Outils

Informatiques

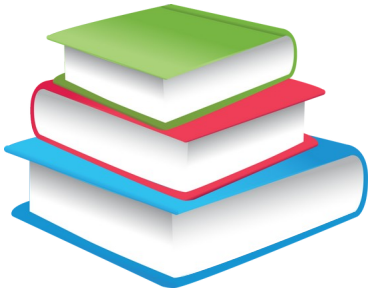
ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

P Lefebvre - 2022

L'École des INGÉNIEURS Scientifiques

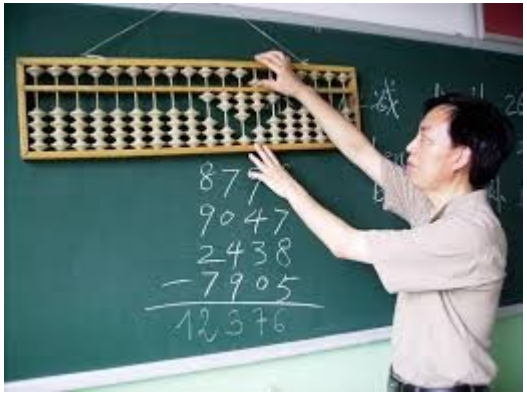
Bibliographie

- « Langage C », de *Brian Kernighan et Dennis Ritchie*
- « Introduction à l’algorithmique », de Cormen, Leiserson, Rivest
- openclassrooms.com : « Apprenez à programmer en C !”



plan
28h + 2h examen

- Architecture d’un ordinateur (2h)
  - codage des nombres entiers, des flottants, des caractères
  - architecture de Von Neuman
- système d’exploitation et multi-tâche (2h)
- Langage C (14h)
  - Types de données
  - Instructions de contrôle
  - La gestion de la mémoire
  - Gestion des fichiers et des entrées sorties
- Réseau (4h)

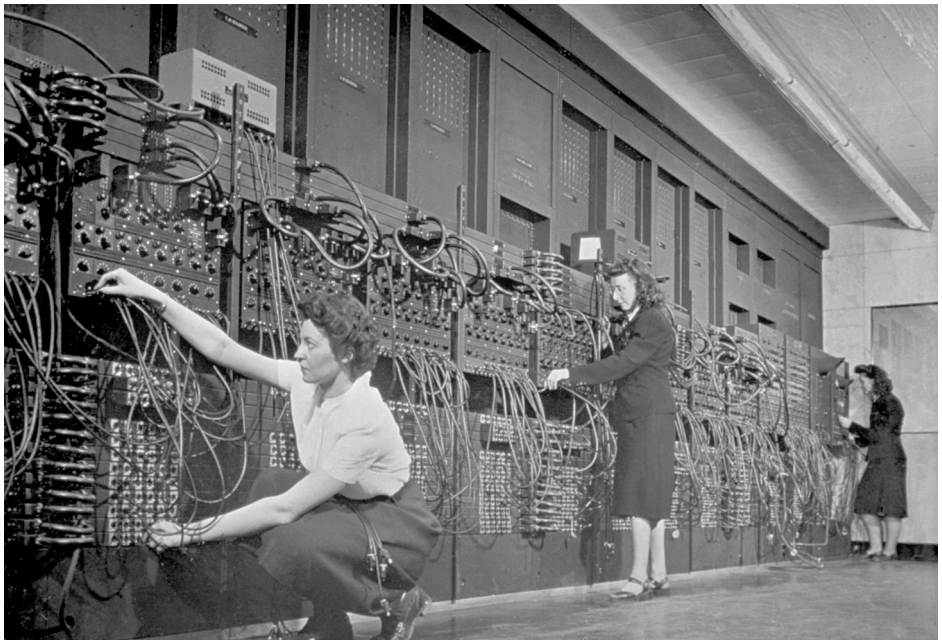


(pré)Histoire

- Ve siècle av J-C, la table de Salamine, abaque babylonienne servant à la conversion des monnaies
- au XIIIe siècle boulier Chinois.
- 1642, la Pascaline, inventé par Blaise Pascal pour aider son père, nommé surintendant de Haute Normandie.
- 1673 Leibniz, machine mécanique pouvant effectuer les 4 opérations
- 1854 création de l'algèbre binaire par Boole.
- 1896 création de Tabulating Machine Corporation qui deviendra IBM.
- 1937 Alan Turing, test de calculabilité : la machine de Turing.
- 1938 Shannon définit le bit (Binary digit)

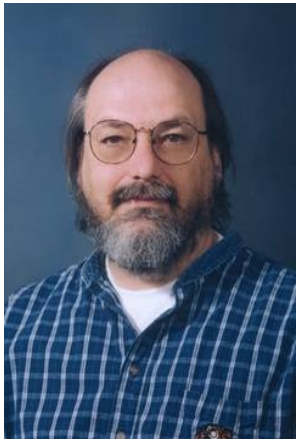
Histoire (suite)

- 1941, Konrad Zuse fabrique le premier ordinateur (Z3) électromécanique facilement programmable par ruban perforé.
- 1945 Von Neumann met au point son architecture, qui a été longtemps la référence pour la conception des processeurs.
- 1947 Bardeen, Walter et Brattain fabriquent le transistor
- 1950 ENIAC, premier ordinateur entièrement électrique… mais difficilement programmable. Jean Jennings Bartik (photo) fut la première à dompter la bête.
- 1950 Invention de l'assembleur à Cambridge par Wilkes
- 1957 Création du Fortran, premier langage de haut niveau par John Backus d'IBM.



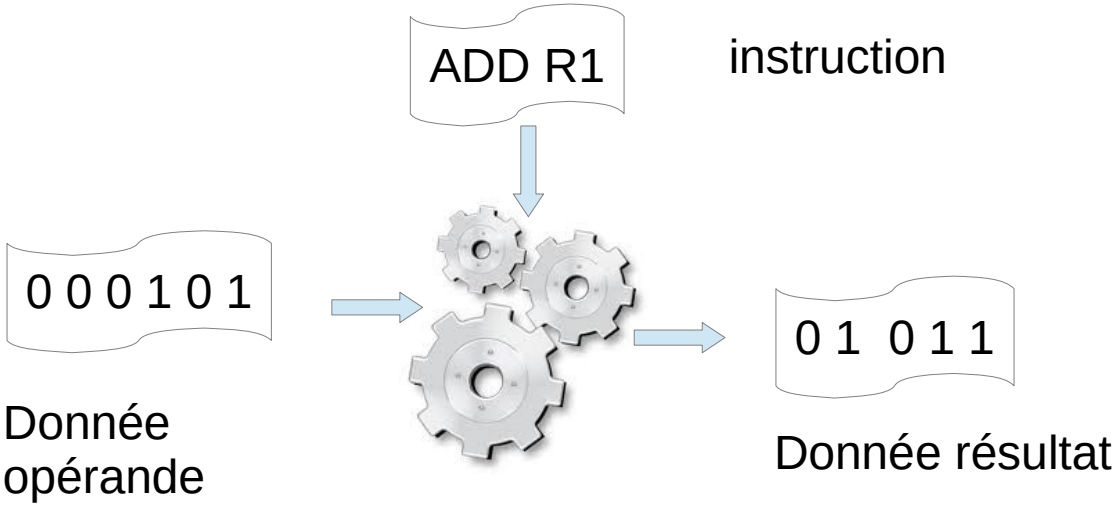
Histoire (suite)

- Années 70, multi-utilisateur, réseau.
  AT&T Labs avec Ken Thomson et Dennis Ritchie crée **Unix**
- Années 80 : début des ordinateurs individuels
  Tanenbaum crée un microkernel Unix appelé **Minix** pour ses étudiants
  Apple 2 conçu par Steve Wozniak
- Années 90 : PC, interactivité, plug & play...
  Windows
  Linus Thorvald, inspiré par Minix crée le noyau **Linux** en 1991
  GNU/Linux = système d'exploitation complet intégrant des outils du projet **GNU** (compilateur, IHM, éditeurs, shell...) initié par Richard Stallman
- Années 2000, **OS embarqués** :
  smartphones, tablettes, voitures, box ADSL...
  Android...



Fonctionnement d'une machine

Ordinateur = machine qui traite des données à l'aide d'un programme et qui produit des résultats.



La moulinette est le plus souvent **séquentielle**, c'est à dire que les instructions sont traitées l'une après l'autre en fonction d'une horloge.

Fonctionnement d'une machine

Exemple d'instruction en langage haut niveau :

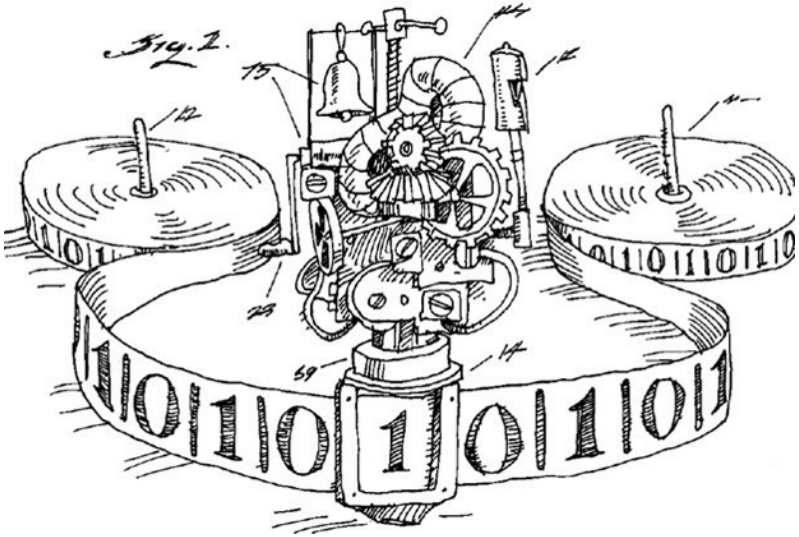
```
i = i + 1 ;
```

traduite en langage de bas niveau

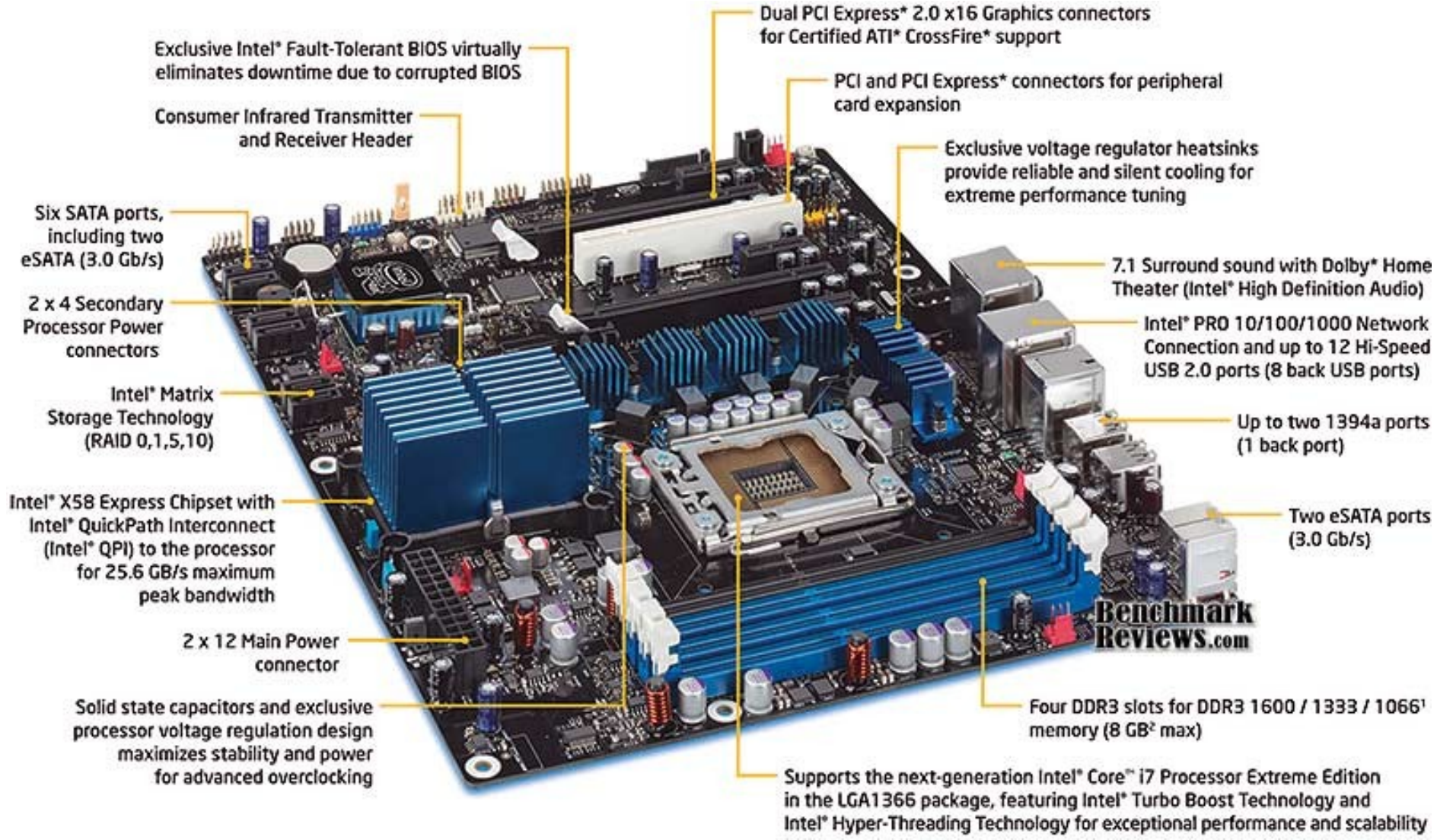
```
mov 0x00004000, Reg
inc Reg
mov Reg, 0x00004000
```

traduite en binaire

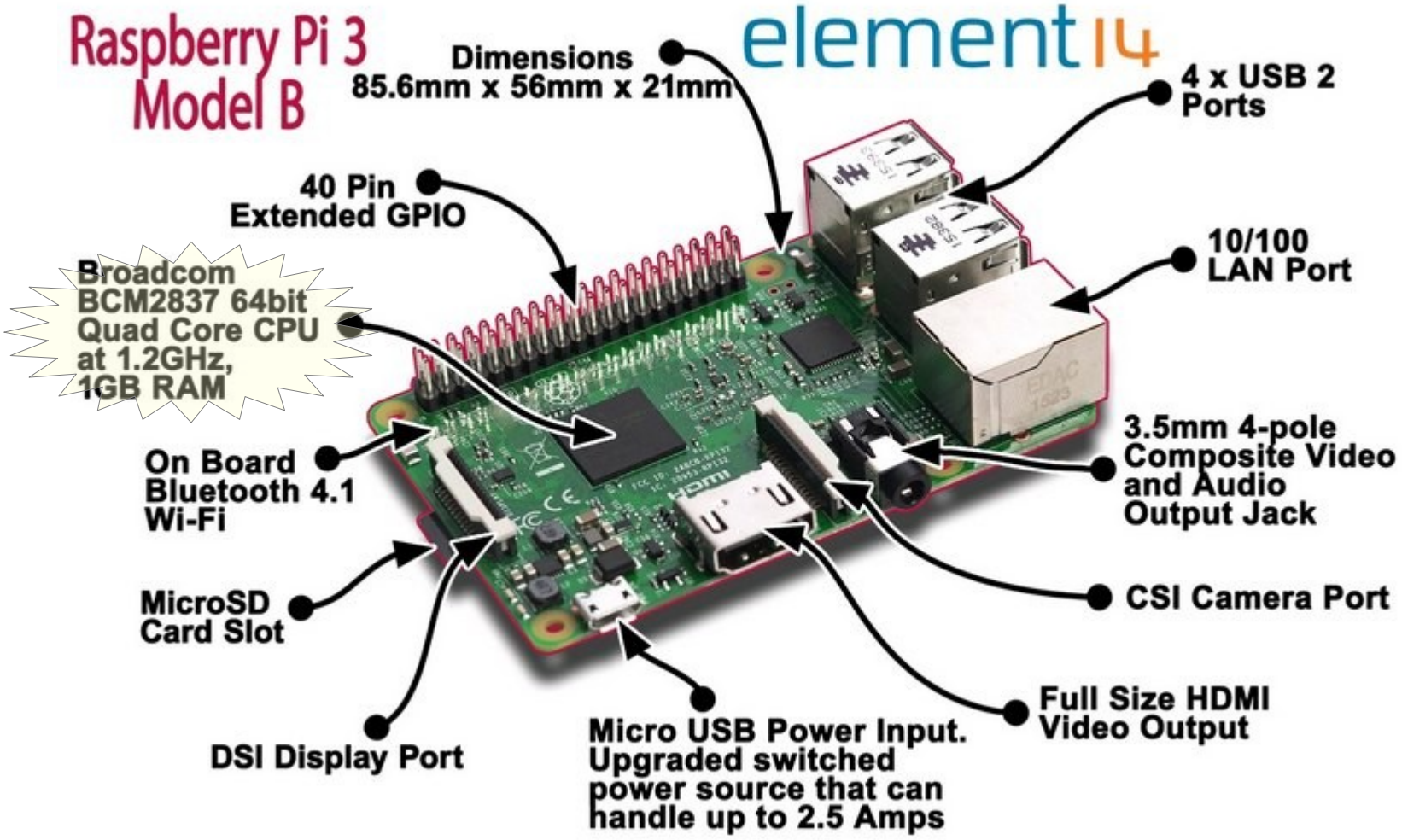
```
00000020
00040000
00000010
00000021
00040000
```



Fonctionnement d'une machine

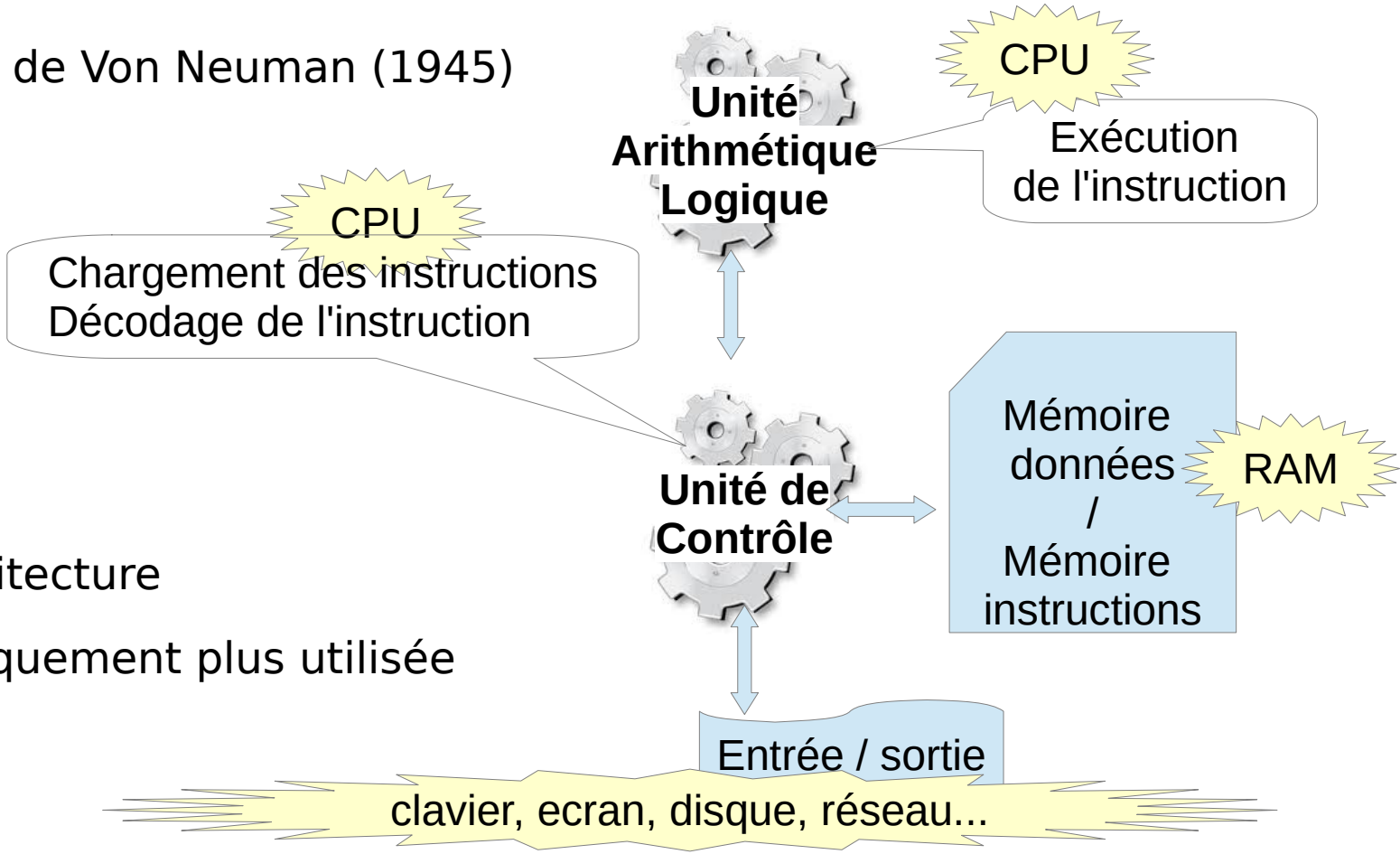


Fonctionnement d'une machine



Fonctionnement d'une machine

Ordinateur de Von Neuman (1945)



