Plan

Plan

Programme minimal

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

```
1 int main(){
2     // declaration de variables
3     // instructions du programme
4 return 0;
5 }
```

Le C : langage typé et impératif

Comme dans un langage algorithmique :

- On manipule des **expressions** et des **variables** qui ont un type (ex : entier, flottant)
- Les instructions sont des expressions particulières modifiant l'environnement :
  - des déclarations
  - des affectations
  - des appels de fonction
  - des entrées/sorties
  - des structures de contrôle pour réaliser des instructions complexes

Plan

Les types de base

- entiers : des parties finies `char`, `short`, `int`, `long` des entiers naturels (`unsigned`) ou relatifs (`signed`)
- réels : des approximations bornées : `float`, `double`
- booléens : `_Bool`

Des conversions implicites d'un type à l'autre.

Le codage binaire des entiers (naturels)

Les entiers naturels sont exprimés sous format binaire

$$231 = 128 + 64 + 32 + 4 + 2 + 1 = 2^7 + 2^6 + 2^5 + 2^2 + 2^1 + 2^0$$

1	1	1	0	0	1	1	1

Le nombre de bits (case) fixe l'ensemble d'entiers représentables.

Sur 8 bits on représente les entiers positifs dans $[0, 255]$

- le plus petit : 0

0	0	0	0	0	0	0	0

- le plus grand : $255 = 2^8 - 1$

1	1	1	1	1	1	1	1

Le codage binaire des entiers relatifs

- Le bit de poids fort indique le signe :
  bit à 0 $\Rightarrow$ entier positif ou nul, bit à 1 $\Rightarrow$ entier négatif.

s	0	0	1	1	0	0	1

- Les entiers positifs sont codés comme les entiers naturels en utilisant les n-1 bits restants (le premier est à 0).

0	0	0	1	1	0	0	1

code l'entier **25**

- Les entiers négatifs sont représentés en complément à deux.

1	0	0	1	1	0	0	1

code -**103** (et non -25)

- Avec n bits on représente les entiers relatifs $[-2^{n-1}, 2^{n-1} - 1]$

Le complément à 2

La représentation en complément à 2 facilite les opérations arithmétiques (l'addition reste classique).

Représentation par complément à 2 d'un nombre négatif -v

1. représenter comme un positif son opposé, soit v
2. calculer le complément à 1 (inverser les bits)
3. ajouter 1

Exemple : Pour représenter -103 sur 8 bits :

1	on représente 103	0	1	1	0	0	1	1	1
2	on inverse les bits	1	0	0	1	1	0	0	0
3	on ajoute 1	1	0	0	1	1	0	0	1

Limitations du codage des entiers relatifs

On ne code qu'une partie finie des entiers relatifs $[MIN, MAX]$ que l'on rend "circulaire" !

(MIN et MAX sont les bornes dépendandantes du nombre de bits choisi)

$\hookrightarrow$ Le successeur de MAX est donc MIN !

Exemple sur 8 bits :

1	127	0	1	1	1	1	1	1	1
2	+ 1	0	0	0	0	0	0	0	1
3	= -128	1	0	0	0	0	0	0	0

Remarque

L'entier négatif $-v$ est représenté sur n bits par le codage classique du positif $2^n - v$ (si le nombre n de bits est suffisant !)

0 n'est codé que par sa "valeur positive"

0	0	0	0	0	0	0	0

L'opposé de MIN est MIN sur l'intervalle d'entiers représentables.

Les types entier en C

- Plusieurs variantes qui diffèrent par leur taille en octets : `char`, `short int`, `int`, `long int`, `long long int`.
  ↪ un octet (*byte* en anglais) regroupe 8 bits (*Binary Digits*).
- Cependant, le standard C ne spécifie pas le nombre exact d'octets alloué à chaque type entier.
  ↪ Généralement, sur une architecture classique 64 bits, on a :

type en C (nom court)	**taille** (nbr octets)
`char`	1
`short`	2
`int`	4
`long`	8
`long long`	8

Les intervalles entier représentables en C

- entiers relatifs : le type entier par défaut, codage en complément à 2.
  ↪ On peut expliciter ce choix en ajoutant le préfixe **signed**.
- entiers naturels : on préfixe le type par le mot-clef **unsigned**, codage binaire classique

