EXPRESSIONS

1. Instruction de base
2. Évaluation
3. Caractères
4. Entiers
5. Types flottants
6. Booléens
7. Le type `void`
8. Compatibilité des types prédéfinis
9. Portabilité des types
10. Constructeurs de types
11. Liste des opérateurs
12. Précédence et associativité
13. Opérateurs arithmétiques
14. Comparateurs
15. Opérateurs logiques
16. Opérateurs bit à bit
17. Expressions conditionnelles
18. Affectations
19. Taille
20. Composition

Instruction de base

$\boxed{\textit{expression};}$

Une expression est formée à partir de constantes et d'identificateurs à l'aide d'**opérateurs**.

Une expression

- détermine une **valeur**
  (évaluation en respectant les priorités des opérateurs)
- possède un **type**
  (évaluation au moyen des règles de typage)
- peut avoir une **action**
  (effet de bord)

Exemple

```
   ...
   short h = 2; int x, y, i = 10;
   x = y = 3 * h + i;
/* expression qui vaut 16, de type int, et qui
   attribue aux variables x et y la valeur 16.
*/
   ...
```

Évaluation

L'évaluation d'une expression se fait en appliquant

- les règles de priorité et d'associativité des opérateurs
- les règles définissant le type des sous-expressions

`4 + 2 * 3` vaut 10
`6 / 2 * 3` vaut 9
`5 / 2` vaut 2
`(double) 5 / 2` vaut 2,5

NOTE : La valeur d'une expression dépend du type des éléments

Exemple

```
   ...
   y = 5;
   x = y / 2;
/* evaluation impossible sans le type de y */
   ...
```

Caractères

Les types `char` sont des sous-ensembles des types entiers.
Un caractère est assimilé à son code ASCII (ou autre).

1– DÉSIGNATIONS et DOMAINES

- `char` représente selon la machine
  - l'ensemble {-128, ..., +127} (signé)
  - l'ensemble {0, ..., +255} (non signé)
- `unsigned char` représente toujours
  - l'ensemble {0, ..., +255}
- `signed char` représente toujours
  - l'ensemble {-128, ..., +127}

2– OPÉRATIONS : comme pour les entiers

3– FONCTIONS dans la bibliothèque <ctype.h> :

```
tolower, toupper, isalpha, isalnum, iscntrl,
isdigit, isxdigit, islower, isupper, isgraph,
isprint, ispunct, isspace
```

Toutes de paramètre `int` pour la portabilité !

4– CONSTANTES

- caractères en quotes : 'A', '*', ...
- caractères en décimal : 65, 42, ...
- caractères en octal : '\101', '\52', ...
- caractères en hexadécimal : '\x41', '\x2A', ...
- notation abrégée de caractères spéciaux :

'\n'	newline	'\t'	tab
'\v'	VT	'\b'	BS
'\r'	CR	'\f'	FF
'\a'	BEL	'\\'	\
'\?'	?	'\''	'
'\"'	"	'\0'	NUL

Exemple

```
#include <stdio.h>
main(){
   char a, b, c, d;
   a = 'A'; b = 65; c = '\101'; d = '\x41';
   printf("%ch! %ch! %ch! %ch!\n", a, b, c, d);
   printf("%c %d %o %x\n", a, a, a, a);
}
```

```
Ah!  Ah!  Ah!  Ah!
A 65 101 41
```

Entiers

1– DÉSIGNATIONS et DOMAINES

- `int` en général 2 ou 4 octets
- entiers courts
  - `short int` ou simplement `short`
  - en général, `sizeof(short)` = 2, {-32768, ..., +32767}
- entiers longs
  - `long int` ou simplement `long`
  - en général, `sizeof(long)` = 4, {-2147483648, ..., +2147483647}
- entiers signés (par défaut)
  - `signed int`
  - `signed short int`
  - `signed long int`
- entiers sans signe (calcul modulo 2^n où n est le nombre de bits : il n'y a donc pas de débordement - pas d'*overflow*)
  - `unsigned int`
  - `unsigned short int`, {0, ..., +65535}
  - `unsigned long int`
- entiers "très courts": `char`

