

# Tableaux et chaînes de caractères

Marie Pelleau

`marie.pelleau@univ-cotedazur.fr`

Basé sur les transparents de Jean-Charles Régim

# Type énuméré

```
#include <stdio.h>

enum Lights {green, yellow, red};
enum Cards {diamond = 1, spade = -5, club = 5, heart};
enum Operator {Plus = '+', Min = '-', Mult = '*', Div = '/'};

int main (void) {
    enum Lights feux = red;
    enum Cards jeu = spade;
    enum Operator op = Min;

    printf("L = %d %d %d\n", green, yellow, red);
    printf("C = %d %d %d %d\n", diamond, spade, club, heart);
    printf("O = %d %d %d %d\n", Plus, Min, Mult, Div);

    jeu = yellow;
    printf("%d %d %c\n", feux, jeu, op);

    return 0;
}
```

# Type énuméré

- Un type énuméré est considéré comme de type `int` : la numérotation commence à 0, mais on peut donner n'importe quelles valeurs entières
- On peut affecter ou comparer des variables de types énumérés
- Pas de vérification

# Type structurés

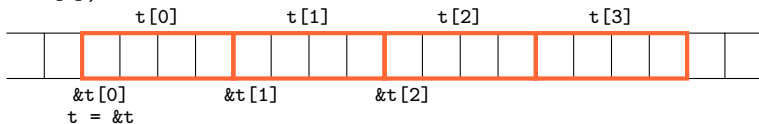
```
#include <stdio.h>
int main (void) {
    int t[4];
    int u[] = {0, 1, 2, 3};
    float x[3][10];
    char w[][3] = {{ 'a', 'b', 'c'}, {'d', 'e', 'f'}};

    int i;
    for (i = 0; i < 4; i++) {
        t[i] = 0;
        printf("t[%d] = %d ", i, t[i]);
    }
    fputc('\n', stdout);

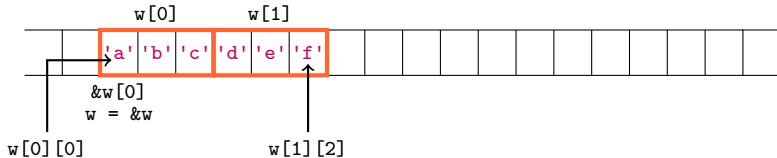
    for (i = 0; i < 2; i++) {
        int j;
        for (j = 0; j < 3; j++) {
            w[i][j] = 'a';
            fprintf(stdout, "w[%d][%d] = %c ", i, j, w[i][j]);
        }
        fputc('\n', stdout);
    }
    return 0;
}
```

# Tableaux

```
int t[4];
```



```
char w[][3] = {{ 'a', 'b', 'c' }, { 'd', 'e', 'f' } };
```



# Tableaux

- Tableaux à une seule dimension : possibilité de tableaux de tableaux
- Dans ce cas, la dernière dimension varie plus vite
- Indice entier uniquement (borne inférieure toujours égale à 0)
- On peut initialiser un tableau lors de sa déclaration (par agrégat)
- Dimensionnement automatique par agrégat (seule la première dimension peut ne pas être spécifiée)
- Les opérations se font élément par élément
- Aucune vérification sur le dépassement des bornes

# Tableaux

- La dimension doit être connue **statiquement** (lors de la compilation)  

```
int n = 10;  
int t[n]; /* INTERDIT */
```
- Ce qu'il faut plutôt faire  

```
#define N 10  
...  
int t[N]; /* c'est le préprocesseur qui travaille*/
```
- On verra plus tard comment définir des tableaux de façon **dynamique** (taille connue à l'exécution)

# Tableaux : initialisation

```
#include <stdio.h>

void init (int t[], int n, int v) {
    int i;
    for (i = 0; i < n; i++) {
        t[i] = v;
    }
}

void aff (int t[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        printf("%d ", t[i]);
    }
    printf("\n");
}

int main (void) {
    int tab[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
    aff(tab, 5);
    init(tab, 5, 0);
    aff(tab, 10);
    return 0;
}
```



