



# Chapitre 3

## Structures de contrôle



# Buts du chapitre (conditions)

- Écrire des programmes incluant des **décisions** qui aiguillent vers l'une ou l'autre **branche** de code
- Représenter ces décisions sous forme **d'organigrammes**
- Effectuer des **comparaisons** sur les types vus aux chapitres précédents
- **Combiner** des valeurs **booléennes** pour prendre des décisions plus complexes
- **Combiner** des **décisions** pour effectuer des branchements plus complexes



# Buts du chapitre (boucles et saut)

- Mettre en œuvre des **boucles** **while**, **for** et **do ... while**
- Éviter
  - de rester coincé dans une **boucle infinie**
  - de se **tromper de un** dans le nombre d'itérations
- Choisir l'instruction de boucle **la plus appropriée** à votre problème
- Mettre en œuvre des **boucles imbriquées**
- Apprendre à utiliser (parcimonieusement) les instructions de **saut** **break**, **continue** et **goto**



## Conditions

1. Organigrammes [5-7]
2. Blocs de code [8-12]
3. if..else [13-30]
4. Les conditions [31-39]
5. Opérations logiques [40-46]
6. switch [47-51]
7. enum [52-59]

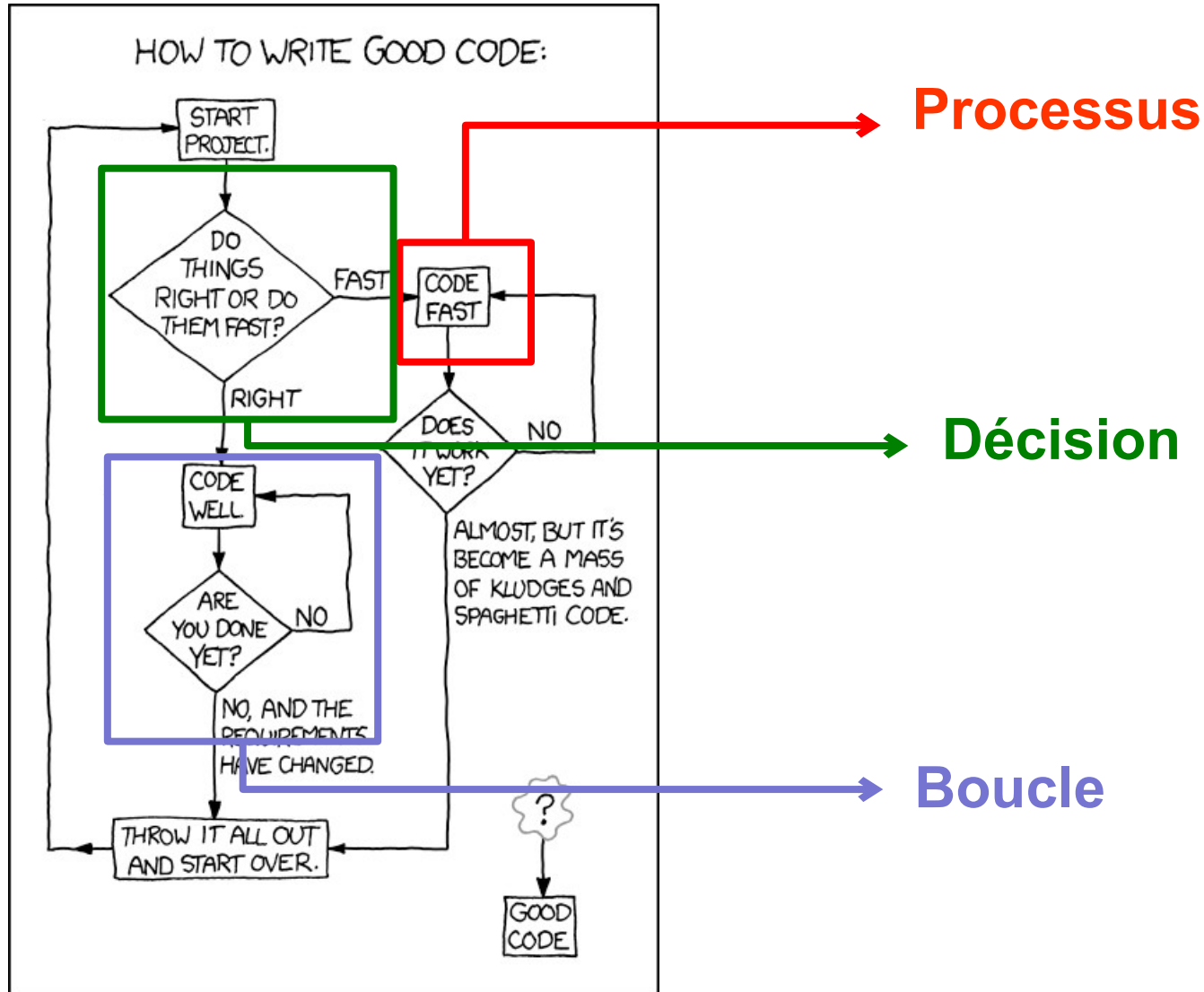
## Boucles

8. while [62-79]
9. for [80-95]
10. do..while [96-105]
11. Boucles imbriquées [106-112]
12. Instructions de saut [113-122]
13. Résumé [123-126]

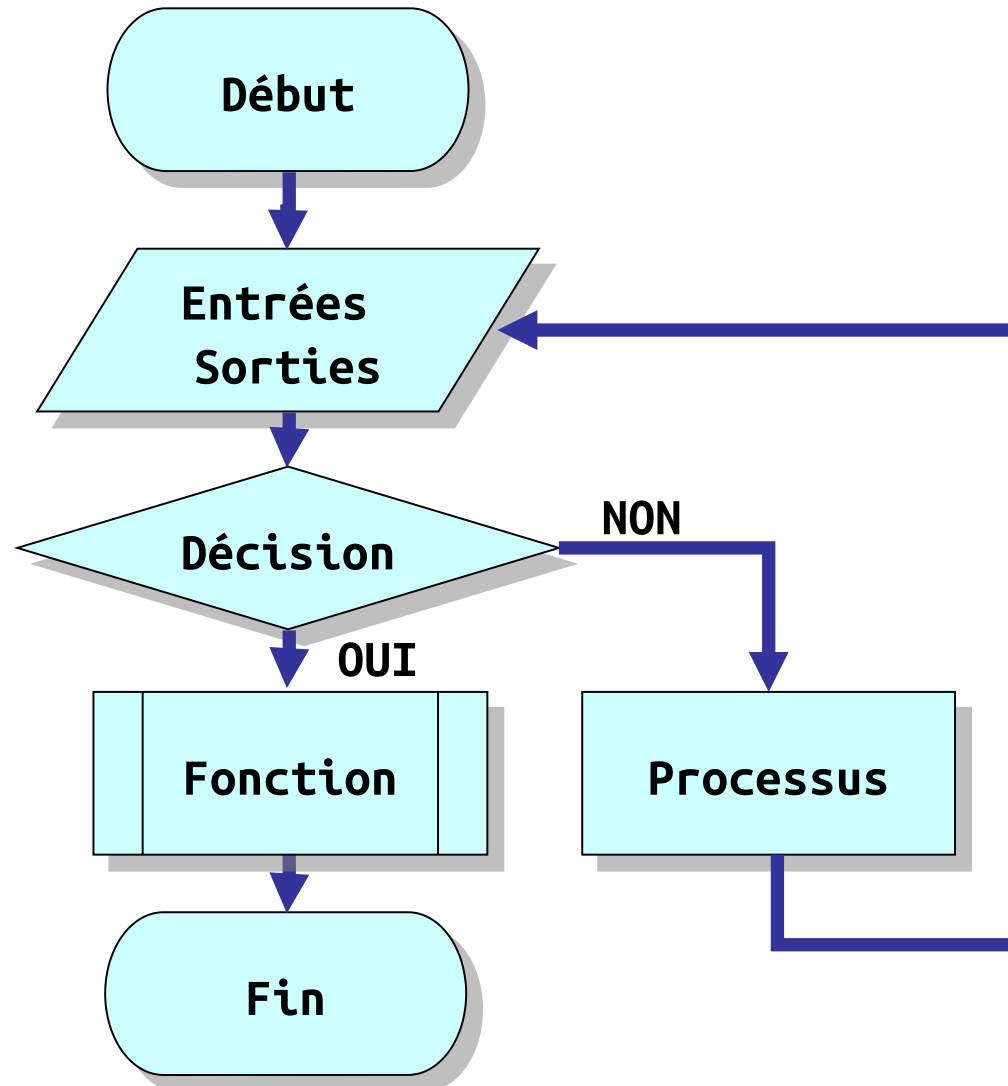


# 1. Organigrammes

# HE<sup>VD</sup> IG Les organigrammes (flowcharts)



# HE<sup>VD</sup> IG Organigrammes, formes courantes





## 2. Blocs de code

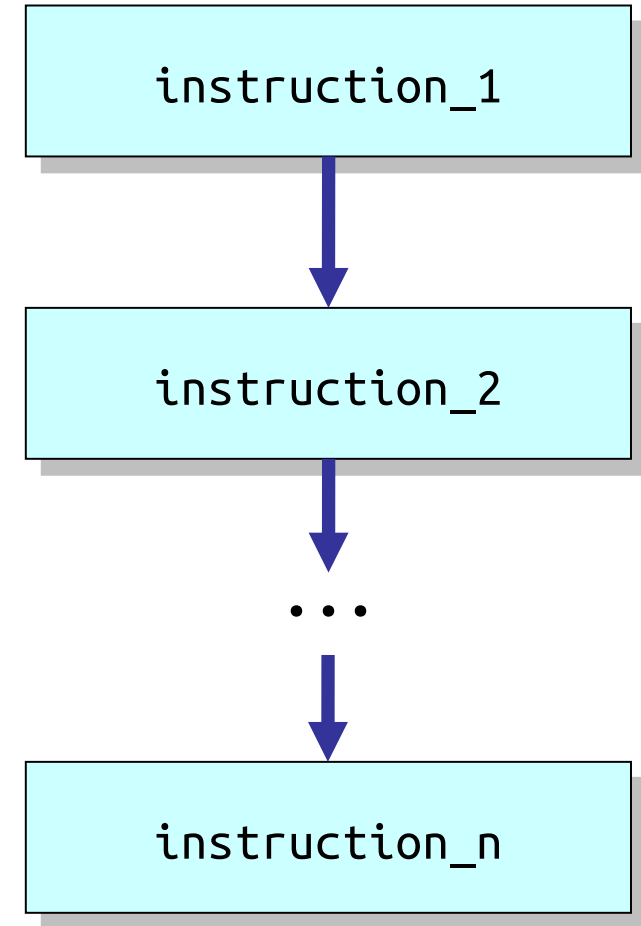


# HE<sup>VD</sup> IG Instructions simples



- Les instructions d'un code se terminent par un ;
- Elles s'enchaînent de manière **séquentielle**

```
instruction_1;  
instruction_2;  
...  
instruction_n;
```



# HE<sup>VD</sup> IG Un bloc de code



- 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
```

# HE<sup>VD</sup> IG Visibilité des variables



Depuis l'intérieur d'un bloc, on voit donc

- Les variables déclarées **dans** le bloc
- Les variables déclarées **avant** le bloc
- Les variables déclarées **hors de tout** bloc, appelées variables globales

```
int globale;

int main() {
    int avant;
    {
        int invisible1;
    }

    {
        int dedans;

        // d'ici, on voit globale,
        // avant et dedans, mais pas
        // invisible1 ni invisible2
    }
    int invisible2;
}
```

# HE<sup>VD</sup> IG Blocs imbriqués



- Vu de l'extérieur, un **bloc fonctionne comme une seule instruction**
- Un bloc peut donc **inclure d'autres blocs**
- On peut éventuellement déclarer des variables de même nom dans des blocs différents. Dans ce cas, seule **la variable la plus imbriquée est visible**, elle masque les autres

