http://nic.volanschi.free.fr/teaching/c

Programmation impérative

N. Volanschi *

Introduction à la programmation impérative en langage C

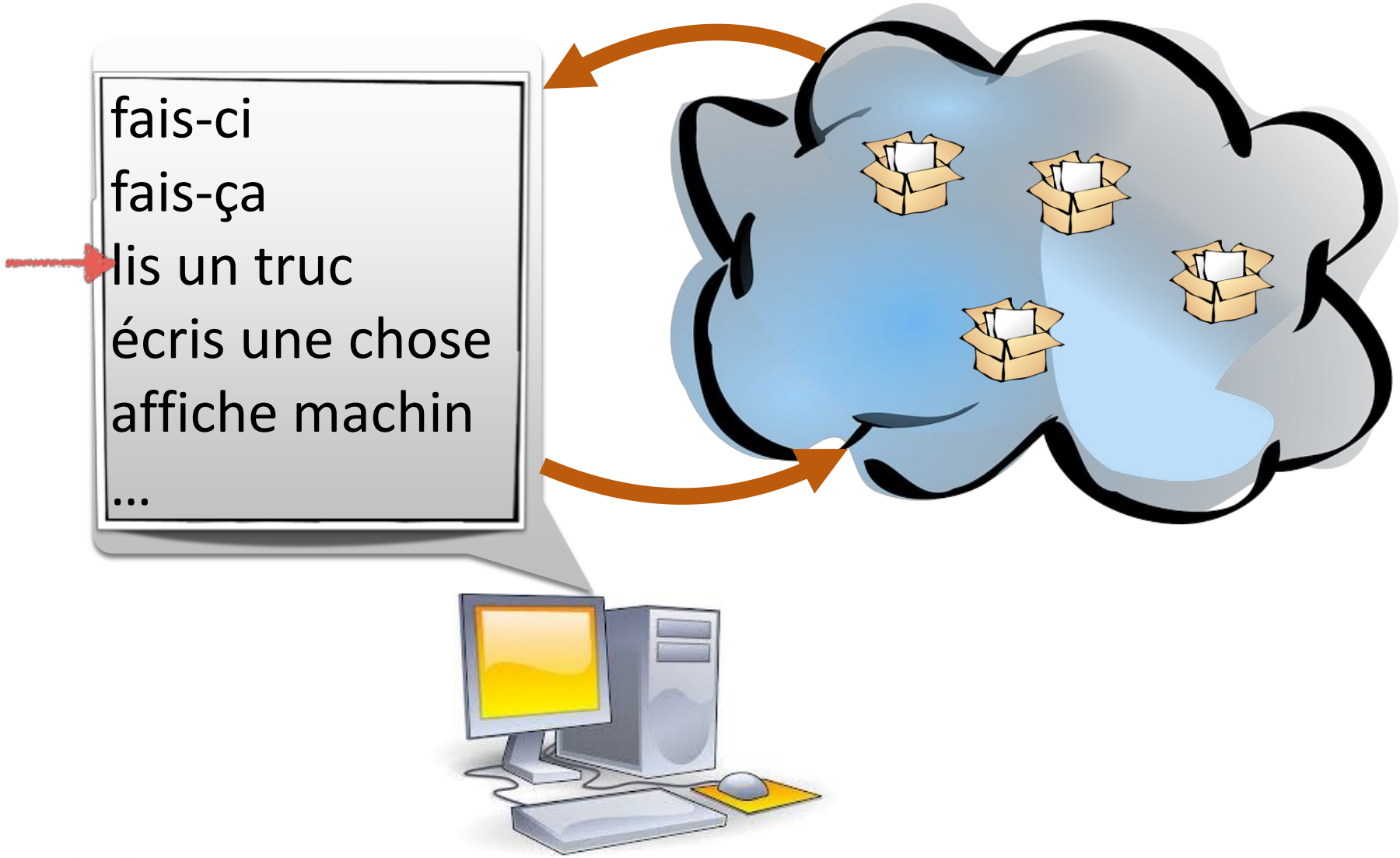
15 octobre 2019

- En collaboration avec G. Mercier, L. Réveillère, F. Morandat, B. Rouzeyre

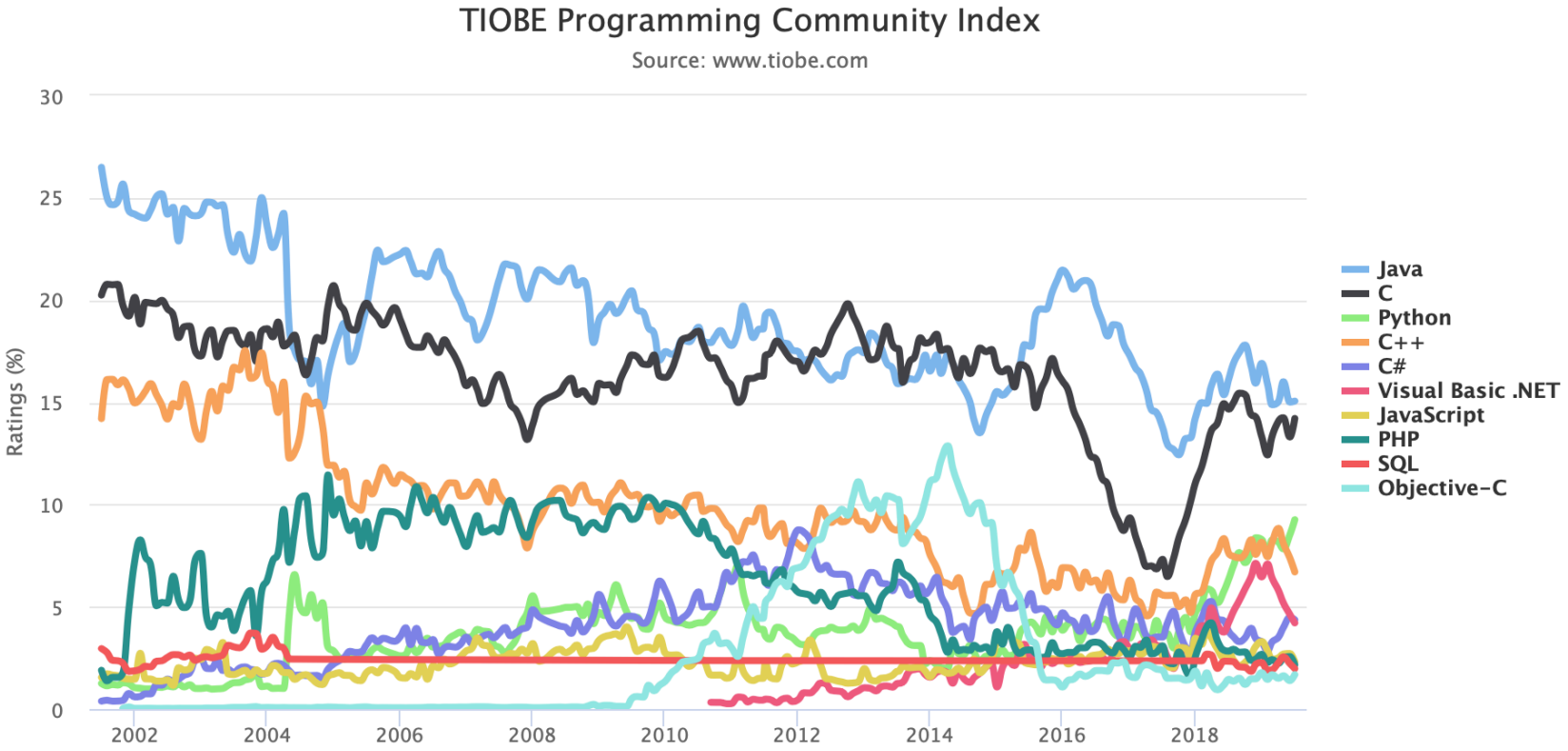
COURS 1-2: UN SURVOL DU C

Contexte
Hello world
Compilation
Variables
Expressions
If
While

Programmation impérative



Le langage C



- Toujours très populaire (#2)
- Apprécié pour sa proximité avec le système & le matériel
- Utilisé pour les systèmes embarqués ou contraints
- Portable (… mais pas tout seul !)

Caractéristiques du C

- Langage compilé
- Typage statique
- (Assez) fortement typé
- «Haut» niveau (mais le plus bas)
- Gestion de la mémoire explicite
- Pas de surcoût à l'exécution
  - Il ne fait rien d'autre que ce que vous lui dites!

Histoire

- Inventé en 1972 (Dennis Ritchie, Bell Labs)
- Spécification : C K&R (1978)
- Normalisation ANSI : Ansi C / C89 (1989-1990)
- **Normalisation ISO : C99 (1999)**
- Mise-à-jour : C11 (ou C1x) (2011)

Mon premier programme

example-01.c

```c
#include <stdio.h>

/*
        This program prints ``Hello world''
*/
int main(int argc, char **argv) {
        printf("Hello world\n");
        return 0;
}
```

```
$ cc -std=c99 -Wall —o example-01 example-01.c
$ ./example-01
Hello world
```

Options de compilation

- -o <output> (nom du programme généré)
- -Wall (montre plus d'erreurs)
- -std=c99 (moins restrictif que -ansi)
- -Werror (attention => erreurs)

S’il vous plaît …

- Utilisez un VRAI éditeur de texte (emacs, vim)
- Mettez de la couleur
- Utilisez l’indentation automatique
- Restez cohérent
- Utilisez des noms évocateurs
- Respecter l’indentation (espacement
- Testez régulièrement
- Commentez (et en anglais c’est mieux !)

Instructions

- Instruction simple
  - Se termine par un ;
- Bloc d'instructions
  - { instructions }
  - pas de ;

Exemples d'instructions

- Expression
- Déclaration de variable
- Structure conditionnelle (test)
- Structure répétitive (boucle)
- Retourner une valeur
- Ne rien faire

Expressions

- Caractéristiques
  - Évaluable => Valeur
  - Typé
- Forme
  - Constante
  - Variable
  - Expression composée
  - Appel de fonction

Expressions

		Assoc
x( )	Appel de fonction	
( )	Groupe	
* / %	Multiplier, diviser, modulo (reste de la div)	G
+ -	Addition et soustraction (binaire)	G
> >= < <=	Plus grand (ou égal) Plus petit (ou égal)	G
== !=	Égalité, différent	G
=	Affectation	D

Variables

- Variable : élément de mémorisation nommé
- Toutes les variables doivent être déclarées suivant un type

```
int a, b;
```

ou bien

```
int a;
int b;
```

```
float x;
char caractere;
```

- Initialisation:
  - Par affectation :

```
char a;
int un;
a = 'a';
un = 1;
a = '1';
```

  - Dès la définition :

```
char a = 'a';
int i = 0;
```

Identificateurs

- Nom d'un objet (variable, fonction, type…)
  - Le premier caractères doit être une lettre
  - Les autres caractères sont les lettres (majuscules et minuscules), les chiffres et le caractère _
  - Majuscule / minuscule significatifs
  - Ne peut être un des mots réservés du C:

auto	default	float	register	struct	volatile
break	do	for	return	switch	while
case	double	goto	short	typedef	
char	else	if	signed	union	
const	enum	int	sizeof	unsigned	
continue	extern	long	static	void	

- Exemples :

  Pi, pi3_14, a2B3 : corrects

  2x,i-E : incorrects

  A1 ≠ a1

Structure conditionnelle

```
if (expression)
    instruction
```

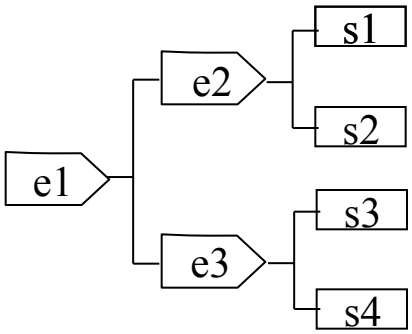
```
if (expression)
    instruction
else
    instruction
```

- Si l'expression est vraie,
  alors on exécute l'instruction
  sinon on exécute l'autre instruction
- !!! Attention à l'imbrication !!!
- else se réfère au if le plus proche
- Conseil: utilisez les {}

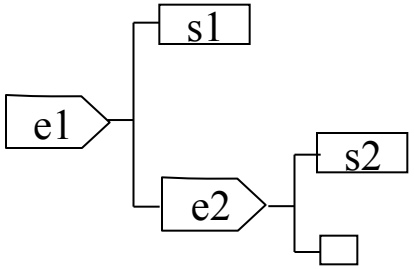
Compléments sur les instructions de contrôle

- Instruction if : imbrication

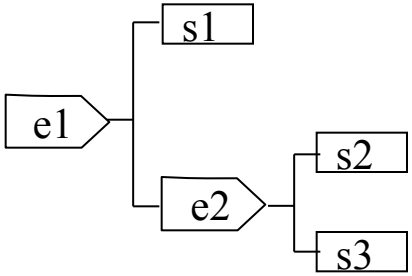
ex1 : if(e1) if(e2) s1;
else s2;
else if (e3) s3;
else s4;



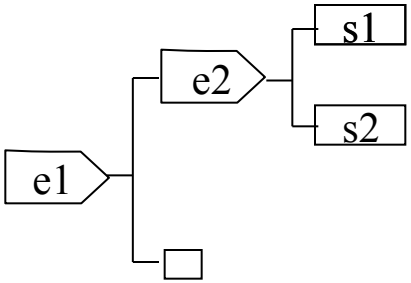
ex2 : if(e1) s1;
else if (e2) s2;

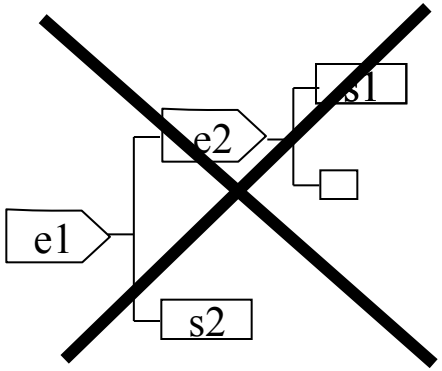


ex3 : if(e1) s1;
else if (e2) s2;
else s3;



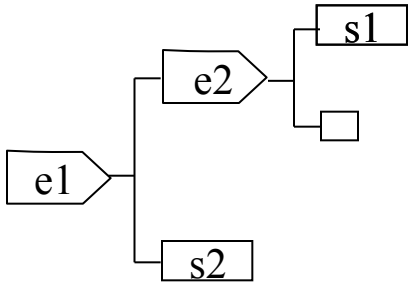
ex4 : if(e1) if(e2) s1;
else s2;





Compléments sur les instructions de contrôle

- Règle : le else se rapporte au if le + imbriqué




```
if(e1) {
   if(e2) s1;
} else s2;
```

```
if(e1)
   if(e2) s1;
   else ;
else s2;
```

example-02.c

```
#include <stdio.h>

int main(int argc, char** argv) {
  int a = 5;
  int b = 3;
  int res;
  if (a > b) // Les parenthèses sont OBLIGATOIRES
    res = a;
  else
    res = b;
  printf("Le max est: %i\n", res);
  return 0;
}
```

Structures répétitives

```
while (expression)
    instruction
```

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

- Tant que expression est vraie
  faire instruction
- !!! Attention, l'éternité c'est long vers la fin !!!
- Faire instruction tant que expression est vraie
- !!! Attention la deuxième forme ne se finit pas par une instruction !!!

Exemple

```
//example-03.c
#include <stdio.h>

int main(int argc, char* argv[]) {
  int a = 5;
  int b = 3;
  int res = 0;
  while (b > 0) {
    res = res + a;
    b = b - 1;
  }
  printf("Le produit est: %i\n", res);
  return 0;
}
```

Make

- Utilitaire pour piloter la chaîne de compilation d'une application donnée (cf. un « makefile »)
- Repose sur un langage dédié (domain-specific language, ou DSL)
- Règles de construction (build) d'une application
  - Implicites (prédéfinies)
  - Explicites (définies par le programmeur – plus tard)
- Build incrémental:
  - On reconstruit juste ce qui a changé

```
# Makefile
CC=gcc
CFLAGS=-Wall –std=c99
```

```
ex/$ make example-03
gcc -Wall    example-03.c   -o example-03
ex/$ make example-03
make: `example-03' is up to date.
ex/$ rm example-03
ex/$ make example-03
gcc -Wall    example-03.c   -o example-03
ex/$ make example-03
make: `example-03' is up to date.
ex/$ touch example-03.c
ex/$ make example-03
gcc -Wall    example-03.c   -o example-03
```

Exercices

- Rajouter une vérification du résultat de la multiplication
  - Utiliser l'opérateur '*'
- Écrire une version qui calcule a modulo b et vérifie le résultat
  - Utiliser l'opérateur '%'

COURS 3-4:
TABLEAUX, FONCTIONS

Instruction for
Tableaux
Fonctions

Structures répétitives 2

- Initialiser
- Tester
- Faire
- Incrémenter

```
i = 0;
while (i < 10) {
  instruction;
  i = i + 1;
}
```

- Boucle for équivalente:

```
for (i = 0; i < 10; i = i + 1)
  instruction;
```

- Ou même:

```
for (int i = 0; i < 10; i = i + 1)
  instruction;
```

Tableaux

- Lorsque on veut mémoriser plusieurs données de même type, on peut utiliser un tableau c-à-d on regroupe sous un même nom plusieurs informations
- Exemple de déclaration d'un tableau d'entiers

```
int tab[100];
```

int : type des éléments du tableau

tab : identificateur (nom du tableau)

100 : nombre d'éléments du tableau (dimension)

tab peut recevoir 100 entiers identifiés de 0 à 99 (attention !)

le premier est tab[0]

le second tab[1]

..

le dernier tab [99]

Tableaux

- Utilisation

  chaque élément du tableau est accessible par un indice qui doit être de type entier, quelque soit le type des éléments du tableau

  exemples :

  int i ;

  tab[2] 3eme élément du tableau

  tab[2+3] 6eme élément du tableau

  tab[i] i+1eme élément du tableau

- Exemples :

  stocker les 100 premiers nombres pairs : 0,2,4,...,196,198

```
int i, t[100];
for (i=0; i < 100; i=i+1)
    t[i]= 2*i;
```

Tableaux

- Remarques:

  1/ chaque élément du tableau s'utilise comme une variable

  tab[3] = 2;

  2/ le nombre maximum d'éléments du tableau (dimension)

  1/ doit être fixé à la définition

  2/ ne peut pas être modifié par la suite

  3/ Pas d'opérations sur les tableaux en tant que tels

  4/ … sauf déclaration initialisée:

  float tab[3] = {1.0, 2.0, 3.5};

  float tab[] = {1.0, 2.0, 3.5};

Parcours des éléments d'un tableau

Parcours du premier au dernier

```
int i;  /* l'indice de balayage doit être un entier */
float t[100];  /* le type du tableau est quelconque, ici réel */
for (i=0; i < 100; i=i+1)  // ou bien for (i=0; i <= 99; i=i+1)
    t[i]= …….;
```

Parcours du dernier au premier

```
int i;  /* l'indice de balayge doit être un entier */
float t[100];  /* le type du tableau est quelconque */
for (i=99; i >= 0; i=i-1)
    t[i]= …….;
```

La dimension

Bonne pratique de programmation

```
int i;
int t[100];
for (i=0; i < 100; i=i+1)
    t[i]= 100;
```

Pb : modification du pgm, changement de la taille du tableau malaisée

```
#define TAILLE 100
int i;
int t[TAILLE];
for (i=0; i < TAILLE; i=i+1)
    t[i]= 100;
```

Il suffit de changer TAILLE

Exemples

Point de l'espace

1ere solution :

float x,y,z;

2eme solution

float pt[3];

pt[0] pour x, pt[1] pour y, pt[2] pour z

Mémorisation des 100 premiers nombres pairs et impairs:

```
int pairs[100], impairs[100];
int i;
for (i=0; i<100;i=i+1) {
    pairs[i]=2*i;
    impairs[i]=2*i+1;
}
```

Exercice

- Écrire un programme (max.c) qui calcule la valeur maximale dans un tableau

Solution: max.c

```
#include <stdio.h>

#define N 5
int tab[N] = {10, 20, 42, 40, 12};

int main(int argc, char **argv) {
  int max = tab[0];
  for (int i = 1; i < N; i = i + 1) {
    if(tab[i] > max)
      max = tab[i];
  }
  printf("Max du tableau: %i\n", max);
  return 0;
}
```

Les fonctions

- Une fonction permet de :
  - Remplacer une partie qui se répète
  - Découper un programme en parties isolées -> débogage, lisibilité, etc..
- Exemples : fonctions d'E/S (scanf, printf, …), mathématiques (sin, cos, …)
- Organisation d'un programme :

```
type fonction1 (arguments) {
  Déclarations de variables et de types locaux à la fonction
  Instructions
}
type fonction2 (arguments) {
  Déclarations de variables et de types locaux à la fonction
  Instructions
}
...
int  main (int argc, char *argv[]) {
  Déclarations de variables et de types locaux à la fonction
  Instructions
}
```

Exemple

```
#include <stdio.h>

int max(int a, int b) {
  if (a > b) return a;
  return b;
}

int main(int argc, char* argv[]) {
  int a = 5;
  int b = 3;
  printf("Le max de %i et %i est: %i\n", a, b, max(a, b));
  return 0;
}
```

Type de la valeur de retour

Arguments

Instructions

Valeur renvoyée

Appel de la fonction

Définition de fonction : syntaxe

type_resultat nom_fonction (type_arg1 arg1, …, type_argn argn) {

…

return (valeur retournée);

}

Dans l'exemple précédent :

type_resultat : int, c'est le type de la valeur renvoyée par return

nom_fonction : max

Le nombre d'arguments est quelconque, éventuellement aucun, les parenthèses doivent toujours figurer (ex: main ())

Type de la fonction

- Une fonction peut ne pas renvoyer de valeur.
  - Dans ce cas, le type de la fonction est : void
  - Exemple (void.c):

```
void print_table(int x[], int n) {
 printf("{");
 for (int i = 0; i < n; i = i + 1) {
   if (i > 0)
     printf(", ");
   printf(”%i", x[i]);
 }
 printf("}\n");
}
```

- Le type du résultat de la fonction ne peut pas être un tableau

Instruction return

1/ Indique la valeur de retour de la fonction.

