3TC31 - Introduction au langage C

Guillaume Duc

Le fameux polycopié de la deuxième partie du module 3TC31

August 24, 2026

Table des matières

1. Note pour les élèves

La deuxième partie du cours 3TC31 vise à vous apprendre les bases de la programmation dans le langage C. Les supports de cours se basent sur un livre disponible au CRDN : *Effective C - An Introduction to Professional C Programming*, Robert C. Seacord, No Starch Press, 2020, ISBN 978-1-71850-104-1. Ce livre est récent, bien écrit et complet, permettant ainsi à la fois de comprendre certaines parties du cours qui auraient été trop rapidement abordées à l'oral mais également d'aller plus loin et de voir d'autres notions que celles strictement au programme. Néanmoins, pendant plusieurs années, vos prédécesseurs n'ont eu de cesse de demander un polycopié, visiblement un objet rassurant à consulter la veille de l'examen. Ce document permet donc de combler ce vide.

Avertissement : Ce document couvre les notions au programme de 3TC31, qui sont un sous-ensemble du langage C. De nombreuses notions ou possibilités du langage ne sont donc pas présentées ici. De plus, certains points sont volontairement simplifiés pour des raisons pédagogiques.

Malgré une relecture attentive, des erreurs peuvent subsister dans ce document, n'hésitez pas à nous les signaler.

2. Introduction

2.1. Architecture Harvard vs. Architecture von Neumann

Le processeur construit dans la première partie du module interagissait avec deux mémoires :
- Une mémoire contenant les instructions à exécuter (représentées en code machine, c'est-à-dire pour chaque instruction, les 32 bits représentant celle-ci ainsi que ses arguments)
- Une mémoire contenant les données (le processeur interagit avec celle-ci lors de l'exécution des instructions *load* et *store*)

Une architecture où le programme (ses instructions) et ses données sont stockés dans deux mémoires séparées est appelée une architecture *Harvard*.

À l'opposé, il est possible d'avoir une seule mémoire, contenant à la fois les instructions et les données. Il s'agit des architectures de *von Neumann*. Elles sont très largement majoritaires actuellement et donc nous nous placerons dans ce modèle pour la suite du cours.

Note (hors programme) : pour être précis, la majorité des architectures actuelles sont hybrides. Le cœur même du processeur a deux bus mémoire distincts, un pour les instructions et un pour les données, et pourtant il n'y a réellement qu'une seule mémoire stockant instructions et données. L'unification se fait traditionnellement au niveau des caches du processeur (les caches de premier niveau étant généralement séparés, les caches de niveaux supérieurs unifiés).

On considérera donc dans la suite qu'il n'y a qu'une seule mémoire, contenant à la fois les instructions et les données.

2.2. Du code source à l'exécution : la compilation

Comme nous l'avons vu dans la première partie du cours, pour que le processeur exécute un programme, il faut :
- charger en mémoire les instructions du programme, sous forme de code machine ;
- prévoir de la place en mémoire pour les données (variables du programme notamment) et éventuellement charger leurs valeurs initiales ;
- et finalement mettre l'adresse de la première instruction du programme dans le compteur ordinal (*program counter*) du processeur.

Toutes ces étapes sont fastidieuses. Par exemple, écrire du code machine, certes directement exécutable par le processeur mais illisible pour un humain, est fastidieux et irréaliste au delà de quelques instructions. Une première étape, vue dans la première partie du module, consiste à écrire le programme en langage assembleur. Ce langage assembleur peut alors être traduit automatiquement par un outil, l'assembleur, vers du code machine prêt à être chargé en mémoire pour être exécuté. Néanmoins, écrire un programme en assembleur reste un exercice long et pénible. En effet, le langage assembleur n'utilise que les instructions disponibles sur le processeur, qui sont extrêmement simples (arithmétique, logique, chargement mémoire, sauts), et ne propose pas de construction de plus haut niveau (boucles, fonctions, etc.), ne connaît pas la notion de variable ni de type, etc.

Pour faciliter le développement, des langages de haut niveau, comme le C (mais il en existe des centaines), ont été inventés.

Pour passer du code source écrit en langage C vers un programme qui s'exécute, il y a plusieurs étapes :

- le code source est écrit (avec un éditeur de texte) dans un (ou plusieurs) fichier ;
- puis le (ou les) fichier contenant le code source est *compilé* (transformé) par un outil nommé *compilateur* en un fichier *exécutable* ;
- et enfin, lorsque l'utilisateur veut exécuter le programme, le fichier exécutable est chargé en mémoire par un composant du système d'exploitation : le *chargeur* (*loader*).

Le fichier exécutable, qui est produit par le compilateur, contient :

- du code machine, résultat de la traduction de la partie du code source décrivant des opérations à réaliser en instructions pour le processeur ;
- et des informations concernant les données manipulées par le programme (tailles, adresses, valeurs initiales).

Pour exécuter le programme à partir du fichier exécutable, le chargeur du système d'exploitation (à noter que ces étapes seront détaillées dans la troisième partie du cours) :

- place en mémoire les instructions en code machine ;
- réserve la mémoire pour les données (et place éventuellement les valeurs initiales) ;
- et met l'adresse de la première instruction à exécuter dans le compteur ordinal du processeur.

2.3. Pourquoi le langage C ?

Le langage C est depuis de très nombreuses années, un langage extrêmement populaire.

C'est un langage bas niveau permettant d'utiliser directement certaines ressources de l'ordinateur, ce qui en fait un langage adapté pour développer des systèmes d'exploitation, programmer des systèmes embarqués, etc.

Le C est un langage simple à prendre en main. Néanmoins, cette simplicité apparente est en fait un gros défaut pour les débutants. Il est très facile d'écrire des programmes contenant des bugs, qui vont provoquer des plantages, voire pire, des failles de sécurité. De plus, le compilateur est très permissif, et n'est pas d'une grande aide pour détecter les erreurs.

2.4. Bref historique du langage C

- 1972: Invention par *Dennis Ritchie* et Ken Thompson (Bell Telephone Laboratories) pour développer leur propre système d'exploitation … Unix (voir troisième partie du cours)
- 1989: Premier standard (ANSI C ou C89), adopté par l'ISO l'année suivante (C90)
- 1999: Nouveau standard ISO (C99) - très largement supporté (type Boolean, types entiers avec tailles fixes)
- 2011: Nouveau standard ISO (C11) - bien supporté de nos jours (support de l'Unicode, support du multi-threading et des types atomiques)
- 2017: Nouveau standard ISO (C17) - principalement des corrections
- 2024: Nouveau standard ISO (C23)

2.5. Standards

Les standards définissent le langage (sa grammaire notamment) et le contenu de la bibliothèque standard (un ensemble de types et de fonctions utiles).

Il y a parfois des différences entre le standard et son implémentation par les compilateurs : les compilateurs n'implémentent pas nécessairement l'intégralité des standards (en particulier les standards les plus récents) et ajoutent également des extensions non standardisées. De plus, les standards laissent sur certains points une marge de manœuvre aux compilateurs.

En particulier, on distingue les comportements :

- *Définis par l'implémentation* (*implementation-defined*) : comportements qui ne sont pas précisés dans le standard et donc sont laissés au choix de l'implémentation qui doit les documenter. Il s'agit souvent de comportements liés à l'architecture vers laquelle le programme est compilé, comme par exemple, la taille de certains types (`char`, `int`).
- *Non spécifiés* (*unspecified*) : comportements laissés libre au compilateur, et qui peut varier, y compris au sein d'un même programme, notamment pour des raisons d'optimisation, comme par exemple l'ordre d'évaluation des arguments d'une fonction.
- *Non définis* (*undefined*) : comportements non prévus par le standard en cas de programmes erronés. Le compilateur est libre de ne rien faire, de générer une erreur, de produire du code qui va planter… Ces situation sont à éviter à tout prix. Exemple : déréférencement d'un pointeur `NULL`, débordement de la valeur d'un entier…

Dans la suite du cours, nous utiliserons le standard C11, relativement moderne et très bien supporté par les compilateurs.

2.6. En pratique : compiler un premier programme (sous Linux/Unix)

On suppose que le fichier `hello.c` contient le code source suivant (on détaillera ultérieurement sa signification) :

```c
/* hello.c */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  printf("Hello world!\n");
  return EXIT_SUCCESS;
}
```

Pour pouvoir exécuter ce premier programme, il faut le compiler pour produire un fichier exécutable puis demander l'exécution de ce fichier exécutable :

```
ouessant:~/tmp$ ls
hello.c

ouessant:~/tmp$ gcc -Wall -pedantic -std=c11 -g -o hello hello.c

ouessant:~/tmp$ ls
hello  hello.c

ouessant:~/tmp$ ./hello
Hello world!
```

Pas de panique, nous allons expliquer pas-à-pas l'ensemble de ces commandes.

2.6.1. Explications pas à pas

```
ouessant:~/tmp$ ls
hello.c
```

- `ouessant:~/tmp$` est l'invite de commande (*prompt*) affichée par votre interpréteur de commande (*shell*) pour vous… inviter à taper une commande. Elle peut varier en fonction de votre configuration et est donc représentée par `$` dans la suite du cours.
- `ls` est la commande tapée par l'utilisateur. Utilisée sans arguments, cette commande affiche la liste des fichiers contenus dans le répertoire courant.
- `hello.c` est le résultat affiché par cette commande : il y a donc un seul fichier dans le répertoire courant et il est nommé `hello.c` (c'est le fichier contenant le code source).

```
ouessant:~/tmp$ gcc -Wall -pedantic -std=c11 -g -o hello hello.c
```

- `gcc` est le compilateur que l'on utilisera en TP. Il est suivi d'un certain nombre d'arguments qui permettent de contrôler son exécution :
  - `-Wall` active tous les avertissements liés à des constructions souvent problématiques
  - `-pedantic` active les avertissements liés au respect strict du standard
  - `-std=c11` indique au compilateur de se baser sur le standard C11
  - `-g` indique au compilateur d'ajouter des informations au fichier exécutable pour faciliter le débogage par la suite
  - `-o hello` indique que le fichier exécutable produit doit s'appeler `hello` (sinon par défaut il s'appellera `a.out`)
  - `hello.c` indique le nom du fichier source à compiler
- Le compilateur n'affiche rien, tout s'est donc bien passé. Si ce n'était pas le cas, lisez attentivement les messages, en commençant toujours par le début (et non la fin).

```
ouessant:~/tmp$ ls
hello  hello.c
```

- `ls` nous montre maintenant qu'il y a deux fichiers dans le répertoire courant :
  - `hello` le fichier exécutable produit par le compilateur
  - `hello.c` le fichier source présent précédemment

```
ouessant:~/tmp$ ./hello
Hello world!
```

- Pour lancer l'exécution de notre programme, il faut taper son nom : `hello`
- Cependant, pour des raisons expliquées ultérieurement, il faut dans notre cas préciser explicitement son emplacement d'où `./hello` :
  - `.` est le répertoire courant
  - `/` est le délimiteur de chemin
  - `hello` est le nom de l'exécutable
  - `./hello` est donc un chemin relatif : le fichier nommé `hello` dans le répertoire courant
- `Hello world!` a été affiché par notre programme

Attention : C'est le fichier exécutable qui est exécuté, pas le code source (une fois compilé, vous pourriez supprimer le fichier source). Si vous changez le code source, il faut explicitement le compiler de nouveau pour mettre à jour l'exécutable. C'est un piège classique : vous

corrigez le code source mais oubliez de recompiler et vous ne comprenez pas pourquoi vos modifications ne sont pas prises en compte.

2.7. Les commentaires

Le premier élément de syntaxe que nous aborderont sont les commentaires.

Il existe deux façons d'introduire des commentaires dans un fichier source en C :
- `//` introduit un commentaire qui va jusqu'à la fin de la ligne
- `/*` introduit un commentaire qui va jusqu'à rencontrer `*/`

Exemple :

```
int a; // Ceci est un commentaire qui va jusqu'à la fin de la ligne

/*
Tout ceci est un commentaire
Encore le même commentaire
Toujours
*/
```

Le contenu des commentaires est tout simplement ignoré par le compilateur.

Ils sont néanmoins très utiles pour le développeur. N'oubliez pas qu'on passe plus de temps à lire/relire un programme pour essayer de le comprendre (ou de trouver des erreurs) qu'à l'écrire.

3. Les types de base

3.1. Introduction

Le *type* d'une donnée définit la nature des valeurs que peut prendre la donnée, ainsi que les opérations qui peuvent s'appliquer sur cette données.

Parmi les langages de programmation, on distingue :
- *typage dynamique* : le type des données est déterminé et vérifié à l'exécution (lorsque le programme s'exécute). C'est notamment le cas de Python, JavaScript, etc.
- *typage statique* : le type des données est déterminé et vérifié à la compilation. C'est le cas du langage C, de Rust, C++, etc.

Nous verrons dans ce chapitre les types de base du langage. Les types composés (tableaux, structures, pointeurs, etc.) seront abordés ultérieurement.

3.2. Le type `void`

Le type `void` est un peu particulier puisqu'il ne possède aucune valeur.

Il est utilisé :
- pour indiquer qu'une fonction ne renvoie pas de valeur ;
- pour indiquer qu'une fonction ne prend pas d'argument ;
- et pour créer des pointeurs dont le type de l'objet pointé n'est pas spécifié.

3.3. Le type booléen

Depuis le standard C99, le C possède le type `_Bool` capable de stocker deux valeurs : 0 et 1.

Le fichier d'entête `stdbool.h` définit trois macros :
- `bool` qui est un alias du type `_Bool` (il est donc possible d'écrire `bool` au lieu de `_Bool`) ;
- `true` qui est évaluée à la valeur 1 ;
- `false` qui est évaluée à la valeur 0.

Pour utiliser ces macros, il suffit d'inclure le fichier d'en-tête `stdbool.h` à votre programme en ajoutant `#include <stdbool.h>` en début du fichier.

Hors programme : dans le standard C23, `bool` est devenu un type à part entière (équivalent à `_Bool`) et `true` et `false` des mots clés. Le fichier d'en-tête n'est donc plus nécessaire.

3.4. Les types entiers

3.4.1. Les types signés

Le C définit 5 types entiers signés (pouvant stocker des entiers négatifs et positifs) standards :

Type	Alias	Plage minimale représentable d'après le standard
`signed char`		-127 à 127 $-(2^7-1)$ à 2^7-1

Type	Alias	Plage minimale représentable d'après le standard

`short int`	`short`
	`signed short`
	`signed short int`
		-32.767 à 32.767
		$-(2^{15}-1)$ à $2^{15}-1$

`int`	`signed` `signed int`	-32.767 à 32.767 $-(2^{15}-1)$ à $2^{15}-1$

`long int`	`long`
	`signed long`
	`signed long int`
		$-2.147.483.647$ à $2.147.483.647$
		$-(2^{31}-1)$ à $2^{31}-1$

`long long int`	`long long`
	`signed long long`
	`signed long long int`
		$-9.223.372.036.854.775.807$
		à $9.223.372.036.854.775.807$
		$-(2^{63}-1)$ à $2^{63}-1$

Le standard donne des contraintes sur ces types :

- Une plage minimale de valeurs représentables pour chacun de ces types (la plage exacte dépend de l'implémentation)
- Chaque type doit pouvoir représenter au moins toutes les valeurs représentables par le type précédent (dans l'ordre du tableau ci-dessus)
- Un `signed char` doit occuper la même place en mémoire qu'un `char` (voir plus loin)
- Un `int` a la taille naturelle des entiers sur l'architecture d'exécution

Par contre, le standard ne précise pas comment sont représentés les nombres en mémoire (complément à deux, signe et valeur absolue, etc.). Néanmoins, la totalité des architectures récentes utilise la représentation en complément à deux pour les nombres signés.

Comme nous venons de le voir, la plage des valeurs représentables dépend de l'architecture cible de notre programme. Sur les machines de TP de l'école, on obtient :

Type	Taille (en bits)	Plage représentable
`signed char`	8	-128 à 127 -2^7 à 2^7-1
`short int`	16	-32.768 à 32.767 -2^{15} à $2^{15}-1$
`int`	32	$-2.147.483.648$ à $2.147.483.647$ -2^{31} à $2^{31}-1$
`long int`	64	$-9.223.372.036.854.775.808$ à $9.223.372.036.854.775.807$ -2^{63} à $2^{63}-1$
`long long int`	64	$-9.223.372.036.854.775.808$ à $9.223.372.036.854.775.807$ -2^{63} à $2^{63}-1$

3.4.2. Les types non signés

Le C définit également 5 types non signés (ne pouvant donc stocker que des entiers positifs) :

Type	Alias	Plage minimale représentable d'après le standard
`unsigned char`		0 à 255 0 à $2^8 - 1$
`unsigned short int`	`unsigned short`	0 à 65.535 0 à $2^{16} - 1$
`unsigned int`		0 à 65.535 0 à $2^{16} - 1$
`unsigned long int`	`unsigned long`	0 à 4.294.967.295 0 à $2^{32} - 1$
`unsigned long long int`	`unsigned long long`	0 à 18.446.744.073.709.551.615 0 à $2^{64} - 1$

Comme pour les types signés, le standard ne précise qu'une plage minimale de valeurs représentables, la plage exacte dépendant de l'architecture cible. Sur les machines de TP de l'école, on obtient :

Type	Taille (en bits)	Plage représentable
`unsigned char`	8	0 à 255 0 à $2^8 - 1$
`unsigned short int`	16	0 à 65.535 0 à $2^{16} - 1$
`unsigned int`	32	0 à 4.294.967.295 0 à $2^{32} - 1$
`unsigned long int`	64	0 à 18.446.744.073.709.551.615 0 à $2^{64} - 1$
`unsigned long long int`	64	0 à 18.446.744.073.709.551.615 0 à $2^{64} - 1$

3.4.3. Aparté : Littéraux entiers

Un littéral est une valeur fixe dans le code source, par exemple la valeur `42` dans le code suivant :

```
a = 42;
```

Par défaut, les littéraux entiers sont exprimés en base 10. Donc `42` ici représente bien $4 \times 10 + 2$.

Il est possible d'écrire un littéral entier dans d'autres bases en le préfixant avec :

- `0x` : la base utilisée est alors la base 16 (hexadécimale), exemple `0x1A` qui représente la valeur 26 en base 10
- `0` : la base utilisée est alors la base 8 (octale), exemple `010` qui représente la valeur 8 en base 10

Attention : la base est celle dans laquelle est représentée (écrite) la valeur dans le code source. Quoi qu'il arrive, elle sera stockée en mémoire et manipulée dans le processeur en base 2.

