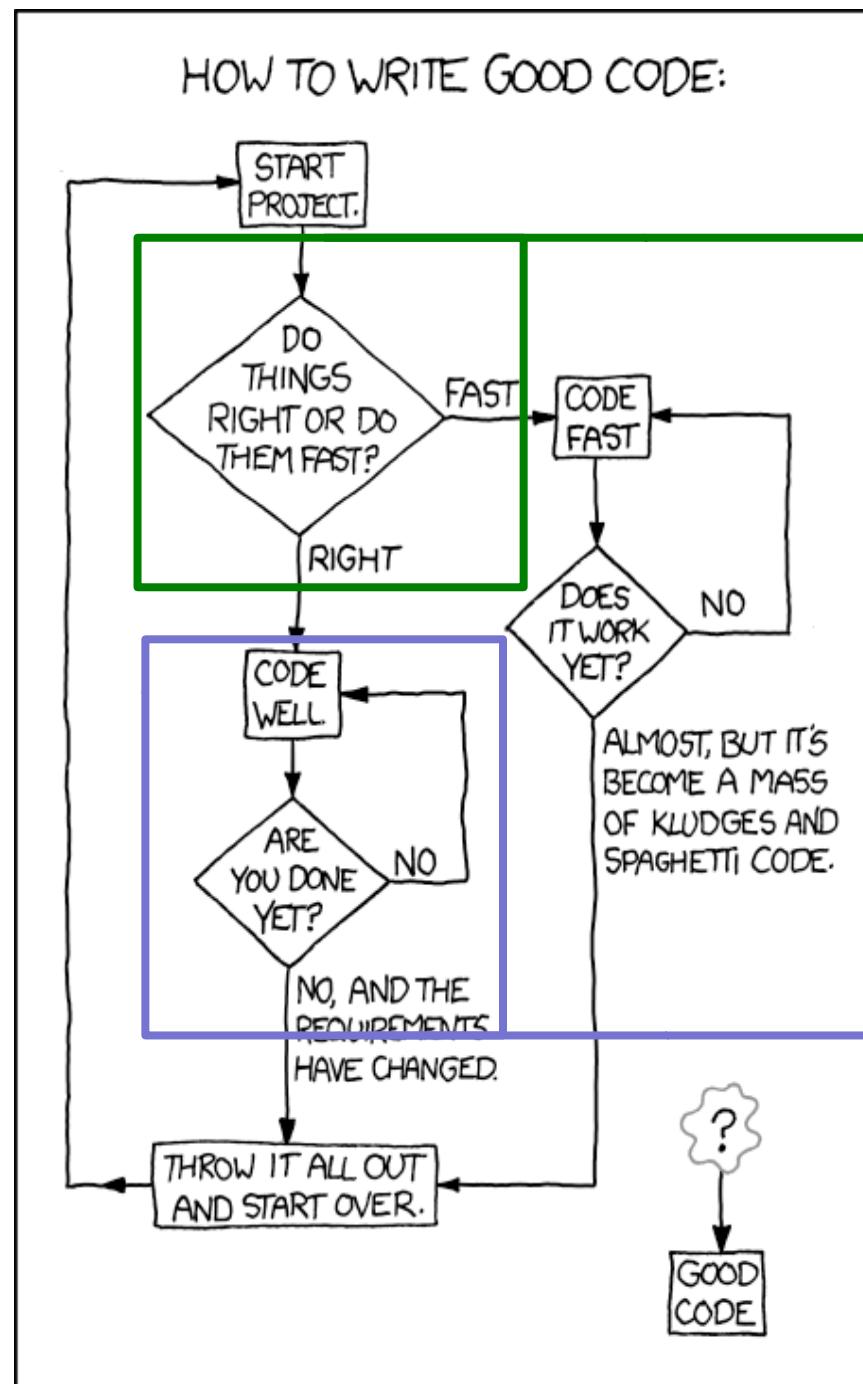
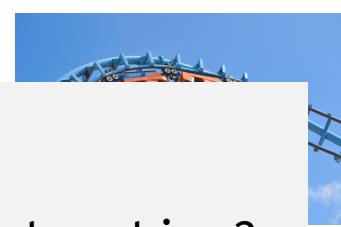


A photograph of a roller coaster track against a clear blue sky. The track is blue with orange supports and cars. The cars are orange with black seats and are filled with passengers. The track curves upwards and to the right.