```
int a = 1;

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



## 3. if ... else

# HE<sup>VD</sup> IG Un problème d'ascenseur



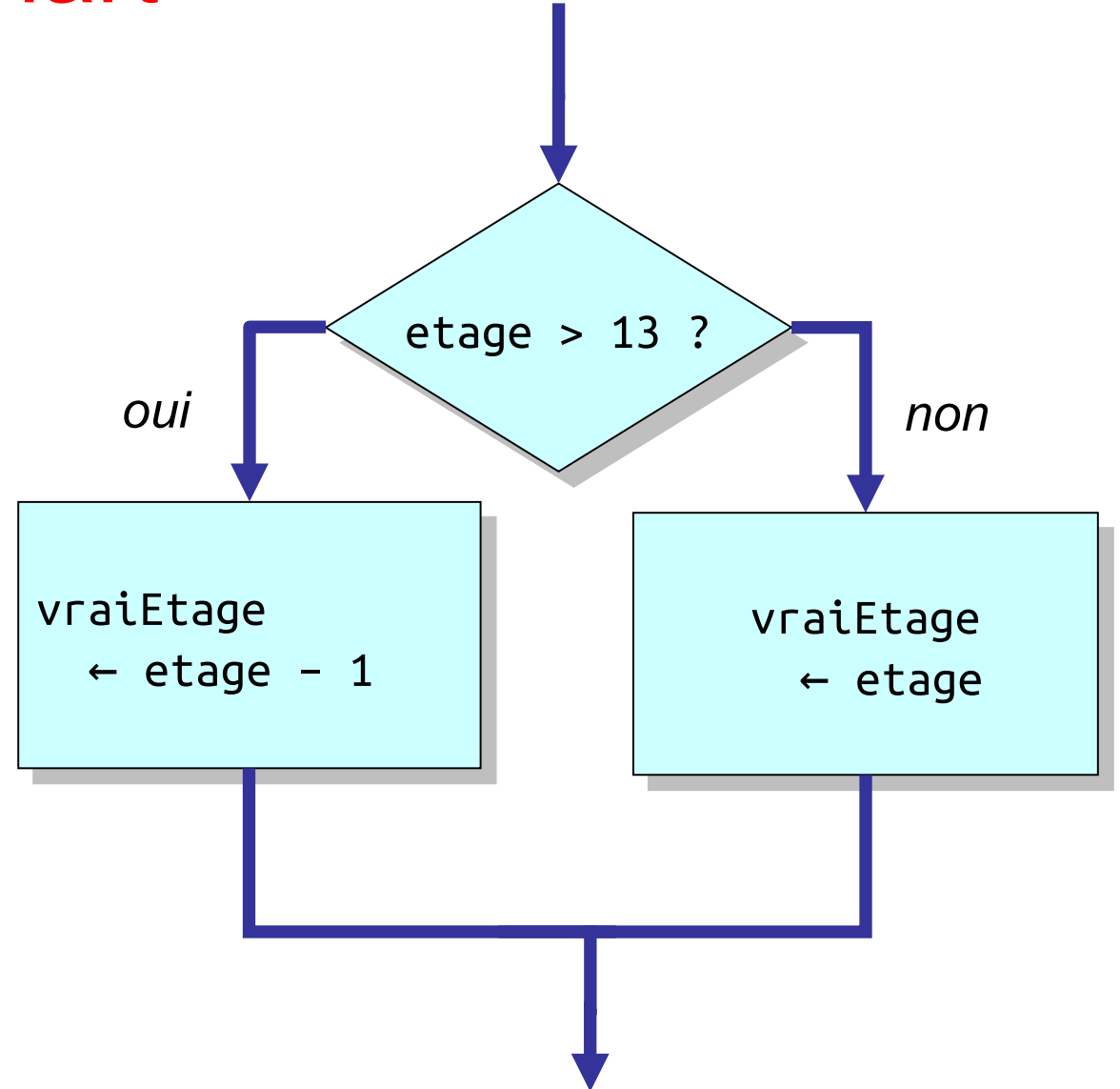
- Il est courant — pour épargner les **triskaïdékaphobiques** — de se passer du nombre 13 dans la numérotation des étages
- Écrivons le code permettant de calculer `vraiEtage`, l'étage réel, à partir de `etage`, demandé par l'utilisateur



# HE<sup>VD</sup> IG Pseudo-code et flowchart



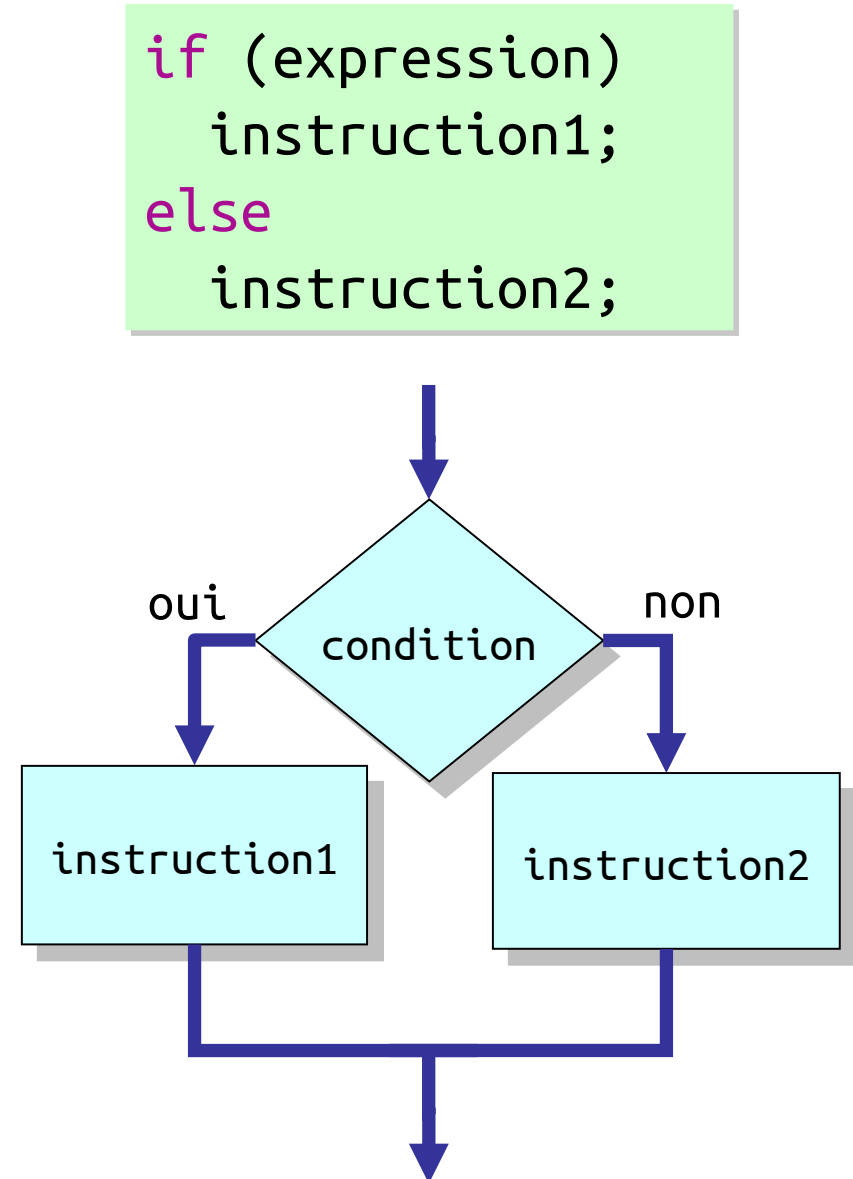
```
si etage > 13  
    vraiEtage ← etage - 1  
sinon  
    vraiEtage ← etage  
fin si
```



# HE<sup>VD</sup> IG if ... else



- Pour le mettre en œuvre, C++ dispose de la paire d'instructions `if else`.
- Notons:
  - La présence de **parenthèses obligatoires** autour de l'expression
  - L'absence de « ; » après la parenthèse fermant l'expression et après le `else` « ; » serait une instruction vide.





# HE<sup>VD</sup> IG if ... else



- Le code calculant le bon étage est donc ...
- Il est recommandé de toujours utiliser des **blocs** de code et pas des instructions simples après **if** et **else**. Cela simplifie notamment les **modifications** ultérieures
- **Commenter** la condition qui mène au **else** rend le code plus lisible

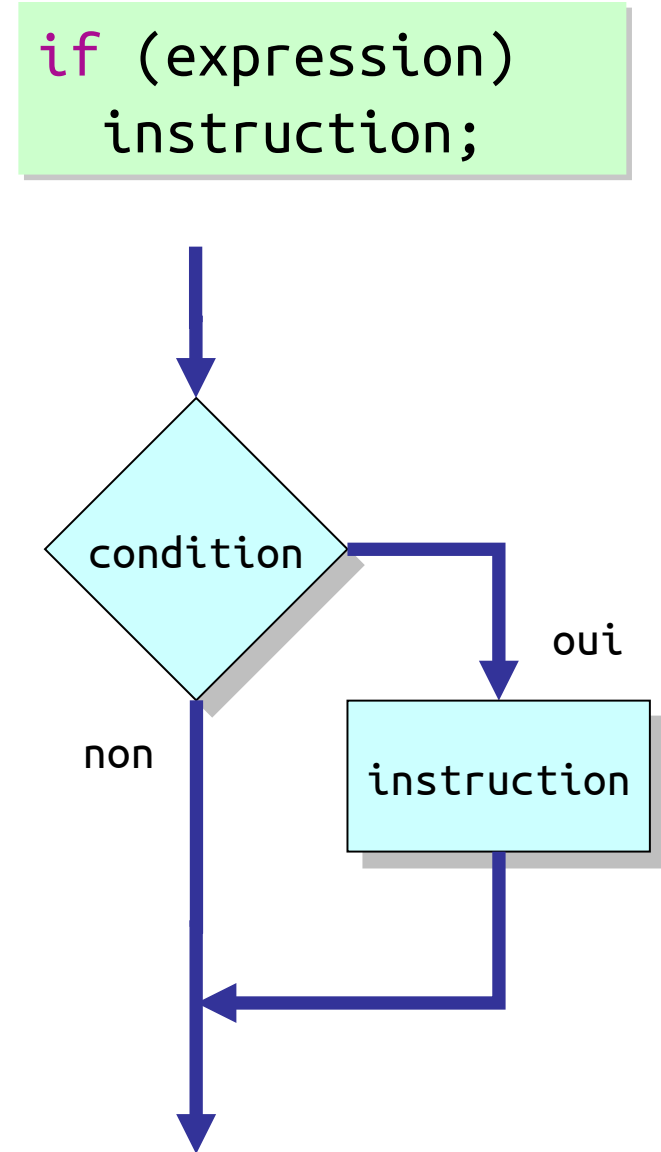
```
if (etage > 13)
    vraiEtage = etage - 1;
else
    vraiEtage = etage;
```

```
if (etage > 13) {
    vraiEtage = etage - 1;
}
else { // etage <= 13
    vraiEtage = etage;
}
```

# HE<sup>VD</sup> IG if seul



- Parfois, il n'y a **rien à faire** dans la branche « **sinon** »
- La partie **else** n'est **pas obligatoire**