Il est possible d'ajouter un suffixe à la constante pour guider son typage :
- `u` (ou `U`) : le type du littéral sera non signé
- `l` (ou `L`) : le type du littéral sera au moins `long`
- `ll` (ou `LL`) : le type du littéral sera `long long`

Des règles standardisées, dépendante de la base dans laquelle la constante est exprimée et d'un éventuel suffixe, déterminent le type exact du littéral. Ces règles dépassent le cadre de ce cours.

3.5. Les types flottants

Un nombre flottant, ou plus précisément à virgule flottante, est un nombre réel représenté sous la forme $s \times m \times 2^e$ où s représente le *signe* (-1 ou 1), m la *mantisse* (avec le plus souvent $1 \leq m < 2$) et e l'*exposant* (entier signé). La représentation et la manipulation de ces nombres sont spécifiées dans le standard IEEE 754, implémenté dans de nombreuses architectures.

Le C dispose de trois types flottants :
- `float` : simple précision (en général 32 bits)
- `double` : double précision (en général 64 bits)
- `long double` : précision étendue (128 bits si supporté, sinon au moins 64 bits)

3.5.1. Littéraux flottants

Tout comme les littéraux entiers, il est possible d'écrire une valeur flottante dans le code.

Quelques exemples :
- `.5` $(0,5)$
- `1.` (1)
- `1.3e3` (1300)
- `2e-1` $(0,2)$

Par défaut, ces littéraux sont de type `double`. Un suffixe peut être ajouté pour modifier ce comportement : `f` pour `float` et `l` pour `long double`.

3.6. Les types caractères

Historiquement, le stockage et la représentation des caractères, en particulier pour ceux en dehors du jeu de caractères de base (lettres latines, chiffres et quelques symboles de ponctuation), a été un sujet complexe et une source d'incompatibilités. Cette complexité se retrouve dans le langage C du fait de son ancienneté.

3.6.1. Le type `char`

Le type `char` est défini comme un type devant pouvoir stocker au minimum les caractères d'un jeu de caractères de base (lettres latines minuscules et majuscules, chiffres et certains symboles).

Au sein d'un variable de type `char`, les caractères sont représentés sous la forme d'un nombre entier. Le standard précise que ces caractères de base doivent avoir une valeur non négative (donc positive ou nulle).

En fonction de la plate-forme, `char` est équivalent à un `signed char` ou un `unsigned char` (les types entiers vus précédemment).

Classiquement, `char` est stocké sur un octet (8 bits) et peut donc stocker des valeurs entre −128 et +127 ou entre 0 et 255.

Classiquement, la table ASCII est utilisée pour faire correspondre les caractères de base et des entiers. Cette table fait la correspondance entre un caractère couvrant le jeu de caractères de base et des entiers entre 0 et 127. Par exemple, le caractère `a` est représenté par la valeur 97.

3.6.2. Les littéraux caractères

Il est possible d'exprimer un caractère littéral dans le code source en l'encadrant entre `'`, exemple : `'a'` (valeur 97), `'0'` (valeur 48).

Certains caractères spéciaux sont difficilement représentables et nécessite l'utilisation d'une séquence d'échappement. Par exemple, le littéral `'\n'` (valeur 10) représente le caractère de contrôle *Line Feed* (saut de ligne).

Les principales séquences d'échappement sont :

Séquence	Description	Code ASCII
`'\n'`	*Line Feed* (saut de ligne)	10
`'\r'`	*Carriage Return* (retour à la ligne)	13
`'\''`	Simple quote (`'`)	39
`'\"'`	Double quote (`"`)	34
`'\\'`	Backslash (`\`)	92
`'\t'`	Tabulation horizontale	9
`'\`n`'`	Caractère de valeur n (en octale)	n

3.6.3. Les types caractères étendus (hors programme)

Pour stocker des jeux de caractères en dehors des caractères de base (par exemple les lettres accentuées, l'alphabet grec, les émojis…), plusieurs types ont été introduits au cours du temps :

- `wchar_t` (défini dans `wchar.h`, C95) : type caractère étendu, taille définie par l'implémentation

- `char16_t` (défini dans `uchar.h`, C11) : taille 16 bits, permettant de stocker des caractères Unicode avec la représentation UTF-16 (c'est un encodage à taille variable donc un caractère unicode peut nécessiter plusieurs `char16_t` pour être représenté)
- `char32_t` (défini dans `uchar.h`, C11) : taille 32 bits, permettant de stocker des caractères Unicode avec la représentation UTF-32
- `char8_t` (défini dans `uchar.h`, C23) : taille 8 bits, permettant de stocker des caractères Unicode avec la représentation UTF-8 (c'est un encodage à taille variable donc un caractère unicode peut nécessiter plusieurs `char8_t` pour être représenté)

3.7. Les chaînes de caractères littérales

Les chaînes de caractères (zéro, un ou plusieurs caractères qui se suivent) littérales sont écrites entre doubles quotes (`"`).

Par exemple :
- `""` : chaîne vide
- `"a"` : chaîne contenant un seul caractère (attention, c'est différent de `'a'`)
- `"Vive le C"` : chaîne contenant plusieurs caractères

Le type des chaînes de caractères sera expliqué plus loin. Dans un premier temps, on considérera qu'il s'agit d'un tableau de `char`.

4. Déclarations et définitions de variables et de fonctions

4.1. Généralités

Une *déclaration* permet d'introduire dans le programme un identifiant et de spécifier sa signification et ses propriétés.

Une *définition* est une déclaration qui en plus :
- pour une fonction, précise le corps de la fonction ;
- pour une variable, demande au compilateur de prévoir l'allocation mémoire.

Pour chaque identifiant déclaré (ou défini), on associe plusieurs propriétés :

- Tout d'abord sa *portée* (*scope*), c'est-à-dire la portion du programme où l'identifiant est visible et donc utilisable. Dans le cadre de ce cours, on distingue trois portées : un fichier source, une fonction ou un bloc de code.
- Ensuite son comportement lors de l'édition des liens ou liaison (*linkage*). On distingue trois comportements : liaison externe (*external linkage*), liaison interne (*internal linkage*) et sans liaison (*no linkage*). Au sein de l'ensemble d'un programme (pouvant s'étendre sur plusieurs fichiers sources), toutes les déclarations d'un identifiant avec liaison externe correspondent au même objet ou fonction. Au sein d'un fichier source (unité de traduction), toutes les déclarations d'un identifiant avec liaison interne correspondent au même objet ou fonction. Une déclaration d'un identifiant sans liaison correspond à un objet unique.
- Enfin une *durée de stockage* ou *durée de vie* (*storage duration*). Cette durée représente l'intervalle de temps durant lequel l'objet est utilisable (il existe, a une adresse fixe et conserve sa valeur). Dans le cadre de ce cours, on distingue deux durées de stockage : *automatique* (jusqu'à la fin du bloc dans lequel il est défini) et *statique* (toute la vie du programme).

Enfin, on distingue les déclarations *globales* (au niveau d'un fichier source, en dehors de toute fonction) et *locales* (à l'intérieur du corps d'une fonction).

4.2. Identifiants

En C un identifiant doit commencer par une lettre ou un caractère `_` et se poursuit ensuite par zéro ou plus lettres, chiffres et caractère `_`.

L'utilisation d'un mot clé du langage comme identifiant est interdit.

4.3. Déclarations et définitions de variables

La syntaxe de base d'une déclaration de variable est la suivante :

`<Type> <Identifiant>;`

par exemple :

```
// Déclaration (définition) d'une variable i de type int
int i;
```

L'exemple déclare (en réalité définit mais on y reviendra juste après) une variable dont l'identifiant est `i` et son type `int`.

À partir de cette déclaration, et jusqu'à la fin de la portée de l'objet (ici la variable), le compilateur connaît l'identifiant `i` et son type. Quand il rencontrera cet identifiant, il est donc capable de faire les vérifications imposées par le système de type, les conversions implicites si nécessaire et générer le code correspondant aux opérations sur cette variable.

Tout identifiant doit être connu par le compilateur au moment où il le rencontre. Chaque identifiant doit donc être déclaré, ou défini, avant sa première utilisation.

Il est possible de préfixer la déclaration par un ou plusieurs mots clés :
- `const` : la valeur de la variable ne peut pas être modifiée ultérieurement (*constante*)
- `static` : change la durée de stockage de la variable ou sa liaison (voir plus bas)
- `extern` : change la liaison de la variable (voir plus bas)

Localisation	Mots clés	Portée	Liaison	Durée de stockage	Déclaration ou définition
Globale (à l'extérieur d'une fonction)	Aucun	Fichier (jusqu'à la fin du fichier)	Externe	Statique (jusqu'à la fin de l'exécution du programme)	Définition (allocation mémoire)
	`extern`	Fichier (jusqu'à la fin du fichier)	Externe	Statique (jusqu'à la fin de l'exécution du programme)	Déclaration uniquement (pas d'allocation mémoire)
	`static`	Fichier (jusqu'à la fin du fichier)	Interne	Statique (jusqu'à la fin de l'exécution du programme)	Définition (allocation mémoire)
Locale (à l'intérieur d'une fonction)	Aucun	Bloc (jusqu'à la fin du bloc)	Sans	Automatique (jusqu'à la fin du bloc)	Définition (allocation mémoire)
	`extern`	Bloc (jusqu'à la fin du bloc)	Externe (sauf redéclaration d'un identi-	Statique (jusqu'à la fin de l'exécution	Déclaration uniquement (pas d'allocation mémoire)

Localisation	Mots clés	Portée	Liaison	Durée de stockage	Déclaration ou définition
			fiant déjà interne)	du programme)	
	`static`	Bloc (jusqu'à la fin du bloc)	Sans	Statique (jusqu'à la fin de l'exécution du programme)	Définition (allocation mémoire)

Sauf cas particuliers, l'utilisation du mot clé `extern` réalise uniquement une *déclaration*, c'est-à-dire que le compilateur ne prévoit pas, à ce moment là, de mémoire pour la variable (autrement dit ne réserve pas de mémoire). Pour que la variable soit utilisable, il faut qu'une définition soit réalisée ailleurs (en général dans un autre fichier source constituant le même programme). Pour simplifier, `extern` dit au compilateur : la variable est déjà définie ailleurs (dans un autre fichier), donc tu n'as pas besoin de réserver une deuxième fois la mémoire pour cette variable.

Au contraire, sans l'utilisation du mot clé `extern`, on réalise une *définition*, c'est-à-dire une déclaration et en plus, une demande au compilateur d'allouer (réserver) de la mémoire pour cette variable. Il ne doit y avoir qu'une seule définition pour chaque variable (dans le cas contraire, le compilateur va émettre un avertissement). Nous verrons ultérieurement pourquoi cette distinction entre déclaration et définition existe.

Dans le cas d'une définition, il est possible d'affecter une valeur initiale, par exemple :

```
// Définition d'une variable i de type int ayant pour valeur initiale 42
int i = 42;
```

Il est également possible de déclarer ou définir plusieurs identifiants en les séparant par des virgules, par exemple :

```
// Définition d'une variable a de type int ayant pour valeur initiale 42
//  et d'une variable b de type int ayant pour valeur initiale 84
int a = 42, b = 84;
```

Si la valeur initiale d'une variable n'est pas indiquée alors elle dépend de sa durée de stockage :
- si la durée est statique (durée de vie du programme), elle est initialisée à zéro,
- si la durée est automatique (jusqu'à la fin du bloc), elle est indéterminée (**attention**, il s'agit d'une source fréquente de problèmes).

4.4. Déclarations et définitions de fonctions

4.4.1. Déclaration

La syntaxe pour une *déclaration* d'une fonction est la suivante :

```
<Type de retour> <Identifiant>(<Type argument> <Identifiant argument>...);
```

Exemple :

```
// Déclaration d'une fonction qui prend deux arguments de type int en entrée (a
et b)
//  et qui renvoie une valeur de type int
int add(int a, int b);
```

Pour une déclaration, les identifiants (noms) des arguments ne sont pas requis. Il est donc possible d'écrire :

```
// Déclaration d'une fonction qui prend deux arguments de type int en entrée
//  et qui renvoie une valeur de type int
int add(int, int);
```

Une fonction qui ne prend pas d'arguments est censée être déclarée avec le mot clé `void` entre parenthèses (mais ce n'est pas indispensable et souvent omis) :

```
// Déclaration d'une fonction qui ne prend pas d'arguments
//  et qui renvoie une valeur de type int
int f(void);
```

La valeur du type de retour peut être `void` pour indiquer qu'une fonction ne renvoie pas de valeur :

```
// Déclaration d'une fonction qui prend un argument de type int et un de type
float
//  et qui ne renvoie pas de valeur
void add(int, float);
```

4.4.2. Définition

Une *définition* de fonction est une déclaration pour laquelle on spécifie en plus le corps de la fonction à l'aide d'un bloc de code encadré par des accolades :

```
// Définition d'une fonction qui prend deux arguments de type int en entrée : a
et b
//  et qui renvoie une valeur de type int
int add(int a, int b) {
   // Corps de la fonction
   return a + b;
}
```

Dans une définition, l'identifiant (nom) des arguments, s'il y en a, est obligatoire.

La définition d'une fonction ne peut apparaître qu'à l'extérieur d'une fonction. Les fonctions imbriquées ne sont pas autorisées (en tout cas dans le standard).

4.4.3. Généralités

La portée d'une déclaration globale ou d'une définition d'une fonction est celle du fichier : depuis la déclaration ou définition et jusqu'à la fin du fichier. Le compilateur sait alors que l'identifiant se réfère à une fonction, connaît son type de retour ainsi que le nombre et le type de ses arguments. S'il rencontre cet identifiant, il peut vérifier qu'il est bien utilisé comme

une fonction, peut vérifier qu'il y a bien le bon nombre d'arguments, faire les conversions si nécessaire des arguments, faire la conversion du type de retour si nécessaire…

Il est donc nécessaire de déclarer (ou de définir) une fonction avant sa première utilisation dans un fichier.

Par exemple :

```
void g() {
   f(); // f est appelée ici mais est déclarée ultérieurement => erreur
}

void f() { // Définition de la fonction f
   // ...
}
```

Dans le code ci-dessus, la fonction `f` est appelée à l'intérieur de la fonction `g` mais n'est déclarée (plus précisément définie dans l'exemple) que plus loin. Le compilateur va donc émettre un avertissement ou une erreur (en fonction de sa configuration) indiquant qu'il ne connaît pas le symbole `f` au moment où il le rencontre.

Une première solution pour résoudre ce problème est de définir la fonction avant son utilisation dans la fonction `g` :

```
void f() { // Définition de la fonction f
   // ...
}

void g() {
   f(); // f est appelée ici et a été définie précédemment => OK
}
```

Mais, il est également possible de ne mettre qu'une déclaration de la fonction `f` avant la fonction `g` :

```
void f(void); // Déclaration de la fonction f

void g() {
   f(); // f est appelée ici et a été déclarée précédemment => OK
}

void f() { // Définition de la fonction f
   // ...
}
```

Pour une fonction, la liaison est par défaut externe, sauf si la définition de la fonction est débute par le mot clé `static`.

5. Les expressions

Une expression est une séquence d'opérateurs et leurs opérandes, qui spécifie un calcul.

Les expressions peuvent produire un résultat (par exemple `1 + 2` produit la valeur `3`) et peuvent également produire des effets de bord (par exemple `i = 3` modifie la valeur de la variable `i`, `printf("Bonjour")` affiche un message à l'écran).

Les opérandes d'un opérateur peuvent être :
- des littéraux (par exemple `1 + 2`) ;
- des noms de variable (par exemple `a + 2`), auquel cas le nom de la variable est remplacé à l'exécution par sa valeur ;
- des appels de fonction (par exemple `carre(a) + 2`), auquel cas, la fonction est exécutée et remplacée par sa valeur de retour ;
- le résultat d'autres expressions.

5.1. Les opérateurs

5.1.1. Introduction

La langage C possède un certain nombre d'opérateurs que nous allons décrire dans la suite de cette section. Chaque opérateur est associé à deux propriétés :
- la *priorité*, qui permet de connaître dans quel ordre les opérations sont effectuées lorsque plusieurs opérateurs ont des priorités différentes ;
- l'*associativité*, qui permet de connaître dans quel ordre les opérations sont effectuées lorsque plusieurs opérateurs ont des priorités identiques.

Voici les principaux opérateurs du C :

Priorité	Opérateur	Description	Associativité
1 (la plus forte)	`++` et `--`	Post incrémentation et décrémentation	Gauche à droite
	`()`	Appel de fonction	
	`[]`	Accès à un tableau	
	`.`	Accès à un membre d'une structure	
	`->`	Accès à un membre d'une structure via un pointeur	
2	`++` et `--`	Pré incrémentation et décrémentation	Droite à gauche
	`+` et `-`	Plus et moins unaires	
	`!` et `~`	NON logique et bit-à-bit	
	`(type)`	Cast explicite	

Priorité	Opérateur	Description	Associativité
	`*`	Déréférencement	
	`&`	Adresse de	
	`sizeof`	Taille d'un type ou d'un objet	
	`_Alignof`	Alignement d'un type ou d'un objet	
3	`*`	Multiplication	Gauche à droite
	`/`	Division	
	`%`	Reste	
4	`+`	Addition	Gauche à droite
	`-`	Soustraction	
5	`<<`	Décalage à gauche	Gauche à droite
	`>>`	Décalage à droite	
6	`<` et `<=`	Inférieur et inférieur ou égal	Gauche à droite
	`>` et `>=`	Supérieur et supérieur ou égal	
7	`==` et `!=`	Égal et différent	Gauche à droite
8	`&`	ET bit-à-bit	Gauche à droite
9	`^`	OU exclusif bit-à-bit	Gauche à droite
10	`|`	OU bit-à-bit	Gauche à droite
11	`&&`	ET logique	Gauche à droite
12	`||`	OU logique	Gauche à droite
13	`? :`	Opérateur conditionnel ternaire	Droite à gauche
14	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `^=`, `|=`	Affectation simple et affectations combinées	Droite à gauche
15 (la plus faible)	`,`	Virgule	Gauche à droite

Quelques exemples concernant la priorité et l'associativité :

- `5 * -a` sera évaluée comme `5 * (-a)` car l'opérateur `-` (moins unaire, c'est-à-dire l'opposé) est plus prioritaire que l'opérateur `*` de multiplication ;
- `5 + b - 3` sera évaluée comme `(5 + b) - 3` car les opérateurs `+` et `-` (addition et soustraction) ont la même priorité mais leur associativité est de gauche à droite ;

- `a = b = 42` sera évaluée comme `a = (b = 42)` car l'associativité de l'opérateur `=` (affectation) est de droite à gauche.

Pour certains opérateurs, en particulier pour les opérateurs arithmétiques, la priorité et l'associativité sont bien celles attendues. Dans d'autres cas, elles sont moins évidentes. Dans le doute, il vaut toujours mieux mettre des parenthèses pour faciliter la lecture et éviter des erreurs.

