonctions */
11
12 void personne_saisir(Personne t[], int n)
13 {
14     int i;
15
16     for (i = 0; i < n; i++) {
17         printf("Nom de la personne %d :", i);
18         scanf("%s", t[i].cnom);
19         printf("Âge de %s :", t[i].cnom);
20         scanf("%d", &t[i].cage);
21     }
22 }
23
24 /* Définition des autres fonctions du module... */
```

### SECTION

## 3

## Directives du préprocesseur

Les directives pour le préprocesseur sont des instructions qui sont exécutées avant la compilation (page 26). Nous en connaissons déjà deux importantes : `#include` et `#define`.

### 1 #define

La directive `#define` permet de définir des constantes, des macros. Elle doit figurer en tête de fichier.

```
1 #define PI 3.141592 /* La macro constante PI */
```

Le préprocesseur substitue à l'identificateur `PI` la valeur `3.141592`.

Attention, lors de la définition de constantes, à ne pas terminer cette directive par un « ; » ! En effet, à travers cette commande, le préprocesseur agit comme un simple outil de chercher/remplacer. Une erreur classique est de définir une constante de cette manière :

#### Mauvaise utilisation de `#define`

```
1 #define PI 3.141592;
```

Dans ce cas, toutes les occurrences de `PI` seront remplacées par `3.141592;`, ce qui mène, après remplacement, à des instructions du type :

#### Code (erroné) avant et après remplacement de la constante `PI` mal définie

```
1 if (angle > PI)
2   printf("Angle supérieur à PI...\n");
```

```
1 if (angle > 3.141592;)
2   printf("Angle supérieur à 3.141592;...\n");
```

Comme on le voit après remplacement, les résultats de la substitution contiennent un « ; » de trop... On remarque également au passage que la substitution se fait partout, le préprocesseur ne pouvant pas différencier les contextes d'utilisation des constantes...

Conventionnellement, pour éviter des substitutions malheureuses, on choisit de définir toutes les constantes introduites grâce à `#define` en majuscules. Cela ne supprime pas tous les effets de bord, comme l'exemple ci-dessus le montre, mais permet de les limiter.

## 2 #include

La directive `#include` permet d'accéder aux fonctionnalités d'une bibliothèque. Elle doit figurer en tête de fichier. Les bibliothèques que nous avons déjà vues sont déclarées dans `stdio.h` (comme *Standard Input Output*) qui gère les entrées-sorties, `string.h` pour les chaînes de caractères et `math.h` pour les fonctions mathématiques. Ainsi :

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

permet d'inclure la bibliothèque `stdio.h` et donc d'utiliser les fonctions standards d'entrées/sorties, telles que `printf`, `scanf`... `#include` permet également d'inclure les bibliothèques personnelles :

```
1 #include "personne.h"
```

Dans ce cas, on note que le nom de la bibliothèque figure entre guillemets "" et non pas entre crochets < >, comme pour `stdio.h`. Cela correspond à un répertoire de recherche différent. Le principe est le suivant : lorsque le nom de la bibliothèque est spécifié avec < et >, la recherche du fichier à inclure se fait par défaut dans le répertoire où sont installées les bibliothèques standards de C ; lorsque le nom de la bibliothèque est spécifié avec " et ", la recherche se fait dans le répertoire courant. Il faut donc utiliser



""" pour inclure une bibliothèque personnelle. Dans le cas où la bibliothèque personnelle ne figure pas dans le répertoire courant, il faut expliciter le chemin. Par exemple, pour lire un fichier d'en-tête présent sur un périphérique désigné par `a` :

```
1 #include "a:personne.h"
```

### 3 Compilation conditionnelle

Plusieurs inclusions de la même bibliothèque provoque une erreur, car dans ce cas, les déclarations correspondantes sont dupliquées. Pour éviter cela, nous définissons une *variable drapeau* (un *flag*) lors de la première lecture du fichier d'en-tête. Par la suite, il suffira de tester l'existence de cette variable drapeau pour inclure ou non le fichier d'en-tête. Cela se fait avec les directives `#ifndef...#endif`, qui testent si une macro n'a pas été définie, ou `#ifdef...#endif`, qui testent si une macro a été définie.

```
1 #ifndef H_PERSON /* Teste si H_PERSON n'est pas définie */
2 #define H_PERSON /* Dans ce cas, définir H_PERSON */
3
4 /* ... Fichier d'en-tête ... */
5
6 #endif /* Fin de si (c'est-à-dire de #ifndef) */
```

Ces directives du préprocesseur peuvent être utilisées n'importe où dans un programme. La directive `#define VGA` définit la variable drapeau `VGA`. Nous pouvons tester son existence par `#ifdef (VGA)` (ou `#if defined VGA`).

**Exercice 9** Écrire dans le programme ci-dessous les lignes de codes supplémentaires qui permettent d'afficher les valeurs des variables `n` et `m` si la variable drapeau `DEBUG` est définie.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int n,m,p;
6     float x;
7
8     printf("Valeur de m et n : ");
9     scanf("%d %d", &m, &n);
10    x = (float) m / n; /* Calcul de la division en réel */
11    p = 100 * x;
12    printf("p = %d\n", p);
13
14    return 0;
15 }
```

### 4 Exemple de compilation conditionnelle

Voici un exemple où l'utilisation de la compilation conditionnelle est indispensable. Considérons les programmes suivants :

Fichier `a.h`

```
1 typedef struct {
```

```

2     int elt;
3 } A;
4
5 int fonction_a (A e);

```

Fichier a.c

```

1 #include "a.h"
2
3 int fonction_a (A e)
4 {
5     return(e.elt) * 3;
6 }

```

Fichier b.h

```

1 #include "a.h"
2
3 typedef struct {
4     int elt;
5     A   eltA;
6 } B;
7
8 int fonction_b(B e);

```

Fichier b.c

```

1 #include "b.h"
2
3 int fonction_b(B e)
4 {
5     return (e.elt + fonction_a(e.eltA));
6 }

```

Fichier c.h

```

1 #include "a.h"
2
3 typedef struct {
4     int elt ;
5     A   eltA ;
6 } C;
7
8 int fonction_c(C e);

```

Fichier c.c

```
1 #include "c.h"
2
3 int fonction_c(C e)
4 {
5     return (((int) e.elt) - fonction_a (e.eltA));
6 }
```

Fichier main.c

```
1 #include <stdio.h>
2
3 #include "a.h"
4 #include "b.h"
5 #include "c.h"
6
7 int main()
8 {
9     A a;
10    B b;
11    C c;
12
13    a.elt = 1;
14    b.elt = 5;
15    b.eltA = a;
16    c.elt = 6;
17    c.eltA = a;
18
19    printf("%d\n", fonction_b(b) + fonction_c(c));
20
21    return 0;
22 }
```

On compile et on obtient le résultat suivant :

```
1 royal test 58 : gcc -c a.c
2 royal test 59 : gcc -c b.c
3 royal test 60 : gcc -c c.c
4 royal test 61 : gcc -c main.c
5 In file included from b.h:1,
6     from main.c:4:
7 a.h:3: error: conflicting types for 'A'
8 a.h:3: error: previous declaration of 'A' was here
9 a.h:5: error: conflicting types for 'fonction_a'
10 a.h:5: error: previous declaration of 'fonction_a' was here
11 In file included from c.h:1,
12     from main.c:5:
13 a.h:3: error: conflicting types for 'A'
14 a.h:3: error: previous declaration of 'A' was here
```

```
15 a.h:5: error: conflicting types for 'fonction_a'
16 a.h:5: error: previous declaration of 'fonction_a' was here
17 main.c: In function 'main':
18 main.c:15: error: incompatible types in assignment
```

Nous avons plusieurs erreurs à cause de l'inclusion multiple du fichier `a.h` dans le fichier `main.c`. Pour résoudre ce problème, il est nécessaire d'utiliser la compilation conditionnelle. Voici les corrections correspondantes :

#### Fichier `a.h` corrigé

```
1 #ifndef _A
2 #define _A
3
4 typedef struct {
5     int elt ;
6 } A;
7
8 int fonction_a (A e);
9
10 #endif
```

#### Fichier `b.h` corrigé

```
1 #ifndef _B
2 #define _B
3
4 #include "a.h"
5
6 typedef struct {
7     int elt;
8     A   eltA;
9 } B;
10
11 int fonction_b(B e);
12
13 #endif
```

#### Fichier `c.h` corrigé

```
1 #ifndef _C
2 #define _C
3
4 #include "a.h"
5
6 typedef struct {
7     int elt;
8     A   eltA;
9 } C;
```

```
10  
11 int fonction_c(C e);  
12  
13 #endif
```



## SECTION

## 1

## Introduction

`make` est un logiciel permettant d'automatiser les étapes de construction de programmes, de fichiers, de ressources... Il sert à appeler des commandes créant des fichiers, en invoquant ces commandes uniquement si elles sont nécessaires. Dans le contexte particulier de la construction de programmes C, `make` permet d'automatiser la production des fichiers objets et des programmes en eux-mêmes, en facilitant les étapes de compilations et d'éditions de liens nécessaires. Pour parvenir à ces automatisations, `make` utilise un fichier de configuration appelé `makefile` ou plus souvent `Makefile`. En se basant sur ce fichier, `make` détermine quels fichiers doivent être générés et comment le faire.

L'objectif de ce chapitre est de fournir une initiation à la conception de fichiers `Makefile`. La connaissance de `make` est vivement conseillée pour tout projet de programmation. C'est une bonne manière d'économiser du temps et d'être efficace, quelle que soit la taille du projet.

La suite de ce chapitre explique, pas à pas, comment obtenir un fichier `Makefile` générique, utilisable sur n'importe quel projet de petite et moyenne envergure. Il est possible d'utiliser directement le résultat obtenu, correspondant aux deux derniers `Makefiles` décrits, et de survoler les explications techniques menant à ces résultats. Les explications fournies peuvent n'être utilisées qu'en cas de besoin, typiquement lorsqu'il devient nécessaire de faire évoluer les `Makefiles` génériques fournis pour intégrer des contraintes plus avancées.

## SECTION

## 2

## Historique de GNU Make

GNU `make` est une œuvre de Richard M. Stallman et Roland McGrath, conformément aux normes POSIX 2, ensemble de règles de standards UNIX. La version et l'auteur d'un utilitaire `Make` peut changer d'une plate-forme à l'autre, mais la syntaxe du `Makefile` est standard et le résultat est le même. La principale raison pour ceci est que le nom du compilateur n'est pas toujours le même et certains paramètres peuvent varier. Un simple script `shell` n'est pas efficace pour compiler, puisque le type de `shell` est loin d'être standard sous Unix (`ash`, `csh`, `ksh`, `bash`, etc.).

