Info 111 : Programmation Impérative

Nicolas M. Thiéry

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

Faculté des Sciences d'Orsay

Inspiré de cours de
Frédéric Vernière, Laurent Simon, Florent Hivert, ...

31 janvier 2021

Première partie 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 ?

A. 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 ?

$$\text{Conducteur} \neq \text{Garagiste} \neq \text{Physicien}$$

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

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

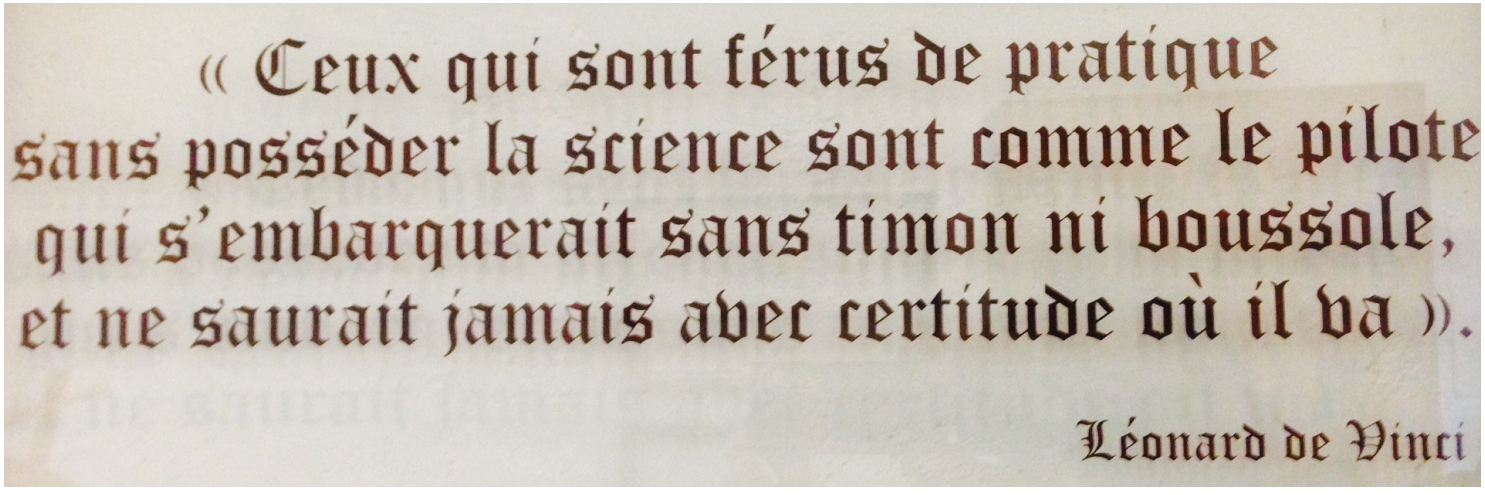
C'est quoi l'informatique en fait ?

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 *la science*
- 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 ?

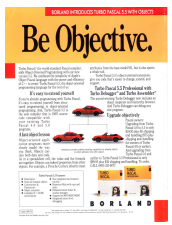
- Évidence : tous les jeunes savent utiliser un ordinateur
- Vraiment ? les-enfants-ne-savent-pas-se-servir-dun-ordinateur

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 ...

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

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 ?

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} octet !)
- 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 ...

Grands thèmes de l'informatique

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, Structures de données

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

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, ...

B. 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

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

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

C. 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, ...

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

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* : information 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`

Exemples de programmes

Ingrédients

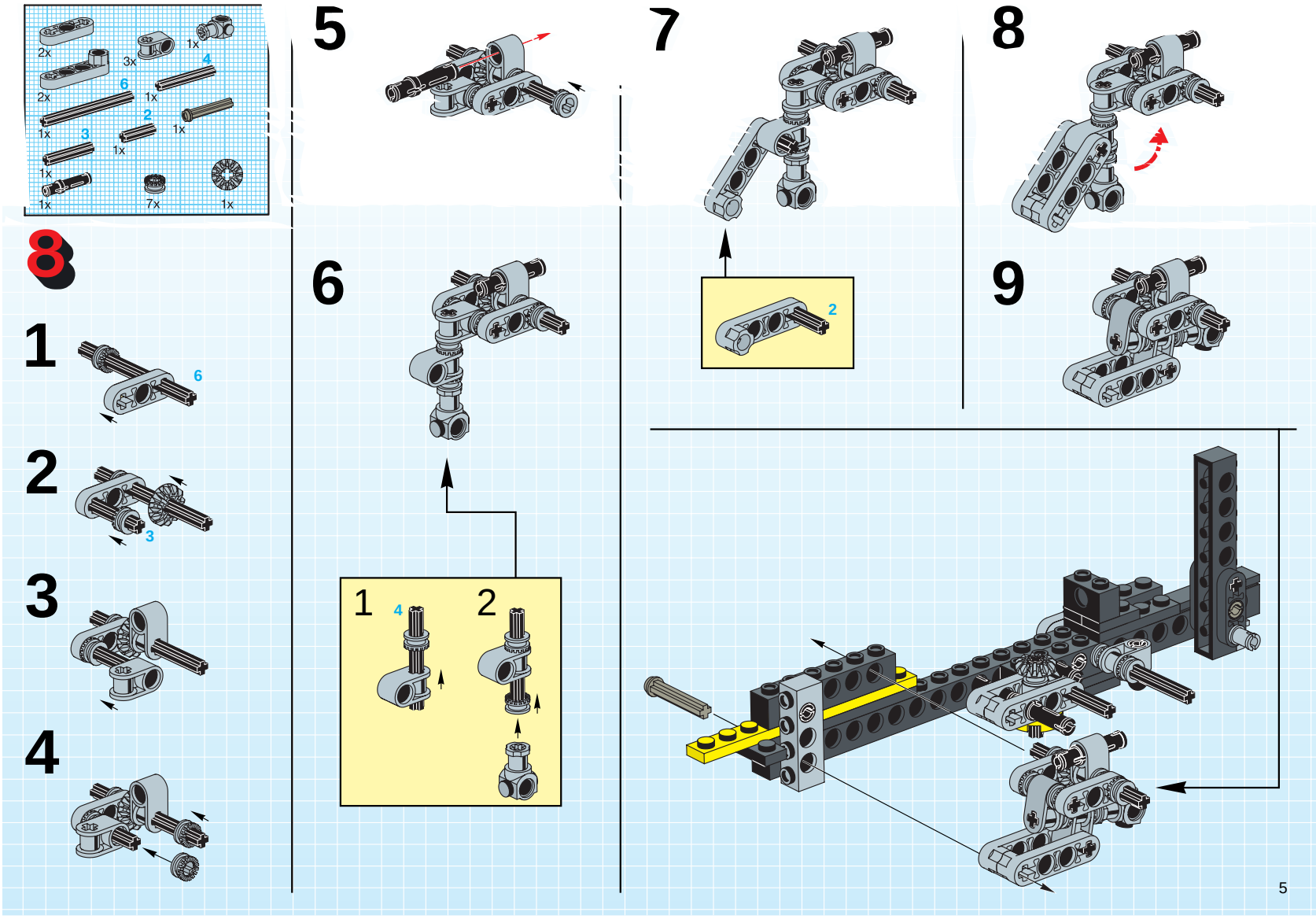
250g de chocolat, 125g de beurre, 6 œufs, 50 g de sucre, café

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 ?

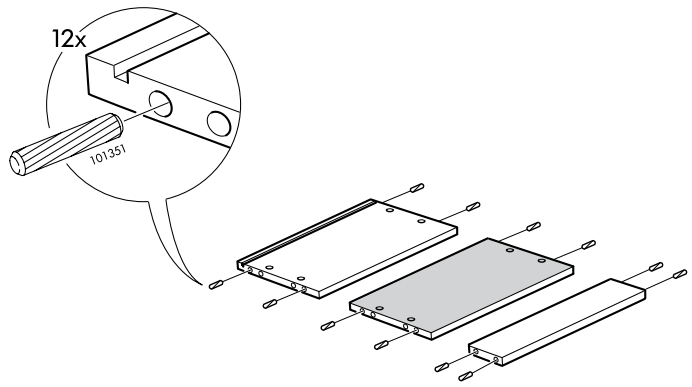
Exemples de programmes



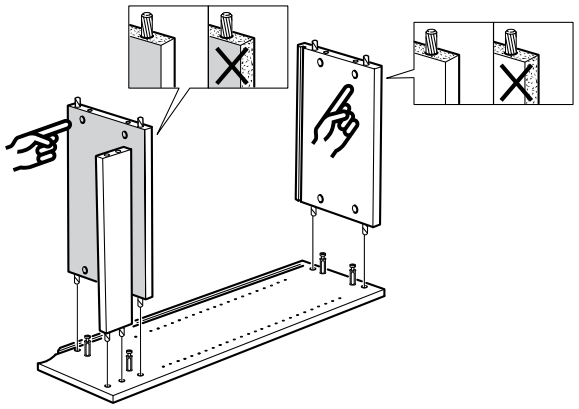
Entrée ? Sortie ?

Exemples de programmes

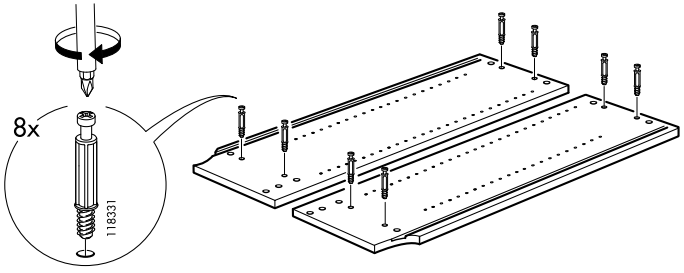
1



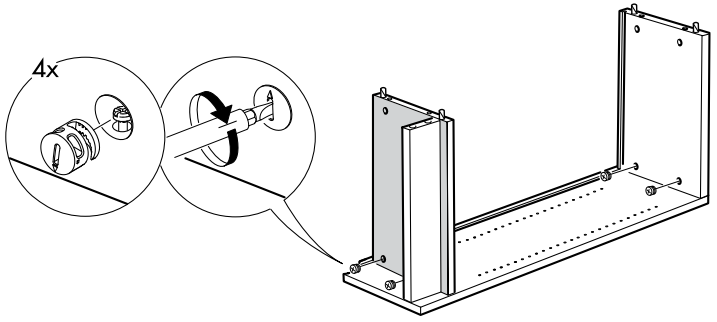
3



2



4



9 10 AA-981964-4

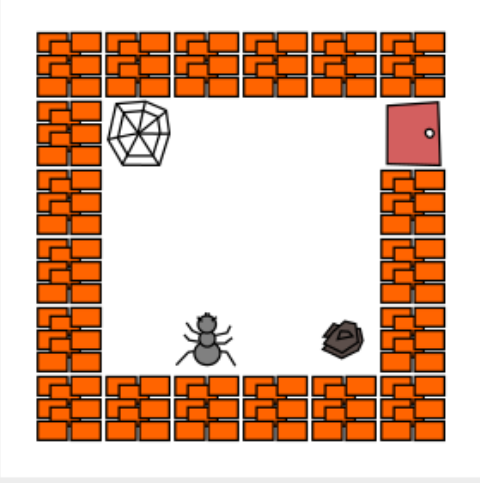
Entrée ? Sortie ?

Exemples de programmes

laby

Démo

Ce labyrinthe sert de démonstration. Le programme initial permet au robot fourmi de sortir de ce labyrinthe. Il va s'amuser un peu avec le caillou et éviter la toile d'araignée. Appuie sur le bouton qui permet de charger le programme dans le robot, et fait défiler le temps avec les flèches.



Langage : c

Niveau : 0.laby

Programme :

```
1  #include "robot.h"
2
3  void fourmi()
4  {
5    droite();
6    avance();
7    prend();
8    gauche();
9    avance();
10   pose();
11   droite();
12   avance();
13   gauche();
14   avance();
15   avance();
16   droite();
17   ouvre();
18 }
19
```

Exécuter

Précédent Suivant Rembobiner Lire Avancer

Entrée ? Sortie ?

Un exemple de programme C++

puissance-quatre.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++

puissance-quatre-jupyter.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

E. Expressions

Expressions

Combinaison de **valeurs** par des **opérations** donnant une nouvelle **valeur**

Exemples

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

Expressions booléenes

Définition (Expression booléenne)

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

Exemples

`true` `false` `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`

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

F. Variables

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.

Variables

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 sucre dans un **saladier**
  - Ajouter la farine dans le **saladier**
  - Laisser reposer ...
  - Verser le contenu du **saladier** dans ...

Variables

Notes

En C++, une variable possède quatre propriétés :

- un *nom* (ou *identificateur*)
- une *adresse*
- un *type*
- une *valeur*

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

Notion de type

Les variables peuvent contenir toutes sortes de données différentes :

- nombres entiers, réels, booléens, ...
- textes
- relevés de notes, images, musiques, ...