5.1.2. Opérateurs arithmétiques

Les opérateurs arithmétiques sont les suivants :

Opérateur	Description
`+`	Plus unaire
`-`	Moins unaire (opposé)
`*`	Multiplication
`/`	Division
`%`	Reste
`+`	Addition
`-`	Soustraction

5.1.2.1. Opérandes

Ces opérateurs s'appliquent sur des entiers et sur des flottants (à l'exception de `%`). *Note* : l'addition et la soustraction peuvent également s'appliquer sur des pointeurs, ce qui sera vu ultérieurement (arithmétique des pointeurs).

5.1.2.2. Débordement

Sur des entiers non signés, les opérations sont réalisées modulo 2^n (où n est le nombre de bits du type sur lequel les opérations sont réalisées). Par exemple, sur des `unsigned char` sur 8 bits, `255 + 1` est égal à `0` et `0 - 1` est égal à `255`.

Sur des entiers signés, le comportement lors d'un débordement (c'est-à-dire su le résultat n'est pas représentable sur la plage de valeur du type résultat) est *non défini*. Si on veut écrire un programme portable, il est nécessaire de faire attention de ne pas se retrouver dans ce cas.

5.1.2.3. Division (`/`) et Reste (`%`)

Dans les deux cas, si l'opérande de droite vaut `0`, le comportement est *non défini*.

Sur des entiers, le résultat de la division est tronqué vers `0` :
- `11 / 2` est égal à `5`
- `-11 / 2` est égal à `-5`

Le reste est tel que : `(a/b)*b + a%b == a`. Par conséquence, le reste a toujours le même signe que le membre de gauche (dividende).

- `11 % 2` est égal à `1`

- `-11 % 2` est égal à `-1`
- `11 % -2` est égal à `1`
- `-11 % -2` est égal à `-1`

5.1.3. Opérateurs bit-à-bit

Les opérateurs bit-à-bit sont les suivants :

Opérateur	Description
`~`	NON bit-à-bit (unaire)
`&`	ET bit-à-bit
`|`	OU bit-à-bit
`^`	OU exclusif bit-à-bit

Chaque bit du résultat de l'opérateur unaire NON (`~`) est la négation du bit correspondant de son opérande.

Exemple : Si `a` est de type `unsigned short` (supposé sur 16 bits) et vaut `0x0123` (en binaire : `0000 0001 0010 0011`), `~a` vaut `0xFDEC` (en binaire : `1111 1110 1101 1100`).

Pour les trois autres opérateurs, chaque bit du résultat est le résultat de l'opération sur les bits correspondants des deux opérandes.

Exemple : Si `a` est de type `unsigned short` (supposé sur 16 bits) et vaut `0x0123` (en binaire : `0000 0001 0010 0011`) et si `b` est de type `unsigned short` et vaut `0x00FE` (en binaire : `0000 0000 1111 1110`) :

- `a & b` vaut `0x0022` (en binaire : `0000 0000 0010 0010`)
- `a | b` vaut `0x01FF` (en binaire : `0000 0001 1111 1111`)
- `a ^ b` vaut `0x01DD` (en binaire : `0000 0001 1101 1101`)

5.1.4. Opérateurs de décalage

Les opérateurs de décalage sont les suivants :

Opérateur	Description
`<<`	décalage à gauche
`>>`	décalage à droite

Ces deux opérateurs s'appliquent sur des entiers.

L'opérateur `a << k` renvoie la valeur `a` décalée à gauche de `k` bits. Les `k` bits de poids faibles du résultat sont mis à `0`. Les `k` bits de poids forts de `a` sont perdus. Si `a` est non signé, le résultat correspond à $a \times 2^k$ modulo 2^n où n est le nombre de bits du résultat. Si `a` est signé et positif, le résultat correspond à $a \times 2^k$ si cette valeur est représentable sur le type résultat. Dans le cas contraire, le comportement n'est pas défini.

Exemple : si `a` est de type `unsigned short` (supposé sur 16 bits) et vaut `0x0123` (en binaire : `0000 0001 0010 0011`, en décimal : 291), `a << 2` vaut `0x048C` (en binaire : `0000 0100 1000 1100`, en décimal : $1164 = 291 \times 4$).

L'opérateur `a >> k` renvoie la valeur `a` décalée à droite de `k` bits. Les `k` bits de poids faibles de `a` sont perdus. Si `a` est non signé ou signé et de valeur positive, les `k` bits de poids forts du résultat sont mis à `0`. Dans le cas contraire (`a` de signe négatif), le comportement dépend de l'implémentation. Dans la vaste majorité des implémentations, les `k` bits de poids forts du résultat sont mis à `1` pour conserver la signe négatif (voir la règle d'extension de signe en complément à 2). Si `a` est non signé ou signé positif, le résultat correspond à la partie entière de $\frac{a}{2^k}$.

Exemple : si `a` est de type `unsigned short` (supposé sur 16 bits) et vaut `0x0123` (en binaire : `0000 0001 0010 0011`, en décimal : 291), `a >> 2` vaut `0x0048` (en binaire : `0000 0000 0100 1000`, en décimal : $72 = \frac{291}{4}$).

5.1.5. Opérateurs logiques

Les opérateurs logiques sont les suivants :

Opérateur	Description
`!`	NON logique
`&&`	ET logique
`||`	OU logique

Ces trois opérateurs renvoient un entier de type `int` qui vaut `1` ou `0`.

`!a` renvoie la valeur `1` si l'expression `a` est égale à `0` (si `a==0`), ou `0` dans le cas contraire.

`a && b` renvoie la valeur `1` si les expressions `a` et `b` sont toutes les deux différentes de `0`, ou `0` dans le cas contraire. Si `a` est égale à `0`, l'expression `b` n'est pas du tout évaluée (donc si par exemple l'opérande de droite est un appel de fonction, la fonction n'est pas appelée).

`a || b` renvoie la valeur `1` si au moins une des expressions `a` ou `b` est différentes de `0`, ou `0` dans le cas contraire. Si `a` est différente de `0`, l'expression `b` n'est pas du tout évaluée.

Exemples :
- `!42` vaut `0`
- `42 && 12` vaut `1`
- `42 && 0` vaut `0`
- `0 && g(17)` vaut `0` et la fonction `g` n'est pas appelée

5.1.6. Opérateurs de comparaison

Les opérateurs de comparaison sont les suivants :

Opérateur	Description
`==`	Égalité

Opérateur	Description
`!=`	Différence
`<`	Inférieur
`<=`	Inférieur ou égal
`>`	Supérieur
`>=`	Supérieur ou égal

Ces opérateurs renvoient un entier de type `int` qui vaut `1` si la comparaison est correcte, ou `0` dans le cas contraire.

5.1.7. Opérateurs d'affectation

5.1.7.1. Affectation simple

L'opérateur d'affectation est `=`.

Le membre de gauche doit être une valeur modifiable (par exemple une variable). L'expression de droite est évaluée et sa valeur est affectée au membre de gauche.

Les types du membre de droite et du membre de gauche doivent être identiques ou il doit être possible de convertir implicitement le type de droite vers le type de gauche.

L'ensemble de l'expression est évaluée à la valeur du membre de droite. Il est donc possible d'écrire `a = b = 42`, qui, d'après les règles d'associativité se traduit par `a = (b = 42)`. `b = 42` affecte la valeur `42` à la variable `b` et renvoie la valeur `42`, ce qui donne ensuite `a = 42`.

⚠ **Attention** : ne pas confondre l'opérateur d'affectation `=` et l'opérateur de comparaison `==`.

5.1.7.2. Affectations combinées

Il s'agit de raccourcis d'écriture :

Opérateur	Équivalent à
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`
`a %= b`	`a = a % b`
`a &= b`	`a = a & b`
`a |= b`	`a = a | b`
`a ^= b`	`a = a ^ b`
`a <<= b`	`a = a << b`

Opérateur	Équivalent à
`a >>= b`	`a = a >> b`

Par exemple, l'expression `a += 1` est donc identique à `a = a + 1`.

5.1.8. Opérateurs d'incrémentation et décrémentation

Il existe 4 opérateurs :
- Pré-incrémentation ou décrémentation :
  - `++a` et `--a`
- Post-incrémentation ou décrémentation :
  - `a++` et `a--`

Dans les deux cas, l'expression `a` est incrémentée de 1 (équivalent à `a = a + 1`) ou décrémentée de 1 (équivalent à `a = a - 1`).

La différence réside dans la valeur renvoyée par l'expression complète :
- Pour l'opérateur de pré-incrémentation (respectivement décrémentation), la valeur renvoyée par l'expression est la valeur de l'expression `a` **après** incrémentation (respectivement décrémentation)
- Pour l'opérateur de post-incrémentation (respectivement décrémentation), la valeur renvoyée par l'expression est la valeur de l'expression `a` **avant** incrémentation (respectivement décrémentation)

Par exemple :

```
int a = 42;
int b = a++;
// Ici, a vaut maintenant 43 (elle a été incrémentée)
//  et b vaut 42 (la valeur de a avant son incrémentation)
```

```
int a = 42;
int b = ++a;
// Ici, a vaut maintenant 43 (elle a été incrémentée)
//  et b vaut 43 (la valeur de a après son incrémentation)
```

5.1.9. Appel de fonction

L'opérateur d'appel de fonction a la syntaxe suivante :

fonction `(` *arguments* `)`

fonction est le nom (identifiant) de la fonction et *arguments* est une série d'expressions, séparées par des virgules.

Le nombre d'arguments doit être identique au nombre d'arguments attendus par la fonction (voir sa déclaration). Chaque expression passée en argument doit pouvoir être convertie implicitement vers le type attendu pour l'argument correspondant par la fonction (voir également sa déclaration). Les expressions sont évaluées dans un ordre non spécifié. La valeur de chaque expression est ensuite copiée dans l'argument de la fonction (nous reviendrons sur ce point important dans la suite). La fonction est ensuite exécutée et sa valeur de retour devient la valeur de l'expression.

Exemple :

```
int add(int a, int b) {
   return a + b;
}

void f() {
   int c = 42;
   int d = 1;
   int e = add(c, d + 1);
   // e vaut 44
}
```

Dans l'exemple ci-dessus, deux expressions sont passées en argument pour la fonction `add` :

- `c` qui est évaluée à la valeur `42`
- `d + 1` qui est évaluée à la valeur `2`

Ces deux expressions sont donc évaluées et leur valeur est copiée dans les arguments `a` et `b` de la fonction.

La fonction est exécutée et renvoie la valeur `44`.

Cette valeur vient remplacer l'appel de la fonction, ce qui donne `int e = 44`.

⚠️ **Attention : ne confondez pas la syntaxe de déclaration (ou définition) d'une fonction et celle pour appeler une fonction.** C'est une erreur très fréquente.

Pour appeler une fonction, il suffit de mettre son nom et entre parenthèse la valeur de ses arguments. Le type de sa valeur de retour et le type de ses arguments ne doivent pas être rappelés :

```
add(42, 2);             // Oui
add(c, d + 1);          // Oui
int add(42, 2);         // NON !
add(int 42, int 2);     // NON !
int add(int 42, int 2); // NON !
```

5.1.10. Opérateurs d'accès

Opérateur	Description
`a[b]`	Accès au b-ième élément du tableau `a`
`*a`	Déréférencement du pointeur `a`
`&a`	Adresse de l'objet `a`
`a.b`	Accès au champ `b` de la structure `a`
`a->b`	Accès au champ `b` de la structure pointée par `a`

Seul le premier opérateur sera étudiée ici, les quatre autres seront abordés ultérieurement.

Si `a` est le nom d'un tableau et `b` une expression entière dont la valeur est strictement inférieure au nombre d'éléments du tableau, `a[b]` permet d'accéder au b-ième élément du tableau.

Note : les éléments des tableaux sont numérotés à partir de `0`.

Exemple :

```
int tab[3] = {1, 2, 3};
int a = tab[0];  // 1
int b = tab[2];  // 3
```

5.1.11. Cast

La syntaxe de l'opérateur de conversion explicite (*cast*) est la suivante :

`(` *type* `)` *expression*

L'expression *expression* est évaluée et son type est explicitement converti vers le type *type*.

Exemple :

```
float f = 3.14;
int p = (int) f; // 3
```

5.1.12. Opérateur conditionnel ternaire

L'opérateur conditionnel ternaire a la syntaxe suivante :

condition `?` *expression-vrai* `:` *expression-faux*

L'expression *condition* est tout d'abord évaluée.

- Si elle est évaluée à une valeur différente de zéro, l'expression *expression-vrai* est évaluée et l'opérateur renvoie sa valeur.
- Si elle est évaluée à une valeur égale à zéro, l'expression *expression-faux* est évaluée et l'opérateur renvoie sa valeur.

Dans les deux cas, l'autre expression n'est pas évaluée (ce qui signifie par exemple que s'il s'agit d'appels de fonctions, celles-ci ne sont pas appelées).

Exemple :

```
int double(int a) {
   printf("double\n");
   return a * 2;
}

int triple(int a) {
   printf("triple\n");
   return a * 3;
}

void f(int b) {
   int c = (b == 2) double(5) : triple(5);
}
```

Si `b` vaut 2, la variable `c` vaudra `10` et le message `double` sera affiché. Si `b` est différent de 2, la variable `c` vaudra `15` et le message `triple` sera affiché.

5.2. Ordre d'évaluation

Les notions de *priorité* et d'*associativité* permettent de savoir dans quel ordre sont réalisées les opérations. **Néanmoins**, l'ordre d'évaluation des opérandes de ces opérateurs est *non spécifié*.

Par exemple :

```
int double(int a) {
   printf("double (%d)\n", a);
   return a * 2;
}

void f(int b) {
   int c = double(2) + double(3) * double(4);
}
```

De part la priorité des opérateurs, l'expression `double(2) + double(3) * double(4)` est interprétée comme `double(2) + (double(3) * double(4))`. Néanmoins, l'évaluation des trois opérandes (`double(2)`, `double(3)` et `double(4)`) peut avoir lieu dans n'importe quel ordre, et donc l'appel de ces fonctions, et leur effet de bord (affichage grâce à `printf`) peut se faire dans n'importe quel ordre. Donc bien que dans tous les cas, le résultat vaudra bien 52 ($(2 \times 2) + ((2 \times 3) \times (2 \times 4))$), tous les affichages suivants sont possibles :

```
double (2)
double (3)
double (4)
```

ou

```
double (2)
double (4)
double (3)
```

ou

```
double (3)
double (4)
double (2)
```

et ainsi de suite (les 6 cas sont possibles).

5.2.1. Cas particulier : les opérateurs logique (court-circuit) et l'opérateur conditionnel ternaire

Les opérateurs logiques `&&`, `||` sont dits à court-circuit. En effet, on a la garantie que leur second opérande n'est pas évalué si :

- pour `&&` le premier opérande est égal à 0,
- pour `||` le premier opérande est différent de 0.

De même, on a la garantie que l'opérateur conditionnel ternaire n'évalue qu'un seul de ses deux derniers opérandes.

5.2.2. Comportements non définis

Le comportement de certaines expressions est explicitement marqué comme non défini (donc ne devrait pas être utilisé), en particulier quand il y a plusieurs effets de bord sur une même variable, par exemple :

```
a = a++ + a++;
f(++a, ++a);
```

5.3. Conversions implicites

Si nécessaire, des conversions de type sur les opérateurs sont réalisées lors d'évaluation d'expressions.

Les règles exactes sont présentes dans le standard mais dépassent le cadre de ce cours.

6. Le corps des fonctions : les instructions

6.1. Introduction

Le corps d'une fonction, fourni lors de sa définition entre accolades (`{` et `}`), est constitué d'une liste d'*instructions* (*statements*) et de déclaration (ou définition) de variables. *Attention*, ici le terme instruction a un sens légèrement différent de celui utilisé dans la première partie du cours (il ne s'agit pas des instructions élémentaires exécutables par le processeur).

Il existe plusieurs catégories d'instructions qui seront détaillées dans la suite de cette section.

6.2. Les instructions expressions (*expression-statement*)

Une *instruction expression* est simplement une expression terminée par un point-virgule (`;`).

Par exemple :

```
void f() {
   int i;
   i = 4; // Instruction expression
}
```

Dans le code ci-dessus, `i = 4` est une expression, qui affecte la valeur `4` à la variable `i` et renvoie la valeur affectée : `4`. Pour pouvoir mettre cette expression dans le corps d'une fonction, il faut la transformer en instruction, en ajoutant simplement un `;` après cette expression. La valeur renvoyée par l'expression est perdue.

Si l'expression n'a pas d'effet de bord, la transformer en instruction ne produit aucun effet, par exemple :

```
void f() {
   int i = 2;
   i + 4; // Inutile, ne produit aucun effet
   // i vaut toujours 2 ici
}
```

`i + 4` est une expression qui renvoie la valeur de la variable `i` incrémentée de 4 (mais sans changer la variable `i`), donc la transformer en instruction est parfaitement autorisé mais ne produit aucun effet.

Si on avait voulu modifier la valeur de `i` en l'incrémentant de 4, il aurait fallu utiliser une instruction qui a un effet :

```
void f() {
   int i = 2;
   i = i + 4; // i vaut 6 après cette instruction
}
```

6.3. Les instructions composées (*compound-statement*)

Certaines constructions présentées ultérieurement n'autorisent qu'une seule instruction (par exemple dans la branche principale d'un `if`).

Si l'on souhaite mettre plusieurs instructions, il est nécessaire de les regrouper dans une instruction composée en les mettant entre accolades : `{` et `}` .

Par exemple :

```
void f() {
   int i = 2;
   {
      // Instruction composée regroupant les deux instructions suivantes
      i = i + 1;
      i = i * i;
   }
}
```

6.4. Les instructions de sélection (*selection-statement*)

Il existe trois instructions de sélection : `if`, `if / else` et `switch`.

Elles permettent d'exécuter ou non certaines instructions en fonction de la valeur d'une expression.

6.4.1. `if` et `if / else`

Les syntaxes sont les suivantes :

`if (` *expression* `)` *instruction1*

`if (` *expression* `)` *instruction1* `else` *instruction2*

À l'exécution, l'expression *expression* est évaluée. L'instruction *instruction1* est exécutée si et seulement si la valeur de cette expression est différente de zéro. Si un `else` est présent (deuxième syntaxe ci-dessus), l'instruction *instruction2* est exécutée si et seulement si la valeur de l'expression est égale à zéro.

Note : *instruction1* et *instruction2* correspondent chacune à une seule instruction. Si vous souhaitez que plusieurs instructions soient exécutées conditionnellement, il faut les grouper dans une instruction composée (les mettre à l'intérieur d'un bloc `{ }`).