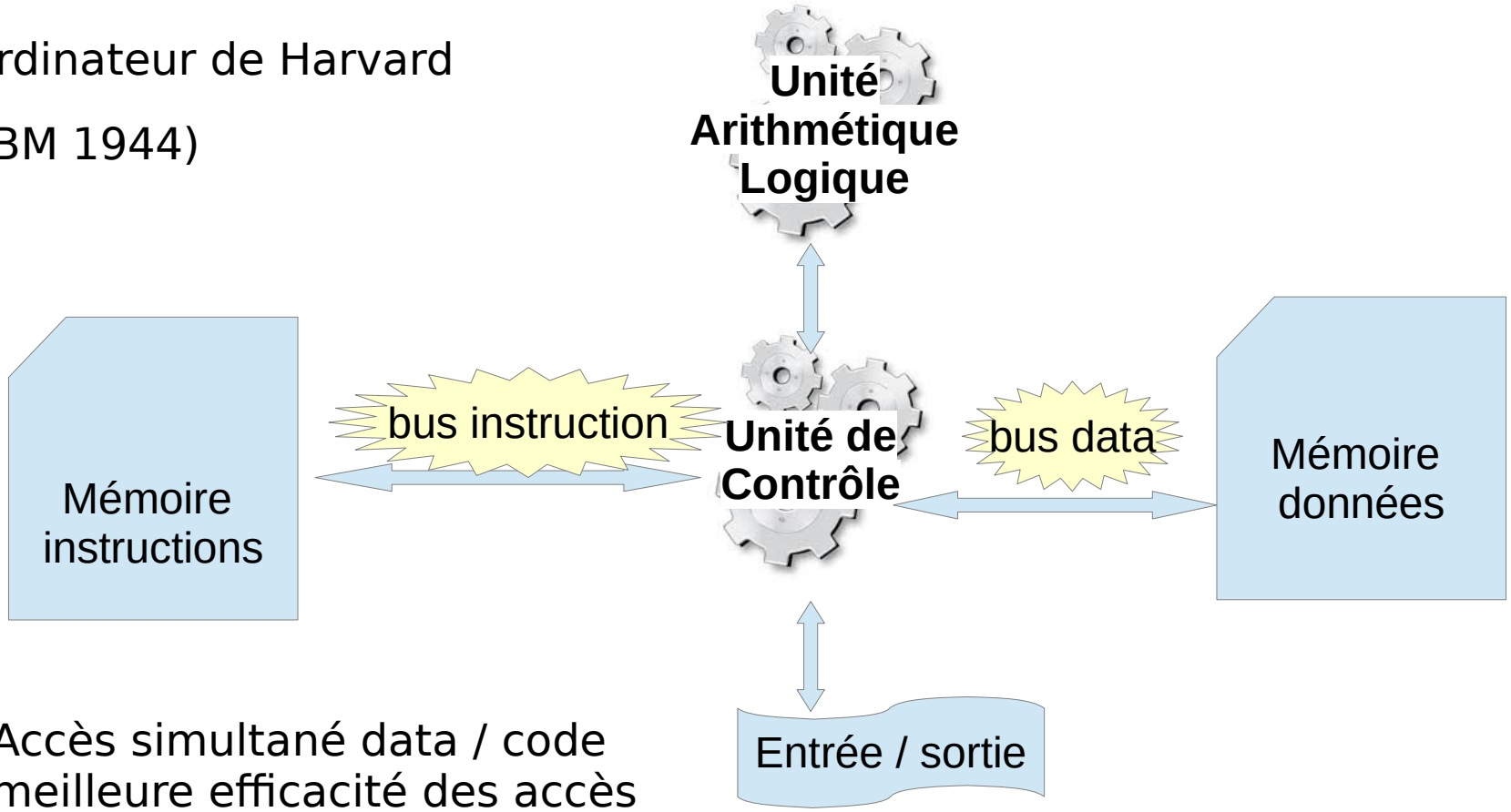
Cette Architecture

n’est pratiquement plus utilisée

Fonctionnement d'une machine

Ordinateur de Harvard

(IBM 1944)



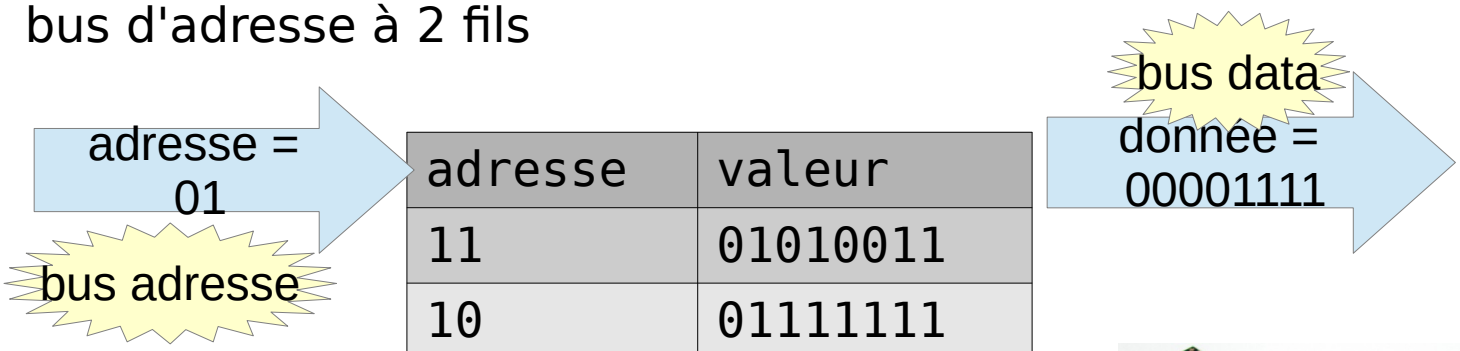
- Accès simultané data / code
- meilleure efficacité des accès

Fonctionnement d'une machine : la mémoire

- La mémoire est un ensemble de cases pouvant stocker 8 bits, soit 1 octet.
- Chaque case comporte un numéro permettant l'accès à son contenu. c'est **l'adresse** de la case.
- L'accès à la mémoire par le processeur se fait par un ensemble de fils, appelé **bus**. Il y a un bus d'adresse et un bus de données.
- Sur un système 64 bits, le bus d'adresse comporte 64 fils.
- La capacité d'une mémoire se compte en puissance de 2.
  2^{10} octets = 1024 octets = 1 kio.
  2^{20} = 1Mio , 2^{30} = 1Gio, 2^{40} = 1Tio
  Notez le « i » pour les différencier des unités du système international.

Fonctionnement d'une machine, la mémoire

- Exemple avec un bus d'adresse à 2 fils



adresse	valeur
11	01010011
10	01111111
01	00001111
00	01011111

- Combien faut il de fils pour adresser 4 Gio de mémoire ?

Les limites actuelles des machines

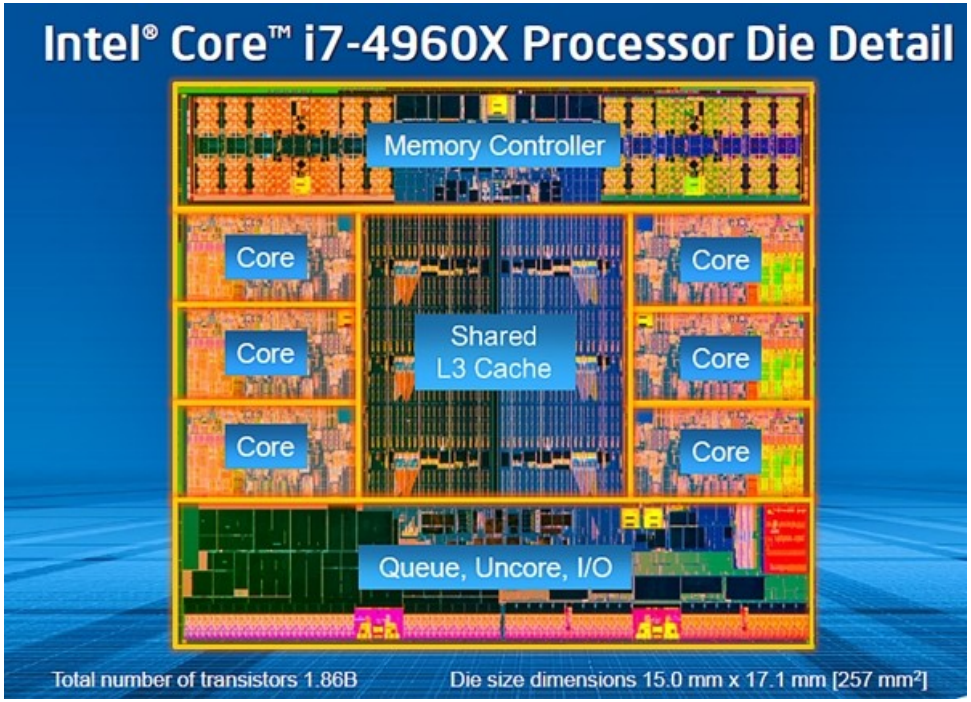
Maille du silicium :0,54 nm
En technologie 10 nm : 20 mailles de Si forment la grille du transistor*

** pour 2018*

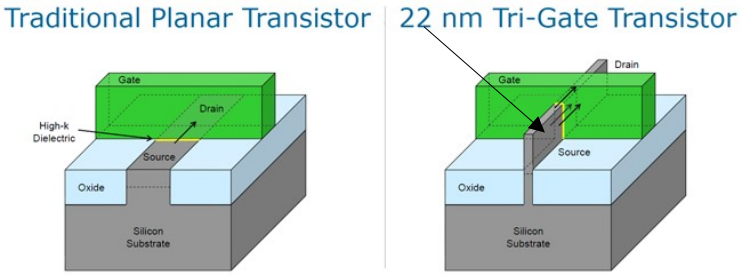
Fréquence 4,7 GHz → ~ 1 instruction / 0,2 ns
temps A/R d'un bout à l'autre dans le i7
$t=d/c = 2x0,02/3.10^8 = 0,13$ ns
le temps A/R « mange tout le temps de cycle »

→ paralléliser les architectures
… mais aussi réduire la consommation.
… et sécuriser les échanges de données.

Autre avenir, l'ordinateur quantique



consommation : 95 W

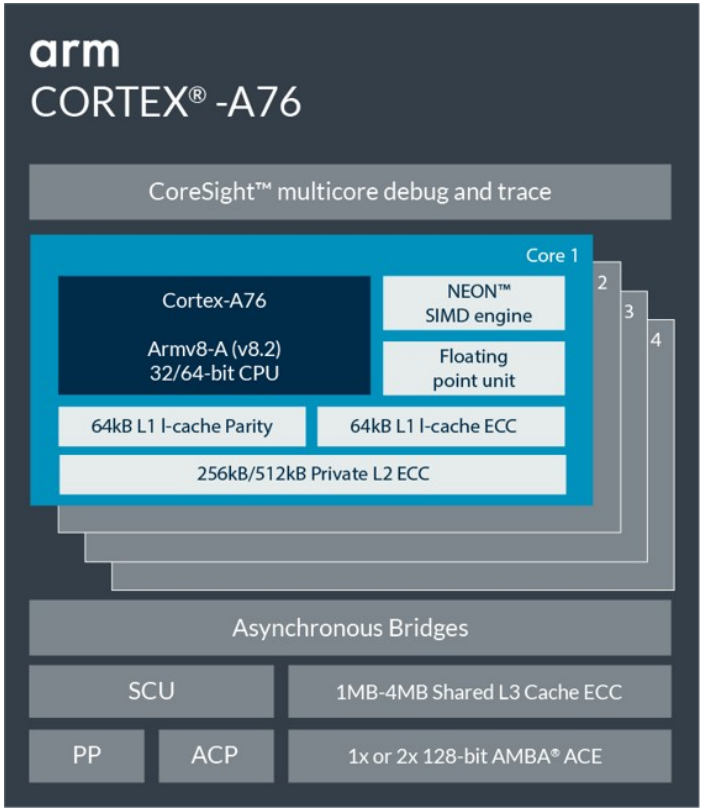


Hiérarchie mémoire

Mémoire Cache : La mémoire principale (RAM DDR) est trop lente pour le processeur. Il faut donc une mémoire plus proche du CPU (mémoire cache) qui lui permet de travailler efficacement.

Exemple chez Arm

Cache Level 1 : 2 par cœur :
data et instructions séparés
Cache Level 2 : un par cœur
Cache Level 3 : un par processeur.



Hiérarchie mémoire

Plus la mémoire est loin du CPU,

plus elle est de grande capacité… mais moins elle est rapide !

Exemple i7 Haswell:

	taille	Latence (en cycle)
register	64 bits	1
Data cache L1	32 KB	4 si simple accès
Instruction cache L1	32 KB	idem
L2 cache	256 KB	12
L3 cache	8 MB	36
RAM	512 GB	36*(nb_mots) + 57 (en ns)

Il faut gérer le « va et vient » entre la mémoire de travail (registre, cache) et la mémoire principale (RAM).

→ c’est le CPU et l’OS qui gèrent cela.

Fonctionnement d'une machine : le CPU

- A chaque coup d'horloge (ou presque) le CPU
  - va chercher une instruction en mémoire grâce au registre « Instruction Pointer » qui connaît l'adresse de la prochaine instruction à exécuter.
  - décode l'instruction pour savoir à quel circuit donner l'instruction pour exécution (unité de calcul, unité de chargement mémoire, unité d'entrée/sortie…)
  - exécute l'instruction (calcul ou écriture en mémoire)
  - écrit le résultat dans le registre destination si besoin
- Ces étapes sont faites par des **unités fonctionnelles différentes :**
  - Charger, Decoder, Execution, Write Back
- Ces unités peuvent travailler en **parallèle** *(illustration ci-après)*
  - *On a donc un pipeline d'instructions qui augmente les performances du CPU. Il exécute ainsi 4 instructions en même temps.*

Fonctionnement d'une machine : le CPU

Mémoire

instruction 1
instruction 2
instruction 3
instruction 4

CPU

charger l' instruction	**décoder** l'instruction	**exécuter** l'instruction	**écrire** le résultat dans un registre
instruction 4	instruction 3	instruction 2	instruction 1

coups d'horloge

unité fonctionnelle	*T0*	*T1*	*T2*	*T3*	*T4*	*T5*	*T6*
chargeur	*Ins 1*	*Ins 2*	*Ins 3*	*Ins 4*	*Ins 5*	*Ins 6*	*Ins 7*
décodeur		*Ins 1*	*Ins 2*	*Ins 3*	*Ins 4*	*Ins 5*	*Ins 6*
Exécution			*Ins 1*	*Ins 2*	*Ins 3*	*Ins 4*	*Ins 5*
Ecriture				*Ins 1*	*Ins 2*	*Ins 3*	*Ins 4*

L’ordonnancement des tâches

Le processeurs peut exécuter plusieurs tâches en parallèle :

- car il a plusieurs cœurs
- mais il peut simuler du parallélisme en effectuant des “morceaux de tâche” les unes après les autres.

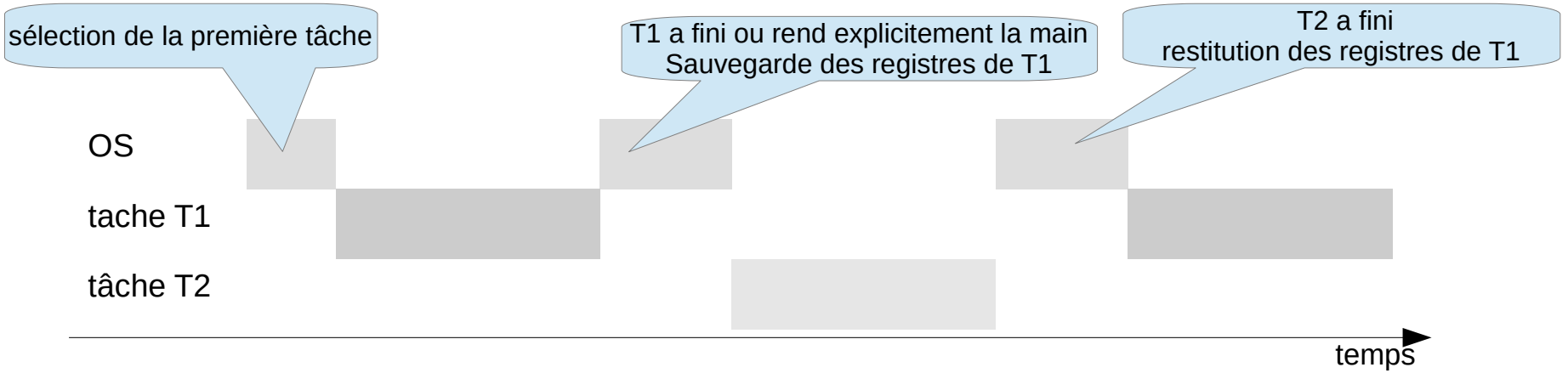
Le système d’exploitation organise le séquencement des tâches en fonction

- des priorités de tâches
- des ressources disponibles

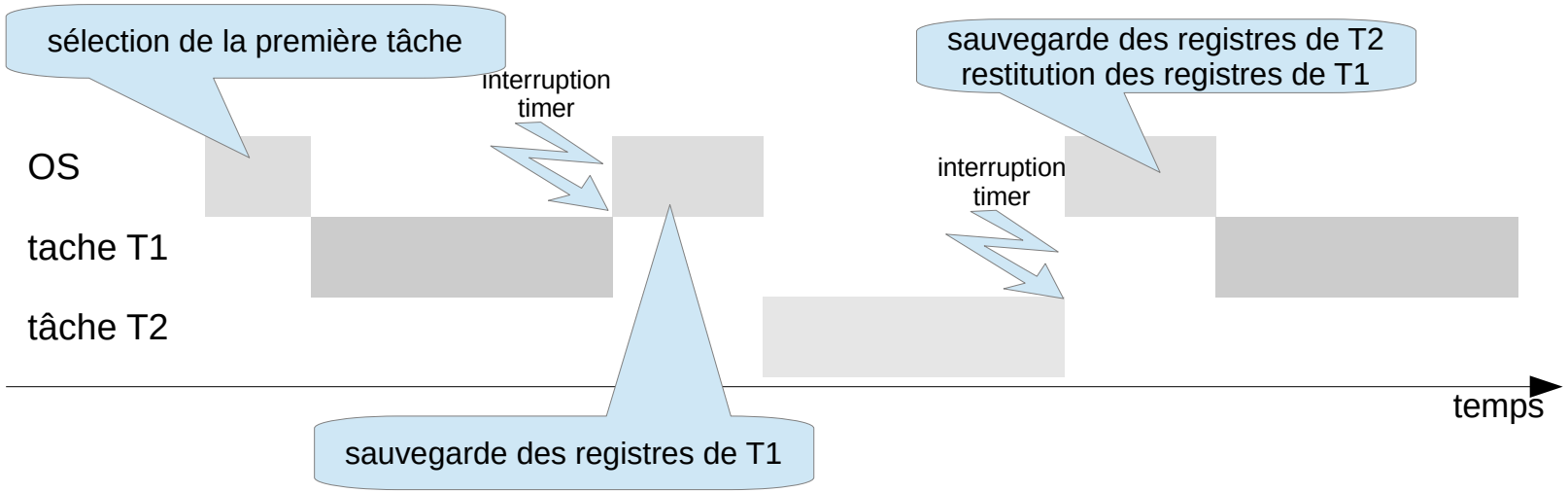
- Lorsque le CPU passe d’une tâche à une autre on appelle cela la **commutation de contexte.**

