Introduction au C pour la MP2I

Pierre Karpman

Version du 26 septembre 2025

1 Introduction

L'objectif (à terme) de ce document est d'être une ressource à peu près complète pour l'apprentissage du langage C en filière MP2I/MPI des classes préparatoires aux grandes écoles. On ne présuppose aucune connaissance antérieure avec le C, ni avec aucun autre langage de programmation. Le niveau de description adopté cherche à faire prendre un certain recul avec les mécanismes à l'œuvre dans l'écriture & l'exécution d'un programme C, tout en se limitant essentiellement au contenu du programme des CPGE. Certaines précisions plus techniques seront parfois écrites en petit, et peuvent alors être ignorées en première lecture.

2 Bases du développement

Cette section fait parfois référence à des notions décrites dans des sections ultérieures.

Le C est un langage de programmation *compilé* ; son utilisation s'effectue en deux temps : l'écriture du *code source* (ou juste *code*, ou juste *source* ; en anglais : *source code*, ou juste *source*, ou juste *code*) du programme, puis la *compilation* de celui-ci par un *compilateur* (en anglais : *compiler*) en un programme *exécutable*. Le rôle du compilateur est à peu près littéralement de traduire le code source du programme, écrit en C, en un code *machine* ou *binaire* (en anglais : *machine* ou *binary*) qui peut être exécuté par un ordinateur (généralement : le vôtre). Le développement en C nécessite donc essentiellement deux outils : un *éditeur de texte* permettant l'écriture (efficace) du code source, et un compilateur (ou plus généralement, un ensemble d'outils de compilation). L'ensemble de ces outils est parfois regroupé (ou présenté depuis) un seul gros logiciel, un *IDE* (pour *integrated development environment*).

§ Éditeurs de texte

Il existe *beaucoup* d'éditeurs de texte pour le développement logiciel, qui sont tous meilleurs que les autres. Un bon éditeur doit pouvoir fournir une *coloration syntaxique* décente (qui permet de mettre en évidence les mots-clefs & constructions syntaxiques du langage) ainsi qu'une aide à l'*indentation* (*cf.* ci-dessous). À part ça, la qualité essentielle d'un éditeur est qu'il vous plaise.

Certains éditeurs courants sont *vim*, *Emacs* (éditeurs « historiques » mais toujours d'actualité (surtout le premier)), *VS Code*, ou encore *Sublime Text* (payant), *nano* ou *notepad++*.

§ Compilation d'un programme C

La compilation d'un programme C consiste essentiellement à correctement invoquer un compilateur, avec les *options* souhaitées.

¶ **Compilateurs.** Les qualités essentielles d'un compilateur C sont de supporter (à peu près) la dernière version de la norme du langage, actuellement C23, et de produire du code correct & efficace. Les deux principaux compilateurs C sont gcc et clang, mais il en existe bien d'autres (dont certains comme CompCert visent des cas d'utilisation plutôt niche, en l'occurrence la compilation de logiciels critiques). Si vous souhaitez vous amuser avec plein de compilateurs, un bon point de départ est le compiler explorer.

¶ **Compilation & exécution.** La façon la plus élémentaire d'utiliser un compilateur est de l'invoquer depuis la *ligne de commande* (depuis un émulateur de terminal) pour compiler un programme dont l'intégralité du code source hors éventuelle utilisation de *fonctions de bibliothèque* se trouve dans un unique fichier. Traditionnellement le nom d'un fichier contenant du code source C se termine par l'extension `.c` (par exemple : `ohai.c`), et bien que suivre cette pratique ne soit pas *strictement nécessaire*, ne pas le faire serait certainement une mauvaise idée. En supposant qu'un compilateur peut s'invoquer avec la commande `cc`, compiler (par exemple) `ohai.c` n'est pas plus difficile qu'écrire :

Terminal 2.1

```
> cc ohai.c
```

depuis un terminal placé dans le même répertoire que le fichier `ohai.c`, où `>` désigne l'*invite de commande* (en anglais : *prompt*), c'est à dire un ensemble de caractères indiquant que le terminal est prêt à recevoir une entrée.

Si la compilation se déroule sans erreur, elle produit un fichier exécutable `a.out` dans le même répertoire, qui peut alors s'exécuter comme `> ./a.out` depuis le terminal. Si `ohai.c` contient le Programme 3.4, l'effet de l'exécution de `a.out` sera d'afficher le message `OHAI` (suivi d'un retour à la ligne) sur la *sortie standard* ; l'ensemble du processus de compilation et d'exécution ressemble alors à :

Terminal 2.2

```
> cc ohai.c
> ./a.out
OHAI
```

¶ **Options de compilation.** Les compilateurs (C ou autre) acceptent typiquement de nombreuses *options* de compilation permettant de piloter ou personnaliser la compilation. Nous ne décrirons pour l'instant qu'un très petit nombre d'entre elles, qui sont le plus immédiatement utiles en début d'apprentissage.

L'option `-o` permet de spécifier un autre nom pour l'exécutable que le nom par défaut `a.out`. La plupart du temps, on utilise simplement le nom du fichier principal privé de son extension `.c`, par exemple `> cc -o ohai ohai.c`. Faites toutefois attention à ne pas écraser votre source par erreur en faisant `> cc -o ohai.c ohai.c`, ce serait dommage (la décompilation n'est pas une science exacte, et ne pourra pas toujours vous sauver) !

Les options `-Wall` et `-Wextra` activent un certain nombre d'*avertissements* (en anglais : *warnings*). Ceux-ci sont émis par le compilateur lorsqu'il détecte une erreur probable (par exemple un fragment de code pouvant amener à un bug) ou une construction à la pertinence suspecte. Malgré leurs noms, il existe beaucoup de types d'avertissements qui ne sont *pas* activés par ces options ; avec *clang*, on pourra par exemple encore ajouter `-Wmost` et `-Wtype-limits`.

La famille d'option `-std=...` permet de spécifier une version de la norme C à utiliser pour la

compilation. Le sous-ensemble du langage au programme en MP2I/MPI est défini relativement à la « norme C99 ou plus récente », et l'on pourra donc utiliser cette option sous la forme `-std=c99`, `-std=c11`, `-std=c17` ou `-std=c23`. La version exacte de la norme choisie aura relativement peu d'impact dans le cadre des CPGE, mais mieux vaut en choisir une à la compilation. Cela étant dit, la norme C23 étant encore récente (elle n'est par exemple pas encore complètement implémentée par *clang*), mieux vaut éviter pour l'instant d'en exploiter des aspects trop spécifiques (même dans deux ans, rien ne vous garantit que vous disposerez d'un compilateur compatible lors des oraux de travaux pratiques). L'option `-std` peut être accompagnée de l'option `-pedantic`, qui aura pour effet d'émettre des avertissements lorsque votre programme dévie (trop) de la norme indiquée.

Lors de la compilation de programmes utilisant des tableaux (*cf.* Section 4.2), des pointeurs (Section 5) ou effectuant des allocations mémoire (Section 6), on conseille vivement de compiler avec l'option `-fsanitize=address`, dont l'effet est d'activer l'*address sanitizer*, un outil précieux pour diagnostiquer de nombreux bugs. Ces diagnostiques seront également plus précis si cette option est utilisée conjointement avec l'option `-g`, qui enrichit l'exécutable produit de symboles aidant à identifier les différents points du code source. Dans un contexte plus général, on peut également utiliser l'option `-fsanitize=undefined` afin d'aider la détection de certains comportement non définis par le langage.

¶ **`make`.** Il est rapidement fastidieux d'écrire à la main et à répétition la même commande de compilation, et cela devient encore plus pénible quand des options longues et compliquées deviennent nécessaires. Pour cette raison, les invocations du compilateur sont souvent déléguées à un outil d'aide à la compilation ou à un IDE. Un outil courant sous plateforme UNIX est le logiciel utilitaire `make`, qui à l'aide d'un fichier de description `Makefile` invoque automatiquement le compilateur de la façon appropriée.

On propose ci-dessous un `Makefile` personnalisable minimaliste permettant de compiler les programmes écrits dans un seul fichier source. On suppose ici que l'on se trouve dans un répertoire contenant deux fichiers source `ohai.c` et `bgcd.c`, décrivant chacun un programme indépendant ; il suffit alors, dans le même répertoire, de créer un fichier `Makefile` contenant :

Programme 2.3

```
CC=clang
CFLAGS= -Wall -Wextra -Wmost -Wtype-limits -std=c23 -pedantic \
        -fdiagnostics-color=always
EXECS= ohai bgcd

all: $(EXECS)

clean:
    rm $(EXECS)

ohai: ohai.c

bgcd: bgcd.c
```

Attention : suite à une conception légèrement archaïque, le caractère précédent `rm` (après la ligne `clean:`) doit être une *tabulation*, (touche `Tab`) et non une espace.
Une fois ce fichier créé, `make` ou `make all` tentera de compiler chacun des programmes listés après

`EXECS=` (mais s'arrêtera après la première erreur), avec le compilateur indiqué après `CC=` et les options indiquées après `CFLAGS=` ; `make ohai` ou `make bgcd` permettront de compiler spécifiquement le programme `ohai` ou `bgcd` respectivement, et `make clean` supprimera les exécutables résultant.

§ Conventions de style

Le langage C impose relativement peu de contraintes de *style d'écriture*, et c'est tant mieux. Cela ne veut pas pour autant dire qu'on peut faire n'importe quoi.

Il existe de nombreuses convention de style en C, dont certaines sont assez extrêmes. Dans le cadre des CPGE, il n'est pas nécessaire d'adhérer strictement à une convention particulière (et d'abord, laquelle ?), mais on s'efforcera de suivre un certain nombre de règles de bases :

- adoptez un style homogène au sein d'un même fichier (et plus généralement, au sein d'un ensemble de fichiers appartenant au même projet), d'un même devoir ;
- laissez faire l'outil d'indentation automatique de vote éditeur de texte s'il en possède un, et sinon installez en un ou utilisez un outil externe comme *clang-format* ;
- n'écrivez jamais plus d'une instruction par ligne ;
- ne déclarez pas plusieurs variables en une seule fois ;
- limitez la longueur des lignes, la taille des fonctions, le nombre d'imbrications ;
- essayez de bien nommer vos fonctions & variables ;
- aérez le code (mais pas trop) en insérant judicieusement des espaces ou sauts à la ligne.

Les mêmes consignes s'appliquent pour l'écriture de code sur papier, si ce n'est que vous ne bénéficierez évidemment plus d'outil d'indentation.

Si vous souhaitez ne *pas* respecter ces règles, vous trouverez peut-être un exécutoire en l'objet de l'international obfuscated C code contest, mais pensez bien que dans un contexte scolaire ou professionnel, toute patience a ses limites ; bien qu'on soit en droit de trouver que le code source suivant est « rigolo », on espère qu'on conviendra aisément qu'il est également ~~plus~~ moins lisible que le Programme 3.4 qu'on peut d'ailleurs presque retrouver en lui appliquant *clang-format* avec le style *WebKit*.

Programme 2.4

```c
#include  <stdio.h>
#include <stdlib.h>
int main(void){puts
(  "O""H""A""I"  );
return EXIT_SUCCESS
                ;}
```

§ Documentation

Les programmes écrits dans le cadre des CPGE sont relativement courts, et ne nécessitent pas *a priori* d'être accompagnés d'une documentation abondante ; on se contentera généralement d'une documentation intégrée au code source, à l'aide de commentaires et d'`assert`. Un bon niveau par défaut (à éventuellement moduler en fonction des besoins) et d'*a minima* décrire le rôle de chaque fonction par un commentaire, valider ses éventuelles préconditions & postconditions par des `assert`, et faire de même pour chaque portion de code (par exemple une grosse boucle) ou invariant significatif. Il n'est cependant pas nécessaire (et même contre-productif) d'inclure des commentaires tautologiques semblables à `int a = 1; // déclaration de la variable a de type int, initialisée à 1`.

§ **Tests**

Tout processus de développement sérieux (et le contexte scolaire des CPGE n'y fait pas exception) nécessite l'écriture de tests, dans le but d'aider à valider l'absence de bugs et la correction des programmes. Nous aborderons le sujet plus en détail dans la Section 11, et pour l'heure nous contentons de formuler quelques conseils élémentaires :

— le code doit être testé : l'écriture de test fait intégralement partie du processus de développement, et il est *normal* d'y consacrer du temps. À titre d'exemple, l'infrastructure de test du moteur SQL *SQLite* utilise 590 fois plus de ligne de codes que le moteur lui-même ;
— les tests doivent être écrits et exécutés au fur et à mesure de l'avancement du développement : cela « dilue » le travail, et surtout permet de détecter les éventuelles erreurs au plus vite ;
— les tests doivent avoir une bonne *couverture* des différents cas possible (par exemple, pour une fonction renvoyant `true` ou `false` sous certaines conditions, il faut *a minima* vérifier qu'elle renvoie `true` quand c'est attendu, mais *aussi* `false` quand ça l'est…).

3 Bases du langage

Dans toute cette partie, on suppose que les programmes sont définis dans un unique fichier, hors éventuelle utilisation de fonctions de la bibliothèque C standard.

3.1 Structure d'un programme C

§ **Fonction `main`.**

Un programme C comporte toujours un unique point d'entrée, sous la forme d'une *fonction* spéciale appelée `main`. Lorsqu'on exécute un programme, c'est cette fonction qui est *appelée*.

Un programme peut être lancé avec ou sans *arguments*. Les éventuels arguments sont transmis à la fonction `main`, qui peut cependant choisir de les ignorer ; c'est ce que nous ferons dans un premier temps.

La fonction `main` est aussi (en général) le point de sortie du programme : lorsque `main` *termine*, l'exécution du programme termine également. Il se peut cependant que le programme termine pour d'autres raisons : le plus souvent à cause d'une erreur irrattrapable lors de l'exécution, mais aussi parce qu'une fonction de sortie du programme a été appelée (*cf.* Section 3.13) ; nous ignorerons pour l'instant cette dernière possibilité (et il n'y a malheureusement pas grand chose à faire en ce qui concerne la première, à part écrire des programmes sans bugs).

Lorsqu'elle termine son exécution, la fonction `main` *renvoie* une valeur qui indique si cette exécution s'est bien déroulée ou non. Cette valeur est *un entier de type* `int` valant conventionnellement `0` pour indiquer que l'exécution s'est bien déroulée, et une valeur différente de `0` dans le cas contraire. Ces valeurs sont également définies par les symboles `EXIT_SUCCESS` et `EXIT_FAILURE`, qu'on s'astreindra à utiliser pour plus de clarté et de *portabilité* (c'est à dire, pour qu'un programme se comporte de la même manière quand compilé dans des environnements (compilateur, système d'exploitation, architecture matérielle…) différents).

Lorsque la fonction `main` ignore les éventuels arguments du programme (ce qui est pour l'instant ce que l'on souhaite faire), sa *signature* est celle d'une fonction ne prenant aucun paramètre, et renvoyant un entier de type `int`. Ceci s'écrit `int main(void)`, en suivant la syntaxe :

Syntaxe 3.1

```
nom_du_type_de_renvoi nom_de_la_fonction(paramètres de la fonction)
```

`void` est un type particulier, qui ici indique que la fonction ne prend pas de paramètres (on détaillera plus tard le cas plus général de fonctions prenant un nombre non nul de paramètres).

Un programme comporte généralement d'autres fonctions que `main` (qui doivent alors être définies), mais ce n'est pas *nécessaire*.

¶ **Directive `#include`.** Un programme C utilise souvent des objets qui ont déjà été définis *ailleurs*, typiquement dans la bibliothèque C standard, ou d'autres bibliothèques logicielles. Pour pouvoir utiliser ces objets, il est nécessaire que le programme en connaisse l'existence ; cette information est en général communiquée en utilisant la *directive* `#include`, qui sert à recopier des *déclarations* ou des *définitions* se trouvant dans d'autres fichiers. Par exemple, les symboles `EXIT_SUCCESS` et `EXIT_FAILURE` sont définis dans le fichier `stdlib.h` : le fichier d'*en-têtes* (en anglais : *headers*) de la bibliothèque C standard (en anglais : *standard C library*). Un programme C commence alors usuellement par un certain nombre de directives `#include`, une pour chacun des fichiers nécessaires, en procédant ainsi :

Syntaxe 3.2

```
#include <nom_du_fichier>
#include "nom_du_fichier"
```

On ne peut écrire qu'une seule directive par ligne, sans aucune espace après le début de la ligne et avant le début de la directive. Les deux syntaxes `<nom_du_fichier>` et `"nom_du_fichier"` correspondent à des familles de *chemins* différentes dans lesquelles le compilateur tente de trouver un fichier s'appelant `nom_du_fichier`. Pour l'instant, nous n'avons pas de raison d'utiliser la seconde.

Pour l'écriture de code papier dans le cadre des CPGE, on supposera toujours que les fichiers d'en-tête `assert.h`, `stdbool.h`, `stddef.h`, `stdint.h`, `stdio.h`, `stdlib.h` et —sauf mention du contraire— `string.h` ont été inclus, et il n'est donc pas nécessaire d'écrire les directives `#include` correspondantes.

¶ **Commentaires.** Il est rare qu'un programme soit immédiatement compréhensible à la lecture du seul code source. Il est donc capital de *documenter* celui-ci à l'aide de *commentaires*. Il existe deux façons usuelles d'introduire des commentaires en C :

— les commentaires uni-lignes introduits par `//` : tout ce qui suit ces deux symboles jusqu'à la fin de la même ligne est ignoré à la compilation ;
— les commentaires multi-lignes : tout ce qui se trouve entre la paire de symboles `/*` et la paire de symboles `*/` est ignoré à la compilation, mais on ne peut pas imbriquer deux commentaires multi-lignes.

§ **Far niente**

Nous pouvons maintenant écrire un programme C complet qui ne fait rien : il suffit que celui-ci soit constitué d'une unique fonction `main` qui termine immédiatement son exécution (avec succès, tant qu'à faire...). Ceci s'écrit en faisant suivre l'écriture de la signature de `main` par la suite de ses *instructions*, entre une paire d'accolades « `{` » et « `}` ». L'instruction permettant à une fonction de terminer son exécution

utilise le mot-clef `return`, optionnellement suivi (sans parenthèses nécessaires !) d'une *expression* dont la valeur sera celle renvoyée par la fonction, et enfin d'un point-virgule « ; » dénotant la fin de l'instruction.

Finalement, on obtient :

Programme 3.3

```c
// ne fait rien, avec succès
#include <stdlib.h>

int main(void)
{
    return EXIT_SUCCESS;
}
```

3.2 Types

Le C est un langage *typé* : les éléments manipulés ont un *type*, qui détermine ce qu'il est possible ou non de faire avec. Par exemple, il est possible d'ajouter deux éléments de type entier entre eux mais pas deux fonctions, ou un entier et une fonction. Le typage en C est *statique* : les fonctions & objets possèdent *un* type déterminé à l'écriture du programme ; il est cependant *faible* : il se *peut* qu'un objet d'un certain type soit convertit en un objet d'un autre type *compatible*, parfois au prix d'une perte d'information.

3.3 Sorties élémentaires

Un programme qui n'interagit pas avec son environnement est ultimement peu utile ; l'une des interactions les plus simples est l'affichage ou la lecture de texte, par exemple dans ou depuis un *terminal*. Ceci peut être fait en utilisant les fonctions d'entrées/sorties standard C (en anglais : *standard input/output*), qui sont déclarées dans le fichier d'en-tête `stdio.h`. Nous ne nous intéresserons pour l'instant qu'aux *sorties* les plus simples, c'est à dire… l'affichage de texte dans un terminal. On décrit quatre fonctions (essentiellement deux) permettant de faire cela : `puts`, `fputs`, `printf` et `fprintf`.

¶ **Fonction `puts`.** Comme son nom l'indique, la fonction `puts` affiche une *chaîne de caractères* (en anglais : *(character) string*) sur la *sortie standard* `stdout`, suivie d'un retour à la ligne. Une chaîne de caractère s'écrit en C comme du texte entre « *double quotes* », c'est à dire le symbole « " ». Le texte peut être vide, donnant la chaîne vide `""`, et éventuellement contenir des caractères jouant un rôle particulier. Si l'on écrit plusieurs chaînes de caractère à la suite (comme `"O" "H" "A" "I"`), celles-ci sont concaténées.

L'*appel* de la fonction `puts` se fait simplement (comme pour n'importe quelle fonction à un seul paramètre) en écrivant le nom de la fonction, suivi de son argument entre parenthèses. On peut ensuite placer un tel appel dans une instruction, en ajoutant un point-virgule.

Par exemple, le programme ci-dessous affiche le texte `OHAI` suivi d'un retour à la ligne sur la sortie standard ; lorsque le programme est lancé depuis un terminal, la sortie standard correspond simplement au terminal lui-même.

Programme 3.4

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

```c
#include <stdio.h>

int main(void)
{
    puts("OHAI");

    return EXIT_SUCCESS;
}
```

¶ **Fonction `fputs`.** La fonction `fputs` est identique à `puts`, à part deux différences :

— Elle prend deux paramètres plutôt qu'un seul, et le second désigne un *flux de sortie* (en anglais : *output stream*). Ceci permet d'afficher le texte sur une autre sortie que la sortie standard, notamment la sortie standard d'*erreur* `stderr`. Nous verrons également plus tard comment construire un flux correspondant à un fichier.
— Elle ne rajoute pas elle-même de retour à la ligne. Si désiré, ce dernier doit être ajouté « à la main » avec le caractère spécial « `\n` ».

L'appel d'une fonction à plusieurs paramètres se fait comme pour une fonction à un seul, en séparant les arguments fournis par des virgules.

Le programme ci-dessous utilise `fputs` pour afficher le texte `OHAI` suivi d'un retour à la ligne, à la fois sur la sortie standard et sur la sortie d'erreur standard. Par défaut, lorsque le programme est lancé depuis un terminal, cette dernière affiche également le texte sur le terminal. Les deux flux sont néanmoins distincts, et sous un système UNIX peuvent par exemple être *redirigés* séparément : `stdout` avec `>`, et `stderr` avec `2>`. Comme son nom l'indique, la sortie d'erreur standard sert principalement à afficher des messages d'erreur, et c'est une bonne pratique de l'utiliser pour cela.

Programme 3.5

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

int main(void)
{
    fputs("OHAI\n", stdout);
    fputs("OHAI\n", stderr);

    return EXIT_SUCCESS;
}
```

¶ **Fonction `printf`.** Comme son nom l'indique, la fonction `printf` permet d'afficher du texte *formaté* sur la sortie standard. Contrairement à `puts`, `printf` peut prendre plusieurs arguments, *en nombre quelconque* (on parle techniquement de *fonction variadique*, et c'est avec ses comparses probablement la seule fonction de ce type que vous serez amenés à manipuler couramment).

Le premier argument de `printf` est une chaîne de caractères qui peut contenir zéro à plusieurs (en anglais) *conversion specifications* qui ne seront pas affichées littéralement, mais auxquelles on substituera

une chaîne de caractères représentant une certaine valeur.

Si aucune « spécification de conversion » n'est fournie, `printf` est presque équivalente à `puts`, à la différence que comme `fputs` elle ne rajoute pas elle-même de retour à la ligne.

Pour chaque spécification de conversion, indiquée dans la chaîne de caractères qui est son premier argument, `printf` doit prendre un argument supplémentaire dont le *type* est compatible avec la spécification de conversion ; ces arguments doivent être fournis dans le même ordre que les spécifications de conversion.

Nous ne donnons pour l'instant que deux exemples :

— spécification de conversion `%s` : l'argument supplémentaire doit être une chaîne de caractères ;
— spécification de conversion `%d` : l'argument supplémentaire doit être un entier de type `int`.

Le programme suivant utilise `printf` pour afficher le texte `OHAI 1` suivi d'un retour à la ligne.

Programme 3.6

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

int main(void)
{
    printf("%s %d\n", "OHAI", 1);

    return EXIT_SUCCESS;
}
```

¶ **Fonction `fprintf`.** Comme son nom l'indique, la fonction `fprintf` est à `printf` ce que `fputs` est à `puts`. Pour que les choses ne soient pas trop simples et afin de limiter la ressemblance, le paramètre supplémentaire indiquant le flux de sortie est maintenant le *premier*.

Le programme suivant est fonctionnellement identique en tous points au précédent, mais utilise `fprintf`.

Programme 3.7

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

int main(void)
{
    fprintf(stdout, "%s %d\n", "OHAI", 1);

    return EXIT_SUCCESS;
}
```

¶ **Affichage et tampon.** La gestion interne (au système d'exploitation) des mécanismes d'affichage fait qu'un message non immédiatement suivi d'un retour à la ligne peut *temporairement* ne pas être affiché sur sa sortie. Certains émulateurs de terminal peuvent aussi mal réagir si le dernier message affiché avant

la terminaison d'un programme n'est pas non plus suivi d'un retour à la ligne. Il est donc préférable de toujours terminer ses messages (ou groupes de messages) par un tel caractère ou si l'on sait ce qu'on fait & ce qu'on veut, utiliser `fflush`.

3.4 Définition de fonction

Tout langage de programmation un tant soit peu avancé possède une notion de *fonction*, mais celle-ci peut désigner des choses bien différentes. Dans le cas des langages à dominance « impérative » comme le C, une fonction peut se voir comme un fragment de code définissant une action précise à effectuer ; qui le sera lors de l'*appel* de la fonction.

¶ **Principe d'une fonction C.** Comme nous l'avons vu plus haut, une fonction C peut prendre zéro, un, … plusieurs *paramètres*. Pour pouvoir appeler une fonction à n paramètres, il faut fournir autant d'*arguments* de type correspondant, dans le même ordre que les paramètres. Par exemple, si une fonction `fun` prend deux paramètres, un premier de type t_a et un second de type t_b, un appel à `fun` devra être de la forme `fun(a, b)` avec `a` un argument de type t_a et `b` un argument de type t_b.

Le fait qu'une fonction puisse prendre des paramètres permet de faire dépendre l'action effectuée par celle-ci des arguments fournis à l'appel ; même si cela n'est pas nécessaire (comme nous l'avons vu pour `main`, par exemple), cela correspond à la grande majorité des cas.

¶ **Utilité des fonctions.** Hors exemples élémentaires, un programme C bien écrit définira nécessairement un certain nombre de fonctions qui lui seront propre, chaque fonction servant un but bien précis aidant à la réalisation du but global du programme. Ceci n'est pas une contrainte technique car (comme on l'a déjà dit) seule l'*unique fonction* `main` est réellement nécessaire à tout programme C, mais utiliser plusieurs fonctions contribuera immensément à améliorer la *lisibilité* du programme (qui est une qualité essentielle) ainsi que sa capacité à être modifié ou réutilisé plus tard.

Si l'on peut donc raisonnablement dire que tout « bon » programme C non élémentaire utilise plusieurs fonctions, la réciproque n'est (sans grande surprise) malheureusement pas vraie. Notamment, choisir un « bon » ensemble de fonctions pour « découper » les tâches à réaliser dans un programme fait partie intégrale du processus de développement, et n'est pas toujours une étape facile.

§ **Syntaxe de déclaration & définition**

Pour beaucoup d'éléments du langage (dont les fonctions), le langage C distingue la notion de *déclaration* (« déclarer qu'une chose existe ») de celle de définition (« décrire ce en quoi elle consiste »). Dans le cas des fonctions, il est possible de déclarer une même fonction (identifiée par son nom) plusieurs fois, mais elle ne peut être définie qu'à un unique point dans tout le programme.

Comme nous l'avons déjà (partiellement) vu, la syntaxe (usuelle) de la signature d'une fonction en C est :

Syntaxe 3.8

```
nom_du_type_de_renvoi nom_de_la_fonction(paramètres de la fonction)
```

où `paramètres de la fonction` consiste en :

— `void` si la fonction prend zéro paramètres ;

— une liste `t1 p1, t2 p2, ..., tn pn` pour une fonction à n paramètres, avec `t1`, `t2`… les noms des types des paramètres 1, 2, …, et `p1`, `p2`… les noms des *variables* sous lesquels les arguments fournis à l'appel seront connus dans le *corps* de la fonction.

Soit `sig` une telle signature d'une fonction `fun`, on peut alors déclarer `fun` en faisant simplement suivre `sig` d'un « `;` » et dans ce cas, fournir les *noms* des paramètres est en fait facultatif ; cette déclaration d'une signature (éventuellement allégée) est un *prototype* de la fonction, mais on n'emploiera ce terme que rarement dans ces notes de cours. (ceci peut être fait plusieurs fois), et la définir en faisant suivre `sig` de son *corps*, c'est à dire des instructions spécifiant ce que fait la fonction, placées entre « `{` » et « `}` » (ceci ne peut être fait qu'une seule fois).

Comme nous l'avons déjà évoqué plus haut et détaillerons un peu plus ci-dessous, l'intérêt d'une déclaration est de rendre *visible* le nom d'une fonction « avant » qu'elle ne soit définie : ainsi, pour pouvoir appeler une fonction en une ligne du programme, il faut que celle-ci ait été :

— **définie** *en amont* (« en des lignes précédentes »), ou ;
— **déclarée** en amont, et **définie** quelque part ailleurs dans le fichier (en anglais, on parle de *forward declaration*) ou bien dans la bibliothèque C standard.

Dans le cas d'un programme constitué d'un unique fichier (hors utilisation de fonctions de bibliothèque), la seconde possibilité n'est que modérément utile puisqu' à l'exception de fonctions *mutuellement récursives*, très peu idiomatiques en C il est toujours possible (et souvent plus clair) de définir une fonction avant sa première utilisation. En revanche, elle joue un rôle important dans l'organisation d'un programme en plusieurs fichiers, et (comme brièvement déjà évoqué) les fichiers d'en-tête sont typiquement constitués de déclarations de fonctions (définies *ailleurs*) afin de permettre leur utilisation.

Le programme suivant déclare une fonction `print_greeting` prenant un paramètre `n` de type entier `int` et ne renvoyant rien : son type de renvoie est `void` ; appelle cette fonction depuis la fonction `main` avec l'argument `12` ; définit la fonction `print_greeting` comme affichant sur la sortie standard le message `OHAI n` suivi d'un retour à la ligne, où `n` est substitué par l'argument fourni à la fonction lors de son appel.

Programme 3.9

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

void print_greeting(int n);

int main(void)
{
    print_greeting(12);

    return EXIT_SUCCESS;
}

void print_greeting(int n)
{
    printf("OHAI %d\n", n);
}
```


À l'exécution, ce programme affiche `OHAI 12` sur la sortie standard, suivi d'un retour à la ligne.

