Tu aimes la frangipane ?

De 0 à Rust

Présenté par :
strawberries

Document compilé le 02 07 2026

Temps de travail total: 13h15

Table des matières

1

Introduction

Ce livre considère que tu n'as aucune expérience en programmation et t'amène à un niveau correct dans le langage Rust.
C'est une objectif ambitieux mais je le crois possible.
Nous passerons par le C avant d'arriver au Rust, pour bien comprendre les enjeux et pourquoi le langage fonctionne comme il le fait.

2

L’ordinateur

Le développement se fait sur ordinateur. Les applications et logiciels créés fonctionnent sur ordinateur mais ont souvent besoin de fonctionner sur d’autres machines telles que des téléphones portables ou des microcontrôleurs. Tous ces types d’appareils ont des composants en commun: un processeur (CPU), de la mémoire vive (RAM), et souvent du stockage. De ce fait, il se peut que j’utilise le mot « machine » pour désigner des ordinateurs ou appareils.

Ce chapitre résumme les bases de l’ordinateur, tu peux le sauter si tu es déjà à l’aise avec le matériel.

2.0.1 Le processeur

Le processeur (ou CPU[*]) est le centre calculatoire de la machine. Il ordonne des données dans la mémoire vive (voir section suivante) et effectue des calculs mathématiques. C’est tout.

Le processeur impose une contrainte conséquente pour la programmation: son architecture. Selon le fabriquant, l’architecture du processeur peut différer et il est important d’adapter notre code à un certain niveau. On parlera plus en détail de pourquoi cela est important lorsqu’on parlera de compilation dans {TODO: link to chapter}.

2.0.2 La mémoire vive

La mémoire vive, aussi appelée RAM — Random Access Memory, est ce que les programmes que l’on créé manipulent principalement. A l’inverse du disque (voir section suivante), les données stockées dans la RAM sont perdues losque la machine est éteinte. Cependant, l’accès en lecture et en écriture sur la RAM est beaucoup plus rapide que sur un disque, et c’est donc elle que l’on utilise pour stocker des données éphémères lors de l’exécution de programmes / logiciels / applications.

2.0.3 Le disque

Ici je parle de disque pour généraliser « solution de stockage » mais de nos jours ce n’est plus toujours physiquement un disque.

Le disque est capable de conserver des données entre les sessions de mise sous tension de la machine. On l’utilise pour faire persister des données entre les exécutions de nos programmes / logiciels / applications.

[*] **CPU :** Central Processing Unit - l'unité centrale de traitement

2.0.4 Récapitulatif

On peut développer des programmes pour plusieurs types d'appareils, allant du microcontrolleur que l'on trouve dans presque tout ce qui est électronique, jusqu'à l'ordinateur classique, en passant par les smartphones.

Ils ont généralement 3 composants manipulables en commun: le processeur, la mémoire vive et parfois des disques.

Losqu'on programme, on s'intéresse surtout à la manipulation de la mémoire vive / RAM.

3

Les bases (binaire, hexadecimal)

Un processeur ne peut pas comprendre les langages que nous utilisont pour communiquer. Encore pire, il ne peut pas comprendre directement des chiffres ou des lettres. La seule chose qu'il peut manipuler et mesurer c'est une tension forte ou une tension faible: le binaire.

3.1 Le binaire

On représente le binaire avec deux chiffres: le 0 et le 1. Un chiffre binaire (0 ou 1 donc) est appelé un *bit*.

Contrairement à ce que l'on pourrait penser, le binaire est amplement suffisant pour coder absolument tout ce que l'on peut imaginer.

Voici les nombres de 0 à 15 codés en binaire:

Représentation décimale	Représentation binaire
0	`0000`
1	`0001`
2	`0010`
3	`0011`
4	`0100`
5	`0101`
6	`0110`
7	`0111`
8	`1000`
9	`1001`
10	`1010`
11	`1011`
12	`1100`

13	`1101`
14	`1110`
15	`1111`

Premièrement, comment est-ce qu'on écrit en binaire… La valeur représentée par chaque bit selon sa position est une puissance de 2. De droite à gauche, les valeurs sont:

$2^0 = 1$,

$2^1 = 2$,

$2^2 = 4$,

$2^3 = 8$

Maintenant, on peut remarquer que les 16 nombres entiers contenus entre 0 et 15 inclus peuvent être représentés en binaire sur 4 bits. Il est possible de représenter des nombres sur plus de bits, augmentant ainsi leur portée. L'unité la plus utilisée est l'octet. Comme son nom semble l'indiquer, il est composé de 8 bits.

Voici quelques nombres supplémentaires, codés en tirant avantage des 8 bits:

Représentation décimale	**Représentation binaire**
15	`0000 1111`
16	`0001 0000`
17	`0001 0001`
18	`0001 0010`
…	…
253	`1111 1101`
254	`1111 1110`
255	`1111 1111`

On remarque que le nombre maximal encodable sur un octet est 255, ce qui élève le nombre de valeurs possibles à 256 lorsqu'on inclut 0 (`0000 0000`).

Lorsqu'on écrit dans différentes bases, on utilise cette syntaxe:

1110_2 — binaire
16_8 — octal
14_{10} — décimal
E_{16} — hexadécimal

3.2 L'hexadecimal

Il existe une autre base que l'on utilise pour représenter des valeurs en informatiques: l'hexadécimal. Comme son nom l'indique, il est composé de 16 valeurs différentes.

