Introduction au C/C++

Cours 2

Rémi Watrigant
(fortement inspiré du cours de V. Boudet, P. Giorgi et M. Joab de l'Université Montpellier 2)

Université de Nîmes

2013-2014

Rappel : programme minimal

Un programme C minimum se compose d'une fonction **main**, dont la version la plus simple est :

```
1 int main(){
2
3   déclaration de variables
4   instructions
5
6 return 0;
7 }
```

Rappel : programme minimal

Comme dans le langage algorithmique :

- les variables ont un type (ex : entiers, flottant)
- les instructions sont :
  - des instructions simples (ex : affectation)
  - des expressions (ex : $a+b\times c$)
  - des structure de contrôle (ex : tant que)

Les composants de base d'un programme C

Un programme en langage C est constitué des composants élémentaires suivants :

- les types de données,
- les constantes,
- les variables,
- les opérateurs,
- les structures de contrôle,
- les appels de fonctions

Plan

Les types de données

- **des nombres** : entiers, approximations flottantes des réels
- **et des lettres** : caractères

type	**type en C**	**valeurs**
entiers	`int`	$[-2^{31}, 2^{31}-1]$
réels (simple précision)	`float`	$\approx$ 7 chiffres significatifs
réels (double précision)	`double`	$\approx$ 15 chiffres significatifs
caractère	`char`	

- pas de type booléen en C on utilise les entiers :
  false $\rightarrow$ 0
  true $\rightarrow$ tout entier $\neq 0$
- en fait c'est un peu faux : type `bool` disponible depuis C99

Les types de données entiers

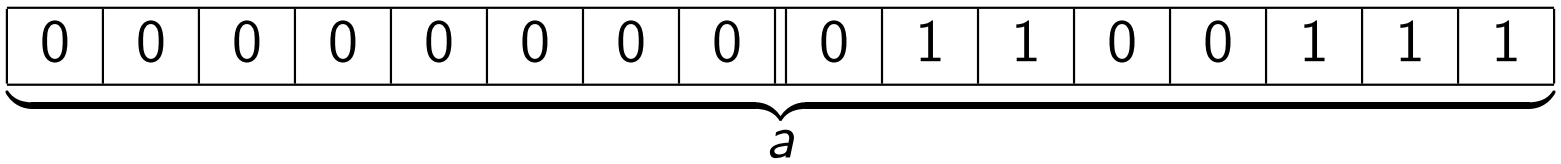
Il existe différentes variantes pour les entiers qui utilisent plus ou moins d'octets et qui ont des plages de valeurs différentes.

type entier	**type en C**	**nbr octets**	**valeurs**
court	`short int`	2	$[-2^{15}, 2^{15}-1]$
standard	`int`	4	$[-2^{31}, 2^{31}-1]$
long (64bits)	`long int`	8	$[-2^{63}, 2^{63}-1]$
long (32bits)	`long long int`	8	$[-2^{63}, 2^{63}-1]$

Exemple

```
short int a= 103;
```

$$103 = 64 + 32 + 4 + 2 + 1 = 2^6 + 2^5 + 2^2 + 2^1 + 2^0$$



Les types de données entiers

On peut étendre la valeur maximum des entiers en utilisant des entiers positifs (versions non signées)

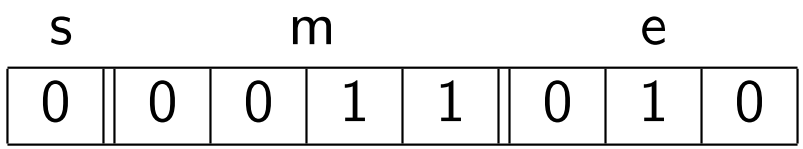
type en C	**nbr octets**	**valeurs**
`unsigned short int`	2	$[0, 2^{16} - 1]$
`unsigned int`	4	$[0, 2^{32} - 1]$
`unsigned long int`	8	$[0, 2^{64} - 1]$
`unsigned long long int`	8	$[0, 2^{64} - 1]$

Les types de données réels

Les réels ne sont pas représentables sur un ordinateur (à cause du codage binaire). Pour représenter un réel x, on utilise une approximation dite flottante (un rationnel particulier) :

$$x \approx (-1)^s \times m \times 2^{-e}$$

exemple : $0.75 = (-1)^0 \times 3 \times 2^{-2}$
Cela permet d'avoir un codage binaire car s (signe),m (mantisse),e (exposant) sont des entiers et donc représentables en binaire

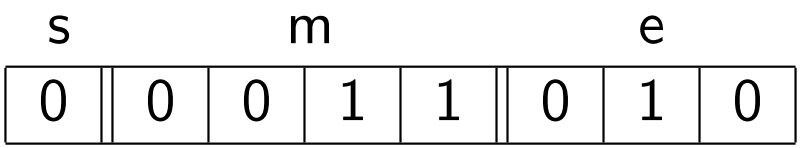


Les types de données réels

Les réels ne sont pas représentables sur un ordinateur (à cause du codage binaire). Pour représenter un réel x, on utilise une approximation dite flottante (un rationnel particulier) :

$$x \approx (-1)^s \times m \times 2^{-e}$$

exemple : $0.75 = (-1)^0 \times 3 \times 2^{-2}$
Cela permet d'avoir un codage binaire car s (signe),m (mantisse),e (exposant) sont des entiers et donc représentables en binaire



En C :

- `float` (4 oct.) : 1 bit pour s, 23 bits pour m, 8 bits pour e
- `double` (8 oct.) : 1 bit pour s, 52 bits pour m, 11 bits pour e

Le type de données caractère

Le mot-clef `char` désigne un objet de type caractère codé à partir de la table de conversion 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

Plan

Les constantes

Chaque type de données possède des constantes pouvant être utilisées dans un programme C pour :

- affecter une valeur à une variable,
- effectuer un calcul,
- former une expression booléenne.

Exemple

```
int a=10;
1+2+3+4+5;
1<2;
```

Les constantes entières

Elles sont définies en accord avec le type natif des entiers (`int`) et appartiennent donc à l'intervalle $[-2^{31}, 2^{31} - 1]$. ($2^{31} = 2147483648$)

```
1 int a;
2 a= 2147483647;    (OK    -> a=2^31 -1)
3 a= 2147483648;    (FAUX -> a= -2^31 au lieu de 2^31)
4 a= -2147483648;   (OK    -> a= -2^31)
5 a= -2147483649;   (FAUX -> a= 2^31-1 au lieu de -2^31-1)
```

ATTENTION : l'utilisation de constantes en dehors des valeurs autorisées ne provoquera aucune erreur lors de la compilation, les valeurs seront simplement tronquées pour satisfaire l'intervalle.

Les constantes entières

- plusieurs représentations sont possibles pour les constantes entières
  - décimal (en base 10) - ex : 1998
  - octal (en base 8) - ex : 03716 (commence par un zéro)
  - hexadécimal (en base 16) - ex : 0x7CE (commence par 0x)

Les constantes réelles (flottantes)

syntaxe :

- représentation par mantisse et exposant
- l'exposant est introduit par la lettre e
- ex : 2.34e4 représente 2.34×10^4

Les constantes caractères

Elles sont formées par l'expression utilisant les guillemets simples : '...' où les ... sont remplacés par :

- un caractère alphanumérique :

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
        1234567890;,.?!@*()+-[]:;<>  ETC.
```

- un caractère spécial :

  $\backslash n$ $\rightarrow$ retour à la ligne
  $\backslash t$ $\rightarrow$ tabulation
  $\backslash b$ $\rightarrow$ backspace
  ...

Plan

Manipulation de la mémoire

Rappel :

- la mémoire est binaire (un grand tableau de 0 et de 1)
- un langage impératif manipule directement la mémoire

Comment savoir ou se trouve les données en mémoire ?

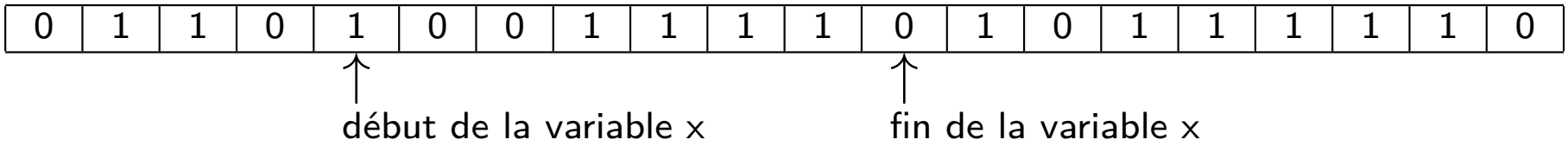
0	1	1	0	1	0	0	1	1	1	1	0	1	0	1	1	1	1	1	0

Manipulation de la mémoire

Rappel :

- la mémoire est binaire (un grand tableau de 0 et de 1)
- un langage impératif manipule directement la mémoire

Comment savoir ou se trouve les données en mémoire ? les variables

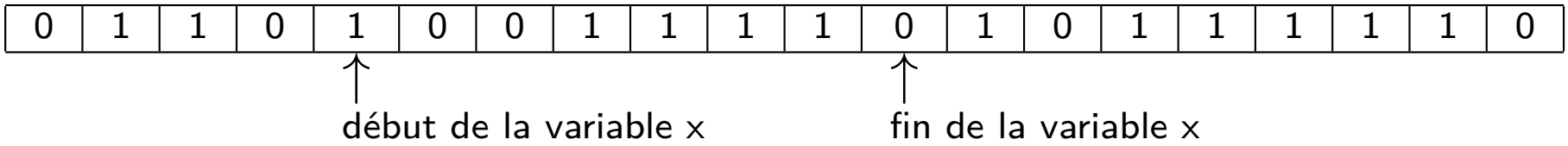


Manipulation de la mémoire

Rappel :

- la mémoire est binaire (un grand tableau de 0 et de 1)
- un langage impératif manipule directement la mémoire

Comment savoir ou se trouve les données en mémoire ? les variables



On identifie une variable par :

- une adresse de début dans l'espace mémoire
- une taille indiquant l'espace mémoire occupé par la variable

Les variables en C

Definition

On appelle variable une zone mémoire de l'ordinateur à laquelle on a donné un nom, ainsi qu'un type de données.

- le nom de la variable, appelé *identificateur*[1], permet de manipuler facilement les données stockées en mémoire.
- le type de données permet au compilateur de réserver l'espace mémoire suffisant pour stocker les valeurs.

(1) on utilise un identificateur plutôt que l'adresse mémoire mais l'on peut facilement récupérer l'adresse d'une variable à partir de son identificateur.

Les variables en C

Exemple

