# Apprendre le langage C et son bon usage

Julien Bernard
julien.bernard@univ-fcomte.fr

version 1 (septembre 2017)

Ce document s'adresse à des étudiants ayant déjà une expérience de programmation et souhaitant découvrir le langage C. Le plus difficile dans le langage C n'est pas sa syntaxe mais les bonnes pratiques qui permettent de bien programmer. Aussi le présent document attachera une importance toute particulière à mettre en lumière ces bonnes pratiques à travers des recommandations bien identifiées.

Nous supposerons également que les étudiants disposent d'une station de travail avec un système GNU/Linux installé, soit directement, soit via une machine virtuelle.

# Sommaire

# Chapitre 1

# Le langage C

Dans ce chapitre, nous allons parcourir la syntaxe et la sémantique du langage C en essayant d'identifier les pièges potentiels et comment les éviter.

## 1.1 Introduction

Le langage C est un langage impératif procédural. Un programme en C est découpé en plusieurs fonctions qui peuvent être appelées depuis n'importe quelle autre fonction. Les opérations à l'intérieur des fonctions sont décrites en termes de séquences d'instructions pour modifier l'état du programme.

Dans cette section, nous allons voir un bref historique du langage C ainsi que notre premier programme en C.

### 1.1.1 Historique

#### C K&R

Le C naît quasiment en même temps que Unix dans les années 70. En effet, jusque là, les systèmes d'exploitation étaient écrit en assembleur pour chaque machine. Brian Kernighan et Ken Thompson décide d'inventer un langage de programmation pour ré-écrire le système Unix indépendamment de la machine. Par la suite, Dennis Ritchie apporte également son aide et stabilise le langage dans le livre *The C Programming Language* sorti en 1978. La version de C issue de ce livre est généralement appelée C K&R (pour Kernighan et Ritchie). Sa syntaxe diffère un peu du C moderne que nous connaissons même si on peut encore rencontrer de nos jours des programmes écrits en C K&R.

#### C89

L'organisme de normalisation américain, l'ANSI, décide de normaliser le langage dans les années 80. Ce long travail aboutit en 1989 à la version C89, aussi appelée ANSI C. Cette version est très importante parce qu'elle apporte des nouveautés syntaxiques tout en restant compatible avec le C K&R. En 1990, l'organisme de normalisation mondial, l'ISO, reprend le développement de la norme sous son aile et adopte le C89. Cette version est donc également connue sous le nom C90 ou ISO C.

Certains développeurs sont très attachés à cette version du C et prennent un soin particulier pour rester compatible avec C89. Pendant longtemps, cette version était d'ailleurs la version par défaut dans beaucoup de compilateurs C. Dans le présent document, les constructions qui requiert une version supérieure à C89 seront indiquées.

#### C99

Le travail de normalisation continue jusqu'en 1999 où une nouvelle version du C est éditée, le C99. Cette version apporte beaucoup d'améliorations qui permettent de coder de manière plus propre et plus sûre. Aussi, nous nous focaliserons sur cette version du C. Il est à noter qu'à cette époque, le C++ est déjà né. Il est largement compatible avec le C et les deux langages évoluent en ayant une influence réciproque l'un sur l'autre. Des constructions en provenance du C++ intègrent donc le langage C.

#### C11

En 2011, la norme est à nouveau mise à jour vers la version C11. Les principaux ajouts portent sur des aspects avancées de la programmation qui dépasse le cadre du présent document. De plus, certains compilateurs grand public n'implémentent pas encore entièrement la norme C99, et donc encore moins la norme C11.

### 1.1.2 Hello World !

Voici notre premier programme en C.

```c
#include <stdio.h>

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

Il s'agit d'un simple programme qui affiche la chaîne de caractère «Hello World !» sur la console.

La première ligne est une directive du préprocesseur, elle permet l'inclusion du fichier d'en-têtes `stdio.h` dans lequel se trouve la fonction `printf`. L'inclusion de fichier est abordée en section 1.6.2.

Puis, la fonction principale `main` du programme est définie, c'est le point d'entrée du programme. La section 1.5.4 vous dira tout ce qu'il faut savoir sur cette fonction principale.

Enfin, la chaîne de caractères est affichée via la fonction `printf`, puis le programme s'arrête en renvoyant un code d'erreur 0 (qui signifie qu'il n'y a pas eu d'erreur). La fonction `printf` est décrite en section 2.2.

### 1.1.3 Compilation

Pour compiler un programme sous GNU/Linux, vous devez disposer d'un compilateur C. Généralement, GCC est fourni par défaut mais vous pouvez

également opter pour Clang. Ces deux compilateurs partagent une grande partie de leurs options et sont tous les deux de très bonne qualité. Dans la suite, nous utiliserons la commande `cc` qui est un alias pour le compilateur principal sur les systèmes GNU/Linux.

La commande qui permet de compiler le programme précédent qui aura été placé dans un fichier `hello.c` est :

```
cc -std=c99 -Wall -g -O2 -o hello hello.c
```

L'option `-std=c99` permet d'indiquer au compilateur que notre programme suit la norme C99. En particulier, cela signifie que les extensions au langage du compilateur ne sont pas autorisées. Hormis dans des cas très spécifiques et dûment justifiés, ces extensions sont inutiles pour l'usage courant du langage C.

L'option `-Wall` signifie que le compilateur doit activer tous les avertissements qu'il peut *(Warning all)*. Avoir un code qui compile proprement, c'est-à-dire sans avertissement du compilateur, est la première étape vers un code correct. C'est donc l'objet de notre première recommandation.

> **Recommandation #1** S'assurer de toujours compiler sans aucun avertissement du compilateur. Chaque avertissement du compilateur cache une erreur potentielle.

Attention cependant à ne pas vouloir corriger les avertissements en dépit du bon sens. Il ne faut pas faire plaisir au compilateur, il faut comprendre pourquoi le compilateur génère un avertissement et le corriger intelligemment.

L'option `-g` permet d'activer les symboles de *debug*. Quand vous aurez une grosse erreur qui fait planter votre programme, vous serez alors capable de déterminer la ligne fautive grâce à ces symboles de *debug*. Dans le cas contraire, vous n'aurez qu'une information partielle sur l'erreur.

L'option `-O2` indique au compilateur d'optimiser le programme. Il existe plusieurs niveaux d'optimisation : le niveau 0 indique qu'il ne faut faire aucune optimisation ; le niveau 1 indique qu'il faut faire les optimisations les plus simples ; le niveau 2 indique qu'il faut optimiser le programme sans le compromettre ; le niveau 3 indique qu'il est possible d'utiliser des optimisations très agressives qui peuvent prendre du temps et potentiellement être néfastes pour le programme.

L'option `-o` définit le nom du programme à créer (ici, `hello`). Si cette option est absente, le programme résultant de la compilation s'appellera `a.out`.

## 1.2 Types et opérateurs

Dans cette section, nous allons passer en revue les différents types existants dans le langage C.

### 1.2.1 Types primitifs

Les types primitifs sont des types fournis directement par le langage.

| Nom | Caractère | Valeur | Utilité |
|---|---|---|---|
| NUL | `\0` | `0x00` | Fin de chaîne de caractère |
| BEL | `\a` | `0x07` | *Bell* |
| BS | `\b` | `0x08` | *Backspace* |
| HT | `\t` | `0x09` | *Horizontal Tab* |
| LF | `\n` | `0x0A` | *Line Feed* |
| VT | `\v` | `0x0B` | *Vertical Tab* |
| FF | `\f` | `0x0C` | *Form Feed* |
| CR | `\r` | `0x0D` | *Carriage Return* |
| | `\"` | `0x22` | Guillemet double |
| | `\'` | `0x27` | Guillemet simple |
| | `\?` | `0x3F` | Point d'interrogation |
| | `\\` | `0x5C` | Antislash |
| | `\$$$` | | Valeur octale représentée par `$$$` |
| | `\x$$` | | Valeur hexadécimale représentée par `$$` |

TABLE 1.1 – Séquences d'échappement en C

### Caractères

En C, le type `char` sert à représenter des caractères sur un octet. Plus précisément, il peut représenter tous les encodages dont l'unité de code *(code unit)* est sur un octet, en particulier UTF-8 qui est un encodage quasiment universel. La norme ne précise pas si le type `char` est signé ou non-signé.

Le langage C fournit en plus les deux types `signed char` et `unsigned char` qui représente respectivement des caractères signés et non-signés. Le type `char` n'est synonyme d'aucun des deux, il s'agit bien de trois types distincts. Parmi ces deux types, le seul qui est utilisé régulièrement est `unsigned char`, il sert souvent à représenter un octet de données binaires brutes.

Un littéral de type caractère s'écrit avec des guillemets simples. Cependant, il faut s'assurer que le littéral est bien codé sur un seul octet. Par exemple, `'é'` n'est pas un littéral valide car, en UTF-8, le caractère «é» est codé sur deux octets. De manière générale, seuls les caractères ASCII sont codés sur un seul octet.