# Chaînes de caractères

- Ce sont des tableaux de caractères : pas un vrai type
- Par convention, elles se terminent par le caractère nul `'\0'`
- Il n'y a pas d'opérations pré-définies (puisque ce n'est pas un type), mais il existe des fonctions de bibliothèque, dont le fichier de déclarations s'appelle `string.h`

# Chaînes de caractères

## Exemple

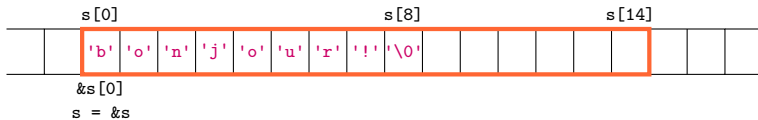
```
char string[100];  
char s[15] = {'b', 'o', 'n', 'j', 'o', 'u', 'r', '!', '\\0'}  
}; /* "bonjour!" */  
char c[] = "0123456789";  
char *c2 = "0123456789";
```

## Ecriture sur la sortie standard

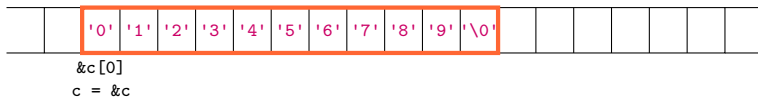
```
fprintf(stdout, "%s", s);  
printf("%s", s);  
  
fputs(s, stdout);  
puts(s);
```

# Chaînes de caractères

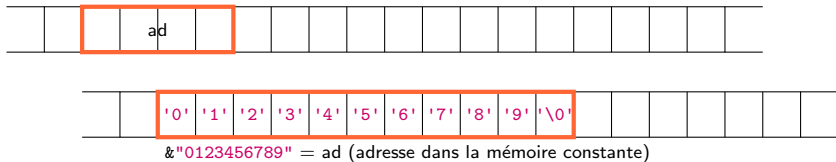
```
char s[15] = {'b', 'o', 'n', 'j', 'o', 'u', 'r', '!', '\\0'};
```



```
char c[] = "0123456789";
```



```
char *c2 = "0123456789";
```



# Chaînes de caractères : initialisation

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

int main (void) {
    char chaine[] = "bonjour ";
    char chRes[256];

    printf("%s\n", strcpy(chRes, chaine));
    printf("%s\n", strcat(chRes, "tout le monde!"));
    printf("%s\n", chRes);

    return 0;
}
```

# Structures

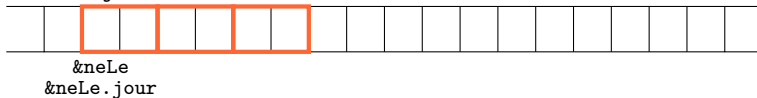
```
struct date {  
    short jour, mois, annee;  
};
```

```
struct etudiant {  
    int num;  
    struct date neLe;  
    short tn[3];  
};
```

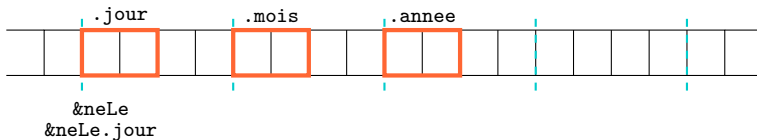
- Objet composite formé d'éléments de types quelconques
- La place réservée est la somme de la longueur des champs (au minimum, à cause de l'alignement)
- On peut affecter des variables de types structures
- Lors de la **déclaration** de variables de types structures, on peut initialiser avec un agrégat

# Structures

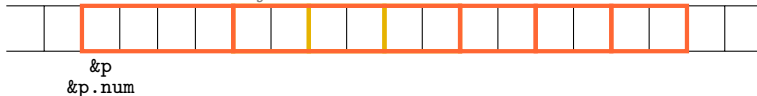
```
struct date neLe;
    .jour .mois .annee
```



```
struct date neLe; // avec alignement
```



```
struct etudiant p;
    .num .jour .date .mois .annee .tn[0] .tn[1] .tn[2]
```



# Structures

