Introduction à la Programmation Impérative

Nicolas Thiéry et al.

sept. 27, 2022

Programme hebdomadaire

CHAPITRE 1

Programmation impérative (Info 111)

Nicolas Thiéry

http://Nicolas.Thiery.name/Enseignement/Info111

Pourquoi enseigner l'informatique?

Évidence : l'ordinateur est partout!

- Combien d'ordinateurs dans la salle?
- Combien d'ordinateurs possédez vous?
- Le mot «assisté par ordinateur» a disparu
- Usage constant des ordinateurs, pour le travail comme le reste

Évidence : tous les jeunes connaissent déjà l'informatique!

Vraiment?

1.1 C'est quoi l'informatique?

Une petite analogie :

- Mr Einstein, vous qui êtes un excellent *physicien*, vous devez savoir changer la roue de ma voiture, non?
- Mr Alonso, vous qui êtes un excellent *conducteur* de F1, vous devez savoir réparer le carburateur de ma voiture, non?

Conducteur $\neq$ Garagiste $\neq$ Physicien

Et pourtant, loin d'être Einstein ou Alonso, …

- Mr Thiéry, vous qui êtes *professeur en informatique*, vous devez savoir réparer mon W…..s, non?

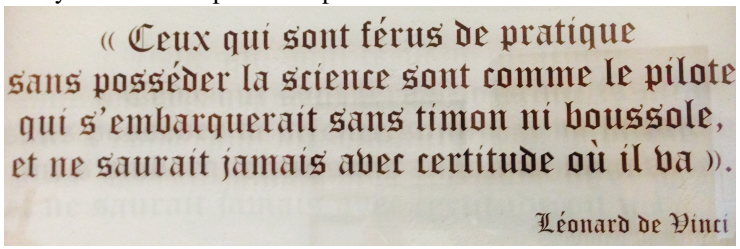
C'est quoi l'informatique en fait?

Suite de la petite analogie :

L'usage	La technologie	La science
Conduite	Réparation, Conception	Physique
Consommation	Cuisine	Chimie, Biologie
Utilisation	Programmation, …	Informatique

Qu'est-ce qu'on apprend à l'école?

- Principalement
- Et il y a des raisons profondes pour cela



- Et il y a des pressions pour que ce ne soit pas le cas …

Quelle école pour la société de l'information? :

Une conférence de François Élie

À lire ou écouter … et méditer …

Tous les jeunes connaissent déjà l'informatique?

L'usage?
- Évidence : tous les jeunes savent utiliser un ordinateur
- Vraiment? les-enfants-ne-savent-pas-se-servir-dun-ordinateur

La technologie?
- Qui sait programmer? Configurer un réseau?

La science?

Ma petite expérience :
- 6ème : 3ème
- Fac : apprendre la *science* a chamboulé ma programmation
- 2018 : après 30 ans et 300000 lignes de code, j'apprends encore …

1.2 La science informatique?

- ***Science du calcul et de l'information***
- Notion fondamentale : *étude des systèmes en évolution*
  - État du système avant
  - Étape de calcul
  - État du système après
- Modèles de calcul

1.2.1 Grands thèmes de l'informatique

Calculabilité : Que peut, ou ne peut pas faire, un ordinateur?
- Indépendamment du langage
- Indépendamment du matériel
- Miracle : tous les langages sont équivalents!

Complexité : Combien de ressources pour résoudre un problème?
- Indépendamment du langage
- Indépendamment du matériel
- Indépendamment de l'algorithme?

1.2.2 Grands problèmes de l'informatique

Maîtriser les systèmes extrêmement complexes :
- Internet avec des milliards d'ordinateurs
- Programmes avec des millions de lignes
- Données occupant des petaoctets (10^{15} octets!)
- Services gérant des millions de clients
- Passage à l'échelle

Abstraction :

Exemple : Couches OSI pour les réseaux

Difficulté :

Apprendre des outils conçus pour les programmes de 100000 lignes en travaillant sur des programmes de 10 lignes …

1.2.3 Grands thèmes de l'informatique (suite)

Conceptions des langages de programmation :
- Java, C++, Python, Ada, Pascal, Perl, Camel, Haskell, Go, Rust…
- Un nouveau langage par semaine depuis 50 ans!
- Heureusement les concepts sont presque toujours les mêmes :
  - Programmation impérative
  - Programmation objet
  - Programmation fonctionnelle
  - Programmation logique
  - Orchestration de flots de données
  - Apprentissage
  - Algorithmique et structures de données

Comment mieux s'exprimer pour que l'ordinateur résolve nos problèmes?

1.2.4 Autres grands thèmes de l'informatique

- Architecture des ordinateurs, parallélisme
- Réseaux, transmission de données
- Bases de données
- Langages formels, automates
- Modèles et structures de données
- Sureté et sécurité du logiciel :spécification, test, preuve
- Sureté et sécurité des données :codage, cryptographie
- Mathématiques discrètes : graphes, combinatoire, …

1.3 À propos de ce cours

Au programme :
- *Science* : concepts de la programmation structurée
- *Technologie* : programmation C++ (simple)
- *Usage* : environnement de programmation, GNU/Linux

Ce que l'on va voir :
- Les briques de bases, les règles de compositions
- Les constructions usuelles
- Les problèmes déjà résolus, les erreurs les plus courantes

Du TD? Pour quoi faire??
- Bénéficier de l'expérience de plus de 50 ans de programmation
- Intuition de ce qui est possible … ou pas
- Intuition de comment résoudre un nouveau problème

1.3.1 Organisation du cours

1h30 amphi, 1h30 TD, 2h TP

Du TD? pour quoi faire???
- Apprendre la science informatique, en utilisant un ordinateur, pour programmer …
- Comme apprendre la physique, au volant d'une voiture …
- C'est pas facile …

Une difficulté : la forte hétérogénéité de niveau : Ce module s'adresse à tous, *débutants* comme *expérimentés*

Évaluation :
- 25% : Examen mi-semestre (dans l'axe des TD)
- 40% : Examen final (vision d'ensemble)
- 20% : Projet en fin de semestre
- 15% : Exercices en ligne, notes de TP

1.4 Ordinateurs et traitement automatique des informations

Exemples d'ordinateurs :
- Calculatrice (programmable)
- Ordinateur personnel (PC, Mac, …)
- Station de travail (Sun, DEC, HP, …)
- Super-ordinateur (Cray, IBM-SP, …)
- Clusters d'ordinateurs

Mais aussi :
- Puce (programme fixe)
- Tablettes
- Téléphones portables, appareils photos, GPS, lecteurs MP3, …
- Box, routeurs wifi, …
- Téléviseurs, …
- Arduino, Raspberry Pi, …

1.4.1 Caractéristiques principales d'un ordinateur

Absolument stupide :
- Il obéit strictement aux ordres reçus
- Est-ce qu'il fait ce que l'on veut?

Très très rapide :
- 2GHz : 2 milliards d'opérations par seconde

Très très bonne mémoire :
- Bible : Mo (million de caractères)
- Mémoire : Go (milliards de caractères)
- Disque : To (1000 milliards de caractères)
- Data center : Po

1.4.2 À quoi sert un ordinateur?

Stocker des informations :
- Documents, musique, photos, agenda, …

Traiter automatiquement des informations :
- ***Entrée*** : informations venant du clavier, de la souris, de capteurs, de la mémoire, d'autres ordinateurs, …
- Traitement des informations en exécutant un ***programme***
- ***Sortie*** : informations envoyées vers l'écran, la mémoire, d'autres ordinateurs, …

Définition :

Informellement, un ***programme*** est une séquence d'instructions qui spécifie étape par étape les opérations à effectuer pour obtenir à partir des ***entrées*** un résultat (la ***sortie***).

Voir aussi : http://fr.wikipedia.org/wiki/Programme_informatique

1.5 Premiers programmes

Exemples de programmes

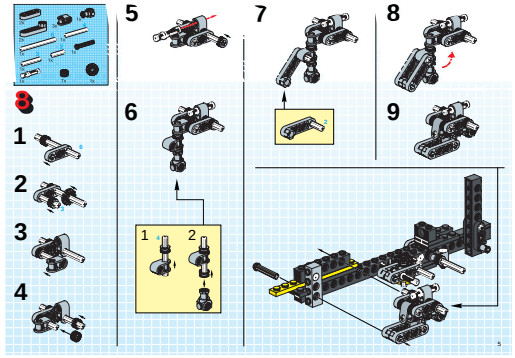
Ingrédients :

250g de chocolat, 125g de beurre, 6 œufs, 50 g de sucre, café

Étapes :
- Faire fondre le chocolat avec 2 cuillères d'eau
- Ajouter le beurre, laisser refroidir puis ajouter les jaunes
- Ajouter le sucre et comme parfum un peu de café
- Battre les blancs jusqu'à former une neige uniforme
- Ajouter au mélange.

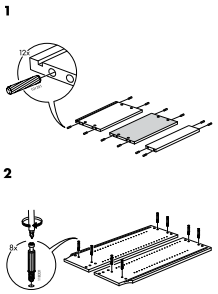
Entrée? Sortie?

Exemples de programmes

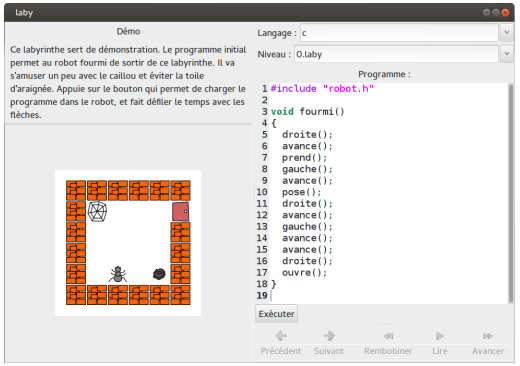


Entrée? Sortie?

Exemples de programmes



Entrée? Sortie?



Entrée? Sortie?

Un exemple de programme C++

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

int main() {
    int x, xCarre, xPuissanceQuatre;

    cout << "Entrez un entier: ";
    cin >> x;

    xCarre = x * x;
    xPuissanceQuatre = xCarre * xCarre;

    cout << "La puissance quatrième de " << x
         << " est " << xPuissanceQuatre << endl;

    return 0;
}
```

Compilation, exécution, … *Un peu lourd? Pas de panique!*

Le coeur du programme C++

```cpp
// Entrée
int x = 5;

// Traitement
int xCarre = x * x;
int xPuissanceQuatre = xCarre * xCarre;

// Sortie
xPuissanceQuatre
```

Exécution dans Jupyter+Cling

Jupyter + Cling : une super calculatrice programmable

Jupyter :
- Un environnement de calcul interactif multi-langage
- http://jupyter.org

Cling :
- Un interpréteur C++
- https://root.cern.ch/cling

Démo

1.6 Résumé

- À propos d'Info 111
  - Qu'est-ce que l'informatique (*Usage, Technologie, Science!*)
  - Objectifs du cours
- Un aperçu de premiers éléments de programmation :
  - Ordinateur
  - Programmes On reviendra dessus!
- Environnement Jupyter+Cling
- Infrastructure du cours

CHAPITRE 2

Premiers éléments de programmation impérative

2.1 Prélude

2.1.1 Résumé des épisodes précédents ...

— Info 111: modalités et infrastructure

— Informatique: usage, technologie, science

— Objectif d'Info 111: initier à la science via la technologie

— Concrètement: bases de la programmation impérative + …

2.2 Rappel

Programme : suite d'instructions exécutées de manière séquentielle (les unes après les autres)

Exemple :

```
debut();
droite();
avance();
prend();
gauche();
avance();
```

(suite sur la page suivante)

(suite de la page précédente)

```
pose();
droite();
avance();
gauche();
avance();
avance();
droite();
ouvre();
```

2.3 Comment rompre la monotonie?

— Faire des calculs : *expressions et variables*

— Découper en petits programmes : les *fonctions*

— S'adapter au contexte : les *instructions conditionnelles*

— Répéter : les *instructions itératives* (boucles)

2.4 Expressions

Expression : Combinaison de *valeurs* par des *opérations* donnant une nouvelle *valeur*.

Exemple : L'expression `3 * (1 + 3) + (1 + 4) * (2 + 4)` vaut `42`

Opérations sur les entiers:

opération	exemple	résultat
opposé	`-(-5)`	5
addition	`17 + 5`	22
soustraction	`17 - 5`	12
multiplication	`17 * 5`	85
division entière	`17 / 5`	3
reste de la division entière	`17 % 5`	2

2.4.1 Aparté : *syntaxe, sémantique, algorithme*

- **Syntaxe** : comment on l'écrit
- **Sémantique** : ce que cela fait
- **Algorithme** : comment c'est fait

Exemple :
- Syntaxe : 17 / 5
- Sémantique : calcule la division entière de 17 par 5
- Algorithme : division euclidienne

2.4.2 Expressions booléennes

Définition :

Une expression dont la valeur est «vrai» ou «faux» (type : `bool`)

Exemples:

```
true
```

```
false
```

- `regarde() == Vide`
- `x > 3.14`
- `2 <= n and n <= 5`

Opérations booléennes usuelles :

opération	exemple	résultat
comparaison	`3 <= 5`	`true`
comparaison stricte	`3 < 5`	`true`
comparaison stricte	`3 > 5`	`false`
égalité	`3 == 5`	`false`
inégalité	`3 != 5`	`true`
négation	`not 3 <= 5`	`false`
et	`3 < 5 and 3 > 5`	`false`
ou	`3 < 5 or 3 > 5`	`true`

2.5 Variables

2.5.1 Exemple

Calculer l'énergie cinétique $\frac{1}{2}mv^2$ d'un objet de masse $14,5$ kg selon qu'il aille à 1, 10, 100, ou 1000 m/s.

```
1./2 * 14.5 * 1 * 1
```

```
1./2 * 14.5 * 10 * 10
```

```
1./2 * 14.5 * 10 * 10
```

```
1./2 * 14.5 * 100 * 100
```

```
double v;
double m;
```

```
v = 1000;
m = 14.5;
```

```
1.0/2.0 * m * v * v
```

2.5.2 Définition

Une ***variable*** est un espace de stockage *nommé* où le programme peut mémoriser une donnée

Le nom de la variable est choisi par le programmeur
- Objectif : stocker des informations durant l'exécution d'un programme
- Analogie : utiliser un récipient pour stocker des ingrédients en cuisine :
  - Verser le sucre dans un *saladier*
  - Ajouter la farine dans le *saladier*
  - Laisser reposer
  - Verser le contenu du *saladier* dans …

2.5.3 Notes

En C++, une variable possède quatre propriétés :
- un **nom** (ou **identificateur**)
- une **adresse**à préciser la semaine prochaine
- un **type**
- une **valeur**

La valeur peut changer en cours d'exécution du programme (d'où le nom de variable)

2.5.4 Notion de type

Les variables peuvent contenir toutes sortes de données différentes :
- nombres entiers, réels, booléens, …
- textes
- relevés de notes, images, musiques, …

Définition :
- Une variable C++ ne peut contenir qu'une seule sorte de données
- On appelle cette sorte le ***type*** de la variable
- On dit que C++ est un langage typé statiquement

Les types de base

Les différents types de base en C++ sont :
- Les ***entiers*** (mots clés `int`, `long int`)Exemples : 1, 42, -32765
- les ***réels*** (mots clés `float`, `double`)Exemples : 10.43, 1.0324432e22
- les ***caractères*** (mot clé `char`)Exemples : "a", "b", " ", "]"
- les ***chaînes de caractères*** (mot clé `string`)Exemples : « bonjour », « Alice aime Bob »
- les booléens (mot clé `bool`)Exemples : `true` (vrai), `false` (faux)

Les entiers, les caractères et les booléens forment les ***types ordinaux***.

2.5.5 La déclaration des variables

Pour chaque variable, il faut donner au programme son nom et son type. On dit que l'on ***déclare*** la variable.

Syntaxe : Déclaration des variables

```
type nomvariable;
type nomvariable1, nomvariable2, ...;
```

Exemples :

```
int x, y, monEntier;
double f, g;
bool b;
```

Note : en C++ (compilé) on ne peut pas redéclarer une variable avec le même nom!

2.5.6 L'affectation

Syntaxe :

```
identificateur = expression;
```

Exemple :

```
x = 3 + 5;
```

Sémantique :
- Calcul (ou évaluation) de la valeur de l'expression
- Stockage de cette valeur dans la case mémoire associée à cette variable.

— *La variable et l'expression doivent être de même type!*

Exemples d'affectations

```
int x, y;
```

On affecte la valeur 1 à la variable `x` :

```
x = 1;
```

On affecte la valeur 3 à la variable `y` :

```
y = 3;
```

Valeurs des variables après l'affectation :

```
x
```

```
y
```

Exemple : affecter la valeur d'une variable à une autre variable

```
x = y;
```

```
x
```

```
y
```

Note

— Affectation `x = y` : copie de la valeur

— `y` garde sa valeur

— L'ancienne valeur de `x` est perdue!

— Différent de transférer un ingrédient d'un récipient à l'autre

Exemple : incrémentation

```
int x;
```

```
x = 1;
```

```
x = x + 1;
```

```
x
```

Variantes :

```
x -= 2
```

```
x--;
```

```
x
```

⍰ Affectation et égalité : deux concepts différents ⍰

L'affectation `x = 5` :

Une instruction modifiant l'état de la mémoire.

Le test d'égalité `x == 5` :

Une expression qui a une valeur booléenne (vrai ou faux) :

«vrai ou faux: `x` est égal à `5` ?»

Autrement dit : est-ce que la valeur contenue dans la variable `x` est `5` ?

2.6 Fonctions

Retour sur notre

Exemple :

Calculer l'énergie cinétique $\frac{1}{2}mv^2$ d'un objet de masse $14,5$ kg selon qu'il aille à 1, 10, 100, ou 1000 km/h.

Comment éviter de retaper chaque fois la formule?

2.6.1 Fonctions

Définition informelle :

Une fonction est un petit programme :
- Entrées
- Traitement
- Sortie

Exemple :

```
double energie_cinetique(double m, double v) {
    return 0.5 * m * v * v;
}
```

```
energie_cinetique(14.5, 10)
```

- Entrées : la masse et la vitesse (des nombres réels)
- Sortie : l'énergie cinétique (un nombre réel)
- Traitement : `0.5 * m * v * v`

Autres exemples de fonctions

Chou-Chèvre-Loup :

```
void transporter(... T) {
    charger(T);
    traverser();
    decharger(T);
}
```

Laby :

```
void avance_tant_que_tu_peux() {
    while ( regarde() == Vide ) {
        avance();
    }
}
```

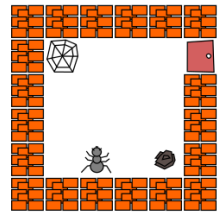
2.7 Structures de contrôle

2.7.1 Rôle des structures de contrôle

Rappel

Les instructions sont exécutées de manière séquentielle (les unes après les autres), dans l'ordre du programme.

Exemple :

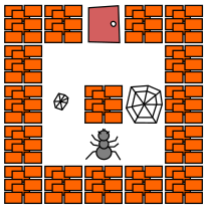


```
        droite();
        avance();
        prend();
        gauche();
        avance();
        pose();
        droite();
        avance();
        gauche();
        avance();
        avance();
        droite();
        ouvre();
```

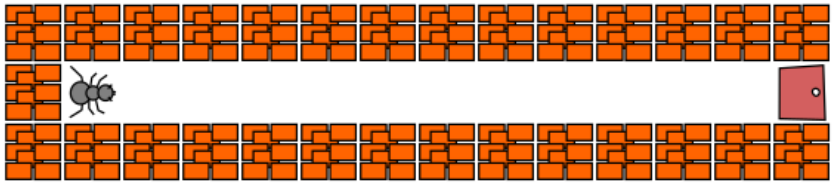
Le problème

On a souvent besoin de *rompre l'exécution séquentielle* :

— Des instructions différentes selon le contexte :



— Des instructions répétées :



Ce sont les **structures de contrôle**

2.7.2 Instructions conditionnelles

En fonction d'une ***condition***, on va exécuter ou non un **bloc d'instructions**.