Exemple :

```
void f(int i) {
   if (i == 2)
      printf("i est égal à 2\n"); // Exécuté si i est égal à 2
   printf("Tout le temps\n");     // Exécuté quelque soit la valeur de i
}
```

Ici `i == 2` est l'expression utilisée comme condition. Pour rappel, l'opérateur de comparaison `==` renvoie 1 (donc une valeur différente de zéro) si la comparaison est vraie. Donc l'instruction `printf("i est égal à 2\n");` (qui est une instruction expression) est exécutée si et seulement si la variable `i` vaut `2`. L'instruction suivante (`printf("Tout le temps\n");`) est en dehors du `if` (non pas à cause de l'indentation comme cela serait le cas en Python, mais car le `if` ne prend qu'une seule instruction) et donc est exécutée dans tous les cas.

Autre exemple :

```
void f(int i) {
   if (i % 2 == 0)
      printf("i est pair\n");   // Exécuté si i % 2 == 0
```

```
    else
        printf("i est impair\n"); // Exécuté si i % 2 != 0
    printf("Tout le temps\n");   // Exécuté quelque soit la valeur de i
}
```

⚠️ **Attention** : ne pas confondre l'opérateur d'affectation `=` et l'opérateur de comparaison `==` dans la condition du `if`.

```
void f(int i) {
    if (i = 42) // Notez le = d'affectation au lieu du == de comparaison
        printf("i vaut 42\n");
}
```

Si vous appelez cette fonction avec comme argument la valeur `1` par exemple, elle affichera `"i vaut 42"`. En effet, `i = 42` est une expression d'affectation. Elle affecte la valeur `42` à la variable `i`. L'expression elle-même renvoie la valeur affectée, c'est-à-dire la valeur `42`. Or `42` est différent de `0`, donc la condition du `if` est considérée comme vraie et donc la branche principale s'exécute. C'est un piège extrêmement classique.

6.4.2. `switch`

La syntaxe usuelle est la suivante :

`switch (` *expression* `) {`
substatement
substatement
... `}`

Le corps du `switch` est une série d'une ou plusieurs instructions qui peuvent être des instructions classiques ou l'une des deux constructions suivantes :

`case` *case-expression* `:` *statement*

ou

`default:` *statement*

Exemple :

```
void f(int i) {
    switch (i) {
    case 0:
        printf("Cas 0\n");
        printf("Cas 0, suite\n");
    case 1:
        printf("Cas 1\n");
        printf("Cas 1, suite\n");
    case 2:
        printf("Cas 2\n");
        printf("Cas 2, suite\n");
    default:
        printf("Cas default\n");
        printf("Cas default, suite\n");
    }
```

```
    printf("Après le switch\n");
}
```

Le fonctionnement du `switch` en C est un peu particulier (comparé à d'autres langages).

L'expression *expression* (dans l'exemple ci-dessus `i`) est évaluée à l'entrée du `switch`. Cette expression doit être de type entier (`int` dans l'exemple ci-dessus). La valeur de cette expression est comparée successivement avec la valeur des expressions *case-expression* des différents `case` présents dans le corps du `switch`. Chacune de ces expressions doit être une expression entière constante et doivent être différentes les unes des autres (c'est-à-dire qu'il ne peut y avoir deux `case 0:` dans un même `switch`). Dans l'exemple ci-dessus, ces *case-expressions* sont : `0`, `1` et `2`.

Dès que la valeur d'une des expressions *case-expression* est égale à la valeur de *expression*, **toutes** instructions suivantes dans le corps du `switch` sont exécutées.

Par exemple, dans le code ci-dessus, si `i` vaut 1, les deux premiers `printf` sont ignorés, par contre, les six derniers (à partir du `case 1:`) sont exécutés. À l'exécution, le programme affichera :

```
Cas 1
Cas 1, suite
Cas 2
Cas 2, suite
Cas default
Cas default, suite
Après le switch
```

Si aucune des *case-expression* ne correspond à la valeur de *expression* :

- si un `default:` est présent (il ne peut y en avoir au plus qu'un seul), les instructions suivant le `default:` sont exécutées
- s'il n'y a pas de `default:`, aucune instruction du corps du `default:` n'est exécutée.

Dans l'exemple précédent, si `i` vaut 3, seuls les deux derniers `printf` sont exécutés. À l'exécution, le programme affichera :

```
Cas default
Cas default, suite
Après le switch
```

L'exécution de toutes les instructions à partir du `case` ou du `default` pertinent n'est en général pas le comportement désiré. Il est possible de sortir prématurément du `switch` grâce à l'instruction `break`.

On peut donc transformer l'exemple ci dessus :

```
void f(int i) {
    switch (i) {
    case 0:
        printf("Cas 0\n");
        printf("Cas 0, suite\n");
        break;
    case 1:
        printf("Cas 1\n");
```

```
        printf("Cas 1, suite\n");
        break;
    case 2:
        printf("Cas 2\n");
        printf("Cas 2, suite\n");
        break;
    default:
        printf("Cas default\n");
        printf("Cas default, suite\n");
    }
    printf("Après le switch\n");
}
```

Si `i` vaut 1, l'exécution de ce nouvel exemple donnera :

```
Cas 1
Cas 1, suite
Après le switch
```

6.5. Les instructions d'itération (*iteration-statement*)

Le C possède trois constructions pour exécuter zéro, une ou plusieurs fois une instruction : `while`, `do ... while` et `for`.

6.5.1. `while`

La syntaxe est la suivante :

`while (` *expression* `)` *instruction*

L'expression *expression* est évaluée une première fois.

- Si elle est égale à 0, l'instruction *instruction* n'est pas exécutée et l'exécution continue à l'instruction suivante.
- Si elle est différente de 0, l'instruction *instruction* est exécutée puis l'exécution du `while` recommence (l'expression *expression* est de nouveau évaluée et en fonction du résultat, on sort du `while` ou on exécute l'instruction et ainsi de suite).

Par exemple :

```
void f(int i) {
    while (i > 0) {
        printf("i = %d\n", i);
        i = i - 1;
    }
    printf("Après while\n");
}
```

Si la fonction `f` est appelée avec la valeur `0` (donc `i` vaut `0` au début de l'exécution de la fonction), l'exécution de la fonction affichera :

```
Après while
```

En effet, la condition `i > 0` étant fausse (donc renvoyant 0) dès la première évaluation, le corps de la boucle n'est jamais exécuté.

Si la fonction `f` est appelée avec la valeur `1` (donc `i` vaut `1` au début de l'exécution de la fonction), l'exécution de la fonction affichera :

```
i = 1
Après while
```

En effet, la condition `i > 0` étant vraie (donc renvoyant 1) à l'entrée de la boucle, le corps de la boucle est exécuté une première fois. L'exécution du corps décrémente `i` qui se retrouve donc à 0. La condition `i > 0` est de nouveau testée. Elle est maintenant fausse donc l'exécution se poursuit après la boucle.

Si la fonction `f` est appelée avec la valeur `2` (donc `i` vaut `2` au début de l'exécution de la fonction), l'exécution de la fonction affichera :

```
i = 2
i = 1
Après while
```

En effet, la condition `i > 0` étant vraie (donc renvoyant 1) à l'entrée de la boucle, le corpsd e la boucle est exécuté une première fois. L'exécution du corps décrémente `i` qui se retrouve donc à 1. La condition `i > 0` est de nouveau testée. Elle est toujours vraie donc le corps de la boucle est exécuté une deuxième fois. À l'issue de cette exécution, i vaut 0. La condition `i > 0` est de nouveau testée. Elle est maintenant fausse donc l'exécution se poursuit après la boucle.

On remarque que pour une boucle `while` le corps de la boucle peut ne jamais être exécutée si la condition est évaluée à 0 dès sa première évaluation.

6.5.2. `do ... while`

La syntaxe est la suivante :

`do` *instruction* `while (` *expression* `);`

Cette boucle fonctionne comme le `while` sauf que le corps de la boucle (*instruction*) est toujours exécuté au moins une fois car l'expression *expression* n'est testée qu'à la fin de chaque itération.

Par exemple :

```
void f(int i) {
   do {
      printf("i = %d\n", i);
      i = i - 1;
   } while (i > 0);
   printf("Après while\n");
}
```

Si la fonction `f` est appelée avec la valeur `0`, l'exécution de la fonction affichera :

```
i = 0
Après while
```

En effet, le corps de la boucle est exécuté une première fois, puis à la fin de l'exécution, la condition est évaluée. Ici, la condition `i > 0` étant fausse (donc renvoyant 0), l'exécution se poursuit après la boucle.

Si la fonction `f` est appelée avec la valeur `1`, l'exécution de la fonction affichera :

```
i = 1
Après while
```

En effet, le corps de la boucle est exécuté une première fois. L'exécution du corps décrémente `i` qui se retrouve donc à 0. La condition `i > 0` est testée. Elle est fausse donc l'exécution se poursuit après la boucle.

Si la fonction `f` est appelée avec la valeur `2`, l'exécution de la fonction affichera :

```
i = 2
i = 1
Après while
```

6.5.3. `for`

La syntaxe est la suivante :

`for (` *expression1* `;` *expression2* `;` *expression3* `)` *instruction*

L'exécution de cette boucle `for` est équivalente au code suivant (sauf en ce qui concerne le traitement de `continue` comme cela sera vu plus loin) :

expression1 `;`
`while (` *expression2* `) {`
    instruction
    expression3 `;`
`}`

L'expression *expression1* est tout d'abord évaluée. Cette expression peut contenir la définition d'une variable dont la visibilité sera limitée à la boucle.

Puis l'expression *expression2* est évaluée. Si elle est égale à 0, l'exécution continue après la boucle. Si elle est différente de 0, le corps de la boucle (*instruction*) est exécuté et à la fin du corps de la boucle, l'expression *expression3* est évaluée.

Enfin, la condition *expression2* est de nouveau évaluée et en fonction du résultat, l'exécution se poursuit après la boucle (si *expression2* est égale à 0) ou le corps de la boucle (et *expression3*) est exécuté de nouveau (si *expression2* est différente de 0) et ainsi de suite.

Exemple :

```
void f() {
   for (int i = 0; i < 5; i = i + 1)
      printf("i = %d\n", i);
   printf("Après for\n");
}
```

L'exécution de cette fonction affichera :

```
i = 0
i = 1
i = 2
i = 3
i = 4
Après for
```

Les expression *expression1* et *expression3* peuvent être omises, par contre les `;` restent nécessaires, par exemple :

```
// Extrait de code
void f() {
   int i = 0;
   for (; i < 5;) {
      printf("i = %d\n", i);
      i = i + 1;
   }
   printf("Après for\n");
}
```

L'exécution de ce code produit le même résultat que l'exemple précédent.

6.5.4. `continue` et `break`

Il est possible d'altérer l'exécution d'une boucle avec les mots clés `continue` et `break`.

`continue`, qui ne peut apparaître qu'à l'intérieur d'une boucle, permet de sauter le reste de l'exécution de l'itération courante de la boucle pour arriver directement à l'évaluation de la condition de sortie. Comme dans le fonctionnement normal, en fonction de l'évaluation de cette condition, on recommencera une exécution du corps de la boucle ou non.

```
void f() {
   for (int i = 0; i < 5; i = i + 1) {
      if (i == 2)
         continue;
      printf("i = %d\n", i);
   }
   printf("Après for\n");
}
```

L'exécution de cette fonction affichera :

```
i = 0
i = 1
i = 3
i = 4
Après for
```

En effet, dans l'itération où `i` vaut `2`, le `continue` permet de sauter le reste du corps de la boucle (donc ici le `printf`) pour continuer à l'itération suivante. On notera que la troisième expression du `for` (`i = i + 1`) est tout de même exécutée avant le test de la condition de continuation (`i < 5`).

`break` permet de sortir immédiatement de la boucle, aucune autre itération ne sera exécutée.

Par exemple :

```
void f() {
   for (int i = 0; i < 5; i = i + 1) {
      if (i == 2)
         break;
      printf("i = %d\n", i);
   }
   printf("Après for\n");
}
```

L'exécution de cette fonction affichera :

```
i = 0
i = 1
Après for
```

En effet, dans l'itération où `i` vaut `2`, le `break` va sortir de la boucle immédiatement, en sautant le reste du corps de la boucle (donc ici le `printf`).

Si plusieurs boucles sont imbriquées, `continue` et `break` s'appliquent sur la boucle la plus imbriquée.

Par exemple :

```
void f() {
   int j = 0;
   while (j < 5) {
      int i = 0;
      while (i < 5) {
         if (i + j >= 5) {
            printf("i = %d, j = %d\n", i, j);
            break; // Sortira de la boucle while (i < 5)
         }
         i = i + 1;
      }
      j = j + 1;
   }
}
```

L'exécution de cette fonction affichera :

```
i = 4, j = 1
i = 3, j = 2
i = 2, j = 3
i = 1, j = 4
```

6.6. L'instruction `return`

Il existe deux formats pour l'instruction `return` :

`return;`

`return` *expression* `;`

Dans une fonction dont le type de retour est `void`, seul le premier format (sans expression) est accepté. Dans une fonction dont le type de retour est différent de `void`, seul le second format (avec expression) est accepté.

Dans les deux cas, l'instruction `return` arrête **immédiatement** l'exécution d'une fonction (le reste du corps de la fonction n'est pas exécuté) et l'exécution se poursuit dans la fonction appelante.

Dans une fonction dont le type de retour n'est pas `void`, la valeur de l'expression *expression* correspond à la valeur renvoyée par la fonction.

Exemple :

```
int g(int i) {
   if (i >= 0)
      return 1; // Cette instruction arrête l'exécution de g en renvoyant la
valeur 1
   // Le reste de la fonction n'est donc exécutée que si i < 0
   return 0;
}

void f(int i) {
   if (i == 0) {
      printf("i == 0\n");
      return;
   }
   printf("i != 0\n");
   if (g(i) == 1) {
      printf("i > 0\n");
      return;
   }
   printf("i < 0\n");
}
```

L'exécution de la fonction `f` avec la valeur `-1` produit l'affichage suivant :

```
i != 0
i < 0
```

L'exécution de la fonction `f` avec la valeur `0` produit l'affichage suivant :

```
i == 0
```

L'exécution de la fonction `f` avec la valeur `1` produit l'affichage suivant :

```
i != 0
i > 0
```

Dans une fonction dont le type de retour est `void`, si l'exécution arrive à la fin du corps de la fonction sans avoir rencontré de `return;`, la fonction se termine normalement comme s'il y avait un `return;` à la toute fin du corps de la fonction.

Dans une fonction dont le type de retour n'est pas `void`, si l'exécution arrive à la fin du corps de la fonction sans avoir rencontré de `return` *expression* `;`, le comportement n'est pas défini (donc ne le faite pas). *Note :* il y a une exception depuis le standard C99 concernant la fonction `main` qui est supposée renvoyer dans ce cas la valeur `0`.

7. Les types dérivés / composés

Les types dérivés sont créés à partir des types de base vus précédemment. Dans cette partie, nous traiterons des pointeurs, des structures, des énumérations et des alias de type.

7.1. Pointeurs

Remarque préliminaire : les pointeurs sont souvent un point délicat l'apprentissage du C.

Un pointeur est une variable qui contient l'*adresse* en mémoire d'un autre objet (une variable la plupart du temps ou une fonction).

De plus, en C, le type du pointeur donne une information sur le *type* de l'objet pointé (`int` , `float` ...).

7.1.1. Déclaration/Définition

Pour définir un pointeur, il suffit d'ajouter une étoile (`*`) devant l'identifiant de la variable dans une déclaration ou une définition.

Par exemple :

```
int a;
int * p;
```

Dans cet exemple, `a` est une variable de type `int` . `p` est une variable de type pointeur vers une variable de type `int` . `a` peut stocker un entier et `p` peut stocker l'adresse en mémoire d'une variable de type `int` .

Autre exemple :

```
int ** pp;
```

Ici, `pp` est une variable de type pointeur vers une variable de type pointeur vers une variable de type `int` .

7.1.2. Initialisation et opérateur `&` (adresse de)

```
int a;
int * p;
```

Nous avons ici la définition de deux variables, `a` et `p` . Comme vu précédemment, en rencontrant ces définitions, le compilateur sait qu'il faut prévoir l'allocation de la mémoire nécessaire pour stocker ces deux variables. On peut donc dès lors stocker des valeurs dans ces deux variables (un entier dans la variable `a` et l'adresse d'un entier dans la variable `p`).

La valeur initiale de ces deux variables dépend de leur durée de stockage :

- si statique (variables globales ou locales avec le mot clé `static`) : 0 pour `a` et l'adresse 0 (invalide) pour `p`
- si automatique (variables locales) : une valeur non définie pour `a` et une adresse non définie pour `p`

Pour pouvoir faire quelque chose d'intéressant avec notre pointeur `p` , il faut y stocker l'adresse d'une variable de type `int` existante. Dans un premier temps (nous verrons d'autres

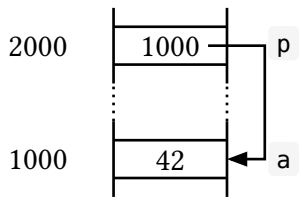
méthodes plus tard, notamment quand nous parlerons d'allocation mémoire dynamique), nous pouvons utiliser l'opérateur `&` (adresse de). Cet opérateur renvoie l'adresse de son opérande.

On peut donc écrire :

```
int a;
int * p;

a = 42;  // La valeur 42 est mise dans la variable a
p = &a;  // L'adresse de la variable a est mise dans la variable p
```

`&a` renvoie l'adresse de la variable `a` et cette adresse est stockée dans la variable `p`.

La situation est expliquée par la figure suivante. On suppose que la variable `a` est stockée en mémoire à l'adresse 1000 et que la variable `p` est stockée à l'adresse 2000 (valeurs totalement arbitraires dans cet exemple). La variable `a` vaut `42`, donc la valeur `42` est stockée à l'adresse 1000. L'adresse de la variable `a` (donc la valeur `1000`) a été affectée au pointeur `p`, donc la valeur `1000` est stockée à l'adresse 2000.



7.1.3. Déréférencement

L'opération la plus importante sur un pointeur est l'opération de *déréférencement*.

Toutes les opérations classiques affectent le contenu du pointeur lui-même (autrement dit l'adresse qu'il contient). Par exemple l'opérateur d'affectation permet de changer le contenu du pointeur :

```
int a = 42;
int b = 43;
int * p = &a; // p contient l'adresse de la variable a

p = &b;       // p contient maintenant l'adresse de la variable b
              // Les valeurs de a et b n'ont pas été modifiées
```

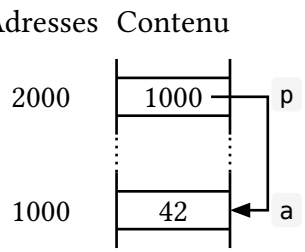
L'opérateur de déréférencement permet de manipuler non pas le contenu du pointeur lui-même mais la case mémoire située à l'adresse contenue dans le pointeur.

