Le langage C

Programmer en C, par Mathéo Allart

Table des matières

Préface

Bienvenue ! Ce cours document détaille comment programmer en utilisant le langage haut niveau C.
En effet, C n'est pas fondamentalement un langage bas niveau. C'est un langage de programmation qui offre une abstraction suffisante du fonctionnement de l'ordinateur pour être considéré comme langage de haut niveau. Pas autant, bien sûr, que des langages tels que Java ou Python, qui ont une abstraction totale de la gestion des composants de l'ordinateur, mais qui reste néanmoins suffisamment haut pour être considéré comme facile à apprendre par des programmeurs débutants.

Certaines notions sont nécessaires pour commencer la programmation, certaines supplémentaires pour le C. L'avantage de ce cours est qu'il ne se penche sur l'historique du langage que lorsque nécessaire, afin de se concentrer sur une mise en pratique rapide et compréhensible desdites notions.

Aucun prérequis n'est nécessaire afin de comprendre et mettre en pratique le contenu de ce document.

Notions prérequises

Certaines notions et sens logiques sont nécessaires pour comprendre la base de la programmation. Certains termes techniques vont être employés plusieurs fois, et il est ainsi nécessaire de les définir. Ce sont les raisons d'être de cette section.
Cette section est très complète et peut être un peu indigeste. Par conséquent, il est conseillé de la sauter en premier lieu et d'y revenir lorsqu'un terme paraît inconnu durant le reste de ce document.

Notions

Un script se lit, et s'exécute de haut en bas et de gauche vers la droite. Afin de rendre son code réutilisable, il est nécessaire de le commenter de manière claire, et de donner des noms **explicites** aux variables et fonctions que l'on crée.

Dans l'optique de rendre son code le plus portable possible (lui permettre de compiler et de tourner sur la majorité des plate-formes disponibles), il est préférable d'utiliser le plus possible des solutions et bibliothèques standards. En C, elles sont généralement précédées du préfixe "std": <stdio.h>, <stdlib.h>, <stdint.h>, <stdbool.h>...

Rendre son code modulaire est aussi une approche préférable. Cela signifie qu'au lieu de "hardcoder" (écrire de manière explicite) un maximum de valeurs dans le code, il est mieux de

le remplacer par des constantes, découper chaque fonctionnalités du programme en sous-programmes (fonctions) distincts et compartimenter chaque unités de fonctions du code en fichiers. Ainsi, si une partie du code peut être utilisée dans un futur programme, ou afin d'étendre les fonctionnalités du programme existant dans le futur, cela devient possible.

Définitions

Premièrement: en programmation, une **instruction** est un ordre donné à l'ordinateur. Tout fonctionne uniquement comme ça, à base de ce que l'on appelle des **interruptions**. Une **interruption**, c'est l'ordre donné à un processeur de pauser sa tâche courante pour en effectuer une autre, avant de reprendre son cycle initial.
Une **interruption** peut par exemple être d'appeler une suite d'autres instructions contenues en mémoire ou en registre.

Chaque **opération**, chaque fonction appelée correspond à une instruction.

Un **script** est une suite d'instructions faisant partie de la même **unité de compilation**. Une **unité de compilation** est une suite d'instructions et de valeurs encodées directement dans le programme, de manière indépendante des autres unités de compilation.

Une **condition**, c'est une instruction qui renvoie une valeur **booléenne**.
Une valeur **booléenne** est une valeur qui peut seulement être de valeur 0 ou 1, correspondant respectivement à faux ou vrai.

Un bloc de code, souvent délimité entre accolades ({ }), correspond à une unité de compilation. C'est à dire que c'est une section qui va être traduite en ordres compréhensibles par l'ordinateur en tant que telle, comme une fonction indépendante. Un bloc de code peut en contenir d'autres.

Une structure de contrôle, c'est l'usage d'une condition pour choisir si on exécute un bloc de code ou non. **Les boucles** sont des structures de contrôle qui permettent de répéter l'exécution d'un bloc de code ou d'une instruction tant que la condition fournie renvoie vrai (1).

Une fonction est un bloc de code exécutable et réutilisable qui prend des **paramètres** en entrée. Un **paramètre**, c'est une valeur que l'on passe à une fonction afin qu'elle devienne accessible dans son **scope**. Le **scope**, c'est l'ensemble des **variables** et **valeurs** accessibles depuis un bloc de code.

Une variable est le nom qu'on associe à une valeur. La valeur est le contenu de la variable: on peut la changer, opérer dessus, la récupérer.

Un pointeur est une variable ayant pour valeur l'adresse en mémoire d'une autre variable. **Un pointeur** peut aussi pointer (avoir pour valeur l'adresse de) d'une fonction.

Une pratique **standard** est une pratique considérée comme faisant partie intégrante du langage et standard défini à l'international C, et est censé ainsi être supporté par l'ensemble des compilateurs et des plate-formes supportant le langage C.

Les **directives préprocesseur** sont des instructions qui sont compilée avant le reste du programme. Ainsi, cela permet de modifier le contenu du programme ou la manière dont il va être compilé. Par exemple, la directive #define permet de remplacer toutes les occurrences d'un lexème (token, suite de caractères) par un autre.

Un **programme** est un ensemble de scripts.

Un **header** ou un fichier **d'en-tête** est un fichier écrit dans le même langage de programmation servant à apporter une description de ce qui se trouve dans le script correspondant. Cela est très utile quand des fonctions en appellent d'autres, cela permet de les affranchir de l'ordre de compilation. En effet, une fonction *a* déclarée avant une fonction *b* ne pourra appeler *b* que si cette dernière est aussi déclarée avant *a*. C'est pour cela qu'on fait des **forward declarations** dans les fichiers d'en-tête.

Une **forward declaration** ou une déclaration anticipée est une technique de programmation qui consiste à déclarer l'entête d'un sous programme ou d'une structure sans en déclarer le corps (bloc de code et instructions). Ainsi, elle devient appelable même si elle n'est pas encore compilée.

