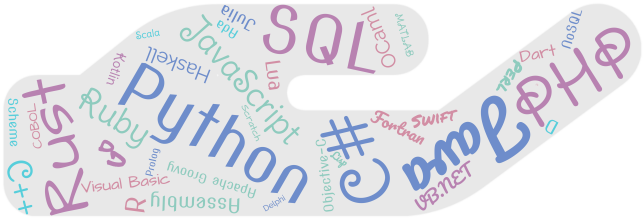
Programmation en langage C

Notes de cours et exercices



Thomas Place, Marc Zeitoun

Version du 9 octobre 2025

Ce document présente les notions abordées dans le cours de l'Université de Bordeaux :

Programmation C, 2025–2026,
L2 *Mathématiques-informatique*,
CMI *Optimisation Mathématique et Algorithmes*,
CMI *Ingénierie de la Statistique et Informatique*.

Il suppose une connaissance de base de la programmation impérative, concernant par exemple les notions de variables, de structures de contrôle (tests et boucles), et de fonctions.

Il n'a pas pour but de remplacer des documents de référence, comme la norme du langage ou les documentations des divers logiciels utilisés dans ce cours (éditeur, compilateur, débogueur, etc.), ni les nombreux excellents livres sur le langage C.

Ce document met l'accent sur ces principes fondamentaux de ce langage. Ces principes sont relativement simples. De plus, les maîtriser permet de comprendre plus facilement le fonctionnement interne d'autres langages, car le langage C demande au programmeur ou à la programmeuse de manipuler directement la mémoire utilisée.

A contrario, de nombreux détails rendent le langage C compliqué. Ces détails ont souvent des raisons historiques, et sont en général sans importance quand on souhaite apprendre à programmer en C. Bien sûr, il faut avoir connaissance de certains de ces points et conscience des pièges potentiels, mais ce document évite de s'étendre dessus. Vous pouvez faire de même dans un premier temps, de façon à vous concentrer sur les points fondamentaux 😉.

En résumé, ce document a pour objectif de présenter le plus simplement possible les aspects importants du langage. Il contient également des exercices. Il est conçu pour *accompagner la pratique* de la programmation, indispensable dans tout apprentissage.

Conventions typographiques

Les définitions les plus importantes sont indiquées en *cette couleur et cette police*. Le texte contient plusieurs hyperliens.

- Les termes renvoyant à une de ces définitions sont indiqués en de cette façon (comme le mot norme ci-dessus).
- Les hyperliens externes sont souvent indiqués par l'icône ↗.

Table des matières

1 Introduction au langage C

1.1 Configuration avant de commencer

Un programme C est écrit dans un ou plusieurs fichiers texte avec l'extension « `.c` ». Lors de la programmation, il est fortement recommandé d'utiliser un *éditeur* tel que *VSCode*, l'éditeur le plus utilisé en licence à l'Université de Bordeaux. Il y en a bien sûr d'autres, comme par exemple GNU Emacs. À l'inverse, évitez les logiciels comme `nano`, qui ne permet que de saisir du texte. Un éditeur comme VSCode est appelé environnement de développement intégré (IDE, *Integrated Development Environment*), car il permet non seulement de taper un programme, mais aussi d'identifier certains bogues du code avant même la compilation. De plus, il colorie divers éléments et reformate automatiquement le code, ce qui améliore considérablement sa lisibilité. Enfin, il intègre des fonctionnalités comme la possibilité de lancer un débogueur.

Configurer votre éditeur

Pour utiliser un éditeur, vous devez d'abord le configurer. Si vous utilisez VSCode, vous pouvez par exemple installer trois extensions Microsoft spécifiques pour gérer les programmes C (icône C/C++) :

- *C/C++ IntelliSense*,
- *C/C++ Extension Pack*, et
- *C/C++ Themes*.

Une fois ces extensions installées, l'éditeur reconnaîtra automatiquement votre fichier comme un programme C, à condition que le fichier se termine par « `.c` » ou « `.h` ».

Il est fortement conseillé de configurer l'éditeur pour reformater automatiquement le code, par exemple à chaque sauvegarde. Pour cela, ouvrez les préférences de VSCode (via la roue dentée *Settings* en bas à gauche, ou la combinaison de touches ctrl + , sous Linux), cherchez **`format on save`** et activez l'option. Alternativement, vous pouvez éditer le fichier de configuration `json`, accessible via la combinaison ctrl + ⇧ + P en cherchant **`user (JSON)`**. Un contenu minimal pour ce fichier peut être le suivant (la première ligne permet une prise en charge correcte du clavier si vous utilisez `X2go` pour vous connecter au CREMI).

```
{
  "keyboard.dispatch": "keyCode",
  "editor.suggestSelection": "first",
  "editor.tabSize": 4,
  "editor.rulers": [
    120
  ],
  "editor.formatOnSave": true,
  },
  "editor.fontSize": 14,
  "workbench.colorTheme": "Visual Studio Light - C++",
  "editor.formatOnSave": true,
}
```

Pour plus d'information sur la configuration de VSCode, reportez-vous à l'aide.

1.2 Compilation et exécution d'un programme C

1.2.1 Qu'est-ce qu'un programme C ?

Un programme C se compose de définitions de fonctions. Le programmeur peut définir autant de fonctions qu'il le souhaite, mais il doit y avoir au moins une fonction spéciale appelée « `main` ». Lorsque le programme est exécuté, la fonction « `main` » est appelée automatiquement, et toutes les instructions qu'elle contient sont exécutées séquentiellement, en commençant par la première. Examinons un exemple simple.

Un premier programme C : `Hello World!`

```c
// Pour la déclaration de printf
#include <stdio.h>

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

Compilation et exécution du programme

```
$ gcc -Wall -o hello fichier_hello.c
$ ./hello
Hello World!
$
```

Dans ce programme, la fonction « `main` » contient *deux* instructions :

1. La première instruction, `printf("Hello, World!\n");`, appelle la fonction `printf` de la bibliothèque standard. Cette fonction est utilisée pour afficher à l'écran de l'ordinateur une chaîne de caractères (c'est-à-dire une suite de caractères). Ici, nous l'utilisons pour afficher la chaîne `Hello, World!` suivie d'un caractère de saut de ligne (indiqué dans le programme par la suite de deux caractères `\n`). Le saut de ligne a pour effet de déplacer le curseur à la ligne suivante. Si on l'avait oublié, le *prompt* `$` du shell aurait été affiché juste après le dernier caractère « `!` » de la chaîne.
2. La deuxième instruction, `return 0;`, termine la fonction et renvoie la valeur `0`. Nous discuterons des fonctions et des valeurs de retour plus tard, ainsi que de la particulatité de la fonction `main()`.

Deux éléments fondamentaux de la syntaxe C

- Chaque instruction doit se terminer par un symbole point-virgule `;`. Cela marque la fin de l'instruction et indique au compilateur où une instruction s'arrête et où la suivante commence.
- Les instructions sont regroupées en blocs, qui sont entourés d'accolades `{...}`. Ces accolades définissent la portée du bloc. Cela a des implications importantes pour les variables, que nous aborderons plus tard.

Règles clés pour le programmeur C

- Contrairement à Python, l'indentation des lignes n'affecte pas l'exécution d'un programme C. Cependant, il est fortement recommandé d'utiliser une indentation appropriée, car cela améliore considérablement la lisibilité du programme. Cela est généralement géré automatiquement par votre éditeur.
- Des commentaires peuvent être ajoutés au code. Ceux-ci sont ignorés par le compilateur et sont utilisés pour améliorer la lisibilité du programme. Il y a deux syntaxes possibles pour les commentaires :
  - La séquence `/*` commence un commentaire, qui sera terminé à la prochaine séquence `*/`.
  - La séquence `//` commence aussi un commentaire, qui sera terminé en fin de ligne.

1.2.2 Compilation

Contrairement à Python, les programmes C doivent être compilés. Un programme C ne peut pas être exécuté directement : il doit d'abord être « traduit » en langage machine. Ce processus de traduction s'appelle la compilation, et il est réalisé à l'aide d'un compilateur. Si la compilation réussit, c'est-à-dire qu'aucune erreur n'est détectée, le compilateur créee un fichier exécutable qui pourra ensuite être exécuté.

Nous utiliserons le compilateur `gcc` (il y en a d'autres : sous MacOS, le compilateur par défaut est `clang`).

Illustrons le processus de compilation en utilisant notre programme `Hello World` comme exemple.

Compilation du programme `Hello World`

```
$ ls
fichier_hello.c
$ gcc fichier_hello.c -o hello
$ ls
hello     fichier_hello.c
$./hello
Hello World!
```

Le programme est écrit dans un fichier dont le nom est `fichier_hello.c` (qui est le seul fichier du répertoire). Pour le compiler, nous utilisons la commande `gcc fichier_hello.c -Wall -o hello`. Cette commande appelle le compilateur pour traduire le programme en un fichier exécutable nommé `hello` (on a choisi ce nom grâce à l'option `-o`). Une fois la compilation réussie, nous pouvons exécuter le programme compilé en lançant ce fichier avec la commande `./hello`.

Le standard C

Il existe de nombreux compilateurs C, comme `gcc` ou `clang`. Cependant, le langage C ne change pas d'un compilateur à l'autre, car il est régi par une norme internationale. Une *norme* est simplement l'ensemble des règles que les compilateurs doivent suivre. Cela sert à garantir que le code C écrit sur un système pourra être compilé et exécuté sur un autre (on dit que le code est *portable*). La norme C est mise à jour périodiquement. On parle de la norme C90, C99, C11, C18, C23, etc. Le compilateur comprend plusieurs normes. Par exemple, l'option `-std=c18` de `gcc` demande l'utilisation de la norme C18.

Remarque

La norme est un document de référence, mais :

1. Elle n'est pas accessible gratuitement. Des versions préliminaires (*drafts*) peuvent être trouvées sur `https://www.iso-9899.info/wiki/The_Standard`. Par exemple, le dernier *draft* de la norme C23 est disponible sur `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3220.pdf`.
2. La norme est faite pour être un document de référence, pas pour être pédagogique. Elle est difficile à lire, et on ne l'utilise que sur des points ponctuels.

1.2.3 Erreurs et avertissements

La compilation d'un programme échoue si le programme contient des *erreurs* (par exemple, s'il ne respecte pas la syntaxe du langage). Dans ce cas, aucun fichier exécutable n'est produit, et le compilateur affiche un message d'erreur indiquant où et pourquoi la compilation a échoué. Par exemple, modifions notre programme `Hello World` en supprimant le point-virgule après la première instruction.

```c
#include <stdio.h>

int main(void) {
  printf("Hello, World!\n")
      // Il manque un point-virgule.
      return 0;
}
```

```
$ gcc fichier_hello.c -o hello
fichier_hello.c:4:30: error: expected
 ↪  ';' after expression
4 |     printf("Hello, World!\n")
  |                              ^
  |                              ;
1 error generated.
```

Le compilateur indique que le programme n'a pas pu être compilé car il manque un point-virgule à la ligne 4, colonne 30. Ce message d'erreur aide le programmeur à identifier rapidement et corriger le problème. Remarquez aussi que l'éditeur, qui indente le texte au fur et à mesure, a décalé horizontalement le commentaire et l'instruction `return 0;`. C'est également une aide pour visualiser le problème.

En plus des erreurs, un compilateur C peut également générer des *avertissements* (« warnings » en anglais). Un avertissement met en évidence des problèmes potentiels dans votre code qui n'empêchent pas la compilation du programme, mais qui peuvent entraîner un comportement inattendu ou des bogues. Parmi les exemples courants, citons les deux situations suivantes, que nous aborderons plus tard lors de l'introduction des variables :

- Utilisation de variables non initialisées.
- Conversions implicites de types qui pourraient entraîner une perte de données ou des résultats inattendus.

Un avertissement n'empêche pas la compilation du programme : un fichier exécutable est produit s'il n'y a pas d'erreur. Cependant, il est important de traiter ces avertissements, car les ignorer peut entraîner des erreurs difficiles à déboguer ou un comportement indéfini (c'est-à-dire, qui pourra différer d'une exécution à l'autre).

Demander au compilateur de générer plus d'avertissements

Vous pouvez demander au compilateur de générer des messages d'avertissement supplémentaires pour mieux identifier les situations pouvant conduire à des bogues. On utilisera les options suivantes :

- `-Wall` active tous les avertissements standard.
- `-Wextra` active des avertissements utiles non activés par `-Wall`.
- `-pedantic` génère des avertissements lorsque le code s'écarte de la norme.
- `-Werror` traite tout avertissement comme erreur : un avertissement empêchera de créer l'exécutable.

Ces options doivent toujours être utilisées, car elles aident à produire un code fiable. Il est donc fortement recommandé de compiler votre code par la commande suivante :

```
gcc program.c -o executable -Wall -Wextra -pedantic -Werror
```

Règles-clés pour programmer : les erreurs et avertissements sont vos amis

En programmant, il est courant de commettre des erreurs. Rencontrer des messages d'erreur ou des avertissements est normal et attendu. Cependant, ces erreurs doivent être corrigées. Un programme qui ne se compile pas ou génère des avertissements est considéré comme incorrect. Un programme ne peut pas être « presque correct » ; il fonctionne ou il ne fonctionne pas. Il est crucial de suivre ces deux règles :

1. Le programme doit être compilé fréquemment pour détecter et corriger rapidement les erreurs. Si vous attendez que tout le programme soit écrit avant de le compiler, vous risquez de vous retrouver face à des dizaines d'erreurs à corriger d'un coup, ce qui peut être décourageant et chronophage.
2. Il est important de lire et comprendre minutieusement les messages d'erreur et les avertissements. Ils sont conçus pour **aider** le programmeur.

1.3 Progresser et trouver de l'aide

Le langage C repose sur un petit nombre de principes qu'il est facile de maîtriser. Mais il est aussi facile de passer à côté, auquel cas on ne sait pas vraiment ce qu'on fait quand on programme. Il ne faut surtout pas écrire du code par imitation, et encore moins déléguer cette tâche à une IA comme `ChatGPT` ou `copilot` : cela ne vous apprendra rien. Les principes pour s'améliorer sont simples.

Principes pour bien programmer en C

Pour progresser en programmation C, il faut connaître ces principes, et en particulier :

- Comprendre les phases principales de la compilation. En effet, le compilateur enchaîne plusieurs tâches, et tout message d'erreur est émis par l'une d'elles. Pour comprendre un tel message, il est utile de savoir quelle composante du compilateur nous informe. Ces phases sont expliquées au chapitre 2.
- Comprendre les messages du compilateur. N'en ignorez aucun ! Si vous ne comprenez pas un message, internet est votre ami. Les IA peuvent probablement expliquer les messages les plus courants de `gcc` (mais il faut lire les réponses avec un œil critique, car elles répondent parfois n'importe quoi).
- Surtout, il est important de bien comprendre l'effet des instructions écrites sur la mémoire. Se représenter mentalement la mémoire lors de l'exécution du programme qu'on développe est essentiel. En effet, le langage C demande aux programmeurs et programmeuses de gérer la mémoire. Un programme ne fait donc que transformer ou explorer de la mémoire. Pour cette raison, il est fondamental de dessiner la mémoire, dès que les structures employées deviennent complexes.

Lorsque vous avez une question précise, vous pouvez consulter la documentation du langage ou des outils.

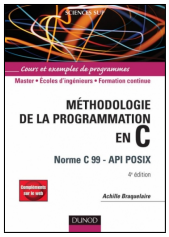
- La norme est décrite dans des documents publiés par l'International Organization for Standardization. Ces documents sont payants (si !) mais on peut accéder gratuitement à des versions préliminaires (*drafts*), par exemple sur `https://open-std.org`. Ainsi, ce document est une version récente de la norme du langage C (746 pages). Voir aussi `https://en.cppreference.com/w/c.html`.

- La documentation des compilateurs est également disponible :
  - Ici pour `gcc` (1214 pages) et là pour le précompilateur `cpp` (82 pages). La précompilation est essentiellement la première étape lancée par `gcc`.
  - Ici pour `clang`, ou, pour une documentation pdf d'une version antérieure, là (1057 pages).

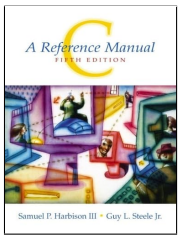
La documentation de `gcc` est aussi disponible via le `shell`, en lançant `info gcc`. Pour apprendre à utiliser la commande `info` elle-même, vous pouvez lancer `info info`.

Ces documents sont peu digestes. Ils sont conçus pour être des références (du langage C pour la norme, des outils pour les compilateurs ou précompilateurs). Il ne sont pas faits pour être lus linéairement, mais pour s'y reporter en cas de doute ou de question précise. Il existe de nombreux bons livres présentant le langage. Nous indiquons les suivants : le premier [1] commence à un niveau débutant. Le second [2] est plus proche d'une explication de la norme. Le troisième, [3], est très complet et contient plus de détails que [1]. Il est disponible sur `https://inria.hal.science/hal-02383654v2/file/modernC.pdf` (voir aussi la page du livre).

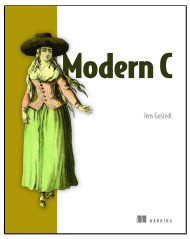
[1] Achille Braquelaire Méthodologie de la programmation en C (4e éd.), 2005. *Norme C99 - API POSIX*.



[2] Samuel P. Harbison III et Guy L. Steele Jr. C, A reference manual (5e éd.), 2002. Prentice Hall.



[3] Jens Gustedt, Modern C, 2025. Manning.



⚠ Un mot sur l'utilisation d'outils d'intelligence artificielle

Utiliser une IA pour générer du code n'aide pas à progresser. Toute aide externe étant interdite pendant les TP notés de cet enseignement, il vaut mieux travailler en désactivant ce type d'outil (ChatGPT, Copilot).

Ces outils peuvent proposer des explications de messages du compilateur ou produire du code de fonctions simples, mais actuellement, la qualité des réponses varie de bonne à catastrophique (elles sont par contre toujours présentées poliment et avec beaucoup d'assurance). Quant à la production de code, elle reste encore moyenne. À vos risques et périls, donc... et à utiliser avec un œil critique.

2 La compilation

L'objectif de ce chapitre est de comprendre le rôle des différentes parties du compilateur. Le *compilateur* est le programme qui convertit un ensemble de programmes sources C en fichier exécutable (ou en bibliothèque).

2.1 Principes généraux

Un programme C est écrit dans un ou plusieurs fichiers qu'on appelle ***fichiers sources***. Les noms des fichiers qui contiennent des instructions (placées à l'intérieur de définitions de fonctions) se terminent par `.c`. D'autres fichiers, de nom terminé par `.h`, sont appelés ***fichiers d'en-tête*** (*headers*, en anglais, d'où l'extension : `.h`). Ils peuvent contenir des déclarations de fonctions (aussi appelés prototypes), de variables ou de types.

Pour obtenir un programme exécutable, on traduit les programmes en langage machine avec un programme appelé ***compilateur***. Si le programme est complet et correct, le compilateur peut créer un ***fichier exécutable***. Le compilateur que nous utiliserons s'appelle `gcc`.

2.1.1 Définitions et déclarations

En C, une fonction ou variable doit avoir été déclarée avant d'être utilisée.

Qu'est-ce qu'une déclaration ?

Une *déclaration* (d'une fonction, d'une variable, d'un type) peut se comprendre comme une « promesse » faite au compilateur que l'objet déclaré (la fonction, la variable, le type) sera bien défini dans le programme. La *définition* de l'objet peut se trouver dans le même fichier source (après la déclaration), ou dans un autre.

La différence de syntaxe entre déclaration et définition est facile à visualiser pour les fonctions : une définition de fonction contient le **code** de la fonction (ses **instructions**), alors que sa déclaration, aussi appelée *prototype*, ne contient que :

- le nom de la fonction,
- le type de la valeur renvoyée (ou `void` si la fonction ne renvoie pas de valeur), et
- le type des paramètres de la fonction (avec éventuellement leur nom).

Par exemple, le cadre de gauche suivant définit la fonction `fact`, dont le prototype est donné dans chacun des cadres de droite (en nommant les paramètres dans le premier cadre, sans les nommer dans le second) :

Fonction factorielle : définition

```
int fact(int n) {
  if (n <= 1)
    return 1;
  else
    return n * fact(n - 1);
}
```

Fonction factorielle : prototype (ou déclaration)

```
int fact(int n);
```

Fonction factorielle : autre syntaxe du prototype

```
int fact(int);
```

Pour les variables globales, la différence syntaxique est moins nette entre définition et déclaration. Considérons les trois sources suivants :

Déclaration de variable

```
extern int x;
```

Définition de variable

```
int x = 0;
```

Déclaration ou définition

```
int x;
```

Le premier fichier contient une déclaration d'une variable `x` de type `int`. C'est le mot-clé `extern` qui indique que c'est une déclaration, et non une définition. Rappelons qu'une telle déclaration joue le rôle qu'une promesse faite au compilateur que la variable est définie ailleurs : dans un autre fichier, ou dans une bibliothèque.

Le second fichier contient une définition d'une variable `x` de type `int`, en raison de l'initialisation `= 0`. La différence entre déclaration et définition pour une variable globale est qu'une définition demande au compilateur de réserver de la mémoire pour la variable. Si la variable est initialisée (*i.e.*, si on indique une valeur initiale en même temps que son nom et son type), il s'agit d'une définition. Il faut donc éviter d'utiliser à la fois le mot-clé `extern` (qui indique une déclaration) et une initialisation (qui indique une définition). Le compilateur émet un avertissement si cela arrive. Enfin, une définition de variable joue aussi le rôle de déclaration.

Le troisième fichier

On peut compiler un fichier pourvu qu'il contienne toutes les déclarations des variables et fonctions utilisées. En particulier, on peut compiler un fichier même s'il manque des définitions de fonctions utilisées. Par contre, il n'est possible de créer un exécutable que si toutes les fonctions utilisées sont définies, soit par la personne qui écrit le programme, soit dans une bibliothèque (contenant des fonctions déjà écrites et pré-compilées). Ainsi, le fichier suivant peut être compilé, même si la fonction `fact` n'est pas définie :

Fichier pouvant être compilé avec `gcc -c`

```c
#include <stdio.h>

int fact(int);

int main(void) {
  printf("%d\n", fact(5));
  return 0;
}
```

Notez que ce fichier ne contient ni la définition de la fonction `fact`, ni celle de la fonction `printf`. Mais on a donné explicitement le prototype de la fonction `fact`, et celui de la fonction `printf` se trouve dans le fichier d'en-tête `stdio.h` (qui se trouve quelque part sur le système, à un endroit que le compilateur `gcc` sait trouver).

Compilation du fichier précédent sans créer l'exécutable

```
$ gcc -std=c18 -Wall -Wextra -pedantic -c fact_usage.c
# Aucune erreur du compilateur (mais aucun exécutable créé).
$ ls -l fact_usage.o
.rw-r--r-- 784 mz staff 2025-09-10 17:53 fact_usage.o
# Le compilateur a créé un fichier .o.
```

L'option `-c` de `gcc` arrête la compilation avant la phase d'édition de liens (*linkage*, en anglais). Cette phase vérifie que tout objet qui a été promis dans un fichier (par une déclaration) a une unique définition dans un des sources C ou dans une bibliothèque. Si c'est bien le cas, elle crée l'exécutable. A contrario, l'option `-c` ne crée pas d'exécutable. Si on omet cette option dans notre exemple, on obtient une erreur. En effet, on demande alors de créer l'exécutable, pour lequel le compilateur a besoin de la définition de la fonction `fact`, non fournie :

Tentative de compilation complète du fichier précédent