L'ordonnancement des tâches

ordonnancement coopératif



ordonnancement préemptif



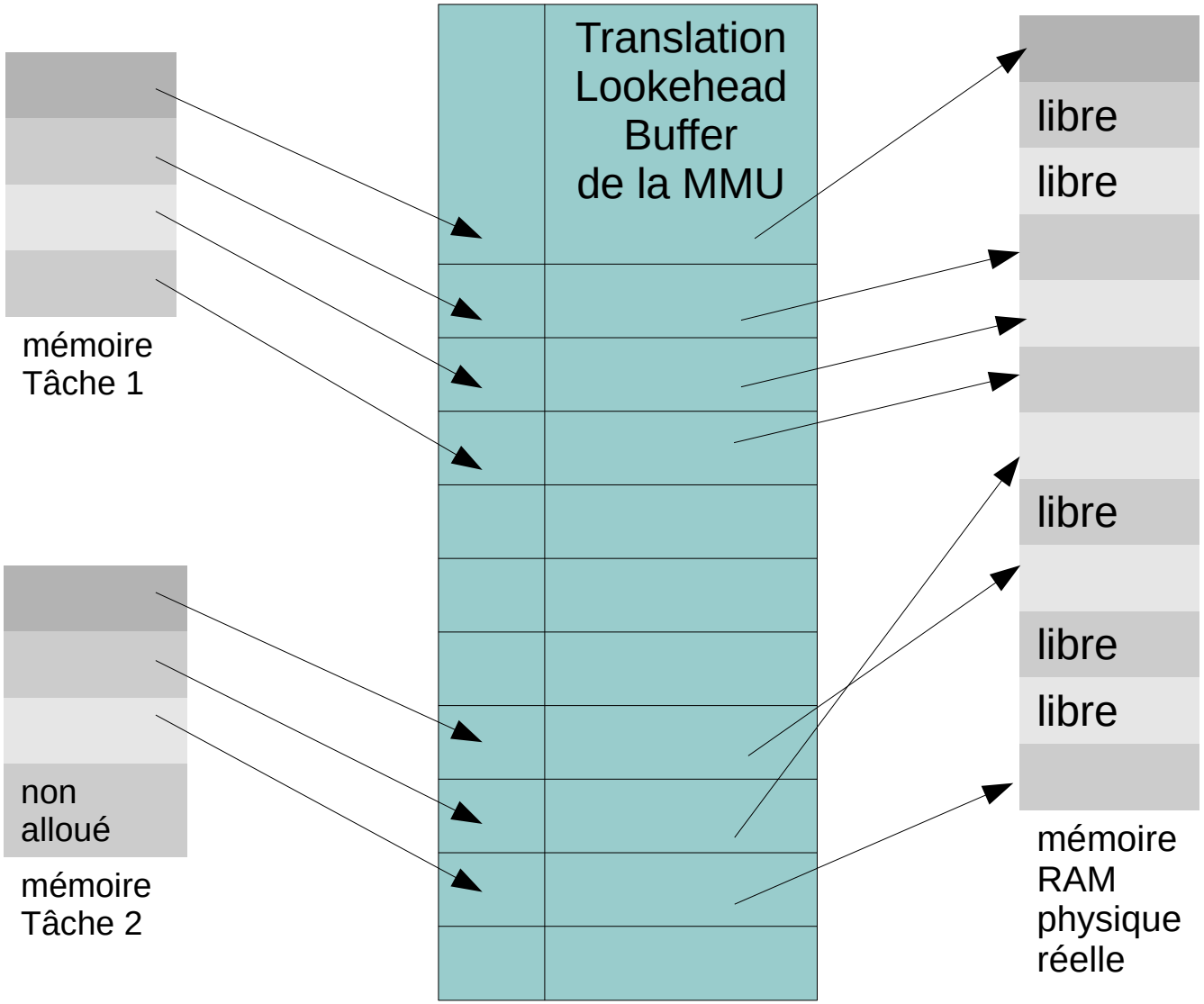
L’ordonnancement des tâches

Le CPU possède une unité fonctionnelle particulière appelée MMU : Memory management Unit

- garanti qu’**une tâche ne peut accéder aux données d’une autre** (il existe des failles comme furent Meltdown et Spectre)
- facilite la gestion mémoire en offrant aux programmes un **espace mémoire plan virtuel**.
- La mémoire est découpée en **segments** (par exemple 2 Mo) et une table de correspondance en mémoire (cf. planche suivante) est gérée par la MMU.
- L’OS fonctionne en mode privilégié (appelé **mode kernel**) et peut accéder à la totalité de la mémoire ainsi qu’aux entrées/sorties
- Les applications (tâches) fonctionnent en **mode user** et ne peuvent accéder aux E/S qu’en faisant des appels systèmes, c’est à dire en appelant des fonctions du systèmes d’exploitation
- Une tâche qui accède à un segment mémoire non alloué est arrêtée : le fameux “access violation” de Windows !

Mais alors comment faire communiquer les tâches entre elles ? → fonctionnalités de l’OS tels que les segments de mémoire partagée, les tubes, les sémaphores...

Virtualisation de la mémoire



Swap mémoire

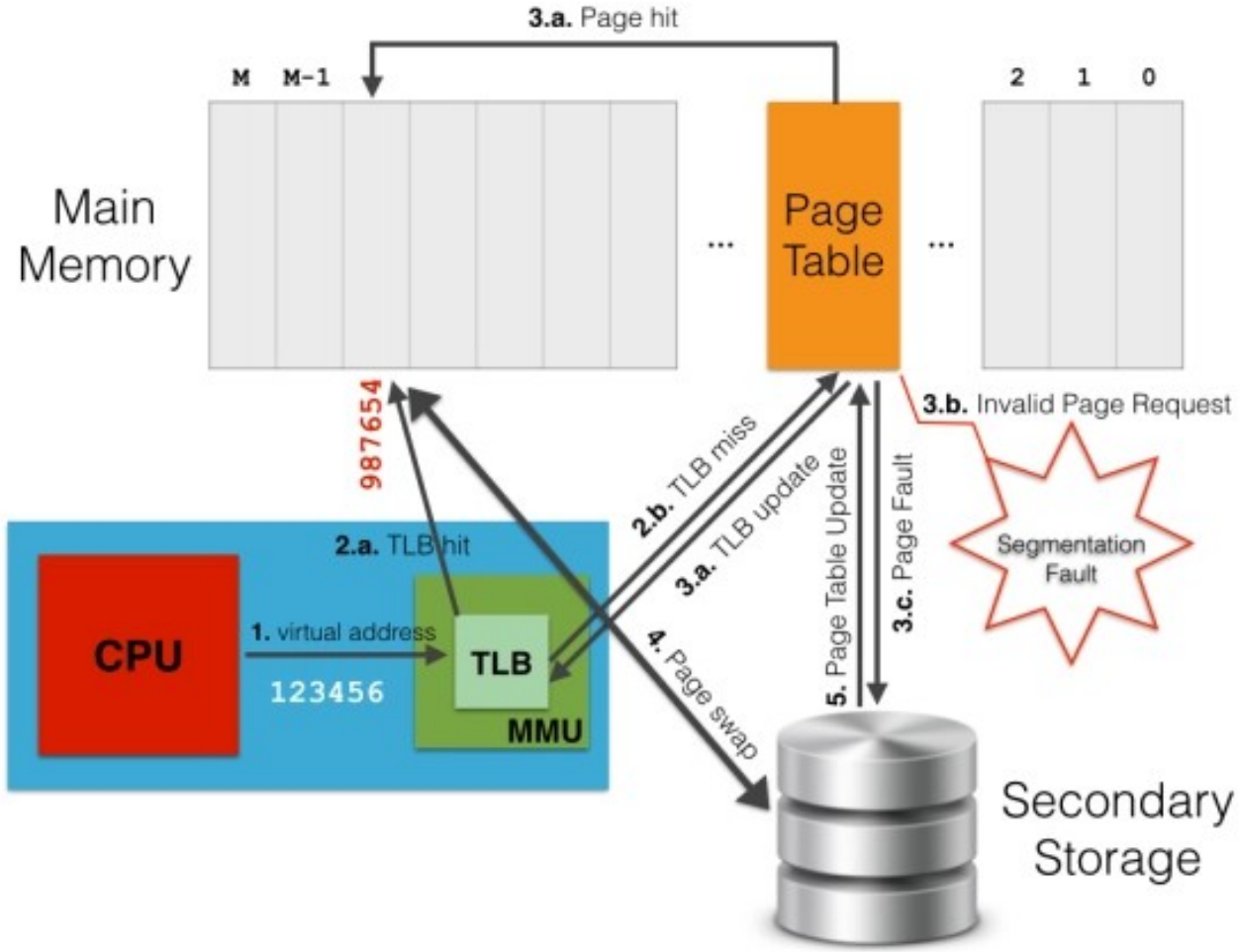
Lorsque la RAM ne suffit plus aux applications, l'OS peut décharger la mémoire utilisée mais non utile à l'instant sur le disque dur. Ce mécanisme s'appelle le **swap.**

Lorsque **l'OS réveille une application** (clic de souris…) mais que ses segments mémoires sont sur le disque, il faut remettre en RAM les segments mémoires.

windows : fichier swap « pagefile.sys ». L'emplacement de ce fichier doit être sur le disque le plus rapide car il impacte grandement les performances de la machine.

Linux : partition particulière appelée « partition swap ». même remarque.

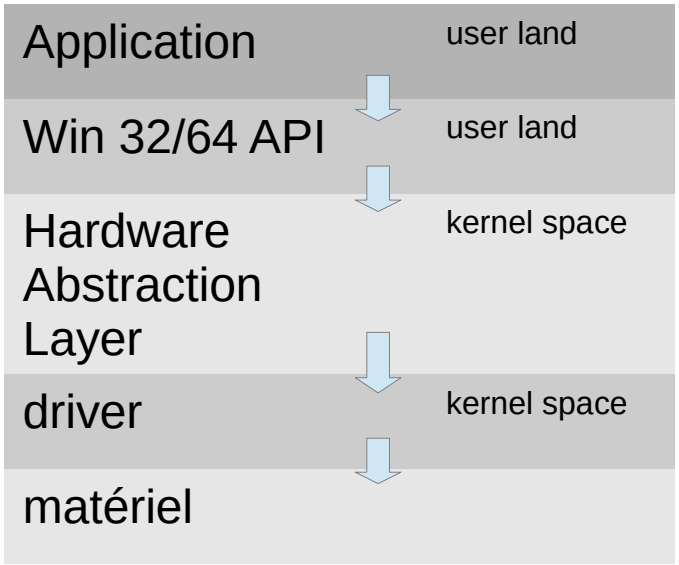
Virtualisation et swap mémoire



gabrieletolomei.wordpress.com

Les appels systèmes

- L'OS propose aux développeurs des bibliothèques de fonctions permettant de gérer la machine.
  Win32 / Win64 API pour windows et glibc pour Linux.
- Offre une interface unifiée aux programmes quelque soit le matériel → couche d'abstraction matériel offerte aux programmeurs.
- Le fabricant d'un matériel offre un programme (qui peut être certifié) appelé pilote qui permet de gérer le matériel.
  Il doit se conformer à la couche HAL.
- Facilite la gestion mémoire en offrant aux programmes un espace mémoire plan virtuel.
- Augmente la robustesse des systèmes puisque ces appels systèmes ont été maintes fois testés.



Fonctionnement d'une machine : le CPU

Les registres généraux des processeurs INTEL.

Un registre est une mémoire locale au CPU, et utilisé pour des instructions particulières. Sa taille dépend du CPU. Sur un processeur 64 bits, le registre fait aussi 64 bits.

Nom 64 bits	
RAX	accumulateur
RBX, RCX, RDX	registres généraux
RBP	Base Pointer : pointe le début de la pile au moment de l'entrée dans une fonction
RSP	Stack Pointer : pointeur de pile
RIP	Instruction Pointer : pointe sur l'instruction en cours d'exécution.

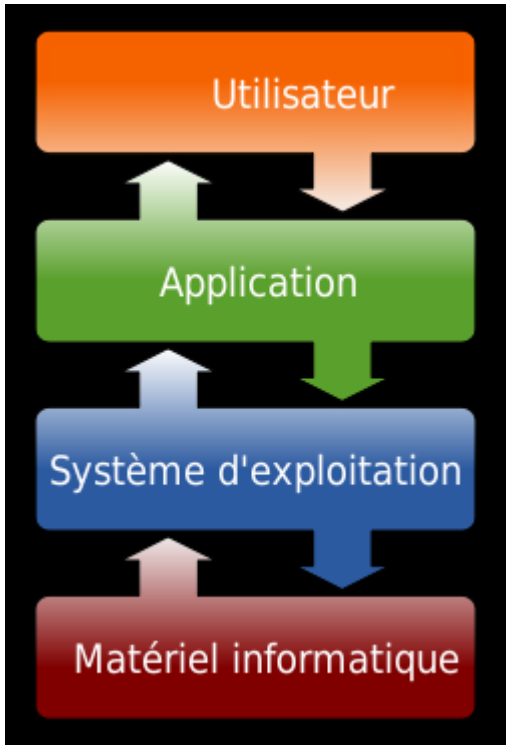
Fonctionnement d'une machine

Le système d'exploitation

Fournit des fonctions pour exploiter le **matériel**, comme la **mémoire** (allocation dynamique, cache...), les **entrées sorties** (écran, clavier, réseaux, souris, disques durs...), le **multi-tâche**...

Peut fournir également une **interface utilisateur** pour gérer la machine. Cette interface peut être graphique (Android, Mac OSX, Window's) ou textuelle, comprenant un **interpréteur de commande**.

Dialogue Application / Système d'exploitation souvent fait au travers de **bibliothèques de fonctions** écrites en langage C !

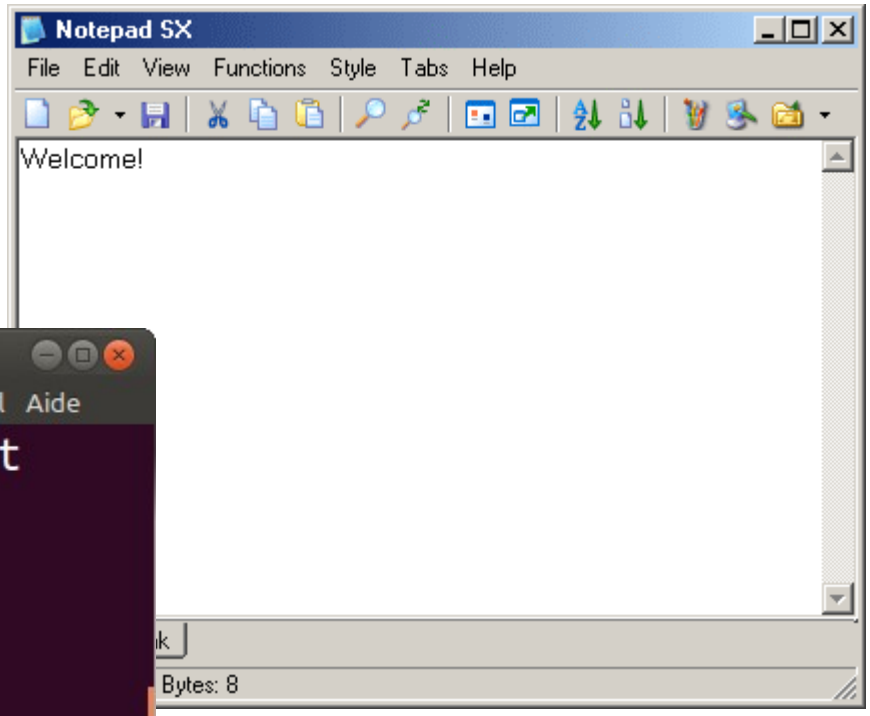


* image de Golftheman

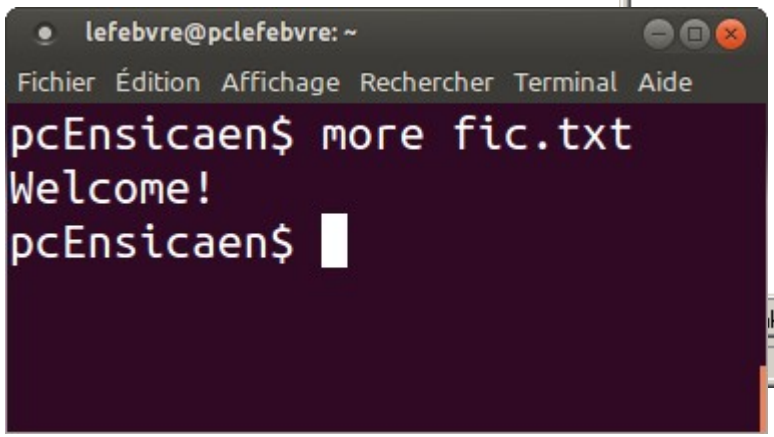
Fonctionnement d'une machine

Exemple : Ouverture d'un fichier

- Je double clic...



- en Shell



- en C

```
File *fp ;
fp = fopen ("fic.txt", "r") ;
```

Fonctionnement d'une machine

La compilation

Traduction nécessaire entre le programme écrit en langage compréhensible par l'homme et le **langage machine**, compréhensible par le processeur.

Traduction au cours de l'exécution = **langages interprétés** : PHP, Shell, Excel basic, Java*...

+ indépendant de la plateforme
- vitesse d'exécution

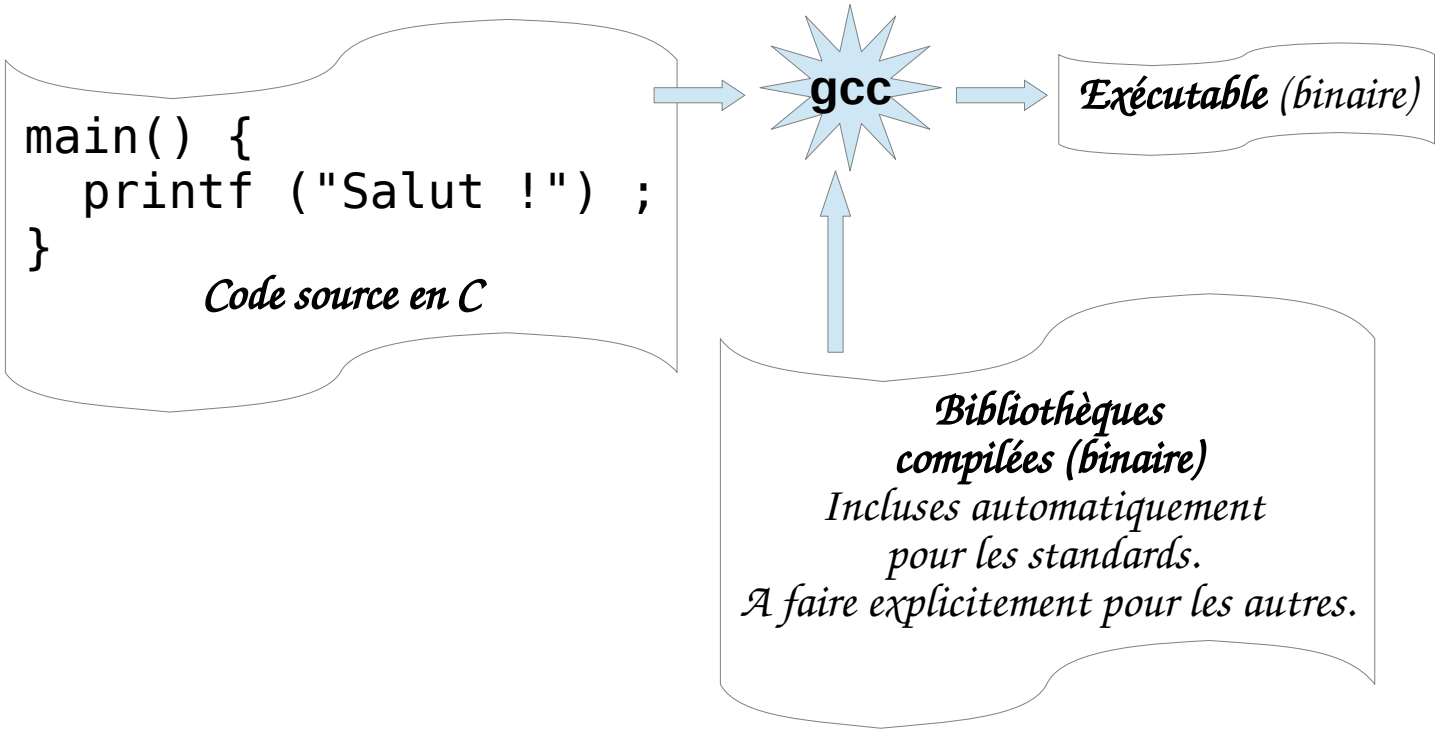
Traduction avant l'exécution : = **langages compilés** : C, C++, ADA...

