



## Premiers exemples de traitements

```
#include <cmath>
#include <iostream>

void main()
{
    float b(0.0);
    float c(0.0);
    float delta(0.0);
    cin >> b >> c;
    delta = b*b - 4*c;
    if (delta < 0.0){
        cout << "Pas de solutions réelles !" << endl;
    }
    else if (delta == 0.0) {
        cout << "Une solution unique: " << -b/2.0 << endl;
    }
    else {
        cout << "Deux solutions: [" << (-b-sqrt(delta))/2.0
        << ", " << (-b+sqrt(delta))/2.0 << ']' << endl;
    }
}
```



## Une instruction simple: l'affectation

➔ L'instruction d'**affectation** consiste à **attribuer** (*affecter*)  
une **valeur** à une **variable**.

En C++, la syntaxe d'une affectation est:

```
<identificateur> = <valeur> ;
```

où *valeur* est une valeur **littéral** ou une **expression** de même type que la variable  
référéncée par *identificateur*.

**Exemple:**

```
i = 3;
```



## Interagir avec l'extérieur: entrées/sorties

Les **instructions d'entrées/sorties** (E/S) permettent les interactions du programme avec son environnement (extérieur).

Les mécanismes d'E/S doivent en particulier donner au programmeur la possibilité:



De prendre en compte, au sein du programme, des données issues de l'extérieur, comme des informations saisies au clavier ou lues dans des fichiers.

⇒ requiert des fonctions spécifiques, appelées **fonctions d'entrées**, permettant d'associer des informations externes aux variables définies dans le programme.



D'afficher des données à l'écran

⇒ requiert des fonctions spécifiques, appelées **fonctions de sorties**, permettant de visualiser des valeurs, en spécifiant éventuellement un format d'affichage.



De sauvegarder les résultats produits pour d'éventuelles réutilisations.

⇒ requiert des fonctions spécifiques, appelées **fonctions de lecture/écriture**, permettant de stocker les données produites par les programmes dans des *fichiers*, et de récupérer ces données par la suite (notion de *persistance* des données).



# Instructions d'entrées/sorties (1)

## Instructions élémentaires pour l'affichage à l'écran:

```
cout << <expression1> << <expression2> << ... << <expressionn>;
```

affiche à l'écran<sup>1</sup> les valeurs des expressions <expression<sub>i</sub>>.



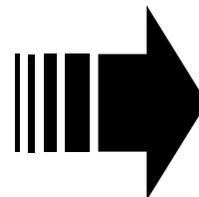
Les chaînes de caractères doivent être indiquées entre guillemets anglais «"», tandis que les caractères simples sont eux indiqués entre apostrophes «'».



Les valeurs affichées ne sont, pas délimitées par défaut (pas d'espaces séparateurs). Les délimiteurs sont donc à indiquer explicitement, par exemple par ' ' ou " ".

### Exemple

```
int a(1);
cout << '[' << -a << ", "
      << a << ']' << end;
```



[-1, 1]



1. En réalité sur la *sortie standard* (c.f. plus tard dans le cours).



## Instructions d'entrées/sorties (2)

### Exemple d'utilisation de cout:

Avec l'initialisation:

```
int i(2);  
float x(3.14);
```

l'instruction:

```
cout << "=> " << 2*i+5 << ", " << x << endl;
```

produira à l'écran l'affichage suivant: => 9, 3.14



Le manipulateur endl représente le *retour de ligne*.  
Il permet donc de «passer à la ligne» suivante lors de  
l'affichage.



## Instructions d'entrées/sorties (3)

Instructions élémentaires pour la lecture au clavier:

```
cin << <var1> << <var2> << ... << <varn>;
```

permet à l'utilisateur de saisir au clavier<sup>2</sup> une liste de valeurs  $val_1, val_2, \dots, val_n$  qui seront stockées dans les variables  $\langle var_i \rangle$ .

### Exemple

Avec l'initialisation:

```
int i;  
double x;
```

l'instruction:

```
cin >> i >> x;
```

permettra de lire au clavier un entier suivi d'un réel, et affectera l'entier lu à la variable  $i$  et le réel à la variable  $x$ .



