CentraleSupélec

C

Pierre Wilke

6 septembre 2021

CentraleSupélec

Plan

CentraleSupélec

Introduction

CentraleSupélec

Qu'est-ce que C ?

Un langage de programmation **impérative**.

Très utilisé

- systèmes d'exploitation (Windows, OS X, Linux (donc Android), BSD)
- systèmes de bases de données (MySQL, Oracle, PostgreSQL)
- systèmes embarqués

Un langage de bas niveau = "proche de la machine"

\+ comportement prévisible

\+ performance

+/- permet/nécessite de comprendre comment un ordinateur fonctionne

\- gestion **manuelle** de la mémoire

\- pas d'exceptions

CentraleSupélec

Historique

- En 1972, Dennis Ritchie et Ken Thompson (Bell Labs) développent le système d'exploitation Unix, en assembleur pour le PDP-11.
- Thompson crée le langage B pour développer des programmes pour Unix. Mais les programmes écrits en B sont lents et ne peuvent pas tirer parti de toutes les fonctionnalités du PDP-11.
- Ritchie améliore le langage B, et finit par créer le langage C.
- En 1973, le noyau Unix (v4) est réécrit en majeure partie en C.
- En 1978, Brian Kernighan et Dennis Ritchie publient « The C Programming Language », une spécification informelle du langage C.
- 1989 : Première standardisation du langage : ANSI-C ou C89.
- Adoption par l'International Organization for Standardization (ISO) en 1990 : C90 (= C89)
- Le standard évolue : C99, C11, C18, C2x... (2021 ?)

CentraleSupélec

Hello, World!

```c
#include <stdio.h>
```
← Inclusion de librairies

```c
int main(){
```
← Début de la fonction principale

```c
  printf("Hello, World!\n");
```
← Affichage avec retour à la ligne

```c
  return 0;
```
← Retour de fonction (0 = tout va bien)

```c
}
```

Voir le fichier `001-hello.c`.

CentraleSupélec

Compilation : kézako ?

Votre ordinateur ne sait exécuter que du code machine : une suite d'octets qui représentent des instructions *élémentaires*

La **compilation** consiste à transformer votre code C (lisible et compréhensible par vous, humains) en du code machine (illisible pour vous, mais compréhensible et **exécutable par la machine**)

Mais...

Mais... en Python, je n'ai jamais dû compiler mon programme Mon ordinateur sait exécuter du Python directement ?

Python est un langage **interprété** : à l'exécution, vous appelez un **interpréteur** qui :

- lit votre fichier `.py`
- transforme **à la volée** chaque instruction python en du code exécutable par la machine
- exécute ce code machine

Langages interprétés :

+ pas besoin d'appeler un compilateur (simplicité d'utilisation)
+ votre programme peut être exécuté sur n'importe quelle architecture munie d'un interpréteur Python
- le temps d'analyse du programme et de compilation est payé à chaque exécution

CentraleSupélec

Et en Java ?

Le slogan de Java est « *Compile one, Run Everywhere* ».

Pour ne pas payer le coût de la compilation à chaque exécution, le code Java est **compilé** vers du *bytecode Java*.
Ce bytecode est ensuite exécuté par une JVM (Java Virtual Machine) : un *interpréteur de ce bytecode*.

Compromis entre portabilité et vitesse d'exécution.

En C, pas de compromis : on privilégie la vitesse d'exécution et on compile pour une architecture donnée.

Compiler un programme C

Le fichier que vous compilez : `monfichier.c`
L'exécutable que vous souhaitez obtenir : `monexecutable`

```
$ gcc -Wall monfichier.c -o monexecutable
# erreurs, avertissements, ou rien du tout
$ ./monexecutable
```

- `gcc` : le nom du compilateur (`clang` est un autre choix commun)
- `-Wall` : afficher tous les avertissements (*warnings*)
  Les avertissements du compilateur sont importants : tenez-en compte !
  On pourrait ajouter le flag `-Werror` qui considère les avertissements comme des erreurs et empêche la compilation du programme.
- `monfichier.c`
- `-o monfichier` : désigne le nom du fichier exécutable (*o = output*).

CentraleSupélec

Types, valeurs, expressions

Types

Types

- **`char`** (1 octet) : `'a'`, `'0'`, `'\n'`, `'\t'`...
- **`int`** (4 octets) : `123`, `0xfeb1`, `-247`...
- **`long`** (8 octets) : `123L`, `0xfdaab34cfc645edb`...
- **`float`** (4 octets) : `1.32`...
- **`double`** (8 octets)
- **`void`** (pas de valeur)

Les tailles peuvent varier en fonction de l'architecture.
`sizeof``(`**`long`**`)` donne la taille du type **`long`**.
Voir le fichier `002-sizes.c`.

Pas de type booléen : Par convention, 0 vaut faux. N'importe quel autre entier vaut vrai.

Entiers signés par défaut.
Par exemple, **`int`** = **`signed int`** : $[0; 2^{32} - 1]$
Le type **`unsigned int`** : $[-2^{31}; 2^{31} - 1]$

Encodage des nombres

Un **`unsigned char`** est un nombre entre 0 et 255. Les nombres sont encodés en représentation binaire.

En représentation décimale, le nombre 3562 correspond à $2 \cdot 10^0 + 6 \cdot 10^1 + 5 \cdot 10^2 + 3 \cdot 10^3$
L'idée de la représentation binaire est similaire, mais en utilisant 2 comme base, plutôt que 10. On utilise donc seulement les chiffres 0 et 1, qu'on nomme bits.

Ainsi, le nombre binaire $(01101010)_2$ est égal à

$$\begin{aligned} & 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 + 1 \cdot 2^6 + 0 \cdot 2^7 \\ = \; & 0 + 2 + 0 + 8 + 0 + 32 + 64 + 0 \\ = \; & 106 \end{aligned}$$

Encodage des nombres

La notation $(01101010)_2$ est un peu encombrante, on préfère souvent la notation hexadécimale (base 16).
On y utilise les chiffres de 0 à 9, puis les lettres A = 10, B = 11, . . ., F = 15.
Un digit hexadécimal (*hexit*?) correspond à 4 bits.

Ainsi, on notera $(01101010)_2 = (6A)_{16}$.
On notera plus souvent `0x6A` (ou `0x6a`).

Encodage des nombres

Le type **`unsigned int`** est codé sur 4 octets, c'est-à-dire 32 bits.
Ainsi, 1 million ($(1000000)_{10}$) est représenté en hexadécimal `0x000f4240`.

On peut vérifier en calculant :

$$\begin{aligned} & 0 \cdot 16^0 + 4 \cdot 16^1 + 2 \cdot 16^2 + 4 \cdot 16^3 + 15 \cdot 16^4 \\ = \; & 0 + 64 + 512 + 16384 + 983040 \\ = \; & 1000000 \end{aligned}$$

Addition en binaire

$$
\begin{array}{cccccccc}
 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
+ & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
\hline
 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1
\end{array}
$$

Vérification :

- $(00010010)_2 = 16 + 2 = 18$
- $(00011011)_2 = 1 + 2 + 8 + 16 = 27$
- $(00101101)_2 = 1 + 4 + 8 + 32 = 45 = 27 + 18$

Interlude : les nombres négatifs

Comment sont encodés les nombres négatifs ?

Pour les nombres non signés : on encode le nombre en binaire et on a un intervalle de 0 à $2^{32} - 1$

CentraleSupélec

Interlude : les nombres négatifs

Comment sont encodés les nombres négatifs ?
Pour les nombres non signés : on encode le nombre en binaire et on a un intervalle de 0 à $2^{32} - 1$

Première idée : On pourrait utiliser le bit de poids fort comme bit de signe et utiliser les 31 bits restants pour encoder la valeur absolue du nombre (*approche dite « signe - magnitude »*)
Exemple : -7 = 10000000 00000000 00000000 00000111

Problème :
0 = 00000000 00000000 00000000 00000000
- 0 = 10000000 00000000 00000000 00000000 ?
Deux représentations pour le même nombre : pas terrible...

Addition en signe-magnitude

	5	0	0	0	0	0	1	0	1
+	-7	1	0	0	0	0	1	1	1
	?	1	0	0	0	1	1	0	0

Vérification :

- $(10001100)_2 = -(4+8) = -12$: ça ne marche pas...
- il faudrait utiliser un autre algorithme pour additionner deux nombres, selon que les nombres que l'on veut additionner sont positifs ou négatifs...

Interlude : le complément à 1

Une autre approche pour représenter les nombres négatifs : on prend la représentation binaire et on applique une négation binaire.
Pour encoder -7 :

- on encode 7 : $(00000111)_2$
- on nie (négationne ?) : $(11111000)_2$

	5	0	0	0	0	0	1	0	1
+	-7	1	1	1	1	1	0	0	0
	?	1	1	1	1	1	1	0	1

Vérification :

- on nie le résultat : $(00000010)_2 = (2)_{10}$
- on prend l'opposé : -2

C'est bien le bon résultat, on peut donc utiliser le même algorithme quels que soient les signes des opérandes !

CentraleSupélec

Complément à 1

En revanche, on a toujours deux représentations de 0.
On représente -0 :

- On encode 0 : $(00000000)_2$
- On nie : $(11111111)_2$

Ça complique le test d'égalité entre entiers...

Complément à deux

L'encodage de $-x$ est obtenu en prenant l'encodage de x, puis en appliquant une négation binaire à tous les bits, puis en ajoutant 1.