```c
char c = 'a';
unsigned char data = 42;
```

Il existe un certain nombre de séquence d'échappement *(escape sequence)* qui correspondent à des caractères qui n'ont généralement pas de représentation visuelle. Pour pouvoir écrire ces séquences d'échappement, on utilise le caractère d'échappement `\` suivi d'un code qui permet de savoir quel est le caractère concerné. La table 1.1 donne les séquences d'échappement accessible en C.

> **Recommandation #2** Pour indiquer un passage à la ligne, on utilise toujours le caractère `'\n'`.

| Nom usuel | Synonymes |
|---|---|
| `short` | `short int`<br>`signed short`<br>`signed short int` |
| `unsigned short` | `unsigned short int` |
| `int` | `signed`<br>`signed int` |
| `unsigned int` | `unsigned` |
| `long` | `long int`<br>`signed long`<br>`signed long int` |
| `unsigned long` | `unsigned long int` |

TABLE 1.2 – Les différents types d'entiers en C

| Types | ILP32 | LLP64 | LP64 |
|---|---|---|---|
| `short`, `unsigned short` | 16 | 16 | 16 |
| `int`, `unsigned int` | 32 | 32 | 32 |
| `long`, `unsigned long` | 32 | 32 | 64 |
| pointeurs | 32 | 64 | 64 |

TABLE 1.3 – Taille en bits des types entiers suivant le modèle de données

**Entiers**

Les entiers peuvent être représentés par trois genres, chaque genre existant en version signée ou non-signée, ce qui représentent six types. La figure 1.2 détaille les différents types entiers ainsi que tous leurs synonymes.

Généralement, le type `int` convient pour la plupart des usages. Même quand les données à représenter ne peuvent être que positives, on n'utilisera pas nécessairement un type non-signé. En effet, la valeur `-1` peut alors être utilisée comme valeur particulière représentant l'absence de donnée. De plus, le type `int` est généralement le type entier le plus adapté à la machine et sa manipulation est donc extrêmement rapide.

> **Recommandation #3** On utilise toujours le type `int` pour représenter un entier, sauf cas particulier qu'il faut justifier.

Pour chaque genre d'entiers, la taille de la version signée et non-signée est la même. Malheureusement, elle peut être différente d'un système à l'autre. La table 1.3 fournit la taille en bits pour chaque genre d'entiers en fonction du modèle de données utilisé par le système. La table 1.4 donne les modèles de données utilisés par les principaux systèmes d'exploitation.

Dans certains cas, comme par exemple la sérialisation de donnée ou la transmission de données sur un réseau, il est nécessaire de connaître précisément la taille des entiers qu'on utilise. Dans ce cas, on peut inclure `stdint.h` pour avoir accès à des synonymes qui donnent la taille. La table 1.5 donne les noms de ces types.

| Modèle | Unix | | Windows | |
|---|---|---|---|---|
| | 32 bits | 64 bits | 32 bits | 64 bits |
| ILP32 | ✓ | | ✓ | |
| LLP64 | | | | ✓ |
| LP64 | | ✓ | | |

TABLE 1.4 – Modèles de données des principaux systèmes

| Taille | Signé | Non-signé |
|---|---|---|
| 8 | `int8_t` | `uint8_t` |
| 16 | `int16_t` | `uint16_t` |
| 32 | `int32_t` | `uint32_t` |
| 64 | `int64_t` | `uint64_t` |

TABLE 1.5 – Types entiers fournis par stdint.h

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

### Flottants

Les nombres flottants sont représentés par deux types : `float` et `double`. Le langage C suit la norme IEEE 754 sur la représentation des nombres flottants et donc, quel que soit le système, un `float` est un flottant simple précision qui est représenté sur 32 bits, tandis qu'un `double` est un flottant double précision qui est représenté sur 64 bits.

Généralement, on utilise le type `double` pour représenter un nombre flottant. L'exception majeure à cette règle concerne les jeux vidéos, et plus généralement toutes les applications qui vont utiliser la carte graphique. En effet, les cartes graphiques sont optimisées pour traiter des `float` et donc, ces applications utilisent également des `float` en interne pour faciliter les transferts vers la carte graphique.

> **Recommandation #4** On utilise le type `double` pour représenter un flottant, sauf dans le cas d'applications graphiques.

### Vide

Le langage C dispose d'un type vide appelé `void`. Ce type n'a aucune valeur possible, il n'est donc pas possible de déclarer une variable de type `void`. Ce type a deux usages principaux. Premièrement, il permet de spécifier le type de retour d'une fonction qui ne renvoie aucun résultat. Deuxièmement, il sert à former le pointeur générique `void *` (voir la section 1.3.2).

### Booléen

Il n'existait pas de type booléen en C89. Cet oubli a été réparé en C99 par l'ajout d'un type booléen nommé `_Bool`. Heureusement, ce type est également accessible via le nom `bool` si on inclut `stdbool.h`. Dans ce cas, les deux constantes `true` et `false` sont également définie.

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

Le type booléen est en fait un type entier particulier et les deux constantes `true` et `false` ont pour valeurs respectives `1` et `0`.

#### Alias

Il existe des alias pour certains types entiers : `size_t` et `ptrdiff_t`.

`size_t` est défini comme un entier non-signé suffisamment grand pour représenter la taille de n'importe quel objet en mémoire. Il sert également à indicer les tableaux. Quant à `ptrdiff_t`, il est défini comme un entier signé suffisamment grand pour représenter une différence entre deux adresses en mémoire. Il joue un rôle dans l'arithmétique des pointeurs (voir la section 1.3.2).

> **Recommandation #5** On utilise toujours `size_t` pour représenter une taille ou un indice dans un tableau.

Ces deux types sont définis dans le fichier `stddef.h`. Mais généralement, ce fichier est inclus indirectement du fait de l'utilisation de ces deux types dans la bibliothèque standard. Il est donc rare de l'inclure directement.

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

### 1.2.2 Opérateurs

Le langage C dispose des principaux opérateurs : opérateurs arithmétiques (`+`, `-`, `*`, `/`, `%`), opérateurs relationnels (`==`, `!=`, `<`, `>`, `<=`, `>=`), opérateurs logiques (`&&`, `||`, `!`), opérateurs binaires (`&`, `|`, `^`, `~`, `<<`, `>>`). Après avoir vu la priorité et l'associativité des opérateurs, nous ferons un zoom sur certains d'entre eux dont l'utilisation peut cacher des pièges.

#### Priorité et associativité

La table 1.6 présentent les opérateurs du langage C avec leur priorité et leur associativité.

La priorité d'un opérateur permet de savoir quel opérateur sera appliqué en premier s'il n'y a pas de parenthèses. Par exemple, dans l'expression `3 + 4 * 5`, l'opérateur `*` est prioritaire par rapport à l'opérateur `+` (sa priorité est inférieure), et donc il faut comprendre cette expression de la manière suivante : `3 + (4 * 5)`.

Si une expression contient des opérateurs de même priorité, alors c'est la règle d'associativité qui va déterminer l'opérateur à appliquer en premier. Dans le cas d'une associativité à gauche, on commencera par appliquer l'opérateur le plus à gauche, et inversement pour une associativité à droite. Par exemple, dans l'expression `3 + 4 - 5`, les opérateurs `+` et `-` ont la même priorité et ont une associativité à gauche. Il faut donc comprendre cette expression de la manière suivante : `(3 + 4) - 5`.

Connaître les règles de priorité et d'associativité permet d'éviter de surparenthéser une expression.

| Priorité | Opérateur | Description | Associativité |
|---|---|---|---|
| 1 | `++` | Incrémentation postfixé | Gauche |
| | `--` | Décrémentation postfixé | |
| | `[]` | Indice de tableau | |
| | `.` | Accès | |
| | `->` | Accès par un pointeur | |
| 2 | `++` | Incrémentation préfixé | Droite |
| | `--` | Décrémentation préfixé | |
| | `+` | Plus unaire | |
| | `-` | Moins unaire | |
| | `!` | NON logique | |
| | `~` | NON binaire | |
| | `*` | Déréférencement (1.3.2) | |
| | `&` | Référencement (1.3.2) | |
| | `sizeof` | Taille | |
| 3 | `*` | Multiplication | Gauche |
| | `/` | Division | |
| | `%` | Modulo | |
| 4 | `+` | Addition | |
| | `-` | Soustraction | |
| 5 | `<<` | Décalage à gauche | |
| | `>>` | Décalage à droite | |
| 6 | `<` | Test d'infériorité | |
| | `<=` | Test d'infériorité ou égalité | |
| | `>` | Test de supériorité | |
| | `>=` | Test de supériorité ou égalité | |
| 7 | `==` | Test d'égalité | |
| | `!=` | Test d'inégalité | |
| 8 | `&` | ET binaire | |
| 9 | `^` | XOR binaire | |
| 10 | `\|` | OU binaire | |
| 11 | `&&` | ET logique | |
| 12 | `\|\|` | OU logique | |
| 13 | `?:` | Opérateur ternaire | Droite |
| 14 | `=` | Affectation | |
| | `+=` | Affectation avec addition | |
| | `-=` | Affectation avec soustraction | |
| | `*=` | Affectation avec multiplication | |
| | `/=` | Affectation avec division | |
| | `%=` | Affectation avec modulo | |
| | `<<=` | Affectation avec décalage gauche | |
| | `>>=` | Affectation avec décalage droite | |
| | `&=` | Affectation avec ET binaire | |
| | `^=` | Affectation avec XOR binaire | |
| | `\|=` | Affectation avec OU binaire | |

TABLE 1.6 – Principaux opérateurs en C

### Opérateurs d'incrémentation et décrémentation

Les opérateurs d'incrémentation (`++`) et de décrémentation (`--`) existent en version préfixée et postfixée. La différence entre les deux tient dans le résultat d'une expression utilisant une incrémentation ou une décrémentation. Dans la version préfixée, l'expression prend la valeur de la variable après l'opération. Dans la version postfixé, l'expression prend la valeur de la variable avant l'opération. Dit autrement, `++i` est équivalent à `{ i = i + 1; return i }` tandis que `i++` est équivalent à `{ tmp = i; i = i + 1; return tmp; }`.

```
int a = 1;
int b = a++; // a = 2, b = 1
int c = 1;
int d = ++c; // c = 2, d = 2
```

> **Recommandation #6** En cas de choix, il faut toujours préférer la version préfixée des opérateurs d'incrémentation et de décrémentation.

### Opérateur `sizeof`

L'opérateur `sizeof` permet d'obtenir la taille en octets de n'importe quel type ou expression *à la compilation*. La taille renvoyée a pour type `size_t`.

```
size_t s1 = sizeof(char); // s1 = 1

int x;
size_t s2 = sizeof(x); // s2 = sizeof int
```

### Opérateurs binaires

Les opérateurs binaires `&`, `|` et `^` ont historiquement une priorité inadéquate. En effet, dans les premières version de C, les opérateurs logiques n'existaient pas et on se servait des opérateurs binaires pour réaliser des combinaisons logiques d'expressions. Par la suite, les opérateurs logiques ont été introduits mais les opérateurs binaires n'ont pas changé de priorité.

En particulier, la priorité des opérateurs d'égalité et d'inégalité étant inférieure, un problème apparaît dans des expressions pour vérifier la présence de certains bits dans un *flag*. L'expression `a & b != 0` est alors comprise comme `a & (b != 0)` alors qu'on s'attend plutôt à `(a & b) != 0`. Dans ce cas, les parenthèses sont obligatoires.

```
#define READABLE 0x01
#define WRITABLE 0x02

unsigned int flag = READABLE | WRITABLE;

if ((flag & READABLE) != 0) {
  printf("Read");
}
```

```c
if ((flag & WRITABLE) != 0) {
  printf("Write");
}
```

#### Opérateurs de comparaison

La comparaison entre des entiers utilisant des opérateurs d'inégalité peut parfois poser problème quand un des entiers est signé et l'autre est non-signé. En effet, le compilateur ne sait pas s'il faut convertir l'entier signé en non-signé ou l'inverse. Les compilateurs émettent généralement un avertissement mais compilent quand même le code en faisant un choix arbitraire qui peut ne pas convenir.

> **Recommandation #7** On ne compare jamais un entier signé avec un entier non-signé.

#### Opérateur d'affectation

L'affectation `a = b` est une expression comme les autres dans ce sens où elle renvoie une valeur. La valeur de cette expression est le résultat de l'évaluation de sa partie droite. Cette particularité qui n'existe pas dans tous les langages permet notamment de chaîner les affectations. L'opérateur `=` ayant une associativité à droite, l'expression `a = b = c` signifie qu'on fait l'affectation de `c` à `b` puis qu'on prend la valeur de cette affectation et qu'on l'affecte à `a`. Dit autrement, cette expression est équivalente à `a = (b = c)`.

### 1.2.3 Types structurés

Il est possible de définir un type structuré (ou type agrégé ou plus simplement une structure) grâce au mot-clef `struct`. La liste des champs est définie entre accolades avec un type et un nom pour chaque champs suivi d'un point-virgule. Enfin, il ne faut pas oublier le point-virgule à la fin de la définition.

