Méthodes informatiques pour physiciens

introduction à C++ et
résolution de problèmes de physique par ordinateur

Leçon # 2 : Conditions et Boucles

Alessandro Bravar Alessandro.Bravar@unige.ch
tél.: 96210 bureau: EP 206

assistants

Johanna Gramling Johanna.Gramling@unige.ch
tél.: 96368 bureau: EP 202A

Mark Rayner Mark.Rayner@unige.ch
tél.: 96263 bureau: EP 219

http://dpnc.unige.ch/~bravar/C++2015/L2
pour les notes du cours, les exemples, les corrigés, …

Plan du jour #2

Récapitulatif et corrigé de la leçon #1

Déroulement d'un programme et structures de contrôle

texte Micheloud et Rieder chap. 5 et 6

Exécutions conditionnelles :

instruction `if`

`if ... else`

Opérateurs relationnels et logiques

Itérations et développement d'un algorithme

Les boucles :

instructions `while`

`do... while`

`for`

Récapitulatif de la leçon #1

Ecriture, compilation et exécution d'un programme

Entrée / Sortie (clavier / écran)

```
cin >> x >> y;
cout << "Hello" << y;
```

Déclaration, initialisation et affectation des variables

```
int
double
```

Commentaires !

```
//
/* . . . */
```

Opérations arithmétiques

```
+, -, *, /, %
```

Fonctions mathématiques

Mémorisez le programme suivant !

Il contient tous les élément importants vus la semaine passée.

```
//calcul de la racine carree
#include <iostream>
#include <cmath>

using namespace std;

int main() {
  double x, y;
  cout << "Entrez deux nombres > 0 :" << endl;
  cin >> x >> y;

  double z = sqrt(x+y);
  cout << "La racine carree de " << x+y << " est " << z << endl;

  //on peut aussi ecrire :
  /*
  cout << "La racine carree de " << x+y << " est " << sqrt(x+y) << "\n";
  */

  return 0;
}
```

Racine.cpp

Déroulement du programme

exécution séquentielle

chaque instruction est exécutée une fois dans son ordre d'apparition

exécution conditionnelle

l'exécution d'une certaine instruction dépend de conditions qui peuvent changer ou être modifiées en cours de processus

instructions: `if`

`if . . . else`

exécution itérée

la même instruction (ou bloc d'instructions) est répétée plusieurs fois jusqu'à ce qu'une condition prédéterminée se vérifie;

elles sont appelées *boucles* et sont à la base des algorithmes

instructions: `while`

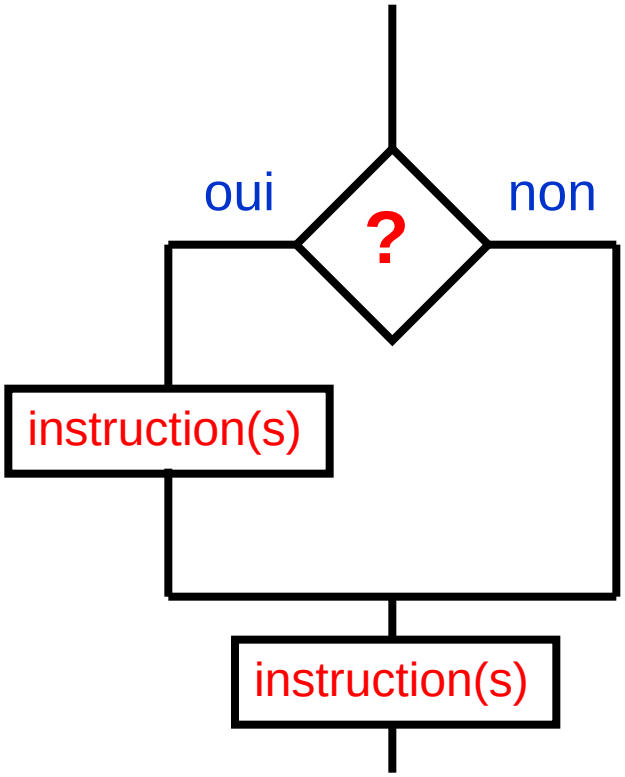
`do . . . while`

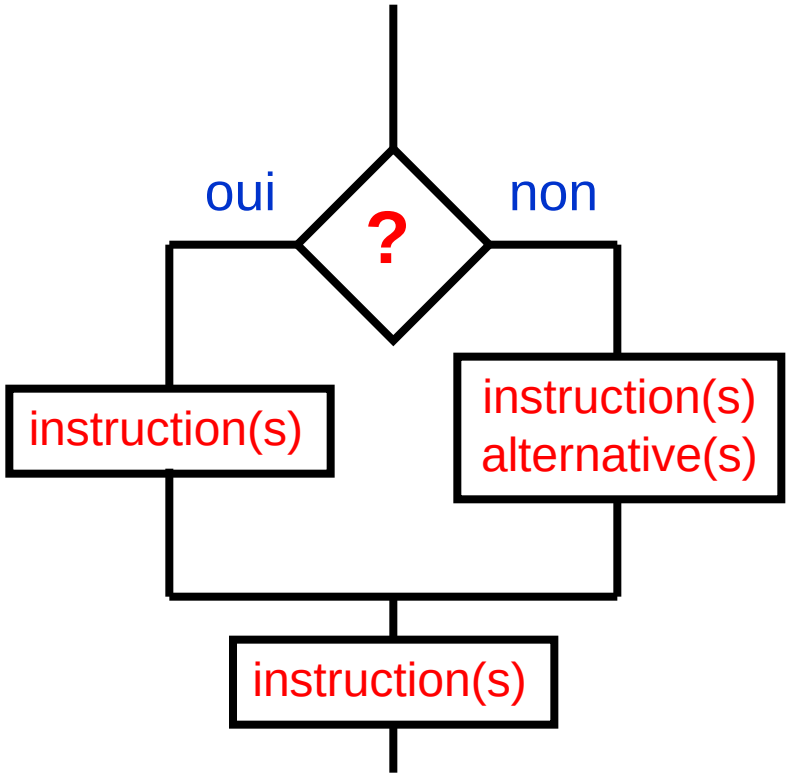
`for`

Instructions `if` et `if ... else`

L'instruction `if` permet d'effectuer des exécutions soumises à des conditions.