Définition (Notion de *type de donnée*)

- 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` ou `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*

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;
float f, g;
bool b;
```

Note : on ne peut pas redéclarer une variable avec le même nom !

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

opération	instruction	valeur de la variable après
affecter la valeur 1 à la variable `x`	`x = 1`	`x` : 1
affecter la valeur 3 à la variable `y`	`y = 3`	`y` : 3

Notes

- Affectation `x = y` : copie de la valeur
- L'ancienne valeur de `x` est perdue !
- $\neq$ transférer un ingrédient d'un récipient à l'autre

opération	instruction	valeur de la variable après
affecter la valeur `x + 1` à la variable `x`	`x = x + 1`	`x` : 2
affecter la valeur `y + x` à la variable `y`	`y = y + x`	`y` : 5

Affectation et égalité : deux concepts différents

Exemple

Exécution répétée de `x = x + 1`

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) :
« Est-ce que `x` est égal à 5 ? »
Autrement dit : est-ce que la valeur contenue dans la variable `x` est 5 ?

Fonctions

Retour sur

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 ?
Exemple sur Jupyter

Fonctions

Définition informelle

Une fonction est un petit programme :

- Entrées
- Traitement
- Sortie

Exemple

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

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

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
  - Expressions

  On reviendra dessus !
- Environnement Jupyter+Cling
- Infrastructure du cours

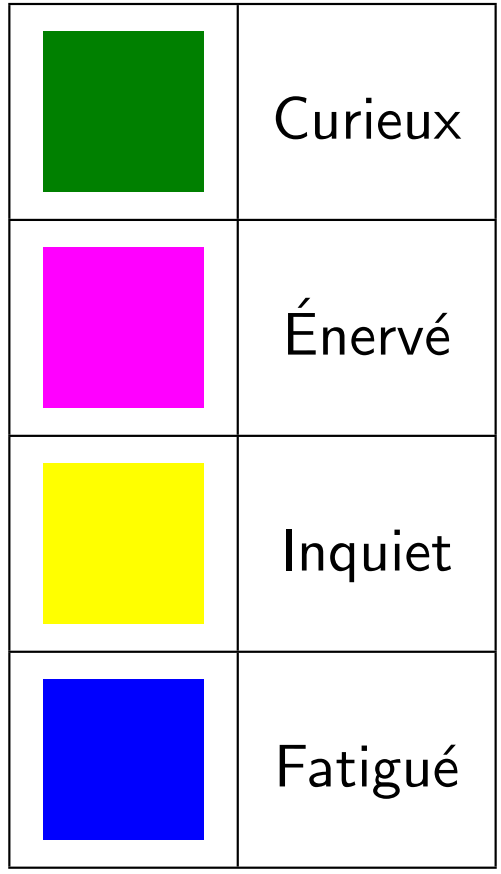
Deuxième partie 2

Langages de programmation, structures de contrôle

Résumé des épisodes précédents . . .

- Informatique : Usage, Technologie, Science
- Objectif d'Info 111 : initier à la science via la technologie
- Concrètement : bases de la programmation impérative + ...
- Premiers programmes

Comment vous sentez-vous en ce début de cours ?



A. Mémoire et variables

Un ordinateur traite de l'information.

- Il faut pouvoir la stocker : la **mémoire**
- Il faut pouvoir y accéder : les **variables**

Mémoire

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





Modèle simplifié

- Une suite contiguë de 0 et de 1 (les *bits*, groupés par *octets*)
- Pour 1Go, à raison de un bit par mm, cela ferait 8590 km
  Plus que Paris-Pékin !
- Le processeur y accède par *adresse*

Variables

Définition

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 mémoire
    En 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 contres 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 (2)

Notes

- 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 ?

non-initialisation.cpp

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

- 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 :

initialisation.cpp

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

B. Structures de contrôle

Rappel

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

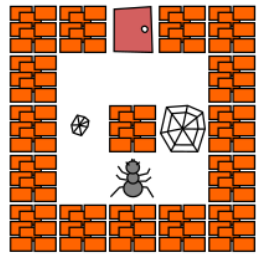
Exemple

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

Le problème

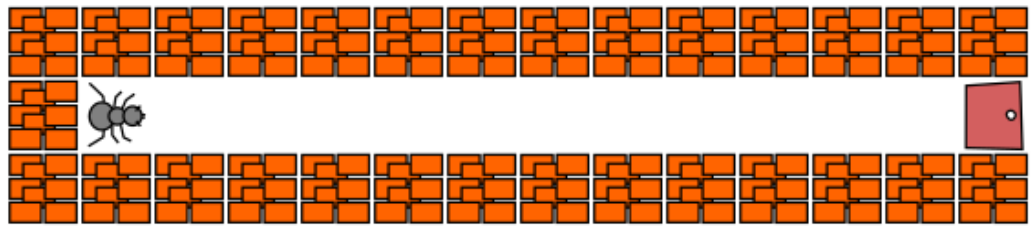
On a souvent besoin de **rompre l'exécution séquentielle** :

- Des instructions différentes, selon le contexte :



Instructions conditionnelles

- Des instructions répétées :

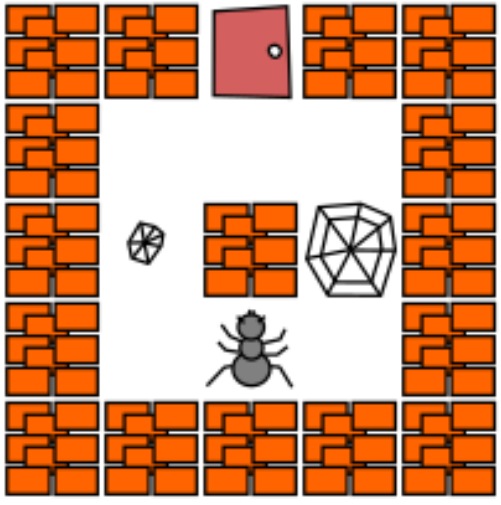


Instructions itératives

Ce sont les *structures de contrôle*

B. 2. Instructions conditionnelles

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



laby3a.cpp

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

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

Expressions booléennes, blocs d'instructions

Définition

Une *condition* est une *expression booléenne*
i.e. dont le résultat est de type booléen : vrai (`true`) ou faux (`false`)

Exemples

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

Définition

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 `true`, 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 `true`, exécution du bloc d'instructions 1
3. Si sa valeur est `false`, 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, y;                          // Les entrées
int maximum, minimum;              // Les sorties

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

Erreurs classiques avec les conditionnelles

Exemple

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

Utiliser une expression booléenne à la place !

```
bool estPositif =  x >= 0;
```

Erreurs classiques avec les conditionnelles (2)

Exercice

Que fait :

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

Attention !

Ne pas confondre « = » (affectation) et « == » (égalité) !

Erreurs classiques avec les conditionnelles (3)

Exercice

Que fait :

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

La même chose que :

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

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

Attention !

- le point-virgule est un séparateur d'instruction !
- `if (...) {...} else {...}` forme une seule instruction
- **Jamais de point-virgule avant un bloc d'instructions !**

Tests imbriqués

Exemple

Que se passe-t-il lorsque `x == 5` et `y == 4` dans l'exemple suivant :

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

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

L'indentation

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

Rappel

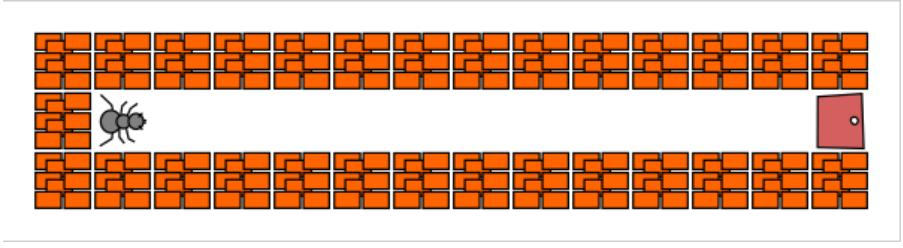
- Un programme s'adresse à un **lecteur**
- La **lisibilité** est un objectif essentiel

Notes

- 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**.

Instructions itératives : exemple



laby2a-mauvais.cpp

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

laby2a.cpp

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

B. 3. Instructions itératives

Rappel

La force d'un ordinateur est de savoir faire des tâches répétitives très rapidement et sans s'ennuyer

Exemples

- On veut afficher tous les nombres entre 1 et 1000.
- Dans un jeu sur ordinateur, à la fin d'une partie, on veut demander « voulez vous rejouer ? » et si oui recommencer une nouvelle partie.
- Tous les 1/24ème de seconde on veut afficher une image d'un film (s'il en reste)

Les instructions itératives

Définition

Les *instructions itératives* 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 groupe d'instructions.

Instructions itératives

- Boucles while : « tant que ... faire ... »
- Boucles do ... while : « Faire ... tant que ... »
- Boucles for : « Pour ... de ... à ... faire ... »

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

Syntaxe

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

Sémantique

1. Évaluation de la condition
2. Si la valeur est `true` :
   - 2.1 Exécution du bloc d'instructions
   - 2.2 On recommence en 1.

La boucle while : exemple

Exemple (Compter de 1 à 5)

```
int n = 1;

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

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 !

Exemple

```
int n = 1;

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 ! (boucle infinie)

Exemple (Que fait ce programme ?)

```
int n = 1;

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

Exemple (Erreur typique : oublier l'incrémentation !)

```
int n = 1;

while ( n <= 10 ) {
    cout << n << endl;    // Affiche la valeur de n
}
```

Une source d'erreur classique en fin de boucle

Exemple

Que vaut n à la fin du programme suivant ?

```
int n = 1;

while ( n <= 10 ) {
   n = n + 1;
}

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

Rappel

On sort de la boucle quand la condition est `false`

Le compteur est donc « un cran trop loin »

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

Exemple

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

- Jouer la partie **au moins une fois**
- Tester la condition **après** la partie

Syntaxe

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

Sémantique

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

La boucle do ... while : exemples

Exemple

```
char reponse;

do {
    ...
    cout << "Voulez-vous rejouer (o/n) ?" << endl;
    cin >> reponse;
} while ( reponse == 'o' );
```

La boucle for : « pour ... de ... à ... faire ... »

Exemple (Compter de 1 à 10)

```
int n = 1;
while ( n <= 10 ) {
    cout << n << endl;    // Affiche la valeur de n
    n = n + 1;
}
```

Schéma classique avec un **compteur** :

```
initialisation;
while ( condition ) {
    bloc d'instructions
    incrementation
}
```

Gestion du compteur dispersée ! Avec une boucle `for` :

```
for ( int n = 1; n <= 10; n = n + 1 ) {
    cout << n << endl;    // Affiche la valeur de n
}
```

La boucle for : « pour ... de ... à ... faire ... »

Syntaxe

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

Sémantique

1. Exécution de l'instruction d'initialisation
2. Évaluation de la condition
3. Si sa valeur est `true` :
   - 3.1 Exécution du bloc d'instruction
   - 3.2 Exécution de l'instruction d'incrémentation
   - 3.3 On recommence en 2.

Remarque

- **Centralise** la gestion du compteur
- Strictement équivalent à while, mais exprime une **intention**

La boucle for : exemples

Exemple

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

Variante compacte :

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

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

La boucle for : calcul de la factorielle

Exemple

On veut calculer $7! = 1 \cdot 2 \cdots 7$:

factorielle-7.cpp

```
int resultat = 1;

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

Problèmes

- Ce code sent mauvais (répétitions) !
- Et si on veut calculer 10! ou 100! ?