2/ Arrête l'exécution de la fonction

```
void print_table(int x[], int n) {
 if(n == 0) return;
 printf("{");
 ...
 printf("}\n");
}
```

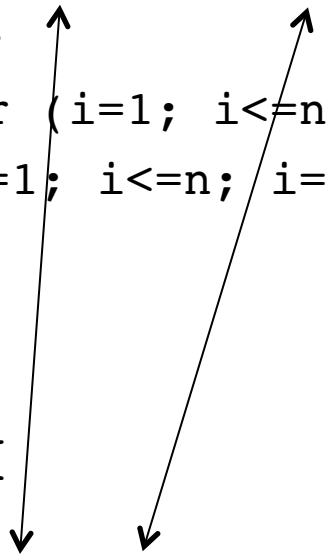
Pour les fonction de type void, le return à la fin est facultatif

Appel des fonctions

- L'appel d'une fonction se fait en donnant son nom, suivi de la liste des paramètres entre parenthèses. L'ordre des paramètres correspond à celui des arguments.
- Exemple

```
float puiss (float x, int n) {
  float y=1.0;
  if (n>0) for (i=1; i<=n; i=i+1) y = y*x;
  else for (i=1; i<=n; i=i+1) y = y/x;
  return (y);
}

void main () {
  float z,t;
  z = puiss(10.7, 2);
  t = puiss (z, -6);
...
}
```



Appel des fonctions

- Un appel de fonction peut se faire comme opérande d'une expression, ou comme paramètre d'un autre appel de fonction.
- Exemple

```
int max (int a, int b) {
  if (a > b) return a;
  return b;
}
void main () {
  int v1=11, v2=22, v3=33, m1;
  m1 = max (v1,v2);
  m1 = max (m1,v3);
  printf("valeur maximale %i\n", m1);
}
```

ou bien

```
  m1 = max(v1,v2);
  printf("valeur maximale %i\n", max(m1,v3));
```

ou bien

```
  printf("valeur maximale %i\n", max(max(v1,v2),v3));
```

Exemple: fibo.c

```
// fibo.c: Fibonacci Series using Dynamic Programming
#include <stdio.h>

int fibo(int n) {
  int f[n+1];
  int i;

  if(n < 1)
    return 0;

  f[0] = 0; // Initial values
  f[1] = 1;
  for (i = 2; i <= n; i = i + 1) {
      /* Add the previous 2 numbers in the series
         and store it */
      f[i] = f[i-1] + f[i-2];
    }
  return f[n];
}
```

Exercice: éviter d'utiliser un tableau

Commentaires

- Pas de paraphrase de code
- Commentaires multi-lignes
  - commencent par /*
  - finissent au PREMIER */
- Commentaires mono-lignes (depuis c99)
  - Commencent par //
  - finissent à la fin de la ligne

Règles de déclaration et d'appel

- Toute fonction ne peut appeler que des fonctions déclarées avant elle ou elle-même.

```
... f1 (..) {
...
}
... f2 (...) {
...
}
... f3 (...) {
...
}
int main (int argc, char *argv[]) {
...
}
```

  la fonction main peut appeler f1,f2,f3 (et elle-même, mais ça n'a pas beaucoup de sens)
  la fonction f3 peut appeler f1,f2,f3
  la fonction f2 peut appeler f1, f2
  la fonction f1 peut appeler f1

- Lorsqu'une fonction s'appelle elle-même, on dit qu'elle est "récursive".

Déclarations en "avance"

- Règle précédente contraignante
- Solution : Prototype

  En début de programme on donne le type de chaque fonction , son nom, le nombre et les types des arguments
- Information suffisante pour le compilateur.

```
int pair(int n);                          /*Prototype */
int impair(int n);

int pair(int n) {
if (n == 0) return 1;
return impair(n - 1); }                   /*Appel avant définition*/

int impair(int n) {                       /*Définition de la fonction */
if (n == 0) return 0;
return pair(n - 1); }
```

Exemple (maxf.c)

```
#include <stdio.h>
int max(int a, int b) {
  if (a > b) // Parenthèses OBLIGATOIRES
    return a;
  else
    return b;
}

int maxTableau(int tableau[],
                        int nbElements) {
  int resultat = tableau[0];
  int i = 1;
  while (i < nbElements) {
    resultat = max(resultat, tableau[i]);
    i = i + 1;
  }
  return resultat;
}
```

```
int main(int argc, char **argv) {
  int tab[] = {10, 20, 42, 40};
  int res = maxTableau(tab, 4);
  printf("Max du tableau: %i\n", res);
  return 0;
}
```

Exercices

- Dans l'exemple maxf.c:
  - Rajouter une fonction minTableau() qui retourne la valeur minimale du tableau
  - Rajouter une fonction secondTableau() qui retourne la 2nde valeur la plus grande du tableau (peut être égale à la valeur maximale)
- Écrire un programme (incseq.c) qui affiche toutes les séquences croissantes d'un tableau
  - Exemple: int tab[] = {21, 10, 10, 42, 40, 11, 2, 12, 50, 1};

```
ex/$ ./incseq
21
10 10 42
40
11
2 12 50
1
```

COURS 5-6: TEXTE

Type caractère
Chaînes de caractères
Ligne de commande
Fonction atoi()

Type caractère

- Caractère : Symboles et alphanumériques (?,$,a,z,1,9) + caractères spéciaux (retour à la ligne, beep, etc..)
- Un caractère est représenté sur un octet (8 bits) suivant la table ASCII (American Standard Code for Information Interchange)
- ex : 'a' = 97_{10} = 61_{16} = 0110 0001_2
- Table ASCII
  ex : code ASCII du 'A' = 65
  'A'<'B'<……..< 'Z'
  '0'<'1'<'2'<…..<'9'
  'a'<'b'<…….<'z'
- Déclaration de variable de type caractère
  ```
  char c;
  c = 'a';
  ```
- Constante de type caractère
  ```
  #define caractere_a 'a'
  ```

Code ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.asciitable.com

Type caractère

- Caractères spéciaux (retour à la ligne, tabulation etc..)
- Exemple :

  retour à la ligne = CR = code ASCII 13

  char retour;

  retour = 13; ou bien retour = '\n';
- Conventions

  ‘\n’ : retour à la ligne

  ‘\t’ : tabulation

  ‘\f’ : nouvelle page

  ‘\’’ : apostrophe

  ‘\0’ : caractère nul

Chaînes de caractères

- En C, pas de type prédéfini chaîne de caractères. En pratique on utilise des tableaux de caractères.
- Convention : le dernier caractère utile est suivi du caractère ‘\0’ (de code ascii 0)
- Exemples :

  char t[10]; (9 caractères max, puisque une case occupée par \0;

  strcpy(t,"abcdefghi")

'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'	'i'	0

  chaque lettre est accessible par l'indice

  char t[12];

  strcpy(t,"abcdefghi");

'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'	'i'	0	?	?

- Initialisation

  char t[12]= "abcdefghi"; ou char t[]= "abcdefghi";

- Notation avec pointeurs:

  char *t= "abcdefghi";

Fonctions de manipulations de chaînes de caractères

```
#include <string.h>
char s1[] = "abcd", s2[20];
s2 = s1 ; incorrect
strcpy(s2, s1);

void mystrcpy(char *dest, char *source){
  int i = 0;
  while(source[i] != '\0') {
     dest[i] = source[i];
     i = i + 1;
   }
  dest[i] = source[i]; // ne pas oublier de copier le zero!
}
```

Fonctions de manipulations de chaînes de caractères

- Toutes les fonctions de manipulation de chaines de caractères suivent ce gabarit.
- strlen(char * s) : donne la longueur de s, le 0 final non compris
- strcat (char * s1, char * s2): ajoute s2 à la fin de s1
- strcmp(char * s1, char * s2): compare les chaines suivant l'ordre lexicographique. Valeur de retour :

  >0 si s1 > s2

  =0 si s1 = s2

  <0 si s1 < s2

  "aa" vient avant "b"

  "aa" vient avant "ab"

  "abc" vient avant "abcd"

et bien d'autres fonctions (voir string.h)

Exercice

- Mise en œuvre des fonctions précédentes (string.c)
  - int mystrlen(char *s);
    retourne la longueur de s, le 0 final non compris
  - void mystrcat(char * s1, char * s2);
    ajoute s2 à la fin de s1
  - int mystrcmp(char * s1, char * s2);
    retourne :
    - >0 si s1 > s2
    - =0 si s1 = s2
    - <0 si s1 < s2

La ligne de commande

```
// File: sum.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int i, sum = 0;
  for (i = 1; i < argc; i = i + 1)
    sum = sum + atoi(argv[i]);
  printf("La somme du programme %s est : %i\n", argv[0], sum);
}
```

Nombre d'arguments

Les arguments
NB: Le premier est **toujours** le nom du programme.

ex/$ ./sum 3 4 6
La somme du programme ./sum est : 13

Ecriture

- 3 fonctions de base : putchar(), puts() et printf()
  - putchar(caractère): écrit un caractère
  - puts(chaîne): écrit une chaîne et passe à la ligne suivante

- Exemple

```
char c='a';
putchar ( c );
putchar ('\n');
puts("bla");
puts(".");
```

Affichage :

```
a
bla
.
```

Printf()

- Format

  printf(format,arg1,arg2,…..,argn);

  les arguments sont des valeurs d'expressions à imprimer
  le format donne le texte fixe et le mode de décodage des arguments
  le format est une chaîne de caractères

- Exemples :

```
printf("bon"); printf("jour");printf("\n");        bonjour
int i=1, j=1;
printf("i=%i\n", i);                               i=1
printf("%i%i%i\n", i, j, i+j);                     112
printf("%i  %i  %i\n", i, j, i+j);                 1  1  2
float x=3.0;
printf("%f  %i\n", x, i);                          3.000000  1
printf("%i\n%i\n", i, i+j);                        1
                                                   2
```

Printf

- Caractères spéciaux du format

  %i (ou %d) : imprime les entiers

  %f : imprime les réels avec 6 chiffres de partie décimale

  %e : imprime les réels en notation exponentielle

  %c : imprime un caractère

  %s : imprime une chaîne de caractères jusqu'à rencontrer le caractère de fin de chaine 0

  …..

  \n : saut à la ligne

  \t : tabulation

  ….

Printf : mises en forme

- Forçage du nombre de caractères

  entiers :

  %5i l'entier est imprimé sur 5 caractères au moins (blancs) avec cadrage à droite

  %-5i l'entier est imprimé sur 5 caractères au moins (blancs) avec cadrage à gauche

  réels :

  %10f le réel est imprimé sur 10 caractères au moins (en tout) avec 6 chiffres en partie décimale (cadrage à droite)

  %-10f idem + cadrage à gauche

  limitation de la partie décimale

  %20.3f le réel est imprimé sur 20 caractères (en tout) avec 3 chiffres en partie décimale

Printf

- La valeur de retour du printf est le nombre de caractères écrits, ou une valeur négative si il y a eu un problème.

- Exemple :

```
int a,x;
a=32;
x = printf ("%i\n",a);
printf ("%i\n",x);   ->  3
```

Autres fonctions d'E/S

- Beaucoup d'autre fonctions d'E/S

  voir "stdio.h" (cf. « man stdio »)

  – atoi, atof: convertissent une chaîne en entier/flottant
  – gets: lit une chaîne de caractères

  Exemple :

```
#include "stdio.h"
int main(int argc, char *argv[]) {
    char ligne[80];
    gets(ligne);
    puts(ligne);
    int i = atoi(argv[1]);
}
```

Exercice (command.c)

- Programme qui prend N mots sur la ligne de commande (N > 1) et écrit le mot le plus long et le mot le plus court (command.c)

COURS 7-8: OPÉRATEURS (1)

Types entiers
Expressions
Opérateurs d’affectation
Opérateurs booléens

Les entiers : entiers naturels

- Type « unsigned int »: codage sur 2 ou 4 octets suivant le calculateur

$$x = x_{n-1}...x_1x_0 = \sum_{i=0}^{n-1} x_i 2^i, x_i \in \{0,1\}, \text{n} = 16 \text{ ou } 32$$

- Sur deux octets on peut coder les nombres de 0 à 2^{16}-1 (0 à 65535)
- Nombre représenté en base 2, les bits sont rangés dans des cellules correspondant à leur poids, on complète à gauche par des 0
- Exemple :

  13 = 8 + 4 +1 = $1*2^3+1*2^2+0*2^1+1*2^0$

  0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1
- Déclarations d'une variable entier naturel:

```
unsigned char uc = 'A';  // 8 bits
unsigned short int us = (unsigned short)65U; // min 16 bits
unsigned int ui = 65U; // min 16 bits
unsigned long int ul = 65UL; // min 32 bits
unsigned long long int ull = 65ULL; // min 64 bits
printf("uc=%hhu us=%hu ui=%u ul=%lu ull=%llu\n", uc, us, ui, ul, ull);
```

Type entier relatif

- Codage complément à 2

$$x = x_{n-1}x_{n-2}\ldots x_1x_0 = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i, x_i \in \{0,1\}$$

- Si $x_{n-1} = 0$: nombres positifs, de 0 à $2^{n-1}-1$
- Si $x_{n-1} = 1$: nombres négatifs, de $-2^{n-1}+0$ à $-2^{n-1}+(2^{n-1}-1) = -1$
- Le plus grand entier positif est $2^{n-1}-1$
- Le plus petit entier négatif est -2^{n-1}
- Représentation non symétrique : le plus petit nombre n'a pas d'opposé : sur n bits
- Sur 16 bits (2 octets)

  $-32768 \leq x \leq 32767$
- Sur 32 bits

  $-2147483648 \leq x \leq 2147483647$

Somme des puissances de 2

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

$$2^n = 2^{1+(n-1)} = 2^1 * 2^{n-1} = 2^{n-1} + 2^{n-1}$$

$$2^n = 2^{n-1} + 2^{n-1}$$

$$2^n = 2^{n-1} + 2^{n-2} + 2^{n-2}$$

$$\dots$$

$$2^n = 2^{n-1} + 2^{n-2} + \dots + 2^2 + 2^1 + 2^1$$

$$2^n = 2^{n-1} + 2^{n-2} + \dots + 2^2 + 2^1 + 2^0 + 2^0$$

$$2^{n-1} - 1 = 2^{n-2} + 2^{n-3} + \dots + 2^2 + 2^1 + 2^0$$

Type entier relatif

- Codage complément à 2

$$x = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i, x_i \in \{0,1\}$$

- Exemple sur 16 bits

  +5 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 0000 0000 0000 0101

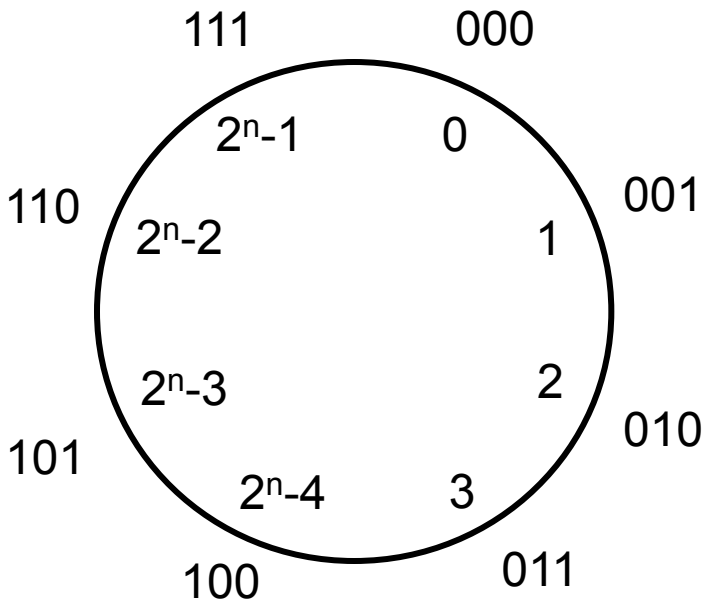
  -3 = -32768 + 32765

  = -2^{15}+ 2^{14} + 2^{13} + 2^{12} + 2^{11} + 2^{10} + 2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 1.2^2 + 0. 2^1 + 1.2^0

  = 1111 1111 1111 1101

Complément à 2

- Exemple: pour n=3 bits



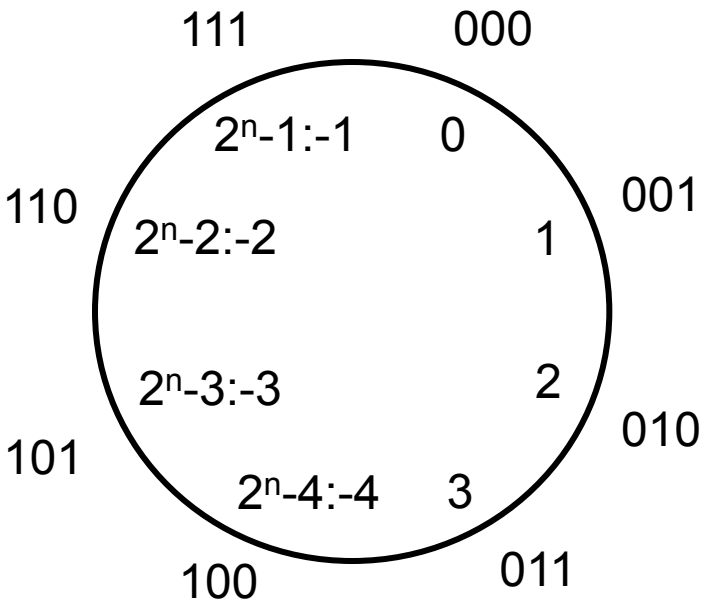
Notation:
~x = complément à 1 de x

- $\sim x + x = 2^n - 1$
- $\sim x = 2^n - (x + 1)$
- $\sim(x-1) = 2^n - x$

$$x_{n-1}\ldots x_1 x_0 + \overline{x}_{n-1}\ldots\overline{x}_1\overline{x}_0 = 1_{n-1}\ldots 1_1 1_0 = 2^n - 1$$

Complément à 2

- Exemple: pour n=3 bits



Notation:
~x = complément à 1 de x

- $\sim x + x = 2^n - 1$
- $\mathbf{\sim x} = 2^n - (x + 1)$: **-(x+1)**
- $\mathbf{\sim(x-1)} = 2^n - x$: **-x**

$$x_{n-1}\ldots x_1 x_0 + \overline{x}_{n-1}\ldots\overline{x}_1\overline{x}_0 = 1_{n-1}\ldots 1_1 1_0 = 2^n - 1$$

Codage complément à 2

- Remarques

  1/ Complément à 2 de x = (Complément à 1 de x) + 1 -x : ~(x-1)

  représentation de –3 ?