Pour encoder -7 :

- On encode 7 : $(00000111)_2$
- On applique la négation binaire : $(11111000)_2$
- On ajoute 1 : $(11111001)_2 = (F9)_{16}$

	5	0	0	0	0	0	1	0	1
+	-7	1	1	1	1	1	0	0	1
	?	1	1	1	1	1	1	1	0

Vérification :

- on nie le résultat : $(00000001)_2$
- on ajoute 1 : $(00000010)_2 = (2)_{10}$
- on prend l'opposé : -2

CentraleSupélec

Complément à deux

Une seule représentation de 0, essayons d'encoder « - 0 » :

- On encode 0 : $(00000000)_2$
- On applique la négation binaire : $(11111111)_2$
- On ajoute 1 : $(1\ 0000\ 0000)_2$
- On tronque le résultat à 8 bits : $(00000000)_2 = (00)_{16}$

C'est cette représentation qui est utilisée dans les processeurs actuels.

On peut donc représenter des entiers signés dans l'intervalle $[-2^{N-1}, 2^{N-1} - 1]$ ou des entiers non signés dans l'intervalle $[0, 2^N - 1]$.

- $N = 8$: $[-128, 127]$ ou $[0, 255]$
- $N = 32$: $[-2\ 147\ 483\ 648, 2\ 147\ 483\ 647]$ ou $[0, 4\ 294\ 967\ 295]$

Attention au débordement ! $2^{31} \times 2 = 0$!

Variables

Déclaration de variables :

```
int x;  // déclaration
signed int x;
int y = 12; // déclaration + initialisation
unsigned int y = 12;

long j = 92384623498;
long k = 0x0123456789abcdef; // Notation hexadécimale

char c = 'a';
char d = 18; // Les caractères sont en fait des entiers.
```

Une **variable** est :

- un **nom** (`x`) associé à un emplacement de la mémoire
- a un **type** déclaré explicitement
- contient une **valeur**, qui change au cours de l'exécution du programme

Les caractères - table ASCII

American Standard Code for Information Interchange

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

Afficher du texte : la fonction `printf`

Spécifieurs de format : `%d`, `%f`...

Voir `https://www.cplusplus.com/reference/cstdio/printf/` pour une liste plus complète.

```
printf("%d", 3);
printf("%f", 3.14);
printf("%ld", 3876340937498);
printf("%lf", 3.149823749328749238);

printf("%d est une approximation grossiere de %f\n", 3, 3.14);
```

Caractères d'échappement

```
printf("\n"); // Retour à la ligne
printf("\t"); // Tabulation
printf("\\"); // Backslash \
```

Attention : sortie « bufferisée ». Utilisez `flush` ou un caractère `'\n'` pour forcer l'écriture.

CentraleSupélec

Opérateurs arithmétiques

Voir le fichier `003-ops.c`.

- Opérateurs : `+`, `-`, `*`, `/`, `%`
  - Attention : la division par zéro est interdite. Voir le fichier `004-div0.c`.
- Incrémentation / décrémentation : `i++`, `++i`, `i--`, `--i`
- Décalages : `i >> 3`, `i << 5`
- Comparaisons : `==`, `!=`, `<`, `<=`, `>`, `>=`
- Opérateurs binaires (bit-à-bit) : `&`, `|`, `^`, `~`
- Opérateurs booléens : `!`, `&&`, `||`
- Affectation : `+=`, `-=`, `&=`, `|=`, `>>=`, `<<=`
  - `a += b;` est équivalent à `a = a + b;`

Opérateurs binaires

Négation : `~ 0x12345678`

0x12								0x34								0x56								0x78							
0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	0	0
1	1	1	0	1	1	0	1	1	1	0	0	1	0	1	1	1	0	1	0	1	0	0	1	1	0	0	0	0	1	1	1
0xee								0xcc								0xaa								0x87							

ET binaire (conjonction) : `0x12345678 & 0xff001efc`

0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	0	0
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	1	1	1	1	1	1	0	0
0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	0	0	0

OU binaire (disjonction) : `0x12345678 | 0xff001efc`

0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	0	0
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	1	1	1	1	1	1	0	0
1	1	1	1	1	1	1	1	0	0	1	1	0	1	0	0	0	1	0	1	1	1	1	0	1	1	1	1	1	1	0	0

Opérateurs booléens

```
if(a == 0 || a++){
  ...
}
if(b != 0 && 10 / b > 3){
  ...
}
```

Les opérateurs `&&` et `||` sont **court-circuitants**.
Si `a == 0`, l'expression `a++` n'est pas évaluée.
L'expression `10 / b` n'est évaluée que si `b != 0`. (Pratique !)

CentraleSupélec

Opérateur ternaire

```c
int max = (a > b) ? a : b;

// équivalent à :

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

Instructions de contrôle

Conditions if .. else

```c
if ( i > j ){
  printf("%d > %d\n", i, j);
} else {
  printf("%d >= %d\n", j, i);
}
```

La branche **else** est optionnelle.

```c
if ( i > j ){
  printf("%d > %d\n", i, j);
}
```

Voir le fichier `05-cond.c`.

On peut enchaîner les
if .. else if .. else

```c
if ( i > j ){
  printf("%d > %d\n", i, j);
} else if (j > i) {
  printf("%d > %d\n", j, i);
} else {
  printf("%d == %d\n", j, i);
}
```

Boucles while

```c
int i = 20;
while ( i > 0 ){
  printf("Itération %d\n", i);
  i--; // i = i - 1;
}

int j = 0;
while (j < 20){
  printf("Itération %d\n",j);
  j++; // j = j + 1;
}
```

Voir le fichier `06-while.c`.

CentraleSupélec

Boucles while et `break`

`break` permet de sortir d'une boucle.

```
int i = 20;
while ( i > 0 ){
  printf("Itération %d\n", i);
  i--; // i = i - 1;
  if (f(i) == 0){
    break;
  }
}
```

Voir le fichier `07-while-break.c`.

Boucles while et `continue`

`continue` permet de passer à l'itération suivante d'une boucle.

```
int i = 20;
int sum = 0;
while ( i > 0 ){
  printf("Itération %d\n", i);
  i--; // i = i - 1;
  if (f(i) == 0){
    continue;
  }
  sum += i;
}
```

On calcule la somme des entiers i entre 0 à 20 pour lesquels $f(i) \neq 0$.

Voir le fichier `08-while-continue.c`.

Boucles for

```c
for (int i = 0; i < 20; i++){
  printf("Itération %d\n", i);
}
```

Équivalent à :

```c
{
  int i = 0;
  while (i < 20){
    printf("Itération %d\n", i);
    i++;
  }
}
```

Note : les instructions **`break`** et **`continue`** fonctionnent de la même manière avec les boucles **`for`** qu'avec les boucles **`while`**.

Boucles do .. while

Permet de faire au moins un tour de boucle avant de tester la condition.

```c
int i = ...;
do {
  printf("Itération %d\n", i);
  i--;
} while (i > 0);
```

Équivalent à :

```c
int i = ...;
printf("Itération %d\n", i);
i--;
while (i > 0) {
  printf("Itération %d\n", i);
  i--;
}
```

Switch

Permet de choisir quoi exécuter en fonction des différents valeurs d'une expression.

```
switch (i){
  case 0:
    printf("C'est nul !\n");
    break;
  case 1:
  case 3:
    printf("C'est un petit nombre impair");
    break;
  case 2:
    printf("Les 2 font la paire !\n");
  default:
    printf("Je ne sais pas, c'est trop grand.\n")
}
```

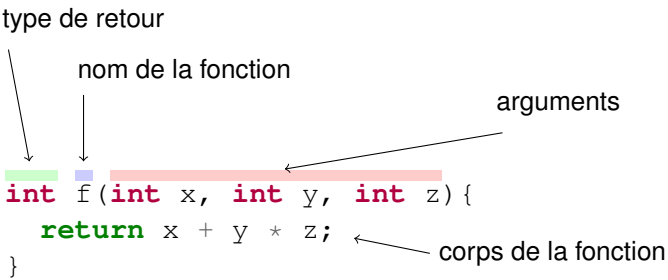
Le **break** est essentiel pour ne pas exécuter le reste du code.

Voir le fichier `09-switch.c`.

CentraleSupélec

Fonctions et programme

Fonctions


type de retour
nom de la fonction
arguments
int f(int x, int y, int z){
return x + y * z;
}
corps de la fonction

Appels de fonctions

```c
int f(int x, int y, int z){
  return x + y * z;
}

int g(int x, int y, int z){
  // Les variables a, b et c sont locales à la fonction g.
  // Elles n'existent pas en dehors de la fonction g.
  int a = x + y;
  int b = y + z;
  int c = f(a, b, a + b);
  return c * 2;
}
```

Déclaration et Définition de fonction

```c
// Déclaration
int f(int x, int y, int z);
int f(int, int, int); // pas besoin de nommer les arguments

// Définition
int f(int x, int y, int z){
  return x + y * z;
}
```

- Une fonction doit être déclarée avant d'être appelée.
- Une définition compte comme une déclaration.

Les fichiers d'en-tête `stdio.h`, `stdlib.h` contiennent des déclarations de fonctions.

- Par exemple, `stdio.h` contient la déclaration de la fonction `printf`.

Portée des variables

Variables globales :

- définies en dehors d'une fonction
- accessibles partout après leur définition

Variables locales :

- définies dans une fonction
- locales au bloc (délimité par des accolades `{ }`) dans lequel elles sont définies

```c
int glob = 123;
int f(){
  int y = 8;
  {
    int x = 3;
    printf("%d\n", x);
  }
  // x n'existe plus ici.
}
// y n'existe plus ici.
// glob existe toujours
```

Retour de fonction

Lorsque le type de la fonction n'est pas **void**, on doit retourner une valeur.

