Comportements machines dépendants

- Arrondi de /: machine dépendant pour opérande(s) négative(s)
  par défaut sinon
- Signe de %: machine dépendant pour opérande(s) négative(s)
  positif sinon
- Division par 0 (/ et %): comportement machine dépendant
- Valeurs de e << e' et e >> e': machine dépendant si e' est supérieur au nombre de bits de e, ou si e'<0
- Valeurs de e << e' et e >> e': machine dépendant si e<0
- Dépassements de capacité

Int

Int:

Nombre de bits de la représentation = taille du mot machine

Le type int est le type naturel des entiers pour la machine
=> MACHINE DEPENDANT !

Il est signé par défaut

Tailles typiques:

i8088	: 16 bits
PDP-7	: 18 bits
intel pentium	: 32 bits
Sun UltraSparc III	: 64 bits

Bornes (<limits.h>)
INT_MIN, INT_MAX

Vals. min. ANSI
-32767..+32767

Des int d'autres tailles

Short int Long int

Comme les int, signés par défaut

Tailles: machines dépendantes.

Seules certitudes:
- 2<=sizeof(short)
- 4<=sizeof(long)
- sizeof(short)<=sizeof(int)<=sizeof(long)

Bornes (<limits.h>)
SHRT_MIN, SHRT_MAX
LONG_MIN, LONG_MAX

Vals min. ANSI
-32767..+32767
-2147483647L..+2147483647L

Les entiers non signés

Type d'entier *t*: unsigned t

Arithmétique modulo 2^n (n nombre de bits codant l'entier)

Jamais de dépassement de capacité

Les valeurs communes à t et unsigned t sont codées de la même manière

Bornes supérieures (<limits.h>)
U<type_desc>_MAX
Exemple: UCHAR_MAX

Les entiers signés

Type d'entier *t*: signed t

Les entiers sont signés par défaut, sauf ceux du type char (machine dépendant)

Exemple: signed int <=> int

Bornes (<limits.h>)
SCHAR_MIN
SCHAR_MAX
Toute borne non U

Les réels

- Float: simple précision
- Double: double précision
- Long double: précision étendue

précision(float)<=précision(double)<=précision(long double)

Pas de non signés

Bornes et autres informations: <float.h>
(FLT+DBL)_(MIN+MAX): bornes
(FLT+DBL)_EPSILON: plus petit x tel que 1+x=x
(FLT+DBL)_DIG: précision en chiffres décimaux
...

Vals. min. ANSI: FLT_MIN: 1e-37, FLT_MAX: 1e+37, FLT_DIG: 6, FLT_EPSILON: 1e-5
DBL_MIN: 1e-37, BDL_MAX: 1e+37, DBL_DIG: 10, DBL_EPSILON: 1e-9

Promotion entière

(signed+unsigned)(char+short int) $\xrightarrow{\text{Promotion entière}}$ int+unsigned int

Si int contient le type d'origine, alors conversion en int, sinon en unsigned int

La promotion entière est donc machine dépendante

Conversion entière

entier x:t ⟶ entier x':t' non signé

x' = x [Ut'_MAX+1]

entier x:t ⟶ entier x':t' signé

x' = x si possible dans t', machine dépendant sinon

Exemples

3:unsigned int ⟶ 3:signed char

SMAX_CHAR+1: unsigned int ⟶ machine dépendante:signed char

Conversions entières

Entier -> entier non signé de n bits x -> x [2^n]

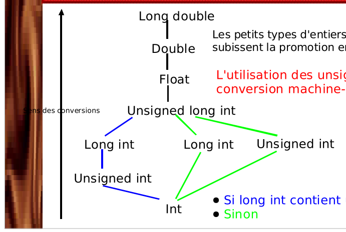
Exemples:

unsigned char c=3;	unsigned int i=c;	3
signed char c = -3;	unsigned int i=c;	UINT_MAX+1-3
char c = 129;	unsigned int i=c;	Machine dépendant
int i=4;	unsigned char c=i;	4
int i=-4;	unsigned char c=i;	UCHAR_MAX+1-4
int i=-31237;	unsigned char c=i;	-31237+(UCHAR_MAX+1) -(-31237/UCHAR_MAX)
int i=31237;	unsigned char c=i;	31237- (31237/UCHAR_MAX)