Cet opérateur de déréférencement est l'opérateur `*` unaire. Son unique opérande, situé à droite de l'opérateur, doit être un pointeur contenant une adresse valide (et dont le type n'est pas `void *`).

Considérons l'exemple suivant :

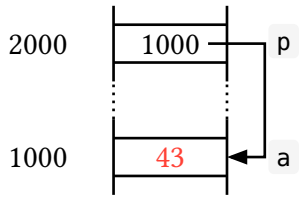
```
int a = 42;
int * p = &a;

*p = 43;
```

Avant l'exécution de la dernière instruction, la mémoire est dans la situation suivante :



`*p` représente donc la case mémoire dont l'adresse est celle contenue dans `p`, c'est-à-dire la case mémoire située à l'adresse 1000 (donc la case mémoire contenant la variable `a`). Cette opération de déréférencement est située à gauche d'une affectation (`*p = 43`), donc la valeur 43 est stockée à cette adresse 1000.

Après exécution, on a donc la situation suivante :



La variable `a` vaut donc maintenant la valeur 43.

L'opération de déréférencement permet d'accéder à la valeur pointée par le pointeur en lecture ou en écriture :

```
int a = 42;
int * p = &a;

*p = 43; // Écriture, la valeur de a est modifiée
int b = *p; // Lecture, la valeur de a est lue et est affectée à la variable b
```

Le type de l'expression `*p` est le type de l'objet pointé par `p`. Par exemple, si `p` est un pointeur vers un `int` (`int * p`), `*p` sera de type `int`.

Il n'est pas possible de déréférencer un pointeur `void` (`void *`).

7.1.4. Pointeur nul

Les pointeurs ont une valeur nulle. Un pointeur dont la valeur est nulle ne pointe pas vers un objet ou une fonction. Déréférencer un pointeur ayant la valeur nulle est un comportement non défini (la plupart du temps le programme va planter).

Cette valeur nulle peut être obtenue grâce à la constante `NULL`.

7.1.5. Dangers des pointeurs

⚠️ **ATTENTION !** Pour pouvoir déréférencer un pointeur, il faut qu'il contienne l'adresse d'une zone mémoire correctement allouée (réservée), de taille suffisante, et dont les persmissions sont correctes (ex. en lecture/écriture si on souhaite faire une écriture).

Aucun contrôle n'est effectué par le compilateur ou à l'exécution. Si le pointeur contient une adresse invalide, la lecture ou l'écriture va se faire avec cette adresse (et donc n'importe où en mémoire), ce qui entraînera soit un plantage, soit un comportement inattendu de votre programme, soit ouvrira une faille de sécurité.

En particulier, un pointeur non initialisé ne contient jamais une adresse d'une zone mémoire valide. Vous ne devez donc pas le déréférencer.

```
int * p; // Définition du pointeur p mais sans valeur initiale
*p = 43; // DANGER !
```

Dans le code ci-dessus, `p` est une variable de type pointeur vers un `int`. La première ligne réalise la définition de cette variable, le compilateur alloue donc suffisamment de mémoire pour stocker la valeur de `p`. Néanmoins, cette valeur n'est pas précisée. La valeur de `p` sera soit 0 (donc `NULL`) si c'est une variable globale, ou une valeur non définie si c'est une variable locale. Donc dans aucun des deux cas, la valeur de `p` correspond à l'adresse d'une zone mémoire où l'on pourrait légitimement lire ou écrire. Il est donc très dangereux de déréférencer ce pointeur en écriture (ni en lecture) comme c'est fait dans l'exemple, car on se met à écrire la valeur `43` dans une zone où l'on ne devrait pas. Le plus souvent le code ci-dessus entraînera l'arrêt brutal du programme par le système d'exploitation avec une *erreur de segmentation* (*segmentation fault*) pour des raisons expliquées dans la troisième partie du cours.

Si vous souhaitez déréférencer un pointeur, assurez-vous qu'il contient bien l'adresse d'une zone mémoire valide. Cela peut se faire en le faisant pointer sur une autre variable allouée par le compilateur, par exemple :

```
int a;        // Le compilateur alloue la mémoire pour la variable a
int * p = &a; // Le compilateur alloue la mémoire pour la variable p
              // et la valeur de p vaut l'adresse de la variable a
*p = 43;      // OK
```

Il est également possible de stocker dans le pointeur l'adresse d'une zone mémoire allouée dynamiquement (voir plus loin dans le cours) à l'aide de la fonction `malloc` par exemple.

7.2. Structures

Un structure permet de créer un nouveau type par regroupement d'un ou plusieurs membres pouvant avoir des types différents.

7.2.1. Définition d'une structure

La définition d'une structure a la syntaxe suivante :

`struct` *nom* `{` *membres* `};`

où *membres* est une liste de déclarations de la forme :

type nomchamp `;`

Par exemple :

```
struct s {
   int a;
```

```
    float b;
};
```

La définition ci-dessus introduit un nouveau type `struct s` (**attention**, le nom du type est bien `struct s` et non pas simplement `s`) composé de deux membres : `a` de type `int` et `b` de type `float`.

À noter qu'une fois le type défini, il n'est pas possible de modifier une structure en ajoutant ou supprimant un membre : dans le reste du programme, une structure de type `struct s` contiendra toujours ces deux membres et uniquement ceux-là.

7.2.2. Création d'une variable de ce type

Une fois le type défini, il est alors possible de créer des variables de ce nouveau type de la manière classique :

```
struct s s1;
```

Ici `s1` est une variable de type `struct s`.

Il est également possible d'initialiser la valeur des membres dès la déclaration de la variable avec les deux syntaxes suivantes :

```
struct s s1 = {1, 2.0};            // Ancienne syntaxe, a = 1, b = 2.0
struct s s2 = {.a = 3, .b = 4.0};  // Nouvelle syntaxe, a = 3, b = 4.0
```

Note : Il est possible de créer une variable dès la définition de la structure avec la syntaxe :

```
struct s {
    int a;
    float b;
} s1;
```

On réalise alors à la fois la définition de la structure `struct s` et la définition de la variable `s1` de type `struct s`.

Il est également possible dans ce cas de ne pas nommer la structure (la variable reste nommée) :

```
struct {
    int a;
    float b;
} s1;
```

7.2.3. Accès aux champs d'une structure (opérateurs `.` et `->`)

L'opérateur `.` permet d'accéder à un champ d'une structure :

```
struct s s1;
s1.a = 42;
s1.b = 3.14;
```

Si on dispose d'un pointeur vers une variable de type structure, il faut tout d'abord déréférencer le pointeur pour pouvoir ensuite accéder aux membres :

```
struct s s1;
struct s *p = &s1;
```

```
(*p).a = 42;
(*p).b = 3.14;
```

Ici les parenthèses sont nécessaires car l'opérateur `.` est prioritaire sur l'opérateur de déréférencement `*`.

Cependant, il existe un raccourci d'écriture très utilisé, l'opérateur `->`. Il réalise un déréférencement de son opérande de gauche (qui doit donc être un pointeur) puis l'accès au membre indiqué par son opérande droit. Le code ci-dessus peut donc être écrit plus lisiblement :

```
struct s s1;
struct s *p = &s1;
p->a = 42;
p->b = 3.14;
```

7.3. Énumérations

Une énumération permet de créer un nouveau type et de donner explicitement des noms aux différentes valeurs que ce type peut prendre.

7.3.1. Définition

La syntaxe de la définition d'une énumération est la suivante :

`enum` *nom* `{` *enumerator-list* `};`

où *enumerator-list* est une liste d'identifiants, séparés par des virgules.

Par exemple :

```
enum color {RED, BLUE, GREEN};
```

Cette définition créé un nouveau type `enum color` (**attention**, le nom du type est `enum color` et non simplement `color`) et trois constantes `RED`, `BLUE` et `GREEN` qui sont des valeurs admissibles pour cette énumération.

7.3.2. Usage

Une fois le type défini, il est possible de créer une variable de ce type :

```
enum color c1;
c1 = RED;
if (c1 == GREEN) {
   // ...
}
```

7.3.3. Pour aller plus loin

Une valeur numérique est affectée à chaque constante lors de la définition, à partir de 0 (donc dans l'exemple ci dessus, `RED` a la valeur 0, `BLUE` la valeur 1 et `GREEN` la valeur 2).

Il est possible de spécifier une valeur explicite pour une des constantes à l'aide de la syntaxe suivante :

```
enum color2 {RED, BLUE = 3, GREEN};
```

`RED` aura alors la valeur 0, `BLUE` la valeur 3 et `GREEN` la valeur 4.

7.4. Alias de type (`typedef`)

Il est possible de donner un nouveau nom à un type existant à l'aide de `typedef` grâce à la syntaxe :

`typedef` *type nouveau-nom* `;`

Exemple :

```
typedef int myint;

myint i = 3;
```

`typedef` est le plus souvent utilisé pour donner un nom plus court aux types structure et énumération.

Par exemple, avec la définition suivante :

```
struct s {
   int a;
   float b;
};
```

Le nom du type est `struct s` et donc si on veut déclarer une variable de ce type, il faut écrire :

```
struct s s1;
```

Il est alors possible de donner un nom plus court grâce à `typedef` :

```
struct s {
   int a;
   float b;
};

typedef struct s s_t;
```

Ici, `s_t` est un alias pour `struct s`. Il est donc par la suite possible de l'utiliser :

```
s_t s1 = {1, 2.0};
```

Il est également possible de combiner `typedef` et la définition de la structure :

```
typedef struct s {
   int a;
   float b;
} s_t;
```

ou même :

```
typedef struct {
   int a;
   float b;
} s_t;
```

8. La mémoire

8.1. Introduction

La mémoire occupée par un programme est décomposée en trois grandes zones :

- Une zone qui contient le code et les données allouées statiquement par le compilateur pour toute la durée de l'exécution. Sa taille ne change pas lors de l'exécution. On y trouve notamment :
  - Le code machine résultant de la compilation des instructions du corps des fonctions
  - Les variables dont la durée de stockage est statique (c'est-à-dire principalement les variables globales)
- Le **tas** (*heap*) qui va contenir les zones mémoires allouées dynamiquement par le programmeur à l'exécution
- La **pile** (*stack*) qui va contenir toutes les données relatives à l'exécution des fonctions (variables automatiques, arguments, etc.)

Cette section va notamment présenter la pile et le tas.

8.2. Opérateurs `sizeof` et `_Alignof`

8.2.1. `sizeof`

Il est souvent nécessaire, en particulier lorsque l'on doit manipuler directement la mémoire, de connaître la taille d'un objet. L'opérateur `sizeof` permet de le faire. Il présente deux syntaxes :

- `sizeof(` *type* `)`
- `sizeof` *expression*

Dans les deux cas, `sizeof` renvoie la taille, en nombre d'octets, que prendrait un objet du type indiqué ou du type de l'expression indiquée.

Attention, la taille de certains types dépend de l'architecture sur laquelle le programme est exécuté. C'est notamment le cas de la plupart des types entiers de base du C (par exemple `int`).

Exemple :

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
  printf("char:\t\t%zu\n", sizeof(char));
  printf("short:\t\t%zu\n", sizeof(short));
  printf("int:\t\t%zu\n", sizeof(int));
  printf("long:\t\t%zu\n", sizeof(long));
  printf("long long:\t%zu\n", sizeof(long long));
  printf("float:\t\t%zu\n", sizeof(float));
  printf("double:\t\t%zu\n", sizeof(double));
  printf("long double:\t%zu\n", sizeof(long double));
  return EXIT_SUCCESS;
}
```

Exécuté sur une machine de TP de l'école, le programme précédent produit l'affichage suivant :

```
char:           1
short:          2
int:            4
long:           8
long long:      8
float:          4
double:         8
long double:    16
```

À noter : appliqué à un tableau, `sizeof` renvoie la taille totale du tableau. Appliqué à un pointeur, `sizeof` renvoie la taille du pointeur et non la taille de l'objet pointé par le pointeur.

Illustration :

```
int tab[10];
printf("tab: %zu\n", sizeof tab); // Affiche "tab: 40"
short *ptr;
printf("ptr: %zu\n", sizeof ptr); // Affiche "ptr: 8"
```

8.2.2. `_Alignof`

Chaque type possède une contrainte d'alignement. Il s'agit d'un entier positif, puissance de 2, exprimé en nombre d'octets. Un objet d'un type donné doit être stocké en mémoire à une adresse multiple de la contrainte d'alignement pour ce type.

Il est possible d'obtenir cette contrainte grâce à l'opérateur `_Alignof` avec la syntaxe :

`_Alignof(` *type* `)`

On peut également utiliser la macro `alignof` définie dans le fichier `<stdalign.h>`.

Exemple :

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
  printf("char:\t\t%zu\n", _Alignof(char));
  printf("short:\t\t%zu\n", _Alignof(short));
  printf("int:\t\t%zu\n", _Alignof(int));
  printf("long:\t\t%zu\n", _Alignof(long));
  printf("long long:\t%zu\n", _Alignof(long long));
  printf("float:\t\t%zu\n", _Alignof(float));
  printf("double:\t\t%zu\n", _Alignof(double));
  printf("long double:\t%zu\n", _Alignof(long double));
  return EXIT_SUCCESS;
}
```

Exécuté sur une machine de TP de l'école, le programme précédent produit l'affichage suivant :

```
char:           1
short:          2
int:            4
long:           8
long long:      8
float:          4
```

```
double:         8
long double:    16
```

8.2.3. Taille d'un structure

À cause des contraintes d'alignement, la taille en mémoire d'une structure peut être supérieure à la somme de la taille de chacun de ses membres.

Exemple :

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

struct { int a; char b; int c; } s;

int main(int argc, char *argv[])
{
  printf("%zu\n", sizeof(s));
  return EXIT_SUCCESS;
}
```

Exécuté sur une machine de TP (où la taille des `int` est de 4 et celle des `char` est de 1), on obtient 12.

En effet, le premier membre `a` doit être stocké à une adresse multiple de 4 et fait 4 octets. Le deuxième membre, `b` peut être stocké à n'importe quelle adresse et fait 1 octet. Le troisième membre, `c` doit être stocké à une adresse multiple de 4 et fait 4 octets.

Le compilateur va donc insérer trois octets de bourrage (*padding*) entre `b` et `c` de telle sorte à ce que `c` puisse être correctement aligné.

8.3. La pile

8.3.1. Utilité de la pile

Considérons la fonction suivante :

```c
int f(int a) {
   int b = a - 1;
   if (b == 0)
      return 0;
   else
      return f(b) + 1;
}
```

Le compilateur ne peut pas savoir, à la compilation, combien de place en mémoire est nécessaire pour l'exécution de cette fonction. Lors de son exécution, la fonction `f` a besoin au minimum de 4 octets pour stocker son argument `a` et de 4 octets pour stocker sa variable locale `b` (en supposant que les `int` sont stockés sur 4 octets). Cependant, comme c'est une fonction récursive, plusieurs appels à `f` peuvent être nécessaires et chaque appel à cette fonction aura besoin de sa propre copie de son argument `a` et de sa variable locale `b`.

Les données relatives à l'exécution des fonctions vont donc être stockées dans une zone mémoire dédiée nommée la **pile** (*stack*).

On va trouver classiquement dans la pile (les détails peuvent varier en fonction des architectures et des optimisations) :

- la valeur des arguments ;
- les variables locales (sauf celles `static`) ;
- des informations sauvegardées pour permettre la reprise de l'exécution à la fin de la fonction (par exemple l'adresse de retour) ;
- et la valeur de retour de la fonction.

La taille de cette zone croit lors de l'appel d'une fonction pour y stocker les informations liées à son exécution. Elle décroît une fois la fonction terminée supprimant ainsi les données qui ne sont plus utiles.

Au niveau matériel, le plus souvent, un registre du processeur, nommé le *pointeur de pile* (*stack pointer*), est dédié à la gestion de la pile. Il contient l'adresse du sommet de la pile (soit la dernière valeur mise sur la pile, soit la prochaine case libre de la pile en fonction des architectures). Des instructions, insérées par le compilateur au début et à la fin des fonctions ainsi qu'au niveau des appels de fonction, permettent de gérer cette zone en modifiant la valeur de ce pointeur.

8.3.2. Exemple

On considère le programme suivant :

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

int g(int a, int b) {
   // <-- Étape 5
   return a * b;
}

int f(int c, int d) {
   // <-- Étape 3
   int e;
   e = g(c, d); // <-- Étape 4
   // <-- Étape 6
   return e;
}

int main(int argc, char *argv[]) {
   // <-- Étape 1
   int t = f(4, 5); // <-- Étape 2
   // <-- Étape 7
   printf("%d\n", t);
   return EXIT_SUCCESS;
}
```

Voici l'état de la pile à l'entrée de la fonction `main` (étape 1) :

Adresse	Valeur	Description
`7fffd97c`	`xxxxxxxx`	Variable locale `t` (sommet de la pile)

Adresse	Valeur	Description
`7fffd980`	`7fffda88`	Valeur de `argv`
`7fffd984`	`00000001`	Valeur de `argc`
`7fffd988`	`0007f29d`	Adresse de retour de `main`
	…	

Voici l'état de la pile lors de la préparation de l'appel à la fonction `f` (étape 2) :

Adresse	Valeur	Description
`7fffd970`	`00000005`	Deuxième argument de `f` (sommet de la pile)
`7fffd974`	`00000004`	Premier argument de `f`
`7fffd978`	`0007f354`	Adresse de retour de `f` (dans `main`)
`7fffd97c`	`xxxxxxxx`	Variable locale `t`
`7fffd980`	`7fffda88`	Valeur de `argv`
`7fffd984`	`00000001`	Valeur de `argc`
`7fffd988`	`0007f29d`	Adresse de retour de `main`
	…	

Lors de la préparation de l'appel à une fonction, le compilateur met sur le sommet de la pile la valeur des différents arguments de la fonction (ici les valeurs `4` et `5`), ainsi que d'autres informations (par exemple ici l'adresse de retour, c'est-à-dire l'adresse de l'instruction à exécuter une fois que la fonction sera terminée).

Voici l'état de la pile à l'entrée de la fonction `f` (étape 3) :

Adresse	Valeur	Description
`7fffd96c`	`xxxxxxxx`	Variable locale `e` (sommet de la pile)
`7fffd970`	`00000005`	Deuxième argument de `f` (`d`)
`7fffd974`	`00000004`	Premier argument de `f` (`e`)
`7fffd978`	`0007f354`	Adresse de retour de `f` (dans `main`)
`7fffd97c`	`xxxxxxxx`	Variable locale `t`
`7fffd980`	`7fffda88`	Valeur de `argv`
`7fffd984`	`00000001`	Valeur de `argc`
`7fffd988`	`0007f29d`	Adresse de retour de `main`
	…	

À l'entrée de la fonction, le compilateur réserve sur la pile l'espace nécessaire pour les variables locales (ici `e`). De plus, la valeur des arguments qui ont été placés sur la pile lors de l'appel à

la fonction, sont accessibles par le nom qui leur a été attribué dans la signature de la fonction (`d` et `e`).