```
int a;
```

a est une variable de type entier :

- le compilateur lui réservera 4 octets en mémoire pour stocker ses valeurs
- on utilisera le nom « a » pour travailler avec l'espace mémoire attribué à la variable a.

Les variables en C

Exemple

```
double f;
```

f est une variable de type réel :

- le compilateur lui réservera 8 octets en mémoire pour stocker ses valeurs
- on utilisera le nom « f » pour travailler avec l'espace mémoire attribué à la variable `f`.

Plan

Les opérateurs arithmétiques

les opérateurs classiques

- addition : $+$
- soustraction : $-$
- division : $/$
- multiplication : $*$
- opposé : $-$
- modulo : $\%$

Attention : les opérateurs utilisent des opérandes du même type et renvoie un résultat du même type que les opérandes

- `int + int = int;`
- `float * float = float;`
- `int + float =???`

Priorité des opérateurs arithmétiques

Les règles de priorités mathématiques sont conservées :

- $a + b \times c$ sera interprété $a + (b \times c)$
- $(a + b) \times c$ sera interprété correctement car les parenthèses fixent l'ordre de priorité

nb : les opérations entre parenthèses sont prioritaires et seront toujours évaluées en premier

Opérateurs de comparaisons

les opérateurs classiques sur les types numériques :

égalité	==
différent	!=
supérieur	>
inférieur	<
supérieur ou égal	>=
inférieur ou égal 1	<=

Attention : le langage C ne dispose pas de type booléen en standard, le résultat d'un expression booléenne est un entier :

- 0 si l'expression est fausse (ex : $3 < 2$)
- 1 si l'expression est vraie (ex : $3 > 2$)

Opérateurs logiques

les opérateurs logiques en C :

- et : `&&`
- ou : `||`
- non : `!`

les opérandes sont de type numériques (pas de booléen en C)

- 0 correspond à faux
- toute valeur numérique non nulle correspond à vrai

Exemple

`(2.5 > 3.5) && (1 < 3)` est égale à 0

`0.5 || (1 >= 3)` est égale à 1

Opérateur d'affectation

Cette opérateur permet d'affecter la valeur d'une expression à une variable. En C l'affectation se fait avec =

- `var = exp`
  - `exp` : expression du même type que `var`
  - `var` : nom d'une variable déclarée
  - la variable `var` prend la valeur de l'expression `exp`
  - ex : `a=2+3;` `a` prend la valeur de l'expression `2+3` donc `5`

- comme en algorithmique, l'opérande de gauche ne peut être qu'une variable : l'affectation `a+b=3;` n'est pas correcte syntaxiquement.

Autres opérateurs

- incrémentation/décrémentation d'une variable entière `a` :
  - $a++$ incrémente la valeur de `a` par 1
  - $a--$ décrémente la valeur de `a` par 1

- affectations élargies : `+=`, `-=`, `*=`, `/=`
  - `a+=3;` correspond à l'expression `a=a+3;`

- taille mémoire des variables : `sizeof`
  - `sizeof(a)` renvoie la taille en octets de la variable `a`.
- et beaucoup d'autres ...

Opérateur d'adresse mémoire

Chaque variable est stockée en mémoire à une adresse précise. L'opérateur d'adresse & permet de récupérer l'adresse associée à une variable

- si a est une variable définie :
  `&a` renvoie la valeur de l'adresse de a

Attention

- l'adresse des variables n'est pas choisie par le programmeur : `&a=...` est interdit ! ! !
- l'adresse des variables peut être stockée dans une variable : `b=&a` si b a le bon type

Plan

Conversions de types

Attention, en C les types de données peuvent être changés implicitement.
L'ordre des conversions est le suivant :

```
char -> int -> float -> double
        signed -> unsigned
```

- lors de l'évaluation d'une expression :
  ex : `3.5+1` sera calculé comme `3.5+1.0`
  ↪ pas de perte d'information

- lors d'une affectation :
  `int a = 3.5;` sera effectué comme `int a= 3;`
  ↪ perte d'information possible

Plan

Affichage écran

L'intérêt majeur d'un programme est de faire effectuer les calculs par l'ordinateur. Cependant, le résultat doit être récupérable par l'utilisateur.

Une possibilité offerte par la plupart des langages compilés est l'affichage à l'écran :

en C, cela correspond à la fonction `printf`.

printf

Definition

`printf(chaine, var1, var2, ... , varn)`

- var1, var2, ... , varn sont les variables dont on souhaite afficher les valeurs.
- le nombre de ces variables est quelconque

printf

Definition

`printf(chaine, var1, var2, ... , varn)`

Le `printf` :

- n'utilise pas le type des variables pour afficher leurs valeurs
- considère les variables comme une zone mémoire non typée

- `chaine` est une chaine de caractère servant de modèle pour l'affichage des variables. Ce modèle fixe :
  - le format d'interprétation des variables
  - le texte fixe à afficher en plus des variables

printf : exemple 1

Exemple

Afficher une variable de type entier à l'écran

```c
#include <stdio.h>

int main(){

   int a = 3;
   printf("l'entier a est égal à %d \n", a);

 return 0;
}
```

Ce code affiche à l'écran :

l'entier a est égal à 3

`printf` : les formats d'affichage

un format doit être inclus dans la chaîne du `printf` pour chaque variable à affiché.

les formats disponibles sont :

- %c → char
- %d → int
- %f → float (écriture décimale)
- %e → double ou float (écriture scientifique)
- %s → chaîne de caractère (passage variable par adresse)

printf : exemple 2

On peut afficher plusieurs variables en même temps :

```
1 #include <stdio.h>
2
3 int main(){
4   int a = 3;
5   float b = 0.0234;
6   printf("l'entier a= %d \n le flottant b= %e \n", a, b);
7  return 0;
8 }
```

Ce code affiche à l'écran :

l'entier a= 3

le flottant b= 2.340000e-2

`printf` : maîtriser son affichage

On peut spécifier au `printf` le gabarit d'affichage d'une variable (nbr caractères).

- on ajoute un entier entre le % et le format de la variable
- ex : `"%10d"`

L'entier spécifie le nombre minimum de chiffres à afficher : des blancs sont ajoutés si le gabarit est supérieur au nombre de chiffres de la variable.

`printf` : maîtriser son affichage

On peut spécifier au `printf` la précision d'affichage des variables flottantes.

- on ajoute un point suivi d'un entier après le gabarit d'une variable
- ex : `"%3.8f"`

L'entier spécifie le nombre de chiffres après la virgule à afficher : des zéros sont ajoutés si la précision est supérieure au nombre de chiffres de la variable.

printf : exemple 3a

```
1  #include <stdio.h>
2
3  int main(){
4    int a1,a2,a3;
5    a1=1;
6    a2=100;
7    a3=10000;
8    printf("voici 3 entiers:\n%5d\n%5d\n%5d\n",a1,a2,a3);
9   return 0;
10 }
```

Ce code affiche à l'écran :

```
voici 3 entiers :
    1 (avec 4 blancs devant)
  100 (avec 2 blancs devant)
10000
```

printf : exemple 3b

```
1  #include <stdio.h>
2
3  int main(){
4     float b1,b2,b3;
5     b1=0.1;
6     b2=0.001;
7     b3=0,00001;
8     printf("voici 3 réels:\n%5.3f\n%5.3f\n%5.3f\n",b1,b2,b3);
9   return 0;
10 }
```

Ce code affiche à l'écran :

voici 3 réels :
0.100
0.001
0.000

Plan

Intérêt des structures de contrôles

Contrôler l'exécution des instructions du programme :
notion d'algorithmes

- exécution conditionnelle (si alors sinon)
- exécuter plusieurs fois les mêmes instructions (boucle pour)
- exécuter des instructions tant que (boucle tant que)

Plan

if ... else

Definition