Entier -> entier signé

Si l'entier de départ est représentable dans le type de destination, sa valeur est inchangée. Conversion machine dépendante sinon

Conversions arithmétiques



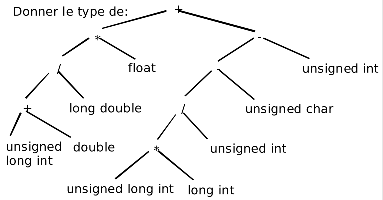
Les petits types d'entiers (char, short) subissent la promotion entière

L'utilisation des unsigned rend la conversion machine-dépendante

- Si long int contient unsigned int
- Sinon

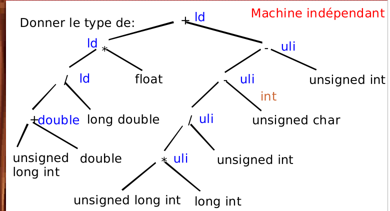
Exemple

Donner le type de:



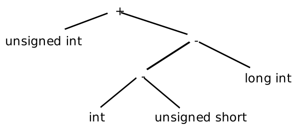
Correction

Donner le type de:



Exemple

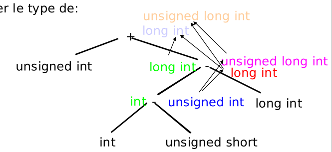
Donner le type de:



Que dire du résultat si on avait mis des valeurs dans l'arbre ?

Correction

Donner le type de:



Que dire du résultat si on avait mis des valeurs dans l'arbre ?

Conversions entiers <-> réels

Réel *r::t* ⟶ Entier *e::t'*

Si la valeur tronquée *r'* de *r* est représentable dans *t'*, *e*=*r'*
Machine dépendant sinon

Entier *e::t* ⟶ Réel *r::t'*

Si *e* est représentable exactement dans *t'*, *r*=*e*
Si *e* est dans le domaine de t' sans y être représentable exactement, *r*=valeur la plus proche de *e* représentable dans *t'* (machine dépendant)
Si *e* n'est pas dans le domaine de *t'*, machine dépendant

L'opérateur de conversion sur les valeurs arithmétiques

Syntaxe: (*type*)*expression*

Sémantique: après l'évaluation de expression, sa valeur est convertie dans le type *type*

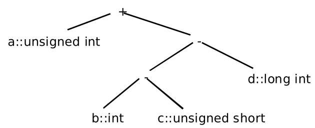
Permet de convertir **explicitement** une valeur d'un type donné vers un autre type.

A utiliser pour éviter les surprises des promotions et conversions automatiques faites par le compilateur.

A utiliser aussi pour désambiguer

L'opérateur de conversion sur les valeurs arithmétiques: exemple

On a l'assurance que les valeurs employées sont petites [SCHAR_MIN,SCHAR_MAX]



Plutôt que d'écrire a+((b-c)-d)   on écrit ((int)a)+((b-(int)c)-d)

Les constantes entières

Trois bases de représentation:

- décimale (10): normale

-128 -56 -10 -9 0 15 16 +17 +300

- octale (8): [+-<rien>]0...

-0200 -070 -012 -011 00 017 020 +021 +0454

- hexadécimale (16): [+-<rien>]0[xX]...

-0xff -0X38 -0x10 -0X9 0x0 0XF 0x10 +0X11 +0x12c

Suffixes:

- long int: [Ll] (-200L, -1L, 200L)
- unsigned int: [Uu] (-200U, -1U, 200u)
- unsigned long int: [Uu]L (200UL,-1UL, 200uL)

Typage des constantes entières

Premier type valable rencontré dans l'ordre de calcul et pouvant contenir la constante

Ordre de calcul →

	int	unsigned int	long int	unsigned long int
Sans suffixe, décimal				
Sans suffixe,autre				
Suffixe [Uu]				
Suffixe [Ll]				
Suffixe [Uu]L				

Exemple: sur une machine 16 bits, typer $(32768)_{10}$ et $(8000)_{16}$

Les caractères constants

Trois formes:

- Le caractère entre quotes: 'a'
- Le code du caractère (en base 10,8,16) précédé de \, le tout entre quotes: '\0xff'
- Une séquence d'échappement entre quotes:

fin de ligne	NL (LF)	\n
tabulation horizontale	HT	\t
tabulation verticale	VT	\v
retour en arrière	BS	\b
retour chariot	CR	\r
saut de page	FF	\f
signal sonore	BEL	\a
antislash	\	\\
point d'interrogation	?	\?
quote	'	\'
guillemet	"	\"

Les constantes réelles

[+-][<entier>].[<entier>][Ee][+-][<entier>][FfLl]

Certaines parties sont optionnelles, à condition que d'autres soient présentes.