La condition peut être modifiée pendant l'exécution du programme.






```
if (condition) instruction;
```

```
if (condition) instructionA;
else instructionB;
```

La condition est une expression logique (entière), l'instruction sera exécutée uniquement si la condition est **true** (vraie), c'est-à-dire si l'expression a une valeur différente de zéro.

N'oubliez pas les parenthèses obligatoires autour de la condition !

Opérateurs relationnels et logiques

Pour exprimer et évaluer les conditions, on utilise des symboles appelés

opérateurs relationnels

qui produisent des valeurs lorsqu'ils sont combinés aux expressions.

Le résultat d'une opération relationnelle (comparaison) est de type booléen et vaut:

true (vrai) ou **false** (faux)

<	inférieur à
<=	inférieur ou égal à
==	égal à !!!
>	supérieur à
>=	supérieur ou égal à
!=	différent de

p. ex. a = 3, b = 5

(a <= b)	(a > b)	(a==b)	(a != b)
vrai	faux	faux	vrai

On peut combiner plusieurs conditions en utilisant les trois opérateurs logiques :

&& = AND (et) || = OR (ou) ! = NOT (négation)

p.ex. vérifier si n est positif et inférieur à 5: (n > 0 && n < 5)

vérifier si n est négatif ou supérieur à 3: (n < 0 || n > 3)

Pour étudier les conditions complexes on utilise les tableaux de vérité.

Instruction `if`

```
if (condition) instruction;
else instruction alternative;
```

Pour tester la condition, l'expression dans les parenthèses est évaluée:
si la condition est vraie (true), l'instruction est exécutée;
si la condition est fausse (false), le programme passe à la prochaine instruction.

Pour évaluer les conditions, on utilise les opérateurs relationnels.

Écrivez un programme pour tester la divisibilité de m par n (m, n sont entiers) !

```
cin >> m >> n;
if (m%n == 0) cout << "n divise m";
```

```
cin >> m >> n;
if (m%n == 0) cout << "n divise m";
else cout<< "n ne divise pas m";
```

Ici la condition est : `m%n` est égal à 0 ? Si `m%n == 0`, la condition est vraie.

`if (x = 3)` est différent de `if (x == 3)`
(`==` est un opérateur relationnel !, `=` est l'opérateur d'affection !)
la première expression est toujours vraie, la valeur de l'expression entre () ≠ 0 !),
la deuxième dépend de la valeur de `x`.

S'il n'y a pas d' instructions alternatives, le bloc `else` peut être omis.

Exemple

Ce programme vérifie si deux nombres sont divisibles.

```
#include <iostream>

using namespace std;

int main() {
  int n;
  cout << "Entrez un nombre a diviser : ";
  cin >> n;

  int div;
  cout << "Entrez un diviseur : ";
  cin >> div;

  if (n%div==0) cout << n << " est divisible par " << div << endl;
  else cout << n << " n'est pas divisible par " << div << endl;

  return 0;
}
```

voir Ex_if.cpp et Ex_if-else.cpp

d'abord on évalue `n%div`, puis on vérifie si le résultat est égal à zéro (notez le == !);

si `n` est divisible par `div`, la condition est vraie et l'instruction suivante sera exécutée.

si `n` n'est pas divisible par `div`, la condition est fausse et l'instruction alternative sera exécutée.

Quelques exemples de conditions

`if (x <= 5)` vrai, si x ≤ 5

`if (x == 3)` vrai, si x = 3

`if (x = 3)` toujours vrai, la valeur de l'expression entre parenthèses ≠ 0 !

`if (x = 0)` toujours faux, parce que `x=0` et la valeur de l'expression entre ( ) = 0 !

attention: == et = sont des opérateurs très différents (relationnel et d'affectation)

La valeur zéro équivaut à `false`, alors que les autres valeurs sont égales à `true`.

```
x = 0;  if (x)
```
faux, la valeur de l'expression entre parenthèses est 0, donc la condition est fausse.

```
        if (3+5)
```
vrai, la valeur de l'expression est différente de zéro.

Les opérateurs logiques ont la priorité sur les opérateurs relationnels :
la condition `x > 5 && y > 5 || z > 5`
est différente de `(x > 5) && (y > 5 || z > 5)`

Pour changer les priorités, on utilise les parenthèses.
Dans le doute, écrivez un petit programme pour tester la condition.

L'opérateur logique `NOT` (`!`) donne un résultat vrai, si l'expression évalué est fausse :
`if (!(x==3))` est vrai, si `x` n'est pas égal à 3 ;
la condition précédente est équivalente à `if (x != 3)`.

* Variables booléennes

Etant donnée que le résultat d'une conditions est vrai (`true`) ou fausse (`false`)
on a introduit un nouvelle type de variable, dite booléenne, pour enregistrer ce résultat.
[Les variables booléennes sont enregistrés dans un octet, donc ils ne occupent pas beaucoup de mémoire.]

```
bool cond1 = true;
if (cond1) ......
```
vrai, la valeur de la variable booléenne `cond1` est `true`

```
bool cond2 = false;
if (cond2) ......
```
faux, la valeur de la variable booléenne `cond2` est `false`

La valeur **`false`** correspond à tous les bits dans l'octet mis à zéro,
n'importe quelle valeur ≠ zéro est considéré comme `vrai` .