```
$ gcc -std=c18 -Wall -Wextra -pedantic fact_usage.c
/usr/bin/ld : /tmp/cache-mzeitoun/ccMxtPQN.o : dans la fonction « main » :
fact.c:(.text+0xa) : référence indéfinie vers « fact »
collect2: error: ld returned 1 exit status
```

⚠ Fichiers d'en-tête et bibliothèques

Les fichiers d'en-tête ne contiennent pas de définitions de fonctions. Le code de la fonction `printf` ne se trouve pas dans le fichier `stdio.h`. Il n'y a que son prototype. Le code lui-même se trouve, déjà pre-compilé, dans la « bibliothèque standard » que le compilateur sait trouver sur le système.

Une *bibliothèque* est simplement un ensemble de fonctions déjà écrites, et pré-compilées. Par défaut, le compilateur cherche le code des fonctions non définies dans une bibliothèque particulière, appelée la *bibliothèque standard C* (ou *bibliothèque standard C*, ou *libc*). C'est dans cette bibliothèque que se trouve le code déjà compilé de la fonction `printf`.

2.2 Phases de la compilation d'un programme C

Lorsqu'on compile un programme C, on utilise en général une seule commande, par exemple `gcc`. En réalité, le programme `gcc` enchaîne plusieurs phases dans un ordre précis, et on peut lui demander d'arrêter son travail après une de ces phases. La Figure 2.1 indique ces principales phases.

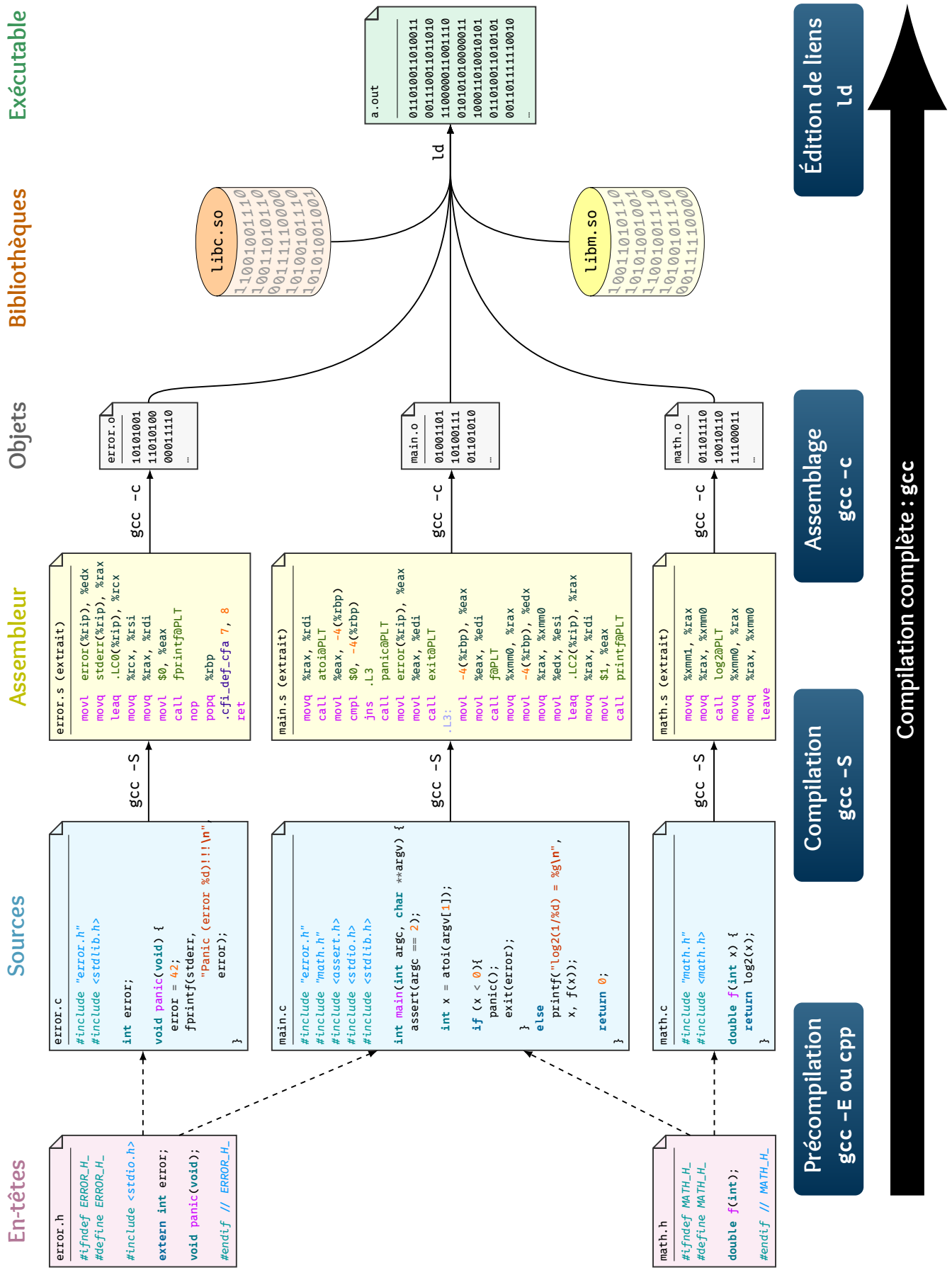


FIGURE 2.1 : Phases de la compilation

Ces phases sont les suivantes :

1. La *pré-compilation* : cette première phase ne fait « que » des manipulations de texte, sans analyser le programme. En particulier, elle n'est pas responsable de détecter d'éventuelles erreurs de syntaxe. En particulier, elle :
   - supprime les commentaires,
   - fusionne les constantes « chaînes de caractères »,
   - traite les directives destinées au pré-compilateur, qui commencent par un caractère `#`, comme `#define`, `#if`, `#ifdef`, `#ifndef` ou `#include`.
2. La *compilation*, qui transforme le code obtenu après précompilation en code assembleur. L'assembleur est un langage de programmation dont les instructions sont très simples. Il est encore lisible par un humain et il est possible d'écrire en assembleur, mais cela est long et fastidieux.
3. L'*assemblage*, qui permet de compiler un fichier écrit en assembleur, sans vérifier que toutes les variables et fonctions utilisées sont bien définies, soit dans le code écrit, soit dans une bibliothèque.
4. L'*édition de liens* (*linkage* en anglais), qui crée l'exécutable, si toutes les variables et fonctions sont définies.

Nous allons illustrer les trois premières phases en utilisant l'exemple suivant, composé de deux fichiers :

Fichier `f1.h`

```
#ifndef F1_H_
#define F1_H_

// Déclaration de la variable globale y
extern int y;

// Prototype de la fonction f1
extern void f1(int x);

#endif // F1_H_
```

Fichier `f1.c`

```
#include "f1.h"

#define X 10

int y = X;

// Définition de la fonction f1
void f1(int x) {
  y += x;
}
```

2.2.1 La précompilation

On peut compiler le fichier `f1.c` en demandant à `gcc` de s'arrêter après la phase de pré-compilation :

```
$ gcc -E f1.c > f1.i
```

On obtient le fichier C ci-dessous encore lisible :

Fichier `f1.i`

```
# 1 "f1.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 466 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "f1.c" 2
# 1 "./f1.h" 1

extern int y;

extern void f1(int x);
# 2 "f1.c" 2

int y = 10;

void f1(int x) {
  y += x;
}
```

Remarquez que :

- Il n'y a plus de commentaires : le pré-compilateur les a supprimés.
- Les lignes qui commençaient par un caractère `#` dans les fichiers sources, appelées *directives*, ont été traitées de façon particulière.

Les directives les plus courantes sont les suivantes :

- Lorsque le pré-compilateur rencontre une ligne comme :

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

  il remplace cette ligne par le contenu du fichier d'en-tête standard `stdio.h`, en ajoutant quelques informations sur les fichiers inclus et les numéros de ligne, pour pouvoir les indiquer dans ses messages d'erreur. Il est possible de lui indiquer explicitement où chercher ces fichiers d'en-tête, mais si on utilise que la bibliothèque standard, la configuration par défaut du compilateur fait qu'il connaît leur localisation.

  Lorsque le pré-compilateur rencontre une ligne comme

  ```
  #include "f1.h"
  ```

  il remplace cette ligne par le contenu du fichier local `f1.h`, cherché par défaut dans le même répertoire que le fichier dans lequel se trouve cette directive de précompilation
- Lorsque le pré-compilateur rencontre une ligne comme :

  ```
  #define X 10
  ```

  il remplace les occurrences de `X` qu'il trouve dans la suite du code par `10`. On appelle cela une ***macro***. On peut écrire une macro avec des arguments, par exemple :

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

  Dans ce cas, `max(10,5)` est remplacé par `((10) > (5)) ? (10) : (5)`, qui est une ***expression*** qui s'évalue en `10`.

> **⚠ Attention aux macros**
>
> Plusieurs points concernant les macros demandent de la vigilance :
>
> - La macro précédente est donnée comme exemple, mais elle n'est pas idéale. En effet, elle peut conduire à une double évaluation. Si un des arguments effectue un ***effet de bord***, cela peut être problématique. Par exemple, `max(x++,y)` peut incrémenter `x` de deux unités, ce qui n'est pas forcément souhaité. Il faut donc faire attention aux macros qui peuvent évaluer deux fois l'un des arguments.
> - Les macros doivent être écrites sur une seule ligne. Si la macro est longue, on peut utiliser le caractère `\` en fin de ligne.
> - On parenthèse habituellement les arguments des macros. En effet, si on écrit
>
>   ```
>   #define MULT(a,b) a * b
>   ```
>
>   alors `MULT(1+2, 3)` s'évaluera en `1+2 * 3`, ce qui n'est probablement pas ce qu'on a en tête.

Pour définir des macros qu'on peut utiliser avec une syntaxe similaire à une fonction, on utilise la forme suivante :

```
#define F() do { \
    f();       \
    g();       \
  } while (0)
```

Une macro peut aussi prendre un nombre variable d'arguments. Nous n'insisterons pas sur cet aspect dans ce cours.

- On peut aussi définir des macros sans leur donner une valeur, pour tester ensuite si elles sont ou non définies. C'est ce mécanisme qui est utilisé au début du fichier `f1.h` pour éviter qu'il soit inclus deux fois. À la seconde inclusion, la macro `F1_H_` est déjà définie, et la directive `#ifndef` s'évalue en faux. Le contenu du fichier n'est donc pas inclus une seconde fois.

- La macro `#if` évalue une condition dont la valeur peut être calculée à la compilation, et le code jusqu'au `#endif` correspondant n'est conservé que si cette condition est vraie. On peut donc commenter un bloc de code en l'entourant de `#if 0` et `#endif`. De façon similaire, `#ifdef` permet de tester qu'une macro est définie, et `#ifndef` qu'une macro n'est pas définie.

2.2.2 La compilation

Pour arrêter la compilation à la phase de génération du code assembleur, on utilise l'option `-S` de `gcc`. Cela produit un fichier avec l'extension `.s`. Ce fichier contient du code toujours lisible, mais dans un langage très « bas niveau ».

2.2.3 L'assemblage

Pour arrêter la compilation à la phase d'assemblage, on utilise, on utilise l'option `-c` de `gcc`. Cela produit un fichier avec l'extension `.o`. Ce fichier n'est plus lisible par un humain. Nous verrons que cette option est très utile.

2.2.4 La création de l'exécutable

Pour créer l'exécutable, on fournit à `gcc` plusieurs fichiers C qui peuvent être des sources, ou être déjà compilés par l'option `-c`. Le compilateur vérifie que chaque variable, fonction, type qui a été déclaré a aussi une unique définition dans un des fichiers. Si ce n'est pas le cas, il l'indique. Sinon, il crée l'exécutable.

2.3 La compilation séparée

Lorsqu'on développe des projets importants, il est utile de découper le code en plusieurs fichiers. Habituellement, chaque fichier a une fonction spécifique. Par exemple, on peut avoir un fichier implémentant une petite bibliothèque de manipulation de listes, et d'autres fichiers utilisant cette bibliothèque. L'avantage de la séparation du code en plusieurs fichiers est multiple : d'une part, cela permet de garder des fichiers ayant chacun une fonction logique. Cela structure ainsi le projet. D'autre part, cela rend la compilation plus rapide. En effet, si on dispose d'un projet contenant plusieurs fichiers et que seul l'un d'eux change, on peut ne compiler que celui-ci et refaire seulement l'édition de liens.

La figure suivante indique comment sont compilés ces fichiers (les pointillés indiquent les `#include`, les flèches les étapes de compilation).

Résumé

Un programme C est écrit dans un ou plusieurs fichiers qu'on appelle ***fichiers sources***. Les noms des fichiers qui contiennent des instructions (placées à l'intérieur de définitions de fonctions) se terminent par l'extension `.c`. Ces fichiers peuvent aussi contenir des définitions ou déclarations de variables ou de types. D'autres fichiers, dont le nom est terminé par `.h`, peuvent contenir des déclarations de fonctions (aussi appelés prototypes), de variables ou de types. En revanche, les fichiers `.h` ne doivent pas contenir de définitions de fonctions.

En résumé, pour obtenir un programme exécutable, on utilise donc un ***compilateur*** (comme `gcc`), qui est un logiciel qui transforme les fichiers sources en un fichier exécutable. On peut compiler un fichier pourvu qu'il contienne toutes les déclarations des variables et fonctions **utilisées**. En particulier, on peut compiler un fichier même s'il ne contient pas une définition d'une des fonctions utilisées. Par contre, bien sûr, il ne sera pas possible de créer un fichier exécutable s'il manque des définitions de fonctions. Ainsi, le fichier suivant peut être compilé : Pour cela, in utilise l'option `-c` du compilateur.

Compilation du fichier `f1.c` sans créer l'exécutable

```
$ gcc -std=c18 -Wall -c f1.c
# aucune erreur du compilateur (mais aucun exécutable créé)
$ ls -l f1.o
-rw-r--r-- 784 mz staff 2024-09-05 07:09 f1.o
# le compilateur a créé un fichier .o.
```

L'option `-c` de `gcc` demande ainsi de s'arrêter avant la phase de *linkage* (celle qui crée l'exécutable). Avec cette option, aucun exécutable n'est donc créé. Si on omet cette option, on obtient une erreur. En effet, dans ce cas, on demande la création de l'exécutable, mais le compilateur ne sait pas où trouver le code de la fonction `main` :

Tentative de compilation complète du fichier précédent

```
$ gcc -std=c18 -Wall f1.c
Undefined symbols for architecture arm64:
  "_main", referenced from:
      <initial-undefines>
ld: symbol(s) not found for architecture arm64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Exercice 1 On reprend l'exemple précédent avec les fichiers `f1.c`, `f2.c`, `main.c`, `f1.h`, `f2.h`.

1. Quelles sont les commandes à lancer pour compiler séparément `f2.c` et `main.c` ? Pourquoi peut-on utiliser la fonction `f2` dans `main.c` ?
2. Quelles sont les commandes à lancer pour créer l'exécutable à partir des fichiers `.o` ? À partir des fichiers sources `.c` et `.h` ?

Exercice 2 Écrire en C **dans deux fichiers séparés** les algorithmes correspondant aux programmes mutuellement récursifs vus en algorithmique :

```
1 : procédure TEST3(entier n)
2 :     si n = 0 alors
3 :         afficher 0
4 :     sinon
5 :         afficher n
6 :         TEST4(n − 1)
7 :     fin si
8 : fin procédure
```

```
1 : procédure TEST4(entier n)
2 :     si n = 0 alors
3 :         afficher 0
4 :     sinon
5 :         TEST3(n − 1)
6 :         afficher n
7 :     fin si
8 : fin procédure
```

3

Premiers éléments du langage

Dans ce chapitre, nous proposons une introduction à l'écriture de programmes de base en C. Nous expliquons comment déclarer ou définir des variables et des fonctions, et les manipuler. Afin de pouvoir écrire des fonctions intéressantes, nous présentons aussi l'utilisation des structures conditionnelles et des boucles. Les concepts sont illustrés par des exemples pouvant être testés sur n'importe quel ordinateur avec un compilateur C installé.

3.1 Expressions et Variables

Comme dans d'autres langages de programmation, les *variables* en C sont utilisées pour stocker et manipuler des valeurs. Cependant, en C, les variables doivent être déclarées avec une instruction avant de pouvoir être utilisées. Une déclaration de variable nécessite d'indiquer son type et de choisir son nom. Pour déclarer une variable, le type est écrit en premier, suivi du nom de la variable, puis d'un point-virgule qui termine l'instruction :

```
type variableName;
```

Vous pouvez également déclarer plusieurs variables du même type simultanément :

```
type variableName1, variableName2, variableName3, ... ;
```

Lorsqu'une variable est déclarée, un espace mémoire est alloué [1] pour stocker sa valeur, la taille de cet espace mémoire dépendant du type de la variable. Voici quelques exemples pour illustrer cela.

```c
int n;          // Déclaration d'une variable n de type entier.

float x, y, z; // Déclaration de trois variables x, y et z de type flottant.

bool a, b;      // Déclaration de deux variables a et b de type Booléen.
```

⚠ Règles clés pour le programmeur C : choisir des noms de variables pertinents

Les noms de variables n'ont de signification que pour le programmeur et n'affectent pas le programme lui-même. Ils ne sont pas inclus dans le fichier exécutable généré par le compilateur. Cependant, il est considéré comme une bonne pratique de choisir des noms de variables clairs, pertinents par rapport à leur fonction, et cohérents avec les conventions de nommage globales utilisées dans le programme.

1. Conceptuellement, c'est plutôt la définition de variable qui demande l'allocation mémoire.

Définition vs. déclaration

Il y a souvent confusion entre la « définition » et la « déclaration » d'une variable. Cela vient du fait que souvent, une définition de variable joue aussi le rôle de déclaration.

La différence entre les deux concepts est la suivante : une *déclaration de variable* est simplement une promesse faite au compilateur que la variable existe, et sera définie dans le programme ou dans une bibliothèque. Une *définition de variable* demande au compilateur de réserver de la place pour cette variable.

Pour les variables locales, définir une variable se fait avec la même syntaxe que pour la déclarer. La variable sera créée et accessible lorsque la fonction ou bloc dans laquelle elle se trouve sera exécutée.

La situation est différente pour les variables globales, c'est-à-dire définies en dehors du corps de toute fonction. En effet, une telle variable peut être utilisée dans plusieurs fichiers. Pour cela, elle doit être définie dans l'un d'eux, et déclarée dans les autres. Autrement dit, dans un des fichiers, on définit cette variable globale, et dans les autres fichiers où on veut pouvoir l'utiliser, on la déclare, c'est-à-dire qu'on promet au compilateur qu'elle est définie ailleurs. Pour préciser qu'on déclare une variable définie ailleurs, on utilise le mot-clé `extern`, comme dans la première ligne du code suivant qui déclare la variable `a`. La seconde ligne, qui ne contient pas ce mot-clé, définit une variable globale `b`, initialisée à `42`.

```
extern int a;
int b = 42;
```

Que signifie « allouer » ? « désallouer » ?

Les termes *allouer* et *désallouer* sont fréquemment utilisés.

- « allouer » signifie « réserver de la place en mémoire » (par exemple, pour stocker une variable).
- « désallouer » signifie rendre la place qui a été allouée.

Souvent, l'allocation et la désallocation est faite automatiquement lors de l'exécution d'un programme. Nous verrons que le programmeur peut aussi demander explicitement l'allocation ou la désallocation de zones en mémoire.

Initialisation de variables

Lorsqu'on définit une variable, on peut préciser une valeur initiale avec laquelle la variable sera créée. La syntaxe est la suivante (comme dans la ligne qui définit la variable `b` de l'exemple précédent) :

```
type variableName1 = initialValue;
```

Initialisation de variables « `static` »

Nous verrons plus tard que par défaut, les variables locales d'une fonction sont :

- allouées lorsque la fonction est appelée (c'est-à-dire que de la place est réservée par le compilateur pour stocker leur valeur au moment où on commence à exécuter la fonction),
- désallouées lorsque la fonction se termine (c'est-à-dire qu'à ce moment, la place qui leur avait été allouée à l'appel de la fonction est reprise par le système).

Une exception apparaît si on déclare une variable avec le mot-clé `static`. Dans ce cas, de la place pour la variable est allouée une seule fois, à la compilation (c'est aussi le cas pour les variables globales). Dans ce cas précis, il ne faut pas confondre initialisation lors de la définition et affectation après cette définition. Considérons les deux codes suivants, placés dans le corps d'une fonction :

```
{
  static int a;
  a = 3;
  //...
}
```

```
{
  static int a = 3;
  //...
}
```

Dans le premier cas, la variable est remise à la valeur 3 à chaque appel de fonction (le fait que la variable soit `static` n'a donc aucun intérêt dans ce cas). Au contraire, dans le deuxième cas, elle ne prend la valeur 3 qu'une seule fois, à sa création. Il ne faut donc pas confondre initialisation de variable, et affectation.

3.1.1 Types

En C, chaque variable a un type, qui est spécifié par le programmeur lors de la déclaration de la variable. Le type d'une variable détermine les éléments suivants :

- L'étendue des valeurs que la variable peut contenir, spécifiant le type de données qu'elle stocke (*par exemple*, entier, flottant, Booléen, ...).
- La quantité de mémoire utilisée pour stocker les valeurs de la variable et la façon dont ces valeurs sont représentées[2].
- Les opérations qui peuvent être effectuées sur la variable.

Le langage C fournit plusieurs types prédéfinis. Voici quelques exemples importants.

Entiers. Le type `int` stocke des valeurs entières. Les entiers sont signés, ce qui signifie qu'ils peuvent être soit négatifs, soit positifs. *L'étendue réelle n'est pas fixée par la norme C* et dépend du compilateur. Les compilateurs classiques comme `gcc` utilisent souvent 4 octets pour stocker une variable `int`, ce qui lui permet de représenter des valeurs allant de −2147483648 à 2147483647.

Flottants. Le type `double` stocke des valeurs dites flottantes (*par exemple*, 3.14, −56.7, ...).

Caractères. Le type `char` stocke des caractères écrits entre guillemets simples : `'A'`, `'x'`, `'8'`, etc. Cela peut prêter à confusion : ne pas confondre `'3'` (qui est un caractère) avec `3` (qui est un entier). Les caractères sont stockés en utilisant un seul octet, ce qui est stocké en mémoire est souvent leur code ASCII. Ainsi, tout caractère a aussi une valeur entière.

Booléens. Le type `bool` a seulement deux valeurs possibles : `true` et `false`. Il est utilisé pour représenter le résultat d'une expression logique.

Booléens et valeur nulle

En C, la valeur nulle est considérée comme fausse, et toute valeur non nulle comme vraie.

Utilisation des Booléens

Le type `bool` n'est pas directement disponible. Pour l'utiliser, inclure la directive `#include <stdbool.h>` au début du code source.

2. Cela n'est pas toujours fixé par la norme C et dépend donc du compilateur.

? Autres types

Ce ne sont pas les seuls types disponibles en C :

- Il existe d'autres types entiers, comme `short`, `long` ou `long long`. La principale différence entre ces types et le type `int` est leur étendue. Elle dépend du compilateur : la norme ne précise pas sur quelle taille ces types doivent être représentés, ni comment. Elle indique des valeurs minimales. Par exemple, une variable de type `char` doit être codée sur au moins 8 bits (et est, en pratique, souvent codée sur un octet). Une variable de type `long` doit être codée sur au moins 32 bits, une variable de type `long long` sur 64 bits, etc. Le fichier `limits.h` précise les tailles utilisées sur le système.
- Le mot-clé `unsigned` permet de spécifier des entiers « non signés », représentant des valeurs positives ou nulles. Si un type est représenté sur n bits, sa version `unsigned` permet de représenter les entiers de 0 à $2^n - 1$. Par exemple, si le type `long long int` est représenté sur 8 octets, soit 64 bits, les variables de type `unsigned long long int` représentent des valeurs de 0 à $2^{64} - 1$. Il est conseillé d'utiliser ce type lorsqu'on en a vraiment besoin : pour représenter de (très) grands entiers, ou pour faire des opérations bit à bit.
- Il existe également d'autres types flottants, comme `float` ou `long double`. La différence entre ces types et le type `double` est leur précision (qui dépend également du compilateur).
- Il existe des types pointeurs.
- Le programmeur peut enfin définir ses propres types personnalisés.

Nous reviendrons sur les deux derniers points dans le chapitre 4 et le chapitre 6).

3.1.2 Expressions

Pour expliquer comment les variables sont manipulées dans un programme, il est d'abord nécessaire de discuter des *expressions*. C'est un concept fondamental en informatique et dans les langages de programmation.

> Une expression est une construction syntaxique qui peut être évaluée en une valeur typée.

Concrètement, une expression est formée en combinant des constantes, des variables, des opérateurs et des fonctions de manière à ce que le langage de programmation puisse l'évaluer pour produire une valeur. Voici quelques exemples d'expressions :

- `42` consiste en une seule constante et s'évalue à la valeur `42` de type `int`.
- `'B'` consiste en une seule constante et s'évalue à la valeur `'B'` de type `char`.
- `13 < 23` consiste en deux constantes et un opérateur, et s'évalue à la valeur `true` de type `bool`.
- Si `n` est une variable de type `int` avec la valeur `7`, alors `1 + 3 * n` est une expression composée de deux constantes, une variable et deux opérateurs, et s'évalue à la valeur `22` de type `int`.

Les fonctions peuvent également être utilisées dans les expressions. Nous reviendrons sur ce point plus tard. Pour l'instant, présentons quelques opérateurs standard.

Opérateurs arithmétiques

Ces opérateurs effectuent des opérations arithmétiques sur des valeurs numériques (*par exemple*, de type `int` ou `double`). Voici une liste des principaux opérateurs, classés par ordre croissant de priorité :

1. *Addition* (`+`) : Ajoute deux nombres. Exemples :
   - `3 + -4` s'évalue à `-1` de type `int`.
   - `3.14 + 23.1` s'évalue à `26.24` de type `double`.
2. *Soustraction* (`-`) : Soustrait un nombre d'un autre. Exemples :
   - `7 - 18` s'évalue à `-11` de type `int`.
   - `2.05 - 23.1` s'évalue à `-21.05` de type `double`.
3. *Multiplication* (`*`) : Multiplie deux nombres. Exemples :
   - `3 * 4` s'évalue à `12` de type `int`.
   - `3.0 * 1.5` s'évalue à `4.5` de type `double`.
4. *Division* (`/`) : Divise un nombre par un autre. Le comportement de cet opérateur varie selon que les opérandes sont des entiers ou des flottants. Exemples :
   - `15 / 2` s'évalue à `7` de type `int` (une division entière est effectuée).
   - `15.0 / 2.0` s'évalue à `7.5` de type `double`.
5. *Modulo* (`%`) : Renvoie le reste de la division entière. Cet opérateur est uniquement défini pour des opérandes entiers. Exemple :
   - `27 / 4` s'évalue à `3` de type `int` (car $27 = 4 \times 6 + 3$).

Les opérateurs sont listés par ordre croissant de priorité, ce qui signifie que dans l'expression `x + 3 * y`, la multiplication a une priorité plus élevée que l'addition, et cette expression correspond donc à `x + (3 * y)`.

Opérateurs logiques

Ces opérateurs sont utilisés pour effectuer des opérations logiques sur des valeurs Booléennes. Voici une liste des principaux opérateurs logiques en C, classés par ordre croissant de priorité :

1. *OU logique* (`||`). Exemple : « `true || false` » s'évalue à `true` de type `bool`.
2. *ET logique* (`&&`). Exemple : `true && false` s'évalue à `false` de type `bool`.
3. *NON logique* (`!`). Exemple : `!true` s'évalue à `false` de type `bool`.

Les opérateurs sont classés par ordre croissant de priorité. Ainsi, dans l'expression `!x && y || z`, l'opérateur *NON* a la priorité la plus élevée, suivi de l'opérateur *ET*, puis de l'opérateur *OU*. Par conséquent, l'expression est interprétée comme `((!x) && y) || z`.

Évaluation paresseuse

En C, les expressions impliquant les opérateurs « `||` » et `&&` sont évaluées de manière paresseuse. Plus précisément, lors de l'évaluation de `<expression1> || <expression2>`, les étapes suivantes se produisent :

- `<expression1>` est évaluée.
- Si elle s'évalue à `true`, `<expression2>` n'est **pas évaluée**. L'expression entière s'évalue à `true`.
- Si elle s'évalue à `false`, `<expression2>` est ensuite évaluée.

De même, pour l'expression `<expression1> && <expression2>`, l'évaluation se déroule comme suit :

- `<expression1>` est évaluée.
- Si elle s'évalue à `false`, `<expression2>` n'est **pas évaluée**. L'expression entière s'évalue à `false`.
- Si elle s'évalue à `true`, `<expression2>` est ensuite évaluée.