```c
int f (int x){
  return x * 2;
}
```

```c
int f (int x){
  if (x > 0)
    return x * 2;
  else
    return x - 1;
}
```

Tous les chemins d'exécution doivent atteindre un **return** !

```c
int f (int x){
  while(x > 0){
    if (x == 2) return x;
    x--;
  }
}
```

```
$ gcc while-ret.c -o while-ret -Wall
while-ret.c: In function ‘f’:
while-ret.c:9:1: warning: control reaches end of
 ↪  non-void function [-Wreturn-type]
```

Voir le fichier `10-while-ret.c`.

Retour de fonction `void`

On peut utiliser **`return`** sans valeur de retour dans une fonction **`void`**.
Quitte la fonctione prématurément.

```c
void f(int x){
  while(x > 0){
    if (x * 2 == 24) return;
    printf("%d\n",x);
    x--;
  }
}
```

```
# f(15)
$ gcc void-ret.c -o void-ret -Wall
$ ./void-ret
15
14
13
$
```

```
# f(7)
$ ./void-ret
7
6
5
4
3
2
1
$
```

Voir le fichier `11-void-ret.c`.

Passage de paramètres : par valeur

On dit que les paramètres d'une fonction sont passés **par valeur**.
Lorsque l'on appelle `f(i, j)`, on passe en paramètre une copie des valeurs de `i` et `j`.
La fonction appelée ne modifie pas les variables locales de la fonction appelante.

```c
int f(int i, int j){
  i++;
  j = 0;
}
int main(){
  int i = 3;
  int j = 4;
  int r = f(i,j);
  printf("i = %d, j = %d, r = %d\n", i, j, r);
  return 0;
}
```

Un programme entier

Un programme complet comporte plusieurs définitions de fonctions, dont une fonction `main`.
C'est la fonction `main` qui sera exécutée quand vous lancerez votre programme.

```c
// librairie I/O (printf)
#include <stdio.h>

int f(int x){
  ...
}

int main(){
  printf("Résultat: %d\n", f(4));
  // Code d'erreur 0 : tout va bien
  return 0;
}
```

- La fonction `main` a toujours un type retour **`int`**.
- La valeur de retour est appelée **code d'erreur**.
- Par convention, un code d'erreur de `0` signifie que tout s'est bien passé.
- On utilisera d'autres codes de retour (-1, -2, 1, 2, ...) pour indiquer divers types d'erreurs.

Exercice

Complétez le fichier `12-facto.c` dans lequel :

- vous définissez une fonction `facto`
  - qui prend un entier `n` en paramètre
  - qui renvoie la factorielle de `n`
- vous complétez la fonction `main` pour
  - calculer la factorielle de 6
  - afficher le résultat.

Rappel : la factorielle (notée $n!$) d'un entier positif n est le produit de lui-même et de tous ses prédécesseurs.
On peut la définir mathématiquement comme la fonction :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n * (n-1)! & \text{si } n > 0 \end{cases}$$

CentraleSupélec

Tableaux et pointeurs

Les tableaux

Un tableau (*array*) est une collection de valeurs d'un même type.

```
// Déclaration d'un tableau d'entiers, de taille 10, nommé tab
int tab[10];

// Écriture dans des cases du tableau
tab[0] = 8;
tab[1] = 12;

// Accès illégal: seuls tab[0] à tab[9]
tab[10] = 42;

// Accès aux éléments
printf("%d\n", tab[0] + tab[1]);
```

Les tableaux - Initialisation

Par défaut, les cases du tableau ne sont pas initialisées, c'est à vous de le faire.

```c
#include <stdio.h>

int main(){
  int tab[10];
  tab[0] = 12;
  tab[1] = 8;
  for (int i = 0; i < 10; i++)
    printf("tab[%d] = %d\n",
           i, tab[i]);
  return 0;
}
```

Voir le fichier `13-tab.c`.

```
$ gcc -Wall tab.c -o tab
$ ./tab
tab[0] = 12
tab[1] = 8
tab[2] = 0
tab[3] = 0
tab[4] = -458636864
tab[5] = 21884
tab[6] = -458637216
tab[7] = 21884
tab[8] = -207342432
tab[9] = 32765
```

CentraleSupélec

Euh... quoi ?

Que s'est-il passé ? D'où viennent ces valeurs ?

La mémoire d'un programme peut être vue comme un gigantesque tableau d'octets :

	0	1	2	3	4	5	6	7
0	0x6d	0xcd	0xe1	0x8f	0xf7	0x9b	0xdc	0x32
8	0x87	0x66	0x65	0x7b	0xff	0x01	0x3e	0x4d
16	0x87	0xd6	0x12	0x00	0xb3	0x00	0x01	0x02
24	0xf3	0xde	0x18	0xba	0xc7	0xe1	0x32	0x89
32	0x21	0x34	0x12	0xe3	0x8b	0xbc	0xfd	0xe4
40	0x88	0x7d	0x01	0x02	0x03	0x04	0x05	0x00
48	0x90	0x87	0xaa	0x00	0xde	0xad	0xbe	0xef

Lors de la déclaration d'un tableau `int tab[10]`, le compilateur va choisir un emplacement dans la mémoire pour ce tableau.

(pas au hasard, mais il est trop tôt pour vous dire comment)

Les valeurs observées dans l'exemple précédent sont simplement les valeurs qui se trouvaient dans la mémoire à cet endroit au préalable.

Les tableaux - Initialisation

Par défaut, les cases du tableau ne sont pas initialisées, c'est à vous de le faire.

```c
#include <stdio.h>

int main(){
  int tab[10] = { 12, 8, 5, 4, 3, 2,
   ↪  -8, 12, 7, 10 };
  for (int i = 0; i < 10; i++)
    printf("tab[%d] = %d\n",
           i, tab[i]);
  return 0;
}
```

Voir le fichier `14-tab2.c`.

```
$ gcc tab2.c -o tab2
$ ./tab2
tab[0] = 12
tab[1] = 8
tab[2] = 5
tab[3] = 4
tab[4] = 3
tab[5] = 2
tab[6] = -8
tab[7] = 12
tab[8] = 7
tab[9] = 10
$
```

Tableaux et sizeof

On a vu que **sizeof** permettait d'obtenir la taille (en octets) d'un type. Cela fonctionne aussi pour des variables, y compris des tableaux.

```
int x = 3;
int tab[] = {0, 1, 2, 3, 4, 5};
printf("taille du tableau = %ld\n", sizeof(tab));
printf("nombre d'éléments du tableau = %ld\n", sizeof(tab) /
 ↪  sizeof(int));
printf("taille de x = %ld\n", sizeof(x));
return 0;
```

```
taille du tableau = 24
nombre d'éléments du tableau = 6
taille de x = 4
```

Voir le fichier `15-tab-size.c`.

Les tableaux - Pas de vérification des bornes

```c
#include <stdio.h>

int main(){
  int tab[10];
  printf("tab[100] = %d\n",
   ↪  tab[100]);
  printf("tab[1000] = %d\n",
   ↪  tab[1000]);
  return 0;
}
```