2. En réalité sur l'entrée standard (c.f. plus tard dans le cours).



## Instructions d'entrées/sorties (4)

### Exemple de programme:

```
#include <iostream>
void main()
{
    int i;
    double x;
    cout << "Valeurs pour i et x: " << flush;
    cin >> i >> x;
    cout << "=> " << i << ", " << x << endl;
}
```



Pour pouvoir utiliser les instructions d'E/S, il faut inclure, en début de programme, le fichier *header* *iostream*, au moyen de la directive:  
`#include <iostream>`



Le manipulateur `flush` permet d'assurer l'affichage de la ligne courante, en l'absence de saut de ligne (il ordonne le vidage du tampon d'affichage)



```
#include <cmath>
#include <iostream>

void main()
{
    float b(0.0);
    float c(0.0);
    float delta(0.0);
    cin >> b >> c;
    delta = b*b - 4*c;
    if (delta < 0.0){
        cout << "Pas de solutions réelles !" << endl;
    }
    else if (delta == 0.0) {
        cout << "Une solution unique: " << -b/2.0 << endl;
    }
    else {
        cout << "Deux solutions: [" << (-b-sqrt(delta))/2.0
            << ", " << (-b+sqrt(delta))/2.0 << ']' << endl;
    }
}
```



## Opérateurs et expressions (1)

➔ En plus des instructions, tout langage de programmation fournit des **opérateurs** permettant de manipuler les éléments prédéfinis. Les définitions des opérateurs sont souvent étroitement associées au type des éléments sur lesquels ils sont capables d'opérer.

Ainsi, les *opérateurs arithmétiques* (+, -, \*, /, ...) sont définis pour les types numériques (entiers et réels par exemple<sup>3</sup>), et les *opérateurs logiques* (!, &&, ||) sont définis eux (en particulier) pour les types booléens.

Les **expressions** sont des séquences bien formées combinant (éventuellement à l'aide de parenthèses) des opérateurs et des arguments (qui peuvent être des valeurs littérales, des variables ou des expressions).

Par exemple:  $(2 * (13 - 3) / (1 + 4))$  est une expression numérique bien formée, alors que  $) 3 + ) 5 * (-2$  ne l'est pas.

3. En fait, et on le verra par la suite, il est également possible de définir les opérateurs arithmétiques pour des types non-élémentaires, comme par exemple les nombres complexes.



## Opérateurs et expressions (2)

Les *opérateurs arithmétiques* sont (dans leur ordre de priorité d'évaluation):

*	multiplication
/	division
%	modulo
+	addition
-	soustraction

▼ Le *modulo* est le reste de la division entière.

▼ Ces opérateurs sont tous *associatifs à gauche*

Du fait que les opérateurs ont non seulement une *syntaxe*, mais aussi une *définition opératoire*, une expression peut être **évaluée**, c'est-à-dire associée à la *valeur littérale* correspondant au *résultat* de l'expression.

Ainsi, l'évaluation de l'expression  $(2 * (13 - 3) / (1 + 4))$  correspond à la valeur littérale de type entier 4.





### Exemple d'utilisation d'expressions pour l'affectation:

- $Z = (x+3) \% y;$
- $Z = (3*x+y)/10;$
- $Z = (x = 3*y)+2;$

### Remarque:

C++ (tout comme C) fournit un certain nombre de **notations abrégées** pour des affectations particulières:

<i>Affectation</i>	<i>Notation abrégée</i>
$x = x + y$ idem pour $\{-, *, /, \%$	$x += y$
$x = x + 1$ idem pour $\{-$	$++ x$

4. Ou  $x++$ , la différence entre les deux notations sera expliquée dans les mini-références.



### Remarques à propos de l'opérateur de division en C++:

⇒ Si a **et** b sont *entiers*,  $a/b$  est le quotient de la division entière de a par b.

Exemple:  $5/2 = 2$

et  $a\%b$  est le reste de cette division.

Exemple:  $5 \% 2 = 1$



donc, si a et b sont *entiers*, on a:

$$(a/b) * b + (a\%b) = a$$

⇒ Si a **et/ou** b est *réel*,  $a/b$  est le résultat de la division de a par b.

Exemple:  $5.0/2 = 2.5$



Au sens strict, une **expression logique** (ou *condition*) est une **expression booléenne** (i.e. de type *booléen*), c'est-à-dire une expression qui peut avoir comme résultat :  
soit la valeur «vraie» (`true`) – dans ce cas on dit que la condition est *vérifiée* –  
soit la valeur «fausse» (`false`) – et dans ce cas, la condition n'est *pas vérifiée*.