- Revenons à notre problème d'ascenseur  
On peut le réécrire différemment

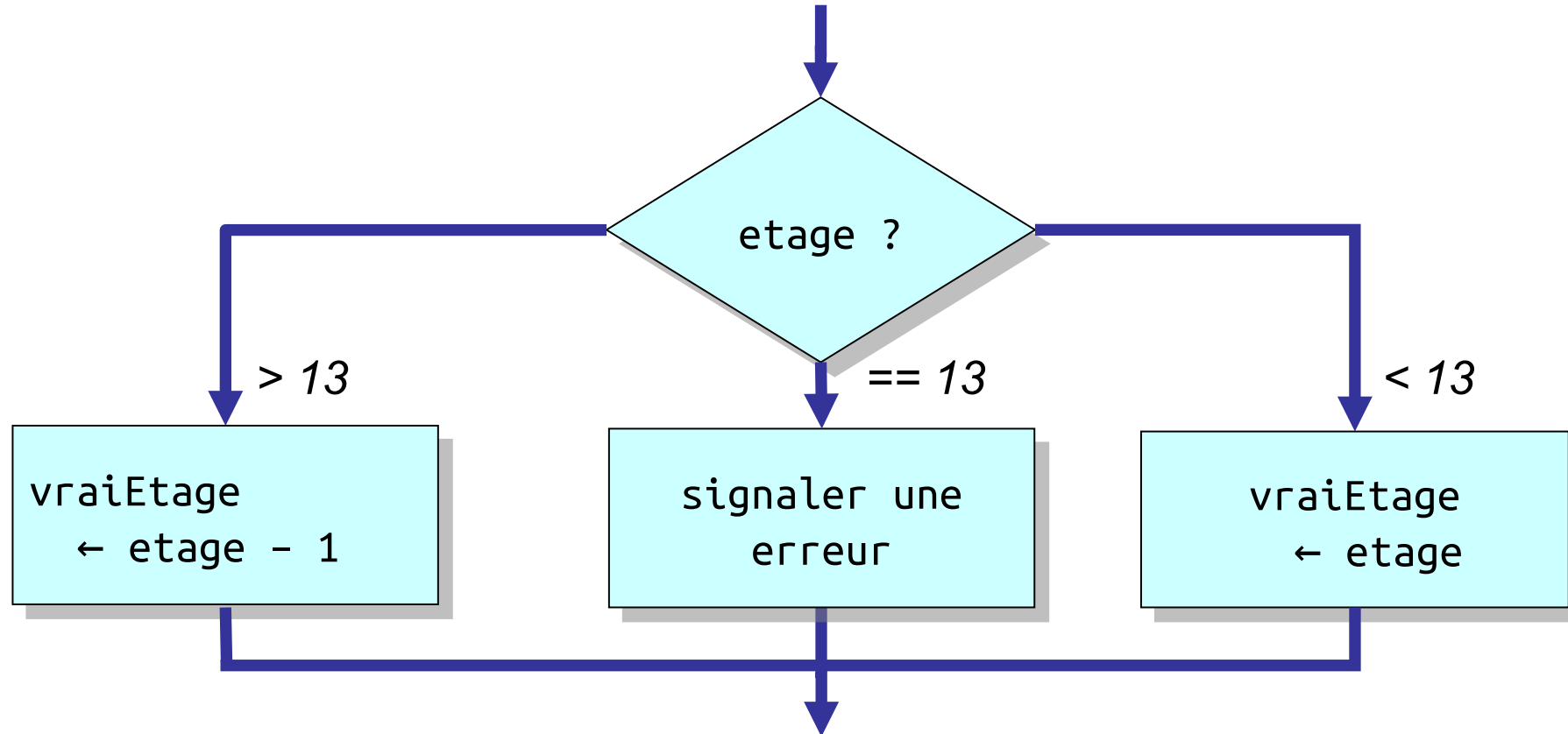
*Il ne faut décrémenter l'étage que s'il est plus grand que 13*

- On peut donc écrire le code en affectant la valeur de vraiEtage avant le test, et en ne la **modifiant que si nécessaire**.

```
vraiEtage = etage;  
if (etage > 13) {  
    --vraiEtage;  
} // pas besoin de else
```



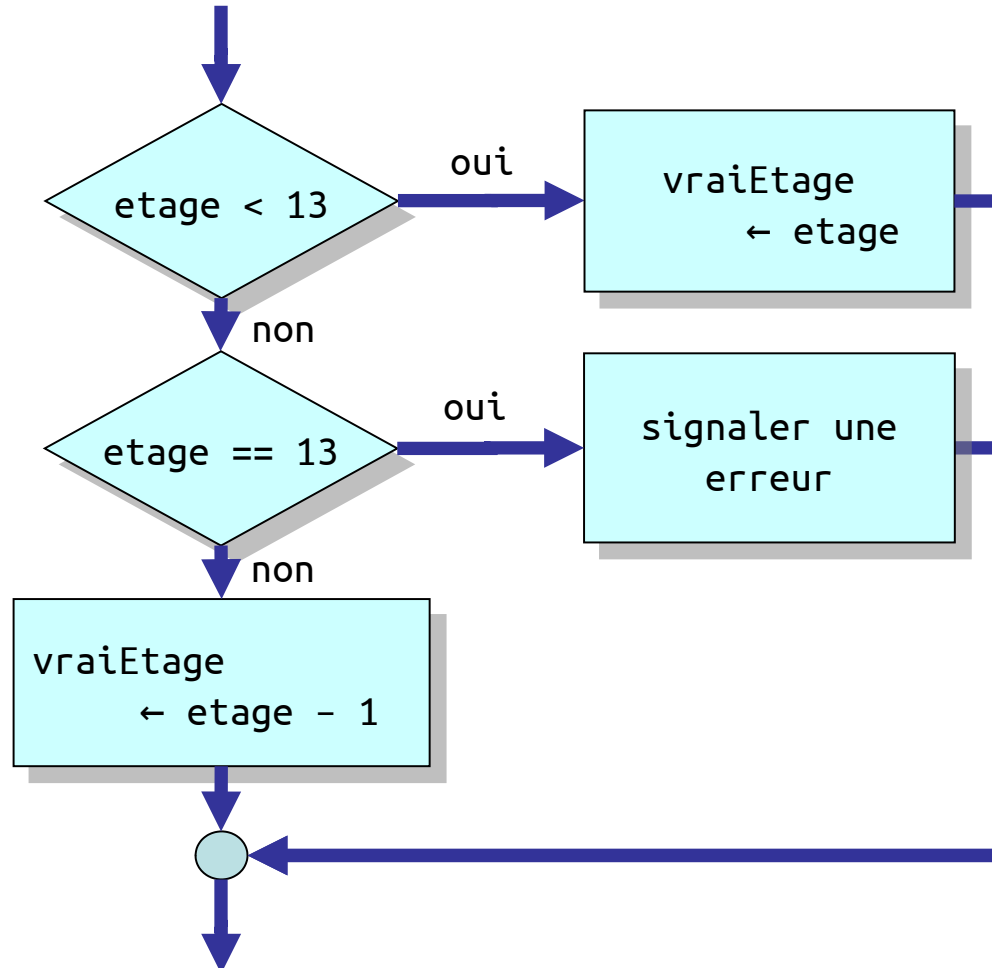
- Comme tout bon programmeur, vous vous méfiez de vos utilisateurs
- Que se passerait-il si la valeur 13 était saisie ?



# HE<sup>VD</sup> IG if imbriqués



- C++ ne fournit pas de choix ternaire
- Il faut faire deux tests imbriqués



```
if (etage < 13) {  
    vraiEtage = etage;  
}  
else if (etage == 13) {  
    cerr << "Erreur"  
         << " etage vaut 13" << endl;  
}  
else { // etage > 13  
    vraiEtage = etage - 1;  
}
```

# HE<sup>VD</sup> IG if imbriqués



- Notons qu'en respectant notre règle de bonne pratique qui ne fait suivre `if` et `else` que par des blocs, on aurait ...
- `else` immédiatement suivi de `if` est l'unique exception à cette bonne pratique

```
if (etage < 13) {  
    vraiEtage = etage;  
}  
else {  
    if (etage == 13) {  
        cerr << "Erreur"  
            " etage vaut 13" << endl;  
    }  
    else { // etage > 13  
        vraiEtage = etage - 1;  
    }  
}
```

# HE<sup>VD</sup> IG if imbriqués



- Par contre, si vous voulez **modifier le code suivant** pour ajouter un test dans la branche du **if**
- On peut penser qu'il suffit d'ajouter le nouveau test comme ici
- **Erreur !** Voici ce que vous avez écrit indenté correctement

La branche **else** ne se rapporte plus au bon test **if**

```
if (etage <= 13)
    vraiEtage = etage;
else // etage > 13
    vraiEtage = etage - 1;
```

```
if (etage <= 13)
    if (etage != 13)
        vraiEtage = etage;
else // etage > 13
    vraiEtage = etage - 1;
```

```
if (etage <= 13)
    if (etage != 13)
        vraiEtage = etage;
else // etage == 13
    vraiEtage = etage - 1;
```

# HE<sup>VD</sup> IG if imbriqués



- Cette erreur ne risque pas d'arriver si vous utilisez **systématiquement des blocs** de code après **if** et **else**

```
if (etage <= 13) {  
    vraiEtage = etage;  
}  
else { // etage > 13  
    vraiEtage = etage - 1;  
}
```

```
if (etage <= 13) {  
    if (etage != 13) {  
        vraiEtage = etage;  
    }  
}  
else { // etage > 13  
    vraiEtage = etage - 1;  
}
```





# Modifions encore notre code...

- On veut afficher la valeur de vraiEtage une fois qu'elle est calculée  
Il est assez naturel de l'écrire ainsi
- **Horreur !!**  
l'instruction cout est **dupliquée**
- Si du code doit **s'exécuter dans toutes les branches**, il faut le sortir du **if else**

```
if (etage <= 13) {  
    vraiEtage = etage;  
    cout << vraiEtage << endl;  
}  
else {  
    vraiEtage = etage - 1;  
    cout << vraiEtage << endl;  
}
```

**Duplication de code !**

```
if (etage <= 13) {  
    vraiEtage = etage;  
}  
else {  
    // etage > 13  
    vraiEtage = etage - 1;  
}  
cout << vraiEtage << endl;
```

# HE<sup>VD</sup> IG Sus au code dupliqué...



- En fait, il reste du code dupliqué...
- On voudrait pouvoir écrire
- Mais ce n'est **pas possible**  
L'instruction **if** n'est **pas une expression**  
Elle ne renvoie pas de valeur

```
if (etage <= 13)  
    vraiEtage = etage;  
else  
    vraiEtage = etage - 1;
```

```
vraiEtage =  
    if (etage <= 13) etage;  
    else etage - 1;
```

