Introduction à la Programmation des Algorithmes

1.3. Langage C – Mémoire, variables et types

François Fleuret
https://fleuret.org/11x001/

Un programme C minimal a la forme suivante:

```c
#include <stdio.h>

int main(void) {
  printf("Bonjour\n");
  return 0;
}
```

- La ligne `#include <stdio.h>` indique que l'on inclut le contenu du fichier `stdio.h`, qui est disponible en standard avec le compilateur C.
- La ligne `int main(void) {` marque ce que l'ordinateur devra exécuter en premier, qui se termine par le `}` correspondant.
- La ligne `printf("Bonjour\n");` indique à l'ordinateur d'afficher le texte `Bonjour` et de revenir à la ligne. `printf` est définie dans `stdio.h`.
- La ligne `return 0;` indique que le programme se termine sans erreur.

Les lignes du programme sont donc lues et exécutées dans l'ordre. En particulier les lignes qui se terminent par `;` dans le `main`.

Nous allons clarifier ces différents composants dans ce cours, et voir comment remplacer le `printf` par des instructions plus complexes.

Pour exécuter ce programme nous devons d'abord compiler le **fichier source** que nous écrivons, pour créer un **fichier exécutable** qui contiendra des instructions en langage machine que le CPU peut comprendre.

Si je fais ces opérations depuis un terminal en utilisant `emacs` comme éditeur de texte et `clang` comme compilateur C, et en appelant mon fichier source `test.c` et mon exécutable `test`, cela donne:

```
~ emacs test.c
~ clang -o test test.c
~ ./test
Bonjour
```

Nous utiliserons une interface web pour les TPs.

Variables

Un premier élément clé en programmation est le concept de “variable”, qui permet de manipuler des quantités comme des nombres ou des caractères.

Pour pouvoir l'aborder, nous devons d'abord revenir sur la structure de la **mémoire vive** de l'ordinateur.

La mémoire vive peut être vue comme une longue série de nombres compris entre 0 et 255, que l'on appelle des **octets**.

Cette curieuse plage de valeurs correspond à ce qui est représentable en binaire avec huit chiffres ou **bits** (*Binary digITS*).

En base 10, il y 10 chiffres $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, et chaque rang dans un nombre “vaut” 10 fois plus que le rang à sa droite:

$$203 = 2 \cdot 10^2 + 0 \cdot 10^1 + 3 \cdot 10^0$$

En binaire, qui est la base 2, il n'y a que deux chiffres $\{0, 1\}$ et chaque rang dans un nombre vaut le double du rang à sa droite:

$$10011000 = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$$

0	00000	10	01010
1	00001	11	01011
2	00010	12	01100
3	00011	13	01101
4	00100	14	01110
5	00101	15	01111
6	00110	16	10000
7	00111	17	10001
8	01000	18	10010
9	01001	19	10011

La représentation en base 2 est particulièrement adaptée à des systèmes physiques dans lesquels chaque bit correspond à la présence ou l'absence d'une charge électrique ou magnétique.

La mémoire vive peut donc être vue comme une longue séquence de groupes de 8 bits appelés **octets**, chacun avec une adresse.

Pour faire références à un groupe d'octets, on utilise les unités

kilo-octets	2^{10}	$1'024$	$\simeq 10^3$
méga-octets	2^{20}	$1'048'576$	$\simeq 10^6$
giga-octets	2^{30}	$1'073'741'824$	$\simeq 10^9$
téra-octets	2^{40}	$1'099'511'627'776$	$\simeq 10^{12}$

On peut donc se représenter la mémoire comme une suite de “cases”, chacune correspondant à un octet, avec une adresse (ici en rouge) et contenant une valeur comprise entre 0 et 255, ici écrite en binaire (en noir). Les valeurs sont sur cette figure complètement arbitraires.

0 00011001	1 11110010	2 10001100	3 10111111	4 00011010	5 01000000	6 00101001	7 00010101
8 10000000	9 11001111	10 10110110	11 01111011	12 00001000	13 01110011	14 00010110	15 01110101
16 00011011	17 00101000	18 10001111	19 00100101	20 10100101	21 00110100	22 10100000	23 11101101
24 10110111	25 11100111	26 10011100	27 00000010	28 10111110	29 10000100	30 10111100	31 11101000

Plusieurs octets consécutifs peuvent représenter une grandeur entière avec une plage de valeurs plus étendue.

	En mémoire	Valeur min	Valeur max
Entier positif	1 octet	0	255
Entier signé	1 octet	-128	127
Entier positif	4 octets	0	4294967295
Entier signé	4 octets	-2147483648	2147483647

Par exemple un groupe de 4 octets qui représente un entier positif est vu comme composé de $4 \times 8 = 32$ bits, ayant des valeurs individuels allant de 2^0 à 2^{31}.

Remarque: par convention, sur les processeurs Intel et ARM, les octets sont rangés en mémoire par importance croissante. La convention inverse est utilisée sur d'autres machines.

En programmation, une **variable** est une zone mémoire à laquelle on associe un **identifiant**, qui est un nom pour y faire référence, et un **type**, qui spécifie comment interpréter les octets qui la composent.