  -3 : c1(3-1) = c1(2)

  2 = 0000 0000 0000 0010

  c(2) = 1111 1111 1111 1101

  2/ Représentation 16 bits => 32 bits

  x > 0 = 0000 0000 0000 0011 => x = 0000 0000 0000 0000 0000 0000 0000 0011

  x < 0 = 1111 1111 1111 1101 => x = 1111 1111 1111 1111 1111 1111 1111 1101

- Déclaration

```
char c = -'A';
short int s = -65; // min 2 octets
int i = -65; // min 2 octets
long int l = -65L; // 4 octets
long long int ll = -65LL; // 8 octets
printf("c=%hhi s=%hi i=%i l=%li ll=%lli\n", c, s, i, l, ll);
```

Taille des objets (taille.c)

- sizeof(...) : donne la taille d'un objet, en octets

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  char c = 42;
  short s = 42;
  int i = 42;
  long l = 42;
  long long ll = 42;
  printf("c: %zu\ns: %zu\ni: %zu\nl: %zu\nll: %lu\n",
        sizeof(c), sizeof(s), sizeof(i), sizeof(l), sizeof(ll));
  return 0;
}
```

Opérateur de taille : sizeof

- Donne la taille de l'implantation
- 2 syntaxes

  1/ sizeof (expression)

  Exemple :

  int i,j ;

  char tab[100];

  j = sizeof (i); -> 2 ou 4 (octets)

  j = sizeof(tab) -> 100

  2/ sizeof (type)

  Exemples :

  int n;

  n = sizeof(int), -> 2 ou 4 (octets)

  n = sizeof(char[100]) -> 100

Conversion de type

- Conversion explicite : (type) expression

  Exemple :

  ```
  int a; float x; char c;
  a=2;
  x=(float) a;
  x=2.3;
  a= (int) (x+1);
  a = 98;
  c = (char) a; // -> c='b'
  ```

- Conversion implicite

  Exemple :

  ```
  int a; float x; char c;
  a=2;
  x= a;
  x=2.3;
  a= x+1;
  a = 98;
  c = a; -> c='b'
  ```

Conversion de types

- Exemples :

```
char c; int i; float f;
// conversion entier vers char.
c=98; // implicite : c prend le code ASCII 98 c-à-d ’b'
c = (char) 98;  // explicite plus propre
// char vers entier
i= 'a' ; // i prend la valeur 97
i= (int) 'a' ; //plus propre
// entier vers réel
f=3; // f prend la valeur 3.0;
f=(float) 3; //+ propre
// réel vers entier, attention : troncature
i = 3.6; // i prend la valeur 3
i= -3.6; // i prend la valeur -3
i = (int) 3.6; // + propre
```

Conversion de types : application

- Passer au caractère suivant

```
char c;
c ='a';
c = c+1; // calcul fait en entier puis résultat converti en char
c = (char) ((int) c+1) ; //+ propre
```

- Conversions majuscule <-> minuscule

```
char c;
c='t';
// conversion en Majuscule
c=c-32; // c contient 'T'
// ou mieux
c=c-('a'-'A');
c=c+1; // c contient 'U'
// conversion en minuscule
c=c+32;
ou c=c+('a'-'A') // c contient ‘u’
```

Exercice

- Écrire un programme (complement.c) qui calcule le nombre de bits d'un entier, et sa forme binaire:
  - int nbits(unsigned int n);
  - void bin(unsigned int n, char buf[]);
  - void reverse(char buf[], int len);

ex/$ ./complement 8

8 has 4 bits

8 in reversed binary = 0001

8 in binary = 1000

ex/$ ./complement -3

-3 has 32 bits

-3 in reversed binary = 10111111111111111111111111111111

-3 in binary = 11111111111111111111111111111101

Expressions

- Une expression représente:
  - une donnée élémentaire : constante, variable
  - l'application d'un opérateur: un élément de tableau, conversion de type, opérateurs arithmétiques, de taille, relationnels et logiques, affectation, bit-à-bit, conditionnels, adresse, appel de fonction, …
- Exemples

  3

  a+b

  x=y

  c = a+b

  x <= y

  x == y

  i++

  sin(3.14)
- Toute expression a une valeur

Opérateurs arithmétiques

- Opérateurs bi-opérandes

  + , -

  * , / , % (modulo)

  Les opérandes doivent être des valeurs numériques.

  entier OP_2 entier -> résultat entier

  réel OP_2 réel -> résultat réel

  entier OP_2 réel, réel OP_2 entier -> résultat réel

- Exemples

int a,b;		float a,b;	
a=10; b= 3		a=12.6; b= 3.0	
a+b	13	a+b	15.6
a-b	7	a-b	9.6
a*b	30	a*b	37.8
a/b	3 (division euclidienne)	a/b	4.2 (division réelle)
a%b	1	a%b	erreur de syntaxe

Opérateurs arithmétiques

- Opérateur % :
  - int a; float x;

  (a+x) % 4 incorrect. ((int) (a+x))%4 correct
  - si l'un des opérandes est négatif, le résultat est négatif.
- Si l'un des opérandes est de type caractère, c'est la valeur du code ASCII qui est prise (conversion implicite char vers int ou float)

Exemple :

```
char c = 'a';
c = c+1;  => c = 'b'
```

Opérateurs arithmétiques

- Opérateurs unaires (un opérande)

  a/ signe : + , -

  exemple : max(a, -a);

  b/ incrémentation, décrémentation : ++ (+1) , -- (-1)

  exemple :

```
int i =1;
++i;
printf("%i", i) ; // -> 2;
```

  Syntaxes : ++i ou i++

  ++i : la valeur de i est d'abord incrémenté, la valeur résultat est utilisée dans l'expression courante

  i++ : la valeur courante de i est utilisée dans l'expression courante, puis i est incrémenté

++ et --

- Exemples

```
i=1;
printf("i=%i\n",i);  //-> i=1
printf("i=%i\n",++i);  //-> i=2
printf("i=%i\n",i);  //-> i=2
```

```
i=1;
printf("i=%i\n",i);  //-> i=1
printf("i=%i\n",i++);  //-> i=1
printf("i=%i\n",i);  //-> i=2
```

- Conclusions :

  1/ apprendre la règle (pour comprendre des programmes)

  2/ éviter de mélanger calculs et effets de bord :

  x=y+z++; // à éviter

  x++; // pas de risque: raccourci pour x = x + 1

  stack[++top]=val; // peut avoir du sens, si bien maîtrisé…

Opérateurs d'affectation

- Affectation simple

  syntaxe : var = expr

  la valeur de l'expression expr est stockée dans la mémoire à l'endroit réservé pour la variable var

  NB: la valeur de l'expression « var = expr » est cette même valeur !

  Exemples :

  a = 2; b=1; c=0;

  a = b+c;

  Attention : ne pas confondre affectation et test d'égalité !

  if (a =1) instruction1; else instruction2;

  -> L'instruction1 est toujours déclenchée.

  a = b = 3; // évaluation de droite à gauche: a = (b = 3);

Opérateurs d'affectation

- Affectation et opération : +=, -=, *=, /=, %=,<<= , >>=, &=, |=, ^=

Syntaxe : var OP_2= expr

équivalent à : var = var OP_2 expr

Exemple :

```
int i;
i = 3;
i += 2;  // équivalent à: i = i + 2
printf("%i\n",i); // -> 5
```

Opérateurs relationnels et logiques

- Valeur logique :
  0 : faux
  ≠ 0 : vrai
  exemple : if (3) traitement1 ; else traitement 2;
  équivalent à : traitement1;

- Relationnels : >= , > , == , <, <= , !=
- La valeur de l'expression est 1 si l'expression est vraie, 0 si elle est fausse
  Exemple : 2 < 3 vaut 1 , 2 > 3 vaut 0

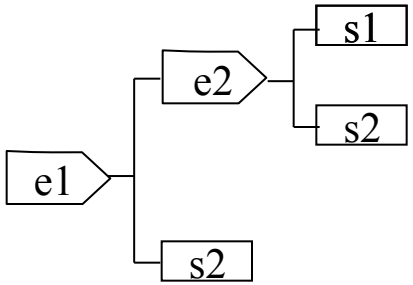
- Attention à la confusion : test d'égalité == et l'affectation =
  ex : if (x=0) traitement 1; // au lieu de x==0
  else traitement 2;
  Conséquence: non seulement le traitement 1 ne sera jamais exécuté mais en plus x vaudra 0 quelle que soit sa valeur initiale

- Logiques : && "et" logique , || "ou" logique, ! "non" logique
  Dans l'évaluation de l'expression, 0 est considéré comme la valeur logique "faux", toute valeur ≠ 0 comme la valeur logique "vraie"
  La valeur de l'expression est 1 ou 0
  Exemples:
  2 && 0 vaut 0 et donc est faux
  2 || 0 vaut 1 et donc est vrai
  !0 vaut 1 !4 vaut 0

Compléments sur les opérateurs logiques

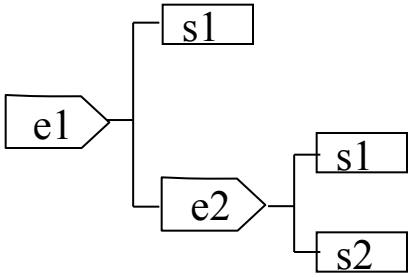
- &&, || ⇔ if imbriqués (sémantique « paresseuse »)

```
if(e1&& e2) s1;
else s2;
```




```
if(e1) if (e2) s1;
       else s2;
else s2;
```

```
if(e1 || e2) s1;
else s2;
```



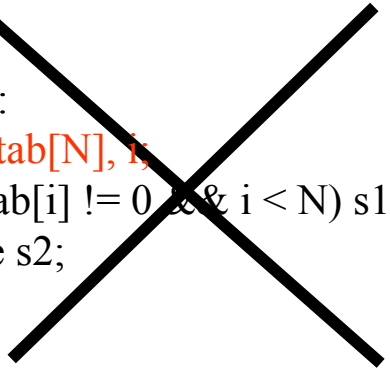

```
if(e1) s1;
else if(e2) s1;
     else s2;
```

Ex :

```
int tab[N], i;
if(i < N && tab[i] != 0) s1;
else s2;
```

Ex :

```
int tab[N], i;
if(tab[i] != 0 && i < N) s1;
else s2;
```



Exercice

- Calcul des années bissextiles (bissextile.c):
  - soit divisibles par 4 mais non divisibles par 100 ;
  - soit divisibles par 400.
- Exemples:

```
ex/$ ./bissextile
usage: ./bissextile year
ex/$ ./bissextile 2016
L'annee 2016 est bissextile
ex/$ ./bissextile 2100
L'annee 2100 n'est pas bissextile
ex/$ ./bissextile 2000
L'annee 2000 est bissextile
```

COURS 9-10: OPÉRATEURS (2)

Opérateur ternaire

Opérateurs bits à bits

Constantes numériques

Opérations sur les flottants

Opérateurs bit à bit

- Opèrent sur les représentations des valeurs
- & et , | ou, ^ ou-exclusif, ~ complément à 1 ,
- << décalage à gauche, >> décalage à droite,
- Attention : & ≠ &&
- Exemples

5	0000 0000 0000 0101
20	0000 0000 0001 0100
5 & 20	0000 0000 0000 0100 => 5 & 20 => 4
5 | 20	0000 0000 0001 0101 => 5 | 20 => 21
5 ^ 20	0000 0000 0001 0001 => 5 ^ 20 => 17
~5	1111 1111 1111 1010 => -6

- Affectation/bit-à-bit : &=, |=, ^=, ~=

Décalages

- Décalages
  - à gauche a << b : a est décalé à gauche de b bits (les bits ajoutés valent 0)

5 0000 0000 0000 0101

5 << 2 0000 0000 0001 0100 20

un décalage d'une position à gauche correspond à une multiplication par 2

  - à droite a >>b : a est décalé à droite de b bits (les bits insérés valent le bit de poids fort, si le type est signé; sinon 0)

14 0000 0000 0000 1110

14 >> 2 0000 0000 0000 0011 3

-6 1111 1111 1111 1010

-6 >> 1 1111 1111 1111 1101 -3

un décalage d'une position à droite correspond à une division par 2 (en respectant le signe, s'il existe)

Constantes numériques

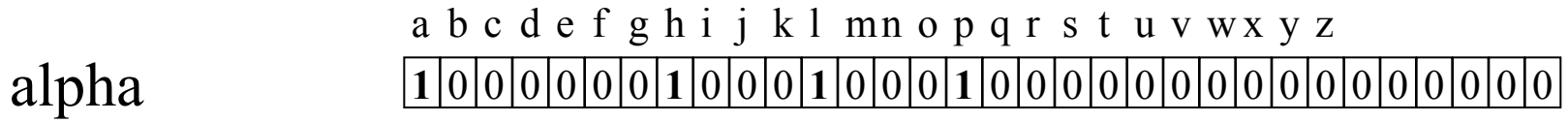
- Chaque type numérique fournit ses propres constantes.
  - entier décimal — int — 1234
  - entier octal — int — 02322
  - entier hexadécimal — int — 0x4d2
  - entier héxadécimal long — long int — 0x4d2**L**
  - flottant double précision — double — 12.3, 1233e-1
  - flottant — float — 12.3F, 1233e-1**F**

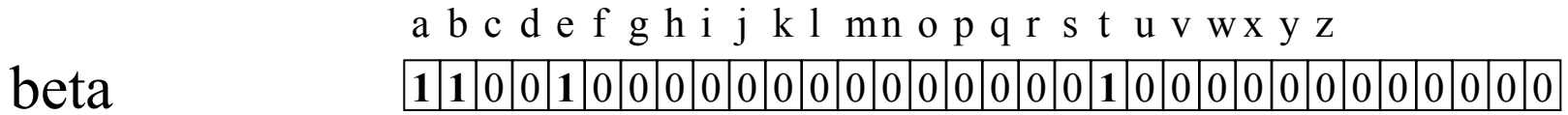
- Des règles déterminent le type d'une constante entière. Par exemple, une constante décimale est du premier type qui peut la représenter, parmi int, long ou long long.

Exemple

- Programme (lettersets.c) qui effectue des opérations sur des ensembles de lettres de l'alphabet:

```
ex/$ ./lettersets alpha + beta
abehlpt
ex/$ ./lettersets alpha x beta
a
ex/$ ./lettersets alpha - beta
hlp
```





lettersets.c (1/2)

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

int mask(char c) {
  return (1U << 31) >> (c - 'a');
}

int setof(char *s) {
```


```
}
```

```
void print_set(int set) {
```


```
}

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


```
}
```

Exercice

- Optimiser le programme précédent en pré-calculant le masque de chaque lettre

Opérateur conditionnel

- Syntaxe

  expression1 ? expression2 : expression3

  à peu près équivalent à :

  ```
  if (expression1) expression2; else expression3;
  ```

- Exemple :

  ```
  maximum = (x>y) ? x : y;
  ```

  ```
  if (x>y) maximum = x ; else maximum = y;
  ```

- Conseil : ne pas y inclure des effets de bords (affectations)

  x > y ? ~~maximum = x~~ : maximum = y;

Opérateurs divers

- ( ) : force l'ordre des calculs

  ex : 1 + 2 * 3 -> 7

  (1+2) * 3 -> 9

- [ ] pour les tableaux

  t[2] est le 3ème élément de t

- Liés aux pointeurs (étudiés plus tard) :
  - &x (adresse de x, à ne pas confondre avec le et par bits)
  - *x (dont l'adresse est x, à ne pas confondre avec la multiplication)
  - -> et . (opérateurs sur structures)

Priorité des opérateurs

Priorité	Opérateurs	Description	Associativité
15	() [ ] -> . ++ --	opérateurs postfix	N/A
14	++ --	incrément/décrément (prefix)	N/A
	~	complément à un (bit à bit)	
	!	non unaire	
	& *	adresse et valeur (pointeurs)	
	(*type*)	conversion de type (cast)	
	+-	plus/moins unaire (signe)	
13	* / %	opérations arithmétiques	→
12	+ -	opérateurs binaires	→
11	<< >>	décalage bit à bit	→
10	< <= > >=	opérateur relationnels	→
9	== !=	""	→
8	&	et bit à bit	→
7	^	ou exclusif bit à bit	→
6	|	ou bit à bit	→
5	&&	et logique	→
4	||	ou logique	→
3	?:	conditionnel	N/A
2	= += -= *= /= %= >>= <<= &= ^= |=	assignations	←
1	,	séquence	→

Priorité des opérateurs

- « Associativité » (ordre d'évaluation):
  - Opérateurs binaires:
    - → x+y+z ⇔ (x+y)+z
    - ← x=y=z ⇔ x=(y=z)
  - Opérateurs unaires: N/A
    - Prefix f(x)[y]
    - Postfix -++x
- Priorité:

```
-i++  // ⇔ -(i++)
a – b /c *d
(a-b) / (c-d)
i = j = k = 0;

int a=1, b[]={0};
! --a == ++ b [0]   // => 1 (vrai)
```

Priorité des opérateurs (exercices)

```
main(){
int x, y , z;
x = 2;
x += 3 + 2; printf("%d\n",x);
x -= y = z = 4; printf("%d%d%d\n",x,y,z);
x = y == z; printf("%d%d%d\n",x,y,z);
x == (y = z); printf("%d%d%d\n",x,y,z);

x = 3; y = 2 ; z = 1;
x = x && y || z ; printf("%d\n", x);
printf ("%d\n", x || ! y && z);
x = y = 0;
z = x++ - 1; printf ("%d, %d\n", x, z);
z += -x++ + ++y; printf ("%d, %d\n", x, z);

x = 1 ; y = 1;
printf("%d\n", !x | x);
printf("%d\n", ~x | x);
printf("%d\n",  x ^ x);
x <<= 2 + 1 ; printf("%d\n", x);
```

```
x = 1 ; y = 1; z = 1;
x += y += z;
printf("%d\n", x < y ? y : x) ;
printf("%d\n", x < y ? x++ : y++) ;
printf("%d, %d\n", x , y);
printf("%d\n", z += x < y ? x++ : y++) ;

printf("%d, %d\n", y , z);

x = 3; y = z = 4;
printf("%d\n",( z >=  y >= x) ? 1 : 0) ;
printf("%d\n", z >=  y  && y >= x ) ;
x = y = z = 0;
}
```

Type réel

- Déclaration

```
float x,y;
x = 3.14;
y = 2.0e+3;
```

- Représentation selon norme IEEE simple précision
  (mantisse et exposant)
  – Capacité: $10^{-38} < x < 10^{38}$
- Les différentes précisions :

```
float f = 3.14F;  // codé sur 4 octets
double d = 3.14;  // codé sur 8 octets
long double ld = 3.14L; // codé sur min 8 octets
printf("f=%f, d=%f ld=%Lf\n", f, d, ld);
```

- La précision par défaut est double !

Affichage des flottants (printf.c)

L	f	e	g
Modificateur pour long double	notation décimale	notation scientifique	choisit la notation la plus adaptée
	0.0 .0 0.	1e0 1.0e0 .0e1	

```
float f = 1.0 / 3.0;
printf("%f %e %g\n", f, f, f);
f = 1 / 3;
printf("%f %e %g\n", f, f, f);
f = 1.0 / 3;
printf("%f %e %g\n", f, f, f);
double d = 1.0 / 3.0;
printf("%5.2f\n", d);
```

```
0.333333 3.333333e-01 0.333333

0.000000 0.000000e+00 0

0.333333 3.333333e-01 0.333333

 0.33
```

Nombre de caractères minimum à afficher

Nombre maximum de décimales à afficher

FONCTIONS SUR LES FLOTTANTS

- #include <math.h>
- NB: ajouter l'option de compilation « -lm » (lib math)

  - Puissance : pow(), sqrt()
  - Arrondis : floor(), ceil()
  - Troncature : trunc()
  - Valeur absolue: fabs() (notez le f au début!)
- Par défaut : argument(s) & résultat = double
  - Version float : suivie d'un f ; ex: truncf()
  - Version long double: suivie d'un l ; ex: truncl()

- Pour convertir des chaines de charactères en réels:
  - atof() // n'oublier pas le #include <stdlib.h>

Exemple (distance.c)

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

int main(int argc, char *argv[]) {
  if(argc != 5) {
    printf("usage: %s x1 y1 x2 y2\n", argv[0]);
    return 1;
  }
  double x1 = atof(argv[1]);
  double y1 = atof(argv[2]);
  double x2 = atof(argv[3]);
  double y2 = atof(argv[4]);
  printf("Distance entre (%.2f,%.2f) et (%.2f,%.2f) = %f\n",
         x1, y1, x2, y2, sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1)));
}
```

Exercice

- Écrire un programme (average.c) qui calcule
  - La moyenne d'une série de valeurs réelles
    float average(float t[], int n);
  - L'écart type de la série
    float stddev(float t[], int n);

$$\sigma_X := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

- Exemples:

ex/$ ./average 1 2 3 4 5

la moyenne des valeurs est 3.000000

l'ecart type est 1.414214

ex/$ ./average 1 2 3

la moyenne des valeurs est 2.000000

l'ecart type est 0.816497

COURS 11-12: POINTEURS (1)

Pointeurs

Exemple: scanf

Exemple: boutisme

Structures

Structures & pointeurs

Pointeurs (Références)

- Type des adresses mémoire
- Permet d'accéder à cette adresse (R/W)
- Les pointeurs sont typés (adresse d'un certain type)
- Convention: NULL (0) => Absence de valeur
- Toujours la même taille (en général sizeof(int))
- Utilisations
  - « Passage par référence » d'un argument à une fonction (= variable à modifier par la fonction)
  - « Retourner » plus d'une valeur (tableau, structure)
  - Effectuer des traitements géneriques (ex: manipuler des objets de taille variable, passer un nombre variable d'arguments)
  - Définir des structures de données « chaînées »
  - ...