```
#include <laby/global_fr.hpp>
LABY("3a")
```

```
debut();
droite();
avance();
gauche();
```

Solution complète :

```
LABY("3a")
```

```
droite();
avance();
gauche();

if ( regarde() == Toile ) {
    gauche();
    avance();
    avance();
    droite();
    avance();
    avance();
    droite();
    avance();
    gauche();
} else {
    avance();
    avance();
    gauche();
    avance();
    droite();
}
ouvre();
```

Définition : bloc d'instructions

Un ***bloc d'instructions*** est une suite d'instructions à exécuter successivement. Il est décrit par la syntaxe suivante :

```
    {
        instruction 1;
        instruction 2;
        ...
        instruction n;
    }
```

Une instruction toute seule est considérée comme un bloc

Instruction conditionnelle simple : «si ... alors ...»

Syntaxe :

```
if ( condition ) {
    bloc d instructions;
}
```

Sémantique :

1. Évaluation de la condition
2. Si sa valeur est vraie, exécution du bloc d'instructions

Exemples :

```
if ( regarde() == Toile ) {      // Au secours, fuyons!
    gauche();
    gauche();
}

if ( x >= 0 ) gauche();
```

Instruction conditionnelle : «si ... alors ... sinon ...»

Syntaxe :

```
if ( condition ) {
    bloc d instructions 1;
} else {
    bloc d instructions 2;
}
```

Sémantique :

1. Évaluation de la condition
2. Si sa valeur est «Vrai», exécution du bloc d'instructions 1
3. Si sa valeur est «Faux», exécution du bloc d'instructions 2

Exemples d'instruction alternative

Exemple :

```
if ( regarde() == Toile ) {      // Au secours, fuyons!
    gauche();
    gauche();
} else {                         // Tout va bien
    avance();
}
```

Exemples d'instruction alternative (2)

Exemple : Calcul du maximum et du minimum de `x` et `y`

```
int x = 3, y = 5;                // Les entrées
int maximum, minimum;            // Les sorties

if ( x > y ) {
    maximum = x;
    minimum = y;
} else {
    maximum = y;
    minimum = x;
}
```

2.8 Résumé

Un aperçu de premiers éléments de programmation :
- Expressions, valeurs et types
- Variables (*affectation ≠ égalité!*)
- Fonctions
- Conditionnelles (`if`)

On reviendra dessus!

CHAPITRE 3

Mémoire, conditionnelles et itératives

3.1 Prélude

3.1.1 Résumé des épisodes précédents

— Info 111 : modalités objectifs et infrastructure

— Un aperçu des premiers éléments de la programmation impérative :
  — expressions, variables
  — instructions conditionnelles : `if`
  — instruction itérative : boucle `while`

3.1.2 Au programme

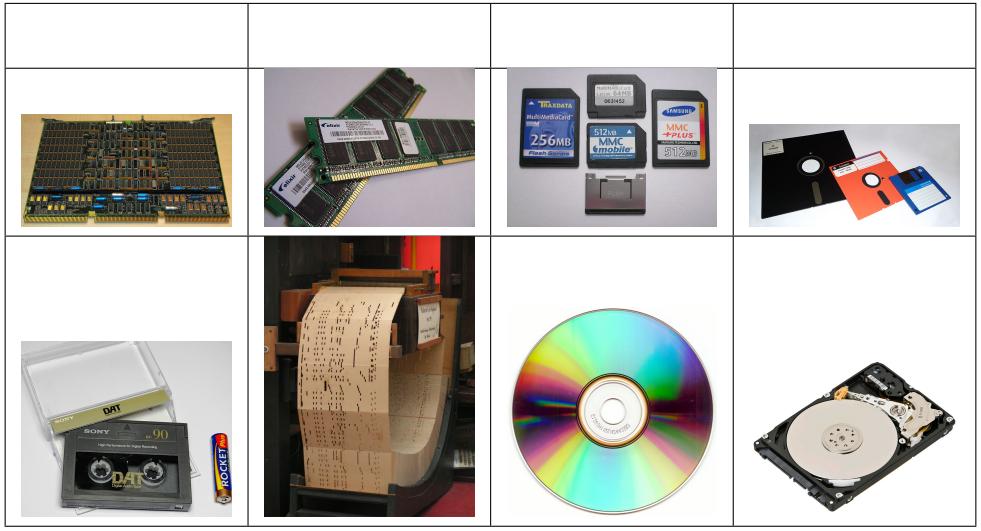
1. Démo exercices PLaTon
2. *Mémoire et variables*
3. *Digression: types entiers*
4. *Conditionnelles enchaînées*
5. *Instructions itératives*
6. *Contionnelles: erreurs classiques*

3.2 Mémoire et variables

Un ordinateur traite de l'information.
  — Il faut pouvoir la stocker: la *mémoire*
  — Il faut pouvoir y accéder: les *variables*

3.2.1 Mémoire

https://fr.wikipedia.org/wiki/Mémoire_(informatique)



Modèle simplifié

- Une suite contiguë de `0` et de `1` Les ***bits***, groupés par paquets de huit appelés ***octets***
- Pour 1 Go, à raison de un bit par mm, cela ferait 8590 kmPlus que Paris-Pékin!
- Le processeur y accède par **adresse**

3.2.2 Variables

Une **variable** est un espace de stockage *nommé* où le programme peut mémoriser une donnée; elle possède quatre propriétés:

- Un **nom** (ou **identificateur**) : Il est choisi par le programmeur
- Une **adresse** :Où est stockée la variable dans la mémoire
- Un **type** qui spécifie:
  - La **structure de donnée** : comment la valeur est représentée en mémoireEn particulier combien d'octets sont occupés par la variable
  - La **sémantique** des opérations
- Une **valeur** :Elle peut changer en cours d'exécution du programme

Règles de formation des identificateurs

Les noms des variables (ainsi que les noms des programmes, constantes, types, procédures et fonctions) sont appelés des **identificateurs**.

Syntaxe: règles de formation des identificateurs

- suite de lettres (minuscules `'a'...'z'` ou majuscules `'A'...'Z'`), de chiffres (`'0'...'9'`) et de caractères de soulignement (`'_'`)
- premier caractère devant être une lettre
- longueur bornée

`Exemples et contre-exemples d'identificateurs :`

- `c14_T0` est un identificateur
- `14c_T0` n'est pas un identificateur
- `x*y` n'est pas un identificateur

Formation des identificateurs : bonnes pratiques

- Donnez des noms *signifiants* aux variables
- Dans le cas de plusieurs mots, par convention dans le cadre de ce cours on mettra le premier mot en minuscule et les suivants avec une majuscule: `maVariable`
- Autre convention possible: `ma_variable`
- Mauvais noms : `truc`, `toto`, `temp`, `nombre`
- Bons noms courts: `i`, `j`, `k`, `x`, `y`, `z`, `t`
- Bons noms longs: `nbCases`, `notes`, `moyenneNotes`, `estNegatif`

Initialisation des variables

Quelle est la valeur de ces variables après leur déclaration?

```
double d;
long l;
int i;
```

```
l
```

```
d
```

— Certains langages ou compilateurs garantissent que les variables sont initialisées à une valeur par défaut.

— *En C++, pas forcément!*
Typiquement, la valeur de la variable correspond à l'état de la mémoire au moment de sa déclaration

Bonne pratique : systématiquement initialiser les variables au moment de leur déclaration:

```
int i = 0;
long l = 1024;
double d = 3.14159;
```

3.3 Les types entiers en C++

```
#include <climits>
#include <cmath>
```

3.3.1 Type `unsigned short int`

```
unsigned short int u;
```

```
u = 1
```

```
u = 10000
```

```
u = 100000
```

```
USHRT_MAX
```

```
pow(2,16) - 1
```

```
u = 65533
```

```
u = u + 1
```

```
u = u + 1
```

```
u = u + 1
```

```
u = u - 1
```

3.3.2 Type `signed short int`

```
short int s;
```

```
s = 32767
```

```
s = s + 1
```

3.3.3 Type `int`

```
int i;
```

```
i = 1
```

```
i = 1000
```

```
i = 1000000
```

```
i = 1000000000
```

```
i = 1000000000000
```

Quel est le plus grand entier que l'on peut représenter avec un int?

```
INT_MAX
```

```
i = INT_MAX
```

Que se passe-t-il si on lui ajoute `1` ?

```
i + 1
```

```
i = INT_MIN
```

```
i - 1
```

Conséquence amusante:

```
-i
```

```
abs(i)
```

3.3.4 Type `long`

```
long l = LONG_MAX
```

```
l + 1
```

```
l = LONG_MIN
```

```
l - 1
```

```
- l
```

3.3.5 À retenir

— Les entiers des types `int`, `long`, … sont des **entiers machine** ; ce sont des **approximations** des entiers mathématiquesIl est possible de représenter les entiers mathématiques arbitraires avec un type adéquat

— De même pour `float`, `double`, … : ce sont des **approximations** des nombres réèlsIl n'est pas possible de représenter tous les nombres réels!

3.4 Cours et exercices: les instructions conditionnelles enchaînées

Lors des séances précédentes, nous avons manipulé des conditionnelles simples, permettant de distinguer entre deux cas à traiter avec des actions différentes.

```
    if ( condition ) {
        bloc d'instruction 1;   // Instructions dans le premier cas
    } else {
        bloc d'instruction 2;   // Instructions dans le deuxième␣
↪cas
    }
```

Comment faire lorsqu'il y a plus de deux cas à traiter? Nous allons voir qu'il n'y a pas besoin de nouvelle instruction; il suffit d'"**enchaîner les instructions conditionnelles**.

Exercice :

— Observez le programme suivant, sans l'exécuter, que fait-il?
— Vérifiez votre intuition en l'exécutant avec plusieurs valeurs de `note`.

```
int note;
std::string resultat;
```

```
note = 14;
```

```
if ( note < 0  or note > 20 ) {
    resultat = "La note n'est pas correcte.";
} else {
    if ( note < 10 ) {
        resultat = "Vous avez raté l'examen!";
    } else {
        resultat = "Vous avez réussi l'examen!";
    }
}

resultat
```

Comment est-ce que cela marche?

Pour gérer trois cas, nous avons simplement distingué entre deux cas (note incorrecte ou correcte), puis distingué ce deuxième cas en deux (examen raté ou réussi).

Comparez maintenant le programme ci-dessus avec le programme ci-dessous:

```
if ( note < 0  or note > 20 ) {
    resultat = "La note n'est pas correcte.";
} else if ( note < 10 ) {
    resultat = "Vous avez raté l'examen!";
} else {
    resultat = "Vous avez réussi l'examen!";
}

resultat
```

La seule chose qui a changé, c'est une paire d'accolades en moins (laquelle?), ainsi que l'"**indentation** (les espaces en début de ligne). Les deux programmes sont en fait équivalents (vérifiez le!) car:

— le deuxième `if` forme une seule instruction;
— les accolades ne sont pas requises en C++ lorsqu'un bloc est formé d'une seule instruction.

Cependant le deuxième programme est plus lisible, et **exprime une intention différente** : il y a trois cas, tous sur le même plan, et non deux cas dont le deuxième avec sous-cas.

3.4.1 À vous de jouer

Recopiez le programme précédent ci-dessous en l'adaptant pour qu'il affiche en plus la mention (de 12 à 14, mention *assez bien*, de 14 à 16 mention *bien* et au dessus de 16, mention *très bien*) :

```
// VOTRE CODE ICI
resultat
```

3.4.2 Bilan

Bravo, vous maîtrisez maintenant les instructions conditionnelles avec cas multiples.

3.5 Les instructions itératives

3.5.1 Motivation : retour sur laby

```
#include <laby/global_fr.hpp>
LABY("2a")
```

Voilà une première proposition de solution. Est-elle correcte? Est-elle satisfaisante?

```
debut();
avance();
avance();
avance();
avance();
```

(suite sur la page suivante)

(suite de la page précédente)

```
avance();
avance();
avance();
avance();
avance();
avance();
avance();
ouvre();
```

Inconvénients:

— C'est *long*, *répétitif* ;
— Ce n'est pas *général* : il faut compter le nombre de cases du couloir et écrire une solution différente selon le cas.

En français, on dirait plutôt : «Tant que c'est vide devant toi, avance!».

En C++, cela se traduit par :

```
debut();

while ( regarde() == Vide ) {
    avance();
}

ouvre();
```

3.5.2 Répéter ou ne pas se répéter

Un ordinateur est capable d'effectuer des tâches répétitives très rapidement et sans s'ennuyer.

Pas nous!

Aussi le principe suivant de programmation va revenir comme un leitmotiv tout au long de ce cours :

«Ne te répètes pas».

En anglais cela donne l'acronyme mnémotechnique «**DRY**»:

Don't Repeat Yourself

3.5.3 Les instructions itératives

Dans cette feuille, nous abordons un premier outil pour ne pas se répéter : les **instructions itératives**. Elles permettent de répéter un certain nombre de fois l'exécution d'un bloc d'instructions sous certaines conditions. De façon imagée, on appelle **boucle** cette méthode permettant de répéter l'exécution d'un bloc d'instructions.

Il y a trois cas typiques d'utilisation de boucles, illustrés par les exemples suivants :

— Tant que l'on est pas à la fin du film, afficher une image puis attendre 1/24ème de seconde;
— Dans un jeu sur ordinateur, à la fin d'une partie, demander «voulez vous rejouer?» et si oui recommencer une nouvelle partie;
— Afficher tous les nombres entre 1 et 1000.

Il serait tout à fait possible de n'avoir qu'une seule instruction itérative pour couvrir tous les usages. L'expérience a cependant montré qu'il était plus pratique, plus sûr et surtout plus **expressif** d'avoir trois types d'instructions itératives distincts dans un language de programmation :

— Boucles «while» : «tant que … faire …»;
— Boucles «do … while» : «Faire … tant que …»;
— Boucles «for» : «Pour … de … à … faire …»;

Nous allons maintenant les voir tour à tour.

3.5.4 La boucle while : «tant que ... répéter ...»

La boucle while : syntaxe et sémantique

Syntaxe :

```
while ( condition ) {
    bloc d instructions;
}
```

Sémantique :

1. Évaluation de la condition
2. Si sa valeur est «Vrai» :
    1. Exécution du bloc d'instructions
    2. On recommence en 1.

La boucle while : exemples

Une petite digression : pour visualiser l'exécution d'une boucle, il est pratique d'utiliser des affichages, pour lesquels nous avons besoin d'une petite incantation magique :

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

Exécutez pas-à-pas le programme ci-dessous en appliquant la syntaxe et sémantique de la boucle «while» :

```
int n = 1;

while ( n <= 5 ) {
    cout << n << endl;     // Affiche la valeur de n
    n = n + 1;
}
```

On a réalisé un programme permettant de compter de 1 à 5. On verra un peu plus loin que ce n'est pas la meilleure instruction itérative pour cet usage.

Exercice :

Modifiez l'exemple ci-dessus pour :
- compter jusqu'à 10 (inclus) ;
- compter de -5 à 5 (inclus) ;
- compter de deux en deux de 0 à 10.

Cas particulier : condition toujours fausse

*Si la valeur de la condition est fausse dès le départ, alors le bloc d'instructions ne sera **jamais exécuté** !*

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

```
int n = 1;
```

(suite sur la page suivante)

(suite de la page précédente)

```
while ( n <= 0 ) {
    cout << n << endl;     // Affiche la valeur de n
    n = n + 1;
}
```

Cas particulier : condition toujours vraie

*Si la valeur de la condition est toujours vraie, alors le bloc d'instructions sera **exécuté indéfiniment** !*

⚠ L'exemple suivant ne va pas s'arrêter! Il faudra redémarrer le noyau (menu Noyau) ⚠

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

```
int n = 1;

while ( true ) {
    cout << n << endl;     // Affiche la valeur de n
    n = n + 1;
}
```

Erreur classique : oublier l'incrémentation

⚠ L'exemple suivant ne va pas s'arrêter! Il faudra redémarer le noyau (menu Noyau). ⚠

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

```
int n = 1;

while ( n <= 5 ) {
    cout << n << endl;     // Affiche la valeur de n
}
```

Erreur classique: fin de boucle

Que vaut n à la fin du programme suivant?

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

```
int n = 1;

while ( n <= 5 ) {
    n = n + 1;
}

cout << n << endl;    // Affiche la valeur de n;
```

À retenir :

— On sort de la boucle quand la condition vaut «Faux»; le compteur est donc «un cran trop loin».

3.5.5 La boucle do ... while : «faire ... tant que ...»

La boucle do ... while : motivation

Dans un jeu sur ordinateur, à la fin d'une partie, on veut demander «voulez vous rejouer?» et si oui recommencer une nouvelle partie.

Cela peut s'écrire avec une boucle `while`, mais ce n'est pas très pratique car :

— On veut jouer la partie *au moins une fois*

— On veut tester la condition *après* la partie

Pour traiter ce cas d'usage classique de façon plus élégante, la plupart des langages de programmation introduisent une variante de la boucle «while» : la boucle **«do … while»** ; littéralement : «faire … tant que …»;

La boucle do ... while : syntaxe et sémantique

Syntaxe :

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

Sémantique

1. Exécution du bloc d'instructions
2. Évaluation de la condition
3. Si sa valeur est «Vrai», on recommence en 1.

La boucle do ... while : exemple

```
#include <iostream>
using namespace std;
std::string reponse;
```

```
do {
    cout << "Une partie de jeu ..."          << endl;
    cout << "Voulez-vous rejouer (oui/non)?" << endl;
    cin >> reponse;                  // Lit la réponse
} while ( reponse == "oui" );
```

3.5.6 La boucle for : «pour ... de ... à ... faire ...»; compteurs

La boucle for : motivation

Revenons sur notre programme pour compter de 1 à 5 :

```
int n = 1;
while ( n <= 5 ) {
    cout << n << endl;    // Affiche la valeur de n
    n = n + 1;
}
```

Il suit un schéma classique de programme avec un *compteur* :

```
initialisation;

while ( condition ) {
    bloc d instructions;
    incrémentation;
}
```

— L'***initialisation*** (`n = 1`) détermine à partir de quelle valeur on compte;
— L'***incrémentation*** (`n = n + 1`) détermine par pas de combien on compte;
— La ***condition*** (`n <= 10`) détermine jusqu'à quelle valeur on compte.

Problème :

La gestion du compteur est dispersée : le lecteur doit chercher à trois endroits pour avoir toutes les informations sur le compteur. L'incrémentation notamment se retrouve loin si le bloc d'instructions est long. Nous avons vu les conséquences si on l'oublie accidentellement!

Pour pallier à cela, la plupart des langages de programmation introduisent la boucle «**for**» : «pour … de … à … faire …». Voici ce que cela donne sur notre exemple :

```
for ( int n = 1; n <= 5; n = n + 1 ) {
    cout << n << endl;    // Affiche la valeur de n
}
```

Exercice :

Comparer les deux programmes ci-dessus pour compter de 1 à 5 ; retrouver les différents éléments : initialisation, incrémentation, condition, instruction.

À retenir :
— Avec la boucle «for», toute la gestion du compteur est centralisée sur une seule ligne.
— La boucle «for» est strictement équivalent à while, mais elle **exprime une intention** : on utilise un compteur.

3.5.7 La boucle for : syntaxe et sémantique

Syntaxe :

```
for ( initialisation ; condition ; incrémentation ) {
    bloc d instructions;
}
```

Sémantique :

1. Exécution de l'instruction d'initialisation
2. Évaluation de la condition
3. Si sa valeur est «Vrai» :
    1. Exécution du bloc d'instructions
    2. Exécution de l'instruction d'incrémentation
    3. On recommence en 2.

La boucle for : exemples

Notre exemple ci-dessus peut s'écrire de façon plus compacte :

```
for ( int n = 1 ; n <= 5 ; n++ ) {
    cout << n << endl;
}
```

— La variable `n` est locale à la boucle (on y reviendra)
— `n++` est un raccourci pour `n = n + 1`

3.5.8 Compteurs et accumulateurs

On souhaite calculer la factorielle de 7 : $7! = 1 \cdot 2 \cdot 3 \cdots 7$

Cela peut s'écrire de la façon suivante :

```
int resultat = 1;

resultat = resultat * 2;
resultat = resultat * 3;
resultat = resultat * 4;
resultat = resultat * 5;
resultat = resultat * 6;
resultat = resultat * 7;

resultat
```

La variable `resultat` sert d'***accumulateur*** : on y accumule les entiers 1, 2, 3, ... par produit et elle vaut successivement :
— 1

— $1 \cdot 2$
— $1 \cdot 2 \cdot 3$
— …
— $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7$

Problèmes :

— Ce code sent mauvais : il y a beaucoup de répétitions!
— Et si on veut calculer 10! ou 100! ?

Des instructions répétées? Cela suggère une boucle. On compte de 1 à 7? C'est une mission pour la boucle «for»!