+ vitesse d'exécution
- dépendant de la plateforme

* traduction intermédiaire en « byte code »

Fonctionnement d'une machine

La chaîne de compilation :

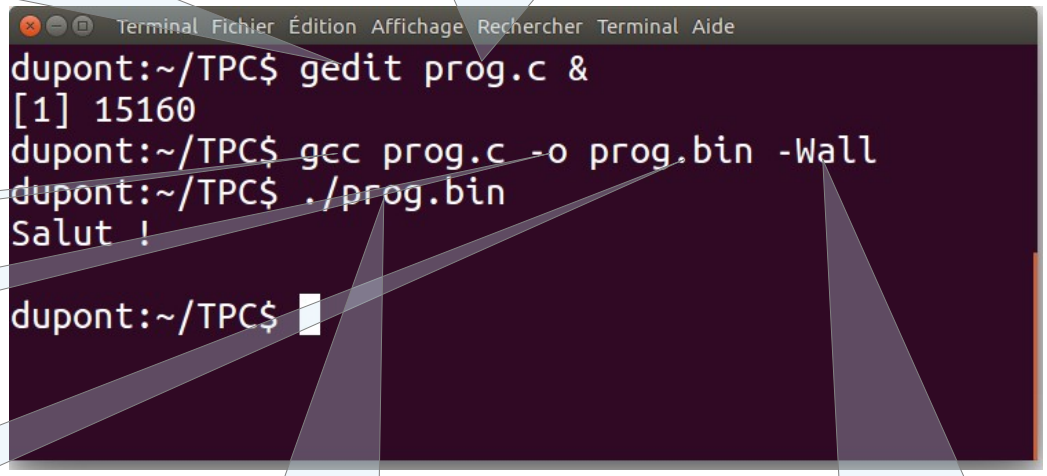


Fonctionnement d'une machine

La chaîne de compilation – exemple sous Linux :

Nom de l'éditeur de fichier :
gedit (il existe aussi sublime, geany...)

Nom du fichier source écrit en C



Nom du compilateur

Option du compilateur :
Nom du fichier binaire

Nom du fichier binaire

Exécution du programme

Option du compilateur :
Afficher tous les avertissements

Types de données

Codage des entiers

- pour les char, short, int, long : binaire complément à 2.
  Exemple :
  $(26)_{10} = (0001\ 1010)_2 = (1A)_{16}$
  $26 = 0x2^7 + 0x2^6 + 0x2^5 + 1x2^4 + 1x2^3 + 0x2^2 + 1x2^1 + 0x2^0 = 1x16^1 + 10x16^0$

- Le complément à 2 s'obtient en faisant l'inversion bit à bit du code naturel et en ajoutant 1. Du coup, **le bit de gauche représente le signe**.

  $(26)_{10} = (0001\ 1010)_2 \rightarrow inv \rightarrow (1110\ 0101)_2 \rightarrow +1 \rightarrow (1110\ 0110)_2 = (-26)_{10} = (E6)_{16}$
  $(-1)_{10} = (1111\ 1111)_2 = (FF)_{16}$
  $(-2)_{10} = (1111\ 1110)_2 = (FE)_{16}$

- Si le type est précédé de **unsigned,** le type est non signé. C'est important pour les comparaisons car :

  $(FF)_{16} < 1$ pour les nombres signés
  $(FF)_{16} > 1$ pour les nombres non signés

Types de données

Codage des nombres flottants 32 bits

pour les float : norme IEEE 754 sur 32 bits

- **Le signe** : bit de gauche (le plus significatif) : vaut 1 si <0
- **L'exposant** : les 8 bits suivants.
- **La mantisse** : les 23 suivants. Le bit le plus significatif vaut 0,5. celui qui suit a le poids 0,25, celui d'après 0,125...

Le nombre obtenu se calcule comme suit :

$$nb = (-1)^{signe} \times 2^{exposant-127} \times (mantisse+1)$$

- Exemple : $(-0,75)_{10}$ se code 10111111010000000000000000000000

  car $-0,75 = -1,5 \times 2^{-1}$

  mantisse = 0,5 et *exposant* = 126

Types de données

Codage des nombres flottants de type “double”

- Le format suit le même principe mais sur 64 bits (11 bits d'exposant et 52 bits de mantisse)

- Le chiffre le moins significatif représente :
  - $2^{-52} = 2{,}22 \cdot 10^{-16}$ pour les doubles, soit 15 chiffres significatif en base 10
  - $2^{-23} = 1{,}19 \cdot 10^{-7}$ pour les floats, soit 6 chiffres significatifs en base 10

- Le nombre le plus grand en valeur absolue est :
  - $2^{1024} = 1{,}8 \cdot 10^{308}$ pour les doubles
  - $2^{128} = 3{,}4 \cdot 10^{38}$ pour les floats

Types de données

Codage des caractères

- Comme la mémoire ne contient que des nombres, il faut une table, dite **table ASCII** (American Standard Code for Information Interchange) faisant correspondre les caractères usuels avec les valeurs comprises entre 0 et 127.
- **Unicode**

  Est une norme internationale qui recense les caractères d'un très grand nombre de langues incluant chinois, japonais, arabe : 1,114,112 caractères

  http://www.unicode.org/charts/ pour une liste complète.

Types de données

Table ASCII valeurs.

Les valeurs > 7F (en hexadécimal) ne sont pas définies dans la table ASCII.

hexa	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Types de données

Codage des caractères

- iso-8859-1 (Latin1) code sur 8 bits les caractères occidentaux. La table ASCII occupe les 128 premiers codes.
- iso-8859-15 (Latin15) idem à Latin1 mais code l' « € » et l' »œ ». Mais il en existe de nombreux autres sur 8bits (comme windows-1256 pour l'arabe...)
- utf-8 code sur un nombre variable d'octets les caractères. Tous les caractères unicodes sont encodables.
  - Ce sont les bits de poids fort du premier octet qui précise la longueur du code.
  - `0xxx xxxx` est compatible avec l'ASCII
  - `110x xxxx 10xxxxxx` encode presque l'ensemble des langues occidentales
  - `1110 xxxx 10xxxxxx 10xxxxxx` encode tout l'Unicode historique (appelé Basic Multilingual Plane)
  - `11110xxx 10xxxxxx 10xxxxxx 10xxxxxx` encode le reste de l'Unicode.

  utf-16 et utf-32 reprennent le principe de l'utf-8 en plus efficace, mais sont incompatibles avec la table ASCII.

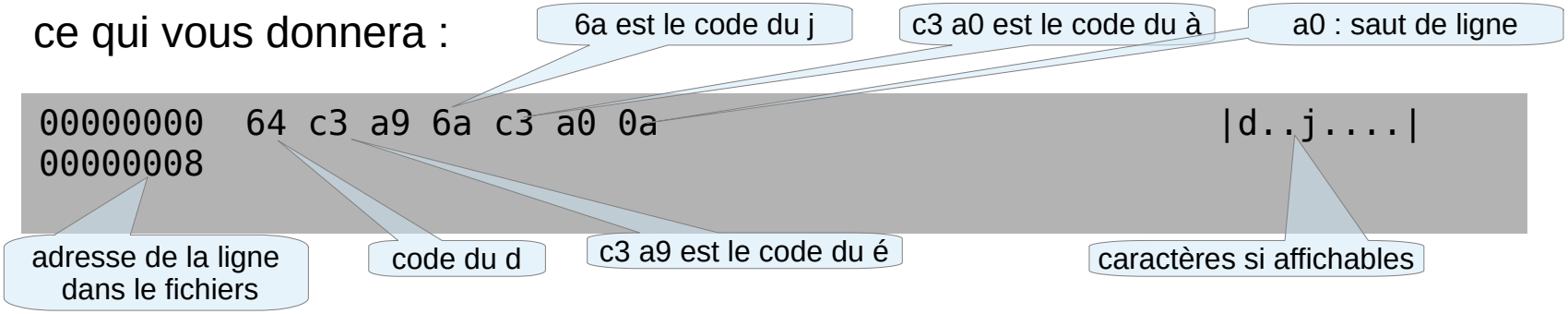
Encodage des caractères

Exercice

Avec gedit saisissez le mot « déjà » dans un fichier que vous nommerez « deja_utf8.txt » en l'enregistrant en choisissant « encodage utf-8 ».
Dans une console, taper

```
hexdump -C deja_utf8.txt
```

ce qui vous donnera :



Faites la même chose en enregistrant cette fois-ci votre fichier en choisissant « encodage iso8859-15 ». Analyser le fichier de nouveau avec hexdump.

Quel est la taille du fichier ? pourquoi ?

Le langage C

Historique

- Langage inventé par **Denis Ritchie** pour aider **Ken Thomson** à développer UNIX en 1971.
- C'est un langage **procédural**, comme le langage machine dont il est très proche : les instructions sont exécutées de manière séquentielle
- Il comprend entre autres :
  - des variables
  - des instructions de contrôle (test)
  - des instructions de boucle

- Les variables sont **typées**.
  - Permet au compilateur de **repérer plus facilement les erreurs** de programmation.
  - Permet également de **modifier le comportement des opérateurs** en fonction du type des données.
- Le C standard, s'appelle le C ANSI C89. Divers adaptations sont nées pour rendre plus de services aux programmeurs.
- La plus connue est C99. Elle est moins portable que la norme ANSI C89.
- La dernière version est C11.

Format

Les identifiants.

Ils sont utilisés pour nommer les variables, les fonctions, les nouveaux types. Seuls les caractères : [a-z] [A-Z] [0-9] et _ sont autorisés. Les identifiants ne peuvent pas commencer par un chiffre.

Les mots réservés :

Types : void char short int long float double
Modificateurs : auto extern register static signed unsigned const volatile
Constructeurs : enum struct typedef union
Boucles : do for while
Conditions : case default else if switch
Branchements : break continue goto return
Autres : asm entry fortran sizeof

Mise en page

- Les lignes contenant des instructions se terminent par un point virgule.
- Un programme est un **ensemble de fonctions**.
- Les fonctions sont structurées de cette manière en C ANSI :
  - un **prototype** définissant les paramètres d'entrée et ceux de sortie.
  - une **accolade ouvrante** {
  - la **déclaration des variables ;**
  - des **instructions ;**
  - une **accolade fermante** }
- La fonction main() est la première fonction exécutée.
- Les commentaires sont encadrés par /* et */ ou commencent par // et se terminent à la fin de la ligne

Mise en page

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

int main () {
    int a = 0 ;
    int A = 3 ;
    printf ("a=%d A=%d \n", a, A+A ) ;
    return (EXIT_SUCCESS) ;
}
```

Inclusion entête de la bibliothèque pour printf

Inclusion entête de la bibliothèque pour EXIT_SUCCESS

On peut omettre les paramètres du main

Attention à la casse !

Une tabulation devant chaque instruction à l'intérieur d'un bloc : 4 espaces recommandés

Alignement de l'accolade

produit :

```
a=0 A=6
```

Types de données

Les types primitifs sur gcc Linux Intel

- **char** : 8 bits, de -128 à + 127
- **short** : 16 bits, de -32768 à +32767
- **int** : 32bits, de -2 147 483 648 à +2 147 483 647
- **long** : 64 bits, de -9 223 372 036 854 775 808 à 9 223 372 036 854 775 807
- **float** : 32 bits, flottant à la norme IEEE 754 (max = 1e38), précision sur 6 chiffres significatifs
- **double** : 64 bits, flottant à la norme IEEE 754 (max = 1e308), précision sur 15 chiffres significatifs
- A partir de C99, il existe le **__int128** et le **long double** sur 128 bits. Ils ne sont pas supportés par toutes les architectures. Par exemple, Intel n'utilise que 80 bits sur les 128 du long double.

Types de données

La norme C99 de définit en fait qu'une taille minimum :

- **char** : minimum 8 bits
- **short** : minimum 16 bits
- **int** : minimum 16 bits
- **long** : minimum 32 bits
- **long** : minimum 64 bits

Types de données

Formats des nombres (par l'exemple)

langage C	signification
12	entier de valeur 12
12.3	Double 12,3 (64 bits)
1e3	1000
1.4e-2	0,014
012	(12) en octal donc 10 en décimal
0x1A	(1A) en hexadécimal, donc 26 en décimal
2L	Entier long (64 bits)
3UL	Entier non signé long
4.5F	Flottant (32 bits)
6.7L	Long double (128 bits)

Types de données

Codage des caractères

- On l'a déjà vu, comme la mémoire ne contient que des nombres, il faut une table de correspondance dite **table ASCII.**
- Le type char est utilisé pour stocker les caractères. Les valeurs au-delà de 127 sont utilisées pour coder les caractères particuliers à la langue installée sur le système. L'encodage utilisé en France pour coder ces caractères est l'**iso-8859-15** (Latin15).
- On notera que le type char ne peut stocker que les 128 premiers caractères de l'UTF-8
- on obtient le code en entourant le caractère par des simples quotes ' .

```
char c1 = 65 ;

char c2 = 'A' ;
```

ici, c1 et c2 contiennent la même valeur car 65 est le code ASCII de A.

- Les caractères spéciaux comme la tabulation sont précédés de « back slash \ » suivi d'une lettre. Par exemple, '\t' est une tabulation.

Types de données

extraits de la table ASCII étendue 8859-15

code en décimal	caractère	observation
0	'\0'	caractère NULL
9	'\t'	tabulation
10	'\n'	new line
32	' '	espace
34	'\"'	guillemet ou double quote
39	'\''	quote
48	'0'	
49	'1'	
65	'A'	
66	'B'	
97	'a'	
98	'b'	
92	'\\'	back slash
164	'€'	

Fonction d'affichage : printf

Version simplifiée...pour débuter !

`printf (format, parametres...)`

format est une chaîne de caractères qui sera affichée. Elle peut contenir des caractères spéciaux permettant d'insérer dans la chaîne les valeurs des paramètres.

%d décodé comme un entier en base 10

%ld décodé comme un entier long en base 10
%X décodé comme un entier hexadécimal

%o décodé comme un entier en octal
%f décodé comme un flottant

%lf décodé comme un flottant long
%e flottant en notation ingénieur
%c affiche le caractère dont le code ASCII est contenu dans le paramètre
%s chaîne de caractères

Elle dépend du système d'exploitation. Son code est donc dans une bibliothèque.

Fonction d'affichage : printf

Exemple

```
#include <stdio.h>
int main (void) {
  int a = 65535 ;
  char c = 65 ;
  float x = -0.75 ;
  printf ("a=%d, a=%X, c=%d, c=%c, x=%f\n", a, a, c, c, x);
}
```

compilation puis exécution

```
$ gcc essaiPrintf.c
$ ./a.out
a=65535, a=FFFF, c=65, c=A, x=-0.750000
```

Notez le `#include <stdio.h>` *qui permet au compilateur de connaître le prototype du printf de la bibliothèque du système.*
La vérification par le compilateur du prototype des fonctions utilisées permet de diminuer les erreurs de programmation.

Fonction d'affichage
source http://www.cplusplus.com/reference

A *format specifier* follows this prototype: [see compatibility note below]
`%[flags][width][.precision][length]specifier`

Where the *specifier character* at the end is the most significant component, since it defines the type and the interpretation of its corresponding argument:

specifier	**Output**	**Example**
`d` *or* `i`	Signed decimal integer	`392`
`u`	Unsigned decimal integer	`7235`
`o`	Unsigned octal	`610`
`x`	Unsigned hexadecimal integer	`7fa`
`X`	Unsigned hexadecimal integer (uppercase)	`7FA`
`f`	Decimal floating point, lowercase	`392.65`
`F`	Decimal floating point, uppercase	`392.65`
`e`	Scientific notation (mantissa/exponent), lowercase	`3.9265e+2`
`E`	Scientific notation (mantissa/exponent), uppercase	`3.9265E+2`
`g`	Use the shortest representation: `%e` or `%f`	`392.65`
`G`	Use the shortest representation: `%E` or `%F`	`392.65`
`a`	Hexadecimal floating point, lowercase	`-0xc.90fep-2`
`A`	Hexadecimal floating point, uppercase	`-0XC.90FEP-2`
`c`	Character	`a`
`s`	String of characters	`sample`
`p`	Pointer address	`b8000000`
`n`	Nothing printed. The corresponding argument must be a pointer to a `signed int`. The number of characters written so far is stored in the pointed location.	
`%`	A `%` followed by another `%` character will write a single `%` to the stream.	`%`

Fonction d'affichage
source http://www.cplusplus.com/reference

The *format specifier* can also contain sub-specifiers: *flags*, *width*, *.precision* and *modifiers* (in that order), which are optional and follow these specifications:

flags	**description**
-	Left-justify within the given field width; Right justification is the default (see *width* sub-specifier).
+	Forces to preceed the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceeded with 0, 0x or 0X respectively for values different than zero. Used with a, A, e, E, f, F, g or G it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see *width* sub-specifier).

width	**description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The *width* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	**description**
.number	For integer specifiers (d, i, o, u, x, X): *precision* specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A *precision* of 0 means that no character is written for the value 0. For a, A, e, E, f and F specifiers: this is the number of digits to be printed **after** the decimal point (by default, this is 6). For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for *precision*, 0 is assumed.
.*	The *precision* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

Fonction d'affichage
source http://www.cplusplus.com/reference

The *length* sub-specifier modifies the length of the data type. This is a chart showing the types used to interpret the corresponding arguments with and without *length* specifier (if a different type is used, the proper type promotion or conversion is performed, if allowed):

	specifiers						
length	**d i**	**u o x X**	**f F e E g G a A**	**c**	**s**	**p**	**n**
(none)	int	unsigned int	double	int	char*	void*	int*
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wchar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

Note regarding the c specifier: it takes an int (or wint_t) as argument, but performs the proper conversion to a char value (or a wchar_t) before formatting it for output.

Note: Yellow rows indicate specifiers and sub-specifiers introduced by C99. See <cinttypes> for the specifiers for extended types.

Fonction d'affichage : printf

Exemple

```
#include <stdio.h>
int main (void) {
  double x=1.23456789012345678e23 ;
  double y=1.234e4 ;

  printf ("x=%f\n", x);
  printf ("x=%.3f\n", x);
  printf ("x=%E\n", x);
  printf ("x=%.12e\n", x);
  printf ("x=%14e\n", x);
  printf ("y=%14e\n", y);
  printf ("x=%g\n", x);
  printf ("y=%g\n", y);
}
```

produit

```
x=123456789012345685803008.000000
x=123456789012345685803008.000
x=1.234568E+23
x=1.234567890123e+23
x=  1.234568e+23
y=  1.234000e+04
x=1.23457e+23
y=12340
```

Notez la précision fantaisiste !

3 chiffres après la virgule

Augmentation de la précision standard

justification à droite sur 14 caractères

Choix automatique de l'affichage le plus compact

Fonction lecture clavier : scanf

Version simplifiée...pour débuter !