La **compilation** est un processus complexe qui consiste à transformer un programme écrit dans un langage de programmation haut niveau en instructions binaires compréhensibles par l'ordinateur.

Une **instance** est la ou les valeurs assignée(s) à un type. Par exemple, je peux instancier 3 nombres entiers dans mon programme, les nommant respectivement n1, n2 et n3. Mes instances sont des instances du même type, mais ne partageront pas forcément la même valeur: car ce sont des instances différentes. Si la référence de deux variables et la même, alors elles pointent toutes les deux sur la même **instance**.

\n est un caractère spécial qui permet de demander un retour à la ligne.

Programmer en C

Le langage C est un langage fascinant. C'est un langage de haut niveau permettant aussi bien d'accéder à des fonctionnalités bas-niveau. à un tel point qu'il supporte de manière standard les instructions assembleurs au sein des scripts C.

Il est assez simple à prendre en main.

Un programme exécutable en C se compose d'un ou plusieurs scripts, l'un d'entre eux devant obligatoirement contenir un point d'entrée étant la fonction **main**.

une fonction **main** standard doit renvoyer un int et prendre en paramètres un entier pour le nombre d'arguments qu'on lui passe, et un vecteur de chaînes de caractères:

```
int main(int argc, char** argv)
{
	return 0;
}
```

le vecteur de chaînes de caractères est en fait un tableau itérable de taille *argc* qui correspond à chaque argument qu'on lui passe en console lors de son appel.
Ainsi, pour un script suivant:

```
#include <stdio.h>
int main(int argc, char** argv)
{
	for(int i = 0; i < argc; ++i)
		printf("arg %d: %s", i, argv[i]);
	return 0;
}
```

et l'appel depuis la console suivant:
> app.exe bonjour le monde !

On obtiendra l'output suivant;

arg 0: app.exe
arg 1: bonjour
arg 2: le
arg 3: monde
arg 4: !

Maintenant, analysons ce code ligne par ligne.

```c
// inclut la bibliothèque standard d'entrée / sortie de signal
#include <stdio.h>

// déclaration de l'entête de la fonction d'entrée du programme (main)
int main(int argc, char** argv)
// déclaration du corps de la fonction main
{
// boucle qui affiche chacun des paramètres avec lesquels le programme a
été
// appelé (nous reviendrons sur les boucles plus tard)
      for(int i = 0; i < argc; ++i)
// affichage des arguments
            printf("arg %d: %s", i, argv[i]);
      return 0;
}
```

À savoir que les commentaires (bouts du code qui ne sont pas traités par l'ordinateur) s'écrivent sur une ligne précédés de "//". Pour écrire un commentaire multi lignes, il suffit de l'écrire entre /* (commentaire) */ .

Fonctions

En C, on peut faire une déclaration de fonction soit anticipée, soit immédiate.
La différence réside dans le fait qu'une déclaration anticipée permet un appel à ladite fonction avant même la déclaration du corps de celle-ci, alors que dans le cas d'une déclaration immédiate, seules les fonctions compilées après celles-ci peuvent l'appeler. Par exemple :

```c
int A()
{
      return 1;
}

int B()
{
      return A();
}
```

Dans ce cas, cela fonctionne parfaitement, car A est compilé avant B, et est donc visible pour celui-ci.

Mais imaginons un cas où A devrait aussi pouvoir appeler B, par exemple si le paramètre qu'on

lui passe est égal à 0 (= nul). Dans ce cas, le code nécessite qu'on ajoute une déclaration anticipée de B:

```
int B();

int A(int x)
{
    if( x == 0 )
        return B();
    return 1;
}

int B()
{
    return A(1);
}
```

Récapitulons dans l'ordre: B est une fonction ne prenant aucun paramètre renvoyant un entier, A est une fonction qui renvoie un entier à la fin de son exécution, et si ce paramètre est nul renvoie le résultat de B, sinon 1. B, précédemment déclaré, renvoie le résultat de A prenant pour paramètre 1.

Le mot clé *return* sert à préciser que l'exécution de la fonction s'arrête ici, et que la valeur qu'elle retourne est celle précisée ci-après.

Variables

En C, les variables se déclarent en précisant d'abord leur type, puis leur nom, et enfin une éventuelle assignation de valeur.
par exemple:

```
int annee_courante;
annee_courante = 2024
float mon_compte_bancaire = 204.32;
```

à savoir qu'il n'est pas possible de faire de transtypage implicite sécurisé, c'est-à-dire changer de type en garantissant l'interprétation correcte des données.

Une constante est opposée à une variable, dans le sens ou c'est un mot lié à une valeur qui ne peut être modifiée. Une constante se déclare en rajoutant le mot clé "**const**" au début de la déclaration de variable standard.

Bulle: caster des types

Le *casting* consiste à expliciter au compilateur qu'il doit changer l'interprétation qu'il a de certaines données; sans changer les données en elles-mêmes. En C, toutes les données sont simplement des nombres, de différentes tailles en mémoire mais c'est la seule différence qu'ont tous les types primitifs dans ce langage.
Il est néanmoins possible de forcer l'ordinateur à interpréter certaines données autrement que ce pour quoi elles sont faites:

```c
#include <stdint.h>

int main(int argc, char** argv)
{
	int8_t negative_value = -1;
	printf("Ceci affiche 255, car on caste depuis un type signé vers un type non-signé: %d", (uint8_t)negative_value); // cela vaut -1 aussi
	return 0;
}
```

Ici, nous avons un nombre entier sur 8 bits encodé comme -1. En notation binaire, -1 correspond en fait à "**1111 1111**" (complément à 2). mais si on l'interprète comme un simple entier non signé (naturel, contraire de relatif), cette valeur devient simplement 255.

Structures de contrôle

Les structure de contrôles principales (dont les boucles) sont: *if*, *while*, *switch*, et *for*.
Nous avons brièvement vu la structure *if* dans les précédents exemples, mais il est temps de trouver de meilleures manières de programmer.