Opérateurs relationnels et de comparaison

Ces opérateurs sont utilisés pour comparer des valeurs. Voici une liste des principaux opérateurs :

1. *Inférieur à* (`<`). Vérifie si un nombre est strictement plus petit qu'un autre. Exemple :
   - `2 < 3` s'évalue à `true` de type `bool`.
2. *Supérieur à* (`>`). Vérifie si un nombre est strictement plus grand qu'un autre. Exemple :
   - `2 > 3` s'évalue à `false` de type `bool`.
3. *Inférieur ou égal à* (`<=`). Vérifie si un nombre est plus petit ou égal à un autre. Exemple :
   - `23 <= 42` s'évalue à `true` de type `bool`.
4. *Supérieur ou égal à* (`>=`). Vérifie si un nombre est plus grand ou égal à un autre. Exemple :
   - `23 >= 42` s'évalue à `false` de type `bool`.
5. *Égal à* (`==`). Vérifie si deux valeurs sont égales. Exemples :
   - `true == true` s'évalue à `true` de type `bool`.
   - `7 == 12` s'évalue à `false` de type `bool`.
6. *Différent de* (`!=`). Vérifie si deux valeurs sont différentes. Exemples :
   - `false != true` s'évalue à `true` de type `bool`.
   - `12 != 12` s'évalue à `false` de type `bool`.

Opérateurs bit à bit

Il ne faut pas confondre les opérateurs logiques et les *opérateurs bit à bit*. Ils s'utilisent sur des variables entières `unsigned` (car pour les types signés, l'encodage du signe n'est pas fixé par la norme et peut varier d'un compilateur à l'autre). Ils permettent de manipuler les bits individuels de telles variables.

1. *Décalage à gauche (`<<`)*. Décale les bits de la représentation d'un entier de plusieurs positions vers la gauche, en remplissant par des zéros à droite. Exemple, si `x` est l'`unsigned int` valant 13 :
   - `x << 5` s'évalue en l'entier obtenu en décalant la représentation de 13 de 5 positions vers la gauche. Les 5 bits de gauche de l'entier d'origine sont perdus. Si les entiers sont codés sur 16 bits et l'entier 13 représenté en binaire par `00000000 00001101`, l'expression s'évalue en l'entier dont la représentation est `00000001 10100000`, soit $13 \times 2^5 = 13 \times 32 = 416$.
2. *Décalage à gauche (`>>`)*. Décale les bits de la représentation d'un entier de plusieurs positions vers la droite, en remplissant par des zéros à gauche. Exemple, si `x` est l'`unsigned int` valant 417, représenté sur deux octets par `00000001 10100001` :
   - `x >> 7` est représenté par `00000000 00000011`, soit l'entier 3. Les 7 bits de droite de l'entier d'origine sont perdus, et on complète par 7 zéros à gauche.
3. *ET bit à bit* (`&`). Effectue le ET bit à bit de deux nombres, en alignant les nombres sur la droite. Exemple, pour l'`unsigned int x = 34952` et l'`unsigned int y = 50115`, en les supposant représentés sur 2 octets par `10001000 10001000` (pour `x`) et `11000011 11000011` (pour `y`) :
   - `x & y` vaut l'entier représenté par `10000000 10000000`.
4. *OU bit à bit (`|`)*. Effectue le OU bit à bit de deux nombres, en alignant les nombres sur la droite. Exemple, pour les mêmes représentations des `unsigned int` `x` et `y` :
   - `x | y` vaut l'entier représenté par `11001011 11001011`.
5. *XOR bit à bit (`^`)* Effectue le OU EXCLUSIF bit à bit de deux nombres, en alignant les nombres sur la droite. Exemple, pour les mêmes représentations des `unsigned int` `x` et `y` :
   - `x ^ y` vaut l'entier représenté par `01001011 01001011`.
6. *NON bit à bit (`~`)* Effectue le NON bit à bit sur un nombre. Exemple, pour les mêmes représentations des `unsigned int` `x` et `y` :
   - `~x` vaut l'entier représenté par `01110111 01110111` et `~y` celui représenté par `00111100 00111100`.

⚠ Opérateurs logiques et opérateurs bit à bit

Attention à ne pas confondre ces deux types d'opérateurs ! Ces erreurs sont difficiles à déboguer. En effet :

- Les opérateurs logiques sont légaux sur des entiers, tout comme les opérateurs bit à bit. Rappelons qu'un entier non nul est considéré comme vrai, et l'entier nul comme faux.
- Les notations sont très similaires.

Par exemple, si les `unsigned int` `x` et `y` sont codés par `10000000 00000000` et `00000000 00000001`, alors `x & y` vaut `0`, alors que `x && y` vaut `1` (la valeur logique « vrai »).

3.1.3 Affectation d'une valeur à une variable

Une valeur est *affectée* à une variable (qui a été préalablement déclarée) avec l'instruction suivante :

```
variableName = <expression>;
```

Lorsque cette instruction est exécutée, l'expression est évaluée par le programme et le résultat est ensuite stocké dans l'espace mémoire correspondant à la variable (qui a été alloué lors de la définition de la variable).

```c
#include <stdio.h>

int main(void) {
    int n;
    n = 42;
    printf("n = %d.\n", n);
    n = n + 13;
    printf("n = %d.\n", n);
    n = n / 2;
    printf("n = %d.\n", n);
    return 0;
}
```

```
$ ./variables1
n = 42.
n = 55;
n = 27;
```

De plus, une valeur peut être affectée à une variable au moment de sa déclaration :

```
type variableName = <expression>;
```

Lors de l'exécution, la variable est déclarée, ce qui alloue de la mémoire pour elle. Cette mémoire allouée est ensuite immédiatement remplie avec le résultat de l'évaluation de l'expression. Examinons quelques exemples.

```c
#include <stdio.h>

int main(void){
    int n = 42;
    printf("n = %d.\n", n);
    int k = (n + 1) / 2;
    printf("k = %d.\n", k);
    int i = 5, j = n * k + i;
    printf("i = %d and j = %d.\n", i, j);
    return 0;
}
```

```
$ ./variables
n = 42.
k = 21.
i = 5 and j = 887.
```

⚠ Conversion implicite

Lors de l'affectation d'une valeur à une variable, il n'est pas strictement nécessaire que l'expression s'évalue à une valeur du même type que la variable. Si le type de la variable et le type de l'expression ne sont pas les mêmes, mais sont « compatibles », une conversion est effectuée, du type de la valeur évaluée vers celui de la variable (ou la valeur de retour d'une fonction). Ce processus est appelé *conversion implicite*.
En fonction du contexte, ces conversions implicites peuvent entraîner un comportement imprévisible et indésirable. En général, le compilateur générera un avertissement si une conversion implicite potentielle-

⚠ Conversion implicite (suite)

ment problématique est détectée. Les conversions implicites générant des avertissement sont déconseillées dans ce cours. Même pour celles qui n'en génèrent pas, il faut être conscient de ce processus.

Présentons un exemple impliquant le type **`unsigned int`**, conçu pour stocker *uniquement des entiers positifs*. Son avantage par rapport à **`int`** réside dans sa capacité à stocker une plage plus large d'entiers positifs, car il ne prend pas en compte les valeurs négatives. Cependant, sauf si cette plage accrue est absolument nécessaire, il n'est pas conseillé d'utiliser **`unsigned int`**, car cela peut induire des erreurs.

```c
#include <stdio.h>

int main(void) {
    int a = -1;
    unsigned int k = a;
    a = k / 3;
    printf("a = %d.\n", a);
    return 0;
}
```

```
$ ./conversion1
a = 1431655765.
$
```

Nous n'obtenons pas le résultat auquel on s'attendrait. La raison est la suivante. Dans ce programme, la valeur `-1` est affectée à la variable **`int`** `a`. Cette valeur est ensuite affectée à la variable **`unsigned int`** `k`. Étant donné que `-1` n'est *pas* un **`unsigned int`**, le compilateur C effectue une conversion implicite : `k` reçoit la valeur `4294967295`. Ainsi, le résultat est exactement cette valeur divisée par trois.

Plus précisément, `-1` et `4294967295` ont exactement le même encodage en mémoire sur la machine où le programme a été testé. Selon le type de la variable, cet encodage est interprété différemment : comme la valeur `-1` pour un **`int`** et comme la valeur `4294967295` pour un **`unsigned int`**. Sur cette machine, les **`int`** et **`unsigned int`** sont codés sur 4 octets. Le codage des entiers négatifs est le codage en complément à 2, dans lequel `-1` se représente par les 32 bits (soit 4 octets) suivants : `11111111111111111111111111111111`. Lorsque l'affectation à l'**`unsigned k`** est effectuée, cette valeur est interprétée comme un nombre positif en base 2, soit comme `4294967295`.

En compilant le programme avec `gcc -Wconversion`, on obtiendrait un avertissement à ce sujet :

```
$ gcc -Wconversion conversion1.c
conversion1.c:5:22: warning: conversion to ‘unsigned int' from ‘int' may change the
↪  sign of the result [-Wsign-conversion]
    5 |     unsigned int k = a;
      |                      ^
conversion1.c:6:9: warning: conversion to ‘int' from ‘unsigned int' may change the
↪  sign of the result [-Wsign-conversion]
    6 |     a = k / 3;
      |         ^
```

Modifions le programme en remplaçant le type **`unsigned int`** par le type **`long int`** :

```c
#include <stdio.h>

int main(void)
{
    long int k = 0xffffffff;
    int a = -1;
    a = k / 3;
    printf("a = %d.\n", a);
    return 0;
}
```

```
$ ./conversion2
a = 0.
$
```

Cette fois, nous obtenons le résultat attendu : `(-1)/3` s'évalue en `0`, car le compilateur C détecte une division entière. Cependant, deux conversions ont eu lieu :

⚠ Conversion implicite (suite)

- Une première lors de l'initialisation de `long int k = a;`, l'entier `int a` est converti en `long int` pour être stocké dans `k`. Cette conversion implicite ne fait pas perdre de précision, et n'est donc pas problématique, car la norme indique qu'un `long int` est codé sur au moins la même taille qu'un `int`.
- La seconde conversion, `a = k / 3`, convertit `k / 3`, qui est de type `long int`, en le type de `a`, qui est `int`. Cette conversion ne pose en fait pas de problème ici, car, on l'a vu, `k / 3` vaut `0`. Cependant, si `k / 3` était un entier long plus grand que l'entier maximal pouvant être stocké dans le type `int`, on aurait perdu de la précision. Le compilateur indique donc un problème potentiel :

```
$ gcc -Wconversion conversion2.c
conversion2.c: In function ‘main’:
conversion2.c:7:9: warning: conversion from ‘long int’ to ‘int’ may change
 ↪  value [-Wconversion]
    7 |     a = k / 3;
      |         ^
```

Présentons une dernière variation sur ce programme. Cette fois, on affecte un entier de type `long long int` à un entier de type `int`, ce qui provoque à nouveau une conversion implicite :

```c
#include <stdio.h>

int main(void)
{
    long long int k = 0xffffffffff;
    printf("k = %lld\n", k);
    int a = k;
    printf("a = %d.\n", a);
    return 0;
}
```

```
$ ./conversion3
k = 68719476735
a = -1.
$
```

Ici, la valeur de `k` est donnée en base 16 (cela est indiqué par le préfixe `0x` du nombre, qui est `fffffffff` en base 16, soit 68719476735 en base 10). La chaîne `%lld` passée à la fonction `printf` indique le format de l'entier à imprimer : entier au format `long long` (`ll`) à afficher en base 10 (`d`). À nouveau, le compilateur nous avertit si on utilise l'option `-Wconversion` :

```
$ gcc -Wconversion conversion3.c
conversion3.c: In function ‘main’:
conversion3.c:7:13: warning: conversion from ‘long long int’ to ‘int’ may
 ↪  change value [-Wconversion]
    7 |     int a = k;
      |             ^
```

Nous terminons par un dernier exemple concernant le mécanisme de conversion implicite. Supposons qu'on veut écrire une fonction prenant deux nombres en arguments `a` et `b`, et qui renvoie un entier `int` ayant même signe que `a-b`. Autrement dit, la fonction doit renvoyer

- `0` si `a == b`,
- Un entier strictement positif si `a > b`,
- Un entier strictement négatif si `a < b`.

La fonction suivante remplit parfaitement ce rôle :

```c
int signe(int a, int b) { return a - b; }
```

Imaginons maintenant qu'on se rende compte que les arguments de la fonction `signe` doivent en fait être de type `double`. Si on adapte la fonction sans réfléchir, on peut aboutir au code suivant :

Conversion implicite (fin)

```
int signe(double a, double b) { return a - b; }
```

Le problème est qu'une conversion implicite se produit au retour de la fonction. Par exemple, si `a` vaut `1.0` et `b` vaut `0.7`, alors `(a - b)` vaut `0.3`. Mais comme la fonction renvoie un `int`, cette valeur, `0.3`, est convertie en l'entier `0` au retour de la fonction. Puisque la fonction aurait dû renvoyer un entier strictement positif dans ce cas, elle est incorrecte. À nouveau, ceci est indiqué par le compilateur :

```
$ gcc -c -Wall -Wextra -Wconversion conversion4.c
conversion4.c: In function 'signe':
conversion4.c:1:42: warning: conversion from 'double' to 'int' may change
 ↪  value [-Wfloat-conversion]
    1 | int signe(double a, double b) { return a - b; }
      |                                          ~~^~~
```

Conversion explicite

On peut demander une conversion explicite d'un type à un autre. La syntaxe est la suivante :

(typeVersLequelConvertir) valeur

Nous reviendrons sur ce mécanisme plus tard.

Que se passe-t-il lorsqu'une variable n'a pas été initialisée ?

Examinons le programme suivant, qui génère un avertissement lors de la compilation :

```c
#include <stdio.h>

int main(void) {
  int n;
  printf("n = %d.\n",n);
  return 0;
}
```

```
$ gcc prog.c -o prog -Wall
uninitialized.c:5:25: warning: variable
 ↪  'n' is uninitialized when used here
 ↪  [-Wuninitialized]
5 |     printf("n = %d.\n", n);
|                          ^
1 warning generated.
```

Lorsqu'une variable est déclarée, de la mémoire est allouée pour stocker sa valeur. Si cette valeur n'est pas initialisée, elle sera égale à celle qui était précédemment stockée à cet emplacement mémoire. Le programmeur n'a aucun contrôle sur cette valeur : c'est pourquoi un avertissement est généré.

3.1.4 Les affectations sont des expressions

Une caractéristique du langage C est que les affectations de variables sont également des expressions. Par exemple, si `x` est une variable de type `int`, alors l'instruction `x = 3` est *une expression de type* `int`, ce qui signifie qu'elle s'évalue à une valeur. Plus précisément, lorsque l'expression `x = 3` est évaluée, deux choses se produisent :

1. La variable `x` se voit affecter la valeur `3`.
2. L'expression elle-même s'évalue à la valeur `3`.

En conséquence, le programme suivant est correct en C.

```c
#include <stdio.h>

int main(void) {
  int x, y;
  x = y = 42;
  printf("x = %d, y = %d.\n",x,y);
  return 0;
}
```

```
$ ./aexps
x = 42, y = 42.
```

Lorsque `x = y = 42;` est exécutée, l'expression `y = 42` est d'abord évaluée. Cela affecte la valeur `42` à `y` et s'évalue en `42`, qui est ensuite affectée à `x`.

Dans la pratique, il n'est pas recommandé d'utiliser cette fonctionnalité. Elle complique souvent le code et le rend plus difficile à comprendre. L'un des bogues les plus courants est lié à cette fonctionnalité.

⚠ Ne pas confondre les affectations avec les tests d'égalité

Une erreur fréquente en C est de confondre l'opérateur de test d'égalité `==` avec l'opérateur d'affectation `=`. Considérons qu'une variable `int x` a été déclarée et initialisée à la valeur `3`. Dans ce cas,

1. L'expression `x == 42` s'évalue à `false`.
2. L'expression `x = 42` affecte `42` à `x` et s'évalue à `42`.

Utiliser la deuxième expression au lieu du test d'égalité souhaité peut entraîner deux problèmes majeurs. Premièrement, la valeur de `x` est modifiée, ce qui n'était pas l'intention. Deuxièmement, en C, les entiers peuvent être traités comme des valeurs Booléennes : `0` est considéré comme `false`, tandis que toute valeur non nulle est interprétée comme `true`. Par conséquent, du point de vue des Booléens, la deuxième expression s'évalue à `true`, ce qui est l'opposé du résultat attendu.

ℹ Incrémentation et décrémentation

Deux opérateurs supplémentaires sur les entiers sont fréquemment utilisés pour incrémenter un entier (c'est-à-dire ajouter `1`) et décrémenter un entier (c'est-à-dire soustraire `1`). Ces opérateurs peuvent être utilisés sous deux formes. On suppose qu'une variable `int x` est déclarée et initialisée.

- **Postfixe**. L'expression `x++` incrémente `x` et s'évalue à la valeur que `x` avait avant cet incrément. De même, l'expression `x--` décrémente `x` et s'évalue à la valeur que `x` avait avant ce décrément.
- **Préfixe**. L'expression `++x` incrémente `x` et s'évalue à la valeur que `x` a après cet incrément. De même, l'expression `--x` décrémente `x` et s'évalue à la valeur que `x` a après ce décrément.

```c
#include <stdio.h>

int main(void) {
  int x = 0, y = 0;
  printf("Premier x: %d.\n",x++);
  printf("Deuxième x: %d.\n",x++);
  printf("Premier y: %d.\n",++y);
  printf("Deuxième y: %d.\n",++y);
  return 0;
}
```

```
$ ./plusplus
Premier x: 0.
Deuxième x: 1.
Premier y: 1.
Deuxième y: 2.
```

3.1.5 Durée de vie et portée d'une variable

Nous abordons deux concepts importants en C : la portée des variables, et leur durée de vie. Dans ce cours, nous n'aborderons que deux types de variables.

Les deux concepts sont parfois confondus, et donc mal compris. De façon informelle,

- La *portée d'une variable* est définie comme l'ensemble des emplacements dans le code où le programmeur a le droit de faire référence à la variable. C'est donc une notion spatiale (puisqu'elle fait référence aux « emplacements dans le code »).
- La *durée de vie d'une variable* est définie comme les moments pendant lesquels, lors de l'exécution du programme, la variable possède un emplacement mémoire. C'est donc une notion temporelle (puisqu'elle fait référence aux « moments » de l'exécution du programme).

Variables globales. Une variable globale en C est une variable déclarée en dehors de toutes les fonctions, généralement au début du programme, avant les définitions de fonctions. Ce type de variable est accessible depuis n'importe quelle fonction du programme après sa déclaration, ce qui signifie qu'elle peut être utilisée et modifiée par n'importe quelle partie du code après sa déclaration.

Déclaration de variables globales

```c
#include <stdio.h>

int a = 50;
int b = 32;

int main(void) {
  printf("La somme est %d.\n",a + b);
  return 0;
}
```

```
$ ./plusplus
La somme est 72.
```

La durée de vie d'une variable globale est celle de l'exécution du programme. Elle est allouée lorsque le programme démarre et détruite lorsque le programme se termine.

Variables locales. Une variable locale en C est une variable déclarée au sein d'une fonction ou d'un bloc de code (tel qu'une boucle ou une condition) délimité par des accolades `{...}`. Elle est **accessible uniquement dans cette fonction ou ce bloc**. Une fois que la fonction ou le bloc a terminé son exécution, la variable locale sort de la portée et est **détruite**. Par exemple, la compilation du programme incorrect suivant générera une erreur de compilation.

La portée des variables locales

```c
#include <stdio.h>

int main(void) {
  {
    int x = 43;
  }
  printf("%d\n", x);
  return 0;
}
```

```
$ gcc scope.c -o scope -Wall
scope.c:11:20: erreur: utilisation de
 ↪  l'identifiant non déclaré 'x'
11 |     printf("%d\n", x);
|                     ^
1 erreur générée.
```

La portée de `x` est limitée au bloc de code dans lequel il est déclaré (délimité par des accolades `{...}`).

⚠ Masquage de variables

Le masquage de variables se produit lorsqu'une variable locale dans une fonction ou un bloc porte le même nom qu'une variable déclarée à l'extérieur de ce bloc. Dans ce cas, la variable locale « masque » la variable extérieure dans la portée de ce bloc ou de cette fonction. La variable la plus interne prend priorité.

```c
#include <stdio.h>

int main(void) {
    int x = 18, y = 7;
    printf("Début: %d, %d\n", x, y);
    {
        int y = 0;
        printf("Bloc1: %d, %d\n", x,
        ↪  y);
    }
    {
        int x = 43;
        printf("Bloc2: %d, %d\n", x,
        ↪  y);
    }
    printf("Fin: %d, %d\n", x, y);
    return 0;
}
```

```
$ ./masking
Début: 18, 7
Premier bloc: 18, 0
Deuxième bloc: 43, 7
Fin: 18, 7
```

Le masquage peut entraîner de la confusion et des bugs, surtout dans de grands programmes. Il est généralement conseillé d'**éviter d'utiliser le même nom de variable** à la fois dans les portées locales et globales.

3.2 La fonction printf

Avant de continuer, revenons sur la fonction `printf`, utilisée fréquemment dans nos exemples jusqu'à présent. Elle permet d'afficher des données sur la *sortie standard* (généralement le terminal où le programme s'exécute). Cette fonction est déjà écrite pour nous. En effet, en plus du langage de base, la norme C décrit tout un ensemble de fonctions déjà écrites. On appelle cet ensemble de fonctions la *bibliothèque standard C* (aussi appelée par son petit nom libc, pour "C library"). Plus généralement, une *bibliothèque* est un ensemble de fonctions déjà écrites. On verra dans le chapitre 2 comment demander au compilateur de charger une bibliothèque particulière. Pour la bibliothèque standard, on n'a pas besoin de le demander, car le compilateur charge la bibliothèque standard automatiquement.

La fonction `printf` fait partie de la bibliothèque standard C et est déclarée dans le fichier d'en-tête `stdio.h`. En pratique, cela signifie que le code source doit commencer par la directive suivante pour utiliser la fonction :

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

La fonction `printf` est appelée avec la syntaxe suivante :

```c
printf("format_string", suite_d_arguments);
```

- `"format_string"` : Il s'agit d'une chaîne de caractères qui peut contenir du texte, des séquences d'échappement (comme `\n` pour un saut de ligne) et des *spécificateurs de format* (comme `%d`, `%f`, `%s`) qui indiquent comment afficher les arguments fournis.
- `suite_d_arguments` : Ce sont des expressions (séparés par une virgule) dont les valeurs sont à afficher, et elles doivent correspondre aux spécificateurs de format dans la chaîne `"format_string"`.

Voici une liste des spécificateurs de format courants :

- `%d` : Utilisé pour afficher un *entier* (`int`).
- `%lg` : Utilisé pour afficher un *nombre à virgule flottante* (`double`).
- `%c` : Utilisé pour afficher un *caractère* (`char`).
- `%s` : Utilisé pour afficher une *chaîne de caractères* (`char*`, nous aborderons ce type plus tard).

La fonction printf

```c
#include <stdio.h>

int main(void) {
   int num = 10;
   char letter = 'A';
   printf("Un entier %d et un caractère %c.\n", num, letter);

   double pi = 3.14;
   char* str = "test";
   printf("Un nombre à virgule flottante %lg et une chaîne de caractères %s.\n", pi, str);

   return 0;
}
```

```
$ ./printf
Un entier 10 et un caractère A.
Un nombre à virgule flottante 3.140000 et une chaîne de caractères test.
```

Comment savoir quelles sont les fonctions de la bibliothèque standard ?

Vous pouvez consulter la norme, la référence [2], ou `https://en.cppreference.com/w/c.html`.

Comment obtenir la documentation d'une fonction de bibliothèque particulière ?

La commande shell `man` permet d'obtenir la documentation des fonctions de bibliothèques. La documentation est segmentée en sections. Les fonctions de la bibliothèque standard font partie de la section 3. Par exemple, on obtient la documentation de `printf` par :

```
$ man 3 printf
```

En particulier, cela vous indiquera qu'il faut utiliser

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

pour utiliser la fonction `printf` en C.

3.3 Exécution conditionnelle

En C, l'exécution conditionnelle est implémentée à l'aide de l'instruction **`if-else`**, qui permet au programmeur d'exécuter différents blocs de code en fonction de la véracité ou non d'une condition.

3.3.1 L'instruction `if`

L'instruction **`if`** vérifie une condition spécifiée par une expression. Si cette expression s'évalue à `true`, le bloc de code à l'intérieur des accolades est exécuté :

```c
if (<expression>) {
  // Code à exécuter si l'expression s'évalue à true
}
```

Exemple d'instruction « if »

```c
#include <stdio.h>

int main(void) {
  int x = 10;
  if (x == 42) {
    printf("x est égal à 42.\n");
  }
  if (x != 42) {
    printf("x n'est pas égal à 42.\n");
  }
  return 0;
}
```

```
$ ./ifexample
x n'est pas égal à 42.
$
```

3.3.2 L'instruction `if-else`

L'instruction **`if-else`** fournit un bloc de code alternatif à exécuter lorsque la condition est fausse.

```c
if (<expression>) {
  // Code à exécuter si l'expression s'évalue à true
}
else {
  // Code à exécuter si l'expression s'évalue à false
}
```

Avec l'instruction **`if-else`** en main, nous pouvons simplifier l'exemple précédent.

Exemple d'instruction « if-else »

```c
#include <stdio.h>

int main(void) {
  int x = 10;
  if (x == 42) {
    printf("x est égal à 42.\n");
  }
  else {
    printf("x n'est pas égal à 42.\n");
  }
  return 0;
}
```

```
$ ./ifexample
x n'est pas égal à 42.
$
```

3.3.3 Exercices

Nous concluons notre discussion sur les conditionnels avec deux exercices.

Exercice 3 : Un autre exemple Que va afficher le programme suivant lorsqu'il sera exécuté ?

```c
#include <stdio.h>

int main(void) {
  int a = 1, b = 2, c = 3;
  if (a < 6) {
    if (b < c) {
      printf("Programmer");
    }
    else {
      printf("Le gâteau");
    }
    if (c/2 < b){
      printf(" en C");
    }
  }
  if (3 * a + b <= c){
    printf(" est délicieux.\n");
  }
  else {
    printf(" est merveilleux.\n");
  }
  return 0;

}
```

Exercice 4 : Une erreur courante La compilation du programme suivant génère un avertissement. Pourquoi cela se produit-il et que va afficher le programme lorsqu'il sera exécuté ?

```c
#include <stdio.h>

int main(void) {
  int x = 10;
  if (x = 42) {
    printf("x est égal à 42.\n");
  }
  else {
    printf("x n'est pas égal à 42.\n");
  }

  return 0;
}
```

3.4 Boucles

Les boucles en C sont utilisées pour exécuter un bloc de code de manière répétée jusqu'à ce qu'une condition spécifiée soit remplie. Il existe deux types principaux de boucles.