## Les deux étapes de la création d'un programme

Il y a typiquement deux étapes lors de la construction d'un programme (page 26) : la transformation en code machine des instructions, la *compilation*, et la création des liens entre les fonctions définies dans les différents fichiers sources, l'*édition de liens*.

Lors de la première étape, des fichiers objets (\*.o) sont générés. Ils correspondent à la conversion des fichiers sources C en langage machine, mais ils ne peuvent être exécutés en l'état puisque les liens entre les fonctions des fichiers impliqués n'ont pas encore été créés. Par exemple, un appel à la fonction `printf` peut y être inclus, mais les instructions de cette fonction ne sont pas elles-mêmes incluses. Si une erreur se produit lors de cette étape, c'est dans la majorité des cas une erreur de syntaxe dans le code source.

La deuxième étape consiste à faire le lien entre les fonctions définies (dans les différents objets) et leurs appels, pour ensuite générer le fichier exécutable final. C'est à cette étape que l'on retrouve les messages d'erreurs les plus frustrants. Le nom des fonctions, les passages de paramètres et tous les autres détails qui sont relatifs à la communication entre les fonctions sont vérifiés ici.

Lorsqu'un programme conséquent est construit, le débogage afférent est plus efficace si on comprend bien les différentes étapes impliquées dans cette construction. `make` permet de décomposer plus facilement le travail de construction et ainsi d'accéder à des détails très utiles. Il permet par ailleurs d'optimiser les étapes nécessaires, et ainsi le temps et les ressources utilisées.

## Exemple de construction de Makefile

L'idée principale de `make` est d'effectuer uniquement les étapes de construction nécessaires à la création d'un exécutable. Par exemple, si un seul fichier source a été modifié dans un programme composé de plusieurs fichiers, il suffit de recompiler ce fichier et d'effectuer l'édition de liens. Les autres fichiers sources n'ont pas besoin d'être recompilés.

Cet exemple montre la construction progressive d'un fichier Makefile pour un programme en C. À travers l'évolution de ce fichier, nous introduisons au fur et à mesure les différentes fonctionnalités utilisables dans de tels fichiers.

Une première version du fichier Makefile peut être :

```
1 programme: main.o
2     gcc -o programme main.o
3
4 main.o: main.c
5     gcc -c -Wall -ansi -pedantic main.c
```

Nous découvrons dans cet exemple la structure générale des règles qui vont permettre à `make` de comprendre quelle ressource il doit construire. La syntaxe de chaque règle est de la forme :

```
cible: liste de dépendances
<tabulation>première commande permettant de construire la cible
<tabulation>deuxième commande permettant de construire la cible
<tabulation>...
```



Attention, il est **indispensable** d'insérer un caractère de tabulation, obtenu avec la touche du même nom, devant les différentes commandes à exécuter pour reconstruire une cible. Si ce caractère est absent, le fichier `Makefile` ne peut pas être exploité normalement... Dans le détail, par rapport à l'exemple ci-dessus :

- On voit que la *cible* `main.o` dépend du fichier `main.c` (ligne n° 4). Si `main.c` a une date de dernière modification plus récente que `main.o`, `main.o` doit être reconstruit. Pour effectuer cette reconstruction, la commande associée (il peut y en avoir plusieurs, mais il n'y en a qu'une ici) sera exécutée. Cette commande correspond à une simple compilation du fichier `main.c` (ligne n° 5).
- On voit également que la cible `programme` dépend du fichier `main.o` (ligne n° 1). Avant d'examiner si `main.o` est plus récent que `programme`, `make` considère d'abord la règle dont la cible est `main.o` (règle décrite ci-dessus). En effet, `main.o` est le fichier dont dépend `programme`, mais c'est aussi la cible d'une règle particulière. Ainsi, en tenant compte des deux règles, si `main.c` est plus récent que `main.o`, `main.o` sera reconstruit et sera ainsi forcément plus récent que `programme`, qui sera donc reconstruit à son tour... Cette reconstruction sera effectuée à l'aide de la commande associée à la règle dont la cible est `programme`, c'est-à-dire l'édition de liens (ligne n° 2).
- Si `main.c` avait été la cible d'une troisième règle, cette troisième règle aurait également été implicitement utilisée lors de la considération de `main.o` et donc de `programme`. Comme ce n'est pas le cas, `make` se contente de considérer que `main.c` est juste un fichier.

Un simple appel à la commande `make` lance automatiquement l'analyse du fichier `Makefile`. En effet, la commande `make` recherche par défaut dans le répertoire courant un fichier de nom `makefile`, ou `Makefile` si elle ne le trouve pas.

Sans argument, `make` considère par défaut la première règle contenue dans le fichier `Makefile` et essaie de construire (au besoin) la cible correspondante. Il est également possible d'invoquer `make` en lui fournissant en paramètre la cible que l'on souhaite reconstruire (exemple : `make main.o`).

On peut faire évoluer le `Makefile` pour obtenir cette nouvelle version :

```
1 CC = gcc
2 OBJS = main.o
3
4 programme: $(OBJS)
5     $(CC) -o programme $(OBJS)
6
7 main.o: main.c
8     $(CC) -c -Wall -ansi -pedantic main.c
```

Dans cette nouvelle mouture, on cherche à factoriser les informations redondantes pour pouvoir les mettre à jour plus facilement en cas de changement. Cette factorisation peut être effectuée en introduisant des variables, généralement définies en préambule du fichier `Makefile`. C'est le cas en particulier de la variable `CC`, censée contenir le nom du compilateur à utiliser, et de la variable `OBJS`, censée contenir la liste des fichiers objets à considérer durant l'édition de liens. L'utilisation d'une variable définie se fait simplement en plaçant cette variable entre parenthèses et en la préfixant par `$`.

En poussant un peu plus loin la factorisation, on obtient :

```
1 CC      = gcc
2 OBJS    = main.o
3 CFLAGS  = -c -Wall -ansi -pedantic
4 LDFLAGS =
5 PGM     = programme
6
7 $(PGM): $(OBJS)
8     $(CC) -o $(PGM) $(OBJS) $(LDFLAGS)
9
```

```

10 main.o: main.c
11     $(CC) $(CFLAGS) main.c

```

Ici, on introduit les variables CFLAGS et LDFLAGS, contenant respectivement les paramètres liés à la compilation et à l'édition de liens. La variable LDFLAGS est pour le moment vide. Ça n'est pas grave, elle est définie et pourra être utilisée si des paramètres doivent être ajoutés pour les éditions de liens... Le nom du programme fait également l'objet d'une définition de variable.

On peut compléter un peu le Makefile pour considérer d'autres fichiers objets nécessaires à la construction du projet :

```

1 CC      = gcc
2 OBJS    = main.o module1.o module2.o
3 CFLAGS  = -c -Wall -ansi -pedantic
4 LDFLAGS =
5 PGM     = programme
6
7 $(PGM): $(OBJS)
8     $(CC) -o $(PGM) $(OBJS) $(LDFLAGS)
9
10 main.o: main.c module2.h
11     $(CC) $(CFLAGS) main.c
12
13 module1.o: module1.c module1.h
14     $(CC) $(CFLAGS) module1.c
15
16 module2.o: module2.c module2.h module1.h
17     $(CC) $(CFLAGS) module2.c

```

Deux modules ont été ajoutés. Les listes de dépendances ont été mises à jour pour refléter typiquement les inclusions de fichiers d'en-tête réalisées à l'aide de `#include`. C'est tout à fait logique : si un fichier `toto.c` inclut (*via* un `#include`) un fichier `yop.h`, c'est qu'il en dépend. Si le fichier `yop.h` vient à être modifié, il faut donc que le fichier `toto.c` soit recompilé. C'est ce qui est exprimé par les dépendances définies dans le Makefile...

Si le nombre de modules à considérer devient important, l'écriture de toutes les règles gérant les compilations peut devenir très fastidieuse... Il est alors possible d'écrire des *méta-règles* :

```

1 CC      = gcc
2 OBJS    = main.o module1.o module2.o
3 CFLAGS  = -c -Wall -ansi -pedantic
4 LDFLAGS =
5 PGM     = programme
6
7 $(PGM): $(OBJS)
8     $(CC) -o $(PGM) $(OBJS) $(LDFLAGS)
9
10 .c.o:
11     $(CC) $(CFLAGS) $<

```

C'est le cas de la méta-règle de la ligne n° 10 : elle indique comment tous les fichiers à extension `.c` peuvent être transformés en fichiers à extension `.o`. L'avantage est évident : si l'application contient 500 modules, la compilation de ces 500 modules peut être gérée avec cette seule méta-règle. Il y a cependant un inconvénient : les dépendances liées à l'inclusion des fichiers d'en-tête ne peuvent plus être

exprimées. À titre d'information, certains programmes, comme `makedepend`, peuvent calculer automatiquement ce type de dépendances...

La variable de la ligne n° 11 permet de désigner le fichier source à extension `.c`. D'une façon générale, les variables suivantes peuvent être utilisées lors de l'écriture de méta-règles :

- `$@` : le nom de la cible
- `$<` : le nom de la première dépendance
- `$^` : la liste des dépendances
- `$?` : la liste des dépendances plus récentes que la cible
- `$*` : le nom du fichier sans suffixe

Enfin, il est également possible de définir :

- Des règles comportant des cibles sans dépendance : lorsque ces cibles sont recherchées, les commandes correspondantes sont toujours exécutées... C'est en particulier pratique pour gérer des règles comme `clean`.
- Des règles sans commande associée : la règle fait alors office de simple alias. C'est particulièrement utile pour définir une règle comme `all`, censée définir tout ce qui doit être construit dans un programme. On place généralement cette règle `all` en premier (dans la liste de définition des règles) afin qu'elle soit automatiquement appelée lorsque `make` est invoqué sans paramètre.