```
#include <stdio.h>

struct date {
    short jour, mois, annee;
};

struct etudiant {
    int num; /* numéro de carte */
    struct date neLe;
    short tn[3]; /* tableau de notes */
};

int main (void) {
    struct etudiant p;
    struct etudiant etud[2];
    short e, n, somme;

    p.num = 15;
    p.neLe.jour = 5;
    p.neLe.mois = 11;
    p.neLe.annee = 2000;
    p.tn[0] = 10;
    p.tn[1] = 15;
```

```
    p.tn[2] = 20;
    etud[0] = p;

    p.num = 20;
    struct date d = {25, 1, 2001};
    p.neLe = d;
    p.tn[0] = 0;
    p.tn[1] = 5;
    p.tn[2] = 10;
    etud[1] = p;

    for (e = 0; e < 2; e++) {
        somme = 0;
        for (n = 0; n < 3; n++) {
            somme = somme + etud[e].tn[n];
        }
        printf ("moy de %d, né(e) en %hd = %.2f\n", etud[e].num, etud[e].neLe.annee,
            somme/3.0);
    }
}
```

# Union

```
union donnee {  
    int i;  
    short s;  
    char c;  
};
```

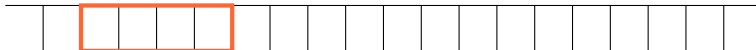
- La place réservée est le maximum de la longueur des champs
- Sert à partager la mémoire dans le cas où l'on a des objets dont l'accès est exclusif
- Sert à interpréter la représentation interne d'un objet comme s'il était d'un autre type



# Union

```
union donnee {  
    int i;  
    short s;  
    char c;  
};
```

```
union donnee data;
```



&data  
&data.i  
&data.s  
&data.c

# Union

```
#include <stdio.h>

union donnee {
    int i;
    short s;
    char c;
};

int main (void) {
    union donnee data;
    data.i = 123456;
    printf ("%d, %hd, %c\n", data.i, data.s, data.c);

    data.s = 42;
    printf ("%d, %hd, %c\n", data.i, data.s, data.c);

    data.c = 'Z';
    printf ("%d, %hd, %c\n", data.i, data.s, data.c);
    return 0;
}
```

# Champs de bits

- Pour les champs de bits, on donne la longueur du champ en bits; longueur spéciale 0 pour forcer l'alignement (champ sans nom pour le remplissage)
- Attention aux affectations (gauche à droite ou vice versa)
- Utiliser pour coder plusieurs choses sur un mot :
  - Je veux coder les couleurs sur un mot machine (ici 32 bits)
  - 3 couleurs donc 10 bits par couleur au lieu d'un octet

# Champs de bits

```
#include <stdio.h>

typedef struct {
    unsigned rouge : 10;
    unsigned vert : 10;
    unsigned bleu : 10;
    unsigned : 2;
} couleur;

int main (void) {
    couleur rouge = {0x2FF, 0x000, 0x000};
    couleur vert = {0x133, 0x3FF, 0x133};
    couleur bleu = {0x3FF, 0x3FF, 0x2FF};

    printf("rouge = %x\tvert = %x\tbleu = %x\n", rouge, vert, bleu);
    printf("rouge = %u\tvert = %u\tbleu = %u\n", rouge, vert, bleu);

    return 0;
}
```

# Champs de bits

- Dépend fortement de la machine, donc difficilement transportable
- Architectures droite à gauche : **little endian** (petit bout)
  - Octet de poids faible en premier (adresse basse)
  - x86
- Architectures gauche à droite : **big endian** (gros bout)
  - Octet de poids fort en premier (adresse basse)
  - Motorola 68000
- Souvent remplacé par des masques binaires et des décalages

# Définition de types : typedef

Permet de définir de nouveaux types (aide le compilateur)

```
typedef int Longueur;
```

```
typedef char tab_car[30];
```

```
typedef struct people {  
    char Name[20];  
    short Age;  
    long IdNumber;  
} Human;
```

```
typedef struct node *Tree;  
typedef struct node {  
    char *word;  
    Tree left;  
    Tree right;  
} Node;
```