Exemples: -.5E6 vaut -500000, 12E-5F vaut 0.00012

Typage directement d'après le suffixe:
- [Ff]: float
- [Ll]: long double
- rien: double

Du bon usage des entiers: observations

- char et short peuvent induire des promotions entières
  - Prenant du temps
  - éventuellement machines dépendantes
- long int n'est pas nécessairement plus long qu'int
- unsigned induit des conversions arithmétiques éventuellement machines dépendantes
- les opérations de décalage sur les entiers sont machines dépendantes

Du bon usage des entiers: conclusions

- On préférera utiliser le type int pour les entiers
- Les types char et short sont utilisés uniquement pour économiser la mémoire, seulement quand c'est absolument nécessaire.
  Par exemple, pour un énorme tableau de petits entiers.
- Pour faire des opérations de manipulation de bits, on utilisera de préférence unsigned
- Pour stocker des valeurs positives ou nulles, par exemple des tailles, on pourra utiliser unsigned si les opérations arithmétiques dans lesquelles elles interviennent sont toujours simples

Les entiers comme booléens

Pas de booléens en C

Les entiers peuvent être interprétés comme des booléens:
- 0: faux
- Toute valeur différente de 0: vrai

```
while (1)
    printf("boucle infinie");
```

Les opérateurs logiques

En C, la valeur d'une expression logique
- vraie est 1
- fausse est 0

Opérateurs de logique booléenne: ! && ||
- !: négation logique
- &&: ET logique (à ne pas confondre avec &)
- ||: OU logique (à ne pas confondre avec |)

érateurs de comparaison: == != < <= > >=

Attention: ne pas confondre l'opérateur de comparaison == avec l'opérateur d'affectation =

Attention: la valeur entière de !!a peut être différente de a (mais la valeur logique est identique)

L'évaluation des opérateurs logiques || et && est paresseuse !

L'évaluation d'expressions logiques s'effectue de gauche à droite, et s'arrête dès que possible

Permet de s'assurer qu'une ressource est disponible, ou qu'une variable n'a pas une valeur interdite, par exemple, avant de faire une opération dessus.

```
/* Retourne vrai ssi a divise b */
int estDiviseur(unsigned int a, unsigned int b)
{
    return ( a!=0 && !(b%a) );
}
```


Expression

En C, une expression est n'importe quoi ayant une valeur

- Constante
- Valeur d'une variable
- Résultat d'une opération
  - Arithmétique: +, -, +, -, /, *, %
  - Arithmétique avec effet de bord: ++, ++, --, --
  - Bit à bit: &, |, ^, ~, <<, >>
  - Logique: &&, ||, !
  - Affectation: = et autres
  - Taille d'emplacement mémoire: sizeof
- Appel de fonction: ()
- Accès à un élément de tableau: []
- Accès à un élément de structure: .
- Déréférencement: *
- Accès à un élément de structure pointée: ->
- Transtypage: (type)
- Prise d'adresse: &
- Séquençage d'expression: ,
- Expression conditionnelle: ?:

Expression conditionnelle

Syntaxe: *expression booléenne* ? *expression1* : *expression2*

Sémantique: Si *expression booléenne* est vraie, alors la valeur de l'expression conditionnelle est celle de *expression1*, sinon la valeur de l'expression conditionnelle est celle de *expression2*.

Soit *expression1*, soit *expression2* est évaluée (pas les deux), en fonction de la valeur de *expression booléenne*.

expression1 et *expression2* doivent être de types compatibles.

Exemple:

```
unsigned long int factorielle(unsigned int n)
{ return n==0 ? 1 : n*factorielle(n-1); }
```


Sizeof

Syntaxe: size_t sizeof(*expression*)
size_t sizeof(*type*)

Sémantique: Calculer la taille (en cellules) de son argument.
Dans sa première forme, calcule le type de *expression* (**sans l'évaluer**), puis la taille en cellules des objets du type obtenu.
Dans sa seconde forme, calcule la taille en cellules des objets du type *type* (ne doit pas être un type de fonction).