```c
struct date {
  int day;
  int month;
  int year;
};
```

Si on définit une structure `date` comme dans l'exemple précédent, le nom du type est bien `struct date` et pas juste `date`. Donc, on utilise le nom `struct date` quand on déclare une variable de type `date`.

L'accès aux champs d'une structure se fait grâce à l'opérateur `.` (point). L'initialisation d'une structure, au moment de sa déclaration peut être réalisée à l'aide d'une expression entre accolades avec les valeurs des différents champs.

```c
struct date d1;
d1.day = 14;
```

```
d1.month = 7;
d1.year = 1789;

struct date d2 = { 14, 7, 1789 };

struct date d3 = d2;
```

L'affectation d'une variable utilisant un type structuré affecte en réalité tous les champs un par un, ce qui permet d'éviter d'oublier un des champs.

### 1.2.4 Types énumérés

Il est possible de définir un type énuméré (ou plus simplement une énumération) grâce au mot-clef `enum`. La liste des membres de l'énumération est définie entre accolades avec un nom, généralement écrit en majuscule (voir la section 3.2.1), éventuellement suivie d'une valeur entière, et suivie d'une virgule, y compris le dernier membre. Enfin, il ne faut pas oublier le point-virgule à la fin de la définition.

```
enum color {
  COLOR_RED,
  COLOR_GREEN,
  COLOR_BLUE,
};
```

Si aucune valeur n'est donnée explicitement, le premier membre a pour valeur 0 et les suivants sont incrémentés de 1 à chaque fois. Généralement, il n'y a pas besoin de donner une valeur explicite. Les seuls cas où une valeur explicite est utilisée concernent le détournement des types énumérés pour définir des constantes, notamment des masques de bits.

Les noms des membres d'une énumération sont plongés dans l'espace de nom commun. Si on n'y prend pas garde, il peut arriver qu'un même nom soit utilisé dans deux énumérations différentes auquel cas la compilation s'arrête à cause d'un conflit de noms. Une manière de contourner ce problème est de préfixer les noms des membres d'une énumération, généralement avec le nom de l'énumération ou une version abrégée. Un avantage de cette technique est qu'il est possible de savoir rapidement à quelle énumération se rapporte le nom du membre.

> **Recommandation #8** Le nom des membres d'une énumération doivent toujours avoir un préfixe commun pour éviter un conflit de nom.

Si on définit une énumération `color` comme dans l'exemple précédent, le nom du type est bien `enum color` et pas juste `color`. Donc, on utilise le nom `enum color` quand on déclare une variable de type `color`.

```
enum color c1 = COLOR_RED;
enum color c2 = COLOR_GREEN;
enum color c3 = c2;
```

### 1.2.5 Type union

Il est possible de définir un type union (ou plus simplement une union) grâce au mot-clef `union`. La liste des champs se définit exactement de la même manière que pour une structure. Cependant, contrairement à une structure, les champs d'une union partagent le même emplacement mémoire. La conséquence est qu'on ne peut utiliser qu'un seul champ de l'union à la fois.

```
union number {
  double real;
  long integer;
};
```

Pour savoir quel est le champs qui est actif à chaque instant, on a plusieurs possibilités : soit la logique du programme fait qu'on connaît cette information ; soit on utilise une union taguée.

Une union taguée est l'association d'une union et d'une énumération. L'énumération sert alors à déterminer le champs actif. Dans l'exemple suivant, si `tag` vaut `TAG_REAL`, alors c'est le champs `real` qu'il faut utiliser dans l'union suivante, et ainsi de suite.

```
struct number {
  enum { TAG_REAL, TAG_INTEGER } tag;
  union {
    double real;
    long integer;
  } value;
};
```

### 1.2.6 Types constants

Il est possible de définir une variable d'un type constant avec le mot-clef `const`. Dans ce cas, la valeur de la variable est définie à la déclaration et ne peut plus changer par la suite. Le compilateur vérifie que cette contrainte est respectée et émet une erreur dans le cas contraire.

```
const int i = 42;
const struct date d = { 14, 7, 1789 };
```

### 1.2.7 Alias de type

Il est possible de définir des alias de type à l'aide du mot-clef `typedef`, c'est-à-dire de substituer le nom d'un type par un autre.

Il est d'usage de nommer les alias de types avec le suffixe `_t`. La bibliothèque standard utilise cette convention de nommage : `size_t`, `int8_t`, etc.

> **Recommandation #9** Le nom d'un alias de type doit terminer par le suffixe `_t`.

```c
typedef int integer_t;
typedef struct date date_t;
typedef enum color color_t;

typedef struct {
  double real;
  double imaginary;
} complex_t;
```

Une fois défini, l'alias de type s'utilise directement, sans mot-clef devant. Comme il s'agit d'un alias, les variables ont en réalité le type d'origine. Dit autrement, on ne crée pas un nouveau type avec un alias de type, juste un synonyme parfois plus facile à utiliser.

```c
integer_t i = 3;
date_t d = { 14, 7, 1789 };
color_t c = COLOR_RED;

complex_t j = { 0., 1. };
```

L'intérêt des alias est de pouvoir changer plus simplement un type dans un ensemble de fichier : il suffit de changer la définition de l'alias. Cependant, il ne faut pas non plus abuser des alias (notamment les alias de types primitifs) au risque de rendre le programme confus.

## 1.3 Tableaux, pointeurs et chaînes de caractères

### 1.3.1 Tableaux

Un tableau est un espace mémoire contenant plusieurs éléments contigus du même type. Il existe deux types de tableaux en C, les tableaux statiques dont la taille est connue à la compilation et les tableaux dynamiques dont la taille est connue à l'exécution. Les tableaux sont indicés à partir de 0.

#### Tableaux statiques

Un tableau statique est défini en indiquant le type des éléments, le nom du tableau et le nombre d'éléments entre crochets. Ce nombre d'éléments peut être omis si le tableau est initialisé, sa taille est alors calculée automatiquement à la compilation. Si le nombre d'éléments indiqué est plus grand que le nombre d'éléments dans l'initialisation, alors c'est le nombre d'éléments indiqué qui est pris en compte et des éléments du tableau ne sont alors pas initialisés.

```c
enum color colors[3]; // array of 3 colors
char vowels[] = {
  'a', 'e', 'i', 'o', 'u'
}; // array of 5 chars
int array[10] = { 1, 2 }; // array of 10 ints
```

Pour accéder aux éléments du tableau, on utilise les crochets. Les compilateurs ne vérifient pas si l'indice donné correspond bien à un élément légitime

du tableau, il faut donc faire attention et s'assurer que c'est bien le cas. Cette erreur est une source de bugs très fréquente.

```
enum color c = colors[0]; // first color
char v = vowels[10]; // BOOM!
```

> **Recommandation #10** On vérifie toujours que l'indice utilisé pour accéder à un élément d'un tableau correspond à un élément valide, c'est-à-dire que l'indice est bien strictement inférieur au nombre d'éléments du tableau.

#### Tableaux à plusieurs dimensions

Il est possible de définir des matrices et plus généralement des tableaux à plusieurs dimensions. Dans ce cas, il faut préciser toutes les dimensions, à l'exception de la première qui peut être déduite de l'initialisation du tableau.

```
float transform[3][3];
int directions[][2] = {
  {  1,  0 },
  {  0,  1 },
  { -1,  0 },
  {  0, -1 }
};
```

#### Taille d'un tableau statique

Pour connaître à la compilation le nombre d'éléments d'un tableau, on peut utiliser `sizeof`. Cependant, `sizeof` avec le nom du tableau renvoie la taille en octets du tableau, pas le nombre d'éléments. Une solution consiste alors à diviser la taille du tableau par la taille de son premier élément. Cette solution fonctionne uniquement avec les tableaux à une dimension.

```
int loto[] = { 42, 23, 18, 14, 1 };
size_t count = sizeof(loto) / sizeof(loto[0]);
// count = 5
```

Le plus simple et le plus sûr reste de définir une constante pour définir la taille du tableau et d'utiliser cette constante dans la suite.

### 1.3.2 Pointeurs

#### Définition et représentation

Le langage C est un langage bas niveau, ce qui signifie qu'il peut travailler directement avec des adresses en mémoire. Un pointeur est une adresse vers un espace mémoire typé.

Plus précisément, pour tout type `T`, on peut définir un type «pointeur sur `T`», noté `T *`. Une variable de type `T *` peut contenir l'adresse d'un objet en


FIGURE 1.1 – Représentation graphique d'un pointeur

mémoire de type `T`. Le type `T *` étant un type comme les autres, on peut définir un type «pointeur sur pointeur sur `T`», noté `T **`. Et ainsi de suite.

La figure 1.1 montre la représentation graphique usuelle d'un pointeur. La variable `p` est un pointeur qui contient l'adresse de la variable `a`, ce que montre la flèche. Dans ce cas, on dit que «`p` pointe sur `a`».

Quand on déclare une variable de type pointeur, l'étoile ne fait pas réellement partie du type, ce qui peut conduire à des erreurs. Pour éviter ces erreurs, l'usage est de coller l'étoile au nom de la variable plutôt qu'au nom du type.

```
int *p;
// p is a pointer on an int
// *p is an int
int *p, q;
// q is an int!
int *p, *q;
// *p and *q are ints
```

> **Recommandation #11** Généralement, dans la déclaration d'un pointeur, on colle l'étoile au nom de la variable.

### Opérateurs de référencement et déréférencement

L'opérateur de référencement, noté `&`, permet de récupérer l'adresse d'une variable, ou plus généralement d'un objet en mémoire. Cet opérateur est le premier pourvoyeur de pointeur puisqu'il permet de créer un pointeur à partir de n'importe quel objet en mémoire, qu'il s'agisse d'une simple variable, d'un membre d'une structure ou d'un élément dans un tableau.

L'opérateur de déréférencement, noté `*`, permet de récupérer le contenu de la variable pointée, ou plus généralement le contenu de l'objet en mémoire pointé.

```
int a = 3;
int *p = &a;
// p has the address of a
int b = *p;
// b = 3
```

Dans le cas où le type de l'objet pointé est une structure ou une union, on peut utiliser l'opérateur `->` pour accéder directement aux champs de la structure. Si `p` est un pointeur sur une structure contenant un champs `f` alors, `p->f` est exactement équivalent à `(*p).f`.

```
struct date today = { 18, 3, 2016 };
```

```c
struct date *birthday = &today;
birthday->day = 21; // same as (*birthday).day = 21;
```

### Pointeur et `const`

Dans la déclaration d'une variable de type pointeur, il est possible de combiner le type pointeur avec le mot-clef `const` qui donne une signification différente suivant où il est placé dans le nom du type. Si le mot-clef `const` se trouve avant l'étoile, alors il porte sur la donnée pointée, c'est-à-dire que la donnée pointée est constante ; tandis que s'il se trouve après l'étoile, il porte sur la variable, c'est-à-dire que c'est la variable qui est constante. Il peut également apparaître deux fois avant et après l'étoile pour cumuler les deux propriétés.