§ **Flot d'exécution & valeur de retour**

Les fonctions ne faisant pas d'appel à `setjmp` ont un unique *point d'entrée* : l'exécution commence toujours par la première instruction de leurs corps. En revanche, elles peuvent avoir plusieurs *points de sortie* : chaque instruction de saut `return` a pour effet de terminer l'exécution de la fonction.

L'exécution termine également immédiatement après avoir exécuté la dernière instruction du corps d'une fonction, et dans ce cas ne renvoie rien. Autrement dit, tout corps de fonction est implicitement terminé par une instruction `return;`. Ceci n'est pas un problème si la fonction a type de retour `void` (qui veut exactement dire qu'elle ne renvoie rien), mais peut dans le cas contraire entraîner l'un des très nombreux *comportements non définis* (en anglais : *undefined behaviour*, ou « UB ») du langage si l'on cherche à en utiliser la valeur de retour, *cf.* MSC37-C. Ensure that control never reaches the end of a non-void function. De façon générale, on ne doit pas utiliser d'instruction `return` non suivie d'une expression dans une fonction dont le type de retour n'est pas `void`, et à l'inverse on ne doit pas utiliser d'instruction `return` suivie d'une expression dans une fonction dont le type de retour est `void`.

3.5 Variables

Les variables jouent un rôle central dans tout langage de programmation un tant soit peu avancé. Dans un langage impératif, elles forment une interface essentielle entre le code (tel qu'on l'écrit) et l'état d'un programme (lors de son exécution).

§ **Définition de variable**

En C une *variable* est un identifiant pour un *objet*. Informellement, un *objet* est une valeur numérique « brute » existant pendant toute ou partie de l'exécution d'un programme.

Toute variable possède un *type*, qui détermine comment *interpréter* la valeur de l'objet qu'elle *référence* lors de l'exécution. Ce type est décidé *statiquement* à la déclaration de la variable (et non pas par exemple *dynamiquement* à l'exécution du programme).

Tout comme une fonction, pour pouvoir être utilisée en un point d'un programme une variable doit avoir été déclarée en amont. Cette déclaration peut notamment avoir été faite par :

— la signature d'une fonction, pour les variables correspondant à ses paramètres ;
— une structure de boucle `for` (*cf.* Section 3.9) ;
— une déclaration explicite (*cf.* ci-dessous).

La syntaxe de déclaration d'une variable est similaire à celle d'une fonction ; elle consiste en :

Syntaxe 3.10

```
nom_du_type nom_de_la_variable;
```

Tout comme pour une fonction et *sauf cas particulier* (que nous évoquerons plus bas), cette déclaration *ne définit pas* la variable ; c'est à dire qu'elle ne définit pas de valeur initiale pour l'objet référencé par celle-ci. En C, lire une variable qui n'a pas été définie est un comportement non défini (en clair, c'est interdit), *cf.* EXP33-C. Do not read uninitialized memory. Une bonne pratique de programmation (qu'on s'astreindra à suivre pendant toute la scolarité en CPGE) consiste à ne *jamais seulement déclarer une variable*, mais à toujours la déclarer avec un « initialiseur » (en anglais : *initializer*) qui lui définit une

valeur initiale. À la rigueur, on pourra se permettre d'enfreindre cette règle si l'affectation initiale (*cf.* ci-dessous) de la variable est visible depuis sa déclaration. La syntaxe pour ceci est alors :

Syntaxe 3.11

```
nom_du_type nom_de_la_variable = initialiseur;
```

où `initialiseur` est une *expression* (*cf.* Section 3.8) du même type que la variable.

On peut déclarer une variable à n'importe quel point d'un programme, mais l'emplacement de la déclaration influencera sa *portée* et donc sa *visibilité* (*cf.* Section 3.10). On ne peut pas « redéclarer » une variable de même nom qu'une variable existante de même visibilité et de même portée.

Le programme ci-dessous utilise une variable `tw` de type entier `int` déclarée explicitement et initialisée à 12, pour afficher le message `OHAI 12 HAI 12` sur la sortie standard, suivi d'un retour à la ligne.

Programme 3.12

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

int main(void)
{
    int tw = 12;

    printf("OHAI %d ", tw);
    printf("HAI %d\n", tw);

    return EXIT_SUCCESS;
}
```

§ **Utilisation**

On peut essentiellement utiliser des variables de deux façons : en *lecture* (typiquement dans une expression) ou en *écriture*.

Comme leurs noms l'indiquent, une lecture consulte la valeur de l'objet référencé par une variable, et une écriture modifie cette dernière. L'écriture d'une variable se fait *via* une *instruction d'affectation*, dont la syntaxe est :

Syntaxe 3.13

```
nom_de_la_variable = valeur;
```

où `valeur` est une expression du même type que la variable.

Le programme ci-dessous est semblable au Programme 3.12 si ce n'est qu'il n'initialise pas la variable `tw` à la déclaration, mais lui affecte une valeur immédiatement après. Bien qu'une initialisation à la déclaration soit ici possible et préférable, le comportement du programme est clair et ce style d'écriture n'est pas vraiment problématique.

Programme 3.14

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

int main(void)
{
    int tw;
    tw = 12;

    printf("OHAI %d ", tw);
    printf("HAI %d\n", tw);

    return EXIT_SUCCESS;
}
```

De façon générale, tout programme C non complètement élémentaire utilise des variables. Vous n'êtes pas *vraiment* limités dans le nombre de variables que vous pouvez déclarer, et une bonne heuristique est que si vous écrivez la même « chose » (qui n'est pas juste un nom de variable ou de fonction) plus de deux fois, vous pouvez probablement avantageusement remplacer cette « chose » par une variable dont la valeur lui correspond.

Il est également souvent possible & utile d'introduire des variables dites *intermédiaires* pour simplifier des fragments de code complexe.

En revanche il faut prendre garde à ne pas inutilement dupliquer les variables ; avant d'introduire une nouvelle variable, il est sain de se demander si son rôle ne peut pas être rempli par une variable existante.

Trouver de bons noms de variables (et dans une moindre mesure, de bons noms de fonctions) est malheureusement une tâche difficile, qu'il vous faudra surmonter avec adresse. À ce jeu il ne faut pas chercher à être original ; un « bon » nom de variable est typiquement assez court pensez bien que même en tapant vite au clavier ou en utilisant un outil d'autocomplétion, `tableau_des_plus_courtes_distances_entre_paires_de_sommets` est un peu pénible à écrire… Et puis la norme C ne garantit pas qu'un nom de plus de 31 ou 63 caractères sera pris en compte, *cf.* DCL23-C. Guarantee that mutually visible identifiers are unique ; pas trop abrégé vous souviendrez-vous facilement dans un mois que `herd` signifie « réordonnancement heuristique », en particulier si vous n'avez pas documenté votre programme ? ; descriptif si une variable s'appelle `tableau`, mieux vaut qu'elle en soit un ; s'il y a plusieurs tableaux de rôles différents, mieux vaut préciser ces derniers dans le nom plutôt que d'avoir des `tableau1`, `tableau2` etc. ; consensuel inspirez vous par exemple des noms que vous rencontrerez dans la documentation de la bibliothèque standard.

Il est tout à fait possible de réutiliser un même nom de variable en différents points du programme, si le contexte est similaire. Par exemple, si deux fonctions `fun1` et `fun2` distinctes prennent toutes-deux un paramètre représentant la taille d'un certain objet, autant simplement nommer ce paramètre `size` dans les deux cas et il serait contre-productif de les appeler respectivement `size_in_fun1` et `size_in_fun2`, par exemple… Nous pourrons justifier cela un peu plus quand nous aurons défini les règles de portée des variables.

En C moderne, il n'y a pas de contraintes sur les points d'un programme où une variable peut être déclarée ; cela ne veut cependant pas dire qu'il faut faire n'importe quoi, et tout comme les choix des noms, les choix des points de déclarations influencent la lisibilité d'un code. *Une* approche possible est de déclarer les variables au plus proche de leur première utilisation, tout en regroupant les déclarations

de variables logiquement liées entre elles.

§ **Qualification `const`**

Lors d'une déclaration de variable, on peut enrichir le type d'un « qualifieur » (en anglais : *qualifier*) qui précise ou restreint l'usage de la variable. Nous n'évoquerons (pour l'instant) que le qualifieur `const`, dont l'effet est de rendre illégale toute modification. En clair, la variable est alors déclarée en « lecture seule » (en anglais : *read only*).

La syntaxe d'utilisation de ce qualifieur (ou d'un autre) est simple : il suffit de faire précéder le nom du type de la variable du mot-clef `const`. On illustre ceci avec le programme suivant, identique au Programme 3.12 si ce n'est que la variable `tw` est « (qualifiée) `const` » (en anglais : `const`(*-qualified*)).

Programme 3.15

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

int main(void)
{
    const int tw = 12;

    printf("OHAI %d ", tw);
    printf("HAI %d\n", tw);

    return EXIT_SUCCESS;
}
```

On peut remarquer que puisqu'une variable `const` ne peut pas être modifiée, il devient encore plus important de l'initialiser à sa déclaration ; en effet seule une utilisation en lecture serait *a priori* légale, et toute lecture d'une variable non initialisée mène à un comportement non défini. S'il n'est donc pas *illégal* de déclarer une variable `const` non initialisée, cela est peu utile…

Utiliser un qualifieur `const` dès que possible (c'est à dire dès qu'une variable n'a pas besoin ou ne doit pas être modifiée) peut être une bonne pratique. Les avantages escomptés sont une amélioration de la lisibilité du code (on sait dès la déclaration que la variable ne sera pas modifiée), une diminution du risque de bugs (tenter de modifier une variable `const` (par étourderie) déclenche une erreur à la compilation, et permet donc de détecter l'erreur très tôt), et éventuellement une amélioration des performances du code produit (le compilateur peut par exemple remplacer chaque utilisation de la variable par sa définition (en anglais, on parle d'*inlining*)) bien qu'à un niveau d'optimisation suffisant, il est probable que cela ne fasse pas une grosse différence ..

Attention : bien qu'on puisse par abus de langage parler de « constantes » pour les variables `const`, celles-ci sont néanmoins distinctes des vraies constantes, notamment des *constantes littérales* dont nous parlerons un peu plus tard. Si la précision est importante (quand ne l'est-elle pas ?), mieux vaut employer le terme de variable (qualifiée) `const`.

3.6 Types entiers

Nous n'avons pour l'instant mentionné qu'un type entier, *viz.* `int`, mais le langage C en possède plus d'une douzaine (nous ne les décrirons pas tous). On distingue notamment les types « historiques » (dont `int`) des types « modernes » (comme `uint64_t`).
Tous les types entiers définis par le langage possèdent des caractéristiques communes :

- chaque type est ou bien *signé* (permettant alors de représenter des valeurs négatives) ou *non signé* (ne permettant alors que de représenter des valeurs supérieures ou égales à zéro) ;
- chaque type ne permet de représenter qu'un **nombre fini de valeurs**, dans un intervalle consécutif ; ces intervalles sont de taille fixe pour les types modernes, et de taille variable (en fonction du compilateur, de l'architecture matérielle…) pour les types historiques ;
- un calcul effectué avec des entiers **signés** d'un type donné dont le résultat n'est pas représentable par ce même type (c'est à dire, est hors de l'intervalle des valeurs représentables ; on parle de *dépassement de capacité*) mène à un comportement non défini, et est donc à éviter, *cf.* INT32-C. Ensure that operations on signed integers do not result in overflow ;
- un calcul effectué avec des entiers **non signés** d'un type donné dont le résultat r n'est pas représentable par ce même type est bien défini, et a pour résultat la valeur représentable congrue à r modulo le nombre de valeurs représentables par le type. Par exemple, le type `uint32_t` permet de représenter les 2^{32} valeurs consécutives de l'intervalle $[\![0, 2^{32}-1]\!]$; si un calcul sur des entiers de ce type devait avoir 2^{32} comme résultat quand effectué « dans $\mathbb{Z}$ », alors il aura 0 pour résultat réel ; s'il devait avoir -1 comme résultat dans $\mathbb{Z}$, alors il aura $2^{32}-1$ comme résultat réel.

¶ **`limits.h`.** Le fichier d'en-tête `limits.h` définit des symboles pour les valeurs minimales et maximales représentables par certains types entier historiques. Par exemple, les symboles `INT_MIN` et `INT_MAX` représentent des valeurs de type `int` respectivement égales à la valeur minimale et maximale représentable par un `int`.

Quand ces valeurs peuvent varier (comme c'est le cas pour `int`), les symboles représentent toujours les valeurs exactes pour la configuration utilisée. Autrement dit, s'il se peut qu'un programme comportant l'instruction `printf("%d\n", INT_MIN);` affiche des valeurs différentes quand exécuté dans deux contextes (compilateur, architecture matérielle…) distincts, il affichera toujours la valeur minimale représentable par un `int` *dans le contexte où il est exécuté*.

¶ **`int`.** Le type entier `int` permet de représenter des entiers signés de valeur entre `INT_MIN` et `INT_MAX`, qui pour la plupart des compilateurs et sur la plupart des architectures modernes valent respectivement $-2\,147\,483\,648 = -2^{31}$ et $2\,147\,483\,647 = 2^{31}-1$ et valent au moins $-32\,767 = -2^{15}+1$ et $32\,767 = 2^{15}-1$.
Le type `int` peut être utilisé pour représenter de petites quantités entières pouvant être négatives.
La spécification de conversion pour l'affichage avec `printf` d'une valeur de type `int` en écriture décimale est `%d`.

¶ **`unsigned int`.** Le type entier `unsigned int`, que l'on écrit usuellement de façon abrégée comme `unsigned`, permet de représenter des entiers non signés de valeur entre `0` et `UINT_MAX`, qui sur la plupart des architectures modernes vaut $4\,294\,967\,295 = 2^{32}-1$.
Le type `unsigned` peut être utilisé pour représenter de petites quantités entières qui ne sont jamais négatives. Utiliser `unsigned` plutôt que `int` dans ce cas a l'avantage de documenter le fait que la quantité est garantie d'être toujours non-nulle ; il a l'inconvénient d'être source d'erreurs potentielles dans des situations où il serait naturel de vouloir diminuer une quantité jusqu'à ce qu'elle soit strictement inférieure à zéro, puisque cela n'est *jamais le cas* pour un entier non signé.

La spécification de conversion pour l'affichage avec `printf` d'une valeur de type `unsigned` en écriture décimale est `%u`, et `%x` ou `%X` pour un affichage en écriture hexadécimale (en base 16).

Le fichier d'en-tête `stdint.h` définit les types modernes suivant, ainsi que certaines informations s'y rapportant (par exemple divers symboles de limite de représentation) :

¶ `int8_t`. Le type entier `int8_t` utilise exactement 8 bits pour représenter des entiers signés de valeur entre `INT8_MIN` et `INT8_MAX`, qui valent toujours respectivement $-128 = -2^7$ et $127 = 2^7 - 1$.

¶ `int16_t`. Le type entier `int16_t` utilise exactement 16 bits pour représenter des entiers signés de valeur entre `INT16_MIN` et `INT16_MAX`, qui valent toujours respectivement $-32\,768 = -2^{15}$ et $32\,767 = 2^{15} - 1$.

¶ `int32_t`. Le type entier `int32_t` utilise exactement 32 bits pour représenter des entiers signés de valeur entre `INT32_MIN` et `INT32_MAX`, qui valent toujours respectivement $-2\,147\,483\,648 = -2^{31}$ et $2\,147\,483\,647 = 2^{31} - 1$.

¶ `int64_t`. Le type entier `int64_t` utilise exactement 64 bits pour représenter des entiers signés de valeur entre `INT64_MIN` et `INT64_MAX`, qui valent toujours respectivement $-9\,223\,372\,036\,854\,775\,808 = -2^{63}$ et $9\,223\,372\,036\,854\,775\,807 = 2^{63} - 1$.

¶ `uint8_t`. Le type entier `uint8_t` utilise exactement 8 bits pour représenter des entiers non signés de valeur entre `0` et `UINT8_MAX`, qui vaut toujours $255 = 2^8 - 1$.

¶ `uint16_t`. Le type entier `uint16_t` utilise exactement 16 bits pour représenter des entiers non signés de valeur entre `0` et `UINT16_MAX`, qui vaut toujours $65\,535 = 2^{16} - 1$.

¶ `uint32_t`. Le type entier `uint32_t` utilise exactement 32 bits pour représenter des entiers non signés de valeur entre `0` et `UINT32_MAX`, qui vaut toujours $4\,294\,967\,295 = 2^{32} - 1$.

¶ `uint64_t`. Le type entier `uint64_t` utilise exactement 64 bits pour représenter des entiers non signés de valeur entre `0` et `UINT64_MAX`, qui vaut toujours $18\,446\,744\,073\,709\,551\,615 = 2^{64} - 1$.

§ **Typage et écriture des littéraux entiers**

En C, les littéraux entiers (c'est à dire les valeurs s'écrivant littéralement, comme `12`) peuvent s'écrire notamment en base dix (c'est le comportement usuel), mais aussi en base seize (de chiffres 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*, *B*, *C*, *D*, *E*, *F*) en les préfixant par `0x`, en base huit en les préfixant par `0`, ou encore en base deux en les préfixant par `0b`. Le fragment de code suivant utilise ces quatre écritures possibles pour définir quatre fois une variable contenant la valeur douze :

Fragment 3.16

```
int douze_dix   = 12;
int douze_seize = 0xC; // ou 0xc
int douze_huit  = 014;
int douze_deux  = 0b1100;
```

Par défaut les littéraux sont de type `int`, c'est à dire que si la valeur dénotée par la constante est représentable par le type `int` alors c'est celui qu'elle aura. Sinon, le type sera un autre type « historique » typiquement parmi `unsigned`, `long` (que nous ne détaillerons pas plus que ça) ou `unsigned long` (*ditto*) en fonction d'une règle logique (quoique...) que nous ne détaillerons pas non plus. Si l'on souhaite changer ce comportement par défaut (ce qui est occasionnellement pratique), il suffit de suffixer le littéral par la lettre `U` pour forcer le type à un type non signé et par `L` ou `LL` pour forcer le type à un type `long` ou `long long`, les deux pouvant se combiner. Par exemple, `12` est de type `int`, `12U` de type `unsigned`, et `12UL` (ou `12LU`) de type `unsigned long`.

§ **Conversion entre types entiers**

Du fait de l'existence de (très) nombreux types entiers en C, il se peut au cours d'un programme que l'on cherche à affecter une valeur d'un certain type (par exemple `int`) à une variable d'un autre type (par exemple `unsigned`), ou bien qu'une expression (*cf.* 3.8) fasse intervenir des valeurs de types différents. Toutes ces situations donneront lieu à une ou plusieurs *conversions de type* explicites (syntaxiquement « demandées » par la personne écrivant le programme, *cf.* Section 3.14) ou implicites (« décidées » par le langage).

Qu'elle soit explicite ou implicite, une conversion entre deux types entiers peut être source de multiples problèmes, typiquement de dépassement de capacité ou de changement d'interprétation.

Par exemple, le comportement du programme ci-dessous est *implementation defined* ce qui n'est pas tout à fait aussi catastrophique que causer un comportement non défini, mais pas terrible pour autant à cause d'un dépassement de capacité sur type signé :

Programme 3.17

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

int main(void)
{
    uint8_t a = 160;
    int8_t b = a;

    return EXIT_SUCCESS;
}
```

Plus amusant est le programme suivant qui tente de déterminer si zéro est supérieur ou non à moins un ; son comportement est cette fois parfaitement défini, mais ne permet pas vraiment de conclure puisque les comparaisons s'évaluent deux fois à `true` et deux fois à `false` :

Programme 3.18

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

int main(void)
{
```



```
    printf("%d\n", 0 > -1); // 1
    printf("%d\n", 0U > -1); // 0
    printf("%d\n", 0U > -1L); // 1
    printf("%d\n", 0UL > -1L); // 0

    return EXIT_SUCCESS;
}
```

En général, on choisira donc avec soin les variantes des types entiers utilisés afin d'éviter les conversions autant que possible (*cf.* également INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data), et si une conversion *non triviale* est inévitable on essaiera de documenter son (bon) comportement. On pourra cependant largement s'autoriser les conversions implicites « usuelles » comme :

Fragment 3.19

```
unsigned a = 0; // pas nécessaire d'utiliser 0U
```

qui ne sont sources d'aucune erreur ou comportement étrange, et sont de toutes façons extrêmement courantes.

§ **Affichage des valeurs de types entier modernes**

La façon dont les types entiers modernes sont définis en pratique fait qu'il est un peu plus délicat de les afficher « proprement » que les types historiques. En pratique, on pourra sans grand problème utiliser les mêmes spécifications de conversions que pour `int` et `unsigned` pour tous les types respectivement signés et non signés autres que `int64_t` et `uint64_t` ; pour ces derniers, on peut utiliser `%ld` ou `%lld` et `%lu` ou `%llu` & `%lx` ou `%llx` & `%lX` ou `%llX` au prix d'un éventuel avertissement du compilateur aisé à éliminer.

Une solution « propre » (mais un peu lourde) est d'utiliser des symboles définis dans le fichier d'en-tête `inttypes.h` et procéder comme dans l'exemple ci-dessous.

Programme 3.20

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

int main(void)
{
    int8_t i8 = INT8_MIN;
    printf("[int8_t] valeur min %" PRId8 "\n", i8);
    i8 = INT8_MAX;
    printf("[int8_t] valeur max %" PRId8 "\n", i8);

    int16_t i16 = INT16_MIN;
```

```c
    printf("[int16_t] valeur min %" PRId16 "\n", i16);
    i16 = INT16_MAX;
    printf("[int16_t] valeur max %" PRId16 "\n", i16);

    int32_t i32 = INT32_MIN;
    printf("[int32_t] valeur min %" PRId32 "\n", i32);
    i32 = INT32_MAX;
    printf("[int32_t] valeur max %" PRId32 "\n", i32);

    int64_t i64 = INT64_MIN;
    printf("[int64_t] valeur min %" PRId64 "\n", i64);
    i64 = INT64_MAX;
    printf("[int64_t] valeur max %" PRId64 "\n", i64);

    uint8_t u8 = UINT8_MAX;
    puts("[uint8_t] valeur min 0");
    printf("[uint8_t] valeur max %" PRIu8 "\n", u8);

    uint16_t u16 = UINT16_MAX;
    puts("[uint16_t] valeur min 0");
    printf("[uint16_t] valeur max %" PRIu16 "\n", u16);

    uint32_t u32 = UINT32_MAX;
    puts("[uint32_t] valeur min 0");
    printf("[uint32_t] valeur max %" PRIu32 "\n", u32);

    uint64_t u64 = UINT64_MAX;
    puts("[uint64_t] valeur min 0");
    printf("[uint64_t] valeur max %" PRIu64 "\n", u64);

    return EXIT_SUCCESS;
}
```

Une autre solution propre mais nécessitant un compilateur implémentant (au moins partiellement) la dernière version de la norme C (*viz.* C23) est de préfixer la spécification de conversion par `%wn`, avec `n` le nombre de bits du type ; par exemple, on peut utiliser `%w64u` pour afficher une valeur de type `uint64_t` en écriture décimale.

3.7 Type booléen

Le fichier d'en-tête `stdbool.h` définit un type « booléen » `bool`, comportant exactement deux valeurs `true` (« vrai ») et `false` (« faux »).

Il n'existe actuellement pas de spécification de conversion des valeurs de type `bool` dans le standard C ; celles-ci s'affichent habituellement comme des entiers, avec la conversion `true` ↪ `1` et `false` ↪ `0`.

3.8 Expressions (arithmétiques & booléennes)

§ Expressions

Les *expressions* font partie des constructions syntaxiques de base dans de nombreux langages de programmation. En C les expressions sont peu indépendantes des *instructions* (dont nous parlerons à la Section 3.9), mais en sont néanmoins distinctes.

Informellement, une expression est « quelque chose » qui peut s'*évaluer* pour produire une *valeur*. En C toute expression a un *type*, qui comme pour une variable détermine comment interpréter le résultat de son évaluation.

Les expressions les plus courantes sont :

— les *constantes littérales* (ou *littéraux*), c'est à dire les « valeurs de base » des types ; dans ce cas, l'expression s'évalue simplement en la valeur du littéral. Par exemple, `12` est une constante littérale de type entier `int`, et est donc également une expression de type `int` qui s'évalue en `12` ;
— les (noms de) variables ; dans ce cas, l'expression s'évalue en la valeur courante de l'objet référencé par la variable. Par exemple, si l'on vient de déclarer une variable `tw` de type `int` et initialisée à `12`, alors `tw` est une expression de type `int` s'évaluant à `12` ;
— les expressions arithmétiques et booléennes, formées en combinant entre elles plusieurs *sous-expressions* avec des *opérateurs* ; dans ce cas, évaluer l'expression revient à effectuer un calcul (suivant des règles dépendant des types des sous-expressions et des définitions des opérateurs). Par exemple, `10 + 2` est une expression de type `int` obtenue en combinant les deux expressions `10` et `2` avec l'opérateur `+`, qui s'évalue à `12`. Une curiosité du C est qu'il n'existe pas de littéraux représentant les nombres négatifs, mais un opérateur unaire de négation permet de facilement construire des expressions de valeur négative. Par exemple, `-12` n'est pas un littéral, mais une expression de type `int` obtenue en appliquant l'opérateur `-` à la sous-expression (littérale) `12` ;
— les appels de fonctions ; dans ce cas, l'expression s'évalue en la valeur renvoyée par la fonction.