Voici chacune de ces valeurs et leur représentation binaire et décimale:

Décimal	Binaire	Hexadécimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Ainsi, on peut représenter des octets en 2 caractères en hexadécimal: par exemple $146_{10} = 1110\ 0110_2 = \mathrm{E6}_{16}$

3.3 ASCII

Maintenant qu'on a des bases pour représenter des valeurs, on peut essayer de représenter du texte. La table *ASCII* existe exactement pour ça.

En voici un extrait:

Décimal	Hexadécimal	Caractère
65	41	A
66	42	B
65	43	C
120	78	x
121	79	y
122	7A	z
64	40	@

61	3D	=

3.4 Récapitulatif

Les ordinateurs ne peuvent comprendre que le binaire. On le représente avec des 0 et des 1, et parfois en hexadécimal avec les caractères de 0 à F. Grâce à cette base, on peut représenter des lettres en binaire avec la table ASCII.

On peut maintenant se demander pourquoi il est possible de programmer en Python ou en C si le processeur ne peut pas comprendre ces langages…

4

La compilation

4.1 Introduction

Il est possible de programmer en binaire. Le processeur comprendrait directement et la vie serait belle. Seulement… Quel humain sain d'esprit voudrait encoder chacune de ses instructions en binaire pour programmer un navigateur internet ? Evidemment, ce processus a été automatisé: la compilation.

On peut programmer avec le langage C graces à des compilateurs comme GCC et Clang qui font le travail de traduction en binaire pour nous (et bien plus !). On peut programmer en Rust grace à rustc, en C++ avec G++ ou Clang, etc…

4.2 L'interprétation de langages

Tous les langages ne sont pas compilés directement en langage machine pour être interprétés par le processeur. Il existe beaucoup de langages qui sont dits *interprétés*: un moteur tourne et exécute ce que le langage lui dit d'exécuter.
L'exemple le plus commun est le Python.
Cela apporte des avantages conséquents en sécurité et en simplicité pour le développement: l'interpréteur est celui qui gère la mémoire, ce n'est plus directement la personne qui programme.
Les langages interprétés prennent cependant un coût en performances.

4.2.1 Java

Le langage *Java* est un cas hybride intéressant: il est compilé en *bytecode* pour être ensuite exécuté par la *JVM* (Java Virtual Machine) qui gère l'utilisation de mémoire et l'optimisation. Ainsi, le Java n'est généralement pas soumis au même coût en performance, tout en assurant une sécurité en mémoire.

4.3 Architectures

J'ai grossi sur plusieurs niveaux en disant que la compilation convertissait simplement le code en binaire. Le plus gros raccourci que j'ai pris est que le binaire résultant est en réalité dépendant de l'*architecture* du processeur.

Voici quelques architectures de processeurs bien répandues:

Architecture	Alias / membres	Où on les trouve
32-bit x86	i386, i486, i586, i686, x86, x86_32, …	Ordinateurs avant 2005, embarqué
64-bit x86	x86_64, amd64, AMD64, x64, …	Ordinateurs modernes, serveurs
32-bit ARM	arm, arm32, AArch32, ARMv7, …	Smartphones et tablettes avant 2015, embarqué
64-bit ARM	arm64, aarch64, …	Smartphones modernes, Apple Silicon, serveurs ARM
RISC-V 32	riscv32, RV32, …	Embarqué

Le langage de programmation de plus bas niveau compris par les processeurs des différentes architectures, qui n'est pas du binaire est l'*assembleur*. Je ne parlerai pas de l'assembleur ici mais un compilateur sait quel assembleur utiliser selon le processeur ciblé par la compilation. Et oui le premier compilateur a du être codé en assembleur, puis généralement *bootstrappé* dans son propre langage. Je t'invite à faire des recherches sur les langages d'assembleur et le bootstrapping de compilateur pour aller (beaucoup) plus loin.

5

Introduction au C

Pour plonger ensuite dans la manipulation de la mémoire, il est selon moi bon de commencer par comprendre le C, un langage de bas niveau qui offre le contrôle total sur la mémoire.
Je vais utiliser des exemples de code, mais pas besoin d'avoir peur car je vais tout expliquer.

5.1 Variables

Une variables est un moyen de stocker une valeur. Elle réserve un emplacement dans la RAM, et est identifiable par un nom:

```
int age = 20;
char lettre_preferee = 'J';
float budget = 299.05;
```

Ici, j'ai déclaré 3 variables différentes: `age`, `lettre_preferee` et `budget`.
Le C utilise un syntaxe spécifique pour déclarer des variables: on spécifie le type (e.g. `int`), puis le nom, puis on dit combien vaut la variable lors de l'initialisation.

Ici, j'ai déclaré:
- `age`, un `int`: c'est un nombre entier, généralement stocké sur 4 octets (32 bits) avec un bit indiquant si il est négatif ou non. Sa valeur maximale est donc 2,147,483,647 et sa valeur minimale est −2,147,483,648.
- `lettre_preferee`, un `char`: un caractère stocké sur un seul octet.
- `budget`, un `float`: c'est un nombre à virgule flottante, permettant les nombres décimaux. Il est stocké sur 4 octets ici également, mais cela dépend du langage et du système.