La boucle for : calcul de la factorielle (2)

Exemple

Entrée : un entier n

factorielle-for.cpp

```
int resultat = 1;

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

Exécution pour $n = 3$ puis pour $n = 0$

Techniques classiques de boucles

- Utilisation d'un *compteur* : `k`
  Varie toujours de la même façon
- Utilisation d'un *accumulateur* : `resultat`
  Accumule progressivement des valeurs par produit, somme, ...

Résumé

Mémoire, variables, types

- Mémoire : suite de 0 et de 1
- Variable : nom, adresse, type, valeur

Structures de contrôles

- **Instructions conditionnelles**
  - if : « si ... alors ... »
  - if/else : « si ... alors ... sinon ... »
  - Erreurs classiques
- **Instructions itératives**
  - Boucles while : « tant que ... faire ... »
  - Boucles do while : « faire ... tant que ... »
  - Boucles for : « pour ... de ... à ... faire ... »
  - Compteurs, accumulateurs
  - Erreurs classiques

Importance de la lisibilité du code

- Indentation, ...

Troisième partie 3

Fonctions

Comment vous sentez-vous en ce début de cours ?

	Curieux
	En rogne
	Inquiet
	Fatigué

A. Où en est-on ?

	Je connais la **syntaxe** et la **sémantique** de la boucle **while** ?
	J'ai pu utiliser le serveur JupyterHub depuis « chez moi »
	J'ai fini les TD et TP de la semaine dernière (hors ♣)

Rappels ...

Taux de mémorisation d'un message

1. lu : 10 %
2. entendu : 20 %
3. vu : 30 %
4. vu et entendu : 50 %
5. reformulé par soi-même : 80 %

Mémorisation sur le long terme

- Cours relu le soir même : 80 %
- Cours relu plus tard : 10 %

Objectif : **Travailler efficacement**

- Voir la page web pour des recommandations

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`

Remarque

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 ?

Passage 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

Qu'est-ce qui ne va pas ?

Duplication

- Longueurs
- En cas d'erreur ou d'amélioration : corriger plusieurs endroits !

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, ...)

- 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, pâte brisée, 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)

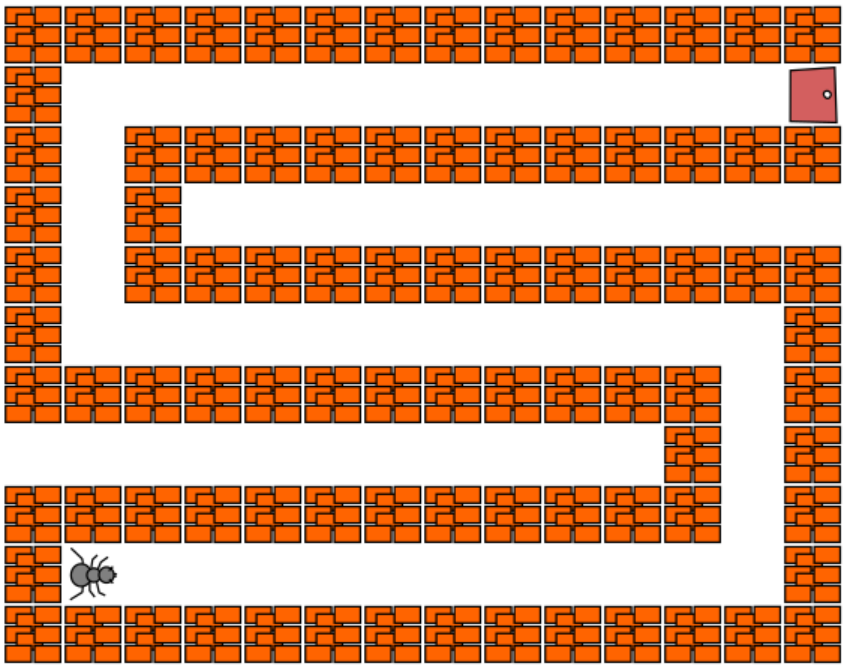
Non duplication

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

Niveau d'abstraction

- Programmes plus **concis** et **expressifs**

Une impression de déjà vu ?



laby2c-mauvais.cpp

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

laby2c.cpp

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

Une impression de déjà vu ?

Fonctions que vous avez déjà écrites

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

Appel de fonctions usuelles

fonctions-usuelles.ipynb

Chargement de la bibliothèque de fonctions mathématiques usuelles:

```
In [1]: #include <cmath>
```

Fonctions trigonométriques:

```
In [2]: cos(3.14159)

Out[2]: -1
        type: double
```

Fonction exponentielle:

```
In [3]: exp(1.0)

Out[3]: 2.71828
        type: double
```

Fonction puissance:

```
In [4]: pow(3, 2)

Out[4]: 9
        type: double

In [5]: pow(2, 3)

Out[5]: 8
        type: double
```

Appel de fonctions usuelles

On remarque

- La présence de `#include <cmath>`
  C'est pour utiliser la bibliothèque de fonctions mathématiques
  On 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**

Écrire ses propres fonctions : la fonction factorielle

fonction-factorielle.ipynb

La fonction factorielle

À la main

Calculons 5!

```
In [1]: int resultat;

In [2]: resultat = 1;
        for ( int i = 1; i <= 5; i++ ) {
            resultat = resultat * i;
        }
        resultat

Out[2]: 120
        type: int
```

Calculons 7!

```
In [3]: resultat = 1;
        for ( int i = 1; i <= 7; i++ ) {
            resultat = resultat * i;
        }
        resultat

Out[3]: 5040
        type: int
```

Écrire ses propres fonctions : la fonction factorielle

fonction-factorielle.ipynb

Avec une fonction

```
In [4]: int factorielle(int n) {
            int resultat = 1;
            for ( int i = 1; i <= n; i++ ) {
                resultat = resultat * i;
            }
            return resultat;
        }

In [5]: factorielle(5)

Out[5]: 120
        type: int

In [6]: factorielle(7)

Out[6]: 5040
        type: int

In [7]: factorielle(5) / factorielle(3) / factorielle(2)

Out[7]: 10
        type: int
```

Écrire ses propres fonctions : la fonction factorielle

fonction-factorielle.cpp

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

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

Sémantique simplifiée de l'appel de fonction

Revenons sur la fonction `max`

max.cpp

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

- `max` n'est utilisée que si elle est **appelée**
- Pour appeler cette fonction on écrit, par exemple,

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

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

D. Documentation et tests

Documentation d'une fonction (syntaxe javadoc)

Exemple

fonction-factorielle-doctests.cpp

```
/** 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 toutes les façons il faut réfléchir à ce qu'elle va faire !

Tests d'une fonction

- Pas d'infrastructure standard en C++ pour écrire des tests
- Dans ce cours, on utilisera une infrastructure minimale

Exemple

fonction-factorielle-doctests.cpp

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

Tests d'une fonction

Astuces pour être efficace

- **Commencer par écrire les tests d'une fonction**
  De toutes les façons 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

Remarque

- Les effets de bord sont durs à tester !

E. Modèle d'exécution

On considère la fonction `incremente` :

fonction-exemple-passage-par-valeur.cpp

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

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

fonction-exemple-passage-par-valeur.cpp

```
    int a, b;

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

E. Modèle d'exécution

Objectif

Comprendre précisément l'**appel de fonction**

Exemple : appel de la fonction factorielle

fonction-factorielle.cpp

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

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

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

Appel de fonctions : formalisation

Syntaxe

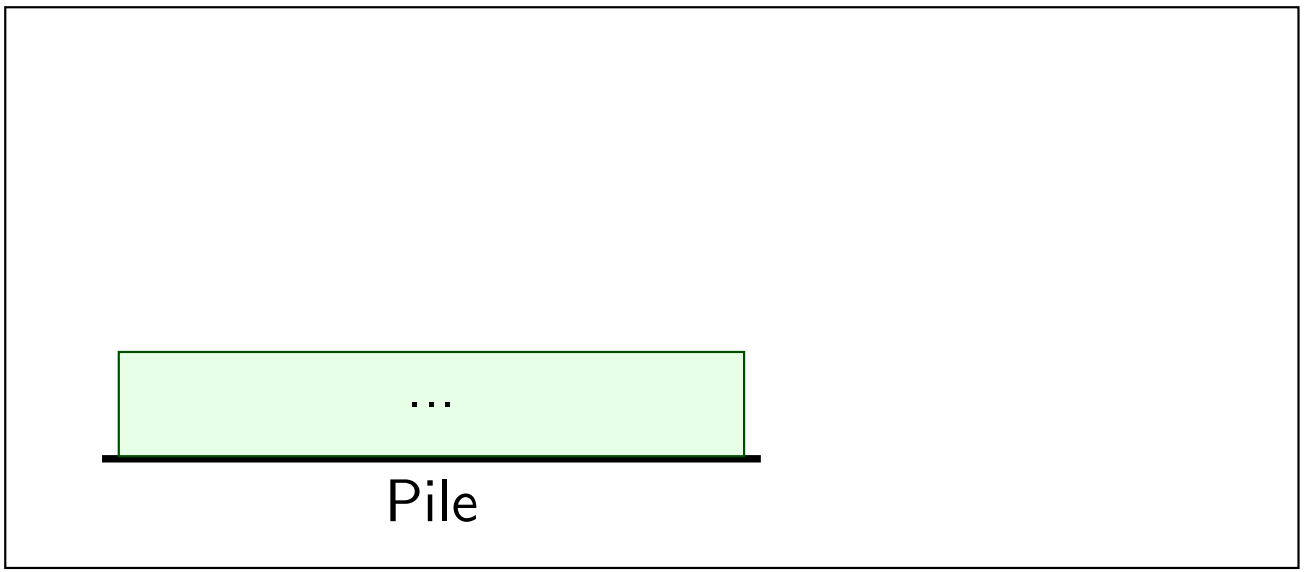
nom(expression1, expression2, ...)

Sémantique

1. Evaluation des expressions
2. Leurs valeurs sont les *paramètres réels*
3. Allocation de mémoire sur la *pile* pour :
   - Les variables locales
   - Les paramètres formels
4. Affectation des paramètres réels aux paramètres formels
   (par copie ; les types doivent correspondre !)
5. Exécution des instructions
6. Lorsque « `return expression` » est rencontré, évaluation de l'expression qui donne la *valeur de retour de la fonction*
7. Désallocation des variables et paramètres sur la pile
8. La valeur de l'expression `nom(...)` est donnée par la valeur de retour

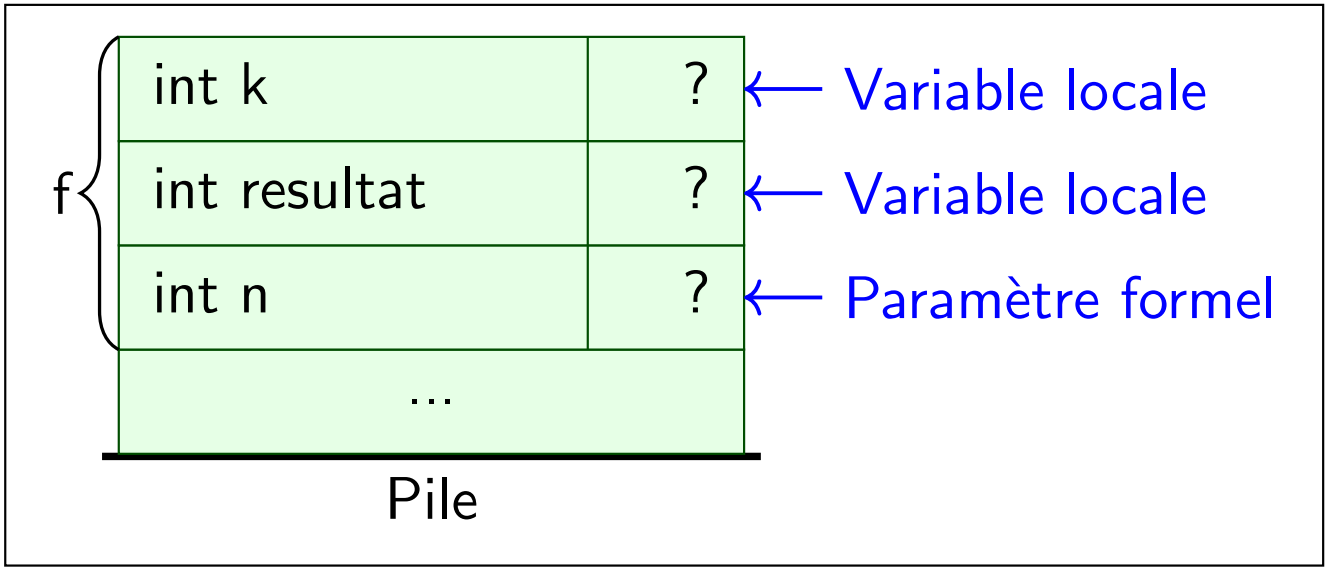
Évolution de la pile sur l'exemple :

1. État initial



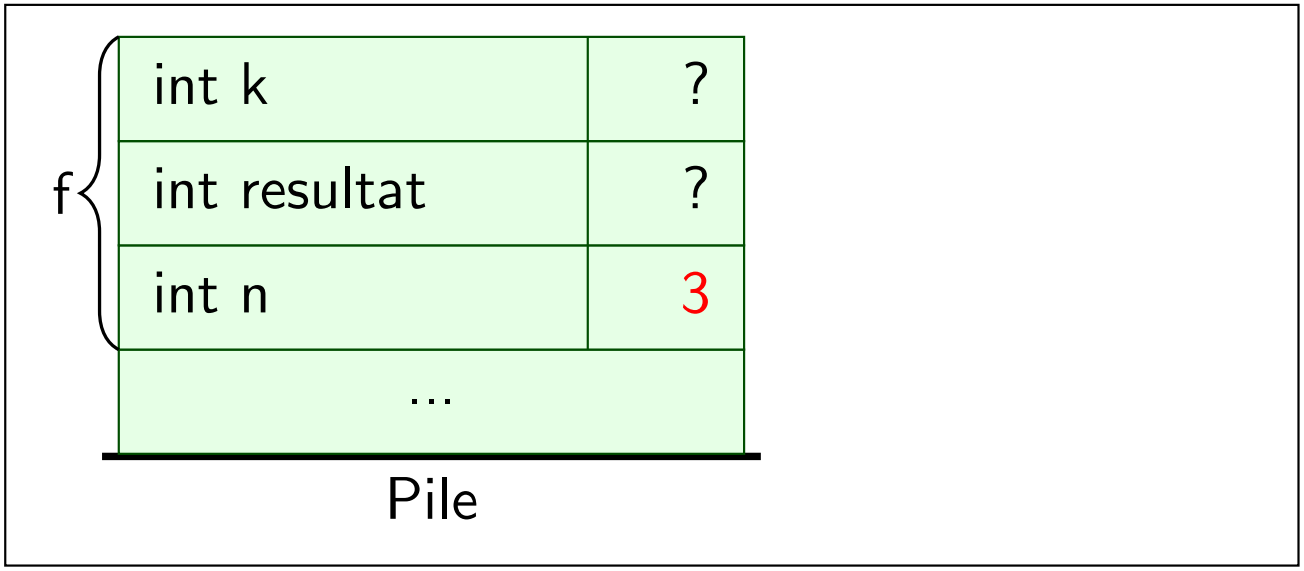
Évolution de la pile sur l'exemple :

3. Allocation de la mémoire sur la pile



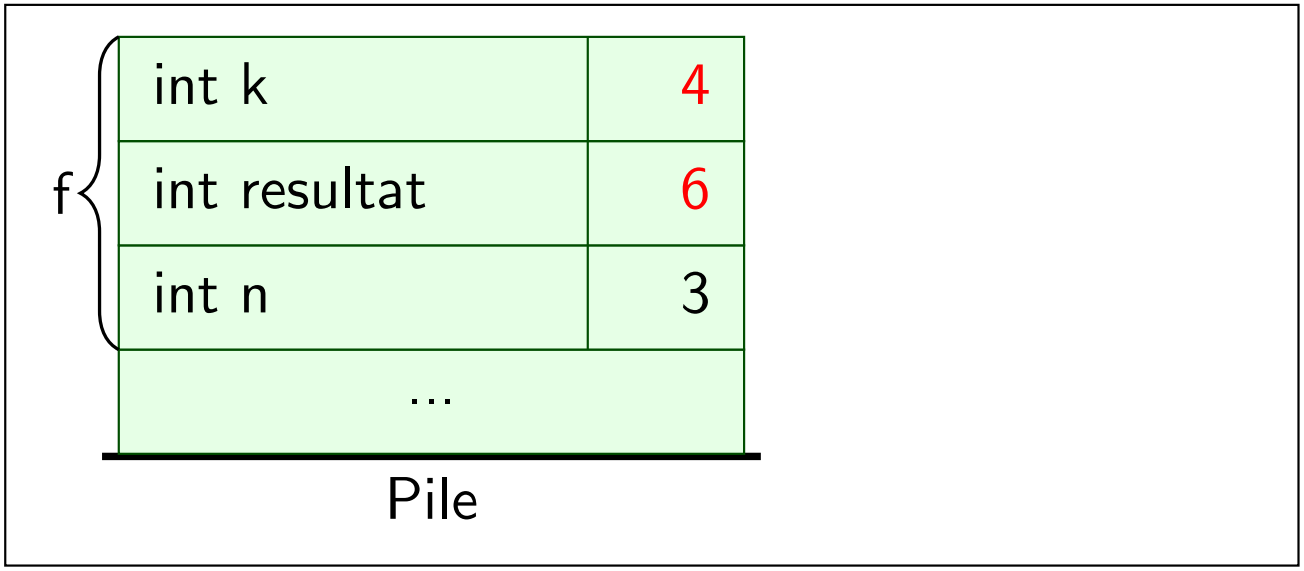
Évolution de la pile sur l'exemple :

4. Affectation du paramètre réel au paramètre formel



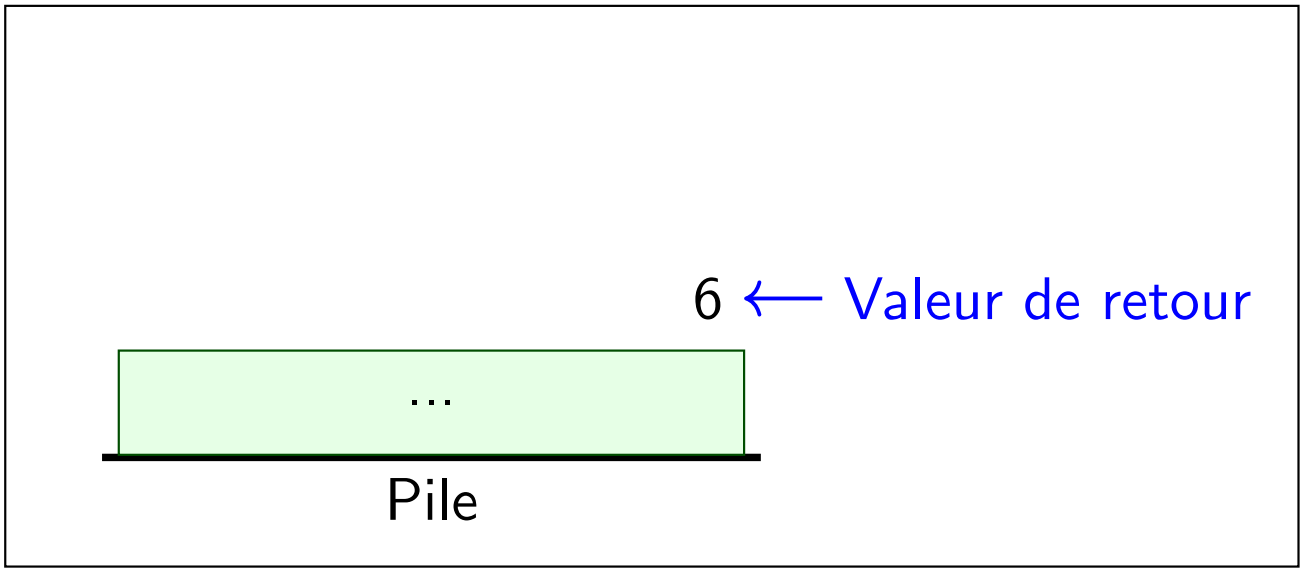
Évolution de la pile sur l'exemple :

5. Exécution des instructions



Évolution de la pile sur l'exemple :

6-8. Désallocation de la mémoire sur la pile et valeur de retour



Appel de fonctions : exercice

On considère la fonction `incremente` :

fonction-exemple-passage-par-valeur.cpp

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

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

fonction-exemple-passage-par-valeur.cpp