La structure *if*, comme vu précédemment, évalue une condition et exécute l'instruction ou le bloc de code la suivant si la condition donnée est vraie.

While

La boucle while agit à peu près de la même manière, mais en exécutant en boucle les instructions qu'on lui donne *tant que* la condition qu'on lui donne est vérifiée.
Par exemple, si on souhaite faire entrer un mot de passe à l'utilisateur mais qu'il fasse nécessairement plus de 8 caractères, on peut procéder comme suit grâce à la fonction scanf qui permet de mettre le contenu du flux de données entrant (l'input console) dans un buffer (nous reviendrons sur la notion de buffer plus tard):

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

int main(int argc, char** argv)
{
	char buffer[128];
	// on initialise la première valeur du buffer en tant que valeur nulle, afin de correctement lire la longueur de la chaîne avec strlen
	buffer[0] = '\0';
	// on vérifie la longueur de la chaîne actuelle contenue dans buffer
	while(strlen(buffer)) < 8)
	{
		printf("Veuillez entrer un nouveau mot de passe plus long:\n");
		// on met le mot de passe écrit dans le buffer, quitte à réécrire par dessus le contenu d'avant
		scanf("%s", buffer);
	}
	// On remercie l'utilisateur pour avoir compris la consigne; c'est important.
	printf("Merci !\n");
	return 0;
}
```

Une fois que le mot de passe rentré fera 8 caractères ou plus de long, la boucle cessera de s’exécuter et l’utilisateur pourra enfin arrêter d’entrer des mots de passe à n’en plus pouvoir.

For

Une boucle *for* est une extension d'une boucle *while*. C'est une structure de contrôle très pratique qui permet de garder un contrôle aisé sur ses variables ainsi que leurs scopes.
Une boucle *for* exécute répétitivement une action de la même manière que la boucle while, mais nous ajoute la possibilité d'instancier une variable en première itération tout en prenant le soin de garder celle-ci invisible pour tout ce qui sortirait du scope de la boucle.

Une itération, c'est le parcours d'instructions qu'effectue le programme à chaque lecture des instructions de ladite boucle.

Ce type de boucle permet aussi d'effectuer automatiquement une opération en fin de chaque itération. Par exemple, pour parcourir un tableau **tableau** d'une longueur fixe de 50 caractères, plutôt que de marquer 50 fois printf(...), nous allons utiliser l'approche suivante:

```
#include <stdio.h>

int main(int argc, char** argv)
{
	char tableau[50];
	for(int i = 0; i < 50; i++)
	{
		printf("caractère à l'index %d: %c\n", i, tableau[i]);
	}
	return 0;
}
```

Nous avons donc introduit pas mal de nouveaux éléments dans notre méthode de programmation: premièrement, nous utilisons l'index du tableau pour le parcourir, c'est à dire qu'en écrivant tableau[i], on écrit en fait: "le premier index du tableau + i". Mais nous y reviendrons quand nous parlerons des pointeurs.

Ensuite, un nouvel opérateur apparaît: **++**. **i++** signifie que nous souhaitons incrémenter i. C'est-à-dire que l'on souhaite assigner à i sa valeur actuelle + 1, tout simplement.

Comme on peut le voir, la structure d'une boucle for est donc:
(variable à déclarer; condition pour que la boucle s'exécute; action en fin d'itération)
On parcoura donc tous les index du tableau, partant de l'index 0 et finissant à l'index 49, affichant à chaque fois un nouveau caractère.

À savoir que chaque partie de la boucle for (initialisation initiale, condition d'exécution, action de fin d'itération) peut être omise. Si elle est non renseignée, la partie centrale (condition) est vraie par défaut. Il est quand même important de les séparer avec un point-virgule (for(;1;)).

Switch

Un switch est une série de if opérant sur une même variable de type intégral. C'est-à-dire que pour qu'un switch puisse fonctionner, il doit opérer sur un type représentable en nombre entier. Pour chacun des cas prévus, on peut prévoir un *case*, une série d'instructions à exécuter si la condition donnée est respectée:

```c
#include <stdio.h>

int main(int argc, char** argv)
{
	int a;
	scanf("%d", &a);
	switch(a)
	{
	case 0: return 0;
	case 1: printf("La valeur 0 était attendue\n"); break;
	default: return 1; // cas de défaut, si aucun autre cas du switch n'a été prouvé vrai
	}
}
```

Un nouveau mot-clé entre en jeu: break. C'est un mot clé qui sert à indiquer une sortie prématurée de boucle, et peut s'appliquer à toutes les structures de contrôle que l'on a vu.

Les types de données

Maintenant qu'on a survolé les concepts avec lesquels on va pouvoir composer l'architecture de nos programmes, il est temps de plonger dans les abysses.

Un type de donnée en informatique, c'est la manière dont on va agencer les bits afin de pouvoir représenter sans perte de l'information, que celle-ci soit abstraite ou non. Encore une fois, nous allons nous concentrer sur la pratique afin d'arriver au plus vite sur les points cruciaux de la programmation en C.

Les types primitifs

Les types primitifs sont des types qui sont directement représentables en tant que séquences de bits simples. Ils sont opposés aux types dits "'complexes", tels que les types abstraits de données, les unions, les flags… Que nous verrons ensuite.

Ils se déclinent selon le tableau qui suit:

Nom	Usage	Taille (en octets)	Aliases
char	stockage de caractères	1	int8_t, bool
short int	stockages de petits nombres ([−32767 ; +32767])	2	int16_t
int	stockage de nombres standards	4	int32_t
long int	stockage de grands nombres	4-8	int32_t, int64_t
long long int	stockage de très grands nombres	8	int64_t
float	nombre flottant (à virgule) $[3.4*10^{-38};3.4*10^{38}]$	4	
double	nombre flottant précis $[1.7*10^{-308};1.7*10^{308}]$	8	
long double	nombre flottant très précis $[3.4*10^{-4932};3.4*10^{4932}]$	10	

À savoir qu'il est possible de multiplier l'étendue des résultats interprétables pour chaque type en choisissant qu'il soit *non signé*: cela signifie que pour le même nombre de bits en mémoire, on va choisir d'interpréter le tout comme un ensemble numérique uniquement, en omettant le

fait que le dernier bit soit censé servir de bit de signe (+ / -). Ainsi, on peut compter deux fois plus loin, mais sans interpréter de signe pour notre valeur.

Les pointeurs

Il fallait bien en parler un moment ou un autre.
Les pointeurs, de leurs doux noms, impliquent de *pointer* sur quelque chose.
En vérité, un pointeur n'est rien d'autre qu'une variable ayant pour valeur l'adresse mémoire d'une autre variable. Si on étend le raisonnement, nous en avons déjà utilisé plusieurs fois, dont une de manière explicite; chaque fonction que l'on appelle est en fait la demande d'exécution d'une commande supposément stockée à l'adresse que l'on renseigne (celle de la fonction). On a utilisé un pointeur de manière plus explicite lors de l'utilisation de la fonction scanf(), où on a renseigné l'adresse de la variable *a* pour dire à scanf d'y écrire la valeur qu'elle récupérerait par l'entrée utilisateur.