Après l'exécution de ces 3 lignes, mon programme a donc alloué 9 octets de RAM pour stocker mes 3 premières variables (en excluant le padding qui est un autre sujet).

> **Note sur les tailles de `int`**: Le `int` est un type à taille variable selon le système. On peut avoir besoin d'utiliser un type à taille fixe en embarqué par exemple. En C, la librairie `stdint.h` contient de tels types comme `int32_t` ou `int16_t`.

5.2 Pointeurs

En C, on peut pointer l'emplacement (adresse) mémoire d'une variable à l'aide d'un pointeur:

```
int ma_variable = 3;

int *mon_pointeur = &ma_variable;
```

Ici, j'ai déclaré une variable de type entier `ma_variable` initialisée à 3, puis j'ai déclaré un pointeur sur entier `mon_pointeur` initialisé à l'adresse mémoire de `ma_variable`.
Si on lit `mon_pointeur` actuellement, il contient en effet l'adresse de `ma_variable`.

On peut retrouver `ma_variable` à partir du pointeur avec un *déréférencement*:

```
int autre_variable = *mon_pointeur;
```

Ici, `autre_variable` est dont un entier égal à 3.

5.3 Tables

Une table est une liste de valeurs d'un même type contiguës en mémoire.

```
int ma_table[10];
```

Ici j'ai déclaré une table d'entiers de taille 10.

> On remarque qu'en C, il est possible de déclarer des variables sans les initialiser. Elles contiennent alors la valeur qui se trouvait à cet emplacement mémoire. Lire la valeur d'une variable non-initialisée est dit *undefined behavior* car on ne peut pas prédire ce qui sera lu. Il faut dont éviter de le faire !

On peut accéder à des valeurs de la table par index:

```
ma_table[0] = 20;
ma_table[1] = 12;
ma_table[9] = ma_table[0] + ma_table[1];
```

La plupart des langages de programmation ont un type « string » (une chaîne de caractères). Le C n'en a pas, il utilise simplement les tables avec le type `char`:

```
char ma_chaine = "Hello, World!;
```

Ici, `ma_chaine[1]` est le caractère `'e'`.

> Pour pouvoir lire les chaînes de caractères, le C place automatiquement le caractère `'\0'` à la fin lorsqu'on les initialise comme ci-dessus.

Puisque les valeurs d'une table sont contiguës en mémoire, il est possible d'effectuer de l'*arithmétique sur les pointeurs* pour y accéder par adresse plutôt que par index:

```
int ma_table[3] = {10, 20, 30};

// 'ma_table' tout seul est un pointeur vers le premier élément
(&ma_table[0])
int *p = ma_table;

int premier = *p;           // vaut 10
int deuxieme = *(p + 1);    // vaut 20
int troisieme = *(p + 2);   // vaut 30
```

En C, l'expression `ma_table[i]` est équivalente à `*(ma_table + i)`: un déréférencement de `ma_table + 1`. L'arithmétique de pointeurs est intelligente: quand on ajoute 1 à un pointeur, le compilateur ne rajoute pas simplement 1 à l'adresse mémoire, mais il rajoute la taille du type pointé (par exemple 4 octets pour un `int`).

> **Attention :** Le C ne vérifie jamais si on dépasse la taille de la table. Si on tente d'accéder à ma_table[10] alors que la table fait une taille 3, le programme lira ce qui se trouve juste après en mémoire. Cela peut causer un plantage (Segfault) ou, pire, des failles de sécurité. On parlera de cela dans Chapitre 6.3.

6

La gestion de mémoire

La mémoire d'un programme est divisée en deux zones: la *pile* (stack) et le *tas* (heap). Comprendre les deux est important car elles ne remplissent pas les mêmes fonctions.

6.1 La pile

La pile est la zone mémoire où sont placées les variables que l'on déclare simplement comme on l'a fait précédemment. Voici un exemple de code, puis une représentation de la pile:

```
int var_1 = 3;
char var_2 = 'A';
float var_3 = 123.456;
```

La pile étant réellement une pile, voici sa représentation:

Taille	Nom	Valeur
4	var_3	123.456
1	var_2	'A'
4	var_1	3

6.2 Le tas

Le tas est une zone mémoire plus grande, permettant d'allouer de grandes tailles de données dynamiquement.
En C, on utilise `malloc` pour allouer dans le tas:

```
int *ma_variable = malloc(sizeof(int));
```

Ici, j'appelle `malloc` en lui donnant la taille du segment mémoire que je souhaite allouer: `sizeof(int)` qui sera dans notre cas égal à 4 octets. `malloc` est une fonction qui retourne un pointeur vers la zone mémoire qu'elle a alloué: c'est pour cela qu'on déclare un pointeur avec `int *ma_variable`.