```
int n = 10;
```

```
int resultat = 1;
for ( int i = 2; i <= n; i = i + 1) {
     resultat = resultat * i;
}
```

```
resultat
```

Exécutez pas-à-pas le code ci-dessus pour $n = 3$; pour $n = 0$.

3.5.9 Résumé

Nous avons vu les **instructions itératives** – les **boucles** – pour répéter des instructions :

— la boucle «while»
— la boucle «do while»
— la boucle «for»

Elles sont toutes les trois strictement équivalentes. Mais chacune est mieux adaptée à un type d'utilisation, et exprime une **intention**.

Plus tard nous verrons une quatrième boucle, la boucle «foreach», lorsque nous voudrons parcourir les éléments d'une collection.

Nous avons aussi vu deux techniques classiques de boucles :

— L'utilisation d'un ***compteur*** : `k` qui parcours les valeurs de … à … par pas de …
— L'utilisation d'un ***accumulateur*** : `resultat` qui accumule progressivement des valeurs par produit, somme, …

3.6 Conditionnelles : erreurs classiques

Nous allons maintenant voir quelques bonnes pratiques et erreurs classiques lorsque l'on utilise des instructions conditionnelles.

3.6.1 Exemple 1

Dans cet exemple, on souhaite affecter à la variable `estPositif` la valeur «Vrai» si $x \geq 0$ et «Faux» sinon. Cela se traduit litéralement par :

```
int x = 4;
```

```
bool estPositif;

if ( x >= 0 ) {
    estPositif = true;
} else {
    estPositif = false;
}

estPositif
```

Mais ne pourrait-on pas faire mieux?

Si!

Grâce à George Boole qui a réalisé que «Vrai» et «Faux» sont des valeurs comme les autres avec lesquelles on peut calculer (expressions booléennes) et que l'on peut stocker (variables booléennes).

Exécutez les cellules ci-dessus et ci-dessous pour différentes valeurs de `x` et vérifier que le résultat est le même :

```
x = 4
```

```
x >= 0
```

```
bool estPositif =  x >= 0;

estPositif
```

C'est plus concis. Plus efficace. Avec moins de risque d'erreur.

Pourquoi cela nous paraît-il moins naturel au premier abord? Parce qu'il n'y a pas d'équivalent naturel dans la langue française! Au mieux on peut utiliser une périphrase comme : stocker dans `estPositif` la valeur de vérité de l'expression «x est plus grand que zéro».

À retenir :

— Chaque fois que possible, utiliser une expression booléenne plutôt qu'une instruction conditionnelle.

3.6.2 Exemple 2

Devinez le contenu de la variable `y` à la fin de l'exécution du programme suivant. Vérifiez en l'exécutant :

```
int x = 0;
int y = 0;

if ( x = 1 ) {
    y = 4;
}

y
```

Pourtant `x` n'est pas égal à 1! L'instruction `y=4` n'aurait pas dû être exécutée!?!

Sauf que … Nous avons utilisé une affectation = et non un test d'égalité == ! Cela se voit à la valeur de `x` :

```
x
```

Jupyter + Cling nous ont prévenu avec une alerte (*warning*) : ce n'est probablement pas ce que l'on souhaitait faire.

Il se trouve cependant que le code ci-dessus est valide en C++, ce qui est une cause classique de bogue.

Pour les curieux : ♣

En toute logique le code ci-dessus devrait être invalide et déclencher une erreur. En effet, une affectation, c'est juste une **instruction** qui décrit une action à effectuer. Une action, cela n'a pas de valeur. Cependant de nombreux languages, dont C++, choisissent de faire de l'affectation une **expression**, en décidant de lui attribuer une valeur : celle qui a été affectée.

Cela permet par exemple d'affecter une valeur à deux variables simultanément :

```
x = y = 17
```

Analysez pourquoi cela marche, en l'interprétant sous la forme : `x = ( y = 17 )`.

Maintenant nous savons que `x = 1` est une expression qui vaut `1` tout le temps; elle est en particulier de type `int`. Là encore le code devrait être invalide puisque l'on a dit que `if` prenait comme condition une expression booléenne.

Sauf que … C++ hérite de C une vieille convention qui date de l'époque où les languages n'avaient pas de type `bool`. On représentait alors les valeurs booléennes par des entiers, en représentant «Faux» par `0` et «Vrai» par n'importe quel nombre entier non nul. De ce fait l'expression `x = 1` a pour valeur `1` qui est interprétée comme «Vrai» dans le contexte d'une condition.

Exercice : ♣

Quelles sont les valeurs de `x` et `y` à la fin du programme suivant :

```
int x = 1;
int y = 2;
if ( x = 0 ) {
    y = 3;
}
```

3.6.3 Erreurs classiques avec les conditionnelles

Devinez ce qu'affiche le programme suivant? Vérifiez en l'exécutant :

```
int x = 0;
int y = 0;

if ( x == 1 ); {
    y = 4;
}

y
```

Ce programme est équivalent à :

```
if ( x == 1 );
y = 4;
```

qui ne tient pas compte du `if` et affecte toujours `4` à `y` (quel que soit `x`).

À retenir :

— Le point-virgule `;` est un **séparateur d'instruction** ;
— Le bloc d'instructions d'un `if` peut être vide en C++;
— Il ne faut **jamais de point-virgule avant un bloc d'instructions!**

3.6.4 Tests imbriqués

Devinez ce qu'affiche le programme suivant? Vérifiez en l'exécutant :

```
int x = 5;
int y = 4;
std::string resultat = "";

if ( x >= y ) {
    if ( x == y ) {
        resultat = "égalité";
    }
else {
    resultat = "x est plus petit que y";
```

(suite sur la page suivante)

(suite de la page précédente)

```
}
}

resultat
```

Oups. Il y a un bogue.

Explication : en C++, la structuration est entièrement déterminée par les accolades : `{`, `}`. Les espaces et sauts de lignes ne jouent aucun rôle. Aussi, ici, le `else` se rapporte au deuxième `if`, contrairement à ce que la disposition visuelle du code pourrait faire croire.

Maintenant, **indentons** correctement le code : c'est-à-dire décalons chaque ligne de quatre espaces par niveau d'imbrication dans des accolades :

```
resultat = "";

if ( x >= y ) {
    if ( x == y ) {
        resultat = "égalité";
    }
    else {
        resultat = "x est plus petit que y";
    }
}

resultat
```

La source du bogue est devenue claire, et on peut facilement le corriger :

```
resultat = "";

if ( x >= y ) {
    if ( x == y ) {
        resultat = "égalité";
    }
} else {
    resultat = "x est plus petit que y";
}

resultat
```

À retenir :
- Un `else` se rapporte au dernier `if` rencontré;
- En C++, la structuration est déterminée pas les accolades;
- La mauvaise indentation induit en erreur le lecteur!

3.6.5 Apparté : l'indentation

- L'***indentation*** consiste à espacer les lignes de code par rapport au bord gauche de la fenêtre de saisie de texte;
- L'espacement doit être proportionnel au *niveau d'imbrication* des instructions du programme;
- Quatre espaces par niveau d'imbrication est un bon compromis.

La plupart des éditeurs de texte offrent des facilités pour réaliser une bonne indentation. *Apprenez-les.*

3.6.6 De la lisibilité des programmes

> «Programs must be written for people to read, and only incidentally for machines to execute.»
>
> – Harold Abelson, Structure and Interpretation of Computer Programs 1984

- Un programme s'adresse à un *lecteur*
- La *lisibilité* est un objectif essentiel

3.6.7 Conclusion

Nous avons vu les bonnes pratiques et erreurs classiques suivantes sur les instructions conditionnelles :
- Bonne pratique : lorsque cela est possible, utiliser une expression booléenne plutôt qu'une instruction conditionnelle;
- Erreur classique : insérer accidentellement une instruction vide `;` ;
- Erreur classique : imbriquer incorrectement `if` et `else`.

Nous avons aussi vu l'importance de l'**indentation** pour la **lisibilité** des programmes.

CHAPITRE 4

Fonctions

4.1 Prélude

4.1.1 Résumé des épisodes précédents ...

Pour le moment nous avons vu :

- Expressions: `3 * (4+5)`, `1 < x and x < 5 or y == 3`
- Variables, types, affectation: `int n = 1 + 1`
- Instruction conditionnelle: `if`
- Instructions itératives: `while`, `do ... while`, `for`

Tout ce qui est calculable par un ordinateur peut être programmé uniquement avec ces instructions(ou presque : il faudrait un accès un peu plus souple à la mémoire)

Pourquoi aller plus loin?

Pour passer à l'échelle!

Motivation: l'exemple du livre de cuisine (1)

Recette de la tarte aux pommes

- Ingrédients : 250 g de farine, 125 g de beurre, 1 œuf, 2 cl d'eau, une pincée de sel, 100 g de sucre en poudre, 5 belles pommes
- Mettre la farine dans un récipient puis faire un puits
- Versez dans le puits 2 cl d'eau
- Mettre le beurre
- Mettre le sucre et le sel
- Pétrir de façon à former une boule
- Étaler la pâte dans un moule
- Peler les pommes, les couper en quartiers et les disposer sur la pâte
- Faire cuire 30 minutes

Motivation : l'exemple du livre de cuisine (2)

Recette de la tarte aux poires

- Ingrédients : 250 g de farine, 125 g de beurre, 1 œuf, 2 cl d'eau, une pincée de sel, 100 g de sucre en poudre, 5 belles poires
- Mettre la farine dans un récipient puis faire un puits
- Versez dans le puits 2 cl d'eau
- Mettre le beurre

— Mettre le sucre et le sel
— Pétrir de façon à former une boule
— Étaler la pâte dans un moule
— Peler les poires, les couper en quartiers et les disposer sur la pâte
— Faire cuire 30 minutes

Motivation : l'exemple du livre de cuisine (3)

Recette de la tarte tatin

— Ingrédients : 250 g de farine, 125 g de beurre, 1 œuf, 2 cl d'eau, une pincée de sel, 200 g de sucre en poudre, 5 belles pommes
— Mettre la farine dans un récipient puis faire un puits
— Versez dans le puits 2 cl d'eau
— Mettre le beurre
— Mettre le sucre et le sel
— Pétrir de façon à former une boule
— Verser le sucre dans une casserole
— Rajouter un peu d'eau pour l'humecter
— Le faire caraméliser à feu vif, sans remuer
— Verser au fond du plat à tarte
— Peler les pommes, les couper en quartiers
— Faire revenir les pommes dans une poêle avec du beurre
— Disposer les pommes dans le plat et étaler la pâte au dessus
— Faire cuire 45 minutes et retourner dans une assiette

Qu'est-ce qui ne va pas?

Duplications

— Longueurs
— En cas d'erreur ou d'amélioration :plusieurs endroits à ajuster!

Manque d'expressivité

— Difficile à lire
— Difficile à mémoriser

Essayons d'améliorer cela

Recettes de base

Recette de la pâte brisée

— Ingrédients : 250 g de farine, 125 g de beurre, 1 œuf, 2 cl d'eau, une pincée de sel
— Mettre la farine dans un récipient puis faire un puits
— Versez dans le puits 2 cl d'eau Mettre le beurre
— Mettre le sucre et et une pincée de sel
— Pétrir de façon à former une boule

Recette du caramel

— Ingrédients : 100 g de sucre
— Verser le sucre dans une casserole
— Rajouter un peu d'eau pour l'humecter
— Le faire caraméliser à feu vif, sans remuer

Recettes de tartes

Tarte aux fruits (pommes, poires, prunes, ...)

— Ingrédients : 500g de fruits, ingrédients pour une pâte brisée
— *Préparer une pâte brisée*
— Étaler la pâte dans un moule
— Peler les fruits, les couper en quartiers et les disposer sur la pâte
— Faire cuire 30 minutes

Tarte tatin

— Ingrédients : 5 belles pommes, ingrédients pour pâte brisée et caramel
— *Préparer une pâte brisée*
— *Préparer un caramel* et le verser au fond du plat à tarte
— Peler les pommes, les couper en quartiers
— Faire revenir les pommes dans une poêle avec du beurre
— Disposer les pommes dans le plat, et étaler la pâte au dessus
— Faire cuire 45 minutes et retourner dans une assiette

Les *fonctions* : objectif

Modularité

— Décomposer un programme en programmes plus simples
— Implantation plus facile
— Validation (tests)
— Réutilisation
— Flexibilité (remplacement d'un sous-programme par un autre)

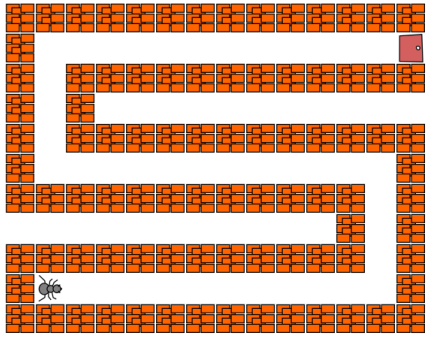
Non duplication

— Partager (*factoriser*) du code
— Code plus court
— Maintenance plus facile

Niveau d'abstraction

— Programmes plus *concis* et *expressifs*

Une impression de déjà vu? (1)



```
while ( regarde() == Vide ) {
    avance();
}
gauche();
while ( regarde() == Vide ) {
    avance();
}
gauche();
```

```
while ( regarde() == Vide ) {
    avance();
}
```

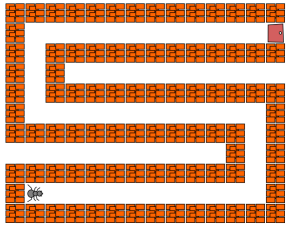
(suite sur la page suivante)

(suite de la page précédente)

```
droite();
while ( regarde() == Vide ) {
    avance();
}
droite();
```

```
while ( regarde() == Vide ) {
    avance();
}
ouvre();
```

Une impression de déjà vu? (2)



```
void avance_tant_que_tu_peux() {
    while ( regarde() == Vide ) {
        avance();
    }
}
```

```
avance_tant_que_tu_peux();
gauche();
avance_tant_que_tu_peux();
gauche();
avance_tant_que_tu_peux();
droite();
avance_tant_que_tu_peux();
droite();
avance_tant_que_tu_peux();
ouvre();
```

Une impression de déjà vu? (3)

Fonctions que vous avez déjà écrites :

- TD1 : `transporte(Chèvre)`
- TP1 : `avance_tant_que_tu_peux()`
- TD2 : `max2(note1, note2)`, `max3(note1, note2, note3)`, ...
- TP3 : `factorielle(n)`, ...

4.2 Fonctions

4.2.1 Appels de fonctions usuelles

Exemples

Chargement de la bibliothèque de fonctions mathématiques usuelles :

```
#include <cmath>
```

Fonctions trigonométriques :

```
cos(3.14159)
```

Fonction exponentielle :

```
exp(1.0)
```

Fonction puissance :

```
pow(3, 2)
```

```
pow(2, 3)
```

On remarque

- La présence de `#include <cmath>`C'est pour utiliser la bibliothèque de fonctions mathématiquesOn y reviendra …
- L'ordre des arguments est important
- Le type des arguments est important
- On sait ce que calcule `cos(x)` !
- On ne sait pas **comment** il le fait
- **On n'a pas besoin de le savoir**

4.2.2 Écrire ses propres fonctions : la fonction factorielle

À la main

Calculons 5! :

```
int resultat = 1;
for ( int i = 1; i <= 5; i++ ) {
    resultat = resultat * i;
}
resultat
```

Calculons 7! :

```
int resultat = 1;
for ( int i = 1; i <= 7; i++ ) {
    resultat = resultat * i;
}
resultat
```

Avec une fonction

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

(suite sur la page suivante)

(suite de la page précédente)

```
    return resultat;
}
```

```
factorielle(5)
```

```
factorielle(7)
```

```
factorielle(5) / factorielle(3) / factorielle(2)
```

4.2.3 Syntaxe d'une fonction

Syntaxe :

```
type nom(type1 parametre1, type2 parametre2, ...) {
    déclarations de variables;
    bloc d instructions;
    return expression;
}
```

- `parametre1, parametre2, ...` : les **paramètres formels**
- Le type des paramètres formels est fixé
- Les variables sont appelées **variables locales**
- À la fin, la fonction **renvoie** la valeur de `expression`Celle-ci doit être du type annoncé

4.2.4 Sémantique simplifiée de l'appel de fonction

Revenons sur la fonction `max`

```
float max(float a, float b) {
    if ( a >= b ) {
        return a;
    } else {
        return b;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

— `max` n'est utilisée que si elle est *appelée*

— Pour appeler cette fonction on écrit par exemple

```
max(1.5, 3.0)
```

1. les **paramètres** `a` et `b` sont initialisés avec les valeurs `1.5` et `3.0`
1. le code de la fonction est exécuté
1. l'exécution s'arrête au premier `return` rencontré
1. le `return` spécifie la **valeur de retour** de la fonction :la valeur de l'expression `max(1.5, 3.0)`

4.3 Documentation et tests

4.3.1 Documentation d'une fonction (syntaxe javadoc)

Exemple :

```
/** Fonction qui calcule la factorielle
 *  @param n un nombre entier positif
 *  @return n!
 **/
int factorielle(int n) ...
```

Une bonne documentation :
— Est concise et précise
— Donne les *préconditions* sur les paramètres
— Décrit le résultat (ce que fait la fonction)

Astuce pour être efficace :
— *Toujours commencer par écrire la documentation*De toute façon il faut réfléchir à ce que la fonction va faire!