```
typedef float (*arith) (int, int);
```

# Définition de types : typedef

```
#include <stdio.h>
#include <math.h>
typedef double (*unary) (double);
typedef double (*binary) (double, double);

double add (double a, double b) {
    return a + b;
}

double mult (double a, double b) {
    return a * b;
}

double div2 (double a) {
    return a / 2.0;
}

double moyenne (double a, double b, binary binop, unary unop) {
    double aux = (*binop)(a, b);
    double moy = (*unop)(aux);
    return moy;
}

int main (void) {
    double moy_arith = moyenne(15.0, 12.0, add, div2);
    printf("moyenne arithmétique = %lf\n", moy_arith);

    double moy_geom = moyenne(15.0, 12.0, mult, sqrt);
    printf("moyenne géométrique = %lf\n", moy_geom);
    return 0;
}
```

## Instruction vide

- Dénotée par le point-virgule
- Le point-virgule sert de terminateur d'instruction : la liste des instructions est une suite d'instructions se terminant toutes par un ;

```
for (; (fgetc(stdin) != EOF); nb++) /* rien */ ;
```

### Remarque

- Ne pas mettre systématiquement un ; après la parenthèse fermante du `for`
- Mettre toujours des { }, avec les `if` et les `for`



# Expression, instruction et affectation

## En général

- instruction  $\equiv$  action
  - expression  $\equiv$  valeur
- 
- Une expression a un type **statique** (c'est-à-dire qui ne dépend pas de l'exécution du programme) et une **valeur**
  - L'affectation en C est dénotée par le signe =

# Bloc

- Sert à grouper plusieurs instructions en une seule
- Sert à restreindre (localiser) la visibilité d'une variable
- Sert à marquer le corps d'une fonction

```
#include <stdio.h>
int main (void) {
    int i = 3;
    int j = 5;
    {
        int i = 4;
        printf("i = %d, j = %d \n", i, j);
        i++;
        j++;
    }
    printf ("i = %d, j = %d \n", i, j); /* valeur de i ? de j ? */
    return 0;
}
```

# Conditionnelle

- Il n'y a pas de mot-clé *alors* et la partie *sinon* est facultative
- La condition doit être parenthésée

## Remarque

En C, il n'y a pas d'expression booléenne ; une expression numérique est considérée comme **faux** si sa valeur est égale à 0, **vrai** sinon

# Conditionnelle

```
#include <stdio.h>
#include <stdlib.h>
int main (void) {
    int x, y = -3, z = -5;
    printf("valeur de x ?");
    scanf("%d", &x);
    if (x==0)
        x = 3;
    else if (x==2) {
        x=4; y = 5;
    }
    else
        x = 10;
    printf ("x = %d, y = %d, z = %d \n", x, y, z);
    if (0) x = 3; else x=4;
    return 0;
}
```

# Conditionnelle

```
#include <stdio.h>
#include <stdlib.h>
int main (void) {
    int x, y = -3, z = -5;
    printf("valeur de x ? ");
    scanf("%d", &x);
    if (x==0) {
        x = 3;
    } else if (x==2) {
        x=4; y = 5;
    } else {
        x = 10;
    }
    printf ("x = %d, y = %d, z = %d \n", x, y, z);
    if (0) {
        x = 3;
    } else {
        x=4;
    }
    return 0;
}
```

# Aiguillage (Switch)

- L'expression doit être entre parenthèses
- Les étiquettes de branchement doivent avoir des valeurs calculables à la compilation et de type discret
- Pas d'erreur si aucune branche n'est sélectionnée
- **Exécution des différentes branches en séquentiel** : ne pas oublier une instruction de débranchement (`break` par exemple)

# Aiguillage (Switch)

```
#include <stdio.h>

int main (int argc, char *argv[]) {
    short i = 0, nbc = 0, nbb = 0, nba = 0;
    if (argc != 2) {
        fprintf(stderr, "usage: %s chaîne \n", argv[0]);
        return 1;
    }
    while (argv[1][i]) {
        switch (argv[1][i]) {
            case '0' :
            case '1' :
            case '2' :
            case '3' :
            case '4' :
            case '5' :
            case '6' :
            case '7' :
            case '8' :
            case '9' : nbc++; break;
            case ' ' :
            case '\t' :
            case '\n' : nbb++; break;
            default : nba ++;
        }
        i++;
    }
    printf ("chiffres = %hd, blancs = %hd, autres = %hd\n", nbc, nbb, nba);
}
```

# Les boucles