```c
const int *p;
// p is a pointer to a const int
// p can be modified
// *p can NOT be modified
int * const q;
// q is a constant pointer to an int
// q can NOT be modified
// *q can be modified
const int * const r;
// r is a constant pointer to a const int
// r can NOT be modified
// *r can NOT be modified
```

### Pointeur générique

Un pointeur qui a le type `void *` est un pointeur générique. Il a la particularité de pouvoir être affecté à n'importe quel autre pointeur sans transtypage, et inversement on peut lui affecter n'importe quel autre pointeur sans transtypage.

```c
double data = 42.0;
double *pointer = &data;
void *generic = pointer; // no need to cast
uint64_t *bits = generic; // no need to cast
```

### Pointeur `NULL`

`NULL` est une valeur de pointeur invalide, généralement elle vaut 0. Elle peut servir à initialiser une variable de type pointeur qu'on ne saurait pas initialiser autrement. Le pointeur `NULL` est généralement un pointeur générique, qui permet de l'affecter à n'importe quelle valeur de type pointeur.

Déréférencer un pointeur `NULL` provoque une erreur à l'exécution et l'arrêt immédiat du programme. C'est une erreur commune quand on débute la programmation en C. La section 3.2 couvre les bonnes pratiques pour éviter ces erreurs, et les outils pour permettre de les corriger rapidement.

**Recommandation #12** Un pointeur sans valeur initiale valide doit être initialisé à `NULL`.

### 1.3.3 Arithmétique sur les pointeurs

Le langage C permet de réaliser des opérations avec les pointeurs. Un principe de base de toute ces opérations est qu'elle tiennent compte du type pointé et donc de la taille des objets pointés.

#### Addition d'un pointeur et d'un entier

Étant donné une variable `p` de type pointeur sur un type `T`, il est possible d'ajouter un entier `n` à `p`. Le résultat `p + n` est un pointeur du même type que `p` dont la valeur vaut l'adresse `p` à laquelle on a ajouté `n` fois la taille de `T`. Ainsi, `p+1` désigne un élément de type `T` qui se situe juste après l'élément de type `T` pointé par `p`, etc.

```
int array[10];
int *p = &array[0];
// p points to the first element of array
int *q = p + 1;
// q points to the second element of array
q++;
// q points to the third element of array
```

#### Soustraction de deux pointeurs

Étant donné deux variables `p` et `q` de type pointeur sur un type `T`, il est possible de soustraire `p` à `q`. Le résultat `p - q` est un entier signé de type `ptrdiff_t` dont la valeur est le nombre d'objet de type `T` entre `p` et `q`.

```
int array[10];
int *p = &array[0];
int *q = p + 3;
ptrdiff_t diff1 = q - p; // diff1 = 3
ptrdiff_t diff2 = p - q; // diff2 = -3
```

### 1.3.4 Équivalence tableau/pointeur

Un tableau est un pointeur vers le premier élément. On peut donc réaliser une conversion implicite d'un tableau vers un pointeur. À l'inverse, un pointeur peut être vu comme le début d'un tableau, en ce sens qu'il ne pointe pas vers un élément unique mais vers un ensemble d'éléments contigus. Un tableau `a` de type `T` a pour type `T * const`.

L'accès à un élément d'un tableau `a[i]` est en fait complètement équivalent à `*(a + i)`. De même, si on dispose d'un pointeur `p` qui représente un tableau, on peut accéder à un élément avec `p[i]` qui est équivalent à `*(p + i)`. En particulier, `p[0]` est équivalent à `*p`.

```c
int array[10];
// array is a int * const
int *p = array;
// implicit conversion
int x = p[2];
// equivalent to *(p + 2)
int y = p[0];
// equivalent to *p
```

Si tableau et pointeur ont une certaine équivalence, il faut cependant faire attention. Si on dispose uniquement d'un pointeur `p` qui représente un tableau, on ne peut pas savoir la taille du tableau juste avec le pointeur. En particulier, l'opérateur `sizeof` qui permet de connaître la taille d'un tableau statique ne permet absolument pas de renvoyer la taille d'un tableau auquel on a accès via un pointeur. Dans ce cas, `sizeof(p)` renverra la taille d'un pointeur.

> **Recommandation #13** Il ne faut jamais utiliser `sizeof` pour obtenir la taille d'un tableau qu'on connaît via un pointeur.

Pour régler ce problème, on n'utilise que très rarement un pointeur seul pour désigner un tableau, il est très souvent accompagné d'une variable de type `size_t` pour connaître la taille du tableau.

### 1.3.5 Chaînes de caractères

Une chaîne de caractères est un tableau de caractères de type `char` dont le dernier élément est le caractère `'\0'`. En tant que tableau, tout ce qui est réalisable sur un tableau est également réalisable sur une chaîne de caractère. Mais une chaîne de caractères a cette particularité d'avoir un élément final distinctif qui permet de connaître la fin de la chaîne. Tous les algorithmes qui manipulent des chaines de caractères utilisent le `'\0'` final pour savoir si la fin de la chaîne est atteinte. Le type d'une chaine de caractères est généralement `char *` ou `const char *`.

Il existe deux chaînes de caractères particulières. Premièrement la chaîne nulle, qui est représentée par le pointeur `NULL`. Comme une chaîne de caractère est représentée par un pointeur, le pointeur `NULL` est un cas particulier de chaîne de caractères. Deuxièmement, la chaîne vide, notée `""`, qui est un tableau d'un seul caractère, le caractère `'\0'`.

Un litéral de type chaîne de caractères s'écrit généralement entre guillemets double mais ce n'est absolument pas une obligation.

```c
char str1[] = { 'a', 'b', 'c', '\0' };
// a valid string
char str2[] = "abc";
// same as str1
char *str3 = "abc";
// same as str1 but can not be modified
```

## 1.4 Structures de contrôle

Les structures de contrôle du langage C sont quasiment les mêmes que dans les autres langages. On peut les séparer en deux grandes catégories : les structures de contrôle conditionnelles (`if`, `switch`) et les structures de contrôle itératives, aussi appelées boucles (`while`, `for`).

Dans tous les cas, la structure de contrôle contient (au moins) un bloc d'instruction à éxécuter de manière conditionnelle. Un usage déraisonnable consiste, quand ce bloc ne contient qu'une seule instruction, à se passer des accolades. C'est la porte ouverte à des erreurs futures où on va ajouter une seconde instruction et oublier d'ajouter les accolades.

> **Recommandation #14** On utilise toujours des accolades pour le bloc d'instructions d'une structure de contrôle, y compris quand il ne contient qu'une seule instruction.

Les conditions utilisées dans les structures de contrôle peuvent être des expressions booléennes mais aussi des expressions entières ou des pointeurs. Dans ces derniers cas, l'expression `e` qui sert pour la condition est équivalente respectivement à `(e != 0)` ou `(e != NULL)`.

### 1.4.1 Structures conditionnelles

#### Structure `if`

La structure `if` permet d'exécuter un bloc d'instructions sous une certaine condition.

```
if (i > 0) {
  // i is positive
}
```

On peut ajouter un second bloc d'instructions après le mot-clef `else` qui est exécuté si la condition est fausse.

```
if (i % 2 == 0) {
  // i is even
} else {
  // i is odd
}
```

Si la dernière instruction de la fin du premier bloc est un `return` (ou une autre instruction de rupture de flot comme `break` ou `continue`), alors on n'utilisera pas le mot-clef `else` qui devient inutile puisque si la condition est vraie, les instructions après le bloc d'instruction ne seront pas exécutée quoiqu'il arrive.

> **Recommandation #15** On ne met jamais de `else` après un `return`, un `break` ou un `continue`.

#### Structure `switch`

Dans le cas où on doit comparer une variable de type caractère ou entier à un ensemble de valeurs litérales, on peut utiliser la structure **`switch`**. On n'utilise jamais **`switch`** sur des chaînes de caractères, ce qui reviendrait à comparer des pointeurs (et donc à se tromper). De même, on n'utilise jamais **`switch`** sur des flottants puisque l'égalité entre flottants est une notion à manier avec beaucoup de précaution en programmation.

> **Recommandation #16** Dès qu'on doit comparer une variable à plus de deux valeurs, on préfère utiliser **`switch`** plutôt qu'une suite de **`if-else`**.

Chaque cas est introduit par le mot-clef **`case`**. On peut regrouper plusieurs cas à la suite en mettant un **`case`** sur chaque ligne. Chaque cas finit par une instruction **`break`**. Enfin, le cas par défaut **`default`** permet de récupérer tous les cas non-traités auparavant.

> **Recommandation #17** Dans un **`switch`**, il faut toujours mettre un **`default`** et gérer le cas par défaut correctement.

```
switch (i) {
  case 0:
    // i is 0
    break;
  case 1:
  case 2:
    // i is 1 or 2
    break;
  default:
    // i is something else
    break;
}
```

Le mot-clef **`break`** peut être omis si on veut exécuter les instructions du cas suivant. Cette situation arrive peu fréquemmment mais dans ce cas, on l'indique généralement par un commentaire «fallthrough» à la place du **`break`** pour indiquer au lecteur que ce n'est pas un oubli.

### 1.4.2 Structures itératives

#### Structure `for`

La structure **`for`** permet d'exécuter un bloc d'instructions en boucle sous le contrôle d'une condition. Plus précisément, la structure **`for`** est contrôlé par trois parties :

1. l'initialisation, qui contient généralement une instruction d'initialisation d'une variable et qui est exécutée une seule fois avant la boucle ;

2. la condition, qui permet de savoir via un test sur la variable si on reste dans la boucle ou non et qui est évaluée au début de la boucle ;
3. la mise à jour, qui permet généralement de mettre à jour la variable et qui est exécuté à la fin de la boucle.

```
for (int i = 0; i < 10; ++i) {
  // do something with i
}
```

Chacune de ces trois parties est optionnelle. Il arrive fréquemment que la variable qui sert à itérer n'ait pas d'autre utilité dans le reste du programme. Dans ce cas, on la déclare directement dans la partie initialisation du `for`.

> **Recommandation #18** Une variable de boucle spécifique à une structure `for` doit être déclarée dans la partie initialisation du `for`.

#### Structure `while` et `do-while`

La structure `while` permet d'exécuter un bloc d'instructions en boucle sous le contrôle d'une condition évaluée au début de la boucle. Le bloc peut donc ne jamais être exécuté.

```
while (i != 0) {
  i = i / 2;
}
```

La structure `do-while` permet d'exécuter un bloc d'instructions en boucle sous le contrôle d'une condition évaluée à la fin de la boucle. Le bloc est donc toujours exécuté au moins une fois.

```
do {
  i = i + 2;
} while (i % 7 != 0);
```

#### Instructions `continue` et `break`

Le bloc d'instructions des structures itératives peut être interrompu à tout moment par deux instructions spécifiques.

L'instruction `continue` permet d'interrompre le flot et d'aller directement à la fin du bloc. Elle permet donc d'initier la boucle suivante, en examinant la condition de boucle.

L'instruction `break` permet d'interrompre le flot et de terminer la boucle immédiatement.

#### Boucle infinie

Une boucle infinie est une boucle qui ne termine jamais, sauf quand le programme s'arrêtera. Il y a deux manières de réaliser une boucle infinie : soit avec un «`for (;;)`», soit avec un «`while (true)`» (ou «`while (1)`»). On rencontre régulièrement les deux manières de faire.

## 1.5 Fonctions

Les fonctions sont les éléments de base de la construction d'un programme en C.

### 1.5.1 Déclaration et définition

Une fonction doit être soit définie, soit déclarée avant d'être utilisée. Dit autrement, on ne peut pas appeler une fonction qui n'a pas été définie ou déclarée auparavant. C'est pour cette raison qu'il existe des en-têtes (voir la section 3.1.1).

La définition d'une fonction comprend, dans l'ordre, son type de retour, son nom, ses paramètres, puis le corps de la fonction qui est une suite d'instructions.