```
if (exp)
   instr1
else
   instr2
```

- exp est une expression booléenne
- instr1 et instr2 sont :
  - une instruction
  - une suite d'instruction entre accolades (bloc d'instructions)

Exemple: if ... else

Exemple

Déterminer si un entier est pair ou impair

```
1  #include <stdio.h>
2
3  int main(){
4     int a;
5     a= 17;
6
7     if (a%2 == 0)
8        printf("%d est pair\n",a);
9     else
10       printf("%d est impair\n",a);
11
12    return 0;
13 }
```

Exemple: if ... else

Exemple

Déterminer le plus grand et le plus petit entre deux entiers

```
1  #include <stdio.h>
2  int main(){
3     int a,b,max,min;
4     a=17;   b=13;
5     if (a > b){
6         max=a;
7         min=b;
8     }
9     else {
10        max=b;
11        min=a;
12    }
13    printf("le min est %d le max est %d \n",min,max);
14    return 0;
15 }
```

Plan

Répétition d'instructions : la boucle pour

Intérêt : répéter un nombre de fois donné une même suite d'instructions.

Exemple

calculer la somme des entiers entre 1 et 10

```
s :=0 ;
pour i de 1 à 10 faire
    s :=s+i ;
fin pour ;
```

En C, les boucles **pour** sont effectuées avec l'instruction `for`

L'instruction `for`

Definition

```
for (exp1; exp2; exp3)
     instr;
```

- `exp1` est une expression quelconque évaluée une seule fois au début de la boucle
- `exp2` est une expression booléenne qui permet d'arrêter la boucle
- `exp3` est une expression quelconque évaluée à chaque tour de boucle (en dernier).

- `instr` est une instruction ou un bloc d'instructions

L'instruction `for`

Definition

```
for (exp1; exp2; exp3)
     instr;
```

- `exp1` est une expression quelconque évaluée une seule fois au début de la boucle

On se sert de exp1 pour initialiser la variable de boucle préalablement déclarée.
ex : exp1 est remplacée par `i=1`

L'instruction `for`

Definition

```
for (exp1; exp2; exp3)
     instr;
```

- exp2 est une expression booléenne

Si exp2 est :

- vrai : la boucle `for` continue et on exécute `instr`
- faux : on sort de la boucle `for` sans exécuter `instr`

ex : exp2 est remplacée par `i<11`

L'instruction `for`

Definition

```
for (exp1; exp2; exp3)
     instr;
```

- exp3 est une expression quelconque évaluée à chaque tour de boucle (en dernier).

On sert de exp3 pour modifier la variable de boucle (incrémentation ou décrémentation) :
ex : exp3 est remplacée par `i=i+1`

L'instruction `for` : schéma d'exécution

Definition

```
for (exp1; exp2; exp3)
     instr;
```

1. exp1

   ...
2. exp2 $\rightarrow$ on continue car exp2=true
3. instr
4. exp3

   ...
5. exp2 $\rightarrow$ sort de la boucle car exp2=false

L'instruction `for` : exemple 1

Exemple

calculer la somme des entiers entre 1 et 10

```
1  #include <stdio.h>
2  int main(){
3     int i,s;
4     s=0;
5     for (i=1;i<11;i++) // i++ est équivalent à i=i+1
6        s=s+i;
7
8     printf("la somme est: %d",s);
9     return 0;
10 }
```

ce programme affiche : la somme est 55

L'instruction `for` : exemple 2

Exemple

afficher les entiers entre 1 et 10 qui sont multiples de 2 ou de 3

```c
#include <stdio.h>
int main(){
   int i;
   for (i=1;i<11;i++){
      if (i%2==0) || (i%3==0)
         printf("%d ",i);
   }
   printf("\n");
   return 0;
}
```

ce programme affiche : 2 3 4 6 8 9 10

L'instruction `for` : ce qu'il ne faut pas faire

Attention aux boucles qui ne se terminent jamais ! ! !
les boucles infinies ...

```
1  int i,s;
2  s=0;
3  for (i=1;i<11;s=s+i)
4     ...
```

→ la variable de boucle n'est pas incrémentée

```
1  int i,s;
2  for (i=1;i<11;i--)
3     ...
```

→ la condition d'arrêt de la boucle n'est jamais atteinte

Plan

Répétition d'instructions : la boucle tant que

Intérêt : répéter une instruction tant qu'une condition est vérifiée

Exemple

calculer la somme des entiers entre 1 et 10

```
s :=0 ;
i :=1 ;
tant que i<11 faire
    s :=s+i ;
    i :=i+1 ;
fin tant que ;
```

En C, les boucles **tant que** sont effectuées avec l'instruction `while` !

L'instruction `while`

Definition

```
while (exp)
  instr;
```

- `exp` est une expression booléenne permettant de contrôler la boucle
- `instr` est une instruction ou un bloc d'instructions

L'instruction `while`

Definition

```
while (exp)
  instr;
```

- `exp` est une expression booléenne permettant de contrôler la boucle

Si `exp` est :

- vrai : la boucle `while` continue et on exécute `instr`
- faux : on sort de la boucle `while` sans exécuter `instr`

ex : exp est remplacée par `i<11`

L'instruction `while` : schéma d'exécution

Definition

```
while (exp)
   instr;
```

1. `exp` → on rentre dans la boucle car `exp`=true
2. `instr`

   ...
3. `exp` → on continue car `exp`=true
4. `instr`

   ...
5. `exp` → on sort de la boucle car `exp`=false

   ...

L'instruction `while` : exemple 1

Exemple

calculer la somme des entiers entre 1 et 10

```
1  #include <stdio.h>
2  int main(){
3     int i,s;
4     s=0;
5     i=1;
6     while (i<11){
7        s=s+i;
8        i++; // on peut aussi écrire i=i+1
9     }
10    printf("la somme est: %d",s);
11    return 0;
12 }
```

ce programme affiche : la somme est 55

L'instruction `while` : exemple 2

Exemple

afficher les entiers entre 1 et 10 qui sont multiples de 2 ou de 3

```
1  #include <stdio.h>
2  int main(){
3     int i;
4     i=1;
5     while (i<11){
6        if (i%2==0) || (i%3==0)
7           printf("%d ",i);
8        i++;
9     }
10    printf("\n");
11    return 0;
12 }
```

ce programme affiche : 2 3 4 6 8 9

L'instruction `while` : exemple 3

Exemple

calcul de la plus petite puissance de 2 supérieure à un entier

```
1  #include <stdio.h>
2  int main(){
3     int a,p;
4     a=27;
5     p=1;
6     while (p<a){
7        p=2*p;
8     }
9     printf("%d est supérieur à %d \n",p,a);
10    return 0;
11 }
```

ce programme affiche : 32 est supérieur à 27

L'instruction `while` : exemple 4

cas où la boucle tant que est nécessaire : *calcul du pgcd de 2 entiers*

```
1  #include <stdio.h>
2  int main(){
3     int a,b,c,r1,r2;
4     a=30; b=18;
5     r1=a; r2=b;
6     while (r2 != 0){
7        c=r2;
8        r2=r1%r2;
9        r1=c;
10    }
11    printf("le pgcd de %d et %d est %d \n",a,b,r1);
12    return 0;
13 }
```

ce programme affiche : le pgcd de 30 et 18 est 6

Plan

L'instruction `do while`

Parfois, il est souhaitable d'exécuter le corps de boucle avant la condition de boucle (`instr` avant `exp`).

Dans ce cas, on peut utiliser l'instruction `do {} while;`

Definition

```
do {
   instr;
} while (exp);
```

- `instr` et `exp` sont identiques à ceux utilisés dans la boucle `while` classique

RAPPEL : schéma d'exécution du `while`

Definition

```
while (exp)
   instr;
```

1. `exp` → sort de la boucle si `exp`=false
2. `instr`

   ...
3. `exp` → sort de la boucle si `exp`=false
4. `instr`

   ...
5. `exp` → sort de la boucle si `exp`=false
6. `instr`

   ...

L'instruction do{} while : schéma d'exécution

Definition

```
do {
   instr;
}
while (exp);
```

1. instr
   ...
2. exp → sort de la boucle si exp=false
3. instr
   ...
4. exp → sort de la boucle si exp=false
5. instr
   ...

L'instruction do{} while : exemple

calculer le plus grand entier tel que son carré est inférieur à un entier x.