# HE<sup>VD</sup> IG Opérateur conditionnel



- C++ permet de faire ce qui précède, mais avec une autre syntaxe  
L'expression

```
condition ? valeur1 : valeur2
```

renvoie **valeur1** si **condition** est vraie, **valeur2** sinon

- On peut donc écrire le code précédent sous la forme

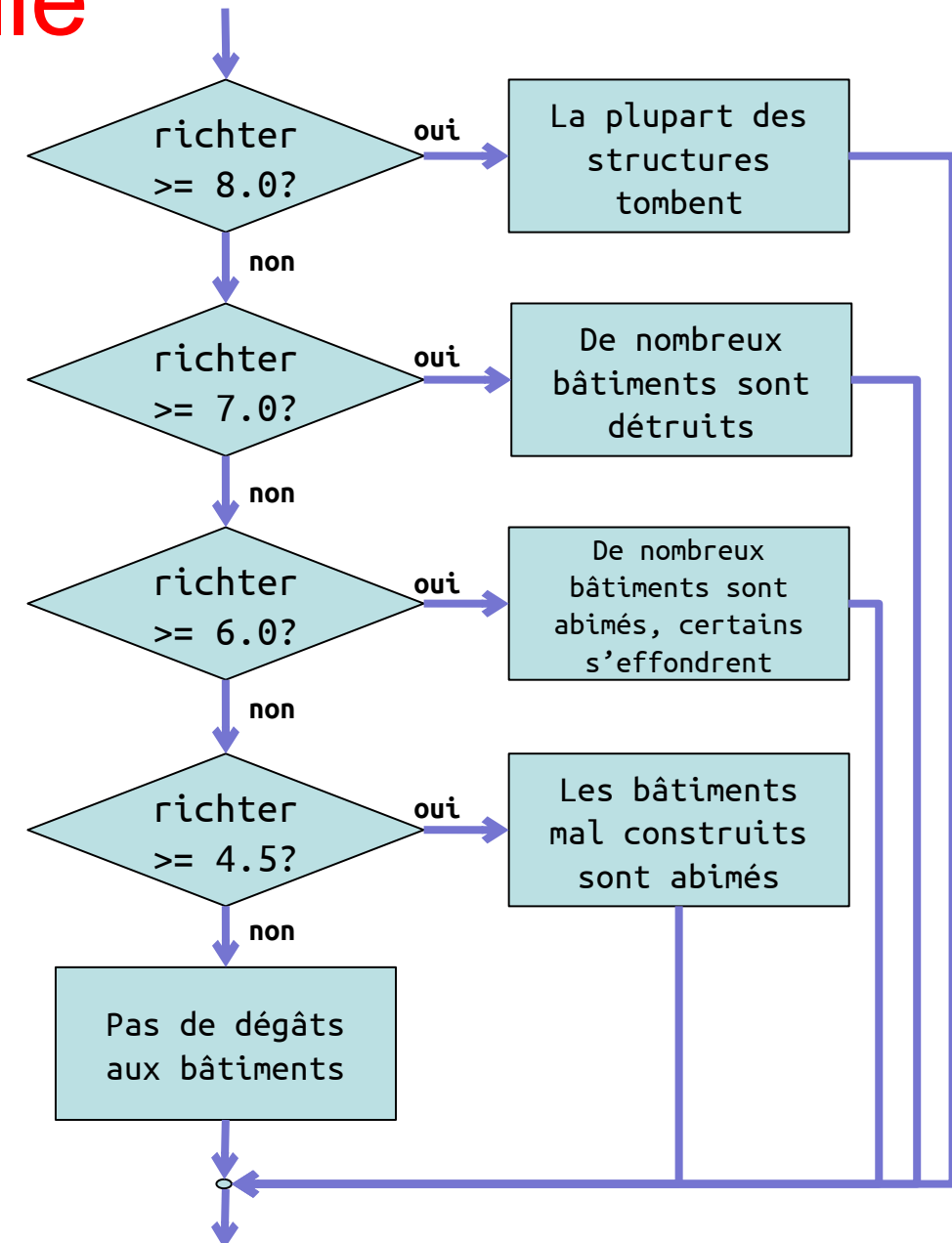
```
vraiEtage = etage <= 13 ? etage : etage - 1;
```

- C'est le **seul opérateur ternaire** du langage

# HE<sup>VD</sup> IG Recherche d'un intervalle



- L'organigramme ci-contre décrit l'effet d'un tremblement de terre selon son classement sur l'échelle de Richter



# HE<sup>VD</sup> IG Recherche d'un intervalle



```
if (richter >= 8.0) {           // intervalle [ 8.0 , ∞ ]
    cout << "La plupart des structures tombent";
}
else if (richter >= 7.0) {      // intervalle [ 7.0 , 8.0 [
    cout << "De nombreux batiments sont detruits";
}
else if (richter >= 6.0) {      // intervalle [ 6.0 , 7.0 [
    cout << "De nombreux batiments sont abimes, certains s'effondrent";
}
else if (richter >= 4.5) {      // intervalle [ 4.5 , 6.0 [
    cout << "Les batiments fragiles sont abimes";
}
else {                          // intervalle [ - ∞ , 4.5 [
    cout << "Pas de degats aux batiments";
}
```

# HE<sup>VD</sup> IG Recherche d'un intervalle



- Dès que l'un des 4 tests `if` réussit, le bloc qui suit est exécuté et aucun des tests suivants n'est testé
- Si aucun test ne réussit, le bloc `else` final est exécuté
- Cet ordre d'exécution nous oblige à faire très attention à l'ordre des tests : du plus strict au moins strict

```
if (richter >= 4.5) {           // mauvais ordre !!  
    cout << "Les batiments fragiles sont abimes";  
} else if (richter >= 6.0) {  
    cout << "De nombreux batiments sont abimes";  
} else if (richter >= 7.0) {  
    cout << "De nombreux batiments sont detruits";  
} else if (richter >= 8.0) {  
    cout << "La plupart des structures tombent";  
}
```



## 4. Les conditions



- Les **expressions** utilisées par les conditions sont soit **vraies**, soit **fausses**
- C++ représente ces **valeurs booléennes** avec le type **bool**, qui peut prendre les valeurs **true** ou **false**
- C'est en fait un type numérique qui peut prendre les valeurs
  - 0 pour **false**
  - 1 pour **true**
- Toute valeur numérique non nulle correspond à **true**



# HE<sup>VD</sup> IG Affichage des booléens



- Par défaut, afficher un booléen avec cout affiche 0 ou 1
- Ce comportement peut être modifié avec les modificateurs de flux boolalpha et noboolalpha

```
bool vrai = true, faux = false;

cout << vrai << " " << faux << endl; // 1 0
cout << boolalpha;
cout << vrai << " " << faux << endl; // true false
cout << noboolalpha;
cout << vrai << " " << faux << endl; // 1 0
```

# HE<sup>VD</sup> IG Opérateurs de comparaison



- C++ fournit 6 opérateurs de comparaison
  - < plus petit que
  - > plus grand que
  - <= plus petit ou égal
  - >= plus grand ou égal
  - == égal (ne pas confondre avec l'affectation =)
  - != différent de
- Ils permettent de comparer des
  - entiers
  - réels
  - chaînes de caractères (string)
- Ils renvoient des valeurs de type `bool`



# Attention à ne pas confondre = et ==

- Le code suivant est **syntactiquement correct**, mais ne fait pas ce que le programmeur voulait !

```
int etage;  
  
etage == 11; // comparaison dont on n'utilise  
            // pas la valeur retournée  
  
if (etage = 13) {  
    // affectation qui renvoie l'entier 13, ce  
    // qui correspond à la valeur booléenne true  
    cout << etage << endl;  
}
```

- Le compilateur émettra des avertissements, selon les options choisies



# Attention aux types non signés

- Pour une variable `n` de type `unsigned`,
  - `n < 0` est toujours `false`
  - `n >= 0` est toujours `true`

```
unsigned int n = -1;

if (n < 0) {
    cout << "Erreur: nombre negatif" << endl;
}
else {
    cout << n << endl;
}
```

4294967295



# Attention à l'égalité (==) avec les réels

- Tester l'égalité de deux nombres réels est délicat à cause de la **limite de précision** de leur représentation

```
cout << boolalpha;  
cout << 1.0 / 3.0 << endl;           // 0.333333  
cout << (1.0 / 3.0 == 0.333333) << endl; // false
```

- Plutôt que d'utiliser uniquement l'opérateur ==, on **compare leur différence** à une très petite valeur qui dépend du volume et du type des calculs effectués pour obtenir les valeurs à comparer

```
const double EPSILON = numeric_limits<double>::epsilon(); // par exemple  
a == b or fabs((a-b))/(fabs(a)+fabs(b)) < EPSILON
```



- Vous verrez souvent

```
if (n != 0) {  
    cout << n << " est non nul" << endl;  
}
```

remplacé par

```
if (n) {  
    cout << n << " est non nul" << endl;  
}
```

- Ceci est **correct** au vu de la règle de conversion des types numériques vers le type **bool**



# Comparer des chaînes de caractères

- La comparaison de chaînes (string) fonctionne comme l'ordre d'un dictionnaire
  - On compare le premier caractère des 2 chaînes
  - S'il est identique, on compare le second
  - S'il est identique, on compare le troisième
  - ...
  - Jusqu'à ce qu'un caractère diffère ou qu'on arrive au bout d'au moins une des chaînes.
- Une fois le premier caractère différent atteint, on compare leurs codes ASCII, et donc
  - ' ' < '0' < '9' < 'A' < 'Z' < 'a' < 'z'
- Si au contraire on atteint la fin d'une des chaînes, la plus courte est plus petite que la plus longue



## 5. Opérateurs logiques



# HE<sup>VD</sup> IG Opérateurs logiques



- C++ dispose des opérateurs logiques suivants

- not
- or
- and

| not « ! » |   |
|-----------|---|
| 0         | 1 |
| 1         | 0 |

| or «    » |   |   |
|-----------|---|---|
| 0         | 0 | 0 |
| 0         | 1 | 1 |
| 1         | 0 | 1 |
| 1         | 1 | 1 |

| and « && » |   |   |
|------------|---|---|
| 0          | 0 | 0 |
| 0          | 1 | 0 |
| 1          | 0 | 0 |
| 1          | 1 | 1 |

```
cout << boolalpha;
cout << ( (5 == 5) && (3 > 6) ) << endl; // false
cout << ( (5 == 5) || (3 > 6) ) << endl; // true
cout << ( (5 == 5) and (3 < 6) ) << endl; // true
cout << ( (5 != 5) or (3 > 6) ) << endl; // false
cout << !(5 == 5) << endl; // false
cout << not(5 != 5) << endl; // true
```

# HE<sup>VD</sup> IG Évaluation court-circuit



- L'évaluation des expressions **and** (&&) et **or** (||) se fait **de gauche à droite** et **s'arrête dès que possible**

- (a **and** b) n'évalue pas b si a est faux
- (a **or** b) n'évalue pas b si a est vrai



- Cela permet d'écrire sans risque des expressions telles que

```
((n != 0) and (p < m / n))
```

- Mais attention aux **effets de bord** de la non évaluation du terme de droite

```
(i or ++j)
```

# HE<sup>VD</sup> IG == true, == false



- Il arrive fréquemment d'avoir une variable **booléenne** à évaluer
- On pourrait donc écrire

```
if (test == true)    // test est-il vrai ?  
if (test == false)  // test est-il faux ?
```

- Mais il est plus élégant d'écrire

```
if (test)            // test est-il vrai ?  
if (!test)           // test est-il faux ?  
if (not test)        // test est-il faux ?
```

# HE<sup>VD</sup> IG Priorités des opérateurs



- L'opérateur **not** (!)  
a la même priorité que les autres **opérateurs unaires préfixes** (+, -, ++, --)  
Il est donc plus prioritaire que les opérateurs arithmétiques
- Les opérateurs de **comparaison** < > <= >=  
sont moins prioritaires que les opérateurs arithmétiques
- Les opérateurs d'**égalité** == et !=  
sont moins prioritaires que les opérateurs de comparaison
- L'opérateur de **conjonction** and (&&)  
est moins prioritaire que les opérateurs d'égalité
- L'opérateur de **disjonction** or (||)  
est moins prioritaire que le **and**

# HE<sup>VD</sup> IG Lois de De Morgan



- Le code suivant calcule des frais d'expédition hors des USA continentaux

```
fraisExpedition = 10.0;
if (not (pays == "USA" and etat != "AK" and etat != "HI")) {
    // Alaska           Hawaï
    fraisExpedition = 20.0;
}
```

- On peut le réécrire plus simplement

```
fraisExpedition = 10.0;
if (pays != "USA" or etat == "AK" or etat == "HI") {
    // Alaska           Hawaï
    fraisExpedition = 20.0;
}
```

# HE<sup>VD</sup> IG Lois de De Morgan



- La négation de la conjonction de deux propositions est équivalente à la disjonction des négations des deux propositions