2– CONSTANTES

- on préfixe par `0` (zéro) pour octal

  `032 = 26` et `077 = 63`
- on préfixe par `0x` pour hexadécimal

  `0x32 = 50` et `0x3F = 63`
- on suffixe par `l` ou `L` pour `long`

  `0xaf9fL = 44959`
- on suffixe par `u` ou `U` pour `unsigned`

  `0xffffu = 65535`

3– OPÉRATIONS

```
+ - * / %
== != < <= > >=
++ -- (sur variables)
```

4– FONCTIONS : voir fonctions du type flottant

5– REPRÉSENTATION

La représentation des entiers relatifs dépend de la machine (complément à 2 ou à 1). Sur un octet on représente

- l'ensemble {-128, ..., 127} en complément à 2
- l'ensemble {-127, ..., 127} en complément à 1

Types flottants

1– DÉSIGNATIONS et DOMAINES

- `float`
  - réels simple précision
  - en général, `sizeof(float)` $= 4$
  - $10^{-37} \leq abs(f) \leq 10^{38}$, 6 chiffres significatifs
- `double`
  - réels double précision
  - en général, `sizeof(double)` $= 8$
  - $10^{-307} \leq abs(f) \leq 10^{308}$, 15 chiffres significatifs
- `long double`
  - réels "extra" double précision
  - en général, `sizeof(long double)` $= 12$
  - $10^{-4931} \leq abs(f) \leq 10^{4932}$, 18 chiffres significatifs

2– CONSTANTES

- les constantes sont en double par défaut

  126.7 1.267E2 1267E-1 .1267e+3 .12 12.

- suffixe `f` ou `F` pour `float`, `12.8f`
- suffixe `l` ou `L` pour `long double`, `1289.34e13L`

3– OPÉRATIONS

```
+ - * /
== != < <= > >=
```

4– FONCTIONS

Dans la bibliothèque <math.h> :
`sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, exp, log, log10, pow, sqrt, ceil, floor, fabs, ldexp, frexp, modf, fmod`

5– REPRÉSENTATION

Standard IEEE, en général

Booléens

1– DÉSIGNATION : inexistante

2– DOMAINE : {faux, vrai}

3– CONSTANTES

- FAUX = 0
- VRAI = tout entier non nul

4– OPÉRATIONS

&&	||	!
==	!=	

Déclaration possible :

```
typedef enum {faux = 0, vrai = 1} booleen;
```

Le type `void`

- une conversion en type `void` d'une expression signifie que sa valeur est ignorée
- utilisé pour le type pointeur
- utilisé par les fonctions sans valeur, ou sans paramètre
- sert comme type générique de pointeur : pas de contrôle de type avec un pointeur de type `void`

Compatibilité des types prédéfinis

RETENIR: Tous les types arithmétiques, entiers, caractères, réels, sont compatibles entre eux

- le compilateur admet un mélange d'objets de ces divers types dans une expression, en particulier dans une affectation, mais effectue des conversions implicites
- à l'évaluation d'une expression, les conversions implicites respectent les règles suivantes
  - en l'absence de `unsigned`, les objets sont convertis dans le type le plus fort selon l'ordre décroissant :

    `long double, double, float, long int, int.`
  - les règles pour les opérandes `unsigned` dépendent de l'implémentation
  - la conversion en `unsigned` ne se fait qu'en cas de nécessité
  - **promotion entière** : l'arithmétique se fait au moins en `int`, c'est-à-dire que `char` et `short` sont promus en `int`
  - **attention**, suivant l'implémentation un `char` peut être promu en un entier négatif !

```
char c; short s; long l;
float f; long double g;
c = s*l;
f = 2*g;
```

`s` est converti en `long` dans l'évaluation de `s*l`
le résultat de `s*l` est converti en `char` pour l'affectation à `c`
2 est converti en `long double` dans l'évaluation de `2*g`
le résultat de `2*g` est converti en `float` pour l'affectation à `f`

- **Coercition** : le type peut être forcé (*cast* en anglais). La syntaxe est

  (*nom_du_type*) *expression*

  Exemple

```
#include <stdio.h>
main(){
   int i, j; double x, y, z, t;
   i = 5/2;  x = 5/2; y = (double) (5/2);
   j = (double) 5/2;  z = (double) 5/2;  t = 5./2;
   printf("%d, %g, %g, %d, %g, %g\n",
            i,  x,  y,  j,  z,  t);
}
```

```
2, 2, 2, 2, 2.5, 2.5
```