4.3.2 Tests d'une fonction

— Pas d'infrastructure standard en C++ pour écrire des tests
— Dans ce cours, on utilisera doctest

Exemple :

```
int factorielle(int n) {
    int resultat = 1;
    for ( int i = 1; i <= n; i++ ) {
        resultat = resultat * i;
    }
    return resultat;
}
```

```
CHECK( factorielle(0) == 1  );
CHECK( factorielle(1) == 1  );
CHECK( factorielle(2) == 2  );
CHECK( factorielle(3) == 6  );
CHECK( factorielle(4) == 24 );
```

Note : jusqu'en 2020-2021, ainsi que les trois premières séances de 2021-2022, ce cours utilisait le nom `ASSERT` au lieu de `CHECK`. Le changement a eu lieu pour assurer la compatibilité avec l'infrastructure `doctest`. Pour simplifier la transition, dans l'environnement Jupyter de ce cours, on peut utiliser les deux indifféremment.

Tests d'une fonction

Astuces pour être efficace :
— *Commencer par écrire les tests d'une fonction*De toute façon il faut réfléchir à ce qu'elle va faire!
— Tester les cas particuliers
— Tant que l'on est pas sûr que la fonction est correcte :
  — Faire des essais supplémentaires
  — Capitaliser ces essais sous forme de tests
— Si l'on trouve un bogue :
  — Ajouter un test caractérisant le bogue
— Les *effets de bord* sont durs à tester!

4.4 Modèle d'exécution

4.4.1 Motivation

Exercice

On considère la fonction `incremente` :

```
int incremente(int n) {
    n = n + 1;
    return n;
}
```

Quelles sont les valeurs de `a` et `b` après l'exécution des lignes suivantes :

```
int a, b;

a = 1;
b = incremente(a);
```

Deux possibilités paraissent envisageables :
- `a=1` et `b=2`
- `a=2` et `b=2`

Laquelle en C++?

Motivation

Comprendre précisément l'*appel de fonction*

Exemple : appel de la fonction factorielle

```
int factorielle(int n) {
    int resultat = 1;
    for ( int k = 1; k <= n; k++ ) {
        resultat = resultat * k;
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    return resultat;
}
```

Que se passe-t-il lorsque l'on évalue l'expression suivante?

```
factorielle(1+2)
```

4.4.2 Appel de fonctions : formalisation

Syntaxe

```
nom(expression1, expression2, ...)
```

Sémantique

1. Evaluation des expressionsLeurs valeurs sont les **paramètres réels**

1. Allocation de mémoire sur la **pile** pour :
   - Les variables locales
   - Les paramètres formels

1. Affectation des paramètres réels aux paramètres formels(par copie; les types doivent correspondre!)

1. Exécution des instructions

1. Lorsque `return expression` est rencontré, évaluation de l'expression qui donne la **valeur de retour de la fonction**

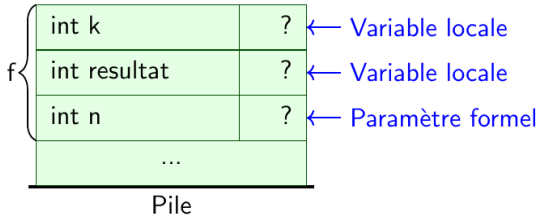
1. Désallocation des variables et paramètres sur la pile

1. La valeur de l'expression `nom(...)` est donnée par la valeur de retour

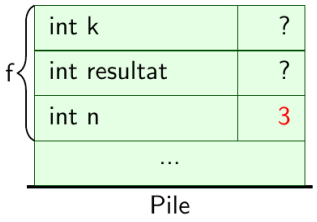
Évolution de la pile sur l'exemple

1. État initial

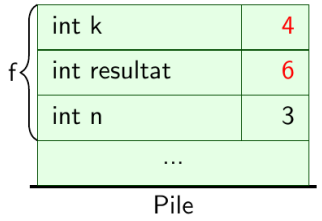
2 : Allocation de la mémoire sur la pile



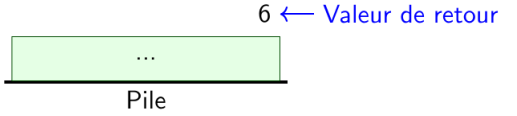
3 : Affectation du paramètre réel au paramètre formel



4 : Exécution des instructions



5,6,7 : Désallocation de la mémoire sur la pile et valeur de retour




```c
int incremente(int n) {
    n = n + 1;
    return n;
}
```

Appliquez la sémantique détaillée de l'appel de fonction pour déterminer les valeurs de `a` et `b` après l'exécution des lignes suivantes :

```c
int a, b;

a = 1;
b = incremente(a);
```

```c
a
```

```
b
```

4.4.3 Passage des paramètres par valeur

— Les paramètres formels d'une fonction sont des variables comme les autres.
— On peut les modifier.
— Mais …

Rappel

Lors d'un appel de fonction ou de procédure, la valeur du paramètre réel est **copiée** dans le paramètre formel.

Conséquence

— Une modification du paramètre formel n'affecte pas le paramètre réel
— Si la variable est volumineuse (tableaux, chaîne de caractères, etc.), cette copie peut être coûteuse

On dit que les paramètres sont passés **par valeur**.

Au second semestre, vous verrez le passage de paramètres **par référence**.

4.5 Fonctions particulières

1. Procédures
2. Fonctions récursives

4.5.1 Fonctions particulières : procédures

Besoin de sous-programmes qui *agissent* au lieu de *calculer* :
— On veut produire un effet (affichage, musique, *etc*)
— On veut modifier l'état interne d'une structure de donnée
On parle d'"**effet de bord**

Exemple :

```
void avance_tant_que_tu_peux() {
    while ( regarde() == Vide ) {
        avance();
    }
}
```

— Cette fonction ne renvoie rien
— On le dénote en C++ par le type `void`
— Dans d'autres langages on distingue **fonctions** et **procédures**

— Autres exemples : `transporte(...)`, `gauche()`

4.5.2 Fonctions particulières : fonctions récursives ♣

```
int factorielle(int n) {
    if ( n == 0 ) {
        return 1;
    } else {
        return n * factorielle(n-1);
    }
}
```

Une ***fonction*** récursive est une fonction qui s'appelle elle-même.

Cela peut paraître étrange de définir un objet à partir de lui-même, mais c'est comme pour les suites définies par récurrence en mathématiques: il faut et il suffit de s'assurer d'avoir un cas de base et une étape de récurrence bien posée.

Exercice :
— Exécuter pas à pas :
  — `factorielle(0)`
  — `factorielle(3)`
  — `factorielle(-1)`

4.6 Résumé

4.6.1 Motivation

— Modularité
— Lutte contre la duplication

— Programmes plus concis et expressifs

4.6.2 Fonctions

— Combinaison d'instructions élémentaires donnant une instruction de plus haut niveau
— Modèle d'exécution (pile)
— Procédures, fonctions récursives

4.6.3 Trilogie

— *Documentation* : ce que fait la fonction (entrées, sorties, …)
— *Tests* : ce que fait la fonction (exemples)
— *Code* : comment elle le fait

CHAPITRE 5

Tableaux, compilation, portée des variables

5.1 Prélude

5.1.1 Au programme

1. *Tableaux (introduction)*
2. *Premiers programmes compilés*
3. *Portée des variables*

5.2 Résumé du cours

5.2.1 Les tableaux

— **Valeurs composites homogènes** pour manipuler des collections de données
— Construction: **déclaration**, **allocation**, **initialisation**
— Utilisation

5.2.2 Fonctions: la portée des variables

— Variables locales
— Variables globales
— Portée définie par les blocs (contexte lexical)

5.2.3 Premiers programmes compilés

— Édition d'un fichier `programme.cpp` avec une fonction `main`
— Compilation: `g++ programme.cpp -o programme`
— Exécution: `./programme`

5.3 Premiers programmes compilés en C++

Jusqu'à présent, vous avez utilisé l'environnement Jupyter, qui est conçu pour du calcul interactif, où l'utilisateur manipule quelques lignes de code s'intégrant dans un texte narratif tel que ces notes de cours. Les calculs peuvent éventuellement faire intervenir des bibliothèques massives (par exemple `ROOT`, développée et utilisée au CERN pour analyser des téraoctets de données provenant d'expériences de physique des particules). Cependant le code apparaissant dans la feuille Jupyter reste court.

Pour écrire des programmes plus gros il est souhaitable de s'affranchir de Jupyter et de passer à des **programmes compilés**. C'est en fait l'usage le plus courant de `C++`.

Dans cette feuille, vous allez compiler vos premiers programmes C++. Dans les semaines qui viennent, nous allons progressivement basculer d'un usage majoritairement dans Jupyter à un usage majoritairement en compilé, tout particulièrement pour le projet.

5.3.1 Premier exemple

Votre répertoire `ProgImperative/Semaine5/` contient un fichier max.cpp, dont voici le contenu (ignorer le `%% file max.cpp`) :

```
%%file max.cpp
#include <iostream>
using namespace std;

float max(float a, float b) {
    if ( a >= b ) {
        return a;
    } else {
        return b;
    }
}

int main() {
    cout << max(1.5, 3.0) << endl;
    cout << max(5.2, 2.0) << endl;
    cout << max(2.3, 2.3) << endl;

    return 0;
}
```

C'est un programme indépendant. Il n'est pas directement compréhensible par un ordinateur ; il faut au préalable le transformer, le **compiler**. Pour cela, nous allons utiliser le compilateur g++ (GNU C++ Compiler) avec la commande suivante du terminal :

```
g++ max.cpp -o max
```

Exécuter cette commande produit un fichier max qui. lui, est en binaire, illisible pour un humain, mais directement exécutable par l'ordinateur. Nous pouvons l'exécuter depuis un terminal avec

```
./max
```

Syntaxe et sémantique; fonction `main`

Revenons au fichier max.cpp. Vous remarquerez qu'il y a plusieurs fonctions, mais aucune instruction en dehors de ces fonctions. Si on regarde l'affichage, on constate que la fonction `main` a été appellée (et qu'elle a appelé à son tour la fonction `max`). Il est temps d'expliquer tout cela en détail.

Syntaxe :

Un programme à compiler est composé d'un fichier avec extension `.cpp` contenant :
- Éventuellement un entête : inclusion de bibliothèque, déclaration de variables, …
- Une ou plusieurs fonctionsUne de ces fonctions doit s'appeler `main` (*fonction principale*). Elle ne prend pas de paramètre et renvoie un entier :

```
int main () {
    ...
    return 0;
}
```

Sémantique :
- Au lancement du programme, la fonction `main` est exécutée
- Convention : la fonction renvoie :
  - 0 si l'exécution du programme s'est déroulée normalement
  - Un entier différent de 0 en cas d'erreurCet entier indique quel genre d'erreur s'est produite

Si vous travaillez sur ce cours de façon autonome, cela peut être un bon moment de vous entraîner sur la feuille d'exercice `Programmes compilés` du TP.

5.3.2 La fonction `main` : paramètres ♣

Les commandes du terminal, telles que `cd` ou `mkdir` peuvent prendre des paramètres. Par exemple ici on passe le paramètre `fetch` et `Semaine5` à la commande `info-111` :

```
info-111 fetch Semaine1
```

On peut souhaiter de même écrire en C++ des programmes qui prennent des paramètres sur la ligne de commande.

Exemple :

1. Copiez le programme suivant dans un fichier `bonjour-nom.cpp` :

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

int main(int argv, char ** args) {
    string nom1 = args[1];
    string nom2 = args[2];

    cout << "Bonjour " << nom1 << "!" << endl;
    cout << "Bonjour " << nom2 << "!" << endl;
    return 0;
}
```

1. Compilez-le
2. Éxecutez le sous la forme :

```
./bonjour-nom Jean Paul
```

3. Observez le résultat ; cela devrait être :

```
Bonjour Jean!
Bonjour Paul!
```

Explications :

Les trois premières lignes ne sont pas très compréhensibles, n'est-ce pas? pour le moment, vous pouvez prendre ces lignes comme des incantations magiques.

Très brièvement : cela provient du `C` dont hérite `C++`. Il se trouve qu'il n'y a pas en `C` de chaîne de caractères à proprement parler ; on les émule avec des *pointeurs sur des caractères* : `char *`. De même pour les tableaux. Ensuite :

— `argv` contient le nombre de paramètres sur la ligne de commande : ici 2
— `**args` contient les paramètres eux-même

Tout cela vous sera expliqué en détail au second semestre.

5.3.3 Résumé

Nous avons vu comment compiler un programme `C++` et le rôle de la fonction `main`. C'est tout ce que vous avez besoin de savoir pour cette semaine. Nous reviendrons sur le rôle de la compilation plus tard dans le semestre.

5.4 Tableaux (introduction)

5.4.1 Résumé des épisodes précédents

Pour le moment nous avons vu :

— Expressions : `3 * (4+5)` `1 < x and x < 5 or y == 3`
— Variables, types, affectation : `variable = expression`
— Instruction conditionnelle : `if`
— Instructions itératives : `while`, `do ... while`, `for`
— Fonctions, procédures

Pourquoi aller plus loin?

Pour passer à l'échelle!

Manipulation de collections de données

5.4.2 Les tableaux

Exemple : Mini annuaire

On souhaite implanter un mini annuaire. Pour cela, on va stocker, pour chaque personne, un nom et un numéro de téléphone. Nous utiliserons pour chacun d'entre eux une chaîne de caractères (`string`). Noter que le numéro n'est pas un nombre : il peut contenir des espaces, etc.

Commençons par les incantations magiques usuelles :

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

Dans un premier temps, notre annuaire aura trois personnes (les noms et numéros sont factices!) :

```cpp
string nom1 = "Jeanne";
string telephone1 = "04 23 23 54 56";
```

```
string nom2 = "Franck";
string telephone2 = "03 23 42 34 26";
```

```
string nom3 = "Marie";
string telephone3 = "06 52 95 56 06";
```

Écrivons un petit programme pour afficher le contenu de l'annuaire :

```
cout << nom1 << " " << telephone1 << endl;
cout << nom2 << " " << telephone2 << endl;
cout << nom3 << " " << telephone3 << endl;
```

Est-ce satisfaisant comme manière de procéder?

Non : ce code est très redondant. Imaginez s'il y avait 100 personnes dans l'annuaire. La ligne d'affichage serait répétée 100 fois.

Répéter 3 fois (quasiment) la même ligne cela s'appelle une boucle `for`. Essayons :

```
for ( int i=1; i <= 3; i++ ) {
    cout << nom1 << " " << telephone1 << endl;
}
```

Ce n'est pas encore ce que l'on veut. En effet, pour $i = 1$, on voudrait utiliser la variable `nom1`, pour $i = 2$ la variable `nom2`, etc. Mais ce sont des variables distinctes!

La solution va être de regrouper les trois variables `nom1`, `nom2`, `nom3` contenant chacune un nom en une seule variable contenant les trois noms. Pour cela, on va utiliser un **tableau**. En C++, on utilisera le type `vector<string>` pour représenter un tableau de chaînes de caractères. Il faut au préalable charger la bibliothèque `vector` :

```
#include <vector>
using namespace std;
```

On construit un tableau pour les trois noms et un pour les trois numéros de téléphone :

```
vector<string> noms;
noms = vector<string>(3);

vector<string> telephones;
telephones = vector<string>(3);
```

`noms` et `telephones` contiennent chacun trois chaînes vide (ne vous préoccuppez pas du type affiché en deuxième ligne; c'est essentiellement `vector<string>`.

```
noms
```

```
telephones
```

Commençons à remplir le tableau. Noter que, pour accéder au i-ème nom, on utilise la syntaxe `noms[i]`.

```
noms[1] = "Jeanne";
telephones[1] = "04 23 23 54 56";
```

```
noms[2] = "Franck";
telephones[2] = "03 23 42 34 26";
```

Voyons ce que cela donne :

```
noms
```

Oups! Que s'est-il passé????

Explication : en C++, comme dans la plupart des langages, les tableaux commencent à 0. Ainsi, le premier nom est `noms[0]`, le deuxième `noms[1]`, etc.

Reprenons :

```
noms[0] = "Jeanne";
telephones[0] = "04 23 23 54 56";
```

(suite sur la page suivante)

(suite de la page précédente)

```
noms[1] = "Franck";
telephones[1] = "03 23 42 34 26";

noms[2] = "Marie";
telephones[2] = "06 52 95 56 06";
```

Maintenant, nos tableaux sont remplis correctement :

```
noms
```

```
telephones
```

Nous pouvons enfin écrire notre boucle `for` :

```
#include <iostream>
```

```
for ( int i=0; i < 3; i++ ) {
    cout << noms[i] << " " << telephones[i] << endl;
}
```

Notez bien que nous avons fait varier `i` de `0` à `2` !

Si vous êtes aventureux, regardez ce qui se passe si on fait varier `i` de `1` à `3`. Soyez prêt à redémarrer votre noyau!

Nommage : `telephone` ou `telephones` ?

Il pourrait être tentant de nommer notre variable `telephone`, pour que `telephone[i]` ressemble à `telephonei`. Mais ce n'est pas la bonne convention. Le nom d'une variable doit refléter ce qu'elle contient. Ici c'est plusieurs numéros de téléphones. Donc `telephones`.

Un annuaire avec seulement trois personnes, c'est un peu triste. Ajoutons une quatrième personne :

```
noms[3] = "Joël";
telephones[3] = "07 23 63 92 38"
```

```
noms
```

Oups! Que s'est-il passé? On a essayé de rajouter un quatrième nom dans un tableau prévu pour trois noms, et ça a tout planté.

Redémarrez votre noyau.

Reprenons à zéro : incantations magiques, création de tableau de taille 4, remplissage :

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

vector<string> noms;
noms = vector<string>(4);
vector<string> telephones;
telephones = vector<string>(4);

noms[0] = "Jeanne";
telephones[0] = "04 23 23 54 56";
noms[1] = "Franck";
telephones[1] = "03 23 42 34 26";
noms[2] = "Marie";
telephones[2] = "06 52 95 56 06";
noms[3] = "Joël";
telephones[3] = "07 23 63 92 38";
```

Réutilisons notre programme pour afficher l'annuaire :

```
for ( int i=0; i < 3; i++ ) {
    cout << noms[i] << " " << telephones[i] << endl;
}
```

Oups! Quel est le problème?

Forcément, on ne peut pas réutiliser exactement le même programme. Il faut changer le `3` en `4` :

```
for ( int i=0; i < 4; i++ ) {
    cout << noms[i] << " " << telephones[i] << endl;
}
```

Ce n'est pas très pratique de devoir à chaque fois ajuster notre programme. Imaginez le vendeur de téléphone s'il devait changer son programme chaque fois qu'il rajoute un client!