$\neg (A \text{ and } B)$  est équivalent à  $(\neg A \text{ or } \neg B)$

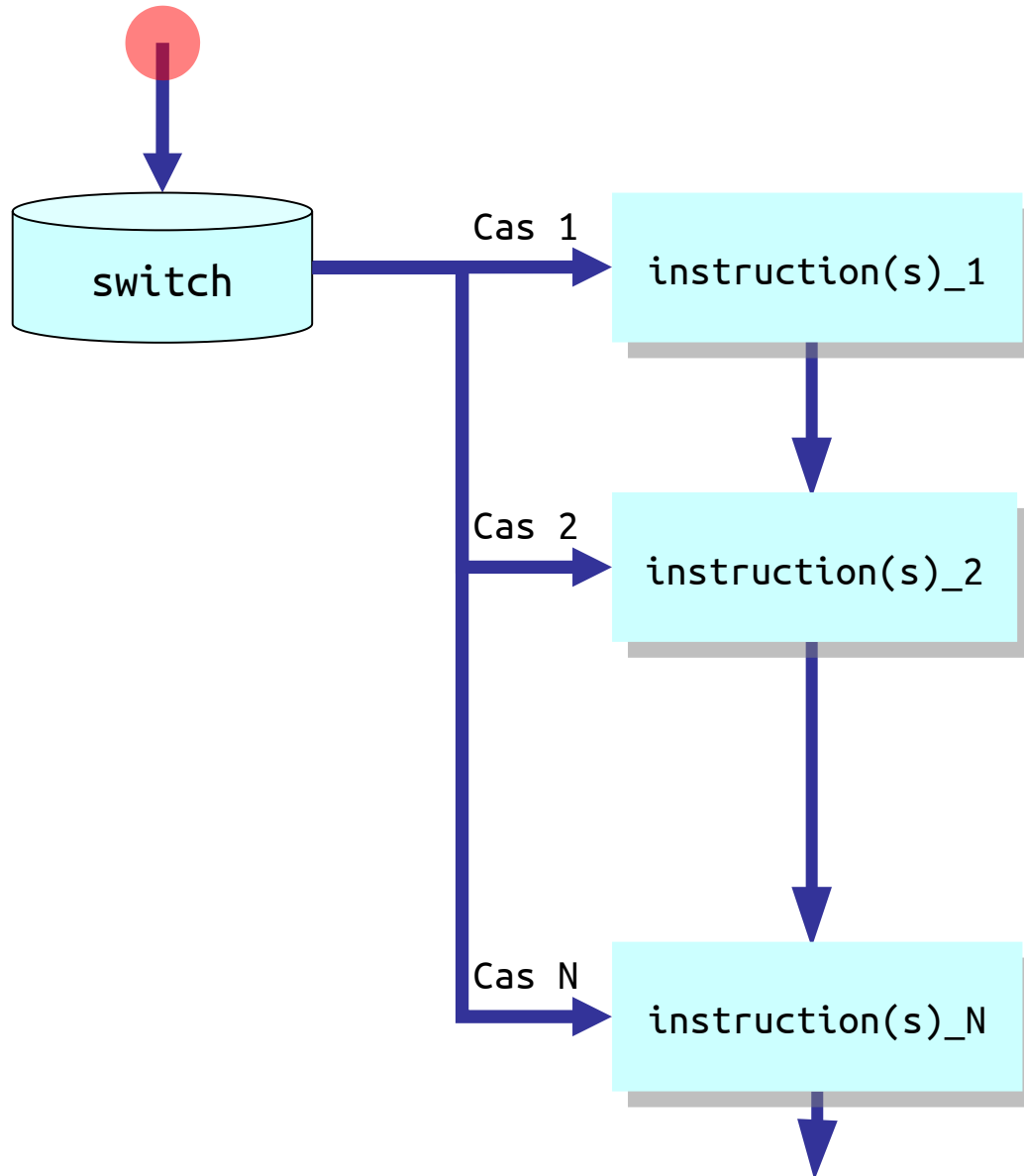
- La négation de la disjonction de deux propositions est équivalente à la conjonction des négations des deux propositions

$\neg (A \text{ or } B)$  est équivalent à  $(\neg A \text{ and } \neg B)$



## 6. switch

# HE<sup>VD</sup> IG switch, un aiguillage



```
switch (expression) {  
  case cas1:  
    instruction(s)_1;  
  
  case cas2:  
    instruction(s)_2;  
  
  case casN:  
    instruction(s)_N;  
}
```



# HE<sup>VD</sup> IG switch



Considérons ce code

```
int chiffre; string nom;
if (chiffre == 1) { nom = "un"; }
else if (chiffre == 2) { nom = "deux"; }
else if (chiffre == 3) { nom = "trois"; }
else if (chiffre == 4) { nom = "quatre"; }
else if (chiffre == 5) { nom = "cinq"; }
else if (chiffre == 6) { nom = "six"; }
else if (chiffre == 7) { nom = "sept"; }
else if (chiffre == 8) { nom = "huit"; }
else if (chiffre == 9) { nom = "neuf"; }
else { nom = ""; }
```

```
switch (chiffre) {
    case 1: nom = "un"; break;
    case 2: nom = "deux"; break;
    case 3: nom = "trois"; break;
    case 4: nom = "quatre"; break;
    case 5: nom = "cinq"; break;
    case 6: nom = "six"; break;
    case 7: nom = "sept"; break;
    case 8: nom = "huit"; break;
    case 9: nom = "neuf"; break;
    default: nom = "";
}
```

- tous les tests
  - portent sur la **même variable entière**
  - sont des **tests d'égalité**

- le **switch** permet d'écrire cela différemment

# HE<sup>VD</sup> IG switch



- Le mot réservé **switch** est suivi d'une expression de type **énumérable** (par ex. un entier) placée entre **parenthèses**
- Le corps de l'aiguillage est un bloc **{...}**
- Chaque cas est introduit par un **case** suivi d'une **expression constante** et terminée par « : »
- Le **traitement** d'un cas peut comporter zéro, une ou plusieurs **instructions** (sans qu'il soit nécessaire de les mettre entre accolades)
- La **fin d'un traitement** doit être explicitée avec l'instruction **break**; sinon on passe au cas suivant
- On peut spécifier un cas **default**:  
si **aucun** des **cas** listés par **case** n'est **atteint**

# HE<sup>VD</sup> IG switch - exemple



```
switch (jourSemaine) {  
    case 1 : cout << "premier ";  
    case 2 :  
    case 3 :  
    case 4 :  
    case 5 : cout << "jour de travail" << endl;  
             break;  
    case 6 :  
    case 7 : cout << "week-end" << endl;  
             break;  
    default: cerr << "jour incorrect" << endl;  
}
```



## 7. enum



- Le code précédent regorge de **nombre magiques** !!

- Pour l'écrire plus proprement,  
on pourrait définir des constantes entières

```
const int LUNDI      = 1;  
const int MARDI      = 2;  
const int MERCREDI   = 3;  
const int JEUDI       = 4;  
const int VENDREDI   = 5;  
const int SAMEDI      = 6;  
const int DIMANCHE   = 7;
```

- Mais C++ propose une solution plus élégante : les énumérations

```
enum JourDeLaSemaine { LUNDI, MARDI, MERCREDI, JEUDI,  
                       VENDREDI, SAMEDI, DIMANCHE };
```



La forme générale  
d'une déclaration **enum**

```
enum [ nom_de_type ]  
{ identificateur [ = expression ]  
  [ , identificateur [ = expression ] ... ]  
} [ identificateur [ , ... ] ];
```

- **nom\_de\_type**  
optionnel, donne un **nom au type** énuméré, ce qui permet de le réutiliser plus tard
- **identificateur**  
entre { } liste toutes **les valeurs possibles**
- **= expression**  
optionnel, donne une **valeur entière équivalente** à l'identificateur  
Par défaut les valeurs croissantes à partir de 0 sont utilisées
- **identificateur**  
après }, optionnel, **déclare des variables** de ce type



- Pour déclarer deux variables saison1 et saison2 pouvant prendre les valeurs PRINTEMPS, ETE, AUTOMNE, HIVER, on peut écrire :

```
enum Saison { PRINTEMPS, ETE, AUTOMNE, HIVER };  
enum Saison saison1, saison2; // comme en C
```

```
enum Saison { PRINTEMPS, ETE, AUTOMNE, HIVER };  
Saison saison1, saison2; // pas valable en C
```

```
enum Saison { PRINTEMPS, ETE, AUTOMNE, HIVER }  
    saison1, saison2; // une seule ligne
```

```
enum { PRINTEMPS, ETE, AUTOMNE, HIVER }  
    saison1, saison2; // si on ne réutilise pas le type
```



- Pour affecter la valeur ETE à saison1, on peut écrire

```
saison1 = ETE;  
saison1 = saison2;    // si saison2 vaut ETE;  
saison1 = (Saison)1;  // PRINTEMPS = 0, ETE = 1, ...  
saison1 = Saison(1);
```

Mais pas

```
saison1 = 1;           // possible en C
```

- Par contre, la **conversion de type** énuméré en entier se fait implicitement

```
int entier  = saison1; // entier vaut 1  
int entier2 = ETE;     // entier2 vaut 1
```



# HE<sup>VD</sup> IG enum - limitations



- Il n'y a **pas de vérification** lors de la conversion depuis un type numérique

```
enum Saison { PRINTEMPS, ETE, AUTOMNE, HIVER };  
Saison saison1;  
saison1 = Saison(12); // valide, alors que seules les  
                      // valeurs de 0 à 3 ont un sens.
```

- De plus, **une constante d'énumération ne peut pas être réutilisée**

```
enum Couleur {VERT, ROSE, BLEU};  
enum Fleur   {MARGUERITE, ROSE, VIOLETTE};  
              // Error: Redefinition of enumerator 'ROSE'
```



- Pour pallier à ces problèmes, C++11 a introduit des **types énumérés fortement typés** `enum class`

```
enum class Saison { PRINTEMPS, ETE, AUTOMNE, HIVER };  
Saison saison1;
```

- Pour utiliser les constantes d'énumération, il faut spécifier le nom de l'énumération

```
saison1 = Saison::ETE; // il faut spécifier Saison::
```

- Le reste de l'affectation reste valable

```
saison1 = saison2;           // si saison2 vaut ETE;  
saison1 = (Saison)1;         // PRINTEMPS = 0, ETE = 1, ...  
saison1 = Saison(1);
```



- Il n'y a **plus de conversion implicite** de l'énumération vers les entiers

```
int entier = saison1; // Error: cannot initialize a
                      // variable of type 'int' with
                      // an lvalue of type 'Saison'
```

- Mais la conversion explicite est possible

```
int entier = (int)saison1;
```

- On peut réutiliser des constantes entre énumérations

```
enum class DirH {GAUCHE, CENTRE, DROITE};
enum class DirV {HAUT, CENTRE, BAS};
DirH horizontal = DirH::CENTRE;
DirV vertical   = DirV::CENTRE;
```



# Instructions de boucle

# HE<sup>VD</sup> IG Les boucles en C++



- On a souvent besoin d'exécuter **plusieurs fois la même série d'instructions**
- Typiquement, on va **répéter** jusqu'à ce qu'un **but soit atteint**
- C++ met trois boucles à disposition