3.4.1 La boucle `for`

Commençons par un programme simple qui utilise une boucle `for`.

Un programme utilisant une boucle for

```c
#include <stdio.h>

int main(void) {
  for (int i = 0; i < 7; i++) {
    printf("i = %d.\n",i);
  }
  return 0;
}
```

```
$ ./forexample
i = 0.
i = 1.
i = 2.
i = 3.
i = 4.
i = 5.
i = 6.
```

Dans ce cas, le bloc de code à l'intérieur de la boucle **`for`** est exécuté *sept* fois. Lors de la première itération, la variable `i` prend la valeur `0`, et elle est incrémentée après chaque itération.

Maintenant, introduisons la syntaxe générale pour déclarer une boucle **`for`**.

```c
for (<initialisation>; <condition>; <mise à jour>) {
  // Code à exécuter à chaque itération
}
```

Comme décrit dans la définition, une boucle **`for`** dépend de *trois composants* :

1. `<initialisation>` : Cette partie est exécutée une seule fois, au début de la boucle (avant la première itération). Elle est généralement utilisée pour initialiser les variables qui contrôlent la boucle. Par exemple, dans l'exemple précédent, la variable `i` est déclarée et initialisée à `0`.
2. `<condition>` : Cette expression Booléenne détermine si la boucle continue ou se termine. Elle est évaluée avant chaque itération. Si la condition est évaluée comme `true`, la boucle continue. Si elle est évaluée comme `false`, la boucle se termine.
3. `<mise à jour>` : Cette partie est exécutée à la fin de chaque itération, juste avant que la suivante commence. Elle est généralement utilisée pour mettre à jour les variables qui contrôlent la boucle. Par exemple, dans l'exemple précédent, la variable `i` est incrémentée à la fin de chaque itération.

Une boucle for n'a pas nécessairement besoin de se terminer

L'exemple fourni plus tôt illustre l'utilisation standard et la plus courante d'une boucle **`for`**. Cependant, il existe très peu de restrictions sur ce que `<initialisation>`, `<condition>` et `<mise à jour>` peuvent contenir. Par exemple, le code suivant est parfaitement valide en C :

```c
#include <stdio.h>

int main(void) {
  for ( ; true ; ) {
    printf("J'aime la programmation en C.\n");
  }
  return 0;
}
```

Dans ce cas, la `<condition>` est toujours évaluée comme `true` (`<initialisation>` et `<mise à jour>` sont vides). En conséquence, la boucle continuera indéfiniment et le programme déclarera éternellement son amour pour le C sans jamais se terminer.

3.4.2 La boucle `while`

Encore une fois, commençons par un programme simple qui utilise une boucle **`while`**.

Un programme utilisant une boucle while

```c
#include <stdio.h>

int main(void) {
  int i = 1;
  while (i < 300) {
    printf("i = %d.\n",i);
    i = 2 * i;
  }
  return 0;
}
```

```
$ ./whileexample
i = 1.
i = 2.
i = 4.
i = 8.
i = 16.
i = 32.
i = 64.
i = 128.
i = 256.
```

Dans ce cas, le bloc de code à l'intérieur de la boucle **`while`** est exécuté exactement *neuf* fois. Avant que la boucle ne commence, la variable `i` est initialisée à `1`. À la fin de chaque itération, `i` est multipliée par `2`. La boucle se termine lorsque `i` dépasse `300`, car c'est la condition spécifiée pour la fin de la boucle.

Maintenant, introduisons la syntaxe générale pour déclarer une boucle **`while`**.

```c
while (<condition>) {
/* Code à exécuter à chaque itération */
}
```

L'exécution d'une boucle **`while`** se déroule comme suit :

1. La `<condition>`, une expression Booléenne, est évaluée avant chaque itération.
2. Si la condition est évaluée comme `false`, la boucle se termine et l'exécution continue avec l'instruction suivante après la boucle.
3. Si la condition est évaluée comme `true` :
   - Le bloc de code à l'intérieur de la boucle **`while`** est exécuté.
   - Après l'exécution du bloc de code, le processus retourne à l'étape 1.

Une boucle while n'a pas nécessairement besoin de se terminer

Typiquement, la `<condition>` dépend de variables modifiées dans le bloc de code de la boucle **`while`**. Il faut en effet s'assurer que `<condition>` finira par être évaluée comme `false`, provoquant ainsi la fin de la boucle. Sinon, la boucle pourrait potentiellement s'exécuter indéfiniment (même s'il existe d'autres moyens de sortir d'une boucle, que nous aborderons plus tard). Par exemple, le code suivant est valide en C :

```c
int main(void) {
  int i = 0;
  while (i < 100) {
    i = (i+1) % 50;
  }
  return 0;
}
```

Dans ce cas, `i` reste strictement inférieur à `50`. Par conséquent, `<condition>` sera toujours évaluée comme `true`, ce qui provoquera une boucle infinie.

Les boucles for et while peuvent être utilisées de manière interchangeable

Typiquement, les boucles **for** sont utilisées lorsque le nombre d'itérations est connu à l'avance, tandis que les boucles **while** sont utilisées lorsque le nombre d'itérations n'est pas prédéfini. Dans la pratique, respecter cette convention améliore la lisibilité de vos programmes.
Cependant, en C, ces deux types de boucles sont interchangeables : les boucles **for** peuvent être simulées par des boucles **while** et vice versa.

Simulation d'une boucle for par une boucle while

```
<initialisation>
while (<condition>) {
  // Code à exécuter
  <mise à jour>
}
```

Simulation d'une boucle while par une boucle for

```
for ( ; <condition> ; ) {
  // Code à exécuter
}
```

3.4.3 Exercices

Nous concluons notre discussion sur les boucles avec trois exercices.

Exercice 5 : Un programme mystérieux Que produit le programme suivant ?

```c
#include <stdio.h>

int main(void) {
  int m,n;
  // Code qui initialise n et m (des valeurs sont affectées)

  int res = 0;
  for (int i = n + (1-n%2); i <= m; i = i + 2) {
    res = res + i;
  }
  printf("Le résultat est %d.\n", res);
  return 0;
}
```

Exercice 6 : La boucle do-while Il existe un troisième type de boucle en C : la boucle **do-while**. C'est une variante de la boucle **while**, mais elle garantit que la boucle s'exécutera au moins une fois. Une boucle **do-while** est déclarée comme suit :

```
do {
  // Code
} while (<condition>);
```

L'exécution d'une boucle **do-while** se déroule comme suit :

1. Le bloc de code à l'intérieur de la boucle **do-while** est exécuté.
2. La `<condition>`, une expression Booléenne, est évaluée :
   - Si la condition s'évalue à `false`, la boucle se termine, et l'exécution continue avec l'instruction suivante après la boucle.
   - Si la condition s'évalue à `true`, le processus retourne à l'étape 1.

Considérez le code suivant qui utilise une boucle **do-while** :

```c
do {
  printf("x = %d.\n");
  x++;
} while (x < 4);
```

1. Déterminez la sortie de ce code pour les valeurs suivantes de `x` : `1`, `2`, `3`, et `4`.
2. Réécrivez le code en utilisant une boucle **while** au lieu d'une boucle **do-while** tout en maintenant

la même fonctionnalité.
3. Plus généralement, expliquez comment toute boucle `do-while` peut être convertie en une boucle `while` équivalente.
4. À l'inverse, décrivez comment toute boucle `while` peut être transformée en une boucle `do-while` équivalente.

L'utilisation des boucles `do-while` est principalement bénéfique pour améliorer la lisibilité, car elles permettent d'éviter la duplication de code dans certaines situations. Cependant, cela dépend du contexte, et dans la plupart des cas, il est inutile d'utiliser de boucles `do-while`.

Exercice 7 : Continue et break Les instructions `continue;` et `break;` offrent plus de contrôle sur les boucles. Elles peuvent être utilisées dans le bloc de code de la boucle comme suit :

- Lorsque `continue;` est rencontrée, l'itération actuelle de la boucle est immédiatement terminée, et l'exécution passe à l'itération suivante.
- Lorsque `break;` est rencontrée, la boucle est complètement terminée, et l'exécution passe à la première instruction après la boucle.

Que produisent les deux programmes suivants ?

```c
#include <stdio.h>

int main(void) {
  for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) {
      continue;
    }
    printf("i = %d.\n",i);
  }
  return 0;
}
```

```c
#include <stdio.h>

int main(void) {
  int i = 0;
  while (true) {
    if (i > 5) {
      break;
    }
    printf("i = %d.\n",i);
    i++;
  }
  return 0;
}
```

3.5 Fonctions

En C, une fonction est un bloc de code réutilisable conçu pour accomplir une tâche spécifique. Les principaux avantages des fonctions sont les suivants :

- *Réutilisabilité* : Les fonctions peuvent être appelées plusieurs fois, évitant ainsi la duplication de code.
- *Modularité* : Elles permettent de diviser des programmes complexes en parties plus simples et plus petites.
- *Facilité de débogage* : Le débogage et les tests de fonctions individuelles sont plus simples que de travailler avec un programme monolithique.
- *Lisibilité du code* : Les fonctions rendent les programmes plus faciles à lire et à comprendre.

Voyons deux exemples simples de fonctions.

```c
int max_int(int i, int j) {
  if (i < j) {
    return j;
  }
  else {
    return i;
  }
}
```

Cette fonction `max_int` a deux arguments, tous deux de type `int`, et retourne une valeur de type `int`, qui est le maximum de ses deux arguments.

```c
void display_c(void) {
  printf("Je programme en C.\n");
}
```

Cette fonction `display_c` n'a aucun argument et ne retourne aucune valeur (ce qui est indiqué par le type spécial `void`).

3.5.1 Prototypes

Avant de définir une fonction, il est recommandé (bien que non obligatoire) de fournir un prototype pour celle-ci. Cela ressemble à la déclaration d'une variable sans l'initialiser. Un prototype de fonction spécifie le nom de la fonction, son type de retour, ainsi que les types et le nombre de ses arguments (le cas échéant). Il est déclaré comme suit :

```
<type de retour> <nom> (<type1> <arg1>, <type2> <arg2>, ... , <typeN> <argN>);
```

Par exemple, voici les prototypes des fonctions `max_int` et `display_c` définies plus haut :

```c
int max_int(int i, int j);
```

```c
void display_c(void);
```

Enfin, il n'est pas nécessaire de spécifier des noms pour les arguments dans un prototype (et si des noms sont spécifiés, ils n'ont pas à être les mêmes que ceux utilisés dans la définition ultérieure de la fonction). Autrement dit, le prototype suivant est équivalent à celui de `max_int` :

```c
int max_int(int, int);
```

3.5.2 Définition d'une fonction

Une fonction est définie en fournissant son nom, son type de retour et ses paramètres (le cas échéant), suivis de son corps encadré par des accolades `{ ... }`. Si un prototype pour la fonction a été déclaré précédemment, le type de retour et les types des paramètres doivent correspondre à ceux spécifiés dans le prototype. Le corps contient les instructions qui exécutent la tâche de la fonction :

```
<type de retour> <nom> (<type1> <arg1>, <type2> <arg2>, ... , <typeN> <argN>) {
  // Corps de la fonction qui dépend de ses arguments
}
```

3.5.3 Appeler une fonction en C

Pour appeler une fonction en C, on utilise simplement le nom de la fonction suivi de parenthèses et on fournit les arguments nécessaires (le cas échéant) à l'intérieur des parenthèses. Si la fonction n'a pas d'arguments, on utilise des parenthèses vides. Illustrons cela avec la fonction `max_int` :

```c
#include <stdio.h>

int max_int(int,int); // Prototype (nécessaire car la définition de max_int vient après main)

int main(void){ /* La fonction principale */
  int a = 42, b = 17;
  printf("Et le maximum est %d.\n", max_int(a,b)); /* max_int est appelée */
  return 0;
}

int max_int(int i, int j) { /* La définition de max_int */
  if (i < j) {
    return j;
  }
  else {
    return i;
  }
}
```

Plus généralement, si `fun` est une fonction quelconque, elle est appelée comme suit :

```
fun(<expression1>, <expression2>, ... , <expressionN>)
```

Bien sûr, le nombre d'expressions à l'intérieur des parenthèses doit correspondre au nombre de paramètres définis dans la fonction `fun`. Lorsque cette expression est évaluée, les étapes suivantes sont effectuées :

1. Les arguments passés à `fun` sont évalués et stockés.

2. L'exécution du programme passe à la première instruction à l'intérieur de `fun`.
3. Lorsqu'on rencontre l'instruction **`return`** (ou la parenthèse fermante du bloc de code de la fonction), l'exécution revient au point suivant l'appel de `fun`.

⚠ Les arguments sont passés par valeur

Lorsqu'une fonction est appelée, ses arguments sont fournis sous forme d'expressions qui sont évaluées. Les valeurs résultantes sont ensuite assignées aux paramètres correspondants définis dans la fonction. Cela peut parfois prêter à confusion, notamment lorsque des noms de variables similaires sont utilisés.

```c
#include <stdio.h>

void plus(int);

int main(void){ /* La fonction principale */
  int x = 0;
  plus(x);
  printf("x = %d.\n", x);
  return 0;
}

void plus(int x) {
  x++;
}
```

Ce programme affiche « `0` » dans le terminal. Lorsque la fonction `plus` est appelée, son seul argument, l'expression `x`, est évaluée à la valeur « `0` », qui est ensuite assignée à la variable locale `x` de la fonction `plus`. Cela peut prêter à confusion : le deuxième `x` est local à la fonction et masque celui dans `main`.

3.5.4 L'instruction return

À l'intérieur du bloc de code d'une fonction, une instruction spéciale appelée **`return`** peut être utilisée. Lorsque cette instruction est exécutée, la fonction se termine et le contrôle revient immédiatement après l'appel de la fonction. L'utilisation de **`return`** diffère suivant le type de retour de la fonction :

1. Si le type de retour est **`void`** (c'est-à-dire que la fonction ne retourne pas de valeur), l'instruction **`return`**`;` peut être utilisée pour terminer explicitement la fonction. Toutefois, cela est optionnel, car la fonction se termine automatiquement lorsque la parenthèse fermante de son bloc de code est atteinte.

```c
#include <stdio.h>

void test42(int);

int main(void) {
  test42(4);
  test42(42);
  return 0;
}

void test42(int n) {
  if (n != 42) {
    printf("Je suis déçu...\n");
    return;
  }
  printf("Wow, c'est 42!\n)");
}
```

```
$ ./voidexample
Je suis déçu...
Wow, c'est 42!
$
```

Dans ce cas, une expression qui appelle la fonction ne retourne pas de valeur lorsqu'elle est évaluée. Par conséquent, elle ne peut pas être utilisée dans une affectation de variable. Par exemple, dans le programme ci-dessus, l'instruction `int k = test42(1);` n'est pas permise et provoquerait une erreur de compilation.

2. Si le type de retour n'est pas **`void`**, l'utilisation de **`return`** est obligatoire selon la norme C; sinon, le compilateur générera un avertissement. Dans ce cas, l'instruction **`return`** `<expression>;` doit être utilisée, où `<expression>` s'évalue à une valeur qui correspond au type de retour de la fonction.

```c
#include <stdio.h>

int sy(int);

int main(void) {
  printf("sy(2) = %d."\n, sy(2));
  printf("sy(7) = %d."\n, sy(7));
  return 0;
}

int sy(int n){
  if (n % 2 == 0){
    return n/2;
  }
  return 3 * n + 1;
}
```

```
$ ./intexample
sy(2) = 1.
sy(7) = 22.
$
```

Dans ce cas, une expression qui appelle la fonction s'évalue à une valeur de son type de retour. Cette valeur correspond au résultat de l'expression suivant l'instruction `return` qui termine l'exécution de la fonction.

⚠ Conversions implicites de type dans les fonctions

Si l'expression suivant une instruction `return` ne correspond pas au type de retour de la fonction, C effectuera une conversion implicite de type. Comme mentionné précédemment, cela peut être risqué.

3.6 Exercices

Exercice 8 : Somme-produit-moyenne Écrire une fonction avec le prototype `void spm(int n);`. Si `n` est positif, la fonction doit calculer et afficher la somme, le produit et la moyenne des `n` premiers entiers sur le terminal. Si `n` est négatif, la fonction n'affiche rien.

```c
#include <stdio.h>

void spm(int n);

int main(void) {
  spm(6);
  spm(10);
  return 0;
}

  /* Définir spm ici */
```

```
$ ./spm
Somme: 21, produit: 720, moyenne: 3.
Somme: 55, produit: 3628800, moyenne: 5.
$
```

Exercice 9 : Tables de multiplication

1. Écrire une fonction avec le prototype `void mline(int n,int rx);`. Si `n` et `rx` sont strictement positifs, elle doit afficher les multiplications `1*n`, `2*n`, `3*n`, ..., `rx*n` sur une seule ligne du terminal :

```c
#include <stdio.h>

void mline(int n, int rx);

int main(void) {
  mline(4, 5);
  return 0;
}

  /* Définir mline ici */
```

```
$ ./table
4     8     12    16    20
```

Pour assurer un bon alignement des nombres, vous pouvez utiliser le spécificateur de format `%4d` au lieu de `%d` dans `printf`. Cela garantit que chaque nombre occupe au moins 4 caractères, avec des espaces ajoutés si nécessaire.

2. Définir une fonction avec le prototype `void mtable(int rx, int ry);`. Si `rx` et `ry` sont tous les deux strictement positifs, la fonction doit imprimer la table de multiplication jusqu'à `rx` horizontalement et `ry` verticalement sur le terminal. Vous pouvez réutiliser la fonction `mline` de la première question.

```c
#include <stdio.h>

void mline(int n, int rx);
void mtable(int rx, int ry);

int main(void) {
mtable(5,9);
return 0;
}

  /* Définir mline et mtable ici */
```

```
$ ./table
1     2     3     4     5
1     1     2     3     4     5
2     2     4     6     8    10
3     3     6     9    12    15
4     4     8    12    16    20
5     5    10    15    20    25
6     6    12    18    24    30
7     7    14    21    28    35
8     8    16    24    32    40
9     9    18    27    36    45
```

Exercice 10 : Plus grand chiffre Écrire une fonction `int largestDigit(int n);`. Si `n` est positif, elle retourne le plus grand chiffre contenu dans l'écriture décimale de `n`. Si `n` est négatif, elle retourne `-1`.

```c
#include <stdio.h>

int largestDigit(int n);
int main(void) {
printf("%d.\n", largestDigit(2814));
printf("%d.\n", largestDigit(536));
printf("%d.\n", largestDigit(-129));
return 0;
}
  /* Définir largestDigit ici */
```

```
$ ./largestdigit
8.
6.
-1.
```

Exercice 11 : Nombres premiers Écrire une fonction `bool isPrime(int n);`. Si `n` est positif et un nombre premier (*i.e.*, les seuls diviseurs de `n` sont `1` et `n` lui-même), elle retourne vrai. Sinon, elle retourne faux.

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

bool isPrime(int);
int main(void) {
for (int i = 0; i < 10; i++) {
if (isPrime(i)) {
printf("%d est premier.\n", i);
}
}
  return 0;
}
  /* définir isPrime ici */
```

```
$ ./prime
2 est premier.
3 est premier.
5 est premier.
7 est premier.
```

Exercice 12 : Nombres palindromes Un nombre palindrome reste le même lorsqu'on inverse ses chiffres. Par exemple, `343` est un palindrome, mais `376` ne l'est pas. Définir `bool isPalindrome(int n);` pour retourner `true` si `n` est positif et un palindrome.

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

bool isPalindrome(int);
int main(void) {
if (isPalindrome(341143)) {
printf("%d est palindrome.\n", i);
}
  if (isPalindrome(37612)) {
  printf("%d est palindrome.\n", i);
}
}
  return 0;
}
  /* définir isPalindrome ici */
```

```
$ ./palindrome
341143 est un palindrome.
```

4

Pointeurs et tableaux

Un pointeur est une variable qui stocke une adresse mémoire. Cela peut être, en particulier, l'adresse d'une autre variable. Ce concept est fondamental en C et possède plusieurs applications. L'une des principales applications concerne les tableaux, qui sont étroitement liés aux pointeurs et seront abordés dans ce chapitre. Avant cela, revenons sur la gestion des variables, ce qui nous aidera à motiver l'introduction des pointeurs.

4.1 Durée de vie des variables : la pile d'exécution

Afin de comprendre l'un des principaux usages des pointeurs, il est essentiel de clarifier le fonctionnement des variables locales en C. Commençons par revoir ce que nous avons déjà mentionné à propos de la durée de vie des variables. Supposons qu'un programme en C soit en cours d'exécution :

- Lorsqu'une variable locale est déclarée, un espace mémoire est alloué pour stocker sa valeur.
- La durée de vie d'une variable locale correspond à son bloc de code, délimité par des accolades `{}`. Lorsque l'exécution atteint l'accolade fermante, la variable est automatiquement désallouée.

? Que signifie allouer et désallouer une variable ?

Allouer une variable signifie réserver un espace mémoire pour stocker sa valeur. Tant que la variable reste allouée, cette mémoire est « protégée, » c'est-à-dire qu'elle ne sera pas utilisée pour stocker d'autres données.
Désallouer une variable signifie libérer cet espace mémoire, permettant ainsi son utilisation pour d'autres données. En revanche, la désallocation ne « supprime » pas sa. Celle-ci reste en mémoire jusqu'à ce qu'elle soit remplacée par de nouvelles données, ce qui se produit à un moment imprévisible.

La région mémoire où sont stockées les variables locales s'appelle la pile d'exécution. C'est également l'endroit où sont stockés les paramètres des fonctions.

? Qu'en est-il des variables globales ?

Les *variables globales* sont stockées dans une région mémoire distincte, indépendante de la pile d'exécution, appelée « *mémoire statique* ». Elles restent allouées pendant toute la durée d'exécution du programme.

Comme montré dans la Figure 5.1 (à droite), la pile d'exécution est organisée en cadres de pile. Chaque cadre correspond à un *appel de fonction* et est utilisé pour stocker les variables déclarées dans cette fonction. Chaque fois qu'une fonction `f` est appelée pendant l'exécution du programme, un nouveau cadre est ajouté en haut de la pile d'exécution pour `f`. Ce cadre reste sur la pile jusqu'à ce que `f` retourne.

Il est important de noter que le cadre le plus haut de la pile correspond toujours à la fonction la plus récemment appelée qui est encore en cours d'exécution. À l'inverse, le cadre le plus bas correspond toujours à la fonction initiale appelée au début du programme, à savoir la fonction `main`.

C'est pourquoi le terme « *pile* » est utilisé. Cette structure mémoire suit le principe « dernier entré, premier sorti » (LIFO pour Last In, First Out), ce qui signifie que le cadre de pile le plus récemment alloué est également le premier à être désalloué.

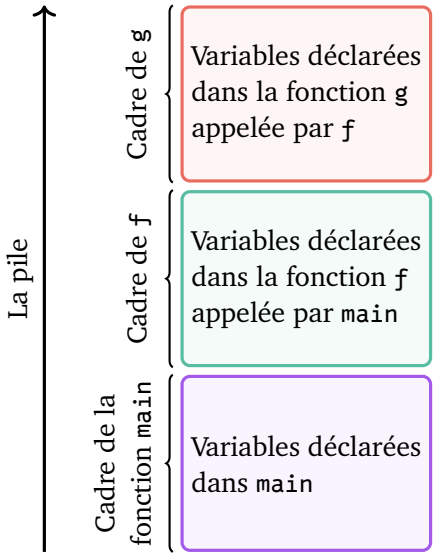


FIGURE 4.1 : La pile d'exécution en C.

4.1.1 Les appels de fonction

Lorsqu'une fonction est appelée dans un programme en C, les étapes suivantes sont effectuées :

1. Une nouvelle frame est ajoutée au sommet de la pile d'exécution. De l'espace est alloué pour stocker trois types d'objets : les arguments de la fonction (le cas échéant), ses variables locales (celles déclarées dans le bloc de code de la fonction) et la valeur retournée (s'il y en a une).
2. Les expressions passées en tant qu'arguments à la fonction sont évaluées, et les valeurs résultantes sont assignées aux variables correspondantes dans la frame de la fonction.
3. Le code de la fonction est exécuté.

Prenons l'exemple du programme suivant.

Exemple de programme qui appelle une fonction

```c
#include<stdio.h>
int modif(int a, int b);

int main(void) {
  int x = 1, y = 2, z = 5;
  int r = modif(x + y, z);
  return 0;
}

int modif(int a, int b){
  int x = b - a;
  b = 2 * x;
  return b;
}
```

Nous illustrons l'évolution de la pile d'exécution pendant l'exécution de ce programme dans la Figure 4.2.