```
$ gcc out-of-bounds.c -o
 ↪  out-of-bounds
$ ./out-of-bounds
tab[100] = -1095940154
erreur de segmentation (core dumped)
$
```

Voir le fichier `16-tab-out-of-bounds.c`.

C'est au programmeur (à vous) de s'assurer que l'indice demandé est dans les bornes du tableau.

Les tableaux - Pas de vérification des bornes

Donc on peut dépasser un peu, mais pas trop ? Mais... l'accès à `tab[100]` n'a pas provoqué d'erreur ?

Toute la mémoire n'est pas accessible au programme C :

	0	1	2	3	4	5	...	1023
0	0x6d	0xcd	0xe1	0x8f	0xf7	0x9b	...	0x32
1024	0x87	0x66	0x65	0x7b	0xff	0x01	...	0x4d
2048	0x87	0xd6	0x12	0x00	0xb3	0x00	...	0x02
3072	0xf3	0xde	0x18	0xba	0xc7	0xe1	...	0x89
4096							...	
5120							...	
6284	0x90	0x87	0xaa	0x00	0xde	0xad	...	0xef

Les tableaux - Pas de vérification des bornes

Donc on peut dépasser un peu, mais pas trop ? Mais... l'accès à `tab[100]` n'a pas provoqué d'erreur ?

Toute la mémoire n'est pas accessible au programme C :

	0	1	2	3	4	5	...	1023
0	0x6d	0xcd	0xe1	0x8f	0xf7	0x9b	...	0x32
1024	0x87	0x66	0x65	0x7b	0xff	0x01	...	0x4d
2048	0x87	0xd6	0x12	0x00	0xb3	0x00	...	0x02
3072	0xf3	0xde	0x18	0xba	0xc7	0xe1	...	0x89
4096							...	
5120							...	
6284	0x90	0x87	0xaa	0x00	0xde	0xad	...	0xef

Le compilateur s'assure d'avoir suffisament de place pour les 10 éléments déclarés du tableau.

Donc `tab[0]` à `tab[9]` sont forcément dans une ligne « blanche ».

Les tableaux - Pas de vérification des bornes

Donc on peut dépasser un peu, mais pas trop ? Mais... l'accès à `tab[100]` n'a pas provoqué d'erreur ?

Toute la mémoire n'est pas accessible au programme C :

	0	1	2	3	4	5	...	1023
0	0x6d	0xcd	0xe1	0x8f	0xf7	0x9b	...	0x32
1024	0x87	0x66	0x65	0x7b	0xff	0x01	...	0x4d
2048	0x87	0xd6	0x12	0x00	0xb3	0x00	...	0x02
3072	0xf3	0xde	0x18	0xba	0xc7	0xe1	...	0x89
4096							...	
5120							...	
6284	0x90	0x87	0xaa	0x00	0xde	0xad	...	0xef

Le compilateur s'assure d'avoir suffisament de place pour les 10 éléments déclarés du tableau.

Donc `tab[0]` à `tab[9]` sont forcément dans une ligne « blanche ».

Et `tab[100]`, **par chance**, se trouvait encore sur la même ligne.

Les tableaux - Pas de vérification des bornes

Donc on peut dépasser un peu, mais pas trop ? Mais... l'accès à `tab[100]` n'a pas provoqué d'erreur ?

Toute la mémoire n'est pas accessible au programme C :

	0	1	2	3	4	5	...	1023
0	0x6d	0xcd	0xe1	0x8f	0xf7	0x9b	...	0x32
1024	0x87	0x66	0x65	0x7b	0xff	0x01	...	0x4d
2048	0x87	0xd6	0x12	0x00	0xb3	0x00	...	0x02
3072	0xf3	0xde	0x18	0xba	0xc7	0xe1	...	0x89
4096							...	
5120							...	
6284	0x90	0x87	0xaa	0x00	0xde	0xad	...	0xef

Le compilateur s'assure d'avoir suffisament de place pour les 10 éléments déclarés du tableau.

Donc `tab[0]` à `tab[9]` sont forcément dans une ligne « blanche ».

Et `tab[100]`, **par chance**, se trouvait encore sur la même ligne.

Mais `tab[1000]` a dépassé et s'est retrouvé dans une ligne rouge, à laquelle notre programme n'a pas accès.

Les tableaux - Pas de vérification des bornes

Donc on peut dépasser un peu, mais pas trop ? Mais... l'accès à `tab[100]` n'a pas provoqué d'erreur ?

Toute la mémoire n'est pas accessible au programme C :

	0	1	2	3	4	5	...	1023
0	0x6d	0xcd	0xe1	0x8f	0xf7	0x9b	...	0x32
1024	0x87	0x66	0x65	0x7b	0xff	0x01	...	0x4d
2048	0x87	0xd6	0x12	0x00	0xb3	0x00	...	0x02
3072	0xf3	0xde	0x18	0xba	0xc7	0xe1	...	0x89
4096							...	
5120							...	
6284	0x90	0x87	0xaa	0x00	0xde	0xad	...	0xef

Le compilateur s'assure d'avoir suffisament de place pour les 10 éléments déclarés du tableau.

Donc `tab[0]` à `tab[9]` sont forcément dans une ligne « blanche ».

Et `tab[100]`, **par chance**, se trouvait encore sur la même ligne.

Mais `tab[1000]` a dépassé et s'est retrouvé dans une ligne rouge, à laquelle notre programme n'a pas accès.

RDV en cours de Systèmes d'exploitation pour mieux comprendre à quoi correspondent ces « lignes rouges ».

Passage de tableaux en paramètre

Contrairement aux types de base (**int**, **float**...), les tableaux ne sont pas passés par valeur (copie), mais par adresse (référence).
Cela signifie que le contenu d'un tableau peut être modifié par une autre fonction.

```c
#include <stdio.h>

void f (int tab[2]){
  tab[0] += 1;
}

int main(){
  int tab[2] = { 3, 8 };
  printf("tab[0] = %d\n", tab[0]);
  f(tab);
  printf("tab[0] = %d\n", tab[0]);
  return 0;
}
```

```
$ gcc funtab.c -o funtab
$ ./funtab
tab[0] = 3
tab[0] = 4
$
```

Voir le fichier `17-funtab.c`.

CentraleSupélec

Tableaux multi-dimensionnels

```
int matrice[4][4] = {
      {1,  2,  3,  4},
      {5,  6,  7,  8},
      {9,  10, 11, 12},
      {13, 14, 15, 16}
    };
matrice[1][3]; // 8
```

CentraleSupélec

Exercice

Complétez le fichier `18-sumtab.c` :

- Complétez la fonction `int sumtab(int tab[], int n)` qui calcule la somme des éléments d'un tableau `tab` constitué de `n` entiers.

CentraleSupélec

Les pointeurs

Les pointeurs sont des valeurs qui contiennent une adresse dans la mémoire.

Pointeurs = **LE** concept vraiment compliqué dans C.

Une variable qui contient un pointeur vers un entier est de type `int *`.
Une variable qui contient un pointeur vers un caractère est de type `char *`.
Une variable qui contient un pointeur vers un élément de type `T` est de type `T *`.

Deux opérateurs : adressage `&` et déréférencement `*`.

On récupère un pointeur sur une variable `x` avec l'opérateur `&`.
Exemple : `&x` est un pointeur sur `x`.

On **déréférence** un pointeur `ptr` (on accède à la valeur pointée par ce pointeur) avec l'opérateur `*`.
Exemple : `*ptr` accède à la valeur pointée par `ptr`.
Si `p` est de type `T *`, alors `*p` est de type `T`.

Cas particulier : le pointeur `NULL` ne pointe sur rien.
Déréférencer un pointeur nul provoque une erreur à l'exécution.

Les pointeurs

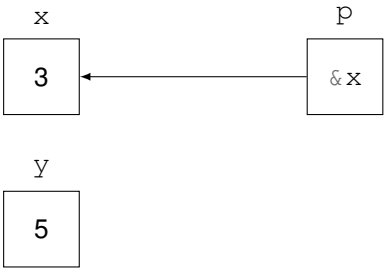
Les pointeurs sont des valeurs qui contiennent une adresse dans la mémoire.
Une variable qui contient un pointeur vers un entier est de type `int *`.
Une variable qui contient un pointeur vers un caractère est de type `char *`.

```c
int main(){
  int x = 1234567;
  int * y = &x;
  printf("1: %d\n", *y);
  *y = 4;
  printf("2: %d\n", *y);
  printf("3: %d\n", x);
  return 0;
}
```

```
1: 1234567
2: 4
3: 4
```

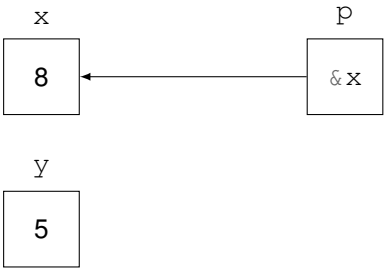
Voir le fichier `19-ptr.c`.

Pointeurs



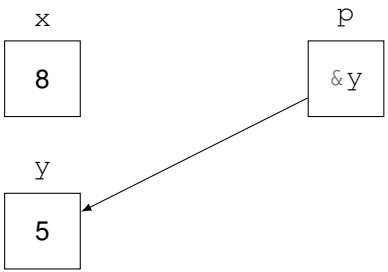

```c
int x = 3;
int y = 5;
int * p = &x;
```

Pointeurs



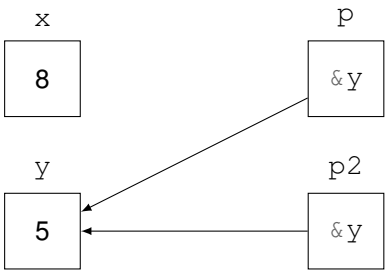

```
int x = 3;
int y = 5;
int * p = &x;
*p = 8;
```

Pointeurs




```c
int x = 3;
int y = 5;
int * p = &x;
*p = 8;
p = &y;
```

Pointeurs




```
int x = 3;
int y = 5;
int * p = &x;
*p = 8;
p = &y;
int * p2 = p;
```

Mémoire et pointeurs

	0	1	2	3	4	5	6	7
0	0x6d	0xcd	0xe1	0x8f	0xf7	0x9b	0xdc	0x32
8	0x87	0x66	0x65	0x7b	0xff	0x01	0x3e	0x4d
16	0x87	0xd6	0x12	0x00	0xb3	0x00	0x01	0x02
24	0xf3	0xde	0x18	0xba	0xc7	0xe1	0x32	0x89
32	0x21	0x34	0x12	0xe3	0x8b	0xbc	0xfd	0xe4
40	0x88	0x7d	0x01	0x02	0x03	0x04	0x05	0x00
48	0x90	0x87	0xaa	0x00	0xde	0xad	0xbe	0xef

Supposons que `x` soit stocké à l'adresse 16.