Opérateurs d'adressage

- « Adresse de » : &

  Syntaxe : &variable , donne l'adresse mémoire de la variable

  Exemple :

  ```
  adr = &i;
  ```

  ne pas confondre avec le "et" bit à bit 

- « L'objet à l'adresse » : *

  Syntaxe : *expression : donne l'objet en mémoire à l'adresse donnée par l'expression

  Exemple :

  ```
  i=1;
  adr = &i;
  printf("%i", *adr);  -> 1
  ```

Adresses et pointeurs

- Déclaration de pointeurs
- Syntaxe

```
type * variable    /* type : char, float, int, … */
ex :
int * pi;          /* pi est un pointeur d'entier */
float * px;    /* px est un pointeur de réel */
char * pc ;   /* pc est un pointeur de caractère */
```

- Interprétations

```
int * pi ;
```

1. L'objet à l'adresse pi est un entier => pi est un pointeur d'entier

```
int * pi ;
```

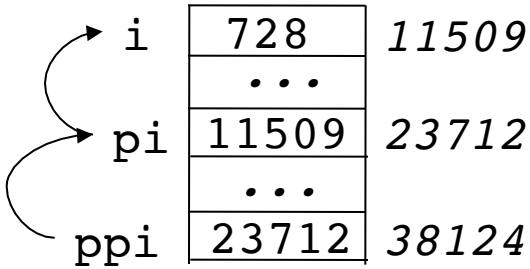
2. pi est une adresse d'entier

Adresse des pointeurs (!)

- Types de pointeurs

  type * variable // type = int, float, char, **pointeur**, …

```
int i;
int * pi;
int * * ppi ;/* adresse d'adresse d'entiers */
pi= &i;
ppi = &pi;
```



- Pas de pointeurs de tableau (le nom du tableau = adresse constante !)

```
int t[10];
int * * ppi = &t; // erreur
int * pi = t; // ok
```

Pointeurs et opérations

- Sur variable pointée : toute opération valide sur le type

```
float x,y,* px; ...
px= &x;
y = sinus (*px);
(*px) += PI / 2
```

- Sur pointeurs

Remarque : La valeur d'un pointeur n'a pas d'intérêt en elle-même, d'autant qu'elle change à chaque exécution.

1/ affectation

```
int * pi, * pj;
short * ps;
...
pi = pj;
ps = (short *) pi; /*conversion de type de pointeur */
```

Pointeurs et opérations

2/ comparaison d'égalité, d'inégalité

```
int * pi, * pj;
...
if (pi==pj) ......;
```

3/ comparaison >, < , ... mais plus rarement d'intérêt.

```
if (pi < pj) ......;
```

4/ addition et soustractions (« arithmétique des pointeurs »)

```
int t[10]; // t est une adresse d'entier
int i = *(t+2); // ⇔ i = t[2]
```

Exemple: inverser deux valeurs (swap.c)

```
#include <stdio.h>

void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char *argv[]){
  int a = 42;
  int b = 24;
  printf("avant swap %i %i\n", a, b);
  swap(&a, &b);
  printf("après swap %i %i\n", a, b);
}
```

```
void bad_swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

scanf

- Saisie de données : scanf(…)
  - Format identique à printf(…)
  - !!! ATTENTION au type des paramètres: pointeurs !!!
  - Les valeurs lues sont délimitées par des espaces (!)
  - Retourne le nombre **de valeurs** lues
    (Rappel: printf(…) retourne le nombre de caractères affichés)

```
int main(int argc, char **argv) {
  int a, res;
  float b;
  char s[50];
  res = scanf("%i %g %s", &a, &b, s);
  printf("lu: %i variables: a=%i, b=%g et s=%s\n", res, a, b, s);
  return 0;
}
```

Application: boutisme (endianness)

- A partir du moment où il y a échange de données, l'ordre des octets compte
  - Transmission sur le réseau entre deux machines
  - Écriture dans un fichier
- Deux grandes familles (représentation du nombre O3O2O1O0):
  - **Big-Endian**: Octet de poids fort en premier (ordre réseau, Motorola 68k, Java, jpeg, ...): {O3, O2, O1, O0}
  - **Little-Endian**: Octet de poids faible en premier (intel, FAT, ext2, ...): {O0, O1, O2, O3}
  - + Mixed-Endian: {O2, O3, O0, O1}, …
- Il faut donc interpréter les entiers dans le bon ordre

Exemple: endian.c

```
#include <stdio.h>

void print_bytes(char *b, int n) {
  printf(”char b[%i]={", n);
  for(int i = 0; i < n; i++) {
    printf(”0x%x ", *(b+i)); // ou: b[i]
  }
  printf("}\n");
}

int main() {
  int i = 0x01020304;
  printf("i=0x%x\n", i);
  print_bytes((char *)&i, sizeof(i));
}
```

- Programme qui affiche les octets d’un entier, pour tester le boutisme de la machine:

Exemple (machine little endian):

```
> ./endian
i=0x1020304
char b[4]={0x4 0x3 0x2 0x1 }
```

Exemple (machine big endian):

```
> ./endian
i=0x1020304
b[4]={0x1 0x2 0x3 0x4 }
```

Exercice

- Écrire un programme (puissances2.c) qui
  - Définit une fonction fois2() qui prend un entier en argument et le double **sans rien retourner** (type retour = void)
  - Affiche les puissances de 2 entre 2^1 et 2^n, en utilisant la fonction fois2()
- Exemple:

  > ./puissances2 5

  2^1=2

  2^2=4

  2^3=8

  2^4=16

  2^5=32
- Option: tester le débordement de capacité du type entier

Structures

- Déclaration d'une structure : syntaxe

```
struct nomdelastructure {
        typemembre1 nommembre1 ;
        typemembre2 nommembre2 ;
        …
        typemembren nommembren ;
        }
```

- Exemple : compte bancaire

```
struct compte {
       int no_compte ;
       char etat ;
       char nom[80];
       float solde;
       };

struct compte a,b,c; /*déclaration de 3 variables de ce type*/
```

Déclarations de variables

- Différentes façons de déclarer des variables structure

```
struct compte {
       int no_compte ;
       char etat ;
       char nom[80];
       float solde;
       } a, b;   /*déclaration de 2 variables de ce type*/
struct compte c; /*déclaration de 1 variable de ce type*/


struct          { /* le nom de la structure est facultatif */
       int no_compte ;
       char etat ;
       char nom[80];
       float solde;
       } a,b,c;   /*déclaration de variables de ce type ici */
/* mais plus de possibilité de déclarer d'autres variables de
ce type*/
```

Déconseillé

Déclarations de variables

- Autres façons de déclarer des variables structure

```
typedef struct compte {
        int no_compte ;
        char etat ;
        char nom[80];
        float solde;
        } cpt ;
/* cpt est alors un type équivalent à struct  compte*/

cpt a,b,c; /*déclaration de variables de ce type*/
```

Recommandé

- Dans ce cas puisque on ne se sert plus de "struct compte" par la suite

```
typedef struct {
        int no_compte ;
        char etat ;
        char nom[80];
        float solde;
        } cpt ;
```

Structures imbriquées

- Une structure peut être membre d'une autre structure

```
struct date {
        int jour;
        int mois;
        int annee;
        };

struct compte {
        int no_compte ;
        char etat ;
        char nom[80];
        float solde;
        struct date dernier_versement;
        };
```

- Remarque : ordre de déclaration des structures

Structures

- Tableaux de structures

```
struct compte client[100];
```

- La portée du nom d'un membre est limité à la structure dans laquelle il est défini. On peut avoir des membres homonymes dans des structures distinctes.

```
struct s1 {
        float x;
        int y ;
        };

struct s2{
        char x;
        float y;
        };
```

Pas de confusion

Manipulation des structures

- Initialisation à la compilation

```
struct compte {
       int no_compte ;
       char etat ;
       char nom[80];
       float solde;
       struct date dernier_versement;
       };

struct compte c1 = {12345,'i',"Dupond",2000.45,{01,11,2009}};
```

- Accés aux membres : opérateur '.' Syntaxe : variable.membre

```
1/  c1.solde = 3834.56;

2/  struct compte c[100];
    y=c[33].solde;

3/ c1.dernier_versement.jour = 15;
   c[12].dernier_versement.mois = 11;
```

Manipulation des structures

- Sur les structures elles-mêmes
  - Affectation :

```
c[4] = c1
```

  - Pas de comparaison => comparer chaque membre

Structures et pointeurs

- L'adresse de début d'une structure s'obtient à l'aide de l'opérateur &

```
typedef struct {
        int no_compte ;
        char etat ;
        char nom[80];
        float solde;
        struct date dernier_versement;
        } cpt ;
cpt c1 , * pc;
```

- c1 est de type cpt, pc est un pointeur sur une variable de type cpt

```
pc = &c1;
```

- Accés au membres à partir du pointeur

```
*pc.no-compte = ...
```

Incorrect . est plus prioritaire que *

```
(*pc).no-compte = ...
```

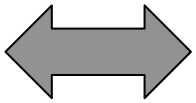
- Raccourci: Opérateur ->

```
pc->no-compte = ...; // équivalent
```

Structures et pointeurs: exemple

```
struct Point {
  int x;
  int y;
};
```

```
void swap(Point *p) {
  int tmp = (*p).x;
  (*p).x = (*p).y;
  (*p).y = tmp;
}
```



```
void swap(Point *p) {
  int tmp = p->x;
  p->x = p->y;
  p->y = tmp;
}
```

Structures et fonctions

- Les membres d'une structure peuvent être passés comme paramètres à des fonctions avec ou sans modification
- Ex1 (sans modification)

```
float ajoute_au_compte(float solde1, float somme1) {
  solde1 = solde1 + somme1;
  return solde1;
}

void main () {
  ......
  cpt c1;
  c1.solde = 0.;
  ajoute_au_compte(c1.solde, 1000.0);
  printf("%f\n", c1.solde);  // -> 0.000000
  c1.solde = ajoute_au_compte(c1.solde, 1000.0);
  printf("%f\n", c1.solde);  // -> 1000.000000
}
```

Structures et fonctions

- Ex2 (avec modification)

```
void ajoute_au_compte(float * solde1, float somme1) {
  *solde1 = *solde1 + somme1;
}

void main () {
  ......
  cpt c1;
  c1.solde = 0.;
  ajoute_au_compte(&(c1.solde), 1000.0);  // ou &c1.solde
  printf("%f\n", c1.solde);  // -> 1000.000000
}
```

Structures et fonctions

- Un argument de fonction peut être de type structure

```
float ajoute_au_compte(cpt c, float somme1) {
  return c.solde + somme1;
}

void main () {
  cpt c1;
  c1.solde = ajoute_au_compte(c1, 1000.0);
  printf("%f\n", c1.solde);  // -> 1000.000000
}
```

- Ou pointeur sur structure

```
void ajoute_au_compte (cpt * c, float somme1) {
  c->solde = c->solde + somme1;
}

void main () {
  cpt c1;
  ajoute_au_compte(&c1, 1000.0);
  printf("%f\n", c1.solde);  // -> 1000.000000
}
```

Structures et fonctions

- La valeur de retour d'une fonction peut être une structure

```
cpt ajoute_au_compte(cpt c, float somme1) {
  c.solde = c.solde + somme1; // c est une copie locale
  return c;
}

void main () {
  ......
  cpt c1;
  c1.solde = 0.;
  c1 = ajoute_au_compte(c1, 1000.0);
  printf("%f\n", c1.solde);  // -> 1000.000000
}
```

Exercice

- Écrire un programme (ingredients.c) qui calcule tous les ingrédients à acheter étant donné N recettes

ex/$ **cat clafoutis.txt**
500 cerises
4 oeufs
50 sucre
125 farine
250 lait
30 beurre
ex/$ **cat 4quarts.txt**
250 beurre
250 farine
250 sucre
4 oeufs
ex/$ **cat clafoutis.txt 4quarts.txt | ./ingredients**
500 cerises
8 oeufs
300 sucre
375 farine
250 lait
280 beurre

```
#define N 20 // nbre max d'ingrédients différents
#define SIZE 32 // taille max d'un nom d'ingrédient
struct ingredient {
  char name[SIZE];
  int quant;
};
struct ingredient tab[N]; // liste d'ingrédients totalisés
int n = 0; // nbre d'ingrédients dans tab[]

int read_ingredient(struct ingredient *s) {…}
void print_ingredient(struct ingredient *s) {…}
void add_ingredient(struct ingredient *s) {...}

int main(int argc, char *argv[]) {
  struct ingredient ingredient1;
  while (read_ingredient(&ingredient1)) {
    add_ingredient(&ingredient1);
  }
  for (int i = 0; i < n; i++) {
    print_ingredient(&tab[i]);
  }
}
```

COURS 13-14: POINTEURS (2)

Pointeurs et tableaux

Arithmétique des pointeurs

Allocation dynamique

Les tableaux: rappels

- Tableau = regroupement de données de même type sous un même nom, accessibles par un indice (0,..,dim-1)

- Initialisation à la définition:

```
int t[10] = {1,2,3,4,5,6,7,8,9,10};
float x[4] = {0.,0.25,3.14,2.57};
char couleur[4]= {'r','v','b','j'};
char texte[10]="abcd";
int t1[10] = {1,2,3};
```

1	2	3	4	5	6	7	8	9	10

0.	0.25	3.14	2.57

r	v	b	j

a	b	c	d	\0	?	?	?	?	?

1	2	3	?	?	?	?	?	?	?

- Dimension par défaut:

```
int t[ ]={0,0,0} => dimension = 3
char t[ ]={'r','v','b','j'}; => dimension = 4
char t[ ]= "rvbj" => dimension = 5
```

par contre int t[ ] sans initialisation est incorrect

Les tableaux en mémoire

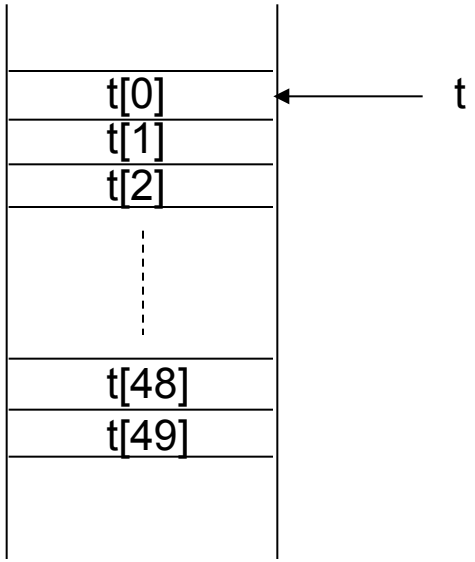
Déclaration et implantation mémoire :

int t[50]; => réservation dans la mémoire de 50 cases contiguës d'entiers.

L'adresse de la première case est t

&t[0] ⇔ t

*t ⇔ t[0]



Tableaux: accès

- Accès aux éléments d'un tableau

```
int t[50];

syntaxe 1
// accès à la (i+1)ème case avec i compris entre 0 et 49
t[i];

syntaxe 2
puisque t est l'adresse de la iere case
t[0] ⇔ *t        // entier à l'adresse t
t[1] ⇔ *(t+1)  // rem : priorité des opérateurs
…
t[i] ⇔ *(t+i)  //
```

```
*t+i ⇔ t[0]+i
```

Arithmétique de pointeurs

Tous les opérateurs arithmétiques d'addition, de soustraction et de comparaison.
Valeur incrémentée de **sizeof(type)**

```
unsigned strlen(char* str) {
unsigned len = 0;
 while (*str) {
len ++;
str ++; }
return len;
}
```

```
unsigned strlen(char* str) {
unsigned len = 0;
while (*str ++) len ++;
return len;
}
```

```
unsigned strlen(char* str) {
char *ptr = str;
while (*ptr++) { }
return ptr - str;
}
```

Passage d'un tableau à une dimension en paramètre

- Rappels:
  - Lorsqu'on déclare un tableau, par ex int t[10], t est l'adresse du 1er élément du tableau
  - Chaque élément du tableau peut être accédé par t[i] ou *(t+i)
  - Le tableau ne connaît pas sa taille => il faut la passer séparément
- Exemple

```
void printtab (int t1[], int n){
  for (int i=0; i<n; i++)
    printf("%i", t1[i]);
}

void main () {
  int t[50],t2[100];
  printtab(t,50);    // affiche toutes les cases de t de 0 à 49
  printtab(t2,100);  // affiche toutes les cases de t2 de 0 à 99
  printtab(t+20,30); // affiche toutes les cases de t de 20 à 49
  printtab(t+20,10); // affiche toutes les cases de t de 20 à 29
}
```

Passage d'un tableau à une dimension en paramètre

Puisqu'en fait t1 est une adresse, on peut le déclarer comme tel

```
void printtab (int * t1, int n) {
  int i;
  for (i=0; i<n; i++)
    printf("%i", t1[i]);
}
```

Passage d'un tableau à une dimension en paramètre

Conséquence :

Si un argument est de type tableau (c.-à-d. une adresse), la fonction peut modifier les cases du tableau