On obtient ainsi :

```
1 CC      = gcc
2 OBJS    = main.o module1.o module2.o
3 CFLAGS  = -c -Wall -ansi -pedantic
4 LDFLAGS =
5 PGM     = programme
6
7 all: $(PGM)
8
9 $(PGM): $(OBJS)
10         $(CC) -o $(PGM) $(OBJS) $(LDFLAGS)
11
12 .c.o:
13         $(CC) $(CFLAGS) $<
14
15 clean:
16         rm -f $(PGM)
17         rm -f $(OBJS)
18         rm -f *~
```

Enfin, comme précisé ci-dessus, un programme comme `makedepend` peut générer automatiquement les dépendances d'un ensemble de sources C dans un `Makefile`. Cela permet de cumuler la puissance des méta-règles à l'intérêt des dépendances. Il est alors utile d'ajouter une règle pour appeler plus facilement `makedepend`. Ce dernier ajoute automatiquement les dépendances calculées en fin de `Makefile` (à partir de la ligne n° 22 — incluse — dans l'exemple ci-dessous). Cette dernière partie ne doit alors pas être éditée manuellement naturellement.

```
1 CC      = gcc
2 OBJS    = main.o module1.o module2.o
3 CFLAGS  = -c -Wall -ansi -pedantic
4 LDFLAGS =
5 PGM     = programme
6
7 all: $(PGM)
8
```

```

9 $(PGM): $(OBJS)
10     $(CC) -o $(PGM) $(OBJS) $(LDFLAGS)
11
12 .c.o:
13     $(CC) $(CFLAGS) $<
14
15 depend:
16     makedepend *.c
17
18 clean:
19     rm -f $(PGM)
20     rm -f $(OBJS)
21     rm -f *~
22 # DO NOT DELETE
23
24 main.o: /usr/include/stdio.h /usr/include/features.h
25 main.o: /usr/include/bits/predefs.h /usr/include/sys/cdefs.h
26 main.o: /usr/include/bits/wordsize.h /usr/include/gnu/stubs.h
27 main.o: /usr/include/gnu/stubs-32.h /usr/include/bits/types.h
28 main.o: /usr/include/bits/typesizes.h /usr/include/libio.h
29 main.o: /usr/include/_G_config.h /usr/include/wchar.h
30 main.o: /usr/include/bits/stdio_lim.h /usr/include/bits/sys_errlist.h
31 main.o: module2.h
32 module1.o: module1.h
33 module2.o: module2.h module1.h

```

## SECTION

# 5

## Définition des Makefile

Au travers des différents exemples du paragraphe précédent, on voit qu'il est possible de définir un fichier `Makefile` de plusieurs manières différentes, suivant que l'on souhaite apporter une solution *ad-hoc* ou générique à un problème donné. Suivant le contexte, le programme considéré, il convient donc de définir avec discernement les fichiers `Makefile`.

On peut toutefois considérer que, quel que soit le problème considéré, la définition de variables telles que `CC`, `OBJS` et `CFLAGS` est largement recommandée. Suivant le nombre de modules à prendre en compte, il faut ensuite voir s'il est intéressant de définir des méta-règles de compilation (qui peuvent alors être utilisées conjointement avec `makedepend` pour la gestion des dépendances comme précisé plus haut)...

# Les pointeurs

## SECTION

## 1

## Une explication (\*, & et NULL)

Quelle est la signification d'un pointeur ? À chaque variable est associée une adresse mémoire, à laquelle est inscrit le contenu de cette variable. Un pointeur est simplement une variable qui contient cette adresse mémoire. Ainsi, un pointeur `ptr` sur un réel contient l'adresse d'une case mémoire dans laquelle il y a un réel, `r` par exemple. On dit alors que `ptr` *pointe* sur `r`.

La définition d'un pointeur se fait à l'aide de l'opérateur `*`, en précisant le type du contenu de la cellule pointée. Voici quelques exemples de définition :

```
1 int    *ptr_n;    /* ptr_n est un pointeur sur un entier */
2 float *ptr_r;    /* ptr_r est un pointeur sur un reel  */
```

L'association d'une variable existante à un pointeur se fait en affectant l'adresse de cette variable au pointeur. Cela se fait avec l'opérateur `&`, qui prend en argument une variable et retourne l'adresse mémoire de cette variable :

```
1 float r;
2
3 ptr_r = &r;
```

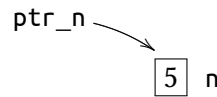
`ptr_r` vaut ainsi l'adresse de `r`. En d'autres termes, `ptr_r` pointe sur `r`. On accède au contenu de la case pointée par un pointeur en appliquant l'opérateur `*`, appelé *opérateur de déréférencement* dans ce contexte. Cela peut servir, par exemple, à affecter une valeur à la case pointée :

```
1 *ptr_r = 14.76;
```

Nous pouvons résumer tout cela avec l'exemple suivant :

```
1 int n;
2 int *ptr_n;
3
4 n = 5;
5 ptr_n = &n;
```

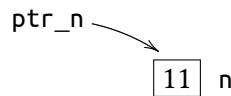
Ces instructions se comprennent ainsi : `n` est une variable de type `int` dont le contenu est 5. `ptr_n` est un pointeur sur un élément de type `int`. Son contenu est précisé par l'expression `ptr_n = &n` : `ptr_n` contient `&n`, l'adresse du contenu de la variable `n`. Cela revient à dire que `ptr_n` pointe sur `n`. À ce stade de l'exécution, la valeur de `*ptr_n` est donc 5.



Voici un autre exemple d'affectation *via* un pointeur :

```
1 *ptr_n = *ptr_n + 6;
```

Ici, nous ajoutons 6 dans la case pointée par `ptr_n`, *i.e.* à la variable `n`. Cette instruction a exactement le même effet que `n = *ptr_n + 6;`, `*ptr_n = n + 6;` ou encore `n = n + 6;`. Dans tous les cas, `n` prend la valeur  $5 + 6 = 11$ , ainsi que `*ptr_n`.



On peut initialiser, ou modifier à n'importe quel moment, un pointeur en lui affectant la valeur spéciale `NULL` <sup>❶</sup>. Cette valeur indique conventionnellement que le pointeur n'est pas valide et qu'il ne doit donc pas être déréférencé :

```
1 int *ptr;  
2  
3 ptr = NULL;
```

Dans ce cas, aucune adresse mémoire n'est associée à `ptr`. L'appel `*ptr` est alors prohibé car il provoque une erreur dans ce contexte. Nous verrons au chapitre suivant que l'initialisation est importante pour la construction des listes chaînées.

## SECTION

# 2 Arithmétique des pointeurs

Il peut paraître surprenant d'associer un type de données aux pointeurs. En effet, quel que soit le type de données pointé, un pointeur correspond toujours à une adresse, donc à un entier... Il y a cependant un intérêt non négligeable à utiliser des pointeurs typés : celui de pouvoir utiliser ce que l'on qualifie d'*arithmétique des pointeurs*.

En effet, certaines opérations arithmétiques sont autorisées sur les pointeurs et d'une manière générale sur les adresses mémoire : l'addition et la soustraction. Ces opérations se comportent cependant de manière particulière dans ce contexte. Ainsi, si on additionne un entier à un pointeur (ou à une adresse), cet entier est **automatiquement** et **implicitement** multiplié par la taille en octets du type de données pointé. Considérons par exemple le code suivant :

<sup>❶</sup> L'inclusion d'un *header* tel que `stdio.h` ou `stdlib.h` est nécessaire pour pouvoir utiliser `NULL`.

```

1  int main()
2  {
3      int i;
4      int *pi;
5
6      pi = &i;
7      pi++;
8
9      return 0;
10 }

```

Au cours d'une exécution, le pointeur `pi` peut (par exemple) prendre la valeur `0xB00000`, adresse de la variable `i`, en ligne 6. À la ligne suivante, `pi` est incrémenté : sa valeur ne va pas passer à `0xB00001`, mais à `0xB00004`, si la taille d'un entier (de type `int`) est de 4 octets sur la plate-forme utilisée...

#### SECTION

## 3

## Les pointeurs et les tableaux

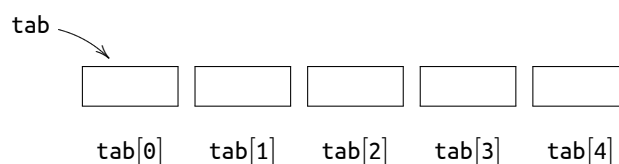
Les pointeurs sont omniprésents en C et il arrive même qu'on en manipule sans s'en rendre compte. C'est également le cas pour certaines adresses mémoire. C'est ce qui se produit quand on définit un tableau et qu'on utilise l'identificateur correspondant sans l'opérateur `[]`. Que passe-t-il lors de la déclaration de `tab`, dans l'exemple ci-dessous ?

```

1  int tab[5];

```

Il y a réservation d'un bloc mémoire contigu pour cinq variables de type `int`. Formellement, `tab` renvoie l'adresse du premier élément du tableau (page 21)<sup>②</sup> En considérant l'arithmétique des pointeurs, l'adresse du *i*<sup>e</sup> élément est `tab + i`, c'est-à-dire que `*(tab + i)` est exactement `tab[i]`.



Cette gestion de la mémoire est toutefois transparente pour le programmeur. Il est en effet toujours possible de manipuler les tableaux en C en ignorant la structure sous-jacente. L'opérateur `[]` permet d'accéder directement à chaque élément du tableau.

On peut également préférer mettre à profit la représentation à partir des pointeurs, quand cela peut simplifier l'écriture du programme. Ainsi, les instructions suivantes :

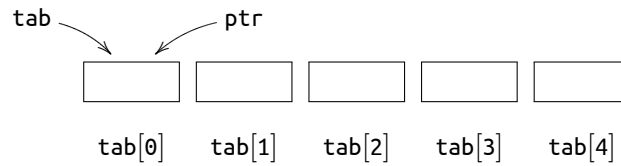
```

1  int tab[5];
2  int *ptr;
3
4  ptr = tab;

```

<sup>②</sup> Attention, `tab` ne peut pas être qualifié pour autant de *pointeur*, même si son utilisation sous cette forme – sans l'opérateur `[]` – renvoie une adresse. Il ne correspond en effet pas à une variable : il est impossible en particulier de lui affecter une nouvelle valeur, donc d'écrire une instruction du type `tab = &a` (`a` étant une variable entière).

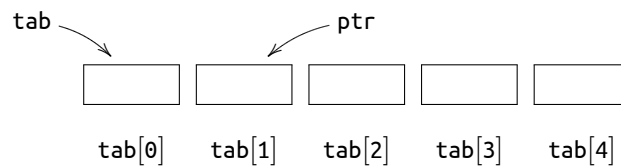
sont tout à fait valides. En voici la représentation au niveau de la mémoire :



On peut ensuite incrémenter le pointeur :

```
1 ptr++;
```

ce qui a pour effet de déplacer le pointeur ptr au bloc suivant, correspondant à l'élément d'indice 1 dans le tableau.