```
while (expression entière) {  
    instructions  
}
```

```
#include <stdio.h>  
#define MAX 30
```

```
int main (int argc, char *argv[]) {  
    char tab[MAX], c;  
    int i;  
    i = 0;  
    while ((i < MAX -1) && (c = fgetc(stdin)) != EOF) {  
        tab[i++] = c;  
    }  
    printf("\n%s\n", tab);  
    return 0;  
}
```



# Les boucles

```
do {  
    instructions  
} while (expression entière);
```

```
#include <stdio.h>  
#include <string.h>  
#include <ctype.h>  
#define NB 3
```

```
int main (int argc, char *argv[]) {  
    char rep[NB];  
    do {  
        printf("Avez-vous fini ?");  
        fgets(rep, NB, stdin);  
        rep[0] = toupper(rep[0]);  
    } while (strcmp(rep, "O\n"));  
    return 0;  
}
```

# Les boucles

```
for (init; condition; pas) {  
    instructions  
}
```

```
init;  
while (condition) {  
    instructions  
    pas;  
}
```

```
#include <stdio.h>  
#define MAX 30
```

```
int main (int argc, char *argv[]) {  
    char tab[MAX] = "toto";  
    printf ("%s*\n", tab);  
    int i;  
  
    for (i = 0; i < MAX; i++) {  
        tab[i] = '\\0';  
    }  
    printf ("\n%s*\n", tab);  
    return 0;  
}
```

# Les boucles

```
#include <stdio.h>

int main (int argc, char *argv[]) {
    int i;
    i = 0;
    while (i < 10) {
        i ++;
    }
    printf("i = %d\n", i);

    i = 0;
    do {
        i ++;
    } while (i < 10);
    printf("i = %d\n", i);

    for (i = 0; i < 10; i++);
    printf("i = %d\n", i);
    return 0;
}
```

# Instructions de débranchement

- **break**
  - Utilisée dans un **switch** ou dans une boucle
  - Se débranche sur la première instruction qui suit le **switch** ou la boucle
- **continue**
  - Utilisée dans les boucles
  - Poursuit l'exécution de la boucle au test (ou au rebouclage)
- **goto**
  - Va à l'étiquette donnée
  - L'étiquette, etq par exemple, est placée comme suit  
etq : instruction
- **return**
  - Provoque la sortie d'une fonction
  - La valeur de retour est placée derrière le mot-clé
- **exit**
  - Met fin à l'exécution d'un programme
  - 0 en paramètre indique l'arrêt normal

# Instructions de débranchement

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

void f (int j) {
    int i = 0;
    while (i < j) {
        /* instruction */
        i++;
    }
    etq: printf("fin de f, i = %d\n", i);
}

int main (int argc, char *argv[]) {
    int i = 100;
    f(i);
    printf("fin de main, i = %d\n", i);
    return 0;
}
```

/\* instruction \*/

- fin de f, i = 100  
fin de main, i = 100
- break;  
fin de f, i = 0  
fin de main, i = 100
- continue;  
**boucle infinie !**
- goto etq;  
fin de f, i = 0  
fin de main, i = 100
- return;  
fin de main, i = 100
- exit(0);

# Opérateurs

## Affectation

Signe =, dont les opérandes sont de tout type (**attention** si de type tableau)

## Opérateurs unaires

Incrémentation (++) et décrémentation (--), **attention** l'ordre d'évaluation n'est pas garanti

- `t[i++] = v[i++]; /* à éviter */`
- `i = i++; /* n'est pas défini */`

# Opérateurs de calcul

- Arithmétiques      +, \*, -, /, %
- Relationnels      <, <=, >, >=, ==, !=
- Logiques          !, &&, ||

