Instructions de contrôle et tableaux

Retour sur les types

Que vaut l'expression suivante ?

```
( (unsigned int) 1000 < -1 ) == 0
```

Affichages dans le terminal

Fonction `printf()` de la librairie `stdio`

```
printf( chaine_de_caractere);
```

Affichages dans le terminal

Fonction `printf()` de la librairie `stdio`

```
printf( chaine_de_caractere);
```

Pour afficher la valeur d'une variable entière :

```
printf("valeur de l'entier x: %d \n", x);
```

Affichages dans le terminal

Fonction `printf()` de la librairie `stdio`

```
printf( chaine_de_caractere);
```

Pour afficher la valeur d'une variable entière :

```
printf("valeur de l'entier x: %d \n", x);
```

Remplacer `%d` par :

- `%f` pour le type double,
- `%c` pour le type char.

Instruction-expression

Il s'agit d'une expression suivie d'un **point virgule** :

```
expression;
```

Instruction-expression

Il s'agit d'une expression suivie d'un **point virgule** :

```
expression;
```

L'expression est évaluée et sa valeur est ignorée.

Instruction-expression

Il s'agit d'une expression suivie d'un **point virgule** :

```
expression;
```

L'expression est évaluée et sa valeur est ignorée.

Cela n'a de sens que si l'expression réalise un **effet de bord**, comme l'affectation.

Instruction-expression

Il s'agit d'une expression suivie d'un **point virgule** :

```
expression;
```

L'expression est évaluée et sa valeur est ignorée.

Cela n'a de sens que si l'expression réalise un **effet de bord**, comme l'affectation.

Il est valide d'écrire

```
i+1;
```

(mais pas très utile...)

Bloc d’instructions

Les instructions peuvent être regroupées dans des blocs.

```
{
   liste de déclarations
   liste d’instructions
}
```

Bloc d'instructions

Les instructions peuvent être regroupées dans des blocs.

```
{
   liste de déclarations
   liste d'instructions
}
```

Les instructions dans le bloc peuvent être des instruction-expressions ou des bloc d'instructions.

Bloc d'instructions

Les instructions peuvent être regroupées dans des blocs.

```
{
   liste de déclarations
   liste d'instructions
}
```

Les instructions dans le bloc peuvent être des instruction-expressions ou des bloc d'instructions.

Les variables déclarées dans un bloc ne sont visibles qu'à l'intérieur, et leur identifiant est prioritaire par rapport aux variables externes.

Bloc d'instructions

```
int a = 4;

{
  int a;
  a = 5;
}

printf("%d\n", a);
```

Combien vaut `a` ?

Branchement conditionnel

```
if( expression ) instruction1
if( expression ) instruction1 else instruction2
```

Compare la valeur de `expression` à 0.
`expression` ne contient pas nécessairement d'opérateur de comparaison.

Branchement conditionnel

```
if( expression ) instruction1
if( expression ) instruction1 else instruction2
```

Compare la valeur de `expression` à 0.
`expression` ne contient pas nécessairement d'opérateur de comparaison.

```
if( expression1 ) if( expression2 ) instr1 else instr2
```

équivalent à

```
if( expression1 ){ if( expression2 ) instr1 else instr2 }
```

IF avec plus de 2 alternatives

```
if(n < 0){
  instruction1;
}
else if(n == 0){
  instruction2;
}
else{
  instruction3;
}
```

Deux programmes identiques ?

```
if(! n%2){
  n = n/2;
}
else{
  n = 3*n+1;
}
```

```
if(n%2 == 0){
  n = n/2;
}
if(n%2 == 1){
  n = 3*n+1;
}
```

Instructions itératives

- `for`
- `while`
- `do`

Instruction FOR

```
for(expr1; expr2; expr3) instruction
```

- `expr1` est évaluée une seule fois **au début**.
- `expr2` est évaluée **avant** chaque exécution de `instruction`. Si sa valeur est non nulle, `instruction` est exécutée, sinon la boucle se termine.
- `expr3` est évaluée **à la fin** de chaque exécution de `instruction`.

Instruction FOR

```
for(expr1; expr2; expr3) instruction
```

- `expr1` est évaluée une seule fois **au début**.
- `expr2` est évaluée **avant** chaque exécution de `instruction`. Si sa valeur est non nulle, `instruction` est exécutée, sinon la boucle se termine.
- `expr3` est évaluée **à la fin** de chaque exécution de `instruction`.

La valeur de `expr1` et `expr3` n'est pas utilisée.
Seul leur effet de bord peut avoir une incidence sur le programme.

Instruction FOR

```
for(expr1; expr2; expr3) instruction
```

- `expr1` est évaluée une seule fois **au début**.
- `expr2` est évaluée **avant** chaque exécution de `instruction`. Si sa valeur est non nulle, `instruction` est exécutée, sinon la boucle se termine.
- `expr3` est évaluée **à la fin** de chaque exécution de `instruction`.