On peut aussi affecter des valeurs numériques à des variables booléennes

`cond1 = 5;` (donc **`true`**)

`cond2 = 0;` (donc **`false`**)

ou des résultat des opérations logiques

`cond1 = (n%m == 0);` **`true`**, si n est divisible par m

`cond2 = n%m;` **`false`**, si n est divisible per m, car le reste est 0

voir Bool.cpp

Conditions complexes

Et si je veux exécuter plusieurs instructions soumises à la même condition ?

```
if (condition)
  instruction1;
instruction2;
instruction3;
```

est différent de

```
if (condition) {
  instruction1;
  instruction2;
}
instruction3;
```

Dans le programme à gauche, les instructions 2 et 3 sont toujours exécutées tandis que l'instruction 1 sera exécutée seulement si la condition est vraie.
Dans le programme à droite, l'instruction 3 est toujours exécutée,
et les instructions 1 et 2 seront exécutées seulement si la condition est vraie.

Si on veut exécuter plusieurs instructions soumises à la même condition,
il faut les entourées avec des accolades **{** et **}** .
On utilise les accolades **{** et **}** pour construire des instructions composées.
Les instructions dans les accolades sont appelées bloc ou bloc d'instructions.
De l'extérieur, le bloc peut être vu comme une seule instruction.

On utilise aussi les accolades pour rendre le programme plus lisible
et pour éviter des situations ambiguës.
Dans ce cas, Il faut faire attention à la portée des variables définies dans le bloc d'instructions (les variables définis le bloc existent seulement dans le bloc).

* Blocs d'instructions et portée

Un *bloc d'instructions* est une suite d'instructions encadrées par des accolades **{ }** , de l'extérieur, le bloc peut être interprété comme une seule instruction.

p.ex.
```
if (x > y) {int temp = x; x = y; y = temp;}
max = y;
```

Les trois instructions dans ce bloc trient les valeurs de `x` et `y` dans l'ordre croissant. Ce programme exécute les trois instructions ou aucune d'entre elles.

La variable `temp` est déclarée à l'intérieur du bloc. Cette dernière est locale au bloc ; elle n'existe que pendant l'exécution du bloc.
Une fois sortie du bloc la variable `temp` est efface de la mémoire de l'ordinateur.
Si la condition est fausse `(x < y)`, la variable `temp` ne sera pas créée.

La *portée* d'une variable représente la partie du programme dans laquelle la variable peut être utilisée à partir de l'emplacement de sa déclaration jusqu'à la fin du bloc que cette déclaration contrôle. Cette variable existe dans ce bloc et disparaît une fois le bloc d'instruction exécuté. Vous pouvez utiliser un bloc pour limiter la portée d'une variable, permettant ainsi l'utilisation du même nom pour différentes variables dans divers endroits d'un programme.

voir Bloc.cpp et Portee.cpp

Conditions imbriquées

L'instruction conditionnelle peut être utilisée dans une autre instruction conditionnelle.

Faîtes très attention !

```
if (condition1)
  if (condition2)
    instructionA;
  else
    instructionB;
instructionC;
```

est différent de

```
if (condition1) {
  if (condition2)
    instructionA;
}
else
  instructionB;
instructionC;
```

L'`instructionC` est toujours exécutée,

à gauche: l'`instructionB` est exécutée si la `condition1` est vraie et si la `condition2` est fausse.

à droite: l'`instructionB` est exécutée si la `condition1` est fausse.

L'instruction `else` est toujours associée à la dernière instruction `if`.

Nous avons utilisé des accolades `{ }` pour modifier la portée des conditions.

Pour éviter des résultats non voulus, utilisez toujours les accolades pour résoudre les ambiguïtés.
Faîtes attention à la portée.

Le code suivant est identique au code à gauche, mais ce qui se passe est plus clair.

```
if (condition1) {
  if (condition2)
    instructionA;
  else
    instructionB; }
instructionC;
```

* L'instruction `switch`

Afin d'implémenter une série d'alternatives, on peut utiliser l'instruction `switch` au lieu de la structure `else if` :

```
switch (expression) {
  case const1: instructions1;
  . . .
  case constn: instructionsn;
  default: instructions0;
}
```

L' `expression` est évaluée et sa valeur recherchée parmi les constantes `case`.

Si sa valeur est rencontrée, le programme saut à cette ligne et toutes les instructions suivantes sont exécutées. Dans le cas contraire, si un `default` est présent (facultatif), l'exécution saute à la dernière instruction précédée per `default`.

L'évaluation de l'`expression` est effectuée dans un type entier (y compris le `char`) et les constantes sont des constantes entières.

Pour exécuter la seule instruction correspondante à `case`, ajouter l'instruction `break`; après `break` le programme saute en-dehors de la structure `switch`,

e.g.

```
case constj: instructionsj; break;
```

voir: Calculatrice_v0.cpp et Calculatrice.cpp

* L'opérateur conditionnel ? :

Les opérateurs rencontrés jusqu'ici agissent sur un (unaire) ou deux (binaire) termes. L'opérateur conditionnel (? :) est le seul opérateur C++ qui comprend trois termes (opérateur ternaire).

Il prend en charge trois expressions et retourne une valeur:

```
(condition) ? (expression1) : (expression2)
```

si `condition` est vraie, il retourne la valeur de `expression1`;
sinon, il retourne la valeur de `expression2`.
La valeur renvoyée est généralement affectée à une variable.

Au lieu d'écrire

```
if (x > y) {int temp = x; x = y; y = temp;}
max = x;   cout << max;
```

on obtient le même résultat avec

```
max = (x > y) ? x : y;   cout << max;
```

En général, la première version est préférable parce que plus lisible.