Remarque (à des fins de compréhension) : l'opérateur[] peut être utilisé sur un pointeur de la même manière que sur l'identificateur d'un tableau. Son comportement est identique dans les deux cas : on ajoute le déplacement entre crochets à l'adresse de base, selon l'arithmétique des pointeurs, et on dé-référence ensuite. L'exemple suivant affiche ainsi bien la valeur « 3 » après compilation et exécution.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int tab[5];
6     int *p;
7
8     p = tab;
9     p[1] = 3;
10
11     printf("%d\n", tab[1]);
12
13     return 0;
14 }
```

Pour autant, au niveau des conventions d'écriture de programmes, on évite généralement d'utiliser l'opérateur [] avec des pointeurs, car cela ne facilite pas la lisibilité des programmes. On préfère ainsi confiner l'usage de cet opérateur à la manipulation de tableaux.

**Exercice 10** Quelles sont les valeurs des variables après l'exécution du programme ci-dessous ? Pour cela, vous préciserez la valeur des variables à chaque étape du programme.

```
1 int main()
2 {
3     int t[2];
```



```

4   int a, *b, *c;
5
6   t[0] = 3;
7   t[1] = 5;
8   a = 10;
9   b = t;
10  c = &a;
11  *c = *b;
12  a++;
13  b++;
14  *b = 2 * a;
15
16  return 0;
17 }

```

**Exercice 11** *Quel est le résultat du programme suivant ?*

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main()
5  {
6      char x[10];
7      char *pa, *pb;
8
9      strcpy(x, "235849617");
10     pa = x;
11     pb = pa + 3;
12     printf("%s \n", pb);
13
14     return 0;
15 }

```

**Remarque :** Une chaîne de caractères en C se termine toujours par le caractère `'\0'` qui est appelé le *caractère de fin de chaîne*. L'absence du caractère de fin de chaîne provoque des erreurs graves, même avec un simple `printf`, qui va afficher une chaîne depuis son premier caractère (dont l'adresse lui est passée en paramètre) jusqu'à ce caractère de fin de chaîne (qui, s'il est omis, peut être rencontré très loin dans la mémoire après la fin de l'emplacement mémoire réservé pour la chaîne).

## SECTION

# 4

## Le passage par adresse

### ① Principe

Une première utilisation des pointeurs est de permettre de passer les paramètres d'une fonction *par adresse* et de pouvoir ainsi modifier les valeurs des arguments en exécutant la fonction. Considérons l'exemple suivant avec cette définition (inefficace) d'une fonction `swap` :

```

1 void swap(int n, int m)
2 {
3     int tmp;
4
5     tmp = n;
6     n = m;
7     m = tmp;
8 }

```

L'intention de la fonction `swap` est manifestement d'échanger les valeurs passées en paramètre. Cependant, avec cette définition, `swap` échoue. Testons la avec les lignes suivantes :

```

1 int a, b;
2
3 a = 4;
4 b = 7;
5 swap(a, b);

```

On peut vérifier que les valeurs de `a` et `b` restent inchangées : `a` vaut toujours 4 et `b` vaut toujours 7. Comment expliquer cela ? L'appel `swap(a, b)` provoque la création de deux nouvelles cases mémoire pour les paramètres `n` et `m`. Ces deux cases sont initialisées par les valeurs de `a` et `b` respectivement. Dans le cours de la fonction, seules les cases correspondant à `n` et `m` sont utilisées. Les instructions de `swap` n'affectent donc pas `a` et `b`. À l'issue de l'exécution, il y a destruction de `n` et `m` et leur valeur est donc perdue.

Pour autoriser la modification des valeurs passées en paramètre, la solution est de passer en argument les *adresses* de `n` et `m`. De cette manière, la fonction `swap` accède directement au contenu des cases mémoire correspondant aux paramètres effectifs. Pour notre exemple, le prototype de la fonction `swap` devient :

```

1 void swap(int *ptr_n, int *ptr_m);

```

`ptr_n` et `ptr_m` sont deux pointeurs sur un entier. Dans le corps de `swap`, il faut ensuite veiller à ce que les opérations sur les valeurs pointées se fassent en déréférençant :

```

1 void swap(int *ptr_n, int *ptr_m)
2 {
3     int x;
4
5     x = *ptr_n;
6     *ptr_n = *ptr_m;
7     *ptr_m = x;
8 }

```

Dans la suite du programme, pour appliquer la fonction `swap`, on passe en paramètre les adresses des variables dont on souhaite échanger les valeurs :

```

1 int a, b;
2
3 a = 4;

```

```

4  b = 7;
5  swap(&a, &b);

```

Après l'exécution de ces instructions, la valeur de `a` est 7 et celle de `b` est 4.

**Remarque :** Il existe des fonctions standards, dont le passage des paramètres se fait par adresse. Ainsi en est-il de `scanf` :

```

1  int n;
2
3  scanf("%d", &n);

```

La fonction lit au clavier une valeur et l'affecte à `n`, qui se trouve ainsi modifié.

**Remarque :** Le fait qu'un tableau soit en fait identifié par l'adresse de son élément de tête fait que le passage d'un tableau comme paramètre d'une fonction est systématiquement un passage par adresse, comme cela a déjà été vu ([page 17](#)).

**Exercice 12** *Quelles sont les valeurs des variables après exécution du programme suivant ?*

```

1  int f(int x, int *y, int z)
2  {
3      x = x + *y;
4      *y = *y - x;
5      z = x - *y;
6
7      return z;
8  }
9
10 int main()
11 {
12     int a, b, d, e;
13     int *c;
14
15     a = 5;
16     b = 9;
17     c = &e;
18     *c = 23;
19     d = f(a, &b, *c);
20
21     return 0;
22 }

```

## 2 Fonctions renvoyant plusieurs résultats

En C, comme dans la majeure partie des langages de programmation, les fonctions ne peuvent renvoyer qu'un seul résultat. Il y a cependant des situations où on souhaite qu'une fonction puisse renvoyer plusieurs résultats. En utilisant les pointeurs, il est possible de simuler ce comportement. Il suffit dans ce

cas d'ajouter autant de paramètres que de résultats attendus, *en les passant par adresse*. La fonction appelée pourra ainsi, en plus du résultat qu'elle renvoie, utiliser ces paramètres pour fournir des résultats supplémentaires.

À titre d'illustration, voici un exemple d'une telle fonction et du programme principal appelant cette fonction :

```
1  #include <stdio.h>
2
3  int calcul(float a, float b, float *somme, float *diff, float *div)
4  {
5      *somme = a + b;
6      *diff = a - b;
7
8      if (b != 0) {
9          *div = a / b;
10         return 1;
11     }
12     else {
13         return 0;
14     }
15 }
16
17 int main()
18 {
19     float x , y, somme, difference, ratio;
20     int ok;
21
22     printf("Valeur de x : ");
23     scanf("%f", &x);
24     printf("Valeur de y : ");
25     scanf("%f", &y);
26
27     ok = calcul(x, y, &somme, &difference, &ratio);
28
29     printf("Somme : %f\n", somme);
30     printf("Différence : %f\n", difference);
31
32     if (ok)
33         printf("Ratio : %f\n", ratio);
34     else
35         printf("Calcul du ratio impossible...\n");
36
37     return 0;
38 }
```

## Gestion dynamique de la mémoire

Jusqu'ici, nous avons manipulé des pointeurs dont le contenu était l'adresse d'une variable existante. Il est également possible d'associer à un pointeur un nouvel espace mémoire. C'est l'*allocation dynamique de mémoire*, qui permet de créer des nouvelles variables au cours de l'exécution d'un programme.

Le principe est le suivant. Après avoir défini un pointeur, on demande au système un bloc mémoire grâce à une fonction d'allocation. Il y a ainsi une réservation de mémoire, à laquelle on accède par l'intermédiaire du pointeur. Lorsque cet espace mémoire n'est plus utile, on doit le détruire explicitement, en rendant au système la mémoire qui avait été allouée auparavant. C'est la *libération*.

### 1 Allocation : la fonction `malloc`, l'opérateur `sizeof`

L'allocation dynamique de mémoire se fait à l'aide de la fonction système `malloc`, définie dans la bibliothèque `stdlib.h`. En voici un squelette typique d'utilisation :

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 ...
4 T *ptr_t;
5
6 ptr_t = (T*) malloc(taille);
7 if (ptr_t == NULL) {
8     printf("Erreur d'allocation...\n");
9 }
```

Ces instructions définissent un pointeur sur une variable de type `T` puis lui alloue un bloc mémoire de taille `taille`. Plus précisément, le résultat de la fonction `malloc` est le suivant. Dans le cas où la mémoire est correctement allouée, `malloc` retourne l'adresse du bloc mémoire alloué. Il faut ensuite convertir cette adresse en un pointeur sur le bon type. C'est le rôle de l'expression `(T*)`, réalisant un *transtypage*, encore appelé *cast*. En cas d'échec, lorsque l'allocation est impossible, `malloc` retourne la valeur `NULL`.

L'utilisation de `malloc` est souvent accompagnée de celle de `sizeof`, qui prend comme argument un type, ou une variable, et retourne le nombre d'octets nécessaires au système pour stocker une variable de ce type. Ainsi, l'instruction :

```
1 ptr_t = (float *) malloc(sizeof(float));
```

permet d'allouer de la mémoire à une variable de type `float`. Les instructions suivantes permettent d'allouer de la mémoire à une variable de type `int`, pointée par `ptr_p`. Dans le cas où l'allocation est effectuée avec succès, on affecte ensuite la valeur 2.

Voici l'exemple complet d'utilisation de la fonction `malloc` :

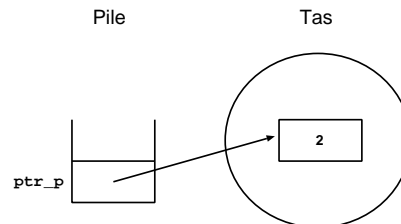
```
1 #include <stdlib.h>
2 #include <stdio.h>
3 ...
4 int *ptr_p;
5
6 ptr_p = (int *) malloc(sizeof(int));
7
```