Ce que l'on alloue dans le tas n'est jamais libéré automatiquement en C, et permet donc de faire persister des données peu importe le code.
Voici une comparaison entre pile et tas pour comprendre:

```c
void ma_fonction() {
  int ma_variable_pile = 3;
}

void autre_fonction() {
  int *ma_variable_tas = malloc(sizeof(int));
}
```

J'ai déclaré 2 *fonctions* (morceaux de code qui s'exécutent lorsqu'on les appelle). L'une alloue un entier sur la pile, et l'autre sur le tas.
Maintenant, je les appelle et on va observer ce qui se produit:

```c
ma_fonction();
autre_fonction();

// On étudie ce point de l'exécution.
```

Lors de l'exécution de la première fonction, `ma_fonction`, un `int` est alloué sur la pile. Puis il y a une accolade fermante `}`, ce qui fait sortir `ma_variable` du *scope* (périmètre) d'exécution interne à la fonction. Puisqu'elle est sur la pile, elle est désallouée et libérée. Elle n'existe simplement plus.
A l'inverse, `autre_fonction` alloue un `int` sur le tas, et stocke un pointeur vers elle dans la variable `ma_variable_tas`. Lors de la sortie du *scope* de la fonction, le pointeur `ma_variable_tas`, ce qui stocke l'adresse de la variable, est libéré. Nous n'avons donc plus de moyen de connaître l'adresse de la variable dans le tas. Cependant cette dernière n'est **pas** libérée du tas et les données restent. On appelle ça une *fuite mémoire* ou *memory leak*, sujet dont on reparlera.

Pour libérer un espace du tas alloué en C, on utilise la fonction `free` en lui donnant un pointeur:

```c
free(ma_variable_tas);
```

6.3 Sécurité

La mémoire est un espace de stockage partagé entre tout ce qui est exécuté sur un système, et même au sein d'un processus elle partage non seulement des adresses de variables mais aussi de fonctions, de fil d'exécution etc... De ce fait, il est extrêmement critique d'assurer que des valeurs ne s'échappent pas de leur zone réservée.
Voici quelques exemples d'erreurs catastrophiques que l'on peut faire en C:

6.3.1 Buffer overflow

Le buffer overflow se produit lorsqu'on écrit à coté de l'espace réservé pour un tableau par exemple:

```c
int tableau[10];
tableau[99] = 1234; // Erreur! L'index va de 0 à 9
```

Ici on écrit à une adresse arbitraire dans la mémoire, qui n'est pas allouée pour notre tableau, ce qui peut permettre toutes sortes de redirection d'exécution malveillantes.

6.3.2 Use after free

Si on libère un espace mémoire avec `free` mais que l'on continue d'utiliser le pointeur ensuite, on accède alors à une zone mémoire que ne nous appartient plus et qui peut avoir été allouée à d'autres choses:

```
char *ma_variable_tas = malloc(1); // On alloue un octet
*ma_variable_tas = 'A'; // On utilise le pointeur

free(ma_variable_tas); // On libère la mémoire

// Et plus tard:
*ma_variable_tas = 'B'; // Erreur! Cette zone mémoire ne nous appartient
plus
```

Après le `free`, `ma_variable_tas` est un *dangling pointeur* (« pointeur pendouillant »). Cela nous permet d'accéder et de modifier des zones mémoire qui sont en utilisation par d'autres taches, ce qui permet également toutes sortes d'actions malveillantes.

6.3.3 Solutions en dehors du C

A cause de ces dangers, l'allocation dynamique (dans le tas) est parfois interdite dans les domaines critiques comme l'aeronautique, le médical et l'automobile, et seule la pile est utilisée, assurant que sortir du scope libère les variables du scope.

Comme on en a rapidement parlé dans la section sur la compilation, beaucoup de langages de plus haut niveau ont des stratégies pour assurer que la mémoire ne se retrouve pas sans appartenance. Voici les quelques approches principales avec leurs langages les plus connus:

- **Garbage collecting**: Des langages comme le Java, le Python, le C# ou le Go ont un *garbage_collector* ou « ramasse-miettes ». C'est un processus qui garde la possession des zones mémoire allouées, et les libère automatiquement lorsqu'il s'appercoit qu'elles ne sont plus utilisées. Cela rend le développement plus facile en enlevant la nécessité de s'occuper de la gestion de mémoire, et assure la sécurité parceque toute zone mémoire est soit utilisée, soit reste allouée en attente, soit nettoyée par le garbage collector. Cependant l'exécution du garbage collecting cause des chutes de performance selon la quantité de mémoire allouée, et ne pas libérer la mémoire directement est également un coût en espace mémoire.
- **Gestion par scope (RAII & Ownership)**: Le Rust prend une approche très spéciale à la gestion mémoire, qui lui vaut d'être un langage particulièrement apprécié pour ce qui est de la sécurité. Son compilateur très strict suit l'appartenance de toute variable à au moins un « owner » et s'assure que toute zone mémoire allouée appartient toujours à un scope. Si ce n'est pas le cas, la compilation n'est même pas possible. Le C++ a une approche hybride entre le C et le Rust sur ses fonctionnalités objet avec une stratégie

qui libère les données lorsqu'elles sortent du scope. Le principal désavantage de cette stragégie est simplement que les langages sont plus difficiles à apprendre et utiliser.

Il existe d'autres stratégie de gestion de mémoire, souvent utilisées en C pour aider dans la sécurité et l'efficacité pour ce langage permissif (gestion par régions / arènes, pools, etc…).

7

Structures de données

Représenter et stocker des variables de types simples (dits *primitifs*) est utile, mais on peut parfois avoir besoin de stocker des données sous des organisations plus spécifiques et riches. La plus simple structure de données est le tableau: une suite contiguë de valeurs ou d'autres tables. Cependant on peut parfois avoir besoin de structures comme le dictionnaire pour pouvoir rechercher des nombres à partir de chaînes de caractères, ou de listes chaînées pour optimiser la vitesse d'insertion.