`while`

`for`

`do ... while`



## 8. while

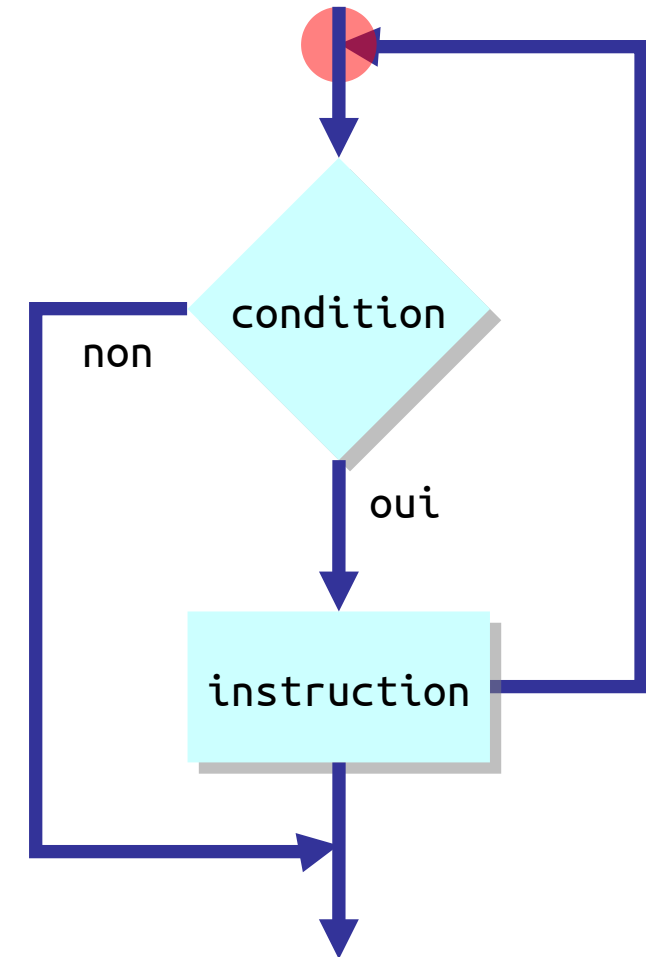
# HE<sup>VD</sup> IG *while* - ... *tant que*



- Si la **condition** est satisfaite
  - effectue **l'instruction** (le bloc d'instructions)
  - puis **revient sur la condition**

```
while (condition)  
    instruction;
```

```
nbre = 3;  
while (nbre <= 10) {  
    ++nbre;  
}
```





# Exemple – doubler son investissement

*Vous investissez la somme de 10'000 CHF au taux de 5% par an.  
Combien d'années faut-il pour au moins doubler votre mise ?*

## Algorithme

- Initialiser **annee** à 0 et **solde** à 10000.00 CHF
- Tant que le **solde** est plus < 20000.00 CHF, répéter
  - Incrémenter **annee**
  - Calculer les intérêts de 5% du **solde**
  - Mettre à jour **solde** en y ajoutant les intérêts
- Afficher **annee**, le temps nécessaire pour doubler le solde





# Exemple – doubler son investissement

- Le cœur de la boucle s'écrit, avec une constante double **TAUX** qui vaut 5.0

```
++annee;  
interet = solde * TAUX / 100.0;  
solde   = solde + interet;
```

- La **condition à tester** pour continuer d'itérer s'écrit,  
avec une constante **SOLDE\_CIBLE** qui vaut 20000.00 CHF

```
solde < SOLDE_CIBLE
```



# Exemple – doubler son investissement

La boucle complète est donc

```
while (solde < SOLDE_CIBLE) {  
    ++annee;  
    double interet = solde * TAUX / 100.0;  
    solde = solde + interet;  
}
```

- La variable **interet** peut être déclarée localement dans la boucle  
**Attention**, cette pratique peut être coûteuse avec des types composés (classes)
- Les variables **annee** et **solde** doivent être déclarées et initialisées en dehors



# Exemple – doubler son investissement

```
int main() {  
    const double TAUX          = 5.0;  
    const double SOLDE_INITIAL = 10000.0;  
    const double SOLDE_CIBLE   = 2.0 * SOLDE_INITIAL;  
  
    double solde = SOLDE_INITIAL;  
    int     annee = 0;  
  
    while (solde < SOLDE_CIBLE) {  
        ++annee;  
        double interet = solde * TAUX / 100.0;  
        solde = solde + interet;  
    }  
  
    cout << "L'investissement double apres " << annee << " annees." << endl;  
  
    return EXIT_SUCCESS;  
}
```



# Qu'affiche ce code ?

```
int i = 5;
while (i > 0) {
    cout << i << " ";
    i--;
}
```

5 4 3 2 1

- i vaut successivement 5, 4, 3, 2, 1, ...
- Quand i vaut 0
- (i > 0) vaut false
- La boucle s'arrête



# Qu'affiche ce code ?

```
int i = 5;
while (i >= 0) {
    cout << i << " ";
    i--;
}
```

5 4 3 2 1 0

- i vaut successivement 5, 4, 3, 2, 1, 0, ...
- Quand i vaut -1
- (i >= 0) vaut false
- La boucle s'arrête
- Bien choisir entre > et >= est essentiel



# Qu'affiche ce code ?

```
int i = 5;  
while (i < 0) {  
    cout << i << " ";  
    i--;  
}
```

- Quand i vaut 5
- (i < 0) vaut false
- La boucle s'arrête avant même de commencer  
=> Elle n'est **jamais exécutée !**





# Qu'affiche ce code ?

```
unsigned int i = 5;  
while (i >= 0) {  
    cout << i << " ";  
    i--;  
}
```

5 4 3 2 1 0  
4294967295  
4294967294  
4294967293  
...

- Quand i vaut -1,  
il vaut en fait 4294967295
- (i >= 0) vaut toujours true
- La boucle ne s'arrête jamais,  
c'est une boucle infinie



# Qu'affiche ce code ?

```
int i = 5;
while (i > 0); {
    cout << i << " ";
    i--;
}
```

- Une autre **boucle infinie** causée par un **point-virgule mal placé** (instruction vide)
- La valeur de `i` n'est **pas modifiée** par la boucle
- La valeur de `(i > 0)` ne change pas, et reste **true**



# HE<sup>VD</sup> IG Qu'affiche ce code ?



```
int i;  
while (i >= 0) {  
    cout << i << " ";  
    i--;  
}
```

- i n'est **pas initialisé**  
=> sa valeur est indéterminée
- Le passage dans la boucle est **aléatoire**



- 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

| n    | sum | digit |
|------|-----|-------|
| 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  $1729 \% 10 = 9$

- **sur**

| n    | sum          | digit |
|------|--------------|-------|
| 1729 | <del>0</del> |       |
|      | 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  $1729 / 10 = 172$
- On barre les anciennes valeurs et indique la nouvelle valeur

| n               | sum          | digit |
|-----------------|--------------|-------|
| <del>1729</del> | <del>0</del> |       |
| 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        |
|-----------------|--------------|--------------|
| <del>1729</del> | <del>0</del> |              |
| <del>172</del>  | <del>9</del> | <del>9</del> |
| 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        |
|-----------------|---------------|--------------|
| <del>1729</del> | <del>0</del>  |              |
| <del>172</del>  | <del>9</del>  | <del>9</del> |
| <del>17</del>   | <del>11</del> | <del>2</del> |
| <del>1</del>    | <del>18</del> | <del>7</del> |
| 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  $1 + 7 + 2 + 9$



9. for



# HE<sup>VD</sup> IG for - *pour tous les ...*



- 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;        // incrémentation
}
```

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



- Rien n'oblige à se restreindre à parcourir des intervalles  
De manière générale, la boucle « for » s'écrit

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

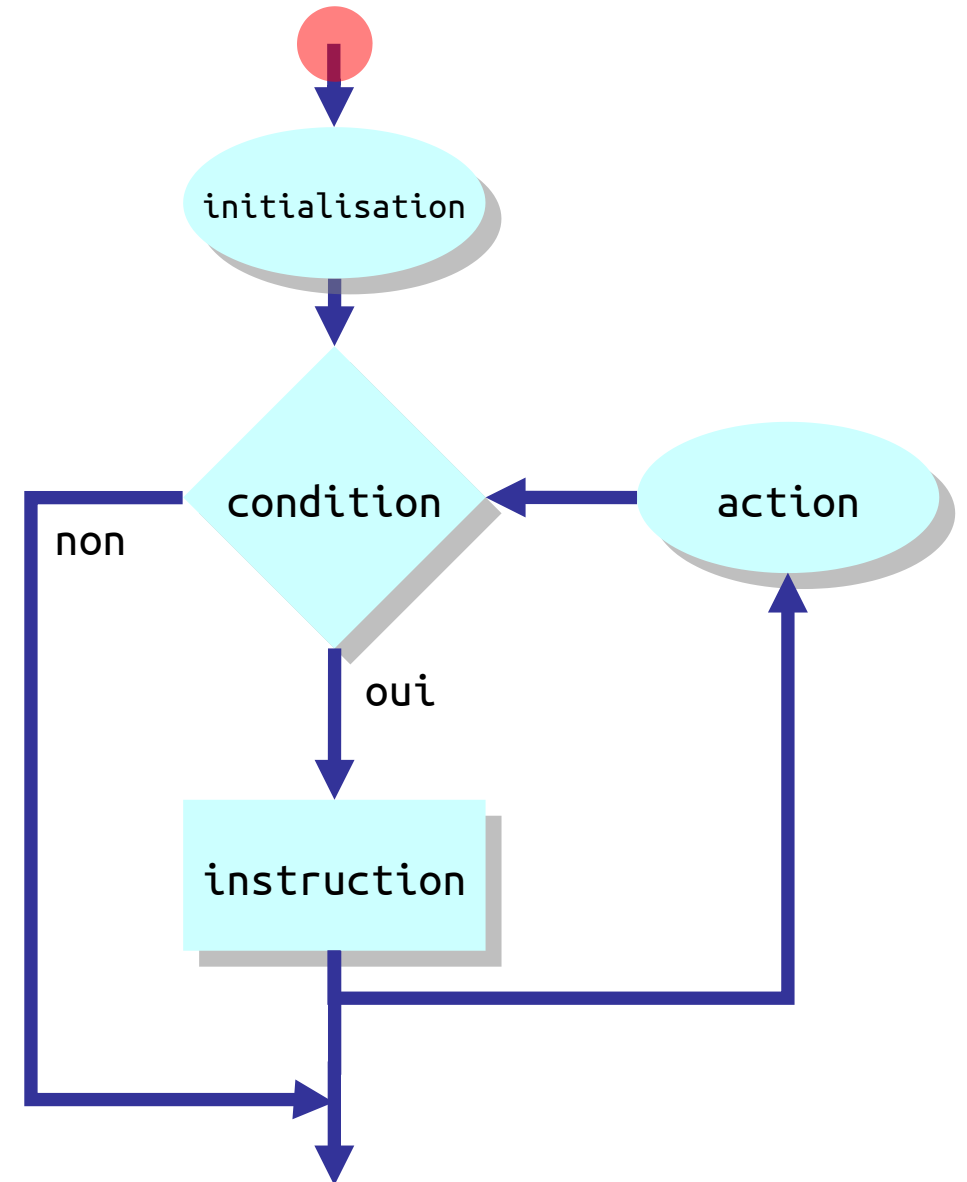
- **initialisation** s'évalue **une seule fois**, en général pour initialiser (voire déclarer) la variable de boucle
- **condition** expression **évaluée avant chaque itération** pour déterminer s'il faut continuer à boucler
- **action** est évaluée **à la fin de chaque itération**, sert en général à modifier la variable de boucle



```
for (i = 5; i <= 8; ++i) {  
    cout << "i = " << i << endl;  
}
```

Valeur de i : 9

```
i = 5  
i = 6  
i = 7  
i = 8
```





- La présence des 3 expressions n'est pas obligatoire

```
for ( ; condition; action)  
    instruction;
```

initialisation avant la boucle

```
for (initialisation; ; action)  
    instruction;
```

une condition vide retourne **true**

```
for (initialisation; condition; )  
    instruction;
```


action vide (intégrée dans la boucle)

```
for ( ; ; )  
    instruction;
```

boucle infinie  
une boucle **while** (**true**)  
serait plus lisible



- Les **déclarations de variables** peuvent intervenir partout dans le code
- Dans le cas d'une boucle **for**, la **variable de boucle** ne devrait **exister que pour la boucle et dans la boucle**



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

- **Attention**  
vous n'avez alors plus accès à la valeur de `i` après la boucle



# Exemple – investir pendant 5 ans

*Vous investissez la somme de 10'000 CHF au taux de 5% par an*

*~~Combien d'années faut-il pour au moins doubler votre mise?~~*

Comment évolue le solde sur une période de 5 ans?

## Algorithme

- Initialiser **solde** à 10000.00 CHF
- Pour **annee** allant de 1 à 5 inclus, répéter
  - Mettre à jour **solde** en y ajoutant 5% d'intérêts
  - Afficher **solde**



# Exemple – investir pendant 5 ans

La boucle **for** s'écrit, avec une constante **DUREE** valant 5 et une variable de boucle **annee** définie localement.