Développement d'un programme

Ecrivez un programme qui trouve les racines des polynômes de second degré :

$$a\,x^2 + b\,x + c = 0 \quad \text{(a, b, c sont des nombres réels)}$$

En général, avant de taper le programme il faut
- comprendre et analyser le problème
- identifier toutes les variables impliquées
- faire tous les calculs avec le crayon (ou trouver toutes les équations nécessaires)
- dessiner le programme sur une feuille (*flow chart* : diagramme de flux),
  i.e. considérer en détail le déroulement du programme :

1. déclarer des variables pour les coefficients a, b, c
   puis saisir les valeurs de a, b, c depuis le clavier (`cin`)

2. vérifier la valeur de a : si a est nul, l'équation n'est pas du second degré
(situation banale, mais si a est le résultat d'un calcul précédent, il peut être égale à 0 :
$\rightarrow$ division par 0 $\rightarrow$ erreur d'exécution ; difficile à trouver dans le code !!!)

3. vérifier la valeur du discriminant $\Delta = b^2 - 4ac$:
   si Δ est positif, il y a deux solutions réelles,
   si Δ est égal à 0, les deux solutions sont identiques,
   si Δ est négatif, il n'y a pas de solutions réelles.

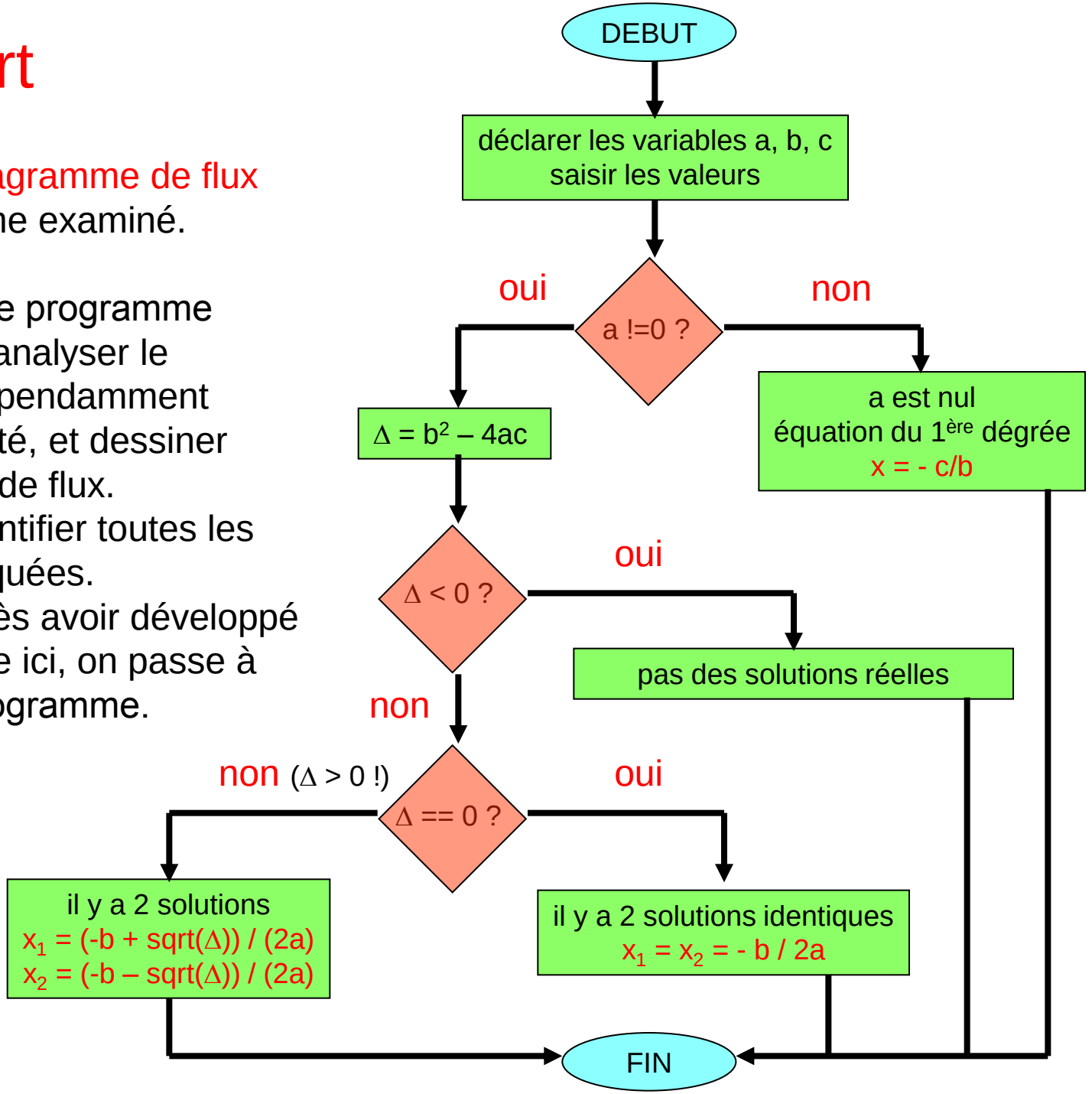
4. afficher les résultats sur l'écran (cout)

Cette analyse nous permet de développer le «pseudo-code» avant d'écrire le programme.

Flow chart

Exemple de diagramme de flux pour le problème examiné.