```

8  if (ptr_p == NULL) {
9      printf("Erreur d'allocation...\n");
10     exit(1);
11 }
12
13 *ptr_p = 2;

```

À l'issue de l'exécution de cet exemple, on obtient ainsi un schéma mémoire similaire à celui présenté (Fig. 5.1).



**Figure 5.1** : Représentation mémoire après allocation dynamique.

*Commentaire / discussion* : allouer une seule variable de manière dynamique<sup>③</sup> ne présente généralement aucun intérêt. Ainsi, si l'exemple précédent devait être inclus dans un programme, il serait nettement plus simple et avantageux d'utiliser une allocation statique, ce qui reviendrait simplement à déclarer un entier... En revanche, cela est nettement plus utile pour un tableau dont on ne connaît pas la taille à la compilation. On peut en effet se servir de `malloc` pour effectuer une allocation de plusieurs variables à la fois. Dans l'exemple ci-dessous, la fonction effectue une allocation de mémoire pour un tableau de  $n$  entiers :

```

1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main()
5  {
6      int n;
7      int *tab;
8
9      printf("Taille de tableau souhaitée : ");
10     scanf("%d", &n);
11
12     tab = (int *) malloc(sizeof(int) * n);
13
14     if (tab == NULL) {
15         printf("Erreur d'allocation...\n");
16         exit(1);
17     }
18
19     /* Traitement sur le tableau */
20
21     return 0;
22 }

```

<sup>③</sup> On rappelle que « statique » fait référence à ce qui est décidé avant la compilation, alors que « dynamique » fait référence à ce qui est décidé à l'exécution.

Dans le traitement du tableau, les éléments du tableau peuvent être manipulés de la même manière qu'avec un tableau statique (page 55). Par exemple, `tab[0] = 3;`.

## 2 Allocation dynamique de structures

L'allocation dynamique de mémoire ne se restreint pas à la création de cellules contenant des valeurs de type simple, tels que `int` ou `float`. Il est également possible d'allouer de la mémoire à des objets de types composites. Considérons le type suivant, qui permet de modéliser une planète, grâce à son nom et son rayon :

```
1 #define MAX_NOM 20
2
3 typedef struct {
4     char nomplanet[MAX_NOM];
5     int rayon;
6 } Planete;
```

Le nombre d'octets nécessaires pour mémoriser une planète est `sizeof(Planete)`. On peut d'ailleurs connaître ce nombre en l'affichant `printf("%d", sizeof(Planete))`. La création d'une planète est obtenue ainsi :

```
1 #include <stdlib.h>
2 ...
3 Planete *ptr_planet;
4
5 ptr_planet = (Planete *) malloc(sizeof(Planete));
6 if (ptr_planet == NULL) {
7     printf("Erreur d'allocation...\n");
8     exit(1);
9 }
```

Les informations relatives à la nouvelle planète doivent être affectées à `(*ptr_planet).nomplanet`, pour le nom, et `(*ptr_planet).rayon`, pour le rayon. On peut également avantageusement utiliser l'opérateur binaire `->`, qui est équivalent à `(*...)`. Cela donne, par exemple :

```
1 strcpy(ptr_planet->nomplanet, "Uranus");
2 ptr_planet->rayon = 47600;
```

**Exercice 13** Écrire une fonction permettant de saisir les données relatives à une planète (on suppose que la mémoire est déjà attribuée) de prototype `void saisir(Planete *xp)`.

**Exercice 14** Écrire une fonction qui duplique une planète avec la fonction `malloc`. Le prototype est : `Planete *dupliquer(Planete p)`.

Nous pouvons essayer les deux fonctions précédentes avec le programme suivant :

```
1 int main()
2 {
3     Planete p, *ptr_p;
4 }
```

```

5   saisir(&p);
6   ptr_p = dupliquer(p);
7   printf("%s %d \n", p.nomplanet, p.rayon);
8   printf("%s %d \n", ptr_p->nomplanet, ptr_p->rayon);
9
10  return 0;
11 }

```

Les types des différentes références s'analysent comme suit :

Référence	Type
ptr_p	Planete *
*ptr_p	Planete
ptr_p->nomplanet	char [] = char *
ptr_p->rayon	int
p	Planete
&p	Planete *
p.nomplanet	char [] = char *
p.rayon	int

Enfin, voici comment allouer la mémoire pour un tableau de  $n$  structures :

```

1  #include <stdlib.h>
2
3  Planete *allouer_struct(int n)
4  {
5      Planete *ptr_planet;
6
7      ptr_planet = (Planete *) malloc(sizeof(Planete) * n);
8
9      if (ptr_planet == NULL) {
10         printf("Erreur d'allocation...\n");
11         exit(1);
12     }
13
14     return ptr_planet;
15 }

```

### 3 Libération : la fonction free

La fonction free permet de libérer de la mémoire qui a été allouée par une fonction système comme malloc. Voici un exemple typique d'utilisation :

```

1   Planete *p;
2
3   p = (Planete *) malloc(sizeof(Planete));
4   ...
5   free(p);

```

Attention, quelques informations **importantes** à propos de cette fonction free :

- les seules adresses mémoire acceptées sont celles qui ont été précédemment fournies par la fonction malloc,



- il n'est pas possible de faire des libérations partielles de mémoire : impossible en particulier de ne restituer que 100 Ko à partir d'une allocation de 500 ko<sup>④</sup>,
- dès qu'un bloc mémoire est rendu par l'intermédiaire de la fonction `free`, il n'appartient plus au programme mais au système : il n'est ainsi plus possible d'y accéder.

## SECTION

# 6

## Les périls de la programmation avec pointeurs

Le langage C permet avec les pointeurs un accès direct à la mémoire. Cela s'avère être très utile pour le passage de paramètres dans une fonction ou pour la définition de structures de données dynamiques, comme nous le verrons à la section suivante avec les listes chaînées. Mais cette liberté a une contre-partie : il est facile pour un programmeur distrait de faire des actions d'écriture illicites. De plus, la plupart des erreurs liées à une mauvaise manipulation des pointeurs **ne sont pas détectées par le compilateur**. Nous dressons ici une liste des erreurs les plus courantes.

Premier cas de figure : le programme s'exécute, mais son comportement est imprévisible. Typiquement, la cause est un oubli d'initialisation, ou une confusion dans une instruction entre le contenu du pointeur (adresse) et le contenu de la case pointée. Cette confusion peut résulter d'une mauvaise connaissance des priorités des opérateurs.

Une autre situation pathologique est celle où le programme interrompt son exécution brutalement. C'est souvent dû à une mauvaise gestion de la mémoire. Il faut à ce moment vérifier que pour chaque pointeur, la mémoire est correctement allouée (`malloc`) puis libérée (`free`). Une liste non exhaustive de malversations qui entraînent une fin fatale est présentée (page 66).

## SECTION

# 7

## valgrind

Les problèmes les plus embêtants rencontrés lors du développement d'un programme en C avec allocation dynamique de mémoire sont liés à des fautes de gestion de la mémoire (faute d'allocation, viol de protection d'un segment, ...) En effet, à l'exécution, le programme va afficher un message du type « *bus error* » ou « *segmentation fault* »... sans donner d'indication sur la source exacte du problème. On retombe ainsi sur les mêmes problèmes que ceux évoqués (page 32).

Là encore, l'outil `valgrind` permet la détection et le diagnostic des erreurs de gestion de mémoire. Lorsqu'un programme est exécuté sous la supervision de `valgrind`, nous avons vu que toutes les lectures et écritures réalisées en mémoire sont tracées. C'est également le cas de tous les appels de fonctions relatives à la gestion de la mémoire dynamique (`malloc`, `calloc`, `free`...), qui sont interceptés.

Grâce à cela, `valgrind` peut détecter des problèmes comme l'utilisation de zones de mémoire non-allouées, l'accès à des zones de mémoire déjà libérées (suite à un `free()`), le débordement de la zone allouée, l'accès à des zones de mémoire protégées...

Pour rappel, pour utiliser cet outil, il faut :

1. Compiler le programme à tester avec l'option `-g`, permettant de générer des informations de débogage (et surtout de faire le lien des erreurs avec la ligne correspondante dans le fichier source) (page 28).

<sup>④</sup> Ainsi, si seuls 400 Ko de mémoire deviennent requis sur un bloc alloué de 500 Ko, la bonne manipulation consiste à faire une nouvelle allocation de 400 Ko, de recopier les 400 Ko de données utilisées de l'ancien bloc vers le nouveau et de libérer l'ancien bloc de 500 Ko.

référence à un bloc non alloué	int *a; int i = *a;
référence à la valeur d'un pointeur non initialisé	int *a = (int *) malloc(sizeof(int)); int i = *a;
référence à un pointeur sur une variable libérée	int i; int *a = (int *) malloc(sizeof(int)); ... free(a); i = *a;
libérer un pointeur non alloué	int *a; free(a);
libérer une variable allouée statiquement	int a[10]; free(a);
libérer deux fois le même pointeur	free(a); free(a);
oublier de libérer la mémoire allouée	int main() { int *a = (int *) malloc(sizeof(int)); }
déclarer un pointeur sur une variable locale	int *a; int i; { int b; a = &b; } i = *a;

**Table 5.1** : Tableau des erreurs fréquentes liées aux pointeurs.

2. Encapsuler l'exécution du programme dans `valgrind` au moyen de la ligne de commande :

`valgrind ./executable options de l'exécutable`.

À titre d'exemple, considérons le programme suivant, volontairement truffé d'erreurs à ne pas faire en C avec les pointeurs :

```

1  #include <stdlib.h>
2
3  int main() {
4      int *q = (int *) 0x100000;
5      void *p;
6
7      p = malloc(10);
8      free(q);
9      (*q)++;
10
11     return 0;
12 }
```

Ce programme compile sans problème, même avec les options de compilation préconisées dans ce polycopié. Lors de l'exécution, le résultat ne se fait pas attendre :

```

1  royal 56 : gcc -ansi -pedantic -Wall testvalg.c -o test
2  royal 57 : ./test
```

```
3 zsh: segmentation fault (core dumped) ./test
```

Naturellement, compte-tenu de la nature du programme, il n'est pas étonnant d'obtenir ce résultat. Le point important à noter ici est qu'*aucune indication n'est donnée sur la ou les lignes incriminées...* Ici, le programme compte une douzaine de lignes, mais le même résultat pourrait être obtenu avec un programme comptant plusieurs dizaines de milliers de lignes... Sans outil supplémentaire, il est très difficile dans ces conditions d'identifier les instructions posant problème. Le même exemple lancé sous la supervision de valgrind donne :