# Chapitre 3 : Décisions et boucles



// 3 syntaxes de décisions en C++

```
if (condition) instruction1; else instruction2;  
  
variable = condition ? valeur1 : valeur2;  
  
switch (valeur_enumerable) {  
    case constante1 : instruction1; break;  
    case constante2 : instruction2;  
        instruction3; break;  
    default : instruction4; break;  
}
```

// 3 syntaxes de boucles en C++

```
while (condition) instruction;  
  
for (initialisation; condition; action)  
    instruction;  
  
do instruction; while (condition);
```



- On peut grouper plusieurs instructions dans un bloc pour former une **instruction composée**
- Il suffit de les entourer **d'accolades** { }
- Il est inutile de faire suivre un bloc par ;
- Un bloc peut contenir ses propres **déclarations** de variables
- Une variable déclarée dans un bloc n'est **visible** que depuis l'intérieur de ce bloc

```
{ // début du bloc  
  
    int maValeur;  
  
    instruction_1;  
    instruction_2;  
    ...  
    instruction_n;  
  
} // fin du bloc
```



- Vu de l'extérieur, un bloc fonctionne comme une seule instruction
- Un bloc peut donc **inclure** d'autres blocs
- L'**indentation** des blocs, ignorée par le compilateur, aide l'humain qui lit le code
- Si des variables ont le même nom dans des blocs imbriqués, seule **la variable la plus imbriquée est visible**, elle masque les autres

*Note : Même masquée par une variable locale, la variable globale reste toujours visible sous le nom ::a*

```
int a = 1;

int main() {
    cout << a; // 1
    int a = 2;
    cout << a; // 2
{
    int a = 3;
    cout << a; // 3
{
    int a = 4;
    cout << a; // 4
}
    cout << a; // 3
}
cout << a; // 2
cout << ::a; // 1
}
```

# 1. Décisions



- if ... else
  - Syntaxe
  - if imbriqués, recherche d'intervalle
  - C++17: if avec initialisation et if constexpr
- Opérateur ternaire ?:
- Switch
  - Syntaxe
  - Instruction break et fallthrough



# if ... else    si ... sinon



- Instruction de **branchement conditionnel**, qui permet d'exécuter une ou l'autre branche selon que la **condition booléenne** est vraie ou pas
- Syntaxe :

```
if (condition)
    instruction_si_vrai;
else
    instruction_si_faux;
```

```
if (condition) {
    instructions_si_vrai;
} else {
    instructions_si_faux;
}
```

- La condition booléenne doit être entourée de **parenthèses**
- La branche **else** est optionnelle
- La syntaxe avec **accolades** (blocs de code) est plus robuste aux modifications ultérieures du code

# if ... else (exemples)



```
int x; cin >> x;
if (x == 0)
    cout << "Valeur nulle \n";
else
    cout << "Valeur non nulle \n";

int y; cin >> y;
if (cin) { // conversion du flux cin en booléen
    cout << "Valeur correctement lue\n";
} else {
    cout << "Erreur dans le flux d'entrée\n";
    y = 42;
}

if (x < y) {
    cout << "La première valeur est plus petite que la seconde\n";
} // pas de branche else
```

# if imbriqués



- Une condition booléenne ne peut distinguer que 2 alternatives
- Quand il y a **plus de 2 possibilités**, il faut imbriquer d'autres branchements dans les branches du `if ... else`
- 🐚 de style :
  - On utilise les accolades
  - Sauf pour une instruction `if` ou `if ... else` unique dans la branche `else`

```
if (x < 0) {  
    cout << "négatif";  
} else if (x == 0) {  
    cout << "nul";  
} else { // x > 0  
    cout << "positif";  
}
```



