# **Résumé de programmation – semaines 0-5**

# Variables

Si on veut résoudre une équation du type

1. Demander les coefficients b et c
2. Calculer le Delta =
3. Si Delta < 0 afficher " pas de solution "
4. Si Delta = 0 afficher " solution unique "

Une variable possède 3 caractéristiques

* Son identificateur (le nom par lequel la variable est désignée)
* Son type (de quel "genre" est la variable)
* Sa valeur

## Déclaration de variable

int n(4); 🡪 n est l'identificateur

int n\_carre; 🡪 int est le type (ici type entier) 🡪 n\_carre ne peut contenir que des entiers

double x(1.0); 🡪 double est le type des variables avec décimales

Une déclaration de variable suit le schéma:

type identificateur(valeur\_initiale) ;

## Types de variables

int valeurs entières

double nombres à virgule

insigned int entiers positifs

char caractères (A, Z etc.)

bool valeur true ou valeur false

const type constante

## Ordre des types

Les types ont une hiérarchie. Celle-ci est utile en cas d'expression contenant deux types de variables différents.

Exemple ( 2 / 3.0 ) va renvoyer une variable double alors que (2 / 3) va renvoyer une variable int, et l'expression vaudra ici 0. Cela est dû au fait que double est plus haut dans la hiérarchie que int.

int 🡪 long 🡪 long long 🡪 float 🡪 double 🡪 long double

## Affectation

L'affectation s'effectue au moyen du signe =

Exemple:

n\_carre = n\*n ; ATTENTION ce n'est pas une égalité mathématique

L'affectation suit le schéma:

nom\_de\_variable = expression

## Ecrire à l'écran

cout << "la variable n contient" << n << "." << endl ;

cout représente la fenêtre terminal

ce qui est affiché entre guillemets est affiché littéralement

ATTENTION on ne peut utiliser cout et endl (dans leur forme simple) SEULEMENT si le début du programme contient la ligne

using namespace std

sinon il faudrait écrire std: :cout et std: :endl

## Demande d'entrer une valeur

int n ;

cout << "Entrez une valeur pour n" ;

cin >> n ;

## Nombres et leurs types

1. int
2. double
3. double

On peut utiliser la notation scientifique

Exemple: 2e3 = 2\*10^3 = 2000

ATTENTION dans le type int:

1 / 2 = 0, 5 / 2 = 2, mais 1 / 2.0 = 0.5

+= -= \*= /= sont des raccourcis pour ne pas devoir trop écrire

Exemple

a += 5 ⬄ a = a + 5

a \*= b ⬄ a = a\*b

## Opérateur modulo %

% renvoie le reste de la division entière 🡪 11 % 4 = 3 car 11 = (4 \* 2) + 3

++ et – sont les signes de l'incrémentation et de la décrémentation

Exemple ++i ⬄ i = i + 1

ATTENTION pour inclure les fonctions mathématiques dans un programme il faut qu'au début du programme il y ait la ligne

#include <cmath>

Variables dans des blocs

Quand une variable est initialisée dans un bloc, par exemple

if (condition) {

int variable\_une (0);

…

}

elle ne peut pas être utilisée dans la suite du programme. Ce type de variable est appelé variable locale. A l'inverse, les variables globales sont les variables initialisées à l'extérieur de tout bloc. Ne jamais utiliser de variables globales (sauf pour certaines constantes) est une bonne pratique car elles sont accessibles à l'ensemble du programme. Il est préférable de déclarer une variable au plus près de son utilisation.

Lorsque plusieurs variables portent le même nom (ce qui n'est pas une bonne idée), le compilateur va choisir la variable "la plus proche" pour exécuter les instructions demandées.

# Structures de contrôle

## Types de structures de contrôle

Il existe trois types de structures

* Les branchements conditionnels
* Les itérations
* Les boucles conditionnelles

## Branchements conditionnels (if)

if (condition) { si la condition est vérifiée, exécuter les instructions

…

} else { sinon, exécuter ces instructions-ci

…

}

Le bloc else est optionnel. Si tel est le cas et que la condition n'est pas vérifiée, on ignore le bloc if et on continue.

ATTENTION la condition est toujours exprimée entre parenthèses !

## Valeur des conditions

Une condition ne peut prendre que deux valeurs: true ou false

Exemple n = 0 et condition: ( n < 5 ) 🡪 la condition renvoie la valeur true

## Opérateurs de comparaison

< inférieur à

> supérieur à

== égal à ATTENTION à ne pas confondre == et = ( qui attribue une valeur)

<= inférieur ou égal à

>= supérieur ou égal à

!= différent de

On peut relier ces conditions simples grâce à des opérateurs logiques (and, or et not)

(a < b ) and ( c < d ) est vraie uniquement si les deux conditions sont vraies

(a < b ) or ( c < d ) est vraie si au moins une des deux conditions est vraie

not ( a < b ) est vraie si a >= b et fausse si a < b

not peut aussi s'écrire ! ⬄ != veut dire "différent de"

## Type bool

Le type bool permet de déclarer des variables contenant la valeur d'une condition (true ou false)

Exemple int a(1) ;

int b(2) ;

bool test1(a == b) ; test1 prend la valeur false

bool test2(a < b) ; test2 prend la valeur true

if (test2) {

…

}

## Conversion entre type de données et type booléen

Dans une expression logique :

* Motif binaire nul (0, 0.0, …) 🡪 Booléen false
* Autre motif binaire 🡪 Booléen true

## Dans une expression arithmétique :

* False 🡪 0
* True 🡪 1

## Style

Pour une cascade de conditions, utiliser la syntaxe else if

# Itérations

## Boucle for

La boucle for permet de répéter une même série d'instructions.

