Fin du cours introduction au langage C

Construction d'une boucle

initialisation des variables de boucle

tantque (non condition d'arrêt)

faire instruction de boucle.

↳ résolution du problème évolue vers la condition d'arrêt

Bon usage des boucles en C

while → boucle générale

for → compteur for(i=0; i<=n; i++)

do while → quand on est sûr d'exécuter au moins une fois les actions de boucle

Choix : lisible et claire pour rapport au pb.

L'instruction break permet de sortir d'une boucle.

En LO21, on évitera d'utiliser cette instruction afin de garder une bonne lisibilité des programmes.

Équivalence entre les boucles

```
for (A-expr1 ; expr 2 ; A-expr-3)
    instructions
```

⇔

```
A-expr1;
while (expr 2) {
    instructions
    A-expr 3;
}
```

```
do
    instructions
while (expression);
```

⇔

```
instructions
while (expression)
    instructions;
```

6.5. L'instruction switch()

L'instruction switch est une instruction de choix multiple qui permet d'effectuer un aiguillage direct vers les actions (instructions) en fonction d'un cas.

```
switch (expression)      {

        case constante 1 : instruction 1
        case constante 2 : instruction 2
        case ……..
        case constante n : instruction n
        default : instruction défaut
        }
```

Principe : On évalue l'expression, si l'expression est égale à la constante 1 on exécute l'instruction 1, si l'expression est égale à la constante 1 on exécute l'instruction 1, si l'expression est égale à la constante 2 on exécute l'instruction 2 et ainsi de suite, on exécute l'instruction n.
Si l'on ne veut pas tester les autres cas il faut impérativement mettre un break dans l'instruction.

```
char c ;

printf ("entrez un caractère SVP : ") ;
scanf("%c", &c);
switch (c) {

      case '+': printf ("addition \n ") ; break ;
      case '-' :  printf ("soustraction \n ") ; break ;
      case '*' : printf (" multiplication  \n ") ; break ;
      case '/': printf   (" division \n ") ; break ;
      default : printf (" opération inconnue\n ") ;
}
```

……..

```
if (expression == constante1)
        instruction 1
else if (expression == constante2)
        instruction 2
else if (expression == constante 3)
        instruction 3

        ---

    if (expression == constante n)
            instruction n
    else
            instruction défaut
```

```
case const1 : case const2 :
        instruction 1 et 2
case const3 :
        instruction 3
```

7. Les fonctions d'entrée-sortie.

échanger des informations entre le programme et les périphériques (clavier, écran, sons, disque dur, ...)

clavier → entrée ; écran → sortie

fonctions définies dans <stdio.h>

printf( )
scanf( ) — formatte les E/S

%d
%f
%c
%s

scanf (convertit les caractères en un type spécifié par le format (%).
printf (et inversement

les E/S standard : lecture et écriture de caractère

lecture getchar()

```
char c;
c = getchar();   scanf("%c", &c);
```

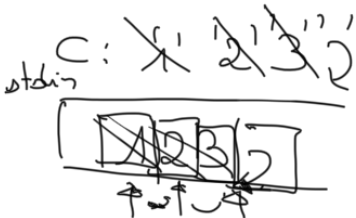
fonction sans paramètre (void) — retourne la valeur du caractère lu à la position courante de l'entrée standard (stdin ↔ clavier)

écriture putchar()

```
char c;
c='a';
putchar(c);
putchar('\n');
putchar('!');
```

écrit le caractère passé en paramètre sur la sortie standard (stdout : écran)

```
printf("entrer une suite de caractères");
while ((c = getchar()) != '\n')
{
....
```



Chapitre 3

Types, opérateurs, expressions

1. Quelques compléments sur les types simples

char — caractère

int, long, short — entier

↓

unsigned int
unsigned long
unsigned short

float, double — réel

2. Quelques compléments sur les évaluations des expressions

vues en TD

points de vigilance

- / (entière ou réelle)
- logique ≠ 0 : vrai
  = 0 : faux.
- le résultat d'une affectation : valeur affectée
- l'évaluation d'une expression s'arrête dès que le résultat est connu.

```
x = 0;
if (x == 0)
    vrai → 1
if (x == 10)
    faux → 0
```

ex : if (A && B && C)

A faux : 0 — B et C ne seront pas évalués.

3. Conversion dans les expressions

Il y a conversion de type lorsqu'une expression fait intervenir des éléments de types différents. En C elles sont largement permises.

Conversions implicites

Expression.