Avant d'écrire le programme il faut toujours analyser le problème, indépendamment de sa complexité, et dessiner un diagramme de flux.
Il faut aussi identifier toutes les variables impliquées.
Seulement après avoir développé le code, comme ici, on passe à l'écriture du programme.



```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main() {
  cout << "Entrez les coefficients de l'equation :" << endl;
  double a, b, c;
  cin >> a >> b >> c;

  if (a != 0.) {
    double delta;
    delta = b*b - 4.*a*c;
    if (delta < 0.)
      cout << "Il n'y a pas des solutions reelles !" << endl;
    else if (delta == 0.) {
      double x;
      x = -b/(2.*a);
      cout << "Il y a deux solutions identiques : " << x << endl;
    }
    else {       //delta > 0.
      double x1, x2;
      x1 = (-b + sqrt(delta)) / (2.*a);
      x2 = (-b - sqrt(delta)) / (2.*a);
      cout << "Les solutions sont : " << x1 << " et " << x2 << endl;
    }
  }
  else cout << "a est nul, il s'agit d'une equation de 1ere degree !\n";

  return 0;
}
```

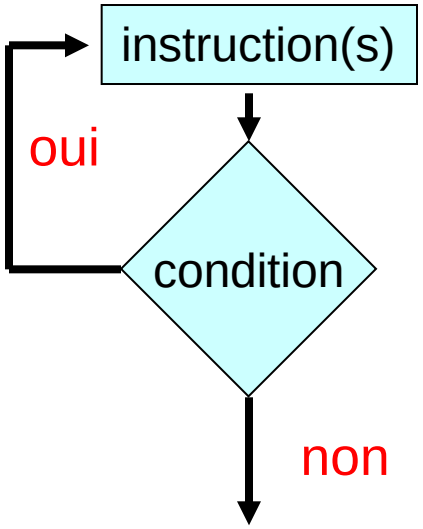
voir : RacinesPolynome.cpp

Itérations

Répétition d'une instruction ou d'un bloc d' instructions
1) plusieurs fois ou
2) tant qu'une condition est vérifiée

développement des algorithmes

boucle



par exemple:

1) addition des n premiers termes de la série $\sum_{i=1}^{n} \frac{1}{i^2}$

```
S = 0, i = 1
S = S + 1. / i2
i = i + 1
si (i <= n)
```

attention : `1.` et non `1` !

n fois, i représente aussi la variable de « contrôle »

2) calcul de $\sqrt{2}$ avec l'algorithme Babylonien :

```
y = 1.
y = y / 2. + 1. / y
si (|y – 2./y| >  ε)
```

condition ε

itération	y (= √2)	|y-2/y|
0	1.0	0.41421
1	1.5	0.16666
2	1.41666	0.00490
3	1.41421	4×10^{-6}

voir Babylon.cpp, Fibonacci.cpp, Euclide.cpp

Instruction `while`

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

Le programme évalue d'abord la `condition`. Si elle est `true` (vraie), l'`instruction` ou bloc d'instructions est exécutée et la condition est évaluée à nouveau.
Dans le bloc d'instruction on peut modifier les données utilisées pour évaluer la condition.
Le bloc d'instructions sera répété (itéré) tant que la `condition` sera `true` (vraie).

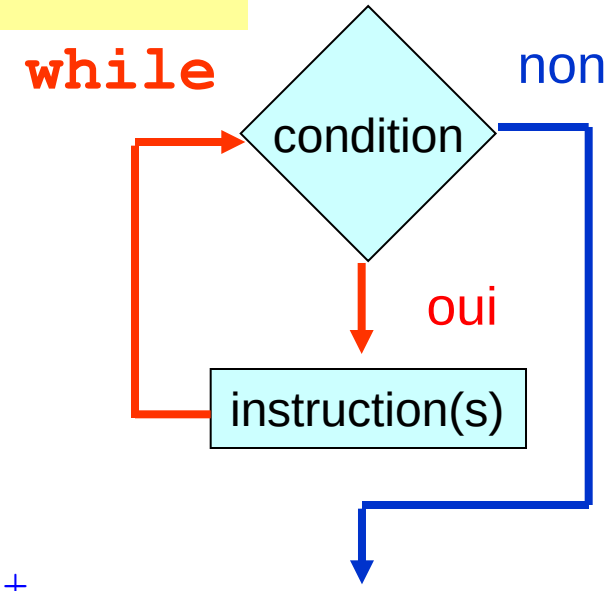
```
cout << "Entrez des entiers > 0, terminez par 0 ";
int n;    cin >> n;
while (n > 0) {
  cout << "Le carre est" << n * n << endl;
  cout << "Entrez n ";    cin >> n; }
```

Ex. : addition des n premiers termes de la série $\sum_{i=1}^{n} \frac{1}{i^2}$

```
int n;    cin >> n;
double s = 0.;
int i = 1;
while (i <= n) {
  s = s + 1./(i*i);
  i++;
}
```

le résultat 1 / i est un entier,
donc 1 / (i*i) = 0 !!!
le résultat 1. / i est un float,
donc 1. / (i*i) ≠ 0 !!!

variable de contrôle `i` avec l'opérateur `++`



voire Serie.cpp

Les opérateurs ++ et --

Dans les boucles on a souvent besoin d'incrémenter (ou décrémenter) la variable de contrôle de 1 (i.e. le compteur de nombre d'itérations).
Pour en faciliter la tache, on a introduit les opérateurs `++` e `--` :