§ Expressions arithmétiques

Les opérateurs arithmétiques usuels s'écrivent `+` ; `-` ; `*` ; `/` ; `%` en C, pour tous les types entiers. Ils correspondent respectivement à l'addition, la soustraction et la négation unaire, la multiplication, et un quotient & reste dans une division entière.

Ces opérateurs permettent de construire des *expressions arithmétiques* à partir de variables de types entiers, de littéraux entiers, ou en général d'autres expressions de type entier. Ils s'emploient en *notation infixe* (ce qui est usuel), c'est à dire que l'opérateur est placé entre ses opérandes ; on note par exemple `3 * 4` pour créer l'expression construisant le produit de `3` et `4`. On peut séparer les opérandes de l'opérateur par une ou plusieurs espaces ; profiter de cette possibilité rend souvent le code plus agréable à lire.

Tant que les expressions construites avec ces opérateurs s'évaluent en des valeurs représentables, celles-ci sont « les mêmes que sur $\mathbb{Z}$ ». Cependant, à l'exception du dernier (et de l'avant-dernier pour les types non signés), tous permettent de construire des valeurs non représentables par le type de leurs opérandes, ce qui (on le rappelle) est un comportement non défini dans le cas de types signés. Les deux derniers opérateurs peuvent (sans surprise ?) également être source de comportement non défini si leur seconde opérande est nulle, *cf.* INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors.

Les règles de priorité entre opérateurs sont les mêmes que celles apprises à l'école, et l'on peut également utiliser des parenthèses pour changer le comportement par défaut ou le rendre explicite (ce qui peut ou non améliorer la lisibilité de l'expression, en fonction des cas & des goûts). Par exemple, `b*b - 4*a*c`

est équivalent à `(b*b) - (4*a*c)`, `(b*b) - (4*(a*c))` ou encore `b* b-4 *a* ((c))` ~~; on espère qu'il est évident que la dernière version est la plus lisible~~.

¶ **Quotient & reste.** L'opérateur `%` permet de construire des expressions `x % y` qui s'évaluent en le reste dans une division entière (« euclidienne ») de `x` par `y`. Seul le comportement dans le cas d'opérandes positives est au programme : le reste calculé appartient alors à l'intervalle $[\![0, y-1]\!]$.

L'opérateur `/` permet de construire des expressions `x / y` qui s'évaluent en le quotient dans une division entière de `x` par `y`. Contrairement à l'opérateur `%`, le programme n'émet pas de restriction sur les comportements à connaître, ce qui est absurde étant donné que les règles se déduisent de celles pour le reste et *vice-versa*.

De façon générale, la valeur absolue du reste r calculé par `%` appartient à l'intervalle $[\![0, y-1]\!]$ et il a le même signe que x ; le quotient q calculé par `/` est tel que l'on a l'égalité $q \times y + r = x$ (avec le bon signe), et est donc négatif ssi. exactement l'une des deux opérandes l'est.

Le plus simple reste cependant de n'utiliser `/` & `%` que pour des opérandes positives, quitte à introduire des calculs supplémentaires si l'on désire ultimement calculer quelque chose de différent.

Le programme suivant permet de visualiser les différents cas possibles. Quels comportements non-définis pourraient être déclenchés par une utilisation imprudente de la fonction `quorem_examples` ?

Programme 3.21

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

void quorem_examples(int x, int y)
{
    printf("%d / %d = %d, reste %d (%d * %d + %d = %d)\n",
            x,   y,   x/y,     x%y,  y, x/y, x%y, y*(x/y) + x%y);
    x = -x;
    printf("%d / %d = %d, reste %d (%d * %d + %d = %d)\n",
            x,   y,   x/y,     x%y,  y, x/y, x%y, y*(x/y) + x%y);
    y = -y;
    printf("%d / %d = %d, reste %d (%d * %d + %d = %d)\n",
            x,   y,   x/y,     x%y,  y, x/y, x%y, y*(x/y) + x%y);
    x = -x;
    printf("%d / %d = %d, reste %d (%d * %d + %d = %d)\n",
            x,   y,   x/y,     x%y,  y, x/y, x%y, y*(x/y) + x%y);
}

int main(void)
{
    quorem_examples(12, 3);
    puts("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    quorem_examples(13, 3);

    return EXIT_SUCCESS;
}
```

§ Expressions booléennes

Les opérateurs de comparaison entre valeurs de types entier s'écrivent `==` ; `!=` ; `<` ; `<=` ; `>` ; `>=` pour tous les types entiers. Ils correspondent respectivement au test d'égalité, de différence, d'infériorité stricte & non stricte, de supériorité stricte & non stricte. Ils construisent des expressions *booléennes*, qui s'évaluent en une valeur `true` ou `false`. Ils s'emploient en notation infixe, et sont moins prioritaires que tous les opérateurs arithmétiques.

Il est *possible* d'« enchaîner » plusieurs opérateurs de comparaison, mais ceci n'a probablement pas le résultat que vous espérez. Par exemple, `3 == 3 == 3` s'évalue à `false`. En général, on évitera de recourir à de telles constructions.

Les opérateurs usuels de conjonction, disjonction, négation permettent de construire des expressions booléennes à partir de variables booléennes, de littéraux booléens, ou encore d'autres expressions de type booléen ; ils s'écrivent `&&`, `||` et `!`. L'opérateur de négation s'utilise en notation *préfixe*, c'est à dire comme « `!x` ».

Les opérateurs `&&` et `||` ont la particularité d'être *paresseux* : ils n'évaluent leur seconde opérande (opérande de droite) que si celle-ci est nécessaire au calcul du résultat. Ceci a de l'importance si cette évaluation coûte cher ou a des *effets* (comme par exemple déclencher un comportement non défini). Le programme ci-dessous donne un exemple d'utilisation classique de cette paresse :

Programme 3.22

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

bool is_multiple(unsigned x, unsigned y)
{
    return (y != 0) && (x % y == 0);
}

bool is_multiple_unsafe(unsigned x, unsigned y)
{
    return x % y == 0;
}

int main(void)
{
    printf("%d\n", is_multiple(12, 3));
    printf("%d\n", is_multiple(13, 3));
    printf("%d\n", is_multiple(13, 0));
    printf("%d\n", is_multiple_unsafe(13, 0)); // boum

    return EXIT_SUCCESS;
}
```

3.9 Instructions & structures de contrôle

§ Instructions

Les *instructions* (ou parfois *commandes* ; en anglais : *statements*) regroupent un ensemble de constructions syntaxiques dont le but est généralement de contrôler l'état et le *flot d'exécution* d'un programme.

Tout comme pour les expressions, ce qu'est ou n'est pas une instruction dépend du langage de programmation ; il n'y a pas de « vérité générale », et ce qui est une instruction en C peut par exemple ne pas en être une (pour une construction équivalente) en OCaml.

En tant que langage à dominance impérative, la plupart des constructions syntaxiques en C sont des instructions, et l'exécution d'un programme peut s'assimiler à une suite d'exécution d'instructions.

¶ **Instructions élémentaires.** Nous avons déjà rencontré un certain nombre d'instructions, parmi lesquelles :

- l'instruction `return`, qui permet de terminer l'exécution d'une fonction (et d'optionnellement faire renvoyer une valeur à celle-ci) ;
- l'affectation de variable ;
- l'appel de fonction suivi d'un « `;` » ;
- toute liste d'instructions, c'est à dire des instructions littéralement « collées » les unes aux autres ;
- tout *bloc* d'instructions, formé d'une liste d'instructions (éventuellement vide...) placées entre « `{` » et « `}` » ; en plus des instructions, on peut également déclarer des variables au sein d'un bloc.

§ Instructions de sélection

Les instructions de *sélection* permettent de choisir la prochaine instruction à exécuter, en fonction d'une ou plusieurs *conditions*. Le langage C définit deux instructions de sélection : `if` et `switch` ; seule la première est au programme de MP2I/MPI, et c'est donc la seule que nous décrirons.

L'instruction `if` possède deux syntaxes possibles :

Syntaxe 3.23

```
if (expression) instruction_si_vrai
```

et

Syntaxe 3.24

```
if (expression) instruction_si_vrai else instruction_si_faux
```

Dans le cadre des CPGE, nous rajouterons les contraintes suivantes :

- l'expression de contrôle `expression` **doit** être une expression booléenne (c'est à dire de type `bool`) ;
- les instructions contrôlées par le `if` (`instruction_si_vrai` et `instruction_si_faux`) se **doivent** d'être des instructions bloc. Avec ces contraintes, on obtient (dans la seconde forme, plus générale) :

Syntaxe 3.25

```
if (expression_booléenne)
{
    instructions_si_vrai
}
else
{
    instructions_si_faux
}
```

Les noms `instructions_si_vrai` et `instructions_si_faux` deviennent ici techniquement trompeurs car un bloc peut également contenir des déclarations, *cf.* ci-dessus. On omet sciemment ce dernier point dans la discussion suivante afin de ne pas trop la complexifier.

La *sémantique* de l'instruction `if` (c'est à dire : « ce qu'elle fait ») est intuitive : on commence par évaluer `expression_booléenne` ; si celle-ci s'évalue à `true`, on exécute les instructions du bloc suivant immédiatement le `if`, c'est à dire `instructions_si_vrai` ; si celle-ci s'évalue à `false`, on exécute les instructions `instructions_si_faux` si elles sont présentes (c'est à dire, s'il y a un `else`), et sinon on ne fait rien.

Puisqu' `if` est une instruction, elle peut notamment faire partie d'une liste d'instructions ; on peut donc faire suivre un `if` d'instructions qui seront exécutées « quoi qu'il arrive » (indépendamment de la valeur en laquelle s'évalue la condition), à condition que le programme ou la fonction n'ait pas déjà terminé (par exemple parce que `instruction_si_vrai` contient une instruction `return`). Ces éventuelles instructions ne font cependant pas partie du `if` lui-même.

Pour la même raison, `instructions_si_vrai` (disons) peuvent elles-mêmes être constituées d'un ou plusieurs `if` ; on parle alors de `if`s *imbriqués*. Il est également possible d'enchaîner plusieurs `if`s à la suite, et l'on pourra dans ce cas (et seulement ce cas) se dispenser des accolades supplémentaires autour de `instruction_si_faux` ; ceci donne le motif suivant :

Fragment 3.26

```
if (cond1)
{
    // instructions si cond1
}
else if (cond2)
{
    // instructions si !cond1 && cond2
}
else if (cond3)
{
    // instructions si !cond1 && !cond2 && cond3
}
// ...
else
```



```
{
    // instructions si aucune des conditions n'est vraie
}
```

Puisque la liste d'instructions dans un bloc peut être vide, on *peut* utiliser un `if` de la façon suivante si l'on n'a rien à faire quand la condition s'évalue à `true` :

Fragment 3.27

```
if (cond)
{
    // rien
}
else
{
    // instructions si !cond
}
```

Si un tel cas d'usage se présente, Il est cependant bien plus léger & idiomatique d'inverser la condition et d'utiliser la forme de `if` sans `else` :

Fragment 3.28

```
if (!cond)
{
    // instructions si !cond
}
```

Le programme suivant illustre l'utilisation de `if`s (et d'expressions booléennes) dans des cas variés.

Programme 3.29

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

int max(int x, int y)
{
    if (x > y)
    {
        return x;
    }
    else
    {
```

```
        return y;
    }
}

int max_var(int x, int y)
{
    if (x > y)
    {
        return x;
    }
    return y;
}

int max3(int x, int y, int z)
{
    if (x > y)
    {
        if (x > z)
        {
            return x;
        }
        else
        {
            return z;
        }
    }
    else if (y > z)
    {
        return y;
    }
    else
    {
        return z;
    }
}

int max3_var(int x, int y, int z)
{
    if (x > y && x > z)
    {
        return x;
    }
    if (y > z)
    {
        return y;
```

```
    }
    return z;
}

bool test_max3(void)
{
    bool all_okay = true;

    puts("Testing max3...");

    if (max3(1, 2, 3) != 3)
    {
        all_okay = false;
        fputs("FAILURE: max3(1, 2, 3) != 3\n", stderr);
    }
    if (max3(3, 1, 2) != 3)
    {
        all_okay = false;
        fputs("FAILURE: max3(3, 1, 2) != 3\n", stderr);
    }
    if (max3(2, 3, 1) != 3)
    {
        all_okay = false;
        fputs("FAILURE: max3(2, 3, 1) != 3\n", stderr);
    }
    if (max3(1, 3, 2) != 3)
    {
        all_okay = false;
        fputs("FAILURE: max3(1, 3, 2) != 3\n", stderr);
    }
    if (max3(2, 1, 3) != 3)
    {
        all_okay = false;
        fputs("FAILURE: max3(2, 1, 3) != 3\n", stderr);
    }
    if (max3(3, 2, 1) != 3)
    {
        all_okay = false;
        fputs("FAILURE: max3(3, 2, 1) != 3\n", stderr);
    }
    if (all_okay)
    {
        puts("SUCCESS: all tests OK\n");
    }
    else
```

```
    {
        fputs("FAILURE: some tests NOK\n\n", stderr);
    }

    return all_okay;
}

int main(void)
{
    test_max3();

    return EXIT_SUCCESS;
}
```

Histoire vraie : l'écriture & exécution de `test_max3` (qui peut sembler un peu rébarbative) a permis de détecter une erreur d'étourderie dans la version initiale de `max3`. *Tout le monde* fait des erreurs (j'ai un jour entendu une légende prétendant l'existence d'une personne écrivant du code toujours correct *du premier coup*, mais je n'y crois pas vraiment), et les tests sont là pour nous aider à les corriger.

§ Instructions d'itération

Les instructions d'*itération* permettent de répéter une instruction plusieurs fois, en fonction d'une condition. Le langage C définit trois instructions d'itération : `while`, `do while` et `for` ; bien que parfois très pratique, la seconde n'est pas au programme de MP2I/MPI et nous ne la décrirons donc pas :(

¶ **Instruction `while`.** La syntaxe de l'instruction `while` est la suivante :

Syntaxe 3.30

```
while (expression) instruction
```

Tout comme pour `if`, on restreindra l'usage à la forme plus contrainte :

Syntaxe 3.31

```
while (expression_booléenne)
{
    instructions
}
```

Comme précédemment, `instructions` peut en fait également inclure des déclarations…

La sémantique du `while` est naturelle : *tant que* `expression_booléenne` s'évalue à `true`, on exécute le *corps de boucle* `instructions` à répétition. Autrement dit on commence par évaluer la condition `expression_booléenne` ; si elle s'évalue à `false` l'exécution de l'instruction `while` est terminée, sinon on exécute le corps de boucle, puis (à supposer que la boucle `while` n'a pas terminé pour d'autres raisons) on évalue à nouveau `expression_booléenne` ; si elle s'évalue à `false` l'exécution de l'instruction `while` est terminée, sinon on exécute le corps de boucle, puis (à supposer que la boucle `while` n'a pas terminé pour d'autres raisons) on évalue à nouveau `expression_booléenne` ; si elle s'évalue à `false` l'exécution

de l'instruction `while` est terminée, sinon on exécute le corps de boucle, puis (à supposer que la boucle `while` n'a pas terminé pour d'autres raisons) on évalue à nouveau `expression_booléenne` ; si elle s'évalue à `false` l'exécution de l'instruction `while` est terminée, sinon on exécute le corps de boucle, puis (à supposer que la boucle `while` n'a pas terminé pour d'autres raisons) on évalue à nouveau `expression_booléenne` ; si elle s'évalue à `false` l'exécution de l'instruction `while` est terminée, sinon on exécute le corps de boucle, puis (à supposer que la boucle `while` n'a pas terminé pour d'autres raisons) on évalue à nouveau `expression_booléenne` ; si …

En pratique, il est important d'éviter les *boucles infinies* dont l'exécution ne termine jamais. Un (bon) algorithme utilisant une boucle `while` doit donc typiquement être associé à une *preuve de terminaison* de celle-ci.

Il est cependant parfois utile d'écrire une boucle *a priori* infinie, dont la terminaison est décidée par un phénomène extérieur (par exemple la lecture d'une valeur adéquate par un capteur de pression). Il existe plusieurs façons d'écrire une telle boucle en C, mais celle à favoriser dans le cadre des CPGE (qui n'est pas la plus drôle…) est :

Fragment 3.32

```
while (true)
{
    // corps de boucle
}
```

la terminaison de la boucle peut alors se faire depuis son corps en utilisant par exemple une instruction de saut `return` (qui a l'effet habituel, *viz.* celui de faire terminer l'exécution de la fonction contenant le `while`) ou une instruction de saut `break`, dont l'effet est exactement de faire terminer l'instruction `while`.

Il existe également une instruction de saut `continue` similaire à `break`, dont l'effet est d'interrompre l'exécution des instructions du corps de boucle et d'immédiatement évaluer à nouveau l'expression de contrôle ; celle-ci jouera alors son rôle habituel de décider si l'exécution du `while` termine ou pas.

Un corps de boucle peut naturellement contenir d'autres boucles *imbriquées*. Un `break` ou `continue` n'a d'effet que sur la boucle à laquelle il appartient, et pas sur d'éventuelles boucles englobantes.

¶ **Instruction `for`.** La syntaxe de l'instruction `for` est la suivante :

Syntaxe 3.33

```
for (clause ; expression1 ; expression2) instruction
```

Tout comme pour `if` et `while`, on restreindra l'usage à une forme plus contrainte :

Syntaxe 3.34

```
for (clause ; expression_booléenne ; expression_d_affectation)
{
    instructions
}
```

où dans le détail :

— `clause` est ou bien une déclaration de variable (pour nous : avec initialisation), ou bien une *expression d'affectation*, qui est identique à une instruction d'affectation dans sa syntaxe et ses effets, si ce n'est qu'elle ne se termine pas par un « ; » ;
— `expression_d_affectation` est une expression d'affectation, dans le même sens que ci-dessus.

La sémantique du `for` est assez subtile et très *expressive* elle permet notamment de faire strictement plus de choses que `while`. Si l'on appelle `init`, `test` et `incr` les trois composantes d'un `for` (dans l'ordre d'écriture), on a que :

— `init` est exécutée *une fois* au début de l'exécution du `for`. Si elle déclare une variable, celle-ci aura pour *portée* le corps de la boucle (*cf.* Section 3.10) ;
— `test` joue exactement le même rôle que l'unique expression booléenne contrôlant une instruction `while` ;
— `incr` est exécutée à la fin de *chaque* exécution du corps de la boucle.

L'utilisation typique d'une boucle `for` est alors la suivante : `init` déclare une nouvelle variable (typiquement de type entier) qui servira de compteur ; `test` vérifie que le compteur ne dépasse pas une certaine borne ; `incr` modifie la valeur du compteur.

La modification du compteur (appelons le `i`) se fait typiquement en utilisant une expression d'affectation de la forme `i = i + 1`, ou `i = i - 1`, ou (plus rarement) `i = i / 2`, etc. Ces expressions d'affectation sont les seules officiellement au programme, mais elles ne sont pas très idiomatiques ; on pourra se permettre à la fois d'utiliser le *sucre syntaxique* `i += 1` qui est un « raccourci » pour `i = i + 1` (on a de même `i -= 1`, `i /= 2` etc.) ainsi que (**uniquement dans ce cas**) les expressions obtenues à partir des opérateurs unaires d'incrémentation et de décrémentation `++` et `--` : `i++` (resp. `i--`) est *dans ce cas* fonctionnellement équivalent à `i = i + 1` (resp. `i = i - 1`).

Les instructions de saut `break` et `continue` peuvent s'utiliser dans un `for` de la même façon que dans un `while`.

Les fonctions suivantes illustrent une utilisation basique d'une boucle `for` ainsi que d'une boucle `while` fonctionnellement équivalente :

Fragment 3.35

```
unsigned sum_n_while(unsigned n)
{
    unsigned res = 0;
    unsigned i = 0;

    while (i <= n)
    {
        res = res + i;
        i = i + 1;
    }

    return res;
}

unsigned sum_n_for(unsigned n)
{
```



```
    unsigned res = 0;

    for (unsigned i = 0; i <= n; i++)
    {
        res = res + i;
    }

    return res;
}

unsigned sum_n_math(unsigned n)
{
    return (n * (n + 1)) / 2;
}
```

Quel est un (très léger) inconvénient de `sum_n_math` par rapport à `sum_n_for` ? En quoi les conditions de `sum_n_while` et `sum_n_for` sont potentiellement dangereuses ?

Toute ou partie de `init`, `test`, `incr` peut être laissée vide :

— `for (; test; incr)` peut être utile pour enchaîner plusieurs boucles qui partagent un même compteur qu'on ne souhaite pas réinitialiser à chaque boucle ;
— `for (;;)` est équivalent à `while (true)` (et bien plus rigolo, mais on s'abstiendra de l'utiliser) ;
— `for (; test;)` est équivalent à `while (test)`, qui est dans ce cas à privilégier ;
— les autres cas sont plus rarement utiles.

Le programme suivant illustre l'utilisation d'un `for` avec initialisation vide.

Programme 3.36

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

void print_n_star(unsigned n)
{
    for (unsigned i = 0; i < n; i++)
    {
        printf("*");
    }
    puts("");
}

void star_sym(unsigned n)
{
    unsigned i = 1;

    if (n == 0)
```

```
    {
        return;
    }

    for (; i < n; i++)
    {
        print_n_star(i);
    }
    for (; i > 0; i--)
    {
        print_n_star(i);
    }
}

void star_asym(unsigned n)
{
    unsigned i = 1;

    if (n == 0)
    {
        return;
    }

    for (; i < n; i++)
    {
        print_n_star(i);
    }
    for (i = 1; i < n; i++)
    {
        print_n_star(i);
    }
}

int main(void)
{

    star_sym(12);
    puts("");
    star_asym(12);

    return EXIT_SUCCESS;
}
```

§ **Saut au dessus du programme**

Le langage C définit une instruction `goto` de saut inconditionnel. Une légende tenace prétend que cette instruction est dangereuse et ne devrait jamais être utilisée (*cf.* notamment `https://xkcd.com/292/`). Pourtant, certaines situations (comme sortir en une fois de plusieurs boucles imbriquées ou libérer un ensemble de ressources en cas d'erreur) se traitent plus efficacement & élégamment avec un `goto` que de n'importe quelle autre façon, *cf.* MEM12-C. Consider using a goto chain when leaving a function on error when using and releasing resources.

Quoi qu'il en soit, `goto` n'est ~~mal~~heureusement pas au programme, et nous n'en parlerons pas.

3.10 Portée des variables

Les identifiants de variable ont une *portée* (en anglais : *scope*) qui détermine les points du programme où ceux-ci sont *visibles*, et donc utilisables dans des instructions, des expressions…

En cas de conflit de nom entre plusieurs variables (c'est à dire, si plusieurs variables de même nom peuvent *a priori* être visibles en le même point), c'est uniquement la variable correspondant à la déclaration la plus *interne* (ou « récente ») qui reste visible, et l'on dit dans ce cas qu'elle *masque* la ou les autres variables de même nom. Quand elle se produit, une telle situation tend à rapidement rendre le code difficile à lire et peut ainsi entraîner des bugs ; elle est donc à éviter autant que possible.

¶ **Portée fichier.** La portée fichier (*file scope*) est celle des variables dites *globales*, déclarées en dehors de toute fonction ; une variable globale peut être référencée dans tout le fichier où elle est déclarée.

Les variables globales sont à utiliser avec une extrême parcimonie ; leur seule présence nuit considérablement à la lisibilité d'un programme et augmente les risques de bugs. Si l'on ressent un besoin irrépressible d'utiliser une variable globale, on essayera *a minima* de la qualifier `const`.

Les variables globales ont une *durée de stockage* (en anglais : *storage duration*) de type `static` (c'est à dire, pour toute la durée de l'exécution du programme), ce qui implique notamment qu'elles sont initialisées par défaut à leur déclaration même si aucun initialiseur n'est fourni ; on s'abstiendra cependant d'exploiter ce fait. Nous aborderons la notion de durée de stockage & durée de vie des objets plus en détails à la Section 6.1.

¶ **Portée bloc.** Les variables correspondant aux paramètres d'une fonction, déclarées dans l'initialisation d'un `for`, ou au sein d'un bloc ont toutes une portée bloc (*block scope*) correspondant respectivement au corps de la fonction, au corps de la boucle, ou au bloc lui-même. C'est le type de portée rencontré usuellement.

Il existe d'autres types de portée (notamment *function* et *function prototype*) dont la description dépasse le cadre du programme, et nous n'en parlerons donc pas.

Le programme suivant illustre les portées fichier & bloc ainsi que différents phénomènes de masquage. Il utilise les symboles `__func__` et `__LINE__` qui sont remplacés à la compilation respectivement par le nom de la fonction et le numéro de ligne où il se trouvent *littéralement* dans le code.

Programme 3.37

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

const int n = 10;
```



```c
void print_arg(int i)
{
    printf("%s@%d: n = %d\n", __func__, __LINE__, n);
    printf("%s@%d: i = %d\n", __func__, __LINE__, i);
}

unsigned incr(unsigned n)
{
    printf("%s@%d: n = %u\n", __func__, __LINE__, n);
    n = n + 1;
    printf("%s@%d: n = %u\n", __func__, __LINE__, n);

    return n;
}

void silly(void)
{
    int i = 0;
    printf("%s@%d: i = %d\n", __func__, __LINE__, i);
    {
        printf("%s@%d: i = %d\n", __func__, __LINE__, i);
        int i = 1;
        printf("%s@%d: i = %d\n", __func__, __LINE__, i);
    }
    printf("%s@%d: i = %d\n", __func__, __LINE__, i);
}

void very_silly(int i)
{
    if (i != 0)
    {
        int i = 0;
        printf("%s@%d: i = %d\n", __func__, __LINE__, i);
    }
    printf("%s@%d: i = %d\n", __func__, __LINE__, i);
}

int main(void)
{
    printf("%s@%d: n = %d\n", __func__, __LINE__, n);
    print_arg(19);
    incr(12);
    silly();
    very_silly(257);
```

```
    return EXIT_SUCCESS;
}
```

3.11 Politique de passage des arguments d'une fonction

En C, la politique de passage des arguments de fonction est par *valeur* (en anglais : *call-by-value*). Ceci veut dire que lors d'un appel de fonction (disons à un paramètre) `fun(e)` avec `e` une expression, on évalue d'abord celle-ci, puis on affecte sa valeur v à la variable dénotant le paramètre de `fun` ; cette variable est alors accessible (avec portée bloc) dans le corps de la fonction, et référence initialement un objet de valeur v.

Cette politique de passage (qui n'a rien de spécifique au langage C, et est relativement courante) a notamment pour conséquence qu'une modification de la variable-paramètre à l'intérieur d'une fonction *ne change pas* la valeur d'une éventuelle variable passée en argument (on verra cependant un peu plus tard qu'on peut dans certains cas changer le contenu *référencé* par un argument).

Le programme suivant illustre les effets de cette politique de passage dans un cas simple.

Programme 3.38

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

void no_incr(unsigned x)
{
    printf("%s@%d: x = %u\n", __func__, __LINE__, x);
    x = x + 1;
    printf("%s@%d: x = %u\n", __func__, __LINE__, x);
}

int main(void)
{
    unsigned x = 0;

    printf("%s@%d: x = %u\n", __func__, __LINE__, x);
    no_incr(x);
    printf("%s@%d: x = %u\n", __func__, __LINE__, x);

    return EXIT_SUCCESS;
}
```

3.12 Entrées élémentaires

La gestion des *entrées* utilisateur en C est assez complexe, et source fréquente de bugs. Dans le cadre des CPGE nous n'utiliserons que deux mécanismes, même pas forcément « correctement ».

§ `scanf`

La fonction `scanf` est déclarée dans le fichier d'en-tête `stdio.h`. Elle permet d'effectuer des lectures *formatées* sur l'*entrée standard* (en anglais : *standard input*) `stdin` et de stocker les valeurs lues dans des objets référencés par des variables.

Tout comme `printf`, `scanf` est une fonction variadique dont le premier paramètre est une chaîne de caractères contenant des spécifications de conversion (similaires à celles utilisées par `printf`) et les paramètres suivant (un pour chaque spécification de conversion) des *adresses* de variables de type compatible avec la spécification de conversion qui lui correspond.