Voici l'état de la pile à lors de la préparation de l'appel à la fonction `g` (étape 4) :

Adresse	Valeur	Description
`7fffd960`	`00000005`	Deuxième argument de `g` (sommet de la pile)
`7fffd964`	`00000004`	Premier argument de `g`
`7fffd968`	`0007f498`	Adresse de retour de `g` (dans `f`)
`7fffd96c`	`xxxxxxxx`	Variable locale `e`
`7fffd970`	`00000005`	Deuxième argument de `f` (`d`)
`7fffd974`	`00000004`	Premier argument de `f` (`e`)
`7fffd978`	`0007f354`	Adresse de retour de `f` (dans `main`)
`7fffd97c`	`xxxxxxxx`	Variable locale `t`
`7fffd980`	`7fffda88`	Valeur de `argv`
`7fffd984`	`00000001`	Valeur de `argc`
`7fffd988`	`0007f29d`	Adresse de retour de `main`
	…	

Comme précédemment, lors de la préparation de l'appel à la fonction `g`, le compilateur empile la valeur des arguments passés à cette fonction ainsi que l'adresse de retour.

Voici l'état de la pile à l'entrée de la fonction `g` (étape 5) :

Adresse	Valeur	Description
`7fffd960`	`00000005`	Deuxième argument de `g` (`b`) (sommet de la pile)
`7fffd964`	`00000004`	Premier argument de `g` (`a`)
`7fffd968`	`0007f498`	Adresse de retour de `g` (dans `f`)
`7fffd96c`	`xxxxxxxx`	Variable locale `e`
`7fffd970`	`00000005`	Deuxième argument de `f` (`d`)
`7fffd974`	`00000004`	Premier argument de `f` (`e`)
`7fffd978`	`0007f354`	Adresse de retour de `f` (dans `main`)
`7fffd97c`	`xxxxxxxx`	Variable locale `t`
`7fffd980`	`7fffda88`	Valeur de `argv`
`7fffd984`	`00000001`	Valeur de `argc`
`7fffd988`	`0007f29d`	Adresse de retour de `main`
	…	

Voici l'état de la pile à la sortie de la fonction `g` (étape 6) :

Adresse	Valeur	Description
`7fffd96c`	`00000014`	Variable locale `e` (sommet de la pile)
`7fffd970`	`00000005`	Deuxième argument de `f` (`d`)
`7fffd974`	`00000004`	Premier argument de `f` (`e`)
`7fffd978`	`0007f354`	Adresse de retour de `f` (dans `main`)
`7fffd97c`	`xxxxxxxx`	Variable locale `t`
`7fffd980`	`7fffda88`	Valeur de `argv`
`7fffd984`	`00000001`	Valeur de `argc`
`7fffd988`	`0007f29d`	Adresse de retour de `main`
		...

En sortant d'une fonction, les informations relative à l'exécution sont dépilées et l'adresse de retour est récupérée pour pouvoir revenir à la fonction appelante.

Voici l'état de la pile à la sortie de la fonction `f` (étape 7) :

Adresse	Valeur	Description
`7fffd97c`	`00000014`	Variable locale `t` (sommet de la pile)
`7fffd980`	`7fffda88`	Valeur de `argv`
`7fffd984`	`00000001`	Valeur de `argc`
`7fffd988`	`0007f29d`	Adresse de retour de `main`
		...

8.3.3. Passage des arguments d'une fonction

Lors d'un appel de fonction, les valeurs des arguments sont placées sur la pile (les détails peuvent varier en fonction des architectures...). La fonction appelée a accès (et peut modifier) ces valeurs sur la pile.

Quand la fonction se termine, ces arguments sont supprimés de la pile. Toute modification faite par la fonction sur ses arguments est donc perdue. La fonction appelante n'a pas été affectée.

Les arguments des fonctions sont dits passés *par valeur*.

8.3.3.1. Exemple

Voici un exemple pour illustrer le mécanisme et ses conséquences :

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

void swap(int a, int b) {
```

```c
    // <-- Étape 3
    // On échange a et b
    int c = a;
    a = b;
    b = c;
    // <-- Étape 4
}

int main(int argc, char *argv[]) {
    // <-- Étape 1
    int a = 3;
    int b = 4;
    // <-- Étape 2
    swap(a, b);
    // <-- Étape 5
    printf("a = %d, b = %d\n", a, b);
    return EXIT_SUCCESS;
}
```

L'exécution de ce code affiche à l'écran `a = 3, b = 4`, c'est-à-dire que la fonction `swap` n'a eu aucun impact sur les variables locales `a` et `b` de la fonction `main`.

Explications (on simplifie en supposant que la pile ne contient que les variables locales et les arguments de fonctions pour simplifier la lecture) :

État de la pile à l'entrée de la fonction `main` (étape 1) :

Adresse	Valeur	Description
`7fffd978`	`xxxxxxxx`	Variable locale `b` (fonction `main`)
`7fffd97c`	`xxxxxxxx`	Variable locale `a` (fonction `main`)
`7fffd980`	`7fffda88`	Valeur de `argv`
`7fffd984`	`00000001`	Valeur de `argc`
	...	

À l'entrée de la fonction `main`, le compilateur va faire en sorte de réserver dans la pile de la place pour stocker les deux variables locales `a` et `b` de la fonction `main`.

État de la pile lors de la préparation de l'appel de la fonction `swap` (étape 2) :

Adresse	Valeur	Description
`7fffd970`	`00000004`	Deuxième argument de la fonction `swap`
`7fffd974`	`00000003`	Premier argument de la fonction `swap`
`7fffd978`	`00000004`	Variable locale `b` (fonction `main`)
`7fffd97c`	`00000003`	Variable locale `a` (fonction `main`)
`7fffd980`	`7fffda88`	Valeur de `argv`
`7fffd984`	`00000001`	Valeur de `argc`

Adresse	Valeur	Description
	...	

On note que les variables `a` et `b` ont pris les valeurs `3` et `4` lors de l'exécution des deux premières lignes de la fonction `main`. De plus, en préparation de l'appel à la fonction `swap`, les valeurs des deux arguments (`3` et `4`) ont été mises sur le sommet de la pile.

État de la pile lors à l'entrée de la fonction `swap` (étape 3) :

Adresse	Valeur	Description
`7fffd96c`	`xxxxxxxx`	Variable locale `c` (fonction `swap`)
`7fffd970`	`00000004`	Deuxième argument de la fonction `swap` (`b` vu de `swap`)
`7fffd974`	`00000003`	Premier argument de la fonction `swap` (`a` vu de `swap`)
`7fffd978`	`00000004`	Variable locale `b` (fonction `main`)
`7fffd97c`	`00000003`	Variable locale `a` (fonction `main`)
`7fffd980`	`7fffda88`	Valeur de `argv`
`7fffd984`	`00000001`	Valeur de `argc`
	...	

Le compilateur fait en sorte de réserver de la place pour la variable locale `c` de la fonction `swap`. De plus, les valeurs des deux arguments de la fonction sont maintenant accessibles avec les identifiants `a` et `b` tels que déclarés dans la définition de la fonction `swap`.

État de la pile à la fin de la fonction `swap` (étape 4) :

Adresse	Valeur	Description
`7fffd96c`	`00000003`	Variable locale `c` (fonction `swap`)
`7fffd970`	`00000003`	Deuxième argument de la fonction `swap` (`b` vu de `swap`)
`7fffd974`	`00000004`	Premier argument de la fonction `swap` (`a` vu de `swap`)
`7fffd978`	`00000004`	Variable locale `b` (fonction `main`)
`7fffd97c`	`00000003`	Variable locale `a` (fonction `main`)
`7fffd980`	`7fffda88`	Valeur de `argv`
`7fffd984`	`00000001`	Valeur de `argc`
	...	

À la fin de la fonction, les valeurs des arguments `a` et `b` ont bien été échangées. Néanmoins, cet échange a affecté les copies réalisées et non les valeurs vues depuis la fonction `main`.

État de la pile lors du retour dans la fonction `main` (étape 5) :

Adresse	Valeur	Description
`7fffd978`	`00000004`	Variable locale `b` (fonction `main`)
`7fffd97c`	`00000003`	Variable locale `a` (fonction `main`)
`7fffd980`	`7fffda88`	Valeur de `argv`
`7fffd984`	`00000001`	Valeur de `argc`
	...	

De retour dans la fonction `main`, toutes les valeurs qui avaient été empilées pour l'appel de la fonction `swap` sont supprimées de la pile. Les valeurs des deux variables locales `a` et `b` n'ont pas été modifiées.

8.3.3.2. Passage par adresse

Pour qu'une fonction puisse avoir, par l'intermédiaire de ses arguments, des effets visibles sur la fonction appelante, il est possible de passer les arguments *par adresse*.

Le passage par adresse consiste à fournir à la fonction appelée non pas la valeur de l'argument mais l'adresse à laquelle se situe l'argument. La fonction appelée pourra donc utiliser l'adresse fournie pour récupérer ou modifier la valeur de l'argument. Cela permet donc à la fonction appelée d'avoir des effets visibles en dehors d'elle-même via ses arguments. Cette méthode à également d'autres avantages, par exemple, si la taille du type d'un argument est importante, le passage par adresse permet d'éviter une copie de la valeur de l'argument, copie qui peut être coûteuse en temps ou en espace mémoire.

8.3.3.2.1. Exemple

Nous modifions l'exemple précédent en utilisant un passage par adresse. La fonction `swap` reçoit donc maintenant l'adresse de ses deux arguments entiers, et donc elle reçoit deux pointeurs vers des `int`.

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

void swap(int *a, int *b) { // Notez les pointeurs
   // <-- Étape 3
   int c = *a;
   *a = *b;
   *b = *c;
   // <-- Étape 4
}

int main(int argc, char *argv[]) {
   // <-- Étape 1
   int a = 3;
   int b = 4;
   // <-- Étape 2
   swap(&a, &b);
   // <-- Étape 5
   printf("a = %d, b = %d\n", a, b);
```

```
    return EXIT_SUCCESS;
}
```

L'exécution de ce code affiche à l'écran `a = 4, b = 3`. Les valeurs des deux variables locales de la fonction `main` ont bien été échangées par la fonction `swap`.

Explications (on simplifie en supposant que la pile ne contient que les variables locales et les arguments de fonctions pour simplifier la lecture) :

État de la pile à l'entrée de la fonction `main` (étape 1) :

Adresse	Valeur	Description
`7fffd978`	`xxxxxxxx`	Variable locale `b` (fonction `main`)
`7fffd97c`	`xxxxxxxx`	Variable locale `a` (fonction `main`)
`7fffd980`	`7fffda88`	Valeur de `argv`
`7fffd984`	`00000001`	Valeur de `argc`
	...	

État de la pile lors de la préparation de l'appel de la fonction `swap` (étape 2) :

Adresse	Valeur	Description
`7fffd970`	`7fffd978`	Deuxième argument de la fonction `swap` (adresse de `b`)
`7fffd974`	`7fffd97c`	Premier argument de la fonction `swap` (adresse de `a`)
`7fffd978`	`00000004`	Variable locale `b` (fonction `main`)
`7fffd97c`	`00000003`	Variable locale `a` (fonction `main`)
`7fffd980`	`7fffda88`	Valeur de `argv`
`7fffd984`	`00000001`	Valeur de `argc`
	...	

En préparation de l'appel à la fonction `swap`, les valeurs des deux arguments, c'est-à-dire dans notre cas l'adresse des variables locales `a` (adresse `7fffd97c`) et `b` (adresse `7fffd978`), sont mises sur le sommet de la pile.

État de la pile lors à l'entrée de la fonction `swap` (étape 3) :

Adresse	Valeur	Description
`7fffd96c`	`xxxxxxxx`	Variable locale `c` (fonction `swap`)
`7fffd970`	`7fffd978`	Deuxième argument de la fonction `swap` (`b` vu de `swap`)
`7fffd974`	`7fffd97c`	Premier argument de la fonction `swap` (`a` vu de `swap`)
`7fffd978`	`00000004`	Variable locale `b` (fonction `main`)
`7fffd97c`	`00000003`	Variable locale `a` (fonction `main`)

Adresse	Valeur	Description
`7fffd980`	`7fffda88`	Valeur de `argv`
`7fffd984`	`00000001`	Valeur de `argc`
	...	

État de la pile à la fin de la fonction `swap` (étape 4) :

Adresse	Valeur	Description
`7fffd96c`	`00000003`	Variable locale `c` (fonction `swap`)
`7fffd970`	`7fffd978`	Deuxième argument de la fonction `swap` (`b` vu de `swap`)
`7fffd974`	`7fffd97c`	Premier argument de la fonction `swap` (`a` vu de `swap`)
`7fffd978`	`00000003`	Variable locale `b` (fonction `main`)
`7fffd97c`	`00000004`	Variable locale `a` (fonction `main`)
`7fffd980`	`7fffda88`	Valeur de `argv`
`7fffd984`	`00000001`	Valeur de `argc`
	...	

Via les opérations de déréférencements de `a` et `b` (`*a` et `*b`), le corps de la fonction `swap` va échanger les valeurs des cases mémoires situées aux adresses `7fffd97c` et `7fffd978` (c'est-à-dire là où sont situées les variables locales `a` et `b` de la fonction `main`).

État de la pile lors du retour dans la fonction `main` (étape 5) :

Adresse	Valeur	Description
`7fffd978`	`00000003`	Variable locale `b` (fonction `main`)
`7fffd97c`	`00000004`	Variable locale `a` (fonction `main`)
`7fffd980`	`7fffda88`	Valeur de `argv`
`7fffd984`	`00000001`	Valeur de `argc`
	...	

De retour dans la fonction `main`, toutes les valeurs qui avaient été empilées pour l'appel de la fonction `swap` sont supprimées de la pile. Néanmoins, cette fonction ayant modifié directement d'autres zones mémoires, ces modifications restes visibles, en l'occurrence ici, les modifications des variables locales `a` et `b` de la fonction `main`.

8.4. Le tas

Dans de nombreuses situations, il est nécessaire de disposer de mémoire pour stocker des informations dont la taille n'est pas connue à la compilation (quelques exemples : analyse d'une page web reçue via une connexion réseau, saisie d'un texte arbitrairement long par l'utilisateur, etc.). Sauf à prévoir un pire cas (taille maximale des données très grande), ce qui

n'est pas efficace ou même parfois impossible, il est donc nécessaire de pouvoir demander à allouer (réserver) explicitement (c'est au développeur de le faire) de la mémoire dynamiquement (c'est-à-dire à l'exécution). Toutes ces allocations dynamiques ont lieu dans une zone de la mémoire, nommée le *tas* (*heap* en anglais), dont la taille varie au cours de l'exécution.

Les trois fonctions principales pour manipuler le tas sont les suivantes :
- `malloc` : alloue de la mémoire
- `free` : libère de la mémoire précédemment allouée
- `realloc` : permet de modifier la taille d'une zone précédemment allouée

Elles sont toutes les trois déclarées dans le fichier d'en-tête `stdlib.h`.

8.4.1. `malloc`

La fonction `malloc` a la signature suivante :

```
void *malloc(size_t size);
```

La fonction `malloc` permet d'allouer (réserver) de la mémoire dynamiquement dans le tas. Elle prend un seul argument `size` de type `size_t` indiquant le nombre d'octets à allouer.

Note : `size_t` est un type entier, non signé (donc positif ou nul), de taille suffisante pour représenter la taille théorique maximale d'un objet de n'importe quel type.

En cas de succès, la fonction retourne la première adresse de la zone allouée. Cette adresse est retournée sous forme d'un pointeur de type `void *` (donc un pointeur non typé). En effet, la fonction ne peut pas savoir quel va être le type des données qui seront stockées. Le contenu de la zone n'est pas initialisé (les valeurs présentes dans la zone sont non définies).

En cas d'échec (par exemple s'il n'y a plus de mémoire disponible), la fonction retournera la valeur `NULL`. Il est important à chaque usage de la fonction `malloc` de vérifier que la valeur de retour n'est pas `NULL`. En effet, en cas d'oubli, le déréférencement d'un pointeur dont la valeur est `NULL` entraîne un comportement indéterminé (le plus souvent arrêt brutal du programme avec une erreur de segmentation).

8.4.1.1. Exemple d'utilisation

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

int main(int argc, char *argv[]) {
   int * p = malloc(sizeof(int)); // Ligne 1
   if (p == NULL) {
      perror("malloc");
      return EXIT_FAILURE;
   }
   *p = 42;
   printf("Valeur : %d\n", *p); // Affiche Valeur : 42
   return EXIT_SUCCESS;
}
```

Ligne 1, le programme, grâce à la fonction `malloc`, réserve une zone mémoire dont la taille est celle du type `int`, donc assez de mémoire pour pouvoir y stocker un `int`. L'adresse de la

zone allouée est stockée dans la variable locale `p` de type pointeur vers un `int`. On notera que la conversion de `void*` vers `int*` est ici implicite.

Le programme teste ensuite si la fonction `malloc` a eu un problème en comparant la valeur de retour avec `NULL`. Dans ce cas, un message d'erreur approprié est affiché à l'aide de la fonction `perror`, puis le programme se termine avec un code de retour indiquant une erreur.

Si tout s'est bien passé, le programme stocke la valeur `42` dans la zone précédemment allouée, puis affiche son contenu à l'aide de la fonction `printf`.

8.4.1.2. Fuites mémoire

La zone mémoire allouée par la fonction `malloc` est accessible pour tout le reste de l'exécution du programme ou jusqu'à sa libération explicite par un appel à la fonction `free` (voir plus loin).

Contrairement à d'autres langages (par exemple Java) qui dispose d'un ramasse-miette (*garbage collector*), la mémoire allouée dynamiquement en C n'est **jamais** libérée automatiquement (sauf à la toute fin de l'exécution du programme où le système d'exploitation libère toutes les ressources utilisées par le programme et en particulier la mémoire), même si, du fait de la logique du programme, il n'est plus possible d'y accéder (par exemple si l'adresse de la zone allouée est perdue).

La mémoire étant une ressource finie, il est de la responsabilité du programmeur de libérer la mémoire allouée lorsqu'elle n'est plus utilisée, surtout pour les programmes qui ont vocation à s'exécuter pendant une longue période (serveur par exemple) sinon il y a un risque qu'au fur et à mesure des allocations non libérées, la mémoire se remplisse.

Exemple :

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

void f() {
   int * p = malloc(sizeof(int)); // Ligne 1
   if (p == NULL) {
      perror("malloc");
      return EXIT_FAILURE;
   }
   *p = 42;
   printf("Valeur : %d\n", *p);
}

int main(int argc, char *argv[]) {
   f();

   // ...

   return EXIT_SUCCESS;
}
```

La fonction `f`, appelée par `main`, réalise une allocation mémoire, puis utilise la zone allouée. À la fin de la fonction `f`, l'adresse de cette zone (contenue dans le pointeur `p` qui est une

variable locale à `f`) est perdue. La mémoire allouée **n'est pas** libérée à ce moment là bien que dans la suite de l'exécution, il n'est plus possible d'accéder à la zone (car le programme a perdu son adresse). Il y a donc une fuite mémoire qui pourrait être problématique si la fonction `f` était appelée en boucle par exemple.