```
    int a, b;

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

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

En conséquence

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

On dit que les paramètres sont passés *par valeur*
Au second semestre, on verra le passage de paramètres *par référence*

F. Fonctions particulières

1. Procédures
2. Fonctions récursives

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

laby2c.cpp

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

Remarques

- 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()`

Fonctions particulières : Fonctions récursives ♣

Exercice

fonction-factorielle-recursive.cpp

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

Résumé

Motivation

- Modularité
- Lutte contre la duplication
- Programmes plus concis et expressifs

Fonction

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

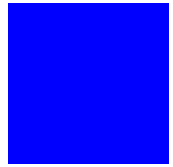
Trilogie

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

Quatrième partie 4

Tableaux (introduction)

Comment vous sentez-vous en ce début de cours ?

	Curieux
	Scrogneugneu
	Stressé
	Fatigué

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 ?

Passage à l'échelle !

Manipulation de collections de données

A. 1. Motivation

Exemple (Fil conducteur)

Implantation d'un mini annuaire

annuaire.ipynb

```
In [1]: #include <iostream>
        #include <vector>
        using namespace std;

In [2]: void afficheAnnuaire(vector<string> noms, vector<string> telephones)
            for ( int i = 0; i < noms.size(); i++ )
                cout << noms[i] << ": " << telephones[i]  << endl;
        }

In [3]: vector<string> noms =
            { "Jean-Claude", "Alban", "Tibo", "Célestin" };
        vector<string> telephones =
            { "0645235432", "0734534534", "+1150343234", "0634534534"};

In [4]: afficheAnnuaire(noms, telephones)

Jean-Claude: 0645235432
Alban: 0734534534
Tibo: +1150343234
Célestin: 0634534534
```

A. 2. Les tableaux

À retenir

- *Un tableau est une valeur composite formée de plusieurs valeurs du même type*
- *Une valeur (ou élément) d'un tableau t est désignée par son indice i dans le tableau ; on 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 tableaux de six entiers :

1	4	1	5	9	2

- Avec cet exemple, t[0] vaut 1, t[1] vaut 4, t[2] vaut 1, ...
- Noter que l'ordre et les répétitions sont importantes !

Les tableaux en C++

Exemple

tableaux.cpp

```
    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;

    resultat = t[1] + 2 * t[3];
```

Avec au préalable :

tableaux.cpp

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

A. 3. Construction des tableaux

Déclaration d'un tableau d'entiers

tableaux.cpp

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

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

Allocation d'un tableau de six entiers

tableaux.cpp

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

Initialisation du tableau

tableaux.cpp

```
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

A. 4. 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;
```

Attention !

- En C++ **les indices ne sont pas vérifiés** !
- Le comportement de `t[i]` n'est pas spécifié en cas de débordement
- Source no 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.push_back(3); // Ajout d'un élément à la fin
```

Fonctions et tableaux

tableau-fonction.ipynb

```
In [1]: #include <vector>
        using namespace std;

In [2]: int somme(vector<int> v) {
            int s = 0;
            for ( int i = 0; i < v.size(); i++) {
                s = s + v[i];
            }
            return s;
        }

In [3]: vector<int> v = { 1, 2, 3 };

In [4]: somme(v)

Out[4]: 6
        type: int
```

- Un tableau est une valeur comme les autres
- Il peut être passé en paramètre à ou renvoyé par une fonction

B. Retour sur les fonctions

- Programmes compilés et fonction `main`
- Variables locales, variables globales
- Fonctions récursives

Programmes compilés : un exemple

Jusqu'ici nous avons exécuté notre code C++ dans Jupyter.

On peut aussi écrire des programmes indépendants :

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;
}
```

Programmes compilés : la fonction `main`

Exemple

bonjour.cpp

```
int main() {
    cout << "Bonjour !" << endl;
    return 0;
}
```

À retenir

- *Un programme compilé peut être composé de plusieurs fonctions*
- *Une des fonctions doit s'appeler `main` (fonction principale)*
- *Au lancement du programme, la fonction `main` est exécutée*
- *Cette fonction doit renvoyer une valeur entière*
- *Convention :*
  - *0 si l'exécution du programme s'est déroulée normalement*
  - *Un entier différent de 0 en cas d'erreur*
    Cet entier indique quel genre d'erreur s'est produite

La fonction `main` : paramètres ♣

bonjour-nom.cpp

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

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

À l'exécution :

```
> bonjour-nom Jean Paul
Bonjour Jean !
Bonjour Paul !
```

Note

Le « `char **` » est un résidu des chaînes de caractères en C

Vous verrez au second semestre ce que font les trois premières lignes

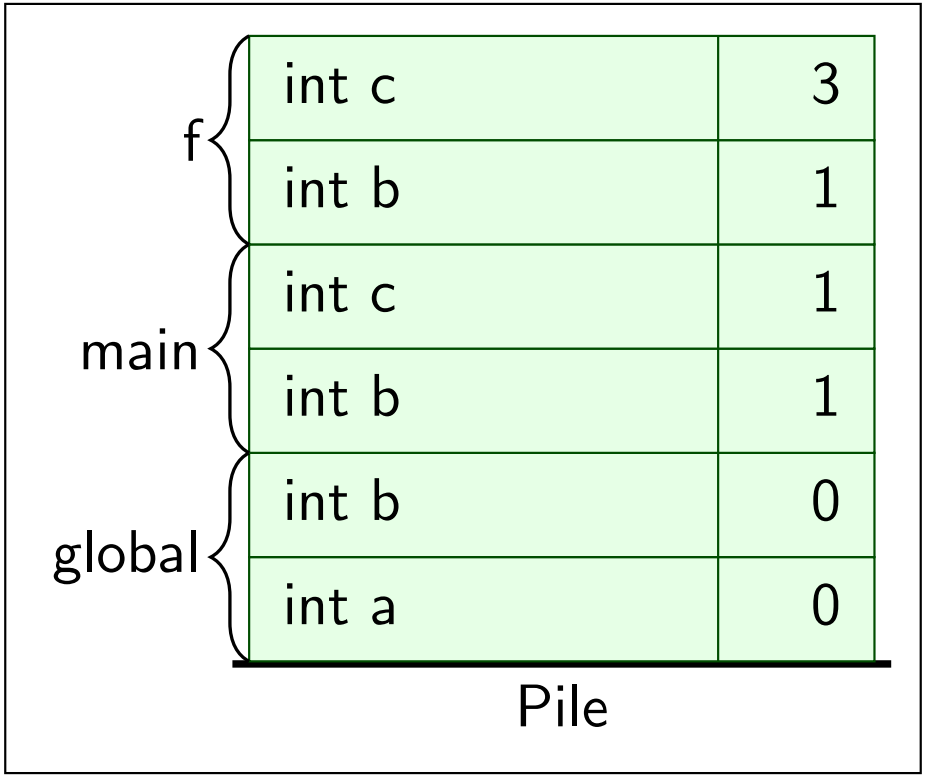
Portée des variables : un exemple

Exécutons pas à pas le programme suivant :

variables-locales-globales.cpp

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

int f(int b) {         // paramètre formel (donc local à f)
    int c = 3;         // variable locale à f
    return a + b + c;
}