Nous étudierons la notion d'adresse plus en détails plus tard, et il est pour l'instant suffisant d'admettre que l'adresse d'une variable de type entier s'obtient en préfixant son nom par une esperluette « `&` ».

Les spécifications de conversion indiquent à la fois un type attendu et un format dans lequel l'entrée doit être écrite. Par exemple, `%d` (l'un des cas les plus courant) indique que l'entrée devra pouvoir être représentable par un `int` et être écrite en base 10.

La fonction `scanf` renvoie un entier `int` indiquant le nombre d'entrées qui ont été lues avec succès. Il est difficile de poursuivre l'exécution d'un programme lorsque ce nombre ne correspond pas à celui voulu ; dans notre cas, le plus simple est alors encore d'interrompre l'exécution (si possible de façon « propre », par exemple en utilisant `exit` avec un code d'erreur, *cf.* Section 3.13).

Le fragment suivant illustre une utilisation simple de `scanf` :

Fragment 3.39

```
unsigned g = 0;

int rd = scanf("%u", &g);
if (rd != 1)
{
    // gestion d'erreur
}
```

§ **Arguments du `main`**

Nous n'avons pour l'instant que présenté des programmes utilisant une fonction `main` de signature `int main(void)`. Il existe une autre signature possible, qui permet de récupérer les arguments fournis à l'invocation du programme en *ligne de commande* (depuis un terminal) ; celle-ci peut s'écrire de plusieurs façons équivalentes, et nous privilégierons : `int main(int argc, char *argv[argc + 1])`.
Les deux paramètres associés à cette signature jouent les rôles suivants :

— `argc` est un entier égal au nombre d'arguments fournis à l'invocation, plus 1
— `argv` est un *tableau* de chaînes de caractères ; nous reparlerons des tableaux plus en détail à la Section 4, et pour l'heure il suffit d'admettre que :
  — le zéroième élément du tableau, auquel on accède par la syntaxe `argv[0]` est la chaîne donnant le nom *complet* sous lequel le programme a été appelé (rarement utile pour nous, même si cela permet de faire des choses rigolotes) ;
  — le *i*ème élément du tableau `argv[i]` pour `i` allant de `1` à `argc - 1` est une chaîne de caractères égale au *i*ème argument fourni à l'invocation ;
  — le `argc`-ième élément du tableau `argv[argc]` est une valeur spéciale `nullptr` somme toute assez inutile, et nous ne l'utiliserons pas.

Le programme suivant illustre une utilisation élémentaire de ces paramètres : il se contente d'afficher sur la sortie standard le nom sous lequel il a été invoqué ainsi que tous ses arguments.

Programme 3.40

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

int main(int argc, char *argv[argc + 1])
{
    for (int i = 0; i < argc; i++)
    {
        printf("%s ", argv[i]);
    }
    puts("");

    return EXIT_SUCCESS;
}
```

Par exemple, si appelé `pif` et invoqué comme `> ./pif paf pouf` depuis un terminal, ce programme affichera `./pif paf pouf`.

Le fait que les arguments sont fournis au `main` sous forme de chaîne de caractères implique qu'une conversion préalable peut être nécessaire, typiquement si l'on souhaite lire un argument numérique. On peut par exemple utiliser pour cela la fonction `atoi` («*ASCII to integer*»), déclarée dans le fichier d'en-tête `stdlib.h`. Cette fonction convertit son unique paramètre (de type chaîne de caractères) qui doit représenter un entier écrit en base 10 en un entier de type `int` de valeur correspondante, et renvoie cette valeur ou `0` en cas d'erreur (ce qui rend la détection d'erreur bien malaisée si `0` est *aussi* une valeur pouvant être légitimement lue).

Le programme suivant illustre une utilisation simple des arguments en ligne de commande pour contrôler un nombre d'étoiles à afficher sur la sortie standard. Afin de limiter la longueur des lignes (ce qui peut améliorer la lisibilité du code), il utilise le fait que des chaînes de caractère littérales successives sont concaténées.

Programme 3.41

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

void print_n_star(unsigned n)
{
    for (unsigned i = 0; i < n; i++)
    {
        printf("*");
    }
    puts("");
}
```

```c
void star_sym(unsigned n)
{
    unsigned i = 1;

    if (n == 0)
    {
        return;
    }

    for (; i < n; i++)
    {
        print_n_star(i);
    }
    for (; i > 0; i--)
    {
        print_n_star(i);
    }
}

int main(int argc, char *argv[argc + 1])
{
    if (argc < 2)
    {
        printf("utilisation : %s n\n", argv[0]);
        puts("              n : entier >= 0; "
             "nombre maximum d'étoiles sur une ligne");
        return EXIT_FAILURE;
    }

    int n = atoi(argv[1]);
    if (n < 0)
    {
        fprintf(stderr, "erreur : n doit être non négatif\n");
        return EXIT_FAILURE;
    }

    star_sym((unsigned)n);

    return EXIT_SUCCESS;
}
```

3.13 Interruption d'exécution

§ Macro `assert`

Le fichier d'en-tête `assert.h` définit une *macro* `assert`, qu'on peut assimiler à une fonction. Celle-ci prend en entrée un argument qui doit pouvoir s'évaluer comme expression booléenne qui, si elle s'évalue à `false`, affiche un message de diagnostic et interrompt brutalement l'exécution de tout le programme.

C'est une bonne pratique d'utiliser `assert` pour vérifier les préconditions & postconditions identifiées dans les fonctions ou les invariants, car cela aide à détecter au plus tôt un éventuel bug. Par exemple, le fragment de code suivant fait l'hypothèse d'une fonction `add257` dont les arguments doivent être compris dans l'intervalle $[\![-128, 128]\!]$, et vérifie ces conditions au début de la fonction grâce à deux `assert`.

Fragment 3.42

```
int32_t add257(int32_t x, int32_t y)
{
    assert(x >= -128 && x <= 128);
    assert(y >= -128 && y <= 128);
    // ...
}
```

La logique d'une telle utilisation est que si le résultat de la fonction ne peut pas être correctement calculé pour des arguments violant la précondition, il ne sert à rien de le faire et d'espérer que le résultat du programme sera tout de même correct : autant interrompre brutalement l'exécution au plus près du *point de divergence* à partir duquel le comportement du programme n'est plus bien défini, afin d'essayer de résoudre les problèmes à leur source.

Une telle utilisation d'`assert` fournit également tout simplement une documentation claire et efficace sur ce qui est attendu du programme. En somme, on peut voir un `assert` comme un commentaire exécutable.

La macro `assert` n'a cependant pas *en principe* vocation à être utilisée pour traiter des problèmes surgissant à l'exécution de façon « banale ». Ce n'est par exemple pas un bon outil pour la gestion d'erreur du Fragment 3.39, quand bien-même celle-ci serait irrécupérable. Dans un contexte professionnel, les `assert` sont typiquement utilisés lors de la phase de développement, puis *désactivés* dans les version de production du logiciel ; s'ils sont toujours présents dans le code pour jouer leur rôle de documentation, ils ne sont alors plus exécutés.

§ Fonction `exit`

La fonction `exit` a pour effet de terminer presque immédiatement le programme, avec le même code de retour que la valeur de son argument. Autrement dit, appeler `exit(r)` depuis n'importe quel point du programme fait terminer celui-ci comme si sa fonction `main` venait de terminer en renvoyant `r`.

Le programme ci-dessous illustre les conséquences d'un appel à `exit` sur le déroulement de l'exécution : seul le message `OHAI` sera affiché sur la sortie standard.

Programme 3.43

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

```c
void print_and_quit(void)
{
    puts("OHAI");
    exit(EXIT_SUCCESS);
}

int main(void)
{
    print_and_quit();
    puts("coucou");

    return EXIT_SUCCESS;
}
```

§ **Interruption manuelle**

On peut interrompre manuellement un programme déjà lancé de plusieurs façons. On en décrit ici trois relativement courantes & pratiques, plutôt spécifiques aux environnements UNIX, et qui n'ont en fait rien de spécifique aux programmes écrits en C. On les illustrera avec le programme « `dot` » suivant qui affiche 12 points sur la sortie standard, un toutes les secondes :

Programme 3.44

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h> // pour sleep sous UNIX

int main(void)
{
    for (int i = 0; i < 12; i++)
    {
        printf(".");
        fflush(stdout); // purge le tampon de la sortie standard
        sleep(1); // dort pendant environ une seconde
    }
    puts("");

    return EXIT_SUCCESS;
}
```

Une exécution non interrompue produit la sortie :

Terminal 3.45

```
> ./dot
............
```

¶ `Ctrl+C`. On peut interrompre un programme lancé depuis un émulateur de terminal en écrivant la combinaison de touches `Ctrl+C` (qui s'affichera généralement `^C`) directement dans le terminal. Une interruption au bout d'environ deux secondes produit alors la sortie :

Terminal 3.46

```
> ./dot
..^C
```

où le `^C` n'est pas écrit par le programme `dot` lui-même, mais le terminal.

¶ `timeout`. Le programme utilitaire `timeout` permet d'exécuter une commande (et donc par exemple de lancer un programme) dont l'exécution sera interrompue au bout d'un temps prédéterminé. L'utilisation la plus basique consiste à simplement préfixer la commande par `timeout d` avec `d` une durée en secondes :

Terminal 3.47

```
> timeout 13 ./dot
............
> timeout 2 ./dot
..%
```

où le `%` est affiché par le terminal pour indiquer le fait que le message écrit sur la sortie standard ne se termine pas par un retour à la ligne.

¶ `killall`. Le programme utilitaire `killall` permet de « tuer » tous les *processus* d'un certain nom. L'utilisation la plus basique consiste simplement à exécuter la commande `> killall prog` pour tuer les processus s'appelant `prog`. Généralement, cette commande doit être exécutée depuis un autre terminal que celui où `prog` a été lancé (sauf s'il l'a été en tâche de fond, par exemple...). Dans notre exemple, si l'on exécute `> killall dot` au bout d'environ trois secondes, on obtient par exemple (sur le terminal exécutant `dot`) :

Terminal 3.48

```
> ./dot
...zsh: terminated  ./dot
```

3.14 Conversion explicite de type

Lors de l'exécution d'un programme C, un objet peut être amené à être *converti* en un objet d'un autre type (on trouve parfois le terme de *transtypage*, et en anglais de *(type) casting*), comme évoqué à la Section 3.6 pour les types entiers. Ces conversions sont parfois implicites « sans soucis » (dans le sens où le compilateur n'avertira pas qu'une conversion a été effectuée, mais pas que celle-ci ne peut être source d'aucun bug), mais aussi parfois source d'avertissement. Par exemple, la compilation du programme ci-dessous (qui fait intervenir des éléments du langage décrits à la Section 5) :

Programme 3.49

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

int main(void)
{
    unsigned a = 12;
    int *p = &a;

    printf("%u @ %p\n", *p, p);

    return EXIT_SUCCESS;
}
```

produit les avertissements ci-dessous :

Terminal 3.50

```
> clang -std=c23 -pedantic -Wall -Wextra -Wmost cast.c
cast.c:7:10: warning: initializing 'int *' with an expression of type
'unsigned int *' converts between pointers to integer types with
different sign [-Wpointer-sign]
    7 |     int *p = &a;
      |          ^   ~~
cast.c:9:29: warning: format specifies type 'void *' but the argument
has type 'int *' [-Wformat-pedantic]
    9 |     printf("%u @ %p\n", *p, p);
      |                    ~~       ^
2 warnings generated.
```

Le premier avertissement nous indique que le programme effectue (*via* un pointeur) une conversion « brute » entre un type entier non signé et un type entier signé. Ici, cette conversion n'aura pas de conséquence imprévisible car (en C23) les représentations `int` et `unsigned` de la valeur 12 sont identiques. Il est cependant largement préférable qu'un programme évite une telle conversion ; ici une remédiation facile est d'homogénéiser les types de `a` et `*p`.

Le second avertissement nous indique l'utilisation d'un argument de type `int *` dans `printf` alors qu'un `void *` est attendu. On peut supprimer cet avertissement en convertissant explicitement le type de l'argument `p` vers le type `void *`, en suivant la syntaxe :

Syntaxe 3.51

```
(nouveau_type)expression
```

Avec ces deux changements, on obtient (par exemple) le programme :

Programme 3.52

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

int main(void)
{
    int a = 12;
    int *p = &a;

    printf("%u @ %p\n", *p, (void *)p);

    return EXIT_SUCCESS;
}
```

qui compile sans avertissement.
Malheureusement, cette absence d'avertissement n'est pas une garantie d'absence de *problème* ! Le rôle d'un avertissement est précisément de signaler qu'une conversion éventuellement problématique a été effectuée, et expliciter cette conversion ne change rien à cela (puisque c'est la *même* conversion qui est faite qu'implicitement). Dans le cas présent, une consultation de la norme du langage nous dit que :

> A pointer to `void` may be converted to or from a pointer to any object type. A pointer to any object type may be converted to a pointer to `void` and back again ; the result shall compare equal to the original pointer.

La conversion est donc licite et produit le résultat voulu (ici : afficher une représentation de l'adresse contenue dans le pointeur).

4 Types structurés

Le langage C permet de définir des types *structurés*, qui servent à agglomérer plusieurs types en un seul. Les deux tels types au programme sont les tableaux de longueur fixe et les enregistrements « `struct` », mais le langage en définit d'autres comme les `union` (dont nous ne parlerons pas).

4.1 Enregistrements

Un type enregistrement permet de définir un unique type pouvant posséder plusieurs *champs* (en anglais : *fields*) de types différents. Ceci est souvent utile en relation avec les paramètres et valeur de retour des fonctions. Supposons une fonction ayant 10 paramètres, son invocation ressemblera à :

Fragment 4.1

```c
fun(a, b, c, d, e, f, g, h, i, j);
```

ce qui est assez long et pénible à écrire, et le sera encore plus avec des variables de nom explicite. Une solution peut alors être de regrouper plusieurs paramètres au sein d'un seul ; par exemple, si les 5

premiers (resp. derniers) paramètres de la fonction du Fragment 4.1 sont logiquement reliés entre eux, on pourrait les fusionner en un seul paramètre, et une invocation ressemblera alors à :

Fragment 4.2

```
fun2(s_1, s_2);
```

qui est nettement plus lisible.
Un autre problème potentiel est celui d'une fonction devant retourner plusieurs valeurs, par exemple une valeur de type `unsigned` et deux de type `int` : le type de retour doit alors pouvoir regrouper ces trois valeurs en une seule. On pourrait également résoudre cette situation en émulant un passage d'arguments par référence en utilisant des paramètres de type pointeur, ce qu'on verra à la Section 5.
Enfin de façon générale, on peut souhaiter vouloir regrouper ensemble plusieurs variables intrinsèquement reliées entre elles afin de matérialiser ces relations, et ainsi rendre le code plus lisible.

§ **Définition & utilisation**

La définition en C d'un type enregistrement se fait avec la syntaxe suivante :

Syntaxe 4.3

```
struct nom_de_la_struct
{
    type_champ1 nom_champ1;
    type_champ2 nom_champ2;
    // ...
};
```

Le type ainsi défini a une visibilité dépendant de son point de définition. En général, ce dernier est en dehors de toute fonction, et la visibilité est donc de type fichier.

La déclaration de variables de type enregistrement se fait de façon similaire aux variables d'autres types, mais la syntaxe peut être légèrement contre-intuitive : le nom du type défini par l'exemple ci-dessus est `struct nom_de_la_struct` (en deux mots), et non pas juste `nom_de_la_struct`. On en illustre les conséquences avec le fragment de code suivant :

Fragment 4.4

```
struct gcof
{
    // ...
};

struct gcof xgcd(unsigned a, unsigned b)
{
    struct gcof res;
    // ...
    return res;
```



```
}
```

Les types des champs définis peuvent eux-mêmes être de genre `struct` ; on peut tout à fait envisager quelque chose comme :

Fragment 4.5

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

struct gcof
{
    unsigned d;
    struct int_pair uv;
}
```

¶ `typedef.` Si l'on trouve la syntaxe ci-dessus un peu verbeuse, on *peut* (mais aucunement *doit*) définir un *alias* pour le nom du type, en utilisant la construction `typedef`, de syntaxe suivante :

Syntaxe 4.6

```
typedef struct ma_struct  nouveau_nom_pour_ma_struct;
```

On peut en fait utiliser cette construction pour n'importe quel type, et par exemple renommer le type `int` en `petit_poney`, mais on s'interdira une telle (relativement) mauvaise pratique dans le cadre des CPGE.
En utilisant cette construction, le Fragment 4.4 peut se réécrire :

Fragment 4.7

```
struct gcof_s
{
    // ...
};

typedef struct gcof_s gcof;

gcof xgcd(unsigned a, unsigned b)
{
    gcof res;
    // ...
    return res;
}
```

¶ **Affectation & passage par valeur.** De par leur nature différente des types non structurés (comme `int`), on pourrait éventuellement supposer que la politique de passage des arguments de fonction de genre `struct` soit différente (c'est à dire, ne soit pas par valeur), mais il n'en est rien.

De même, on pourrait douter que l'affectation d'une variable de genre `struct` dans une autre de même type se passe différemment que pour les types de base non structurés, mais encore une fois il n'en est rien. Par exemple, le fragment suivant :

Fragment 4.8

```
void fun(void)
{
    struct s a;
    struct s b;
    // ...
    a = b;
}
```

est tout à fait légal, et copiera le contenu des champs de `b` dans les champs de `a`.

¶ **Absence de comparaison.** On ne peut pas comparer deux variables de genre `struct` avec les opérateurs usuels, même pour l'égalité ou la différence.

¶ **Manipulation des champs.** Les champs d'une variable de type enregistrement s'accèdent *via* la syntaxe :

Syntaxe 4.9

```
var.nom_champ
```

où `var` est le nom de la variable, et `nom_champ` le nom d'un champ valide pour le type de `var`, tel qu'il a été défini avec la Syntaxe 4.3. On illustre ceci avec le fragment suivant :

Fragment 4.10

```
struct gcof
{
    unsigned d;
    int u;
    int v;
};

struct gcof xgcd(unsigned a, unsigned b)
{
    struct gcof res;
    res.d = 0;
    res.u = 0;
    res.v = 0;
```



```
    // ...
    return res;
}
```

Si un champ est lui-même d'un type de genre `struct`, on itère simplement le procédé. Par exemple, pour utiliser le type défini dans le Fragment 4.5, on peut être amené à écrire `res.uv.x`.

¶ **Initialisation.** On peut initialiser une variable de type enregistrement à la déclaration en utilisant la syntaxe :

Syntaxe 4.11

```
struct ma_struct var = {.nom_champ1 = val1, .nom_champ2 = val2 /*...*/};
```

Il n'est pas *nécessaire* de spécifier les noms des champs de l'initialiseur dans le même ordre que la définition du type, mais ce n'est pas forcément une bonne idée d'utiliser des ordres différents.
Une initialisation peut également être *incomplète* et ne mentionner qu'un sous-ensemble (éventuellement vide) des champs. Si tel est le cas, les champs non mentionnés sont initialisés à zéro, ou ce qui s'en rapproche le plus pour le type du champ.
On peut par exemple réécrire le Fragment 4.10 pour expliciter l'initialisation de la variable `res` :

Fragment 4.12

```
struct gcof xgcd(unsigned a, unsigned b)
{
    struct gcof res = {.d = 0, .u = 0, .v = 0};
    // ...
    return res;
}
```

ou `struct gcof res = {};` pour une initialisation implicite à zéro, qui sera identique ici.

§ **Enregistrements & pointeurs**

Cette section fait appel à des notions décrites à la Section 5.

¶ **Définitions récursives.** Lors de sa définition, un type de genre `struct` peut inclure des champs dont le type est un *pointeur* vers le type en train d'être défini. Ceci se fait suivant la syntaxe habituelle de définition d'un enregistrement et des types pointeurs :

Fragment 4.13

```
struct node
{
    void *data;
    struct node *next;
};
```

Ce mécanisme permet de définir des types de données récursifs : dans l'exemple du fragment ci-dessus on peut (si l'on veut) définir une `struct node` qui contient des données et un pointeur vers une `struct node` qui contient des données et un pointeur vers une `struct node` qui contient des données et un pointeur vers une `struct node` qui contient des données et un pointeur vers une `struct node` qui contient des données et un pointeur vers une `struct node` qui…
La définition & manipulation de types récursifs reste cependant plus agréable dans un langage fonctionnel comme OCaml.

¶ **Accès aux champs depuis un pointeur vers enregistrement.** Manipuler des pointeurs vers des types de genre `struct` est chose assez courante en C, et le langage fournit une syntaxe allégée supplémentaire pour accéder aux champs d'un enregistrement pour lequel on possède seulement un pointeur :

Syntaxe 4.14

```
pointeur_vers_s->champ_de_s
```

Le sens de cette syntaxe est d'à la fois déréférencer le pointeur puis d'accéder au champ ; autrement dit, elle fait strictement la même chose que `*(pointeur_vers_s).champ_de_s`.

4.2 Tableaux

Un *tableau* (en anglais : *array*) est un type structuré qui permet de stocker plusieurs valeurs d'un **même** type ; un tableau de longueur fixe (en anglais : *fixed-length array* ou *FLA*) stocke un nombre fixe (…) d'éléments (sa *longueur*), dont la valeur *fait partie du type*. Ainsi, on peut par exemple définir un type `int [10]` des tableaux contenant 10 valeurs de type `int`.

§ **Tableaux à une dimension**

On commence par ne décrire que les tableaux *à une dimension*, c'est à dire les tableaux stockant des éléments qui ne sont pas des tableaux, et l'on peut alors considérer sans perte de généralité que ces éléments sont de type `int`.

¶ **Déclaration.** La déclaration d'une variable de type tableau se fait par la syntaxe :

Syntaxe 4.15

```
type var[longueur];
```

où type est le nom du type des éléments (par exemple `int`), `var` le nom de la variable, et `longueur` un *littéral* entier strictement positif ou techniquement, une expression entière constante.

Par exemple, le fragment suivant déclare une variable `t` du type `int [10]` des tableaux contenant 10 `int`s :

Fragment 4.16

```
int t[10];
```

¶ **Accès aux éléments.** L'accès aux éléments d'une variable de type tableau (pour lecture, écriture…) se fait par la syntaxe :

Syntaxe 4.17

```
var[i]
```

où `var` est le nom de la variable, et `i` une expression s'évaluant en un entier positif. Cet accès est valide ssi. `i` s'évalue en une valeur appartenant à l'intervalle $[\![0, n-1]\!]$ des entiers entre 0 et $n-1$, avec n la longueur (fixe) du type de `var`. Autrement dit, les n éléments d'un tableau `t` à n éléments s'accèdent comme `t[0]`, `t[1]`, …, `t[n-1]`.

Toute tentative d'accès à un élément hors de l'intervalle valide $[\![0, n-1]\!]$ est une erreur menant à un comportement non défini, *cf.* ARR30-C. Do not form or use out-of-bounds pointers or array subscripts.

¶ ***Sanitizers.*** Une erreur courante lors de l'utilisation de tableaux est de chercher à accéder à un hypothétique élément `t[n]` d'un tableau à n éléments, comme le fait par exemple le programme suivant :

Programme 4.18

```c
#include <stdlib.h>

int main(void)
{
    int t[10];
    t[10] = 0;

    return EXIT_SUCCESS;
}
```

En fonction de votre compilateur, l'exécution du programme résultant peut immédiatement déclencher une erreur, par exemple :

Terminal 4.19

```
> ./a.out
*** stack smashing detected ***: terminated
zsh: IOT instruction (core dumped)  ./a.out
```

On peut aussi obtenir un diagnostique plus précis (et plus systématique) en utilisant l'*address sanitizer*, ce qui se fait en compilant le programme avec l'option `-fsanitize=address` :

Terminal 4.20

```
> cc -fsanitize=address oob.c
> ./a.out
=================================================================
```

```
==68395==ERROR: AddressSanitizer: stack-buffer-overflow on address
0x7688bf200058 at pc 0x64f940465247 bp 0x7ffec27e7cd0 sp 0x7ffec27e7cc0
< message tronqué >
```

L'utilisation de l'*address sanitizer* est **très vivement recommandée** dès qu'un programme manipule des tableaux.

¶ **Initialisation.** On peut initialiser une variable de type tableau à la déclaration en utilisant la syntaxe :

Syntaxe 4.21

```
type var[n] = {val0, val1, /*...*/};
```

où `n` est un littéral entier strictement positif et `val0` etc. sont n expressions de type `type` qui initialisent dans l'ordre les n éléments du tableau. Le fragment suivant effectue une telle initialisation d'un tableau de type `int [10]` :

Fragment 4.22

```
int t[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Comme pour l'initialisation de `struct`s, cette initialisation peut être incomplète et n'initialiser que les premiers éléments ou certains éléments en particulier, ou peut être entièrement vide. Dans tous ces cas, les éléments non explicitement initialisés le sont à « zéro ».

¶ **(Non-)affectation.** On peut lire et écrire les éléments d'une variable de type tableau comme n'importe quelle variable habituelle, ce qu'on illustre dans le fragment suivant :

Fragment 4.23

```
int t[10];
int a
// ...
a = t[7];
t[3] = 1;
t[2] = t[0] + t[1];
```

Il est en revanche impossible d'affecter quoi que ce soit dans une variable de type tableau « toute entière » : le programme suivant ne compile pas :

Programme 4.24

```
#include <stdlib.h>

int main(void)
```

```c
{
    int t[5] = {0, 1, 2, 3, 4};
    int s[5];

    s = t; // impossible

    return EXIT_SUCCESS;
}
```

par exemple avec le message d'erreur :

Terminal 4.25

```
ar.c:8:7: error: array type 'int[5]' is not assignable
    8 |     s = t; // impossible
      |     ~ ^
1 error generated.
```

¶ **Passage d'arguments tableau.** Les mécanismes en jeu lors du passage d'un argument de type tableau à une fonction sont un peu compliqués, et nous les détaillerons dans la Section 5. Une conséquence est notamment que plusieurs syntaxes sont possibles pour écrire le type d'arguments tableaux ; nous n'en verrons pour l'instant que deux, en fonction de si la longueur du tableau doit être la même pour tous les arguments ou non. Dans le premier cas la syntaxe peut être essentiellement identique à celle de la déclaration, et pour par exemple indiquer un paramètre de type `int [10]` il suffit de faire :

Fragment 4.26

```c
void fun(int t[10]);
```

Le « 10 » ci-dessus est en fait surtout décoratif ; si l'on souhaite le rendre plus utile, on peut le faire précéder du mot-clef `static`, pour obtenir : `void fun(int t[static 10])`.
Dans le second cas, il **faut** fournir la longueur *via* un autre argument, car au sein de la fonction ayant pris le tableau comme argument il n'est plus possible de directement accéder à sa longueur. Ceci peut par exemple se faire comme :

Fragment 4.27

```c
void fun(size_t tn, int t[tn]);
```

où `size_t` est un type entier non signé servant spécifiquement à indiquer la taille des objets, et `tn` la longueur du tableau `t`. Attention : l'ordre des paramètres est important, et l'on ne peut pas les inverser (avoir `t` comme premier paramètre et `tn` comme second). Une invocation de cette fonction `fun` avec un argument `t` de longueur 10 se ferait alors comme :

Fragment 4.28

```
fun(10, t);
```

Une seconde conséquence est que même si les arguments de type tableau sont toujours passés par valeur, une modification d'un élément d'un argument tableau *sera visible en dehors de la fonction où cette modification a lieu*. On illustre ce phénomène avec le programme suivant :

Programme 4.29

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

void init10(int t[10])
{
    for (int i = 0; i < 10; i++)
    {
        t[i] = i;
    }
}

int main(void)
{
    int t[10];

    init10(t);
    for (int i = 0; i < 10; i++)
    {
        printf("%d ", t[i]);
    }
    printf("\n");

    return EXIT_SUCCESS;
}
```

dont l'exécution affiche (en toute légalité) :

Terminal 4.30

```
> ./a.out
0 1 2 3 4 5 6 7 8 9
```

¶ **Limitations.** Toujours pour des raisons qui seront détaillées à la Section 7, l'utilisation de tableau est soumise à deux limitations : la taille maximale des tableaux pouvant être utilisés est en pratique relativement petite, et il est illégal pour une fonction de renvoyer un tableau déclaré dans son corps. On illustre ces deux limitations avec les programmes suivants :

Programme 4.31

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

int main(void)
{
    int t[2100000];
    t[0] = 0;

    return EXIT_SUCCESS;
}
```

dont l'exécution peut produire :

Terminal 4.32

```
> ./a.out
zsh: segmentation fault (core dumped)  ./a.out
```

et :

Programme 4.33

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

int *alloc10(void)
{
    int t[10];

    return t;
}

int main(void)
{
   int *t = alloc10();
   t[0] = 0;

   return EXIT_SUCCESS;
}
```

(dont la syntaxe de certaines constructions sera expliquée à la Section 5) qui si compilé avec l'*address sanitizer* peut produire :

Terminal 4.34

```
> ./a.out
==84888==ERROR: AddressSanitizer: stack-use-after-return on address
0x71544da00020 at pc 0x57b0a0c81dad bp 0x7ffe40a95b30 sp 0x7ffe40a95b28
< message tronqué >
```

§ **Tableaux multidimensionnels**

Later

5 Type pointeur

En tant que langage relativement « bas niveau », le C permet de manipuler directement les *adresses mémoires* des objets qui en possèdent. Nous verrons plus en détails dans la Section 7 comment s'organise cette mémoire, et nous contentons pour l'instant de décrire les types *pointeurs*, qui permettent cette manipulation d'adresses.