Un **identifiant** est un label composé de chiffres, lettres minuscules, lettres majuscules et du "underscore", c'est à dire le caractère `'_'`.

Un identifiant ne peut pas commencer par un chiffre.

Exemples:

- `n`
- `x0`
- `nombre_de_voitures`
- `compteurDeTours`
- `_tmp`

Types

Un **type** est soit pré-défini dans le langage lui-même ("type de base"), soit défini dans le programme.

Il y a de nombreux types de base en C, mais nous n'en utiliserons que trois:

- Le type `int` occupe 4 octets et représente une valeur entière comprise entre -2147483648 et 2147483647 inclues.
- Le type `float` occupe 4 octets et représente une valeur réelle comprise entre $-3.4 \cdot 10^{38}$ et $3.4 \cdot 10^{38}$.
- Le type `char` occupe 1 octet et représente un caractère, par exemple une lettre ou un chiffre.

Nous verrons plus tard comment nous pouvons créer nos propres types en combinant des types existants.

La conversion entre un octet et un caractère se fait avec une table. La plus standard étant la table ASCII ("American Standard Code for Information Interchange") qui remonte aux années 60s.

0 NUL	1 SOH	2 STX	3 ETX	4 EOT	5 ENQ	6 ACK	7 BEL	8 BS	9 HT	10 LF	11 VT	12 FF	13 CR	14 SO	15 SI
16 DLE	17 DC1	18 DC2	19 DC3	20 DC4	21 NAK	22 SYN	23 ETB	24 CAN	25 EM	26 SUB	27 ESC	28 FS	29 GS	30 RS	31 US
32 SP	33 !	34 "	35 #	36 $	37 %	38 &	39 '	40 (	41)	42 *	43 +	44 ,	45 -	46 .	47 /
48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7	56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G	72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W	88 X	89 Y	90 Z	91 [	92 \	93]	94 ^	95 _
96 `	97 a	98 b	99 c	100 d	101 e	102 f	103 g	104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w	120 x	121 y	122 z	123 {	124 |	125 }	126 ~	127 DEL

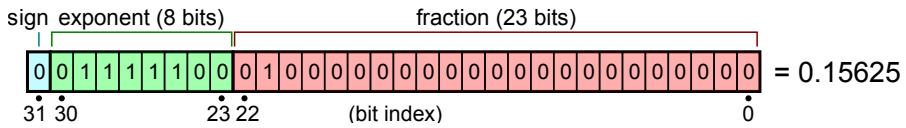
Des conventions de codage sur plusieurs octets permettent de représenter d'autres alphabets avec des milliers de caractères. Le standard est Unicode.

Autant la représentation des entiers est simple, autant celle des réels est complexe. Elle est définie précisément dans la norme IEEE 754.

Essentiellement, elle ressemble à la "notation scientifique": un réel est représenté comme une paire d'entiers: la **mantisse** m et l'**exposant** e, et la valeur qu'ils représentent est

$$\frac{m}{m_{max}} 10^e.$$

Cette représentation permet de travailler à des échelles petites ou grandes. En plus de ces valeurs, un nombre flottant peut être `+inf`, `-inf` ou `nan`.



Nous n'entrerons pas plus dans les détails, mais il peut être nécessaire de se documenter précisément pour programmer des logiciels si la manipulation des réels est critique (simulations physiques, finance).

Avant d'être utilisée dans un programme une variable doit être **déclarée** en indiquant son type puis son identifiant.

Par exemple dans ce programme

```c
#include <stdio.h>

int main(void) {
  int bob_le_nombre_entier;
  return 0;
}
```

on indique à l'ordinateur "réserve un groupe de 4 octets auquel je ferai référence dans la suite avec l'identifiant `bob_le_nombre_entier`, et que j'utiliserai comme une valeur entière."

Comme le programme ne fait rien de plus ici, cette zone mémoire est ensuite libérée.

0 00011001	1 bob_le_nombre_entier	2	3 10111111	4 00011010	5 01000000	6 00101001	7 00010101
8 10000000	9 11001111	10 10110110	11 01111011	12 00001000	13 01110011	14 00010110	15 01110101
16 00011011	17 00101000	18 10001111	19 00100101	20 10100101	21 00110100	22 10100000	23 11101101
24 10110111	25 11100111	26 10011100	27 00000010	28 10111110	29 10000100	30 10111100	31 11101000

Plusieurs variables du même type peuvent être déclarées d'un coup en séparant leurs identifiants par des virgules

```c
#include <stdio.h>

int main(void) {
  int bob_le_nombre_entier, jackUnAutreEntier, n;
  return 0;
}
```


Affectation

Lorsqu'une variable a été **déclarée**, on peut ensuite lui **affecter** une valeur avec le symbole `'='`.

```
#include <stdio.h>

int main(void) {
  int bob_le_nombre_entier;
  bob_le_nombre_entier = 1025;
  return 0;
}
```

En C une variable déclarée a une valeur indéfinie tant qu'une valeur ne lui a pas été explicitement affectée.