int main() {
    int b = 1, c = 1; // variables locales à main
    a + b + c;         // b et c: locales à main, a: globale
    {
        long a = 2, c = 2;
        a + b + c;     // a et c: locales au bloc, b: locale à main
    }
    a + b + c;         // b et c: locales à main, a: globale
    cout << f(b) << endl;
}
```

Portée des variables : un exemple (2)



Portée des variables

Contexte lexical

- Une variable est visible depuis sa déclaration jusqu'à la fin du bloc où elle est déclarée
- Elle peut masquer des variables issues des contextes englobants
- *Variable locale* : définie dans le bloc d'une fonction
- *Paramètre formel* : définie dans l'entête d'une fonction
  se comporte comme une variable locale
- *Variable globale* : définie ailleurs (entête du programme)

À retenir

- *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 être récupérée lors d'un appel ultérieur*

Variables globales

- Accessible à l'intérieur de toutes les fonctions

Attention !

- On peut modifier la valeur d'une variable globale
  Ceci est déconseillé (*effet de bord*)
- Une variable locale masque une variable globale du même nom
  Ceci est déconseillé (ambiguïté à la lecture rapide)
- On évitera ces pratiques dans le cadre de ce cours

Fonctions récursives ♣

Exercice

On considère la fonction :

fonction-factorielle-recursive.cpp

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

Exécuter pas-à-pas l'exécution de `factorielle(3)`

C. Résumé

Tableaux

- Motivation : manipulation de collections de données
  Exemple : un annuaire
- **Tableau** : valeur composite formée de plusieurs valeurs du même type
- Construction 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)`

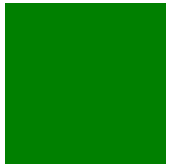
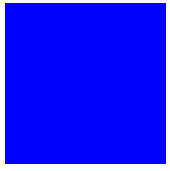
Retour sur les fonctions

- Programmes indépendants, fonction main
  On va les apprivoiser progressivement ...
- Variables locales et globales
- Fonctions récursives

Cinquième partie 5

Collections

Comment vous sentez-vous en ce début de cours ?

	Curieux
	Grrrr
	Inquiet
	Gros bâillement

A. Résumé de l'épisode précédent

Motivation

Manipulation de collections de données

Exemple (Fil conducteur)

Implantation d'un annuaire

Tableau

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

Fonctionnement ? Sémantique ?

B. Tableaux et allocation mémoire

Modèle de mémoire

L'espace mémoire d'un programme est partagé en deux zones :

- La **pile** : variables locales des fonctions
- Le **tas** : le reste

Exemple

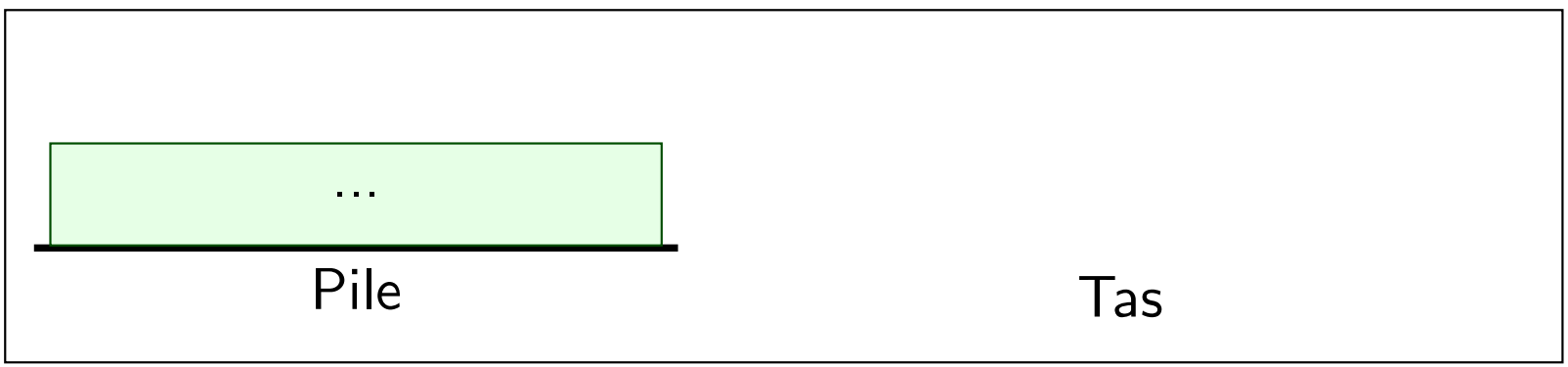
tableaux.cpp

```
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;
```

Exemple de construction d'un tableau

Étapes de la construction en mémoire

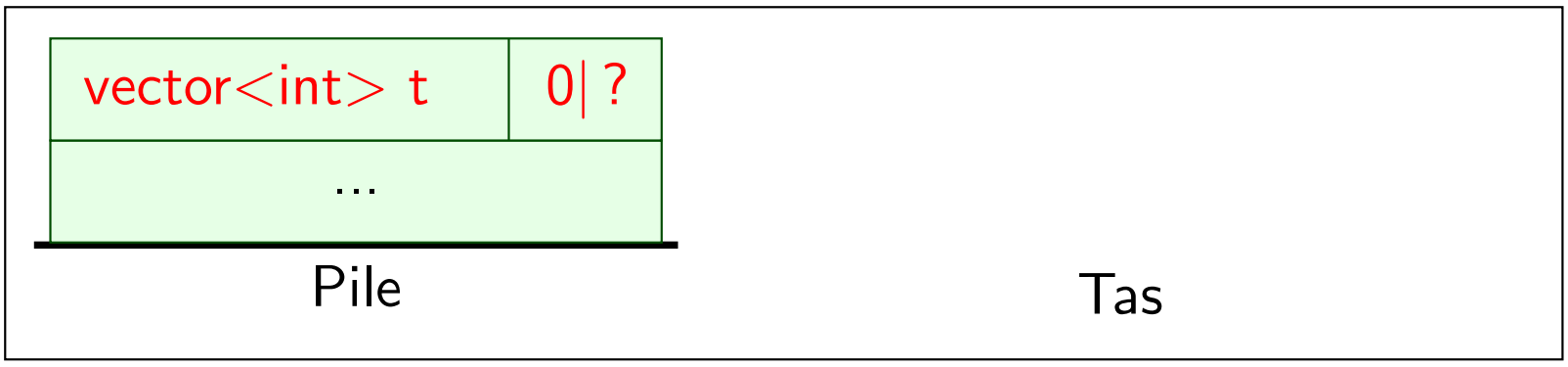
1. Déclaration du tableau
2. Allocation du tableau
3. Initialisation



Exemple de construction d'un tableau

Étapes de la construction en mémoire

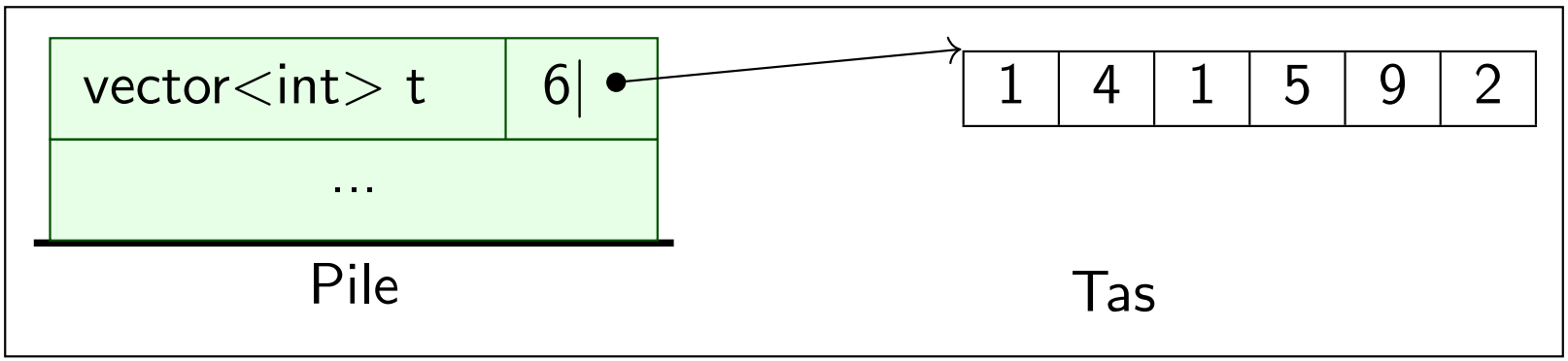
1. Déclaration du tableau
2. Allocation du tableau
3. Initialisation



Exemple de construction d'un tableau

Étapes de la construction en mémoire

1. Déclaration du tableau
2. Allocation du tableau
3. Initialisation



Sémantique (Allocation d'un tableau)

tableaux.cpp

```
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`

Sémantique (Lecture et écriture dans un tableau)

```
t[i]
```

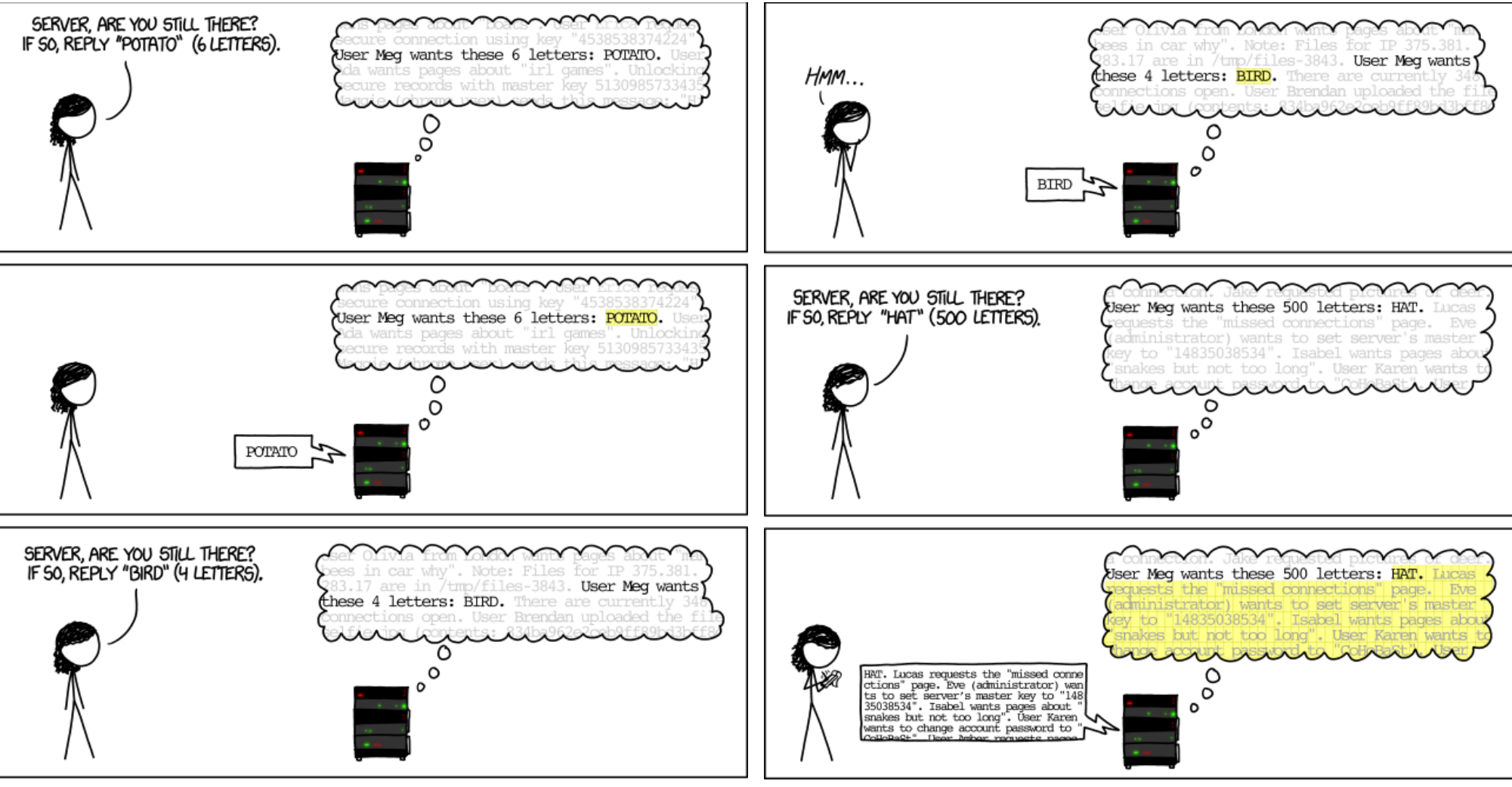
- Donne la `i`-ème case du tableau
- Obtenue en suivant la référence et se décalant de i cases
- Rappel : **pas de vérifications ! ! !**

Exemple de piratage par débordement ♣

login.cpp

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

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



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 !)*

Affectation de tableaux (sémantique)

Exemple

Qu'affiche le programme suivant ?

tableaux-copie.cpp

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

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 !*

Affectation de tableaux (sémantique ♣)