Exemple: for (int i(0) ; i < 5 ; ++i) {

cout << "le carre de " << i << "vaut" << i\*i << endl ;

}

Ici " for (int i(0) ; i < 5 ; ++i) " veut dire que l'on va répéter l'instruction pour i allant de 0 à 5

ATTENTION la variable i n'existe que dans le corps de la boucle, on ne peut pas l'utiliser dans la suite du code

La boucle for suit la syntaxe

for (declaration\_et\_initialisation; condition; incrémentation) {

instructions

{

On peut incrémenter i de plus que 1 à la fin de la boucle for (par exemple i += 2, i \*=5, --i)

Attention à ce que la condition de répétition de la boucle ne soit pas toujours vraie car dans ce cas la boucle va se répéter indéfiniment !

On peut imbriquer des boucles les unes dans les autres, mais attention à ne pas abuser de ce processus car le code devient vite illisible !

La boucle for est utile quand on connaît à l'avance le nombre de répétitions. Si ce n'est pas le cas on utilise la boucle while

## La boucle while (ou do… while)

do {

instructions

} while (condition) ;

La condition renvoie une valeur true ou false. Tant que la condition renvoie true la boucle va se répéter.

La condition utilise des opérateurs logiques (comme les autres conditions)

Les instructions à l'intérieur de la boucle do… while sont toujours exécutées au moins une fois, puisque la condition se situe à la fin de la boucle.

Si la condition de la boucle n'est jamais fausse, les instructions seront répétées indéfiniment !

Le boucle while existe aussi sous la forme

while (condition) {

…

}

Cela permet, avec les mêmes propriétés, de tester la condition avant d'entrer dans la boucle, et donc de sauter la boucle si la condition est fausse.

ATTENTION dans la boucle do… while il y a un point-virgule à la fin de la ligne while, ce qui n'est pas le cas dans la boucle while (avec la condition au début)

# Fonctions

Si une tâche doit être exécutée plusieurs fois dans un programme, il faut la mettre dans une fonction au lieu de simplement la recopier

ATTENTION ne jamais recopier ("copier-coller") de lignes de code !

Une fonction est une portion de programme réutilisable caractérisée par:

* Un corps (portion de programme à réutiliser)
* Un nom (par lequel on désigne u appelle la fonction)
* Des paramètres (ensemble de variables extérieures à la fonction dont le corps dépend pour fonctionner)
* Une valeur de retour (la valeur que la fonction renvoie au programme)
* Un type (int, double etc. 🡪 dépend de la valeur renvoyée)

L'utilisation de la fonction dans une autre partie du programme se nomme un appel de fonction.

## Facettes d'une fonction

"prototype" 🡪 Résumé, contrat

double fonction\_1(double x, int y) ; fonction\_1 est le nom de la

fonction, x, y , a et b sont des

"Définition" 🡪 Construction, création paramètres

double fonction\_1(double a, double b)

{

Instructions du corps de la fonction:

Return expressions:

}

"Appel" 🡪 Utilisation

z = fonction\_1(2\*u, v+3);

## Appel d'une fonction

1. Les expressions passées en argument sont évaluées (argument = paramètre lors de l'appel de la fonction)
2. Les valeurs correspondantes sont affectées aux paramètres de la fonction
3. Le corps de la fonction est exécuté
4. L'expression suivant la première commande return rencontrée est évaluée…
5. … et retournée comme résultat de l'appel. Cette valeur remplace l'expression de l'appel

Une fonction peut tout à fait appeler une autre fonction. Il faut simplement respecter le fait de prototyper la fonction avant l'appel.

## Passages d'arguments

**Passage par valeur :**

void f(int x) {

x = x + 1;

}

…

int main () {

int val(1);

f(val);

}

Ici la variable x est locale à la fonction f, c’est-à-dire qu'on l'utilise comme une copie de la variable val. A la fin de ce programme x = 2 et val = 1.

**Passage par référence:**

void f(int& x) {

x = x + 1;

}

…

int main () {

int val(1);

f(val);

}

Ici on définit x comme un autre nom pour val, cela veut dire que lorsqu'on incrémente x on incrémente val également. A la fin de ce programme x = val = 2.

Le passage par référence est plus "typé" que le passage par valeur. Cela signifie qu'il exige plus de précision concernant le type des valeurs envoyées dans la fonction. On ne peut pas envoyer un double si la fonction attend un int par exemple (ce qui est possible lors d'un passage par valeur).

## Arguments par défaut

Certains paramètres n'ont pas besoin d'argument. La syntaxe d'un paramètre avec valeur par défaut est

type nom = valeur

ATTENTION les paramètres avec valeur par défaut doivent apparaître en dernier dans la liste des paramètres d'une fonction.

Exemple:

void affiche\_ligne( char elt, int nb = 5) ;

Si on ne précise rien, la valeur par défaut est utilisée, mais on peut changer cette valeur lors de l'appel de la fonction.

Affiche\_ligne('A' , 8); ici nb passe à 8

ATTENTION les arguments par défaut se spécifient dans le prototype de la fonction et pas dans la définition.

## Surcharge de fonctions

Il est possible de définir plusieurs fonctions de même nom qui n'ont pas les mêmes paramètres. Cela est utile pour utiliser un même traitement sur plusieurs types de données.

Exemple

void affiche( int x) {

}

void affiche( double x) {

}

## Jargon

* "Appeler la fonction f" = utiliser la fonction f
* "3 est passé en argument" = Lors d'un appel la valeur 3 est copiée dans un paramètre de la fonction
* "La fonction retourne la valeur de y" = L'expression de l'appel de la fonction sera remplacé par la valeur retournée