```
int scanf (format, parametres...)
```

- ***format*** est une chaîne de caractères qui permet de dire au compilateur comment interpréter les donnés entrées par l'utilisateur et quel codage utiliser. La chaîne de caractères peut comprendre les mêmes caractères spéciaux que printf : **%d** ,**%X, %f, %c, %s**
- **paramètres** est une liste d'adresses des zones mémoire où la machine doit stocker ce qui a été lu.

- On obtient l'adresse d'une variable en faisant précéder son nom par **&**.

- **retourne**, par un entier, le nombre d’occurrences lues.

Fonction scanf

Exemple

```
#include <stdio.h>
int main (void) {
    int a, n ; char c; float x;
    printf ("Entrez un nombre entier: ");
    scanf ("%d", &a);
    scanf ("%c", &c); // bizarrerie
    printf ("Entrez un caractère suivi d'un nombre flottant
séparés par un espace: ");
    n=scanf ("%c %f", &c, &x);
    printf ("a=%d, c=%c, x=%f, n=%d\n", a, c, x, n);
}
```

produit

Entrez un nombre entier: 46
Entrez un caractère suivi d'un nombre flottant séparés par un espace: v 1e4
a=46, c=v, x=10000.000000, n=2

Notez le « `scanf ("%c", &c)` *» qui doit absorber le caractère invisible « retour chariot » dû à l'appui sur la touche entrée... les ennuis commencent !*

Fonction scanf

Pourquoi doit-on passer l'adresse des variables à scanf ?

Car le nom de la variable représente sa valeur.
et si le nom est précédé de & cela représente l'adresse de la variable

Par exemple

```
int a ;
```

déclaration de la variable

```
a=7 ;
```

valeur = 7

```
printf ("%d", a);
```

printf affiche la valeur

```
scanf ("%d", &a);
```

scanf a besoin non pas de la valeur, mais de de l'endroit où se trouve la variable

Structures de contrôle

Si alors sinon

syntaxe :

```
if (condition) {
    instruction 1 ; // exécutées si condition est vraie
    instruction 2 ;
    ...
}
```

```
if (condition) {
    instruction 1 ; // exécutées si condition est vraie
    instruction 2 ;
    ...
}
else {
    instruction 1 ; // exécutées si condition est fausse
    instruction 2 ;
    ...
}
```

Structures de contrôle

Si - alors - sinon

condition est un entier interprété comme
- VRAI si *condition* ≠ 0
- FAUX si *condition* = 0

condition peut être calculé par des opérateurs logiques
- a > b retourne 1 si a est **supérieur** à b, et zéro sinon
- a < b retourne 1 si a est **inférieur** à b, et zéro sinon
- a >= b retourne 1 si a est **supérieur ou égal** à b, et zéro sinon
- a <= b retourne 1 si a est **inférieur ou égal** à b, et zéro sinon
- a == b retourne 1 si a est **égal** à b , et zéro sinon
- a != b retourne 1 si a est **différent de** b , et zéro sinon

Conjonction / disjonction / négation
- condition1 **||** condition2 veut dire condition1 **OU** condition2
- condition1 **&&** condition2 veut dire condition1 **ET** condition2
- **!(**condition) inverse la condition

– Lors de l'évaluation, **!** est prioritaire sur **&&** lui même prioritaire sur **||** ; *mais mieux vaut utiliser les parenthèses pour plus de clarté.*
– L'évaluation de l'expression se fait de la gauche vers la droite.

Structures de contrôle

Que produisent ces programmes ?

```
#include <stdio.h>
int main (void) {
    int a=3, b=2 ;
    if ((a==3) && ( b>=2))
        printf ("Vrai\n") ;
    else
        printf ("Faux\n") ;
}
```

```
#include <stdio.h>
int main (void) {
    int a=3 ;
    if (a=2)
        printf ("Vrai\n") ;
    else
        printf ("Faux\n") ;
}
```

```
#include <stdio.h>
int main (void) {
    int a=5 ;
    if (1<a<3) {
        printf ("Vrai\n") ;
    }
    else
        printf ("Faux\n") ;
}
```

```
#include <stdio.h>
int main (void) {
    int a=0, b=7 ;
    if (!a) {
        if ( b )
           printf ("Vrai\n") ;
        else
           printf ("Faux\n") ;
    }
}
```

Structures de contrôle

switch

syntaxe :

```
switch (variable) {
    case valeur1 : instruction1 ;
                   instruction2. . . ;
                   break ;
    case valeur2 : instructions ;
                   break ;
    ...
    default : instructions ;
}
```

remarques

*Ne pas oublier le **break** sinon les instructions d'après sont exécutées également indépendamment de la valeur de la variable.*

Structures de contrôle

Que produit ce programme ?

```
#include <stdio.h>
int main (void) {
    char rep ;
    printf ("Aimez-vous ce cours ?\n") ;
    scanf ("%c", &rep) ;
    switch (rep) {
        case 'n' :
        case 'N' : printf ("Courage ! ") ;
                   printf ("L'année va être longue... ") ;
                   break ;
        case 'o' :
        case 'O' : printf ("Vil flatteur ? ") ;
                   break ;
        default : printf ("Je n'ai pas compris la réponse") ;
    }
}
```

Les opérateurs

opérateur	signification
*	multiplication
/	division
+	addition
-	soustraction
<< n	décalage à gauche de n bits. Remplissage avec des 0
>> n	décalage à droite de n bits. Remplissage avec des 0
%	modulo
var++	exécute l'instruction puis incrémente la variable *var (idem avec --)*
++*var*	incrémente la variable *var puis* exécute l'instruction *(idem avec --)*
&	opération ET bit à bit
|	opération OU bit à bit
~	inversion bit à bit
^	opération XOR (OU exclusif)
a += 2	equivalent à « a=a+2 ». *idem avec les autres opérateurs*
v = t ? b : c	v sera égal à b si t est vrai et sera égal à c sinon
,	Virgule : Sépare deux expressions. Évalué de gauche à droite.

Les opérateurs

Exemples

expression	résultat
`5%3`	
`(3==2) || (3==3)`	
`i=3 ; printf("%d", i++);`	
`printf("%d", ++i);`	
`0x0D & 0x06`	
`0x0D | 0x06`	
`0x0F << 1`	
`~0x2F`	
`r=((3+2)==5) ? 8 : 9 ;`	
`r /= 2 ;`	
`b = ((a = 7), (a + 10));`	

Les opérateurs

Priorité (la plus grande priorité d'abord)

( ) [ ] -> .
! ~ ++ -- -(unaire) (type) *(indirection) &(adresse) sizeof
* / %
+ -(binaire)
<< >>
< <= > >=
== !=
&(et bit-à-bit)
^
|
&&
||
? :
= += -= *= /= %= &= ^= |= <<= >>=
,

Les opérateurs

Exemple

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

int main( )
{
        int a = 020 | 001 ;
        printf("a = %d\n", a) ;

        int i = (7%3==1) ? 8 : 6 ;
        i <<=  2 ;
        printf("i = %d\n", i) ;
    return (EXIT_SUCCESS) ;
}
```

Affiche :

Boucles

Boucle « for »

syntaxe :

```
for (initialisation ; condition ; instructions_etape_suiv ) {
    instructions ;
    ...
}
```

Les *instructions* sont exécutées tant que la *condition* est vrai. A la fin d'un tour de boucle, les *instructions_etape_suiv* sont exécutées. Typiquement on met ici des instructions d'incrémentation.

Exemple le programme suivant calcule factoriel(n), et donc affiche « n=24 » car il s'arrête à i=4.

```
#include <stdio.h>
int main (void) {
    int i, int n=1 ;
    for (i=1 ; i<5 ; i++)
        n=n*i ;
    printf ("n=%d\n", n) ;
}
```

Boucles

Boucle « while »

syntaxe :

```
while ( condition ) {
    instructions ;
    ...
}
```

Les *instructions* sont exécutées tant que la *condition* est vrai.

Exemple le programme suivant calcule factoriel(n), et donc affiche « n=24 » car il s'arrête à i=4.

```
#include <stdio.h>
int main (void)
{
    int i=1 ; int n=1 ;
    while (i<5) {
        n=n*i ;
        i = i+1 ;
    }
    printf ("n=%d\n", n) ;
}
```

Boucles

Boucle « do – While »

syntaxe :

```
 do {
    instructions ;
    ...
} while ( condition ) ;
```

Les *instructions* sont exécutées **au moins une fois**, puis tant que la *condition* est vrai.

Exemple le programme suivant calcule factoriel(n), et donc affiche « n=24 » car il s'arrête à i=4.

```
#include <stdio.h>
int main (void)
{
    int i=1 ; int n=1 ;
    do {
        n=n*i ;
        i = i+1 ;
    } while (i<5) ;
    printf ("n=%d\n", n) ;
}
```

Boucles

– L'instruction ***break*** permet de sortir d'une boucle for ou while.

– L'instruction ***continue*** permet d'aller à la fin et continuer une boucle for ou while.

Ce programme affiche « n=8 ». Pourquoi ?

```
#include <stdio.h>
int main (void)
{
    int i; int n=1 ;
    for (i=1 ; i<7 ; i++) {
       if (i==3) continue ;
       if (i==5) break ;
       n=n*i ;
    }
    printf ("n=%d\n", n) ;
}
```

Fonctions

Sous partie **indépendante** d'un programme :

– permet d'appeler **plusieurs fois** le sous programme

– évite les copiers/collés de code identique

– L'exécution peut être différente à chaque appel grâce aux paramètres

– Améliore grandement la **lisibilité** des programmes car permet de faire ressortir les grandes lignes de l'algorithme. Les détails sont alors écrits dans la fonction.

Fonctions

– **Déclaration d'une fonction** : Elle commence par la déclaration des types de paramètres et du type de la valeur retournée. Puis le code est inséré entre { }.

```
Type_retour NomFonction (Type1 param1, Type2 param2...) {
    code...
}
```

– Une fonction qui ne renvoie rien retourne un type **`void`**

– **Appel de la fonction** : Il se fait en utilisant le nom de la fonction et en remplaçant les paramètres par des valeurs.

– le mot clé **return** suivi d'une valeur permet de renvoyer la valeur calculée par la fonction. Sans valeur, il met simplement fin à la fonction.

– **Prototype** de la fonction : Il sert au compilateur à vérifier que les types utilisés au moment de l'appel sont corrects. Il doit être écrit avant le code de la fonction l'utilisant. Il n'est pas obligatoire si la déclaration est faite avant l'appel.

```
Type_retour NomFonction (Type1, Type2 ...) ;
```

Fonctions

Exemple

Qu'affiche ce programme ?

```
#include <stdio.h>
float module (float, float ) ;

main() {
    float re = 4.0, im = 3.0, ro ;
    ro = module (re, im) ;
    printf (" mod(%f, %f) = %f\n", re, im, ro) ;
    printf (" mod(3, 2) = %e\n", module (3, 2)) ;

}

float module (float x, float y) {
    float modu = x*x + y*y ;
    x = 32 ; // ne sert à rien, mais est expliqué plus tard
    return modu ;
}
```

Fonctions

```
$ ./prog.bin
mod(4.000000, 3.000000) = 25.000000
mod(3, 2) = 1.3000000e1
```

- On remarquera `#include <stdio.h>` qui inclut les prototypes des fonctions d'Entrée/sortie, dont celui de printf qui est utilisé ici.
- Si `module()` avait été défini avant le `main()` , la déclaration de son prototype n'était pas obligatoire.
- on remarquera l'**indentation du programme qui améliore la lisibilité**.
- « x=32 » montre que changer la valeur de « x » n'a pas d'effet sur la valeur de « re »

Fonctions récursives

- On appelle fonction **récursive**, une fonction qui s'appelle elle même (directement ou par le biais d'une autre fonction).
- La programmation de telles fonctions est élégante mais souvent peu efficace car la pile utilisée pour stocker les variables locales des différents appels imbriqués peut croître de manière inconsidérée.
- On remarquera que la programmation de telles fonctions obéit souvent à ce schéma :
  - renvoyer des valeurs pour des cas particuliers ;
  - sinon renvoyer une valeur dépendant d'un autre appel de la fonction.

Exemple, fonction factorielle :

```
int factorielle (int n) {
    if (n < 0)  return (-1) ; // erreur !
    if (n <=1 ) return (1) ;
    return (factorielle (n-1) * n );
}

int main (void) {
    printf (" factorielle(5) = %d\n", factorielle (5));
}
```

fonctions de la bibliothèque mathématique standard

Quelques fonctions utiles de la bibliothèque mathématique standard

La compilation d'un programme utilisants ces fonctions doit se faire avec l'option « -lm ».
Exemple : `gcc -lm prog.c -o prog.bin`

et inclure `<math.h>` en début de fichier.

ceil(x), floor(x), round(x)	arrondi par défaut, arrondi par excès, arrondi au plus proche. *ceil(-0.5)= -1 ; floor(-0.5)=0 ; round(-0.5)=-1*
cos(x), sin(x), tan(x)	cosinus, sinus, tangente *avec x en radian*
cosh(x), sinh(x), tanh(x)	cos, sin et tan hyperbolique,
exp(x), pow(x, y), log(x), sqrt(x)	exponentielle , x puissance y, logarithme, racine carrée
fabs(x)	valeur absolue

Par défaut, ces fonctions prennent des « doubles » en paramètre. Il existe en général des fonctions cousines manipulant des « floats » ou des « long double ». Par exemple :

```
double cosh(double x);
float coshf(float x);
long double coshl(long double x);
```

fonction de génération de nombres aléatoires

int rand(void) retourne un nombre aléatoire entier compris entre 0 et RAND_MAX. Cette constante vaut 2^{31}-1 avec le compilateur gcc utilisé à l'Ensi.

Les nombres aléatoires sont calculés à partir d'une suite dont le premier terme (appelé graine ou seed) est initialisé à 0. **Le premier appel à rand retournera toujours le même résultat !!**

Pour éviter cela, il faut initialiser la graine à une valeur différente à chaque exécution du programme.
On utilise pour cela la fonction ***void srand (unsigned int).***
Le paramètre passé à srand est la nouvelle valeur de graine.

Quelle valeur mettre ?

On peut utiliser la fonction ***time_t time (time_t *t)*** qui retourne le nombre de secondes écoulées depuis le 1er janvier 1970 si t vaut NULL.

time_t est une redéfinition du type **unsigned int**.

génération de nombres aléatoires

Exemple :

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

int main(void){
    int i ;
    srand( time( NULL ) ) ;
    for (i = 0 ; i< 5 ; i++ )
        printf ("%d ", rand()%1000) ;
}
```

produit :

```
146 293 636 356 570
```

Présentation - indentation

```
type fonction (type param1, param2)
{
    Type1 var1 ;
    instruction1 ;
    instruction2 ;
    if ( test1 ){
        instruction3 ;
    }
    do {
        instruction4 ;
        instruction5 ;
    } while (test2) ;
    instruction6 ;
}
```

un espace après la virgule

déclaration des variables après l'accolade ouvrante

accolade ouvrante ici ou là même si une seule instruction

un espace avant et après le point virgule

indentation fixe (4 caractères par exemple) à l'intérieur du bloc délimité par les accolades

accolade fermante alignée sur la ligne de l'accolade ouvrante

Autres bonnes habitudes :

- CONSTANTES en majuscules
- nomDeFonction : mots séparés par une majuscule
- NouveauType : commence par une Majuscule

https://google.github.io/styleguide/

transtypage

Opération de **changement de type** ou « cast »

syntaxe :

```
var1 = (Type_de_var1) var2 ;
```

Exemple :

```
float a = 7.2 ;
int b = (int) a ;
```

Dans cet exemple, « a » est tronqué pour être stocké dans « b ».

→ b vaut alors 7.

A manipuler avec soin cependant.

Adresses et pointeurs

– Les **variables sont stockées en mémoire** à une certaine adresse. *cf p8.*

– L'**adresse est une valeur** sur 16, 32, ou 64 bits, dépendant du système. Avec Linux, MacOS ou Windows, les adresses sont sur 64 bits.

– Elle représente le **numéro du premier octet** où commence le stockage de la variable.

– La valeur de l'adresse d'une variable est obtenue en précédant son nom par un **&**.

ex : **`int i ;`** Dans ce cas **`&i`** représente l'adresse de « i ».

– Une adresse peut être stockée dans une variable, appelée **pointeur**. Dans ce cas la variable a un type. Le type dépend du contenu sur lequel l'adresse pointe. La déclaration d'un pointeur est obtenue en faisant précédé la variable par une ***.**

– NULL est une constante qui vaut 0 et qui permet de représenter un pointeur qui pointe nulle part.

Adresses et pointeurs

Exemple :

```
int *pa ;
```

– Dans ce cas pa est une variable pouvant stocker des adresses de nombres entiers.

– On dit que « pa » est un pointeur d'entiers.

– On dit que « pa » est du « type int* »

– On peut connaître ou modifier la valeur sur laquelle un pointeur pointe en faisant précédé le pointeur par une *****.

* = la valeur pointée par

Exemple :

```
int *pa ;
int a = 2 ;
pa = &a ;
*pa = 3 ;
```

Dans ce cas la nouvelle valeur de « a » est 3.

Adresses

```
int main (void) {
    int b = 7;
    int *pb ;
    pb = &b ;
    printf ("b=%d\n", b) ;
    printf ("pb=%p\n", pb) ;
    *pb = 8 ;
    printf ("b=%d\n", b) ;
    printf ("&pb=%p\n", &pb) ;
}
```

```
$ ./prog.bin
b=7
pb=ABCD0000
b=8
&pb=ABCD0004
```

contenu de la mémoire à la fin du programme :

adresse	valeur	nom de la variable
0xABCD0000	00	
0xABCD0001	00	b
0xABCD0002	00	
0xABCD0003	08	
0xABCD0004	AB	
0xABCD0005	CD	pb
0xABCD0006	00	
0xABCD0007	00	
0xABCD0008		

remarque : ici, les adresses sont sur 32 bits. Sur un système 64 bits, les adresses mémoire sont codées sur 64 bits.

Fonctions – passage par adresse

Comment renvoyer par une fonction plusieurs valeurs ?

→ En passant en paramètre l'adresse des variables que l'on veut que la fonction modifie. On appelle cela le passage par adresse.

Exemple :

```
#include <stdio.h>

void carre (float *x )
{
    *x = (*x) * (*x) ;
}

main()
{
    float re = 4.0 ;
    printf ("re = %f \n", re) ;
    carre ( &re ) ;
    printf ("re = %f \n", re) ;
}
```

Produit :

```
$ ./prog.bin
re = 4.000000
re = 16.000000
```

Fonctions – passage par adresse

Autre Exemple :

1) Écrire int minInt (int a, int b) qui renvoie le min de 2 entiers

2) Écrire void minIntAd (int a, int b, int *min) qui renvoie dans min le min de 2 entiers a et b.

3) Écrire void trierInt (int *a, int *b) qui interverti a et b si a > b.

```
#include <stdio.h>

int intMin (int a, int b) {
    if (a<b) return a ;
    else return b;
}

int main() {
    int x=12, y=8, z ;
    printf ("min de %d, %d est %d\n", x, y, intMin(x,y)) ;
    intMinAd (x,y,&z) ;
    printf ("min de %d, %d est %d\n", x, y, z) ;
    trierInt (&x, &y) ;
    printf ("Affichage dans l'ordre : %d %d\n", x, y) ;
}
```

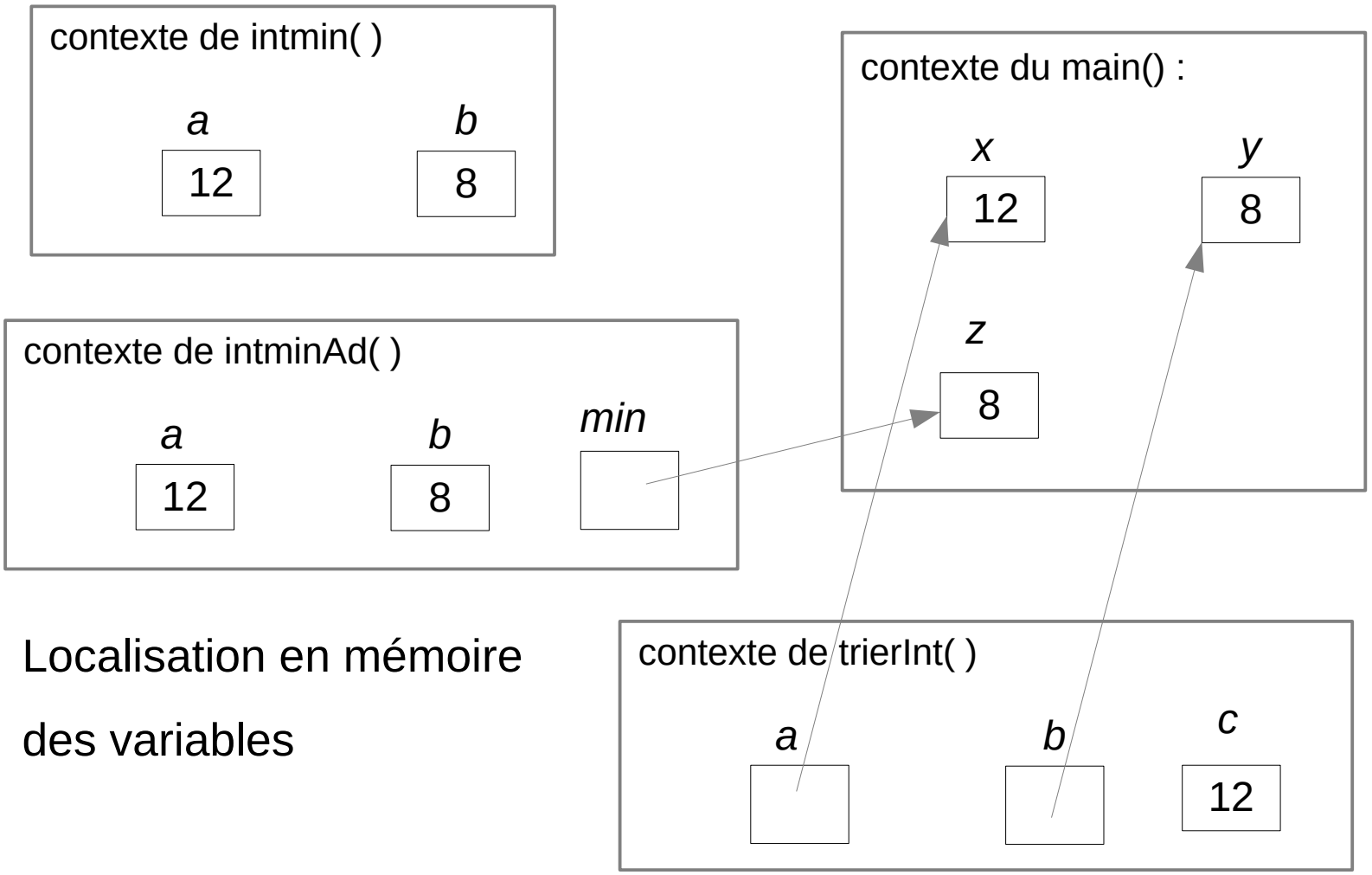
Fonctions – passage par adresse