`++` est l'opérateur de pré / post incrémentation et

`--` est l'opérateur de pré / post décrémentation

`i++` addition `1` à `i` (`i--` soustraie `1` à `i`)

aussi `i = i + 1` `i = i - 1`
ou `i += 1` addition `1` à `i` `i -= 1` soustraie `1` à `i`

mais `i++` et `++i` sont un peu différents, p.ex. les instructions :

```
j = i++;
```

```
j = ++i;
```

sont équivalentes à

```
j = i;
i = i + 1;
```

post – incrémentation

```
i = i + 1;
j = i;
```

pré – incrémentation

Instruction do ... while

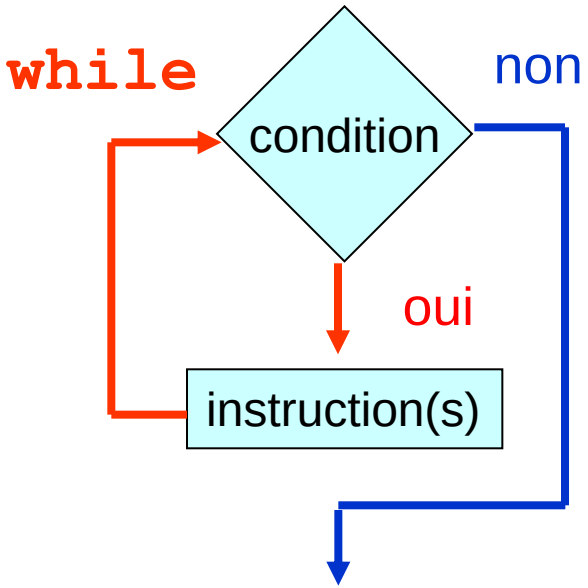
```
do instruction(s) while (condition) ;
```

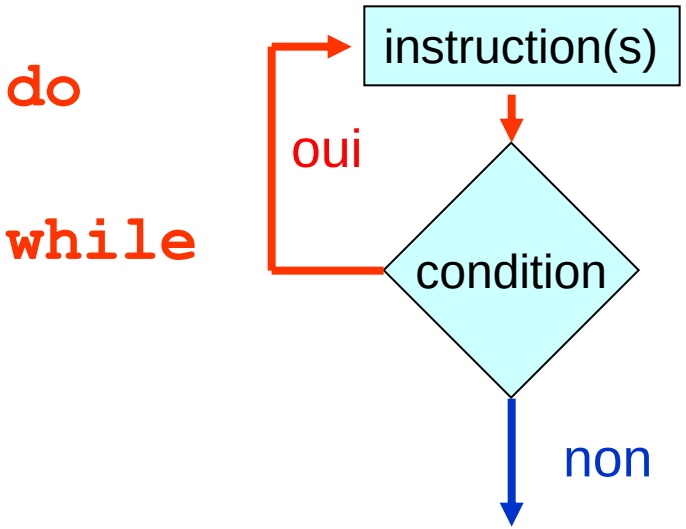
L'instruction `do ... while` est similaire à l'instruction `while`.
La différence principale est que `do ... while` exécute l'`instruction` avant d'évaluer la `condition`.
Donc une boucle `do ... while` sera toujours exécute au moins une fois.

Ex. : Calcul de la fonction factorielle `n!`

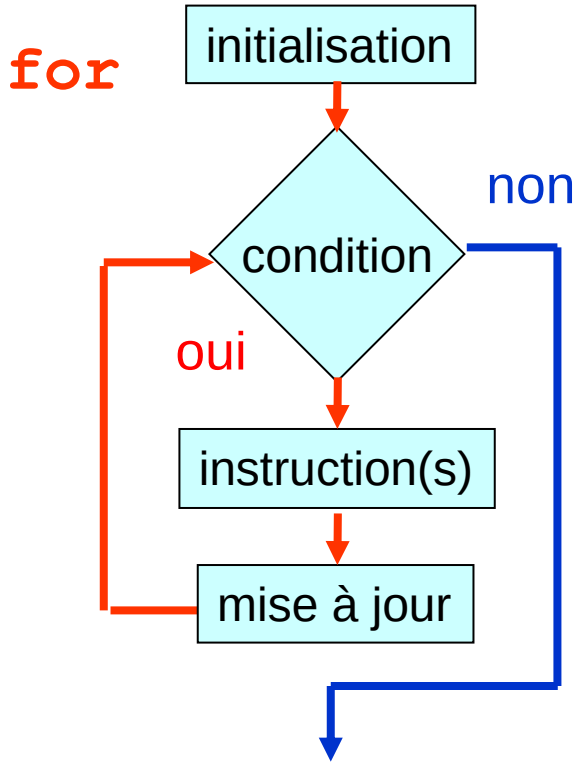
voire Factorielle_do.cpp

```
int n;    cin >> n;
int fact = 1;
do {
   fact = fact * n;
   n--;    //n = n - 1 !
} while (n > 0);
```





Instruction `for`



Une boucle est contrôlée par 3 parties distinctes :

l'*initialisation*

la *condition(s)*

la *mise à jour* (de la variable(s) de contrôle)

Lorsque ces trois étapes sont évidentes,
on utilise la boucle `for` :

```
for(initialisation; condition; mise à jour) instruction(s);
```

Chaque champ correspond à une instruction, donc ils sont séparés par le `;` .

La mise à jour est effectuée à la fin du bloc d'instructions.
Chaque champ peut être vide, mais le `;` doit être présent.

En général tout boucles peuvent être programme avec la boucle **`for`** .

Elle est plus simple à implémenter.
Si vous pouvez choisir entre une boucle `for`, `while` ou `do ... while`,
préférez la boucle `for`.

Exemples des boucles `for`

1) Fonction factorielle :

```
int fact = 1;
for (int i=1; i <= n; i++) {
  fact = fact * i;
}
```

voir Factorielle_for.cpp

Attention : La variable de contrôle `i` est déclarée dans le bloc `for`, donc elle n'existe pas en-dehors de ce bloc `for`. Elle peut être utilisée plusieurs fois, mais elle ne peut pas être déclarée deux fois dans le même bloc !
La variable `fact` doit être initialisée en-dehors de la boucle `for`.
Les parenthèses { } peuvent être omises (une seul instruction)

2) Addition des n (=100) premiers termes d'une série :

```
int somme = 0;
for (int i=0; i <= 100; i++)
  somme = somme + i * i;
```

Les boucles (résumé)

En C++ on peut utiliser trois structures de contrôle différentes pour construire une boucle : `while`, `do-while`, et `for`.