Comme $1025 = 1024 + 1 = 2^{10} + 2^0$, voici l'état de la mémoire après `bob_le_nombre_entier = 1025;`

[0] 00011001	[1] bob_le_nombre_entier	[2]	[3] 10111111	[4] 00011010	[5] 01000000	[6] 00101001	[7] 00010101
[8] 00000001	[9] 00000100	[10] 00000000	[11] 00000000	[12] 00001000	[13] 01110011	[14] 00010110	[15] 01110101
[16] 00011011	[17] 00101000	[18] 10001111	[19] 00100101	[20] 10100101	[21] 00110100	[22] 10100000	[23] 11101101
[24] 10110111	[25] 11100111	[26] 10011100	[27] 00000010	[28] 10111110	[29] 10000100	[30] 10111100	[31] 11101000

On peut également déclarer une ou des variables et faire les affectations de valeurs en même temps:

```
int nb1 = 5, nb2 = 27;
```


Le symbole '=' a donc un sens très différent de celui qu'il a en mathématique. Il ne définit pas une propriété qui est et sera toujours vraie. Il indique simplement de changer quelque chose en mémoire: "copie la valeur spécifiée à droite dans la variable spécifiée à gauche."

Ce programme est parfaitement correct, bien qu'il ne produise aucun résultat:

```
#include <stdio.h>

int main(void) {
  int bob_le_nombre_entier;
  bob_le_nombre_entier = 10042;
  bob_le_nombre_entier = 12;
  bob_le_nombre_entier = 553;
  return 0;
}
```

printf

Un autre concept clé en programmation est celui de **fonction**, qui permet de donner un nom à un bout de programme que l'on veut pouvoir facilement ré-utiliser.

Pour exécuter une fonction, il suffit d'indiquer son identifiant, suivi entre parenthèses de valeurs qui modulent son fonctionnement et que l'on appelle ses **arguments**.

Par convention du langage, quand un programme écrit en C est exécuté, c'est la fonction `main` qui est appelée.

Nous reviendrons extensivement sur cette notion et nous verrons en particulier comment créer nos propres fonctions.

La fonction classique pour faire un affichage en C est `printf`.

On doit indiquer entre parenthèses les **arguments** qui définissent quoi afficher.

S'il y a un seul argument cela doit être une **chaîne de caractères** entre ", et elle est affichée telle quelle.

Les caractères `\n` indique de revenir à la ligne.

```c
#include <stdio.h>

int main(void) {
  printf("Bonjour\n");
  return 0;
}
```



```c
#include <stdio.h>

int main(void) {
  printf("Et de un,\net de deux,\net de trois!\n");
  return 0;
}
```

affiche

```
Et de un,
et de deux,
et de trois!
```

Pour que l'affichage puissent dépendre du fonctionnement du programme, on doit passer plusieurs arguments à `printf`.

Le premier reste une **chaîne de caractères** qui indique un "format" :

1. des caractères à afficher tels quels, et
2. des emplacements spécifiés avec `%d`, `%f` et `%c` qui indiquent où afficher des valeurs entières, réelles, et des caractères respectivement.

Les autres arguments indiquent les valeurs à mettre à la place des `%`.

Remarque: Il existe de nombreux autres formats spécifiables avec `%`.

Par exemple:

- `printf("toto\n")` affiche `'toto'`
- `printf("la valeur est %d\n", 3)` affiche `'la valeur est 3'`
- `printf("pi=%f\n", 3.1415926)` affiche `'pi=3.1415926'`
- `printf("1/%d=%f\n", 50, 0.02)` affiche `'1/50=0.02'`

On peut donc en particulier afficher les valeurs de variables.

```c
#include <stdio.h>

int main(void) {
  int a, b;
  a = 10;
  b = 20;
  printf("a=%d b=%d\n", a, b);
  b = 21;
  printf("a=%d b=%d\n", a, b);
  a = 11;
  printf("a=%d b=%d\n", a, b);
  return 0;
}
```

affiche

```
a=10 b=20
a=10 b=21
a=11 b=21
```


Limitations et dangers des types numériques

Comme nous avons vu, les types numériques souffrent de limitations de représentation:

```
1  int a = 1000000000;
2  printf("a=%d\n", a);
3  a = a + 1000000000;
4  printf("a=%d\n", a);
5  a = a + 1000000000;
6  printf("a=%d\n", a);
```

affiche

```
a=1000000000
a=2000000000
a=-1294967296
```



```
1  float u;
2  u = (0.5 - (0.25 + 0.25)) * 1e16;
3  printf("u=%.16f\n", u);
4  u = (0.3 - (0.1 + 0.2)) * 1e16;
5  printf("u=%.16f\n", u);
6  u = (0.3 - 0.1 - 0.2) * 1e16;
7  printf("u=%.16f\n", u);
```

affiche

```
u=0.0000000000000000
u=-0.5551115274429321
u=-0.2775557637214661
```

De plus, alors que certains langages imposent des contraintes fortes de type, le C est très tolérant, ce qui peut créer des situations problématiques

```
1  float x, y;
2  int n, m;
3  x = 3.4;
4  y = 1e12;
5  m = x;
6  n = y;
7  printf("m=%d n=%d", m, n);
```

affiche

```
m=3 n=-2147483648
```