Ici nous alons parler de complexité algorithmique. C'est la mesure du coût en temps et mémoire pour réaliser certaines tâches. Dans notre cas, ca sera le **nombre d'opérations requises** dans le pire cas pour la recherche et l'insertion de données dans les différentes structures de données qui suivent.
La notation pour la complexité temporelle est le « grand O ». Par exemple, si pour n éléments on effectue n opérations, la complexité est de $O(n)$. Je t'encourage à te renseigner sur la complexité algorithmique si tu n'es pas familier avec ce principe.

7.1 Tableau simple

Un *tableau* simple est une liste contiguë d'éléments en mémoire. On l'appelle aussi *vecteur*.

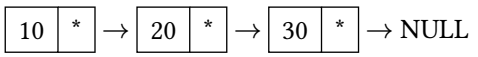
Index	0	1	2	3	4
Valeur	10	32	20	38	123

7.1.1 Complexités

- **Recherche**: Si le tableau contient des valeurs **arbitraires dans le désordre**, la complexité est de $O(n)$ soit linéaire. Cependant si il contient des valeurs **comparables ordonnées** (triées dans l'ordre), des algorithmes de recherche dichotomique (binary search) peuvent trouver une valeur en $O(\log_2(n))$ (complexité logarithmique) en divisant le tableau en deux à chaque comparaison.
- **Insertion**: L'insertion dans un tableau est particulièrement complexe: si l'on souhaite insérer une valeur au centre du tableau, il faut décaler toutes les valeurs à sa droite d'un rang. Ainsi, dans le pire cas la complexité est de $O(n)$, soit linéaire.

7.2 Liste chaînée

La liste chaînée utilise les pointeurs pour structurer un tableau non-contigu en mémoire.



Chaque noeud contient une valeur et un pointeur vers le prochain noeud.

7.2.1 Complexités

- **Recherche**: Que le tableau soit dans le désordre ou dans l'ordre, cette structure de données nous force à parcourir tout le chemin jusqu'à la valeur que l'on cherche, car tous les noeuds ne sont pas contigus en mémoire. Ainsi, la complexité dans le pire cas est $O(n)$.
- **Insertion**: Au contraire, l'insertion dans une liste chaînée est bien plus rapide car il suffit de changer le pointeur d'un noeud en direction du nouveau noeud. De ce fait, la complexité est $O(1)$ (constante).

7.3 Table de hachage

La *table de hachage* est une structure de données dont l'implémentation est plus complexe que celle des tableaux ou des listes. Elle repose sur le concept de **hachage**.
Le hachage est un procédé qui convertit une donnée de façon **déterministe** en une valeur (généralement numérique) de format fixe:
$H(x) = y$, où H est la fonction de hachage, x la valeur d'entrée et y son hash (ou empreinte).
On utilise souvent les fonctions de hachage pour obtenir un hash y de petite taille, censé être unique pour chaque valeur de x (bien que des collisions soient théoriquement possibles).

> En cryptographie, le hachage permet de stocker des données sous une forme qui permet de les valider sans pour autant les révéler. Par exemple:
> Si je stocke le mot de passe `"password123"` dans une base de données, il est risqué de le laisser en clair. À la place, je stocke hash MD5: `"482c811da5d5b4bc6d497ffa98491e38"`.
> Lors d'une tentative de connexion, le système hache le mot de passe saisi et le compare à l'empreinte stockée. Si les deux correspondent, l'accès est accordé. De cette manière, même si la base de données est compromise, le mot de passe original reste difficile à retrouver. (Le MD5 n'est pas une fonction de hashage adaptée à cet usage, pour plusieurs raisons de sécurité. Je t'invite à te renseigner sur les algorithmes de chiffrement.)
> On utilise également des algorithmes de hachage pour garantir l'intégrité de données (signatures numériques), notamment dans le protocole HTTPS ou les blockchains.

Dans un table de hashage, l'objectif est de pouvoir accéder à des données à partir d'une donnée associée. On parle d'association **clé-valeur**. Un algorithme s'occupe de hacher les clés et de leur associer une adresse en mémoire, permettant d'accéder directement à leur valeur.
Voici un exemple de représentation d'un inventaire de fruits qui associe à chaque nom de fruit le nombre disponible en stock:

Clé	Hash de la clé	Adresse mémoire	Valeur
« pomme »	107021353	3	15
« banane »	−1396324207	0	25
« orange »	−1011037684	4	8
« kiwi »	3290659	9	12
« fraise »	−1268573070	7	42

Ici, l'adresse A est déterminée simplement en appliquant un modulo sur le hash: $A = H(x)\%10$.

7.3.1 Complexités

- **Recherche**: Bien que les calculs de conversion de la clé en adresse puissent être lourds, ils ne sont effectués qu'une fois quelle que soit la valeur recherchée. La complexité est donc de $O(1)$.
- **Insertion**: C'est la même chose pour l'insertion, en considérant que la plage de clés disponibles est limitée. (Pour une table de hachage qui grandit dynamiquement, la complexité d'insertion **et** de recherche peut augmenter légèrement.)

> Tous les algorithmes de hachage n'assurent pas l'absence de collisions. Dans la vraie vie, les tables de hachage ont des systèmes pour fonctionner malgré la présence de plusieurs valeurs avec le même hash, et sont généralement hybrides avec des listes chaînées ou autres structures de données.