```
int max(int a, int b) {
  if (a > b) {
    return a;
  }

  return b;
}
```

La déclaration d'une fonction est une description de la fonction sous forme d'un prototype, c'est-à-dire son type de retour, son nom et ses paramètres. Une fonction déclarée permet d'indiquer qu'une fonction existe et qu'on peut l'utiliser.

```
int max(int a, int b);
```

Il existe un cas particulier : les fonctions sans paramètre. Dans ce cas, la liste des paramètres est remplacé par `void`. La définition de la fonction est identique, on indique `void` à la place de la liste de paramètre.

```
int zero_argument(void);
```

En effet, une fonction dont la liste de paramètres est vide signifie que la fonction prend un nombre indéfini de paramètres.

```
int unknown_arguments();
```

> **Recommandation #19** On indique toujours `void` à la place des paramètres pour les fonctions sans paramètre.

### 1.5.2 Arguments

En C, tous les arguments sont passés par valeur. Cela signifie donc qu'une copie des arguments est réalisée à l'intérieur de la fonction. Il importe donc de ne pas passer en paramètre des arguments d'un type d'une taille trop importante. Donc, hormis les types de base qui sont passés directement, on préférera une autre méthode donnée plus loin.

Pour mimer un passage par référence, on utilise un pointeur sur une variable. La fonction peut alors modifier la variable qui se trouve à l'extérieur de la fonction via ce pointeur. L'exemple suivant montre comment échanger le contenu de deux variables. Cette fonction est généralement codée avec des arguments passé par référence. En C, on utilise deux pointeurs.

```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Pour les objets de taille importante, il est possible d'utiliser un pointeur constant en paramètre. Ce pointeur constant assure à l'utilisateur de la fonction que les données pointées ne seront pas modifiées par la fonction.

```
struct big {
  // many fields
};

void function_using_big(const struct big *b) {
  // cannot modify *b
}
```

### 1.5.3 Pointeur de fonction

Un pointeur de fonction est une adresse d'une fonction avec un prototype déterminé. Les pointeurs de fonction sont particuliers et ne peuvent pas être affectés à un pointeur générique.

Il est possible de définir des variables de type pointeur de fonction directement. Cependant, il est plus facile de définir un alias de type car la lecture des types pointeur de fonction peut s'avérer très complexe.

> **Recommandation #20** On utilise très souvent un alias de type pour désigner un pointeur de fonction.

Pour affecter l'adresse d'une fonction à une variable de type pointeur de fonction, on peut utiliser l'opérateur de référencement. Cependant, il existe une conversion automatique depuis le nom d'une fonction vers le pointeur vers cette fonction et donc l'opérateur de référencement n'est pas utile.

```
void f(int x);

void (*fp1)(int) = &f;
void (*fp2)(int) = f; // same as &f

typedef void (*func_t)(int);

func_t fp3 = &f;
func_t fp4 = f; // same as &f
```

Pour appeler la fonction pointée, on peut utiliser l'opérateur de déréférencement mais il est d'usage d'utiliser directement la variable de type pointeur de fonction. Là encore, il existe une conversion automatique car il n'y a aucune ambiguïté étant donné la nature particulière du pointeur.

```
typedef double (*trig_t)(double);
trig_t f = cos;
double a1 = (*f)(0.0);
double a2 = f(0.0); // same as (*f)(0.0);
```

### 1.5.4 Fonction principale `main`

La fonction principale d'un programme en C s'appelle `main`. C'est la fonction qui est appelée lors de l'exécution du programme. Elle peut avoir l'un des deux prototypes suivants :

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

Le paramètre `argc` contient le nombre d'arguments du programme, et `argv` est un tableau de `argc` chaînes de caractères, chaque chaîne représentant un argument du programme. La valeur renvoyée par `main` est la valeur de retour du programme.

Il y a toujours au moins un argument, c'est-à-dire `argc` est toujours au moins égal à 1. `argv[0]` contient le nom du programme tel qu'il a été appelé, avec éventuellement le chemin d'accès au programme.

## 1.6 Directives préprocesseur

### 1.6.1 Le préprocesseur

Le préprocesseur est un programme qui procède à des transformations sur un code source, avant l'étape de compilation. Il interprète des directives qui commencent par le caractère `#` et les remplace par du texte qui est ensuite envoyé au compilateur.

Il n'y a pas besoin d'appeler le préprocesseur directement, le compilateur s'en charge tout seul. Si on veut voir le fichier source à la sortie du préprocesseur, il est possible de passer l'option `-E` à gcc qui arrêtera le processus juste après l'étape du préprocesseur.

### 1.6.2 Inclusion de fichier

La directive `#include` permet d'inclure le contenu d'un fichier dans un autre. Il existe deux types d'inclusion : les inclusions de fichiers systèmes et les inclusions de fichiers locaux.

L'inclusion de fichiers système se fait en mettant le nom du fichier entre chevrons :

```
#include <file.h>
```

Un fichier système est un fichier qui se trouve dans un des répertoires prédéfinis par le compilateur. Sur un système GNU/Linux, le répertoire `/usr/include` fait généralement partie des répertoires connus du compilateur pour contenir des fichiers systèmes.

L'inclusion de fichiers locaux se fait en mettant le nom du fichier entre guillemets :

```
#include "file.h"
```

Un fichier local est un fichier qui se trouve dans le même répertoire ou dans un répertoire fils par rapport au fichier de base. Il est possible d'indiquer un chemin complet jusqu'au fichier à inclure.

Les fichiers qu'on inclue sont toujours des en-têtes, c'est-à-dire des fichiers avec l'extension `.h` *(Header)* qui contiennent des définitions de types et des déclarations de fonctions. La section 3.1.1 indique comment créer des fichiers d'en-tête et les fichiers sources associés.

> **Recommandation #21** On n'inclue jamais des fichiers sources (`.c`), mais toujours des fichiers d'en-tête (`.h`).

Le chapitre 2 indique les fichiers à inclure pour utiliser les fonctions de la bibliothèque standard du C.

### 1.6.3 Définition de constantes

La directive `#define` permet de définir des constantes qui seront remplacées dans le code source. La constante peut ne pas avoir de valeur.

```
#define M_PI 3.14159265358979323846
#define DEBUG
```

Les constantes n'ont pas de type. Une constante contient juste du texte. Le préprocesseur ne fait que de la substitution de texte. Par exemple, supposons qu'on dispose d'un code source qui contient l'instruction suivante.

```
double x = M_PI / 2;
```

Après le passage du préprocesseur, le code source sera le suivant.

```
double x = 14159265358979323846 / 2;
```

Définir des constantes préprocesseurs est une bonne pratique. Dès qu'un litéral (entier, flottant, caractère, parfois même chaîne de caractère) apparaît dans un code source, il peut être transformé en une constante préprocesseur.

> **Recommandation #22** Le plus souvent, un litéral n'apparaît pas directement dans un code source mais provient d'une constante préprocesseur.

Il existe des constantes préféfinies par le préprocesseur. En particulier, les constantes `__FILE__` et `__LINE__` représentent respectivement le fichier courant et la ligne courante du fichier.

### 1.6.4 Compilation conditionnelle

Les directives `#if`, `#ifdef` et `#ifndef` permettent de compiler conditionnellement, c'est-à-dire de laisser après passage du préprocesseur uniquement le code qui remplit une certaine condition. La directive `#else` permet de créer une alternative. La directive `#endif` termine le bloc de code concerné par la condition.

La directive `#if` prend en paramètre une expression booléenne du préprocesseur, c'est-à-dire faisant uniquement intervenir des constantes du préprocesseur. Il est également possible d'utiliser la fonction préprocesseur `defined` qui permet de savoir si une constante préprocesseur est définie.

```
#if LIB_VERSION >= 100
use_advanced_function();
#else
use_current_function();
#endif
```

```
#if defined(DEBUG)
print_state();
#endif
```

L'utilisation de `#if defined` étant tellement courant, il existe un raccourci : `#ifdef`. De même, l'utilisation de `if !defined` peut-être abrégée en `#ifndef`.

```
#ifdef DEBUG
print_state();
#endif
```

### 1.6.5 Définition de macros

La directive `#define` permet de définir une macro. Une macro ressemble à une fonction mais n'en est pas une !

Supposons qu'on définisse une macro de la manière suivante.

```
#define OP(a,b) ((a) + 3 * (b))
```

Elle peut être utilisée dans une instruction.

```
int x = OP(4,6);
```

Après le passage du préprocesseur, le code source sera le suivant.

```
int x = ((4) + 3 * (6));
```

Il existe des règles incontournables quand on définit des macros : une macro ne doit pas dépasser une ligne ; chaque paramètre est parenthésé dans la définition de la macro ; la définition de la macro doit elle-même être parenthésée.

Pour illustrer la nécessité des deux dernières règles, voici deux exemples qui suppriment respectivement les parenthèses autour des paramètres et les parenthèses autour de la macro. Dans les deux cas, le résultat obtenu n'est pas du tout le résultat attendu.

```
#define OP(a,b) (a + 3 * b)
int x = OP(4, 3 + 3);
int x = (4 + 3 * 3 + 3); /* KABOOM! */
```

```
#define OP(a,b) (a) + 3 * (b)
int x = 2 * OP(4, 6);
int x = 2 * (4) + 3 * (6); /* KABOOM! */
```

> **Recommandation #23** Il est nécessaire de parenthéser correctement les macros préprocesseurs

Il existe deux différences fondamentales entre une macro et une fonction. Premièrement, une fonction a un code binaire en mémoire, une macro n'en a pas, c'est une simple substitution de texte. Deuxièmement, une fonction évalue une seule fois ses paramètres, une macro peut évaluer plusieurs fois ses paramètres.

Pour illustrer la seconde différence, on peut définir la macro `MAX` de la manière suivante.

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

Si on utilise la macro `MAX` avec deux appels de fonction, alors une des deux fonctions sera appelée deux fois.

```
MAX(f(), g());
```

# Chapitre 2

# La bibliothèque standard du C

## 2.1 Introduction

### 2.1.1 Bibliothèque standard

La bibliothèque standard est un ensemble de fonctions de base nécessaire dans la plupart des programmes. Pour pouvoir les utiliser, il est nécessaire d'inclure des en-têtes, c'est-à-dire des fichiers qui contiennent les prototypes des fonctions souhaitées. Il existe 24 fichiers d'en-têtes standard. Le code source suivant indique les en-têtes les plus courants.