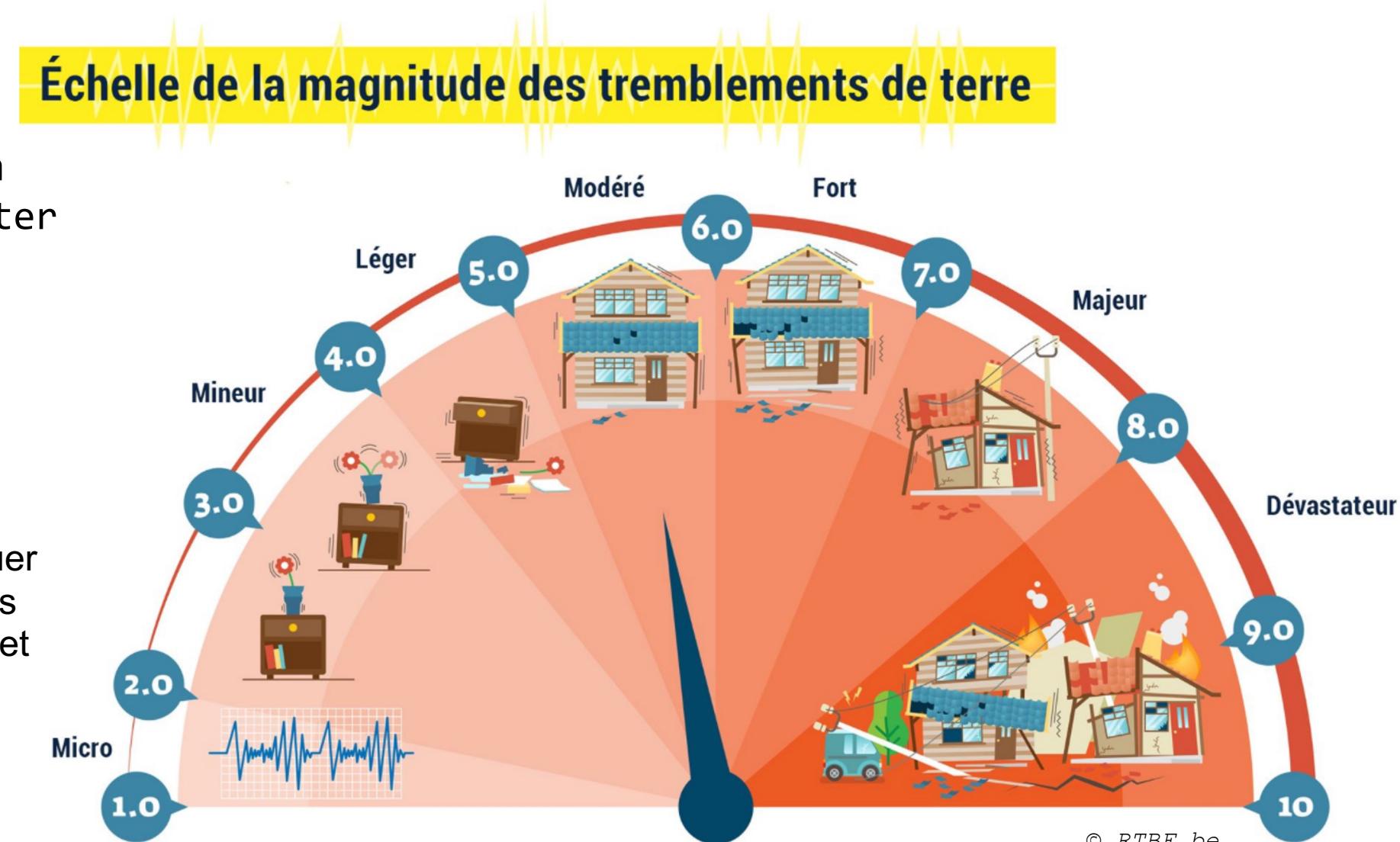
```
if (x < 0) {  
    cout << "négatif";  
} else {  
    if (x == 0) {  
        cout << "nul";  
    } else { // x > 0  
        cout << "positif";  
    }  
}
```





Comment afficher la description d'un tremblement de terre en testant la variable richter de type `double` codant son niveau ?

- Sans erreur
- Sans dupliquer de test
- Y compris sans dupliquer de tests équivalents tels que `(richter < 5.)` et `(richter >= 5.)`