8

La ligne de commande

Ce chapitre fait une parenthèse pratique sur la *ligne de commande*, un outil particulièrement utile dans le développement et l'informatique en général.

Pour ouvrir une ligne de commande, on peut lancer un *émulateur de terminal*. C'est une application disponible sur tous les systèmes d'exploitation. Sur Windows, elle s'appelle *Terminal Windows*. Sur les systèmes UNIX il en existe énormément, par exemple *Kitty*, *Alacritty*, *Konsole*. Sinon un simple TTY peut suffir sous Linux.
Dans un terminal est exécuté un *shell*. C'est un programme qui lit et interprète les **commandes** que l'on lui donne.
Le format d'une commande est le suivant:
`<nom_du_programme> <argument_1> <argument_2> <...>`
Chaque argument est séparé par un espace.

Par exemple pour compiler un programme C en utilisant `gcc`:

```
gcc programme.c -o programme
```

Chaque programme a ses propres arguments. Sur un système Linux ils sont généralement expliqués sur les *man pages*, les pages de manuel.
Pour lire le manuel de gcc:

```
man gcc
```

9

Application du C

Ce chapitre contient un projet guidé réalisé en C. L'objectif est d'observer une réelle utilisation de la mémoire avant d'arriver à une introduction (légèrement brutale) au Rust.
Ce projet est l'implémentatin d'une liste chaînée dans laquelle nous verrons une vraie gestion manuelle de la mémoire.

Notre liste chaînée contiendra des valeurs de type `int`, un nombre entier (positif ou négatif).
De plus, elle sera *doublement chaînée*: c'est le même principe que la liste chaînée que nous avons vu dans le Chapitre 7.2, et cela permet de plus facilement naviger à l'intérieur.

Bien. La première chose à faire est de définir nos `struct`. On commence par définir le noeud:

```
typedef struct Node {
  int val;
  struct Node *prev;
  struct Node *next;
} Node;
```

Ici, j'utilise `typedef`, une fonctionnalité du C qui permet de donner un nom à un type.
On utilise ensuite le type `struct` qui permet de grouper plusieurs types ensemble.
`int val` est notre nombre, et `struct Node *next` est un pointeur sur le noeud suivant.
Cette liste étant doublement chaînée, nous définissons également `struct Node *prev` qui pointera sur le noeud précédent.

Ensuite, nous définissons le type de la liste elle-même pour une manipulation plus facile:

```
typedef struct {
  Node *head;
  size_t length;
} List;
```

On définit un pointeur sur la tête, le premier noeud de la liste, ainsi que la taille de la liste (le nombre de noeuds qu'elle contient) avec le type `size_t`.

Dans un premier temps, il y a 3 fonction simples qu'il nous faut définir.
D'abord, `new()`:

```
List new() {
  return (List){0, flags, NULL};
}
```

La fonction `new()` retourne un type `List`, et ne prend aucun argument.

Ensuite, la fonction `is_empty()` pour savoir si la liste est vide ou non:

```
bool is_empty(List *list) {
  return list->length == 0 ? true : false;
}
```

`is_empty` retourne un type `bool`: un *booléen* (soit vrai soit faux), et prend en argument un pointeur sur une `List`. On utilise un pointeur au lieu de lui passer la liste elle-même car lorsqu'on passe une variable en argument à une fonction en C, elle est *clonée*. Un pointeur est plus petit qu'un `struct` qui contient un pointeur et un `size_t`.
On retourne la valeur de retour d'une expression avec l'*opérateur ternaire*. Il se lit ainsi: « Si `list->length` est supérieur à 0, alors `true`. Sinon, `false`. ».
`list->length` est une syntaxe spéciale: on souhaite accéder à l'attribut `length` du `struct`, mais puisque `list` est en fait un **pointeur** sur le `struct`, il faut d'abord le **déréférencer**. Le `->` fait le déréférencement et l'accès en même temps. On pourrait écrire `(*list).length` au lieu de `list->length`.

Ensuite, la fonction `length()` qui nous donne la longueur de la liste (le nombre de noeuds dans la liste):

```
size_t length(List *list) { return list->length; }
```

La fonction `length()` retourne un type `size_t` et prend en argument un pointeur sur une `List` comme la fonction précédente.
Puisqu'on stocke la taille de la liste en tant qu'attribut dans le `struct List`, cette opération a une complexité de $O(1)$ au lieu de $O(n)$ si l'on avait eu à parcourir tous les noeuds.

Nous pouvons commencer à parcourir notre liste pour vérifier combien de fois elle contient une valeur spécifique:

```
size_t count(List *list, int val) {
  Node *cur = list->head;

  size_t n = 0;

  while (cur != NULL) {
    if (cur->val == val) n++;

    cur = cur->next;
  }

  return n;
}
```

La fonction `count` retourne un type `size_t` et prend en argument un pointeur sur un noeud et une valeur à rechercher.
On déclare une variable « curseur » qui se déplacera sur la chaîne: on l'initialise avec le pointeur sur la tête de la liste (le premier noeud) en utilisant l'attribut `head` de notre `struct List`.
Puis on déclare la variable `n` qui comptera le nombre d'occurences de `val`, et on l'initialise à 0.
Ensuite, on lance une *boucle while*: elle répète son contenu tant que la condition dans les parenthèses est évaluée vraie: `cur != NULL` assure que la boucle s'arrête lorsqu'on atteint la fin de la liste.
Dans la boucle while, on a un `if` qui regarde si la valeur du noeud sur lequel notre curseur pointe est égale à la valeur que l'on recherche. Si c'est le case alors on *incrémente* n en utilisant `n++` (on pourrait aussi faire `n = n + 1;`).
A la fin de chaque *itération* de la boucle, on donne à notre pointeur curseur l'adresse du prochain noeud en récupérant `cur->next`.
Et enfin, après la fin de la boucle, on retourne n qui est le nombre d'occurences de la valeur `val` dans la liste passée par pointeur en argument.