8.4.2. `free`

La fonction `free` a la signature suivante :

```
void free(void *p);
```

Elle reçoit en argument un pointeur. Ce pointeur doit être un pointeur préalablement renvoyé par la fonction `malloc` (ou une fonction équivalente) sinon le comportement est non défini. Cette fonction libère la zone mémoire correspondante. Une fois la zone libérée, une tentative d'accès à la zone entraîne un comportement non défini. De même appeler une deuxième fois la fonction `free` sur le même pointeur entraîne un comportement non défini.

Exemple :

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

int main(int argc, char *argv[]) {
   int * p = malloc(sizeof(int));
   if (p == NULL) {
      perror("malloc");
      return EXIT_FAILURE;
   }
   *p = 42;
   printf("Valeur : %d\n", *p);
   free(p);
   // À partir de ce moment, il n'est plus autorisé
   //  d'accéder à la zone mémoire pointée par p
   return EXIT_SUCCESS;
}
```

8.4.3. `realloc`

La fonction `realloc` a la signature suivante :

```
void *realloc(void *p, size_t size);
```

Cette fonction reçoit deux arguments :
- un pointeur `p` qui pointe vers une zone préalablement allouée par `malloc` (ou une fonction similaire) ;
- un entier `size` qui indique la nouvelle taille désirée.

La fonction `realloc` permet de changer la taille d'une zone mémoire allouée.

En cas de succès, elle renvoie l'adresse de la nouvelle zone (adresse qui peut être identique ou différente de la zone initiale). Les données de la zone initiale (entre le début et le minimum des deux tailles : initiale et nouvelle) sont conservées. Si la nouvelle taille est inférieure à la taille initiale, les données supplémentaires sont perdues. Si la nouvelle taille est supérieure à la taille initiale, les données sont conservées et le reste de la zone est non initialisé. Si la zone

a été déplacée (si `p` et la valeur de retour sont différents), il n'y a pas besoin de libérer la zone initiale, c'est fait par `realloc`.

En cas d'échec, la fonction `realloc` renvoie la valeur `NULL`.

Exemple :

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

int main(int argc, char *argv[]) {
   int *p = malloc(2 * sizeof(int));
   // Allocation d'une zone pouvant contenir 2 int
   if (p == NULL) {
      perror("malloc");
      return EXIT_FAILURE;
   }

   p[0] = 0; // Cette syntaxe sur un pointeur sera vu un peu plus loin
   p[1] = 1;

   int *np = realloc(p, 3 * sizeof(int));
   // On modifie la taille de la zone allouée qui peut
   //  maintenant stocker 3 int
   // On notera que np[0] vaut toujours 0 et np[1] vaut 1.

   if (np == NULL) {
      perror("realloc");
      return EXIT_FAILURE;
   }

   np[2] = 2;

   printf("%d %d %d\n", np[0], np[1], np[2]);
   // Affichage : 0 1 2

   free(np);

   return EXIT_SUCCESS;
}
```

8.5. Pointeurs (suite)

Cette partie aborde les liens entre tableaux et pointeurs en C.

8.5.1. Liens entre tableaux et pointeurs

Les pointeurs et les tableaux sont intimement liés en C.

Tout d'abord une expression de type tableau (comme par exemple le nom d'un tableau), est converti implicitement dans de très nombreux cas, en un pointeur vers son premier élément (c'est-à-dire un pointeur, dont le type cible est celui d'un élément du tableau, et donc la valeur est l'adresse du premier élément du tableau). Il existe néanmoins quelques exceptions, la plus courante étant lorsque le tableau est passé comme opérande à l'opérateur `sizeof`.

Exemple :

```c
int tab[4] = {0, 1, 2, 3};

int *ptr = tab;
// Ici, l'expression tab, utilisée à droite de l'affectation
//  est de type tableau et est donc implicitement convertie
//  en un pointeur vers le premier élément du tableau
// ptr contient donc l'adresse du premier élément
//  du tableau tab

*ptr = 42;
// On stocke la valeur 42 à l'adresse contenue dans ptr
//  C'est-à-dire dans la première case du tableau tab

printf("%d\n", tab[0]); // Affichera 42
```

Attention, il n'est pas possible de modifier l'adresse du tableau par cette méthode. Exemple :

```c
int tab[4] = {0, 1, 2, 3};

int *ptr = malloc(4 * sizeof(int));
// Alloue de l'espace pour stocker 4 entiers

tab = ptr; // Erreur, tab n'est pas modifiable !
```

8.5.2. Arithmétique des pointeurs

Certaines opérations (addition et soustraction) entre pointeurs et entiers sont possibles. Elles ont un comportement qui peut surprendre initialement.

Addition d'un pointeur et d'un entier : si `p` est un pointeur pointant vers l'élément d'index `i` d'un tableau, l'expression `p + n`, où `n` est de type entier (`int` par exemple), est de type pointeur et pointe vers l'élément `i + n` du tableau.

Exemple :

```c
int tab[4] = {0, 1, 2, 3};

int *ptr = tab; // ptr pointe vers le 1er élément du tableau (index 0)

int *ptr2 = ptr + 2; // ptr2 pointe vers le 3e élement du tableau (index 2 = 0
+ 2)
```

Soustraction d'un pointeur et d'un entier : si `p` est un pointeur pointant vers l'élément d'index `i` d'un tableau, l'expression `p - n`, où `n` est de type entier (`int` par exemple), est de type pointeur et pointe vers l'élément `i - n` du tableau. Le comportement n'est pas défini si `i - n` est négatif.

Exemple :

```c
int tab[4] = {0, 1, 2, 3};

int *ptr = &(tab[2]); // ptr pointe vers le 3e élément du tableau (index 2)

int *ptr2 = ptr - 1; // ptr2 pointe vers le 2e élement du tableau (index 1 = 2
- 1)
```

Attention, ces opérations travaillent sur les indices des tableaux. Or, la taille de chaque élément d'un tableau va varier en fonction du type de l'élément de base du tableau. Faire donc `+ 1` sur un pointeur n'incrémente pas nécessairement l'adresse du pointeur de 1, mais de la taille de l'élément pointé.

Exemple :

```
// On suppose dans cet exemple que sizeof(int) == 4
int tab_int[4] = {0, 1, 2, 3};
// On suppose que le tableau tab_int commence à l'adresse 1000

char tab_char[4] = {'a', 'b', 'c', 'd'};
// On suppose que le tableau tab_char commence à l'adresse 2000

int *ptr_int = tab_int; // Pointe vers la 1re case de tab_int
// Donc ptr_int contient l'adresse 1000

int *ptr_int2 = ptr_int + 1; // Pointe vers la 2e case de tab_int
// Donc ptr_int2 contient l'adresse 1002

char *ptr_char = tab_char; // Pointe vers la 1re case de tab_char
// Donc ptr_char contient l'adresse 2000

int *ptr_char2 = ptr_char + 1; // Pointe vers la 2e case de tab_char
// Donc ptr_char2 contient l'adresse 2001
```

On peut donc retenir que, si `p` est un pointeur vers un type `t` et de valeur entière p et `n` un entier de valeur n, écrire `p + n`, revient à faire l'opération *arithmétique* $p + n \times$ `sizeof(t)`.

Incrémentation/Décrémentation : si `p` est un pointeur et `n` un les opérations suivantes sont également définies :

- `p++` et `++p` équivaut à `p = p + 1`
- `p--` et `--p` équivaut à `p = p - 1`
- `p += n` équivaut à `p = p + n`
- `p -= n` équivaut à `p = p - n`

8.5.3. Accès indexé à un tableau et à un pointeur

Enfin, le dernier lien entre tableau et pointeur vient du comportement de l'opérateur d'accès indexé `[]`.

En effet, si `p` est une expression de type pointeur et `n` est de type entier, `p[n]` équivaut à `*((p) + n)`. Ici, le `+` est l'opération d'addition d'un pointeur et d'un entier vu précédemment.

De même, si `e` est une expression de type tableau, dans l'expression `e[n]`, `e` est convertie en un pointeur vers le premier élément du tableau puis le comportement précédent est appliqué.

Exemple :

```
int tab[4] = {0, 1, 2, 3};

int *ptr = tab; // ptr pointe vers le premier élément du tableau tab

// Dans le suite, tab[2] et ptr[2] réfèrent tous les deux au troisième élément
```

```c
//  du tableau tab

printf("tab[2] = %d, ptr[2] = %d\n", tab[2], ptr[2]);
// Affichera tab[2] = 2, ptr[2] = 2

tab[2] = 17;

printf("tab[2] = %d, ptr[2] = %d\n", tab[2], ptr[2]);
// Affichera tab[2] = 17, ptr[2] = 17

ptr[2] = 42;

printf("tab[2] = %d, ptr[2] = %d\n", tab[2], ptr[2]);
// Affichera tab[2] = 42, ptr[2] = 42
```

8.6. Chaînes de caractères

8.6.1. Représentation

Contrairement à certains autres langages (comme Java, Rust, etc.), il n'y a pas de type spécifique en C pour représenter une chaîne de caractères (c'est-à-dire une succession de caractères).

En C, une chaîne de caractères est simplement une zone mémoire où les caractères vont être stockés les uns à la suite des autres. La fin d'une chaîne de caractères est identifié par un caractère particulier : le caractère *nul*, dont la valeur entière est 0, que l'on peut également représenter en C par `'\0'`.

Une chaîne est donc manipulée soit via un tableau de caractères (les éléments d'un tableaux, donc ici les caractères de la chaîne, sont toujours consécutifs en mémoire), soit, le plus souvent, via un pointeur vers le premier caractère (les autres caractères sont aux adresses suivantes et la fin est repérée par le caractère nul).

8.6.2. Chaînes littérales

Les chaînes de caractères littérales sont une succession de zéro ou plus caractères entre double quotes, exemple : `"Hello world!"`. Le type d'une telle chaîne est un tableau de caractères dont la taille est le nombre de caractères de la chaîne plus un (le caractère nul qui est automatiquement ajouté). Comme vu précédemment, une telle expression se convertit implicitement en un pointeur vers son premier élément (c'est-à-dire un pointeur vers un caractère) dans la plupart des cas. *Attention*, les caractères d'une telle chaîne littérale ne sont pas modifiables.

Exemples d'utilisation d'une chaîne littérale :

```c
void f() {
  char t[] = "abc";
  // "abc" est de type tableau de 4 char ['a', 'b', 'c', '\0']
  // La taille du tableau t n'est pas spécifiée donc elle est déduite
  //  de l'initialisation
  // Donc t est un tableau de 4 char (donc 4 octets)
  // t est ici une variable locale donc les 4 octets sont alloués sur la pile
  // Les 4 valeurs de t sont initialisés par copie des 4 valeurs de "abc"
  // Donc "abc" n'est pas modifiable mais le tableau t l'est
```

```
  t[1] = 'g';

  printf("%s\n", t); // Affiche : agc
}
void g() {
  char *p = "abc";
  // "abc" est de type tableau de 4 char ['a', 'b', 'c', '\0']
  // p est une variable locale de type pointeur
  //  (donc occupera 8 octets sur une machine classique)
  // le tableau "abc" se convertit implicitement vers un pointeur
  //  vers son premier élément (i.e. premier caractère)
  // donc p est initialisé avec l'adresse du premier caractère du tableau "abc"
  // or le contenu de ce tableau n'est pas modifiable

  p[1] = 'g'; // équivalent à *(p + 1) = 'g';
  // Comportement non spécifié
  // Le programme compilera très probablement
  //  mais plantera certainement à l'exécution
}
```

8.6.3. Exemple de manipulation de chaînes de caractères

Comme exemple de manipulation des chaînes de caractères en C, nous allons écrire le code d'une fonction calculant le nombre de caractères d'une chaînes de caractères (sans compter le caractère nul).

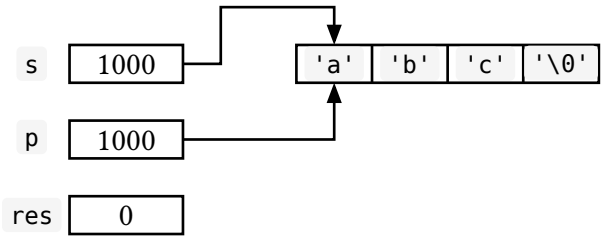
Voici le code de la fonction, qui sera détaillé dans la suite.

```
int len(char * s) {
  char * p = s;
  int res = 0;

  while (*p != '\0') {
    res++;
    p = p + 1;
  }

  return res;
}
```

Cette fonction doit donc recevoir en argument une chaîne de caractères quelconque. Comme le passage d'un tableau de taille inconnu à une fonction est délicat (et hors programme), la fonction va donc recevoir la chaîne via un pointeur vers son premier caractère, c'est-à-dire un `char *`.

Pour les explications suivantes, on suppose que la fonction est appelée par ailleurs de la façon suivante : `len("abc")`. On supposera également que la chaîne `"abc"` est stockée à l'adresse 1000 en mémoire.

À l'entrée de la fonction, on a la situation suivante :

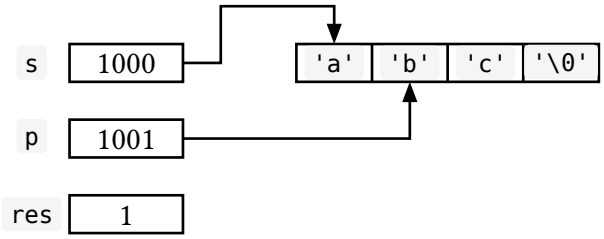


Le pointeur `p` va permettre de parcourir successivement les différents caractères de la chaîne. Il est initialisé lors de sa définition avec la valeur de `s` donc pointe initialement vers le premier caractère de la chaîne.

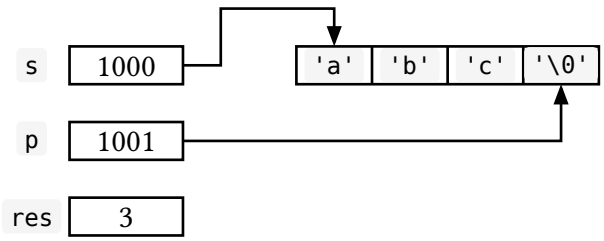
Il faut ensuite vérifier si le caractère courant (celui pointé par `p`) est le dernier de la chaîne. Ce test se fait en comparant le caractère pointé par `p` (obtenu en déréférençant le pointeur : `*p`) et le caractère nul qui indique la fin de la chaîne : `'\0'`.

Si la caractère pointé par `p` n'est pas le dernier, on incrémente le nombre de caractères (`res++`), puis on passe au caractère suivant en incrémentant de 1 le pointeur `p` : `p = p + 1` (voir l'arithmétique des pointeurs).

À la fin de la première itération de la boucle `while`, on obtient donc la situation suivante :



Arrivé au dernier caractère, on sort de la boucle dans la situation suivante :



Note : cette fonction étant utile, elle existe déjà dans la bibliothèque standard et s'appelle `strlen`.

9. La fonction `main`

9.1. Introduction

La fonction `main` est automatiquement appelée au démarrage du programme.

Elle a deux signatures possibles :

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

9.2. Arguments

Lorsque la deuxième forme de la signature est utilisée (celle avec `argc` et `argv`), les valeurs de ces arguments sont passées au programme par l'environnement d'exécution (s'il existe).

`argc` est un entier positif ou nul qui indique le nombre d'arguments passés par l'environnement d'exécution au programme.

`argv` est un pointeur vers le premier élément d'un tableau de `argc` + 1 pointeurs. Le dernier élément du tableau est toujours à la valeur `NULL`. Les autres éléments du tableau, s'ils existent, pointent vers des chaînes de caractères qui représentent la valeur des arguments passés par l'environnement. S'il est n'est pas nul, `argv[0]` représente le nom du programme.

En l'occurrence, dans le cadre de ce cours, les arguments qui sont récupérés via `argv` sont ceux passés lors du lancement du programme sur la ligne de commande.

Exemple :

```
// Fichier args.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  printf("argc = %d\n", argc);
  for (int i = 0; i < argc; i++)
    printf("argv[%d] = \"%s\"\n", i, argv[i]);
  return EXIT_SUCCESS;
}
```

On compile ce programme :

```
$ gcc -Wall -Wextra -std=c11 -o args args.c
```

Si on exécute le programme sans arguments, on obtient :

```
$ ./args
argc = 1
argv[0] = "./args"
```

On peut également passer plusieurs arguments au programme :

```
$ ./args test 42 toto
argc = 4
argv[0] = "./args"
argv[1] = "test"
```

```
argv[2] = "42"
argv[3] = "toto"
```

Note : attention, les arguments sont des chaînes de caractères, y compris si vous avez passé des entiers (42 dans l'exemple précédent).

9.3. Valeur de retour

Normalement, la fonction `main` doit renvoyer un entier. Cet entier est passé implicitement à l'appel système `exit`. Il indique au système d'exploitation si le programme s'est correctement exécuté ou non :

- 0 (ou la macro `EXIT_SUCCESS` définie dans `stdlib.h`) indique une terminaison normale
- une valeur différente de 0 (ou la macro `EXIT_FAILURE`) indique un problème.

10. La bibliothèque standard

10.1. Introduction

La bibliothèque standard est un ensemble de fonctions standardisées disponibles pour tous les programmes C (à quelques exceptions près : noyau d'un système d'exploitation, embarqué *bare-metal*...).

L'objectif de cette section est de présenter quelques unes des principales fonctions de cette bibliothèque (d'autres fonctions ont déjà été présentées précédemment comme les fonctions permettant de gérer la mémoire : `malloc`, `free`...).

10.1.1. Fichiers d'en-tête

Pour rappel, lorsque l'on appelle une fonction, le compilateur doit avoir vu sa déclaration (ou sa définition) au préalable. C'est également le cas pour les fonctions de la bibliothèque standard.

Les déclarations des fonctions de cette bibliothèque sont regroupées dans un ensemble standardisé de fichiers d'en-tête (fichiers portant l'extension `.h`). Donc si vous souhaitez utiliser une fonction de la bibliothèque standard, il faut inclure, à l'aide d'une directive `include` du préprocesseur, le fichier d'en-tête correspondant.