```
1  #include <stdio.h>
2  int main(){
3     int i,s,x;
4     x=1000;
5     i=0;
6     do {
7        i++;
8        s=i*i;
9     } while (s <= x);
10    i=i-1;
11    printf("l'entier est %d \n",i);
12    return 0;
13 }
```

ce programme affiche : l'entier est 31

Plan

Structures de contrôle : erreur classique

L'erreur classique avec les structures de contrôle est l'oubli d'accolade pour définir un bloc d'instructions :

```
1  ...
2     if (a > b){
3         max=a;
4         min=b;
5     }
6     else
7         max=b;
8         min=a; // ATTENTION: n'appartient pas au else
9
10    printf("le min est %d le max est %d \n",min,max);
11 ...
```

Structures de contrôle : erreur classique

L'erreur classique avec les structures de contrôle est l'oubli d'accolade pour définir un bloc d'instructions :

```
1  ...
2     if (a > b)
3         max=a;
4         min=b;
5     else {
6         max=b;
7         min=a; // grâce aux accolades appartient au else
8     }
9     printf("le min est %d le max est %d \n",min,max);
10 ...
```

L'oubli des accolades sur le bloc du `if` génère une erreur de compilation car le `else` se retrouve isolé.

Structures de contrôle : erreur classique

L'erreur classique avec les structures de contrôle est l'oubli d'accolade pour définir un bloc d'instruction :

```
1  ...
2     int i,s;
3     s=0;
4     i=1;
5     while (i<11)
6         s=s+i;
7         i++; // n'appartient pas à la boucle
8
9     printf("la somme est: %d",s);
10 ...
```

Structures de contrôle : erreur classique

L'erreur classique avec les structures de contrôle est l'oubli d'accolade pour définir un bloc d'instruction :

```
1  ...
2     int i,s;
3     s=0;
4     i=1;
5     while (i<11){
6         s=s+i;
7         i++; // grâce aux accolades appartient a la boucle
8     }
9     printf("la somme est: %d",s);
10 ...
```

Récapitulatifs des bases en C

- les variables ont un type (ex : int, float), pas de booléens dans les standards
- on peut calculer grâce aux opérateurs (+,*,%,...)
- on affecte une valeur à une variable avec l'opérateur = (ex : a=6)
- Attention aux conversions de type implicite
- on affiche les valeurs à l'écran avec `printf`
- on peut écrire des algorithmes avec les structures de contrôle classique : `if else`, `for`, `while`

Plan

Plan

A quoi sert une fonction

Les fonctions servent à partitionner les gros traitements en tâches plus petites.

Elle permettent de définir des briques de calcul qui sont :

- identifiables facilement,
- conçues pour faire un traitement précis,
- réutilisables.

Remarque

Les programmes sont plus lisibles et donc plus faciles à maintenir.

Fonction : une brique de calcul parametré

On veut écrire un programme qui infirme le théorème de Fermat :

il n'existe pas $x, y, z \in \mathbb{N}^*$ tel que $x^n + y^n = z^n, \quad \forall n > 2$

```
1  #include "lecture.h"
2  #include <stdio.h>
3  int main(){
4    int x,y,z;
5    x=lire_entier();
6    y=lire_entier();
7    z=lire_entier();
8    if (x*x*x+y*y*y == z*z*z)
9          printf("Fermat a faux !!!");
10   if (x*x*x*x+y*y*y*y == z*z*z*z)
11         printf("Fermat a faux !!!");
12   // IMPOSSIBLE D'ECRIRE TOUTES LES PUISSANCES !!!
13   return 0;
14 }
```

Fonction : une brique de calcul parametré

Remarque

Le seul moyen d'écrire un tel programme est d'utiliser une fonction de calcul de puissance parametré par l'exposant

```
1 // calcul de a^n
2 int puissance(int a, int n)
3 {
4   int i,p;
5   p=1;
6   for (i=1;i<=n;i++)
7     p=p*a;
8   return p;
9 }
```

Fonction : une brique de calcul parametré

On peut alors envisager d'écrire le programme suivant :

```
1  int main(){
2   int x,y,z,nmax,n;
3   x=lire_entier();
4   y=lire_entier();
5   z=lire_entier();
6   nmax=lire_entier();
7
8   for (n=3;n<=nmax;n++)
9   {
10    if (puissance(x,n)+puissance(y,n) == puissance(z,n))
11        printf("Fermat a faux !!!");
12  }
13  return 0;
14 }
```

Qu'est-ce qu'une fonction

Definition

Une fonction est une description formelle d'un calcul en fonction de ses arguments

Exemple

Si on vous donne deux variables entières a et b décrivez les instructions du langage nécessaire pour calculer a^b.

$\rightarrow$ cela rejoint la notion d'algorithme

Qu'est-ce qu'une fonction

Comme en algorithmique, une fonction comportera :

- un nom (l'identifiant de manière non-ambigu),
- des paramètres d'entrées,
- des paramètres de sortie (C sera très restrictif),
- des variables locales,
- une description dans le langage de ce que fait la fonction.

Plan

Définition de fonctions

Definition

```
type_retour nom(liste_param_formel){
   corps
}
```

- `nom` correspond au nom donné à la fonction
- `type_retour` est le type du résultat de la fonction
- `liste_param_formel` est la liste des variables d'entrée de la fonction
- `corps` correspondant aux instructions effectuées par la fonction en fonction des paramètres formels.

Fonctions : nom

Definition

Le nom d'une fonction correspond à un identifiant unique permettant d'effectuer l'appel sans aucune ambiguité

Attention

Une fonction ne peut être assimilée à une variable et vice-versa. On les différencie par la présence de parenthèse après l'identifiant d'une fonction.

```
1 int a();
2 a=3; // ERREUR a n'est pas une variable
3 int b;
4 b(); // ERREUR b n'est pas une fonction
```

Fonctions : nom

Attention

Une variable et une fonction ne peuvent pas partager le même nom au sein d'un même bloc.

```
1 int a(); // déclaration fonction
2 int a;   // déclaration variable
```

Le compilateur générera l'erreur suivante :

error : 'a' redeclared as different kind of symbol

Fonctions : type de retour

Les fonctions en C ne possède qu'un seul paramètre de sortie.
Ce paramètre :

- n'est pas identifié par une variable particulière
- est spécifié uniquement par son type

```
1 int f();      // f renvoie un resultat entier
2 double g(); // g renvoie un resultat flottant
```

Si la fonction ne renvoie rien, on utilise le type `void`.
ex : `void h();`

Fonctions : paramètres formels

La liste des paramètres formels d'une fonction est :

- vide si la fonction n'a aucun paramètre
- de la forme : $type_1$ `p1, ... ,` $type_n$ `pn`

La déclaration des variables formelles ($type_i$ `pi`) est de la forme :

- `type pi` pour une variable simple
- ... (à voir plus loin dans le cours)

où `type` est un type de base et `pi` est le nom de la variable formelle dans la fonction.

Fonctions : paramètres formels

Les paramètres formels d'une fonction permettent :

- de définir le code de la fonction à partir de données inconnues.
- de transmettre (lors de l'appel) les données réelles sur lesquelles doit agir la fonction.

Ils ne sont connus que par leur fonction associée.

```
1 int  f(int a, int b);
2 void g(double a);
3 a=12; // erreur a n'est pas déclaré
```

Fonctions : corps

Definition

Le corps d'une fonction est le bloc défini juste après la déclaration de l'interface de la fonction `nom(liste_param_formel)`

Le corps d'une fonction est composé :

- de la déclaration des variables locales
- de la déclaration du code effectué par la fonction

```
1 int plus(int x,int y){
2   int z;
3   z=x+y;
4   return z;
5 }
```

Fonction : le mot clé `return`

Definition

Dans une fonction, le mot clé `return` *exp* permet :

- d'arrêter l'exécution du corps
- de renvoyer la valeur de l'expression *exp* au bloc appelant

Attention

La valeur de *exp* doit être du même type que le type de retour de la fonction (*sinon conversion implicite*)

```
1 double inverse(int x){
2   return 1/x;
3 }
```

Fonction : le mot clé `return`

Le mot clé `return` peut apparaitre plusieurs fois dans une même fonction. C'est le premier `return` rencontré qui stoppera le corps de la fonction (ex : conditionnelle).

```
1 int impair(int x){
2   if (x%2 == 0)
3     return 0;
4   else
5     return 1;
6 }
```

Fonction : le mot clé `return`

Le mot clé `return` n'est pas obligatoire lorsque le retour de la fonction est `void`. La fonction s'arrête dès qu'elle atteint la fin du bloc.

```
1 void bonjour(){
2   printf("Bonjour\n");
3 }
```

Fonction : le mot clé `return`

On peut toutefois forcer l'arrêt en appelant `return` sans expression.

```
1 void bonjour(int heure){
2   if (heure > 8 && heure < 20){
3     printf("Bonjour\n");
4     return;
5   }
6   printf("Bonsoir\n");
7 }
```

Fonctions : pas d'amalgame

ATTENTION

La déclaration d'une fonction est une description formelle d'un calcul, elle n'exécute rien toute seule : il faut *appeler* la fonction pour que les instructions du corps soient effectuées.