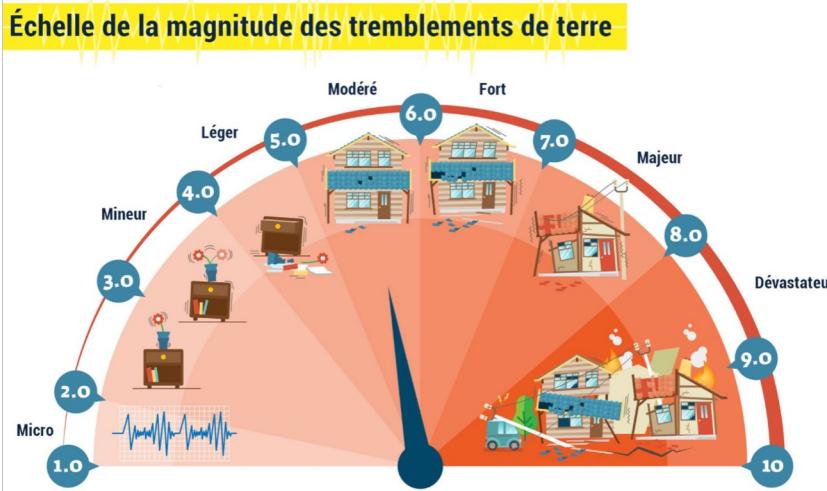


# Recherche d'un intervalle

- Tester **dans l'ordre croissant ou décroissant des valeurs**
- En effectuant **d'abord le test le plus difficile à passer**

```
if (richter < 2.) {
    cout << "micro";
} else if (richter < 4.) {
    cout << "mineur";
} else if (richter < 5.) {
    cout << "léger";
} else if (richter < 6.) {
    cout << "modéré";
} else if (richter < 7.) {
    cout << "fort";
} else if (richter < 8.) {
    cout << "majeur";
} else {
    cout << "dévastateur";
}
```

```
if (richter >= 8.) {
    cout << "dévastateur";
} else if (richter >= 7.) {
    cout << "majeur";
} else if (richter >= 6.) {
    cout << "fort";
} else if (richter >= 5.) {
    cout << "modéré";
} else if (richter >= 4.) {
    cout << "léger";
} else if (richter >= 2.) {
    cout << "mineur";
} else {
    cout << "micro";
}
```





- Depuis C++17, la condition d'une instruction `if` peut également contenir une déclaration.
- La / les variables déclarées n'existent que dans la condition et branches `if` ou `else`

```
string s;
cin >> s;
if (char c = s[0]; c == 'h' or c == 'H') {
    cout << "''" << s << "''"
        << " commence par la lettre H";
} else {
    cout << "''" << s << "''"
        << " commence par la lettre "
        << c << " et pas par H";
}
```



- Depuis C++17, on peut qualifier une instruction `if` de `constexpr`
- La condition doit être évaluable par le `compilateur`
- La branche non sélectionnée n'est pas compilée

```
int grand_nombre;
if constexpr (sizeof(int) >= 4) {
    grand_nombre = 1'000'000'000;
} else {
    grand_nombre = 10'000;
}
```

*Note : Surtout utile aux chapitres 10 et 13 (généricité)*



- Il arrive souvent qu'une **instruction de branchement** serve exclusivement à choisir la valeur d'une expression. Par exemple

```
double pi;  
if (estApproximatif)  
    pi = 3.;  
else  
    pi = 3.141592653589793238;
```

```
if (estFeminin)  
    cout << "Mme";  
else  
    cout << "Mr";
```

- Dans ce cas, il est préférable d'utiliser l'opérateur ternaire, une **expression dont la valeur dépend d'une condition**

```
double pi = estApproximatif ? 3. : 3.141592653589793238;
```

```
cout << ( estFeminin ? "Mme" : "Mr" );
```

# Opérateur ternaire ?:

- Syntaxe :

*condition ? valeur\_si\_vrai : valeur\_si\_faux;*

- Priorité :

- identique aux opérateurs d'affectation (i.e. très faible)

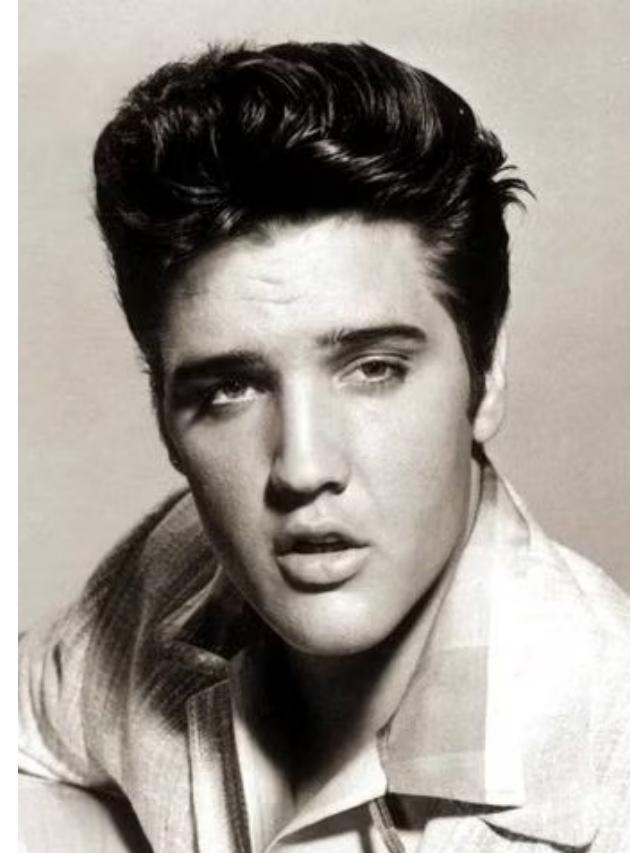
- Type de l'expression déterminé par les 2 valeurs.