```
1 royal 59 : gcc -ansi -pedantic -Wall testvalg.c -g -o test
2 royal 60 : valgrind ./test
3 ==8632== Memcheck, a memory error detector
4 ==8632== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
5 ==8632== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for
   copyright info
6 ==8632== Command: ./test
7 ==8632==
8 ==8632== Invalid free() / delete / delete[]
9 ==8632==     at 0x40257ED: free (vg_replace_malloc.c:366)
10 ==8632==     by 0x8048420: main (testvalg.c:8)
11 ==8632== Address 0x10000 is not stack'd, malloc'd or (recently) free'd
12 ==8632==
13 ==8632== Invalid read of size 4
14 ==8632==     at 0x8048425: main (testvalg.c:9)
15 ==8632== Address 0x10000 is not stack'd, malloc'd or (recently) free'd
16 ==8632==
17 ==8632==
18 ==8632== Process terminating with default action of signal 11 (SIGSEGV):
   dumping core
19 ==8632== Access not within mapped region at address 0x10000
20 ==8632==     at 0x8048425: main (testvalg.c:9)
21 ==8632== If you believe this happened as a result of a stack
22 ==8632== overflow in your program's main thread (unlikely but
23 ==8632== possible), you can try to increase the size of the
24 ==8632== main thread stack using the --main-stacksize= flag.
25 ==8632== The main thread stack size used in this run was 8388608.
26 ==8632==
27 ==8632== HEAP SUMMARY:
28 ==8632==     in use at exit: 10 bytes in 1 blocks
29 ==8632==   total heap usage: 1 allocs, 1 frees, 10 bytes allocated
30 ==8632==
31 ==8632== LEAK SUMMARY:
32 ==8632==     definitely lost: 0 bytes in 0 blocks
33 ==8632==     indirectly lost: 0 bytes in 0 blocks
34 ==8632==     possibly lost: 0 bytes in 0 blocks
35 ==8632==     still reachable: 10 bytes in 1 blocks
36 ==8632==           suppressed: 0 bytes in 0 blocks
37 ==8632== Rerun with --leak-check=full to see details of leaked memory
38 ==8632==
39 ==8632== For counts of detected and suppressed errors, rerun with: -v
40 ==8632== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 11 from 6)
41 zsh: segmentation fault  valgrind ./test
```

On constate que `valgrind` dresse un rapport complet des erreurs *dynamiques*, c'est-à-dire rencontrées durant l'exécution du programme :

- l'usage invalide de la fonction `free` fait en ligne n° 8 du programme,
- une tentative invalide de lecture en mémoire à l'adresse `0x10000` en ligne n° 9 du programme,
- le fait qu'un bloc de 10 octets reste alloué à la fin du programme et n'a pas été restitué au système...

`valgrind` dispose de beaucoup d'autres options qui vous peuvent être utiles. Certaines sont d'ailleurs signalées directement dans la trace d'exécution présentée ci-dessus, faisant ressortir pour chaque bloc mémoire non restitué où l'allocation mémoire correspondante a eu lieu (ligne n° 30 de la trace ci-dessous). D'autres sont accessibles, comme d'habitude sous Linux, grâce au manuel en ligne (`man valgrind`).

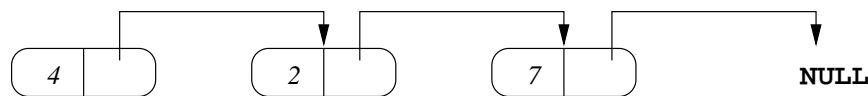
```
1 royal 67 : valgrind --leak-check=yes --show-reachable=yes --track-origins
  =yes ./test
2 ==8711== Memcheck, a memory error detector
3 ==8711== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
4 ==8711== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for
  copyright info
5 ==8711== Command: ./test
6 ==8711==
7 ==8711== Invalid free() / delete / delete[]
8 ==8711==    at 0x40257ED: free (vg_replace_malloc.c:366)
9 ==8711==    by 0x8048420: main (testvalg.c:8)
10 ==8711== Address 0x10000 is not stack'd, malloc'd or (recently) free'd
11 ==8711==
12 ==8711== Invalid read of size 4
13 ==8711==    at 0x8048425: main (testvalg.c:9)
14 ==8711== Address 0x10000 is not stack'd, malloc'd or (recently) free'd
15 ==8711==
16 ==8711==
17 ==8711== Process terminating with default action of signal 11 (SIGSEGV):
  dumping core
18 ==8711== Access not within mapped region at address 0x10000
19 ==8711==    at 0x8048425: main (testvalg.c:9)
20 ==8711== If you believe this happened as a result of a stack
21 ==8711== overflow in your program's main thread (unlikely but
22 ==8711== possible), you can try to increase the size of the
23 ==8711== main thread stack using the --main-stacksize= flag.
24 ==8711== The main thread stack size used in this run was 8388608.
25 ==8711==
26 ==8711== HEAP SUMMARY:
27 ==8711==    in use at exit: 10 bytes in 1 blocks
28 ==8711== total heap usage: 1 allocs, 1 frees, 10 bytes allocated
29 ==8711==
30 ==8711== 10 bytes in 1 blocks are still reachable in loss record 1 of 1
31 ==8711==    at 0x4025BD3: malloc (vg_replace_malloc.c:236)
32 ==8711==    by 0x8048410: main (testvalg.c:7)
33 ==8711==
34 ==8711== LEAK SUMMARY:
35 ==8711==    definitely lost: 0 bytes in 0 blocks
36 ==8711==    indirectly lost: 0 bytes in 0 blocks
37 ==8711==    possibly lost: 0 bytes in 0 blocks
38 ==8711==    still reachable: 10 bytes in 1 blocks
39 ==8711==    suppressed: 0 bytes in 0 blocks
```

```
40 ==8711==  
41 ==8711== For counts of detected and suppressed errors, rerun with: -v  
42 ==8711== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 11 from 6)  
43 zsh: segmentation fault  valgrind --leak-check=yes --show-reachable=yes  
    --track-origins=yes ./test
```



# Représentation des listes par une liste chaînée

Les pointeurs trouvent une application privilégiée avec la définition de listes chaînées. Lorsqu'on ne connaît pas *a priori* la longueur maximum d'une liste, la représentation par un tableau ne convient pas. Dans ce cas, il faut utiliser une structure de données dynamique, qui permet l'ajout et la suppression d'éléments au cours de l'exécution du programme. L'ajout d'un nouvel élément dans la liste se fait en allouant de la mémoire et en le raccordant au reste de la liste, comme un wagon peut être ajouté à un train. La suppression d'un élément se fait avec libération de la mémoire correspondante.



## SECTION

## 1

## Type

La définition en C d'un type pour une liste chaînée se fait de manière récursive, à l'aide de l'instruction `typedef struct`. Une cellule dans une liste est de type composite : un premier champ, disons `val`, correspond à la valeur portée par la cellule et un second champ, disons `suc`, indique la cellule successeur, sous la forme d'un pointeur sur une cellule. Malheureusement, cette intuition ne se traduit pas directement en C. Une déclaration du type :

Déclaration intuitive (mais erronée) d'une structure de liste

```

1 typedef struct {
2     int val;
3     TypeCellule * suc;
4 } TypeCellule ;
  
```

n'est pas acceptée par le compilateur. Il faut recourir à un subterfuge, en décomposant la déclaration avec un nom pour la structure. Par exemple, le type d'une liste d'entiers s'écrit :

```

1 typedef struct TCell {
  
```

```

2   int val;
3   struct TCell *suc;
4 } TypeCellule ;
5
6 typedef struct TCell *Liste;

```

La dernière instruction est également équivalente à

```

1 typedef TypeCellule *Liste;

```

Le type `TypeCellule` est le type d'une cellule de liste. La liste est identifiée par sa tête, qui est un pointeur sur une cellule.

## SECTION

# 2

## Exemple d'application : une bibliothèque Liste

Nous allons construire les opérations de base de manipulation des listes et les regrouper sous la forme d'une bibliothèque `Liste`. Nous choisissons de représenter des listes d'entiers. Les quatre premières fonctions à concevoir dans ce cadre, `liste_initialiser()`, `liste_vide()`, `liste_teteinsérer()` et `liste_tetesupprimer()` permettent respectivement d'initialiser une liste (sans élément), de tester si une liste est vide, d'ajouter un élément en tête et de supprimer l'élément de tête.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "liste.h"
5
6 /* Initialisation d'une liste */
7 Liste liste_initialiser()
8 {
9     return(NULL);
10 }
11
12 /* Test si une liste est vide */
13 int liste_vide(Liste l)
14 {
15     return (l == NULL);
16 }
17
18 /* Ajout d'un élément en tête de liste */
19 Liste liste_teteinsérer(Liste l, int e)
20 {
21     Liste p;
22
23     p = (Liste) malloc(sizeof(TypeCellule));
24     if (p == NULL) {
25         printf("Allocation impossible...\n");
26         exit(1);
27     }
28     p->val = e;

```



```

29     p->suc = l;
30
31     return (p);
32 }
33
34 /* Suppression d'un élément en tête de liste */
35 Liste liste_tetesupprimer(Liste l)
36 {
37     Liste p = NULL;
38
39     if (!liste_vide(l)) {
40         p = l->suc;
41         free(l);
42     }
43
44     return (p) ;
45 }

```

**Exercice 15** Compléter la bibliothèque *Liste* avec les fonctions suivantes :

- affichage des éléments : `void liste_afficher(Liste);`
- insertion d'un élément en queue de liste : `Liste liste_queueinsérer(Liste, int);`
- suppression d'un élément en queue de liste : `Liste liste_queuesupprimer(Liste);`
- recherche si l'élément *a* est dans la liste. La fonction retournera vrai si *a* est bien dans la liste, ainsi que le rang<sup>❶</sup> de *a* dans la liste. Sinon, la fonction retournera faux. Le prototype de la fonction à écrire est `int liste_rechercher(Liste, int, int *)`<sup>❷</sup>.

**Exercice 16** Écrire la fonction `int main()` qui permet de tester toutes ces fonctions. Comme dans les exercices précédents, prévoir la présentation d'un menu regroupant toutes ces fonctions et permettant ainsi une interaction avec l'utilisateur.

❶ Dans ce contexte, le *rang* est la position de l'élément trouvé dans la liste, sachant que le premier élément de cette liste *a* comme rang 0.

❷ Le troisième paramètre de cette fonction est donc un résultat supplémentaire attendu de la fonction. Il est passé par adresse et non pas par valeur (page 59).