Par exemple, la fonction `memcpy` est déclarée dans le fichier `string.h`. Donc si vous voulez l'utiliser, il faudra ajouter `#include <string.h>` en début de votre fichier source.

10.1.2. Gestion des erreurs

La plupart des fonctions de la bibliothèque standard retourne un type (un entier, un pointeur, etc.) et utilise une valeur particulière de ce type (négative, −1, `NULL`...) pour signifier qu'une erreur est survenue. Néanmoins, cette valeur n'indique pas la cause exacte de l'erreur.

Pour cela, en cas d'erreur, les fonctions de la bibliothèque standard stockent dans une variable globale nommée `errno`, de type entier, une valeur indiquant la nature exacte de l'erreur (par exemple 1 signifie une erreur de permission, 2 un fichier ou un répertoire inexistant, etc.). Cette variable est déclarée dans le fichier d'en-tête `errno.h`.

La fonction `perror` permet de traduire la valeur stockée dans `errno` en un message compréhensible par l'utilisateur. Cette fonction prend en unique argument une chaîne de caractères. Elle affiche alors (plus précisément dans `stderr`) la chaîne passée en argument, suivie d'un caractère `:`, suivi du message d'erreur correspondant à la valeur actuelle de `errno`.

Exemple d'utilisation :

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

void f() {
   int res = open("fichier", O_RDONLY);
   if (res == -1) {       // open renvoie -1 en cas d'erreur
      perror("open");     // affiche la cause de l'erreur
      exit(EXIT_FAILURE); // arrête le programme
```

```
    }
    // ...
}
```

Si le fichier `fichier` n'existe pas, l'exécution de la fonction `f` produira le message d'erreur suivant à l'exécution :

```
open: No such file or directory
```

10.1.3. Aide sur les fonctions : `man`

Sous Unix (et Linux en particulier), si vous souhaitez tout savoir sur une fonction de la bibliothèque standard (signature, fichier d'en-tête, signification de la valeur de retour, etc.), vous pouvez utiliser l'outil `man` (manuel).

Par exemple, dans un terminal, pour avoir la page de manuel de la fonction `memcpy`, il suffit de taper :

```
$ man memcpy
```

Parfois, la fonction cherchée se trouve dans plusieurs sections différentes (c'est le cas par exemple de `printf`). Si la page affichée ne correspond visiblement pas à une fonction C, il faut parfois indiquer ajouter la section voulue. Deux sections sont intéressantes :
- la section 3 qui concerne les bibliothèques (et la bibliothèque C en particulier)
- la section 2 qui concerne les appels systèmes (par exemple `read`, `write`, `open`)

Par exemple pour avoir la bonne page de manuel pour `printf` (dans la section 3), il suffit de taper :

```
$ man 3 printf
```

10.2. Entrées / Sorties

10.2.1. `printf` / `fprintf`

Ces deux fonctions sont déclarées dans `stdio.h`. Leurs signatures sont les suivantes :

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

Les `...` signalent que ces fonctions prennent un nombre variable d'arguments (nous n'étudieront pas cette possibilité dans ce cours).

Ces deux fonctions construisent une chaîne de caractères, à partir des indications fournies par la chaîne `format`, et écrivent cette chaîne :
- dans la sortie standard (le plus souvent à l'écran) dans le cas de `printf`
- dans le flux de sortie (le plus souvent un fichier) représenté par `stream` dans le cas de `fprintf`

La chaîne de caractères `format` indique donc à ces fonctions quoi faire :
- les caractères autres que `%` sont recopiés tels quels
- un caractère `%` introduit une instruction de formatage

Chaque instruction de formatage va indiquer comment formater et afficher un argument successif de `printf` ou `fprintf` (la première instruction de formatage introduite par le

premier `%` va traiter le premier argument après `format`, la deuxième instruction de formatage introduite par le deuxième `%` va traiter le deuxième argument après `format` ...).

Les instructions de formatage sont composés :

- du caractère `%`
- optionnellement d'un ou plusieurs drapeaux
  - `-` : le résultat est aligné à gauche (au lieu d'à droite par défaut)
  - `+` : le signe d'une valeur numérique est toujours affiché (par défaut il ne l'est que si la valeur est négative)
  - ` ` (une espace) : si le signe n'est pas présent (négatif), une espace est affichée
  - `0` : si besoin, des `0` sont ajoutés à gauche au lieu d'espaces
- optionnellement, une constante numérique qui indique la taille minimale du champ
- optionnellement, un `.` suivi d'une constante numérique qui indique la précision
- optionnellement, un ou plusieurs caractères modifiant la taille de l'argument
- un caractère indiquant la conversion à effectuer

Les conversions les plus utilisées sont les suivantes.

Conversion	Explications
`%`	Permet d'afficher le caractère `%`
`c`	Affiche un caractère. L'argument est considéré comme un `unsigned char` et le caractère correspondant à la valeur est affiché.
`s`	Affiche une chaîne de caractères. L'argument est considéré comme un pointeur vers un caractère. La précision (optionnelle) indique le nombre maximum de caractères qui seront affichés. Si elle est absente, tous les caractères sont affichés, jusqu'au caractère de fin de chaîne exclu.
`d` ou `i`	Affiche un entier signé en base 10. L'argument est considéré comme un `int`. La précision (optionnelle) indique le nombre minimum de chiffres à afficher. Les modificateurs de taille suivants peuvent être utilisés : • `hh` si l'argument est un `signed char` • `h` si l'argument est un `short` • `l` si l'argument est un `long` • `ll` si l'argument est un `long long`
`x` ou `X`	Affiche un entier non signé en base 16. L'argument est considéré comme un `unsigned int`. La précision (optionnelle) indique le nombre minimum de chiffres à afficher. Les modificateurs de taille suivants peuvent être utilisés : • `hh` si l'argument est un `unsigned char` • `h` si l'argument est un `unsigned short` • `l` si l'argument est un `unsigned long` • `ll` si l'argument est un `unsigned long long`
`u`	Affiche un entier non signé en base 10. L'argument est considéré comme un `unsigned int`. La précision (optionnelle) indique le nombre mini-

Conversion	Explications
	mum de chiffres à afficher. Les modificateurs de taille suivants peuvent être utilisés : • `hh` si l'argument est un `unsigned char` • `h` si l'argument est un `unsigned short` • `l` si l'argument est un `unsigned long` • `ll` si l'argument est un `unsigned long long`
`f`	Affiche un nombre flottant sous la forme ddd.ddd. L'argument est considéré comme un `double`. La précision (optionnelle) indique le nombre de chiffre à afficher après la virgule.
`e`	Affiche un nombre flottant sous la forme d.ddd e dd. L'argument est considéré comme un `double`. La précision (optionnelle) indique le nombre de chiffre à afficher après la virgule.
`p`	Affiche l'adresse contenue dans un pointeur. L'argument est considéré comme un `void*`.

Premier exemple d'utilisation :

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

int main(int argc, char *argv[])
{
  int i = 42;
  int j = 25;
  printf("i = %d, j = %d\n", i, j);
  return EXIT_SUCCESS;
}
```

Une fois compilé, l'exécution de ce code affiche :

```
i = 42, j = 25
```

Explications : la chaîne de formatage est `"i = %d, j = %d\n"`. Elle est analysée caractère par caractère. Tant que le caractère `%` n'est pas rencontré, les caractères sont affichés tels quels. La fonction commence donc par afficher `i =` (y compris l'espace).

Il y a ensuite `%d`. Il s'agit de la première instruction de formatage. Elle indique qu'il faut afficher le premier argument après la chaîne de formatage dans l'appel à `printf` (c'est-à-dire le deuxième argument, ici `i`). `d` indique que l'argument est à traiter comme un `int` (c'est bien le cas), et qu'il faut afficher sa valeur en base 10. La fonction affiche donc `42`.

Elle poursuit ensuite en affichant `, j =` (y compris l'espace).

Il y a ensuite `%d`. Il s'agit de la deuxième instruction de formatage. Elle indique qu'il faut afficher le deuxième argument après la chaîne de formatage dans l'appel à `printf` (c'est-à-dire le troisième argument, ici `j`). `d` indique que l'argument est à traiter comme un `int` (c'est bien le cas), et qu'il faut afficher sa valeur en base 10. La fonction affiche donc `25`.

Elle termine ensuite en affichant `\n`, c'est-à-dire un retour à la ligne.

Autre exemple (précision et consignes d'alignement pour les entiers) :

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

int main(int argc, char *argv[])
{
  int i = 42;
  printf("|%4d|, |% d|, |%+4d|, |%04d|, |%-4d|\n", i, i, i, i, i);
  return EXIT_SUCCESS;
}
```

Cet exemple affiche :

```
|  42|, | 42|, | +42|, |0042|, |42  |
```

Autre exemple (caractères) :

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

int main(int argc, char *argv[])
{
  unsigned char c = 65;
  printf("%c, %hhu\n", c, c);
  return EXIT_SUCCESS;
}
```

Cet exemple affiche :

```
A, 65
```

10.2.2. `scanf` / `fscanf` / `sscanf`

Ces trois fonctions sont déclarées dans `stdio.h`. Leurs signatures sont les suivantes :

```
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *buffer, const char *format, ...);
```

Ces fonctions lisent des données depuis différentes sources, les interprètent selon les indications de la chaîne de caractères `format` et stockent les résultats dans les paramètres supplémentaires passées en arguments (`...`).

Les données sont lues :

- pour `scanf` depuis l'entrée standard (sauf redirection, le clavier dans le cas où le programme est lancé depuis un terminal)
- pour `fscanf` depuis le flux `stream`
- pour `sscanf` depuis la chaîne de caractères `buffer`

La chaîne de caractères `format` se comporte de façon similaire à `printf` et indique comment interpréter les données lues depuis la source.

- Un caractère différent d'un espace (au sens large: espace, tabulation, etc.) ou du caractère `%` : le prochain caractère de la source doit correspondre à ce caractère. Si c'est le cas, le

caractère de la source est « consommé » (on passe au caractère suivant de la source). Dans le cas contraire, la fonction retourne une erreur.
- Un caractère espace (au sens large : espace, tabulation, retour à la ligne, etc.) : si le caractère suivant de la source est un espace, il est consommé ainsi que tous les autres caractères espaces immédiatement à la suite.
- Un caractère `%` introduit une instruction de conversion.

Les instructions de conversion indiquent à `scanf` de lire un ou plusieurs caractères depuis la source, de les convertir (par exemple en entier) et de stocker le résultat dans un des arguments passés à la fonction.

Les instructions de conversion sont composés :
- du caractère `%`
- optionnellement, le caractère `*` indique de faire la conversion (et donc de consommer les caractères en entrée), mais sans stocker le résultat
- optionnellement, une constante numérique strictement positive indiquant le nombre maximum de caractères qui seront consommés depuis la source pendant la conversion
- optionnellement, un ou plusieurs caractères modifiant la taille par défaut de l'argument
- un caractère indiquant la conversion à effectuer

Les conversions les plus utilisées sont les suivantes.

Conversion	Explications
`%`	Permet de consommer le caractère `%`
`c`	Consomme un (par défaut) ou plusieurs caractères (si une indication de nombre de caractères est fournie). L'argument recevant la valeur doit être de type `char *`. Attention, si plusieurs caractères sont capturés, le caractère de fin de chaîne `'\0'` n'est pas ajouté automatiquement
`s`	Consomme une séquence de caractères sans espace. Si une taille est indiquée, elle consomme au plus le nombre indiqué de caractères ou jusqu'au premier caractère espace. L'argument doit être de type `char *`. Un caractère de fin de chaîne `'\0'` est toujours ajouté, donc l'argument doit pouvoir recevoir au plus le nombre de caractères attendus + 1
`d`	Lit un entier représenté en base 10. L'argument recevant la valeur doit être de type `int *`. Les modificateurs de taille suivants peuvent être utilisés : • `hh` l'argument est alors de type `signed char *` • `h` l'argument est alors de type `short *` • `l` l'argument est alors de type `long *` • `ll` l'argument est alors de type `long long *`
`x` ou `X`	Lit un entier non signé en base 16. L'argument recevant la valeur doit être de type `unsigned int *`. Les modificateurs de taille suivants peuvent être utilisés : • `hh` l'argument est alors de type `unsigned char *`

Conversion	Explications
	• `h` l'argument est alors de type `unsigned short *` • `l` l'argument est alors de type `unsigned long *` • `ll` l'argument est alors de type `unsigned long long *`
`u`	Lit un entier non signé en base 10. L'argument recevant la valeur doit être de type `unsigned int *`. Les modificateurs de taille suivants peuvent être utilisés : • `hh` l'argument est alors de type `unsigned char *` • `h` l'argument est alors de type `unsigned short *` • `l` l'argument est alors de type `unsigned long *` • `ll` l'argument est alors de type `unsigned long long *`
`f` ou `e`	Lit un nombre flottant. L'argument est considéré comme un `double *`.

Important : Les arguments recevant les valeurs sont tous de type pointeur. En effet, la fonction `scanf` doit pouvoir y stocker les valeurs lues (voir la section sur le passage par valeur et par adresse des arguments à des fonctions). Ces pointeurs doivent pointer sur des zones mémoires correctement allouées et de taille suffisante pour stocker les résultats (en particulier pour les conversions `c` et `s`).

Les fonctions retournent le nombre de conversions correctement réalisées (qui peut être inférieur au nombre de conversions demandées si une a échoué par exemple) ou la constante `EOF` si une erreur lié à la source (erreur d'entrée sur un fichier par exemple) a eu lieu avant la première conversion.

Exemple :

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

int main(int argc, char *argv[]) {
  char * input = "Abc 42    b, toto titi";
  int i;
  char c;
  char s[16];
  int res;

  res = sscanf(input, "Abc %d %c, %15s", &i, &c, s);

  printf("Valeur de retour de scanf = %d, i = %d, c = %c, s = \"%s\"\n", res, i,
c, s);

  return EXIT_SUCCESS;
}
```

Lors de l'exécution, ce programme affiche :

```
Valeur de retour de scanf = 3, i = 42, c = b, s = "toto"
```

10.2.3. Manipulation de fichiers

La bibliothèque standard fournit deux séries de fonctions permettant de manipuler des fichiers :

- Une série de fonctions bas-niveau (`open` , `close` , `read` , `write` ...)
- Une série de fonctions haut-niveau (`fopen` , `fclose` , `fprintf` , `fscanf`)

Dans le cadre de cette partie nous n'aborderons que cette dernière série de fonctions, la première (fonctions bas-niveau) sera abordée dans la troisième partie du cours (systèmes d'exploitation).

10.2.3.1. `fopen`

Pour pouvoir être manipulé, un fichier doit tout d'abord être ouvert. Cette opération est effectuée à l'aide de la fonction `fopen` déclarée dans `stdio.h` :

```
FILE* fopen(const char* filename, const char* mode);
```

Les arguments sont :

- `filename` est une chaîne de caractères contenant le nom du fichier
- `mode` est une chaîne de caractères indiquant à `fopen` comment le fichier doit être ouvert

Les valeurs possibles pour `mode` sont les suivantes :

Mode	Signification	Comportement si le fichier existe	Comportement si le fichier n'existe pas
`"r"`	Ouvre un fichier en lecture	La lecture commence au début du fichier	`fopen` retourne une erreur
`"w"`	Crée un fichier pour y écrire	Le contenu antérieur est détruit	Le fichier est créé vide
`"a"`	Ajout en fin de fichier	Le fichier est ouvert en écriture et la position courante est définie à la fin du fichier	Le fichier est créé vide
`"r+"`	Ouvre un fichier en lecture/écriture	La lecture commence au début du fichier	`fopen` retourne une erreur
`"w+"`	Crée un fichier pour y lire et écrire	Le contenu antérieur est détruit	Le fichier est créé vide
`"a+"`	Ouvre un fichier en lecture/écriture	La position courante est définie à la fin du fichier	Le fichier est créé vide

`fopen` renvoie un pointeur vers un type `FILE` . En cas d'erreur, ce pointeur vaut `NULL` et la variable `errno` est définie pour permettre de connaître la cause de l'erreur.

Le pointeur renvoyé par `fopen` identifie le fichier ouvert. Toutes les fonctions qui vont manipuler ce fichier prendront en argument ce pointeur.

10.2.3.2. `fclose`

Une fois les opérations terminées sur le fichier, il faut le fermer à l'aide de la fonction `fclose` déclarée dans `stdio.h` :

```
int fclose(FILE* stream);
```

10.2.3.3. Exemple

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

int main(int argc, char *argv[]) {
   FILE *f = fopen("output.txt", "w"); // Ouvre output.txt en écriture seule
   if (!f) { // Équivalent à f == NULL
      perror("fopen failed");
      return EXIT_FAILURE;
   }

   // Écriture dans le fichier
   fprintf(f, "Bonjour\n"); // Écrit "Bonjour" dans le fichier

   if (fclose(f)) { // Ferme le fichier, équivalent à fclose(f) != 0
      perror("fclose failed"); // En cas d'erreur
      return EXIT_FAILURE;
   }
   return EXIT_SUCCESS;
}
```

10.3. Manipulation de chaînes de caractères

Quelques fonctions utiles pour la manipulation de chaînes de caractères (ces fonctions sont déclarées dans `string.h`) :

- `int strncmp(const char* s1, const char* s2, size_t count)` : compare lexicographiquement au plus `count` caractères des deux chaînes (retourne une valeur négative, égale à zéro ou positive si `s1` est plus petite, égale ou plus grande que `s2`)
- `char *strncpy(char *dest, const char *src, size_t count)` : copie au plus `count` caractères de la chaîne `src` vers `dest`
- `char *strncat(char *dest, const char *src, size_t count)` : concatène au plus `count` caractères de la chaîne `src` vers `dest`
- `char *strndup(const char *src, size_t size)` : copie au plus `count` caractères de la chaîne `src` vers une nouvelle chaîne allouée automatiquement sur le tas
- `size_t strlen(const char* str)` : retourne la taille d'une chaîne de caractères (sans le caractère de fin)

Pour `strncpy` et `strncat` il est de la responsabilité de l'appelant de fournir une zone de destination (`dest`) correctement allouée et de taille suffisante pour recevoir la chaîne produite.

10.4. Manipulation de zones mémoires

Quelques fonctions utiles pour manipuler des zones mémoires (ces fonctions sont déclarées dans `string.h`) :

- `int memcmp(const void* lhs, const void* rhs, size_t count)` : compare lexicographiquement le contenu de deux zones mémoires de taille `count` octets (retourne 0 si elles sont égales)
- `void *memcpy(void *dest, const void *src, size_t count)` : copie `count` octets depuis `src` vers `dest`
- `void *memset(void *dest, int c, size_t count)` : remplit les `count` premiers octets de la zone pointée par `dest` avec la valeur de l'octet de poids faible de `c`