```
int a; // on réserve l'espace mémoire pour un entier a
int* a_ptr = &a; // on réserve l'espace mémoire pour un pointeur sur un entier, que l'on appelle a_ptr et qui pointe sur l'adresse de a
*a_ptr = 4; // on utilise l'adresse stockée dans le pointeur 'a_ptr' pour y écrire la valeur 4
// affiche l'adresse de a et la valeur qu'on vient d'y écrire
printf("Valeur stockée en adresse mémoire %p: %d", a_ptr, *a_ptr);
```

À savoir que dans le cas ci-dessus, ces conditions sont vraies:

```
a == *a_ptr;
&a == a_ptr;
```

Car l'opérateur '*' est un opérateur spécial en C qui permet de dire que l'on souhaite récupérer la valeur d'un pointeur.

Et on peut déclarer un pointeur sur n'importe quel type, primitif ou non, tant qu'on le signale correctement en usant du signe '*' lors de sa déclaration !

Fun fact: il est possible de déclarer un pointeur sur pointeur sur entier et l'utiliser comme suit:

```
int a;
int* a_ptr = &a;
int** a_ptr_ptr = &a_ptr;
*(*a_ptr_ptr) = 5;
```

Il n'est pas explicitement possible de déclarer un "pointeur sur pointeur sur pointeur…", car cela ne fait en fait pas sens. Cela reste simplement un pointeur sur pointeur, même si le processus se répète des centaines de fois (à éviter tout de même…)

Note: un pointeur en soi fait toujours la même taille, et celle-ci est fixée par le système d'exploitation et dépend de la quantité de mémoire RAM disponible. Ainsi, il est possible de déclarer un pointeur sur caractère et lui assigner comme valeur l'adresse mémoire d'un int. Ce n'est pas une bonne pratique pour la lisibilité du code, mais c'est possible. C'est ainsi que les pointeurs sur void existent.

Les pointeurs sur void (void pointers, void ptr)

Notion très controversée, et pourtant si indispensable. les pointeurs sur void sont censés être des pointeurs sur un type indéfinis, ce qui veut dire un type non prévu à l'avance. On le déclare comme un pointeur sur tout autre type, mais on marque void à la place du nom de type habituel.

Les tableaux (arrays) et chaînes de caractères (string)

Saviez vous qu'un tableau, en programmation, n'est rien d'autre qu'une suite d'octets adjacents alloués préalablement à toute opération sur ceux-ci ?
C'est pour ça qu'un tableau a une taille fixe.
Et un tableau, en vérité, n'est que l'adresse du premier élément. Je m'explique;
Lorsqu'on déclare un tableau, on réserve un espace en mémoire (dont l'emplacement est choisi par le système d'exploitation) de plusieurs octets.
Dans l'exemple suivant:

```
char chaine[32];
```

on dit simplement au système d'exploitation: "Réserve moi 32 octets en mémoire pour en faire ce que je veux". Et lui, en sa bonté absolue, retourne l'adresse qu'il nous a trouvé correspondante dans la variable 'chaine', qui n'est en fait rien d'autre qu'un pointeur. Un pointeur sur le premier élément du tableau.
Si on fait un bond en arrière et qu'on se rappelle des casts, on sait qu'il est possible d'interpréter de multiples manières la même série de bits. Il en va de même pour les pointeurs: et c'est ce qu'on fait implicitement dès qu'on accède aux données d'un tableau.

```
chaine[0] = 'o';
*(chaine + 1) = 'k';
chaine[2] = '!';
*(chaine + 3) = 0;
printf("%s", chaine); // affiche: "ok!"
```

Ce sont juste deux manières d'accéder à un indice donné du tableau, qui résultent en exactement la même chose. Ce qui est important à comprendre ici, c'est que la mémoire se manipule en octets (chaque indice ajouté est de la taille du type sur lequel le pointeur pointe, dans ce cas un char) au minimum. Il est ainsi pareil d'écrire "**chaine[0]**" et "***chaine**".

Quant aux chaînes de caractères type C (celles que l'on utilise généralement en programmation bas-niveau), ce sont simplement des tableaux de caractères de longueur n + 1. On les appelle aussi chaînes terminées nullement (*null terminated string* ou *zero terminated string*). Au lieu de stocker dans un certain nombre d'octets la taille de la string et de l'écrire après, on écrit simplement la chaîne de caractères et on la termine par un caractère nul (de code ASCII 0, représentable en C comme '\0'). C'est ainsi que le standard C traite les chaînes de caractères, et la raison pour laquelle dans l'exemple précédent, la chaîne "ok!" et terminée par un caractère nul. Par défaut, ce caractère est ajouté à la fin de toute chaîne marquée comme telle:

```
char* chaine = "Je suis une chaîne de caractères !"; // '\0' implicite ajouté à la fin
```

Vous comprenez maintenant pourquoi le tableau d'arguments d'appels en console de la fonction main est de type "char**" ! C'est en fait un tableau de chaînes de caractères terminées nullement.

Les fonctions d'appel différé (callbacks)

Une fonction d'appel différé (qu'on appelera ici callbacks, de part le fait que ce soit le nom couramment employé de cette pratique et qu'il est bien plus court) sont des fonctions passées en paramètres d'autres fonctions. Oui, c'est totalement possible, et c'est généralement ainsi que l'on construit un gestionnaire d'évènements efficace (qui ne consiste pas à vérifier 90 conditions dans une boucle while(1)…).
Construire une callback est très aisé, et pour ce faire, il faut revenir quelques pages avant. On disait précédemment que les fonctions avaient en fait une adresse en mémoire. En réalité, lorsqu'on exécute un programme sur notre ordinateur, celui-ci se fait charger en mémoire par le système d'exploitation, dont toutes ses fonctions et variables. Pour déclarer une callback universelle (applicable à toute autre fonction en C), rien de plus simple:

```
void (*ma_fonction)(int son_parametre); // déclaration d'une fonction "ma_fonction", nécessitant un paramètre integer. Possibilité de ne rien mettre entre parenthèses pour spécifier qu'aucun paramètre n'est attendu
int (*mon_calcul)(short a, short b); // déclaration de fonction permettant un calcul entre a et b, renvoyant un int.
```

Nous avons beau être satisfaits de ceci pour le moment, la sortie console du compilateur va nous montrer des couleurs étranges lorsque l'on essaiera de se servir de ces fonctions. Et pour cause ! Nous n'avons que déclaré ces fonctions, aucune assignation de valeur n'a eu lieu. Si on essaie d'utiliser ces fonctions, elles pointeront dans le vide et l'exécution ne pourra aboutir. Mais comment assigner une valeur valide à ces variables ? Rien de plus simple. Il suffit de déclarer une fonction et d'y assigner l'adresse de cette dernière:

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

int main(int argc, char** argv)
{
	int (*calcul)(int premier_nombre, int second_nombre);
	calcul = &addition; // calcul = addition; aurait aussi été valide
	printf("1 + 2 = %d", calcul(1, 2));
}
```

Et ainsi, il est même possible de passer enfin en paramètre d'une fonction, une autre fonction:

```
void afficher_calcul(int (*f_calcul)(int a, int b), int na, int nb)
{
	printf("Le résultat de la fonction à l'adresse %p pour a = %d et b = %d est: %d", f_calcul, na, nb, f_calcul(na, nb));
}

int addition(int a, int b)
{
	return a + b;
}

int main(int argc, char** argv)
{
	afficher_calcul(&addition, 5, 3);
}
```

Les opérateurs

Maintenant que nous avons vu les types primitifs en C, il est temps de voir comment nous pouvons opérer dessus. C'est assez simple mais complet:

Les opérateurs arithmétiques

Ce sont les opérateurs affranchis des mathématiques binaires. Ces opérations ont le même comportement, quelle que soit la base numérique avec laquelle on opère.
Tous les opérateurs arithmétiques binaires (à deux opérandes) ont leur homologue préfixé de '=', qui permet un assignement immédiat du résultat au premier opérande. (ex: '<<=', '*='...)

Opérateur	Usage	Exemple
a + b	Addition de deux opérandes	a + 5
a - b	Soustraction de deux opérandes	b - 8
a / b	Divise l'opérande a par l'opérande b	7 / 8
a * b	Multiplie l'opérande a par l'opérande b	9 * 6
a % b	Renvoie **a** modulo **b** (reste de la division euclidienne de a par b)	5 % 3
a >> b	Décale **a** de **b** bits vers la droite	4 >> 1 (renvoie 2)
a << b	Décale **a** de **b** bits vers la gauche	4 << 1 (renvoie 8)
a++	Ajoute 1 à **a** et assigne ce résultat à **a** (incrémentation)	val++
a--	Soustrait 1 à **a** et assigne ce résultat à **a** (décrémentation)	val--
a += b	Ajoute **b** à **a** et assigne le résultat à **a**	val += 3
a -= b	Soustrait **b** à **a** et assigne le résultat à **a**	val -= 4
a = b	Assigne la valeur de b à a	val = 5
…	…	…

Les opérateurs logiques

Ce sont les opérateurs qui vont nous aider à faire des calculs à l'échelle du bit. Ils vont aussi nous aider à résoudre des opérations logiques (où le résultat attendu est 0 ou 1, faux ou vrai).

Opérateur	Usage	Exemple
a && b	Renvoie un **ET** logique entre **a** et **b** qui renvoie VRAI ou FAUX.	condition && 1 (renvoie condition)
a || b	Renvoie un **OU** logique entre **a** et **b** qui renvoie VRAI ou FAUX.	condition || 0 (renvoie condition)
!a	Renvoie un **NOT** logique de **a**. Si a est vrai, renvoie faux. Sinon, renvoie vrai.	!1 (renvoie 0)
a & b	Renvoie le masquage logique **ET** de **a** par **b**.	10 & 1 (renvoie 0)
a | b	Renvoie le masquage logique **OU** de **a** par **b**.	11 | 5 (renvoie 15)
a ^ b	Renvoie le résultat de l'opération logique **XOR** de **a** sur **b**.	10 ^ 7 (renvoie 13)
~a	Renvoie le complément à un (inversion binaire totale) de **a**.	~0 (renvoie 255 pour 8 bits, 65535 pour 16 bits…)

Pour les 4 derniers opérateurs, il est possible de les préfixer par le symbole '=' afin d'assigner immédiatement son résultat à la première opérande (la seule dans le cas de l'opérateur '~').

Les opérateurs comparatifs

Cette fois-ci, nous allons étudier les opérateurs avec lesquels nous allons comparer des éléments.