```
1 void bonjour(){
2   printf("Bonjour\n");
3 }
```

Ce code ne fait que déclarer un fonction, il n'affichera rien à l'écran si on l'éxécute.

Plan

Appel de fonction

Definition

```
var = nom(liste des paramètres effectifs);
ou
nom(liste des paramètres effectifs);
```

- la liste des paramètres effectifs correspond à l'ensemble des variables et des constantes que l'on souhaite donner comme argument à la fonction. ex : max(2,3).

Attention : le passage des arguments se fait par copie
la valeur des paramètres effectifs est copiée dans les variables formelles correspondantes.

Appel de fonction : exemple

```c
#include <stdio.h>

int max(int a, int b){
   if (a>b) return a else return b;
}
int main(){
   int x,y;
   x=3;
   y=5;
   printf("le max est : %d\n",max(x,y));
   return 0;
}
```

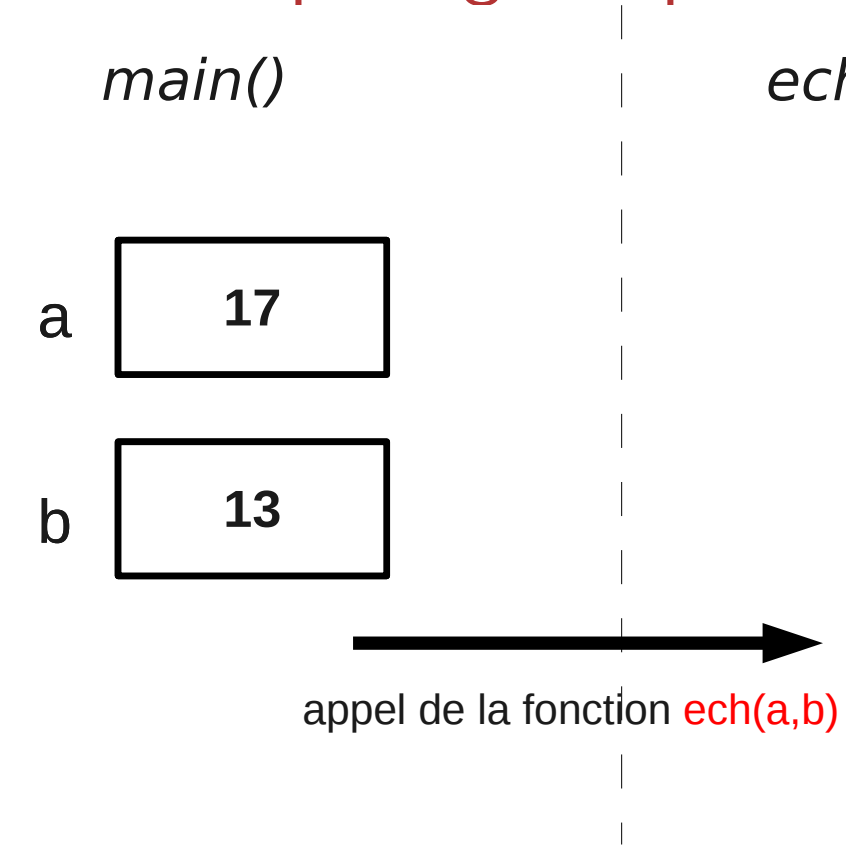
Fonction : passage de paramètres par valeur

Les paramètres d'une fonction sont toujours initialisés par une copie des valeurs des paramètres réels.

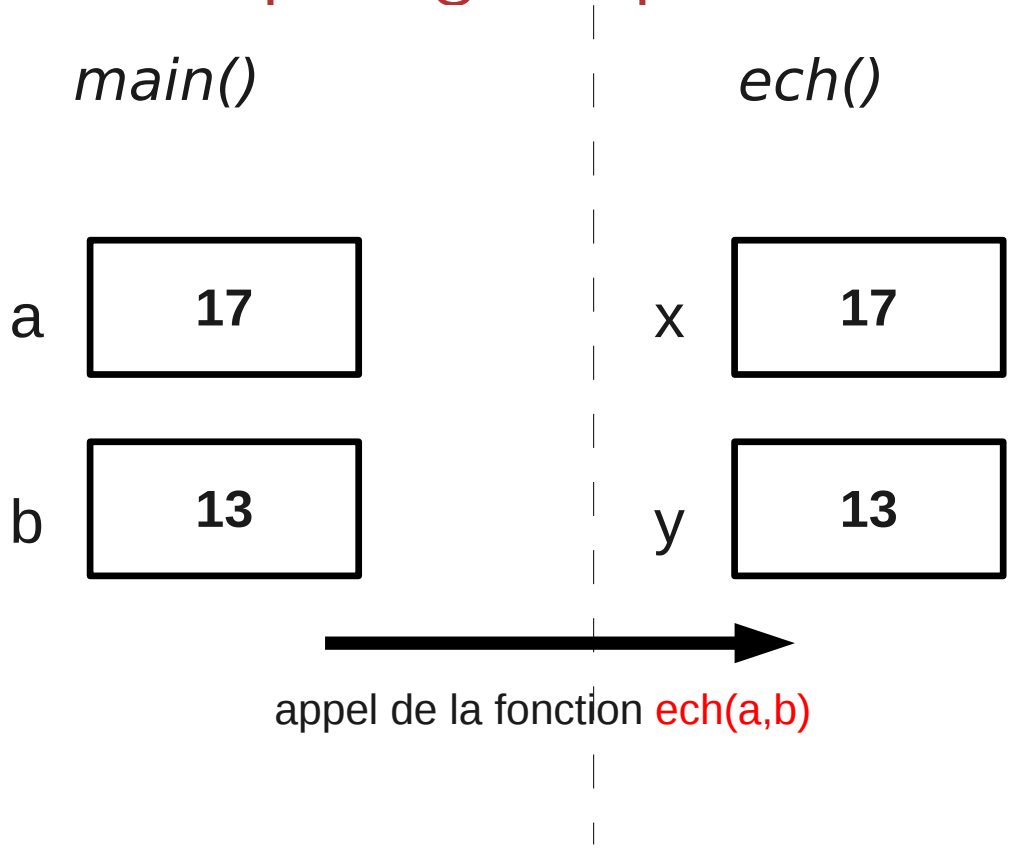
Modifier la valeur des paramètres formels dans le corps de la fonction ne change en aucun cas la valeur des paramètres réels.

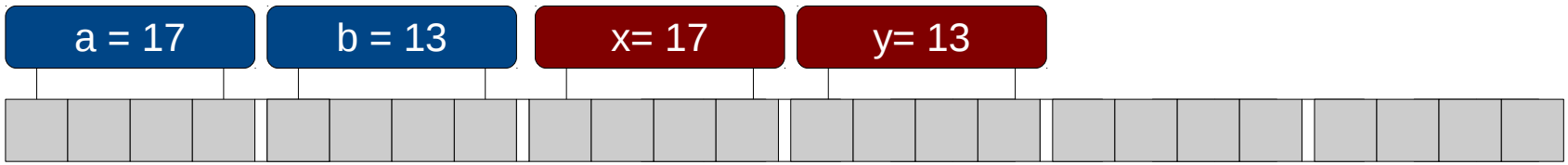
```
1 void ech(int x, int y){
2   int r;
3   r=x; x=y; y=r;
4 }
5 int main(){
6   int a,b;
7   a=17;b=13;
8   ech(a,b); // ne change pas la valeur de a et de b
9 }
```

Fonction : passage de paramètres par valeur

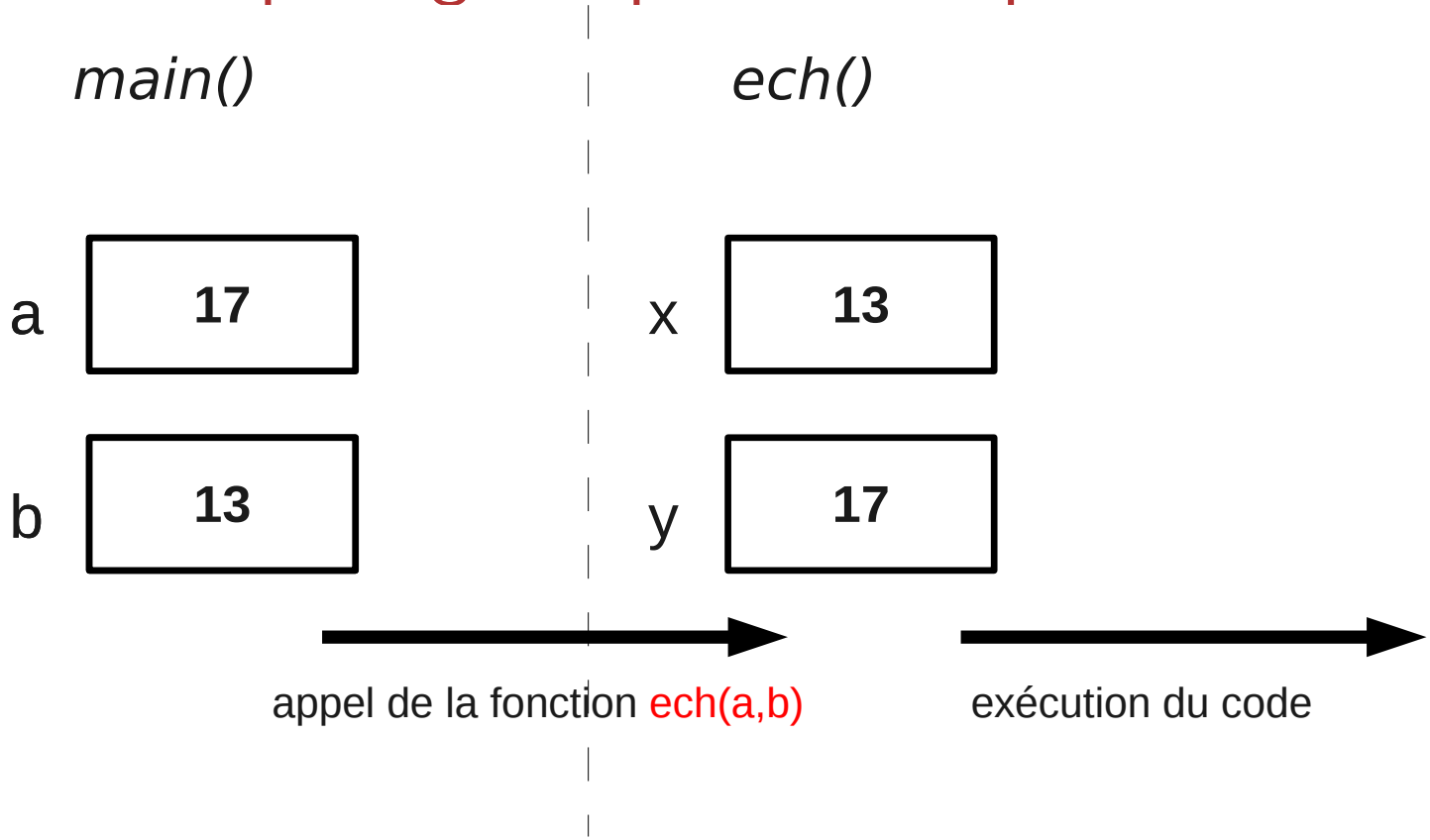


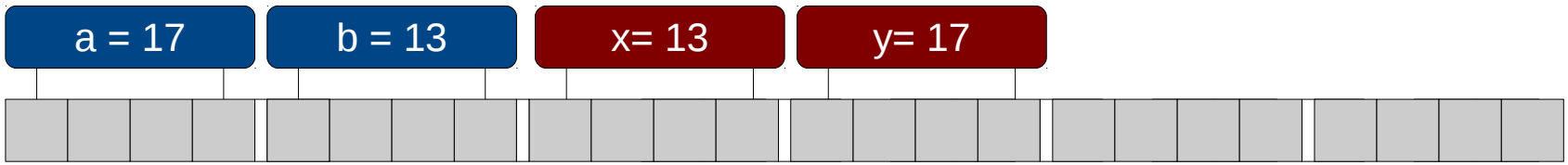
Fonction : passage de paramètres par valeur



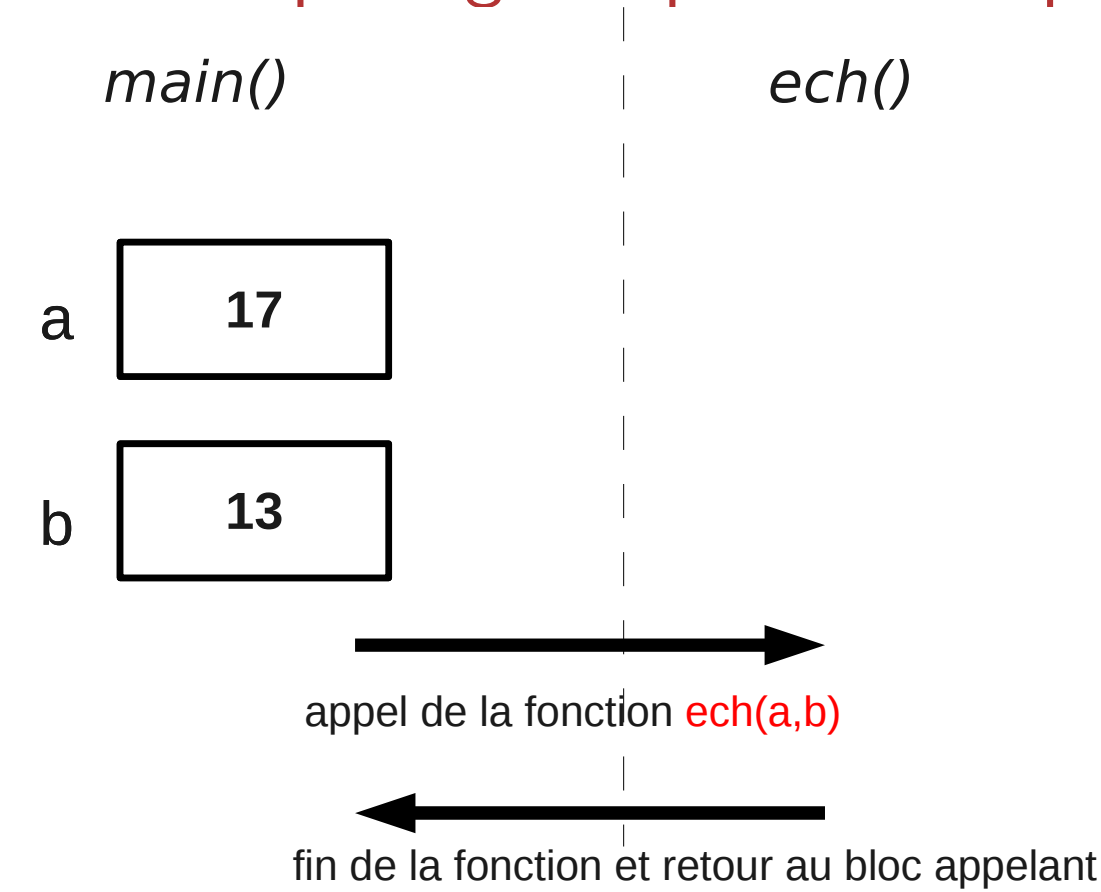


Fonction : passage de paramètres par valeur





Fonction : passage de paramètres par valeur



Plan

Où utiliser une variable ?

Question

Quand on déclare une variable, où-est-elle connue ? où peut-on l'utiliser ?

RÃ©ponse

Dans tout le bloc où elle a été déclarée et à partir de sa déclaration.

Portée d'une variable

Definition

On appelle Portée d'une variable, la portion du code ou cette variable est connue et utilisable.

Exemple 1

Quelles sont les portées de a et de x ?