- dans une affectation, le type du membre droit est converti dans le type du membre gauche (qui est le type du résultat). Peut produire :
  - indétermination
  - perte de précision
    de `double` à `float` ou de `float` à `long int`, par exemple
  - perte de bits de poids fort
    de `long int` à `int`, par exemple

Exemple

`s = c;`

`char c`	`short s`
(-126) 1000 0010	(-126) 1111 1111 1000 0010

`c = s;`

`char c`	`short s`
(1) 0000 0001	(-32767) 1000 0000 0000 0001

`c = s;`

`unsigned char c`	`short s`
(255) 1111 1111	(-1) 1111 1111 1111 1111

`s = c;`

`unsigned char c`	`short s`
(255) 1111 1111	(255) 0000 0000 1111 1111

Portabilité des types

Les fichiers d'inclusion du C ANSI contiennent des caractéristiques de certains types :

- `<limits.h>` : pour les types de base

  entier minimum, entier maximum, nombre de bits par caractère, ...

- `<float.h>` : pour les types flottants

  epsilon-machine, exposant maximum, base de l'exposant, précision décimale, ...

- `<stdlib.h>` contient des fonctions de conversions de types

  chaîne en flottant, chaîne en entier, ...

Constructeurs de types

On construit de nouveaux types à partir des types prédéfinis, en appliquant les constructeurs de types

tableau	`[..]`
structure	`struct`
union	`union`
pointeur	`*`
fonctions	`(..)`

Exemple

```
union LI {long l; int i;};
struct S {char *nom; int taille;};
char * (*tf[5]) (union LI li, struct S s);
```

`tf` est un tableau de 5 pointeurs sur fonctions; chacune de ces fonctions a pour valeur un pointeur sur `char` et pour paramètres une union `LI` et une structure `S`.

Liste des opérateurs

Préc.↘	OPÉRATEUR	SYMBOLE	EXEMPLE	ASSOC.
	indexation	`[]`	`t[i][j]`	
	champ de struct.	`.` `->`	`c.reel`	->
	appel de fonct.	`()`	`max(t)`	
	adresse	`&`	`&i`	
	indirection	`*`	`*p`	
	taille	`sizeof`	`sizeof i`	
	coercition	(*type*)	`(float) p`	
logique	NON logique	`!`	`!(i==2)`	<-
bit à bit	NON bit à bit	~	`~066`	
	pl/moins unaires	`+` `-`	`-2`	
	incrémentation	`++`	`++i, i++`	
arithmé-	décrémentation	`--`	`--i, i--`	
tiques	multiplication	`*`	`a*b`	->
	division/modulo	`/` `%`	`a/b`	
	addition/soustr.	`+` `-`	`a+b a-b`	->
bit à bit	décalages	`<<` `>>`	`x>>n`	->
compa-	inégalités	`< <= > >=`	`a<3`	->
rateurs	égalité/différ.	`==` `!=`	`i==0`	->
	ET bit à bit	`&`	`a & mask`	->
bit à bit	OU excl. b. à b.	`^`	`a^3777`	->
	OU bit à bit	`|`	`a|b`	->
logi-	ET logique	`&&`	`a&&b`	->
ques	OU logique	`||`	`a||b`	->
	expr. condition.	`?:`	`x>0?:x:-x`	<-
	affect. simple	`=`	`i=j=2`	<-
	affect. étendue	`+=` `*=` ...	`i+=2`	
	exp. composée	`,`	`i=2, g=6`	->

Précédence et associativité

- les opérateurs unaires ont tous la même précédence, qui est l'une des plus fortes
- les trois opérateurs arithmétiques multiplicatifs (`* / %`) ont tous la même précédence, plus forte que celle des deux opérateurs additifs (`+ -`)

  `-a*b+c` s'interprète comme `((-a)*b)+c`

- à l'intérieur d'une même classe de précédence les opérateurs s'associent en général de gauche à droite, sauf les opérateurs unaires et les opérateurs d'affectation