La manipulation directe d'adresse est source fréquente de bugs, dont le diagnostique est grandement facilité par l'utilisation de l'*address sanitizer*.

§ **Définition & utilisation**

On peut *dériver* un type pointeur depuis les types de base (par exemple les types entiers) et les types structurés eux-mêmes construits sur ces types de base. Le type pointeur associé à un type nommé `type` se note `type *` ou `type*`, et la déclaration d'une variable de type pointeur se fait en conséquence suivant la syntaxe usuelle, soit :

Syntaxe 5.1

```
type *p;
type* q;
```

Le programme favorise la seconde syntaxe « `type* q` » à la première, mais les deux sont autorisées et la première est peut-être plus fréquente *en général*. La syntaxe `type * r` est également *possible*, mais plus rare.

Il est possible de dériver un pointeur depuis un type pointeur lui-même, ce qui est occasionnellement utile ; on parle alors généralement de « pointeur de pointeur ». Le fragment de code suivant illustre la déclaration de variables de types pointeurs variés :

Fragment 5.2

```
int *p1; // pointeur vers un int
int **p2; // pointeur vers un (pointeur vers un int)
struct gcof *p3; // pointeur vers une struct gcof
```

¶ **Initialisation.** Comme en général, une variable de type pointeur doit être initialisée à *quelque chose* avant de pouvoir être utilisée (rappel : EXP33-C. Do not read uninitialized memory). Dans le cas des

pointeurs, ceci se fait typiquement *via* une *allocation* mémoire, *cf.* Section 6, ou (plus rarement) *via* l'opérateur « *address of* » ou une valeur « nulle » par défaut. *cf.* ci-dessous.

¶ **Pointeur nul.** Il existe une valeur spéciale de *pointeur nul*, que peut prendre n'importe quelle expression de type pointeur (quelque soit le type exact). Cette valeur sert à dénoter un pointeur ne pointant « vers rien », c'est à dire vers aucune zone mémoire, et il sera illégal de le déréférencer (*cf.* ci-dessous). Il y a plusieurs façons d'écrire cette valeur spéciale, la plus courante (pour des raisons historiques) est `NULL`, mais il est préférable d'utiliser `nullptr` (à condition d'utiliser un compilateur suffisamment récent). Le fragment de code suivant définit deux variables de type pointeurs différents, chacune initialisée au pointeur nul :

Fragment 5.3

```
int *p = nullptr;
struct gcof *q = nullptr;
```

¶ **Pointeurs vers `void`.** Il existe un type pointeur particulier « `void *` » des pointeurs vers le type `void`. Celui-ci sert de type pointeur « universel » ou « générique », et s'utilise quand le type des objets pointés est non défini. Tout objet de type pointeur quelconque peut être *converti* vers un objet de type `void *` et vice-versa ; cette conversion n'a pas (toujours) besoin d'être explicite. Dans l'exemple suivant, la fonction `malloc` (que nous verrons en détails dans la Section 6) a signature `void *malloc(size_t size)` et elle est utilisée en toute légalité pour initialiser la valeur d'un pointeur de type `uint64_t *` ; la fonction `free` (que nous verrons également à la Section 6) a pour signature `void free(void *p)`, et l'on peut en toute légalité l'utiliser avec un argument de type `uint64_t *` :

Programme 5.4

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

int main(void)
{
    uint64_t *p = malloc(8);
    free(p);

    return EXIT_SUCCESS;
}
```

Le type `void` ne possédant aucune valeur (il n'est pas *habité*), *déréférencer* (*cf.* ci-dessous) un pointeur vers `void` est vide de sens et illégal. On peut cependant (comme déjà évoqué à la Section 3.14) convertir un pointeur vers `void` depuis et vers un pointeur vers un type quelconque, et vice-versa.

¶ **Affichage.** On peut afficher la valeur d'un pointeur de type `void *` avec `printf` et autres fonctions similaires grâce à la spécifications de conversion %p. Si le pointeur a été initialisée de façon non triviale, cette valeur est une adresse mémoire, affichée dans un format dépendant de l'architecture :

Programme 5.5

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

int main(void)
{
    uint64_t *p = malloc(8);
    int *q = nullptr;

    printf("p:%p\nq:%p\n", (void *)p, (void *)q);
    free(p);

    return EXIT_SUCCESS;
}
```

(où l'on convertit explicitement le type des pointeurs) produit par exemple la sortie :

Terminal 5.6

```
p:0x562712f81310
q:(nil)
```

¶ **Opérateur de déréférencement `*`.** Un objet de type pointeur `type *` sert à *référencer* une zone mémoire qui (si correctement initialisée) contient un élément du type `type`. Étant donné un pointeur vers une telle zone, il est alors possible de lire ou écrire *dans la zone mémoire pointée*. Ceci se fait en *déréférençant* le pointeur, ce qui se fait avec la syntaxe :

Syntaxe 5.7

```
*p
```

Ici, `*` est un opérateur (unaire, en notation préfixe) de *déréférencement* ou d'*indirection*, qu'il ne faut pas confondre avec `*` qui est un opérateur (binaire, en notation infixe) de multiplication.

Soit `x` une opérande de type `type *`, l'expression `*x` obtenue en lui appliquant l'opérateur de déréférencement `*` est de type `type` (on a « retiré l'`*` » du type ; par exemple, si le type de `x` est `int *`, alors `*x` sera de type `int` ; si elle est de type `int **`, alors `*x` sera de type `int *` et `**x` de type `int`) et peut être utilisée « comme une variable » pour lire ou écrire (éventuellement avec certaines restrictions) dans la zone mémoire égale (au résultat de l'évaluation de) `x`. Autrement dit, `*x` est une *lvalue*.

Dans le contexte des CPGE, les opérandes seront généralement de forme très simple. Le fragment de code suivant donne quelques exemples :

Fragment 5.8

```c
int *p;
int a;
// ...
*p = 12; // on écrit 12 dans   la zone mémoire pointée par p
a = *p; // on lit la valeur de la -
p = 13; // erreur de type
```

Une erreur particulièrement commune lors de l'utilisation de pointeurs est de chercher à déréférencer un pointeur invalide, par exemple un pointeur nul, ce qui (sans surprise) est interdit, *cf.* EXP34-C. Do not dereference null pointers. Le programme suivant et son exécution illustrent ce fait :

Programme 5.9

```c
#include <stdlib.h>

int main(void)
{
    int *p = nullptr;
    int a = *p;

    return EXIT_SUCCESS;
}
```

Terminal 5.10

```
> clang -std=c23 ip.c
> ./a.out
zsh: segmentation fault (core dumped)  ./a.out
> clang -std=c23 -fsanitize=address ip.c
> ./a.out
AddressSanitizer:DEADLYSIGNAL
=================================================================
==29153==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000
(pc 0x55c7549c56b0 bp 0x7fff3069d050 sp 0x7fff3069d030 T0)
==29153==The signal is caused by a READ memory access.
==29153==Hint: address points to the zero page.
< message tronqué >
```

¶ **Opérateur d'adresse &.** Il est possible d'obtenir l'adresse mémoire d'un objet qui en possède une (typiquement vrai pour une variable, mais pas pour une constante littérale) *via* la syntaxe :

Syntaxe 5.11

```
&x
```

Ici, `&` est l'opérateur (unaire, en notation préfixe) « *address of* », qui n'a aucun lien avec `&` qui est l'opérateur (binaire, en notation infixe) de ET logique bit-à-bit (*cf.* Section 10.3).

Soit `x` une opérande de type `type`, l'expression `&x` obtenue en lui appliquant l'opérateur *address of* est de type `type *` (on a « ajouté une `*` » au type). Cet opérateur est essentiellement un inverse de l'opérateur de déréférencement dans le sens où les expressions `&(*x)` et `*(&x)` sont toutes deux équivalentes à `x`, à condition qu'elles soient légales (ce qui dépend de la nature de `x`, *cf.* ci-dessus).

Le fragment ci-dessous donne un exemple (assez typique) d'utilisation :

Fragment 5.12

```
int a;
int *p = &a; // pointeur initialisé avec l'adresse de la variable a
```

Il peut être occasionnellement utile (?) d'obtenir l'adresse d'un élément d'un tableau ; l'opérateur `&` fonctionne normalement dans ce cas :

Fragment 5.13

```
int t[12] = {};
int *p = &t[3]; // pointeur initialisé avec l'@ du 4ième élément de t
```

¶ **Passage d'arguments de type pointeur.** Les arguments de fonction de type pointeur sont (encore une fois) passés par valeur. Le mécanisme même des pointeurs permet cependant d'implémenter à la main un passage par *référence*, qui signifie à peu près que quand l'argument de la fonction possède une adresse (par exemple une variable, sauf cas particulier), l'argument manipulé à l'intérieur de la fonction est physiquement identique à celui fourni à l'appel, et une modification du premier sera visible sur le second. On illustre ceci avec le programme suivant, qui échange la valeur de deux variables :

Programme 5.14

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

void swap(int *x, int *y)
{
    int t = *x;
    *x = *y;
    *y = t;
}

int main(void)
```

```c
{
    int a = 3;
    int b = 12;

    swap(&a, &b);
    printf("a:%d\nb:%d\n", a, b);
}
```

dont l'exécution produit :

Terminal 5.15

```
a:12
b:3
```

§ **Arithmétique de pointeur**

Nous verrons à la Section 6 qu'il est courant qu'un pointeur contienne l'adresse du *début d'une grande zone mémoire* permettant de stocker plusieurs valeurs du type pointé. Si `p` est un tel pointeur, on peut accéder au *premier* élément de cette zone en déréférençant `p`, mais un mécanisme supplémentaire est nécessaire pour accéder au second, au troisième, etc. Ceci peut se faire par *arithmétique de pointeur*, qui consiste à (typiquement) ajouter des valeurs entières aux pointeurs suivant la syntaxe :

Syntaxe 5.16

```
p + i
```

avec `i` une expression entière (de valeur typiquement positive).

Si `p` est un pointeur vers un certain type de valeur une zone mémoire permettant de stocker n objets *du même type*, le premier (ou zéroième) est à l'adresse `p`, le second (ou unième) à l'adresse `p + 1`,..., le dernier à l'adresse `p + (n - 1)`. L'arithmétique de pointeur peut (comme toute manipulation des pointeurs) être source de bugs, *cf.* notamment EXP08-C. Ensure pointer arithmetic is used correctly ; elle n'est **pas au programme des CPGE** et l'on s'abstiendra d'y recourir hors cas raisonnables, pour lui préférer la syntaxe suivante :

¶ **Syntaxe à la tableaux.** Si `p` est comme ci-dessus, on peut manipuler (lire, affecter) les n éléments de la zone mémoire comme si `p` était de type tableau, c'est à dire le premier *via* la *lvalue* `p[0]`, le second *via* `p[1]`,..., le dernier *via* `p[n - 1]`.

§ **Dégradation des arguments tableaux**

Comme déjà mentionné à la Section 4.2, les arguments tableaux des fonctions sont passés par valeurs mais de façon « un peu compliquée » : au moment de son passage, un argument de type tableau (d'éléments de type `type`) voit son type se *dégrader* en celui `type *` des pointeurs vers `type`, de valeur l'adresse du tableau. Ceci explique les phénomènes décrits à la Section 4.2, et notamment le fait qu'une modification faite aux éléments d'un tableau passé comme argument d'une fonction persistent au delà de celle-ci : en effet, une telle modification (disons `t[3] = 12`) modifie la zone mémoire d'adresse `&t[3]`, qui n'est autre que l'adresse du quatrième élément du tableau initial.

Ce processus de dégradation peut se visualiser en utilisant la fonctionnalité `whatis` du débuggueur *gdb*, qui donne le type d'une expression. On incite vivement le lecteur ou la lectrice à tester ceci par elle-même avec le programme suivant :

Programme 5.17

```c
#include <stdlib.h>

void dummy(int t[12])
{
    return;
}

int main(void)
{
    int t[12];

    dummy(t);

    return EXIT_SUCCESS;
}
```

où l'on obtient :

Débuggueur 5.18

```
Reading symbols from ./a.out...
(gdb) break main
Breakpoint 1 at 0x1170: file tab.c, line 12.
(gdb) run
Starting program: /tmp/a.out
Breakpoint 1, main () at tab.c:12
12      dummy(t);
(gdb) whatis t
type = int [12]
(gdb) s
dummy (t=0x7fffffffe550) at tab.c:5
5       return;
(gdb) whatis t
type = int *
(gdb) s
main () at tab.c:14
14      return EXIT_SUCCESS;
(gdb) whatis t
type = int [12]
(gdb) s
```

```
[Inferior 1 (process 24069) exited normally]
(gdb) quit
```

§ **Qualification** `const`

Le contenu de cette section est en marge, voire hors du programme des CPGE.

Il est possible de qualifier `const` un type pointeur, notamment en tant que paramètre de fonction, suivant deux syntaxes (dont l'une admet une variante), qui peuvent se combiner :

Syntaxe 5.19

```
const type *p
type const *p // équivalente à la première
type * const p
const type * const p
```

Nous ne décrirons que brièvement la première, qui est la plus souvent utile : lorsqu'un pointeur est de type qualifié `const type *`, on ne peut utiliser un déréférencement que pour *lire* à l'adresse pointée, et non pas y écrire. Ajouter cette contrainte peut être souhaitable (et fournit une documentation utile) pour les paramètres pointeur (ou tableau, puisque cela est équivalent une fois rentré dans la fonction) de fonction lorsque celle-ci ne doit pas modifier la zone mémoire référencée par le pointeur (ou ne la modifie pas de fait). Autrement dit, cette qualification `const` interdit les *effets de bord* rendus possibles par les arguments pointeurs (contrainte ou précision qui est tout à fait inutile pour des paramètres de types arithmétiques, par exemple). Par exemple, une signature possible de la fonction `memcpy`, qui copie n octets d'une zone mémoire *src* vers une zone mémoire *dest* est :

Fragment 5.20

```
void *memcpy(void *dest, const void *src, size_t n)
```

Il n'y a aucune raison de devoir modifier la zone pointée par `src`, qui peut donc être de type qualifié `const`, contrairement à la zone pointée par `dest`, puisque l'on cherche précisément à y écrire le contenu de la première. Une signature moderne de `memcpy` inclura également une qualification `restrict` pour chacun de ses paramètres de type pointeur, qui en substance sert à indiquer que les deux zones mémoires ne doivent pas se chevaucher.

6 Stockage & allocation mémoire

La section précédente a introduit les pointeurs, qui permettent d'accéder aux objets à partir de leur adresse ; nous nous intéressons maintenant notamment à la légalité de ces accès, et à l'une de leurs utilisations principales, *viz.* accéder à une zone mémoire *allouée dynamiquement*.

6.1 Durée de vie des objets

De la même façon que les identifiants ont une portée (*cf.* Section 3.10), en C les objets ont une *durée de vie* (en anglais : *lifetime*) qui détermine à quel moment il est possible d'y accéder (en lecture ou

écriture) lors de l'exécution d'un programme. Référencer un objet en dehors de sa durée de vie est un comportement non défini.

Les notions de portée et de durée de vie possèdent certaines similarités mais sont nettement différentes : ce n'est pas parce qu'un objet est identifiable qu'il est légal d'y accéder, et ce n'est pas parce qu'il ne l'est plus qu'il a forcément cessé d'exister. Ce dernier cas est même (malheureusement) courant en pratique, et correspond généralement à une *fuite mémoire* (en anglais : *memory leak*). Une fuite mémoire n'est pas *directement* problématique dans le sens où le comportement du programme reste parfaitement défini, mais elle peut amener à gâcher des ressources physiques et ainsi nuire aux performances globales du système où le programme s'exécute. Éviter les fuites mémoires fait partie des problèmes important en développement logiciel.

La durée de vie d'un objet est déterminée par sa *durée de stockage* (en anglais : *storage duration*) ; nous décrivons les trois les plus courantes, que l'on s'efforcera d'utiliser à bon escient, *cf.* DCL30-C. Declare objects with appropriate storage durations.

¶ **Durée de stockage *statique*.** La durée de stockage « statique » (en anglais : *static storage duration*) est typiquement celle des variables globales et des variables locales déclarées avec le spécifieur `static`. La durée de vie des objets de cette durée de stockage est simplement toute la durée d'exécution du programme.

¶ **Durée de stockage *automatique*.** La durée de stockage « automatique » (en anglais : *automatic storage duration*) est typiquement celle des variable-paramètres des fonctions et des variables locales qui ne sont pas déclarées avec le spécifieur `static`. Hors cas particulier, la durée de vie des objets de cette durée de stockage coïncide avec la *durée d'exécution* du bloc correspondant ; celle-ci commence lorsque l'on entre dans le bloc et se termine lorsque l'on en sort définitivement, mais *pas* par exemple si une fonction est appelée depuis le bloc. Ceci explique notamment pourquoi il est parfaitement légal d'accéder au contenu d'un tableau passé en argument d'une fonction (comme déjà illustré par le Programme 4.29) mais *pas* au contenu d'un tableau « renvoyé » par une fonction (comme précédemment illustré par le Programme 4.33) ; ce dernier type d'erreur est généralement surnommé *use after return*. L'accès effectué par le programme ci-dessous est également illégal pour la même raison d'utilisation au delà de la durée de vie :

Programme 6.1

```c
#include <stdlib.h>

int main(void)
{
    int *p;
    {
        int a;
        p = &a;
    }
    *p = 12;

    return EXIT_SUCCESS;
}
```

Son exécution sans *sanitizer* ne semble pas soulever de problème (contrairement à sa compilation avec

gcc), mais l'exécution *avec* produira typiquement :

Terminal 6.2

```
==58649==ERROR: AddressSanitizer: stack-use-after-scope on address
0x7bd154700020 at pc 0x5577e29ff765 bp 0x7ffd4f45d150 sp 0x7ffd4f45d148
WRITE of size 4 at 0x7bd154700020 thread T0
<message tronqué>
```

ce qui permet au passage de remarquer que le surnom habituel donné à ce type d'erreur (et en fait plutôt mal adapté) est *use after scope*.

¶ **Durée de stockage *allouée*.** La durée de stockage « allouée » (en anglais : *allocated storage duration*) est typiquement celle des objets alloués par l'une des fonctions d'allocation dynamique de la mémoire (*cf.* Section 6.2 ci-dessous). La durée de vie des objets de cette durée de stockage s'étend de l'allocation (par exemple effectuée avec `malloc`) à la désallocation (ou libération ; par exemple effectuée avec `free`).

Cette durée de stockage est adaptée aux objets dont on souhaite que la durée de vie soit plus longue qu'avec un stockage automatique mais ou bien plus courte qu'avec un stockage statique, ou bien telle que leur taille est inconnue à l'écriture du programme (et ne permet donc pas la déclaration de variables globales de taille appropriée). Pour des raisons indépendantes du langage mais dépendantes des environnement d'exécution typiques décrits à la Section 7, elle est également couramment employée pour des objets dont on aimerait qu'ils aient une durée de stockage automatique, mais concrètement trop gros étant donné comment ce dernier est implémenté.

6.2 Allocation dynamique de la mémoire

En C, l'*allocation dynamique* de la mémoire fait typiquement mais pas exclusivement référence aux objets de durée de stockage allouée. Le caractère *dynamique* vient du fait que ces derniers ne sont pas *statiquement* stockés pendant l'intégralité de l'exécution du programme, et que contrairement aux objets de durée de stockage automatique, leur durée de vie est contrôlée finement et explicitement par les fonctions de gestion de la mémoire. Il existe plusieurs variantes de ces fonctions, mais les deux seules au programme (et les deux seules que nous décriront) sont `malloc` (pour l'allocation) et `free` (pour la libération) ; les alternatives courantes sont `calloc` (allocation & initialisation) et `realloc` (allocation & libération).

La gestion *manuelle* de l'allocation dynamique de la mémoire est une source interminable de bugs (dont on décrit un certain nombre de catégories ci-dessous). Il est donc particulièrement important de tester avec précaution les parties des programmes utilisant ces fonctions, et d'utiliser des outils de diagnostiques comme l'*address sanitizer*. Une bonne discipline de programmation peut également aider à tout simplement limiter leur risque d'occurrence ; par exemple, une bonne pratique consiste à clairement définir *qui* est responsable de l'allocation et de la libération de la mémoire, et à faire cela de façon cohérente au sein d'un même programme, *cf.* MEM00-C. Allocate and free memory in the same module, at the same level of abstraction.

§ `malloc`

La fonction de bibliothèque standard `malloc`, déclarée dans le fichier d'entête `stdlib.h` et de signature `void *malloc(size_t size)` alloue `size` **octets** de mémoire *non initialisée* de durée de stockage allouée et renvoie un pointeur vers le *début* de la zone allouée, ou `nullptr` si l'allocation

échoue. Le contenu de la zone mémoire s'accède *via* le pointeur renvoyé comme décrit à la Section 5 précédente, y compris dans le cas où cette zone est utilisée pour stocker *plusieurs* objets du type pointé.

Lors d'une utilisation de `malloc`, il est évidemment important d'allouer la bonne quantité de mémoire : une allocation trop petite risque d'entraîner un accès hors de la zone allouée, ce qui est un comportement non défini "if an *lvalue* does not designate an object when it is evaluated, the behavior is undefined", *cf.* MEM35-C. Allocate sufficient memory for an object

S'il est aisé de déterminer la quantité nécessaire pour stocker des objets dont les tailles de représentations sont fixes (par exemple un entier non signé de type `uint64_t`, représenté sur huit octets), cela est moins simple pour d'autres types comme `int`, ou surtout les types définis par l'utilisateur ou l'utilisatrice, notamment ceux de genre `struct`. Afin de résoudre ce problème, le langage C fournit un opérateur (ce n'est *pas* une fonction) `sizeof` tel que `sizeof(type)` s'évalue en le nombre d'octets nécessaires au stockage d'un objet du type `type`. Il est également possible d'utiliser `sizeof` avec une *variable* pour opérande, et le résultat de l'évaluation sera alors le nombre d'octets nécessaire au stockage d'un objet du même type que la variable. Le fragment de code suivant illustre une utilisation typique de `malloc` en conjonction avec `sizeof` :

Fragment 6.3

```
int *p = malloc(sizeof(int)); // allocation pour stocker un int
int *q = malloc(10 * sizeof(int)); // - dix -
```

Le type de renvoi de `malloc` est un pointeur vers `void *`, qui comme mentionné à la section précédente et à la Section 3.14 peut être converti vers n'importe quel type pointeur (vers un type objet). Cette conversion peut-être effectuée explicitement mais ce n'est pas nécessaire : les deux utilisations dans le fragment ci-dessous sont légales :

Fragment 6.4

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

§ `free`

La fonction de bibliothèque standard `free`, déclarée dans le fichier d'entête `stdlib.h` et de signature `void free(void *p)` prend en entrée un pointeur *qui doit avoir été précédemment renvoyé par* `malloc` (ou une fonction similaire) *et qui n'a pas déjà été désalloué* et libère (désalloue) la mémoire qui avait été allouée à cet appel (mettant ainsi fin à la durée de vie de l'objet correspondant). L'argument peut aussi éventuellement être un pointeur nul, et dans ce cas `free` ne fait rien.

On peut remarquer qu'il n'est pas nécessaire lors d'un appel à `free` d'indiquer la taille de la zone mémoire devant être libérée ; connaître cette information est de la responsabilité des fonctions d'allocation & libération mémoire elles-mêmes.

Une fois la mémoire désallouée par `free`, il devient illégal de même seulement *lire* la valeur du pointeur ayant été fourni en argument. Une bonne pratique (*cf.* MEM01-C. Store a new value in pointers immediately after free(), mais celle-ci ne fait pas l'unanimité) consiste ainsi à immédiatement réécrire celui-ci (typiquement avec une valeur de pointeur nul) comme dans le fragment ci-dessous :

Fragment 6.5

```
free(p);
p = nullptr;
```

Comme mentionné plus haut, la libération de la mémoire inutilisée est importante, *cf.* MEM31-C. Free dynamically allocated memory when no longer needed.

§ **Fuites mémoires**

Une *fuite mémoire* désigne la présence d'objets de durée de stockage allouée encore vivants qu'on ne peut plus référencer syntaxiquement : on ne peut plus utiliser les objets *ni libérer la mémoire qu'ils occupent*. Le programme ci-dessous illustre deux des mécanismes typiques à l'origine de fuites mémoires : la fin de l'exécution de la fonction `fun` met également fin à la portée syntaxique de la seule variable permettant d'accéder à l'objet alloué, et il n'existe alors plus de variable permettant de référencer ce dernier ; dans la fonction `main`, on écrase la valeur du pointeur permettant de référencer un objet de durée de vie allouée, ce qui encore une fois empêchera d'y accéder (sans même attendre la fin de portée du pointeur).

Programme 6.6

```
#include <stdlib.h>

void fun(void)
{
    int *p = malloc(sizeof(int));
}

int main(void)
{
    fun();
    int *p = malloc(sizeof(int));
    p = nullptr;

    return EXIT_SUCCESS;
}
```

L'utilisation de l'*address sanitizer* permet de détecter un grand nombre de fuites mémoires (sans garantie de les trouver toutes…). L'exécution du programme ci-dessus produit :

Terminal 6.7

```
==15496==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 4 byte(s) in 1 object(s) allocated from:
[...]
```

```
Direct leak of 4 byte(s) in 1 object(s) allocated from:
[...]