Pour éviter cela, nous allons exploiter le fait qu'un tableau connaît sa taille :

```
noms.size()
```

Nous pouvons maintenant écrire notre boucle de sorte qu'elle fonctionne pour un annuaire de taille quelconque :

```
for ( int i=0; i < noms.size(); i++ ) {
    cout << noms[i] << " " << telephones[i] << endl;
}
```

Ce n'est pas encore satisfaisant de devoir reconstruire l'annuaire chaque fois que l'on veut rajouter une personne. Imaginez si vous deviez retaper tous vos contacts sur votre téléphone chaque fois que vous souhaitez en rajouter un.

Pour pallier cela, nous allons voir une dernière opération sur les tableaux: `push_back` ; littéralement, rajouter à la fin :

```
noms.push_back("Zoé");
telephones.push_back("04 12 43 93 27");
```

```
noms
```

Observez ce qui se passe si vous exécutez à nouveau les deux cellules précédentes.

Tout ceci n'est pas encore parfait : on mélange encore beaucoup **données** et **code**. Imaginez si chaque propriétaire de téléphone portable devait réécrire le code pour afficher le contenu de son annuaire.

Un peu plus tard aujourd'hui, nous verrons comment écrire le code une bonne fois pour toutes dans une fonction réutilisable. Dans deux semaines, nous verrons comment regrouper toute l'information dans un tableau unique, où le nom et le numéro de téléphone sont proches l'un de l'autre. Plus tard dans le semestre, nous verrons comment utiliser les **fichiers** pour stocker les données complètement séparément.

Définition

À retenir :

- Un *tableau* est une valeur *composite homogène*C'est-à-dire qu'elle est formée de plusieurs valeurs du même type
- Une valeur (ou *élément*) d'un tableau t est désignée par son *indice* i dans le tableauon la note $t[i]$
- En C++ : cet indice est un entier *entre* 0 *et* $\ell - 1$,où ℓ est le nombre d'éléments du tableau

Exemple :

- Voici un tableau de six entiers : | 1 | 4 | 1 | 5 | 9 | 2 |
- Avec cet exemple, `t[0]` vaut 1, `t[1]` vaut 4, `t[2]` vaut 1, …
- Notez que l'ordre et les répétitions sont importants!

Les tableaux en C++

Exemple :

Au préalable :

```
#include <vector>
using namespace std;
```

```
vector<int> t;
t = vector<int>(6);
t[0] = 1;
t[1] = 4;
t[2] = 1;
t[3] = 5;
t[4] = 9;
t[5] = 2;
```

```
t[1] + 2 * t[3]
```

Construction des tableaux

Déclaration d'un tableau d'entiers :

```
vector<int> t;
```

— Pour un tableau de nombres réels : `vector<double>`, etc.
— ♣ `vector` est un *template*

Allocation d'un tableau de six entiers :

(on réserve de l'espace en mémoire pour ces six entiers)

```
t = vector<int>(6);
```

Initialisation du tableau :

```
t[0] = 1;
t[1] = 4;
t[2] = 1;
```

Les trois étapes de la construction d'un tableau

À retenir :

— Une variable de type tableau se construit en *trois étapes* :

1. *Déclaration*
2. *Allocation*Sans elle : *faute de segmentation* (au mieux!)
3. *Initialisation*Sans elle : même problème qu'avec les variables usuelles

Raccourci : Déclaration, allocation et initialisation en un coup :

```
vector<int> t = { 1, 4, 1, 5, 9, 2 };
```

Introduit par la norme C++ de 2011

Utilisation des tableaux

Syntaxe et sémantique

— `t[i]` s'utilise comme une variable usuelle :

```
// Exemple d'accès en lecture
x = t[2] + 3 * t[5];
y = sin( t[3]*3.14 );

// Exemple d'accès en écriture
t[4] = 2 + 3*x;
```

— En C++ *les indices ne sont pas vérifiés* !

— Le comportement de `t[i]` n'est pas spécifié en cas de débordement

— Source numéro 1 des trous de sécurité!!!

— Accès avec vérifications: `t.at(i)` au lieu de `t[i]`

Quelques autres opérations sur les tableaux

```
t.size();       // Taille du tableau `t`
t.push_back(3); // Ajout d'un élément à la fin de `t`
```

La syntaxe de ces opérations peut vous paraître suprenante : que vient le `.` faire là? Pourquoi n'écrit-on pas plutôt `size(t)`, `push_back(t,3)` ?

En fait, nous sommes en train d'utiliser de la programmation objet sans le dire. Vous n'avez pas besoin de connaître les détails pour le moment; juste de mémoriser la syntaxe un peu particulière.

Fonctions et tableaux

Nous avons dit qu'un tableau était une valeur. Il est donc tout à fait possible d'écrire une fonction qui prend un ou des tableaux en paramètres et/ou qui renvoie des tableaux.

Voici par exemple une fonction qui affiche le contenu de notre annuaire :

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

```
void affiche_annuaire(vector<string> noms, vector<string>␣
↪telephones) {
   for ( int i = 0; i < noms.size(); i++ ) {
      cout << noms[i] << " " << telephones[i] << endl;
   }
}
```

```
vector<string> noms = {"Jeanne", "Franck", "Marie", "Joël"};
vector<string> telephones = {"04 23 23 54 56", "03 23 42 34 26",
↪"06 52 95 56 06", "07 23 63 92 38"}
```

```
affiche_annuaire(noms, telephones)
```

En voici une autre qui calcule la somme des éléments d'un tableau :

```
#include<vector>
using namespace std;
```

```
int somme(vector<int> v) {
    int s = 0;
    for ( int i = 0; i < v.size(); i++) {
        s = s + v[i];
    }
    return s;
}
```

```
vector<int> v = { 1, 2, 3 };
```

```
somme(v)
```

Voir la fiche d'exercices sur les tableaux et fonctions.

5.4.3 Résumé : les tableaux

— Motivation : manipulation de collections de donnéesExemple : un annuaire

— Un **tableau** est une valeur composite formée de plusieurs valeurs du même type

— Un tableau se construit en trois étapes :
  — *Déclaration* : `vector<int> t;`
  — *Allocation* : `t = vector<int>(3);`
  — *Initialisation* : `t[0] = 3; t[1] = 0; ...`

— Utilisation : `t[i] = t[i]+1, t.size(), t.push_back(3)`

— Un tableau est une valeur comme les autres

— Il peut être passé en paramètre à ou renvoyé par une fonction

5.5 Portée des variables : variables locales et globales

Nous avons vu dans les cours précédents que l'on pouvait déclarer des variables à différents endroits :

— Directement dans une cellule Jupyter ou l'entête d'un programme :

```
int a = 1;
```

— Dans une boucle for :

```
for ( int a = 0; i <= 5; i ++ ) {
    ...
}
```

— Comme paramètre d'une fonction :

```
int f(int a) {
    ...
}
```

— Dans le corps d'une fonction :

```
int f() {
    int a;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ...
}
```

— Et en fait, plus généralement, dans un bloc :

```
{
    int a;
    ...
}
```

5.5.1 Quelques exemples

Dans un programme, le même nom peut être utilisé pour plusieurs variables, ce qui peut amener des ambiguïtés.

Devinez le résultat des commandes suivantes :

```
int i = 1;
{
    i = 2;
    i = i + 1;
}
i
```

```
int i = 1;
{
    int i = 2;
    i = i + 1;
}
i
```

```
int i = 1;
for ( int i = 1; i < 10; i++ ) {
    int i;
}
i
```

Exercice

Essayez par exemple, sans les exécuter, de prédire ce qui est affiché par les cinq cellules suivantes :

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

```
int a = 0, b = 0;
```

```
void f(int b) {
    int c = 5;
    cout << "1: " << a << " " << b << " " << c << endl;
}
```

```
void g() {
    int a = 2, c = 2;
    cout << "2: " << a << " " << b << " " << c << endl;
    {
        long a = 3, c = 3;
        cout << "3: " << a << " " << b << " " << c << endl;
    }
    cout << "4: " << a << " " << b << " " << c << endl;
    f(4);
}
```

```
g()
```

Exécutez maintenant les cellules; avez-vous le bon résultat?

La réponse pourrait en fait dépendre du langage! Pour prédire rigoureusement le comportement du programme, nous avons besoin de préciser la sémantique des variables et notamment leur **portée**.

5.5.2 La portée des variables

En C++, comme dans la plupart des langages modernes, la **portée d'une variable** (où elle est définie) est déterminée par le **contexte lexical** (comment le code est écrit) sans avoir à considérer le **contexte d'exécution** (comment le code s'exécute). Cela permet en effet de raisonner localement à une fonction, à un fichier, sans se préocuper du reste du programme.

Sémantique

- Une variable est visible depuis sa déclaration jusqu'à la fin du bloc `{ ... }` où elle est déclarée.
  Dans le cas d'une boucle `for`, le bloc comprend l'entête `for (int i ....) { ... }`.
- Une variable peut masquer des variables issues des contextes englobants.
- Une **variable locale** est définie dans un bloc (par exemple d'une fonction).
- Un **paramètre formel** est défini dans l'entête d'une fonction; il se comporte comme une variable locale.
- Une **variable globale** est définie ailleurs (cellule Jupyter, entête de programme)

Exercice

Reprenez l'exemple ci-dessus, et annotez chaque déclaration et chaque utilisation de variable : quelles variables y sont locales, globales, … Puis exécutez le pas-à-pas en dessinant la pile d'exécution pour prédire à nouveau l'affichage. Vérifiez.

Solution

```
int a = 0, b = 0;                   // Variables globales
```

```
void f(int b) {                     // Paramètre (donc local à f)
    int c = 5;                      // Variable locale de f
    // a: globale; b: paramètre de f; c: local à f
    cout << "1: " << a << " " << b << " " << c << endl;
}
```

```
void g() {
    int a = 2, c = 2;               // Variables locales à g
    // a, c: locales à g; b: globale
    cout << "2: " << a << " " << b << " " << c << endl;
    {
        long a = 3, c = 3;          // Variables locales au bloc
        // a, c: locales au bloc; b: globale
        cout << "3: " << a << " " << b << " " << c << endl;
    }
    // a, c: locales à g; b: globale
    cout << "4: " << a << " " << b << " " << c << endl;
    f(4);
}
```

```
g()
```

Rappels

- Une variable locale à une fonction n'existe que le temps d'exécution de la fonction
- La valeur de cette variable d'un appel à la fonction est perdue lors du retour au programme appelant et ne peut pas être récupérée lors d'un appel ultérieur

5.5.3 Variables globales : bonnes pratiques

- Accessible à l'intérieur de toutes les fonctions
- On peut modifier la valeur d'une variable globaleCeci est déconseillé (*effet de bord*)
- Une variable locale masque une variable globale du même nomCeci est déconseillé (ambiguïté à la lecture rapide)
- On évitera ces pratiques dans le cadre de ce cours

5.5.4 Résumé

- Une variable est définie depuis sa déclaration jusqu'à la fin du bloc la contenant(en considérant l'entête d'une fonction ou d'une boucle for comme dans le bloc)
- Une variable hors de tout bloc est dite globaleEn général, on évite les variables globales
- Sinon elle est locale

CHAPITRE 6

Modèle d'exécution, collections

6.1 Prélude

6.1.1 Résumé de l'épisode précédent

Motivation

Manipulation de collections de données

Fil conducteur : Implantation d'un annuaire

Tableaux

- Un **tableau** est une valeur **composite** formée de plusieurs valeurs du même type
- Construction :
  1. Déclaration
  2. Allocation
  3. Initialisation

Fonctionnement? Sémantique?

Généralisations?

6.1.2 Au programme

1. *Prélude: exemple jouet de piratage par débordement*
2. *Modèle de mémoire et tableaux*
3. *Collections*

6.2 Résumé du cours

- Motivation: un exemple de piratage par débordement
- Un modèle de mémoire raffiné avec pile et tas
- Sémantique:
  - Allocation des tableaux sur le tas
  - Accès à `t[i]`
  - Affectation de tableaux
  - Passage de tableaux aux fonctions par valeur
- Notion de collection
- Autres collections
- Boucle `for each`
- `auto` ♣

6.3 Exemple jouet de piratage par débordement

Le programme suivant implante un écran de connexion simple qui demande à l'utilisateur d'entrer un mot de passe et le compare avec le contenu de la variable `motDePasseSecret`, et répète tant que le bon mot de passe n'a pas été fourni.

Détail technique : il se trouve que, pour minimiser l'exemple, nous avons recours à des tableaux `C` de caractères plutôt que des `string` pour stocker le mot de passe et le mot de passe secret. D'où le `char motDePasseSecret[]`. Vous en saurez plus au deuxième semestre, mais vous pouvez ignorer la distinction pour le moment.

Instructions:

1. Essayez le programme suivant en saisissant successivement les mots de passe «coucou» et «s3*iA3».
2. Essayez le programme suivant en saisissant successivement les mots de passe «ouvre!», «Sesame,ouvre!», et «ouvre!».

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

```
char motDePasse      [] = "XXXXXX";
char motDePasseSecret[] = "s3*iA3";
```

```
do {
    cout << "Entrez le mot de passe: " << endl;
    cin >> motDePasse;
} while ( string(motDePasse) != string(motDePasseSecret) );

cout << "Connexion réussie. Bienvenue chef!" << endl;
```

6.3.1 Explications

Bien sûr, «Sésame» n'est pas une incantation magique. Il y a une explication derrière ce piratage réussi. Pour la comprendre, il vous faudra d'abord avoir lu le début du cours *Modèle de mémoire et tableaux*.

Les deux tableaux sont de longueurs fixes, et le début du tableau `motDePasseSecret` se trouve être 7 cases après le début du tableau `motDePasse`. On peut retrouver cette information par le calcul suivant dit d'"*arithmétique de pointeurs* (vous verrez plus tard pourquoi) :

```
motDePasseSecret - motDePasse
```

De ce fait, lorsque l'attaquant saisi «Sesame,ouvre!» cela déborde de `motDePasse` : le caractère «,» est écrit juste après `motDePasse`, et «ouvre!» est écrit dans `motDePasseSecret`. Cette dernière variable est donc *corrompue* : l'attaquant a pu y mettre le mot de passe de son choix à la place de celui d'origine.

```
motDePasseSecret
```

Du coup, à la tentative de connexion suivante, le mot de passe «ouvre!» est accepté.

6.4 Modèle de mémoire et tableaux

L'espace mémoire d'un programme est partagé en deux zones :
— La *pile* : variables locales des fonctions
— Le *tas* : le reste

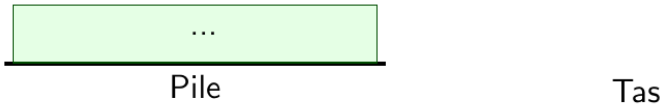
6.4.1 Exemple de construction d'un tableau

```
#include<vector>
using namespace std;
```

```
vector<int> t;       // Déclaration
t = vector<int>(6); // Allocation
t[0] = 1;           // Initialisation
t[1] = 4;
t[2] = 1;
t[3] = 5;
t[4] = 9;
```

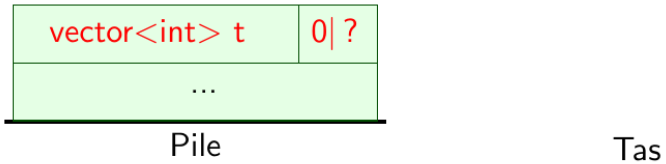
Exemple de construction d'un tableau : mémoire

1. État initial



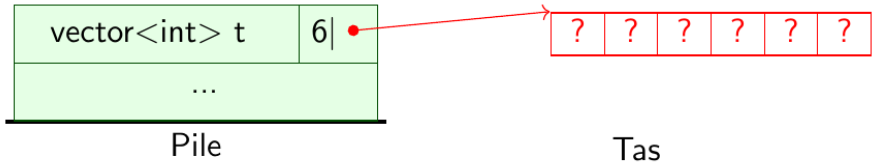
1. Déclaration du tableau

```
vector<int> t;
```



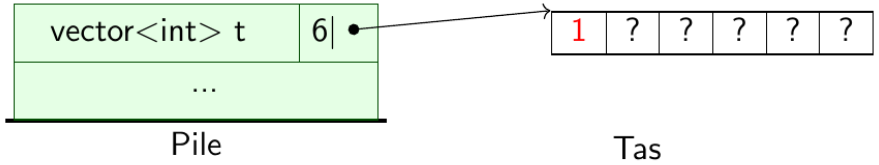
1. Allocation du tableau

```
t = vector<int>(6);
```



1. Initialisation

```
t[0] = 1;
...
```



6.4.2 Sémantique : allocation d'un tableau

```
t = vector<int>(6);
```

1. Une suite contiguë de cases est allouée sur le tas
2. La taille et une référence vers la première des cases est stockée dans `t`

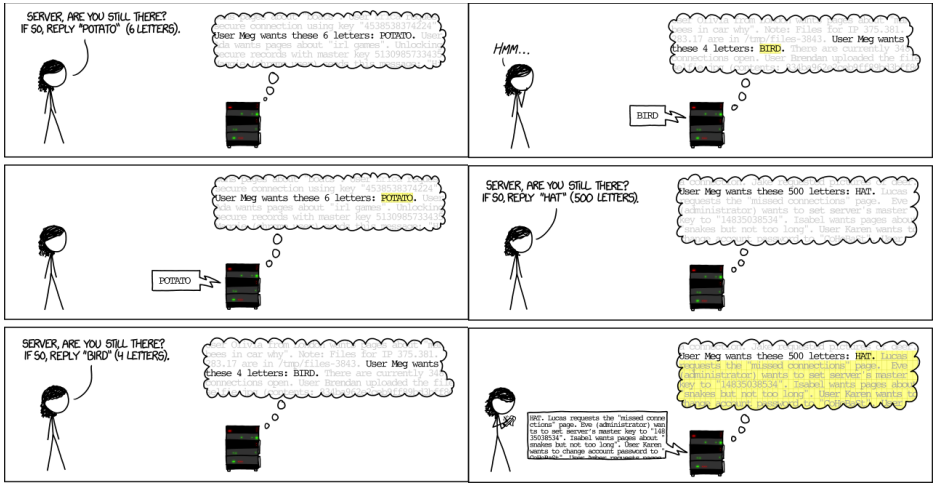
6.4.3 Sémantique : lecture et écriture dans un tableau

```
t[i]
```

— Donne la `i`-ème case du tableau
— Obtenue en suivant la référence et se décalant de i cases
— Rappel : *pas de vérifications!!!*

6.4.4 Retour sur l"exemple de piratage par débordement

Heartblead expliqué (http://xkcd.com/1354/)



6.4.5 Tableaux et allocation mémoire

À retenir :

— Une valeur de type tableau ne contient pas directement les cases du tableau, mais l'adresse en mémoire de celles-ci (référence) et la taille du tableau.

— Une variable de type tableau se construit en trois étapes :

1. Déclaration
2. Allocation Sans cela : *faute de segmentation* (au mieux!)
3. Initialisation Sans cela : même problème qu'avec les variables usuelles

— Lors de l'accès à une case `i` d'un tableau `t`, il faut toujours vérifier les bornes : `0 <= i` et `i < t.size()` Sans cela : *faute de segmentation* (au mieux!)