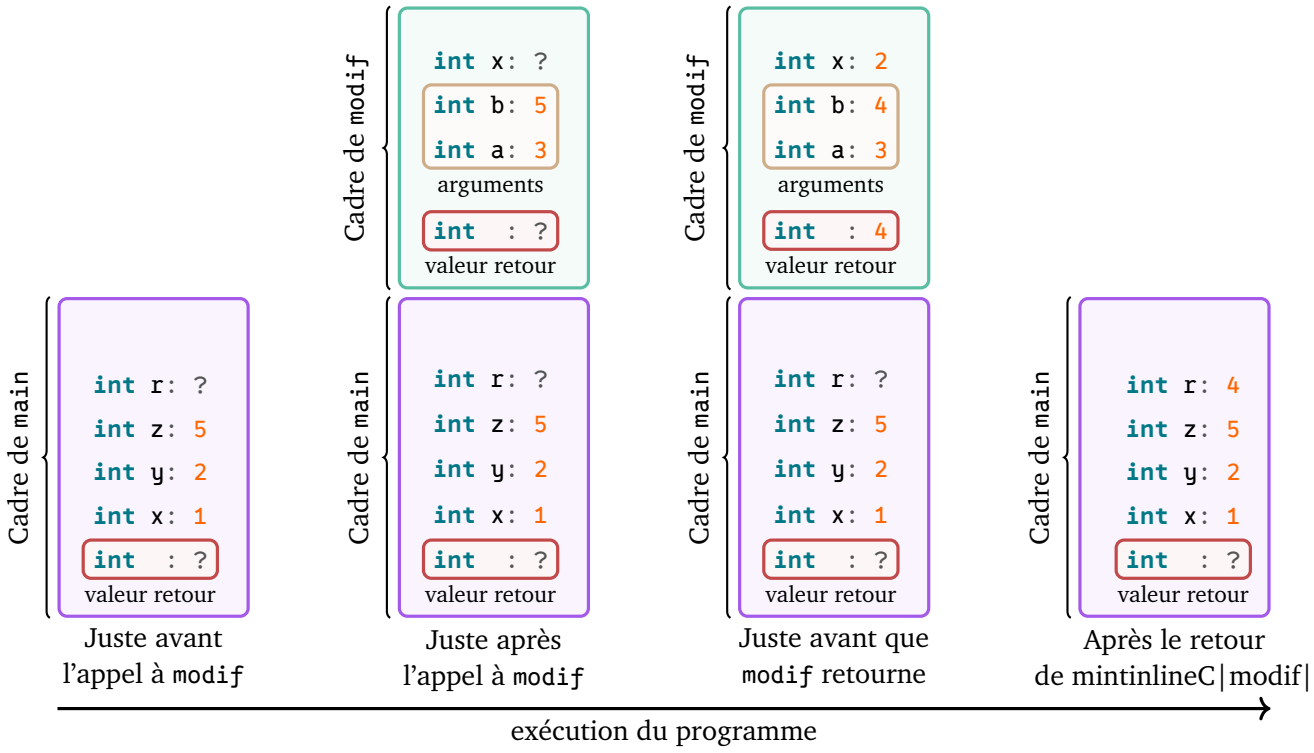


FIGURE 4.2 : Illustration du processus d'appel de la fonction `modif`.

Masquage de variables

Deux variables nommées `x` sont déclarées dans le programme ci-dessus : une dans la fonction `main` et une dans `modif`. Ces variables sont indépendantes et sont allouées dans le cadre de chaque fonction. En particulier, lorsque `modif` est en cours d'exécution, sa variable `x` masque celle de `main`.

Une fonction ne peut pas modifier ses arguments

En C, les arguments sont passés par valeur. Si un appel de fonction passe une variable comme argument, cette variable ne sera pas modifiée par l'exécution de la fonction. Au lieu de cela, la valeur de la variable est copiée et assignée à la variable locale correspondante à l'intérieur de la fonction. Toute modification n'affecte que cette variable locale. C'est ce qui se passe avec la variable `z` dans l'exemple.

Le fonctionnement de la pile d'exécution soulève une question intéressante : en C, est-il possible pour une fonction de modifier des variables déclarées avant son appel, comme celles dans `main` ? La réponse est double :

1. Avec ce que nous avons introduit jusqu'à présent, ce n'est pas possible.
2. Cependant, cela devient possible en utilisant des pointeurs introduits dans la section suivante.

4.1.2 Un aperçu rapide de la récursion

Une *fonction récursive* est une fonction qui s'appelle elle-même dans son bloc de code. Lorsqu'on utilise la récursion, plusieurs frames correspondant à la même fonction peuvent exister simultanément sur la pile d'exécution.

En effet, les mêmes règles s'appliquent aux fonctions récursives : chaque fois qu'une fonction est appelée, un nouveau cadre est ajouté en haut de la pile. Dans le cas de la récursion, chaque appel récursif crée un cadre spécifique à cet appel, qui est ensuite ajouté en haut de la pile. Ce nouveau cadre masque temporairement ceux des appels récursifs précédents.

Schémas de récursion complexes

Des schémas de récursion plus complexes sont également possibles. Par exemple, on peut définir deux fonctions, `f` et `g`, où `f` appelle `g`, et `g` appelle `f`.

Prenons un exemple. Voici le programme suivant :

Calcul de la fonction factorielle

```c
#include<stdio.h>

int facto(int n);

int main(void) {
  int r = facto(4);
  return 0;
}

int facto(int n){
  if (n <= 1) {
    return 1;
  }
  return n * facto(n-1);
}
```

Nous illustrons l'évolution de la pile d'exécution pendant l'exécution de ce programme dans la Figure 4.3 ci-dessous.

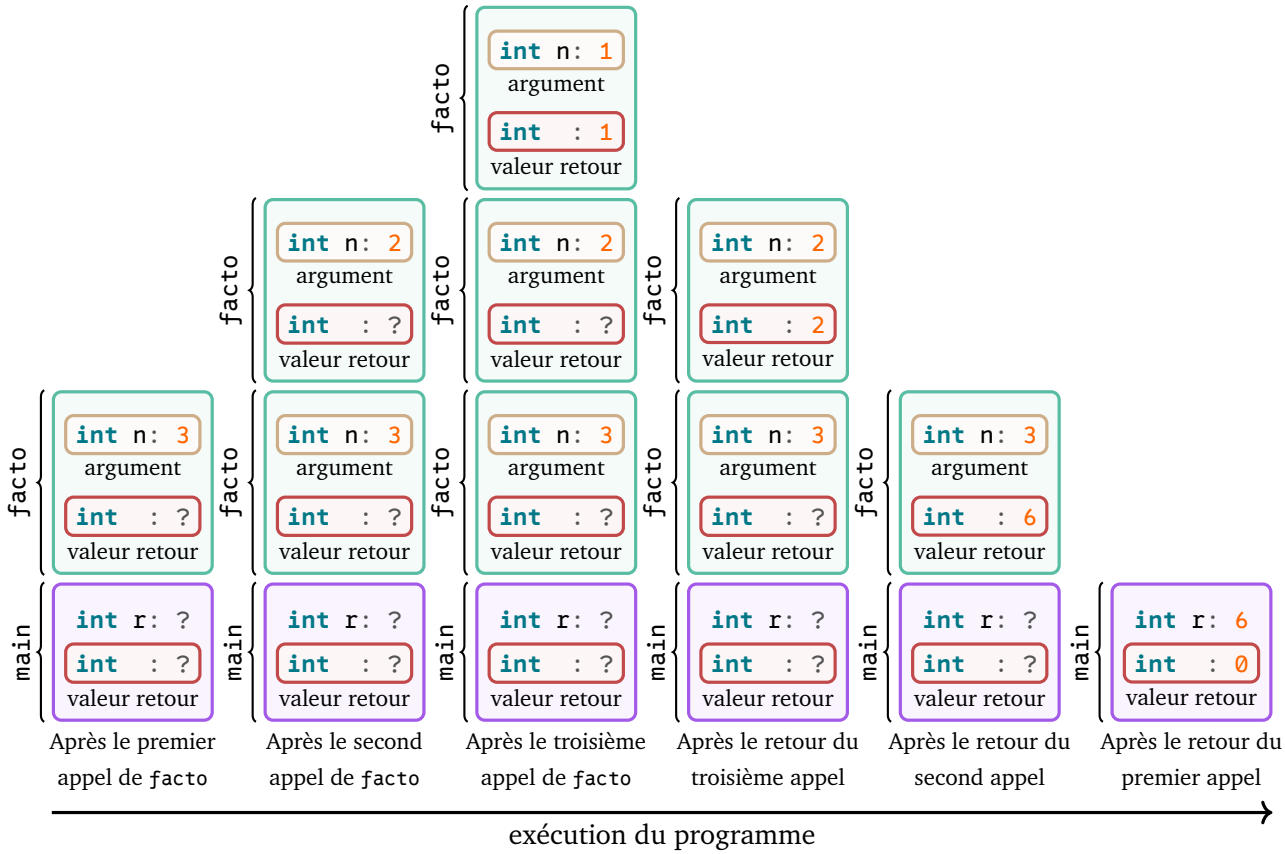


FIGURE 4.3 : Évolution de la pile d'exécution après appel de la fonction `facto`.

4.2 Les pointeurs

Lorsqu'un programme C s'exécute sur un ordinateur, l'espace mémoire qu'il utilise est une séquence continue de bits (abréviation de binary digits, ou chiffres binaires). Chaque bit ne peut prendre que l'une des deux valeurs suivantes : `0` ou `1`. Une représentation graphique est fournie dans la Figure 4.4.

0 1 0 0 1 1 0 1 0 0 1 0 1 1 0 1 0 0 1 0 1 1 1 0 ····· 1 1 0 1 1 1 1 0 0 0 1 0 1 1 0 1

FIGURE 4.4 : Espace mémoire utilisé par un programme C.

La mémoire est divisée en bytes, chaque byte étant composé de plusieurs bits. À proprement parler, le

nombre précis de bits dans un byte peut dépendre de l'architecture de l'ordinateur. Toutefois, un byte est généralement constitué de huit bits (un byte de 8 bits est aussi appelé « octet »), et c'est cette convention que nous suivrons dans nos exemples.

Un byte est considéré comme l'unité de base de la taille mémoire et constitue la plus petite unité adressable de mémoire. Cela signifie que chaque byte dans la mémoire a une adresse unique. Le premier byte a l'adresse `0`, le deuxième byte a l'adresse `1`, le troisième byte a l'adresse `2`, et ainsi de suite. Nous mettons à jour notre représentation graphique dans la Figure 4.5.

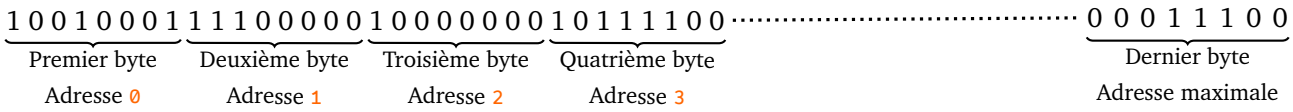


FIGURE 4.5 : Espace mémoire utilisé par un programme C - division en bytes adressables.

Toutes les données nécessaires à l'exécution d'un programme C sont stockées en mémoire. Cela inclut les valeurs des variables et même le programme compilé lui-même. La pile d'exécution est aussi stockée en mémoire. Certaines variables occupent un seul byte (comme les `char`), tandis que d'autres s'étendent sur plusieurs bytes (par exemple, les `int` utilisent généralement quatre bytes [1]). En conséquence, chaque donnée stockée en mémoire a une ad qui correspond à l'adresse de son premier byte.

Par définition, une adresse est une valeur numérique. Ainsi, une valeur stockée en mémoire peut représenter l'adresse d'une autre donnée en mémoire. C'est ce qu'on appelle un pointeur : une variable dont la valeur est interprétée comme une adresse mémoire.

4.2.1 Types de pointeurs

En C, les variables pointeurs sont déclarées de la même manière que les autres variables : le type est écrit en premier, suivi du nom de la variable, puis d'un point-virgule pour terminer l'instruction :

```
type nomVariable;
```

Ainsi, nous devons définir le type d'une variable pointeur. Avant de le faire, discutons des informations que ce type doit fournir au compilateur.

1. Premièrement, le type d'une variable pointeur doit indiquer que la variable est bien un pointeur, c'est-à-dire que sa valeur représente une adresse mémoire.
2. Deuxièmement, une fois que nous savons que la valeur d'une variable est une adresse, nous devrons probablement accéder à la valeur pointée stockée à cette adresse. Pour ce faire, nous devons connaître le type de la valeur pointée, ce qui détermine le nombre de bytes qu'elle occupe en mémoire et comment ces bytes doivent être interprétés. Par conséquent, un type de pointeur doit également inclure le type de la valeur qu'il pointe.

Un type de pointeur est composé de deux éléments : le type de la valeur qu'il pointe et le symbole `*`, qui indique qu'il s'agit d'un type pointeur. Formellement, un type de pointeur s'écrit comme suit :

```
<type pointé> *
```

On donne quelques exemples.

Déclaration de variables pointeurs

```c
/* Variable pointeur vers une valeur de type (int) */
int *x;

/* Variable pointeur vers une valeur de type (char) */
char *y;

/* Variable pointeur vers une valeur de type (double) */
double *z;
```

1. Cela dépend du compilateur. La norme C exige uniquement qu'un `int` utilise au moins deux bytes.

Pointeurs vers des pointeurs

La construction des types de pointeurs permet de déclarer des pointeurs vers des valeurs qui sont elles-mêmes des pointeurs. Par exemple, considérons les déclarations suivantes :

```
/* Variable pointeur vers une valeur de type (int *) */
int **a;

/* Variable pointeur vers une valeur de type (int **) */
int ***b;
```

Déclaration simultanée de plusieurs pointeurs

La syntaxe pour déclarer plusieurs variables dans une seule instruction peut être trompeuse lorsqu'il s'agit de pointeurs. Considérons la déclaration suivante :

```
int *q, r, s;
```

Dans cette instruction, `q` est déclaré comme un pointeur vers un `int` (type `int *`), tandis que `r` et `s` sont déclarés comme des variables non-pointeurs de type `int`.
Pour déclarer les trois variables comme des pointeurs, la syntaxe correcte est :

```
int *q, *r, *s;
```

Cela garantit que `q`, `r` et `s` sont toutes des variables pointeurs de type `int *`.

4.2.2 Manipulation des variables pointeurs

Maintenant que nous comprenons comment déclarer des variables pointeurs, nous devons expliquer comment elles sont manipulées dans un programme C. Il existe deux opérateurs essentiels ce contexte :

Référencement : Pour récupérer l'adresse mémoire d'une variable (pointeur ou non), nous utilisons l'opérateur `&`. Plus précisément, si `x` est une variable de type `<type>`, l'expression suivante :

```
&x
```

s'évalue à l'adresse mémoire de `x`. Le type de cette adresse est `<type> *`.

Déréférencement : Pour accéder à la valeur stockée à l'adresse donnée par un pointeur, nous utilisons l'opérateur `*` : si `px` est une variable pointeur de type `<type> *`, alors ce qui suit :

```
*px
```

représente la valeur stockée à l'adresse mémoire contenue dans `px`. Cette valeur a le type `<type>`.

```c
#include <stdio.h>
int main(void) {
  /* Déclarations */
  int a = 1, b = 2, c = 3;
  int *pa = &a;
  /* Pointeur. Valeur : adresse de a */
  int *pb = &b;
  /* Pointeur. Valeur : adresse de b */

  /* Affectations */
  *pa = *pb;
  pb = &pc;
  *pb = 9;

  printf("a = %d.\n", a);
  printf("b = %d.\n", b);
  printf("c = %d.\n", c);
  return 0;

}
```

```
$ ./pointeurs1
a = 2.
b = 2.
c = 9.
```

Avant que les affectations ne soient effectuées, `pa` est un pointeur vers `a`, et `pb` est un pointeur vers `b`. Décrivons précisément ce que fait chaque assignation :

1. La première remplace la valeur stockée à l'adresse pointée par `pa` par la valeur stockée à l'adresse pointée par `pb`. En conséquence, `a` prend la valeur de `b`.
2. La deuxième met à jour `pb` pour qu'il stocke l'adresse de `c`. Ainsi, `pb` pointe maintenant vers `c`.
3. La troisième remplace la valeur stockée à l'adresse pointée par `pb` par `9`. Ainsi, `c` prend la valeur `9`.

Pour mieux comprendre ce qui se passe en mémoire, on va dessiner la pile d'exécution de ce programme dans la Figure 4.6. Puisque le programmeur n'a aucun contrôle sur les adresses mémoire réelles, nous choisissons de représenter les valeurs des pointeurs à l'aide de flèches pointant vers les données qu'ils référencent. Cela indique visuellement que la véritable valeur du pointeur est l'adresse mémoire des données auxquelles il fait référence.

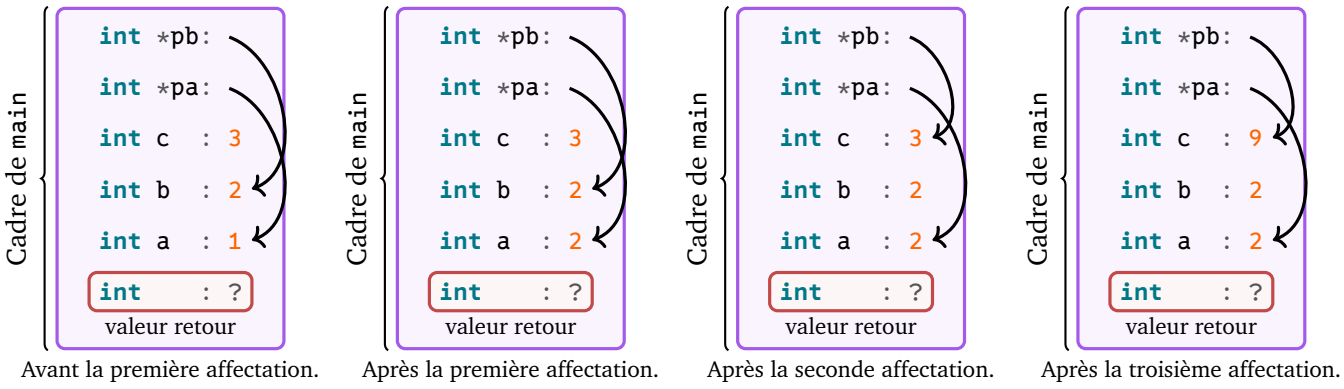


FIGURE 4.6 : Manipulation des pointeurs.

Le symbole `*`

Le symbole `*` est utilisé de deux manières différentes, ce qui peut parfois prêter à confusion :

1. Déclarer des types pointeurs. Par exemple, `int *` est le type d'un pointeur vers un `int`.
2. Déréférencement : accéder à la valeur stockée à l'adresse contenue dans un pointeur. Si `px` est une variable de type `int *`, alors `*px` représente la valeur entière stockée à l'adresse détenue par `px`.

Ces deux usages sont indépendants et ne doivent pas être confondus. Dans l'instruction suivante :

```
int *pa;
```

le `*` fait partie de la déclaration de type, ce qui signifie que `pa` est une variable de type (un pointeur vers un `int`). Cet usage indique que `pa` doit stocker l'adresse d'un entier, comme dans :

```
int *pa = &n; /* avec n étant une variable entière déclarée */
```

D'autre part, dans l'instruction suivante : `*pa = 42;`

le `*` agit en tant qu'opérateur de déréférencement, ce qui signifie que `*pa` fait référence à la valeur stockée à l'adresse mémoire détenue par `pa`. Cet usage permet de modifier la valeur stockée à cette adresse mémoire.

Dessiner la mémoire aide à comprendre les pointeurs

Pour comprendre le fonctionnement d'un programme utilisant des pointeurs, il est très utile de dessiner la mémoire, comme dans la Figure 4.6. Cela est explicitement demandé dans plusieurs exercices.

Variables pointeurs non initialisées - La valeur NULL

Lorsqu'une variable est laissée non initialisée, sa valeur est ce qui a été précédemment stocké à son adresse mémoire. Cela signifie que le programmeur n'a aucun contrôle sur sa valeur initiale. Cela peut être particulièrement dangereux avec les pointeurs. Considérez les instructions suivantes :

```
int *p;
*p = 42;
```

Étant donné que `p` n'a pas été initialisé, la deuxième instruction tente d'accéder à une adresse mémoire imprévisible. Cela peut conduire à deux résultats possibles :

1. Si l'adresse correspond à une zone restreinte de la mémoire, l'accès est interdit, ce qui entraîne un crash du programme – généralement avec un message de type `segmentation fault`.
2. Si l'adresse correspond à l'espace mémoire d'une autre variable, le programme pourrait modifier involontairement cette variable, ce qui entraînerait un comportement indéfini.

Les deux résultats sont fortement indésirables et doivent être évités. Une manière simple de prévenir ce problème est d'utiliser une valeur spéciale pour le pointeur : `NULL`.
La valeur `NULL` indique qu'un pointeur ne référence pas un objet valide (en pratique, c'est un alias pour l'adresse `0`, qui est toujours invalide). Les pointeurs peuvent être initialisés à `NULL` pour signaler qu'ils n'ont pas encore été affectés à une adresse valide. Cela permet de réaliser des vérifications de sécurité :

```
int *p = NULL;

/* Instructions (possiblement assigner une adresse valide à p) */

if (p != NULL) {
  *p = 42;
}
```

En vérifiant si `p` n'est pas `NULL` avant d'utiliser l'opérateur de déréférencement `*`, nous nous assurons que le pointeur contient une adresse valide, évitant ainsi les erreurs potentielles.

4.2.3 Fonctions dont les arguments sont des pointeurs

Nous pouvons maintenant revenir à la question que nous avons soulevée précédemment : en C, est-il possible pour une fonction de modifier des variables déclarées avant qu'elle ne soit appelée ? Cela devient possible si un ou plusieurs arguments de la fonction sont des pointeurs. Considérons le programme suivant :

```
#include <stdio.h>

void plus(int *p);

int main(void) {
  int n = 1;
  plus(&n);
  printf("n = %d.\n", n);
  return 0;
}

void plus(int *p) {
  (*p)++;
}
```

```
$ ./pointers2
n = 2.
```

Dans ce programme, la fonction `plus` modifie avec succès la variable `n` déclarée dans la fonction `main`. Cela fonctionne parce que `plus` prend un paramètre de type pointeur `int *` et est appelée avec l'adresse de `n` en argument. Pour mieux comprendre ce qui se passe, on dessine la pile d'exécution dans la Figure 4.7.

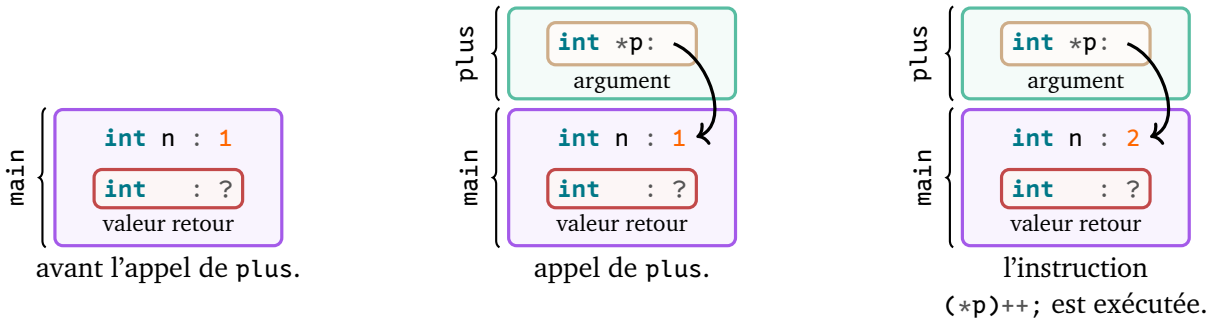


FIGURE 4.7 : Pointeurs et fonctions.

4.2.4 Exercices

Exercice 13 : The great swap Que produisent les programmes suivants ? Dessiner la mémoire.

```c
#include <stdio.h>

void swap1(int a, int b);

int main(void) {
  int a = 1, b = 2;
  swap1(a, b);
  printf("a = %d, b= %d.\n", a, b);
  return 0;
}
void swap1(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}
```

```c
#include <stdio.h>

void swap2(int *a, int *b);

int main(void) {
  int a = 1, b = 2;
  swap2(&a, &b);
  printf("a = %d, b= %d.\n", a, b);
  return 0;
}
void swap2(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
```

```c
#include <stdio.h>

void swap3(int *a, int *b);

int main(void) {
  int a = 1, b = 2;
  swap3(&a, &b);
  printf("a = %d, b= %d.\n", a, b);
  return 0;
}
void swap3(int *a, int *b) {
  int *temp = a;
  a = b;
  b = temp;
}
```

```c
#include <stdio.h>

void swap4(int *a, int *b);

int main(void) {
  int a = 1, b = 2;
  swap4(&a, &b);
  printf("a = %d, b= %d.\n", a, b);
  return 0;
}
void swap4(int *a, int *b) {
  int *temp;
  *temp = *a;
  *a = *b;
  *b = temp;
}
```

Exercice 14 : Manipulation des pointeurs Considérez le programme C suivant :

```c
#include <stdio.h>

int main(void){
  int a = 1, b = 2, c = 3, d = 4;
  int *p1 = &a;
  int *p2 = &b;
  int *p3 = &c;
  int *p4 = &d;
  int *p5;

  p5 = p3;
  *p3 = *p2;
  *p2 = *p5;
  *p4 = *p1;
  *p1 = *p4;
  printf("a, b ,c et d sont égaux à : %d, %d, %d et %d\n",a,b,c,d);
  return 0;
}
```

1. Dessiner l'état de la mémoire après que toutes les variables aient été déclarées.
2. Que produit le programme ?

Exercice 15 : Pointeurs sur pointeurs Considérez le programme C suivant :

```c
#include <stdio.h>

int main(void){
  int a = 1, b = 2, c = 3, d = 4;
  int *p1 = &a;
  int **p2 = &p1;
  int *p3 = &c;
  int *p4 = &b;
  int **p5 = &p3;
  int *p6 = &d;

  *p3 = **p2+6;
  *p2 = p3;
  *p1 = *p4  * 5;
  p5 = &p6;
  **p5 = **p5 + **p2;
  *p4 = *p3 + 14;
  *p5 = p1;
  **p2 = 1 + **p2;
  **p5 = **p5 * **p2;

  printf("a, b ,c et d sont égaux à : %d, %d, %d et %d\n",a,b,c,d);
  return 0;
}
```

1. Dessiner l'état de la mémoire après que toutes les variables aient été déclarées.
2. Redessiner la mémoire après que toutes les affectations aient été réalisées.
3. Que produit le programme ?

Exercice 16 : Fonctions et pointeurs Considérez le programme C suivant :

```c
#include <stdio.h>

int x1, y1, y2, z1, z2;
int *px1 = &x1;
int *py1 = &y1;
int *py2 = &y2;
int *pz1 = &z1;
int *pz2 = &z2;

void aux(int **p,int ***q);

void aux(int **p, int ***q)
{
  **p = 2;
  ***q= 99;
  *p = pz1;
  **p = 12;
  p = &py2;
  *q = &px1;
  **p = 4;
  ***q = 42;
}
```

```c
int main(void)
{
  int **p;
  int **q;
  int ***r;
  int s;
  int *t;

  p     = &py1;
  q     = &px1;
  r     = &q;
  *p  = &s;
  **q   = 1;
  **p   = 34;
  ***r  = 76;
  t     = **r;
  r     = &p;

  aux(p,&q);
  printf("%d %d %d %d %d \n",
         **q, **p, ***r, s, *t);
  return 0;
}
```

1. Dessiner l'état de la mémoire après que toutes les variables ont été déclarées.
2. Redessiner la mémoire après que toutes les affectations aient été réalisées.
3. Redessiner la mémoire après l'appel de la fonction aux.
4. Que produit le programme ?