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

### 2.1.2 Page de manuel

Une page de manuel *(manpage)* est une documentation complète relative à un programme, une fonction, un fichier, etc. Les pages de manuel sont rangées dans différentes sections. Le nom d'une page de manuel est de la forme `page(section)`. Les pages de manuel relatives à la bibliothèque standard sont dans la section 3. Les pages de manuel de chaque fonction indiquent le ou les en-têtes à inclure pour pouvoir utiliser la fonction concernée.

Pour accéder à une page de manuel, il est nécessaire d'appeler la commande `man` suivi du nom de la fonction. Par exemple, pour la page de manuel de la fonction `malloc(3)`, on tape la ligne de commande suivante.

```
man malloc
```

S'il existe une page de manuel avec un nom identique dans les sections 1 ou 2, alors il est nécessaire de préciser le numéro de section. C'est le cas pour `printf(3)`.

| Séquence | Type attendu |
|---|---|
| `"%i"` | `int` |
| `"%u"` | `unsigned int` |
| `"%f"` | `double` |
| `"%c"` | `char` |
| `"%s"` | `char *` |
| `"%p"` | `void *` |
| `"%zu"` | `size_t` |

TABLE 2.1 – Principales séquences de contrôle

```
man 3 printf
```

Avant d'aller voir votre ami Google, il est toujours bon de consulter la page de manuel d'une fonction pour savoir comment bien l'utiliser. Les pages de manuel sont des sources d'information très utiles à ne surtout pas négliger, malgré le caractère un peu austère de leur présentation. Les informations données dans la suite ne sont qu'un court résumé des pages de manuel.

Dans la suite, les fonctions seront nommées selon leur page de manuel.

## 2.2 Fonctions d'entrée/sortie

Les fonctions d'entrée/sortie permettent d'interagir avec l'utilisateur, ou de manière plus générale, avec le système.

### 2.2.1 Sortie formattée

La fonction `printf(3)` permet d'écrire un texte formaté sur la sortie standard. Elle prend un nombre variable d'arguments qui dépend de la chaîne de format.

```
#include <stdio.h>

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

Une chaîne de format est une chaîne de caractères qui contient des séquences de contrôle (commençant par `%`) permettant d'interpréter le type des arguments et de les insérer correctement dans la chaîne. La table 2.1 donne les principales séquences de contrôle pour les types de base. Pour afficher le caractère `'%'`, on utilise la séquence de contrôle `"%%"`.

Le code suivant donne quelques exemples d'utilisation des chaînes de format ainsi que le résultat attendu en commentaire.

```
printf("This is a string without sequence\n");
// This is a string without sequence

printf("%s is %i year old and is %f meter tall\n",
  "Alice", 20, 1.70);
// Alice is 20 year old and is 1.700000 meter tall
```

| Type | Où | Quand | Comment |
|---|---|---|---|
| statique | segment | compilation | implicite |
| automatique | pile *(stack)* | exécution | implicite |
| dynamique | tas *(heap)* | exécution | explicite |

TABLE 2.2 – Types de mémoire en C

```c
char c = 'a';
printf("'%c' has ASCII code %i\n", c, c);
// 'a' has ASCII code 97
```

### 2.2.2 Entrée

La fonction `fgets(3)` permet d'obtenir une chaîne de caractère depuis un fichier, en particulier depuis l'entrée standard `stdin` (le clavier par défaut). Elle prend en paramètre l'adresse d'un tampon *(buffer)* et sa taille, ainsi que le fichier, `stdin` le plus souvent.

```c
#include <stdio.h>

char *fgets(char *s, int size, FILE *stream);
```

La fonction lit au maximum `size` caractère depuis `stream` et les stocke dans le tampon pointé par `s`. La lecture s'arrête après la fin du fichier (EOF) ou la fin de la ligne. Si une fin de ligne a été lue, elle est ajoutée au tampon. Un caractère de fin de chaîne (`'\0'`) est ajouté après le dernier caractère dans le tampon.

La fonction renvoie `s` en cas de succès, et `NULL` en cas d'erreur ou si la fin du fichier a été atteinte et qu'il n'y a plus rien à lire.

```c
#define BUFSIZE 256
char buf[BUFSIZE];

if (fgets(buf, BUFSIZE, stdin) != NULL) {
  // do something with buf
}
```

## 2.3 Gestion de la mémoire

### 2.3.1 La mémoire en C

Il existe trois grands types de mémoire en C : statique, automatique et dynamique. Le tableau 2.2 donne une vision résumée de ces trois types de mémoire.

La mémoire statique est utilisée pour les variables globales. Elle est allouée implicitement à la compilation. La mémoire automatique est utilisée pour les variables locales aux fonctions ainsi que pour les arguments des fonctions. Elle est allouée à l'exécution, au moment des appels de fonctions, de manière implicite. Elle est libérée à la sortie des fonctions.

La mémoire dynamique est allouée à l'exécution de manière explicite. Les fonctions de gestion de la mémoire présentée par la suite permettent d'allouer et

de libérer de la mémoire dynamique. L'utilisateur doit veiller à bien utiliser ces fonctions de manière à ne pas provoquer d'erreur. La section 3.3 montre comment détecter et corriger les erreurs classiques liées à la gestion de la mémoire.

### 2.3.2 Allocation de la mémoire

La fonction `malloc(3)` alloue `size` octets, et renvoie l'adresse de la mémoire allouée via un pointeur. La mémoire allouée n'est pas initialisée. En cas d'échec, la fonction renvoie le pointeur `NULL`.

```
#include <stdlib.h>

void *malloc(size_t size);
```

Pour un exemple d'utilisation de `malloc(3)`, voir la section 2.3.4.

### 2.3.3 Allocation d'un tableau dynamique

La fonction `calloc(3)` alloue la mémoire nécessaire pour un tableau de `nmemb` éléments de `size` octets, et renvoie un pointeur vers la mémoire allouée. Cette fonction est (presque) équivalente à : `malloc(nmemb * size)`. Une différence notable est que la mémoire renvoyée par `calloc(3)` est initialisée à zéro.

```
#include <stdlib.h>

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

Pour un exemple d'utilisation de `calloc(3)`, voir la section 2.3.4.

### 2.3.4 Libération de la mémoire

La fonction `free(3)` libère l'espace mémoire pointé par `ptr` et qui a été allouée précédemment par `malloc(3)` ou `calloc(3)`. Si `ptr` est `NULL`, aucune opération n'est effectuée.

```
#include <stdlib.h>

void free(void *ptr);
```

Voici un exemple complet d'utilisation des fonctions de gestion de la mémoire.

```
int *p = malloc(sizeof(int));
if (p != NULL) {
  *p = 42;
}
free(p);
p = NULL;

struct date *birthday = malloc(sizeof(struct date));
if (birthday != NULL) {
  do_something_with(birthday);
}
```

```c
free(birthday);
birthday = NULL;

char *str = calloc(255, sizeof(char));
if (str != NULL) {
  bla_blah(str);
}
free(str);
str = NULL;
```

## 2.4 Manipulation de chaînes de caractères

Les fonctions de manipulation de chaînes de caractères permettent d'utiliser des chaînes de caractères, c'est-à-dire avec le caractère `'\0'` final. Ce sont des fonctions qui peuvent être dangereuses si elles sont mal utilisées, elle sont à l'origine de nombreuses défaillances de programme. Attention quand vous manipulez des chaînes de caractères !

### 2.4.1 Longueur d'une chaîne de caractères

La fonction `strlen(3)` calcule la longueur de la chaîne de caractères `s`, sans compter le caractère `'\0'` final.

```c
#include <string.h>

size_t strlen(const char *s);
```

### 2.4.2 Copie d'une chaîne de caractères

La fonction `strncpy(3)` copie au plus `n` caractères de la chaîne pointée par `src` dans la chaîne pointée par `dest`. Attention, s'il n'y a pas de caractère `'\0'` dans les `n` premiers caractères de `src`, la chaîne dans `dest` ne sera pas terminée par `'\0'` (et donc ne sera pas une chaîne de caractères). En revanche, si la taille de `src` est strictement inférieur à `n` alors la fonction `strncpy(3)` écrit des caractères `'\0'` jusqu'à la fin du tampon `dest` pour s'assurer que `n` caractères ont été écrits.

```c
#include <string.h>

char *strncpy(char *dest, const char *src, size_t n);
```

### 2.4.3 Conversion d'une chaîne de caractères en entier

La fonction `atoi(3)` convertit le début de la chaîne pointée par `nptr` en entier de type `int`. Ceci est une manière usuelle de récupérer un paramètre entier dans un programme.

```c
#include <stdlib.h>

int atoi(const char *nptr);
```

Le programme suivant montre comment convertir une chaîne passée en paramètre d'un programme en entier.

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

int main(int argc, char *argv[]) {
  int n = 0;

  if(argc > 1) {
    n = atoi(argv[1]);
  }

  printf("n = %i\n", n);
  return 0;
}
```

## 2.5 Fonctions mathématiques

Les principales fonctions mathématiques sont définies dans l'en-tête standard `<math.h>` (attention, pas de s !).

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

Parmi les fonctions qui prennent un `double` et qui renvoie un `double`, on peut citer : `cos(3)`, `sin(3)`, `tan(3)`, `acos(3)`, `asin(3)`, `atan(3)`, `sqrt(3)`, `log(3)`, `log2(3)`, `log10(3)`, `exp(3)`, etc. On trouve aussi d'autres fonctions telles que `fmod(3)` ou `pow(3)` qui prennent deux arguments et dont le nom est assez explicite.

Les fonctions mathématiques sont définies dans la `libm`, il est donc nécessaire de lier le programme avec cette bibliothèque grâce à l'option `-lm` lors de la compilation.

# Chapitre 3

# La construction d'un programme

## 3.1 Découpage logique

### 3.1.1 Unité de compilation

Une unité de compilation est un ensemble cohérent de fonctions dont la définition est dans un fichier source `.c` et la déclaration (quand elle est nécessaire) est dans un fichier d'en-tête `.h`. Au moment de la conception, un exécutable est découpé de manière logique en plusieurs morceaux. Chaque morceau est alors implémenté dans une unité de compilation. Un exécutable est alors constitué de plusieurs unités de compilation. Une unité de compilation ne doit pas avoir plus d'une vingtaine de fonctions. Il faut toujours chercher à avoir des unités de compilation simples de manière à pouvoir retrouver une fonction facilement parmi toutes les unités.

Un fichier d'en-tête *(header)* est un fichier regroupant les déclarations (prototypes) des fonctions d'une unité de compilation ainsi que les types associés aux fonctions de l'unité. Pour rappel, une fonction ne peut être appelée que si la définition (corps) ou la déclaration (prototype) se trouvent avant dans le même fichier. Un fichier d'en-tête sert donc à déclarer les fonctions pour d'autres unités de compilation. Il a vocation à être inclus avec une directive `#include`.

Le fichier d'en-tête doit obligatoirement avoir une garde d'inclusion *(include guard)*. Le code suivant montre un exemple de garde d'inclusion pour un fichier `foo.h`.