En fait, la définition du langage C++ permet de considérer comme expression logique une expression de n'importe quel type, avec la convention suivante:  
si l'évaluation de l'expression est une *valeur nulle*, elle sera considérée comme une condition **fausse**, sinon elle sera considérée comme une condition **vraie**.

Les valeurs nulles sont<sup>5</sup>:

- les zéros numériques: 0 et 0.
- la valeur booléenne `false`

## Exemples d'expressions simples:

- `true`, 1, (2+2)  
sont des expressions évaluées comme *vraies*.
- `false`, (16%2), 0.0  
sont des expressions évaluées comme *fausses*.

5. Il existe une valeur nulle supplémentaire: la valeur *void*; nous la verrons plus tard dans le cours, lorsque nous aborderons la notion de pointeurs.



# Opérateurs de comparaison

Des conditions logiques plus complexes peuvent être construites, à l'aide d'opérateurs spécifiques: les **opérateurs de comparaison** et les **opérateurs logiques**.

Les **opérateurs de comparaisons** (également appelés *opérateurs relationnels*) sont (dans leur ordre de priorité d'évaluation): .

Opérateur	Opération
<	strictement inférieur
<=	inférieur ou égal
>	strictement supérieur
>=	supérieur ou égal
==	égalité
!=	différence (non-égalité)

**Exemples de conditions correspondants à des expressions relationnelles:**

```
(x >= y)
(x+y == 4)
((x > y) == (y > z))
```



# Opérateurs logiques (1)

Les **opérateurs logiques** permettent de combiner plusieurs conditions entre-elles.

<i>Opérateur</i>	<i>Opération</i>
!	NON logique (négation)
&&	ET logique (conjonction)
	OU logique

▼ Toujours selon leur ordre de priorité d'évaluation.

## Exemples de conditions complexes:

- `((z != 0) && (2*(x-y)/z < 3))`
- ≡ • `((z) && (2*(x-y)/z < 3))`
- `((x >= 0) || ((x*y > 0) && !(y*z > 0)))`

≡  
*indique 2 conditions  
logiquement équivalentes*



# Opérateurs logiques (2)

Les opérateurs logiques ET, OU et NON sont définis par les tables de vérités usuelles:

<b><i>X</i></b>	<b><i>Y</i></b>	<b><i>non X</i></b>	<b><i>x ET y</i></b>	<b><i>x OU y</i></b>
v	v	f	v	v
v	f	f	f	v
f	v	v	f	v
f	f	v	f	f



## Evaluation paresseuse (1)

☞ Les opérateurs logiques `&&` et `||` effectuent une **évaluation paresseuse** (*lazy evaluation*) de leur arguments:

⇒ l'évaluation des arguments se fait de la **gauche vers la droite**, et ne sont évalués que les arguments **strictement nécessaires** à la détermination de la valeur logique de l'expression.

### Plus précisément:

- dans l'expression conditionnelle:  $(X_1 \ \&\& \ X_2 \ \&\& \ \dots \ \&\& \ X_n)$ ,  
l'argument  $X_i$  n'est évalué que si les arguments  $X_j$  ( $j < i$ ) sont tous *vrais*:  
l'évaluation se termine au premier  $X_j$  *faux* rencontré, auquel cas l'expression toute entière est *fausse* (si un tel  $X_j$  n'existe pas, l'expression est naturellement *vraie*)
- dans l'expression conditionnelle:  $(X_1 \ || \ X_2 \ || \ \dots \ || \ X_n)$ ,  
l'argument  $X_i$  n'est évalué que si les arguments  $X_j$  ( $j < i$ ) sont tous *faux*:  
l'évaluation se termine au premier  $X_j$  *vrai* rencontré, auquel cas l'expression toute entière est *vraie* (si un tel  $X_j$  n'existe pas, l'expression est naturellement *fausse*)



La notion d'évaluation paresseuse est très utile pour construire des expressions logiques dans lesquelles l'évaluabilité de certains arguments peut être conditionnée par les arguments précédents dans l'expression.

### Exemple:

```
(x != 0) && 4/x > 3)
```

Si l'évaluation n'était pas paresseuse, la détermination de l'expression conduirait à une erreur lorsque x vaut 0 (division par 0)<sup>6</sup>

```
(x <= 0 || log(x) == 4)
```

Dans ce cas également, une évaluation complète conduirait à une erreur lors du calcul de  $\log(x)$  pour tout x inférieur ou égal à 0.

D'une manière générale, il est toujours bon d'économiser du temps processeur, en ne réalisant pas de calculs inutiles.