```
cout << fixed << setprecision(2);  
for (int annee = 1; annee <= DUREE; ++annee) {  
    solde = solde * (1 + TAUX / 100.00);  
    cout << annee << setw(9) << solde << endl;  
}
```

```
1 10500.00  
2 11025.00  
3 11576.25  
4 12155.06  
5 12762.82
```

# HE<sup>VD</sup> IG Qu'affiche ce code ?



```
for (int i = 0; i <= 5; ++i) {  
    cout << i << " ";  
}
```

0 1 2 3 4 5

- i vaut successivement 0, 1, 2, 3, 4, 5, ...
- Quand i vaut 6,  
(i <= 5) vaut *false*  
et la boucle s'arrête
- La boucle s'est *exécutée 6 fois et non 5*



# HE<sup>VD</sup> IG En sens inverse



```
for (int i = 5; i >= 0; --i) {  
    cout << i << " ";  
}
```

5 4 3 2 1 0

- On peut aussi **décrémenter** pour mettre à jour la variable de boucle
- Cela parcourt les valeurs dans l'autre sens
- La boucle s'est **exécutée 6 fois et non 5**

# HE<sup>VD</sup> IG À grands pas



```
for (int i = 0; i < 9; i += 2) {  
    cout << i << " ";  
}
```

- Rien n'oblige à avancer par pas de 1
- On peut ajouter ou retirer de plus grands ou plus petits pas

0 2 4 6 8

# HE<sup>VD</sup> IG On s'emballe...



```
for (int i = 0; i != 9; i += 2) {  
    cout << i << " ";  
}
```

- On peut aussi avoir des **boucles infinies** avec des boucles « for »
- On **évite** souvent les opérateurs **d'égalité** dans la condition d'une boucle « for »

0 2 4 6 8 10 12 14 ...

# HE<sup>VD</sup> IG A pas exponentiels



```
for (int i = 1; i < 20; i *= 2) {  
    cout << i << " ";  
}
```

- Le pas peut être **multiplicatif** ou tout autre expression
- Un pas multiplicatif donne une **évolution exponentielle** de la variable de boucle

**1 2 4 8 16**

# HE<sup>VD</sup> IG À tout petits pas exponentiels



```
for (int i = 0; i < 20; i *= 2) {  
    cout << i << " ";  
}
```

- Avec des pas multiplicatifs, il faut faire **attention à l'initialisation**
- Ici encore on a une **boucle infinie**

0 0 0 0 0 ...



# Bornes symétriques, asymétriques

- parcourt l'intervalle **symétrique**  $[a, b]$ 

```
for (int i = a; i <= b; ++i)
```
- parcourt l'intervalle **asymétrique**  $[a, b[$ 

```
for (int i = a; i < b; ++i)
```
- pour **parcourir les caractères d'une chaîne** `s`,  
on utilise typiquement un intervalle asymétrique  
`s.length()` indique le nombre de caractères contenus dans `s`

```
for (size_t i = 0; i < s.length(); ++i)
```

# HE<sup>VD</sup> IG for each



- C++11 a introduit une **nouvelle syntaxe pour for**
- Elle permet de **parcourir tous les éléments** d'un contenant (foreach en C#, for en Java, ...)
- Exemple  
parcours de tous les éléments d'une string **str** comme ceci

```
string str("Hello");  
for (char c : str) {  
    cout << char(toupper(c));  
}
```

HELLO

- Nous y reviendrons plus en détail après avoir vu les références (chap. 4) et les tableaux (chap. 5)



## 10. do ... while



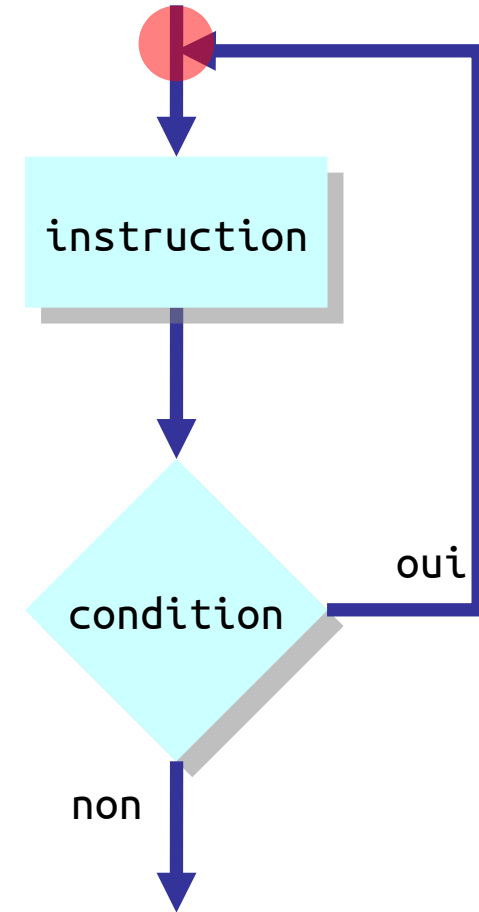
# HE<sup>VD</sup> IG do ... while - tant que...



- Exécute l'instruction ou le bloc une 1<sup>ère</sup> fois
- Recommence tant que la condition est vraie

```
do  
    instruction;  
while (condition);
```

```
cout << "entrez un nombre > 0 ";  
do {  
    cin >> nbre;  
} while (nbre <= 0);
```





# Exemple – gestion des entrées utilisateur

- Un exemple typique d'utilisation consiste à gérer les **entrées utilisateur**
- Une boucle **do...while** permet de répéter le message original en cas d'erreur de l'utilisateur

```
int valeur;  
do {  
    cout << "Entrez un entier < 100: ";  
    cin >> valeur;  
} while (valeur >= 100);  
  
cout << "Valeur = " << valeur << endl;
```

```
Entrez un entier < 100: 2015  
Entrez un entier < 100: 421  
Entrez un entier < 100: 89  
Valeur = 89
```



# Gestion des erreurs d'entrée

- Soit le code suivant

```
int valeur;  
do {  
    cout << "Entrez un entier >= 100: ";  
    cin >> valeur;  
} while (valeur < 100);  
  
cout << "Valeur = " << valeur << endl;
```

```
Entrez un entier >= 100: 13  
Entrez un entier >= 100: a  
Entrez un entier >= 100: Entrez un  
entier >= 100: Entrez un entier >=  
100: Entrez un entier >= 100:  
Entrez un entier >= 100: Entrez un  
entier >= 100: Entrez un entier >=  
100: Entrez un entier >= 100:  
Entrez un entier >= 100: Entrez un  
entier >= 100: Entrez un entier >=  
100: ...
```

- Que se passe-t-il si l'utilisateur **entre une lettre plutôt qu'un nombre ?**



# Gestion des erreurs d'entrée

Que s'est-il passé?

- La valeur saisie « a » ne permet pas de construire un entier
- Le caractère saisi n'est pas consommé et reste dans le flux d'entrée cin
- La lecture n'ayant pas abouti, la variable valeur conserve sa valeur précédente (avant C++11) ou est mise à zéro (C++11)
- La prochaine lecture essaie à nouveau de lire le caractère 'a' ... et on entre dans une boucle infinie

# HE<sup>VD</sup> IG Gestion des erreurs d'entrée



- Il va donc falloir
  1. Détecter l'erreur de lecture
  2. Effacer l'indication d'erreur
  3. Vider le tampon du flux d'entrée

```
cin.fail()
```

renvoie un booléen qui indique s'il y a eu une erreur

```
cin.clear()
```

réinitialise l'indication d'erreur

```
cin.ignore(streamsize n, int delim = EOF)
```

permet de retirer jusqu'à n caractères du flux, ou jusqu'à ce qu'on rencontre le caractère `delim` (par défaut la fin du fichier)



# Gestion des erreurs d'entrée

- On peut donc détecter et corriger l'état du flux suite à une erreur de lecture avec le code suivant

```
cin >> valeur;  
if (cin.fail()) {  
    cin.clear();  
    cin.ignore(numeric_limits<streamsize>::max(), '\\n');  
}
```

# HE<sup>VD</sup> IG Gestion des erreurs d'entrée



En plus de la méthode `fail()`, les méthodes `good()`, `eof()` et `bad()` peuvent aussi nous renseigner sur l'état du flux

| Etat   | <code>good()</code> | <code>eof()</code> | <code>fail()</code> | <code>bad()</code> |
|--|---------------------|--------------------|---------------------|--------------------|
| Pas d'erreur   | true                | false              | false               | false              |
| Fin de fichier sur l'opération de lecture (^D)             | false               | true               | false               | false              |
| Erreur logique sur l'opération d'entrée/sortie             | false               | false              | true                | false              |
| Erreur de lecture/écriture sur l'opération d'entrée/sortie | false               | false              | true                | true               |



# Gestion des erreurs d'entrée

- De manière plus compacte, il est possible de tester le flux lui-même  
La **conversion d'un flux en booléen** est équivalente  
à appeler la méthode `.good()`

```
if (not(cin >> valeur)) {  
    cin.clear();  
    cin.ignore(numeric_limits<streamsize>::max(), '\\n');  
}
```



# HE<sup>VD</sup> IG

## Quelle boucle choisir ?



**while**

- Le **nombre** d'itérations n'est **pas prévisible**
- Les instructions ne sont **peut-être jamais** exécutées

**do ... while**

- Le **nombre** d'itérations n'est **pas prévisible**
- Les instructions doivent s'exécuter **au moins une fois**

**for**

- Le **nombre** d'itérations est **déterminé** à l'avance



# 11. Boucles imbriquées

# HE<sup>VD</sup> IG Boucles imbriquées



- Comment afficher le tableau ci-contre ?  
Une **matrice** de **5 lignes** et **4 colonnes**

|    |    |    |    |
|----|----|----|----|
| a1 | a2 | a3 | a4 |
| b1 | b2 | b3 | b4 |
| c1 | c2 | c3 | c4 |
| d1 | d2 | d3 | d4 |
| e1 | e2 | e3 | e4 |

**faire** pour toutes les lignes

**faire** pour toutes les colonnes

afficher les coordonnées (ligne colonne)

passer à la ligne



# Boucles imbriquées – mise en œuvre

```
const char ESPACE = ' '; // séparations

// pour toutes les lignes
for (char ligne = 'a'; ligne <= 'e'; ++ligne) {
    // pour toutes les colonnes
    for (int col = 1; col <= 4; ++col) {
        // afficher les termes de la matrice
        cout << ligne << col << ESPACE;
    } // for col ...

    // changer de ligne
    cout << endl;
} // for ligne ...
```

# HE<sup>VD</sup> IG Boucles imbriquées



- Qu'affiche ce code ?