```
#ifndef FOO_H
#define FOO_H

// content of foo.h

#endif // FOO_H
```

Quand il est inclus la première fois, la constante `FOO_H` n'est pas définie donc la première condition est vraie. Puis la constante `FOO_H` est définie et le contenu du fichier est inséré. À la fin du fichier, la condition se termine. Le nom de la

condition est généralement une variante du nom du fichier, parfois préfixé par le nom de la bibliothèque pour éviter les doublons. Si le fichier est inclus une seconde fois, la constante étant définie, la condition sera fausse et le contenu ne sera pas inclus.

D'une part, une garde d'inclusion permet d'éviter les inclusions multiples. En effet, un type ne peut pas être déclaré plusieurs fois, la garde d'inclusion évite donc les déclarations multiples en empêchant le contenu du fichier d'être inclus plusieurs fois. D'autre part, une garde d'inclusion permet d'éviter les boucles d'inclusion, c'est-à-dire quand un premier en-tête inclut un second en-tête qui lui même inclut le premier en-tête.

> **Recommandation #24** Il est obligatoire de mettre une garde d'inclusion dans les en-têtes.

Le fichier source associé à un en-tête inclut toujours en premier le fichier d'en-tête correspondant, avant tout autre inclusion. Cela permet de vérifier que le fichier d'en-tête se suffit à lui-même et qu'il inclut bien tout ce qu'il faut pour permettre la définition des types et des fonctions qu'il contient. Le code source `foo.c` suivant correspond au fichier d'en-tête précédemment défini.

```c
#include "foo.h"

// other includes

// definitions of functions
```

### 3.1.2 Les structures comme objets

Une bonne manière de procéder est de considérer une structure et l'ensemble de ses fonctions associées comme une unité de compilation. Plus précisément, les fonctions s'occupent de gérer la structure et notamment l'allocation mémoire interne à la structure. Chaque fonction est nommée en la préfixant par le nom de la structure et prend en premier paramètre un pointeur ou un pointeur constant sur la structure appelé `self`.

Parmi ces fonctions, une fonction suffixée `_create` sera chargée d'initialiser les structures internes, tandis qu'une fonction suffixée `_destroy` sera chargée de nettoyer les structures internes. De manière générale, les fonctions s'occupent uniquement de gérer les structures internes en s'assurant d'être dans un état cohérent, notamment sur la gestion de la mémoire. Elles ne s'occupent pas de gérer la structure en elle-même.

#### Exemple

Considérerons l'exemple suivant : une structure représentant une chaîne de caractères. Cet exemple permet de montrer différents aspects de manière pratique, il importe donc de généraliser les conseils données dans la suite. On définit tout d'abord la structure et on déclare les fonctions dans l'en-tête.

```c
#ifndef STR_H
#define STR_H
```

```c
#include <stddef.h>

// a string of characters
struct str {
  size_t size; // the size of the string
  char *data; // the data of the string
};

void str_create(struct str *self, const char *other);
void str_destroy(struct str *self);

size_t str_length(const struct str *self);
void str_append(struct str *self, const char *other);

#endif // STR_H
```

La structure s'appelle `struct str`, elle contient un champs `size` qui est la taille de la chaîne, ainsi qu'un champs `data` qui est le contenu de la chaîne. La fonction `str_create` prend un paramètre supplémentaire qui est la chaîne de caractères qui sert à intialiser à structure. Cette chaîne est copiée dans la structure. La fonction `str_destroy` libère la mémoire interne, c'est-à-dire celle du contenu de la chaîne. Puis, la fonction `str_length` renvoie la taille de la chaîne, on remarque que `self` est un pointeur constant parce qu'on n'a pas besoin de modifier le contenu de la structure pour calculer la longueur de la chaîne. Enfin, la fonction `str_append` ajoute le contenu d'une chaîne à la chaîne courante.

Le seul fichier inclus ici est `stddef.h` dont on a besoin pour la définition de `size_t`. Aucun autre fichier n'est nécessaire.

Le fichier source suivant montre l'implémentation de ces fonctions.

```c
#include "str.h"

#include <assert.h>
#include <stdlib.h>
#include <string.h>

void str_create(struct str *self, const char *other) {
  assert(self != NULL);

  if (other == NULL) {
    other = "";
  }

  self->size = strlen(other);
  self->data = calloc(self->size + 1, sizeof(char));
  strncpy(self->data, other, self->size + 1);
}

void str_destroy(struct str *self) {
  assert(self != NULL);
```

```
  free(self->data);
  self->data = NULL;
}

size_t str_length(const struct str *self) {
  assert(self != NULL);
  return self->size;
}

void str_append(struct str *self, const char *other) {
  assert(self != NULL);

  if (other == NULL) {
    other = "";
  }

  size_t size = self->size;
  size += strlen(other);
  char *data = calloc(size + 1, sizeof(char));
  strncpy(data, self->data, size + 1);
  strncpy(data + self->size, other,
      size - self->size + 1);
  free(self->data);
  self->data = data;
  self->size = size;
}
```

Le premier en-tête est bien celui correspondant au fichier source. Puis on inclut les en-têtes dont on a besoin pour l'implémentation.

Ensuite, la fonction `str_create` est définie. On commence par vérifier que l'utilisateur a passé un pointeur légitime à la fonction avec `assert`. La section 3.2.2 revient plus en détail sur l'utilisation d'`assert`. On calcule ensuite la taille de la chaîne `other` puis on alloue suffisamment de mémoire pour en faire une copie, en particulier on ajoute un octet pour le `'\0'` final, puis on copie le contenu de la chaîne `other` dans l'espace mémoire qu'on vient d'allouer. On dispose donc d'une copie de la chaîne passée en paramètre qu'on devra gérer nous-même. Si l'utilisateur facétieux décide d'appeler la fonction avec une chaîne nulle, on décide d'initaliser avec une chaîne vide de sorte que `self->data` n'est jamais égal à `NULL`. La fonction `str_destroy` libère simplement la mémoire utilisée.

La fonction `str_length` renvoie directement la taille stockée à l'intérieur de la structure. Il n'y a donc pas besoin d'appeler `strlen` à chaque fois. Il est donc nécessaire de maintenir le champ `size` cohérent avec la taille réelle du contenu de la chaîne.

La fonction `str_append` est un peu plus complexe. On calcule tout d'abord la taille de la chaîne finale qui est la taille de la chaîne courante, ajoutée à la taille de la chaîne à ajouter, plus un pour le caractère `'\0'` final. On alloue un espace mémoire de cette taille puis on copie successivement les deux chaînes : la chaîne courante, puis la chaîne à ajouter. On prend bien garde à copier au bon endroit, en particulier la seconde chaîne qui doit être copiée juste après

la première. Une fois ceci fait, on libère l'espace mémoire qui servait pour la chaîne courante puisqu'on n'en a plus besoin, et on lui affecte le nouvel espace mémoire. Enfin, on met à jour la taille de la nouvelle chaîne ainsi créée.

### Retour sur la création et la destruction

Les deux fonctions de création et de destruction ne s'occupent pas de l'allocation de la structure en elle-même. C'est à l'utilisateur d'allouer la structure et il peut ainsi l'allouer où bon lui semble.

Premièrement, la structure peut être allouée dans la pile avec une variable locale, c'est le cas le plus simple et celui qui est le plus recommandé. En effet, la libération de la mémoire est alors automatique, il n'y a donc rien à faire et aucune erreur possible. L'inconvénient est qu'il faut passer l'adresse de la structure à chaque appel.

```
struct str s1;
str_create(&s1, "stack");
str_destroy(&s1);
```

Deuxièmement, la structure peut être allouée dynamiquement avec `malloc` . L'utilisateur doit dans ce cas là prendre garde à bien libérer la mémoire en appelant `free` après avoir détruit la struture. Cette méthode est déconseillée pour les cas les plus simples. Elle est à réserver pour les cas où on n'a pas le choix, c'est-à-dire assez peu souvent en pratique.

```
struct str *s2 = malloc(sizeof(struct str));
str_create(s2, "heap");
str_destroy(s2);
free(s2);
```

Troisièmement, la structure peut faire partie d'un tableau. Dans ce cas, il faut calculer l'adresse de la structure pour la passer aux fonctions.

```
struct str s3[16];
str_create(&s3[12], "array");
str_destroy(&s3[12]);
```

Quatrièmement, la structure peut faire partie d'une autre structure englobante. Dans ce cas, cette structure englobante possède elle-même une fonction de création qui appelle la fonction de création de la structure. Même chose pour les fonctions de destruction.

```
struct other {
  struct str s;
};