SUMMARY: AddressSanitizer: 8 byte(s) leaked in 2 allocation(s).
```

D'autres outils comme valgrind permettent de détecter efficacement un grand nombre de fuites mémoires (et dans le cas de ce dernier, d'autres types d'erreurs liées à la manipulation de la mémoire).

Un autre mécanisme courant à l'origine de fuites mémoire est la libération trop précoce d'un objet contenant lui-même l'unique référence vers un autre objet alloué. Ce cas est par exemple fréquent lors de la manipulation de chaînes de variables de genre `struct` allouées séparément ; on l'illustre avec le programme (plus simple) ci-dessous :

Programme 6.8

```c
#include <stdlib.h>

struct dbint
{
    int *p;
};

int main(void)
{
    struct dbint *dbi = malloc(sizeof(struct dbint));
    dbi->p = malloc(sizeof(int));
    *(dbi->p) = 12;
    // free(dbi->p); nécessaire
    free(dbi);

    return EXIT_SUCCESS;
}
```

Dans ce cas, la fuite mémoire est causée par la non désallocation de la mémoire accessible *via* le pointeur « interne » `dbi->p` ; comme pour le programme précédent, elle est détectée par l'*address sanitizer*.

La *prévention* de fuite mémoire se fait entre autres à travers une politique rigoureuse d'allocation et de désallocation ; notamment, il est important de réfléchir pour chaque allocation à l'endroit (et le moment) où la libération correspondante sera effectuée, et les désallocations en chaîne doivent être faites dans le bon ordre. Tout ceci doit être pensé au plus tôt lors de la conception d'un programme, par exemple en écrivant des fonctions dédiées d'allocation & libération pour les objets pour lesquels celles-ci sont complexes ; le programme ci-dessus aurait par exemple pu bénéficier de telles fonctions.

On peut également voir l'écriture et l'exécution de tests (sous *sanitizer*, avec *valgrind*...) comme des mesures préventives, qui permettent de détecter (et aider à corriger !) les fuites ; comme toujours, tester régulièrement les programmes dès la phase d'écriture permet de limiter les bugs dans une version « finale ». Dans une certaine mesure, ces outils permettent également de valider l'absence de fuites.

§ **Bestiaire**

L'utilisation conjointe de `malloc` et `free` peut causer de nombreux bugs ~~tous plus drôles les uns que les autres~~ dont les suivants sont parmi les plus courants.

¶ **Dépassement mémoire.** Une erreur classique consiste à chercher à accéder à une zone mémoire qui n'a pas été allouée ; ceci peut se produire suite à des erreurs de manipulation d'indices (comme lors de l'utilisation de tableaux), illustrées au Programme 7.8, ou de façon plus pernicieuse parce que l'on n'a pas alloué suffisamment de mémoire (par exemple suite à un oubli ou une mauvaise utilisation de `sizeof`). Le programme ci-dessous illustre ce second scenario :

Programme 6.9

```c
#include <stdlib.h>

int main(void)
{
    int *p = malloc(1);
    *p = 12;

    return EXIT_SUCCESS;
}
```

Son exécution avec l'*address sanitizer* produit :

Terminal 6.10

```
=================================================================
==11184==ERROR: AddressSanitizer: heap-buffer-overflow on address
0x7b28ff1e0010 at pc 0x56478c2f86b2 bp 0x7ffdfedfae90 sp 0x7ffdfedfae88
WRITE of size 4 at 0x7b28ff1e0010 thread T0
[...]
0x7b28ff1e0011 is located 0 bytes after 1-byte region
<message tronqué>
```

¶ ***Use after free.*** Il est illégal d'accéder à une zone mémoire ayant été libérée par `free`, *cf.* MEM30-C. Do not access freed memory. Le programme suivant et son exécution avec l'*address sanitizer* illustrent ce type d'erreur :

Programme 6.11

```c
#include <stdlib.h>

int main(void)
{
    int *p = malloc(sizeof(int));
```

```c
    free(p);
    *p = 12;

    return EXIT_SUCCESS;
}
```

Terminal 6.12

```
> ./a.out
=================================================================
==90767==ERROR: AddressSanitizer: heap-use-after-free on address
0x7bb04e7e0010 at pc 0x560ee6c641d2 bp 0x7ffdb1972410 sp 0x7ffdb1972400
WRITE of size 4 at 0x7bb04e7e0010 thread T0
<message tronqué>
```

¶ ***Double free.*** Il est également illégal d'appeler `free` avec un pointeur vers une zone ayant *déjà* été libérée, comme illustré par le programme suivant :

Programme 6.13

```c
#include <stdlib.h>

int main(void)
{
    int *p = malloc(sizeof(int));
    free(p);
    free(p);

    return EXIT_SUCCESS;
}
```

dont l'exécution avec l'*address sanitizer* produit :

Terminal 6.14

```
> ./a.out
=================================================================
==92000==ERROR: AddressSanitizer: attempting double-free on
0x7b9a181e0010 in thread T0:
<message tronqué>
```

¶ **Libération de mémoire non allouée avec `malloc`.** Comme dit ci-dessus, la fonction `free` peut seulement être utilisée pour libérer de la mémoire allouée avec `malloc` & *friends*, *cf.* également MEM34-C. Only free memory allocated dynamically. Le programme suivant illustre cette dernière erreur :

Programme 6.15

```c
#include <stdlib.h>

int main(void)
{
    int a;
    int *p = &a;
    free(p);

    return EXIT_SUCCESS;
}
```

via son exécution, toujours avec l'*address sanitizer* :

Terminal 6.16

```
> ./a.out
=================================================================
==92608==ERROR: AddressSanitizer: attempting free on address
which was not malloc()-ed: 0x7b6f18500020 in thread T0
<message tronqué>
```

7 Organisation mémoire

Cette section ne porte pas sur le langage C lui-même, mais sur la gestion et l'organisation mémoire des *processus* (un programme en cours d'exécution, du point de vue du système d'exploitation) telle que typiquement effectuée par les systèmes d'exploitation, notamment UNIX. Nous ne ferons que survoler le sujet, ce qui sera néanmoins suffisant pour décrire certains phénomènes & limitations *généralement* vrais pour tous les programmes, indépendamment du langage de programmation utilisé pour les écrire ; tous les exemples seront toutefois illustrés par des programmes C.

7.1 Mémoire virtuelle

Les adresses mémoires visibles & manipulables par les programmes (typiquement *via* des variables de type pointeur) sont *virtuelles* : elles ne correspondent pas à des emplacements physiques dans la mémoire (typiquement à accès aléatoire ou « vive » ; en anglais : *random-access memory*, ou *RAM*), mais à des identifiants abstraits qui (si valides) pourront être traduits par le système d'exploitation en des adresses *physiques*.

De cette façon, tous les processus s'exécutant sur une machine ont le même *espace d'adressage* (virtuel), c'est à dire les mêmes valeurs possibles pour leurs adresses (virtuelles). Sur une architecture moderne 64-bit, cet espace est typiquement défini par l'ensemble des valeurs que peut prendre un entier non signé de 64 bits, et est donc de taille 2^{64}. La mémoire étant typiquement adressée à la granularité de l'octet, un processus peut alors adresser (environ) 2^{64} octets, soit 2^{32} Go. Il s'ensuit que même sur une machine généreusement dotée en mémoire vive (disons 16 To, soit 2^{12} Go), les adresses virtuelles peuvent être

plusieurs millions *de fois* plus nombreuses que les adresses physiques. Autrement dit, la très grande majorité des adresses virtuelles manipulables par un programme ne correspondent à aucune adresse physique.

L'utilisation d'un espace mémoire virtuel aide à la réalisation d'une des tâches essentielles du système d'exploitation, *viz.* garantir l'*isolation* des différents processus s'exécutant sur une même machine. Pour des raisons évidentes (?) de bon fonctionnement et de sécurité, il n'est en effet pas souhaitable qu'un processus puisse lire ou écrire la mémoire d'un autre processus.

§ **Cartographie**

L'espace mémoire virtuel est typiquement constitué de plusieurs zones. Les adresses *hautes* (les plus grandes, quand interprétées comme un entier) sont réservées à l'usage du noyau (en anglais : *kernel*) du système d'exploitation. Une première zone sous ces adresses est occupée par la *pile d'exécution* (en anglais : *stack*), puis l'on trouve (toujours en descendant) le *tas* (en anglais : *heap*) et enfin les données globales (les variables globales et certaines constantes littérales) et le code machine du programme lui-même. Certaines de ces zones (typiquement celle contenant le code du programme, pour des raisons encore une fois évidentes (?) de sécurité) sont accessibles uniquement en lecture, et toute tentative d'écriture amènera le système d'exploitation à déclencher une erreur.

Le programme ci-dessous permet de visualiser certaines de ces zones, dont on discute des rôles plus en détail ci-après.

Programme 7.1

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

int glo0;
int glo1 = 1;
const int glo2 = 2;
int glo3 = 3;

void dum(void)
{
    return;
}

int main(void)
{
    int a;
    int b;
    int t[10];
    int *p = malloc(sizeof(int));
    int *q = malloc(10 * sizeof(int));
    int *r = malloc(10 * sizeof(int));

    puts("~~stack zone~~");
    printf("@a : %p\n", (void*)&a);
```



```c
    printf("@b : %p\n", (void*)&b);
    printf("@t : %p\n", (void*)t);
    printf("@p : %p\n", (void*)&p);
    printf("@q : %p\n", (void*)&q);
    printf("@r : %p\n\n", (void*)&r);
    puts("~~heap zone~~");
    printf("@*p: %p\n", (void*)p);
    printf("@*q: %p\n", (void*)q);
    printf("@*r: %p\n\n", (void*)r);
    puts("~~data zone~~");
    printf("@glo0: %p\n", (void*)&glo0);
    printf("@glo1: %p\n", (void*)&glo1);
    printf("@glo3: %p\n", (void*)&glo3);
    puts("~~(read-only part)~~");
    printf("@glo2: %p\n\n", (void*)&glo2);
    puts("~~text zone~~");
    printf("@dum : %p\n", (void*)dum); // not clearly standard?
    printf("@main: %p\n", (void*)main);

    return EXIT_SUCCESS;
}
```

Une exécution peut par exemple produire la sortie suivante :

Terminal 7.2

```
~~stack zone~~
@a : 0x7fff7e1f66d0
@b : 0x7fff7e1f66d4
@t : 0x7fff7e1f66f0
@p : 0x7fff7e1f66d8
@q : 0x7fff7e1f66e0
@r : 0x7fff7e1f66e8

~~heap zone~~
@*p: 0x55b19527e310
@*q: 0x55b19527e330
@*r: 0x55b19527e360

~~data zone~~
@glo0: 0x55b16987703c
@glo1: 0x55b169877030
@glo3: 0x55b169877034
~~(read-only part)~~
@glo2: 0x55b169875004
```

```
~~text zone~~
@dum : 0x55b169874169
@main: 0x55b169874170
```

D'autres exécutions du même programme sur la même machine pourront produire des sorties similaires *mais généralement pas identiques* : pour des raisons de sécurité (que nous ne détaillerons pas), les adresses virtuelles utilisées par les processus ne sont typiquement pas complètement déterministes.

§ **Fautes d'accès**

Il se peut qu'un processus cherche à lire ou écrire à une adresse mémoire virtuelle qui ne correspond à aucune adresse physique, ou à écrire à une adresse mémoire se trouvant dans une zone en lecture seule. De telles tentatives sont détectées par le système d'exploitation au moment de la traduction de l'adresse virtuelle en adresse physique, ce qui l'amènera à déclencher une erreur, causant généralement l'interruption brutale du processus.

Le programme ci-dessous illustre le premier type d'erreur :

Programme 7.3

```c
#include <stdlib.h>

int main(void)
{
    int *p = (int *)12;
    *p = 1;

    return EXIT_SUCCESS
}
```

dont l'exécution entraîne une erreur bien méritée :

Terminal 7.4

```
> ./a.out
zsh: segmentation fault (core dumped)  ./a.out
```

Le programme suivant illustre quant à lui le second type d'erreur :

Programme 7.5

```c
#include <stdlib.h>

const int a = 12;

int main(void)
{
```

```c
    *((int *)&a) = 11;

    return EXIT_SUCCESS;
}
```

Son exécution produit également une erreur :

Terminal 7.6

```
> ./a.out
zsh: segmentation fault (core dumped)  ./a.out
```

Dans les deux exemples ci-dessus, une conversion explicite de type est nécessaire pour simplement pouvoir *compiler* les programmes, et l'on aurait pu se douter qu'effectuer une telle conversion afin de supprimer l'erreur :

Terminal 7.7

```
sgflt.c:7:7: error: assignment of read-only variable ‘a’
```

est peu susceptible de produire un programme correct. Malheureusement, il n'y a aucune garantie qu'un compilateur soit capable de détecter ces erreurs *en général*, d'où l'importance d'aussi utiliser des outils de diagnostique comme l'*address sanitizer* (malheureusement aussi faillibles) et de tester ses programmes. Ceci est d'autant plus vrai que certaines erreurs de manipulation de la mémoire (*cf.* ci-dessous) ne sont d'aucun des deux types ci-dessus, et à ce titre ne déclenchent pas d'erreur par le système d'exploitation.

§ Corruption mémoire

Comme l'illustre la sortie du Programme 7.1, les objets manipulés par un programme peuvent posséder des adresses virtuelles proches. Il s'ensuit qu'il est relativement aisé pour un programme d'accéder à d'autres objets que celui souhaité, sans que cela ne déclenche d'erreur par le système d'exploitation puisque les adresses accédées sont légales du point de vue de ce dernier. Ceci constitue cependant bien une erreur du point de vue du langage de programmation (en C, tout du moins, où cela résulte en un comportement non défini caractérisé), et peut avoir des conséquences fâcheuses.

De façon plus basique, le système d'exploitation peut associer des adresses physiques à des adresses virtuelles au delà de ce qui est strictement nécessaire ; le programme ci-dessous illustre ce phénomène :

Programme 7.8

```c
#include <stdlib.h>

int main(void)
{
    int *p = malloc(4*sizeof(int));
    p[4] = 12;
```

```
    return EXIT_SUCCESS;
}
```

Si compilé sans précautions particulières, son exécution ne provoquera typiquement aucune erreur, et se terminera avec succès. L'utilisation de l'*address sanitizer* permet cependant de mettre le problème en évidence :

Terminal 7.9

```
=================================================================
==42323==ERROR: AddressSanitizer: heap-buffer-overflow on address
0x7c0229de0020 at pc 0x56022ae4f6b6 bp 0x7ffe2e56ba30 sp 0x7ffe2e56ba28
WRITE of size 4 at 0x7c0229de0020 thread T0
< message tronqué >
```

Le programme ci-dessous illustre le premier type d'erreur mentionné dans cette section, où l'on accède de façon détournée (et illégale) à un objet existant réellement :

Programme 7.10

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

int main(void)
{
    int *p = malloc(4*sizeof(int));
    int *q = malloc(4*sizeof(int));
    q[0] = 12;
    p[8] = 11;
    printf("%d\n", q[0]);

    return EXIT_SUCCESS;
}
```

Sur une certaine architecture, il produit la sortie :

Terminal 7.11

```
11
```

et termine avec succès. Ce type d'erreur est malheureusement difficile à détecter, y compris par l'*address sanitizer*.

7.2 Zone de la pile

§ Rôle & organisation

Dans l'organisation typique décrite à la section précédente, la pile (en anglais : *stack*) est utilisée pour stocker les contextes (en anglais : *stack frames*) des fonctions *en cours d'exécution* : chaque *exécution* d'une fonction nécessite généralement de stocker des objets en mémoire, typiquement (certains de) ses arguments et l'espace nécessaire pour ses variables locales et de façon interne le point du programme depuis lequel la fonction a été appelée (en anglais : le *return instruction pointer*, ou *rip*), et éventuellement encore d'autres données. Chaque *appel* de fonction donne alors lieu à une *allocation* mémoire pour ces objets (qui sont de durée de stockage automatique), et chaque fin d'exécution à une *libération*.

Ces mécanismes d'allocation et de libération sont typiquement implémentés de la façon suivante : une zone mémoire (à la fois virtuelle, et la mémoire physique correspondant) d'une certaine taille généralement fixe est réservée pour la pile au début de l'exécution du programme, et un *pointeur de pile* (en anglais : *stack pointer*) est initialisé avec l'adresse au début de la pile. À chaque appel de fonction, le pointeur de pile est *décrémenté* (la pile grandit « vers le bas ») d'une certaine quantité (variable pour chaque fonction, en fonction de la taille occupée par les arguments et variables locales), et l'espace mémoire délimité par l'ancienne et la nouvelle valeur du pointeur de pile (la *stack frame*) est utilisé pour stocker le contexte de l'appel effectué qui peut être agrandi au cours de l'exécution de la fonction, pour des raisons variées liées à l'architecture matérielle ou dans certains cas au langage de programmation. Lorsqu'un appel de fonction termine, le pointeur de pile est restauré à sa valeur précédente, ce qui a pour effet de rendre à nouveau disponible la zone mémoire précédemment occupée par le contexte de celle-ci : toute la mémoire occupée par les éventuelles variables locales (notamment les tableaux) est automatiquement libérée. Ces mécanismes sont donc bien adaptés au stockage d'objets de durée automatique, et c'est effectivement ce pour quoi ils sont typiquement employés.

La structure logique implémentée de cette façon est celle d'une pile (...) : l'espace mémoire utilisé pour stocker le contexte d'exécution d'une fonction se trouve immédiatement en dessous (ou au dessus, si l'on préfère considérer une pile logique croissant vers le haut) des contextes de tous les éventuels appels de fonction *ayant déjà eu lieu et n'ayant pas encore terminé*. Symétriquement, les contextes de tous les éventuels appels ultérieurs seront également stockés en dessous (ou au dessus). Ainsi, chaque appel de fonction *empile* un nouveau contexte en bas (au dessus) de la pile, et chaque retour d'appel en *dépile* un.

Une conséquence de cette organisation est que lors de son exécution, une fonction peut concrètement (à condition d'en posséder une adresse) accéder à un objet stocké dans le contexte d'exécution d'une fonction se trouvant sous elle dans la pile, ce qui correspond exactement à une fonction *appelée plus tôt et n'ayant pas encore terminé* ; ceci est aussi légal puisque cela revient effectivement à accéder à des objets de durée de stockage automatique encore vivants, comme précédemment illustré par les Programmes 5.14 et 4.29. En revanche il est illégal de chercher à faire de même pour un appel *ayant déjà terminé*, puisque dans ce cas la durée de vie des objets aura pris fin ; cela posera également un problème concret d'accès mémoire, puisque la zone mémoire utilisée par le contexte a été libérée (et a donc pu être réutilisée, par exemple par une autre fonction lors de son exécution) ; nous avons déjà illustré cette erreur avec le Programme 4.33.

Les mécanismes d'allocation & libération décrits ci-dessus sont les plus courants, mais ne sont pas les seuls (même au sein d'un environnement UNIX) : les contextes de fonctions appelées de façon asynchrone ne peuvent pas (et ne sont pas) stockés dans la pile, et l'utilisation de `longjmp` peut « dépiler » plusieurs contextes en une seule fois.

Lorsqu'exécuté dans un environnement UNIX typique, le programme suivant permet de visualiser approximativement les valeurs prises par le pointeur de pile à chaque appel de fonction :

Programme 7.12

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

void dum2(void)
{
    int psp;
    printf("%s: @psp : %p\n", __func__, (void*)&psp);

    return;
}

void dum1(int a)
{
    int psp;
    printf("%s: @psp : %p\n", __func__, (void*)&psp);

    if (a > 0)
    {
        dum1(a - 1);
    }
    else
    {
        dum2();
    }
}

int main(void)
{
    int psp;

    printf("%s: @psp : %p\n", __func__, (void*)&psp);
    dum1(2);

    return EXIT_SUCCESS;
}
```

Une exécution typique produit la sortie :

Terminal 7.13

```
main: @psp : 0x7ffcaed79484
dum1: @psp : 0x7ffcaed79464
dum1: @psp : 0x7ffcaed79434
```

```
dum1: @psp : 0x7ffcaed79404
dum2: @psp : 0x7ffcaed793d4
```

où chaque appel de fonction affiche l'adresse d'une variable locale se trouvant dans son contexte (et donc à proximité de la valeur courante du pointeur de pile). On peut remarquer que les trois appels de la fonction *récursive* (qui s'appelle elle-même) `dum1` possèdent chacun leur propre contexte.

L'utilisation d'un débuggueur permet également à la fois de consulter l'état courant de la pile d'appel (c'est à dire, l'empilement des appels de fonction en cours) et le contenu individuel des contextes. Dans cette optique et afin d'avoir les informations les plus complètes possibles, il est utile d'utiliser l'option de compilation `-fno-omit-frame-pointer`, qui garantit (en général) que chaque contexte inclura toujours un *frame pointer* vers la fin du contexte immédiatement précédent. Une exécution sous *gdb* du programme suivant peut alors ressembler à :

Débuggueur 7.14

```
Reading symbols from ./a.out...
(gdb) break dum2
Breakpoint 1 at 0x1151: file framezz.c, line 5.
(gdb) run > /dev/null
Starting program: /tmp/a.out > /dev/null
Breakpoint 1, dum2 () at framezz.c:5
5   {
(gdb) bt
#0  dum2 () at framezz.c:5
#1  0x00005555555551ef in dum1 (a=0) at framezz.c:23
#2  0x00005555555551e8 in dum1 (a=1) at framezz.c:19
#3  0x00005555555551e8 in dum1 (a=2) at framezz.c:19
#4  0x0000555555555249 in main () at framezz.c:32
(gdb) info frame 0
Stack frame at 0x7fffffffe460:
 rip = 0x555555555151 in dum2 (framezz.c:5); saved rip = 0x5555555551ef
 called by frame at 0x7fffffffe490
 source language c.
 Arglist at 0x7fffffffe450, args:
 Locals at 0x7fffffffe450, Previous frame's sp is 0x7fffffffe460
 Saved registers:
  rbp at 0x7fffffffe450, rip at 0x7fffffffe458
(gdb) info frame 1
Stack frame at 0x7fffffffe490:
 rip = 0x5555555551ef in dum1 (framezz.c:23); saved rip = 0x5555555551e8
 called by frame at 0x7fffffffe4c0, caller of frame at 0x7fffffffe460
 source language c.
 Arglist at 0x7fffffffe480, args: a=0
 Locals at 0x7fffffffe480, Previous frame's sp is 0x7fffffffe490
 Saved registers:
```

```
  rbp at 0x7fffffffe480, rip at 0x7fffffffe488
(gdb) info frame 4
Stack frame at 0x7fffffffe510:
 rip = 0x555555555249 in main (framezz.c:32); saved rip = 0x7ffff7c27675
 caller of frame at 0x7fffffffe4f0
 source language c.
 Arglist at 0x7fffffffe500, args:
 Locals at 0x7fffffffe500, Previous frame's sp is 0x7fffffffe510
 Saved registers:
  rbp at 0x7fffffffe500, rip at 0x7fffffffe508
```

§ **Taille & débordement**

Par défaut, la taille de la zone mémoire allouée pour la pile d'exécution est généralement fixe et relativement petite, typiquement de l'ordre de quelques méga-octets, souvent 8 sur les systèmes UNIX récents (bien sûr, rien n'empêche d'allouer une pile plus grande, mais si fait de façon globale, cela augmente mécaniquement la consommation mémoire de *tous* les processus, et il y a donc un compromis à trouver ; il est aussi *en principe* possible de créer des piles de tailles variables sur certains systèmes). Ceci n'est pas sans conséquences puisque par conception, la *somme* des tailles des contextes (minorée par la somme des tailles des variables locales) des appels de fonction en cours d'exécution ne doit pas dépasser cette taille, et doit donc elle-même être maintenue relativement petite.

Ceci implique notamment qu'il n'est pas possible d'utiliser des variables locales trop grosses (typiquement de trop grands tableaux), comme précédemment illustré par le Programme 4.31.

Ceci implique également que le nombre total d'appels de fonction (du point de vue de l'environnement d'exécution) en cours d'exécution ne peut pas être arbitrairement grand, puisqu'un contexte n'est jamais de taille nulle. La taille typique de la pile est toutefois assez grande pour accueillir les contextes (de taille raisonnable) de plusieurs centaines de milliers d'appels ; ceux-ci ne peuvent guère être écrits à la main et sont typiquement le fait de fonctions récursives. Cette limitation en est donc surtout une pour les langages de programmation « fonctionnels », dont le style d'écriture favorise l'utilisation de ce type de fonctions. Pour cette même raison, elle motive l'importance de l'élimination automatique des appels récursifs (en anglais : *tail-call optimization*) typiquement effectuée par ces langages, quand elle est aisée à effectuer.

Le programme suivant illustre un dépassement de pile (en anglais : *stack overflow*) causé par un trop grand nombre de contextes, créés par une fonction récursive qui ne termine jamais :

Programme 7.15

```c
#include <stdlib.h>

void loop(void)
{
    loop();
}

int main(void)
{
    loop();
```

```
    return EXIT_SUCCESS;
}
```

Une exécution produira typiquement :

Terminal 7.16

```
> ./a.out
zsh: segmentation fault (core dumped)  ./a.out
```

où l'erreur est causée par la tentative d'écriture d'un contexte d'exécution à la suite (et en dehors) de la zone allouée pour la pile. L'utilisation de l'*address sanitizer* permet d'obtenir un diagnostic assez précis du problème :

Terminal 7.17

```
AddressSanitizer:DEADLYSIGNAL
=================================================================
==88692==ERROR: AddressSanitizer: stack-overflow on address
0x7ffd18362ff8 (pc 0x55858eefc14d bp 0x7ffd18363000 sp 0x7ffd18363000 T0)
    #0 0x55858eefc14d in loop /tmp/loop.c:5
    #1 0x55858eefc151 in loop /tmp/loop.c:5
    #2 0x55858eefc151 in loop /tmp/loop.c:5
    #3 0x55858eefc151 in loop /tmp/loop.c:5
< message tronqué >
```

Quelque soit leur cause, les débordements de pile peuvent être difficiles à anticiper à l'exécution ; ils sont également plus faciles à déclencher sur des architectures embarquées disposant de peu de mémoire et d'une petite pile, ce qui est d'autant plus problématique si ces architectures sont utilisées pour exécuter des logiciels d'importance critique. Certaines entreprises commerciales proposent des analyseurs dont le but est de détecter *statiquement* (c'est à dire, sans exécuter les programmes) les risques de dépassement de pile, ou de prouver leur absence. De façon générale, les outils d'analyse statiques complémentent efficacement les tests & analyses dynamiques (telles qu'effectuées par les *sanitizers*) et sont particulièrement utiles lors du développement de logiciels critiques.

7.3 Zone du tas

De par son fonctionnement, la pile est adaptée au stockage d'objets de petite taille de durée de stockage automatique ; une autre zone mémoire et un autre mécanisme sont nécessaires pour le stockage d'objets de durée de stockage allouée, ainsi que pour le stockage d'objets de durée automatique mais trop gros pour tenir confortablement sur la pile.

Sur les systèmes UNIX, le système d'exploitation fournit des *appels systèmes* (comme `brk`) qui permettent à un processus d'augmenter ou de diminuer la taille d'une certaine zone mémoire non initialisée, et d'autres (comme `mmap` et `munmap`) qui permettent de demander au système d'exploitation d'associer (ou libérer) un stockage physique à une zone d'adresses virtuelles souhaitée. L'ensemble des

plages d'adresses manipulées par ces appels systèmes est communément appelée le *tas* (en anglais : *heap*), qui contrairement à la pile vue précédemment n'a aucun lien avec la structure de données du même nom.

Lez zones mémoires allouées sur le tas par ces appels systèmes doivent être libérées explicitement, et resteront occupées tant que cela n'a pas été fait ; ces mécanismes sont donc adaptés au stockage d'objets de durée allouée. Il n'y a pas non plus de limite *a priori* sur la taille des zones pouvant être allouées, et donc sur la taille des objets qui peuvent être stockés dans le tas. En conséquence, le tas est l'endroit privilégié pour stocker les objets de grande taille, y compris ceux qui remplissent le rôle de variables locales mais sont trop grand pour tenir dans la pile. Ceci explique pourquoi en C il est courant en pratique de créer des objets de durée de stockage allouée (*généralement* stockés sur un tas), alors que ceux-ci sont *moralement* de durée de stockage automatique.

En C sur système UNIX, ces appels systèmes peuvent être (et sont) utilisés pour implémenter les fonctions d'allocation et de libération de mémoire de durée de stockage allouée (par exemple `malloc` et `free`), qui servent alors à s'abstraire des particularités du système d'exploitation. Cette abstraction est nécessaire car la présence d'un tas manipulable par (disons) `brk` *n'est pas garantie par le langage C lui-même*, et un programme en faisant directement l'usage ne serait pas *portable* : il pourrait uniquement s'exécuter dans un environnement présentant ces caractéristiques (qui bien que courantes, ne sont pas universelles).