6. Notez que dans ce cas particulier, on pourrait contourner l'obstacle en testant l'expression équivalente:

```
((x>0) && (4 > 3*x)) || ((x<0) && (4 < 3*x))
```



```
#include <cmath>
#include <iostream>

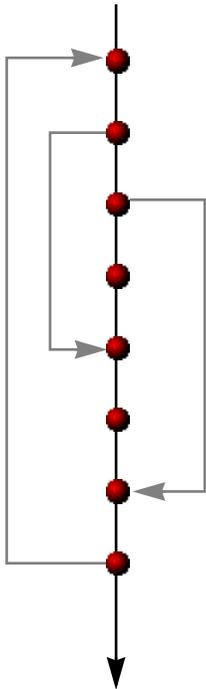
void main()
{
    float b(0.0);
    float c(0.0);
    float delta(0.0);
    cin >> b >> c;
    delta = b*b - 4*c;
    if (delta < 0.0){
        cout << "Pas de solutions réelles !" << endl;
    }
    else if (delta == 0.0) {
        cout << "Une solution unique: " << -b/2.0 << endl;
    }
    else {
        cout << "Deux solutions: [" << (-b-sqrt(delta))/2.0
            << ", " << (-b+sqrt(delta))/2.0 << ']' << endl;
    }
}
```

## Structures de contrôle (2)



☞ Une *structure de contrôle* sert à modifier l'ordre linéaire normal d'exécution (i.e. le *flux d'exécution*) des instructions d'un programme.

Les principales **structures de contrôle** sont:



- Le *branchement* conditionnel: `if`
- La *sélection* à choix multiples: `switch`
- La *boucle*: `while`
- L'*itération*: `for`

Toutes ces structures vont de plus utiliser la notion [importante] de *bloc d'instructions*.



Pour pouvoir être utilisée, les instructions d'un programmes C++ doivent être regroupées en entités<sup>1</sup> appelées **séquences d'instructions** ou **blocs**.

Ces séquences sont explicitement délimitée par les accolades «{» et «}»:

---

```
{
    int tmp(a); // échange du contenu
    a = b;      // de deux variables
    b = tmp;
}
```

---



Par convention, on alignera les accolades l'une sous l'autre<sup>2</sup>, et on indentera la séquence d'instructions correspondant au bloc.

1. Parfois imbriquées les unes dans les autres
2. Il existe cependant de nombreux cas où il paraît judicieux de ne se plier à cette convention. Ainsi, dans la suite du cours elle ne sera parfois pas respectée, pour des raisons de présentation (l'espace disponible sur un transparent étant passablement restreint).



## Branchement conditionnel: `if` (1)

Le **branchement conditionnel** est une structure de contrôle qui permet la *mise en œuvre de traitements variables*, défini par des conditions d'applications spécifiques (i.e. le traitement *d'alternatives*).

La syntaxe générale d'un **branchement conditionnel** est:

```
if ( <condition> ) { <instructions1: si condition vérifiée> }
[ else { <instructions2: si condition non-vérifiée> } ]
```



la notation [ . . . ]  
indique une partie optionnelle

L'expression *<condition>* est tout d'abord évaluée.

Si la condition est vérifiée, la séquence d'instructions *<instructions1:...>* est exécutée; sinon, c'est l'éventuelle séquence alternative *<instructions2:...>*, introduite par le mot réservé `else`, qui sera exécutée.

### Exemple

```
if ( x != 0 ) { cout << 1/x << endl; }
else { cout << "Erreur! (Infinity)" << endl; }
```

## Branchement conditionnel: `if` (2)



Il est également possible d'enchaîner plusieurs conditions:

```

if (<condition 1>)
{
    <instructions 1: si condition 1 vérifiée>
}
else if (<condition 2>)
{
    <instructions 2: si condition 2 vérifiée>
}
...
else if (<condition N>)
{
    <instructions N: si condition N vérifiée>
}
[else
{
    <instructions par défaut>
}]
    
```

- L'expression *<condition 1>* est tout d'abord évaluée; si elle est vérifiée, le bloc *<instructions 1:...>* est exécuté, et les alternatives ne sont pas examinées.
- Sinon (1<sup>ère</sup> condition non vérifiée), c'est l'expression *<condition 2>* qui est évaluée, et ainsi de suite.
- Si aucune des N conditions n'est vérifiée, et que la partie optionnelle `else` est présente, c'est le bloc *<instructions par défaut>* associé qui sera finalement exécuté.