En général, chaque boucle peut être exprimée avec une de ces structures.

calcul de la factorielle avec une boucle `while`

```
int fact = 1;
int i = 1;
while (i <= n) {
  fact = fact * i;
  i++;
}
```

initialisation de la variable de contrôle `i`

évaluation de la condition

calculs

mise à jour de la variable de contrôle `i`

calcul de la factorielle avec une boucle `for`

```
int fact = 1;
for (int i=1; i <= n; i++)
        fact = fact * i;
```

initialisation de la variable de contrôle `i`,

condition,

la mise à jour de la variable de contrôle `i`

sont tous dans le même endroit dans la boucle `for`.

Il est préférable de grouper toutes les instructions de contrôle au même endroit, donc préférez la boucle `for`.

Pour l'instant il faut savoir écrire des boucles.

Les détails et les différences deviennent plus claires avec la pratique.

voir Factorielle_while.cpp, Factorielle_do.cpp, Factorielle_for.cpp

* Les boucles : `break` et `continue`

On peut contrôler ou modifier l'exécution d'une boucle avec les instructions `break` et `continue`.

L'instruction `break` est utilisée pour sortir d'une boucle (i.e. interrompre l'exécution de la boucle). L'instruction `break` ignore le reste des instructions dans la boucle et le programme saut directement à la première instruction en-dehors de la boucle.

```
while (condition1) {
  instruction1;
  if(condition2) break;
  instruction2;}
instruction3;
```

si la `condition2` est vraie (true), l'exécution de la boucle est interrompue et le programme saute à l'`instruction3` en-dehors de la boucle.

L'instruction `continue` est utilisée pour retourner au début de la boucle et re-commencer l'itération suivante sans exécuter les instructions qui suivent le `continue`.

```
while (condition1) {
  instruction1;
  if (condition2) continue;
    instruction2;}
instruction3;
```

si la `condition2` est vraie (true), l'exécution de la boucle retourne au début et l'`instruction2` n'est pas exécutée cette fois; l'exécution de la boucle n'est pas interrompue.

Il existe aussi l'instruction `goto` qui permet de sauter d'un point à un autre dans le programme (spagetti code, à éviter).

voir Tasse.cpp

* Boucles infinies

Somme des n premiers termes d'une série : programmes équivalents

```
while (1) { //boucle infinie
  if (i > n) break;    //condition
  somme = somme + i*i;
  i++;
}
```

```
while (i <= n) {
  somme += i*i;
  i++;
}
```

Les deux boucles sont équivalents, le code à droite est préférable,
mais quelquefois on a besoin d'introduire des contrôles supplémentaires.
à noter : `while (1)`, la condition est toujours vraie !

On peut aussi construire une boucle infinie avec `for` :

```
for(;;) {
  if (i > n) break;
  somme = somme + i*i;
  i++;
}
```

```
for (i=0; i <= n; i++)
  somme += i*i;
```

Les deux boucles sont équivalents entre eux et aussi aux premiers. Le code à droite est toujours préférable (si possible utilisez toujours `for` au lieu de `while`).
à noter : l'instruction `for` utilise trois champs (initialisation, condition, incrémentation) qui peuvent être laissés vides. Dans ce cas la conditions est toujours vrai.

Boucles imbriquées

On peut aussi imbriquer des boucles, p.ex. pour parcourir un tableau de nombres.

Le programme suivant affiche une table de multiplication 12 × 12.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
  for (int i=1; i<=12; i++) {
    for (int j=1, j<=12; j++)
      cout << setw(4) << i*j;
    cout << endl;
  }
  return 0;
}
```

voir Table.cpp

1ère boucle – boucle externe

2ème boucle – boucle interne (boucle imbriquée)
pas d'accolades pars qu' il s'agit d'une seul instruction

(`setw` est une fonction pour la mise en page défini dans la bibliothèque `iomanip`)

Tous les types de boucles rencontrés (`while`, `do-while` et `for`) peuvent être imbriqués.

voir DeuxBoucles.cpp

Mots-clés du langage C++

Les mots-clés sont interprétés par le compilateur comme des éléments intrinsèques au langage de programmation. Les mots-clés ne peuvent donc pas être utilisés comme noms de classes, de variables ou de fonctions.

asm	do	if	return	try
auto	double	inline	short	typedef
bool	dynamic_cast	int	signed	typeid
break	else	long	sizeof	typename
case	enum	mutable	static	union
catch	explicit	namespace	static_cast	unsigned
char	export	new	struct	using
class	extern	operator	switch	virtual
const	false	private	template	void
const_cast	float	protected	this	volatile
continue	for	public	throw	wchar_t
default	friend	register	true	while
delete	goto	reinterpret_cast		

mots-clés déjà vus

mots-clés rencontrés aujourd'hui

Résumé

Ce qu'il faut retenir / savoir faire à la fin de cette leçon :

Le conditions : `if`
`if ... else`

Opérateurs relationnels et logiques

Les boucles : `while`
`do... while`
`for`

Les instructions `break` et `continue`

Toujours analyser le problème et dessiner un diagramme de flux.

Dans la doute toujours écrire un petit programme pour tester les « instructions ».

Mémorisez le programme suivant !

Il contient tous les élément importants vus aujourd'hui (les trois boucles différentes).