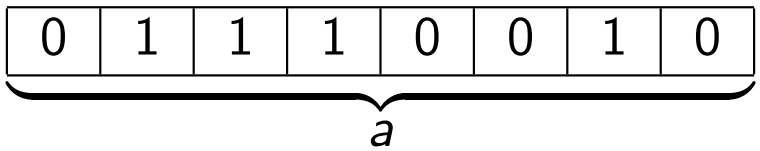
taille	naturels	valeurs	relatifs	valeurs
1	`unsigned char`	$[0, 2^8 - 1]$	`char`	$[-2^7, 2^7 - 1]$
2	`unsigned short`	$[0, 2^{16} - 1]$	`short`	$[-2^{15}, 2^{15} - 1]$
4	`unsigned int`	$[0, 2^{32} - 1]$	`int`	$[-2^{31}, 2^{31} - 1]$
8	`unsigned long`	$[0, 2^{64} - 1]$	`long`	$[-2^{63}, 2^{63} - 1]$

Remarque

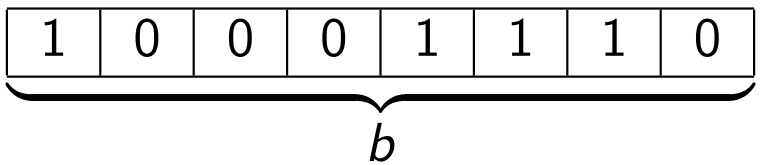
Les constantes `INT_MIN`, `INT_MAX`, `UINT_MIN`, `CHAR_MIN`... permettent de connaitre l'intervalle réel des types **entier** de son architecture.

Exemples

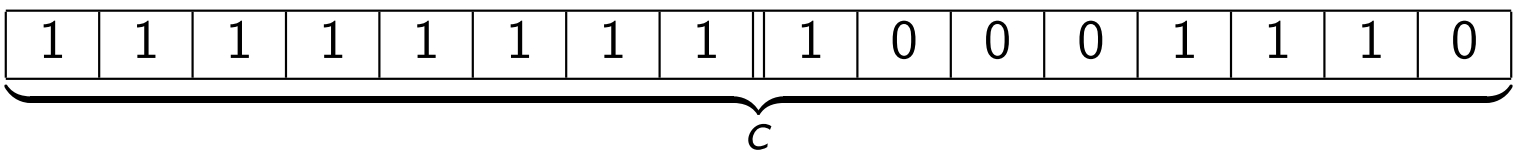
```
char a = 114;
```



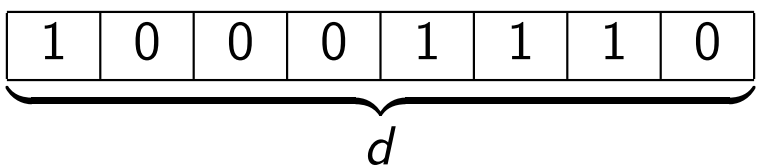

```
char b = -114;
```




```
short int c = -114;
```




```
unsigned char d = 142;
```



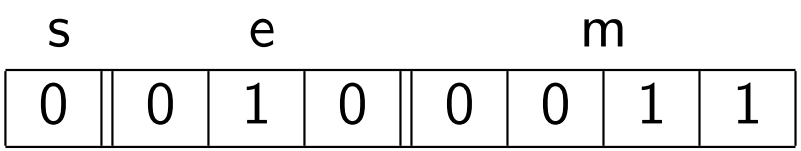
Les types de données réels

Les réels ne sont pas représentables en codage binaire. On représente un réel x par une approximation rationnelle particulière :

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

exemple : $0.75 = (-1)^0 \times \frac{3}{2^2}$

Le codage de x correspond au codage binaire de s,m et e.



Norme IEEE754 :

- `float` (32 bits) : 1 bit pour s, 23 bits pour m, 8 bits pour e
- `double` (64 bits) : 1 bit pour s, 52 bits pour m, 11 bits pour e

Les booléens

Il n'y a pas de base un type booléen en C. On utilise deux valeurs de type **entier** pour représenter les booléens :

- 0 représente la valeur **faux**
- 1 représente la valeur **vrai**

Les règles implicites de conversion considèrent **toute valeur non codée par une suite de bits à 0** comme le booléen **vrai**.

Remarque

Depuis la norme 99, un type `_Bool` a été introduit codé sur 1 octet et n'ayant que deux valeurs 0 et 1.