Opérateur	Usage	Exemple
a < b	Renvoie vrai si **a** est plus petit que **b**.	1 < 5
a > b	Renvoie vrai si **a** est plus grand que **b**.	4 > 3
a <= b	Renvoie vrai si **a** est plus petit ou égal à **b**.	2 <= 2
a >= b	Renvoie vrai si **a** est plus grand ou égal à **b**.	3 >= 3
a == b	Renvoie vrai si **a** est égal à **b**.	5 == 5
a != b	Renvoie vrai si **a** est différent de **b**.	8 != 4

Nous avons fait le tour de la totalité des opérateurs opérant sur des types primitifs définis en C ! Il reste quelques opérateurs à voir en tout, mais le, plus dur est passé. Les autres se dénombrent sur les doigts d'une main.

Les types structurés (structs)

Une structure, potentiellement type structuré, est un type abstrait de données.
Ce n'est pas un objet. C'est une collection de taille fixe de données, regroupées en mémoire au même endroit (par au même endroit, entendre groupées de manière adjacente).
Une struct peut être composée d'un ou plusieurs type primitif, ou d'autres structures.
Pour déclarer une structure en C, on procède comme suit:

```
struct personne
{
      int age;
      char* nom;
      float taille;
};

// et pour l'utiliser:
struct personne personne1;
personne1.age = 24;
personne1.nom = "Catherine";
personne1.taille = 1.84f;
```

Et pour référencer une autre struct à l'intérieur d'une struct:

```
struct moteur
{
      int chevaux_moteur;
      int n_cylindres;
      float taille;
};

struct voiture
{
      int n_roues;
      int couleur;
      struct moteur;
};

struct voiture v1;
v1.n_roues = 4;
v1.couleur = 0xFFFFFFFF;
v1.moteur.chevaux_moteur = 120;
...
```

Afin de construire certains types de données tels que les listes chaînées, il est possible de faire une déclaration anticipée d'une struct:

```c
struct personne;
struct personne
{
	char* nom;
	int age;
	struct personne* parents;
	struct personne* amis;
};
```

Les unions

Les unions permettent de gagner un peu d'espace en mémoire tout en permettant une certaine flexibilité: les unions permettent en fait de réserver un espace mémoire unique, dans lequel stocker toutes les données sont stockées.
Ce qui veut dire que si une donnée est spécifiée non nulle (on a assigné une valeur à l'une d'entre elles), les autres deviennent invalides.
La taille d'une union en mémoire est celle du plus large type qu'elle contient.
On peut construire une union comme suit:

```c
union u
{
	int i;
	char c;
};
union u mon_union;
mon_union.i = 0xFFFF;
mon_union.c = 0;
// mon_union.i va donc être égal à 0xFF00, car il partage son espace
mémoire avec mon_union.c qui est défini comme égal à 0
```

Peu de cas pratiques existent pour les unions, et il est très improbable de tomber dessus ou d'en avoir besoin. Mais au moins la notion est couverte !

Les directives typedef

Les directives *typedef* sont simples à utiliser mais nécessitent de les couvrir afin de raccourcir et rendre plus intuitif le code que l'on écrit.
Elles servent à créer des alias de type, c'est-à-dire à définir des mots-clés personnalisés en couvrant d'autres. Cela est très utile par exemple dans l'utilisation des structs:

```
struct personne
{
	int age;
	char* nom;
	float taille;
};

typedef struct personne Personne;

Personne _pers1;
_pers1.nom = "Yves";
```

Ici, on a créé un alias de la struct personne en la ramenant à un type "Personne".

Il est aussi possible de créer un alias d'un autre type existant:

```
typedef char byte;
```

Quant aux structs, il est aussi possible d'abréger la définition de type et la déclaration de celui-ci en les combinant:

```
typedef struct {
	int age;
	char* nom;
	float taille;
} personne;
```

Les directives préprocesseur

Les directives préprocesseurs sont représentées par du code très simple, que le compilateur va interpréter et exécuter avant la réelle compilation du code du programme. Elles sont généralement utilisées pour ajouter des directives de compilation, telles que la suppression de certaines erreurs en sortie de compilateur, le remplacement de certains mots-clés par d'autres valeurs…
Elles sont très puissantes.

La plus couramment utilisée est déjà couverte dans ce document depuis longtemps: c'est la directive "*include*". cette directive préprocesseur insère virtuellement le contenu du fichier spécifié à la ligne où se trouve la directive. C'est ainsi que le code devient accessible pour le fichier courant.

C'est aussi pour cette raison que l'on ne peut pas *include* de manière circulaire: un fichier a qui include un fichier b, ne peut pas être *include* dans un fichier c qui est *included* dans b. Cela mène à un temps de compilation infini (dit récursif), car chaque écriture de fichier mène à la réécriture d'un autre en boucle.

Une directive qui permet de prévenir ce genre de comportement est *once*. Afin de rajouter des informations sur la manière de compiler un fichier, on peut utiliser les commentaires de compilation avec la directive *pragma*. Chaque compilateur traitera ces commentaires à sa manière cependant, donc il est important de rester sur ses gardes quant à l'utilisation de cette directive:
#pragma once

Une autre manière standard de prévenir la compilation circulaire est d'utiliser les directives #if, #define et #endif:

```
#ifndef MON_FICHIER // si la constante MON_FICHIER n'a pas encore été définie. En première ligne.
#define MON_FICHIER // déclarer MON_FICHIER en constante préprocesseur
#include ... // faire les includes
... // code
#endif // fin du fichier
```

La directive #define est très puissante. Elle permet de déclarer des constantes, certes, mais aussi de définir ce qu'on appelle des macros génératives.