6.4.6 Sémantique : affectation de tableaux

Exemple :

Quelles sont les valeurs de `t[0]` et `t2[0]` après l'exécution du programme suivant?

```
vector<int> t = { 1, 4, 1, 5, 9, 2 };
vector<int> t2;
t2 = t;              // Affectation
t2[0] = 0;
```

À retenir :

— En C++, lors d'une affectation, un `vector` est *copié* !
— On dit que `vector` a une **sémantique de copie**.
— Différent de Java, Python ou des `array` en C!

6.4.7 Sémantique : tableaux et fonctions ♣

Exemple :

Quelle est la valeur de `tableau[0]` après l'exécution du programme suivant?

```
void modifie(vector<int> tableau) {
    tableau[0] = 42;
}
```

```
vector<int> tableau = { 1, 2, 3, 4 };
modifie(tableau);
```

Fonctions et tableaux :

— Affectation des paramètres $\Longrightarrow$ copie
— Donc, les `vector` de C++ sont passés **par valeur** aux fonctions

— Mais la fonction peut renvoyer le tableau modifié!

6.5 Collections et boucle «pour tout ... dans ...»

6.5.1 Quelques exemples de collections

Rappel : tableaux (`vector`)

Nous avons vu qu'un tableau (`vector` en C++) regroupe plusieurs valeurs dans un ordre donné, en tenant compte des répétitions :

```
#include<vector>
using namespace std;
```

```
vector<int> v = { 3, 2, 5, 2 };
```

```
v
```

Cela donne un sens au i-ème élément d'un tableau :

```
v[3]
```

Un tableau est un exemple de **valeur composite homogène**, ou **collection** qui regroupe plusieurs valeurs d'un même type.

Il existe d'autres types de collections. Voyons quelques exemples.

Ensembles (`set`)

Lorsque l'on souhaite ne tenir compte ni de l'ordre, ni des répétitions – comme dans un ensemble mathématique – on peut utiliser un `set` :

```
#include <set>
using namespace std;
```

```
set<int> s = { 3, 2, 5, 2};
```

```
s
```

Du coup, accéder au i-ème élément n'a pas de sens :

```
s[1]
```

Multi-ensembles (`multiset`)

Si l'on ne souhaite pas tenir compte de l'ordre mais tout de même des répétitions, on peut utiliser un multi-ensemble (`multiset`) :

```
multiset<int> m = { 3, 2, 5, 2 };
```

```
m
```

6.5.2 Boucle `for each`

Une opération omniprésente sur une collection est de parcourir toutes ses valeurs : jusqu'ici on l'a fait à l'aide d'un index :

```
#include<vector>
#include<iostream>
using namespace std;
vector<int> v = { 3, 2, 5, 2 };
```

```
for ( int i=0; i < v.size(); i++ )
    cout << v[i] << endl;
```

C'est un peu lourd, et sujet à erreurs. Surtout, cela ne se généralisera pas aux ensembles! (pourquoi?).

À la place nous allons litérallement écrire «pour toute `valeur` dans `v` faire …» :

```
for ( auto valeur: v )
    cout << valeur << endl;
```

C'est l'équivalent du `for valeur in v:` de Python. Et cela fonctionnera tout aussi bien pour toute autre collection :

```
#include<set>
set<int> s = { 3, 2, 5, 2 };
```

```
for ( auto valeur: s )
    cout << valeur << endl;
```

```
multiset<int> m = { 3, 2, 5, 2 };
```

```
for ( auto valeur: m )
    cout << valeur << endl;
```

Digression : à propos de `auto` ♣

Vous vous demandez peut-être quelle est la signification de `auto` ?

On est en train de déclarer une nouvelle variable. Il faut donc préciser son type. On pourrait très bien mettre `int` ci-dessus puisque l'on manipule des collections d'entiers :

```
for ( int valeur: v )
    cout << valeur << endl;
```

`auto` est un type spécial qui indique à C++ de sélectionner automatiquement le type adéquat en fonction du contexte ce qui est facile ici.

```
auto i = 1;
i
```

```
auto pi = 3.14;
pi
```

CHAPITRE 7

Traitement des erreurs et exceptions

Occasionnellement, un programme, ou plus généralement une fonction, peut rencontrer une situation *exceptionnelle* : c'est-à-dire une situation qui n'est pas prévue dans les entrées normales. Que faire dans ce cas? Continuer comme si de rien était pourrait amener toutes sortes de catastrophes. Renvoyer une valeur arbitraire risquerait de passer inaperçu par l'utilisateur du programme ou de la fonction.

On souhaite donc arrêter l'exécution de la fonction, en **signalant** qu'une situation exceptionnelle s'est produite.

7.1 Exemple : gestion d'entrées invalides

Nous avons vu précédemment la fonction factorielle récursive :

```
int factorielle(int n) {
    if ( n == 0 ) return 1;
    return n * factorielle(n-1);
}
```

```
factorielle(4)
```

Que se passerait-il si on calculait `factorielle(-1)` ? Cela appellerait `factorielle(-2)` qui à son tour appellerait `factorielle(-3)`, et ainsi de suite. Jusqu'à ce que le programme plante. Allez-y, essayez pour voir ci-dessus. Et tenez-vous prêt à redémarrer le noyau!

L'entrée `-1` n'est pas prévue par la fonction. C'est une situation **exceptionnelle** que l'on souhaiterait **signaler** immédiatement et de façon plus informative que par un simple plantage.

Note : il vous faudra probablement redémarrer le noyau avant de redéfinir factorielle comme ci-dessous

```
#include <stdexcept>
using namespace std;
```

```
int factorielle(int n) {
    if ( n  < 0 ) throw invalid_argument("n doit être positif");
    if ( n == 0 ) return 1;
    return n * factorielle(n-1);
}
```

```
factorielle(4)
```

```
factorielle(-1)
```

Maintenant, l'utilisateur est satisfait : l'exécution de `factorielle` s'est immédiatement arrêté, avec un message d'erreur explicite sur le problème rencontré.

Nous allons maintenant expliquer les différents éléments utilisés : `throw` (lancer), `invalid_argument` et la bibliothèque `stdexcept`.

7.2 Signaler une exception

Syntaxe :

```
```
throw e;
```
```

Sémantique :

- Une situation exceptionnelle que je ne sais pas gérer s'est produite.
- Je m'arrête immédiatement et je préviens mon «chef» (l'utilisateur ou la fonction appelante).
- On dit qu'on **signale une exception**.
- La situation est décrite par l'exception `ee` est un objet quelconque; par exemple une **exception standard**.
- Si à son tour mon «chef» ne sait pas gérer, il prévient son «chef».
- …
- Si personne ne sait gérer, *le programme s'arrête*.

7.3 Quelques exceptions standard

Les erreurs dans les programmes peuvent être classifiée en plusieurs types classiques; ci-dessus, il s'agissait d'un «argument invalide». Dans d'autre cas, cela peut être un problème d'allocation, une erreur arithmétique, etc. Cette classification aide le lecteur et l'utilisateur à mieux analyser les erreurs et, par exemple, décider du comportement à adopter face à l'une d'elle.

Pour cela, la bibliothèque standard `stdexcept` fournit plusieurs types d'**exceptions standard** qui forment une hiérarchie avec, tout en haut, le type le moins spécifique `exception`; en voici quelques unes :

- `exception`
- `runtime_error`
- `invalid_argument`, `out_of_range`, `length_error`
- `logic_error`, `bad_alloc`, `system_error`
- …

Essayez de deviner le rôle de chacune.

7.4 Gestion d'exception

7.4.1 Exemple 1

Imaginez que vous utilisiez votre traitement de texte, et que vous lui demandiez d'ouvrir un fichier qui n'existe pas. Pour la fonction en charge d'ouvrir ce fichier, c'est une situation exceptionnelle. Pour autant, serait-il acceptable que le traitement de texte plante avec juste un petit message «fichier inexistant»?

Non : pour un traitement de texte, ce n'est pas une situation exceptionnelle. Il sait gérer : prévenir l'utilisateur que le fichier n'existe pas, puis continuer normalement.

Une situation peut-être exceptionnelle pour une fonction, sans qu'elle le soit pour son «chef».

Il nous faut donc un mécanisme par lequel le chef puisse prendre la main et gérer la situation.

7.4.2 Exemple 2

Nous allons considérer un scénario similaire, en plus simple : un programme qui demande un nombre n à l'utilisateur et affiche la factorielle de ce nombre. Pour ce programme, ce n'est pas exceptionnel que l'utilisateur fasse des bêtises en saisissant une valeur invalide. Le programme sait gérer, en affichant un petit message avant de continuer.

Voilà comment cela s'écrit. Pour que tout soit au même endroit, nous rappelons d'abord la définition de la fonction factorielle.

```
#include <stdexcept>
using namespace std;
```

```
int factorielle(int n) {
    if ( n  < 0 ) throw invalid_argument("n doit être positif");
    if ( n == 0 ) return 1;
    return n * factorielle(n-1);
}
```

```
#include <iostream>
```

```
int n;
cout << "Veuillez saisir un nombre entier positif:" << endl;
cin >> n;
try {
    int f = factorielle(n);
    cout << "Factorielle n vaut " << f << endl;
} catch (invalid_argument & e) {
    cout << "Valeur de n invalide: " << n << endl;
}
cout << "Je continue ..." << endl;
```

Les deux éléments nouveaux sont `try` (essayer) et `catch` (rattraper l'erreur). En voici la signification.

7.5 Gestion des exceptions ♣

Syntaxe :

```
try {
    bloc d instructions;
} catch (type & e) {
    bloc d instructions;
}
```

Sémantique :

- Exécute le premier bloc d'instructions
- Si le bloc signale une exception de type `type`, ce n'est pas grave, je sais gérer :
  - L'exécution du premier bloc d'instruction s'interrompt
  - Le deuxième bloc d'instructions est exécuté

7.6 Résumé

Nous avons vu comment **signaler une situation exceptionnelle** (communément appellée erreur) lors de l'exécution d'un programme ou d'une fonction.

Cela se fait avec `throw e` où `e` est une **exception**. Cette dernière est typiquement construite avec l'un des types d'exceptions de la bibliothèque standard `stdexcept`.

Les fonctions appelantes peuvent alors **gérer cette exception**, à l'aide de `try ... catch ...`. Si aucune d'entre elles ne gère l'exception, alors le programme s'arrête.

Il y aurait bien d'autres choses à dire sur les exceptions, mais cela sera suffisant pour notre usage ce semestre.

CHAPITRE 8

Fichiers et flux

8.1 Prélude

8.1.1 Résumé des épisodes précédents

Pour le moment nous avons vu les concepts suivants :
- Instructions conditionnelles et itératives
- Fonctions (avec documentation et tests)
- Variables, tableaux (2D), collections

Pourquoi aller plus loin?

8.1.2 Étude de cas : afficher un annuaire

Revenons sur le programme que nous avions écrit pour afficher un annuaire :

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

vector<string> noms = {
    "Jeanne",
    "Franck",
    "Marie",
    "Joël"
```

(suite sur la page suivante)

(suite de la page précédente)

```
};
vector<string> telephones = {
    "04 23 23 54 56",
    "03 23 42 34 26",
    "06 52 95 56 06",
    "07 23 63 92 38"
};

for (int i=0; i < noms.size(); i++) {
    cout << noms[i] << " " << telephones[i] << endl;
}
```

Imaginons maintenant que le carnet de contacts sur nos téléphones adopte la même approche. Les données étant une partie intégrante du programme, nous devrions chacun avoir un programme différent, alors que seules les données changent d'une personne à l'autre.

De plus, toute modification dans l'annuaire est volatile : toute modification qui a pu avoir lieu pendant l'exécution du programme (par exemple l'ajout d'un contact) est perdue au moment où celui-ci s'arrête.

Nous avons donc besoin :
- d'une séparation claire entre programme et données;
- d'un stockage persistant des données.

C'est ce que les fichiers vont nous apporter. On voudrait pour notre annuaire avoir :
- Un fichier `annuaire.txt` :

```
Jean-Claude 0645235432
Albane      0734534534
Djamila     +1150343234
Tibo        0634534534
```

— Un programme

Reste à définir la notion de fichier et à voir comment les manipuler en C++.

8.2 Fichiers

8.2.1 Qu'est-ce qu'un fichier

Un **fichier informatique** est, au sens commun, une collection d'informations numériques réunies sous un même **nom**, enregistrées sur un support de stockage tel qu'un disque dur, un CD-ROM ou une bande magnétique, et manipulées comme une unité.

Techniquement, un fichier est une **information numérique** constituée d'une **séquence d'octets**, c'est-à-dire d'une séquence de nombres, permettant des usages divers.

En bref, c'est comme la mémoire, mais en persistant!

De même que le type d'une variable indique comment l'information est encodée dans la mémoire, le **format du fichier** indique comment l'information y est encodée :

Voyons maintenant comment on peut écrire et lire dans un fichier.

8.2.2 Écriture dans un fichier

L'exemple suivant écrit `Noel 42` dans le fichier `bla.txt` :

```
#include <fstream>
using namespace std;
```

```
ofstream  fichier;                      // Déclaration
```

```
fichier.open("bla.txt");                // Ouverture
```

```
fichier << "Noel " << 42 << endl;       // Écriture
```

```
fichier.close();                        // Fermeture
```

À retenir :

Écrire dans un fichier se fait en **quatre étapes** :

1. Déclaration
2. Ouverture du fichier
3. Écriture
4. Fermeture du fichier

8.2.3 Lecture depuis un fichier

Nous allons maintenant faire l'opération inverse : lire les informations stockées dans `bla.txt` :

```
#include <fstream>
using namespace std;
```

```
ifstream fichier;          // Déclaration
```

```
fichier.open("bla.txt");   // Ouverture du fichier
```

```
string s;
fichier >> s;              // Lecture
s
```

```
int i;
fichier >> i;
i
```

```
fichier.close();          // Fermeture du fichier
```

À retenir :

De même que l'écriture dans un fichier, la lecture se fait en **quatre étapes** :

1. Déclaration
2. Ouverture du fichier
3. Lecture
4. Fermeture du fichier

8.3 État d'un fichier (ou d'un flux)

Jusqu'ici nous avions principalement fait du calcul. Dans ces derniers, les situations exceptionnelles sont relativement rares : divisions par zéro, préconditions, …

Avec les manipulations de fichiers, nos programmes commencent à interagir avec leur environnement extérieur, environnement que nous ne contrôlons pas forcément. Il va falloir faire face à des situations exceptionnelles ou entâchées d'inconnues :

— le fichier existe-t-il?
— quelle longueur fait-il?
— est-il écrit correctement?
— y a-t-il suffisament de place sur mon disque?

De ce fait, les opérations peuvent échouer.

Dans cette feuille, nous allons voir comment détecter ces échecs pour pouvoir ensuite les gérer. Cela utilisera la notion d'**état d'un fichier**.

8.3.1 Exemple : afficher un annuaire contenu dans un fichier

Revenons à notre annuaire; celui-ci est stocké dans un fichier annuaire.txt dont voici le contenu :

```
!cat annuaire.txt
```

Pour l'afficher, nous pouvons mettre en pratique ce que nous venons de voir :

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

```
ifstream annuaire;
annuaire.open("annuaire.txt");
```

```
string nom;
string tel;
```

```
for ( int i=0; i < 4 ; i++ ) {
    annuaire >> nom;
    annuaire >> tel;
    cout << nom << ": " << tel << endl;
}
```

```
annuaire.close();
```

Mais il y a une **inconnue** sur le fichier : combien contient-il d'entrées?

Notre programme tel quel ne fonctionnera que si le fichier contient exactement quatre entrées.

Au lieu de lire un nombre fixé d'entrées, nous voudrions lire les entrées une à une tant que la lecture se passe bien.

État d'un fichier

Une variable de type fichier peut être dans un **bon état** :
— «jusqu'ici tout va bien»
ou un **mauvais état** :
— fichier non trouvé à l'ouverture, problème de permissions
— lecture ou écriture incorrecte
— fin du fichier atteinte
— plus de place disque
Syntaxe :

```
if ( fichier ) { ...
if ( fichier >> i ) { ...
```

Sémantique :

— le fichier est-il en bon état?

— la lecture s'est elle bien passée?

Remarque : si un fichier n'est pas en bon état, il est bien entendu possible de demander plus d'informations au système pour en déterminer la cause. Pour ce semestre, le test de bon état sera suffisant pour nos besoins.

Exemple : Afficher un annuaire contenu dans un fichier

Voici notre programme d'affichage d'annuaire réécrit pour lire dans le fichier tant que le fichier est en bon état. C'est-à-dire tant que la lecture de l'entrée se passe bien.

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

```
ifstream annuaire;
annuaire.open("annuaire.txt");
```

```
string nom;
string tel;
```

```
while ( annuaire >> nom  and  annuaire >> tel ) {
    cout << nom << ": " << tel << endl;
}
```

```
annuaire.close();
```

Bonne pratique : vérifier l'état d'un fichier

L'autre inconnue est : le fichier existe-t-il? Une bonne pratique est de toujours vérifier l'état d'un fichier après toute opération pouvant échouer, et notamment l'ouverture :

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

```
ifstream fichier;
```

```
fichier.open("annuaire.txt"); // Un fichier existant
```

```
if ( not fichier ) {
    cout << "Erreur à l'ouverture" << endl;
}
```

```
fichier.close();
```

```
fichier.open("oups.txt");    // Un fichier non existant
```

```
if ( not fichier ) {
    cout << "Erreur à l'ouverture" << endl;
}
```

Pour mieux signaler cette situation exceptionnelle, il est recommandé d'utiliser les **exceptions**, que nous avons introduites lors du *cours précédent* :

```
#include <stdexcept>
```

```
if ( not fichier ) {
    throw runtime_error("Erreur à l'ouverture du fichier");
}
```

8.3.2 Résumé

Lorsque l'on manipule des fichiers, les opérations sont succeptibles d'échouer. Dans cette feuille, nous avons vu que la notion d'**état de fichier** permet de détecter de tels échecs pour les traiter. Il est fortement recommandé de **systématiquement vérifier l'état du fichier après toute opération pouvant échouer**, et tout particulièrement après l'ouverture d'un fichier, et de **signaler un tel échec au moyen d'une exception**.

La notion d'état d'un fichier peut aussi être mise à profit avec l'idiome usuel «lire dans un fichier tant que la lecture se passe bien».

8.4 Cours : lecture depuis le clavier

Avant de voir comment lire et écrire dans un fichier, nous avions vu comment afficher; c'est-à-dire écrire à l'écran au moyen de `cout`. Nous avons occasionnellement vu comment lire depuis le clavier `cin`. Voici l'occasion de le formaliser.

8.4.1 Exemple

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

```
string nom;
cout << "Comment t'appelles-tu?" << endl;
cin >> nom;
cout << "Bonjour " << nom << "!" << endl;
```

8.4.2 Syntaxe et sémantique

Syntaxe :

Lecture d'une variable :