```
int x = 1234567;
// 0x0012d687
```

Encodage "little-endian" (x86) : les octets les moins significatifs d'abord.

"big-endian" utilisé sur d'autres architectures, ou dans TCP par exemple.

Mémoire et pointeurs

	0	1	2	3	4	5	6	7
0	0x6d	0xcd	0xe1	0x8f	0xf7	0x9b	0xdc	0x32
8	0x87	0x66	0x65	0x7b	0xff	0x01	0x3e	0x4d
16	0x87	0xd6	0x12	0x00	0xb3	0x00	0x01	0x02
24	0xf3	0xde	0x18	0xba	0xc7	0xe1	0x32	0x89
32	0x21	0x34	0x12	0xe3	0x8b	0xbc	0xfd	0xe4
40	0x88	0x7d	0x01	0x02	0x03	0x04	0x05	0x00
48	0x90	0x87	0xaa	0x00	0xde	0xad	0xbe	0xef

Supposons que `x` soit stocké à l'adresse 16.

```
int x = 1234567;
```

L'adresse de x est 16.

```
&x == 16
```

Mémoire et pointeurs

	0	1	2	3	4	5	6	7
0	0x6d	0xcd	0xe1	0x8f	0xf7	0x9b	0xdc	0x32
8	0x87	0x66	0x65	0x7b	0xff	0x01	0x3e	0x4d
16	0x87	0xd6	0x12	0x00	0xb3	0x00	0x01	0x02
24	0xf3	0xde	0x18	0xba	0xc7	0xe1	0x32	0x89
32	0x21	0x34	0x12	0xe3	0x8b	0xbc	0xfd	0xe4
40	0x88	0x7d	0x01	0x02	0x10	0x00	0x00	0x00
48	0x90	0x87	0xaa	0x00	0xde	0xad	0xbe	0xef

Supposons que `x` soit stocké à l'adresse 16.

```
int x = 1234567;
```

L'adresse de x est 16.

```
&x == 16
```

```
int * p = &x;
```

`p` est un **pointeur** sur `x`.

Tableaux et pointeurs

En fait, les tableaux sont des pointeurs !

```c
int tab[5];
```

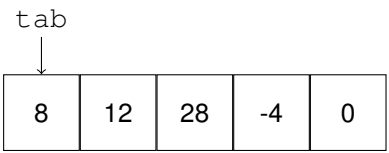
Tableaux et pointeurs

En fait, les tableaux sont des pointeurs !

```
int tab[5];
tab[0] = 8;
```

Tableaux et pointeurs

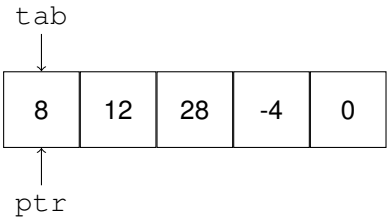
En fait, les tableaux sont des pointeurs !




```
int tab[5];
tab[0] = 8;
tab[1] = 12; tab[2] = 28;
tab[3] = -4; tab[4] = 0;
```

Tableaux et pointeurs

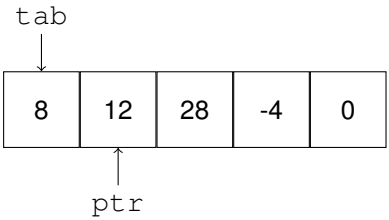
En fait, les tableaux sont des pointeurs !




```
int tab[5];
tab[0] = 8;
tab[1] = 12; tab[2] = 28;
tab[3] = -4; tab[4] = 0;
int * ptr = tab;
```

Tableaux et pointeurs

En fait, les tableaux sont des pointeurs !




```c
int tab[5];
tab[0] = 8;
tab[1] = 12; tab[2] = 28;
tab[3] = -4; tab[4] = 0;
int * ptr = tab;
ptr++;
```

Tableaux et pointeurs

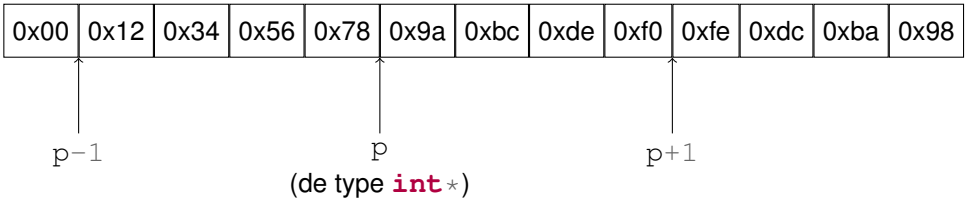
Un accès à un élément d'un tableau `t[i]`, où `t` est un tableau d'entiers :

$$\texttt{t[i]} \equiv \texttt{*(t + i)}$$

Arithmétique de pointeurs

En plus des opérations d'adressage (`&`) et de déréférencement (`*`), on peut faire les opérations suivantes sur des pointeurs :

- Si `p` est de type `T *`, et `i` est un entier (**`char`**, **`int`**, **`long`**...) :
  - `p + i` construit un pointeur qui pointe `i * sizeof(T)` octets plus loin que `p`
  - `p - i` construit un pointeur qui pointe `i * sizeof(T)` octets moins loin que `p`

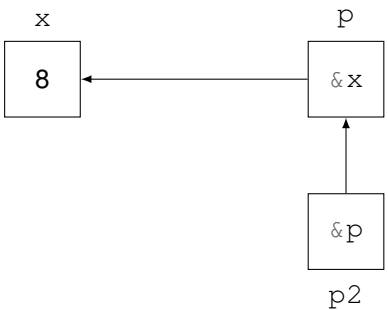


- Si `p1` et `p2` sont deux pointeurs de même type `T*`, alors `p1 - p2` donne le nombre d'éléments de type `T` entre `p2` et `p1`
  Exemple : `(p+1)-(p-1) == 2` (pas 8 !)

Pointeurs : plusieurs niveaux d'indirection

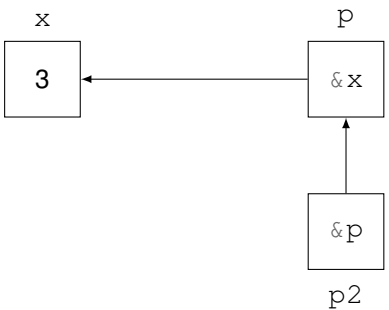

```
int x = 8;
int * p = &x;
```

Pointeurs : plusieurs niveaux d'indirection




```
int x = 8;
int * p = &x;
int ** p2 = &p;
```

Pointeurs : plusieurs niveaux d'indirection




```
int x = 8;
int * p = &x;
int ** p2 = &p;
**p2 = 3;
```

Voir le fichier `20-ptr-indirect.c`.

Pointeurs : plusieurs niveaux d'indirection (vue concrète)

	0	1	2	3	4	5	6	7
0	0x6d	0xcd	0xe1	0x8f	0xf7	0x9b	0xdc	0x32
8	0x87	0x66	0x65	0x7b	0xff	0x01	0x3e	0x4d
16	0x08	0x00	0x00	0x00	0xb3	0x00	0x01	0x02
24	0x10	0x00	0x00	0x00	0xc7	0xe1	0x32	0x89
32	0x21	0x34	0x12	0xe3	0x18	0x00	0x00	0x00
40	0x88	0x7d	0x01	0x02	0x03	0x04	0x05	0x00
48	0x90	0x87	0xaa	0x00	0xde	0xad	0xbe	0xef

```c
int x = 8;
int * p = &x;
int ** p2 = &p;
```

Pointeurs : plusieurs niveaux d'indirection (vue concrète)

	0	1	2	3	4	5	6	7
0	0x6d	0xcd	0xe1	0x8f	0xf7	0x9b	0xdc	0x32
8	0x87	0x66	0x65	0x7b	0xff	0x01	0x3e	0x4d
16	0x03	0x00	0x00	0x00	0xb3	0x00	0x01	0x02
24	0x10	0x00	0x00	0x00	0xc7	0xe1	0x32	0x89
32	0x21	0x34	0x12	0xe3	0x18	0x00	0x00	0x00
40	0x88	0x7d	0x01	0x02	0x03	0x04	0x05	0x00
48	0x90	0x87	0xaa	0x00	0xde	0xad	0xbe	0xef

```c
int x = 8;
int * p = &x;
int ** p2 = &p;

**p2 = 3;
```

Chaînes de caractères

Les chaînes de caractères sont en fait des tableaux de caractères terminé par un caractère nul '\0'

```c
char chaine1[] = { 'H', 'e', 'l', 'l', 'o', '\0'};
printf("%s\n", chaine1);

char chaine2[] = "Bonjour, le Mastère Spécialisé CyberSécurité !";
printf("%s\n", chaine2);

chaine2[7] = '!';
printf("%s\n", chaine2);
```

```
Hello
Bonjour, le Mastère Spécialisé CyberSécurité !
Bonjour! le Mastère Spécialisé CyberSécurité !
```

Voir le fichier `21-str.c`.

Longueur d'une chaîne de caractères

Une chaîne de caractères est un tableau de caractères, terminé par un caractère nul `'\0'`.
On passe une chaîne de caractères à une fonction en donnant le pointeur sur le premier caractère de la chaîne.

```c
int longueur(char * s){
  int lg = 0;
  while (s[lg] != '\0'){
    lg++;
  }
  return lg;
}
```

Voir le fichier `22-longueur.c`.