```
void misea0 (int t1[], int n){
   int i;
   for (i=0; i<n; i++)
     t1[i]=0;
}

void main () {
  int t[10]={1,2,3,4,5,6,7,8,9,10};
  printtab(t,10); -> 1 2 3 4 5 6 7 8 9 10
  misea0(t,10);
  printtab(t,10); -> 0 0 0 0 0 0 0 0 0 0
}
```

VOID * LE TYPE POLYMORPHE

- Tous les pointeurs ont la même taille
- void *var; est valide et représente un pointeur quelconque
- Compatible avec tous les types pointeurs
  *var est ILLEGAL (on ne sait pas ce qu'il y a au bout)
- AUCUNE arithmétique possible

```
void *mymemcpy(void *dst, void *src, size_t bytes) {
  char *dst1 = dst, *src1 = src;
  while (bytes --) {
      *dst1++ = *src1++; // NB: *dst++ = *src++ serait illegal!
  }
  return dst;
}
```

Etats d'un pointeur, NULL

Un pointeur doit contenir une adresse valide pour être déréférencé:

```
int i1 = 0, i2;
int * pi = &i1;
int * pj;  // valeur initiale non-définie => arbitraire
i2 = *pi;  // correct
i2 = *pj;  // non-défini: peut planter, mais pas forcément
```

PB : comment différencier une adresse valide d'une adresse invalide ?

NULL est une valeur spéciale indiquant qu'un pointeur ne pointe vers rien

```
int * pi = NULL;
int i;
if ( ....) pi = &i;
if (pi != NULL) printf ("%i", *pi);
```

Allocation dynamique de mémoire

Jusqu'à maintenant, on a vu que tous les objets (variables) ainsi que leur taille devaient être déclarés au moment de la compilation, càd explicitement dans le code C.

En particulier, la dimension des tableaux doit être connue au moment de l'écriture du pgm et ne peut être modifiée au moment de l'exécution.

Exception: les tableaux locaux à une fonction peuvent avoir une taille dynamique, mais ne subsistent pas a l'appel (ils sont sur la pile => instables):

```
int *bad_table(int n) { int tab[n]; return tab; } // ne JAMAIS faire ça !
```

Les fonctions de gestion "dynamique" de mémoire permettent de remédier à cette limitation.

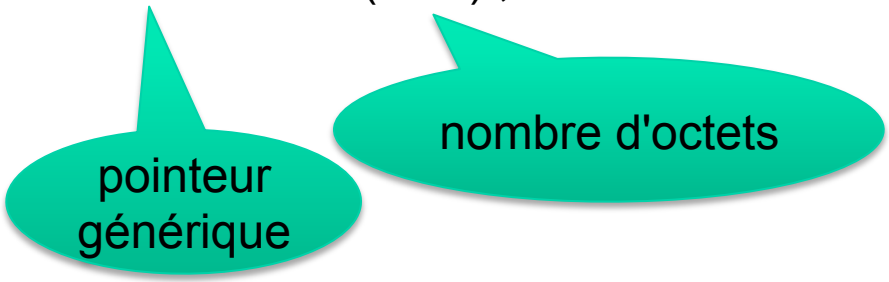
2 fonctions de base :

- malloc() : allocation d'une zone de mémoire
- free() : libération d'une zone mémoire précédemment allouée grâce à malloc

La fonction malloc

Prototype

```
void * malloc (int n) ;
```




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

La fonction malloc

la fonction malloc permet de réserver n octets contigus (un tableau) dans la mémoire.

La valeur de retour est l'adresse du premier octet réservé.

Exemple :

```
int * t; // t est un pointeur d'entier
int n;
printf("combien d'entiers voulez-vous réserver ?\n");
scanf("%i", &n);
t = (int*) malloc (n*sizeof(int));
```

conversion de type

Calcul du nombre d'octets

La fonction malloc

Exemple :

```
int * t; // t est un pointeur d'entier
int n;
….
t = (int*) malloc (n*sizeof(int));
..
// accès aux éléments
*t = …;     // 1ere case ou bien t[0]
*(t+1)=…; // 2eme case ou bien t[1]
*(t+2)=…; // 3eme caseou bien t[2]
…
*(t+i)=…//ieme case ou bien t[i]
```

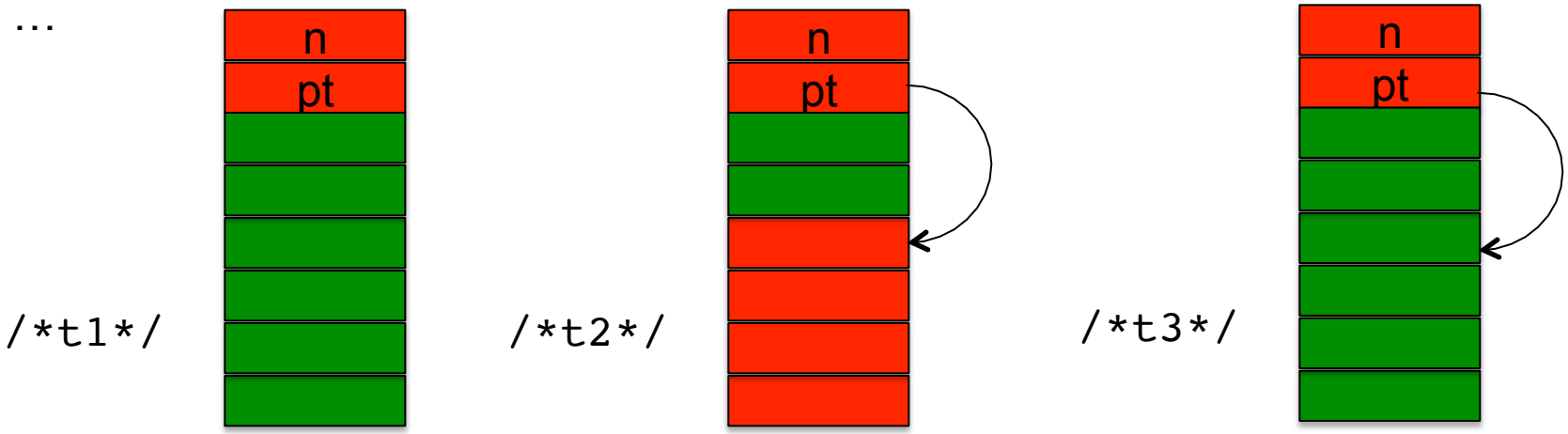
En fait, on a "réservé" un tableau de n cases

La fonction free

La fonction free permet de libérer l'espace mémoire alloué par un malloc précédent.

Ex :

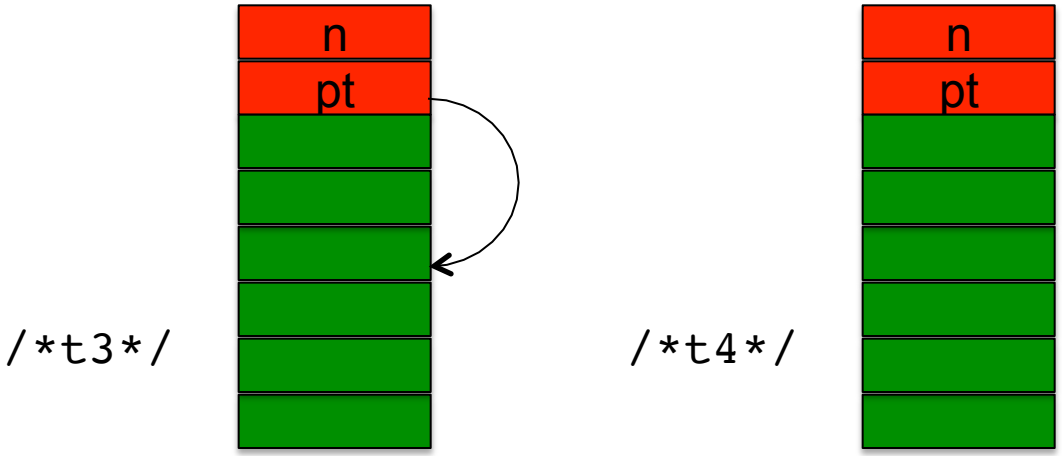
```
int * pt;
int n;
…   /*t1*/
…
pt = (int*) malloc(n*sizeof(int)); /*t2*/
…
free (pt); /*t3*/
…
```



La fonction free

Remarque :

Après free, pt ne vaut pas NULL et il indique pourtant une adresse recyclée.



Faire toujours suivre un free par une mise à NULL du pointeur

free (pt); /*t3*/

pt=NULL;

malloc et free

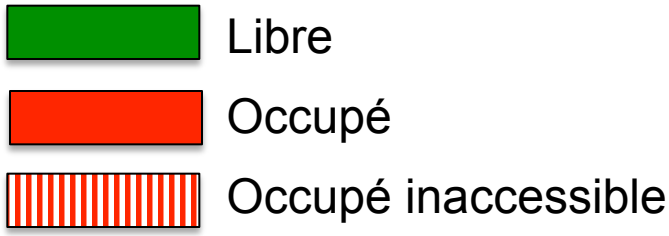
l'adresse donnée comme paramètre à la fonction free doit correspondre à une adresse renvoyée par un malloc précédent

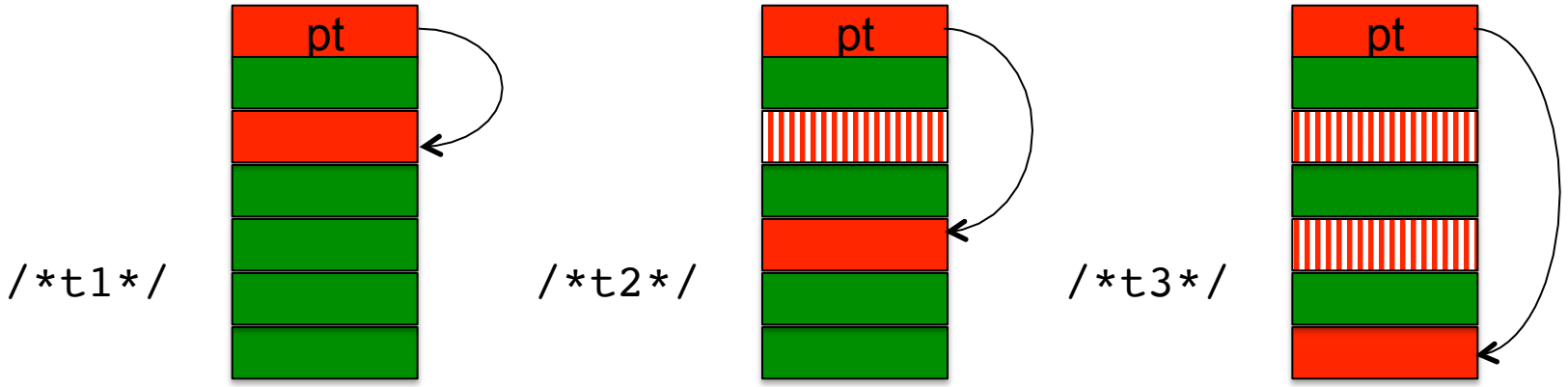
```
int * pt;
int n;
pt = (int*) malloc(n*sizeof(int));
pt = pt+1
free (pt);    => ERREUR
…
```

malloc et free

Il ne faut jamais "oublier" l'adresse renvoyée par un malloc, seul moyen d'atteindre les cases réservées => risque de saturation de la mémoire

```
int * pt;
…
pt = (int*) malloc(sizeof(int)); /*t1*/
pt = (int*) malloc(sizeof(int)); /*t2*/
pt = (int*) malloc(sizeof(int)); /*t3*/
…
```





Structures et allocation dynamique de mémoire : listes, files, piles, etc..

Jusque là, allocation dynamique = **objets de taille variable**

- mais pas plus de variables que de pointeurs déclarés

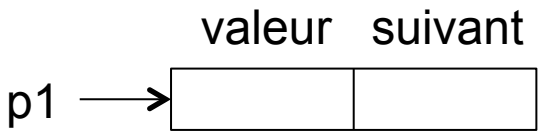
Lorsque on veut créer dynamiquement **un nombre variables d'objets**, on :

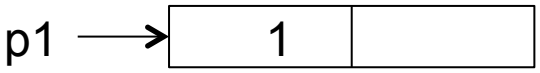
- déclare un pointeur sur structure
- dans la structure on définit un membre qui est lui-même un pointeur (à l'aide duquel on pourra créer dynamiquement une autre structure qui elle-même contient un pointeur, etc …)

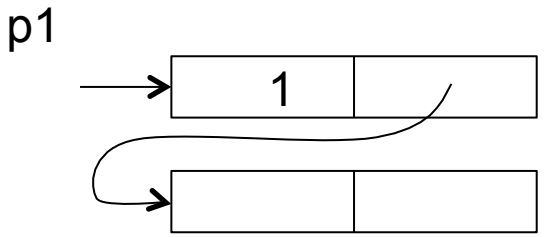
Structures et allocation dynamique de mémoire : listes, files, piles, etc..

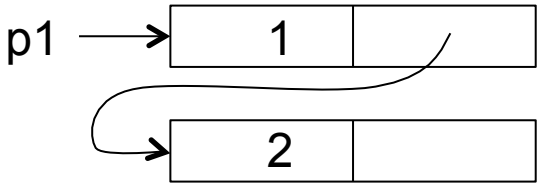
```
struct ent {
  int valeur;
  struct ent * suivant;
};
```

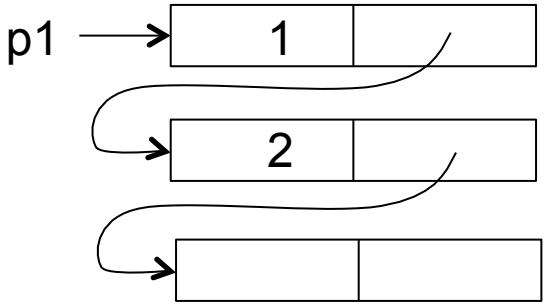
```
p1 = (struct ent *) malloc(sizeof(struct ent));
p1->valeur=1;
p1->suivant= (struct ent *) malloc(sizeof(struct ent));
p1->suivant->valeur=2;
p1->suivant->suivant= (struct ent *) malloc(sizeof(struct ent));
```



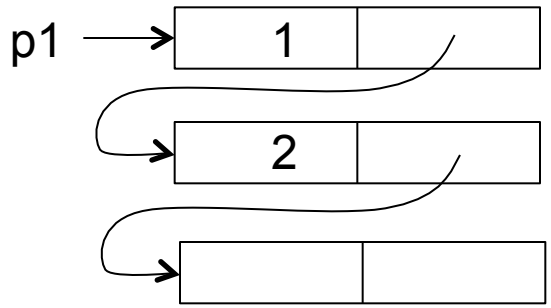








Structures et allocation dynamique de mémoire : listes, files, piles, etc..



Les valeurs sont accessibles par :

p1->valeur

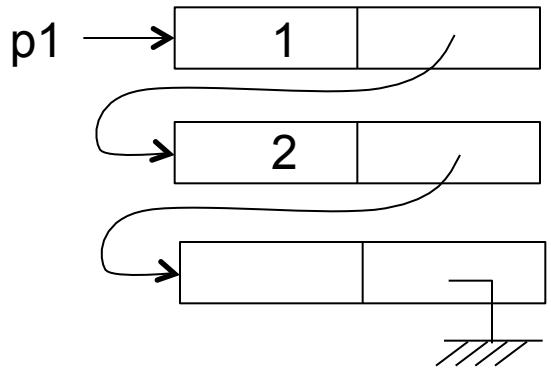
p1->suivant->valeur

p1->suivant->suivant->valeur

Pb : quand doit-on s'arrêter ?

rep : le dernier membre suivant doit être NULL.

Structures et allocation dynamique de mémoire : listes, files, piles, etc..



Les valeurs sont accessibles par :

p1->valeur

p1->suivant->valeur

p1->suivant->suivant->valeur

Pb : quand doit-on s'arrêter ?

rep : le dernier membre suivant doit être NULL.

Structures et allocation dynamique de mémoire : listes, files, piles, etc..

```
struct ent {
  int valeur;
  struct ent * suiv;
};

struct ent *  p1;
p1 = (struct ent *) malloc(sizeof(struct ent));
```

Syntaxe équivalente plus commode:

```
typedef struct ent element, *pt ;  // element et pt sont des types
struct ent {
  int valeur;
  pt suiv;
};
pt p1;
p1 = (pt) malloc(sizeof(element));
```

Listes

Création d'une liste contenant les n premiers entiers:

```
pt deb = NULL, paux;
for(int i=1; i<=n; i++) {
  paux = (pt) malloc (sizeof(element));
  paux->valeur = i;
  paux->suiv = deb;
  deb = paux;
}
```

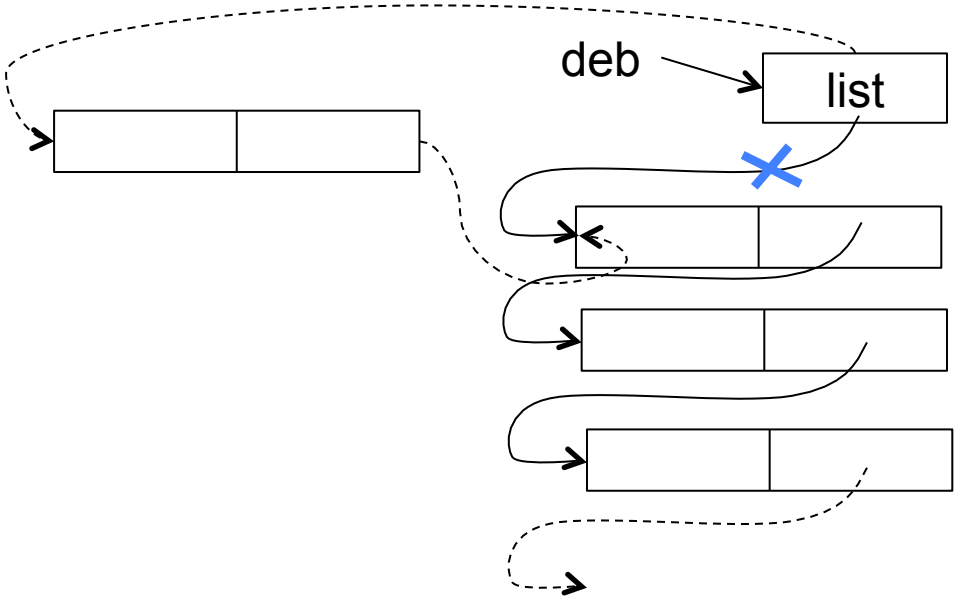
Parcours des éléments de la liste:

```
void affiche(pt deb) {
  pt paux = deb;
  while (paux != NULL) {
    printf("%d", paux->valeur);
    paux = paux->suiv;
} }
```

Listes : exemples de fonction

Fonction ajoutant une donnée en début de liste

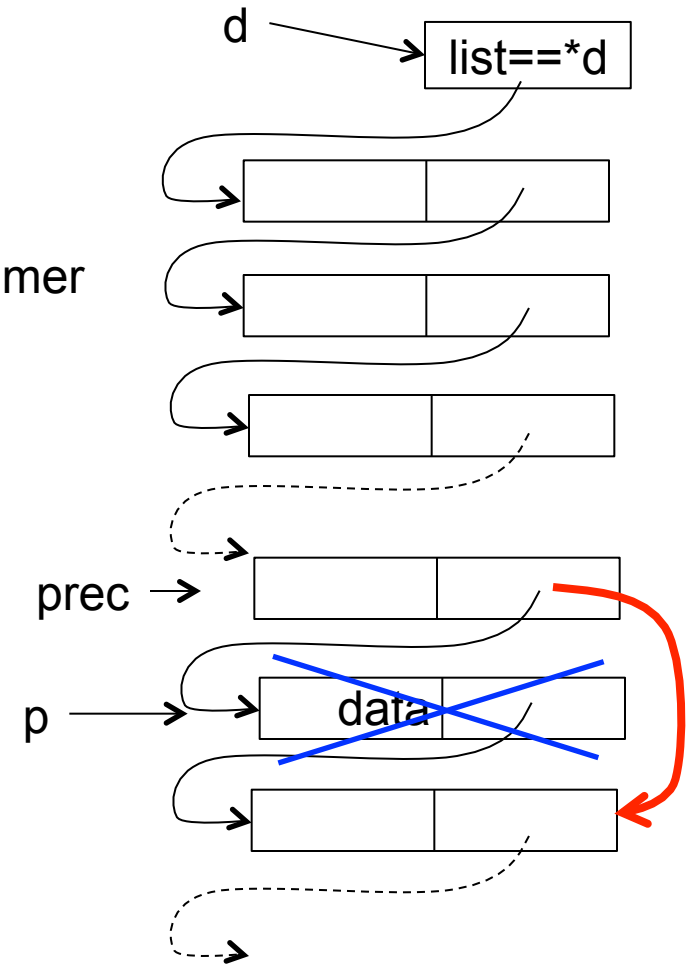
```
// Doit être appelée avec le pointeur de début de la liste, ex : ajoutdeb(&list,56)
void ajoutdeb (pt * deb, int data) {
  pt paux; // pointeur auxiliaire, pour ne pas perdre l'adresse du début de la liste
  paux = (pt) malloc(sizeof(element));
  paux->valeur = data;
  paux->suiv = *deb;
  *deb = paux;
}
```



Listes : exemples de fonction

Fonction supprimant une donnée dans la liste

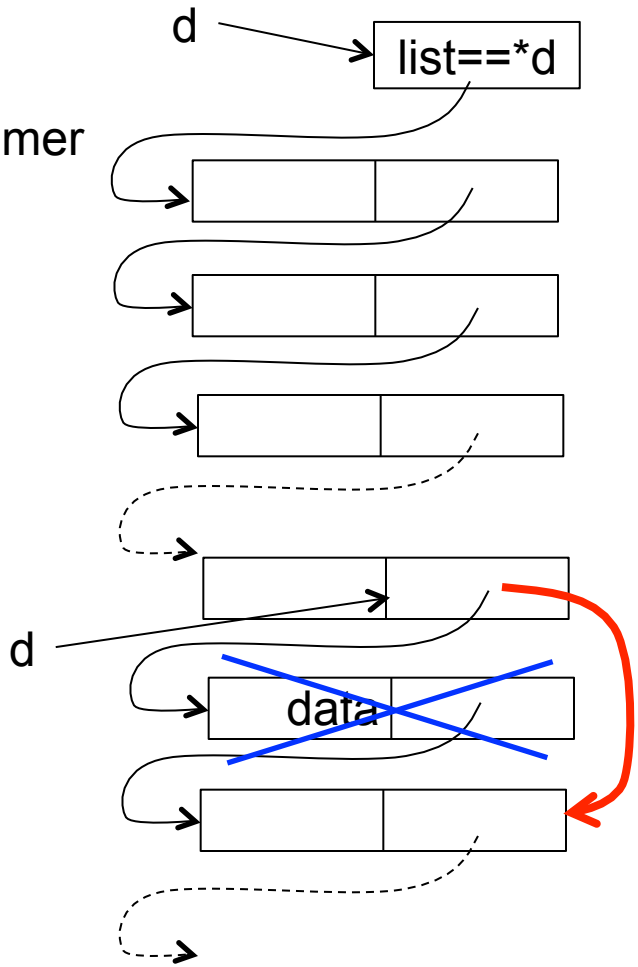
```
// Doit être appelée avec le pointeur de début de la liste ex: supprime(&list,42)
void supprime (pt * d, int data) {
  pt p = *d;
  pt prec = NULL;
  while (p != NULL) {
    if (p->valeur == data) { // trouvé l'élément à supprimer
      if(prec != NULL) prec->suiv = p->suiv;
      else *d = (*d)->suiv;
      free(p);
      return;
    }
    // on avance dans la liste
    prec = p;
    p = p->suiv;
} }
```



Listes : exemples de fonction

Fonction supprimant une donnée dans la liste (v2)

```
// Doit être appelée avec le pointeur de début de la liste ex: suppr(&list,42)
void suppr (pt * d, int data) {
  while (*d != NULL) {
    if ((*d)->valeur == data) { // trouvé l'élément à supprimer
      pt p = *d;
      *d = (*d)->suiv;
      free(p);
      return;
    }
    d = &(*d)->suiv; // on avance dans la liste
  }
}
```



Listes: Exercice

- Modifier le programme du cours (liste.c):
  - Créer une liste contenant les arguments de la commande
  - Afficher la liste
- Écrire des fonctions sur la liste pour:
  - Retourner l'addresse d'un élément si présent dans la liste:
    pt adresse(pt d, int data)
  - Supprimer toute la liste:
    void detruit_liste(pt *d)
  - Rajouter un élément en fin de liste:
    void ajoutfin(pt *d, int data)
- Modifier le programme pour traiter une liste de chaînes de caractères plutôt qu'une liste d'entiers

COURS 15-16: PROGRAMMES STRUCTURÉS

Programmation modulaires

Instruction switch

Type énumération

Flot de contrôle (continue, break, return)

Pré-processeur

- Exécuté avant de compiler le programme
- Transforme les lignes commençant par un #
- #define: définit des constantes NOM et macros NOM(…)
- #include: Inclut un fichier
- #if…#else…#endif: Compilation conditionelle

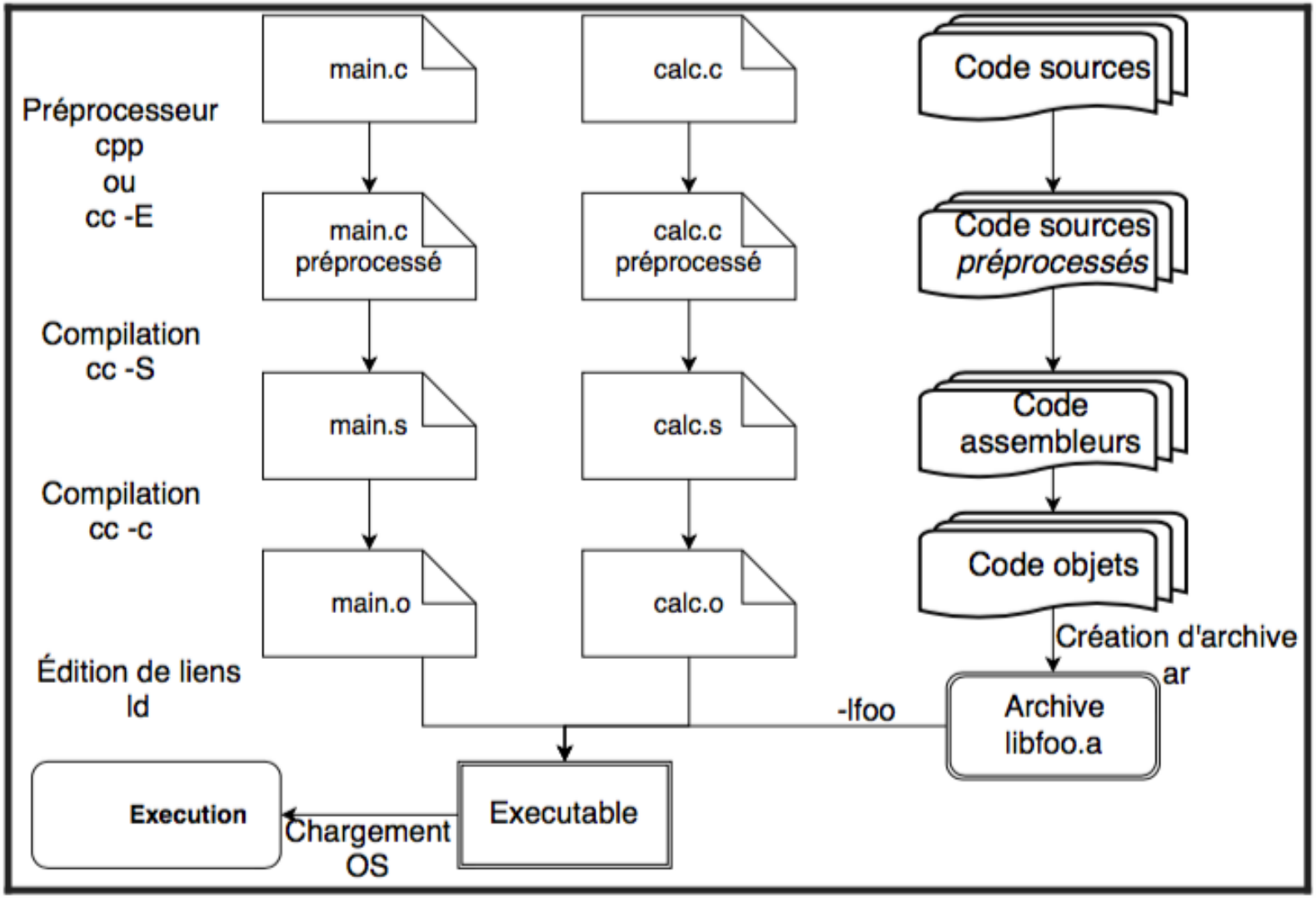
```
#define FOO 42
#define BAR (FOO + 3)
#define BAZ FOO + 3
int a = FOO; // a == 42
int b = BAR * 2; // b == 90
int c = BAZ * 2; // Attention: c == 48
```

```
#define MAX(x, y) ((x) > (y) ? (x) : (y))

int m = MAX(foo(), bar());
// Nombre d'appel de foo et bar ?

int f = foo(), b = bar();
int mm = MAX(f, b); // OK
```

CHAÎNE DE COMPILATION



Fichiers d’en-tête

- Par convention se terminent par .h
- Inclus par #include suivi de :
  - "fichier.h" s'il s'agit d'un fichier perso. (répertoire courant)
  - <fichier.h> s'il s'agit d'un fichier std. (include path)
- Uniquement des déclarations :
  - Prototypes de fonctions (manque le code)
  - Constates/Macros
  - Types (eventuellement incomplets)
  - etc

Préprocesseur et compilation séparée

- Module X = Interface (x.h) + Implantation (x.c)

```
// main.c
#include "calcul.h"
// ...
foo(42);
struct bar b1; // Err: type incomplet
struct bar *b = get_bar(⋯); // OK
```

```
cc -c main.c
cc -c calcul.c
cc -o executable main.o calcul.o
```

```
// calcul.h
#define TAILLE_MAX 42
 void foo(int x); // Fctn. ext.
 struct bar; // Types incomplets
 struct bar * bar_init(⋯);
 // ... autres prototypes
```

```
// calcul.c
#include "calcul.h"
void foo(int n) { ⋯ }
struct bar { ⋯ };
struct bar *x bar_init() { ⋯ }
```

Préprocesseur et compilation séparée

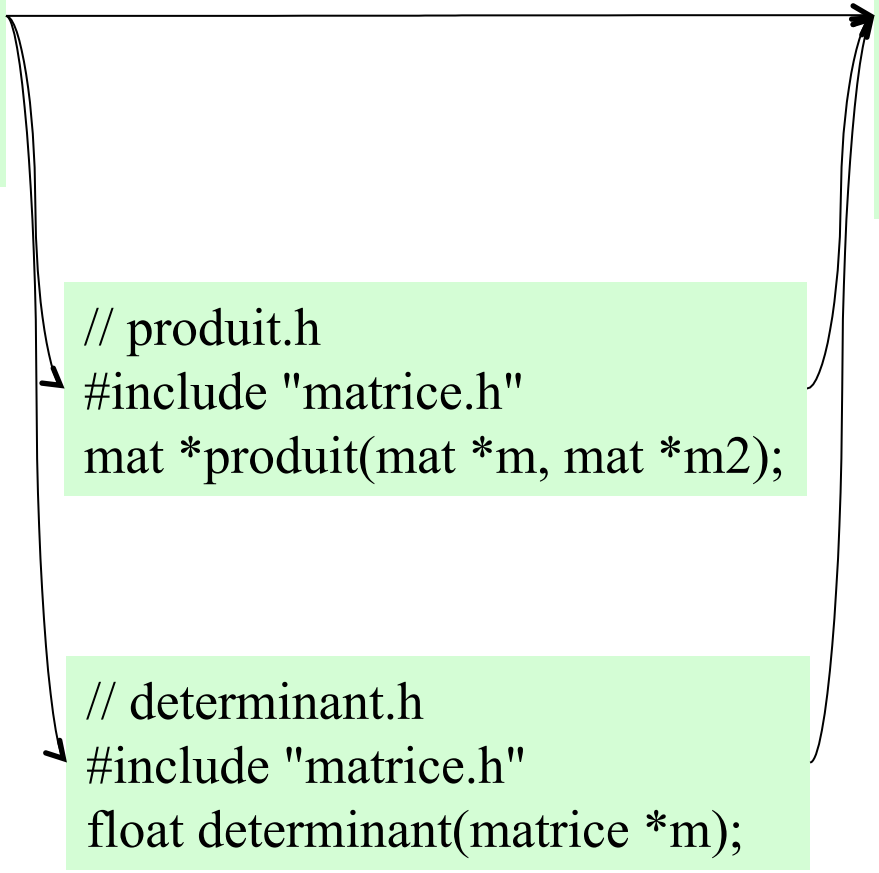
- Éviter les inclusions multiples (astuce des #ifndef)

```
// main.c
#include "matrice.h"
#include ”produit.h"
#include ”determinant.h"
// ...
```

```
// matrice.h
typedef struct {
   int lignes,
   int colonnes;
   float *tab;
 } mat;
```

```
// produit.h
#include "matrice.h"
mat *produit(mat *m, mat *m2);
```

```
// determinant.h
#include "matrice.h"
float determinant(matrice *m);
```



Préprocesseur et compilation séparée

- Éviter les inclusions multiples (astuce des #ifndef)

```
// main.c
#include "matrice.h"
#include ”produit.h"
#include ”determinant.h"
// ...
```

```
// produit.h
#ifndef PRODUIT_H
 # define PRODUIT_H
 #include "matrice.h"
 mat *produit(mat *m, mat *m2);
#endif
```

```
// determinant.h
#ifndef DETERMINANT_H
 # define DETERMINANT_H
 #include "matrice.h"
 float determinant(matrice *m);
#endif
```

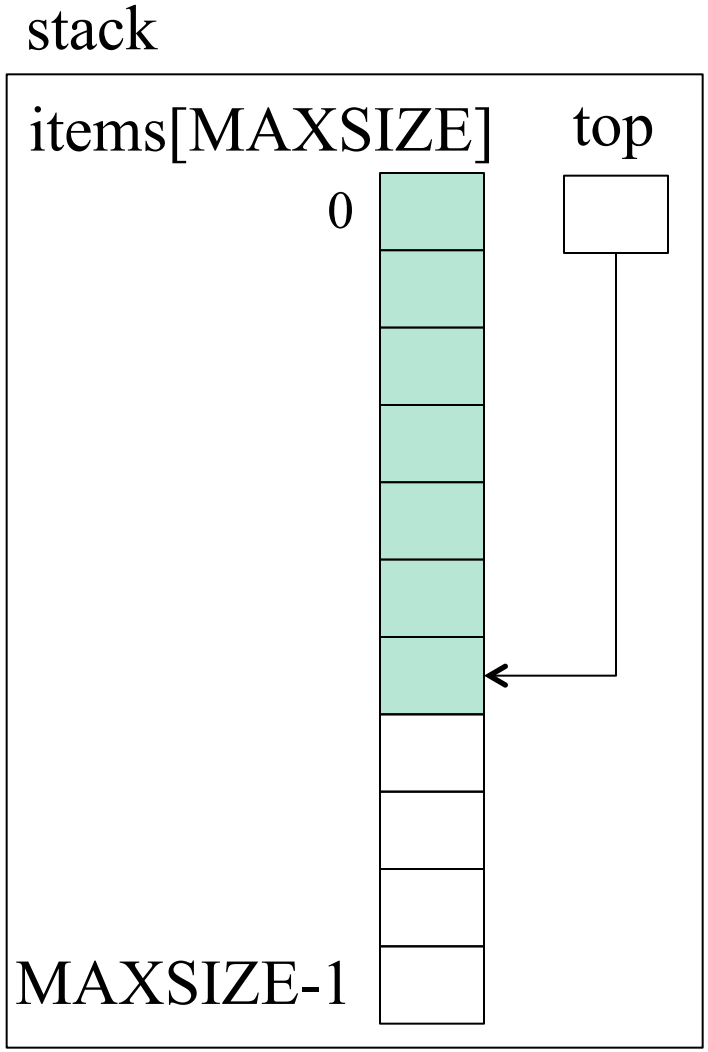
```
// matrice.h
#ifndef MAT_H
 #define MAT_H
 typedef struct {
   int lignes,
   int colonnes;
   float *tab;
 } mat;
#endif
```

Variables, fonctions & compilation séparée

- Variables
  - locales: valables uniquement dans la fonction, voire un bloc (ce qu'on fait depuis le début, et ce qu'il faut maximiser)
  - globales: *déclarées en dehors de toute fonction*
- Tous les noms globaux (variables et fonctions) sont exportés par défaut
  - static sur une variable globale ou fonction la cache
  - extern sur une **déclaration** de variable globale/fonction
    => Il existe une var/fonction, sa définition est ailleurs
- Conclusion: plusieurs **portées** pour une variable:
  - Locale à un bloc: {int x; …} / for(int x; ...; ...) { ... }
  - Locale à une fonction: T foo(int x, …) {int y; …}
  - Globale dans un module: static int x;
  - Globale au programme: int x; / extern int x;

Exemple: pile

- Operations
  - Empiler (push)
  - Dépiler (pop)
  - Regarder (peek)
  - Nombre d'éléments (size)
  - Est Vide / Est Pleine (isempty/isfull)
  - Initialiser (init)
  - Détruire (kill)
- Données:
  - Eléments (items[])
  - Sommet (top)
  - Capacité (MAXSIZE)



Exemple: pile (stack.h)

```
typedef struct stack_s *stack;

extern stack init();
extern kill(stack s);
extern int isempty(stack s);
extern int isfull(stack s);
extern int size(stack s);
extern char * peek(stack s);
extern char * pop(stack s);
extern void push(char * data, stack s);
```

Exemple: pile (stack.c)

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"
#define MAXSIZE 8
struct stack_s {
  char * items[MAXSIZE];
  int top;
};
stack init() {
  stack s = malloc(sizeof(struct stack_s));
  s->top = -1;
  return s;
}
void kill(stack s) { free(s); }
int isempty(stack s) { return size(s) == 0; }
int isfull(stack s) {
 return size(s) == MAXSIZE;
}
int size(stack s) { return s->top + 1; }
```

```
char * peek(stack s) {
  if(isempty(s)) {
    printf("peek: stack is empty.\n");
    exit(1);
  }
  return s->items[s->top];
}

char * pop(stack s) {
  if(isempty(s)) {
    printf("pop: stack is empty.\n");
    exit(1);
  }
  return s->items[s->top--];;
}

void push(char * data, stack s) {
  if(isfull(s)) {
    printf("push: stack is full.\n");
    exit(1);
  }
  s->items[++s->top] = data;
}
```

Exemple: pile (main.c)

```
#include <stdio.h>
#include <string.h>
#include "stack.h"

int main(int argc, char *argv[]) {
  stack s = init();
  for(int i = 1; i < argc; i++)
    if(!strcmp(argv[i], "pop")) pop(s);
    else push(argv[i], s);

  printf("stack contains %d elements\n", size(s));
  printf("Element at top of the stack: %s\n", peek(s));
  printf("Elements: \n");

  while(!isempty(s)) { printf("%s\n", pop(s)); } // print stack data
  printf("Stack full: %s\n" , isfull(s)?"true":"false");
  printf("Stack empty: %s\n" , isempty(s)?"true":"false");
  kill(s);
}
```

Make

- Utilitaire pour piloter la chaîne de compilation d'une application donnée (cf. un « makefile »)
- Repose sur un langage dédié (domain-specific language, ou DSL)
- Règles de construction (build) d'une application
  - Partie déclarative: de quels fichiers (« sources ») dépend chaque fichier (« cible »)
  - Partie impérative: comment produire les fichiers cible

    cible: source …
    commande1
    ...
  - Exécution incrémentale: on ne (re)produit que les parties devant changer
- Variables: user $(X); prédéf.: cible ($@), source(s) ($<, $^)