Plan

Les constantes

- Les constantes permettent de désigner des valeurs dans un programme C
- Ces valeurs sont automatiquement typées en fonction de la forme syntaxique de cette constante
- on distingue :
  - les constantes entières : de type int, unsigned int, long ou unsigned long
  - constantes réelles : de type float ou double
  - constantes caractères : de type int (et non char !)
  - constantes chaînes de caractères : de type tableau de caractères
- Il n'y a pas de constantes de type char et short $\Rightarrow$ des conversions permettront de valuer les variables de ces types.

Les constantes entières

- Des valeurs positives entières données en représentation décimale, octale ou hexadécimale
  - 169, 0251 ou 0xA9 désignent la représentation machine `int` sur 4 octets [00000000|00000000|00000000|10101001]
- la valeur de la constante détermine le type de donnée de l'entier cible
  - de 0 à $2^{31} - 1 \rightarrow$ `int`
  - de 2^{31} à $2^{63} - 1 \rightarrow$ `long`
  - de 2^{63} à $2^{64} - 1 \rightarrow$ `unsigned long`
- on peut forcer une représentation non signée et/ou long avec les suffixes U et L
  - 169UL désigne le `unsigned long` sur 8 octets dont les 56 premiers bits sont à 0 et les 8 derniers sont 10101001

Remarque

Les valeurs négatives des entiers sont obtenues par l'opérateur unaire -

Les constantes réelles (flottantes)

- Des valeurs réelles positives données en représentation scientifique
  - ex : 2.34e4 qui correspond à 2.34×10^4 (le rationnel 23400)
  - ex : 2. qui correspond à 2.0×10^0 (le rationnel 2)
  - ex : 2e-2 qui correspond à 2.0×10^{-2} (le rationnel 0,02)
  - le . ou le e est obligatoire !
- les constantes réelles sont représentées par des double.
- on peut forcer le type `float` en suffixant par F
  - ex : 2.34e4F

Les constantes caractères

Les constantes caractères sont données entre deux quotes '...'

- Elles peuvent contenir un :
  - caractère alphanumérique :
    'a', 'b', ...
    'A', 'B', ...
    '0', '1', ...
    ' ?', ' :', ' ', '<', ...
    mais pas un caractère composé : 'é', 'ç'...
  - un caractère spécial :

    $\backslash n$ $\rightarrow$ retour à la ligne
    $\backslash t$ $\rightarrow$ tabulation
    $\backslash b$ $\rightarrow$ backspace
    $\backslash 0$ $\rightarrow$ fin de chaîne
    ...
- Elles sont représentées par des **valeurs du type `int`** : l'entier correspondant à leur code ASCII.

Codage des caractères

- Les entiers associés aux constantes caractères codent les éléments du jeu de caractères de la machine (souvent ASCII sur 7 bits).

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

les constantes chaînes de caractères

Les constantes chaînes de caractères sont des suites de caractères données entre deux doubles quotes (guillemets) "..."

- Elles contiennent n'importe quel caractère y compris les caractères composés (é, à, ç...) et les caractères spéciaux (\n, \t ...)
- Elles sont représentées par des **séquences de `char`** (octets) dont la longueur dépend du nombre de caractères de la chaîne et du codage des caractères de la machine (ASCII, ASCII étendu, UTF8...)
  - un caractère alphanumérique simple ou un caractère spécial est représenté par un `char`
  - un caractère composé ou spécifique à un alphabet est représenté par plusieurs **char** (selon le codage de caractères utilisé sur la machine)
  - le caractère spécial fin de chaîne \0 est ajouté à la fin
- **"ça va\n"** est représenté en UTF8 par la suite de 8 octets (on donne ici leur valeur décimale) [195|167|97|32|118|97|10 | 0]

  [ç | a | | v | a |\n|\0]

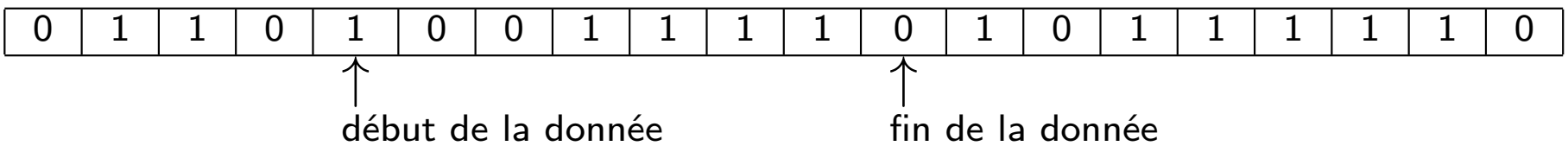
Plan

Stockage des données

Rappel :