  `a/b*c` s'interprète comme `(a/b)*c`,
  et non comme `a/(b*c)`
  `- sizeof i` s'interprète comme `-(sizeof i)`
  `x = y = z` s'interprète comme `x = (y = z)`

- les parenthèses forcent la précédence ou l'associativité
- l'ordre d'évaluation des opérandes dépend de l'implémentation, sauf pour ces quatre opérateurs :

`&&`	`||`	`?:`	`,`

Exemples

- toute expression qui dépend de l'ordre d'évaluation des opérandes doit être jugée incorrecte

- Expressions correctes

```
n * k + m / 5 + 12
'A' + 800
(double) 2 / 3
(c = getchar()) != EOF
x = y = z = t + 18
2 * (x = 3) + 10
t[i++] = x
x>0 ?  x :  -x
t[k = ++i] = x
y = f(x = 17) * 3
x = 17, y = f(y) * 3
++n, f(n)
```

- Expressions incorrectes

```
t[i] = i++
t[i++] = t[i]
(x = 3) * x
f(x = 17) + g(x)
```

Opérateurs arithmétiques

- il y a différentes arithmétiques

  dans les entiers (signés ou non)

  dans les flottants

  sur les pointeurs

- opérateurs

		entiers	flottants	pointeurs
plus unaire	`+`	$\surd$	$\surd$	
moins unaire	`-`	$\surd$	$\surd$	
incrémentation	`++`	$\surd$	$\surd$	$\surd$
décrémentation	`--`	$\surd$	$\surd$	$\surd$
multiplication	`*`	$\surd$	$\surd$	
division	`/`	entière	$\surd$	
modulo	`%`	$\surd$		
addition	`+`	$\surd$	$\surd$	$\surd$
soustraction	`-`	$\surd$	$\surd$	$\surd$

1– Opérateurs d'incrémentation

expression	`++i`	`i++`	`--i`	`i--`
valeur	`i+1`	`i`	`i-1`	`i`
valeur de `i` **après** évaluation	`i+1`		`i-1`	

`int i, t[10];`
`i = 0; t[i++]` *a pour valeur* `t[0]`
`i = 0; t[++i]` *a pour valeur* `t[1]`
dans les deux cas `i` *a pour valeur* `1` *après utilisation, par effet de bord*

Remarque

Les opérateurs `++` et `--` s'appliquent à une expression désignant un objet en mémoire ("L-value"), par exemple un identificateur de variable, mais pas une constante ou une expression arithmétique.

`++5` `--(i+1)` `++i++` **n'ont pas de sens**

Comparateurs

`>`	`>=`	`<`	`<=`	`==`	`!=`

- ils permettent de former des expressions logiques

$$\underbrace{\underbrace{\texttt{4*i+j}}_{\textit{exp. arith.}} \quad \texttt{<=} \quad \underbrace{\texttt{2+z/10}}_{\textit{exp. arith.}}}_{\textit{expression logique}}$$

- toute expression logique est de type `int` et a pour valeur
  - 0 si elle est évaluée à FAUX
  - 1 si elle est évaluée à VRAI

`i>=3` vaut 0 si $i<3$, et vaut 1 si $i\geq 3$

- inversement toute entier a une valeur logique qui est
  - FAUX si l'entier est 0
  - VRAI sinon

`a+b && a-b` est vraie
si et seulement si `a+b` et `a-b` sont vraies,
c'est-à-dire, si et seulement si $a \neq \pm b$

Opérateurs logiques

NON logique	`!`
ET logique	`&&`
OU logique	`||`

Exemple

On suppose que $a \leq b$

conditions	expressions C
$a \leq x \leq b$	`(x >= a) && (x <= b)` `x >= a && x <= b`
$a > x$ ou $x < b$	`x < a || x > b` `!(x >= a && x <= b)`

- Évaluation paresseuse des opérandes de gauche à droite

1– Règles d'évaluation paresseuse

ET logique

`e1`	`e2`	`e1 && e2`
$= 0$	non évaluée	$= 0$
$\neq 0$	évaluée	$= 1$ si `e2` $\neq 0$ $= 0$ si `e2` $= 0$

OU logique

`e1`	`e2`	`e1 || e2`
$\neq 0$	non évaluée	$= 1$
$= 0$	évaluée	$= 1$ si `e2` $\neq 0$ $= 0$ si `e2` $= 0$