- Elles doivent idéalement être de même type

- Sinon, il existe des règles assez complexes pour déterminer si l'expression est valide / quel est son type (*voir chap. 6 pour les types arithmétiques*)

- Par exemple, le code ci-dessous ne compile pas

```
cout << (estFeminin ? "Mme" : 'M' );
```



# switch commutateur

- Lorsque les tests effectués dans une série de `if` imbriqués sont
  - Uniquement des **égalités**
  - Entre une **même expression** et des **constantes**
  - De type **énumérable** (entier ou `enum`)

```
if (i == 0) {  
    cout << "zé";  
    cout << "ro";  
} else if (i == 1) {  
    cout << "un";  
} else {  
    cout << "une autre valeur";  
}
```



- `switch` offre une syntaxe alternative

```
switch (i) {  
    case 0 :  
        cout << "zé";  
        cout << "ro"; break;  
    case 1 :  
        cout << "un"; break;  
    default :  
        cout << "autre valeur"; break;  
}
```



- Syntaxe :

```
switch (condition_enumerable) {  
    case expression_constante1 : instructions_si_1; break;  
    case expression_constante2 : instructions_si_2; break;  
    default : instructions_par_défaut; break;  
}
```

- Le nombre de **case** peut varier. Leur ordre et celui de **default**: n'importe pas
- Les expressions constantes doivent être **toutes différentes**, de type **enumérable** identique au type de la condition (ou convertibles implicitement), **évaluables à la compilation**
- Si la condition n'est égale à aucune des constantes, les instructions suivant **default**: sont exécutées, ou aucune si **default**: est absent



- L'instruction `break` sort de l'instruction `switch`
- En son absence, l'exécution continue au `case` ou `default` suivant

```
switch (i) {  
    case 0 : cout << '0';  
    case 1 : cout << '1'; break;  
    case 2 : cout << '2';  
    default : cout << 'X';  
}
```

i	= affichage
0	01
1	1
2	2X
3	X

- L'option de compilation `-Wimplicit-fallthrough` avertit en cas d'oubli d'un `break`

```
warning: unannotated fall-through between switch labels [-Wimplicit-fallthrough]  
note: insert '[[fallthrough]];' to silence this warning  
note: insert 'break;' to avoid fall-through
```

- L'attribut C++17 `[[fallthrough]]`; supprime l'avertissement pour ce `case`

```
case 0 : cout << '0'; [[fallthrough]];
```

# switch avec initialisation



- Comme pour l'instruction `if`, la condition d'un `switch` peut inclure une déclaration de variable (depuis C++17)

```
unsigned i; cin >> i;

switch (string s = to_string(i); s.size()) {
    case 1 :
        cout << "chiffre";
        break;
    case 2 :
        cout << "nombre entre 10 et 99";
        break;
    default :
        cout << "nombre à 2 chiffres ou plus";
        break;
}
```

# 2. Boucles



- `while(...)` ...
- `for(...; ...; ...)` ...
- `do ... while(...)`
- Suivi manuel
- Boucles imbriquées





- Instruction de **boucle** qui exécute l'instruction qui suit tant que la **condition booléenne** est vraie - aucune fois si la condition de départ est fausse
- Syntaxes :

```
while (condition)
    instruction_répétée_tant_que_vrai;
```

```
while (condition) {
    // bloc d'instructions répété tant que vrai ou saut hors du bloc
}
```

- La condition booléenne doit être entourée de **parenthèses**
- La condition peut aussi être une déclaration de variable
- L'instruction à répéter est le plus souvent un bloc de code

# while (exemples)



```
int i = 4;  
while (i != 0)  
    cout << --i;
```

3210

```
int j = 8;  
while (j > 0) {  
    cout << j;  
    j /= 2;  
}
```

8421

```
int k = 1;  
while (k < 0) {  
    cout << k;  
    ++k;  
}
```

\_\_\_\_\_

```
int n = 42;  
while (int k = n % 10) {  
    cout << k;  
    --n;  
}
```

```
int j = 8;  
while (j >= 0) {  
    cout << j;  
    j /= 2;  
}
```

842100  
000000  
000000  
000000  
000000  
000000  
000000  
000000  
000000  
000000