```
cin >> variable;
```

Sémantique :

- Lit une valeur du type approprié sur le clavier
- Affecte cette valeur à la variable

8.5 Notion de flux de données

- Nous avons utilisé la même syntaxe pour écrire à l'écran ou dans un fichier : `xxx << expression`
- Nous avons utilisé la même syntaxe pour lire au clavier ou depuis un fichier : `xxx >> variable`

Notez que le sens des chevrons << et >> met en valeur dans quel sens se déplacent les informations : de `expression` vers `xxx` ou de `xxx` vers `variable`.

- Un **flux sortant** est un dispositif où l'on peut écrire des données *l'une après l'autre*
- Un **flux entrant** est un dispositif où l'on peut lire des données *l'une après l'autre*

8.5.1 Exemples de flux sortants de données

- `cout` : *sortie standard* du programmeTypiquement : écran♣ Avec tampon
- `cerr` : *sortie d'erreur* du programme♣ Sans tampon
- fichiers (`ofstream`)
- chaînes de caractères (`ostringstream`)
- connexion avec un autre programme …

8.5.2 Exemples de flux entrants de données

- `cin` : *entrée standard* du programmeTypiquement : clavier
- fichiers (`ifstream`)
- chaînes de caractères (`istringstream`)
- connexion avec un autre programme …

8.5.3 Résumé

Dans ce cours, nous avons vu :

- Comment lire et écrire dans des fichiers en C++
- Un concept uniforme pour lire et écrire : les *flux*
  - Entrée et sortie standard d'un programme : `cin`, `cout`
  - Fichiers : `ifstream`, `ofstream`
  - Chaînes de caractères : `istringstream`, `ostringstream`

8.6 Lecture et écriture dans des chaînes de caractères

Nous avons vu comment lire et écrire facilement des valeurs de types divers (entiers, flottants) dans un fichier texte. Sous le capot cela a requis des opérations de natures différentes :

1. interactions avec le système pour lire et écrire du texte dans un fichier
2. conversions de valeurs en texte et réciproquement

Ne pourrait-on pas utiliser ce deuxième point pour lire et écrire des valeurs dans des chaînes de caractères? Après tout, elles contiennent du texte elles aussi! Allons-y.

Au préalable, nous aurons besoin de la bibliothèque suivante :

```
#include <sstream>
using namespace std;
```

8.6.1 Lecture depuis une chaîne de caractères

Exemple

La chaîne de caractère suivante contient – sous forme de texte – plusieurs nombres :

```
string s = "1 2 4 8 16";
```

On souhaiterait extraire ces nombres en les convertissant en entiers. Pour cela on créé un *flux* de type `istringstream` (nous verrons dans la feuille suivante ce qu'est un flux et la signification du nom `istringstream`) :

```
istringstream flux = istringstream(s)
```

Maintenant il est possible de lire le premier nombre, en procédant comme s'il était dans un fichier :

```
int i;
```

```
flux >> i;
i
```

Lorsque l'on effectue de nouvelles lectures, les nombres suivants sont lus un à un :

```
flux >> i;
i
```

```
flux >> i;
i
```

```
flux >> i;
i
```

L'exemple suivant illustre que l'on peut lire successivement des valeurs de types mixtes, et que l'on peut chaîner ces lectures :

```
string s = "truc 33 bidule 2.5 reste"
```

```
istringstream flux = istringstream(s);
```

```
string a, b;
int i;
float f;
```

```
flux >> a >> i >> b >> f;
```

```
a
```

```
b
```

```
i
```

```
f
```

Syntaxe et sémantique

Syntaxe :

```
istringstream flux = istringstream(s);
flux >> variable1;
flux >> variable2;
...
```

**Sémantique **:

Affecte aux variables `variable1`, `variable2`, … les valeurs lues successivement depuis la chaîne de caractères `s`, en tenant compte de leurs types respectifs.

8.6.2 Écriture dans une chaîne de caractères

Réciproquement, on peut souhaiter écrire le contenu de variables non pas directement à l'écran ou dans un fichier mais dans une chaîne de caractères. Le principe sera le même que pour une lecture.

Exemple

On commence par définir un flux de type `ostringstream` (là encore, nous verrons dans la feuille suivante la signification du nom de ce type) :

```
ostringstream flux;
```

Nous pouvons maintenant écrire dans ce flux :

```
flux << 3.53 << " coucou " << 1 << endl;
flux << 42 << endl;
```

et enfin extraire la chaîne de caractère produite :

```
string s = flux.str();
s
```

Syntaxe et sémantique

Syntaxe :

```
ostringstream flux;
flux << variable1;
flux << variable2;
string s = flux.str()
```

**Sémantique **:

Construit une chaîne de caractères `s` en écrivant successivement le contenu des variables `variable1`, `variable2`, … en tenant compte de leurs types respectifs.

8.6.3 Application typique : séparation calcul et entrées sorties

Ce que nous venons de voir sera très utile pour séparer proprement ce qui relève du **calcul** et ce qui relève des **entrées-sorties**. Supposons par exemple que nous souhaitions afficher joliment une entrée d'un annuaire :

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

```
void afficheEntreeAnnuaire(string nom, string prenom, string tel)
 ↪{
    cout << "Nom: " << nom
         << ", Prénom: " << prenom
         << ", Téléphone: " << tel;
}
```

```
afficheEntreeAnnuaire("Lovelace", "Ada", "07 23 23 23 23")
```

— Comment réutiliser cette fonction pour écrire le texte non pas à l'écran mais dans un fichier?
— Comment tester cette fonction?

Cela est rendu difficile par le mélange de deux actions de natures différentes dans une seule fonction :

1. Construire le texte représentant l'entrée dans l'annuaire.
2. Afficher ce texte à l'écran.

Il faut donc **séparer** ces deux actions. L'écriture dans une chaîne va nous aider à le faire.

La fonction suivante se concentre sur la construction du texte représentant l'entrée dans l'annuaire :

```
string formateEntreeAnnuaire(string nom, string prenom, string␣
↪tel) {
    ostringstream flux;
    flux << "Nom: " << nom
         << ", Prénom: " << prenom
         << ", Téléphone: " << tel;
    return flux.str();
}
```

Le texte produit est **renvoyé** par la fonction au lieu d'être **affiché**. Nous pouvons donc maintenant :

— l'affecter à une variable :

```
string s = formateEntreeAnnuaire("Lovelace", "Ada", "07 23 23 23␣
↪23")
```

— l'afficher :

```
cout << s << endl;
```

— calculer avec :

```
"- " + s
```

— ou nous en servir pour toute autre opération, comme l'écrire dans un fichier.

De plus, nous pouvons maintenant aisément tester notre fonction :

```
CHECK( formateEntreeAnnuaire("Lovelace", "Ada", "07 23 23 23 23")
       == "Nom: Lovelace, Prénom: Ada, Téléphone: 07 23 23 23 23"␣
↪);
```

8.6.4 Résumé

Dans cette feuille, nous avons vu comment **lire et écrire** des données de types divers **dans des chaînes de caractères**. Nous l'avons appliqué pour mieux séparer **fonctions qui calculent** et **fonctions d'entrées-sorties**. Enfin nous avons noté que l'on utilise la même syntaxe que pour lire et écrire dans un fichier ou pour interagir avec l'utilisateur. Cela est rendu possible par le concept de flux qui est le sujet de la *feuille suivante*.

CHAPITRE 9

Débogage et Tests

9.1 Prélude

9.1.1 Résumé des épisodes précédents ...

Pour le moment nous avons vu les concepts suivants :

- Contrôle du flot d'exécution : instructions conditionnelles et itératives, fonctions
- Gestion des données : variables, tableaux, collections, entrées et sorties, fichiers
- Méthodologie de développement : fonctions, documentation, test

Pourquoi aller plus loin?

Passage à l'échelle!

Écrire des programmes corrects

9.2 Corriger les erreurs : le débogage

9.2.1 Exemple

Le bout de code suivant contient plusieurs erreurs. Comment pourrait-on s'y prendre pour les trouver?

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

```
/** Teste si mot est un palindrome
 * @param mot une chaîne de charactères
 * @result un booléen
 **/
bool estPalindrome(string mot) {
    int n = mot.size()
    bool result = true;
    for ( int i = 0; i < n/2; i++ ) {
        if ( mot[i] != mot[n-i] ) {
            result = false;
        } else {
    result = true;}
    }
    return result;
}
```

```
CHECK( estPalindrome("a") );
CHECK( estPalindrome("1") );
CHECK( not estPalindrome("am") );
CHECK( not estPalindrome("yo") );
```

(suite sur la page suivante)

(suite de la page précédente)

```
CHECK( estPalindrome("ressasser") );
CHECK( not estPalindrome("jaune") );
CHECK( estPalindrome("aa") );
CHECK( estPalindrome("non") );
CHECK( estPalindrome("ici") );
CHECK( estPalindrome("ala") );
CHECK( estPalindrome("kayak") );
```

Erreurs trouvées :
- Erreur de syntaxe
- Erreurs de sémantique

Stratégies mises en œuvre :
- compiler le code
- exécuter le code et regarder les messages d'erreurs
- faire des tests
- lire en détail le code
- tracer l'exécution du code

9.2.2 Débogage, selon le type d'erreur

Erreurs de syntaxe

Symptômes :

Détectées par l'interpréteur ou le compilateur

Débogage :

1. Bien lire les messages d'erreurs
2. ⚠ Le compilateur pointe vers là où il détecte l'erreur⚠Pas forcément là où est l'erreur

Erreurs à l'exécution

Exemple : exemple-segmentation-fault.cpp

Symptômes :
- Segmentation fault!
- Exceptions : division par zéro, …

Débogage :
- Analyser l'état du programme au moment de l'erreur
- En Python, Java, … : regarder la pile d'appel («strack trace»)En C++ aussi avec la bonne option de compilation :

```
g++ -g -fsanitize=address exemple-segmentation-fault.cpp -o␣
↪exemple-segmentation-fault
```

- En C++ : Utilisation du débogeur!Voir plus loin.

9.2.3 Erreurs sémantiques

Symptômes :
- Le programme s'exécute «normalement»mais le résultat est **incorrect**
- Le programme ne fait pas ce que le programmeur souhaitait
- Rappel : le programme fait ce que le programmeur lui a demandé!

Difficulté : Isoler une erreur glissée dans :
- Un programme de millions de lignes
- De grosses données
- Des milliards d'instructions exécutées

Un travail de *détective!*
- Peut être très frustrant, surtout sous stress
- Peut être une très belle source de satisfaction
- Gérer ses émotions, et celle de son équipe …

9.3 Stratégies de débogage

9.3.1 Stratégie : tracer l'exécution du programme

Étapes :

1. **Instrumenter** le programmeÀ notre niveau : insérer des affichages avec `cout` ou `cerr`
2. Exécuter le programme
3. Analyser le **journal d'exécution** (log)À notre niveau : regarder ce qui est affiché

Exemple :

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

```
bool estPalindrome(string mot) {
    int n = mot.size();
    bool result = true;
    for ( int i = 0; i < n/2; i++ ) {
        // Instrumentation du code
        cout << i << " ";
        cout << result << " ";
        cout << mot[i] << " ";
        cout << mot[n-1-i] << endl;
        //
        if ( mot[i] != mot[n-1-i] ) {
            result = false;
        } else {
    result = true;}
    }
    return result;
}
```

```
estPalindrome("abcddcbz")
```

Avantages :

— Simple à mettre en œuvre

Inconvénients :

— Modification du programme, recompilation et réexécution chaque fois que l'on souhaite changer ce que l'on observe

9.3.2 Stratégie : utiliser le débogueur

Exemple : analyse post-mortem d'une erreur à l'exécution

Dans un terminal, on compile le programme (noter l'option `-g`) :

```
$  g++ -g exemple-segmentation-fault.cpp -o exemple-segmentation-
↪fault
```

Quand on l'exécute, le programme plante :

```
$  ./exemple-segmentation-fault
segmentation fault (core dumped)  ./exemple-segmentation-fault
```

«core dumped» : le programme a laissé un fichier, appelé `core`, décrivant son état au moment du plantage.

Utilisons le débogueur pour analyser cet état *post-mortem* (après plantage) :

```
$ gdb --tui exemple-segmentation-fault

(gdb) core core
...
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x000055ff8b3aa1b3 in main () at exemple-segmentation-fault.
↪cpp:7
7             v[i] = 1;
(gdb) p i
$1 = 10
```

Exemple : exécution pas à pas

Dans un terminal, on compile le programme (noter l'option `-g`) :

```
$  g++ -g est-palindrome.cpp -o est-palindrome
```

Quand on l'exécute, le résultat est incorrect :

```
$  ./est-palindrome
abcddcbz est un palindrome
```

Lançons le débogueur pour **observer l'exécution du programme :**

```
$ gdb --tui est-palidrome
GNU gdb (Ubuntu 8.3-0ubuntu1) 8.3
(gdb) b 13
Breakpoint 1 at 0x1250: file est-palindrome.cpp, line 13.

(gdb) start
Temporary breakpoint 2, main () at est-palindrome.cpp:25
25            string s = "abcddcbz";

(gdb) c
Continuing.
Breakpoint 1, estPalindrome (mot=...) at est-palindrome.cpp:13
13            int n = mot.size();
```

(suite sur la page suivante)

(suite de la page précédente)

```
(gdb) n
14            bool result = true;

(gdb) n
15            for ( int i = 0; i < n/2; i++ ) {

(gdb) n
16                if ( mot[i] != mot[n-i-1] ) {

(gdb) p i
$1 = 0

(gdb) p mot[i]
$2 = ... 'a'

(gdb) p mot[n-i-1]
$3 = ... 'z'

(gdb) n
17                    result = false;

(gdb) n
15            for ( int i = 0; i < n/2; i++ ) {

(gdb) n
16                if ( mot[i] != mot[n-i-1] ) {

(gdb) p mot[i]
$4 =  'b'

(gdb) p mot[n-i-1]
$5 =  'b'

(gdb) n
19            result = true;}
```

Résumé

Un outil essentiel : le débogueur pas à pas

Usages :
- Analyse post-mortem après un plantage du programme
- Observation du programme en cours de fonctionnement

Outils :
- En C++ : `gdb`
- En Python : module `pdb`
- Les Environnements de Développement Intégrés (`Code::Blocks`, …) fournissent des interfaces graphiques confortables pour utiliser le déboggueur.

Inconvénients :
- Un peu lourd à mettre en œuvre
- Pas encore complètement disponible dans un notebook Jupyter
- Seule l'analyse post-mortem est fonctionnelle sur le service JupyterHub

Avantages :
- Contrôle fin de l'exécution :
  - En passant à la ligne suivante (next)
  - En rentrant dans les sous fonctions (step)
  - Jusqu'au prochain points d'arrêts (conditionnels)
- Contrôle fin de l'affichage
  - Suivi des variables
  - Analyse de la pile d'exécution
  - Analyse de l'état de la mémoire
- Pas besoin de recompiler

Mode d'emploi

- Compiler avec l'option `-g` :

```
g++ -g monprogramme.cpp -o monprogramme
```

- Lancer le débogueur avec :

```
gdb --tui monprogramme
```

- Commandes du débogueur (raccourcis : `s`, `n`, `c`, `b l`, `p expr`, `q`, …)

```
start        // lance le programme en pas à pas
next         // instruction suivante
step         // instruction suivante; entre dans les fonctions
```

(suite sur la page suivante)

(suite de la page précédente)

```
continue     // continue jusqu'au point d'arrêt suivant
break l      // ajoute un point d'arrêt ligne `l`
print expr   // affiche la valeur de l'expression
quit         // quitte
```

— Commandes plus avancées ♣

```
display expr // affiche la valeur de l'expression (persistant)
continue     // continue l'exécution du programme
show locals  // affiche les variables locales
where full   // affiche la pile d'appel
help         // aide en ligne
```

9.3.3 Stratégie : réduire le problème par dichotomie

1. Caractériser le bogue : «lorsque j'appelle telle fonction avec tels paramètres, la réponse est incorrecte»En faire un test!
2. Faire une expérience pour déterminer dans quelle «moitié» du programme est l'erreur.
3. Trouver le plus petit exemple incorrectEn faire un test!
4. Exécuter pas à pas la fonction sur cet exemple
5. Trouver l'erreur
6. Corriger l'erreur
7. Vérifier les tests (non régression)
8. Rajouter des tests?

Être efficace dans la boucle essai-erreur!!!

9.3.4 Gagner du temps : développement piloté par les tests

http://fr.wikipedia.org/wiki/Test_Driven_Development

Durant le développement : Pour ajouter une nouvelle fonctionnalité :
- — Écrire les spécifications (typiquement sous forme de javadoc!)
- — Écrire le test correspondant
- — Attention aux cas particuliers!
- — Le développement est terminé lorsque les tests passent

Durant le débogage : Pour corriger un bogue signalé :
- — Écrire un test qui met en évidence le bogue
- — Le débogage est terminé quand les tests passent

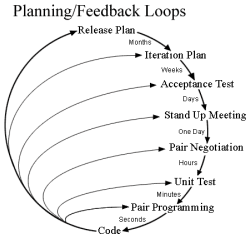
9.4 Tests : pour aller plus loin ♣

9.4.1 Approche qualité traditionnelle ♣

- — Équipes très hiérarchisées :
  - — Analystes programmeurs
  - — Équipe de test
  - — Développeur
- — Procédures strictes (voire lourdes)
- — Évolution lente, planifiée longuement à l'avance

9.4.2 Méthodes agiles 2000- ♣

Exemple : http://fr.wikipedia.org/wiki/Extreme_programming



Objectif : remettre le développeur au coeur du processus
- — Créativité
- — Responsabilité
- — Auto assurance : Tests!

Les tests sont votre outil de libération!

9.4.3 Les objectifs et types de tests ♣

- — Mesurer la qualité du code
- — Mesurer les progrès
- — Formaliser (et anticiper!) les demandes du clientTests fonctionnelsCahier de recette
- — Garantir la robustesse d'une brique avant d'empiler dessusTests unitaires
- — Anticiper la mise en commun de plusieurs briquesTests d'intégration
- — Anticiper la mise en productionTests de charge
- — Alerte immédiate si l'on introduit un bogueTests de non régression

9.4.4 Il faut aller plus loin! ♣

— Article «infecté par les tests»
— Wikipedia : Introduction au test logiciel
— Tests unitaires en Java : JUnit
— Tests unitaires en Python : pytest
— Tests unitaires en C++ : cppunit, doctest

9.5 Résumé

9.5.1 Types d'erreur

— Erreurs de syntaxe
— Erreurs à l'exécution
— Erreurs sémantiques

9.5.2 Stratégies de déboggage

— Réduire le problème
— Dévelopement incrémental :ne jamais trop s'éloigner d'une version qui marche
— Analyser les plantages
  — débogueur
— Observer l'exécution :
  — traçage
  — débogueur pas à pas

9.5.3 Tests

— Pour du code robuste
— Pour éviter ou simplifier le débogage
— Pour être serein

9.5.4 Tests automatique

— Pour être efficace
— Au S1 : `CHECK( f( ... ) == ... );`
— Au S2 : une bibliothèque de tests unitaires plus riche

CHAPITRE 10

Modèle d'exécution, collections

10.1 Prélude

10.1.1 Résumé des épisodes précédents ...

Pour le moment nous avons vu les concepts suivants:
- Contrôle du flot d'exécution: instructions conditionnelles et itératives, fonctions
- Gestion des données: variables, tableaux, collections, entrées et sorties, fichiers
- Méthodologie de développement: fonctions, documentation, test, débogage

Pourquoi aller plus loin?

Passage à l'échelle!

Écrire des «gros» programmes

10.1.2 Au programme

- Cours: *Cycle de vie d'un programme*
- Cours: *Modularité et compilation séparée*
- Cours: *Digressions: surcharge, templates, espaces de noms*
- Cours: *Conclusion*

10.2 Cycle de vie d'un programme

10.2.1 Revenons à la recette de la mousse au chocolat

Ingrédients :

250g de chocolat, 125g de beurre, 6 œufs, 50 g de sucre, café

Étapes :
- Faire fondre le chocolat avec deux cuillères d'eau
- Ajouter le beurre, laisser refroidir puis ajouter les jaunes
- Ajouter le sucre et comme parfum un peu de café
- Battre les blancs jusqu'à former une neige uniforme
- Ajouter au mélange.

Est-ce bien un programme?

Pour répondre à cette question, revenons à la définition :

Programme (rappel) : séquence d"**instructions** qui spécifie, étape par étape, les opérations à effectuer pour obtenir, à partir des **entrées**, un résultat, la **sortie**.

Oui, mais c'est quoi une instruction?

Prenons quelques candidats :

— «Lever le bras de 10 cm, tendre la main vers la gauche, prendre la casserole rouge, …»Trop détaillé, illisible, non portable

— «Préparer une mousse au chocolat»Trop abstrait et non informatif

— «avec deux cuillères d'eau»Ambigu : c'est combien une cuillère?

— «Zwei Eiweiß in Eischnee schlagen»Dans quelle langue?

Cela nous amène naturellement à la question suivante :

Quelles sont les instructions compréhensibles par un ordinateur?

10.2.2 Au commencement était l'"«Assembleur»

L'assembleur est le langage que comprend directement l'ordinateur. Chaque instruction correspond à une opération élémentaire que peut effectuer le processeur :