Pour déclarer une constante, il suffit de marquer le nom de la constante et d'enchaîner ensuite sur la valeur qu'on souhaite lui assigner:

```
#define ANNEE 2024
```

```
// à noter d'une part qu'on ne met pas de point virgule pour les directives préprocesseur, et d'autres part que rien ne l'empêche de générer du code plutôt que '2024'.
```

Quant aux macros génératives, ce sont des *define* qui prennent des paramètres. Ce sont des sortes de fonctions que l'on peut appeler depuis le préprocesseur, et qui vont générer du code correspondant à ce que l'on souhaite faire. On les définit comme suit:

```
#define AFFICHER_MESSAGE(message) printf("Message: %s", message) // à noter que les variables sont non-typées dans les arguments de la directive préprocesseur

void main()
{
	AFFICHER_MESSAGE("Hellow, orld !"); // le code printf("Message: %s", "Hellow, orld !") sera généré
}
```

Une remarque intéressante ? La signature de la fonction main a changé. En réalité, seul le nom de la fonction main compte, pas sa signature complète. Ici, main sera toujours exécuté en tant que point d'entrée du programme.

Les opérateurs relatifs aux directives préprocesseur

Encore d'autres opérateurs existent pour faire des opérations que seul le préprocesseur peut faire !

Rappel sur le fonctionnement d'un compilateur

Pour comprendre comment cela marche, il faut rappeler le fonctionnement d'un compilateur. Tout d'abord, un *lexer* (analyseur lexical) va passer sur tout le code source du programme afin de vérifier que les termes employés font partie du *lexique* (dictionnaire) du compilateur. Quand on crée des variables ou des fonctions, on ajoute des entrées à ce dictionnaire. Ensuite, un *parser* (analyseur syntaxique) va vérifier la structure correcte du programme (s'il manque des points-virgules, si des parenthèses fermantes ne mènent à aucune parenthèses ouvrante et vice-versa…) ainsi que collecter les différents *tokens* (lexèmes) du programme. Une fois ce processus mené à bien, l'unité de translation du compilateur rentre en jeu. Son but va être de transformer tous les tokens que le parser a collecté en code binaire compréhensible par l'ordinateur.

Opérateurs préprocesseur

Maintenant ce fait rappelé, on sait que tout ce qui est traité par le compilateur n'est pour lui rien d'autre que des chaînes de caractères. Et ainsi, il est facile pour lui de tout traiter comme tel. On peut faire ces opérations grâce à certains opérateurs:

```
#define STR(a) #a // pour toute valeur, la renvoie entre guillemets en tant
que chaine nullement terminée
#define CONCAT(a, b) a ## b // concatène (colle) deux mots qu'on lui donne

void main()
{
      printf("%s", STR(ceci est une chaine de caractères !));
      printf("%d", CONCAT(1, 0)); // affiche 10
}
```

Constantes magiques

Certaines constantes arrivent prédéfinies avec le compilateur, dont certaines sont très répandues. Nous allons ici parler de __COUNT__, __FILE__, et __LINE__.

```
#define DEBUG printf("Nombre d'erreurs rencontrées: %d\n Erreur ligne: %d
dans le fichier: %s\n", __COUNT__, __LINE__, __FILE__);

void main()
{
      DEBUG // nb erreurs: 1
      DEBUG // nb erreurs: 2
      DEBUG // nb erreurs: 3
}
```

En effet, la macro __COUNT__ s'incrémente à chaque fois qu'elle est rencontrée dans le code, où que cela soit en dehors des directives préprocesseur.

Quant à la macro __FILE__, elle renvoie une chaîne de caractères représentant le chemin complet vers le fichier où elle est appelée.

Finalement, la macro __LINE__ affiche le numéro de ligne où elle a été appelée.

Interagir avec la bibliothèque standard

La bibliothèque standard C regorge de fonctionnalités essentielles permettant de faire à peu près n'importe quoi. Nous verrons ici quelques fichiers d'entête à inclure dans notre code afin d'accéder à un potentiel encore plus grand du standard C.

Les entrées et sorties avec <stdio.h>

Le header "stdio.h" signifie "standard Input Output header", permettant donc la gestion de flux entrée / sortie.

Accès console

Certaines fonctions permettent d'écrire dans la console ou d'y écrire. On peut à ces fins utiliser les *formats* (qu'on utilise depuis le début de ce document) afin d'insérer différents types de données dans les chaînes. Le tableau des formats disponibles en C peut être décliné selon:

%c	un caractère
%s	une chaîne de caractère terminée 0
%hi	short (signé)
%hu	short (non signé)
%Lf	long double
%n	n'affiche rien
%d	nombre entier décimal (base 10)
%i	nombre entier (base détectée automatiquement)
%o	(base 8) integer octal
%x	(base 16) integer hexadécimal
%p	adresse (ou pointeur)
%f	float
%u	nombre non signé
%e	nombre flottant en notation scientifique
%E	nombre flottant en notation scientifique

%% le symbole %

Il est donc possible d'utiliser ces formats pour toute fonction d'entrée / sortie utilisant les formats (print**f**, scan**f**…), généralement préfixées -f.
Les fonctions les plus utilisées exclusivement pour la console sont printf(const char* format, …) et scanf(const char* format, …), servant respectivement à afficher un message dans la console et l'autre à récupérer des valeurs depuis l'entrée en console.

Lire et écrire dans des flux (fichiers)

Nous allons maintenant apprendre à écrire et lire au sein d'un fichier. Pour ce faire, nous allons utiliser les fonctions fopen(), fwrite(), ftell(), fseek(), fclose() et fread().

```
#include <stdio.h>

void main()
{
	FILE* mon_fichier = fopen("./dossier/mon_fichier.txt", "r"); // ouverture en mode lecture
	if(mon_fichier) // si l'adresse mémoire de mon fichier est non nul, il a été ouvert avec succès.
	{
		fseek(mon_fichier, 0, SEEK_END); // curseur vers fin du fichier
		size_t taille_fichier = ftell(mon_fichier); // on récupère le décalage (offset) avec l'octet de début du fichier pour avoir la taille de celui-ci
		fseek(mon_fichier, 0, SEEK_SET); // curseur vers début du fichier
		char contenu_fichier[65335];
fread(contenu_fichier, sizeof(char), taille_fichier, mon_fichier); // lecture du contenu du fichier et copie de celui-ci dans le buffer "contenu_fichier"
fclose(mon_fichier); // on nettoie la mémoire utilisée par le fichier et sa lecture
printf("Contenu du fichier: \n%s", contenu_fichier);
	}
}
```