```
1 int main()
2 {
3     int a;
4     float x;
5     ...
6     ...
7     return 0;
8 }
```

Exemple 1

Quelles sont les portées de a et de x ?

```
1  int main()
2  {                    <-- début du bloc
3      int a;           <-- début de la portée de a
4      float x;         <-- début de la portée de x
5      ...
6      ...              <-- utilisation possible de a et x
7      return 0;
8  }                    <-- fin de bloc et de portée de a et x
```

Exemple 2

Est-ce que l'utilisation de a et de x est conforme à leur portée ?

```
1 int main()
2 {
3     int a=3;
4     x=2./a;
5     float x;
6     printf("a=%d et x=%f\n",a,x);
7     return 0;
8 }
```

Exemple 2

Est-ce que l'utilisation de a et de x est conforme à leur portée ?

```
1  int main()
2  {
3      int a=3;          <-- début portée de a
4      x=2./a;           <-- on peut utiliser a mais pas x
5      float x;          <-- début portée de x
6      printf("a=%d et x=%f\n",a,x);
7      return 0;
8  }
```

Exemple 2

```
1 int main()
2 {
3     int a=3;
4     x=2./a;
5     float x;
6     printf("a=%d et x=%f\n",a,x);
7     return 0;
8 }
```

Le compilateur vous affiche alors le message suivant :
exemple2.c :4 error : 'x' undeclared (first use in this function)

Portée d'une variable

Definition

On appelle Portée d'une variable, la portion du code ou cette variable est connue et utilisable.

Remarque

Dans le cas de blocs imbriqués, une variable est utilisable dans tout le bloc où elle a été déclarée, à partir de sa déclaration et également dans les blocs internes au bloc de la déclaration.

Exemple

```
1 int main(){
2     int i;
3     for (i=0;i<10;i++){
4         int S;
5         S=2*i*i-1;
6         printf("S=%d\n",S);
7     }
8     return 0;
9 }
```

Exemple

```
1 int main(){
2     int i;                    <-- début portée de i
3     for (i=0;i<10;i++){
4         int S;                <-- début portée de S
5         S=2*i*i-1;
6         printf("S=%d\n",S);
7     }                         <-- fin portée S
8     return 0;
9 }                             <-- fin portée i
```

Exemple

```
1  int main(){
2      int i;
3      for (i=0;i<10;i++){
4          int S;
5          S=2*i*i-1;
6          printf("S=%d\n",S);
7      }
8      printf("S=%d\n",S);      <-- impossible
9      return 0;
10 }
```

Plusieurs variables ?

Question

Que se passe-t-il si on a plusieurs variables de même nom ?

RÃ©ponse

Impossible d'avoir deux variables de même nom dans le même bloc, mais on peut avoir des variables identiques dans des blocs imbriqués.

Exemple

Qu'affiche le code suivant ?

```
1 int main(){
2     int a=3;
3     {
4         int a=5;
5         printf("a=%d\n",a);
6     }
7     return 0;
8 }
```

Exemple

Qu'affiche le code suivant ?

```
1 int main(){
2     int a=3;
3     {
4         int a=5;
5         printf("a=%d\n",a);
6     }
7     return 0;
8 }
```

Le programme affiche :

a=5

Plusieurs variables ?

Question

Que se passe-t-il si on a plusieurs variables de même nom ?

RÃ©ponse

La variable utilisée est celle du bloc englobant le plus proche (au sens de l'inclusion).

Plusieurs variables ?

Question

Et dans le cas de fonctions ?