- Une des boucles les plus fréquentes consiste à **parcourir** un intervalle  $[a, b[$   
Pour cela, il faut
  - **Initialiser** un compteur à la valeur  $a$
  - **Tester** que l'on reste inférieur à  $b$
  - **Incrémenter** le compteur
- La boucle « **for** » permet de **regrouper ces 3 étapes** pour plus de lisibilité

```
int i = a;           // initialisation  
  
while (i < b) { // condition  
  
    instructions; ...  
  
    ++i;           // incrémantation  
}
```

```
for (int i = a; i < b; ++i) {  
  
    instructions; ...  
  
}
```



## ■ Syntaxe :

```
for (initialisation; condition; action)
    instruction_répétée_tant_que_condition_vraie;
```

```
for (initialisation; condition; action) {
    // bloc répété tant que condition vraie ou saut hors du bloc
}
```

## ■ Equivalent à (en l'absence de saut)

```
{ initialisation; while (condition) { instructions; action; } }
```

## ■ Initialisation, condition et action sont optionnels



```
for (int i = 0; i < 4; ++i)  
    cout << i;
```

0123

```
for (int i = 1; i <= 4; ++i)
    cout << i;
```

1234

```
for (int i = 4; i != 0; --i)
    cout << i;
```

4321

```
for (int i = 8; i > 0; i /= 2)  
    cout << i;
```

8421

```
for (int i = 10; i < 0; ++i)  
    cout << i;
```

ANSWER

```
int i = 0;  
for ( ; i < 4; ) {  
    cout << i;  
    ++i;  
}
```

0123

```
int i = 0;  
for (;;) {  
    cout << i;  
}
```



- Parcourir l'intervalle **symétrique**  $[a, b]$ , itérer  $b-a+1$  fois

```
for (int i = a; i <= b; ++i)
```

- Parcourir l'intervalle **asymétrique**  $[a, b[$ , itérer  $b-a$  fois

```
for (int i = a; i < b; ++i)
```

- Pour parcourir les caractères d'une chaîne  $s$ , on utilise typiquement un intervalle asymétrique  $[0, s.length()[$

```
for (int i = 0; i < s.length(); ++i)
    cout << s[i];
```



- Depuis C++11, une boucle `for` peut également parcourir **tous les éléments d'un contenant**
- Exemple : parcours de tous les caractères d'une chaîne `string s("Hello");`

```
for (char c : s) {  
    cout << char(toupper(c));  
}
```

HELLO

- Cette syntaxe est moins source d'erreurs que l'alternative avec des indices

*Note : Nous y reviendrons plus en détail après avoir vu les références, les tableaux, std::vector, ...*



- Il est toujours possible de remplacer une boucle `for` par une boucle `while` et réciproquement. Le choix impacte la **lisibilité** du code.
- On utilise plutôt `for` quand
  - toute la logique d'avancement de la boucle peut être groupée après les instructions à répéter, i.e. dans l'action
  - On connaît a priori le nombre d'itérations qui s'exécuteront
- On utilise plutôt `while` dans les autres cas

# while ou for ? (Exemples)



- Combien d'argent aurais-je en plaçant 10'000 CHF à 5% pendant 8 ans ?

```
double capital = capital_initial;
for (int i = 0; i < duree; ++i)
    capital *= (1 + taux / 100);
cout << capital << " CHF après " << duree << " ans\n";
```

14774.6 CHF  
après 8 ans

- Combien d'année sont nécessaires pour au moins doubler mon investissement de 10'000 CHF en le plaçant à 5% par an ?

```
double capital = capital_initial;
int duree = 0;
while (capital < capital_initial * 2.) {
    capital *= (1 + taux / 100);
    ++duree;
}
cout << "Doublement en " << duree << " ans\n";
```

Doublement  
en 15 ans



- Instruction de **boucle** qui exécute l'instruction **au moins une fois**, puis la répète tant que la condition booléenne est vraie
- Syntaxes :

```
do instruction_répétée; while (condition);
```

```
do { // bloc d'instructions  
} while (condition);
```

- La condition booléenne doit être entourée de **parenthèses**.
- La parenthèse finale est suivie d'un **point-virgule**.