```c
int longueur(char * s){
  char * p = s;
  while (*p != '\0'){
    p++;
  }
  return p - s;
}
```

OU encore

```c
int longueur(char * s){
  char * p = s;
  while (*p++);
  return p - s;
}
```

Fonctions utiles sur les chaîne de caractères

```c
#include <string.h>

// Renvoie 0 si s1 et s2 correspondent à la même chaîne,
// une valeur négative ou positive sinon.
int strcmp(const char *s1, const char *s2);

// Renvoie la longueur de s
size_t strlen(const char *s);

// size_t est un (autre) type d'entiers (8 octets chez moi)
```

```
$ man strcmp
```

Autres fonctions utiles : `https://en.cppreference.com/w/c/string/byte`

CentraleSupélec

Exercice

Écrivez une fonction (`23-nombre-occurrences.c`) qui compte le nombre d'occurrences d'un caractère donné dans une chaîne de caractères.

- 'c' dans "CentraleSupelec" -> 1
- 'e' dans "CentraleSupelec" -> 4
- 'Z' dans "CentraleSupelec" -> 0

CentraleSupélec

Un exemple concret d'utilisation des pointeurs : scanf

On peut demander à l'utilisateur d'entrer des informations, pendant l'exécution du programme.

```c
int main(){
  int x;
  scanf("%d", &x);
  printf ("Vous avez entré le nombre %d.\n", x);
  return 0;
}
```

`scanf` comprend les mêmes *chaînes de format* que `printf`.
On donne un pointeur sur `x` à `scanf` pour qu'elle sache où stocker le nombre entré.
Voir le fichier `24a-input.c`.

Mais... il vaut mieux éviter `scanf`.
Que se passe-t-il si on entre autre chose qu'un entier ? (gestion d'erreur)

scanf est un peu compliqué...

```c
int main(){
  int x = 3;
  scanf("%d", &x);
  printf ("Vous avez entré le
   ↪  nombre %d.\n", x);
  return 0;
}
```

```
$ ./scanf1
2
Vous avez entré le nombre 2.
$ ./scanf1
a
Vous avez entré le nombre 3. # ???
```

Voir le fichier `24b-input.c`.

La fonction `scanf` retourne le nombre d'éléments qui ont *matché*.
On s'attend à 1 (puisqu'on a 1 `"%d"`)
On peut donc réessayer en boucle, jusqu'à ce que l'utilisateur veuille bien rentrer un nombre.

scanf est un peu compliqué...

```c
int main(){
  int x = 3;
  while(1){
    int res = scanf("%d", &x);
    if (res == 1)
      break;
    else {
      printf("Entrez un *nombre* s'il
       ↪  vous plait: \n");
    }
  }
  printf ("Vous avez entré le nombre
   ↪  %d.\n", x);
  return 0;
}
```

Voir le fichier `24c-input.c`.

```
$ ./scanf2
56
Vous avez entré le nombre 56.
$ ./scanf2
a
Entrez un *nombre* s'il vous plait:
Entrez un *nombre* s'il vous plait:
Entrez un *nombre* s'il vous plait:
Entrez un *nombre* s'il vous plait:
...
# Ctrl + C
$
```

CentraleSupélec

scanf est un peu compliqué...

```c
int main(){
  int x = 3;
  while(1){
    int res = scanf("%d", &x);
    if (res == 1)
      break;
    else {
      printf("Entrez un *nombre* s'il
       ↪  vous plait: \n");
    }
  }
  printf ("Vous avez entré le nombre
   ↪  %d.\n", x);
  return 0;
}
```

Voir le fichier `24c-input.c`.

```
$ ./scanf2
56
Vous avez entré le nombre 56.
$ ./scanf2
a
Entrez un *nombre* s'il vous plait:
Entrez un *nombre* s'il vous plait:
Entrez un *nombre* s'il vous plait:
Entrez un *nombre* s'il vous plait:
...
# Ctrl + C
$
```

Explication : `scanf` tente de lire un entier sur l'entrée standard. Lorsque l'entrée standard ne contient pas un nombre, scanf ne consomme rien et renvoie 0. Les itérations suivantes recommencent avec le même contenu sur l'entrée standard...

Une meilleure manière de récupérer un entier sur l'entrée standard

```c
int input_number(){
  char buf[20];
  int num;
  while(1){
    if (fgets(buf, sizeof(buf), stdin)
     ↪  != NULL){
      if(sscanf(buf, "%d", &num) < 1)
        printf("Je n'ai pas
         ↪  compris...\n");
      else
        break;
    } else {
      printf("fgets error\n");
    }
  }
  return num;
}
```

- On lit une chaîne de caractères depuis l'entrée standard.
- On s'arrête dès que l'on rencontre un retour à la ligne (`'\n'`) ou après 20 caractères.
- On utilise `sscanf` sur la chaîne `buf`.
- Si on ne trouve pas d'entier (le résultat est inférieur à 1), alors on affiche un message et on reboucle.
- Sinon, on a lu un entier, donc on sort de la boucle (**`break`**)

Voir le fichier `24d-input.c`.

CentraleSupélec

Une meilleure manière de récupérer un entier sur l'entrée standard

```
$ ./scanf3
Entrez un nombre :
a
Je n'ai pas compris...
b
Je n'ai pas compris...
X
Je n'ai pas compris...
10
Vous avez entré 10
$
```

Générer des nombres aléatoires

On ne sait pas générer du **vrai** aléatoire.
On sait par contre générer des nombres pseudo-aléatoires.
Un tel générateur est paramétré par une *graine* (*seed*), et construit une séquence de nombres aléatoires à partir de cette graine.
Un choix de graine différent à chaque exécution assure que la séquence de nombres n'est pas prédictible.
Important pour la sécurité (notamment en cryptographie, cf votre cours de crypto).

Générer des nombres aléatoires

```c
#include <stdlib.h>  // pour les fonctions srand et rand
#include <time.h>    // pour la fonction time
int main(){
  // Initialisation de la graine
  srand(time(NULL));
  printf("Un nombre aléatoire : %d\n", rand());
  printf("Un autre nombre aléatoire : %d\n", rand());

  int x = rand() % 100;
  printf("Un nombre aléatoire entre 0 (inclus) et 100 (exclus) : %d\n", x);

  return 0;
}
```

Voir le fichier `25-rand.c`.

CentraleSupélec

Exercice : jeu du plus ou moins

Règle du jeu.
Le programme choisit un nombre aléatoire entre 0 et 100.
L'utilisateur doit deviner quel entier a été choisi.
À chaque fois que l'utilisateur propose un entier, le programme :

- répond "C'est gagné" si le nombre a été trouvé, et on quitte le programme
- répond "Non, c'est plus bas" si le nombre à trouver est plus petit que le nombre entré par l'utilisateur, et on recommence
- répond "Non, c'est plus haut" si le nombre à trouver est plus grand que celui entré par l'utilisateur, et on recommence

Écrivez un programme pour jouer au plus ou moins !

Arguments de `main`

Jusqu'à présent on a écrit des programmes qui ne prenaient pas de paramètre.
On peut passer des paramètres à nos programmes via la ligne de commande :

```
$ ./monProgramme arg1 arg2 arg3 ...
```

Et on peut lire ces arguments depuis la fonction `main`, à condition de lui donner la signature suivante :

```
int main(int argc, char **argv){ ... }
```

- `argc` contient le nombre d'arguments passés
- `argv` est un tableau de chaînes de caractères (de `argc` éléments)
  - c'est-à-dire un tableau de tableaux de caractères
  - on peut aussi écrire `char * argv[]`

Attention : `argv[0]` est toujours le nom du programme, pas le premier argument.

Arguments de `main`

```c
#include <stdio.h>

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

```
$ gcc args.c -o args
$ ./args
argv[0] = /path/to/args
$ ./args arg1 arg2 3 4 hello
argv[0] = /path/to/args
argv[1] = arg1
argv[2] = arg2
argv[3] = 3
argv[4] = 4
argv[5] = hello
```

Voir le fichier `26-args.c`.

On reçoit les arguments comme des chaînes de caractères.

Si on veut des entiers, il faut convertir avec les fonctions suivantes (`#include <stdlib.h>`)

- `int atoi(char* s)` (*ASCII to integer*)
- `long atol(char* s)` (*ASCII to long*)
- `double atof(char* s)` (*ASCII to float*)

CentraleSupélec

Exercice : palindrome

Écrivez une fonction `int palindrome(char* s)` qui renvoie 1 si la chaîne pointée par `s` est un palindrome, et 0 sinon.
Faites en sorte que la fonction `main` appelle cette fonction `palindrome` sur le premier argument passé en ligne de commande.
« laval », « radar », « kayak », « ressasser », « SOS », « abcdcdcba » sont des exemples de palindromes.

CentraleSupélec

Plus de types

Énumérations

On peut créer des types ayant un certain nombre de valeurs.

```
enum couleur {
Pique,
Coeur,
Carreau,
Trefle
};
```

```
enum couleur autre_couleur(enum couleur c){
  enum couleur res;
  switch(c){
    case Pique:
      res = Trefle;
      break;
    case Coeur:
      res = Carreau;
      break;
    case Carreau:
      res = Coeur;
      break;
    case Trefle:
      res = Pique;
      break;
    default:
      break;
  }
  return res;
}
```

Énumérations

On peut créer des types ayant un certain nombre de valeurs.

```
enum couleur {
Pique,
Coeur,
Carreau,
Trefle
};

// Type synonyme !
typedef enum couleur couleur;

couleur autre_couleur(couleur c){
  couleur res;
  switch(c){
    case Pique:
      res = Trefle;
      break;
    case Coeur:
      res = Carreau;
      break;
    case Carreau:
      res = Coeur;
      break;
    case Trefle:
      res = Pique;
      break;
    default:
      break;
  }
  return res;
}
```

Structures

On peut empaqueter des données dans des structures.

```c
struct point {
  int x;
  int y;
  char * nom;
};
int main(){
  struct point p1 = { 3, 4, "A" };
  struct point p2;
  p2.x = 12; p2.y = 7;
  p2.nom = "B";
  struct point p3 = { .y = 8, .nom = "C",  .x = 7} ;
  printf("p1: x = %d, y = %d, nom = %s\n", p1.x, p1.y, p1.nom);
  printf("p2: x = %d, y = %d, nom = %s\n", p2.x, p2.y, p2.nom);
  printf("p3: x = %d, y = %d, nom = %s\n", p3.x, p3.y, p3.nom);
  return 0;
}
```

```
$ gcc struct.c -o struct
$ ./struct
p1: x = 3, y = 4, nom = A
p2: x = 12, y = 7, nom = B
p3: x = 7, y = 8, nom = C
```

Voir le fichier `27-struct.c`.