```
1  float cube(float x){
2      float c=x*x*x;
3      return c;
4  }
5  int main(){
6      float c;
7      c=cube(2);
8      printf("le cube de 2 est %f\n",c);
9      return 0;
10 }
```

Plusieurs variables ?

Question

Et dans le cas de fonctions ?

Il n'y a pas de problème : la portée d'une variable étant le bloc où elle a été déclarée, la variable n'existe qu'à l'intérieur de la fonction.

RÃ©ponse

Tout ce qui a été dit précédemment s'applique pour les fonctions.

Plusieurs variables : continuons

Question

L'étudiant qui suit : "Mais si on veut tout de même utiliser la même variable dans plusieurs fonctions ?"

Rien de plus facile, on passe la variable en paramètre ! ! !

Plusieurs variables : continuons

Question

L'étudiant têtu : "C'est tout de même idiot de rajouter une variable à toutes nos fonctions ?"

RÃ©ponse

C'est vrai ! On peut tout de même faire quelque chose : variables globales !

Variables globales

Definition

Une variable est globale si elle est définie en dehors de tout bloc (et donc de toute fonction).

Question

La portée d'une variable est son bloc. Une variable globale n'a pas de bloc, quelle est alors sa portée ?

Variables globales

Une variable globale est connue dans toute la partie du fichier qui suit sa déclaration.

```
1  void uneautrefonction(){
2      // ne peut pas utiliser a
3  }
4
5  int a;
6
7  int mafonction(){
8      // peut utiliser la variable a
9  }
10
11 int main(){
12     // peut utiliser la variable a
13     return 0;
14 }
```

Variables globales : c'est dangereux !

Qu'affiche le code suivant :

```
1  int i;
2
3  void coucou(){
4      for (i=0;i<3;i++)
5          printf("coucou %d fois\n",i);
6  }
7
8  int main(){
9      for (i=0;i<3;i++)
10         coucou();
11     return 0;
12 }
```

Variables globales : c'est dangereux !

On a 3 appels à la fonction `coucou()`. Chaque exécution de cette fonction devrait afficher 3 messages. On s'attend donc à 9 messages. Au lieu de cela, nous obtenons :

coucou 0 fois
coucou 1 fois
coucou 2 fois

Pourquoi c'est dangereux !

Voilà l'exécution du programme :

main : i=0

main : appel de `coucou()`

- coucou : i=0
- coucou : affiche "coucou 0 fois"
- coucou : i=1
- coucou : affiche "coucou 1 fois"
- coucou : i=2
- coucou : affiche "coucou 2 fois"
- coucou : i=3
- coucou : i<3 ? faux, on sort de la fonction

main : retour dans le main, i est toujours égal à 3

main : i<3 ? faux on quitte le programme

Pourquoi c'est dangereux !

Conclusion :

- C'est une très mauvaise idée de modifier une variable globale.
- Mais alors quoi ? les variables globales restent constantes ?
- Pire que ça ! On oublie les variables globales et on définit des constantes.

Plan

Définir une constante

Pour définir une constante, il est possible de définir une macro en tête du programme.

Syntaxe

```
#define NOM valeur
```

Exemple

```
#define MAX 1000
#define PI 3.1415
```

Remarque : par convention, les macros sont notées en majuscules.

Exemple

```c
#include <stdio.h>
#define MAX 10
void produit(int a){
    int i;
    for (i=1;i<=MAX;i++)
        printf("%3d ",a*i);
}
int main(){
    int i;
    for (i=1;i<=MAX; i++){
        produit(i);
        printf("\n");
    }
    return 0;
}
```

Exemple

On obtient alors :

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Définir une constante

Pour définir une constante, il est aussi possible de définir des variables globales constantes.

Syntaxe

```
const type nom=valeur;
```

Exemple

```
const int max=10;
const double pi=3.1415;
```

Exemple

```
1  #include <stdio.h>
2  const int max=10;
3  void produit(int a){
4       int i;
5       for (i=1;i<=max;i++)
6            printf("%3d ",a*i);
7  }
8  int main(){
9       int i;
10      for (i=1;i<=max; i++){
11           produit(i);
12           printf("\n");
13      }
14      return 0;
15 }
```

Exemple

On obtient alors :

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Peut-on modifier une constante ?

```
1 const int max=10;
2 int main(){
3     max=11;
4     return 0;
5 }
```

Le compilateur vous affiche l'erreur suivante :

exemple.c :3 : error : assignment of read-only variable 'max'

Différence entre MACRO et variables constantes

Une variable constante est une vraie variable, avec un espace mémoire réservé. Le compilateur vous interdit juste de la modifier.

Une macro n'est pas une variable. Il n'y a pas d'espace mémoire allouée pour. Au moment de la compilation, le compilateur commence par remplacer toutes les occurences des MACROs par leurs valeurs, puis compile le nouveau fichier obtenu.

Plan

Portée des fonctions

Question

Quand on déclare et définit une fonction, à quel endroit peut-on l'utiliser ?

RÃ©ponse

Partout après sa déclaration !

Autrement dit, si une fonction `f1` appelle une fonction `f2`, la déclaration de `f2` doit se situer avant la fonction `f1`.

Exemple

```c
int f1(){
    printf("je suis f1\n");
}
int f2(){
    printf("je suis f2 et je peux me servir de f1\n");
    f1();
}
int main(){
    f1();// je peux me servir de f1
    f2();// je peux me servir de f2
}
```

Il faut donc bien penser votre programme ainsi qu'à l'ordre dans lequel vous décrivez vos fonctions.

Problème

On souhaite définir ces deux fonctions :

$$pair(n) = \begin{cases} vrai & \text{si } n = 0 \\ impair(n-1) & \text{sinon} \end{cases}$$

et

$$impair(n) = \begin{cases} faux & \text{si } n = 0 \\ pair(n-1) & \text{sinon} \end{cases}$$

Quelle est la fonction qui se place avant l'autre ?

Comment faire ?

RÃ©ponse

On peut séparer la déclaration de la définition d'une fonction.

Dans le corps de la définition d'une fonction, je peux utiliser n'importe quelle fonction qui aura été déclarée précédemment (mais pas forcément définie).

Remarque

Rappelez-vous, on avait dit qu'il était possible d'utiliser une fonction n'importe où après sa *déclaration*...

Comment faire ?

La déclaration de la fonction consiste à donner la signature de la fonction, c'est-à-dire son type de retour et le type de chacun de ses paramètres.

Syntaxe

```
type nom(type,type,...,type);
```

Exemple

```
int pair(int);
int impair(int);
```

Exemple

```
1  int pair(int);
2  int impair(int);
3
4  int pair(int n){
5      if (n==0) return 1;
6      else return impair(n-1);
7  }
8
9  int impair(int n){
10     if (n==0) return 0;
11     else return pair(n-1);
12 }
```

Structure d'un fichier C

```
1  /* Début déclarations des fonctions */
2  int pair(int);
3  ...
4  ...
5  /* Fin déclarations des fonctions */
6
7  /* Début définitions des fonctions */
8  int pair(int n){
9      if (n==0) return 1;
10     else return impair(n-1);
11 }
12 ...
13 ...
14 /* Fin définitions des fonctions */
15
16 int main(){
17     ....
18     return 0;
19 }
```

Programmation modulaire

Remarque

Il est possible de scinder son programme (toutes les fonctions qui le composent) en plusieurs fichiers. On parle alors de programmation modulaire.

Dans les faits, il s'agit de mettre dans un fichier les signatures des fonctions et dans un autre les définitions. Tout programme qui voudra utiliser une des ces fonctions devra seulement inclure le fichier des signatures.

Fichier d'en-tête

Definition

Un fichier d'en-tête (ex : `fonction.h`) contient des déclarations de fonctions.

Exemple

```
1 /* Début déclarations des fonctions */
2 int pair(int);
3 int impair(int);
4 ...
5 /* Fin déclarations des fonctions */
```

Fichier de définition

Definition

Un fichier de définition (ex : `fonction.c`) contient les définitions des fonctions déclarées dans `fonction.h`.

Remarque

Les deux noms de fichiers doivent être les mêmes (`toto.h` et `toto.c`).

Mise en oeuvre

Dans le fichier de définition, il faut inclure le fichier de déclarations.

Syntaxe

```
#include "fonction.h"
```

```
1 #include "fonction.h"
2 /* Début définitions des fonctions */
3 int pair(int n){
4     if (n==0) return 1;
5     else return impair(n-1);
6 }
7 ...
8 ...
9 /* Fin définitions des fonctions */
```

Problème inclusion multiple

Si fichier d'en-tête est inclus plusieurs fois, celà va provoquer une erreur (redéclaration de fonction). Pour éviter celà, il existe des directives pour n'inclure dans un projet qu'une seule fois un ensemble de déclarations.