void other_create(struct other *self) {
  str_create(&self->s, "struct");
}

void other_destroy(struct other *self) {
  str_destroy(&self->s);
}
```

On constate donc que ces fonctions de création et destruction sont très flexibles et peuvent être utilisées dans n'importe quel cas.

## 3.2 Bonnes pratiques

Cette section regroupe pêle-mêle quelques bonnes pratiques quand on code en C. La liste est loin d'être exhaustive mais permet de commencer à coder sans commettre les erreurs ou les maladresses les plus grossières.

### 3.2.1 Les conventions de nommage

Une convention de nommage est un schéma de nommage des différents objets qu'on rencontre dans un programme. Souvent, chaque langage possède une seule convention de nommage à laquelle tout le monde adhère. En C, il en existe plusieurs à cause principalement de l'âge du langage. Toutefois, une de ces conventions se retrouve dans beaucoup de programmes C et c'est celle qu'il est conseillé d'utiliser.

La convention de nommage principale du C exige de nommer les types, les fonctions et les variables en minuscule avec un tiret bas (`_`) pour séparer les mots (cette manière de nommer s'appelle *snake case*) ; les macros et constantes préprocesseur en majuscule avec un tiret bas (`_`) pour séparer les mots. Cette convention est notamment celle qui est (à peu près) utilisée dans la bibliothèque standard.

De plus, pour éviter les doublons de noms entre deux bibliothèques, tous les types, fonctions, macros et constantes sont généralement préfixé par le nom de la bibliothèque (ou un diminutif). Par exemple, dans la bibliothèque OpenGL, toutes les fonctions commencent par le préfixe `gl` (malheureusement, le nom des fonctions ne suit pas la convention de nommage principale) tandis que les constantes commencent par `GL_`.

Quelle que soit la convention de nommage, il convient de toujours nommer les objets de manière explicite. On écrit un code une seule fois mais on le relit de nombreuses fois, il est donc important de faciliter la lecture et donc de bien choisir les noms des objets. L'excuse «je changerai plus tard» n'est jamais valable parce que souvent, ce n'est qu'un prétexte et rien n'est jamais changé plus tard.

> **Recommandation #25** Les noms choisis pour les variables, fonctions, constantes doivent toujours être explicites.

Au delà des conventions de nommage, il existe des conventions de codage qui indique comment présenter un code. Pour cette partie, il n'y a aucune convention majoritaire et les projets codés en C utilisent souvent chacun leur propre variante de convention de codage. Il existe même des programmes qui permettent de formater un code source suivant divers critères de convention de codage. La seule recommendation qu'on peut donc apporter sur les conventions de codage, c'est d'être cohérent c'est-à-dire de toujours conserver la même convention tout au long d'un projet.

**Recommandation #26** Quand on contribue à un projet, on utilise toujours les conventions de codage et de nommage du projet.

### 3.2.2 La programmation défensive

La programmation défensive permet de prévenir des erreurs ou de les détecter le plus tôt possible. L'outil principal pour la programmation défensive consiste à placer des assertions dans le code partout où cela est possible.

Une assertion est une condition qui doit être vérifiée à un endroit donné du code. Si elle ne l'est pas, c'est généralement qu'il se passe quelque chose d'anormal, c'est-à-dire qu'il y a une erreur qu'il faut corriger, et le programme doit s'arrêter immédiatement. Une assertion doit être utilisée pour détecter les erreurs de programmation.

Pour utiliser des assertions en C, il faut tout d'abord inclure le fichier `assert.h`. Ensuite, on peut utiliser la macro `assert` qui prend en paramètre la condition à vérifier à cet endroit du code. Si la condition est fausse au moment où le programme passe à cet endroit, un message d'erreur indiquant la ligne du fichier et l'assertion fautive s'affiche et le programme s'arrête.

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

#### Quelles sont les conditions à vérifier ?

Voici quelques cas caractéristiques où ajouter une assertion est quasiment obligatoire.

- Les préconditions : une précondition est une condition qui doit être vérifiée dès le début d'une fonction. C'est à l'appelant de faire les vérifications nécessaires. Une assertion en début de fonction permet de garantir que la précondition est vérifiée.
- Les pointeurs : pour s'assurer que le pointeur auquel on accède est bien un pointeur valide, on peut ajouter une assertion et vérifier qu'il n'est pas égal à `NULL`. Attention cependant, il est parfois légitime d'avoir un pointeur à `NULL` et dans ces cas là, il faut le traiter normalement.
- Les énumérations de cas : on a parfois un certain nombre de cas à vérifier grâce à une série de conditions `if-else`. Dans la dernière branche, on peut ajouter une assertion pour vérifier qu'on a bien le dernier cas.
- Les invariants : certaines conditions doivent toujours être vérifiées à l'intérieur d'une fonction, on peut s'en assurer en ajoutant des assertions. Les préconditions sont une catégorie d'invariant.

Cette liste n'est évidemment pas exhaustive. On peut ajouter des assertions partout où cela est pertinent. Généralement, quand on arrive à un moment du codage d'une fonction et qu'on se dit «ici, ceci doit être vrai», alors c'est qu'il faut ajouter une assertion. Il ne faut pas hésiter à utiliser ce mécanisme autant que possible car il permet de détecter des erreurs de programmation le plus tôt possible et donc, de les corriger le plus rapidement possible.

Il est à noter qu'il est possible de désactiver toutes les assertions d'un seul coup en ajoutant l'option de compilation `-DNDEBUG`. Les assertions sont alors remplacées par des instructions vides et ne sont plus vérifiées.

### Quelques exemples

Le premier exemple montre une *précondition*. La fonction calcule la fonction de Collatz, qui n'est définie que pour les entiers strictement positifs. Dans ce cas, on s'assure que l'utilisateur de la fonction l'appelle correctement en ajoutant une précondition sur la valeur de `n`.

```
int collatz(int n) {
  assert(n > 0);

  if (n % 2 == 0) {
    return n / 2;
  }

  return 3 * n + 1;
}
```

Le second exemple montre une vérification de *pointeur*. La fonction échange deux valeurs d'entier. En C, comme on l'a vu, on doit passer par des pointeurs. Or, pour que l'échange puisse se faire correctement, il est nécessaire que les deux pointeurs soient valides. Dans ce cas, on utilise donc deux assertions pour vérifier chacun des deux pointeurs. Pourquoi deux ? Parce qu'il sera alors plus facile de voir quel est le pointeur fautif au moment où l'erreur surviendra.

```
void swap(int *a, int *b) {
  assert(a != NULL);
  assert(b != NULL);
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Le troisième exemple montre une *enumération de cas*. On utilise un entier pour représenter une couleur de carte à jouer (pique, cœur, carreau, trèfle). Quand on veut l'afficher, on examine la valeur de `suit`, et le dernier cas est nécessairement trèfle (`CLUB`), donc on l'indique avec une assertion. Si cette condition est fausse, la valeur de `suit` n'est pas parmi les quatre valeurs prédéfinies, c'est qu'il y a une erreur de programmation.

```
#define SPADE 1
#define HEART 2
#define DIAMOND 3
#define CLUB 4

void print_suit(int suit) {
  if (suit == SPADE) {
    printf("spade");
  } else if (suit == HEART) {
    printf("heart");
  } else if (suit == DIAMOND) {
    printf("diamond");
  } else {
```

```
    assert(suit == CLUB);
    printf("club");
  }
}
```

Le quatrième exemple montre un *invariant*. Plus particulièrement, on veut vérifier que l'indice du tableau qu'on utilise est bien dans l'intervalle voulu. Ce genre d'invariant est nécessaire quand on fait des calculs sur les indices d'un tableau, il permet de s'assurer que les calculs effectués sont corrects et/ou que l'indice existe bien. Dans ce cas précis, on fait l'échange de deux valeurs d'un tableau, on vérifie donc que les indices fournis sont bien valides.

```
void swap(int *data, size_t sz, size_t i, size_t j) {
  assert(0 <= i && i < sz);
  assert(0 <= j && j < sz);
  int tmp = data[i];
  data[i] = data[j];
  data[j] = tmp;
}
```

## 3.3 Outillage du programmeur

Les outils permettent de corriger les erreurs qu'on a détectées. Nous traitons ici des erreurs les plus fréquentes en C : les erreurs de mémoire. L'outil principal que nous allons utiliser s'appelle Valgrind et permet d'exécuter un programme et de vérifier la bonne utilisation de la mémoire.

### 3.3.1 Erreur de segmentation

L'erreur de segmentation *(segmentation fault, segfault)* est une erreur dans un programme qui tente d'accéder à une zone mémoire non-autorisée. Le plus souvent, cette erreur survient en utilisant un pointeur à `NULL`, soit en le déréférençant (via l'opérateur `*`), soit en accédant à un champs de la structure pointée (via l'opérateur `->`). En cas d'erreur de segmentation, le programme est immédiatement arrêté.

Soit le programme `segfault.c` suivant qui contient une erreur manifeste :

```
#include <stddef.h>

int main() {
  int *p = NULL;
  *p = 1;
  return 0;
}
```

À l'exécution, ce programme s'arrête et affiche :

```
$ ./segfault
Erreur de segmentation
```

Sans autre précision, il est difficile de savoir où se situe l'erreur. On peut alors utiliser le programme `valgrind` et recommencer.

```
$ valgrind ./segfault
...
==30832== Command: ./segfault
==30832==
==30832== Invalid write of size 4
==30832==    at 0x108530: main (segfault.c:5)
==30832==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
...
Erreur de segmentation
```

On constate alors que valgrind détecte l'erreur mais en plus indique le fichier et la ligne fautive (la ligne 5). On peut donc corriger directement l'erreur.

### 3.3.2 Fuite mémoire

Une fuite mémoire survient quand un espace mémoire allouée n'a pas été libéré avant la fin du programme. C'est une erreur de programmation. Ce type d'erreur est parfois difficile à déceler car un programme avec une fuite mémoire peut fonctionner tout à fait correctement. Cependant, il convient de corriger ce type d'erreur, car un programme qui fuit occupe des ressources du système qui pourrait être utile pour d'autres processus.

> **Recommandation #27** Un programme ne doit jamais avoir de fuite mémoire.

Par exemple, voici un programme avec une fuite mémoire :

```c
#include <stdlib.h>

int main() {
  int *p = malloc(sizeof(int));
  *p = 1;
  return 0;
}
```

Son exécution ne provoque pas d'erreur. Mais si on l'exécute avec Valgrind, voici le résultat :

```
$ valgrind --leak-check=full ./leak
...
==10969== HEAP SUMMARY:
==10969==     in use at exit: 4 bytes in 1 blocks
==10969==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==10969==
==10969== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==10969==    at 0x4C28C20: malloc (vg_replace_malloc.c:296)
==10969==    by 0x400517: main (leak.c:4)
==10969==
==10969== LEAK SUMMARY:
```

```
==10969==    definitely lost: 4 bytes in 1 blocks
==10969==    indirectly lost: 0 bytes in 0 blocks
==10969==      possibly lost: 0 bytes in 0 blocks
==10969==    still reachable: 0 bytes in 0 blocks
==10969==         suppressed: 0 bytes in 0 blocks
==10969==
==10969== For counts of detected and suppressed errors, rerun with: -v
==10969== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

On constate qu'on a perdu quatre octets dans un bloc : notre entier alloué. Pour corriger, il suffit d'ajouter un `free(p);` juste avant le `return` final. Une fois corrigé, on obtient le résultat suivant avec Valgrind :

```
$ valgrind --leak-check=full ./leak
...
==11008== HEAP SUMMARY:
==11008==     in use at exit: 0 bytes in 0 blocks
==11008==   total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==11008==
==11008== All heap blocks were freed -- no leaks are possible
==11008==
==11008== For counts of detected and suppressed errors, rerun with: -v
==11008== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Ce cas d'exemple est très simple. La plupart du temps, corriger une fuite mémoire est un exercice difficile, en particulier quand on n'a pas réfléchi à une méthode pour allouer et libérer la mémoire correctement. De manière générale, pour faciliter la gestion de la mémoire, il vaut mieux savoir où et quand sera libérée la mémoire allouée.

### 3.3.3 Utilisation après libération

L'erreur la plus courante, après la fuite mémoire, est sans doute l'utilisation de mémoire déjà libérée *(use after free)*. Voici un exemple :

```c
#include <stdlib.h>

int main() {
  int *p = malloc(sizeof(int));
  *p = 1;
  free(p);
  *p = 2;
  return 0;
}
```

Valgrind détecte ce genre d'erreur et les signale comme étant des lectures ou des écritures mémoires invalides :

```
$ valgrind --leak-check=full ./user_after_free
...
==11322== Invalid write of size 4
==11322==    at 0x400576: main (user_after_free.c:7)
==11322==  Address 0x51de040 is 0 bytes inside a block of size 4 free'd
```

```
==11322==    at 0x4C29E90: free (vg_replace_malloc.c:473)
==11322==    by 0x400571: main (user_after_free.c:6)
==11322==
==11322==
==11322== HEAP SUMMARY:
==11322==     in use at exit: 0 bytes in 0 blocks
==11322==   total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==11322==
==11322== All heap blocks were freed -- no leaks are possible
==11322==
==11322== For counts of detected and suppressed errors, rerun with: -v
==11322== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Il faut dans ce cas déplacer le `free`, ou faire une copie des données qu'on doit libérer et ne plus utiliser le pointeur. D'une manière générale, mettre à `NULL` un pointeur sur une zone mémoire qu'on vient de libérer est une bonne pratique qui permet de détecter les utilisations après libération encore plus rapidement (puisqu'alors, on a une erreur de segmentation).

> **Recommandation #28** Il faut toujours mettre à `NULL` un pointeur qu'on vient de libérer.

# Table des matières