Exemple: pile

```
# Makefile
CC=gcc
CFLAGS=-Wall
OBJS=stack.o main.o

stack: $(OBJS)
	$(CC) -o $@ $^
stack.o: stack.c stack.h
	$(CC) -c $(CFLAGS) $<
main.o: main.c stack.h
	$(CC) -c $(CFLAGS) $<
clean:
	rm -f $(OBJS) stack
```

```
stack/$ make
gcc -c -Wall stack.c
gcc -c –Wall main.c
gcc -o stack stack.o main.o
stack/$ make clean
rm -f stack.o main.o stack
```

GESTION DES ERREURS

- Pas de mécanisme dédié
- Pour les fonctions système (gestion fichiers, mémoire,…) :
  - variable globale int errno; (errno.h)
    Elle contient le dernier code d'erreur
  - void perror(char* prefix); affiche le prefixe suivi du dernier message d'erreur
  - Couvre toutes les fonctions systèmes
  - etc.
- Règle: toujours tester les résultats
  - Des fonctions système
  - De vos propres fonctions
- Actions possibles en cas d'erreur
  - Irrécupérable: Arrêter le programme (printf(…); exit(N))
  - Rattrapable: Retourner une valeur hors-norme (NULL/<0/…)

ERREURS DURANT LE DÉVELOPPEMENT

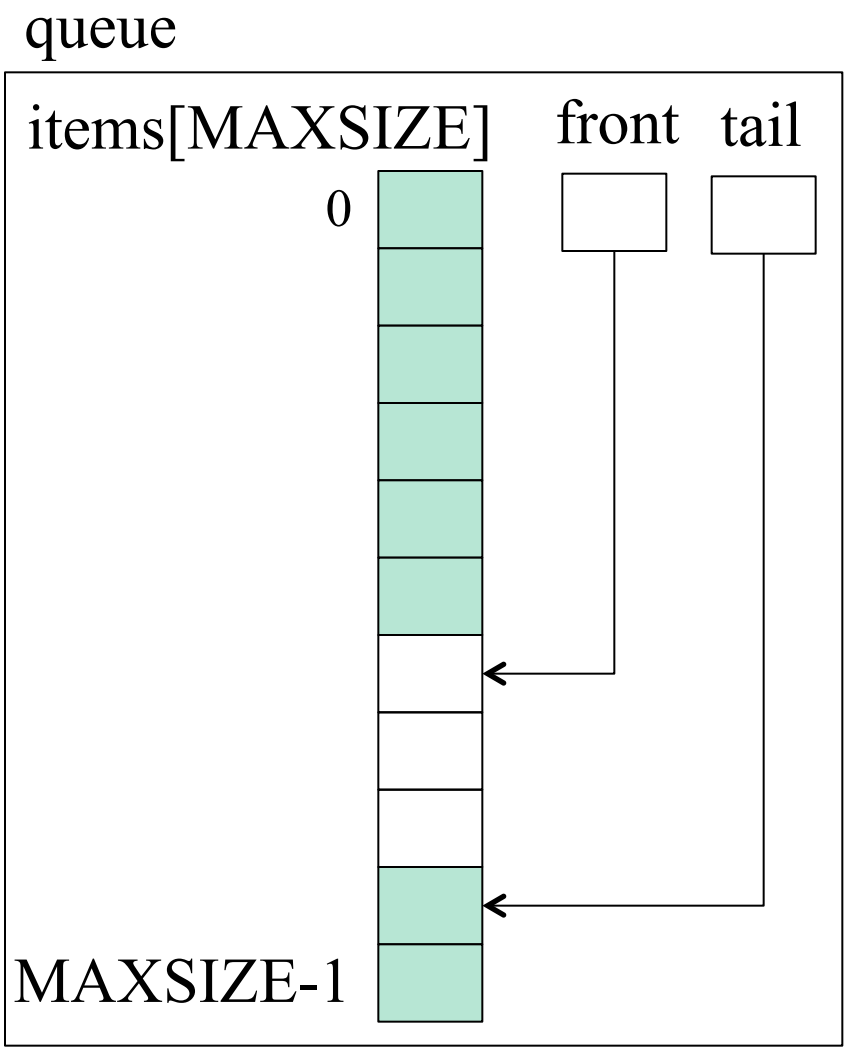
- Pour vous aider lors du développement (assert.h):
  assert(expr);
  arrête le programme si expr est faux
- Sachez que ces lignes peuvent être ignorées (il suffit de compiler avec -DNDEBUG)

```
if ((fd = fopen(file, "r")) == NULL) {
  perror("Error opening file");
  exit(1);
}
```

```
fd = fopen(file, "r");
assert(fd); // C'est mal
```

Exercice: queue de taille fixe

- Implanter une structure de donnée nommé queue, de taille fixe, utilisant un tampon circulaire dans un tableau



Instruction switch

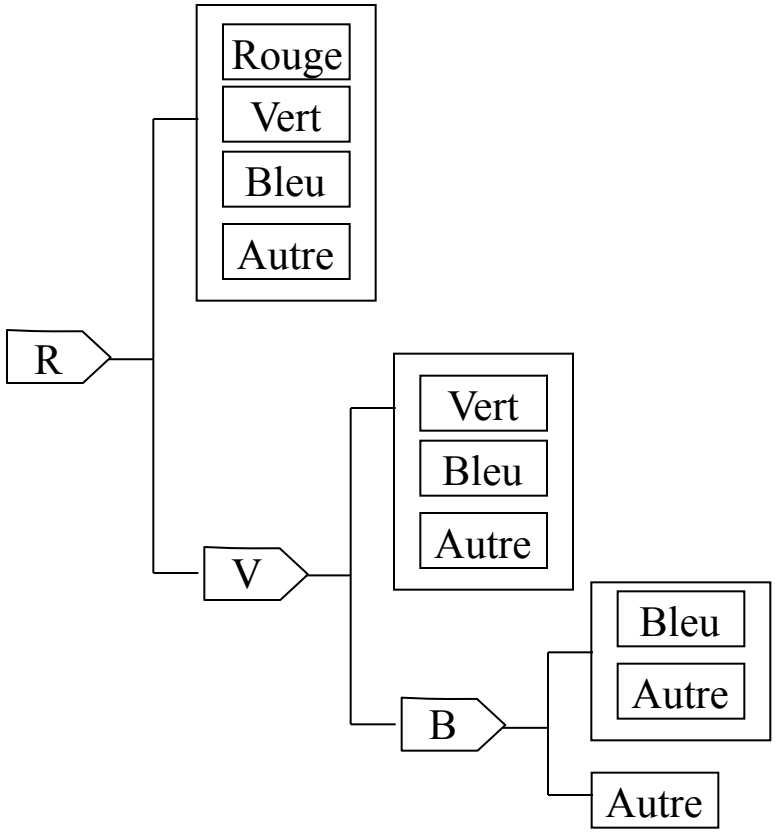
- Sélecteur: expression à valeur entière
- Alternatives: chacune traite une valeur possible du sélecteur
- Syntaxe:

```
switch (expression) {
  case valeur1: instruction1; instruction2; ….;
  case valeur2: instruction1; instruction2; ….;
  …
  case valeurN: instruction1; instruction2; ….;
  default : instruction1; instruction2; ….;  // optionnel
};
```

switch

Exemple :

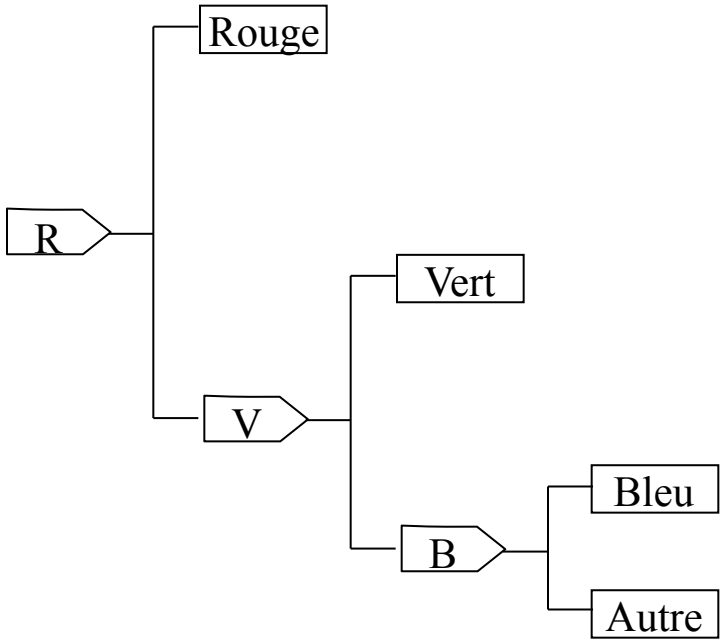
```
char c;
printf("donner une couleur\n");
c=getchar();
if ('a'<=c && c<='z') c += 'A'-'a';
switch (c) {
  case 'R' :
    printf("Rouge\n");
  case 'V' :
    printf("Vert\n");
  case 'B' :
    printf("Bleu\n");
  default :
    printf ("Autre");
}
```



switch + break

Exemple :

```
char c;
printf("donner une couleur\n");
c=getchar();
if ('a'<=c && c<='z') c += 'A'-'a';
switch (c) {
  case 'R' :
    printf("Rouge\n"); break;
  case 'V' :
    printf("Vert\n"); break;
  case 'B' :
    printf("Bleu\n"); break;
  default :
    printf ("Autre");
}
```

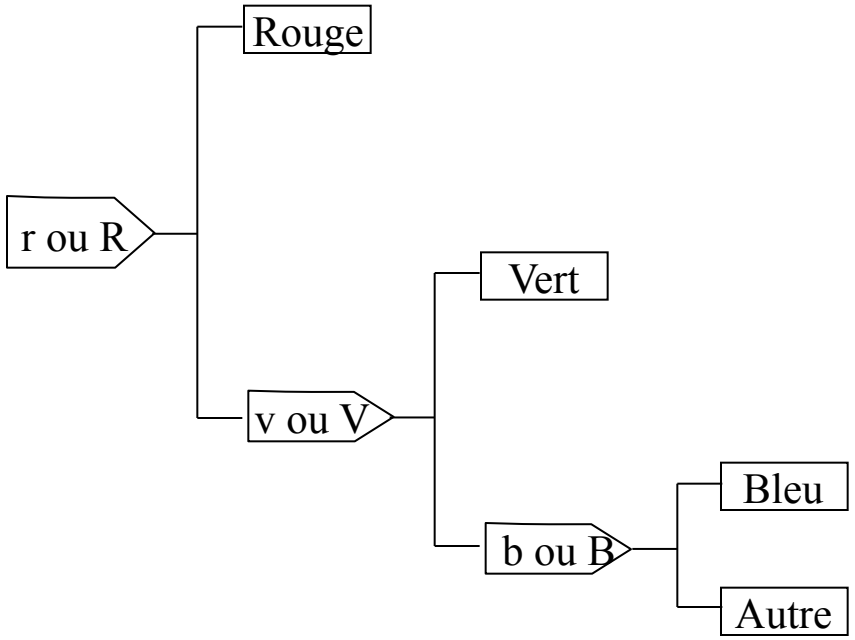


switch + break

Exemple :

```
char c;
printf("donner une couleur\n");
c=getchar();

switch (c) {
  case ‘R’: case ‘r’:
    printf("Rouge\n"); break;
  case ‘V’: case ‘v’:
    printf("Vert\n"); break;
  case ‘B’: case ‘b’:
    printf("Bleu\n"); break;
  default :
    printf ("Autre");
}
```



Boucle + break

- Dans une structure itérative, l'instruction **break** fait sortir de la structure de contrôle dans laquelle elle est imbriquée

- Utilisation dans les boucles : permet de sortir d'une boucle si on détecte une exception

Ex:

```
while (c1) {
  debut_traitement();
  if (c2) break;
  suite_traitement();
}
```

Boucle + continue

- Dans une structure itérative : l'instruction **continue** permet d'arrêter l'itération courante sans sortir de la boucle

- Exemple: Calculer la moyenne des valeurs positives d'un tableau d'entiers relatifs.

```
nb_valeurs=0;
somme = 0;
for (i=0;i<dim;i++) {
  if (T[i]<0) continue;
  somme = somme + T[i];
  nb_valeurs ++;
}
moyenne = somme / nb_valeurs
```

Énumérations

- Entiers nommés
- Numérotation automatique (mais on peut l'aider)
- Affichage, calcul et autre comme pour un entier

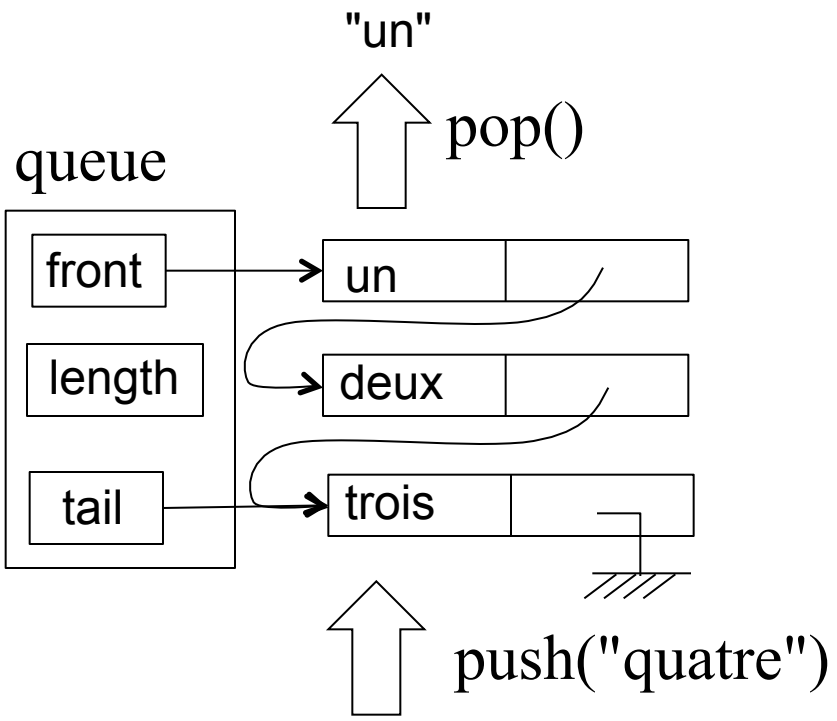
```
enum couleur {
  TREFLE, CARREAU, COEUR,
  PIQUE
};
enum valeur {
  DEUX=2, TROIS, QUATRE,
  CINQ, SIX, SEPT, HUIT, NEUF,
  DIX, VALET, DAME, ROI, AS
};
enum couleur atout = PIQUE;
```

```
struct carte {
  enum valeur valeur;
  enum couleur couleur;
};
struct carte black_jack = { VALET, PIQUE };

int plus_fort(struct carte c1, struct carte c2) {
  return c1.valeur > c2. valeur
         || c1.valeur == c2.valeur
           && c1.couleur > c2.couleur;
};
```

Exercice: queue de taille illimitée

- Implanter une queue de taille illimitée, utilisant une liste chaînée




```
typedef struct queue_s *queue;
extern queue init();
extern void kill(queue q);
extern int isempty(queue q);
extern int isfull(queue q);
extern int size(queue q);
extern char * peek(queue q);
extern char * pop(queue q);
extern void push(char * data, queue q);
```

```
typedef struct cell_s {
  char *val;
  struct cell_s *next;
} cell;
struct queue_s {
  cell *front;
  cell *tail;
  int length;
};
```

COURS 17-18: ASPECTS AVANCÉS

Complément sur static

Mécanisme d'appel d'une fonction

Unions

Pointeurs de fonctions

Tableaux multi-dimensionnels

Compléments : static

- Static sur une variable locale à une fonction: la variable maintient sa valeur à travers les appels de la fonction

```
void inc( )
{
   int i=0;
   i++;
   printf ("%i", i);
}
```

1, 1, 1, 1, …

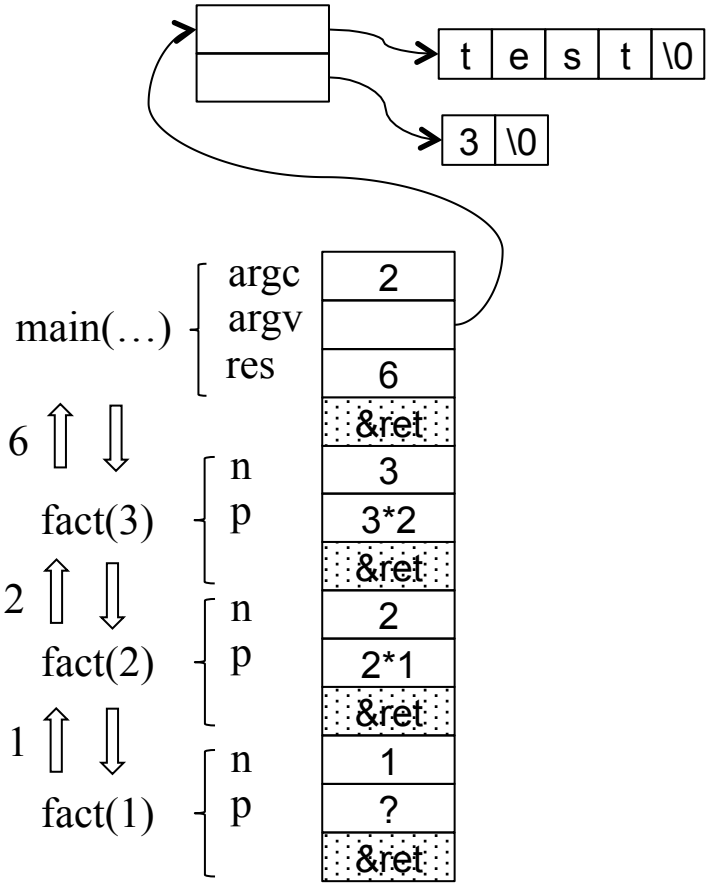
```
void inc( )
{
   static int i=0;
   i++;
   printf ("%i", i);
}
```

1, 2, 3, 4, …

Mécanisme d'appel d'une fonction. Récursion

```
int fact(int n) {
  int p;
  if (n<=1) return (1);
  p = n * fact(n-1); // appel récursif
  return p;
}
int main (int argc, char *argv[]) {
  int res = fact(atoi(argv[1]));
  printf("%i\n", res);
}
```

```
$ make test
gcc -o test test.c
$ ./test 3
6
```



- Empiler:
  - Les arguments
  - Les variables locales
  - L'adresse de retour

UNIONS

- Comme une structure où **un seul** champ peut être utilisé, au choix.
- La taille (sizeof) est au moins égale à la taille du plus grand élément

```
union some_value {
 int i;
 double d;
};
union some_value v;
v.i = 10;
v.d = 20.0;
// v.i écrasé maintenant
printf("i=%i, d=%f", v.i, v.d);
```

- Souvent utilisé en conjonction avec une structure et une énumération

```
struct number {
  enum {INT, REAL} type;
  union some_value v;
};
struct number i = {INT, {.i=10}};
struct number pi ={REAL, {.d=3.14}};

void printnum(struct number n) {
  if (n.type == INT)
    printf("%i\n", n.v.i);
  else printf("%f\n", n.v.d);
}
```

POINTEURS DE FONCTIONS

- Les fonctions sont chargées en mémoire, elles ont donc une adresse
- On peut faire un pointeur sur cette adresse (en utilisant simplement le nom de la fonction)
- On peut appeler la fonction via son pointeur
- Attention à la syntaxe relativement complexe pour déclarer une variable:

  type_de_retour (*__fptr__) (types_des_parametres);

```
int (*compare)(const char*, const char*) = strcmp;
res = (*compare)(s1, s2); // Compare en tenant compte de la case
compare = strcasecmp;
res = (*compare)(s1, s2); // Compare sans tenir compte de la case
```