```
void intMinAd (int a, int b, int *min) {
    *min = intMin (a, b) ;
}

void trierInt (int *a, int *b) {
    int c ;
    if (*a < *b) return ;
    else {
        c= *a ;
        *a = *b ;
        *b = c ;
    }
}
```

produit :

```
min de 12, 8 est 8
min de 12, 8 est 8
Affichage dans l'ordre : 8 12
```

Fonctions – passage par adresse



Localisation en mémoire

des variables

Tableaux

– C'est un ensemble de **cases contiguës** en mémoire ;
– Les cases sont du **même type** et font donc la même taille ;
– On accède aux éléments du tableau par l'**adresse de la première case**.

– L'indice de la case est mis entre [ ] et la première case a l'indice 0 ;

– On déclare un tableau en indiquant le type des éléments, son nom puis sa taille entre [ ] ;

– On peut initialiser un tableau quand on le déclare en mettant entre { } les valeurs initiales.

– Si on a besoin d'un tableau dont on ne connaît pas la taille au moment de la conception du programme, alors on utilise **les tableaux dynamiques**. Cette notion est vue plus loin dans le cours.

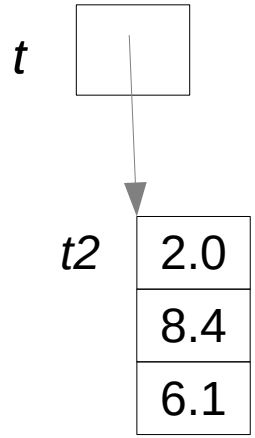
– C'est au programmeur de gérer la taille du tableau et l'indice maxi. Si on cherche à accéder à la case 100 d'un tableau de 10 cases → résultat très incertain !!

Tableaux

Exemple :

```
#include <stdio.h>

void inverserTableau (float t[] ) {
    float dumm = t[0] ;
    t[0] = t[2] ;
    t[2] = dumm ;
}

void main() {
    float t2[] = {2.0, 8.4, 6.1};
    printf("t= [%e, %e, %e]\n", t2[0], t2[1], t2[2]);
    inverserTableau(t2);
    printf ("t= [%.2e, %.2e, %.2e]\n", t2[0], t2[1], t2[2]);
}
```

remarquez le format d'affichage : 2 chiffres après la virgule



produit :

```
t= [2.000000e+00, 8.400000e+00, 6.100000e+00]
t= [6.10e+00, 8.40e+00, 2.00e+00]
```

Tableaux multidimensionnel

Exemple de dimension 2 :

```
float t4[2][3] = {{ 3, 2, 5}, {5.1, 7.0, 8} } ;
t4[1][0] = 14 ;
```

t4

3	2	5
14	7	8

– **La représentation reste linéaire en mémoire.**

t4

3
2
5
14
7.0
8

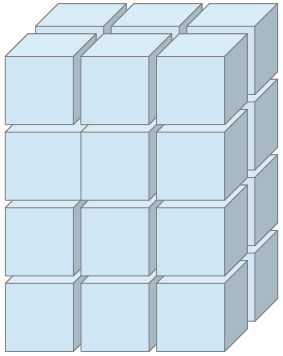
Tableaux multidimensionnel

Exemple de dimension 3 :

```
double t5[4][2][3] ;
t5[3][1][0] = 7 ;
```

t5 est un tableau de 24 cases de type double.
Il occupe 192 octets en mémoire

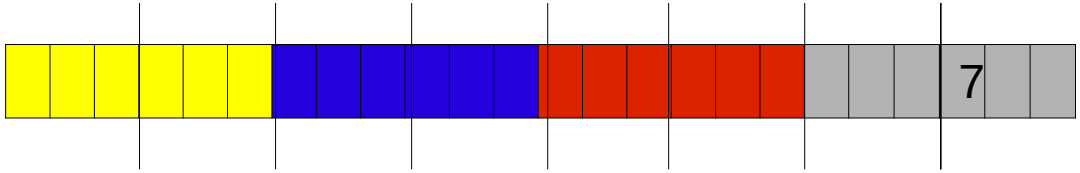
t5



vue de l'esprit

vue en mémoire

t5

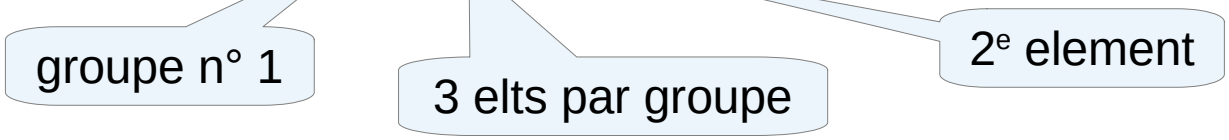


Tableaux multidimensionnel

Lors de la déclaration **des fonctions** manipulant les tableaux multidimensionnelles, le compilateur doit connaître l'agencement du tableau. Par exemple,
– pour t4, c'est 2 groupes de 3 éléments.
– pour t5, c'est 4 groupes de 2 sous groupes de 3 éléments.

Peu importe le nombre de groupes. Pour atteindre la case t4[1][2] le compilateur n'a pas besoin de connaître le nombre de lignes. L'adresse de cette case est :

t4 + (1 x 3 + 2) x taille des éléments



Donc, pour déclarer une fonction manipulant t4 :

```
void trierTableau (float t[][3]) ;
```

Donc, pour déclarer une fonction manipulant t5 :

```
void trierTableau (float t[][3][2]) ;
```

Tableaux multidimentionnels

Exemple :

```
#include <stdio.h>

float det (float t[][2] ){
  return (t[0][0]*t[1][1]-t[0][1]*t[1][0]) ;
}

void main() {
  float t4[2][2] = {{ 3, 2}, { 4 , 7} } ;
  printf("determinant de t4 = %f\n", det (t4)) ;
}
```

produit :

```
$ ./a.out
determinant de t4 = 13.000000
```

Chaînes de caractères

– C'est un tableau dont les cases sont de type char ;
– Une des cases doit contenir le caractère '\0', dont la valeur est 0.
– '\0' est le marqueur de fin de chaîne.

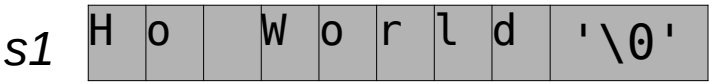
– Certaines fonctions comme printf utilisent le caractère '\0' pour traiter correctement la chaîne.

– A la déclaration, l'ensemble des caractères du tableau peut être mis entre " " pour l'initialisation.

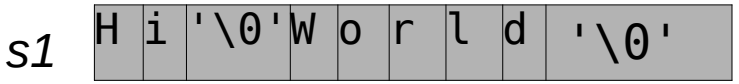
```
void main() {
    char s1[]="Ho World" ;
    s1[1]='i' ;
    printf("%s\n", s1) ;
    s1[2]=0 ; // equiv à s1[2]='\0' ;
    printf("%s\n", s1) ;
}
```

produit :

```
Hi World
Hi
```



A l'initialisation



A la fin du programme

Chaînes de caractères

Exemple :

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

int nbCar (char ch[] ) {
  int lg=0 ;
  while (ch[lg] != '\0' ) {
    lg++ ;
  }
  return lg ;
}

int main (void) {
  char phrase[]="Esope reste ici" ;
  char ph[100] ;
  printf ("longueur de \"%s\" = %d\n", phrase, nbCar(phrase)) ;
  inverser (ph, phrase) ;
  printf ("Inversion de \"%s\" = %s\n", phrase, ph) ;
}
```

produit :

```
longueur de "Esope reste ici" = 15
Inversion de "Esope reste ici" = ici etser eposE
```

Chaînes de caractères

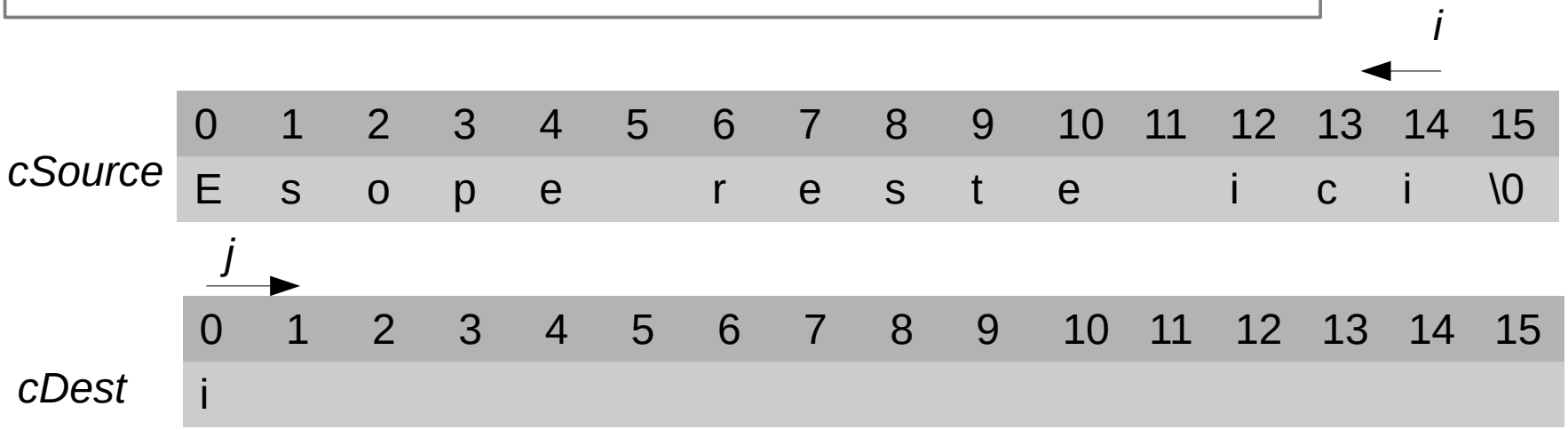
Exemple :

```
void inverser (char cDest[], char cSource[] ) {
  int i=nbCar(cSource)-1 ;
  int j=0 ;
  while ( i>=0 ) {
    cDest[j]=cSource[i] ;
    i-- ;
    j++ ;
  }
  cDest[j]='\0' ;
}
```

On part du caractère n° 14

Traiter également le caractère n° 0

Important pour la fin de chaîne



Chaînes de caractères
fonctions existantes

– Il existe des fonctions pour manipuler les chaînes de caractères.

– ne pas oublier `#include <string.h>` au début du fichier pour déclarer les prototypes des fonctions.

```
int strlen(const char *s);
```

retourne la **longueur** (le nombre de caractères, '\0' exclus) d'une chaîne.

Le modificateur const garanti au programmeur que bien que s soit un pointeur, les données pointées par s ne seront pas modifiées.

Chaînes de caractères
fonctions existantes

```
char *strncpy(char *chDest, const char *chSource, size_t n);
```

copy *n* caractères de la chaîne *chSource* dans *chDest* et retourne un pointeur sur la chaîne *chDest* . Stop la recopie avant si la fin de chSource est rencontrée, c'est à dire si '\0' est détecté dans *chSource*. Dans ce cas, *chDest* est complétée avec des '\0' jusqu'à remplir *n* caractères.

```
int strncmp(const char *s1, const char *s2, size_t n);
```

compare la chaîne *s1* avec *s2* et retourne un entier inférieur, égal ou supérieur à zéro si respectivement, *s1* est inférieur, égal ou supérieur à *s2* au sens lexicographique. Si les longueurs de *s1* et *s2* sont plus grandes que *n*, alors ne compare que les n premiers caractères.

Chaînes de caractères
fonctions existantes

```
char *strncat(char *dest, const char *src, size_t n);
```

concatène, c'est à dire met la chaîne *src* au bout de la chaîne *dest* et met le résultat dans la chaîne *dest* . Retourne un pointeur sur la chaîne dest. La recopie de src est faite sur maximum *n* caractères.

Il existe souvent **deux versions** de ces fonctions, avec un paramètre « n » ou sans. Le paramètre « n » Il limite à n caractères le travail de la fonction.

- strcat / strncat
- strcpy / strncpy
- strcmp / strncpmp

Il est préférable d'utiliser la **version avec « n »** car cela limite les possibilités de plantage sur des chaînes mal formées sans caractère '\0'.

Chaînes de caractères
fonctions existantes

```
int puts(const char *s);
```

Affiche une chaîne à l'écran (plus exactement, sur la sortie standard) et retourne un entier >0 si l'opération a réussi.

```
char *gets(char *s);
```

Lit une chaîne à partir du clavier (entrée standard). Le résultat est stocké dans l'espace mémoire pointé par *s*. Arrête de lire quand il rencontre un '\n'. Le caractère '\n' sera stocké dans s. Retourne la même valeur que s.

La fonction gets ne connait pas la taille de la chaîne s. Il est préférable d'utiliser fgets (expliqué plus loin) avec *stream=stdin*.

```
char *fgets(char *s, int size, FILE *stream);
```

Chaînes de caractères

```
#include <stdio.h>

#include <string.h>

#define LG_PH 100
int main (void) {
    char ph1[LG_PH] , char ph2[LG_PH] ;
    printf ("Entrez une phrase : ");
    fgets (ph1, LG_PH, stdin) ;
    printf ("longueur de \"%s\" = %d\n", ph1, strlen (ph1) );
    ph1[strlen(ph1)-1]= '\0' ;
    printf ("Entrez une phrase : ");
    fgets (ph2, LG_PH, stdin) ;
    strncat (ph1, ph2, LG_PH) ;
    printf ("Concatenation des 2 : = %s\n", ph1);
}
```

définition d'une constante

pour enlever le '\n'

Modifiez le programme pour ajouter un espace

produit par '\n'

produit :

```
Entrez une phrase : Bonjour
longueur de "Bonjour
" = 8
Entrez une phrase : le monde
Concatenation des 2 : = Bonjourle monde
```

Chaînes de caractères
fonctions existantes

```
int sprintf(char *str, const char *format, params...);
```

Copie dans la chaîne pointée par *str* la chaîne pointée par *format.* Comme printf, *format* peut contenir des expressions (ie. %d ou %f...) et les remplacer par les valeurs des paramètres qui suivent. Peut-être utile pour transformer un nombre (entier, flottant...) en chaîne de caractères.

```
int sscanf(const char *str, const char *format, params...);
```

Évalue la chaîne pointée par *str* selon la chaîne pointée par *format.* Comme scanf, *format* doit contenir des expressions (ie. %d ou %f...) permettant d'analyser la chaîne pointée par *str*. Le résultat de l'évaluation est stocké dans les paramètres qui suivent. Retourne le nombre d'occurrences évaluées.
Peut-être utile pour transformer un nombre (entier, flottant...) en chaîne de caractères.

Chaînes de caractères

```
#include <stdio.h>
#define LG_PH 100

int main (void) {
    char ph1[]="14" ;
    int a, b ;
    char ph2[LG_PH] ;
    sscanf (ph1, "%d", &a) ;
    sscanf (ph1, "%o", &b) ;
    sprintf (ph2, "a=%d, b=%d", a, b);
    puts (ph2);
}
```

évalué en décimal

évalué en octal

ces 2 instructions sont équivalentes à
`printf ("a=%d, b=%d\n", a, b);`

produit :

```
a=14, b=12
```

Chaînes de caractères

Exemples en vrac sur les chaînes de caractères constantes

```
main() {
   char *c ;
   puts("salut") ;
   c="toto";
   puts(c);
}
```

produit :

```
salut
toto
```

```
main() {
   char ph1[] = "14" ;

   ph1 = "toto" ;
}
```

AUCUN SENS

ph1 est une étiquette
ph1 n'est pas une variable,
c'est l'adresse du 1er caractère
produit une erreur de compilation

```
main() {
   char *c ;
   puts("salut") ;
   c="toto";
   c[0]='m' ;
   puts(c);
}
```

le caractère c[0] est une constante.
On ne peut le modifier.

produit :

```
salut
segmentation fault (core dumped)
```

Chaînes de caractères

Exemples en vrac sur les comparaisons et copie de chaînes

```
main() {
   char ph1[]="toto" ;
   char ph2[]="toto" ;
   if (ph1 == ph2)
      puts ("VRAI");
   else
      puts("FAUX");
}
```

produit :
Faux

comparaison de 2 adresses

initialisation d'un tableau cette chaîne n'est pas constante contrairement à la page précédente

```
main() {
   char ph1[]= "toto" ;
   char ph2[]={'t','i','t','i','\0'} ;
   char *ch1 = ph1 ;
   char *ch2 = ph2 ;
   ch2=ch1 ;
   ch2[0]='P' ;
   puts (ph1) ;
}
```

modifie ph1 !

produit :
Poto

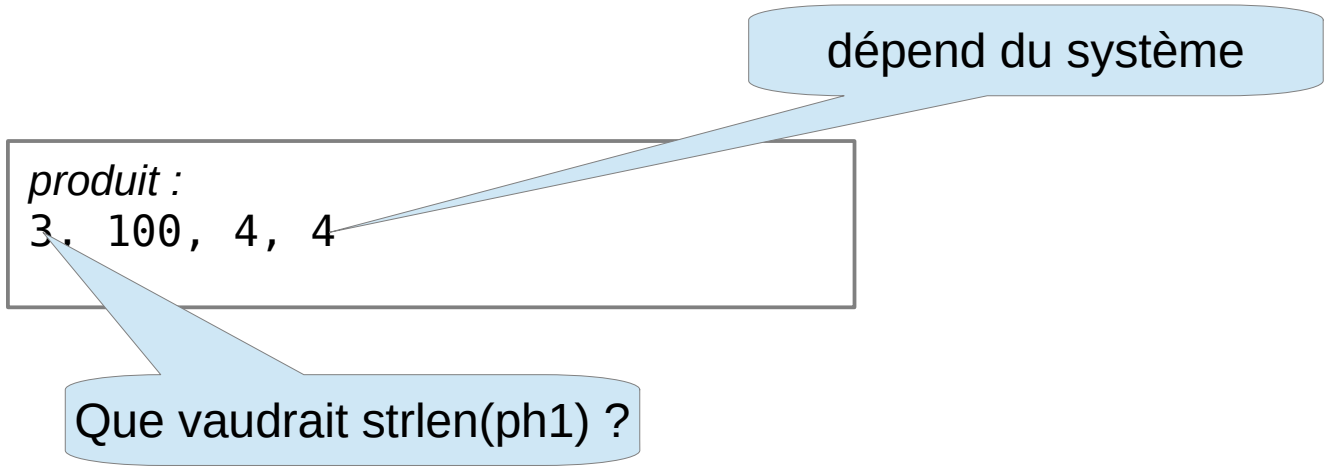
Opérateur sizeof

L'opérateur **sizeof (exp)** retourne la place mémoire occupée par *exp*.

exp peut être un type, un identifiant de variable, de structure ou de tableau.