2– ATTENTION AUX EFFETS DE BORD

```
int i = 0, j = 0;
i++ && j++; /* i=1 et j=0 */
i = 0; j = 0;
j++ && i++; /* i=0 et j=1 */
```

Opérateurs bit à bit

complément	~
décalage à droite	>>
décalage à gauche	<<
ET	&
OU exclusif	^
OU	|

- S'appliquent à des entiers (`char, int, ...`)
- Forcer un ou plusieurs bits à `0`

```
char c;
c = 13 & 3;
```

```
13  0...01101
 3  0...00011
 ----------------
    0...00001  ->1
```

- Forcer un ou plusieurs bits à `1`

```
char c;
c = 13 | 3;
```

```
13  0...01101
 3  0...00011
 ----------------
    0...01111  ->15
```

- Inverser un ou plusieurs bits

```
char c;
c = 13 ^ 3;
```

```
13  0...01101
 3  0...00011
 ----------------
    0...01110  ->14
```

- Créer des masques indépendamment de la taille des entiers (applications au cas où cette taille est inconnue)

```
~ 0
```

1111	sur 4 bits
11111111	sur 8 bits
...	

Expressions conditionnelles

$\boxed{\textit{exp1} \ ? \quad \textit{exp2} \ : \quad \textit{exp3}}$

- si *exp1* est VRAIE, la **valeur** est celle de *exp2*, et *exp3* n'est pas évaluée
- si *exp1* est FAUSSE, la **valeur** est celle de *exp3*, et *exp2* n'est pas évaluée

Exemples

```
abs = x>=0 ? x : -x

max = x>y ? x : y

max = x>y ? x>z ? x : z : y>z ? y : z

printf("Il y a %d element%s", n, n>1 ? "s" : "");
```

Affectations

- **L'affectation est un opérateur**

  `e1 = e2`

  – le type de l'expression est celui de `e1`

  – la valeur de l'expression est celle de `e1` (après évaluation)

- Exemple

  `double x;`

  alors `x = sqrt(9)` est une expression

  – de type `double`, de valeur `3`

  – à effet de bord: la valeur de `x`, après évaluation de l'expression, est égale à `3`

- **Ne pas confondre** `e1 = e2`
  avec l'expression logique `e1 == e2`

```
resultat = 109;
if (resultat = 0) printf("resultat NUL");
else printf("resultat = %d", resultat);
```

  produit

  `resultat = 0`

1– Combiner affectation et test

```
while ((c = getchar()) != EOF)
     putchar(c);
```

L'expression `c=getchar()` a pour valeur le code ASCII du caractère lu, ou `EOF` (`-1`)

2– REGROUPEMENT

`i = j = k = 10` équivaut à `i = (j = (k = 10))` (associativité de `=`)

3– AFFECTATION ÉTENDUE

`e1` *op*`=` `e2` équivaut à `e1 = e1` *op* `(e2)`
(sauf que `e1` n'est évaluée qu'une seule fois)
où *op*`=` est l'un des opérateurs :

`+=`	`-=`	`*=`	`/=`	`%=`
`&=`	`|=`	`^=`	`<<=`	`>>=`

Exemple

`x *= a+b` équivaut à `x = x * (a+b)` et non à `x = (x*a)+b`

Exemple

`i &= i-1` équivaut à `i = i & (i-1)`,
ce qui revient à supprimer le dernier 1 de la représentation binaire de l'entier `i`

Taille

Deux emplois possibles

`sizeof(`*identificateur_de_type*`)`
`sizeof` *expression*

Renvoie la taille en octets du type donné ou du type de l'expression
Sert pour l'allocation dynamique
Utile pour la portabilité des programmes
La valeur est de type `size_t` (type entier défini dans la bibliothèque standard)

Exemple

```
sizeof(short) vaut 2
char t[100];
sizeof t vaut 100
```

Composition

$\boxed{\textit{exp1},\ \textit{exp2},\ \ldots,\ \textit{expn}}$

Exemple

```
i=1, j=2, k=i*j+1
```

- les expressions sont évaluées de la gauche vers la droite
- la valeur de l'expression composée est celle de la dernière expression évaluée, celle de droite
- cet opérateur intervient essentiellement dans les boucles `for`

EXERCICES

1. **Degrés** : faire afficher une table de correspondance entre degrés Fahrenheit et Celsius

   [NOTE : $C = \frac{5}{9}(F - 32)$]

2. **Équation** : écrire une fonction qui calcule les solutions de l'équation $ax^2 + bx + c = 0$, où a, b, c sont des flottants.