## Exemple

- 3 \* 4 + 5
- 3 / 4                    /\* division entière \*/
- 3 % 4                   /\* modulo \*/
- 3.0 / 4
- !(n % 2)                /\* n est pair ? \*/
- (x = 1) || b            /\* vrai \*/
- (x = 0) && b            /\* faux \*/

# Opérateurs de calcul

- Bit à bit     ~, &, |, ^, <<, >>

## Exemple

- ~3                /\* 0000 0011 = 1111 1100 \*/
- 3 & 5            /\* 0000 0011 & 0000 0101 = 0000 0001 \*/
- 3 | 5            /\* 0000 0011 | 0000 0101 = 0000 0111 \*/
- 3 ^ 5            /\* 0000 0011 ^ 0000 0101 = 0000 0110 \*/
- 3 << 5           /\* 0000 0011 << 5 = 0110 0000 \*/
- 73 >> 5          /\* 0100 1001 >> 5 = 0000 0010 \*/



# Opérateurs de calcul

```
#include <stdio.h>

int main (int argc, char *argv[]) {
    char i;
    for (i = 0; i < 10; i++) {
        printf ("i = %hx, ~i = %hx, ", i, ~i);
        printf ("!i = %hx, i << 1 = %hx, ", !i, i << 1);
        printf ("i >> 1 = %hx\n", i >> 1);
    }
    return 0;
}
```

```
i = 0, ~i = ff, !i = 1, i << 1 = 0, i >> 1 = 0
i = 1, ~i = fe, !i = 0, i << 1 = 2, i >> 1 = 0
i = 2, ~i = fd, !i = 0, i << 1 = 4, i >> 1 = 1
i = 3, ~i = fc, !i = 0, i << 1 = 6, i >> 1 = 1
i = 4, ~i = fb, !i = 0, i << 1 = 8, i >> 1 = 2
i = 5, ~i = fa, !i = 0, i << 1 = a, i >> 1 = 2
i = 6, ~i = f9, !i = 0, i << 1 = c, i >> 1 = 3
i = 7, ~i = f8, !i = 0, i << 1 = e, i >> 1 = 3
i = 8, ~i = f7, !i = 0, i << 1 = 10, i >> 1 = 4
i = 9, ~i = f6, !i = 0, i << 1 = 12, i >> 1 = 4
```

## Affectation composée

partie\_gauche  $\diamond$  = expression

avec  $\diamond \in \{+, -, *, /, \%, \wedge, \&, |, \ll, \gg\}$

```
int main (void) {  
    int a, b;  
    a = 3;  
    b = 5;  
  
    a += 3; /* a = a + 3 */  
    a -= b; /* a = a - b */  
    b *= a + 2; /* b = b * (a + 2) */  
    b <<= a; /* b = b << a */  
  
    return 0;  
}
```

# Opérateurs sur les types

## sizeof

- Taille d'un objet (nombre d'octets nécessaires à la mémorisation d'un objet)
- Renvoie une valeur de type `size_t` déclaré dans le fichier de déclarations `stdlib.h`
- `sizeof` (nom\_type)
- `sizeof` expression

# Opérateurs sur les types

```
#include <stdio.h>
#define imp(s, t) printf("sizeof %s = %d\n", s,
    sizeof(t))

int main (int argc, char *argv[]) {
    int t1[10];
    float t2[20];

    imp("char", char);
    imp("short", short);
    imp("int", int);
    imp("long", long);
    imp("float", float);
    imp("double", double);
    imp("long double", long double);

    printf ("sizeof t1 = %d\n", sizeof t1);
    printf ("sizeof t2 = %d\n", sizeof t2);
    printf ("sizeof t1[0] = %d\n", sizeof t1[0]);
    printf ("sizeof t2[1] = %d\n", sizeof t2[1]);

    return 0;
}
```

```
sizeof char = 1
sizeof short = 2
sizeof int = 4
sizeof long = 8
sizeof float = 4
sizeof double = 8
sizeof long double = 16
sizeof t1 = 40
sizeof t2 = 80
sizeof t1[0] = 4
sizeof t2[1] = 4
```