C'est la première fois qu'on le voit ici, mais on l'utilisera en fait souvent: l'opérateur sizeof() est un opérateur unaire (un seul opérande) qui permet de récupérer la taille de n'importe quel type (primitif ou non) en octets. Dans le cas d'un *char*, cela renvoie donc 1.
Et si… On s'avouait que la console n'était en fait qu'un flux, de la même manière que l'est un fichier ?

```c
#include <stdio.h>

void main()
{
	const char msg[] = "Ceci est un message qui s'affiche dans la console.";
	fwrite(msg, sizeof(char), sizeof(msg), stdout); // stdout signifie standard output, ou sortie standard.
}
```

De la même manière qu'il est possible de lire **stdin** (standard input).
Il est possible d'utiliser l'opérateur sizeof() sur des chaînes de caractères calculées à la **compilation** seulement, donc pas avec les chaînes que l'on remplit procéduralement (durant l'exécution du programme).

Les fonctions d'interaction avec l'OS avec <stdlib.h>

Le header standard ajoute des types primordiaux, tels que **size_t** (entier non signé de 64 bits, servant à stocker une taille (de fichier, de buffer…) et des fonctions d'allocation dynamique primordiales.
Pour voir la documentation complète sur stdlib.h, voir ici.

Les allocations mémoire dynamiques

On se rappelle bien de comment définir un tableau avec une taille fixe… Mais parfois, on aimerait bien pouvoir adapter la taille de nos tableaux à celle de ce qu'on souhaite y stocker même au moment de l'exécution. C'est ici qu'entrent en jeu les fonctions **malloc**, **calloc**, et **free**.

```c
#include <stdlib.h>
void main()
{
	if(FILE* f = fopen("./u/wu.txt", "r"))
	{
	fseek(f, 0, SEEK_END);
	size_t fsize = ftell(f);
	fseek(f, 0, SEEK_SET);
	char* buffer = malloc(sizeof(char) * (fsize + 1));
	fread(buffer, sizeof(char), fsize, f);
	buffer[fsize] = 0; // terminer la chaîne par un caractère nul
	printf("%s", buffer);
	free(buffer); // on libère l'espace qu'on a précédemment alloué et réservé : sinon, en sortie du bloc de code du if, le pointeur sur l'adresse
```

```
qu'on a réservé est perdu mais toujours réservé pour notre programme. La
mémoire surcharge inutilement: c'est un memory leak.
      }
}
```

la fonction **calloc**, quant à elle, permet d'allouer un espace en mémoire et retourne l'adresse du premier octet alloué comme **malloc**, mais initialise toute la plage allouée en assignant la valeur **0** à tous les octets de ladite plage. À savoir que les pointeurs retournés par **malloc** et **calloc** peuvent être manipulés de la même manière que tout autre tableau, tant qu'ils sont bien accessibles.

Il est aussi possible d'utiliser la fonction **realloc** afin d'agrandir la mémoire réservée pour certaines données. Parfois, en fonction de l'arrangement courant de la mémoire, il est nécessaire à cette fonction de porter les données vers une autre adresse, renvoyée par cette fonction. Utilisée sur des blocs de mémoire alloués en utilisant la fonction **malloc**.

D'autres fonctions pratiques résident dans cette bibliothèque, comme celles permettant la conversion de chaînes de caractères en nombres entier (par "parsing"), ainsi que la fonction **abs**, renvoyant la valeur absolue de n'importe quel nombre passé en paramètre.

Gérer les chaînes de caractères avec <string.h>

Le site documentaire en programmation Koor.fr dit de string.h:
"
La librairie <string.h> (<cstring> en C++) permet deux grandes catégories de traitements : les manipulations de chaînes de caractères et les manipulations de blocs mémoires. En fait, il n'y a pas tant de différences entre les deux types de traitements dans le sens ou une chaîne de caractères est un bloc d'octets en mémoire : la seule différence étant qu'une chaîne de caractères possède un marqueur de fin particulier '\0' au contraire d'un bloc de mémoire pour lequel il faut connaître sa taille.

Les fonctions de manipulation de chaînes de caractères sont préfixées par *str*. Les fonctions de manipulations de blocs mémoires sont, quand à elles, préfixées par *mem*.
"

Des fonctions telles que strlen() pour compter la longueur d'une chaîne, ou encore memcpy pour copier le contenu d'un buffer dans un autre en mémoire.
[string.h sur Koor.fr]

Les nombres à taille fixe cross-platform avec <stdint.h>

Cette bibliothèque standard définit des règles fixes concernant la taille de chaque type de donnée entier.

Convention de typage

Il est facile de se repérer au sein de cette bibliothèque, car les conventions de nommage sont strictes et bien respectées. Un type de nombre non signé est préfixé **u**, s'ensuit le mot int pour nombre entier, le nombre de bits qu'il prend, et pour finir suffixé **_t** pour préciser que c'est la définition d'un type. Par exemple, un entier de 16 bits (habituellement un short) sera un **int16_t**. Un entier non signé de 64 bits sera un **uint64_t**.

Tout voir ici sur le site d'IBM.

Ajout des booléens avec <stdbool.h>

Ce header ajoute le type **bool** au programme, qui n'est en fait rien d'autre qu'un simple octet (char). À noter l'ajout de la constante **true** et **false**.

Support étendu des nombres flottants avec <float.h>

Permet une manipulation avancée des flottants de différentes bases numériques. En voir plus sur TutorialsPoint.

Ajout de fonctions et constantes mathématiques avec <math.h>

Ce header ajoute certaines constante mathématiques telles que l'exponentielle e (**M_E**) π (**M_PI**), et des fonctions telles que **cos**, **sin**, **tan**, **asin**, **acos**, **atan**, **exp**, **log10**, **log2**, **ln**, **fmod** pour les modulo de flottants…
Voir math.h sur Koor.fr