Exemples: *sizeof(char)* vaut 1, *sizeof(5)* vaut 4 sur un pentium

size_t: entier non signé. Le type de l'entier est machine dépendant.
(unsigned long int, unsigned int, unsigned short ou unsigned char)
Type non natif mais prédéfini (stddef.h)

Attention: malgré sa syntaxe, *sizeof* n'est pas une fonction !

Affectation

Syntaxe: a = e

expression désignant un espace mémoire (l-value) — expression

Sémantique: évaluer *e*, et mettre sa valeur dans *a*. *a* et *e* doivent être de types compatibles. Une conversion vers le type de *a* est réalisée quand le type de *a* et celui de *e* ne sont pas identiques. La valeur et le type de l'expression *a=e* sont ceux de *a* après affectation.

Syntaxes raccourcies pour des affectations: a += e équivalent à a=a+(e)

Autres syntaxes raccourcies: -=, *=, /=, %=, <<=, >>=, &=, ^=, |=

ATTENTION: la localisation de l'espace mémoire peut être faite AVANT l'évaluation de l'expression

Incrémentation / décrémentation préfixée / postfixée

Syntaxe: ++a --a a++ a--
a est une l-value

Sémantique: ++*a* est une expression valant *a*+1, et ayant pour effet de bord d'augmenter la l-value *a* de 1.
a++ est une expression valant *a*, et ayant pour effet de bord d'augmenter la l-value *a* de 1.
La sémantique de --*a* et *a*-- est similaire, mais il s'agit de diminuer *a* de 1 et non de l'augmenter de 1.

Exemple: qu'affichent les lignes suivantes ?

```
int a=1;
printf("%d\n",++a);
printf("%d\n",a--);
```


Expression composée

Syntaxe: *expression1*, *expression2*

Sémantique: *expression1*, *expression2* est une expression formée de deux expressions. *expression1* est évaluée, sa valeur est perdue, puis *expression2* est évaluée. La valeur et le type de *expression1*, *expression2* sont ceux de *expression2*.

La priorité des opérateurs

Opérateur	Symbole	Exemple	Assoc.
Indexation	[]	t[i][j]	
Accès struct.	. ->	c.reel	->
Appel fct.	()	fact(30U)	
Adresse	&	&v	
Déréférecmnt	*	*p	
Taille objets	sizeof	sizeof(int)	
Transtypage	()	(float)4	
Nég. Logique	!	!b	<-
Cplt à un	~	~0022	
pl/neg unaires	+ -	+1 -1	
pré/post inc.	++	++i i++	
pré/post dec.	--	--i i--	

+
Priorité
-

La priorité des opérateurs

<table>
<tr><th>Opérateur</th><th>Symbole</th><th>Exemple</th><th>Assoc.</th></tr>
<tr><td>Produit
div/modulo</td><td>*
/ %</td><td>a*b
a/b a%b</td><td>-></td></tr>
<tr><td>somme/diff.</td><td>+ -</td><td>a+b a-b</td><td>-></td></tr>
<tr><td>Décalages</td><td><< >></td><td>a<<1 a>>1</td><td>-></td></tr>
<tr><td>Ordre</td><td><= =></td><td>a<b ...</td><td>-></td></tr>
<tr><td>égalité/diff.</td><td>== !=</td><td>a==0</td><td>-></td></tr>
<tr><td>Et bit à bit</td><td>&</td><td>a&0022</td><td>-></td></tr>
<tr><td>XOR bit à bit</td><td>^</td><td>a^0022</td><td>-></td></tr>
<tr><td>OU bit à bit</td><td>|</td><td>a|0022</td><td>-></td></tr>
<tr><td>ET logique</td><td>&&</td><td>a&&b</td><td>-></td></tr>
<tr><td>OU logique</td><td>||</td><td>a||b</td><td>-></td></tr>
<tr><td>expr. cond.</td><td>?:</td><td>x>0 ? x: -x</td><td><-</td></tr>
<tr><td>affectation
affect. étendue</td><td>=
+= -= ...</td><td>a=3
a <<= 3U</td><td><-</td></tr>
<tr><td>expr. comp.</td><td>,</td><td>i=2, j=6</td><td>-></td></tr>
</table>

+
Priorité
-