   [NOTE : quand $a \neq 0$, les solutions sont $(-b-\sqrt{b^2 - 4ac})/2a$ et $(-b + \sqrt{b^2 - 4ac})/2a$]

3. **Capital** : écrire une fonction qui donne le capital obtenu en plaçant une somme s au taux de $t\%$ pendant n années.

4. **Bits** : écrire une fonction qui donne le nombre de bits égaux à 1 dans la représentation d'un entier en mémoire.

   [NOTE : utiliser les opérateurs & et `>>`]

5. **Majuscules/minuscules** : écrire les fonctions booléennes `islower`, `isdigit`, et les fonctions `tolower`, `toupper`.

6. **Maximum** : écrire une expression qui donne le maximum de quatre nombres.

Degrés

```
#include <stdio.h>
main(){
   int f; float c;
      printf("Farhenheit Celsius\n");
   for(f=20; f<80; f += 4){
      c = (float) 5 / 9 * (f-32);
      printf("   %3d     %6.2f\n", f, c);
   }
}
```

```
Farhenheit Celsius
    20      -6.67
    24      -4.44
    28      -2.22
    32       0.00
    36       2.22
    40       4.44
    44       6.67
    48       8.89
    52      11.11
    56      13.33
    60      15.56
    64      17.78
    68      20.00
    72      22.22
    76      24.44
```

Équation

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

main(){
   float a, b, c, delta, x, y;

   printf("Trois flottants ? ");
   scanf("%f%f%f", &a, &b, &c);
   printf("Equation %fxX2 + %fxX + %f = 0\n", a, b, c);
   if(a == 0)
      if(b == 0)
         if(c == 0) printf("Tout est solution\n");
         else printf("Pas solution\n");
      else printf("Une solution : %g\n", -c/b);
   else{
      delta = b*b - 4*a*c;
      if(delta >= 0){
         x = (-b - sqrt(delta))/(2*a);
         y = (-b + sqrt(delta))/(2*a);
         printf("Deux solutions : %f, %f\n", x, y);
      } else printf("Pas de solution reelle\n");
   }
}
```

Capital

```
double puissance(double x, int n){

   if(n < 0){n = -n; x = 1/x;}
   if(n == 0)
      return 1;
   else if(n % 2 == 0)
      return puissance(x*x, n/2);
   else
      return x * puissance(x*x, n/2);
}


double capital(double somme,
               double interet,
               int annees){

   return somme * puissance(1+interet/100, annees);
}
```

Bits

```
#include <stdio.h>

int nb1(unsigned n){
   int nb = 0;
   for( ; n!=0; n >>= 1)
      if(n & 1) nb++;
   return nb;
}

main(){
   unsigned n;
   do{
      printf("Un entier ? ");
      scanf("%d", &n);
      printf("Bin(%d) contient %d '1' \n", n, nb1(n));
   } while (n != 0);
}
```

Majuscules/minuscules

```
int toupper(int c){
   return (islower(c) ? (c ^ 0x20) : c);
}


int tolower(int c){
   return (isupper(c) ? (c ^ 0x20) : c);
}
```

Autres solutions

```
int toupper(int c){
   return (islower(c) ? (c-32) : c);
}


int tolower(int c){
   return(isupper(c) ? (c+32) : c);
}
```

Maximum

```
int max4(int i, int j, int k, int l){
   int max;
   if (i>j)
      if(k>l)
         if(i>k) max = i;
         else max = k;
      else
         if(i>l) max = i;
         else max = l;
   else
      if(k>l)
         if(j>k) max = j;
         else max = k;
      else
         if(j>l) max = j;
         else max = l;
   return max;
}

int Max4(int i, int j, int k, int l){

   return i>j ? k>l ? i>k ? i : k : i>l ? i : l
             : k>l ? j>k ? j : k : j>l ? j : l;
}
```