Pointeurs sur structures

On peut avoir des pointeurs sur des structures.

```c
struct point {
  int x;
  int y;
  char * nom;
};
int main(){
struct point p1 = { 3, 4, "A" };
struct point * p2 = &p1;
printf("p1: x = %d, y = %d, nom = %s\n", p1.x, p1.y, p1.nom);
printf("p2: x = %d, y = %d, nom = %s\n", (*p2).x, (*p2).y, (*p2).nom);
printf("p2: x = %d, y = %d, nom = %s\n", p2->x, p2->y, p2->nom);
return 0;
}
```

```
$ gcc structptr.c -o structptr
$ ./structptr
p1: x = 3, y = 4, nom = A
p2: x = 3, y = 4, nom = A
p2: x = 3, y = 4, nom = A
```

Voir le fichier `28-struct-ptr.c`.

Unions

```c
enum number_t { INT, FLOAT };
struct number {
  enum number_t type;
  union {
    int i;
    float f;
  };
};

struct number mon_entier;
mon_entier.type = INT;
mon_entier.i = 37;
// Interdit d'utiliser mon_entier.f
```

- Les unions sont en général utilisées dans des structures, avec un champ supplémentaire qui indique quel champ de l'union est valide.
- Ici par exemple, le champ `type` indique si on stocke un entier (`INT`, champ `i`) ou un flottant (`FLOAT`, champ `f`)
- C'est au programmeur de s'assurer de la cohérence entre ces deux champs.

Typedef

On peut déclarer des synonymes pour les noms de types :

```
typedef char * string;
struct point {
  int x;
  int y;
  string nom;
};
typedef struct point point;

int main(){
point p1 = { 3, 4, "A" };
point * p2 = &p1;
printf("p1: x = %d, y = %d, nom = %s\n", p1.x, p1.y, p1.nom);
printf("p2: x = %d, y = %d, nom = %s\n", p2->x, p2->y, p2->nom);
return 0;
}
```

Transtypage (cast)

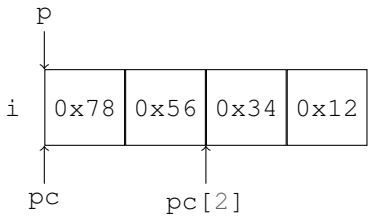
On peut transformer une valeur d'un type t_1 vers un type t_2.

Par exemple, pour transformer un entier en nombre flottant :

```
int i = 3;
float f = (float) i;
float pi = 3.14;
int pi_a_peu_pres = (int) pi;
printf("i = %d, f = %f, pi = %f, pi_a_peu_pres = %d\n", i, f, pi,
 ↪  pi_a_peu_pres);
```

On peut aussi transtyper (**caster**) des pointeurs

```
int i = 0x12345678;
int* p = &i;
char* pc = (char*) p;
for (int j = 0; j < 4; j++){
  printf("pc[%d] = 0x%hhX\n", pc[j]);
}
```



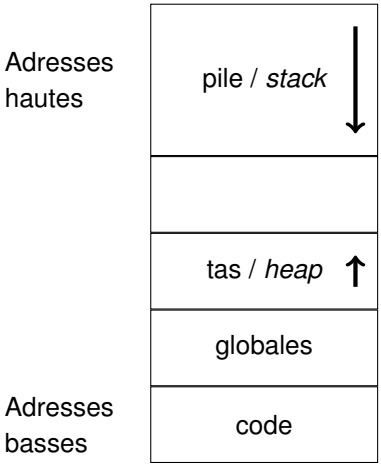
Voir le fichier `29-casts.c`.

CentraleSupélec

Mémoire des programmes et allocation

Gestion de la mémoire

La mémoire d'un processus est organisée de la manière suivante :



- la **pile** contient, entre autres choses, les variables locales de chaque fonction ;
- le **tas** contient des données allouées dynamiquement (on en parle juste après) ;
- les variables globales sont dans une zone dédiée.

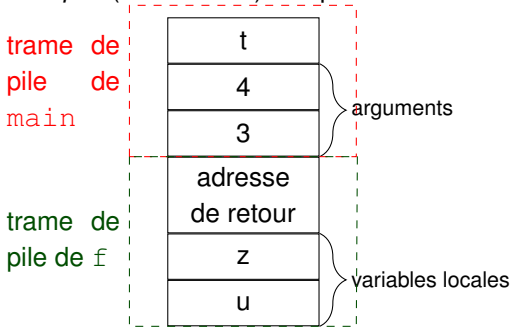
```c
int x = 3;
int main(){
  int i = 2;
  printf("i = %d\n", i);
  return 0;
}
```

- Où est stockée `x` ? `i` ? la chaîne `"i = %d\n"` ?

Appels de fonction et pile

Lors d'un appel à une fonction, on ajoute une *trame de pile* (*stack frame*) à la pile :

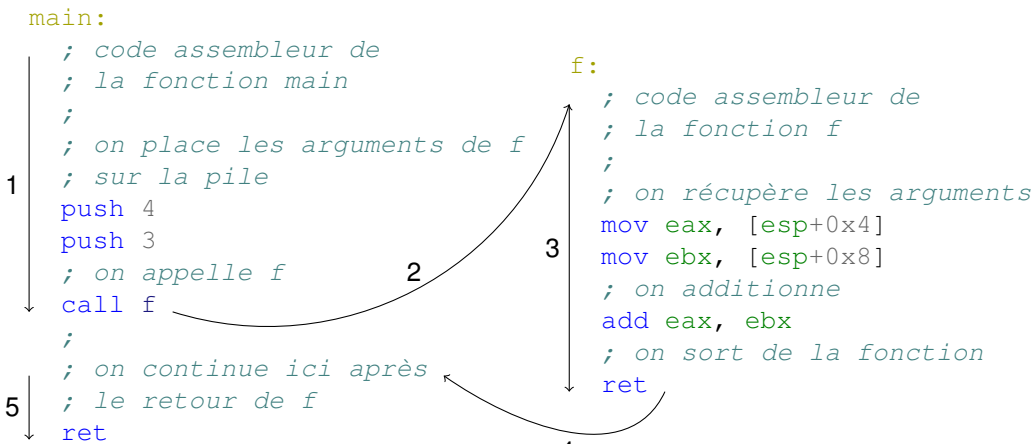
```c
int f(int x, int y){
  int z = x * y;
  int u = z + x;
  return u;
}
int main(){
  int t = f(3,4);
  return t;
}
```



Lors du retour d'une fonction (`return u;`), on dépile cette trame de pile et on continue l'exécution à l'adresse de retour (dans le code).

Important : les variables locales ne sont plus accessibles après le retour de la fonction.

Aparté sur l'adresse de retour



Au point 4, on a besoin de savoir où revenir dans le code de la fonction main. C'est le rôle de l'adresse de retour, stockée sur la pile (en x86).

Variables locales

On ne doit jamais

- retourner un pointeur vers une variable locale
- utiliser un tel pointeur en dehors de la fonction

: la zone pointée sera libérée au retour de la fonction !

```
void f(char **p){
  char s[] = "Bonjour !\n";
  *p = s;
}
int main(){
  char* p;
  f(&p);
  printf("%s\n", p);
  return 0;
}
```

Ce programme n'a pas le comportement attendu ! La variable locale `s` est écrasée par l'appel à `printf`.

Voir le fichier `30-locals.c`.

Allocation dynamique

Jusqu'à maintenant, les allocations de mémoire que l'on a vues étaient **statiques** :

- on connaît la taille à allouer au moment de la compilation
- la portée des variables allouées est le bloc dans lequel elles sont définies

Et si on veut avoir une quantité de mémoire arbitraire, à l'exécution ?
Si on ne connaît pas exactement la quantité de mémoire dont on a besoin ?
Deux possibilités :

- on alloue une quantité fixe, en se disant que "bon quand même, ça devrait suffire" : ça marche, mais
  - et si ça ne suffit pas ?
  - on alloue toujours la quantité maximale
- on demande à l'OS de la mémoire à l'exécution : **dynamiquement**. Cette mémoire sera allouée dans **le tas**.

Allocation dynamique

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

La fonction `void* malloc(size_t size)` alloue de la mémoire dynamiquement.

- la fonction alloue `size` octets
- renvoie un pointeur sur la zone mémoire qui a été allouée.

`void *` signifie : “un pointeur sur n’importe quel type”.

```
int * dyntab = malloc(10 * sizeof(int));
dyntab[0] = 1;
dyntab[3] = 8;
```

Allocation dynamique

La mémoire qui est allouée dynamiquement doit être libérée explicitement.
Fonction `void free(void *ptr)`.