# do ... while (Exemple)



```
int n;
do {
    cout << "Entrez un entier (0 pour sortir): ";
    cin >> n;
    cout << "Vous avez entré " << n << endl;
} while (n != 0);
cout << "Bye" << endl;
```

```
Entrez un entier (0 pour sortir): 3
Vous avez entré 3
Entrez un entier (0 pour sortir): 4
Vous avez entré 4
Entrez un entier (0 pour sortir): 0
Vous avez entré 0
Bye
```



- Pour comprendre un code, il est parfois utile d'en suivre le fonctionnement **pas à pas**
- Considérons le code suivant. Il y a 3 variables, que nous notons dans 3 colonnes
- Les lignes indiquent les itérations

n	sum	digit

```
// somme des chiffres  
  
int n    = 1729;  
int sum = 0;  
while (n > 0) {  
    int digit = n % 10;  
    sum += digit;  
    n /= 10;  
}  
cout << sum << endl;
```



- **n** et **sum** sont initialisées avant d'entrer dans la boucle

<b>n</b>	<b>sum</b>	<b>digit</b>
1729	0	

// somme des chiffres

```
int n    = 1729;
int sum = 0;
while (n > 0) {
    int digit = n % 10;
    sum += digit;
    n /= 10;
}
cout << sum << endl;
```



- **n** est  $> 0$ , on entre dans la boucle
- **digit** prend la valeur  $1729 \% 10 = 9$
- **sum** prend la valeur  $0 + 9 = 9$

<b>n</b>	<b>sum</b>	<b>digit</b>
1729	0	
	9	9

```
// somme des chiffres  
  
int n    = 1729;  
int sum = 0;  
while (n > 0) {  
    int digit = n % 10;  
    sum += digit;  
    n /= 10;  
}  
cout << sum << endl;
```



- Finalement, n prend la valeur  $1729 \div 10 = 172$
- On barre les anciennes valeurs et indique la nouvelle valeur

n	sum	digit
1729	0	
172	9	9

```
// somme des chiffres  
  
int n = 1729;  
int sum = 0;  
while (n > 0) {  
    int digit = n % 10;  
    sum += digit;  
    n /= 10;  
}  
cout << sum << endl;
```



- La condition reste vérifiée,  $172 > 0$
- On effectue **l'itération suivante** en notant les nouvelles valeurs

n	sum	digit
1729	0	
172	9	9
17	11	2

```
// somme des chiffres  
  
int n    = 1729;  
int sum = 0;  
while (n > 0) {  
    int digit = n % 10;  
    sum += digit;  
    n /= 10;  
}  
cout << sum << endl;
```



- On continue les itérations jusqu'à ce que  $n \leq 0$

n	sum	digit
1729	0	
172	9	9
17	11	2
1	18	1
0	19	1

```
// somme des chiffres  
  
int n = 1729;  
int sum = 0;  
while (n > 0) {  
    int digit = n % 10;  
    sum += digit;  
    n /= 10;  
}  
cout << sum << endl;
```

- Le programme affiche donc 19, qui vaut bien  $1 + 7 + 2 + 9$



- Un boucle peut évidemment contenir une autre boucle

```
for (char ligne = 'a'; ligne <= 'd'; ++ligne) {  
    for (int col = 1; col <= 5; ++col) {  
        cout << ligne << col << ' ';  
    }  
    cout << endl;  
}
```

a1	a2	a3	a4	a5
b1	b2	b3	b4	b5
c1	c2	c3	c4	c5
d1	d2	d3	d4	d5

```
for (int y = 0; y < 5; ++y) {  
    for (int x = 0; x <= y; ++x)  
        cout << '*';  
    cout << endl;  
}
```

*
**
***
****
*****



```
char c = 'A';
for (int y = 0; y < 4; ++y) {
    for (int x = 0; x < 16; ++x, ++c)
        cout << c << ' ';
    cout << endl;
}
```

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Q	R	S	T	U	V	W	X	Y	Z	[	\	^	_	`	
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
q	r	s	t	u	v	w	x	y	z	{		~	□	□	

- L'**opérateur virgule** permet de fabriquer une expression en en concaténant deux.
- Il est fréquemment utilisé dans les conditions / actions des boucles **for** ou **while**



- **Syntaxe :**

expression1 , expression2

- **Priorité** : le moins prioritaire de tous. Toujours entre parenthèses dans une expression plus longue
- **Effet** : expression1 est évaluée, son résultat est ignoré. Ensuite, l'expression 2 est évaluée et son résultat donne la valeur de l'expression complète
- **Type** : expression1 et expression2 ne doivent pas nécessairement être de même type. Le type de (expression1 , expression2) est le même que le type de expression2

# Opérateur virgule (Exemples)



```
for (int i = 0, j = 9; i < j; ++i, --j)
    cout << i << " + " << j << " = " << i+j << endl;
```

```
srand(time(nullptr));
int inconnu = rand() % 100;
int nrEssais = 0;
cout << "Devinez un entier < 100 : ";
```

```
while (cin >> i , ++nrEssais, i != inconnu)
    cout << "Non, c'est "
    << ( i < inconnu ? "plus " : "moins")
    << ":";

cout << "Bravo ! Vous avez trouvé en "
    << nrEssais << " essais\n";
```