# Opérateurs sur les types

```
#include <stdio.h>
#include <string.h>
void f (int t[]) {
    printf ("\tf : sizeof t = %lu, sizeof t[0] = %lu\n", sizeof t, sizeof t[0]);
}
void g (char s[]) {
    printf ("\tg : sizeof s = %lu, sizeof s[0] = %lu\n", sizeof s, sizeof s[0]);
    printf ("\tg : longueur de s = %lu\n", strlen(s));
}
int main (int argc, char *argv[]) {
    int t1[10];
    char s1[] = "12345";
    printf ("main : sizeof t1 = %lu\n", sizeof t1);
    f(t1);
    printf ("main : sizeof s1 = %lu, strlen(s1) = %lu\n", sizeof s1, strlen(s1));
    g(s1);
    return 0;
}
```

```
main : sizeof t1 = 40
      f : sizeof t = 8, sizeof t[0] = 4
main : sizeof s1 = 6, strlen(s1) = 5
      g : sizeof s = 8, sizeof s[0] = 1
      g : longueur de s = 5
```

# Opérateurs sur les types

- Conversion explicite (“ casting ” ou transtypage)
- (type) expression

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

```
int main (int argc, char *argv[]) {  
    printf (".2f, ", 3/(double)4);  
    printf ("%d, ", (int)4.5);  
    printf ("%d, ", (int)4.6);  
    printf (".2f, ", (double)5);  
    fputc('\n', stdout);  
  
    return 0;  
}
```

# Opérateur de condition

`condition ? expression_1 : expression_2`

Seul opérateur ternaire du langage

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

int main (int argc, char *argv[]) {
    int x, y, n;
    if (argc != 2) {
        fprintf (stderr, "usage: %s nb\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    x = (n % 2) ? 0 : 1;
    y = (n == 0) ? 43 : (n == -1) ? 52 : 100;
    printf ("x = %d, y = %d\n", x, y);
    return 0;
}
```

# Opérateur virgule

`expr_1, expr_2, ..., expr_n`

- Le résultat est celui de `expr_n`
- `expr_1, expr_2, ..., expr_n-1` sont évaluées, mais leurs résultats oubliés (sauf si effet de bord)

```
#include <stdio.h>
int main (int argc, char *argv[]) {
    int a, b, i, j, t[20];
    for (i = 0, j = 19; i < j; i++, j--) {
        t[i] = j;
        t[j] = i;
    }
    printf ("%d\n", (a = 1, b = 2));
    printf ("%d\n", (a = 1, 2));
    return 0;
}
```



# Opérateurs rangés par ordre de priorité décroissante

Types	Symboles	Associativité
postfixé	<code>()</code> , <code>[]</code> , <code>.</code> , <code>-&gt;</code> , <code>++</code> , <code>--</code>	G à D
unaire	<code>&amp;</code> , <code>*</code> , <code>+</code> , <code>-</code> , <code>~</code> , <code>!</code> , <code>++</code> , <code>--</code> , <code>sizeof</code>	D à G
casting	<code>(type)</code>	D à G
multiplicatif	<code>*</code> , <code>/</code> , <code>%</code>	G à D
additif	<code>+</code> , <code>-</code>	G à D
décalage	<code>&lt;&lt;</code> , <code>&gt;&gt;</code>	G à D
relationnel	<code>&lt;</code> , <code>&lt;=</code> , <code>&gt;</code> , <code>&gt;=</code>	G à D
(in)égalité	<code>==</code> , <code>!=</code>	G à D
et bit à bit	<code>&amp;</code>	G à D
ou ex bit à bit	<code>^</code>	G à D
ou bit à bit	<code> </code>	G à D
et logique	<code>&amp;&amp;</code>	G à D
ou logique	<code>  </code>	G à D
condition	<code>?</code>	D à G
affectation	<code>=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code> , <code>*=</code> , <code>-=</code> , <code>&lt;&lt;=</code> , <code>&gt;&gt;=</code> , <code>&amp;=</code> , <code>^=</code> , <code> =</code>	D à G
virgule	<code>,</code>	G à D