## Sélection à choix multiples: `switch` (1)

La **sélection à choix multiples** permet parfois de remplacer avantageusement un enchaînement de branchements conditionnels, en permettant de sélectionner une instruction en fonction des valeurs possible d'une expression de type entier ou caractère.

La syntaxe générale d'une **sélection à choix multiples** est:

```
switch ( <expression> )  
{  
    case <Constante1> : <instructions1> ;  
    ...  
    case <Constanten> : <instructionsn> ;  
    [ default : <instructions> ; ]  
}
```

L'expression *<expression>* est évaluée, puis comparée successivement à chacune des valeurs *<Constante<sub>i</sub>>*<sup>3</sup> introduites par le mot réservé `case`. En cas d'égalité, le bloc (ou l'instruction) associée est exécutée. Dans le cas où il n'y a aucune *<Constante<sub>i</sub>>* égale à la valeur de l'expression, l'éventuel bloc introduit par le mot réservé `default` est exécuté.

3. Pour spécifier ces valeurs, on ne peut uniquement utiliser des expressions constituées de valeurs littérales ou de constantes (l'expression doit être évaluable lors de la compilation)



## Sélection à choix multiples: `switch` (2)

### Exemple de sélection

```
switch (a+b)
{
    case 0: a = b; // exécuté uniquement lorsque
           break; // (a+b) vaut 0

    case 2:
    case 3: b = a; // lorsque (a+b) vaut 2 ou 3
    case 4:
    case 5: a = 0; // lorsque (a+b) vaut 2,3,4 ou 5
           break;

    default: a = b = 0; // dans tous les autres cas.
}
```



Lorsque l'exécution des instructions d'un `case` est terminée, le contrôle ne passe pas à la fin du `switch`, mais continue l'exécution des instructions suivantes (même si leur valeur associée n'est pas égale à l'expression de sélection). Pour provoquer la terminaison de l'instruction `switch`, il est nécessaire de placer explicitement une instruction **`break`**.



## Boucles: `while` (1)

Les **boucles** permettent la mise en œuvre répétitive d'un traitement, contrôlé *a posteriori* ou *a priori* par une condition de continuation.

La syntaxe générale d'une boucle avec condition de continuation *a posteriori* est:

```
do
{
    <actions>
} while (<condition>);
```

La séquence d'instructions *<actions>* est exécutée **tant que** l'expression de continuation *a posteriori* *<condition>* est vérifiée.

### Exemple

```
do
{
    cout << "valeur de i (>=0): ";
    cin >> i;
    cout << "=> " << i << endl;
}
while (i<0);
```



## Boucles: `while` (2)

La syntaxe générale d'une boucle avec condition de continuation *a priori* est:

```
while ( <condition> )  
{  
    <actions>  
}
```

**Tant que** l'expression de continuation *a priori* `<condition>` est vérifiée, la séquence d'instructions `<actions>` est exécutée.

### Exemple

```
while (x>0)  
{  
    cout << x;  
    x /= 2;  
}
```



## L'itération: `for` (1)

Les itérations permettent l'application itérative d'un traitement, contrôlée par une opération d'initialisation, une condition d'arrêt, et une opération de mise à jour des variables de contrôle de l'itération.

La syntaxe générale d'une **itération** est:

---

```
for ( <initialisation> ; <condition> ; <mise à jour> )
{
    <actions>
}
```

---

Une itération `for` est équivalente à la boucle `while` suivante:

---

```
{
    <initialisation> ;
    while ( <condition> )
    {
        <actions>
        <mise à jour> ;
    }
}
```

---



**Exemple:** affichage du carré des nombres entre 0 et 10

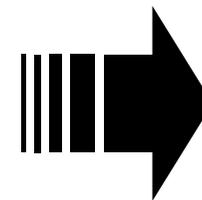
```
for (int i(0); i<=10; ++i)
{
    cout << i*i << endl;
}
```



Il est à noter que la variable de contrôle de la boucle (*i*) est déclarée et initialisée dans la clause `for`.

Lorsque plusieurs instructions d'initialisation ou de mise à jour sont nécessaires, elles sont séparées par des **virgules** (et sont évaluées de la gauche vers la droite)

```
for (int i(0), s(0); i<6; s+=i, ++i)
{
    cout << i << s << endl;
}
```



0	0
1	0
2	1
3	3
4	6
5	10



## Ruptures de séquence (1)

C++ fournit également des instructions de *rupture de séquence*, permettant de contrôler de façon plus fine le déroulement des structures de contrôles.