Exemple :

```
char ph1[]="14" ;
int a ;
char ph2[100] ;
char *c ;

printf ("%d, %d ,%d\n",sizeof (ph1), sizeof (ph2), sizeof (a),
sizeof (c));
```



Types de données

Depuis C99 possibilité de définir la longueur de l'entier dans sa définition. inclure <stdint.h>

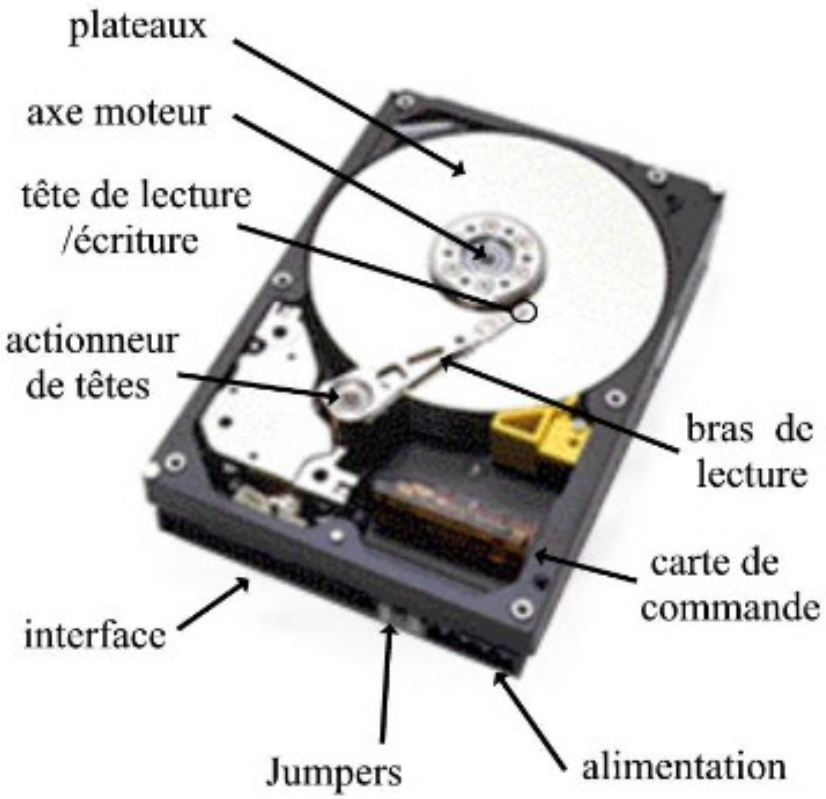
```
#include <stdint.h>

int main()
{
   printf("sizeof char = %lu\n", sizeof(char) );
   printf("sizeof short = %lu\n", sizeof(short) );
   printf("sizeof int = %lu\n", sizeof(int) );
   printf("sizeof long = %lu\n", sizeof(long) );
   printf("sizeof long long = %lu\n", sizeof(long
long) );

   printf("sizeof int8_t = %lu\n", sizeof(int8_t) );
   printf("sizeof int16_t = %lu\n", sizeof(int16_t) );
   printf("sizeof int32_t = %lu\n", sizeof(int32_t) );
   printf("sizeof int64_t = %lu\n", sizeof(int64_t) );
}
```

produit

```
sizeof char = 1
sizeof short = 2
sizeof int = 4
sizeof long = 8
sizeof long long = 8
sizeof int8_t = 1
sizeof int16_t = 2
sizeof int32_t = 4
sizeof int64_t = 8
```

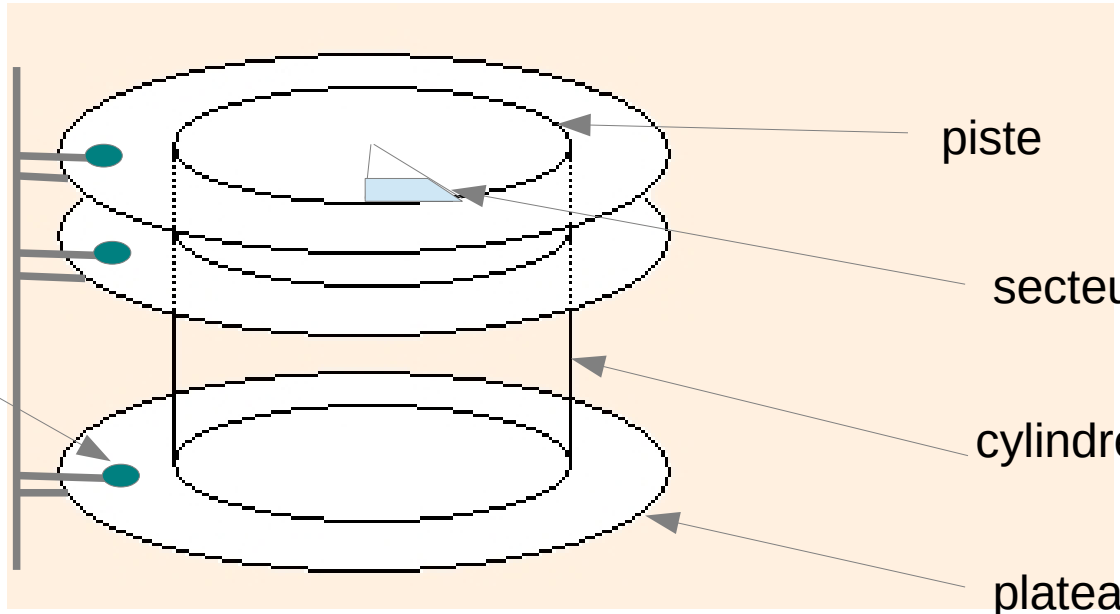
Disque dur

- vitesse de rotation : 7200 tr/min soit un tour toutes les 8,3ms
- Pour trouver une donnée : 8,3/2 + tps de déplacement des têtes = 10 ms => très long !
  → utilisation d'un cache.
- Une alternative : disques sans éléments mécaniques, Solid State Drive. Constitués de mémoire flash, avec des bons temps d'accès et une faible consommation d'énergie.
  Moins fiables que les disques durs mécaniques



Disque dur

Organisation

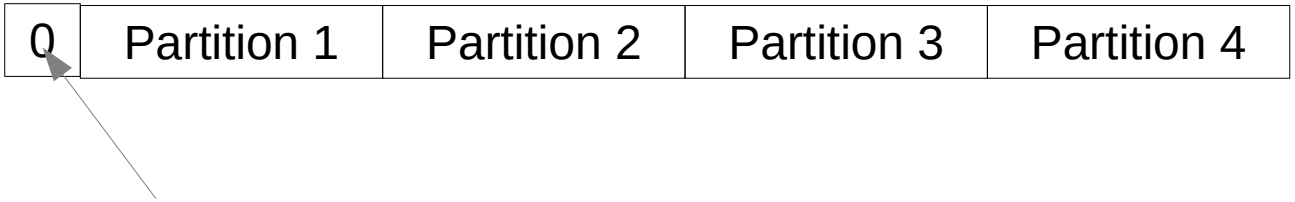


- Secteur : 512 octets
- Cylindre = toutes les pistes alignées de tous les plateaux
- Taille du disque : nb_cylindres (C) x nb_têtes (H) x nb_secteurs_par_piste (S)

Exemple : C/H/S = 65536/16/255
Taille = 127,5 Go (qui est d'ailleurs la taille limite de l'adressage IDE)

Disque dur

Partition d'un disque pour architecture PC avec Legacy BIOS



secteur 0 : Master Boot Record (MBR) : 512 octets !

Lorsque la machine boot, elle exécute le programme du BIOS qui cherche un périphérique bootable. Lorsqu'il est trouvé, le BIOS exécute le programme (loader) du secteur 0 (MBR). Le chargeur du système d'exploitation charge alors le système qui se trouve sur la bonne partition.

Le MBR contient aussi les informations de découpage en partition (*C:*, *D:*,....)

Au maximum, 4 partitions, appelées partitions primaires.

Disque dur

Exemple de format de partition : la FAT-32 **(en cours de remplacement par exFAT soumis à licence Microsoft)**

- Partition :

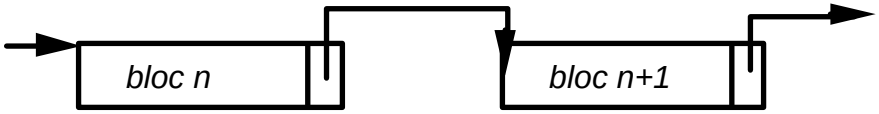
Boot sector	FAT	répertoire racine	cluster 2	cluster 3	...

- Le boot sector décrit la partition : nom du volume, nombre d'octets par secteurs, nb de secteurs par clusters...
- Unité de base : le cluster : ensemble de secteurs. ex, taille d'un cluster pour des disques < 1To : 8 secteurs, soit 4 ko.
- Le répertoire racine : il a une position particulière car il suit la FAT
- comme les autres répertoires, il contient des entrées (sur 32 octets) pour chacun des fichiers ou répertoire qu'il contient. Les informations sont :
  - le nom du fichier/répertoire,
  - sa taille (sur 4 octets, donc taille max d'un fichier = 4Go) ,
  - la date de création,
  - le numéro du premier cluster où trouver le répertoire...

Disque dur

La FAT-32 (suite)

- Un fichier ou un répertoire occupent 1 ou plusieurs blocs qui ne sont pas obligatoirement contigus.
- Dans chaque bloc (cluster), les 4 derniers octets servent à numéroter le bloc suivant. Ces 4 octets reçoivent la valeur ff ff ff ff pour le dernier bloc d'un fichier .



La table d'allocation des fichiers (FAT) :

- Permet de repérer rapidement les blocs libres.
- Permet d'accéder rapidement à des données lors d'un accès aléatoire.
- Chaque bloc du disque est repéré par une entrée sur 4 octets indiquant si le bloc est défectueux (0), libre (1) ou indiquant le bloc auquel il est chaîné. La valeur ffffffff est donnée pour le dernier bloc d'un fichier.

Disque dur

Organisation de la FAT-32 (suite)

Exemple de FAT :

numéro de cluster	valeur
4	1
5	1
6	7
7	9
8	0
9	11
10	1
11	fffffff
12	1

– Dans le cluster contenant les informations de répertoires, on pourra voir que le fichier commence au bloc 6
– puis la FAT renseigne que le fichier est composé des clusters 6, 7, 9 et 11.

- Lors d'un **arrêt brutal** de la machine il se peut que la FAT et le système de fichiers soient **incohérents**. L'OS va donc parcourir tous les blocs du disque et **reconstruire la FAT** lors d'un nouvel allumage de la machine... et vous, prier !

Disque dur

Les inconvénients de la FAT 32

- Nombre de clusters limités : adressage des clusters sur 28 bits. La taille d'un cluster dépend de la capacité du disque. Pour gérer les gros disques : clusters de grandes tailles : 32 Ko
- Donc capacité maximum d'un disque : 2^28*32 = 8 To
- Donc taille minimal d'un fichier : 32 Ko => beaucoup de places perdues
- Aussi, beaucoup de fragmentation.
- Taille maximum d'un fichier : 4 Go... ce n'est même pas l'image ISO d'un DVD...
- Donc mieux vaut utiliser des systèmes de fichiers récents : NTFS, ext4fs...
- et réserver FAT-32 aux clés USB et cartes SD.

Disque dur

Les inconvénients de legacy BIOS

- BIOS, né en 1980, en même temps que le premier PC
- c'est un firmware (code en ROM) exécuté par le CPU à l'allumage. Il vérifie que le matériel minimum est disponible et cherche le premier média sur lequel demarrer
- équipé d'une mémoire réinscriptible sur laquelle l'utilisateur peut enregistrer des configurations de démarrage (disque sur lequel booter, optimisation matériel…)
- la mémoire est sauvegardée grâce à une pile.
- logiciel de configuration du BIOS disponible à l'allumage en appuyant sur « ESC », « del » , « F2 » ou « F12 » selon les machines
- structure d'un autre age avec des champs limité en taille, mal adapté aux architectures modernes : remplacé par **UEFI : Unified Extensible Firmware Interface**

Disque dur

UEFI : Unified Extensible Firmware Interface

- peut gérer des disques > à 2 TB
- Utilise la GUID Partition Table (GPT) à la place du MBR.
- n’oblige pas le secteur d’amorçage à être sur le premier cluster
- indépendant du processeur bien que seuls les CPU little endian soit supportés (Intel, ARM, AMD...)
- indépendant des OS, bien qu’un firmware UEFI 64bits ne puissent booter que des OS 64 bits
- permet de booter à partir du réseau
- permet des « secure boot »

Système de fichiers

- Gérer un disque dur → compliqué
- Il faut laisser faire le système d'exploitation
- Utiliser des fonctions haut niveau du langage C qui utilisent des fonctions bas niveau du système d'exploitation : **Abstraction** !
- Le comportement dépend donc du système d'exploitation et ne sera donc pas le même sur Window's ou sur Linux...

Système de fichiers

fichiers textes / fichiers binaires

- Pourquoi cette distinction alors que tout est binaire !!
  - Un fichier texte stocke la représentation ascii des données.
  - Un fichier binaire stocke la représentation en mémoire des données.
- Prenons un exemple ; on veut stocker deux entiers : -16 et 65536

  Comment l'information est-elle codée sur le disque dur (chaque octet est exprimé en hexadécimal) :

FF	FF	FF	F0	00	01	00	00

  Dans un fichier binaire :

  Dans un fichier texte, chaque octet représente le code ascii des différents caractères :

2D	31	36	20	36	35	35	33	36	0A
-	1	6	espace	6	5	5	3	6	fin de ligne

Système de fichiers

Fichiers textes / fichiers binaires

- **Fichiers textes** : les données sont de **taille variable**
  - Données séparées par un séparateur (espace ou fin de ligne)
  - L'accès ne peut se faire qu'à partir du début en suivant chaque séparateur pour arriver à la donnée voulue.
  - On les appelle aussi des **fichiers à accès sequentiel.**

- **Fichiers binaires** : les données sont de **taille fixe**.
  - Pour accéder à l'enregistrement N, il suffit de se déplacer de N x taille d'un enregistrement.
  - On les appelle aussi des **fichiers** à **accès aléatoire.**

Système de fichiers

Fichiers textes / fichiers binaires

	Avantage	Inconvénient
Fichiers textes	– lisible par un éditeur comme notepad ou gedit	– accès lent à une donnée en fin de fichier
Fichiers binaires	– accès rapide à n'importe quelle donnée – format propriétaire permettant de se protéger de la concurrence	– format propriétaire, non modifiable autrement que par le programme

Système de fichiers

Ouverture/fermeture d'un fichier.

- Lors de la lecture et l'écriture, le système d'exploitation va **optimiser les accès** :
  - en lisant les données par bloc qui seront mises dans un buffer ;
  - en écrivant également les données par bloc à partir d'un buffer.
  - On appelle cette technique : **accès bufferisé.**
- **Problèmes :**
  - Quand vider les buffers lors de l'écriture ?
  - Si le programme s'arrête brutalement on n'est pas sûr que tout est écrit sur disque ;
  - Comment réserver de la place mémoire pour les buffers...

Système de fichiers

fopen

- Ouverture d'un fichier ;
- Demande à l'OS d'allouer les buffers et **retourne** un pointeur sur une zone mémoire où sont stockées les informations relatives à la gestion du fichier.

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

path : chemin et nom du fichier. Absolu ou relatif par rapport au répertoire où est lancé l'exécutable.
mode : type d'accès souhaité :

- "r" (read) : mode lecture. Si non trouvé, fopen retourne NULL.
- "w" (write) : création d'un nouveau fichier vide. Si un fichier de même nom existait déjà, il est écrasé
- "a" (add) : ouverture en ajout à la fin du fichier. Si le fichier n'existait pas il est créé.
- "r+" : mode mise à jour. Lecture et écriture se font au début du fichier.
- "w+" : lecture et écriture. Le fichier est créé s'il n'existait pas.
- "a+" : ouverture ou création d'un fichier. La lecture se fait au début du fichier et l'écriture à la fin.

Système de fichiers

fclose

- Demande à l'OS d'écrire les buffers d'écriture vers le média ;
- Désalloue les buffers alloués par open.

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

fp : Le pointeur de fichier.

Retourne 0 en cas de succès.

pour toutes ces fonctions, ne pas oublier
`#include <stdio.h>`

Système de fichiers

fprintf

- Écriture vers le buffer d'écriture associé à un fichier ;
- Écriture au format texte ;
- Le buffer est écrit sur le média lorsqu'il est plein ou lorsque fprintf écrit un '\n' dans le buffer.

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

stream : Le pointeur de fichier.
format, *params* : même rôle que pour printf et sprintf.

Retourne le nombre de caractères réellement écrits, ou une valeur <0 en cas d'erreur.

Système de fichiers

fscanf

- Lecture d'un fichier au format texte ;

```
int fscanf(FILE *stream, const char *format, params, ...);
```

stream : Le pointeur de fichier.
format, *params* : même rôle que pour scanf et sscanf.

Retourne le nombre d'occurrences lues ou une valeur < ou = à 0 en cas d'erreur (fin de fichier atteinte ou fichier mal formaté..)

Système de fichiers

fwrite

- Écriture vers le buffer d'écriture associé à un fichier ;
- Écriture au format binaire ;
- Le buffer est écrit sur le média lorsqu'il est plein.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

ptr : pointeur vers les données à écrire

size : taille d'un élément des données

nmemb : nombre d'éléments composant les données

stream : Le pointeur de fichier.

Retourne le nombre d'éléments réellement écrits, ou une valeur <0 en cas d'erreur.

Système de fichiers

fread

- Lecture d'un fichier binaire ;

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

ptr : pointeur vers un espace mémoire qui recevra les données

size : taille d'un élément des données à lire

nmemb : nombre d'éléments à lire

stream : Le pointeur de fichier.

Retourne le nombre d'éléments réellement lus ou une valeur <0 en cas d'erreur ou de fin de fichier.

Système de fichiers

fseek

- positionne l'endroit (le « curseur ») où sera fait la lecture ou l'écriture.

```
int fseek(FILE *stream, long offset, int whence);
```

offset : le déplacement mesuré en octet.

whence : la référence du déplacement

- SEEK_SET : début du fichier
- SEEK_CUR : position courante
- SEEK_END : fin du fichier

stream : Le pointeur de fichier.

Retourne 0 en cas de succès.

Système de fichiers

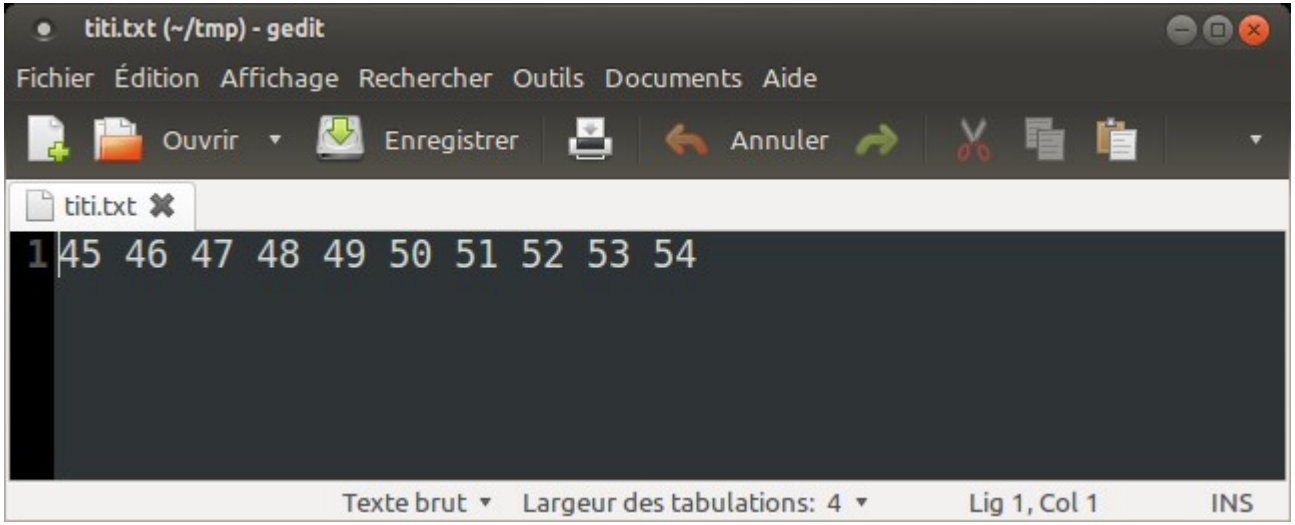
```c
#include <stdio.h>

void main(void) {

  int i;
  FILE *fpb, *fpa;

  fpb = fopen ("titi.bin","w");
  fpa = fopen ("titi.txt","w");
  for(i=45;i<55;i++){
    fwrite(&i,sizeof(int),1,fpb);
    fprintf(fpa,"%d ",i);
  }
  fclose(fpb);
  fclose(fpa);
}
```

écriture en binaire dans « titi.bin »

écriture en mode texte dans « titi.txt »

Système de fichiers

- On peut ouvrir titi.txt avec gedit mais pas titi.bin



Système de fichiers

- **hexdump** est une commande Linux/Mac OS qui permet d'afficher en hexadécimal le contenu d'un fichier octet par octet. (Il existe Hexedit pour Window's).
  - La première colonne donne l'adresse du premier octet de la ligne par rapport au début du fichier ;
  - La deuxième colonne est le contenu du fichier en hexadécimal ;
  - La dernière colonne est la traduction en ASCII de chaque octet quand c'est possible car tous les codes ne sont pas affichables : retour à la ligne...