# Les signaux

## SECTION

## 1

## Introduction

Un *signal* est le type d'interruption logicielle qui est envoyée aux processus par le système pour les informer sur des événements anormaux se déroulant dans leur environnement (violation de mémoire, erreur dans les entrées/sorties...) Il permet également aux processus de communiquer entre eux. À une exception près (le signal SIGKILL), un signal peut être traité de trois manières différentes :

- *Par un comportement par défaut.* Ce comportement dépend alors du type du signal et peut mener, suivant les cas, à la suspension ou la terminaison du processus recevant le signal par exemple.
- *Il peut être ignoré.* Par exemple, le programme peut ignorer les interruptions clavier générées par l'utilisateur (c'est ce qui se passe lorsqu'un processus est lancé en arrière-plan).
- *Il peut être pris en compte par une action spécifique.* Dans ce cas, à la réception d'un signal, l'exécution d'un processus est détournée vers une procédure spécifiée par l'utilisateur, puis reprend là où elle a été interrompue.

Les signaux changent le déroulement d'un programme de manière *asynchrone*, c'est-à-dire à n'importe quel instant lors de l'exécution de ce programme, et donc pas forcément lors d'une instruction permettant d'attendre une communication externe<sup>❶</sup>. Incidemment, les signaux se démarquent des autres formes de communications où les programmes doivent explicitement demander à recevoir les messages externes en attente, par exemple en faisant des lectures dans un tube (*pipe*).

Les signaux véhiculent peu d'information (uniquement leur type). Un processus recevant un signal recherche alors dans une table de correspondance l'action à entreprendre en fonction du signal reçu. Un des intérêts principaux des signaux réside dans le fait qu'il est possible de redéfinir l'action à entreprendre pour un signal donné.

## SECTION

## 2

## Liste des signaux d'un système

Les signaux sont identifiés au niveau du système par un nombre entier. Sous un système Unix, le fichier `/usr/include/signal.h` contient la liste des signaux accessibles. Chaque signal est caractérisé

<sup>❶</sup> Pour considérer une analogie sur les communications dans la vie courante, on pourrait dire qu'aller chercher son courrier dans sa boîte aux lettres est un processus *synchrone*, s'effectuant à un moment choisi. Par opposition, recevoir un appel téléphonique est un processus *asynchrone*, pouvant intervenir à n'importe quel moment, indépendamment des actions réalisées à ce moment précis.

par un mnémonique. Le numéro associé à un signal donné peut varier suivant le système Unix. On utilisera donc toujours le mnémonique pour désigner un signal particulier. La liste des signaux usuels est donnée ci-dessous <sup>②</sup> (liste non exhaustive) :

- SIGHUP (1) Coupure : signal émis aux processus associés à un terminal lorsque celui-ci se déconnecte. Il est aussi émis à chaque processus d'un groupe dont le chef se termine.
- SIGINT (2) Interruption : signal émis aux processus du terminal lorsqu'on frappe la touche d'interruption (INTR ou CTRL-C) de son clavier.
- SIGQUIT (3)\* Abandon : *idem* avec la touche d'abandon (QUIT ou CTRL-D).
- SIGILL (4)\* Instruction illégale : signal émis à la détection d'une instruction illégale, au niveau matériel (exemple : lorsqu'un processus exécute une instruction flottante alors que l'ordinateur ne possède pas d'instruction flottante câblée).
- SIGTRAP (5)\* Piège de traçage : signal émis après chaque instruction en cas de traçage de processus (utilisation de la primitive `ptrace()`).
- SIGIOT (6)\* Piège d'instruction d'E/S : signal émis en cas de problème matériel. Ce signal est aussi appelé SIGABRT.
- SIGBUS (7)\* : signal émis en cas d'erreur sur le bus d'adressage.
- SIGFPE (8)\* : signal émis en cas d'erreur de calcul flottant, comme un nombre en virgule flottante de format illégal. Cela indique presque toujours une erreur de programmation.
- SIGKILL (9) Destruction : arme absolue pour tuer les processus. Ne peut être ni ignoré, ni intercepté (voir SIGTERM pour une mort plus douce).
- SIGUSR1 (10) Premier signal à la disposition de l'utilisateur : utilisé pour la communication inter-processus.
- SIGSEGV (11)\* : signal émis en cas de violation de la segmentation : tentative d'accès à une donnée en dehors du domaine d'adressage du processus actif (ce qui arrive typiquement si on essaie de déréférencer NULL par exemple).
- SIGUSR2 (12) Deuxième signal à la disposition de l'utilisateur : *idem* SIGUSR1.
- SIGPIPE (13) : écriture sur un pipe non ouvert en lecture.
- SIGALRM (14) Horloge : signal émis quand l'horloge d'un processus s'arrête. L'horloge est mise en marche par la primitive `alarm()`.
- SIGTERM (15) Terminaison logicielle : signal émis lors de la terminaison normale d'un processus. Il est également utilisé lors d'un arrêt du système pour mettre fin à tous les processus actifs.
- SIGCHLD (17) Mort d'un fils : signal envoyé au père à la terminaison d'un processus fils.
- SIGPWR (30) Réactivation sur panne d'alimentation.
- SIGSYS (31)\* : argument incorrect d'un appel système.

Certains signaux n'existent que sur certaines architectures et pour plus de portabilité, on peut écrire des programmes utilisant des signaux en appliquant les règles suivantes : éviter les signaux SIGIOT, SIGEMT, SIGBUS et SIGSEGV qui dépendent de l'implémentation. Il est correct de les intercepter pour imprimer un message, mais il ne faut pas essayer de leur attribuer une quelconque signification.

## SECTION

# 3

## Traitement des signaux

### ① La fonction `signal()`

Pour redéfinir l'action à entreprendre lors de la réception d'un signal à l'intérieur d'un processus, il faut utiliser la fonction `signal()`. Cette fonction a le prototype suivant :

```
1 #include <signal.h>
2
```

<sup>②</sup> Les signaux repérés par « \* » génèrent un fichier core sur le disque lorsqu'ils ne sont pas traités correctement.

```

3 typedef void (*sighandler_t)(int);
4 sighandler_t signal(int signum, sighandler_t handler);

```

Par rapport aux notions de base étudiées dans les premiers chapitres, la grande nouveauté réside ici dans la définition du type `sighandler_t`. En effet, cette syntaxe particulière introduit `sighandler_t` comme étant un pointeur de fonction attendant un entier en paramètre et ne renvoyant aucun résultat. Ainsi, en reprenant le prototype de la fonction `signal()`, on s'aperçoit qu'elle attend un tel pointeur comme second paramètre et qu'elle renvoie également un pointeur de ce type comme résultat.

Obtenir un pointeur de fonction en C est une opération très simple : il suffit d'utiliser *seul* l'identificateur d'une fonction respectant les types attendus (au niveau des paramètres comme du résultat renvoyé par cette fonction). Ainsi, il est possible d'appeler la fonction `signal()` comme dans l'exemple suivant :

```

1 #include <stdio.h>
2 #include <signal.h>
3
4 void traitement(int a)
5 {
6     printf("Valeur de a : %d\n", a);
7 }
8
9 int main()
10 {
11     signal(SIGUSR1, traitement);
12
13     while(1)          /* Mauvaise idée, mais fonctionne tout de même... */
14         ;
15
16     return 0;
17 }

```

Remarque : sur cet exemple, une boucle infinie est introduite en lignes 13–14. En effet, en C, l'instruction « ; » tout court représente l'instruction vide. Cet exemple est purement pédagogique : c'est en effet une très mauvaise idée d'introduire ce genre d'instructions dans un programme. Même avec une instruction vide, le processeur est monopolisé pour ne rien faire (!) et un simple examen d'un moniteur système fait apparaître que le processeur exécutant ce programme est occupé à 100 %... pour ne rien faire. Dans ce genre de situations, où on attend un événement extérieur (comme un signal) pour débloquer un processus, on préfère utiliser une fonction comme `pause()` (présentée [page 80](#)) qui présente l'avantage de ne pas monopoliser de ressource CPU.

Si on revient sur l'explication du fonctionnement de la fonction `signal()`, on peut faire les commentaires suivants :

- Le premier paramètre correspond au mnémonique du signal auquel on souhaite associer une nouvelle action, selon la table présentée [page 76](#).
- Le second paramètre peut prendre les valeurs suivantes :
  - `SIG_DFL` : ceci permet de rétablir l'action par défaut pour le signal (dépendant du signal considéré). La réception d'un signal par un processus entraîne alors la terminaison de ce processus, sauf pour `SIGCHLD` et `SIGPWR`, qui sont ignorés par défaut. Dans le cas de certains signaux, il y a création d'un fichier image core sur le disque.
  - `SIG_IGN` : ceci indique que le signal doit être ignoré : le processus est alors immunisé. On rappelle que le signal `SIGKILL` ne peut être ignoré.
  - Un pointeur sur une fonction : ceci implique la capture du signal. La fonction est appelée quand le signal arrive et le traitement du processus reprend où il a été interrompu après son exécution.

On ne peut procéder à un déroutement sur la réception d'un signal `SIGKILL` puisque ce signal n'est pas interceptable. L'entier passé en paramètre à la fonction dont on fournit un pointeur correspond au numéro du signal concerné.

- Enfin, la fonction `signal()` renvoie comme valeur de retour la valeur du pointeur de fonction qui était précédemment associée au traitement du signal concerné.

On voit ainsi qu'il est possible de modifier le comportement d'un processus à l'arrivée d'un signal donné. C'est ce qui se passe pour un certain nombre de processus standard : le shell, par exemple, à la réception d'un signal `SIGINT` affiche le prompt (et n'est donc pas interrompu).

**Attention** : la fonction `signal()` a des comportements différents suivant la famille de système d'exploitation sur laquelle elle est exécutée. Pour les systèmes UNIX, on discerne en particulier à ce niveau deux grandes familles :

- les systèmes *SVID* (*System V Interface Definition*), qui correspondent aux systèmes influencés par le système UNIX initialement développé par les laboratoires AT&T,
- les systèmes *BSD* (*Berkeley Software Distribution*), qui correspondent aux systèmes influencés par le système UNIX initialement développé par l'université Berkeley située en Californie.