§ **Politique de pagination**

Les systèmes d'exploitation modernes utilisent typiquement une politique de *pagination à la demande*. Concrètement, cela veut dire que lorsque certains appels systèmes (par exemple `mmap`) sollicitent le système d'exploitation pour disposer d'une zone mémoire physique, celui-ci la leur accorde seulement *en principe* : seule une tentative d'accès effective à (chaque *page* de) cette zone déclenchera l'attribution *réelle* des ressources physiques correspondantes. Une conséquence de ce mécanisme est que ce n'est pas parce qu'une fonction d'allocation mémoire (typiquement `malloc`) s'est exécutée avec succès que le programme pourra *réellement* disposer de toute la mémoire demandée : un éventuel *surbooking* fait qu'en fonction de la consommation réelle des autres processus, il peut ne pas y avoir assez de mémoire physique de disponible pour remplir toute la zone allouée.

Le programme ci-dessous permet de visualiser ce phénomène :

Programme 7.18

```c
#include <stdlib.h>

int main(void)
{
//    size_t n = 8000000000; // échec de l'allocation
//    size_t n = 4000000000; // tué à l'exécution
    size_t n = 1000000000;
    int *p = malloc(n * sizeof(int));

    for (size_t i = 0; i < n; i++)
    {
        p[i] = 0;
    }
```

```
    return EXIT_SUCCESS;
}
```

Sur un certain ordinateur disposant de 16 Go de mémoire vive, une allocation pour 8 000 000 000 entiers de type `int` échouera car il n'y a *clairement* pas assez de mémoire pour les stocker tous ensemble. En revanche la même demande d'allocation pour 4 000 000 000 se fera avec succès, mais le programme sera typiquement tué à l'exécution quand il cherchera à *réellement* accéder à la zone allouée toute entière. Une allocation pour 1 000 000 000 d'`int`s, quant à elle, se passera bien y compris à l'exécution (si le système n'est pas trop sollicité par ailleurs).

Certains utilitaires de visualisation de charge comme *htop* permettent également de visualiser le même phénomène, en affichant pour chaque processus la quantité de mémoire allouée *en principe* et celle réellement utilisée (parfois très inférieure).

7.4 Format d'exécutable

On termine cette section par brièvement décrire les aspects du *format d'exécutable ELF* (couramment utilisé sur les systèmes UNIX) relatifs au stockage des objets de durée de stockage statique (typiquement les variables globales) : ceux-ci ne sont stockés ni sur la pile ni dans le tas, mais dans une zone mémoire initialisée avec le contenu de l'exécutable lui-même. Autrement dit (contrairement aux objets de durée de stockage automatique ou allouée), ces objets font partie de l'exécutable, au même titre que le code machine. L'emplacement exact au sein de l'exécutable dépend des caractéristiques des objets : ceux correspondant aux variables qualifiées `const` (en lecture seule) sont typiquement placés dans la *section .rodata* (pour « données en lecture seule ») ; ceux correspondant aux variables accessibles en écriture et initialisées à une valeur différente de « zéro » sont typiquement placés dans la section *.data* (pour « données ») ; ceux correspondant aux variables accessibles en écriture et non initialisées ou explicitement initialisées à « zéro » (ce qui du point de vue du langage C est identique) sont typiquement placées dans la section *.bss* (pour *block starting symbol*). Cette dernière différence vient du fait que cette section contient *uniquement* des bits nuls, et n'a donc pas besoin d'être explicitement stockée dans l'exécutable (ce qui permet de limiter la taille de ce dernier).

On peut aisément afficher le contenu de chacune de ces sections d'un exécutable en utilisant l'utilitaire *objdump*, ce qui pour le Programme 7.1 (compilé et *dumpé* sur un certain système Linux) donne pour la section *.rodata* :

Terminal 7.19

```
> objdump -d --section=.rodata a.out
a.out:     file format elf64-x86-64

Disassembly of section .rodata:

0000000000002000 <_IO_stdin_used>:
    2000:   01 00 02 00                                         ....

0000000000002004 <glo2>:
    2004:   02 00 00 00 7e 7e 73 74 61 63 6b 20 7a 6f 6e 65     ....~~stack zone
    2014:   7e 7e 00 40 61 20 3a 20 25 70 0a 00 40 62 20 3a     ~~.@a : %p..@b :
```

```
2024:   20 25 70 0a 00 40 74 20 3a 20 25 70 0a 00 40 70     %p..@t : %p..@p
2034:   20 3a 20 25 70 0a 00 40 71 20 3a 20 25 70 0a 00     : %p..@q : %p..
2044:   40 72 20 3a 20 25 70 0a 0a 00 7e 7e 68 65 61 70    @r : %p...~~heap
2054:   20 7a 6f 6e 65 7e 7e 00 40 2a 70 3a 20 25 70 0a     zone~~.@*p: %p.
2064:   00 40 2a 71 3a 20 25 70 0a 00 40 2a 72 3a 20 25    .@*q: %p..@*r: %
2074:   70 0a 0a 00 7e 7e 64 61 74 61 20 7a 6f 6e 65 7e    p...~~data zone~
2084:   7e 00 40 67 6c 6f 30 3a 20 25 70 0a 00 40 67 6c    ~.@glo0: %p..@gl
2094:   6f 31 3a 20 25 70 0a 00 40 67 6c 6f 33 3a 20 25    o1: %p..@glo3: %
20a4:   70 0a 00 7e 7e 28 72 65 61 64 2d 6f 6e 6c 79 20    p..~~(read-only
20b4:   70 61 72 74 29 7e 7e 00 40 67 6c 6f 32 3a 20 25    part)~~.@glo2: %
20c4:   70 0a 0a 00 7e 7e 74 65 78 74 20 7a 6f 6e 65 7e    p...~~text zone~
20d4:   7e 00 40 64 75 6d 20 3a 20 25 70 0a 00 40 6d 61    ~.@dum : %p..@ma
20e4:   69 6e 3a 20 25 70 0a 00                            in: %p..
```

qui comme on peut le remarquer contient également les chaînes de caractères constantes affichées par le programme ; la section *.data* s'obtient de la même manière :

Terminal 7.20

```
> objdump -d --section=.data a.out
a.out:     file format elf64-x86-64

Disassembly of section .data:

0000000000004020 <__data_start>:
    ...

0000000000004028 <__dso_handle>:
    4028:   28 40 00 00 00 00 00 00

0000000000004030 <glo1>:
    4030:   01 00 00 00

0000000000004034 <glo3>:
    4034:   03 00 00 00
```

ainsi que la section *.bss* :

Terminal 7.21

```
> objdump -d --section=.bss a.out
a.out:     file format elf64-x86-64

Disassembly of section .bss:

0000000000004038 <__bss_start>:
```

```
    4038:   00 00                   add    %al,(%rax)
    ...

000000000000403c <glo0>:
    403c:   00 00 00 00
```

Le code machine contenu dans l'exécutable se trouve quant à lui dans la section *.text* :

Terminal 7.22

```
> objdump -d --section=.text a.out
a.out:     file format elf64-x86-64

Disassembly of section .text:

0000000000001070 <_start>:
    1070:   f3 0f 1e fa             endbr64
    1074:   31 ed                   xor    %ebp,%ebp
    1076:   49 89 d1                mov    %rdx,%r9
    1079:   5e                      pop    %rsi
    107a:   48 89 e2                mov    %rsp,%rdx
    107d:   48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
    1081:   50                      push   %rax
    1082:   54                      push   %rsp
    1083:   45 31 c0                xor    %r8d,%r8d
    1086:   31 c9                   xor    %ecx,%ecx
    [...]
    114d:   00 00 00
    1150:   c3                      ret
    1151:   0f 1f 40 00             nopl   0x0(%rax)
    1155:   66 66 2e 0f 1f 84 00    data16 cs nopw 0x0(%rax,%rax,1)
    115c:   00 00 00 00
    1160:   f3 0f 1e fa             endbr64
    1164:   e9 67 ff ff ff          jmp    10d0 <_start+0x60>

0000000000001169 <dum>:
    1169:   55                      push   %rbp
    116a:   48 89 e5                mov    %rsp,%rbp
    116d:   90                      nop
    116e:   5d                      pop    %rbp
    116f:   c3                      ret

0000000000001170 <main>:
    1170:   55                      push   %rbp
    1171:   48 89 e5                mov    %rsp,%rbp
```

```
1174:   48 83 ec 50             sub    $0x50,%rsp
1178:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
117f:   00 00
1181:   48 89 45 f8             mov    %rax,-0x8(%rbp)
1185:   31 c0                   xor    %eax,%eax
1187:   bf 04 00 00 00          mov    $0x4,%edi
118c:   e8 cf fe ff ff          call   1060 <malloc@plt>
1191:   48 89 45 b8             mov    %rax,-0x48(%rbp)
1195:   bf 28 00 00 00          mov    $0x28,%edi
119a:   e8 c1 fe ff ff          call   1060 <malloc@plt>
[...]
139e:   e8 ad fc ff ff          call   1050 <printf@plt>
13a3:   b8 00 00 00 00          mov    $0x0,%eax
13a8:   48 8b 55 f8             mov    -0x8(%rbp),%rdx
13ac:   64 48 2b 14 25 28 00    sub    %fs:0x28,%rdx
13b3:   00 00
13b5:   74 05                   je     13bc <main+0x24c>
13b7:   e8 84 fc ff ff          call   1040 <__stack_chk_fail@plt>
13bc:   c9                      leave
13bd:   c3                      ret
```

8 Types flottants

Les types entiers sont adaptés à la représentation et au calcul avec des éléments de structures discrètes, comme l'anneau des entiers $\mathbb{Z}$ ou ses quotients $\mathbb{Z}/n\mathbb{Z}$. S'il n'existe pas de type entier standard de *précision arbitraire* en C (permettant de représenter des entiers de valeur absolue arbitrairement grande), il est relativement aisé de construire un tel type « à la main », au moins tant que l'on se contente d'une arithmétique naïve et peu efficace. Et si ce n'est pas le cas, on pourra utiliser une bibliothèque externe comme GMP ou FLINT.On peut également arguer que hors applications spécifiques, il est peu probable d'avoir besoin de calculer avec des entiers de valeur absolue supérieure à 2^{63}, et que les types standards sont donc généralement suffisants. En revanche, il est plus courant d'avoir besoin de représenter et calculer avec des nombres dits « réels », et implémenter efficacement un tel type à partir de types entiers serait un peu pénible. Pour ces raisons, le langage C définit trois types *flottants* standards, *viz.* `float`, `double` (le seul au programme des CPGE) et `long double`. Il définit également trois types flottants optionnels : `_Decimal32`, `_Decimal64` et `_Decimal128`. Ces types sont adaptés à la représentation et au calcul **approché** avec des nombres « réels ».

¶ **Écriture des littéraux flottants.** Les constantes littérales de types flottants, qui par défaut sont de type `double`, peuvent s'écrire avec plusieurs formats. Le plus courant est la notation « pointée » décimale « `i.f` », où `i` dénote la partie entière du nombre « réel », et `f` sa partie fractionnaire, toutes deux écrites en base dix. Si l'une de ces deux quantités est nulle, elle peut être omise dans l'écriture du littéral (on ne peut cependant pas juste écrire « `.` » pour « `0.0` » !). Un autre format relativement courant est la notation « scientifique » décimale « `i.fEe` », où `e` est un exposant (éventuellement négatif) en base dix pour un multiplicateur lui aussi en base dix : ainsi, `i.fEe` désigne le même nombre que `i.f` multiplié par 10^e. Le fragment suivant donne quelques exemples de littéraux :

Fragment 8.1

```
double m_pi = 3.14159265358979323846;
double m_pi_2 = 1.57079632679489661923;
double a = 0.;
double b = .12E-10;
```

¶ **Affichage des valeurs flottantes.** Le spécifieur de conversion pour l'affichage en notation « pointée » décimale es valeurs de type `double` est `%f` ; celui pour la notation « scientifique » décimale est `%e` ou `%E`. Par défaut, cet affichage ne se fait qu'avec une précision limitée, comme l'illustre le programme ci-dessous :

Programme 8.2

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

int main(void)
{
    double m_pi = 3.14159265358979323846;
    double m_pi_2 = 1.57079632679489661923;
    double a = 0.;
    double b = .12E-10;

    printf("%E\n%f\n%f\n%f\n", m_pi, m_pi_2, a, b);

    return EXIT_SUCCESS;
}
```

qui peut afficher :

Terminal 8.3

```
3.141593E+00
1.570796
0.000000
0.000000
```

On constate en particulier que cet affichage ne permet pas de distinguer deux valeurs pourtant différentes ! En remplaçant le dernier spécifieur par `%E`, on obtient :

Terminal 8.4

```
3.141593E+00
1.570796
```

```
0.000000
1.200000E-11
```

ce qui est plus fidèle. Il est également possible de spécifier une précision d'affichage différente de celle par défaut (de six chiffres après ~~la virgule~~ le point) en préfixant le spécifieur par `.p` où `p` est un entier décimal indiquant cette précision. Le programme ci-dessous :

Programme 8.5

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

int main(void)
{
    double m_pi = 3.14159265358979323846;
    double m_pi_2 = 1.57079632679489661923;
    double a = 0.;
    double b = .12E-10;

    printf("%.10f\n%.20f\n%f\n%.15f\n", m_pi, m_pi_2, a, b);

    return EXIT_SUCCESS;
}
```

pourra produire la sortie :

Terminal 8.6

```
3.1415926536
1.57079632679489655800
0.000000
0.000000000012000
```

Ceci permet au passage de constater que l'affichage de la valeur de `m_pi_2` avec la même précision que sa définition produit un résultat différent de cette définition ! Pourtant, ces deux constantes littérales désignent bien la *même* valeur, et se comparent comme égales ! On pourra prendre cela comme un avertissement au fait (développé dans le reste de cette section) que les types flottants ne fournissent que des représentations *approchées* des nombres « réels ».

8.1 Opérations sur les types flottants

Les types flottants peuvent être utilisés dans des expressions de même type ; le langage définit également certaines fonctions mathématiques supplémentaires.

§ Expressions flottantes

Les expressions flottantes partagent leurs opérateurs avec les expressions arithmétiques sur types entiers : l'addition, le produit et la division de flottants s'écrivent respectivement avec `+` ; `*` ; `/`. En

revanche, le calcul du reste dans la division entière de deux flottants (qui pourraient se trouver représenter des valeurs entières) ne peut pas se faire en utilisant l'opérateur %, mais il existe une fonction de bibliothèque dédiée pour cela, *cf.* la section ci-dessous. Les valeurs flottantes peuvent également être comparées avec les opérateurs de comparaison usuelles.

Encore une fois, on prendra garde au fait que les opérations sur les flottants sont des approximations des calculs sur les nombres « réels », comme l'illustre le programme ci-dessous :

Programme 8.7

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

int main(void)
{
    printf("%d\n", 0.1 + 0.2 == 0.3);
    printf("%d\n", 9007199254740992. == 9007199254740992. + 1.);

    return EXIT_SUCCESS;
}
```

qui pourra afficher :

Terminal 8.8

```
0
1
```

§ **Bibliothèque mathématique standard**

Le langage C définit une bibliothèque mathématique standard qui fournit un certain nombre de fonctions de calcul usuelles pour les types flottants. Cette bibliothèque n'est pas au programme des CPGE, et vous ne pouvez *a priori* pas en utiliser les fonctions dans une composition écrite.

Les fonctions de cette bibliothèque sont définies dans le fichier d'en-tête `math.h`, qui définit également quelques (approximations) de nombres notables comme $\pi/2$. Parmi ces fonctions, on trouve notamment :

— des fonctions trigonométriques ;
— des fonctions de division et d'arrondis ;
— des fonctions de calcul de logarithmes ;
— des fonctions d'exponentiation ;
— des fonctions de calcul de racines.

¶ **Compilation.** Pour compiler un programme utilisant des fonctions de la bibliothèque mathématique standard, il est nécessaire de *lier* celle-ci explicitement à la compilation. Ceci se fait en ajoutant une option `-lm` lors de la création de l'exécutable.

8.2 Modèle IEEE 754

La norme C n'impose pas complètement la façon dont les types flottants doivent être implémentés, mais le standard *de facto* est d'utiliser pour cela un modèle définit par la norme IEEE 754. Cette norme

n'est pas au programme des CPGE et aucune connaissance précise à son sujet n'est exigible ; on l'utilise néanmoins comme base pour brièvement décrire la façon dont les nombres flottants sont représentés physiquement.

8.3 Conversions entre types entiers & flottants

Il est possible de convertir des types flottants vers des types entiers, et vice-versa. Ces conversions peuvent à la fois être implicites ou explicitées avec la syntaxe déjà décrite à la Section 3.14.

Dans le cas des conversions implicites dans des expressions de types mixtes, ce sont les types flottants qui ont la précédence sur les types entiers : si une expression fait intervenir des opérandes de plusieurs types et que l'une des opérandes est de type flottant, c'est l'autre opérande qui est convertie vers le (même) type flottant.

Lors d'une conversion d'une valeur de type flottant vers un type entier, l'éventuelle partie fractionnaire n'est pas prise en compte ; l'éventuel arrondi effectué l'est donc « vers zéro ».

Qu'elles soient implicites ou non, les conversions impliquant des types flottants peuvent mener à deux types d'erreur : l'impossibilité de représenter la nouvelle valeur (*cf.* FLP34-C. Ensure that floating-point conversions are within range of the new type) ou (ce qui est un peu moins grave) une perte de précision (*cf.* FLP36-C. Preserve precision when converting integral values to floating-point type).

¶ **Impossibilité de représentation.** Ce type d'erreur est similaire à celles pouvant se produire lors de conversions entre types entiers, quand le type destination ne peut pas représenter la valeur du type source. C'est un comportement non défini de chercher à convertir une valeur flottante vers un type entier ne pouvant pas la représenter (ou un éventuel) arrondi, y compris quand le type destination est non signé.

Le programme ci-dessous met en évidence ce type d'erreur :

Programme 8.9

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

int main(void)
{
    double a = 1E20;
    int b = a;
    uint64_t c = a;

    return EXIT_SUCCESS;
}
```

Lorsque compilé avec l'option `-fsanitize=undefined`, son exécution produit :

Terminal 8.10

```
flp.c:7:13: runtime error: 1e+20 is outside the range of
representable values of type 'int'
```

```
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior flp.c:7:13
flp.c:8:18: runtime error: 1e+20 is outside the range of
representable values of type 'unsigned long'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior flp.c:8:18
```

¶ **Perte de précision.** Comme déjà vu précédemment, une caractéristique des types flottants est qu'ils ne permettent même pas de représenter tous les nombres entiers positifs plus petits que leur valeur maximale représentable. Ceci implique (comme déjà vu) qu'il est possible que des valeurs entières distinctes soient converties vers la même valeur flottante, entraînant une perte de précision. Si ce comportement est assez bien défini (la valeur entière représentée par le type flottant est garantie d'être l'une des plus proches possibles de la valeur initiale qui soit représentable), cela peut encore être source de bugs.

Le programme suivant illustre ce type de phénomène :

Programme 8.11

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

int main(void)
{
    printf("%d\n", (double)UINT64_MAX == (double)(UINT64_MAX - 1));

    return EXIT_SUCCESS;
}
```

en affichant :

Terminal 8.12

```
1
```

9 Manipulation de fichiers

10 Morceaux choisis

10.1 Opérateur ternaire

Le langage C définit un opérateur *ternaire* (à trois opérandes) qui permet de construire une expression dont la valeur d'évaluation est contrôlée par un test. Sa syntaxe est la suivante :

Syntaxe 10.1

```
expression_booléenne ? expression_si_vrai : expression_si_faux
```

et sa sémantique est intuitive : l'évaluation de l'expression `a ? b : c` commence par évaluer la sous-expression `a` et, si celle-ci s'est évaluée à `true` s'évalue à l'évaluation de `b`, et sinon à l'évaluation de `c`.

Par exemple, la première des expressions ci-dessous s'évalue à `12`, et la seconde à `4` :

Fragment 10.2

```
12 < 20 ? 12 : 20;
12 < 4 ? 12 : 4;
```

Cet opérateur est particulièrement pratique lorsque l'on souhaite effectuer une affectation conditionnelle, c'est à dire affecter une valeur à une variable en fonction d'une condition ; son utilisation permet alors d'alléger le code par rapport à l'utilisation d'une instruction de sélection `if`. On illustre ceci avec l'un des cas d'usage canonique pour cet opérateur, *viz.* le calcul du maximum de deux éléments :

Fragment 10.3

```
if (a < b)
{
    max = b;
}
else
{
    max = a;
}
```

Fragment 10.4

```
max = a < b ? b : a;
```

10.2 Génération d'aléa

La génération de données « aléatoires » en informatique est un vaste sujet que nous n'aborderons pas en détails ici. On se contentera de présenter quelques façons courantes (mais pas forcément standard) de générer des entiers non signés « aléatoires » en C suivant une distribution (à peu près) uniforme (tout besoin de génération d'aléa peu ultimement être ramené à de tels cas, mais ce n'est pas forcément facile).

Certaines des fonctions ci-dessous (par exemple `arc4random`) n'étant pas standard, la compilation d'un programme les utilisant échouera si l'on utilise une option de standard stricte comme `std=c23` ; on pourra alors utiliser à la place `std=gnu23` (par exemple), qui dénote un langage enrichi de certaines extensions (dont les fonctions en question).

§ Génération d'aléa déterministe

La bibliothèque standard C Linux fournit plusieurs générateurs *pseudo-aléatoires* (en anglais : *pseudo-random number generators*, ou *PRNG*) qui génèrent (à la demande) une suite d'entiers calculés à partir d'une *graine* (en anglais : *seed*). La graine détermine entièrement la suite générée : utiliser plusieurs fois la même graine donnera à chaque fois la même suite d'entiers. Ce type de générateur est donc très

pratique lorsque l'on souhaite générer des objets donnant l'apparence d'être « aléatoires », mais de façon reproductible ; ils posent par contre un problème d'initialisation évident lorsque l'on souhaite générer des suites différentes à chaque exécution.

¶ **`rand`.** La fonction `rand`, déclarée dans le fichier d'entête `stdlib.h` et de signature `int rand(void)`, renvoie un entier entre `0` et `RAND_MAX` tous deux inclus, où `RAND_MAX` vaut typiquement `INT_MAX`, soit $2\,147\,483\,647 = 2^{31} - 1$ sur les architectures usuelles. Il est donc important de remarquer que (en pratique), le bit de poids fort de la représentation physique de l'entier renvoyé est toujours nul.

Par défaut, la graine utilisée par `rand` est initialisée à `1`, et l'on peut spécifier une autre graine comme l'argument de la fonction `srand`, de signature `void srand(unsigned int seed)`. Chaque appel à `srand` réinitialise l'état interne du générateur, et l'on prendra donc garde à ne pas le réinitialiser « trop souvent ».

Le programme ci-dessous illustre l'utilisation de `rand` et `srand` :

Programme 10.5

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

int main(void)
{
    int r = rand();
    printf("%d\n%d\n", RAND_MAX, r);

    srand(2);
    r = rand();
    printf("%d\n", r);
    r = rand();
    printf("%d\n", r);

    srand(1);
    r = rand();
    printf("%d\n", r);

    return EXIT_SUCCESS;
}
```

Sur une certaine architecture fixée, il produira *toujours* la sortie suivante :

Terminal 10.6

```
2147483647
1804289383
1505335290
1738766719
1804289383
```

¶ **`random`.** La fonction `random` et la fonction d'initialisation associée `srandom` sont similaires à `rand` et `srand` (sur une version récente de Linux, elles sont rigoureusement identiques), mais `random` peut également être initialisée avec une graine plus grande qu'un simple entier non signé, *via* d'autres fonctions que nous ne décrirons pas ici.

§ **Génération d'aléa aléatoire**

La bibliothèque standard C Linux fournit également plusieurs fonctions de génération d'aléa qui *ne peuvent pas* être initialisées, et qui produisent des nombres aléatoires différents à chaque appel. Elles ne sont donc pas directement adaptées aux situations où un (pseudo-)aléa reproductible est nécessaire, mais peuvent dans ce cas servir à générer une graine ensuite utilisée avec (par exemple) `srand` & `rand`.

¶ **`getentropy`.** La fonction `getentropy`, déclarée dans le fichier d'entête `unistd.h` et de signature `int getentropy(void buffer[], size_t length)` écrit `length` octets pseudo-aléatoires dans le tableau `buffer` (qui doit être de longueur (au moins) `length`) ; elle renvoie zéro en cas de succès, et `-1` en cas d'erreur. L'aléa généré utilise le générateur aléatoire du système d'exploitation, qui incorpore certains phénomènes physiques afin de produire de l'aléa « réellement aléatoire » (en anglais : *true random number generator*, ou *TRNG*) ; il est donc de (très) bonne qualité, mais relativement lent, et on lui préférera en pratique `arc4random` (quand ce dernier est disponible). On peut également noter l'existence d'une fonction `getrandom` très similaire à `getentropy`, qui est d'ailleurs implémentée avec cette première.

Le fragment de code suivant illustre l'utilisation de `getentropy` en définissant quatre fonctions de génération d'entiers non signés de taille variable.

Fragment 10.7

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

uint64_t randu64(void)
{
    uint64_t r;
    getentropy(&r, 8);

    return r;
}

uint32_t randu32(void)
{
    uint32_t r;
    getentropy(&r, 4);

    return r;
}

uint16_t randu16(void)
{
```

```c
    uint16_t r;
    getentropy(&r, 2);

    return r;
}

uint8_t randu8(void)
{
    uint8_t r;
    getentropy(&r, 1);

    return r;
}
```

¶ **`arc4random`.** Sous Linux, la fonction `arc4random` déclarée dans le fichier d'entête `stdlib.h` et de signature `uint32_t arc4random(void)` utilise le même générateur que `getentropy` pour initialiser un algorithme de génération de pseudo-aléa similaire à `rand` mais de bien meilleure qualité. Cette fonction est probablement la plus simple à utiliser pour générer des entiers non signés de façon non reproductible, mais elle est relativement récente et ne sera donc pas forcément présente dans tous les environnements que vous rencontrerez.

Il existe également une fonction `arc4random_uniform`, également déclarée dans `stdlib.h` et de signature `uint32_t arc4random_uniform(uint32_t upper_bound)`, telle que pour *upper_bound* un entier supérieur (strictement) à zéro, elle renvoie un nombre tiré uniformément dans l'intervalle $[\![0, \textit{upper\_bound}-1]\!]$. Cette variante est utile car il n'est *pas* vrai que si X est distribuée uniformément dans l'intervalle $[\![0, a]\!]$, alors $X\%b$ (avec $b < a$, non multiple de a) est distribuée uniformément dans l'intervalle $[\![0, b]\!]$. Autrement dit, si vous souhaitez (par exemple) générer un nombre uniforme entre 0 et 10^9 (exclu), la bonne solution (à base de `arc4random`) est d'utiliser `arc4random_uniform(1000000000)` et non pas `arc4random() % 1000000000` qui produira un résultat biaisé car alors les valeurs inférieures à 294 967 295 seront plus probables que les autres. En revanche ce biais serait statistiquement négligeable —et la première solution acceptable— si l'on souhaitait procéder de même pour générer un nombre uniforme entre 0 et 3… Le sujet de la génération d'aléa n'est pas si simple.

10.3 Opérateurs bit-à-bit

En tant que langage relativement « bas niveau », le C fournit un certain nombre d'opérateurs permettant de manipuler les représentations des entiers au niveau bit ; on les nomme généralement « opérateurs bit-à-bit » (en anglais : *bitwise operators*). Bien que définis pour tout type entier, leur usage le plus pertinent l'est pour les types non signés de préférence de taille fixe, c-à-d typiquement les types modernes `uintx_t`.

Ces opérateurs sont souvent utiles lorsque les données manipulées ne servent pas vraiment à représenter des entiers, et qu'une manipulation plus fine que permise par les opérateurs arithmétiques devient nécessaire. Par exemple, la technique du *bitslicing* (*cf.* Section 10.4) consiste à utiliser des types entiers (représentés en binaire) pour stocker des « vecteurs » de valeurs booléennes, et les opérateurs bit-à-bit permettent d'opérer directement sur ces vecteurs ainsi représentés.

Une autre utilisation classique consiste à implémenter efficacement certaines opérations arithmétiques par manipulation directe de la représentation binaire des nombres : typiquement les multiplications et divisions (calcul du quotient et reste) par les puissances de deux. C'est probablement l'utilisation la plus

pertinente qui peut être faite de ces opérateurs dans le cadre des CPGEs, mais on prendra garde à ce *qu'elle ne nuise pas à la lisibilité du code*. En particulier, on s'abstiendra de les utiliser quand les puissances de deux en question sont constantes. Dans ce cas, votre compilateur est assez grand pour choisir lui-même la meilleure façon d'effectuer le calcul.

Une précaution générale à prendre lors de l'utilisation d'opérateurs bit-à-bit est de correctement typer les éventuels littéraux entiers, ou de s'abstenir d'utiliser les opérateurs avec des littéraux. En effet, ces opérateurs agissant sur la représentation binaire, la taille de cette représentation (et donc le type) prend une importance particulière.

¶ **Opérateur ~.** L'opérateur préfixe unaire `~` désigne la complémentation (ou le « non ») bit-à-bit. Autrement dit, l'expression `~x` s'évalue en un entier dont les bits à un sont les bits à zéro dans la représentation binaire de `x`, et vice-versa. Cet opérateur peut notamment être utilisé pour rapidement créer une valeur dont tous les bits sont à un par complémentation de zéro, comme illustré par le programme (perfectible) ci-dessous :

Programme 10.8

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

int main(void)
{
    uint64_t a = ~0UL;
    printf("%lX\n", a);

    return EXIT_SUCCESS;
}
```

dont l'exécution produit :

Terminal 10.9

```
FFFFFFFFFFFFFFFF
```

¶ **Opérateurs >> et <<.** Les opérateurs `>>` et `<<` effectuent un *décalage* respectivement « à droite » (vers les bits de poids faible) et « à gauche » (vers les bits de poids fort) de (la représentation binaire de) leur opérande de gauche par la quantité indiquée par leur opérande de droite, qui *doit être positive et strictement inférieure à la taille de la représentation*, et introduit des bits à zéro aux positions laissées non définies. Autrement dit, l'expression `x << s` s'évalue à un entier dont les bit d'indice 0 à $s-1$ sont nuls, et dont chaque autre bit d'indice $i \geq s$ est égal au bit d'indice $i-s$ de `x`.