La valeur de `expr1` et `expr3` n'est pas utilisée.
Seul leur effet de bord peut avoir une incidence sur le programme.

Les expressions peuvent être vide.
`for(;;);` boucle infinie.

Instruction FOR

Usage courant (`N` entier >0)

```
for(i=0; i<N; i++){
   instruction;
}
```

Combien de fois est exécutée `instruction` ?

Que vaut `i` quand la boucle se termine ?

Instruction FOR

Comment tester plusieurs conditions de continuation ?

Instruction FOR

Comment tester plusieurs conditions de continuation ?

Utiliser l'opérateur `&&` .

Instruction FOR

Comment tester plusieurs conditions de continuation ?

Utiliser l'opérateur `&&` .

Comment effectuer plusieurs instructions de début de FOR et de fin d'instruction ?

Instruction FOR

Comment tester plusieurs conditions de continuation ?

Utiliser l'opérateur `&&` .

Comment effectuer plusieurs instructions de début de FOR et de fin d'instruction ?

Utiliser l'opérateur `,` :

```
expression1 , expression2
```

a la valeur de `expression2` .

Instruction FOR : exemples

```
for(i=0, j=N-1; i<j; i++ ,j--){
    temp = tab[i];
    tab[i] = tab[j];
    tab[j] = temp;
}
```

Instruction FOR : exemples

```
for(i=0, j=N-1; i<j; i++ ,j--){
   temp = tab[i];
   tab[i] = tab[j];
   tab[j] = temp;
}
```

```
for(i=0;  i<4;  printf("*\n"), i++)
   for(j=0; printf(" "), j++, j<4-i; );
```

Instruction WHILE

```
while(expr) instruction
```

`expr` est évaluée **avant** chaque exécution de `instruction`. Si sa valeur est non nulle, `instruction` est exécutée, sinon la boucle se termine.

Instruction WHILE

```
while(expr) instruction
```

`expr` est évaluée **avant** chaque exécution de `instruction`. Si sa valeur est non nulle, `instruction` est exécutée, sinon la boucle se termine.

```
for(expr1; expr2; expr3) instruction
```

équivaut à

```
expr1;
while(expr2){
  instruction;
  expr3;
}
```

Instruction DO

```
do instruction while(expr);
```

`expr` est évaluée **après** chaque exécution de `instruction`. Si sa valeur est non nulle, `instruction` est exécutée, sinon la boucle se termine.

Instruction DO

```
do instruction while(expr);
```

`expr` est évaluée **après** chaque exécution de `instruction`. Si sa valeur est non nulle, `instruction` est exécutée, sinon la boucle se termine.

Utile pour s'assurer que `instruction` est exécutée au moins une fois.

```
do{
   alea = rand();
}
while(alea > 10000);
```

Instruction BREAK

Provoque l'arrêt de la première instruction `for`, `while`, `do` englobante.

```
for (i = 0; i < N; i = i + 1)
   if (t[i] == 0) break;
```

Instruction CONTINUE

Dans une instruction `for`, `while` ou `do`, l'instruction `continue` provoque l'arrêt de l'itération courante, et le passage au début de l'itération suivante.

Instruction SWITCH

```
switch(expr){
  case expr_1 : instr_1; break;
  case expr_2 : instr_2; break;
      ...
  case expr_k : instr_k; break;
  default     : instr;
}
```

Instruction SWITCH

```
switch(expr){
  case expr_1 : instr_1; break;
  case expr_2 : instr_2; break;
      ...
  case expr_k : instr_k; break;
  default     : instr;
}
```

- les valeurs des `expr_i` doivent être connues à la compilation,
- il ne doit pas y avoir deux valeurs égales,
- si la valeur de `expr` n'est égal à aucun des `expr_i`, l'instruction par défaut est exécutée. L'usage de `default` est facultatif.

Instruction SWITCH

c est un caractère.

```
switch(c){
   case '0':
   case '1':
   case '2':
   case '3':
   case '4':
   case '5':
   case '6':
   case '7':
   case '8':
   case '9': nb_chiffres++; break;
   default: nb_non_chiffres++;
}
```

Tableaux

Ensemble d'éléments **du même type**, désignés par un identificateur unique.

A chaque élément est associé un indice.

Tableaux

Ensemble d'éléments **du même type**, désignés par un identificateur unique.

A chaque élément est associé un indice.

Un tableau est un **type dérivé** dont les variables ont :

- un nom
- un type (de base ou dérivé)
- une longueur fixe

Tableaux

Ensemble d'éléments **du même type**, désignés par un identificateur unique.

A chaque élément est associé un indice.

Un tableau est un **type dérivé** dont les variables ont :

- un nom
- un type (de base ou dérivé)
- une longueur fixe

Déclaration :

```
type_éléments nom[ taille_tableau ];
```

Tableaux

Ensemble d'éléments **du même type**, désignés par un identificateur unique.

A chaque élément est associé un indice.