En particulier, **sur les systèmes basés SVID**, si la fonction `signal()` est appelée en passant un pointeur de fonction et que le signal correspondant est envoyé, la fonction considérée sera naturellement appelée, *mais* le signal utilisé sera ensuite réassocié à son action par défaut. En clair : sur ces systèmes, lorsqu'on associe une fonction à un signal, cette fonction n'est valide qu'à la première émission de ce signal. Incidemment, si on souhaite associer de manière durable un signal à une fonction, la première instruction de cette fonction doit être un appel à la fonction `signal()` afin de perpétuer l'association entre le signal et la fonction.

D'un autre côté, **sur les systèmes basés BSD**, ce comportement n'existe pas. Ainsi, si une fonction est associée à un signal donné, elle sera naturellement appelée lors de la première émission de ce signal et, contrairement aux systèmes SVID, elle le restera pour les futures émissions de ce signal, sans avoir à réaliser d'action supplémentaire. Il est **important** de noter que l'implantation de la fonction `signal()` dans la bibliothèque C standard GNU est conforme à la philosophie BSD. **Incidemment, c'est donc le comportement qu'on obtient sous les systèmes GNU/Linux.**

## 2 La fonction `sigaction()`

La fonction `signal()` est historiquement la première implémentation permettant de gérer les associations entre des signaux et des traitements. Cependant, cette fonction souffre d'un certain nombre de problèmes :

- La fonction `signal()` ne bloque pas les autres signaux quand un signal est déjà en cours de traitement, ce qui peut mener à des exécutions non consistantes.
- Sous les systèmes SVID, la fonction `signal()` réinitialise l'action associée à un signal dès lors que ce signal est émis. Si une fonction était associée à un signal, l'association est perdue dès que le signal est émis. Ça n'est pas le cas sous les systèmes basés BSD et cela pose donc un problème de portabilité lorsqu'on souhaite écrire un programme C utilisant les signaux, que l'on souhaite pouvoir exécuter de manière similaire sur ces deux familles d'UNIX.

Pour éviter ces problèmes, une autre fonction permettant de gérer les traitements associés aux signaux a été définie : il s'agit de la fonction `sigaction()`. Cette fonction a le prototype suivant :

```
1 #include <signal.h>
2
3 int sigaction(int signum, struct sigaction *act,
4               struct sigaction *oldact);
```

Cette fonction utilise la structure `struct sigaction` :

```

1 struct sigaction {
2     void      (*sa_handler)(int);
3     void      (*sa_sigaction)(int, siginfo_t *, void *);
4     sigset_t   sa_mask;
5     int        sa_flags;
6     void      (*sa_restorer)(void);
7 };

```

Par rapport à la fonction `signal()`, cette nouvelle fonction présente quelques particularités, même si elle répond aux mêmes objectifs :

- le paramètre `act` de la fonction permet de définir la structure contenant les informations relatives au traitement à mettre en place,
- le paramètre `oldact` de la fonction, s'il est différent de `NULL`, permet de définir une structure qui sera remplie avec les anciennes informations relatives au traitement de ce signal (pratique si on souhaite les rétablir ensuite),
- le champ `sa_handler` permet de définir le pointeur de fonction à utiliser pour le traitement associé au signal,
- le champ `sa_flags` permet de définir finement le comportement de l'association faite par rapport à certains événements système (le processus devient zombie, il crée des fils...),
- le champ `sa_mask` permet de définir un ensemble de signaux qui seront bloqués durant le traitement de ce signal.

Le programme ci-dessous présente un exemple minimaliste d'utilisation de cette fonction (on prend bien soin de mettre à blanc les champs `sa_flags` et `sa_mask` qui ne sont pas utilisés) :

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <signal.h>
5
6 void handler(int signal)
7 {
8     printf("Je reçois le signal %d\n", signal);
9 }
10
11 int main()
12 {
13     struct sigaction action;
14
15     action.sa_handler = handler;
16     action.sa_flags = 0;
17     sigemptyset(&action.sa_mask);
18
19     sigaction(SIGINT, &action, NULL);
20     sigaction(SIGQUIT, &action, NULL);
21     sigaction(SIGTERM, &action, NULL);
22
23     pause();
24     return 0;
25 }

```

Attention, il semble difficile de compiler des programmes C utilisant la fonction `sigaction()` en respectant la norme ANSI. Il est ainsi préférable de ne pas utiliser les options de compilation relatifs à cette norme dans ce cas.

## 4 La fonction pause()

Cette primitive correspond à de l'attente pure, sans monopolisation de ressource CPU. Cette fonction a le prototype suivant :

```
1 #include <unistd.h>
2
3 int pause(void);
```

Elle ne fait rien, et n'attend rien de particulier. Cependant, puisque l'arrivée d'un signal interrompt toute primitive bloquée, on peut tout aussi bien considérer que la fonction `pause()` permet d'attendre un signal. Notons que le plus souvent, le signal que `pause()` attend est l'horloge de la fonction `alarm()`.

## 5 Émission d'un signal

### 1 La fonction kill()

Pour émettre un signal depuis un programme C, il faut utiliser la fonction `kill()` qui, contrairement à ce que son nom pourrait laisser penser, permet d'envoyer tout type de signal et pas seulement les signaux permettant de tuer d'autres processus. Cette fonction a le prototype suivant :

```
1 #include <signal.h>
2
3 int kill(pid_t pid, int sig);
```

Les deux paramètres attendus sont respectivement le `pid` du processus auquel on veut envoyer un signal et le numéro du signal considéré (*via* son mnémonique, c'est toujours plus portable). Cette fonction renvoie 0 si le signal a été envoyé et -1 sinon. Quelques particularités à propos de cette fonction :

- Si la valeur 0 est fournie comme numéro de signal, aucun signal n'est envoyé et la valeur de retour de la fonction `kill()` permet de savoir si le `pid` spécifié correspond ou pas à un numéro de processus.
- Si la valeur -1 est fournie comme `pid`, le signal est envoyé à tous les processus appartenant à l'utilisateur du programme incluant cet appel à `kill()`.

### 2 La fonction raise()

Les programmes souhaitant s'envoyer un signal à eux-mêmes peuvent également utiliser la fonction `raise()`, qui fonctionne comme la fonction `kill()`, mais pour laquelle il n'est pas utile de préciser le `pid`. En voici le prototype :

```
1 #include <signal.h>
2
3 int raise(int sig);
```



## Alarmes

La fonction `alarm()` permet de définir une horloge qui s'activera au bout d'un temps donné. Cette fonction a le prototype suivant :

```
1 #include <unistd.h>
2
3 unsigned int alarm(unsigned int seconds);
```

La valeur retournée par la fonction correspond au temps restant dans l'horloge précédemment définie, si elle n'est pas arrivée à terme, ou 0 si aucune horloge n'était précédemment définie. Techniquement, cette fonction permet l'envoi d'un signal `SIGALRM` au processus qui l'a appelée après un laps de temps passé en argument (en secondes). À l'appel de la fonction, l'horloge est initialisée grâce au paramètre fourni et est ensuite décrétementée toute les secondes jusqu'à 0. Si le paramètre fourni est nul, toute requête en cours est annulée. Cette fonction peut être utilisée, par exemple, pour forcer la lecture au clavier dans un délai donné. Le traitement du signal doit être prévu, sinon le processus est tué. Le programme suivant montre un exemple d'utilisation de cette fonction.

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <signal.h>
4
5 void it_horloge(int sig) /* Fonction exécutée à la réception de SIGALRM
   */
6 {
7     printf("Réception du signal %d : SIGALRM\n", sig);
8 }
9
10 int main()
11 {
12     unsigned sec ;
13
14     signal(SIGALRM, it_horloge);          /* Interception du signal */
15     printf("Alarme dans 5 secondes !\n");
16     sec = alarm(5);
17
18     printf("Valeur retournée par alarm : %d\n", sec);
19     printf("Le programme principal est en attente (CTRL-C pour l'arrêter)\n
20           ");
21     pause();
22
23     return 0;
24 }
```

**Exercice 17** Faire un programme permettant de réviser les tables de multiplication. Le programme comportera une boucle infinie posant des questions à ce sujet. Les opérandes devront être choisies au hasard (voir les fonctions `srandom()` et `random()`). Pour sortir du programme, l'utilisateur devra faire un `CTRL-C`, qui provoquera l'affichage du score obtenu. Le programme devra également s'arrêter si l'utilisateur met plus de 5 secondes à répondre à une question. L'usage des variables globales est recommandé pour cet exercice.



# Index

## Symboles

- `*`, 53
- `-Wall`, 29
- `-ansi`, 29
- `-pedantic`, 29
- `[]`, 11, 14, 19, 55, 56
- `#define`, 39
- `#endif`, 41
- `#ifndef`, 41
- `#include`, 27, 39
- `&`, 21, 53
- `\0`, 16, 57

## A

- adresses mémoire, 21
- arithmétique des pointeurs, 54

## B

- bibliothèque, 37
  - implantation, 37
- booléen, 12

## C

- C++, 5
- C11, 6
- C89, 6
- C99, 6
- `calloc()`, 65
- `char`, 12
- chaînes de caractères, 12, 16
  - calcul de longueur, 17
  - comparaison, 17
  - constante, 17
  - copie, 16
- compilation, 10, 26–28, 48
  - conditionnelle, 41

## D

- `double`, 12

## E

- édition de liens, 26, 28, 48

## F

- fichier d'en-tête, 37
- `float`, 12
- fonctions, 13, 17
  - appels, 18
  - déclaration, 13
  - définition, 13
  - et chaînes de caractères, 20
  - et tableaux, 19
  - passage par adresse, 57
  - portée, 14
  - renvoyant plusieurs résultats, 59
- `free()`, 64, 65

## G

- `gcc`, 10, 28
- `gdb`, 30
- gestion dynamique de la mémoire, 61

## H

- historique, 5

## I

- `int`, 12

## J

- Java, 5

## L

- liste chaînée, 71

## M

- `main()`, 10
- `make`, 47, 48
- `Makefile`, 47, 52
- `malloc()`, 61, 65
- module, 26, 37
  - implantation, 37

## N

- `nm`, 28
- normes, 5
  - C11, 6

C89, 6

C99, 6

`NULL`, 54

## P

pointeurs, 53

arithmétique, 54

et tableaux, 55

portée, 14

`printf()`, 22

programme principal, 10

préprocesseur, 27, 39

## S

`scanf()`, 22, 23

signal, 75

signaux, 75

`sizeof()`, 12, 61

`struct`, 10

structures, 10

## T

tableaux, 14

définition, 14

types de données, 12

définition, 12

## V

`valgrind`, 32, 65

variables, 12

définition, 12

portée, 14

`void`, 12