L'utilisation de ces opérateurs avec des entiers signés est délicate (et l'on s'abstiendra donc de les utiliser pour ces types) : le comportement de l'opérateur `>>` pour des entiers signés négatifs est *implementation defined*, et celui de l'opérateur `<<` peut être cause de dépassement de capacité, qui est un *undefined behaviour*.

Le programme suivant illustre les effets de ces opérateurs dans divers cas, ainsi que (comme indiqué en préambule) l'importance de correctement typer les éventuels littéraux.

Programme 10.10

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

int main(void)
{
    uint64_t a = 1;
    uint64_t b = a << 5;
    uint64_t c = b >> 5;
    uint64_t d = c >> 1;
    // uint64_t e = 1 << 63; // UB
    uint64_t e = 1UL << 63;

    printf("a:%lb\nb:%lb\nc:%lb\nd:%lb\ne:%lb\n", a, b, c, d, e);

    return EXIT_SUCCESS;
}
```

Son exécution produit :

Terminal 10.11

```
a:1
b:100000
c:1
d:0
e:1000000000000000000000000000000000000000000000000000000000000000
```

Une utilisation courante de ces opérateurs est l'écriture aisée d'expressions s'évaluant à des puissances de deux, et donc également de valeurs dont les bits à un sont exactement les n bits de poids faible :

Fragment 10.12

```c
uint64_t t20 = 1 << 20; // 1048576
uint64_t m20 = (1 << 20) - 1; // 0xFFFFF
```

Ainsi que de fonctions de multiplication ou (quotient de la) division par des puissances de deux :

Fragment 10.13

```c
uint64_t mul_pow2(uint64_t x, unsigned e)
```

```c
{
    assert(e < 64);

    return x << e;
}

uint64_t quo_pow2(uint64_t x, unsigned e)
{
    assert(e < 64);

    return x >> e;
}
```

¶ **Opérateur &.** L'opérateur `&` (à ne pas confondre avec `&&` !) calcule un ET logique bit-à-bit entre ses deux opérandes. Autrement dit, l'expression `x & y` s'évalue à un entier dont chaque bit en position i vaut un ssi. le bit de x en position i et le bit de y en position i valent tous-deux un.

Une utilisation courante de cet opérateur est le calcul du reste de la division par une puissance de deux :

Fragment 10.14

```c
uint64_t rem_pow2(uint64_t x, unsigned e)
{
    assert(e < 64);

    uint64_t me = (1 << e) - 1;
    return x & me;
}
```

Il est également fréquemment utile lors de l'utilisation de *bitmasks*, pour ne conserver que les valeurs de certains bits bien choisis.

¶ **Opérateur |.** L'opérateur `|` (à ne pas confondre avec `||` !) est similaire à `&` mais pour le OU logique : l'expression `x | y` s'évalue à un entier dont chaque bit en position i vaut un ssi. au moins le bit de x en position i ou le bit de y en position i (et potentiellement les deux) valent un.

Cet opérateur (tout comme `^` ci-dessous, ou même `+` !) peut par exemple être utilisé en conjonction de `<<` et `>>` pour implémenter un *décalage circulaire* (ou « rotation ») qui applique une permutation cyclique aux bits d'un entier, comme l'illustre le programme ci-dessous (où il faut bien prendre garde à ne pas effectuer de décalage par une valeur égale à la longueur du type !) :

Programme 10.15

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

```c
uint64_t rol64(uint64_t x, unsigned r)
{
    r = r % 64; // ou & 0x3F, mais pas mieux et moins lisible
    return r == 0 ? x : (x << r) | (x >> (64 - r));
}

int main(void)
{
    for (unsigned i = 0; i < 70; i++)
    {
        printf("%064lb\n", rol64(~1UL, i));
    }
    return EXIT_SUCCESS;
}
```

¶ **Opérateur ^.** L'opérateur ^ ~~(à ne pas confondre avec ^^)~~ est similaire à & mais pour le OU exclusif (en anglais : *exclusive OR*, ou *XOR*) logique : l'expression `x ^ y` s'évalue à un entier dont chaque bit en position i vaut un ssi. *exactement un* bit parmi le bit de x en position i et le bit de y en position i vaut un.

Cette opération est linéaire pour les « entiers modulo 2 » (elle est linéaire sur $\mathbb{F}_2$), et est fréquemment utile en théorie des codes et en cryptographie.

10.4 Bitslicing

11 Test de programme

11.1 Mesure de temps de calcul

Table des matières

A Ressources

Le langage C étant *extrêmement* répandu, on trouve aisément de *très nombreuses* ressources à son sujet. La liste de suggestions suivante n'est donc évidemment pas complète, et même pas forcément très adaptée au cadre des CPGE. Elle se veut plus comme un point de départ pour approfondir le (très vaste) sujet.

(Mal)heureusement, la plupart de ces ressources sont en anglais, la *de facto lingua franca* de l'informatique et des sciences exactes.

¶ **Ressource réglementaire.** Le programme d'informatique de la filière MP2I/MPI. L'annexe A définit de façon limitative le fragment du langage C au programme.

¶ **Pages de manuel.** Les *pages de manuel* (en anglais : *manpages*) UNIX ont une section entière (la numéro 3) dédiée à la bibliothèque C standard. On trouve des version variées de ces pages sur l'internet, mais si vous disposez d'un environnement UNIX, le mieux est encore d'utiliser celles installées sur votre machine. Ces pages documentent à la fois des fonctions et des fichiers d'en-tête ; pour par exemple accéder à la page de documentation de `malloc` ou `string.h`, il suffit d'écrire `> man malloc` ou bien `> man string.h` dans un terminal.

¶ **SEI CERT C Coding Standard.** Le site `https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard` regroupe un certain nombre de règles et de conseils pour l'écriture de programmes C exempts de comportements non (ou mal) définis. Il s'adresse avant tout au développement dans un contexte professionnel, et ce serait probablement une mauvaise idée que de chercher à rigoureusement appliquer toutes ses règles dans le contexte des CPGE. C'est par contre une bonne ressource pour en apprendre plus sur le langage, et notamment ses plus de 200 comportements non définis.

¶ **`T-Snippet`.** T-Snippet est un analyseur statique C, disponible en ligne à l'adresse `https://tsnippet.trust-in-soft.com/#`. Il permet d'analyser de petits programmes complets et vise spécifiquement à détecter les comportements non définis.

¶ **Livres.** Aucun de ces livres n'est *vraiment* adapté au contexte des CPGE, mais ils ont l'avantage de présenter beaucoup d'informations intéressantes de façon contextualisée :

- Modern C. Gratuit, très complet, ravira tous les amateurs de corvidés, mais peu accessible pour débuter.
- Effective C. Payant, plein de bons conseils, peut-être plus accessible que le premier.
- Hacker's Delight. Payant, à peu près orthogonal aux CPGE, mais rigolo.

¶ **Norme C.** Les documents décrivant officiellement les différentes normes du C sont payants~~, ce qui veut dire que personne ne sait ce qu'ils contiennent~~, mais certaines versions préliminaires sont disponibles librement à l'adresse `https://iso-9899.info/wiki/The_Standard`. Quoi qu'il en soit, il est peu probable que leur lecture vous soit très utile pendant votre scolarité en CPGE.

B Shell UNIX

On décrit brièvement quelques aspects utiles des environnements en ligne de commande (les « *shells* ») UNIX, ainsi que quelques commandes utiles et utilitaires classiques.

§ **Shell & *prompts***

Il existe un certain nombre de *shells* UNIX (*sh*, *bash*, *zsh*, etc.) qui pour un usage basique sont très similaires. Les plus « récents » et courants proposent des « *prompts* » (invite de commande) enrichis (personnalisables) qui donnent un certain nombre d'informations utiles. Par exemple, une invite de commande par défaut de *bash* ressemble à :

Terminal B.1

```
[pierre@gma ~]$
```

où `pierre` est le nom de l'utilisateur exécutant le shell, `gma` celui de la machine, et `~` un raccourci pour `HOME` (*cf.* ci-dessous) qui indique le répertoire courant dans lequel le shell est situé.

On utilisera parfois un tel *prompt* pour illustrer l'effet de certaines commandes (par exemple `cd`), mais nous nous contenterons la plupart du temps du même *prompt* basique « `>` » que dans le reste de ces notes de cours.

§ **Documentation**

¶ **`man`.** Le programme utilitaire `man` permet d'afficher une page de documentation à propos du programme ou de la commande en argument. Par exemple :

Terminal B.2

```
> man man
```

affiche la page de manuel à propos de `man`.

Les pages de manuel sont organisées en *sections* ; la section 2 est dédiée aux *appels systèmes* (qui permettent de s'interfacer avec le (noyau du) système d'exploitation), et la section 3 aux fonctions de bibliothèque (notamment de la bibliothèque standard C). Dans le cas où plusieurs pages du même nom sont présentes dans différentes sections, on peut indiquer celle voulue en précédent l'argument du numéro de section. Par exemple :

Terminal B.3

```
> man 2 mmap
> man 3 mmap
```

affichent respectivement la page de l'appel système *mmap* et de la fonction du même nom de la bibliothèque C.

§ **Variables d'environnement**

Il est possible de définir des *variables d'environnement* dans un terminal, afin de contrôler l'exécution de certaines commandes ou de certains programmes. De la façon la plus simple, une variable d'environnement `VAR` se définit avec comme valeur la chaîne de caractères `"value"` comme (sans espaces !) :

Terminal B.4

```
> VAR="value"
```

et se référence comme `$VAR`, notamment pour son affichage avec `echo` (*cf.* ci-dessous).

Il existe un certain nombre de variables d'environnement prédéfinies pour des usages fort divers, comme par exemple `HOME` qui contient le chemin du répertoire d'accueil de l'utilisateur ou l'utilisatrice, ou `LINES` et `COLUMNS` qui contiennent respectivement le nombre de lignes et de colonnes du terminal.

§ **Affichage de texte**

¶ `echo`. La commande `echo` permet d'afficher du texte sur la sortie du terminal, ainsi que la valeur des variables d'environnement :

Terminal B.5

```
> echo "coucou"
coucou
> OHAI="ohai"
> echo $OHAI
ohai
```

§ **Arborescence**

¶ **Chemins.** Tous les répertoires et répertoires ont une adresse ou un *chemin*, ancré à la *racine de l'arborescence*, notée `/`. Un fichier `file` se trouvant immédiatement à la racine aura comme chemin `/file` ; un fichier `file` se trouvant dans le répertoire `dir2` qui se trouve lui-même dans le répertoire `dir1` qui se trouve immédiatement à la racine aura comme chemin `/dir1/dir2/file`.

¶ **Chemins relatifs.** Un chemin entièrement donné depuis la racine est dit *absolu*. On peut aussi donner un chemin *relativement* au répertoire dans lequel le shell est situé. Un fichier `file` se trouvant immédiatement dans le répertoire où le shell est situé aura comme chemin `file` (notez l'absence de « / ») ; un fichier `file` se trouvant dans le répertoire `dir` qui se trouve lui-même dans le répertoire où le shell est situé aura comme chemin `dir/file`.

¶ **Parent « . . ».** On peut « remonter » l'arborescence relative grâce à « `..` », qui désigne le répertoire *parent* de celui où le shell est situé. Par exemple, si le shell est placé dans un répertoire `dir2` qui se trouve dans un répertoire `dir1` lui-même à la racine, et que `dir1` possède un fichier `file`, on peut référencer `file` absolument comme `/dir1/file` ou relativement à `dir2` comme `../file`

¶ ***Self* « . ».** On peut de même utiliser `.` dans un chemin pour désigner le répertoire courant du chemin. Ainsi, `file` et `./file` désignent relativement le même fichier (s'il existe). Ceci est surtout utile pour exécuter un programme hors `PATH` (*cf. ci-dessous).

¶ **Joker « * ».** Dans un certain nombre de contextes, il est possible d'utiliser le caractère joker « * » pour désigner l'ensemble des fichiers et répertoires dont le chemin est le préfixe du caractère. Par exemple, `> *` désigne tous les fichiers et répertoires se trouvant dans le répertoire courant, et `/tmp/*` tous les fichiers et répertoires se trouvant dans le répertoire `/tmp`.

¶ **`HOME` et `~`.** Comme donné ci-dessus en exemple, la variable d'environnement `HOME` contient le chemin du répertoire d'accueil. Il est donc par exemple possible d'écrire un chemin absolu « relativement à `HOME` » comme `$HOME/reste_du_chemin`. Ce chemin désigne alors la même intention pour chaque utilisateur & utilisatrice, mais sera concrètement différent pour chacun & chacune. Par exemple, le même chemin `$HOME/.vim` désignera `/home/pierre/.vim` dans l'environnement de l'utilisateur `pierre`, mais `/root/.vim` dans l'environnement de l'utilisateur ou utilisatrice `root`.

Le caractère « `~` » peut généralement également être utilisé comme *alias* pour `$HOME`.

§ **Navigation**

¶ **`cd`.** La commande `cd` permet de changer le répertoire dans lequel le shell est situé, ce qui influence l'écriture des chemins relatifs. L'argument de `cd` doit être le chemin (absolu ou relatif) d'un répertoire, ou être laissé vide ce qui est équivalent à `cd ~`, ou être mis à `-` ce qui a pour effet de changer le répertoire au dernier dans lequel le shell était placé. On illustre ces différents mécanismes ci-dessous :

Terminal B.6

```
[pierre@gma ~]$ cd /tmp
[pierre@gma tmp]$ cd /usr/include
[pierre@gma include]$ cd -
[pierre@gma tmp]$ cd
[pierre@gma ~]$
```

¶ **`ls`.** La commande `ls` affiche les fichiers & répertoires contenus dans un répertoire. Si aucun argument ou option n'est donnée, ce répertoire est celui dans lequel le shell est situé, et seuls les fichiers & répertoires s'y trouvant « immédiatement » sont affichés. Sinon, `ls` affiche le contenu du répertoire fourni en argument :

Terminal B.7

```
> ls /usr
bin  include  lib  lib32  lib64  local  sbin  share  src
```

On peut également utiliser `ls` avec un certain nombre d'options, notamment « `-l` » (pour « liste ») qui enrichit l'affichage d'informations détaillées à propos de chaque fichier & répertoire, ainsi que « `-R` » (pour « récursif ») qui affiche également le contenu des répertoires affichés, et de ceux des répertoires dans ces répertoires etc. L'option `-a` permet d'afficher également les fichiers & répertoires cachés (*cf.* ci-dessous).

¶ **Fichiers & répertoires cachés.** Les fichiers & répertoires dont le nom commence par un « `.` » ne sont pas affichés par défaut par `ls`. Il s'agit en général de fichiers & répertoires à usage interne de certains programmes (par exemple à des fins de configuration), qu'un utilisateur ou utilisatrice est en principe rarement poussé à consulter ou modifier ; les cacher permet donc de limiter l'affichage aux éléments les plus fréquemment utiles, et améliore le confort d'utilisation.

¶ **`pwd`.** La commande `pwd` (« print working directory ») affiche le répertoire courant dans lequel le shell est situé.

Terminal B.8

```
> cd /tmp
> pwd
/tmp
```

§ **Manipulation de fichiers & répertoires**

¶ `mkdir`. La commande `mkdir` permet de créer un répertoire vide au chemin indiqué en argument. Si aucune option n'est fournie, le dernier répertoire du chemin donné en argument ne doit *pas* exister, et tous ceux le précédant *doivent* exister. L'option `-p` permet d'éviter ces deux contraintes.

¶ `rmdir`. La commande `rmdir` permet de supprimer un répertoire *vide* au chemin indiqué en argument.

¶ `rm`. La commande `rm` permet de supprimer des fichiers et des répertoires. Il convient d'être *extrêmement prudent* lors de son utilisation, car les fichiers sont supprimés irrémédiablement (sans passer par une « corbeille »), et certaines options permettent de supprimer un répertoire et *tout ce qu'il contient* (y compris les autres répertoires et tout ce qu'ils contiennent, etc.). Imaginez l'effet d'une telle invocation avec l'argument `$HOME`…

¶ `touch`. La commande `touch` permet de créer un fichier vide au chemin indiqué en argument si aucun fichier n'existe déjà, et sinon de mettre à jour la date de dernière modification de celui-ci.

¶ `cp`. La commande `cp` permet de copier un fichier ou un répertoire à un autre emplacement. Si aucune option n'est fournie `cp` ne permet de copier que des fichiers seuls ; le premier argument correspond alors à un chemin du fichier à copier, et le second à un chemin *où* le copier (attention au cas où l'on écraserait un fichier existant !). Ce second argument peut ou bien désigner un répertoire (dans ce cas le fichier y est copié avec son nom original), ou bien un répertoire suivi d'un nom de fichier (dans ce cas le fichier est copié dans le répertoire, avec ce nom).

L'option `-r` permet de copier un répertoire et tout son contenu, récursivement (c'est à dire que si le répertoire contient des répertoires, ceux-ci seront également copiés avec leur contenu, etc.).

¶ `mv`. La commande `mv` (« *move* ») permet de déplacer un fichier ou répertoire à un autre emplacement. Son fonctionnement est similaire à `cp`, si ce n'est qu'aucune option n'est nécessaire pour déplacer les répertoires. Cette commande permet également de procéder à un « renommage » : renommer un fichier ou un répertoire n'est pas différent que le déplacer au même chemin sous un autre nom.

¶ `ln`. La commande `ln` permet entre autres choses de créer un *lien symbolique*, ou « raccourci », vers un fichier ou répertoire. Un tel lien ne copie pas physiquement le fichier mais permet de le rendre accessible depuis un autre chemin (typiquement un autre chemin relatif plus aisé à utiliser). Ceci permet à la fois d'économiser de la place (si le fichier en question est volumineux) ou d'éviter à multiplier les versions d'un même fichier pouvant être mutualisé. Les liens symboliques sont également couramment utilisés lors de l'installation de programmes ou de bibliothèques, afin de les « enregistrer » sous un nom indépendant d'un numéro de version, par exemple.

La création d'un lien symbolique se fait de façon similaire à une copie, comme :

Terminal B.9

```
> ln -s chemin_source chemin_destination
```

§ **Exécution de programme**

¶ **PATH.** L'exécution d'un programme depuis un shell se fait légèrement différemment en fonction de si le fichier exécutable du programme se trouve dans certains répertoires bien connus ou « n'importe où ». Le premier cas correspond aux fichiers se trouvant dans l'un des répertoires listés par la variable d'environnement `PATH`, et il suffit dans ce cas de donner le nom du fichier (sans qu'il soit nécessaire de préciser l'intégralité de son chemin). Dans le second cas, il faut fournir un chemin (absolu ou relatif) *qui doit contenir au moins un symbole « / »*. Cette dernière règle vise à distinguer clairement deux programmes de même nom dont l'un se trouverait dans l'un des répertoires enregistrés par `PATH` et l'autre non, en s'assurant que le second ne puisse *jamais* être exécuté de la même façon que le premier (ce qui est une bonne chose à éviter pour des raisons de sécurité).

La seconde règle est celle qui s'applique généralement pour vos propres programmes, et explique pourquoi même lorsque le shell est placé dans le même répertoire qu'un exécutable `a.out` venant d'être crée, on ne peut (en général) pas exécuter ce dernier comme `> a.out`. Le « . » prend alors tout son intérêt : `./a.out` désigne trivialement le même chemin relatif que `a.out`, mais puisque le premier inclut un caractère « / » il devient un chemin acceptable pour l'exécution du programme.

¶ **« ? ».** La variable d'environnement « ? » s'évalue en la valeur de retour du dernier programme exécuté. Elle permet ainsi par exemple de distinguer un programme C ayant renvoyé `EXIT_SUCCESS` (qui se traduit par un `0` dans le shell) ou `EXIT_FAILURE` (traduit par un `1`). Les programmes utilitaires `true` et `false` correspondent respectivement à chacun de ces cas :

Terminal B.10

```
> true
> echo $?
0
> false
> echo $?
1
```

On en profite pour remarquer que la correspondance entre *true*, *false*, *0* et *1* est ici l'inverse de celle utilisée dans le langage C lui-même pour les valeurs de type `bool`.

¶ **`htop`.** L'utilitaire `htop` affiche joliment des informations à propos des programmes en cours d'exécution sur le système, notamment le temps depuis lequel ils s'exécutent, leur consommation mémoire et à quel point ils sollicitent le processeur. Il affiche également des informations détaillées sur les ressources utilisées par l'ensemble des programmes.

§ **Redirections**

¶ **>, 2>, >>, 2>>.** Lors d'une exécution d'un programme ou d'une commande depuis le shell, il est possible de *rediriger* la sortie standard `stdout` et la sortie d'erreur `stderr` vers des fichiers. Les redirections de la sortie standard se font avec « > » ou « >> » en fonction de si le fichier pour la redirection est supprimé au

cas où il existerait déjà (dans le premier cas) ou si la sortie est ajoutée à un éventuel fichier déjà existant (dans le second). Les redirections de la sortie d'erreur se font avec « `2>` » et « `2>>` » sur le même mode.

L'exemple suivant illustre utilise une redirection de la sortie d'erreur vers `/dev/null` (décrit un peu plus bas) et de la sortie standard vers un fichier « `ls.out` ».

Terminal B.11

```
> ls > ls.out 2> /dev/null
```

¶ **`<`.** Il est également possible de substituer un fichier à l'entrée standard `stdin` en utilisant « `<` ». Toutes les éventuelles lectures de l'entrée standard seront alors « alimentées » par le fichier fourni, jusqu'à ce que sa fin soit atteinte. Attention cependant à ne pas confondre l'entrée standard (qui n'est pas forcément lue par un programme) et les arguments passés en ligne de commande.

L'exemple suivant illustre l'utilisation d'une telle redirection avec `cat` (décrit un peu plus bas) et un fichier contenant une unique ligne « OHAI ! » ; dans ce cas ; une redirection est en fait assez inutile.

Terminal B.12

```
> cat < ohai
OHAI!
```

¶ **`|`.** Enfin, il est possible de rediriger la sortie standard d'un programme ou d'une commande vers l'entrée standard d'un autre, en utilisant le *pipe* « `|` ». L'effet de `> a | b` est similaire à l'enchaînement de `> a > a.out` et `> b < a.out`, mais évite la création d'un fichier intermédiaire (ce qui peut être pratique, notamment quand la sortie de `a` est potentiellement infinie). Il est également possible « d'enchaîner » les *pipes* au delà de deux programmes ou commandes. L'exemple ci-dessous utilise `grep` (décrit un peu plus bas) pour recherche un motif dans la sortie de `ls` :

Terminal B.13

```
> ls | grep "CP"
CPGE
```

§ **Utilitaires pour fichiers textes**

On décrit brièvement quelques utilitaires classiques, dans leur usage le plus élémentaire. Pour plus d'informations à leur sujet, consultez les pages de manuel !

¶ **`cat`.** Le programme utilitaire `cat` n'a qu'un rapport distant avec les chats. Il sert avant tout à con*cat*éner les fichiers passés en arguments et à afficher le résultat sur la sortie standard. Si aucun argument n'est fourni, `cat` affiche le résultat de l'entrée standard.

¶ **`less`.** Le programme utilitaire `less` affiche le contenu d'un fichier teste dans un environnement ressemblant à celui proposé par les éditeurs de texte pour terminaux. Ceci permet notamment de facilement naviguer dans un grand fichier, et par exemple d'y rechercher une chaîne de caractères. Ce programme (ou d'autres similaires) est aussi typiquement utilisé pour afficher les pages de manuel.

¶ **`wc`.** Le programme utilitaire `wc` (pour *wordcount*) affiche des statistiques sur le nombre de lignes, mots et caractères du fichier texte fourni en argument.

¶ **`grep`.** Le programme utilitaire `grep` permet d'effectuer des recherches de texte avancé suivant des *motifs* spécifiés en argument, dans le fichier texte dont un chemin constitue le dernier argument. Les motifs peuvent notamment être décrits par des *expressions régulières*, qui sont au programme de seconde année de CPGE.

¶ **`diff`.** Le programme utilitaire `diff` permet d'afficher les différences entre deux fichiers texte. Les utilisateur et utilisatrices familières de `vim` pourront également utiliser `vimdiff` pour à peu près le même usage.

¶ **`head` & `tail`.** Les programmes utilitaires `head` et `tail` permettent d'afficher respectivement les quelques premières et dernières lignes d'un fichier texte.

§ **Divers**

¶ **Autocomplétion.** Les shells courants proposent des mécanismes d'*autocomplétion* qui accélèrent grandement l'écriture des commandes et des chemins : il suffit d'appuyer sur la touche de tabulation pour que le shell propose différentes façons de compléter le nom de commande ou de programme ou de chemin partiellement écrit ; si une seule façon de compléter le dernier mot écrit est trouvée, celui-ci est effectivement complété, et sinon il est en général possible d'afficher une liste de candidats et d'alterner entre eux en appuyant à nouveau sur la touche de tabulation.

¶ **Historique.** Les shells courants gardent un historique des dernières actions effectuées. Il est possible d'y naviguer en utilisant les touches de flèches haut et bas, et d'y effectuer une recherche en utilisant la combinaison de touches « `Ctrl + R` ».

¶ **`wget`.** Le programme utilitaire `wget` télécharge le fichier dont l'URL est donnée en argument. Il est notamment pratique pour ainsi récupérer des documents dont l'adresse est connue sans passer par un navigateur WEB.

¶ **`/tmp`.** Le répertoire de chemin `/tmp` est un répertoire temporaire dont le contenu est effacé chaque fois que l'ordinateur s'éteint. Il économise ainsi la suppression manuelle de fichiers dont la durée d'utilisation n'excède pas celle d'une session.

¶ **`/dev/null`.** Le *pseudo-fichier* `/dev/null` supprime tout contenu qui y est écrit. Il est ainsi couramment utilisé comme destination de redirection d'un flux de sortie que l'on ne souhaite pas conserver, comme fait un peu plus haut.

C Représentation des nombres entiers