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

Les données sont stockées à un emplacement précis de la mémoire



Pour retrouver une donnée, il faut donc :

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

Variable C

Définition

Une variable C est une zone de la 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 au programmeur de manipuler facilement les données stockées en mémoire, mais **l'adresse** de la variable serait suffisante
- le **type** permet au compilateur de *réserver l'espace mémoire* suffisant pour stocker les valeurs.
- la suite des bits dans la zone mémoire définit la **valeur** de cette variable.

[1] *on utilise un identificateur plutôt que l'adresse mais l'on peut récupérer l'adresse d'une variable à partir de son identificateur.*

Les identificateurs

En plus de permettre de nommer les variables, ils permettent de nommer les fonctions et les types définis.

Construction d'un identificateur

Une suite de caractères parmi : les majuscules (A..Z), les minuscules (a..z), les chiffres (0..9), le tiret bas (_)

- le premier caractère ne peut être un chiffre,
- les majuscules et minuscules sont différenciées,
- pas de caractères accentués,
- l'identificateur doit être différent des mots clés du langage : `int, char, if, for...`

Déclaration des variables

Déclaration

Pour utiliser une variable il faut auparavant la déclarer en précisant son type et son nom : `type nom ;`
On peut déclarer simultanément plusieurs variables du même type en séparant les noms par des virgules : `type nom1, nom2, ... ;`

Exemple

```
int a ;
```

a est une variable **entier standard** :

- le compilateur réserve 4 octets en mémoire pour stocker ses valeurs
- le programmeur utilise le nom **a** pour travailler avec cette zone mémoire.

```
short b, c ;
```

b et c sont deux variables **entier court** :

- le compilateur réserve 2×2 octets en mémoire pour stocker leurs valeurs.

Plan

Les expressions

Définition

Une expression atomique est soit une constante, soit un nom de variable.
Des expressions plus complexes peuvent être construites à l'aide d'opérateurs.

- Le C est un langage typé ⇒ Toute expression est typée. Le type dépend de l'opérateur et de ses opérandes.
- Le C est un langage faiblement typé ⇒ De nombreuses conversions implicites de type rendent possible l'utilisation d'opérateurs sur des opérandes n'ayant a priori pas le type requis.

Les opérateurs arithmétiques

les opérateurs classiques binaires

- addition : $+$
- soustraction : $-$
- division : $/$
- multiplication : *
- modulo : %
- opposé (unaire) : $-$

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

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

Opérateurs de comparaisons

Les comparateurs classiques sur les types numériques :

égalité	== (2 signes =)
différent	! =
supérieur	>
inférieur	<
supérieur ou égal	>=
inférieur ou égal 1	<=

Rappel : le langage C ne dispose pas d'un type booléen. Les comparateurs retourne une valeur du type `int` :

- 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 : `!`

Exemple

`!1` est évalué à 0 (faux)
`(1 && 0) || 1` est évalué à 1 (vrai)
`(2.5 > 3.5) && (1 < 3)` est évalué à 0
`1 || (1 >= 3)` est évalué à 1

Opérateur d'affectation

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

- `var = exp`
  - `exp` : expression de **même type** que `var` (mais mécanisme de conversion implicite de type)
  - `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 de 5

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

Sizeof

L'opérateur unaire `sizeof` permet de connaitre la taille mémoire (en nombre d'octets) de son opérande qui peut être :

- un type de données : on renvoie le nombre d'octets occupés par une valeur de ce type.
- une expression : on renvoie le nombre d'octets occupés par une valeur du type de l'expression

Exemples :

- `sizeof(2)` vaut 4 (la constante 2 est de type `int`)
- soit a déclaré par : `char a ;`
  `sizeof(a)` vaut 1 (le type char réserve 1 octet)
- `sizeof(a+a)` vaut 4 (pour appliquer l'addition une conversion de la valeur de a en une valeur `int` a été faite)

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

Soit `a` une variable déclarée, `&a` renvoie la valeur de l'adresse de `a` :
un entier codé sur 8 octets quelque soit le type de `a`.
Sa valeur est généralement donnée en hexadécimale.