Exemple

tableau-fonction-valeur.cpp

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

int main() {
    vector<int> tableau = { 1, 2, 3, 4 };
    modifie(tableau);
    cout << tableau[0] << endl;
}
```

Fonctions et tableaux

- Affectation des paramètres $\Longrightarrow$ copie
- Donc les `vector` sont passés *par valeur* aux fonctions
- Mais la fonction peut renvoyer le tableau modifié !

Affectation de tableaux (sémantique ♣)

Exemple

tableaux.cpp

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

Sémantique ≠ fonctionnement interne exact

- Pour préserver les performances, `vector` suit le patron de conception de *copie-en-écriture* : la copie n'a lieu que lorsqu'elle est nécessaire

C. Tableaux, collections et vecteurs en C++

- La bibliothèque standard C++ fournit de nombreuses autres structures de données pour représenter des *collections* : `array`, `list`, `queue`, `stack`, `set`, `multiset`, ...
- Chacune a ses spécificités en terme de **sémantique**, d'**opérations disponibles** et de **performances**
- On se contentera dans ce cours de `vector`
  On en verra plus en S2 !

- Les chaînes de caractères (`string`) se comportent en gros comme des tableaux de caractères
- Les `vector` de C++ ne sont pas des vecteurs au sens mathématique : pas d'opération d'addition, ...

La boucle *for each* (C++ 2011) ♣

Exemple

tableau-foreach.cpp

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

tableau-foreach.cpp

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

tableau-foreach.cpp

```
for ( int valeur: tableau ) {              // Boucle for each
    cout << valeur << " ";
}
cout << endl;
```

Avantages

- Pas de risque d'erreur de manipulation d'indice !
- Fonctionne avec n'importe quelle collection !

D. Tableaux à deux dimensions

Motivation

Jeu de morpion

Remarque

- On représente un tableau à deux dimensions par un tableau de tableaux : `vector<vector<int>>`
- $t_{i,j}$: `t[i][j]`

À retenir

Un tableau à deux dimension se construit en quatre étapes :

1. *Déclaration du tableau*
2. *Allocation du tableau*
3. *Allocation des sous-tableaux*
4. *Initialisation*

Tableaux à deux dimensions : exemple (grille de morpion)

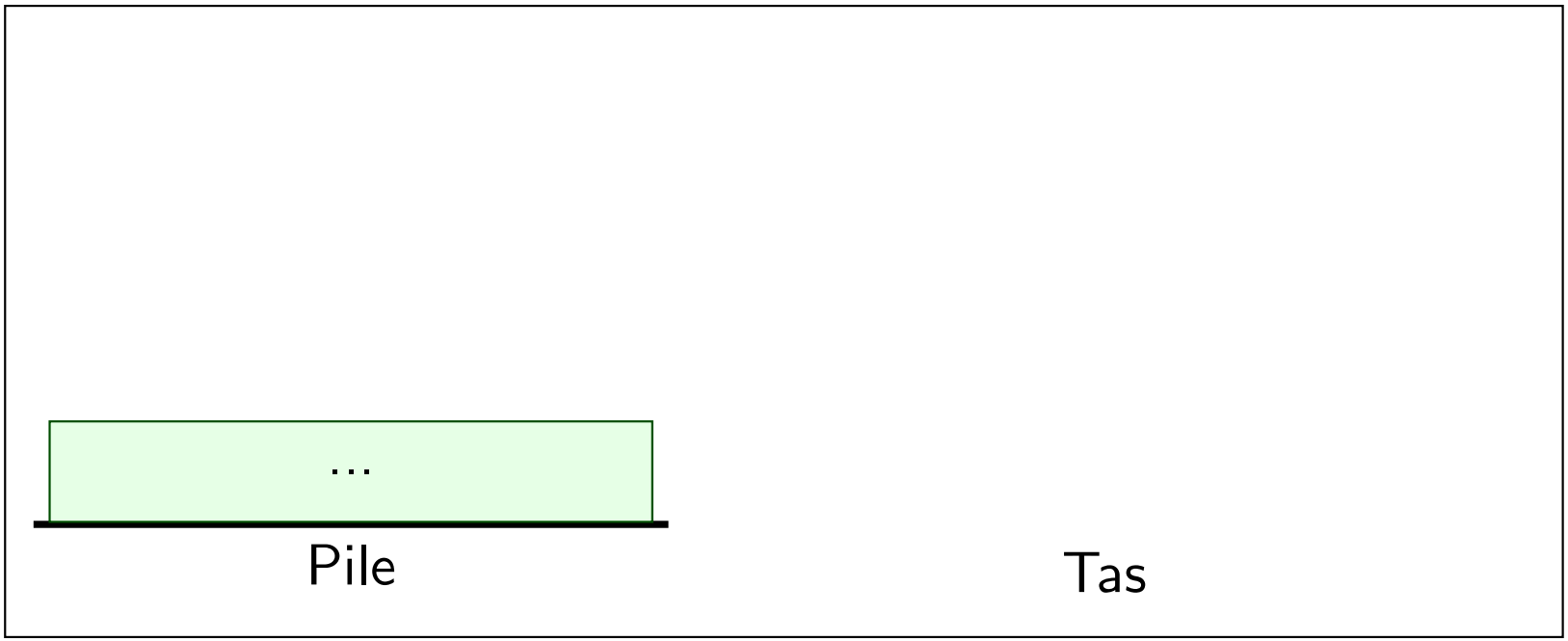
tableaux2D.cpp

```
// Déclaration
vector<vector<int>> t;

// Allocation
t = vector<vector<int>>(3);

// Allocation des sous-tableaux
for ( int i = 0; i < t.size(); i++ )
    t[i] = vector<int>(3);

// Initialization
t[0][0] = 1; t[0][1] = 2; t[0][2] = 0;
t[1][0] = 1; t[1][1] = 1; t[1][2] = 1;
t[2][0] = 0; t[2][1] = 2; t[2][2] = 2;
```

Tableaux à deux dimensions : exemple (grille de morpion)

Étapes de la construction en mémoire

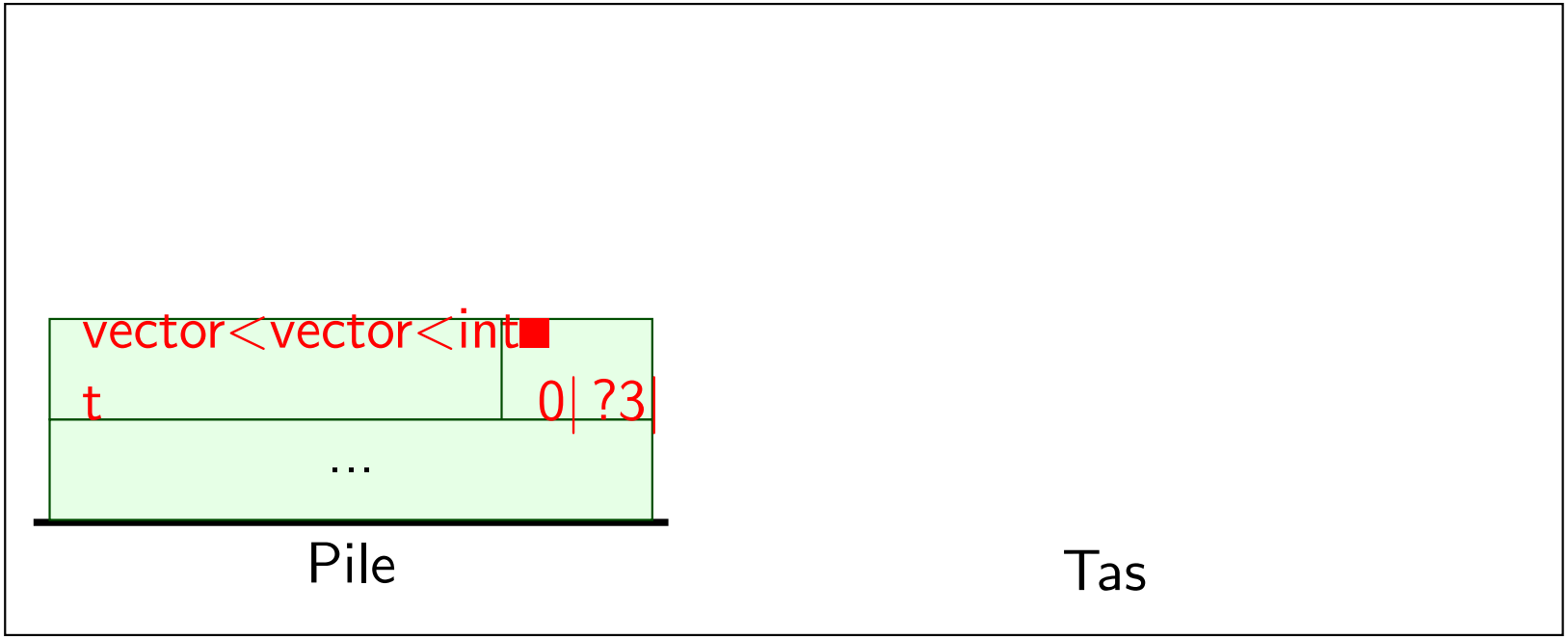
1. Déclaration du tableau
2. Allocation du tableau
3. Allocation des sous-tableaux
4. Initialisation



Tableaux à deux dimensions : exemple (grille de morpion)

Étapes de la construction en mémoire

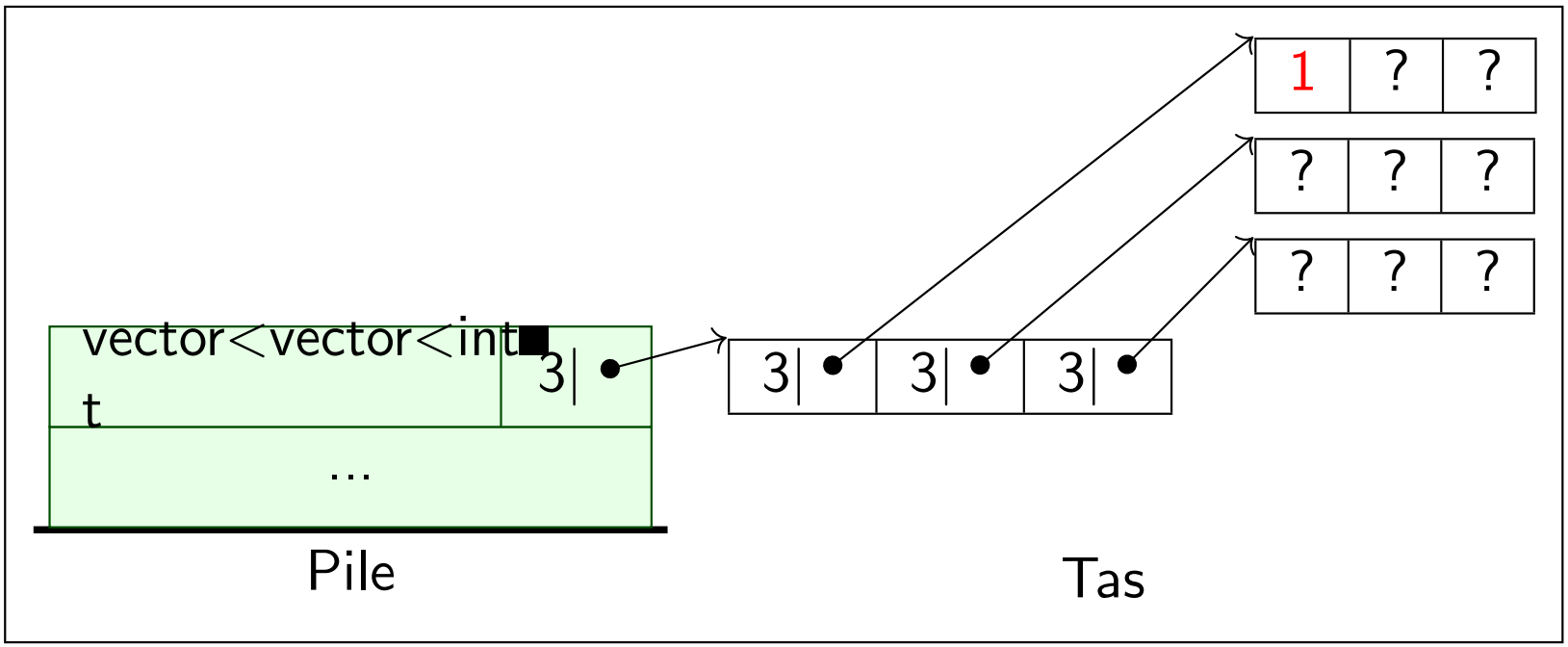
1. Déclaration du tableau
2. Allocation du tableau
3. Allocation des sous-tableaux
4. Initialisation



Tableaux à deux dimensions : exemple (grille de morpion)

Étapes de la construction en mémoire

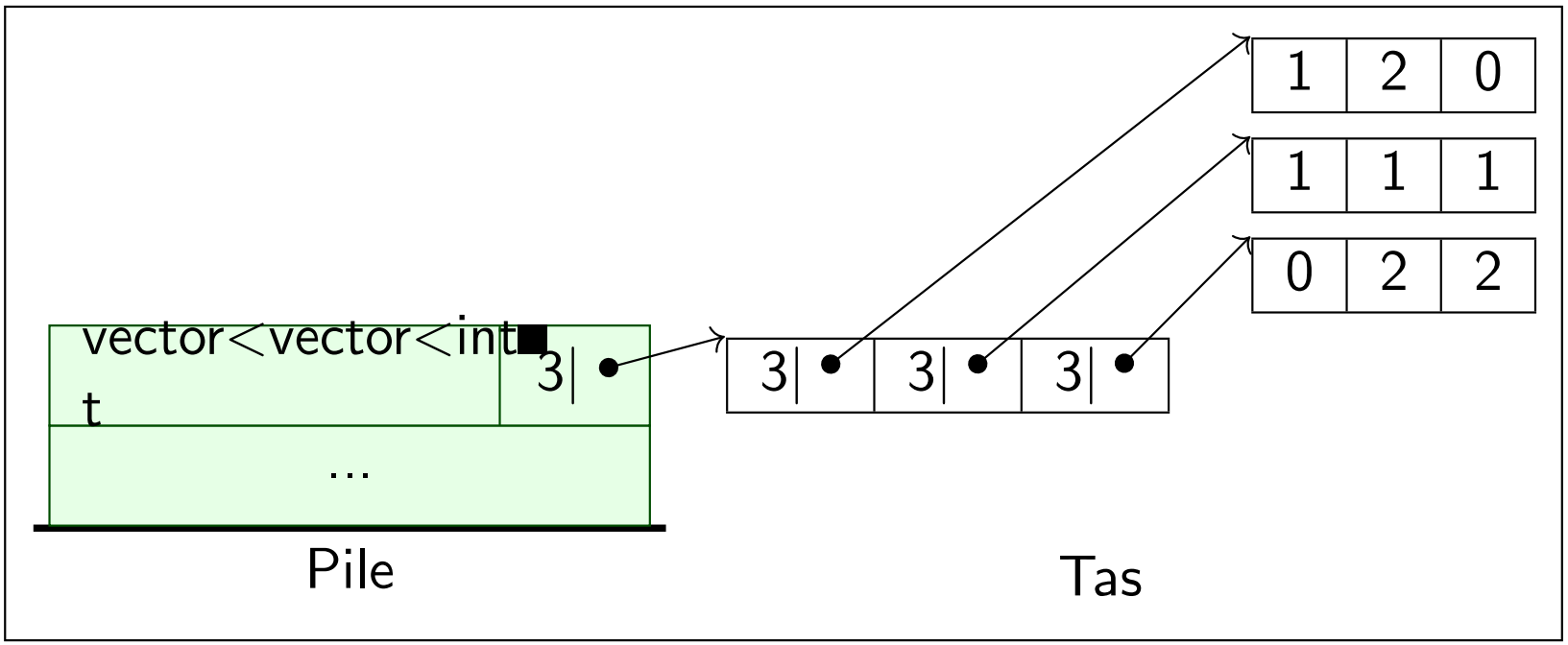
1. Déclaration du tableau
2. Allocation du tableau
3. Allocation des sous-tableaux
4. Initialisation



Tableaux à deux dimensions : exemple (grille de morpion)

Étapes de la construction en mémoire

1. Déclaration du tableau
2. Allocation du tableau
3. Allocation des sous-tableaux
4. Initialisation



Tableaux à deux dimensions : exemple (grille de morpion)

tableaux2D.cpp