0 + 9 = 9
1 + 8 = 9
2 + 7 = 9
3 + 6 = 9
4 + 5 = 9

```
Devinez un entier < 100 : 50
Non, c'est plus : 75
Non, c'est moins: 62
Non, c'est moins: 56
Non, c'est moins: 53
Non, c'est plus : 54
Bravo ! Vous avez trouvé en 6 essais
```

## 3. Sauts



- break
- continue
- goto





- En + des décisions et boucles, C++ fournit des instructions de saut
  - **inutiles** : tout code avec saut peut être ré-écrit sans saut
  - **déconseillées** : sauter est le plus souvent moins clair que l'écriture alternative
  - mais **autorisées** et parfois pratiques.
  - à utiliser avec parcimonie
- Elles sont au nombre de trois :
  - **break;** qui sort du switch ou de la boucle la plus imbriquée
  - **continue;** qui passe à l'itération suivante de la boucle la plus imbriquée
  - **goto ETIQUETTE;** qui saute à la ligne commençant par ETIQUETTE :



- Syntaxe :

```
break;
```

- Sort du **switch** ou de la boucle **for**, **while** ou **do ... while** la plus imbriquée
- Ne permet pas de
  - Sortir de boucles imbriquées
  - Sortir d'une boucle depuis un **switch**

# break (Exemple 1)



- Afficher une barrière avec n = 8 sections et 9 poteaux

| == | == | == | == | == | == | == |

- Avec break;

```
for (int i = 0; true ; ++i)
{
    cout << "|";
    if (i == n) break;
    cout << "==" ;
}
```

- Sans break;

```
cout << "|";
for (int i = 0; i != n; ++i)
    cout << "==" << "|";
```

- Evite de dupliquer cout << "|";

# break (Exemple 2)



- Sortir d'une boucle de recherche quand on a trouvé

```
string s = "Hello, world!";

int i;
for (i = 0; i < s.length(); ++i)
    if (s[i] == 'o') break;

if (i != s.length())
    cout << "Le premier o est à l'indice " << i;
else
    cout << "Pas de o";
```

Le premier o est à l'indice 4

- Permet de séparer la logique du parcours (dans la boucle **for**) et celle de la recherche (dans le **if**)

# break (Exemple 3)



- Validation d'une entrée utilisateur

```
int entree;

while (true) {
    cout << "Entrez un entier positif: ";
    cin >> entree;

    if (entree > 0) {
        break; // entree valide
    } else {
        cout << "Entree invalide. Essayez à nouveau" << endl;
    }
}
```



- Syntaxe :

```
continue;
```

- Sort de l'itération courrante de la boucle la plus imbriquée

- Saute à l'évaluation de la **condition** pour une boucle **while** ou **do...while**
  - Saute à l'**action** pour les boucle **for**

- Les codes suivants sont équivalents

```
while (condition_boucle) {  
    instructions1;  
    if (condition_test) continue;  
    instructions2;  
}
```

```
while (condition_boucle) {  
    instructions1;  
    if (not condition_test) {  
        instructions2;  
    }  
}
```

# for vs. while



```
for (int i = 0; i < 6; ++i) {
    cout << ' ' << i;
    if (i%2) continue;
    cout << i;
}
```

00 1 22 3 44 5

```
int i = 0;
while (i < 6) {
    cout << ' ' << i;
    if (i % 2) continue;
    cout << i; ++i;
}
```

00 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

```
int i = 0;
while (i < 6) {
    cout << ' ' << i;
    if (i % 2) { ++i; continue; }
    cout << i; ++i;
}
```

00 1 22 3 44 5



- Pour être complet ... mais **à contrecoeur**

```
goto ETIQUETTE;
```

- Il est nécessaire de **définir une étiquette** dans le code. L'instruction **goto** y sautera

```
ETIQUETTE: instruction;
```

- Elle est considérée comme une instruction de **programmation de bas niveau**, **qui n'a rien à faire dans un code** écrit en langage de haut niveau comme le C++.

Edsger Dijkstra (March 1968).

"Go To Statement Considered Harmful".

*Communications of the ACM.* 11 (3): 147–148. doi:[10.1145/362929.362947](https://doi.org/10.1145/362929.362947)

## Go To Statement Considered Harmful

**Key Words and Phrases:** go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

**CR Categories:** 4.22, 5.23, 5.24

### EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while B repeat A** or **repeat A until B**). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which