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 *(i.e. un type pointeur)*

Autres opérateurs

- incrémentation/décrémentation d'une variable entière a :
  - $a++$ incrémente la valeur de a par 1 *(i.e. a=a+1)*
  - $a--$ décrémente la valeur de a par 1 *(i.e. a=a-1)*
- affectations élargies : `+=`, `-=`, `*=`, `/=`
  - `a+=3` correspond à l'expression `a=a+3`
- l'opérateur conditionnel : `b ?` e_1 `:` e_2
  - selon la valeur de l'expression booléenne b, l'opérateur calcule la valeur de e_1 si b vrai, ou la valeur de e_2 si faux (e_1 et e_2 doivent être de même type). Ex : `0?-5:5` vaut 5
- les opérateurs bit à bit : &, |, ^ , ~ , <<, >>
- et beaucoup d'autres ...

Priorité des opérateurs arithmétiques

Les règles classiques de priorités mathématiques sont conservées :

- $a + b \times c$ sera interprété $a + (b \times c)$

Les parenthèses permettent d'imposer que certaines opérations soient évaluées avant d'autres :

- $(a + b) \times c$ impose de calculer l'addition avant la multiplication

Priorités des opérateurs : récapitulatif

Opération associative ($\star$) :

- à droite : $a \star b \star c \Rightarrow a \star (b \star c)$
- à gauche : $a \star b \star c \Rightarrow (a \star b) \star c$

Catégorie	**Opérateurs**	**Associativité**
Unaire	$+$, $-$, $++$, $--$, !, $*$, &, sizeof, cast	Droite
Binaire	$*$, $/$, %	Gauche
Binaire	$+$, $-$	Gauche
Comparaison	$<$, $<=$, $>$, $>=$	Gauche
Comparaison	$==$, $!=$	Gauche
Logique	&&	Gauche
Logique	$\|\|$	Gauche
Affectation	$=$, $+=$, $-=$, $*=$, $/=$, $\%=$	Droite

Le tableau est classé par ordre de priorité décroissante.

Priorités des opérateurs : exemples

Expression	**Expression parenthésée**	**Valeur**
2+3*7	2+(3*7)	23
15*3%7/2	((15*3)%7)/2	1
x=y=2	x=(y=2)	x :2, y :2
1<4==7<5 !=9<4	((1<4)==(7<5)) !=(9<4)	0 (faux)

Plan

Conversions de types

On appelle conversion de type le fait de transférer une valeur d'un type dans une valeur d'un autre type.

- Les conversions de type peuvent se faire avec ou sans perte d'information.
  - La perte d'information aura lieu si le codage de la valeur dans le type cible n'est pas possible (car sa taille est trop petite)
- De plus une conversion peut être :
  - explicite : voulue par le programmeur
  - implicite : décidée par le compilateur
    - pour réaliser un calcul
    - pour affecter une variable

Conversions implicites

Calculs réalisés par le processeur

Les calculs ne se font que sur les types : int, long, unsigned int, unsigned long, float et double.

- les valeurs des types `char`, `_Bool`, `short`, `unsigned char`, ou `unsigned short` sont donc implicitement converties en un `int` avant de faire un calcul.

C'est une conversion sans perte.

Attention

Pas de perte, mais des dépassements de capacité peuvent avoir lieu lors des calculs engendrant des résultats non attendus.

- Ex. 2147483647 + 1 vaut -2147483648

Conversions implicites

Résolution d'expression de types mixtes

Le langage étant faiblement typé, on peut avoir à appliquer des opérateurs sur des valeurs n'ayant pas les types attendus.

- Ex : `(1+2.5) && 1`

Des conversions implicites sont faites vers le type le plus riche :

`int -> float -> double` `int -> long -> double`

`signed -> unsigned`

Stockage dans une variable

Lors d'une affectation, la valeur de l'expression doit être rangée dans la variable $\Rightarrow$ conversion si types différents

- `int a = 3.5` sera effectué comme `int a=3`

Cette conversion peut engendrer des pertes.

Conversions explicites

On peut forcer le changement de type en effectuant un `cast`.

Conversion de type par cast

L'expression : `(type) exp`

- calcule la valeur `v` de `exp` dans le type `t_exp`
- puis convertit `v` du type `t_exp` dans le type `type`.

Exemple

```
1 int a=3,b=4;
2 double c= a/b;                    // c=0.0
3 double d= (double)a/(double) b    // d=0.75
```