Parmi ces instructions, on trouve **break** et **continue**.



L'instruction `break`:

Elle ne peut apparaître qu'au sein d'une boucle ou d'une clause de sélection. Elle permet d'interrompre le déroulement de la boucle (quelque soit l'état de la condition de continuation) ou de l'instruction de sélection, en provoquant un saut vers l'instruction suivant la structure de contrôle.



L'instruction `continue`:

Elle ne peut apparaître qu'au sein d'une boucle. Elle interrompt l'exécution des instructions du bloc, et provoque la ré-évaluation de la condition de continuation, afin de déterminer si l'exécution de la boucle doit être poursuivie (avec une nouvelle itération).



# Ruptures de séquence (2)

```
while (condition)
{
    ...
    /* actions de la boucle */
    ...
    break
    ...
    continue
    ...
}
/* actions suivant la
   structure de contrôle
*/
```



### Utilisation de break:

---

```

/*
   une façon de simuler une boucle
   avec condition d'arrêt:
*/
while (true)
{
    /* actions */
    if (<condition d'arrêt>)
    {
        break;
    }
}

```

---

### Utilisation de continue:

---

```

for (int x = 0; x<100; ++x)
{
    /* actions exécutées pour
       tout x entre 0 et 99 incl */
    if (x%2)
    {
        continue;
    }
    /* action exécutées pour
       tout x impair
    */
}

```

---



## Portée: variables locales/globales (1)

Les blocs ont une grande autonomie en C++.  
Ils peuvent contenir leurs propres déclarations (et initialisations) de variables:



les variables déclarées à l'intérieur d'un bloc seront appelées *variables locales* au bloc; elles ne sont accessibles (référençables) qu'à l'intérieur de ce bloc;



les variables déclarées en dehors de tout bloc seront appelées *variables globales* au programme; elles sont accessibles dans l'ensemble du programme.



La portion de code dans laquelle une variable est utilisable est appelée la *portée* de la variable, et s'étend usuellement de la déclaration de la variable jusqu'à la fin du bloc dans lequel est faite la déclaration





# Portée: variables locales/globales (2)

```
// déclarations globales
int z(0);
...
void main()
{
    // déclaration locales
    int x,y;
    {
        // déclaration locales
        int y;
        ... x ...
        ... y ...
        ... z ...
    }
    ... y ...
    ... z ...
}
```

```
// début du programme
int z(0);
...
void main()
{
    // bloc niveau 1
    int x,y;
    ...
    {
        // niveau 2
        int y;
        ... x ...
        ... y ...
        ... z ...
    }
    ... y ...
    ... z ...
}
```



# Portée: variables locales/globales (3)

En particulier, la déclaration d'une variable à l'intérieur d'une boucle est une déclaration **locale** au bloc associé à la structure de boucle.

Ainsi dans la structure de boucle

```
while (<condition> )
{
    int i(0);
    ...
}
```

la variable *i* est **locale** à la boucle

Et de même dans l'itération

```
for (int i(0); i<10; ++i)
{
    int j(0);
    cout << i << j << endl;
}
```

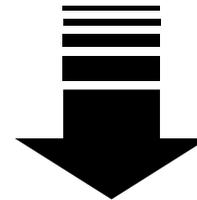
```
{
    int i(0);
    while (i<10)
    {
        int j(0);
        cout << i << j << endl;
        ++i;
    }
}
```



## Portée: variables locales/globales (4)

Exemple: le programme

```
const int MAX(5);  
void main()  
{  
    int i(120);  
    for (int i(1); i<MAX; ++i)  
    {  
        cout << i << endl;  
    }  
    cout << i << endl;  
}
```



donne en sortie:

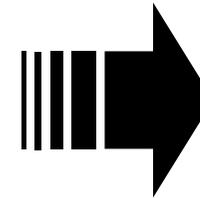
```
1  
2  
3  
4  
120
```



Autre exemple: le programme

```
const int MAX(5);  
void main()  
{  
    int i(120);  
    int j(0);  
    while (j<MAX)  
    {  
        int i(0);  
        ++j;  
        cout << j << ',' << i << endl;  
        i=j;  
    }  
    cout << j << ',' << i << endl;  
}
```

donne en sortie:



1	,0
2	,0
3	,0
4	,0
5	,0
5	,120