Système de fichiers

45 en hexa sur 32 bits, little endian

46 en hexa sur 32 bits, little endian

```
lefebvre@pc:~/tmp$ hexdump -C titi.bin
00000000  2d 00 00 00 2e 00 00 00  2f 00 00 00 30 00 00 00  |-......./...0...|
00000010  31 00 00 00 32 00 00 00  33 00 00 00 34 00 00 00  |1...2...3...4...|
00000020  35 00 00 00 36 00 00 00                           |5...6...|
00000028
lefebvre@pc:~/tmp$
lefebvre@pc:~/tmp$ hexdump -C titi.txt
00000000  34 35 20 34 36 20 34 37  20 34 38 20 34 39 20 35  |45 46 47 48 49 5|
00000010  30 20 35 31 20 35 32 20  35 33 20 35 34 20        |0 51 52 53 54 |
0000001e
```

code ASCII de '4'

code ASCII de '5'

code ASCII de espace

Système de fichiers

Lecture et affichage du fichier binaire

```
#include <stdio.h>

void main(void) {

  int i;
  FILE *fpb ;

  fpb = fopen ("titi.bin","r");
  while ( fread(&i,sizeof(int),1,fpb) >0 ) {
    printf ("%d ", i) ;
  }
  fclose(fpb);
}
```

produit :

```
45 46 47 48 49 50 51 52 53 54
```

Comment charger le fichier en une seule fois (sans boucle) dans un tableau d'entiers ?

Système de fichiers

Lecture et affichage du fichier texte

```
#include <stdio.h>

void main(void) {

    int i;
    FILE *fpa ;

    fpa = fopen ("titi.txt","r");
    while ( fscanf(fpa,"%d", &i) >0 ) {
        printf ("%d ", i) ;
    }
    fclose(fpa);
}
```

produit :

```
45 46 47 48 49 50 51 52 53 54
```

Comment charger le fichier en une seule fois (sans boucle) dans un tableau de char ?

Système de fichiers

Comment gérer le problème de :

- fichiers introuvables
- de droits insuffisants
- de fichiers endommagés
- bref, de problème d'ouverture

→ ***perror***

Cette fonction permettra d'interroger le système d'exploitation et de connaître la raison de l'échec de l'ouverture. ***perror*** n'est pas spécifique au fichier, et permet d'afficher le résultat du dernier appel système.

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

void main(void) {
    int i;
    FILE *fpa ;
    fpa = fopen ("titi.txt","r");
    if (fpa == NULL) {
        perror ("pb fopen") ;
        return (EXIT_FAILURE) ;
    }
```

Système de fichiers

Information sur le fichier : stat, fstat

```
int stat(const char *pathname, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

```
struct stat {
               dev_t     st_dev;         /* ID of device containing file */
               ino_t     st_ino;         /* inode number */
               mode_t    st_mode;        /* protection */
               nlink_t   st_nlink;       /* number of hard links */
               uid_t     st_uid;         /* user ID of owner */
               gid_t     st_gid;         /* group ID of owner */
               dev_t     st_rdev;        /* device ID (if special file) */
               off_t     st_size;        /* total size, in bytes */
               blksize_t st_blksize;     /* blocksize for filesystem I/O */
               blkcnt_t  st_blocks;      /* number of 512B blocks allocated
*/

               /* Since Linux 2.6, the kernel supports nanosecond
                  precision for the following timestamp fields.
                  For the details before Linux 2.6, see NOTES. */

               struct timespec st_atim;  /* time of last access */
               struct timespec st_mtim;  /* time of last modification */
               struct timespec st_ctim;  /* time of last status change */
}
```

Système de fichiers

Exemple d'utilisation de la fonction stat pour récupérer la taille d'un fichier
Ici, le programme retourne la taille en octet de l'exécutable.
En effet, la chaîne de caractère argv[0] contient le nom du fichier exécutable.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main (int argc, char **argv) {
    int err ;
    struct stat info ;
    err = stat (argv[0], &info) ;
    if (err <0) {
         perror("Probleme appel stat") ;
         return (EXIT_FAILURE);
    }
    printf("Taille de %s : %d\n", argv[0], (int) info.st_size) ;
    return (EXIT_FAILURE);
}
```

produit

```
Taille de ./exemp : 8824
```

Définition de type

- Comment alléger la déclaration de certaines variables ?
- Comment donner un nom plus parlant à certains types ?

  → En utilisant **typedef**

- Syntaxe :

```
typedef  définitiondeType NouveauNomDuType ;
```

- Exemple :

```
typedef  int Booleen ;
typedef  unsigned char EntierNonSigne8bits ;

void main() {
    Booleen b=1 ;
    EntierNonSigne8bit w=7 ;
    . . .
}
```

- Voir également l'exemple au chapitre suivant sur les structures

Structures de données

- Comment agréger plusieurs types de données dans une même variable ?

  → En utilisant des structures

- Permet également à une fonction de retourner plusieurs valeurs.

- Déclaration de la structure et de ses champs :

```
struct nomDeLaStruct {
    type1 champ1;
    type2 champ2 ;
    …
} ;
```

- Déclaration d'une variable de ce type :

```
struct nomDeLaStruct nomVariable ;
```

Accès aux champs,
en séparant avec un « point » :

```
nomVariable.champ1 = valeur ;
```

Structures de données

Exemple : Traduction coordonnées cartésiennes / cylindriques

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

typedef struct sCoordCyl { //exemple en utilisant typedef
    float r, alpha, z ;
}  CoordCyl ;

struct sCoordCart { //exemple sans utiliser typedef
    float x, y, z ;
} ;

struct sCoordCart convert (CoordCyl p) {
    struct sCoordCart res ;
    res.x = p.r*cos(p.alpha) ;
    res.y = p.r*sin(p.alpha) ;
    res.z = p.z ;
    return res ;
}
```

Structures de données

Traduction coordonnées cartésiennes / cylindriques

```
void main() {
    struct sCoordCart pointA_cart ;
    CoordCyl pointA_cyl={1, M_PI/3.0, 2 } ;

    pointA_cart = convert(pointA_cyl) ;

    printf ("A en coordonnée cartésienne : [%f %f %f]\n",
        pointA_cart.x, pointA_cart.y, pointA_cart.z ) ;
}
```

produit :

```
A en coordonnée cartésienne : [0.500000 0.866025 2.000000]
```

Allocation Dynamique

Comment déclarer des variables/tableaux supplémentaires pendant l'exécution d'un programme ?

Autrement dit, comment disposer d'un espace mémoire supplémentaire ?

- En C99 : possibilité de créer des variables ou des tableaux n'importe où. Exemple :

```
scanf ("%d", &n) ;
int tab[n] ;
```

  - mais l'emplacement de tab sera dans la pile donc :
  - tab sera détruit quand on sort de la fonction qui l'a déclaré
  - et la pile a une taille limitée, par défaut de 8 Mio sous Linux ou 1 Mio sous Window's

→ Solution : **l'allocation dynamique**

Allocation Dynamique

Allocation dynamique :

- Dans le tas : c'est un espace mémoire persistant jusqu'à l'arrêt du programme.
- La taille du tas dépend du système d'exploitation :
  - Linux 32 bits : 3 Gio
  - Windows 32 bits : 2 Gio
  - OS 64 bits... sous Linux, limité à la taille du swap + taille de la RAM.
- fonctionne en C ANSI
- Certains « petits » OS ne gèrent pas de tas dynamiquement

Allocation Dynamique

Les fonctions de base :

- malloc(), calloc(), realloc() et free()
- prototypes dans <stdlib.h>
- utilise un type prédéfini size_t qui est une redéfinition de « unsigned int »

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

- réserve size octets, sans initialisation de l'espace.

```
void *calloc(size_t nmemb, size_t size);c
```

- réserve nmemb éléments de taille size octets chacun ;
- l'espace est initialisé à 0.

Allocation Dynamique

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

- modifie la taille à size affectée au bloc de mémoire fourni par un précédent appel à malloc() ou calloc().

```
void free(void *ptr);
```

- libére le bloc mémoire pointé par un précédent appel à malloc(), calloc() ou realloc().

- Ces fonctions doivent être transtypées (« castées ») car le type de retour est un pointeur sans type : void*
- Retournent le pointeur NULL (0) si l'espace disponible est insuffisant.

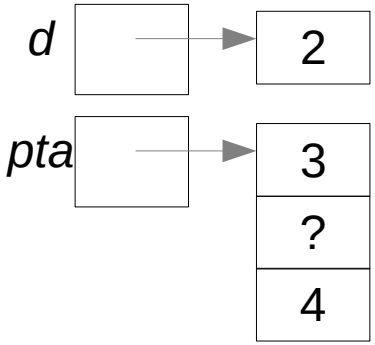
Allocation Dynamique

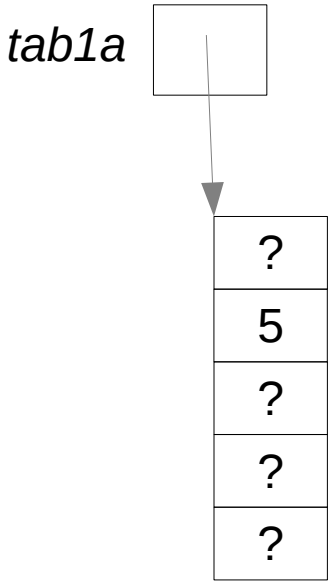
Exemple : une structure et 2 vecteurs créés dynamiquement

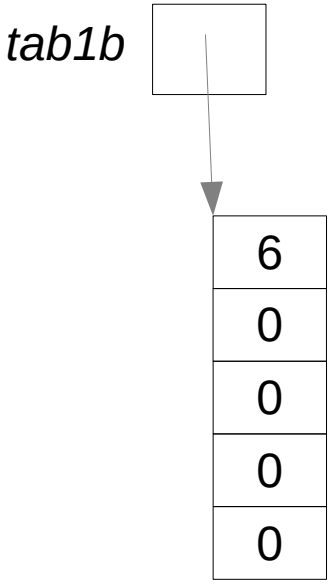
```
void main() {
    double *d ;
    d = (double*) malloc (sizeof (double)) ;
    *d = 2 ;

    CoordCyl *ptA = (CoordCyl *) malloc (sizeof (CoordCyl)) ;
    (*pta).r = 3 ; pta->z = 4 ;

    int *tab1a ;
    tab1a = (int*) malloc (5*sizeof(int)) ;
    tab1a[1] = 5 ;

    int *tab1b = (int*) calloc (5, sizeof(int)) ;
    tab1b[0] = 6 ;


}
```

Allocation Dynamique

Dessinons tout cela !!!







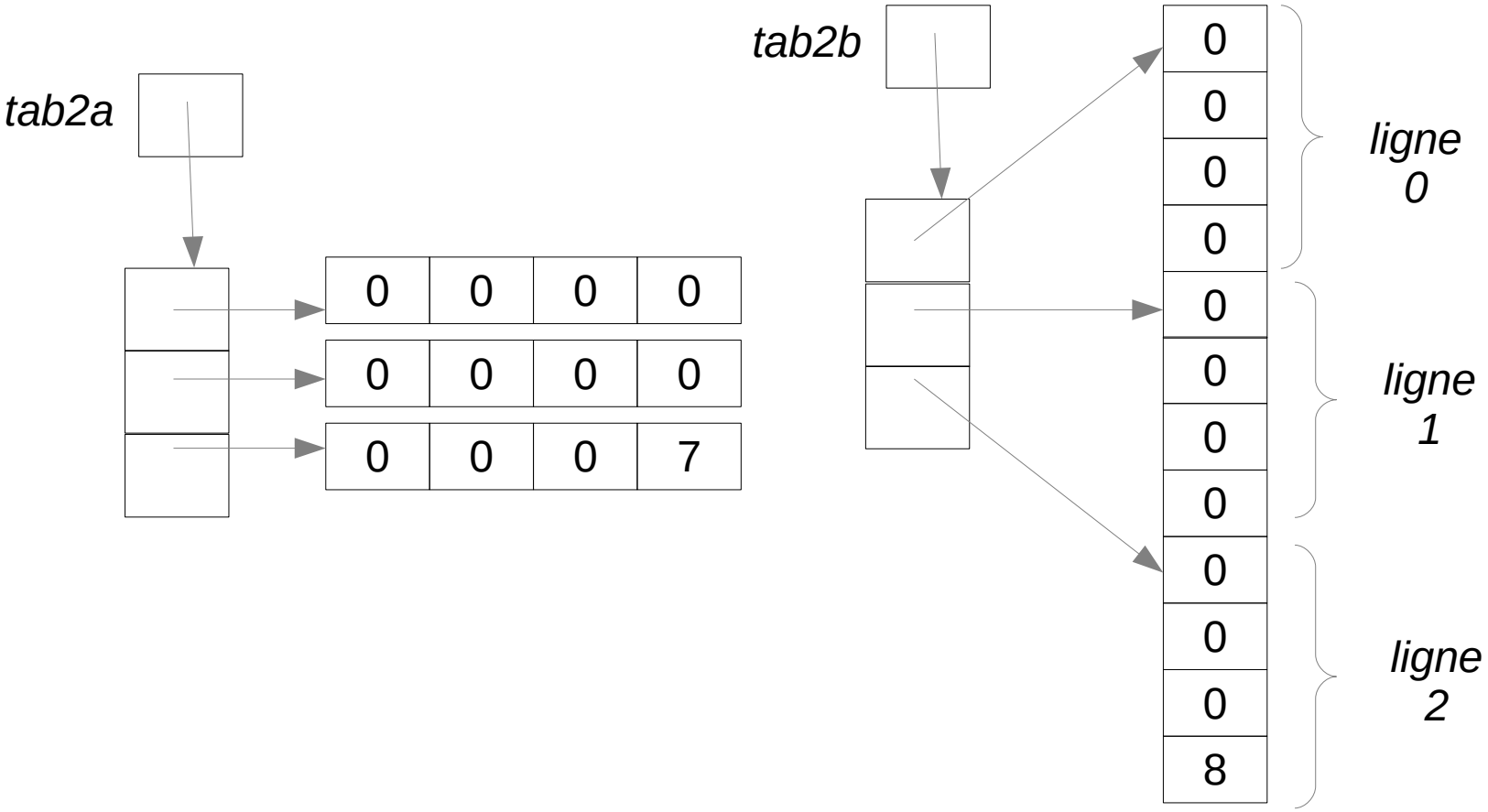
Allocation Dynamique

Exemple : deux matrices créées dynamiquement

```
void main() {

    float **tab2a ;
    tab2a = (float**) calloc (3, sizeof(float*)) ;
    for (i=0 ; i<3 ; i++ )
        tab2a[i] = ((float*) calloc (4, sizeof(float)) ;

    tab2a[2][3]=7 ;

    float **tab2b = (float**) calloc (3, sizeof(float*)) ;
    tab2b[0] = ((float*) calloc (3*4, sizeof(float)) ;
    for (i=1 ; i<3 ; i++ )
        tab2b[i] = tab2b[i-1]+4 ;

    tab2b[2][3]=8 ;

}
```

Allocation Dynamique

Dessinons tout cela !!!



Arithmétique de pointeurs

- **Simplification d'écriture pour les structures**

  `(*ptA).z` est équivalent à `ptA->z`

- **Addition de pointeur**

  si `tab1a=0x0010`

  l'adresse de `tab1a[0]` est `0x0010`

  `*tab1a` est équivalent à `tab1a[0]`

  `*(tab1a+1)` est équivalent à `tab1a[1]`

  L'adresse de `tab1a[1]` est donc `0x0014`, car tab1a est un pointeur sur des entiers, et la taille mémoire d'un entier est 4 octets.

L'addition est typée !

Encore de la gymnastique de pointeurs

Exemple :

```
void main() {
    float a; float t[5]; float *p;
    p = &a;
    *p = 8;
    p = t;
    *p = 9;
    *(p+2) = 10;
    printf("a =  %f, t[0]=%f, t[2]=%f  \n",a , t[0], t[2]);
    printf ("p = %d p = %x @t[0] = %x, @t[2] = %x\n", p, p,
&(t[0]), &(t[2]) );
}
```

produit :

```
a =  8.000000, t[0]=9.000000, t[2]=10.000000
p = -1078918316 p = bfb10354 @t[0] = bfb10354, @t[2] = bfb1035c
```

Encore de la gymnastique de pointeurs

Remarques :

- Dans l'exemple précédent
  - `p = t` a un sens
  - `t = p` n'a aucun sens !!
- Attention, lors de la déclaration,

```
int* a, b ;
```

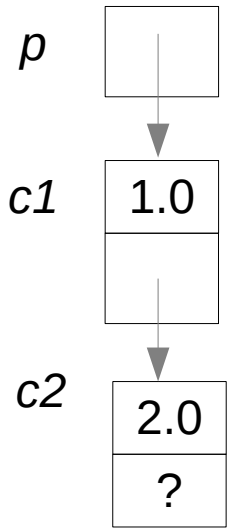
déclare

- a de type int*
- b de type int

Encore de la gymnastique de pointeurs

Exemple :

```
#include <stdio.h>

typedef struct sCell {
    float nb;
    struct sCell *suiv;
}  Cellule;

void main() {
    Cellule c1, c2, *p ;
    c1.suiv=&c2 ;
    p = &c1 ;
    p->nb=1 ;
    p->suiv->nb=2 ;
    printf("c1.nb = %e\n", c1.nb);
    printf("c2.nb = %e\n", c2.nb);
}
```



produit :

```
c1.nb = 1.000000e+00
c2.nb = 2.000000e+00
```

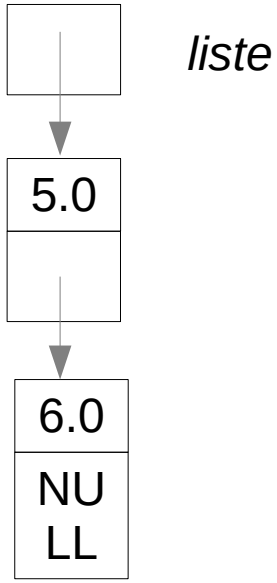
Encore de la gymnastique de pointeurs

Exercice :

1) Écrire une fonction qui crée une cellule et retourne son pointeur. Expliquer pourquoi cela doit se faire avec la fonction malloc (ou calloc...).

2) Écrire la fonction
   *Cellule *ajouter(Cellule *liste, float n)*
   qui ajoute une nouvelle cellule au début de la liste, initialisée avec un flottant passé en paramètre.

3) Écrire la fonction afficherListe.




```
void main() {
    Cellule *liste=NULL ;
    liste = ajouter (liste, 6.0) ;
    liste = ajouter (liste, 5.0) ;
    afficherListe(liste) ;
}
```

produit :

```
liste = 5, 6,
```

Déclaration des variables

- **variables globales** : visible dans tout le fichier
  - Déclarée à l'extérieur de toute fonction.
  - stockée dans le tas (comme les allocations dynamiques)
  - Un autre fichier peut utiliser cette variable s'il la déclare avec le modificateur `extern`.
- **variables locales** : visible uniquement dans le bloc ou la fonction qui est délimité par les accolades. Ces variables sont aussi appelées « automatique » (mot clé obsolète : `auto`)
  - main est une fonction !
  - stockée dans la pile (comme les allocations dynamiques)
  - éphémère : lorsque le programme quitte la fonction, le contenu est perdu. Si on entre de nouveau dans la fonction la variable est n'a plus le contenu précédent
- **variables « statiques »**: Le mot clé « static » s'applique à la déclaration d'une variable dans une fonction. Le contenu de la variable est préservé entre plusieurs appels successifs de cette fonction.

Déclaration des variables

Exemple :

produit :

```
s_glo = 55
somme static = 55
somme locale = 0
```

```
int s_glo=0 ;

void sommeGlo (int n) {
   int i ;
   for (i=n; i>0 ; i-- )
      s_glo = s_glo+ i ;
return ;
}

int sommeStat (int n) {
    static int som=0 ;
    if (n == 0) return som;
    som =som+n ;
    return(sommeStat (n-1 )) ;
}

int sommeLoc (int n) {
    int som=0 ;
    if (n == 0) return som;
    som=som+n ;
    return (sommeLoc (n-1 )) ;
}

void main() {
    sommeGlo (10) ;
    printf("s_glo = %d\n", s_glo) ;
    printf("somme static = %d\n", sommeStat(10)) ;
    printf("somme locale = %d\n", sommeLoc(10)) ;
}
```

Sommaire