Lorsqu'il y a mélange de types dans une expression, les conversions implicites se font en 2 étapes:

1) Les types entiers plus petits que *int* sont convertis en *int*, le type *float* est converti en *double*.
2) Si une des opérandes est *double*, l'autre est converti en *double* et le résultat est un *double*. Sinon, si une des opérandes est *long*, l'autre est converti en *long* et le résultat est *long*.

Exemple :

```
int i,j;
long l;
float f;

          i     *     (   l     +     f   )
1)                          long        double
2)                          double      double
resultat                    double
1)              int                     double
2)              double                  double
resultat              double
```

Affectation : *destination = source ;*

Cas 1 : types entiers

- Si source est plus importante que destination ⇒ perte d'informations
  Ex : *int = long ;*
  char = short|int|long ;
  short= int|long ;

- Si destination + importante que source ⇒ ok

Cas 2 : types entiers et réels

- destination entiere, source réelle ⇒ décimales sont supprimées (troncation)
  Ex : *char|short|int|long=float|double ;*

```
int x ;
x = 3.89 ;
// x vaut 3
```

- destination réelle, source entière ⇒ ok

Ex : *float|double =char|short|int|long;*

Cas 3 : simple précision / double précision

- destination simple précision, source double ⇒ perte d'information (précision)
  Ex : *float = double ;*
- destination double précision, source simple ⇒ ok

Conversions explicites :

On peut forcer explicitement le type d'une variable : casting
(type) expression ou type (expression)

(int) 3.5 => le résultat est 3
(float) 5 => 5.0

int (3.5) => le résultat est 3
float (5) => 5.0

→ plus lisible
→ par obligation

int x = 3;
(float) x
3.0

int x, y;
x = 10;
y = 4;
x / y
2

x / y.
impossible

2.5
(float) x ((float) y
float 10.
float 4.0

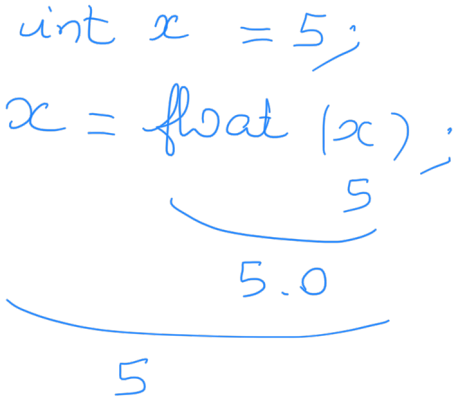
Exemple :

```
int  x, y;
float  f;

x = 23;
y = 5;
f = x/y;    /* x/y vaut 4 et f vaut 4.0 */

f = (float)x /y;      /* (float) x vaut 23.0
                         x vaut toujours 23
                         y vaut 5
                         le resultat sera un
                         double 4.6
                         converti en float
                         f = 4.6 */

f = x /float(y) ;     /* f = 4.6 * /

f = (float)x /(float)y ;  /* f = 4.6 * /

/* interdit     f= x./y.   car x et y sont des
identificateurs et pas des constantes    */
```

Utilisé lorsque des fonctions attendent des arguments de type donné.

z = sin ((double) x) ;

On peut aussi forcer le type pointeur.



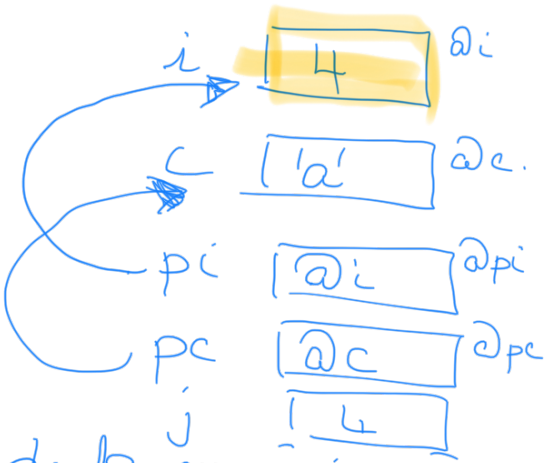
4. Operateurs d'indirection (Pointeurs)

unaire
- `&` = adresse d'une variable
- `*` = contenu de la mémoire à l'adresse donnée

on peut declarer des variables de type pointeur c'est à "adresse de"

- un pointeur: variable qui contient une adresse
- les pointeurs sont typés : adresse de variable d'un type donné
- declaration d'un pointeur

```
int *pi; /* pi est un pointeur d'entier */
char *pc; /* pc " " de caractère */

int i,j; /* i,j un entier */
char c,d; /* c,d un charactère */

i = 4;
pi = &i;
pc = &c;
c = 'a';
j = *pi;
```



4.

le contenu de la mémoire à l'adresse dans pi
le contenu de la mémoire pointé pas pi

```
int* pi;
int *pi;
```