Un tableau est un **type dérivé** dont les variables ont :

- un nom
- un type (de base ou dérivé)
- une longueur fixe

Déclaration :

```
type_éléments nom[ taille_tableau ];
```

En pratique

```
#define N 100
int t[N];
```

Tableaux

Les indices d'un tableau de taille N sont O jusqu'à $N-1$.
`t[i]` contient le $(i+1)-$ème élément.

Tableaux

Les indices d'un tableau de taille N sont O jusqu'à $N-1$.
`t[i]` contient le $(i+1)$−ème élément.

Initialisation d'un tableau d'un type de base :

```
#define N 10
int t[N] = {1, 2};
```

Le premier élément vaut 1, le deuxième vaut 2, les suivants 0.

```
int t[] = {1, 2, 9, 2};
```

Tableau à 4 éléments.

Tableaux

Les indices d'un tableau de taille N sont O jusqu'à $N-1$.
`t[i]` contient le $(i+1)$−ème élément.

Initialisation d'un tableau d'un type de base :

```
#define N 10
int t[N] = {1, 2};
```

Le premier élément vaut 1, le deuxième vaut 2, les suivants 0.

```
int t[] = {1, 2, 9, 2};
```

Tableau à 4 éléments.

Les **tableaux de caractères** peuvent être initialisés par une chaîne de caractère (cf. cours ultérieur).

Tableaux

La **référence** à un élément du tableau est une expression

```
nom_tableau[ expression ]
```

où `expression` doit avoir une valeur entière (l'indice).

Tableaux

La **référence** à un élément du tableau est une expression

```
nom_tableau[ expression ]
```

où `expression` doit avoir une valeur entière (l'indice).

Exemples :

```
t[3] = 5;
t[5]++;
t[0]--;
t[i++]++;
t1[ t2[2] ];

for(i=0, compteur=0; i<taille_tableau;
                          compteur += (t[i]%2==0), i++);
```

Recopie tableau

```
#include <stdio.h>
#define N 10 /* Définition d'une constante */

int main(){
    int t1[N], t2[N] = {1,2};
    int i;
    for(i=0; i<N; i++){
        t1[i] = t2[i];
    }
    return 0;
}
```

Tableau à plusieurs dimensions

Tableau de tableaux :

```
int mat[12][10];
```

est un tableau de 10 éléments de type `int mat[12]`.

Tableau à plusieurs dimensions

Tableau de tableaux :

```
int mat[12][10];
```

est un tableau de 10 éléments de type `int mat[12]`.

Le tableau à deux dimensions `int mat[N][M]` “est équivalent” au tableau unidimensionel `int tab[N*M]`.
On a la correspondance `mat[i][j]` = `tab[i*M+j]`.

Tableau à plusieurs dimensions

Tableau de tableaux :

```
int mat[12][10];
```

est un tableau de 10 éléments de type `int mat[12]`.

Le tableau à deux dimensions `int mat[N][M]` “est équivalent” au tableau unidimensionel `int tab[N*M]`.
On a la correspondance `mat[i][j]` = `tab[i*M+j]`.

C’est pourquoi on peut écrire

```
int tableau[][] =
      {{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15}};
```

Ou

```
int tableau[][5] =
      {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
```

Tableau à plusieurs dimensions

Seule la première dimension peut être omise à la déclaration :

```
int tableau[][] =
       {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
```

génère le message d'erreur suivant à la compilation

```
les éléments du tableau sont de type incomplet
```

Maximum dans un tableau

```
int tab[N]; //N est une constante > 0
int maximum;
int i;
for(i=0; i<N; i++)
    tab[i] = rand();
maximum = tab[0];
for(i=1; i<N; i++){
   if(tab[i] > maximum)
       maximum = tab[i];
}
```

Maximum dans un tableau à 2 dimensions

```
int mat[N][M]; //N,M sont des constantes > 0
int maximum;
int i,j;
maximum = mat[0][0];
for(i=0; i<N; i++){
   for(j=0; j<M; j++){
       if(mat[i][j] > maximum)
           maximum = mat[i][j];
   }
}
```

OU avec une seule boucle :

Maximum dans un tableau à 2 dimensions

```
int mat[N][M]; //N,M sont des constantes > 0
int maximum;
int i,j;
maximum = mat[0][0];
for(i=0; i<N; i++){
   for(j=0; j<M; j++){
       if(mat[i][j] > maximum)
           maximum = mat[i][j];
   }
}
```

OU avec une seule boucle :

```
for(i=1; i<N*M; i++){
   if(mat[i/N][i%N] > maximum)
       maximum = mat[i/N][i%N];
}
```

Maximum dans un tableau à 2 dimensions

```
int mat[N][M]; //N,M sont des constantes > 0
int max;
int i;
max = mat[0][0];
for(max = mat[0][0], i=1; i < N*M;
     max = mat[i/N][i%N] > max ? mat[i/N][i%N] : max , i++);
```