Attention :

- pas de `free` sur un pointeur qui ne vient pas de `malloc`.
- pas deux fois un `free` sur un même pointeur.

```c
int * tab = malloc(...);
//utiliser tab
free(tab);
tab = NULL; // éviter les 'dangling pointers'
```

Listes chaînées

Avec les **`struct`** et l'allocation dynamique, on peut représenter des listes chaînées.

```
struct list {
  int elt;
  struct list* next;
};
```

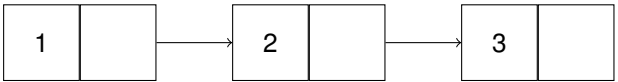
Structure récursive :

- `elt` est le premier élément de la liste.
- `next` est un pointeur vers le reste de la liste (`NULL` si vide).

Note : on peut mettre un champ de type **`struct`** `list*`, mais pas **`struct`** `list` !

Listes chaînées

```
struct list * l1 = malloc(sizeof(struct list));
struct list * l2 = malloc(sizeof(struct list));
struct list * l3 = malloc(sizeof(struct list));
l1->elt = 1;
l2->elt = 2;
l3->elt = 3;
l1->next = l2;
l2->next = l3;
l3->next = NULL;
```



Plus de listes chaînées en TP !

Pointeurs de fonction

Une fonction peut être passée en paramètre d'une autre fonction, sous la forme d'un pointeur de fonction.

```c
int g(int x, int y){ return x * y + 2; }
int applique_fonction(int (*f)(int,int), int x){
  return f(x, x);
}
int main(){
  int res = applique_fonction(g, 4);
  printf("%d\n", res);
  return 0;
}
```

Voir le fichier `31-funptr.c`.

Un cas d'utilisation courant : le tri.

```c
void qsort(void *base, size_t nmemb, size_t size,
                  int (*compar)(const void *, const void *));
```

CentraleSupélec

Préprocesseur et compilation séparée

Préprocesseur

Le préprocesseur est un programme qui va modifier votre code source avant de le donner au compilateur à proprement parler.

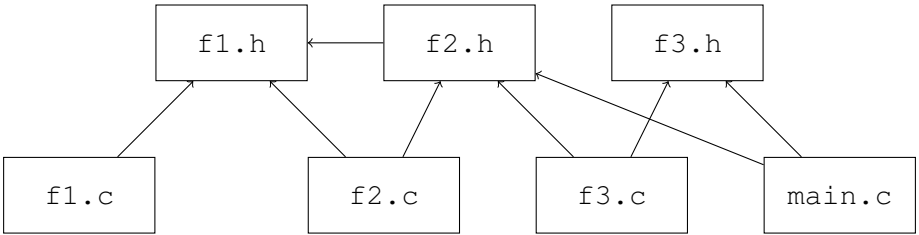
Principalement deux fonctions :

- Directives `#include` : pour inclure d'autres fichiers C
  - `#include <stdio.h>` pour un fichier de la librairie standard C
  - `#include "malibrairie.h"` pour un fichier local
- Directives `#define` : pour définir des constantes, ou des macros
  - Constantes : `#define TAILLE 256`. `TAILLE` sera remplacée partout dans le fichier par `256`.
  - Macros : `#define MUL(x,y) ((x) * (y))`
- Vérifier si une constante est définie :
  - `#ifdef CONST` ... `#endif`
  - `#ifndef CONST` ... `#endif`

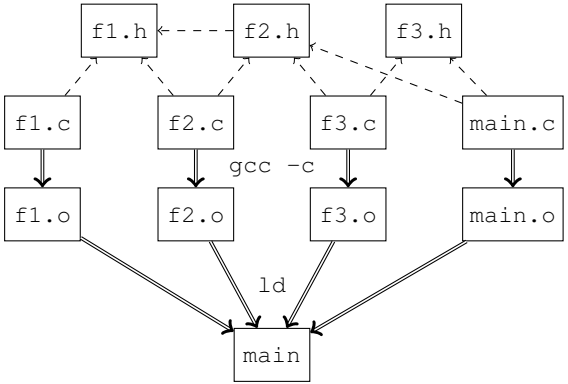
Organisation des fichiers

Votre code sera séparé en plusieurs fichiers `file1.c`, `file2.c`, `main.c`.
Pour compiler tout ça ensemble, il vous faut aussi des fichiers `file1.h` et `file2.h` qui déclarent les types, variables globales et fonctions qui doivent être visibles depuis les autres fichiers.
Les fichiers C incluent les fichiers d'en-têtes (headers), pas les .c

```
#include "file1.h"
#include "file2.h"
...
```



Compilation séparée

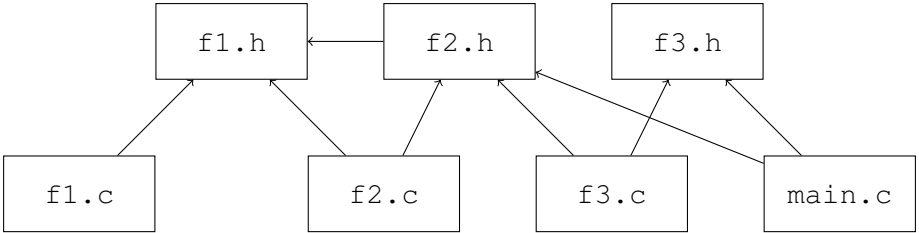



```
$ gcc -c f1.c -o f1.o
$ gcc -c f2.c -o f2.o
$ gcc -c f3.c -o f3.o
$ gcc -c main.c -o main.o
$ ld main.o f1.o f2.o f3.o -o main
```

Ça se passe dans votre dos, il suffit de compiler avec :

```
$ gcc f1.c f2.c f3.c main.c -o main
```

Inclusion multiple



`f2.c` inclut `f1.h` et `f2.h`. Ce dernier inclut lui-même `f1.h` : inclusion multiple.

Solution : chaque fichier d'en-tête est structuré comme cela :

```
#ifndef F1_H
#define F1_H
// les déclarations...
#endif
```

CentraleSupélec

Fichiers

Fichiers

Les fichiers en C sont représentés par une structure **`FILE`**. Cette dernière nécessite d'inclure `<stdio.h>`.

Ouverture de fichier : `FILE * fopen(char* filename, char* mode)`

- `filename` : le nom du fichier (!)
- `mode` : la manière d'ouvrir le fichier :
  - `"r"` : lecture (*read*)
  - `"w"` : écriture en écrasant le fichier (*write*)
  - `"a"` : écriture à la fin du fichier (*append*)

  Si le fichier n'existe pas, et le mode est `"w"` ou `"a"`, il est créé.
  Sinon, `NULL` est renvoyé.

Fermeture de fichier : `int fclose(FILE *fichier)`

Modes d'ouverture des fichiers

Mode	Lecture	Écriture	Fichier	Position
r	oui	non	doit exister	début
r+	oui	oui	doit exister	début
w	non	oui	créé ou vidé	début
w+	oui	oui	créé ou vidé	début
a	non	oui	contenu préservé	fin
a+	oui	oui	contenu préservé	fin

Ajouter un `b` à la fin du mode permet d'ouvrir le fichier en mode *binaire* plutôt que textuel.

Depuis la page de manuel :

> *The mode string can also include the letter 'b' either as a last character or as a character between the characters in any of the two-character strings described above. This is strictly for compatibility with C89 and has no effect; the 'b' is ignored on all POSIX conforming sys- tems, including Linux. (Other systems may treat text files and binary files differently, and adding the 'b' may be a good idea if you do I/O to a binary file and expect that your program may be ported to non-UNIX environments.)*

Fichiers - lecture

```c
// ouverture du fichier en lecture
FILE * fd = fopen("mon_fichier", "r");
char line[256];
// lit jusqu'à un retour à la ligne, ou la fin du fichier, ou 256 octets.
// renvoie line si tout s'est bien passé (ou NULL si on est à la fin du fichier
 ↪  !)
fgets(line, 256, fd);
// lit un caractère
char c = fgetc(fd);
int x;
// lit un entier
fscanf(fd, "%d", &x);
// est-ce la fin du fichier ?
int fini = feof(fd);
```

CentraleSupélec

Fichiers - écriture

```c
// ouverture du fichier en écriture
FILE * fd = fopen("mon_fichier", "w"); // ou a
// écrit une chaîne de caractères
fputs("du texte", fd);
// écrit un caractère
fputc('a', fd);
// écrit un entier
fprintf(fd, "%d", 38);
```

Entrées/sorties sur des fichiers

En mode binaire :

```
// Lit nb blocs de size octets depuis le fichier fd vers buffer.
int fread(char* buffer, int size, int nb, FILE* fd);
// Écrit nb blocs de size octets vers le fichier fd depuis buffer.
int fwrite(char* buffer, int size, int nb, FILE* fd);
```

La valeur retournée est le nombre de blocs effectivement transférés.

En mode texte :

On a parlé de `printf` plus tôt, pour écrire dans la sortie standard.

On peut écrire de la même manière dans un fichier avec la fonction `fprintf`

```
fprintf(fd, "Hello %s !", name);
fscanf(fd, "%d", &my_int);
```

Exercice

Écrivez un programme qui prend un nom de fichier en paramètre, lit ce fichier ligne par ligne et écrit chaque ligne sur la sortie standard. (C'est ce que fait la commande linux `cat`).