Exemple: qsort.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// extern void qsort(void *base, size_t nel, size_t width,
//                   int (*compar)(const void *, const void *));

int cmp(const void *x, const void *y) {
  return strcmp(*(const char**)x, *(const char **)y);
}

int main(int argc, char *argv[]) {
  int i;
  qsort(argv, argc, sizeof(char *), cmp);
  for(i = 1; i < argc; i++) {
    printf("%s ", argv[i]);
  }
  printf("\n");
}
```

Tableaux à plusieurs dimensions

- Tableau dont chaque case est elle-même un tableau

  ex : typedef int t[100] // t est un type

  t matrice [20];

  matrice est un tableau de 20 cases, chacune est un tableau de 100 entiers
  => matrice est un tableau de 20*100 entiers

  autre déclaration :

  int matrice [20][100]; // tableau de 20 "lignes" et 100 "colonnes"

- Accès aux éléments

  par un 1er indice allant de 0 à 19 et par un 2eme indice allant de 0 à 99

  matrice[3] est la 4eme case de tableau. C'est un tableau de 100 cases (entiers)

  matrice[3][48] est un entier.

  matrice [i][j] avec i de 0 à 19 et j de 0 à 99

  matrice est un tableau à 2 dimensions

Tableaux à plusieurs dimensions

- Pas de limitations sur le nombre de dimensions

Ex à 3 dimensions : tableau de coord. de pts de l'espace

typedef float point[3] ; // x: indice 0, y : indice 1, z : indice 2

point tab[100][100]; // tab = matrice de 100 points

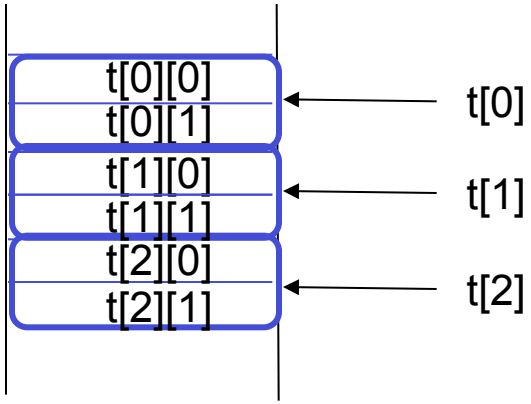
ou bien

float tab[100][100][3];

tab[2][5][1] représente le "y" du point rangé en ligne 2 et colonne 5

- Implantation mémoire

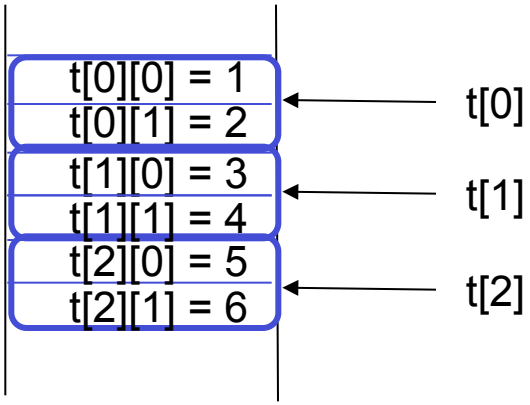
  int t[3][2];



Tableaux à plusieurs dimensions

- Initialisation (à la déclaration)

```
int t[3][2] = {1,2,3,4,5,6};
ou bien  (+ clair)
int t[3][2] = {{1,2},{3,4},{5,6}};
```




```
int t[3][2] = {{1,2},4,{5,6}}; => t[1][1] non initialisé
```

- Initialisation grâce à des boucles

```
for (i=0;i<3;i++)
        for (j=0;j<2;j++)
                t[i][j]=0;
```

Tableaux à plusieurs dimensions

- Accés aux éléments

int t[dim1][dim2] ;
t[i][j] ⇔ * (t+i*dim2+j)

int t[dim1][dim2][dim3];
t[i][j][k] ⇔ * (t+i*dim2*dim3+j*dim3+k)

…

⇒ **la première dimension n'est pas utilisée dans le calcul**
⇒ **On peut laisser implicite la première dimension (mais pas les autres):**

int t[][2] = {{1,2},{3,4},{5,6}}; // ⇔ t[3][2]

void print_pairs(int t[][2], int n) { … }

Allocation dynamique

- Équivalence matrice – pointeur:
  - int tab[n];
    int *ptab = tab;
    int *ptab = malloc(n * sizeof(int));

  - int mat[n1][n2];
    int (*ptab)[n2] = mat;
    int (*ptab)[n2] = malloc(n1 * n2 * sizeof(int))

- Passage en parametre (si dimensions = variables):
  - print(n1, n2, ptab);

```
void print(int lig, int col, int m[][col]) { // NB: ordre des parametres !
  for (int i = 0; i < lig; i++)
    for (int j = 0; j < col; j++)
      printf("%i ", m[i][j]);
}
```

Exercice

- Écrivez un programme (crosswords.c) qui transpose une matrice de mots croisés
- Exemple:

```
ex/$ ./crosswords miels arret ga**a i**or etes*
magie
ira*t
er**e
le*os
star*
```

Exercice

- Écrivez un programme (words.c) qui:
  - Lit l'entrée standard et affiche tous les mots (identifiants C)
  - Insère un mot avec son nombre d'apparitions dans une liste
  - Affiche la liste de mots avec leur nombre total d'apparitions
  - Affiche la liste des mots par ordre alphabétique
  - Calcule le nombre total de mots
  - Trouve le mot le plus utilisé
  - Trouve les N mots les plus utilisés
- Indice:
  - utiliser le format %[a-z_A-Z0-9] de scanf pour avaler un mot entier
- Exemple:

```
ex/$ ./words <words.c
a:3 add:2 argc:1 argv:1 break:1 buf:10 BUFSIZE:2 …
total: 268 words
top cell = pl:19
```

Conclusion

- Le langage C offre un éventail de techniques puissantes, dont nous avons vu un sous-ensemble conséquent
- Il faut bien les employer pour obtenir des programmes fiables et maintenables
- La programmation ne s'arrête pas lorsque le programme passe les premiers tests
  - Tester, re-factoriser, optimiser, … et prendre plaisir !
- Pour aller plus loin
  - Explorer les fonctions standard (fichiers, …)
  - Explorer les autres aspects du langage (constantes, nombre variables d'arguments, champs de bits, …)
  - Apprendre des techniques de programmation en relisant des programmes bien écrits

ANNEXE

Fonctions et tableaux multi-dimensionnels

Fonctions et tableaux à plusieurs dimensions

Rappel: Pour de tableaux à plusieurs dimensions, la première dimension n'est pas utilisée pour le calcul d'adresse, mais la seconde , la troisième etc.. sont nécessaires

int t[dim1][dim2]….[dimn] ;

t[i1][i2]….[in] ⇔ * (t

$+i_1*dim_2*dim_3*dim_{4.....}*dim_n$

$+i_2*\quad dim_3*dim_{4.....}*dim_n$

+…..

$+i_{n-1}\quad *dim_n$

$+i_n)$

Conséquence

Lorsqu'un argument est un tableau à plusieurs dimension il faut donner explicitement sa deuxième, troisième etc...

Exemple

```
int rang_matrice (float t[][40]) {
.....
}
```

Fonctions et tableaux à plusieurs dimensions

```
int rang_matrice (float t[][40]) {
.....
}
```

- Cette fonction n'est utilisable qu'avec des tableaux dont la deuxième dimension est 40

```
float t1[30][40];
float t2[30][50]; /* on ne peut pas utiliser la fonction sur t2 */
```

Ennuyeux ! si l'on manipule des matrices avec une deuxième dimension différente.

Comment contourner la difficulté ?

Faire soit-même l'adressage des cases dans la fonction en gérant les dimensions

Fonctions et tableaux à plusieurs dimensions

Exemple : mise à zéro d'une matrice à l lignes et c colonnes

```
void mat0 (float * t, int l, int c) {
   int i,j;
/* rappel t[i][j] = *(t+i*dim2+j) */
   for (i=0;i<l;i++)
        for (j=0;j<c;j++)
              *(t+i*c+j) = 0.0;
}
void main(){
float t1[20][40],t2[30][10];
mat0 (t1,20,40);
mat0 (t2,30,10);
mat0 (t1,20,5);  /* correct ou incorrect ? */
mat0 (t1,10,40); /* correct ou incorrect ? */
}
```

Fonctions et tableaux à plusieurs dimensions

Calcul correct si c est égal à la deuxième dimension vraie du tableau:

```
void mat0 (float * t, int l , int c) {
   int i,j;
/* rappel t[i][j] = *(t+i*dim2+j) */
   for (i=0;i<l;i++)
        for (j=0;j<c;j++)
              *(t+i*c+j) = 0.0;
}
void main(){
float t1[20][40],t2[30][10];
mat0 (t1,20,40); /* correct */
mat0 (t2,30,10); /* correct */
mat0 (t1,20,5);  /* incorrect */
mat0 (t1,10,40); /* correct */
}
```

Fonctions et tableaux à plusieurs dimensions

Solution : passer la deuxième dimension en paramètre

```
void mat0 (float * t, int l , int c, int dim2) {
   int i,j;
/* rappel t[i][j] = *(t+i*dim2+j) */
   for (i=0;i<l;i++)
        for (j=0;j<c;j++)
              *(t+i*dim2+j) = 0.0;
}
void main(){
float t1[20][40],t2[30][10];
mat0 (t1,20,40,40); // correct mieux: mat0 (&t1[0][0],…)
mat0 (t2,30,10,10); // correct
mat0 (t1,20,5,40);  // correct
mat0 (t1,10,20,40); // correct
}
```

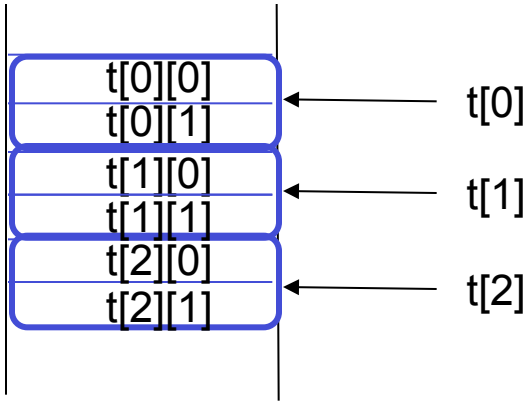
Tableaux à plusieurs dimensions variables

Rappels

t[dim1][dim2] ~ dim1 tableaux de dim2

int t[3][2] : tableau de 3 cases, chaque case est un tableau de 2 entiers

rappel : la première dimension n'est pas utilisée dans le calcul d'adresse
t[i][j] <-> *(t+i*dim2+j)

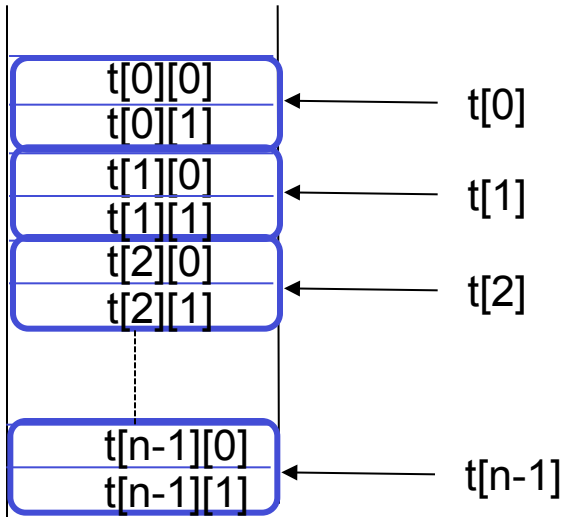


Première dimension "variable"

Première dimension inconnue à la déclaration ~ t[n] [2]

Même principe que pour les tableaux à une dimension

```
typedef int t2[2]; // t2 est un type
t2 * t;   // t est un pointeur sur un objet de type t2
scanf("%d",&n);
t = (t2 *) malloc(n*sizeof(t2));
```



L'accès aux éléments par la notation t[i][j] est utilisable puisque :

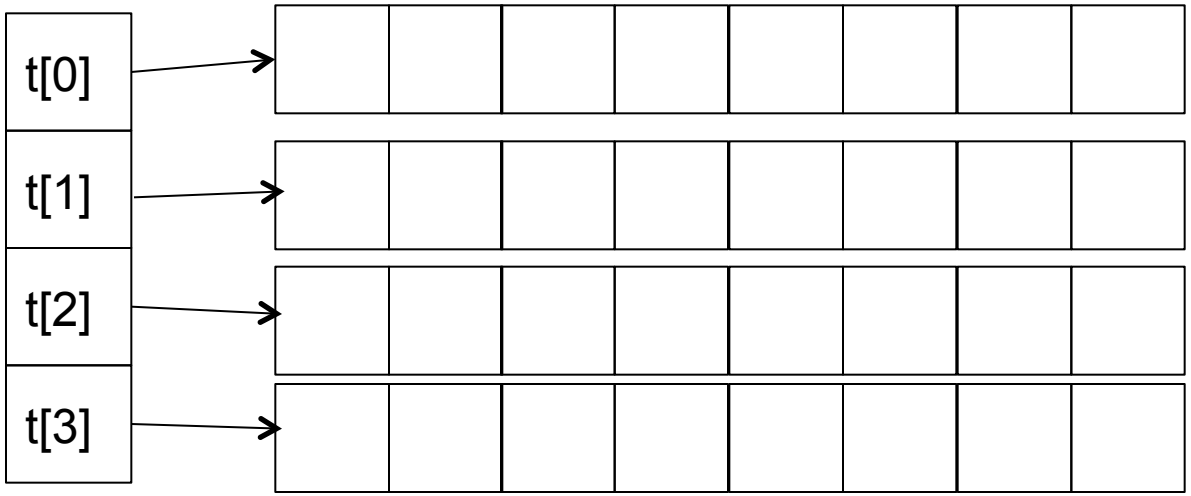
t[i][j] <-> *(t+i*2+j) // 2 est la taille d'un objet de type t2

Deuxième dimension "variable"

Deuxième dimension inconnue à la déclaration ~ t[4] [n]

Principe : on utilise un tableau de pointeurs

```
int * t[4]; // t est tableau de 4 pointeurs
// Création du tableau :
scanf("%d",&n);
for (i=0;i<4;i++)
  t[i] = (int *) malloc(n*sizeof(int));
```



Deuxième dimension "variable"

Accès aux éléménts :

t[i][j] est traduit en *(t[i]+j) : donc notation utilisable

Intérêt : tableau de chaines de caractères

De plus pour les chaines de caractères, l'initialisation est possible :

```
char * semaine[7]= {"lundi","mardi,..,"dimanche"};
```

Tableaux à plusieurs dimensions variables

Deux dimensions inconnues ~ t[n] [m] ~ matrice

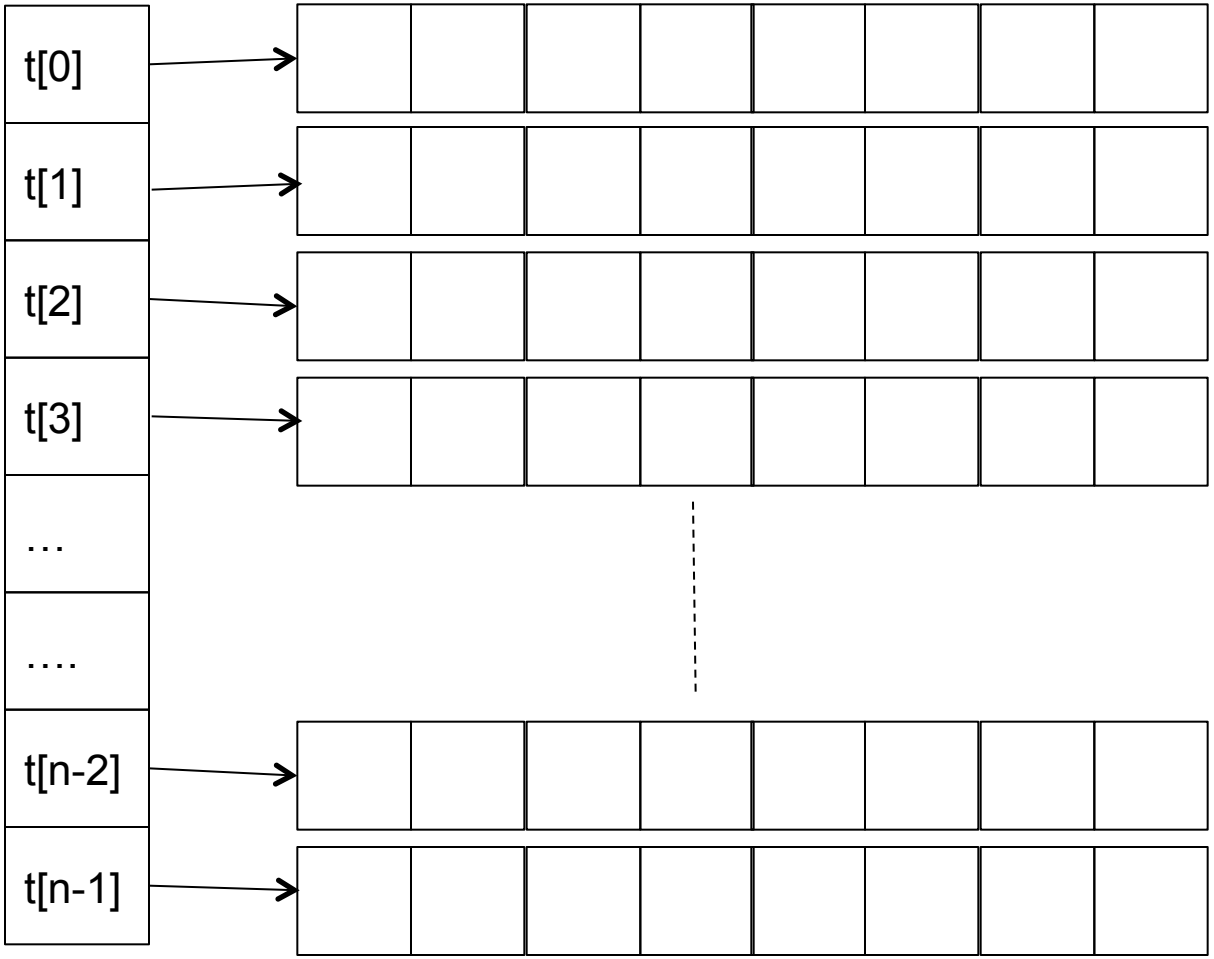
La deuxième dimension est inconnue à la déclaration mais sera constante

```
int ** t; // t est un pointeur sur un tableau de
  pointeurs
```

ex:

```
scanf("%d %d",&n,&m);
// allocation des lignes
  t = (int * *) malloc(n*sizeof(int*));
// allocations des colonnes
  for (i<0;i<n;i++)
   t[i]=(int *) malloc (m*sizeof(int));
```

Tableaux à plusieurs dimensions variables



Accès aux éléménts :

t[i][j] est traduit en *(*(t+i)+j) : donc notation utilisable

Tableaux à plusieurs dimensions

Deux dimensions inconnues, la deuxième peut varier

Exemple : stocker un nombre inconnu de chaines de caractères (tbl_n_m.c):

```
char ** dic;
int cpt = 0;
char tmp[100]; //tableau pour la lecture
int nb = 10;// nombre de chaines prévues par défaut
dic = (char **) malloc(nb * sizeof(char *));
for(int i=0;i<nb;i++) dic[i]=NULL;
printf("donner des noms\n");
while (gets(tmp) && strlen(tmp)!=0) {
  printf("len(%s)=%lu\n", tmp, strlen(tmp));
  dic[cpt] = (char *) malloc(strlen(tmp) + 1); // NB: place aux nuls!
  strcpy(dic[cpt],tmp);
  cpt++;
  if (cpt == nb) { // on augmente la première dimension
    nb=nb+10;
    dic = (char **) realloc(dic,nb*sizeof(char*));
    for(int i=nb-1;i>nb-11;i--) dic[i]=NULL;
  }
}
```

Exercice : trier les mot par ordre alphabétique inverse, protéger contre les attaques par débordement de tampon (cf « man gets »), et améliorer la maintenabilité.