Maintenant on va commencer à insérer des noeuds. On va pour ça avoir besoin de les allouer dans le tas, comme on l'a vu précédemment dans le Chapitre 6.2.
Créons une petite fonction utilitaire qui alloue un noeud et nous retourne un pointeur pour son adresse mémoire:

```
Node *alloc_node(int val, Node *prev, Node *next) {
  Node *new_node = malloc(sizeof(Node));

  if (new_node == NULL) {
    fprintf(stderr, "Memory allocation error\n");
    exit(1);
  }

  new_node->val = val;
  new_node->prev = prev;
  new_node->next = next;

  return new_node;
}
```

La fonction `alloc_node()` retourne un type `Node*` (pointeur sur `Node`), et prend 3 arguments:

- `val`, un entier qui est la valeur contenur par le noeud
- `prev` un pointeur sur le noeud précédent
- `next` un pointeur sur le noeud suivant

La première chose que l'ont fait est allouer la zone mémoire nécessaire pour stocker le `struct Node`: on utilise `malloc` en lui passant `sizeof(Node)`. Puisque `malloc` retourne un *pointeur générique* (de type `void*`), on le stocke dans la variable `new_node` de type `Node*` ce qui le *cast* (ou *transtype*) en **pointeur sur `Node`**.

Il est ensuite nécessaire de vérifier que `malloc` a réussi. Si par exemple la RAM disponible est pleine, `malloc` retourne `NULL`. En C, `NULL` (ou *null-pointer*) est un pointeur sur l'adresse 0 représentant un pointeur sur « rien ». Si le pointeur vaut `NULL`, alors on affiche une erreur et on termine le programme (la ligne `fprintf(..` n'a pas besoin d'être comprise pour le moment).
Ensuite, on définit les différents attributs du `struct Node` que l'on vient d'allouer, et on retourne le pointeur.

Il y a 3 façons d'insérer un noeud: au début, à la fin, et à sa position triée. L'insérer au début est le plus simple et le moins complexe algorithmiquement ($O(1)$), et les deux autres sont au pire cas $O(n)$.
On va commencer par l'insertion en tête:

```
Node *insert_front(List *list, int val) {
    Node *new_node = alloc_node(val, NULL, list->head);

    if (list->head != NULL) {
        list->head->prev = new_node;
    }

    list->head = new_node;
    list->length++;

    return new_node;
}
```

La fonction `insert_front()` retourne un pointeur sur `Node`, bien que ca ne soit pas particulièrement utile.
On utilise notre fonction `alloc_node()` en lui passant la valeur, le pointeur `prev` qui est donc `NULL`, et le pointeur `next` qui pointe sur l'ancienne tête de la liste.
Ensuite si il existait déjà un noeud en tête de la liste (`if (list->head != NULL)`) alors on définit son `prev` qui était `NULL` et qui est maintenant l'adresse de notre nouveau noeud.
On met à jour la tête de la liste pour être notre nouveau noeud, et on incrémente la taille de la liste.

Maintenant qu'on peut allouer des noeuds dans le tas, il est impératif de prévoir de quoi les libérer si notre liste sort du scope d'exécution. Voici une fonction qui libère la liste entière.

```
void list_free(List *list) {
    Node *cur = list->head;

    while (cur != NULL) {
        Node *temp = cur;

        cur = cur->next;
        free(temp);
    }
```

```
    list->head = NULL;
    list->length = 0;
}
```

La fonction `list_free` ne retourne rien, donc on utilise le type `void`.
On déclare comme précédemment un pointeur curseur qui va parcourir la liste. A chaque noeud, on stocke son adresse dans une variable `temp`, et on fait continuer notre curseur avant d'utiliser `free()` sur `temp`.
A la fin, la tête de la liste est mise sur `NULL` et sa taille à 0. Il ne reste plus que le `struct List`, qui est lui alloué sur la pile et sera donc libéré automatiquement en sortie de scope.

Maintenant, voici un exemple d'utilisation de notre liste chaînée:

```
int main() {
  List ma_list = new();

  insert_front(&ma_liste, 1);
  insert_front(&ma_liste, 5);
  insert_front(&ma_liste, 12);
  insert_front(&ma_liste, 5);

  printf(
    "Taille de la liste: %zu\n",
    length(&list)
  );

  printf(
    "Nombre de 5 dans la liste: %zu\n",
    count(&list, 5)
  );
}
```

Dans la fonction `main()`, point d'entrée d'un exécutable en C, on déclare une liste, l'initialise avec notre fonction `new()`, et lui insère quelques entiers.
On passe à `insert_front` la valeur `&ma_liste` qui évalue à l'adresse de `ma_liste` car la fonction prend en argument un pointeur sur `List`.
Enfin, on utilise la fonction `printf()` pour afficher du texte dans le terminal (voir Chapitre 8). `printf()` prend un *string literal* contenant des *escape codes* de formattage. `%zu` s'attend à être remplacé par un type `usize`. On pourrait utiliser `%d` pour le type `int`, `%c` pour le type char, etc... `\n` est un retour à la ligne.