4.3 Passage d'arguments par pointeur au lieu de retourner des valeurs

Avec l'introduction des pointeurs, nous pouvons présenter une technique courante de programmation. Illustrons cela par un exemple. Supposons que nous devions écrire une fonction qui prend deux valeurs `int` en argument et calcule à la fois leur somme et leur produit. En C, une fonction ne peut retourner qu'une seule valeur directement, ce qui rend apparemment impossible de retourner les deux résultats simultanément. Cependant, nous pouvons contourner cette limitation en passant des arguments supplémentaires de type pointeur à la fonction.

L'idée principale est qu'au lieu de retourner des valeurs, la fonction peut modifier des variables existantes en utilisant leurs adresses mémoire, qui sont passées en argument. Cette approche permet de « retourner » plusieurs valeurs. Illustrons cela en écrivant notre programme :

```c
#include <stdio.h>

void sp(int a, int b, int *ps, int *pm);
/* a et b sont les "vrais" arguments */
/* Les résultats sont écrits aux adresses
indiquées par les pointeurs ps et pm */

int main(void){
  int sum, mul;
  sp(3, 14, &sum, &mul);
  printf("Somme : %d.\n", sum);
  printf("Produit : %d.\n", mul);
  return 0;
}

void sp(int a, int b, int *ps, int *pm){
  *ps = a + b;
  *pm = a * b;
}
```

```
$ ./sumprod
Somme : 17.
Produit : 42.
```

⚠ Appeler une fonction avec des arguments de type pointeur

Lors de l'utilisation de cette technique, appeler la fonction nécessite un peu plus de préparation. Comme montré dans l'exemple ci-dessus, il faut d'abord déclarer les variables que la fonction va modifier. Ensuite, il faut passer les adresses de ces variables à la fonction, ce qui lui permet de les modifier.

🎓 Exercice 17 : Chiffres minimum et maximum

1. Écrivez une fonction avec le prototype suivant :

   ```c
   int minmax_digits(int k, int *p_min, int *p_max);
   ```

   Cette fonction doit déterminer le plus petit et le plus grand chiffre de la représentation décimale de l'entier `k` et les stocker aux adresses pointées par `p_min` et `p_max`.
   Par exemple, si `k = 52233678`, le chiffre minimum est `2` et le chiffre maximum est `8`.
2. Écrivez un programme complet qui appelle la fonction `minmax_digits` avec `k = 52233678` et affiche les résultats à la sortie standard.

4.4 Tableaux

Maintenant que nous comprenons les pointeurs, nous pouvons introduire les *tableaux*. En C, un *tableau* est une collection de valeurs du même type, stockées dans des emplacements mémoire contigus. Considérons la déclaration suivante :

```c
int myarray[3]; /* Déclare un tableau de trois valeurs de type int */
```

Lorsque cette instruction est exécutée, *trois valeurs consécutives de type* `int` *sont allouées sur la pile d'exécution*. Cela signifie que la deuxième valeur est stockée immédiatement après la première, et la troisième suit

directement après la deuxième. Ainsi, les tableaux sont souvent représentés comme dans la Figure 4.8.

23	7	54	42	1	-8	-3	76

FIGURE 4.8 : Représentation d'un tableau contenant les `int` 23, 7, 54, 42, 1, -8, -3 et 76 (dans cet ordre).

Cette brève introduction des tableaux soulève plusieurs questions importantes qui doivent être résolues avant que nous puissions utiliser ces structures de données :

1. Dans l'exemple ci-dessus, nous avons déclaré un tableau en utilisant le nom de variable `myarray`. Que représente exactement cette variable et quel est son type ?
2. Comment accéder aux valeurs individuelles stockées dans un tableau pour les lire ou les modifier ?
3. Les tableaux peuvent-ils être passés en argument aux fonctions ?

4.4.1 Déclaration : les tableaux en tant que pointeurs

Pour déclarer un nouveau tableau, il faut spécifier le type commun des valeurs qu'il va stocker. Ce type est suivi d'un nom de variable, d'une expression entourée de crochets `[...]`, et d'un point-virgule :

```
<type des valeurs individuelles> nomVariable[<expression>];
```

Ici, l'expression doit s'évaluer à un entier strictement positif, représentant le nombre de valeurs que le tableau va stocker (nous parlons de la *taille* du tableau). Lorsque cette instruction est exécutée, *le nombre spécifié de valeurs du type indiqué est alloué consécutivement sur la pile d'exécution*.

```c
int array1[42];      /* Déclare un tableau de 42 valeurs de type int */
double array2[2+7]; /* Déclare un tableau de 9 valeurs de type double */
int n = 12;
int *array3[n];      /* Déclare un tableau de 12 valeurs de type int **/
```

Cela nous amène à la première question : que représente exactement la variable utilisée dans la déclaration et quel est son type ? L'explication fournie par la norme C peut être quelque peu déroutante. Cependant, la meilleure manière de la comprendre est que C n'a pas de type « tableau » distinct. En réalité, il n'existe pas de véritables « variables de tableau » en C. Considérons le programme suivant :

```c
#include <stdio.h>

int main(void){
  int arr[4];
  return 0;
}
```

Dans ce programme, la variable `arr` ne stocke pas directement le tableau lui-même. En réalité, `arr` est un pointeur[2] de type `int *`. Sa valeur est l'adresse mémoire du premier élément du tableau alloué. La Figure 4.9 illustre l'état de la mémoire après cette déclaration.

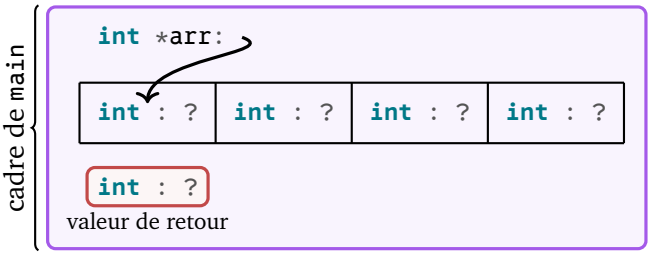


FIGURE 4.9 : État de la mémoire résultant de la déclaration d'un tableau avec `int arr[4];` dans la fonction `main` (le tableau est non initialisé).

2. Techniquement, cela n'est pas entièrement exact selon la norme C. Cependant, dans la pratique, c'est ainsi que les variables de tableau se comportent dans presque tous les cas, ce qui en fait la meilleure manière de les comprendre.

Notez que dans la Figure 4.9, la seule variable est le pointeur `arr` de type `int *`, qui stocke l'adresse du premier élément du tableau. Le tableau lui-même n'est pas directement représenté par une variable.

Cet exemple illustre ce qui se passe lorsqu'un tableau est déclaré sur la pile d'exécution. Plus précisément, lorsqu'un tableau est déclaré en utilisant la syntaxe suivante :

```
<type des valeurs individuelles> nomVariable[<expression>];
```

la variable `variableName` est un pointeur qui stocke l'adresse du premier élément du tableau alloué. Ce fait a des conséquences importantes pour les fonctions, un point que nous aborderons plus tard.

Différences entre la déclaration de tableaux et de pointeurs

Considérez les deux déclarations suivantes :

```c
int *p;
```

```c
int a[4];
```

Comme expliqué précédemment, les variables `p` et `a` déclarées dans ces instructions sont toutes deux de type `int *`. Cependant, il y a deux différences importantes entre ces deux déclarations.

1. Il existe des différences importantes dans l'allocation mémoire :
   - L'instruction `int *p;` alloue de la mémoire pour une seule adresse : la valeur de `p`. De plus, `p` est non initialisé, ce qui signifie que sa valeur est inconnue et probablement une adresse invalide.
   - L'instruction `int a[4];` alloue de la mémoire pour un tableau de quatre valeurs `int` (voir la Figure 4.9). De plus, `a` est automatiquement initialisé à l'adresse du premier élément du tableau.
2. Contrairement à un pointeur classique, `a` est un pointeur constant, ce qui signifie que sa valeur ne peut pas être changée–il doit toujours pointer vers le premier élément du tableau alloué. Tenter d'affecter une nouvelle adresse à `a` entraîne une erreur de compilation. Le code suivant est invalide :

```c
int main(void) {
  int a[4];
  int b[4];
  a = b;  /* Erreur */
  return 0;
}
```

```
$ gcc test.c -o test -Wall
test.c:7:7: error: array type 'int[4]'
 ↪  is not assignable
4 |     a = b;
|     ~ ^
1 error generated.
```

Enfin, lorsqu'un tableau est déclaré, il peut être initialisé simultanément–soit partiellement, soit entièrement–en utilisant des accolades `{}`. La syntaxe pour l'initialisation est :

```
<type des valeurs individuelles> nomVariable[<expression>] = {value1, value2, ..., valueN};
```

Par exemple, considérez le programme suivant :

```c
#include <stdio.h>

int main(void) {
  int arr[4] = { 9, 8, 3, 5 };
  return 0;
}
```

La Figure 4.10 illustre l'état de la mémoire résultant de cette déclaration de tableau lors de l'exécution.

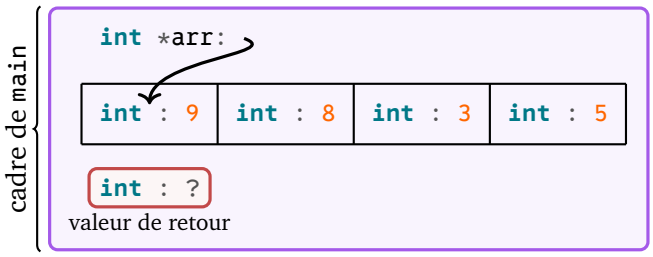


FIGURE 4.10 : État mémoire après déclaration du tableau `int arr[4] = { 9, 8, 3, 5 };` dans `main()`.

Enfin, l'initialisation peut être partielle. Par exemple, la déclaration suivante,

```
int arr[4] = { 9, 8 };
```

initialise les deux premiers éléments du tableau à `9` et `8`. Les éléments restants sont initialisés à `0`.

4.4.2 Accéder aux valeurs dans un tableau

Maintenant que nous savons déclarer des tableaux et comprenons leur nature, il est important d'apprendre comment accéder et modifier les valeurs individuelles stockées dans un tableau.

Si un tableau nommé `arr` est déclaré (ce qui signifie que `arr` est un pointeur vers le premier élément du tableau), alors on accède à l'élément se trouvant à l'index `i` dans ce tableau avec la syntaxe suivante :

```
arr[i]
```

⚠ Indexation des tableaux en C

En C, les indices des tableaux commencent à zéro et vont jusqu'à la taille du tableau moins un. Par exemple, considérons la déclaration suivante : `int arr[n];`. Ici, les indices valides vont de `0` à `n-1` :

- Le premier élément a l'indice `0` et est accessible avec `arr[0]`.
- Le deuxième élément a l'indice `1` et est accessible avec `arr[1]`.
- Le dernier élément a l'indice `n-1` et est accessible avec `arr[n-1]`.

Cette convention d'indexation est une source fréquente d'erreurs, alors soyez toujours prudent.

Illustrons ce concept avec un exemple. Dans le programme suivant, nous déclarons un tableau de taille `12`. Pour chaque indice valide `i < 12` dans le tableau, nous stockons la somme de tous les entiers de `0` à `i` à cet indice. Enfin, nous imprimons toutes les valeurs du tableau à la sortie standard.

```c
#include <stdio.h>

int main(void) {
  int arr[12];
  arr[0] = 0;
  for (int i = 1; i < 12; i++) {
  arr[i] = arr[i - 1] + i;
  }
  for (int i = 0; i < 12; i++) {
  printf("arr[%d]: %d\n", i, arr[i]);
  }
  return 0;
}
```

```
$ ./testarray
arr[0]: 0
arr[1]: 1
arr[2]: 3
arr[3]: 6
arr[4]: 10
arr[5]: 15
arr[6]: 21
arr[7]: 28
arr[8]: 36
arr[9]: 45
arr[10]: 55
arr[11]: 66
```

Comment tout cela fonctionne ? - Arithmétique des pointeurs

Examinons comment C accède aux éléments d'un tableau. Considérons la déclaration suivante :

```
int myarray[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Comme mentionné précédemment, la variable `myarray` est un pointeur de type `int *`, qui pointe vers le premier élément du tableau. De plus, les éléments d'un tableau sont stockés de manière contiguë en mémoire, ce qui signifie que leurs adresses suivent un schéma prévisible.

- L'adresse du premier élément est simplement la valeur du pointeur `myarray`. Accéder à cet élément revient à déréférencer le pointeur : `*myarray`.
- L'adresse du deuxième élément est l'adresse du premier élément plus le nombre de bytes nécessaires pour stocker un seul `int`. De même, l'adresse du troisième élément est l'adresse du premier élément plus l'espace requis pour deux valeurs `int`, et ainsi de suite.

En conséquence, le calcul des adresses de tous les éléments du tableau devient une question d'*arithmétique des pointeurs*, que C gère automatiquement. Plus précisément, si `n` est une valeur entière (de type `int`), alors l'expression suivante est valide :

```
myarray + n
```

Cette expression évalue à l'adresse obtenue en ajoutant la mémoire nécessaire pour stocker `n` éléments de type `int`[3] à l'adresse stockée dans `myarray`. Par exemple, si les valeurs de type `int` occupent *4 octets* en mémoire, alors l'expression `myarray + 5` s'évalue à une adresse qui se trouve à $5 \times 4 = 20$ octets de l'adresse stockée dans `myarray`. Cela correspond exactement à l'adresse du sixième élément du tableau.
Par conséquent, l'adresse de l'élément à l'indice `i` dans notre tableau est calculée par l'expression `myarray + i`, et l'élément lui-même est accédé en déréférenciant ce pointeur : `*(myarray + i)`.
En fait, la notation `myarray[i]` introduite précédemment est simplement un alias de `*(myarray + i)`, et les deux peuvent être utilisés de manière interchangeable. Cependant, il est fortement recommandé d'utiliser `myarray[i]` uniquement pour la clarté et la lisibilité de votre code.

4.4.3 Passage des tableaux aux fonctions

Voyons maintenant comment écrire des fonctions qui prennent des tableaux comme arguments. Puisque la variable utilisée pour manipuler un tableau est en réalité un pointeur, un « argument de tableau » dans une fonction est simplement un pointeur. Par exemple, si une fonction est destinée à traiter un tableau de valeurs `int`, le paramètre correspondant doit avoir le type `int *`. Cela a deux conséquences importantes :

Premièrement, un tableau ne connaît pas intrinsèquement sa taille. Lorsqu'une fonction reçoit un argument de type `int *`, deux informations cruciales manquent :

- Le pointeur pointe-t-il vers une seule variable `int`, ou pointe-t-il vers le premier élément d'un tableau ? La fonction elle-même ne peut pas le déterminer ; c'est au programmeur de l'utiliser correctement.
- Si le pointeur représente le premier élément d'un tableau, quelle est la taille de ce tableau ? Le pointeur seul ne fournit pas cette information.

Cependant, une fonction a besoin de la taille du tableau pour fonctionner correctement. Par exemple, les indices valides pour un tableau dépendent de sa taille (ils vont de `0` à `taille - 1`). En pratique, cela signifie que la fonction doit se voir explicitement fournir la taille du tableau.

Pour manipuler correctement un tableau dans une fonction, on utilise généralement deux paramètres :

1. Un pointeur vers le premier élément du tableau.
2. Un `int` représentant la taille du tableau.

Illustrons cette convention par un exemple. La fonction suivante prend un tableau de valeurs `int` en argument, calcule la somme de ses éléments et retourne le résultat.

3. Dans ce cas, l'expression ajoute la mémoire nécessaire pour stocker `n` valeurs `int` parce que `myarray` est de type `int *`. Ce comportement s'applique également à d'autres types de pointeurs. Par exemple, si `myarray` est de type `double *`, l'expression `myarray + n` ajoute la mémoire nécessaire pour stocker `n` valeurs `double`.

```c
int somme_tableau(int *a, int n){
/* a est un pointeur vers le premier élément du tableau */
/* n est la taille de ce tableau */
int res = 0;
for(int i = 0; i < n; i++) {
  res = res + a[i];
}
return res;
}
```

Assurer la validité des arguments est de la responsabilité du programmeur

Il est tout à fait possible d'appeler la fonction `somme_tableau` avec des paramètres invalides. Par exemple, le programme suivant est syntaxiquement correct :

```c
#include <stdio.h>

int somme_tableau(int *a, int n);
int main(void) {
  int arr[5] = {1, 2, 3, 4, 5};
  printf("Et la somme est %d.\n", somme_tableau(arr, 10));
  return 0;
}
```

Cependant, ce programme risque de planter lors de son exécution. En effet, la fonction `somme_tableau` est appelée avec une valeur incorrecte pour le paramètre `n`, qui est censé représenter la taille du tableau. Le tableau réel ne contient que 5 éléments, mais la fonction est appelée pour en traiter 10. Comme `somme_tableau` n'a aucun moyen de vérifier cela, elle tentera d'accéder à une mémoire hors limites, ce qui peut provoquer un crash.
Il est de la responsabilité du programmeur de s'assurer que `somme_tableau` soit appelée avec des arguments valides–spécifiquement, un pointeur vers le premier élément d'un tableau et la taille correcte de celui-ci.

Un alias pour les types d'argument de tableau

Lors de la déclaration de la fonction `somme_tableau`, une syntaxe alternative peut être utilisée :

```c
int somme_tableau(int a[], int n);
```

On a remplacé la déclaration du paramètre `int *a` par `int a[]`. Ces deux notations sont équivalentes et peuvent être utilisées de manière interchangeable. En réalité, `int a[]` est simplement un alias pour `int *a` ; les deux indiquent que le paramètre est un pointeur vers le premier élément d'un tableau.

De plus, étant donné qu'une fonction qui prend un tableau comme argument reçoit en réalité un pointeur vers son premier élément, elle a la possibilité de modifier le tableau original. En effet, la fonction opère directement sur les emplacements mémoire des éléments du tableau. Considérons le programme suivant :

```c
#include <stdio.h>

void prod_tab(int *a, int n);

int main(void) {
int arr[4] = {2, 2, 1, 2};
prod_tab(arr, 4);
printf("Résultat %d, %d, %d, %d.\n", arr[0], arr[1], arr[2], arr[3]);
return 0;
}

void prod_tab(int *a, int n) {
for (int i = 1; i < n; i++) {
  a[i] = a[i] * a[i - 1];
}
}
```

Lors de son exécution, ce programme produit la sortie suivante sur le terminal :

```
Résultat 2, 4, 4, 8.
```

Ce comportement se produit parce que la fonction `prod_tab` modifie le tableau : chaque élément est remplacé par le produit de toutes les valeurs précédentes. Pour mieux comprendre comment cette modification se produit, on dessine l'état de la pile d'exécution après l'appel de la fonction `prod_tab`.

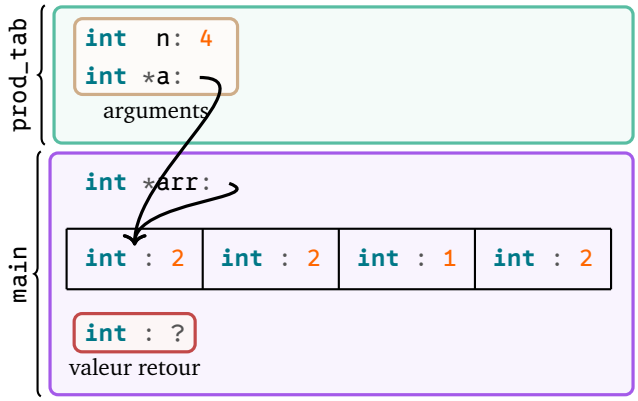


FIGURE 4.11 : État de la mémoire après l'appel de `prod_tab`.

4.4.4 Un tableau alloué sur la pile d'exécution ne peut pas être retourné par une fonction

Une limitation importante des tableaux en C est qu'ils ne peuvent pas être retournés par une fonction s'ils sont alloués sur la pile d'exécution. Considérons le programme suivant :

```c
#include <stdio.h>

int *iota(int n);

int main(void) {
int *arr = iota(3);
printf("Résultat %d, %d, %d, %d.\n", arr[0], arr[1], arr[2], arr[3]);
return 0;
}

int *iota(int n) {
int r[n];
for (int i = 0; i < n; i++) {
  r[i] = i;
}
return r;
}
```

Lors de son exécution, ce programme causera très probablement une erreur lorsque l'instruction `printf` sera exécutée. Le problème survient car, à ce moment-là, l'adresse stockée dans le pointeur `arr` correspond probablement à une zone mémoire invalide. Voici ce qui se passe étape par étape :

1. Lorsque l'instruction `int *arr = iota(3);` est exécutée, la fonction `iota` crée un tableau de taille `3` dans son propre cadre de pile.
2. Cependant, cette allocation est locale à la fonction.
3. Lorsque la fonction retourne, son cadre est supprimé et la mémoire allouée pour le tableau est invalidée.
4. En conséquence, le pointeur renvoyé fait maintenant référence à une zone mémoire invalide, entraînant un comportement indéfini lors de son accès.

Illustrons ces étapes. Nous représentons la pile d'exécution pour ce programme dans la Figure 4.12.

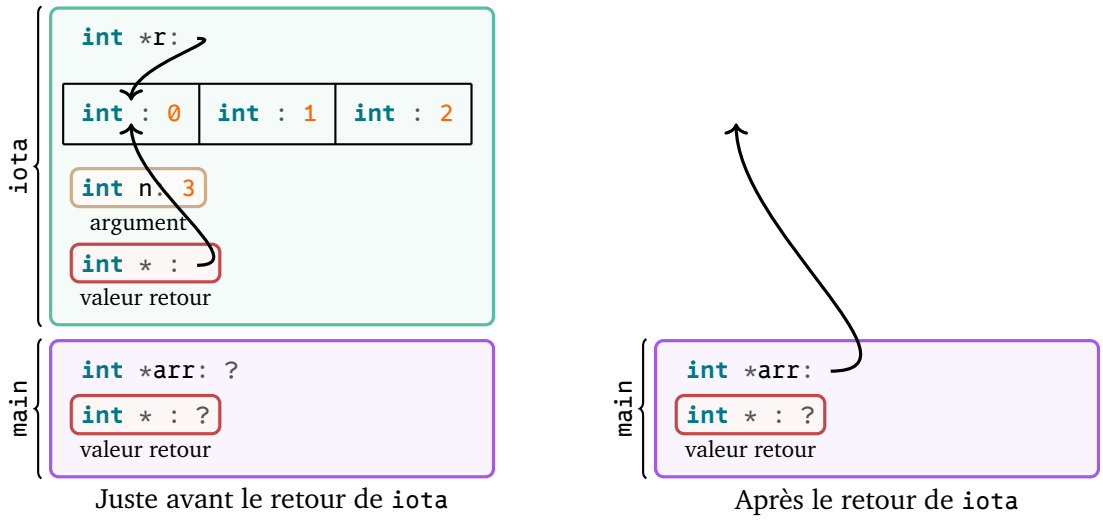


FIGURE 4.12 : État de la mémoire après l'appel de la fonction `iota`.

Une solution : gestion dynamique de la mémoire

Il est possible de « retourner » un tableau en C, mais cela nécessite une notion que nous n'avons pas abordé : *l'allocation dynamique*. En plus de la pile, C fournit une autre région mémoire où les données peuvent être allouées : le *tas*. Contrairement aux données allouées sur la pile, qui sont automatiquement désallouées lorsque la fonction retourne, les données allouées sur le tas persistent jusqu'à ce qu'elles soient explicitement désallouées par le programmeur. En utilisant l'allocation dynamique, nous pouvons allouer un tableau sur le tas à l'intérieur d'une fonction et retourner un pointeur vers celui-ci de manière sécurisée. Cependant, c'est au programmeur de libérer la mémoire allouée une fois qu'elle n'est plus nécessaire.

4.5 Exercices

Exercise 18 : Produit Scalaire Écrivez une fonction avec le prototype suivant :

```
int scalaire(int *u, int *v, int n);
```

Les arguments de cette fonction sont deux tableaux de valeurs `int`, `u` et `v`, tous deux de taille `n`. La fonction doit calculer et retourner leur produit scalaire.
Par exemple, si `u` contient les valeurs `3`, `2`, `4` et `v` contient les valeurs `2`, `-3`, `5`, la fonction doit retourner :

$$3 \times 2 + 2 \times (-3) + 4 \times 5 = 20$$

Exercise 19 : Miroir Écrivez une fonction avec le prototype suivant :

```
void miroir(int *arr, int n);
```

Les arguments de cette fonction sont un tableau de valeurs `int` et sa taille `n`. La fonction doit modifier le tableau en inversant l'ordre de ses éléments.
Par exemple, si `arr` contient les valeurs `3`, `2`, `4`, `7`, `8` (dans cet ordre), la fonction doit le modifier de sorte qu'il contienne `8`, `7`, `4`, `2`, `3` (dans cet ordre).

Exercise 20 : Tri par Sélection Dans cet exercice, nous allons écrire une fonction pour trier un tableau d'entiers dans l'ordre croissant. Cela se fait en deux étapes :

1. Écrivez une fonction avec le prototype suivant :

   ```
   void echanger_min(int *arr, int i, int n);
   ```

   Les arguments de cette fonction sont un tableau de valeurs `int`, un indice valide `i` dans ce tableau, et la taille `n` du tableau. La fonction doit trouver la valeur minimale parmi les éléments de l'indice `i`

à `n-1` et l'échanger avec la valeur à l'indice `i`.
Par exemple, si `arr` contient les valeurs `2, 3, 9, 7, 5, 6` et `i = 2`, la fonction doit échanger `9` avec `5`, ce qui donne `2, 3, 5, 7, 9, 6`.

2. Écrivez une fonction avec le prototype suivant :

```
void tri_selection(int *arr, int n);
```

Les arguments de cette fonction sont un tableau de valeurs `int` et sa taille `n`. La fonction doit trier le tableau dans l'ordre croissant. Réutilisez la fonction écrite à la première question.

Exercise 21 : Recherche Écrivez une fonction avec le prototype suivant :