```
// Utilisation
for ( int i=0; i < t.size(); i++ ) {
    for ( int j=0; j < t[i].size(); j++ ) {
        if      ( t[i][j] == 1 ) cout << "X ";
        else if ( t[i][j] == 2 ) cout << "O ";
        else                     cout << "  ";
    }
    cout << endl;
}
```

Tableaux à deux dimensions : exemple (grille de morpion)

tableaux2D-raccourci.cpp

```
// Déclaration, allocation, allocation des sous-tableaux
// et initialisation en une seule instruction
vector<vector<int>> t = {
    { 1,2,0 },
    { 1,1,1 },
    { 0,2,2 },
};

// Utilisation
for ( int i = 0; i < t.size(); i++ ) {
    for ( int j = 0; j < t[i].size(); j++ ) {
        if      ( t[i][j] == 1 ) cout << "X ";
        else if ( t[i][j] == 2 ) cout << "O ";
        else                     cout << "  ";
    }
    cout << endl;
}
```

E. Types de base et représentation des données

Rappels

- Premier modèle de mémoire : Une suite contiguë de 0 et de 1 (les *bits*, groupés par *octets*)
- Une variable est un segment nommé de cette mémoire

Question

Comment représenter des informations avec juste des bits ?

E. Types de base et représentation des données

Questions

- Quelle information peut-on représenter sur 1 bit ?
- Sur deux bits ?
- Sur quatre bits ?
- Sur huit bits ?
- Sur n bits ?

À retenir

Le type d'une variable décrit la **structure de donnée** *:*
Comment l'information est **représentée** *par une suite de bits*

E. 1. Les entiers : types `int`, `short`, `long`, ...

Rappel : codage des entiers en base B

Exemple

Le nombre qui s'exprime en base B par les quatre chiffres 1101 vaut :

$$1 \times B^3 + 1 \times B^2 + 0 \times B^1 + 1 \times B^0$$

- En base $B = 10$:
  $1 \times 10^3 + 1 \times 10^2 + 0 \times 10^1 + 1 \times 10^0 = 1000 + 100 + 1 = 1101$
- En base $B = 8$:
  $1 \times 8^3 + 1 \times 8^2 + 0 \times 8^1 + 1 \times 8^0 = 512 + 64 + 1 = 577$
- En base $B = 2$: $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 1 = 13$

Entiers non signés et signés sur 3 bits

Exemple

code binaire	entiers non signés	entiers signés
000	0	-4
001	1	-3
010	2	-2
011	3	-1
100	4	0
101	5	1
110	6	2
111	7	3

Remarques

- Non signé : $7 + 1 = 0$
- Signé : $3 + 1 = -4$
- On calcule modulo $2^3 = 8$!
- Calcul de $-x$ en signé : 1 + complément à 2
- $-(-4) = -4$! $\qquad\qquad$ $\text{abs}(-4) = -4$!

Entiers machine vs entiers mathématiques

Entiers machine

- Représentés sur un mot machine
- Sur une machine à n bits on peut représenter 2^n entiers, soit les entiers compris entre -2^{n-1} et $2^{n-1} - 1$
- Les bornes sont données par `-INT_MAX-1` et `INT_MAX`
- Voir `Exemples/int.cpp`

Attention !

- Les entiers machine sont des **approximations** des entiers mathématiques !
- Pour de grosses valeurs : risque de **débordement**
- Il est aussi possible de calculer avec des « vrais » entiers !

Quelques variantes

- Entiers longs : `long` ; voir `Exemples/long.cpp`
- Entiers courts : `short int` ; voir `Exemples/long.cpp`
- Entiers non signés : `unsigned int`, `unsigned long`, ...

Remarque

Le nombre de bits utilisés (et donc les bornes) peuvent dépendre du compilateur, du système d'exploitation, du processeur, ...

E. 2. Les réels : types `float`, `double`

Motivation

- Représenter des nombres réels ?
- Nombres approchés (chiffres significatifs)
- Grande variations d'ordres de grandeur

Nombres à virgule flottante

- Représentation par *mantisse* et *exposant* : *3.423420*e+*05*
- Voir : `Exemples/float.cpp` et `Exemples/double.cpp`.
- Un certain nombre de bits pour la mantisse
- Les bits restant pour l'exposant
- Les détails de la représentation varient suivant les langages de programmation, les machines et les normes utilisées
- Normes IEEE très précises sur les règles d'arrondis

E. 3. Les caractères : type `char`

- Permettent de stocker un seul caractère :
  - Une lettre de l'alphabet (sans accent) : 'a', ..., 'z', 'A', ..., 'Z'
  - Un chiffre '0', ..., '9'
  - Un caractères du clavier ('@', '+', '/', ' ')
  - Quelques caractères spéciaux
- Notés entre apostrophes (exemple : `'A'`) pour distinguer le caractère `'A'` de la variable `A`
- La table ASCII associe un numéro unique entre 0 et 127 à chaque caractère, ce qui permet d'introduire un ordre
- Et les lettres accentuées ? Les caractères chinois ? ...
  Voir : Unicode, UTF-8

Voir `Exemples/char.cpp`

E. 4. Les chaînes de caractères : type `string`

- Permettent de stocker une suite de caractères :
  un mot, une phrase, ...
- Notées entre guillemets doubles
  Exemple : `"Bonjour"`
- Se comportent essentiellement comme des tableaux de caractères

Opérations

opération	exemple	résultat
concaténation	`"bonjour" + "toto"`	`"bonjourtoto"`
indexation	`"bonjour"[3]`	`'j'`
longueur	`"bonjour".length()`	7

E. 5. Les booléens : type `bool`

Notes

Les variables booléennes ne peuvent prendre que deux valeurs :

- vrai (mot clé `true`)
- faux (mot clé `false`)

Les opérations possibles sur les booléens sont :

- la négation (opération unaire, mot clé `not`)
- la conjonction (opération binaire, mot clé `and`)
- la disjonction (opération binaire, mot clé `or`)
- ...

Expressions booléennes : encadrements

Attention !

Les encadrements ne peuvent pas être écrits directement en C++
Ils doivent être réalisés à l'aide de deux comparaisons connectées par l'opérateur `and`

Exemple

L'encadrement mathématique :

$$0 \leq x \leq 15$$

se traduit en C++ par l'expression booléenne :

```
(0 <= x) and (x <= 15)
```

Évaluation paresseuse des expressions booléennes

Exemple

Quelle est la valeur des expressions suivantes :

- `false and ( 3*x + 1 >= 2 or 1/(1+x) < 42 )`
- `true or ( 3*x + 1 >= 2 or 1/(1+x) < 42 )`

Deux possibilités :

- l'**évaluation complète** : évaluer tous les opérandes des expressions booléennes
- l'**évaluation paresseuse** : stopper l'évaluation dès que l'on peut :
  - Pour une conjonction `a and b` on peut s'arrêter si `a` est faux
  - Pour une disjonction `a or b` on peut s'arrêter si `a` est vrai

Résumé

Collections

- Un modèle de mémoire raffiné avec pile et tas
- L'allocation des tableaux, sur le tas
- Les tableaux à deux dimensions
  Construction en quatre étapes !
- D'autres collections

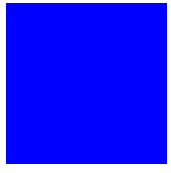
Représentation des données en mémoire

- Types de base : int, long, float, double, char, bool
- Types composites : string, vector, struct
- À partir de ceux-ci, on peut représenter tout type d'information :
  Texte, images, sons, ...
- Le *type* d'une variable décrit la **structure de donnée** :
  Comment l'information est **représentée** par une suite de bits

Sixième partie 6

Débogage

Comment vous sentez-vous en ce début de cours ?

	Curieux, heureux, joyeux, ...
	Groumph, Rostugnududju, ...
	Inquiet
	Fatigué

Résumé des épisodes précédents . . .

Pour le moment nous avons vu les concepts suivants :

- Lecture, écriture
- Instructions conditionnelles et itératives
- Fonctions
- Variables, tableaux, collections

Pourquoi aller plus loin ?

Passage à l'échelle !

Écrire des programmes corrects

A. Corriger les erreurs : le débogage

Exemple

debogage.cpp

```
/** 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;
}
```

A. 1. Débogage, selon le type d'erreur

Erreurs de syntaxe

Détectées par 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

- Segmentation fault !
- Exceptions : division par zéro, ...

Débogage

- Analyser l'état du programme juste avant l'erreur
- En Python, Java, ... : regarder la pile d'appel
- Utilisation du débogeur !

Erreurs sémantiques

- Le programme s'exécute « normalement »
  mais le résultat est incorrect
- Le programme ne fait pas ce que le programmeur souhaitait
- 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 ...

Stratégie : le débogage pas à pas

Un outil essentiel : le débogueur

- Observation du programme en cours de fonctionnement
- Exécution pas à pas :
  - En passant à la ligne suivante (next)
  - En rentrant dans les sous fonctions (step)
- Points d'arrêts (conditionnels)
- Analyse de la pile d'exécution
- Analyse de l'état de la mémoire

Débogueur `gdb` et `Code::Blocks`

- En arrière plan `gdb` : GNU DeBugger
- `Code::Blocks` rend l'utilisation du débogueur facile
- Il n'est disponible que si l'on a créé un projet !

Utilisation du débogueur pas à pas : résumé

- Compiler avec l'option `-g` :

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

- Lancer le débogueur avec :