> La fonction `main()` est la seule fonction n'ayant pas un type de retour `void` autorisée à ne pas retourner de valeur. Cependant il est préférable d'ajouter un `return 0;` à la fin pour respecter la convention.

L'exécution de notre programme entier après ajout des librairies nécessaires et compilation se produirait come ceci:

```
gcc liste.c -o liste # Compilation
./liste # Exécution
```

```
4
2
```

> Cet extrait de shell contient des commentaires (commençant par #), qui sont donc ignorés par le shell.

Cependant, j'ai oublié quelquechose de critique dans ma fonction `main()`. En effet exécutée comme ça, tous les noeuds sont encore alloués en mémoire. Il faut impérativement appeler notre fonction de nettoyage à la fin de `main()`:

```
list_free(&ma_liste);
```

> L'outil *Valgrind* est très utile pour vérifier si un programme a des fuites de mémoire ou autres erreurs de la sorte.

10

Introduction au Rust

Ce qui rend le Rust aussi sécurisé est le fait que le compilateur refusera de compiler si il y a des erreurs qui violent les règles imposées.

Commençons par la syntaxe avec des exemples:

```
fn main() {
  let ma_variable = 3;
}
```

Ici je définis la fonction `main()`, point d'entrée de l'exécutable.
En Rust, on déclare une variable avec le mot clé `let`. Le type est *inféré* car déclarer une variable nous oblige à l'initialiser en même temps.
Par défaut une variable est *immuable*. Si je veux pouvoir la modifier, il faut que je la déclare *mutable* avec le mot clé `mut`: `let mut ma_variable = 3;`.

Maintenant, on va introduire le type `String`, un type spécial et nous allons voir pourquoi:

```
fn ma_fonction(ma_chaine: String) {
    println!("{ma_chaine}");
}

fn main() {
    let ma_chaine = String::from("Hello world");

    ma_fonction(ma_chaine);

    println!("{ma_chaine}"); // Erreur!
}
```

On définit une fonction `ma_fonction()` qui prend en argument un type `String` et ne retourne rien.
Dans `main()` on déclare la variable `ma_chaine` en utilisant la fonction `from` du type `String`.
Puis on appelle `ma_fonction` en lui donnant `ma_chaine`.
Le programme va en effet afficher `"Hello world"` car `ma_fonction()` utilise la *macro* `println!`.
Cependant le type `String` est alloué sur le tas et n'implémente pas `Copy`, ce qui veut dire que les variables de type `String` ne sont pas copiées lorsqu'on les passe dans une fonction (voir Chapitre 9 pour la copie des arguments). A la place, on dit que

`ma_chaine` *move*: elle est déplacée entièrement dans `ma_fonction()`. Elle **n'appartient plus** à `main()`. De ce fait, essayer de print `ma_chaine` après qu'elle ait été déplacée dans `ma_fonction()` cause une erreur à la compilation:

```
error[E0382]: borrow of moved value: `ma_chaine`
```

On aura compris que `ma_fonction()` requiert la possession de son argument. Une solution à cela, d'ailleurs recommandée par le compilateur lors de cette erreur, est de *cloner* la variable pour la passer à `ma_fonction`, ce qui est possible car le type `String` implémente le trait `Clone` (à ne pas confondre avec `Copy` qui est géré différemment). On reparlera des traits dans `ref`.):

```
fn main() {
    let ma_chaine = String::from("Hello world");

    ma_fonction(ma_chaine.clone()); // On utilise .clone()

    println!("{ma_chaine}");
}
```

Cependant cette solution requiert de dupliquer la chaîne de caractères en mémoire, ce qui n'est pas efficace. Mais existe une solution plus propre… Notre fonction `ma_fonction()` actuellement demande que son `String` en argument lui appartienne, et donc ne peut pas le rendre après son exécution. On peut dire à la fonction d'*empreinter* la variable à la place:

```
fn ma_fonction(ma_chaine: &String) {
    println!("{ma_chaine}");
}

fn main() {
    let ma_chaine = String::from("Hello world");

    ma_fonction(&ma_chaine); // On passe une référence

    println!("{ma_chaine}");
}
```

> Les types comme `u32`, `bool` ou `f64` ne sont pas affectés par cette contrainte: ils sont copiés car ils sont alloués sur la pile du fait de leur taille et statique.

Une référence peut également être mutable si une fonction a besoin de modifier la variable passée en argument:

```
fn ma_fonction(ma_chaine: &mut String) {
  ma_chaine.push_str("rld!");
}

fn main() {
    let mut ma_chaine = String::from("Hello wo");
```

```
    ma_fonction(&mut ma_chaine); // On passe une référence mutable
}
```

11

Suite ?

Je ne sais pas comment continuer, donc pour le moment on s'arrête là.
Si le Rust t'intéresse, tu peux lire *The Rust Programming Language*.
Cependant selon les domaines d'utilisation du Rust, il y a des notions que l'on n'a pas vu ici: l'*asynchrone*, le *parallélisme*, les *macros*, etc…