```
bool rechercher(int *arr, int n, int val);
```

Les arguments de cette fonction sont un tableau de valeurs `int`, sa taille `n`, et une valeur `int` arbitraire `val`. La fonction doit retourner une valeur booléenne indiquant si `val` est présente dans le tableau.
Par exemple, si `arr` contient les valeurs `2, 3, 9, 7, 5, 6`, la fonction doit retourner `true` pour `val = 7` et `false` pour `val = 1`.
Essayez d'écrire un programme efficace en minimisant le nombre d'instructions exécutées.

Exercise 22 : Produit maximal de trois Écrivez une fonction avec le prototype suivant :

```
int prod_max(int *arr, int n);
```

Les arguments de cette fonction sont un tableau de valeurs `int` et sa taille `n`. La fonction doit calculer et retourner le produit maximal obtenu en multipliant n'importe quel trio d'éléments du tableau. Notez que le tableau peut contenir des entiers *négatifs*. Par exemple, si `arr` contient les valeurs `2, 3, -9, -7, 5, 6`, la fonction doit retourner `378`, ce qui est le produit de $-9 \times -7 \times 6$.

Exercise 23 : Zéros consécutifs Écrivez une fonction avec le prototype suivant :

```
int max_zero_consecutifs(int *arr, int n);
```

Les arguments de cette fonction sont un tableau de valeurs `int` et sa taille `n`. La fonction doit calculer et retourner le nombre maximal de zéros consécutifs dans le tableau. Par exemple, si `arr` contient les valeurs `0, 4, 0, 0, 0, 1, 2, 0, 0, 0, 0, 9`, la fonction doit retourner `4`.

Exercise 24 : Rotation Écrivez une fonction avec le prototype suivant :

```
void rotation(int *arr, int n, int d);
```

Les arguments de cette fonction sont un tableau de valeurs `int`, sa taille `n` et une valeur positive `int` `d`. La fonction doit modifier le tableau en le faisant pivoter vers la droite de `d` positions. Supposons que `arr` contient les valeurs `2, 3, -9, -7, 5`.

- Si `d = 1`, alors le tableau doit être modifié pour contenir les valeurs `5, 2, 3, -9, -7`.
- Si `d = 3`, alors le tableau doit être modifié pour contenir les valeurs `-9, -7, 5, 2, 3`.

Exercise 25 : Déplacer tous les zéros à la fin Écrivez une fonction avec le prototype suivant :

```
void deplacer_zero(int *arr, int n);
```

Les arguments de cette fonction sont un tableau de valeurs `int` et sa taille `n`. La fonction doit modifier le tableau en déplaçant tous les zéros à la fin du tableau tout en maintenant l'ordre des éléments non nuls. Par exemple, si `arr` contient les valeurs `0, 4, 0, 0, 5, 2, 0, 9`, il doit être modifié pour contenir les valeurs `4, 5, 2, 9, 0, 0, 0, 0`. Écrivez deux versions de la fonction :

1. La première version doit utiliser un tableau auxiliaire (ce qui rend la programmation plus facile).
2. La deuxième version ne doit *pas* utiliser de tableau auxiliaire et doit effectuer la tâche sur place.

4.6 Chaînes de caractères

Dans les langages de programmation, les chaînes de caractères sont utilisées pour stocker du texte ou des séquences de caractères. Nous avons déjà utilisé de nombreuses chaînes dans nos exemples. Chaque appel à la fonction `printf` utilise une chaîne que nous écrivons des chaînes en utilisant la syntaxe suivante :

```
"This is a string!"
```

En C, il n'y a pas de « type chaîne de caractères » dédié. Les chaînes sont implémentées par des tableaux de caractères. Un caractère unique est de type `char`. Par conséquent, comme tous les tableaux, une « variable chaîne » est en réalité un pointeur de type `char *` qui stocke l'adresse du premier caractère de la chaîne.

Cependant, tous les tableaux de caractères ne sont pas des chaînes. Une chaîne valide doit se terminer par un caractère spécial supplémentaire, noté `'\0'`. Ce caractère spécial sert de terminateur, indiquant la fin de la chaîne et signalant qu'aucun autre caractère ne doit être lu. Par exemple, la chaîne `"This is a string!"` est stockée en mémoire comme suit :

'T'	'h'	'i'	's'	' '	'i'	's'	' '	'a'	' '	's'	't'	'r'	'i'	'n'	'g'	'\0'

FIGURE 4.13 : Implémentation d'une chaîne en mémoire.

Le but du caractère spécial `'\0'` est de surmonter l'une des principales limitations des tableaux : un tableau ne connaît pas intrinsèquement sa taille. Dans le cas des chaînes de caractères, déterminer le nombre de caractères est simple : il suffit de compter les caractères jusqu'à ce que le caractère spécial `'\0'` soit rencontré. Cela peut se faire avec la fonction suivante :

```c
int string_length(char *s){
int len = 0;
while (s[len] != '\0'){
  len++;
}
return len;
}
```

La bibliothèque <string.h>

Plusieurs fonctions pour manipuler les chaînes sont déjà implémentées dans la bibliothèque `<string.h>`. Par exemple, la fonction `strlen` est disponible pour calculer la longueur d'une chaîne. Pour utiliser ces fonctions, vous devez inclure la directive suivante au début de votre programme :

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

Tous les tableaux de caractères ne sont pas des chaînes

La fonction `string_length`, ainsi que celles de la bibliothèque `<string.h>`, peuvent accepter n'importe quel tableau de caractères comme argument. Cependant, si le tableau fourni n'est pas une chaîne valide (*i.e.*, s'il ne se termine pas par le caractère spécial `'\0'`), cela entraînera un comportement indéfini. Par exemple, la boucle `while` à l'intérieur de la fonction `string_length` peut continuer au-delà des limites réelles du tableau, ce qui peut provoquer un plantage du programme car il va lire une zone mémoire invalide. Il appartient au programmeur de s'assurer que les tableaux de caractères passés à ces fonctions sont des chaînes valides.

Une chaîne peut être déclarée comme n'importe quel autre tableau. Cependant, il existe une manière spéciale de l'initialiser en garantissant que le caractère de terminaison `'\0'` soit automatiquement placé à la fin. Par exemple, la chaîne `"This is a string!"` peut être déclarée comme suit :

```
char mystring[] = "This is a string!";
```

Notez que la longueur du tableau n'est pas spécifiée explicitement dans les crochets `[]`. Cependant, C détermine automatiquement que `18` valeurs `char` doivent être allouées–`17` pour les caractères de `"This is a string!"` plus une pour le caractère de terminaison spécial `'\0'`.

Ainsi, cette instruction crée un tableau en mémoire comme montré dans la Figure 4.13, et la variable `mystring` est un pointeur (de type `char *`) vers le premier élément du tableau.

5

Allocation dynamique

En C, il existe *deux* principaux emplacements mémoire où les données peuvent être stockées : la *pile* et le *tas*. Nous avons déjà largement discuté de la pile. Dans ce document, nous introduisons le tas. La pile et le tas sont gérés de manière très différente. Ci-dessous, nous résumons les principales différences entre eux :

- **Accès aux données**. La plupart des données stockées sur la pile sont associées à des variables, ce qui permet un accès direct aux valeurs stockées.
  En revanche, les données sur le tas ne sont pas directement associées à des variables. La seule façon d'accéder aux données allouées sur le tas est d'utiliser un pointeur qui contient leur adresse mémoire. Typiquement, ce pointeur est stocké sur la pile et associé à un nom de variable.
- **Durée de vie**. L'allocation et la désallocation de la mémoire sur la pile sont gérées automatiquement : lorsqu'une variable locale est déclarée à l'intérieur d'une fonction, sa durée de vie est déterminée par la portée de cette fonction. Le programmeur n'a aucun contrôle sur ce processus.
  L'allocation et la désallocation sur le tas sont entièrement contrôlées par le programmeur. Pour allouer de la mémoire sur le tas, il faut appeler la fonction `malloc`, qui réserve la quantité de mémoire demandée et retourne son adresse. Cette adresse peut ensuite être stockée dans une variable pointeur sur la pile. La mémoire allouée reste disponible jusqu'à ce qu'elle soit explicitement libérée à l'aide de la fonction `free`.
- **Organisation**. Comme la mémoire de la pile est gérée automatiquement, sa structure suit un schéma prévisible (bien que l'organisation exacte dépende du compilateur).
  En revanche, la mémoire du tas est organisée de manière imprévisible, et le programmeur n'a aucun contrôle direct sur sa disposition. L'allocation de mémoire sur le tas est gérée par le gestionnaire de mémoire du compilateur.

Nous présentons une représentation graphique des deux espaces mémoire dans la Figure 5.1, illustrant plusieurs différences clés entre eux.

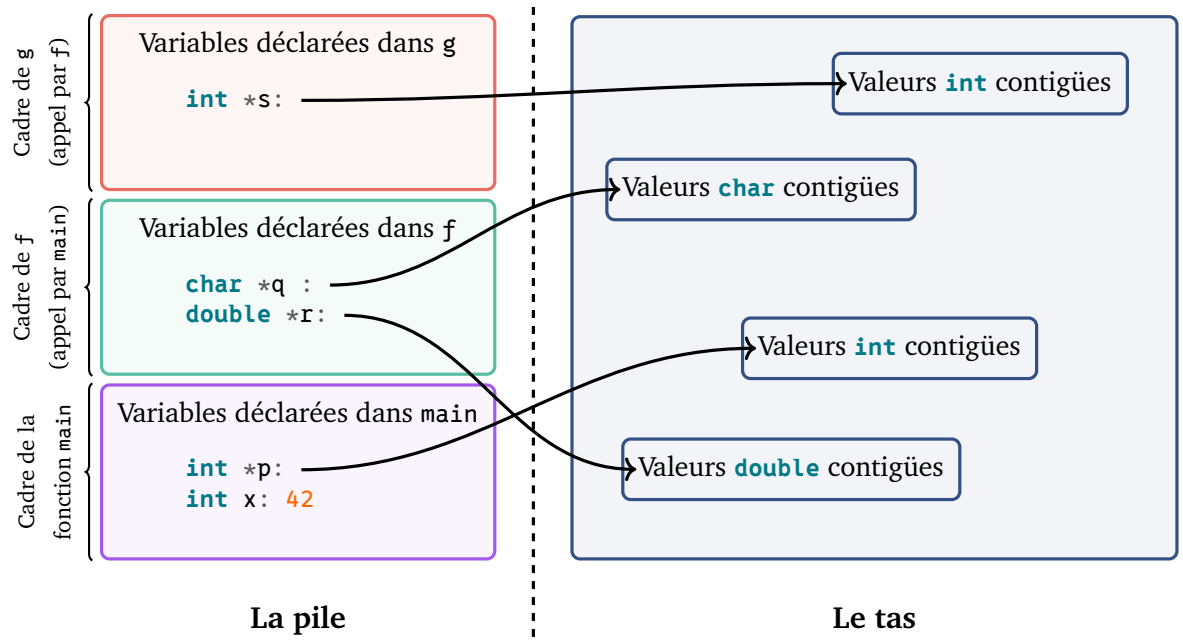


FIGURE 5.1 : Représentation de la mémoire dans un programme C.

Nous allons maintenant décrire l'utilisation des fonctions `malloc` et `free` dans un programme C. Nous allons également aborder un problème courant lié à l'allocation dynamique : les fuites de mémoire.

5.1 Gestion du tas

Pour utiliser la gestion dynamique de la mémoire, la bibliothèque standard de C doit être incluse au début du programme à l'aide de la directive suivante :

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

La bibliothèque standard de C déclare toutes les fonctions utilisées pour allouer et libérer de la mémoire sur le tas. Dans cette section, nous étudions les deux principales fonctions : `malloc` et `free`.

5.1.1 La fonction malloc

La fonction `malloc` est utilisée pour allouer de la mémoire sur le tas. Lorsqu'elle est appelée, le programmeur doit spécifier la quantité de mémoire à allouer en tant qu'argument. Cette quantité doit être donnée en bytes.

Par exemple, pour allouer un tableau de quatre valeurs `int`, on appelle `malloc` avec un argument égal à quatre fois le nombre de bytes utilisés pour stocker un seul `int`. Si l'allocation réussit, `malloc` renvoie un pointeur vers le premier `int` du tableau.

L'opérateur sizeof

Pour allouer de la mémoire pour un type de données spécifique, le programmeur doit connaître le nombre de bytes requis pour stocker une seule valeur de ce type. Comme ce nombre peut varier en fonction du compilateur et de l'architecture du système, l'opérateur `sizeof` permet de le récupérer automatiquement. Si `<type>` est un type de données valide en C, l'expression suivante :

```
sizeof(<type>)
```

s'évalue en le nombre de bytes nécessaires pour stocker une seule valeur de ce type. Par exemple, `sizeof(int)` renvoie le nombre de bytes requis pour stocker un `int`. L'opérateur `sizeof` peut également s'appliquer aux expressions :

```
sizeof(<expression>)
```

Dans ce cas, il s'évalue en le nombre de bytes nécessaires pour stocker le résultat de `<expression>`. Par exemple, si `x` est une variable de type `int`, l'expression `x + 4` donne un `int`. Ainsi, `sizeof(x + 4)` retourne le nombre de bytes nécessaires pour stocker une seule valeur de type `int`.

Nous donc pouvons allouer un tableau de quatre valeurs `int` sur le tas comme suit :

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

Ici, `malloc` alloue exactement le nombre de bytes contigus nécessaires pour stocker quatre valeurs `int` sur le tas et renvoie l'adresse du premier byte. Cette adresse est stockée dans le pointeur `p` alloué sur la pile, qui peut ensuite être utilisée pour accéder au tableau. Il peut être manipulé comme n'importe quel autre tableau. Par exemple, l'instruction `p[2] = 42;` affecte la valeur `42` à l'élément d'indice `2` dans le tableau.

La principale différence avec les tableaux déclarés sur la pile est que, dans ce cas, la mémoire allouée reste sur le tas jusqu'à ce qu'elle soit explicitement libérée par le programmeur avec la fonction `free`.

Le tas nous permet de contourner une limitation des tableaux : l'impossibilité de les retourner depuis des fonctions. Grâce à l'allocation dynamique, nous pouvons écrire des fonctions qui « retournent » des tableaux. Par exemple, on va écrire une fonction qui prend un entier `n` en paramètre, alloue un tableau de taille `n` sur le tas, le remplit avec des valeurs allant de `0` à `n-1`, puis le « retourne » (en réalité, la fonction retourne un pointeur vers le premier élément, plutôt que le tableau lui-même).

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

int *iota(int n);

int main(void){
    int *p = iota(23);
    for(int i = 0; i < 23; i++){
        printf("%d.\n", p[i]);
    }
    free(p);
    return 0;
}

int *iota(int n){
    if (n <= 0){
        return NULL;
    }
    int *t = malloc(n * sizeof(int));
    for (int i = 0; i < n; i++){
        t[i] = i;
    }
    return t;
}
```

```
$ ./iota
0.
1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
```

Pour mieux comprendre ce qui se passe lors de l'exécution de ce programme, visualisons l'état de la mémoire avant et après le retour de la fonction `iota`.

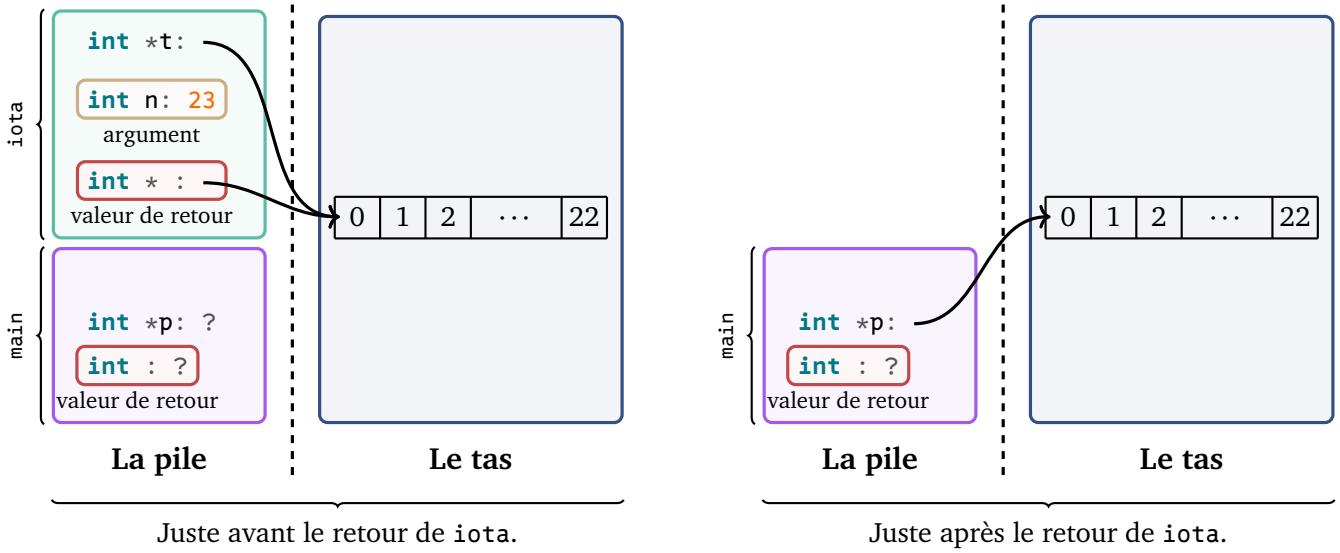


FIGURE 5.2 : État de la mémoire avant et après le retour de la fonction `iota`.

Cet exemple illustre l'utilisation typique de la fonction `malloc`. Étant donné un type `<type>` et un entier positif `n`, un tableau de `n` éléments de type `<type>` peut être alloué sur le tas en utilisant l'instruction suivante :

```
<type> *p = malloc(n * sizeof(<type>));
```

Cette instruction stocke l'adresse du tableau alloué dans la variable pointeur `p`.

⚠ La mémoire allouée avec malloc n'est pas initialisée

Lorsque `malloc` est appelée, elle alloue de la mémoire mais ne l'initialise pas : les données situées à cet emplacement mémoire sont inchangées. Il est donc essentiel d'initialiser manuellement la mémoire allouée. Alternativement, la fonction `calloc` peut être utilisée. Cette variante de `malloc` initialise automatiquement la mémoire allouée en mettant tous les bytes à zéro. Nous discuterons de `calloc` à la fin de cette feuille.

⚠ L'allocation sur le tas peut échouer

Un appel à `malloc` peut échouer (par exemple, si la mémoire disponible est insuffisante). Dans ce cas, `malloc` renvoie un pointeur `NULL`. Cette convention permet de détecter et de gérer les échecs :

```
int *p = malloc(4 * sizeof(int));
if (p == NULL) {
  /* Gérer l'échec de l'allocation */
}
```

5.1.2 La fonction free

Pour désallouer la mémoire stockée sur le tas, le programmeur doit appeler la fonction `free`. Cette fonction prend comme argument l'adresse du bloc mémoire à libérer. Plus précisément, cette adresse doit être celle renvoyée par `malloc` (ou l'une de ses variantes) lors de l'allocation initiale :

```
free(p); /* p doit être l'adresse d'un bloc mémoire valide alloué sur le tas */
```

Cette instruction indique au programme de libérer l'espace mémoire alloué, le rendant ainsi disponible pour une utilisation future. La mémoire libérée peut être réallouée par un appel ultérieur à `malloc`.

Seuls les pointeurs vers des données allouées sur le tas peuvent être passés à free

Passer un pointeur invalide à `free` entraîne un *comportement indéfini*, typiquement un crash :

```c
#include <stdlib.h>
int main(void) {
  int a = 42;
  int *p = &a;
  free(p);
  return 0;
}
```

```
$ ./freetest
freetest(72379,0x1fb9af840)
↪ malloc: *** error for object
↪ 0x16fbd7058:
pointer being freed was not
↪ allocated
```

Parmi les exemples courants de pointeurs invalides, on trouve ceux qui ne référencent plus de données allouées sur le tas (par exemple, après que la mémoire a déjà été libérée, voir la remarque suivante) ou les pointeurs vers des variables situées sur la pile, comme illustré ci-dessus.

La mémoire ne doit être libérée qu'une seule fois

Liberer deux fois le même pointeur (un double `free`) entraîne un *comportement indéfini* et doit être évité. Les conséquences possibles incluent :

- Le programme peut crasher, comme mentionné ci-dessus.
- Si la mémoire est réallouée entre les deux appels à `free`, cela peut entraîner une corruption des données et un comportement imprévisible.

5.1.3 Fuites de mémoire

En C, il existe une « règle d'or » que les programmeurs doivent suivre lorsqu'ils utilisent l'allocation dynamique :

> La mémoire allouée sur le tas doit toujours être libérée lorsqu'elle n'est plus nécessaire.

Comme la libération de la mémoire allouée sur le tas nécessite un appel explicite à la fonction `free`, l'application de cette règle repose entièrement sur le programmeur. Ne pas la respecter peut entraîner une allocation excessive de mémoire pour des données inutilisées. À long terme, cela peut ralentir considérablement le programme, voire provoquer un crash en raison d'une mémoire insuffisante.

Lorsque cette règle d'or est enfreinte, il en résulte une fuite de mémoire. De manière informelle, une fuite de mémoire se produit lorsqu'un programmeur alloue de la mémoire sur le tas à l'aide de `malloc` mais ne la libère pas avec `free` lorsqu'elle n'est plus nécessaire. Malheureusement, les fuites de mémoire peuvent être introduites très facilement, souvent de manière involontaire. Par exemple, considérons la fonction suivante :

```c
void leakfun(int n) {
  int *p = malloc(n * sizeof(int));

  /* Code utilisant le tableau alloué SANS LE LIBÉRER avec free */

  return;
}
```

Chaque fois que `leakfun` est appelée, elle alloue de la mémoire sur le tas. Cependant, la fonction ne libère pas cette mémoire. De plus, l'adresse retournée par `malloc` n'est pas transmise au reste du programme. Une fois que `leakfun` retourne, l'adresse de la mémoire allouée est « perdue », ce qui signifie que ces données deviennent inaccessibles et ne peuvent plus être libérées. Cela entraîne une fuite de mémoire.

Correction de la fonction leakfun

Cet exemple montre que lorsque de la mémoire est allouée sur le tas à l'intérieur d'une fonction, le programmeur doit soit :

- libérer la mémoire allouée avant que la fonction ne retourne, ou
- transmettre l'adresse de la mémoire allouée au reste du programme.

Par exemple, c'est ainsi que fonctionne la fonction `iota` (définie plus haut) : elle alloue un tableau sur le tas et *retourne* son adresse.

5.2 Exercices

Exercice 26 : Renvoyer des pointeurs Considérez les quatre programmes suivants :

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

int *point1(void);

int main(void) {
  int *q = point1();
  printf("%d.\n", *q);
  return 0;
}

int *point1(void) {
  int n = 42;
  return &n;
}
```

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

int *point2(void);

int main(void) {
  int *q = point2();
  printf("%d.\n", *q);
  return 0;
}

int *point2(void) {
  int *p;
  *p = 42;
  return p;
}
```

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

int *point3();

int main(void) {
  int *q = point3();
  printf("%d.\n", *q);
  free(q);
  return 0;
}

int *point3(void) {
  int *p = malloc(sizeof(int));
  *p = 42;
  return p;
}
```

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

void point4(int *p);

int main(void) {
  int *p;
  point4(p);
  printf("%d.\n", *p);
  return 0;
}

void point4(int *p) {
  p = malloc(sizeof(int));
  *p = 42;
}
```

Quel est l'affichage de ces programmes ? Un dessin de la mémoire peut vous aider.

Exercice 27 : Copie de tableau Écrivez une fonction avec le prototype suivant :

```c
int *array_copy(int *a, int n);
```

Cette fonction prend en entrée un tableau d'entiers, `a`, ainsi que sa taille `n`. Elle doit créer une copie de ce tableau sur le tas et renvoyer un pointeur vers cette copie.

Exercice 28 : Tri par fusion Dans cet exercice, nous allons écrire une fonction pour trier un tableau d'entiers en ordre croissant. Cela se fait en deux étapes :

1. Écrivez une fonction avec le prototype suivant :

   ```c
   int *merge_sorted_arrays(int *a1, int n1, int *a2, int n2);
   ```

   Cette fonction prend en entrée deux tableaux *déjà triés*, `a1` et `a2`, ainsi que leurs tailles respectives, `n1` et `n2`. La fonction doit fusionner ces deux tableaux en un seul tableau trié alloué sur le tas et retourner un pointeur vers celui-ci.
   Par exemple, si `a1` contient `2, 3, 3, 9` et `a2` contient `2, 5, 8, 12`, la fonction doit retourner un pointeur vers un nouveau tableau contenant `2, 2, 3, 3, 5, 8, 9, 12`.
2. Écrivez une fonction avec le prototype suivant :

```c
void merge_sort(int *a, int n);
```

Cette fonction prend en entrée un tableau d'entiers et sa taille `n`, et trie le tableau en ordre croissant. Vous devez implémenter l'algorithme du tri par fusion :

- Si le tableau contient au moins deux éléments, triez récursivement ses moitiés gauche et droite.
- Fusionnez les deux moitiés triées dans un nouveau tableau trié (en utilisant la fonction de la première étape).
- Copiez le tableau trié dans le tableau d'origine.

5.3 Variantes de malloc

5.3.1 La fonction calloc

La fonction `calloc` est utilisée pour la même tâche que `malloc` : elle alloue de la mémoire sur le tas. Cependant, il y a une différence essentielle : contrairement à `malloc`, la mémoire allouée avec `calloc` est initialisée–tous les bytes sont mis à zéro.

Pour des raisons historiques, `malloc` et `calloc` suivent des conventions d'appel différentes. Plus précisément, `calloc` prend deux arguments entiers :

- Le premier spécifie le nombre d'éléments à allouer.
- Le second spécifie la taille (en bytes) de chaque élément.

Ainsi, pour allouer un tableau de quatre valeurs `int` sur le tas en utilisant `calloc`, on écrirait :

```c
int *p = calloc(4, sizeof(int));
```

Puisque cette instruction utilise `calloc`, tous les éléments du tableau alloué seront initialisés à `0`.