```
for (int i = 1; i <= 4; ++i) {  
    for (int j = 1; j <= i; ++j) {  
        cout << "*";  
    }  
    cout << endl;  
}
```

```
*  
**  
***  
****
```

- Les **variables de boucles** peuvent être liées
- Les bornes de la boucle intérieure dépendent de la valeur de la variable de boucle extérieure

# HE<sup>VD</sup> IG Boucles imbriquées



- Si nous voulons afficher la matrice suivante
- On peut l'écrire comme ceci

|    |    |    |    |    |
|----|----|----|----|----|
| 0  | 1  | 2  | 3  | 4  |
| 5  | 6  | 7  | 8  | 9  |
| 10 | 11 | 12 | 13 | 14 |

```
const int COLONNES(5), LIGNES(3);
const int LARGEUR(3);

for (int i = 0; i < LIGNES; ++i) {
    for (int j = 0; j < COLONNES; ++j) {
        int ij = j + COLONNES * i;
        cout << setw(LARGEUR) << ij;
    }
    cout << endl;
}
```

# HE<sup>VD</sup> IG Boucles imbriquées



- Si nous voulons afficher la matrice
- ... on peut aussi utiliser un **compteur ij commun** aux deux boucles en plus des compteurs i et j

|    |    |    |    |    |
|----|----|----|----|----|
| 0  | 1  | 2  | 3  | 4  |
| 5  | 6  | 7  | 8  | 9  |
| 10 | 11 | 12 | 13 | 14 |

```
const int COLONNES(5), LIGNES(3);  
const int LARGEUR(3);  
  
for (int i = 0, ij = 0; i < LIGNES; ++i) {  
    for (int j = 0; j < COLONNES; ++j, ++ij) {  
        cout << setw(LARGEUR) << ij;  
    }  
    cout << endl;  
}
```

Qu'est-ce que cela ?

# HE<sup>VD</sup> IG Opérateur virgule



expression1 , expression 2

- Un opérateur binaire qui
  - évalue expression1
  - en ignore la valeur de retour
  - puis évalue expression2
  - en retourne la valeur
- **Attention**  
c'est l'opérateur le moins prioritaire de tous

```
a = 5 , 3;      // a = 5  
a = ( 5 , 3 );  // a = 3
```





## 12. Instructions de saut

# HE<sup>VD</sup> IG Les sauts en C++



- Il arrive parfois, mais **rarement**, qu'un algorithme ne puisse pas être exprimé parfaitement en terme de boucles
- On voudrait effectuer n boucles et demi, **sauter** une partie de certaines itérations, ...
- C++ a **trois instructions** pour mettre en œuvre des sauts à utiliser avec **parcimonie !**

break

continue

goto



- Pour l'instruction de saut goto

```
goto ETIQUETTE;
```

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

```
ETIQUETTE: instruction;
```

- À utiliser avec extrême parcimonie, pour ne pas faire de code spaghetti !





- Exemple possible d'utilisation de l'instruction **goto** : **sortir de boucles imbriquées**

```
int i , j;  
for (i = 1; i <= 5; ++i) {  
    for (j = 1; j <= 5; ++j) {  
        if (i * j > 10)  
            goto finBoucleExterne;  
        cout << i * j << endl;  
    }  
}  
finBoucleExterne:  
cout << "i = " << i  
    << ", j = " << j  
    << endl;
```

```
1  
2  
3  
4  
5  
2  
4  
6  
8  
10  
3  
6  
9  
i = 3, j = 4
```



- Il est parfois utile de **sortir d'une boucle** ailleurs qu'à son début (**while**) ou à sa fin (**do...while**)
- L'instruction **break** permet de sortir de la **boucle courante** à tout moment

```
break;
```

- Notons que nous l'avons déjà rencontrée pour sortir d'un **switch**  
Il est également possible d'avoir **plusieurs instructions break** **dans la même boucle** fournissant plusieurs points de sortie
- **Attention**  
ne permet de **sortir que d'une seule boucle**  
=> ne permet pas de sortir de boucles imbriquées



# Application - afficher une barrière

- Nous voulons écrire un code qui affiche une barrière sous la forme

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

- Il y a 6 poteaux, mais 5 sections horizontales

```
const int SECTIONS(5);  
  
for (int i = 0; i < SECTIONS; ++i) {  
    cout << "|";  
    cout << "==";  
}  
cout << "|";
```

Duplication de code !





# Application - afficher une barrière

- Il faudrait effectuer 5 boucles et demi et sortir au milieu de la 6<sup>e</sup>
- C'est possible avec une boucle infinie dont on sort au milieu avec une instruction `break`

```
const int SECTIONS(5);  
int i = 0;  
while (true) {  
    cout << "|";  
    if (i == SECTIONS) {  
        break;  
    }  
    cout << "=="  
    ++i;  
}
```

Placer une condition `if` dans une boucle n'est pas conseillé

```
const int SECTIONS(5);  
cout << "|";  
for (int i = 0; i < SECTIONS; ++i) {  
    cout << "=="  
}
```



- L'instruction `continue` interrompt l'itération en cours pour commencer la prochaine s'il y a lieu

```
continue;
```

- Les codes suivants sont équivalents

```
while (EXPRESSION) {  
    instructions1;  
    if (CONDITION)  
        continue;  
    instructions2;  
}
```

```
while (EXPRESSION) {  
    instructions1;  
    if (not CONDITION) {  
        instructions2;  
    }  
}
```

```
while (EXPRESSION) {  
    instructions1;  
    if (CONDITION)  
        goto FIN;  
    instructions2;  
FIN : ;  
}
```

- La variante du centre est généralement la plus lisible





- Que fait `continue` dans les différentes boucles ?

`while`

- Passe en début de la boucle pour **évaluer la condition**

`do ... while`

- Passe à la fin de la boucle pour **évaluer la condition**

`for`

- Passe en fin de boucle pour évaluer l'**action** (typiquement l'incrémentation)
- **Puis** passe en début de boucle pour **évaluer la condition**

# HE<sup>VD</sup> IG for vs. while

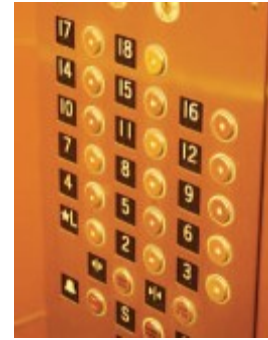
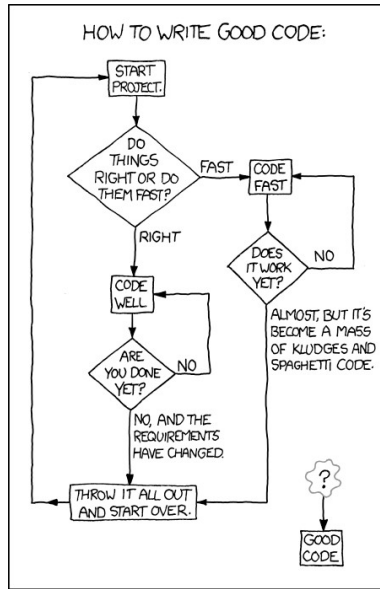


- Les deux codes suivants sont équivalents

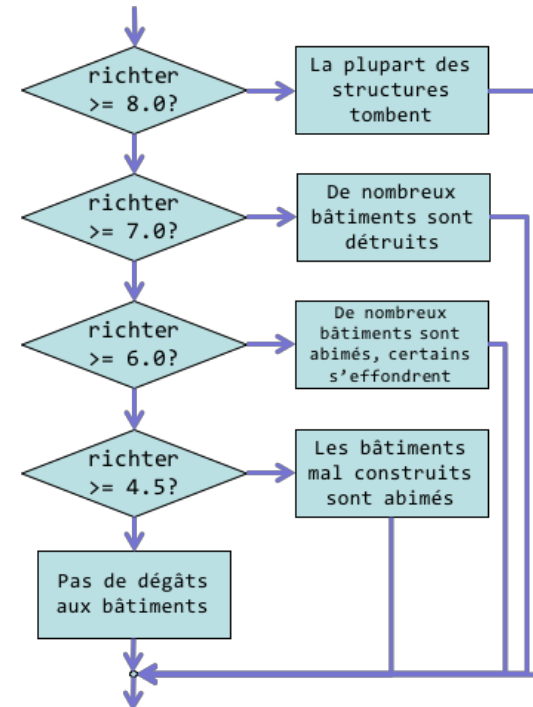
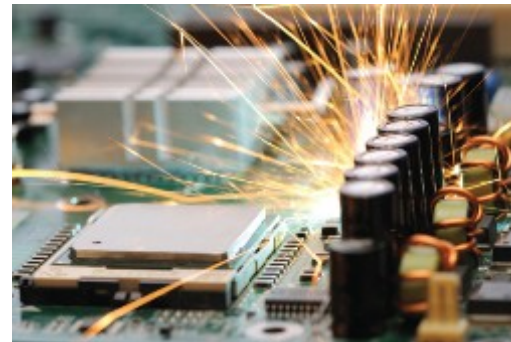
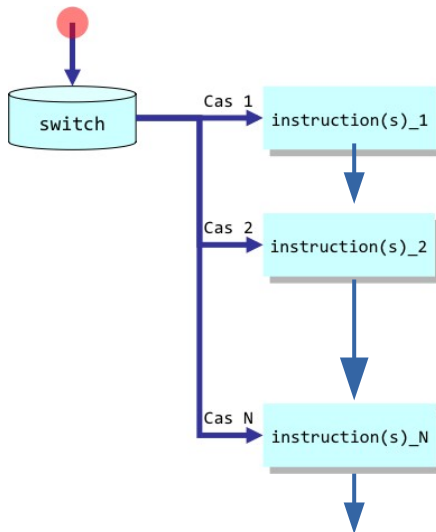
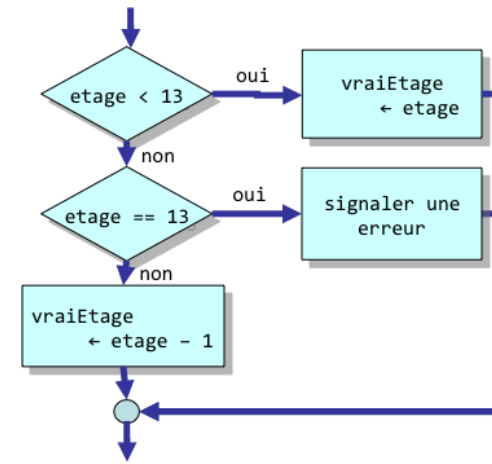
- ```
for (initialisation; condition; action) {  
    instructions;  
}
```

```
{ initialisation;  
    while (condition) {  
        instructions;  
    FIN: action;  
    }  
}  
SUITE: ;
```

- avec `continue` remplacé par `goto FIN` et `break` remplacé par `goto SUITE` dans instructions, le cas échéant



# Résumé



# HE<sup>VD</sup> IG Résumé (conditions)



- Les **organigrammes** permettent de représenter le flux d'exécution d'un programme.
- On peut grouper les instructions par **blocs**. Un bloc se comporte comme une instruction unique du point de vue du reste du programme.
- **if else** permet de coder la notion de « **si, sinon** ». Le **else** est facultatif. Une bonne pratique consiste à **toujours utiliser un bloc** après **if** ou **else**.
- L'**opérateur conditionnel** (condition) `? val1 : val2` permet d'effectuer un test sous la forme d'une **expression**, i.e. en renvoyant une valeur.
- Le type **bool** représente des valeurs vraies / fausses.



# Résumé (conditions)

- Il y a 6 **opérateurs de comparaison** disponibles sur les entiers, réels, chaînes :  
== , != , < , > , <= , >=.
- Attention
  - à ne pas confondre = et ==
  - à ne pas comparer < 0 pour des **types non signés**
  - Quand vous tester **l'égalité** entre deux **réels**
- Il y a 3 **opérateurs logiques**: not (!), and (&&) et or (||), avec évaluation **court-circuit** pour ces derniers.
- **switch** remplace avantageusement des **if else** imbriquées pour tester **l'égalité avec plusieurs constantes**
- **enum** et **enum class** permettent de **nommer** ces constantes plutôt que d'utiliser des nombres magiques.

# HE<sup>VD</sup> IG Résumé (boucles)



- `while` permet de **boucler au minimum zéro fois** tant qu'une condition est respectée.
- `do...while` permet de **boucler au minimum une fois** tant qu'une condition est respectée.
- `for` permet d'écrire tout le code de contrôle d'une boucle `while` en une seule ligne dans le cas particulier où le **nombre d'itérations est déterminé**
- Les boucles peuvent être imbriquées.
- `break` permet de **sortir de la boucle** courante
- `continue` permet de **sauter à l'itération** suivante
- `goto` permet de **sauter à** une étiquette