```
1 #ifndef FONCTIONS_H
2 #define FONCTIONS_H
3
4 /* Début déclarations des fonctions */
5 int pair(int);
6 int impair(int);
7 ...
8 /* Fin déclarations des fonctions */
9 #endif
```

Plan

Fonction récursive

Normalement, tout ceci est un simple rappel de choses vues au premier semestre.

Definition

Une fonction est récursive si elle s'appelle elle-même.

Etude d'un cas simple

Considérons le problème suivant :

- On veut calculer la somme des carrés des entiers compris dans un intervalle (entre m et n).
- Par exemple, `SommeCarres(5,10)`$=5^2+6^2+7^2+8^2+9^2+10^2$.

Première solution

```
1  int SommeCarres(int m, int n){
2      int i,som=0;
3      for (i=m;i<=n; i++)
4          som=som+i*i;
5      return som;
6  }
7  int main(){
8      int m=5, n=10;
9      int sc=SommeCarres(5,10);
10     printf("Carres entre 5 et 10 : %d\n", sc);
11     return 0;
12 }
```

Vision récursive

- Décomposition du problème en sous-problèmes,
- Résolution des sous-problèmes,
- Combinaisons des solutions aux sous-problèmes $=>$ solution du problème

Notre problème

- Si il y a plus d'un nombre dans [m..n], on ajoute le carré de m à la somme des carrés de [m+1..n]
- Si il n'y a qu'un nombre (m=n), le résultat est le carré de m

Mathématiquement,

$$SommeCarres(m,n) = \begin{cases} m*m + SommeCarres(m+1,n) & \text{si } m \neq n \\ m*m & \text{sinon} \end{cases}$$

Solution récursive

```
1  int SommeCarres(int m, int n){
2      if (m==n)
3          return m*m;
4      else
5          return (m*m+SommeCarres(m+1,n));
6  }
7  int main(){
8      int m=5, n=10;
9      int sc=SommeCarres(5,10);
10     printf("Carres entre 5 et 10 : %d\n", sc);
11     return 0;
12 }
```

Trace des appels

$$
\begin{aligned}
\mathit{SommeCarres}(5,10) &= 25 + \mathit{SommeCarres}(6,10) \\
&= 25 + (36 + \mathit{SommeCarres}(7,10)) \\
&= 25 + (36 + (49 + \mathit{SommeCarres}(8,10))) \\
&= 25 + (36 + (49 + (64 + \mathit{SommeCarres}(9,10)))) \\
&= 25 + (36 + (49 + (64 + (81 + \mathit{SommeCarres}(10,10 \\
&= 25 + (36 + (49 + (64 + (81 + 100)))) \\
&= 25 + (36 + (49 + (64 + 181))) \\
&= 25 + (36 + (49 + 245)) \\
&= 25 + (36 + 294) \\
&= 25 + 330 \\
&= 355
\end{aligned}
$$

Principe de construction

- Des instructions résolvant les cas particuliers,
- Des instructions décomposant le problème en sous-problème,
- Des appels récursifs résolvant les sous-problèmes,
- Des instructions pour résoudre le problème à partir des solutions des sous-problèmes.

Un autre problème

La suite de Fibonacci :

$$\begin{cases} F(0) = 1 \\ F(1) = 1 \\ F(n+2) = F(n+1) + F(n) \end{cases}$$

Très facile à mettre en oeuvre !

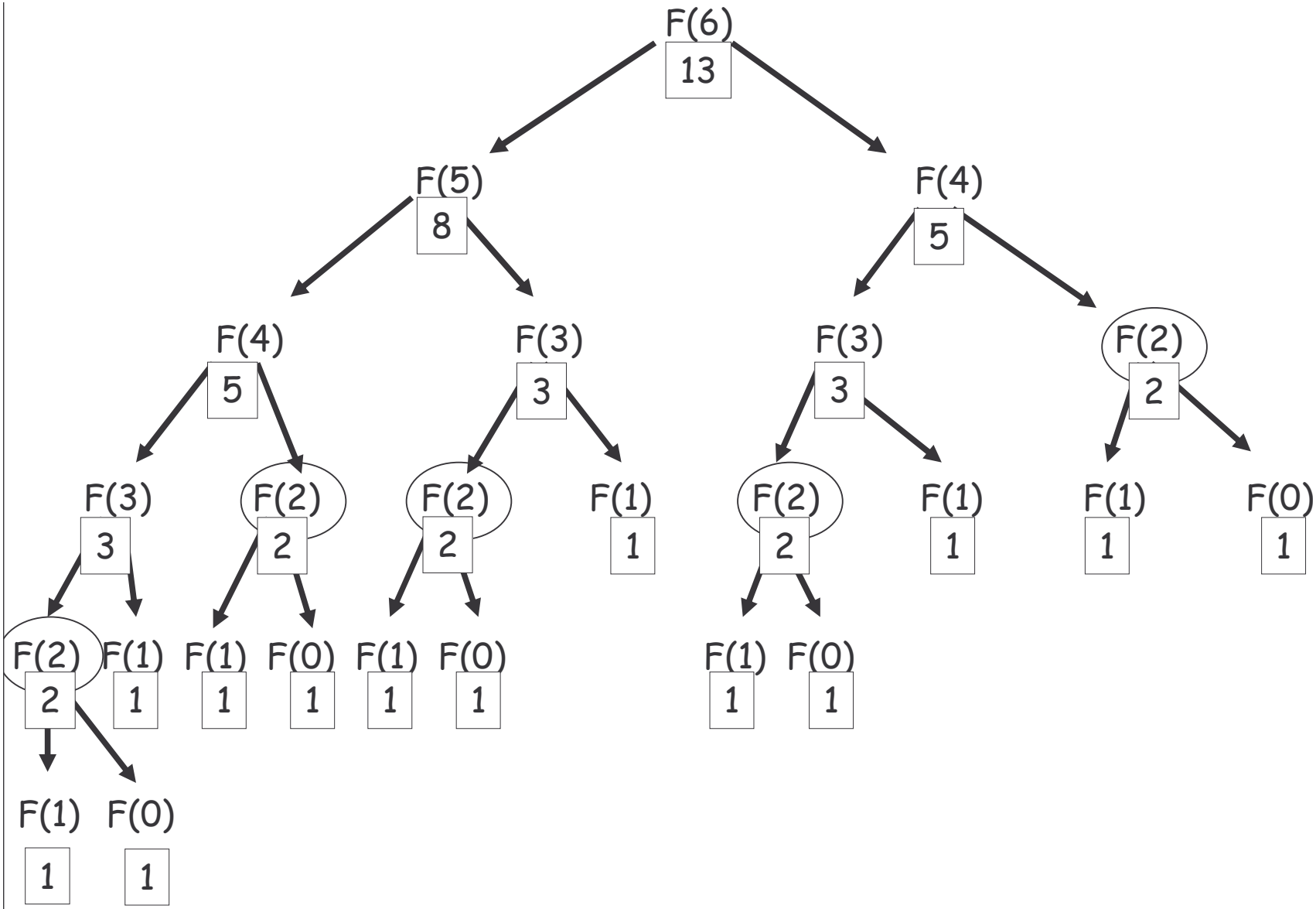
Solution récursive

La version récursive est très proche de l'écriture mathématique :

```
1 int Fibo(int n){
2     if (n==0)||(n==1)
3         return 1;
4     else
5         return Fibo(n-1)+Fibo(n-2);
6 }
```

Exécution

Certains termes sont calculés plusieurs fois :



Solution itérative

La solution itérative est moins intuitive :

```
1  int Fibo(int n){
2      int a,b,F;
3
4      F=a=b=1;
5
6      while (n>1){
7          F=a+b;
8          a=b;
9          b=F;
10         n--;
11     }
12     return F;
13 }
```

F	a	b	n
1	1	1	6

Solution itérative

La solution itérative est moins intuitive :

```
1  int Fibo(int n){
2      int a,b,F;
3
4      F=a=b=1;
5
6      while (n>1){
7          F=a+b;
8          a=b;
9          b=F;
10         n--;
11     }
12     return F;
13 }
```

F	a	b	n
1	1	1	6
2	1	2	5

Solution itérative

La solution itérative est moins intuitive :

```
1  int Fibo(int n){
2      int a,b,F;
3
4      F=a=b=1;
5
6      while (n>1){
7          F=a+b;
8          a=b;
9          b=F;
10         n--;
11     }
12     return F;
13 }
```

F	a	b	n
1	1	1	6
2	1	2	5
3	2	3	4

Solution itérative

La solution itérative est moins intuitive :

```
1  int Fibo(int n){
2      int a,b,F;
3
4      F=a=b=1;
5
6      while (n>1){
7          F=a+b;
8          a=b;
9          b=F;
10         n--;
11     }
12     return F;
13 }
```

F	a	b	n
1	1	1	6
2	1	2	5
3	2	3	4
5	3	5	3

Solution itérative

La solution itérative est moins intuitive :

```
1  int Fibo(int n){
2      int a,b,F;
3
4      F=a=b=1;
5
6      while (n>1){
7          F=a+b;
8          a=b;
9          b=F;
10         n--;
11     }
12     return F;
13 }
```

F	a	b	n
1	1	1	6
2	1	2	5
3	2	3	4
5	3	5	3
8	5	8	2

Solution itérative

La solution itérative est moins intuitive :

```
1  int Fibo(int n){
2      int a,b,F;
3
4      F=a=b=1;
5
6      while (n>1){
7          F=a+b;
8          a=b;
9          b=F;
10         n--;
11     }
12     return F;
13 }
```

F	a	b	n
1	1	1	6
2	1	2	5
3	2	3	4
5	3	5	3
8	5	8	2
13	8	13	1

Récursivité ! ! !

Attention

La récursivité n'est pas toujours efficace, même si elle est plus facile à exprimer.