La différence la plus importante est l'initialisation

La distinction clé entre `calloc` et `malloc` est que `calloc` initialise la mémoire allouée, alors que `malloc` ne le fait pas. Ce point est déterminant dans le choix de la fonction à utiliser.
En revanche, la différence dans la convention des arguments est purement syntaxique, car les deux conventions sont interchangeables. En effet, `calloc` peut être utilisée de la même manière que `malloc`. L'exemple suivant alloue également un tableau de quatre valeurs `int` sur le tas :

```c
int *p = calloc(1, 4 * sizeof(int));
```

5.3.2 La fonction realloc

La fonction realloc est utilisée pour redimensionner un bloc de mémoire qui a déjà été alloué sur le tas. Elle permet soit d'agrandir, soit de réduire le bloc mémoire tout en conservant son contenu existant (autant que possible en cas de réduction). Pour utiliser realloc, deux arguments doivent être fournis :

1. Un pointeur vers le bloc de mémoire précédemment alloué (retourné par un appel antérieur à `malloc`, `calloc` ou `realloc` lui-même).
2. La nouvelle taille (en bytes) du bloc mémoire.

La fonction `realloc` retourne l'adresse du bloc mémoire redimensionné. Deux cas sont possibles :

1. Si suffisamment d'espace est disponible à l'emplacement actuel (ce qui est toujours le cas en cas de réduction), `realloc` redimensionne le bloc sur place. Dans ce cas, l'adresse retournée est la même que l'adresse d'entrée.
2. Sinon, elle alloue un nouveau bloc, copie les anciennes données et libère l'ancien bloc. Dans ce cas, l'adresse retournée pointe vers le nouveau bloc.

Considérons maintenant l'exemple suivant. La fonction ci-dessous prend en entrée un tableau de valeurs `int` alloué sur le tas ainsi que sa taille :

```c
int *square_array(int *a, int n){
  int *p = realloc(a, 2 * n * sizeof(int));
  for(int i = 0; i < n; i++){
    p[n + i] = p[i];
  }
  return p;
}
```

Cette fonction double la taille du tableau d'entrée et copie la première moitié dans la nouvelle seconde moitié.

6 Types structurés

En C, le programmeur peut définir des types de données personnalisés. Dans cette fiche, nous présentons les structures, qui sont le type de données défini par l'utilisateur le plus important [1].

6.1 Structures

Les structures sont des types de données composites définis par le programmeur. Autrement dit, une valeur dont le type est une structure déclarée par le programmeur regroupe *plusieurs valeurs de types éventuellement différents, stockées dand un seul et même bloc mémoire*. Le nombre de valeurs et leurs types sont fixés lors de la déclaration initiale de la structure.

6.1.1 Un premier exemple de structure

Commençons par un exemple. Supposons que nous voulions écrire un programme qui manipule des points sur une grille :

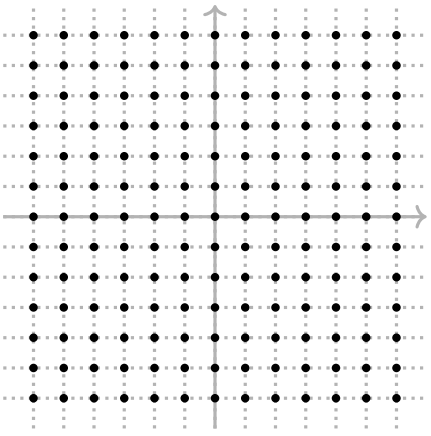


FIGURE 6.1 : Points sur une grille.

Chaque point correspond à *deux* valeurs entières : ses coordonnées. En d'autres termes, les objets fondamentaux que nous voulons manipuler dans notre programme sont des *paires d'entiers*, comme $(0,1)$, $(5,3)$ ou $(-7,3)$. D'après ce que nous savons jusqu'à présent sur C, cela suggère que représenter un *point unique* nécessite *deux variables*, une pour chaque coordonnée. Cependant, les programmeurs peuvent également définir un type personnalisé pour représenter un point : chaque variable de ce type personnalisé représentera un point unique. C'est ce qu'on appelle une *structure*, qui peut être définie avec la syntaxe suivante :

1. Il existe d'autres types de données personnalisés en C, notamment les *unions*, qui sont une variante moins courante des structures, et les *énumérations*, qui permettent d'attribuer des noms à des constantes entières (ces noms améliorent la lisibilité et la maintenabilité du code).

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

Cela définit un nouveau type de données structuré appelé `struct point`, qui possède deux membres nommés `x` et `y`, tous deux de type `int`. Autrement dit, une seule valeur de type `struct point` encapsule deux valeurs de type `int`.

Les variables de type `struct point` sont déclarées comme n'importe quelle autre variable. Par exemple, l'instruction suivante déclare une nouvelle variable nommée `dot` de type `struct point` sur la pile d'exécution :

```
struct point dot;
```

Comme pour les types de données fondamentaux, cette instruction *alloue de la mémoire sur la pile d'exécution* pour stocker un objet `struct point` (*i.e.*, suffisamment d'espace pour stocker deux valeurs de type `int`) et associe cet espace mémoire au nom de variable `dot`.

On accède aux membres d'une variable de type `struct point` avec l'opérateur point « . » suivi des noms `x` et `y`, choisis par le programmeur lors de la déclaration du type. Par exemple, le code suivant déclare une variable de type `struct point` et attribue des valeurs à ses membres pour représenter le point $(4, 2)$ sur la grille :

```
struct point dot;
dot.x = 4;
dot.y = 2;
```

Au-delà de cela, les variables de type `struct point` peuvent être utilisées comme n'importe quelle autre variable. Elles peuvent être passées en argument à des fonctions, être retournées par des fonctions et être affectées les unes aux autres à l'aide de l'opérateur d'affectation `=`.

⚠ Les opérateurs de comparaison ne fonctionnent pas avec les structures

En revanche, les variables de type `struct point` ne peuvent pas être comparées en utilisant les opérateurs de comparaison standard tels que `==`, `<`, ou `>`. En particulier, cela signifie que pour vérifier si deux variables `struct point`, `dot1` et `dot2`, sont égales, chaque membre doit être comparé individuellement. Cela peut être fait à l'aide de l'expression suivante :

```
dot1.x == dot2.x && dot1.y == dot2.y
```

Cette approche est simple ici car `struct point` ne possède que deux membres. Pour des types de données structurés plus complexes, il est recommandé d'écrire une fonction dédiée pour effectuer un test d'égalité.

6.1.2 Définition générale

Un type de structure est déclaré à l'aide du mot-clé `struct`, suivi du *nom de la structure* et d'une liste de ses membres, encadrée par des accolades `{...}`. La déclaration doit se terminer par un point-virgule. Chaque membre est défini en spécifiant son *type*, suivi de son nom et d'un point-virgule :

```
struct nom {
  typeMembre1 nomMembre1;
  typeMembre2 nomMembre2;
  typeMembre3 nomMembre3;
  ...
};
```

Cette déclaration définit un nouveau type de données nommé[2] `struct nom`. Une fois déclaré, on peut :

- Déclarer des variables de type `struct nom` sur la pile. Par exemple, l'instruction `struct nom v;` déclare une variable `v` de type `struct nom`.

2. Remarque : le nom du type est composé de *deux* parties : le mot-clé `struct` et le nom de la structure choisi par le programmeur. Un alias plus simple pour ce type peut être défini, comme expliqué plus loin dans cette fiche.

- Allouer de la mémoire sur le tas pour des valeurs de type `struct nom` en utilisant `malloc`. Ceci sera expliqué plus en détail plus loin dans cette fiche.

Une valeur de type `struct nom` regroupe plusieurs valeurs, une par membre listé dans la déclaration. Ces valeurs sont stockées dans un même bloc mémoire, formant une seule entité de type `struct nom`.

Conventions de nommage

Le choix des noms pour la structure et ses membres suit les mêmes principes que pour les variables : il n'affecte pas le comportement du programme. Cependant, il est fortement recommandé d'utiliser des noms explicites, car cela améliore considérablement la lisibilité et la maintenabilité du code.

Comme expliqué précédemment, l'instruction `struct nom v;` déclare une variable `v` de type `struct nom` sur la pile d'exécution. Chaque valeur correspondant à un membre listé dans la déclaration de la structure peut être accédée en utilisant l'opérateur point « . » suivi du nom du membre. Par exemple,

```
v.nomMembre3
```

représente la valeur associée au troisième membre dans la définition de la structure. Au-delà de cela, les variables de type `struct nom` peuvent être utilisées comme n'importe quelle autre variable. Elles peuvent être passées en argument à des fonctions, retournées par des fonctions et affectées les unes aux autres à l'aide de l'opérateur d'affectation `=`. De plus, il est possible de déclarer des pointeurs vers des valeurs de type `struct nom` ainsi que des tableaux de valeurs de type `struct nom`.

Initialisation d'une variable de structure

Lors de la déclaration d'une variable de structure, il est possible de l'initialiser simultanément en utilisant une syntaxe similaire à celle d'une initialisation de tableau. Illustrons cela avec un exemple :

```c
#include <stdio.h>

struct point {
  int x;
  int y;
};

int main(void) {
  struct point dot = {12, 64};
  printf("x: %d, y: %d.\n", dot.x, dot.y);
  return 0;
}
```

```
$ ./teststruct
x: 12, y: 64.
```

Types de structures imbriquées

Un type de structure peut avoir des membres dont les types sont eux-mêmes des structures préalablement définies. Par exemple, nous pouvons définir une structure dont les membres sont deux points, représentant un segment de droite. La nouvelle structure est définie comme suit :

```c
struct segment {
  struct point start;
  struct point end;
};
```

Si `var` est une variable de type `struct segment`, l'opérateur point « . » est utilisé naturellement pour accéder aux membres des deux points. Par exemple, `var.start.x` représente la coordonnée x du point de départ.

6.2 Pointeurs sur structures

Lorsqu'une nouvelle structure est déclarée, un type de pointeur correspondant pour cette structure est également défini. En pratique, les structures sont souvent manipulées principalement à l'aide de pointeurs.

Les variables pointeurs pour des valeurs de structure sont gérées comme n'importe quels autres pointeurs. Une fois qu'une structure `struct nom` est déclarée, le type de pointeur correspondant est `struct nom*`. Les opérateurs standards des pointeurs peuvent alors être utilisés :

- L'opérateur de référencement `&` récupère l'adresse mémoire d'une variable de structure. Si `v` est une variable de type `struct nom`, alors `&v` est son adresse (de type `struct nom*`).
- L'opérateur de déréférencement `*` permet d'accéder à la valeur de la structure pointée par un pointeur. Si `p` est une variable pointeur de type `struct nom*`, alors `*p` est la valeur de type `struct nom` à laquelle le pointeur fait référence.

```c
#include <stdio.h>

struct point {
  int x;
  int y;
};

int main(void) {
  /* dot est une variable de type struct point */
  struct point dot;
  /* p_dot est un pointeur vers dot (de type struct point*) */
  struct point* p_dot = &dot;
  /* La valeur pointée par p_dot est accédée via l'opérateur "*" */
  (*p_dot).x = 8;
  (*p_dot).y = 2;
  /* Cette instruction affiche "x: 8, y: 2." sur le terminal. */
  printf("x: %d, y: %d.\n", dot.x, dot.y);
  return 0;
}
```

L'opérateur `.` a une priorité plus élevée que `*`

Nous avons utilisé des parenthèses lors de l'affectation de valeurs aux membres de la structure via le pointeur `p_dot`. Plus précisément, nous avons écrit `(*p_dot).x = 8;` et `(*p_dot).y = 2;`.
Ceci est dû au fait que l'opérateur `.` a une priorité plus élevée que l'opérateur de déréférencement `*`. En d'autres termes, l'instruction `*p_dot.x = 8;` serait interprétée comme `*(p_dot.x) = 8;`, ce qui est syntaxiquement incorrect : l'opérateur `.` ne peut pas être appliqué directement à `p_dot` car il s'agit d'un pointeur et non d'une valeur de structure.

6.2.1 L'opérateur flèche

Utiliser à la fois l'opérateur de déréférencement `*` et l'opérateur d'accès aux membres `.`, avec des parenthèses, afin d'accèder aux membres d'une structure via un pointeur peut être fastidieux. Pour simplifier cela, C fournit un opérateur dédié permettant d'accéder directement aux membres d'une structure via un pointeur : l'opérateur flèche `->`.

Si `p` est un pointeur vers une structure, les membres individuels de cette structure peuvent être accédés directement en utilisant l'opérateur flèche suivi du nom du membre :

```
p->nomMembre
```

Utilisation de l'opérateur flèche

L'expression `p->nomMembre` est équivalente à `(*p).nomMembre`. Cela signifie que dans l'exemple précédent, les instructions `(*p_dot).x = 8;` et `(*p_dot).y = 2;` peuvent être écrites de manière équivalente sous la forme `p_dot->x = 8;` et `p_dot->y = 2;`.

6.2.2 Pourquoi utiliser des pointeurs sur structures ?

Les pointeurs sur structures sont fréquemment utilisés en C. L'une des raisons principales est qu'*une seule valeur de structure peut occuper une quantité significative de mémoire*, en particulier lorsque la structure contient de nombreux membres. Par exemple, considérons la structure suivante utilisée pour représenter des fiches d'étudiants dans une base de données :

```
struct etudiant {
  int id;            /* Identifiant de l'étudiant */
  char nom[100];     /* Grand tableau de caractères pour le nom de l'étudiant */
  char adresse[100]; /* Grand tableau de caractères pour l'adresse de l'étudiant */
  double notes[10];  /* Tableau pour stocker les notes de 10 cours */
};
```

Ici, une seule variable de type `struct etudiant` nécessite une quantité importante de mémoire. Cela peut devenir problématique lorsqu'on passe de telles structures en argument à des fonctions. Supposons que nous devions écrire une fonction pour calculer la moyenne des notes d'un étudiant. Bien qu'il soit possible de passer une variable de type `struct etudiant` directement en argument, cela est inefficace et non recommandé.

- *Les arguments de fonction en C sont passés par valeur*, ce qui signifie que lorsqu'on passe une structure en argument, une copie complète de la structure est créée et affectée à une variable locale dans la fonction. Étant donné la taille de la structure `etudiant`, cela est inefficace.
- Il est préférable de passer un *pointeur vers la structure* (*i.e.*, un argument de type `struct etudiant*`). Ainsi, seul le pointeur est copié, ce qui rend les appels de fonction beaucoup plus efficaces.

```
double moyenne_notes(etudiant* e) {
  double somme = 0;
  for(int i = 0; i < 10; i++) {
    somme = somme +  e->notes[i];
  }
  return somme / 10;
}
```

6.2.3 Allocation des structures sur le tas

Les structures sont souvent allouées sur le tas plutôt que sur la pile. Cela signifie qu'elles ne correspondent pas directement à des variables ; elles sont plutôt gérées via une *allocation dynamique de mémoire*, et nous travaillons uniquement avec des pointeurs qui les référencent.

Lors de l'allocation de mémoire sur le tas avec la fonction `malloc`, il est nécessaire de connaître la taille de la structure en octets. Comme pour les types fondamentaux du langage C, on peut utiliser l'opérateur `sizeof` pour déterminer la taille des structures définies par l'utilisateur. Par exemple, l'instruction suivante alloue dynamiquement une structure de type `student` sur le tas et stocke son adresse dans le pointeur `p` :

```
struct student* p = malloc(sizeof(struct student));
```

Cependant, en pratique, la situation est souvent plus complexe. Il est courant que certaines structures aient des membres qui sont eux-mêmes des pointeurs vers d'autres structures. Par exemple, ajoutons un champ représentant la date de naissance de l'étudiant dans notre structure :

```
/* Structure représentant une date */
struct date {
  int month;
  int year;
};

/* Structure représentant un étudiant */
struct student2 {
  int id;            /* Identifiant de l'étudiant */
  char name[100];    /* Tableau de caractères pour le nom de l'étudiant */
  char address[100]; /* Tableau de caractères pour l'adresse de l'étudiant */
  double grades[10]; /* Tableau de notes pour 10 matières */
  struct date* dob;  /* Date de naissance (pointeur sur struct date) */
};
```

Si on utilise `malloc` pour allouer une variable de type `struct student2` sur le tas, cela réserve bien de la mémoire pour *tous les membres* de la structure `student2`. Cependant, il faut noter que l'un de ces membres–`dob`, de type `struct date*`–est en réalité un pointeur vers une autre structure. Dans ce cas, l'appel à `malloc` n'alloue de l'espace que pour le pointeur lui-même, et non pour la structure qu'il est censé pointer.

Pour allouer complètement une structure `student2`, il faut effectuer deux appels distincts à `malloc` :

```c
int main(void){
  struct student2* p = malloc(sizeof(struct student2));  /* Allouer la fiche de l'étudiant */
  p->dob = malloc(sizeof(struct date));                  /* Allouer la date de naissance */
  p->id = 1;
  sprintf(p->name, "Sherlock Holmes");
  sprintf(p->address, "221B Baker Street");
  p->dob->month = 1;
  p->dob->year = 1854;

  /* Code manipulant p */

  return 0;
}
```

Illustrons ces concepts en représentant la mémoire après l'allocation et l'initialisation de la structure.

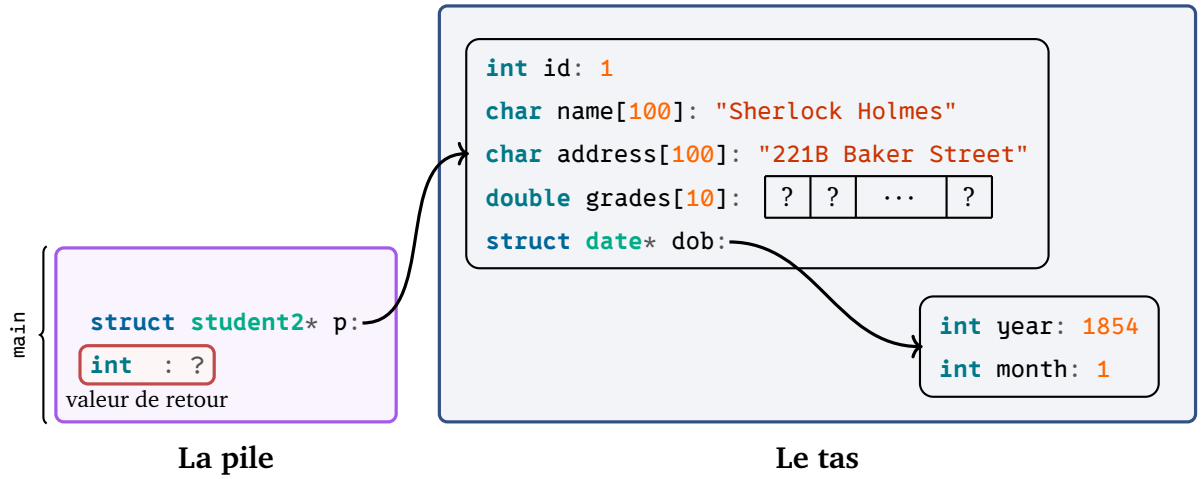


FIGURE 6.2 : La mémoire après l'allocation complète d'une valeur de type `struct student2` sur le tas.

Comme toujours, c'est la *responsabilité du programmeur* de libérer la mémoire allouée dynamiquement une fois qu'elle n'est plus nécessaire. Étant donné que nous avons effectué deux appels à `malloc`, nous devons également effectuer deux appels à `free` afin de libérer correctement la mémoire allouée :

```c
free(p->dob);  /* Libère la structure représentant la date de naissance */
free(p);       /* Libère la structure représentant l'étudiant */
```

Ne pas suivre cette règle peut facilement entraîner des fuites de mémoire. Dans ce cas particulier, il est facile d'oublier de libérer la structure `date` avant de libérer la structure `student2`, ce qui provoquerait une perte du pointeur vers la date de naissance. Pour éviter ce genre de problème, une bonne pratique consiste à définir des fonctions dédiées à l'allocation et à la libération des structures :

```c
/* Allouer une structure student (retourne NULL si l'allocation échoue) */
struct student* create_student(void) {
  struct student* new = malloc(sizeof(struct student));
  if (new == NULL){
    return NULL;
  }
  new->dob = malloc(sizeof(struct date));
  if (new->dob == NULL) {
    free(new);
    return NULL;
  }
  return new;
}

/* Libérer une structure student */
void release_student(struct student* p) {
  if (p == NULL) {
    return;
  }
  free(p->dob);
  free(p);
}
```

⚠ Utiliser l'opérateur flèche

Pour des raisons de lisibilité, il est important d'utiliser l'opérateur flèche plutôt que la combinaison équivalente de `*` et `.`. Supposons que `p` soit un pointeur vers une structure `student` correctement allouée sur le tas, et que nous souhaitions accéder aux membres de la sous-structure représentant la date de naissance. En utilisant l'opérateur flèche, cela peut être fait avec la syntaxe simple suivante :

```c
p->dob->year = 1854;
```

En revanche, utiliser `*` et `.` impose l'utilisation de parenthèses imbriquées, ce qui rend la syntaxe beaucoup plus lourde :

```c
(*(*p).dob).year = 1854;
```

La conclusion essentielle est qu'il faut toujours privilégier l'opérateur flèche lorsque l'on travaille avec des pointeurs de structure afin d'améliorer la lisibilité et la maintenabilité du code.

6.3 Renommage des types de données

Lorsqu'une structure est déclarée, le nom du type correspondant est composé de deux parties : le mot-clé obligatoire **`struct`** suivi d'un nom choisi par le programmeur. Cela peut parfois conduire à des noms longs et peu pratiques.

Heureusement, le mot-clé **`typedef`** permet aux programmeurs de définir des alias pour les types de données existants. Mis à part quelques mots-clés réservés en C, le programmeur peut choisir n'importe quel nom pour ces alias. Une déclaration d'alias suit la syntaxe suivante :

```c
typedef <type existant> <alias>;
```

Cette technique peut être appliquée à n'importe quel type de données, y compris les types fondamentaux déjà définis en C ainsi que les types structurés définis par l'utilisateur. Par exemple, le programme suivant définit un alias pour le type **`int`** :

```c
#include <stdio.h>

typedef int mickey;

mickey main(void) {
  mickey n = 42;
  printf("%d.\n", n);
  return 0;
}
```

Définissons maintenant un alias plus court pour notre type de structure `struct point`. Nous pouvons simplement le nommer `point` pour plus de commodité :

```c
#include <stdio.h>

struct point {
  int x;
  int y;
};

typedef struct point point;

int main() {
  point pt;
  pt.x = 6;
  pt.y = 2;
  printf("x: %d, y: %d.\n", pt.x, pt.y);
  return 0;
}
```

7

Compléments

Ce chapitre présente des compléments. Ceux qui peuvent vous sauver sont indiqués par le logo .
Les autres compléments sont indiqués par le logo .

7.1 Interaction avec le shell

Après avoir compilé un programme C, on obtient un fichier exécutable sous le *shell*. On aimerait alors :

1. Permettre à la commande *shell* créée d'avoir des arguments.
2. Traiter facilement les arguments de cette commande qui sont des options de l'exécutable.
3. Transmettre à la commande l'environnement du *shell*.

Les sections suivantes détaillent ces trois points.

7.1.1 Arguments de la commande shell

On peut transmettre au programme les arguments que l'utilisateur utilise en lançant l'exécutable. Pour cela, on utilise la forme alternative suivante de la fonction `main`.

```
int main(int argc, char **argv){
  // Instructions de la fonction main.
}
```

Lorsque le programme sera lancé, la fonction `main` sera appelée. Ce qui est nouveau est que les deux arguments `argc` et `argv` qui lui seront transmis seront automatiquement initialisés :

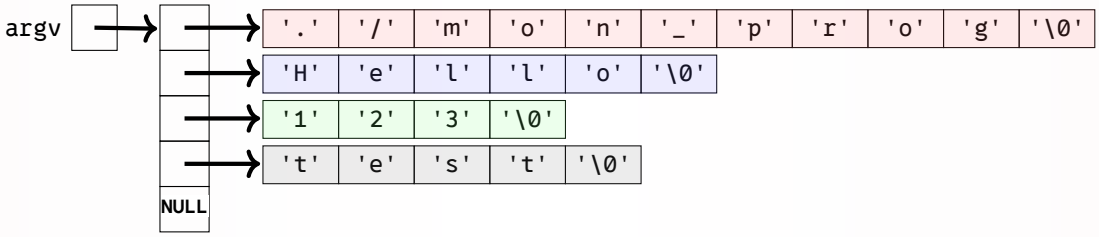
- `argc` est initialisé au nombre d'arguments sur la ligne de commande, plus 1.
- `argv` est initialisé à un tableau de `argc + 1` cases :
  - la case `argv[0]` est initialisée à la chaîne de caractères avec laquelle la commande a été appelée,
  - les cases `argv[1]` à `argv[argc - 1]` sont initialisées avec les arguments de la commande *shell* lancée.
  - la case `argv[argc]` est initialisée à `NULL`.

Schéma

Si vous avez utilisé la forme ci-dessus de la fonction `main`, compilé votre programme et appelé l'exécutable `mon_prog`, et que vous lancez la commande *shell* suivante :

```
$ ./mon_prog Hello 123 test
```

alors, `argc` sera automatiquement initialisé à `4` car il y a 3 arguments sur la ligne de commande : `Hello`, `123` et `test`. De plus, `argv` pointera sur un tableau de 5 cases, dont les 4 premières représentent les chaînes de caractères `"./mon_prog"`, `"Hello"`, `"123"` et `"test"`. La situation est représentée sur la figure suivante.



Programme affichant comment il a été appelé

Une illustration du fait qu'un programme peut savoir comment il a été appelé est donnée ci-dessous :

```c
#include <stdio.h>

int main(int argc, char **argv) {
  printf("Il y a %d arguments.\n\n", argc -
  ↪ 1);
  printf("Commande : %s\n\n", argv[0]);
  for (int i = 1; i < argc; i++) {
    printf("Argument %d : %s\n", i, argv[i]);
  }
  return 0;
}
```

```
$ gcc -Wall argsmain.c -o the_program
$ ./the_program Hello 123 test
Il y a 3 arguments.

Commande : ./the_program

Argument 1 : Hello
Argument 2 : 123
Argument 3 : test
$
```

8 Débogueurs

Chapitre à venir.

8.1 Le débogueur `gcc`

Un *débogueur* est un programme qui permet de localiser et corriger des bogues.

8.1.1 Principe

8.1.2 Interfaces graphiques

8.2 Résoudre les problèmes mémoire : `valgrind`

Index