```
mov    -0x1c(%rbp), %edx
mov    -0x1c(%rbp), %eax
imul   %edx, %eax
mov    %eax, -0x18(%rbp)
mov    -0x18(%rbp), %eax
imul   -0x18(%rbp), %eax
mov    %eax, -0x14(%rbp)
```

Voyons cet exemple de plus près.

Exercice :

Exécuter ce fragment d'"*assembleur* en suivant les indications suivantes :

— `%eax, %edx` sont des registres (cases mémoire du processeur)l'analogue des variables

— `-0x14(%rbp), ..., -0x1c(%rbp)` : autres cases mémoire

— `mov a, b` : copier le contenu de la case a dans la case b

— `imul a, b` : multiplier le contenu de a par celui de b et mettre le résultat dans b

— Initialiser le contenu de la case `%-0x1c(%rbp)` à 3

Vous avez réussi? Bravo, vous pourrez-dire que vous avez fait de l'assembleur! Comme vous le constatez, rien de magique. Mais on est bien content de ne pas avoir à programmer tout le temps en assembleur.

10.2.3 Cycle de vie d'un programme

Motivation

Ce que je veux :

«Calculer la puissance 4 d'un nombre»

Ce que l'ordinateur sait faire :

```
...
mov    -0x1c(%rbp), %edx
mov    -0x1c(%rbp), %eax
imul   %edx, %eax
mov    %eax, -0x18(%rbp)
mov    -0x18(%rbp), %eax
imul   -0x18(%rbp), %eax
mov    %eax, -0x14(%rbp)
...
```

Passer de l'un à l'autre ne va pas se faire tout seul. Il va falloir un peu de méthode; c'est le cycle de vie d'un programme.

Cycle de vie d'un programme

Nous allons passer en revue les différentes étapes du cycle de vie d'un programme, partant d'un problème à résoudre jusqu'à un programme fonctionnel.

Problème

«Calculer la puissance 4 d'un nombre»

Formalisation

Spécification des **entrées** et des **sorties**

Scénario d'utilisation : «l'utilisateur rentre au clavier un nombre entier x ; l'ordinateur affiche en retour la valeur de x^4 à l'écran»

Recherche d'un algorithme

— Comment on résout le problème?
— Quel **traitement** appliquer à l'entrée pour obtenir la sortie désirée?

On note que $x^4 = x * x * x * x = (x^2)^2$

Cela nous donne un **Algorithme** :
— calculer $x * x$
— prendre le résultat et faire de même

Digression : La notion d'algorithme

Définition : Algorithme
— Description formelle d'un procédé de traitement qui permet, à partir d'un ensemble d'informations initiales, d'obtenir des informations déduites
— Succession finie et non ambiguë d'opérations clairement posées

Quelle différence entre un **algorithme** et un **programme** ?

Un algorithme
— doit toujours se terminer!
— est conçu pour communiquer entre humains
— est un concept indépendant du langage dans lequel il est écrit

Cycle de vie d'un programme : implantation

Et maintenant?

L'algorithme s'adresse à un humain

On veut l'expliquer à un ordinateur …

… qui est stupide; et ne comprend pas le français!

Écriture d'un programme :

En assembleur???
— Trop détaillé
— Non portable
— Illisible pour l'humain!

En C++ :

Entrée :

```
int x = 3;
```

Traitement :

```
int xCarre = x * x;
int xPuissanceQuatre = xCarre * xCarre;
```

Sortie :

```
xPuissanceQuatre
```

Niveaux de langages de programmation

Langage machine (binaire) :

Un programme y est directement compréhensible par la machine

Langage d'assemblage (ou assembleur) :

Un programme y est traduisible mot-à-mot en langage machine

Langage de programmation :

En général, un programme doit être **transformé** pour être compris par la machine

Comment faire cette transformation? À la main?

Transformer en binaire

Les interpréteurs

Exemple : interpréteur C++ dans Jupyter (xeus-cling) :

Chaîne de production :

1. L'utilisateur saisit une **instruction source**
2. L'**interpréteur** la convertit en succession d'**instructions machine**
3. Le processeur exécute les instructions

Exemples de langages de programmation avec un interpréteur : Basic, Javascript LISP, Perl, Python, C++, …

Les compilateurs

Exemple : compilation en C++ :
- Programme source : `puissance-quatre.cpp`
- Compilation : `g++ puissance-quatre.cpp -o puissance-quatre`
- Programme objet (ou binaire) : `puissance-quatre`
- Exécution : `./puissance-quatre`
- Fabrication de l'assembleur : `g++ -S puissance-quatre.cpp`
- Programme en assembleur : `puissance-quatre.s`

Exemple :

Observer le contenu de `puissance-quatre.s`, et retrouver les instructions assembleur étudiées plus haut.

Chaîne de production :

1. L'utilisateur saisit un **programme source**
2. Le **compilateur** convertit tout le programme en un **programme objet** (écrit en binaire)
3. L'utilisateur lance l'exécution du programme objet

Exemples de langages de programmation avec un compilateur : Pascal, C, C++, ADA, FORTRAN, Java, …

Exécution et mise au point

Exécuter le programme :
- Autant de fois que l'on veut!

Tester que le programme fonctionne :
- En n'oubliant pas les cas particuliers!!!

Améliorer le programme :
- Correction d'erreurs
- Optimisation du programme (rapidité, consommation mémoire)
- Optimisation de l'algorithme
- Amélioration du programme (lisibilité, généralisation)

À chaque fois, on reboucle sur l'étape de compilation, voire de conception, d'où le terme de cycle de vie.

10.2.4 Cycle de vie d'un programme : résumé

Passer d'un problème à résoudre que l'on a en tête à un programme exécutable par l'ordinateur est un grand saut. Heureusement, on peut s'appuyer sur un peu de …

Méthodologie :

1. Énoncé du problème
2. Formalisation (quel est le problème précisément?)
3. Recherche d'un algorithme (comment résoudre le problème?)
4. Programmation (implantation)
5. Interprétation / Compilation
6. Exécution
7. Mise au point (test, débogage, optimisation, diffusion)

10.2.5 Suite

Maintenant que nous avons clarifié la notion de compilation, vous êtes prêts pour la *compilation séparée*.

10.3 Modularité, compilation séparée

Rappelez vous notre **livre de recettes**. À l'époque, nous avons vu comment **découper un programme en fonctions** pour plus de **modularité**. Cela permet de mieux le comprendre, petit bout par petit bout, d'éviter les redites, …

Nous allons de même **découper un programme en plusieurs fichiers**.

10.3.1 Compilation séparée

Exemple

Considérons les deux programmes suivants :

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

int monMax(int a, int b) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    if ( a >= b )
        return a;
    else
        return b;
}
```

```
int main() {
    cout << monMax(1, 3) << endl;
    return 0;
}
```

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

int monMax(int a, int b) {
    if ( a >= b )
        return a;
    else
        return b;
}
```

```
int main() {
    cout << "Entrez a et b:" << endl;
    int a, b;
    cin >> a >> b;
    cout << "Le maximum est: "
         << monMax(a, b) << endl;
    return 0;
}
```

On constate une répétition : les deux programmes définissent exactement la même fonction `monMax`, qu'ils utilisent ensuite différemment.

Pourrait-on partager la fonction `monMax` entre ces deux programmes?

C'est ce que nous allons faire en définissant une mini bibliothèque. Voyons à quoi cela ressemble.

Exemple : la bibliothèque `max`

Contenu du fichier `max.h` :

```
/** La fonction max
 *  @param x, y deux entiers
 *  @return un entier,
 *  le maximum de x et de y
 **/
int monMax(int a, int b);
```

Contenu du fichier `max.cpp` :

```
#include "max.h"

int monMax(int a, int b) {
    if ( a >= b )
        return a;
    else
        return b;
}
```

Exemple : deux programmes utilisant la bibliothèque max

```
#include <iostream>
using namespace std;
#include "max.h"

int main() {
    cout << monMax(1, 3) << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
#include "max.h"

int main() {
    cout << "Entrez a et b:" << endl;
    int a, b;
    cin >> a >> b;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    cout << "Le maximum est: "
         << monMax(a, b) << endl;
    return 0;
}
```

Exemple : les tests de la bibliothèque `max`

Contenu du fichier `max-test.cpp` :

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

#include "max.h"

/** Infrastructure minimale de test **/
#define CHECK(test) if (!(test)) cout << "Test failed in file " <<␣
 ↪__FILE__ << " line " << __LINE__ << ": " #test << endl
```

```
void monMaxTest() {
    CHECK( monMax(2,3) == 3 );
    CHECK( monMax(5,2) == 5 );
    CHECK( monMax(1,1) == 1 );
}

int main() {
    monMaxTest();
}
```

Qu'avons nous vu?

Déclaration de fonctions

Syntaxe :

```
int monMax(int a, int b);
```

Sémantique :

— Le programme *définit* quelque part une fonction `monMax` avec cette *signature* :type des paramètres et type du résultat
— Cette définition n'est pas forcément dans le même fichier
— Si cette définition n'existe pas ou n'est pas unique, une erreur est déclenchée par le compilateur
— Cette erreur est déclenchée au moment où l'on combine les différents fichiersÉdition de liens; voir plus loin

♠ **Application :**

Deux fonctions qui s'appellent réciproquement

Compilation séparée (1)

— Un programme peut être composé de plusieurs *fichiers source*Contenu :
  — Des *définitions* de fonctions
  — Des variables globales, …
— Chaque fichier source est compilé en un *fichier objet* (extension : .o)Contenu :
  — Le code binaire des fonctions, …
— L'*éditeur de lien* combine plusieurs fichiers objet en un *fichier exécutable*

Voyons cela pour un programme voulant utiliser la bibliothèque `max` :

Les sources sont `max.cpp` et `programme.cpp`

On les compile séparément avec :

```
g++ -c max.cpp
g++ -c programme.cpp
```

Cela produit les fichiers objets `max.o` et `programme.o`. Chacun est un bout incomplet de programmes binaires : `max.o` contient le code binaire de la fonction `max` mais pas la fonction `main`, et réciproquement pour `programme.o`.

Il ne reste plus qu'à combiner ces deux bouts de programmes binaires pour obtenir un programme complet.

```
g++ programme.o max.o -o programme
```

Maintenant, on peut exécuter le programme obtenu autant de fois qu'on le souhaite :

```
./programme
```

Compilation séparée (2)

Au moment de l'édition de lien :
- Chaque fonction utilisée doit être définie une et une seule fois
- La fonction `main` doit être définie une et une seule fois

Quelques variantes autour des fichiers objets : ♣
- Bibliothèques (.a) : Une archive contenant plusieurs fichiers objets .o
- Bibliothèques dynamiques (.so) : Édition de lien dynamique au lancement du programme

Fichiers d'entête

Fichier `.h` (ou `.hpp`) contenant la *déclaration* des fonctions *définies* dans le fichier `.cpp` correspondant

Exemple : Fichier d'entête `max.h`

```
int monMax(int a, int b);
```

Syntaxe : Utilisation d'un fichier d'entête

```
#include "max.h"
```

Sémantique :

Utiliser la bibliothèque `max`

Implantation en C++ :
- Équivalent à copier-coller le contenu de `max.h` à l'emplacement du `#include "max.h"`
- ♣ Géré par le préprocesseur (cpp)

Inclusion de fichiers d'entêtes standards

Syntaxe :

```
#include <iostream>
```

Sémantique :
- Charge la déclaration de toutes les fonctions définies dans la bibliothèque standard `iostream` de C++
- Le fichier `iostream` est recherché dans les répertoires standards du système
- Sous linux : `/usr/include`, …

10.3.2 Résumé

Résumé : implantation d'une bibliothèque en C++

Écrire un fichier d'entête (max.h) :
- La déclaration de toutes les fonctions publiques
- *Avec leur documentation!*

Écrire un fichier source (max.cpp) :
- La définition de toutes les fonctions
- *Inclure le fichier .h!*

Écrire un fichier de tests (maxTest.cpp) :
- Les fonctions de tests
- Une fonction `main` lançant tous les tests

Résumé: utilisation d'une bibliothèque en C++

Inclusion des entêtes :

```
#include <iostream>  // fichier d'entête standard
#include "max.h"     // fichier d'entête perso
```

Compilation :

```
g++ -c max.cpp
g++ -c programme1.cpp
g++ max.o programme1.o -o programme1
```

En une seule étape :

```
g++ max.cpp programme1.cpp -o programme1
```

10.4 Digressions : surcharge, templates, espaces de noms, ...

Nous allons maintenant faire quelques digressions, pour pointer vers des fonctionnalités de C++ que nous avons utilisées sans explications jusqu'ici, et dont l'usage simple est naturel et potentiellement utile dès maintenant.

10.4.1 Surcharge de fonctions ♣

```
using namespace std;
```

Exemple : la fonction `monMax`

Nous avons vu maintes fois la fonction `max` sur les entiers :

```
int monMax(int a, int b) {
    if ( a < b )
        return b;
    else
        return a;
}
```

```
monMax(1,3)
```

On remarquera que le code est générique : on peut par exemple calculer de même le maximum de deux nombres réels :

```
double monMax(double a, double b) {
    if ( a < b)
        return b;
    else
        return a;
}
```

```
monMax(1.0, 3.5)
```

ou même de deux chaînes de caractères :

```
string monMax(string a, string b) {
    if ( a < b)
        return b;
    else
        return a;
}
```

```
monMax("toto", "alfred")
```

Vous remarquerez que l'on a utilisé le même nom de fonction, et que `C++` a appelé automatiquement la bonne version.

```
monMax(3, 4.5)
```

Il serait de même possible d'utiliser le même nom de fonction pour le max à trois arguments (et plus!) :

```
int monMax(int a, int b, int c) {
    return monMax(monMax(a, b), c);
}
```

```
monMax(2,4,3)
```

Surcharge de fonction

- En C++, on peut avoir plusieurs fonctions avec le même nom et des signatures différentes
- Idem en Java, mais pas en C ou en Python par exemple!
- Il est recommandé que toutes les fonctions ayant le même nom aient la même *sémantique*

Digression : templates ♣

L'exemple précédent n'est pas satisfaisant : on a dû implanter la fonction `monMax` pour chaque type sur lequel on peut faire une comparaison, en dupliquant le même code.

Correctif : les *templates* pour écrire du code *générique*

La fonction `monMax` suivante est valide pour tout type sur lequel on peut faire des comparaisons :

```
template<class T>
T monMax(T a,T b) {
    if ( a < b )
        return b;
    else
        return a;
}
```

```
monMax(1,3)
```

```
monMax(string("toot"), string("alfred"))
```

```
monMax(1.4, 0.5)
```

```
monMax(8.0, 4)
```

Les détails seront pour un autre cours!

Digression : espaces de noms ♣

Problème :

Des milliers de développeurs écrivent des bibliothèques, bibliothèques qui sont par la suite combinées entre elles de multiples façons non anticipées à l'origine. Comment s'assurer qu'il n'y a pas de conflit entre deux fonctions portant le même nom et la même signature dans deux bibliothèques différentes?

Solution :

Isoler chaque bibliothèque dans un *espace de noms*

Exemple :

La bibliothèque standard C++ utilise l'espace de nom `std`. Ainsi, dans le programme suivant, il faudrait en principe mettre `std` devant `cout`, et `endl` :

```
#include <iostream>
int main() {
    std::cout << "Bonjour!" << std::endl;
    return 0;
}
```

Notre incantation magique :

```
using namespace std;
```

dit simplement de définir des raccourcis pour rendre accessible directement tout ce qui est défini dans l'espace de nom `std`.

Vous verrez plus tard comment créer vos propres espaces de noms.

10.4.2 Suite

Vous pouvez passer à la *conclusion du cours*.

10.5 Conclusion

10.5.1 Résumé de la séance

Programmes et compilation :
- Algorithme, programme, sources, binaire/assembleur
- Interpréteur, Compilateur
- Cycle de vie d'un programme

Compilation séparée pour la modularité :
- Découper un programme non seulement en fonctions, mais en fichiers
- Bibliothèque de fonctions réutilisables entre programmes
  - fichier d'entêtes : `max.h`
  - fichier source : `max.cpp`
  - fichier de tests : `maxTest.cpp`

Digressions :
- Surcharge
- Templates
- Espaces de noms

10.5.2 Résumé des bonnes pratiques vues ce semestre (méthodologie de développement)

Développement incrémental :
- Toujours être «proche de quelque chose qui marche»
- Gestion de version (`git`, `mercurial`, ...)

Spécifications et tests :
- Spécification : définir précisément la sémantique des fonctions :qu'est-ce qu'elles doivent faire?
- Test : vérifier que la sémantique est respectée sur des exemples
- Analogie Programmer↔Prouver : Documentation↔Énoncé, Test↔Exemple, Code↔Preuve

Modularité :
- Découpage d'un programme en fonctions : *Cours 4*
- Découpage d'un programme en modules : **Aujourd'hui!**
- Découpage d'un programme en espace de noms : **Plus tard**

Conclusion : qu'avez vous vu ce semestre?

Les bases de la programmation impérative
- Contrôle du flot d'exécution : instructions conditionnelles et itératives, fonctions
- Gestion des données : variables, tableaux, collections, entrées et sorties, fichiers

Méthodologie de développement
- Développement incrémental
- Spécifications et tests
- Modularité

Une esquisse de quelques thèmes de l'informatique
- Programmation modulaire (S2)
- Algorithmes et structures de données, complexité (S2)
- Science des données (S2)
- Web, graphisme, architecture, réseau …

Ce n'est que le début de l'aventure!