```
//calcul de la factorielle
#include <iostream>
using namespace std;

int main() {
  int i = 1;   //variable de controle pour la premiere boucle
  int n;
  do {
    //saisi d'un entier > 0
    cout << "Calcul de la factorielle, entrez n (> 0) : " ;
    cin >> n;
    while (n <= 0) {
      cout << "n est <= 0 ! Entrez un autre nombre > 0 : " ;
      cin >> n;
    }
    //calcul de la factorielle (1 * 2 * ... * n-1 * n)
    int fact = 1;
    for (int k=1; k <= n; k++)
      fact = fact * k;
    cout << "La factorielle de " << n << " est : " << fact << endl;
    //refaire le calcul ?
    cout << "Voulez vous continuer (1 pour continuer, 0 pour sortir) ? ";
    cin >> i;
  } while (i);

  return 0;
}
```

Factorielle_v2.cpp

Exercices – série 2

Questions

1. Déterminez si chacune des expressions suivantes est vraie ou fausse :
   a) `!(p || q)` est identique à `!p || !q`
   b) `!!!p` est identique à `!p`
   c) `p && q || r` est identique à `p && (q || r)`

2. Quel est le nombre minimal d'itérations que peut faire une boucle `while` ?
Et une boucle `do . . . while` ?

3. Quel est le problème dans la boucle `while (n<100) sum += n*n;` ?

4. Comment peut-on écrire une boucle de sorte qu'elle se termine avec une instruction située au milieu du bloc correspondant ?

5. Pourquoi utiliser des parenthèses superflues lorsque la priorité des opérateurs est évidente ?

6. Soit *inst* une instruction et *exp* une expression. Comment peut-on écrire les boucles suivantes de manière équivalente à `while` ?
   `for( ; exp; ) inst;` et `for( ; ; exp) inst;`

7. De même, quel est l'équivalent de : `for(exp1; exp2; exp3) inst;`
Que se passe-t-il si *inst* contient une instruction `continue` ?

8. Si les opérateurs relationnels renvoient `true` ou `false`, pourquoi une valeur différente de zéro est-elle considérée comme vraie ?
Les nombres négatifs peuvent-ils posséder les valeurs `true` ou `false` ?

9. Essayez d'utiliser l'opérateur conditionnel `? :`

Trouvez l'erreur !

1.
```
if (x=0) cout << x << "=0\n";
else cout << x << "!= 0 \n";
```

2.
```
if (x<y<z) cout << x << "<" << y << "<" << z << endl;
```

3.
```
if (x==0)
  if (y==0) cout << "x et y sont egaux a zero" << endl;
else cout << "x n'est pas egal a zero" << endl;
```

4.
```
int compteur = 0
while (compteur < 10)
  cout << "compteur : " << compteur;
```

5.
```
for (int compteur = 0; compteur < 10; compteur++);
  cout << compteur << endl;
```

6.
```
int compteur = 100;
while (compteur < 10) {
  cout << "le compteur indique : " << compteur << endl;
  compteur--;
}
```

Exercices avec `if`

1. Etudiez : Ex_if.cpp et Ex_if-else.cpp (p. 9)
   Etudiez : Bloc.cpp et Portee.cpp (p. 13)
   Etudiez et complétez : Calculatrice.cpp (p. 15)

2. Trouvez le maximum et minimum parmi 3 nombres :

```
cin >> a >> b >> c;
if (a > b)
  if (a > c) max = a;
  else max = c;
else
  if (b > c) max = b;
  else max = c;
cout << min  << max;
```

et avec les opérateurs logiques

```
if (a > b && a > c) max = a;
```

Essayez les deux !
(Développez ces programmes)

3. Que fait le code suivant?

```
if (a > 0) if (b > 0) a = a + 1; else if (c > 0)
if (a < 4) b = b + 1; else if (b < 4) c = c + 1;
else a = a - 1; else if (c < 4) b = b - 1; else c = c -1;
else a = 0;
```

C'est un exemple de code très mal écrit et presque impossible à comprendre. Mais il est correct du point de vu de la syntaxe C++. Réécrivez-le de façon plus lisible.

4. Ecrivez un programme qui lit un entier à 6 chiffres et affiche la somme de ces 6 chiffres
Utilisez les opérateurs `/` et `%` pour extraire les chiffres (reste de la division par 10).

Exercices avec les boucles

1. Développez l'exemple 1 à la p. 20 et 21 : addition des n premiers termes d'une série.

2. Développez l'exemple 2 à la p. 20 et 21 : calcul de $\sqrt{2}$ avec l'algorithme Babylonien.

3. Développez un algorithme pour le calcul de la $\sqrt{}$ des nombres entiers.

4. Etudiez et compilez les algorithmes : Babylon.cpp, Fibonacci.cpp, Euclide.cpp

5. Ecrivez un programme pour calculer la factorielle de n de 3 manières différentes (boucles : `while`, `do...while`, `for`) (p. 26);
voir Factorielle_while.cpp, Factorielle_do.cpp, Factorielle_for.cpp

6. Ecrivez un programme pour additionner les n premiers termes d'une série,
p.ex. $\sum_{i=1}^{n} \frac{(-1)^i}{i^2}$, de 3 manières différentes (boucles : `while`, `do...while`, `for`).

Problèmes

1. Rebond d'une balle : à chaque rebond la balle perd une fraction de son énergie initiale. Si la balle tombe d'une hauteur h et la perte d'énergie au rebond est ε ($g = 9.8\ m/s^2$) :

   a) Quelle est la hauteur maximale attente après n rebonds ?

   b) Après combien de rebonds la hauteur maximale sera 1% de la hauteur initiale ?

2. Estimation de π : calculez la surface d'un cercle par « intégration numérique ». Développement de l'algorithme (indications) :

   1) inscrivez un cercle dans un carré de côté 1,

   2) divisez le carré en $n \times n$ petits carrés (la surface de chaque carré est $1/n^2$),

   3) comptez le nombre des ces petits carrés dans le cercle.

Comparez le résultat obtenu à la valeur de π.