```
gdb --tui monprogramme
```

- Commandes du débogueur (raccourcis : `s`, `n`, `p expr`, `q`, ...)

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

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 incorrect
   En 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 ! ! !

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

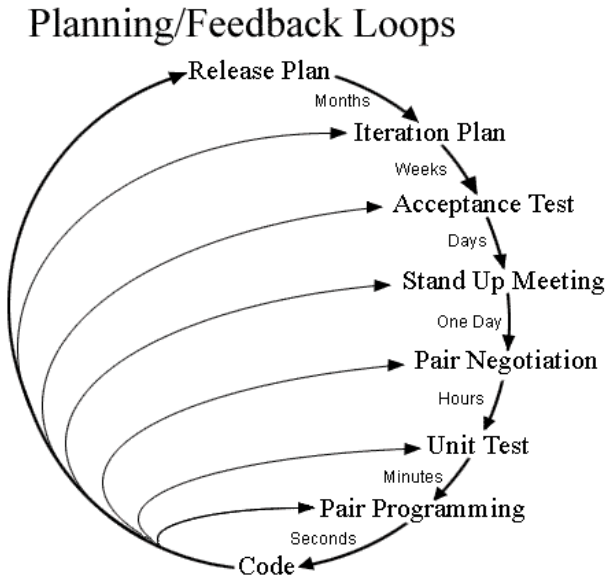
B. Tests : pour aller plus loin ♣

Approche qualité traditionnelle

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

Méthodes agiles ♣

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 !

B.1. Les objectifs et types de tests ♣

- Mesurer la qualité du code
- Mesurer les progrès
- Formaliser (et anticiper !) les demandes du client
  Tests fonctionnels
  Cahier de recette
  `http://fr.wikipedia.org/wiki/Recette_%28informatique%29`
- Garantir la robustesse d'une brique avant d'empiler dessus
  Tests unitaires
- Anticiper la mise en commun de plusieurs briques
  Tests d'intégration
- Anticiper la mise en production
  Tests de charge
- Alerte immédiate si l'on introduit un bogue
  Tests de non régression

B. 2. Il faut aller plus loin ! ♣

- Article « infecté par les tests »
  http://junit.sourceforge.net/doc/testinfected/testing.htm
- http://fr.wikibooks.org/wiki/Introduction_au_test_logiciel

- Tests unitaires en Java :
  http://junit.org/
- Tests unitaires en C++ :
  http://cppunit.sourceforge.net/doc/cvs/cppunit_cookbook.html

Résumé

Stratégies de déboggage

- Réduire le problème
- Dévelopement incrémental :
  ne jamais trop s'éloigner d'une version qui marche
- Débogueur

Tests

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

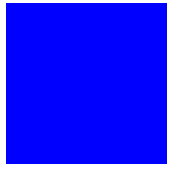
Tests automatique

- `ASSERT( f( ... ) == ... );`
- Pour être efficace

Septième partie 7

Fichiers, flux, exceptions

Comment vous sentez-vous en ce début de cours ?

	Curieux, heureux, joyeux, ...
	Grr, Rostugnududju, ...
	Inquiet
	Flemme

A. Bilan du partiel

- Sujet et corrigé en ligne
- Correction : mardi après-midi, vendredi après-midi
- Notes : à la rentrée, ou plus tôt si moyen sur e-campus
- Minimum : 0, Maximum : ?, Moyenne : ?, écart type : ?
- < 10 ? revoir votre méthode de travail

- Consultation des copies : Jeudi 7 novembre, 12h30-13h45, amphi H2

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 ?

Passage à l'échelle !

Données persistantes

Étude de cas : afficher un annuaire

annuaire.ipynb

```
In [1]: #include <iostream>
        #include <vector>
        using namespace std;

In [2]: void afficheAnnuaire(vector<string> noms, vector<string> telephones)
            for ( int i = 0; i < noms.size(); i++ )
                cout << noms[i] << ": " << telephones[i]  << endl;
        }

In [3]: vector<string> noms =
            { "Jean-Claude", "Alban", "Tibo", "Célestin" };
        vector<string> telephones =
            { "0645235432", "0734534534", "+1150343234", "0634534534"};

In [4]: afficheAnnuaire(noms, telephones)

Jean-Claude: 0645235432
Alban: 0734534534
Tibo: +1150343234
Célestin: 0634534534
```

Problèmes

- Séparation programme / données
- persistance des données

Afficher un annuaire : ce que l'on voudrait

- Un fichier :

annuaire.txt

```
Jean-Claude 0645235432
Alban       0734534534
Tibo        +1150343234
Célestin    0634534534
```

- Un programme :

Comment l'écrire ?

Qu'est-ce qu'un fichier

Définition

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.

Comme la mémoire, mais en persistant !

Format du fichier : comment l'information est-elle encodée ?

Écriture dans un fichier

fichier-ecriture.ipynb

```
In [1]: #include <fstream>
        using namespace std;

In [2]: ofstream  fichier;                          // Déclaration

In [3]: fichier.open("bla.txt");                    // Ouverture

In [4]: fichier << "Noel " << 42 << endl;           // Écriture;

In [5]: fichier.close();                            // Fermeture
```

Quatre étapes :

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

Lecture depuis un fichier

fichier-lecture.ipynb

```
In [1]: #include <fstream>
        using namespace std;

In [2]: ifstream fichier;           // Déclaration

In [3]: fichier.open("bla.txt");    // Ouverture du fichier

In [4]: string s;
        fichier >> s;               // Lecture
        s

Out[4]: "Noel"
        type: std::__cxx11::basic_string<char, std::char_traits<char>, std::

In [5]: int i;
        fichier >> i;
        i

Out[5]: 42
        type: int

In [6]: fichier.close();            // Fermeture du fichier
```

Quatre étapes :

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

Exemple : Afficher un annuaire contenu dans un fichier

annuaire-fichier.ipynb

```
In [1]: #include <iostream>
        #include <fstream>
        using namespace std;

In [2]: ifstream annuaire;
        annuaire.open("annuaire.txt");

In [3]: string nom;
        string tel;

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

Jean-Claude: 0645235432
Alban: 0734534534
Tibo: +1150343234
Célestin: 0634534534
```

Problème

Comment savoir le nombre de lignes ?

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, on peut en savoir plus

Exemple : Afficher un annuaire contenu dans un fichier

annuaire-fichier-while.ipynb

```
In [1]: #include <iostream>
        #include <fstream>
        using namespace std;

In [2]: ifstream annuaire;
        annuaire.open("annuaire.txt");

In [3]: string nom;
        string tel;

In [4]: while ( annuaire >> nom  and  annuaire >> tel ) {
            cout << nom << ": " << tel << endl;
        }

Jean-Claude: 0645235432
Alban: 0734534534
Tibo: +1150343234
Célestin: 0634534534
```

Bonne pratique : vérifier l'état d'un fichier

fichier-ouverture-etat.ipynb

```
In [1]: #include <iostream>
        #include <fstream>
        using namespace std;

In [2]: ifstream fichier;
        fichier.open("annuaire.txt"); // Un fichier existant

In [3]: if ( not fichier ) {
            cout << "Erreur à l'ouverture" << endl;
        }

In [4]: fichier.close();

In [5]: fichier.open("oups.txt");    // Un fichier non existant

In [6]: if ( not fichier ) {
            cout << "Erreur à l'ouverture" << endl;
        }

Erreur à l'ouverture
```

Remarque

Pour mieux signaler l'erreur, on peut utiliser une **exception**. Voir plus loin.

Lecture depuis le clavier

cin.cpp

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

int main () {
    string nom;
    cout << "Comment t'appelles-tu?" << endl;
    cin >> nom;
    cout << "Bonjour " << nom << " !" << endl;
}
```

Syntaxe

Lecture d'une variable :

```
cin >> variable;
```

Sémantique

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

Lecture depuis une chaîne de caractères

istringstream.ipynb

```
In [1]: #include <sstream>
        using namespace std;

In [2]: string s = "1 2 4 8 16";

In [3]: istringstream flux(s);

In [4]: int i, j;

In [5]: flux >> i;

In [6]: i

Out[6]: 1
        type: int

In [7]: flux >> j;

In [8]: j

Out[8]: 2
        type: int

In [9]: i + j

Out[9]: 3
        type: int

In [10]: string bla;
```

Lecture depuis une chaîne de caractère

Syntaxe

Lecture d'une variable :

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

Sémantique

Lit les variables successivement depuis la chaîne de caractères `s`

Écriture dans une chaîne de caractères

ostringstream.ipynb

In [1]:
```
#include <iostream>
#include <sstream>
using namespace std;
```

In [2]:
```
ostringstream flux;
flux << 3.53 << " coucou " << 1 << endl;
flux << 42 << endl;
```

In [3]:
```
string s = flux.str();
```

In [4]:
```
s
```

Out[4]:
```
"3.53 coucou 1
42
"
type: std::__cxx11::basic_string<char, std::char_traits<char>, std::
```

In [ ]:

Notion de flux

Remarque

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

Définition (Flux de données)

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

Exemples de flux sortants de données

- `cout` : **sortie standard** du programme
  Typiquement : écran
  ♣ Avec tampon

- `cerr` : **sortie d'erreur** du programme
  ♣ Sans tampon

- fichiers (`ofstream`)

- chaînes de caractères (`ostringstream`)

- connexion avec un autre programme ...

Exemples de flux entrants de données

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

D. Digression : traitement des erreurs et exceptions

Exemple (gestion d'entrées invalides)

exception.cpp

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

int main() {
    cout << factorielle(-1) << endl;
}
```

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 boss
  c'est-à-dire la fonction appelante
- On dit qu'on **signale une exception**
- La situation est décrite par e
  e est un objet quelconque ; par exemple une **exception standard**
- Si mon boss ne sait pas gérer, il prévient son boss
- ...
- Si personne ne sait gérer, **le programme s'arrête**

Quelques exceptions standard

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

exception-standard.cpp

```
#include <stdexcept>

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

int main() {
    cout << factorielle(-1) << endl;
}
```

Exemple de gestion d'exception

exception-gestion.cpp

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

int main() {
    int n;
    cin >> n;
    try {
        cout << factorielle(n) << endl;
    } catch (invalid_argument & e) {
        cout << "Valeur de n invalide" << endl;
    }
}
```

♣ Gestion des exceptions

Syntaxe

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

Sémantique

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

Résumé

Fichiers

- 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`

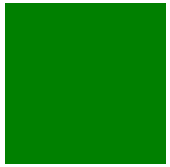
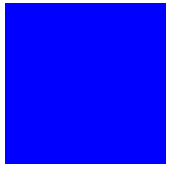
Exceptions

- Comment signaler une erreur dans un programme
- Comment traiter de telles erreurs

Huitième partie 8

Modularité, compilation séparée

Comment vous sentez-vous en ce début de cours ?

	Curieuse, heureux, joyeux, ...
	Grumble, grumble, ...
	Inquiète
	Gros baillement

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
- Entrées-sorties, fichiers
- Complexité

Pourquoi aller plus loin ?

Passage à l'échelle !

Maintenance de « gros » programmes

A. Digression : programmes et compilation

Retour sur 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 ?

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

A. Digression : programmes et compilation

C'est quoi une instruction ?

- « 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 ?

Quelles sont les instructions compréhensibles par un ordinateur ?

Assembleur

Exercice : exécuter ce fragment d'**assembleur**

puissance-quatre-extrait.s

```
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)
```

Indications

- `%eax`, `%edx` : deux *registres* (cases mémoire du processeur)
- `-0x14(%rbp)`, `...`, `-0x1c(%rbp)` : autres cases mémoire
- Initialiser le contenu de la case `%-0x1c(%rbp)` à 3
- `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

Cycle de vie d'un programme : motivation

Ce que je veux :

« Calculer la puissance 4^e d'un nombre »

Ce que l'ordinateur sait faire :

puissance-quatre-extrait.s

```
...
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)
...
```

Cela ne va pas se faire tout seul ...

Cycle de vie d'un programme : formalisation et algorithme

Problème

« Calculer la puissance 4^e 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 $x^4 = x * x * x * x = (x^2)^2$

Algorithme

- calculer $x * x$
- prendre le résultat et faire de même

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

Notes

- Doit donc toujours se terminer !
- Conçu pour communiquer entre humains
- 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 !

Cycle de vie d'un programme (4)

puissance-quatre.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;
}
```

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 directement traduisible en langage machine

Langage de programmation

Un programme doit être *transformé* pour être compris par la machine

Question

Comment faire cette transformation ? À la main ?

Transformer en binaire : les interpréteurs

Exemple : interpréteur C++ dans Jupyter

puissance-quatre.ipynb

```
In [1]: int x;

In [2]: x = 5

Out[2]: 5
        type: int

In [3]: int xCarre = x * x;
        int xPuissance4 = xCarre * xCarre;

In [4]: xPuissance4

Out[4]: 625
        type: int
```

Chaîne de production

Utilisateur $\Longrightarrow$ Instruction source $\Longrightarrow$ Interpréteur $\Longrightarrow$ Instruction machine $\Longrightarrow$ Machine

Exemples (quelques langages interprétés)

Basic, LISP, Perl, Python, C++, ...

Transformer en binaire : 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`

Chaîne de production

Utilisateur $\Longrightarrow$ Programme source $\Longrightarrow$ Compilateur $\Longrightarrow$ Programme objet $\Longrightarrow$ Machine

Exemples (Exemples de langages compilés)

Pascal, C, C++, ADA, FORTRAN, Java, ...

Cycle de vie d'un programme : exécution, mise au point

Exécuter le programme

- Autant de fois que l'on veut !

Tester que le programme fonctionne

- Cas particuliers ! ! !

Améliorer le programme

- Correction d'erreurs
- Optimisation du programme (rapidité, consommation mémoire)
- Optimisation de l'algorithme
- Amélioration du programme (lisibilité, généralisation)

Cycle de vie d'un programme : résumé

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)

B. Compilation séparée

compilation-separee-avant/programme1.cpp

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

int monMax(int a, int b) {
    if ( a >= b )
        return a;
    else
        return b;
}

int main() {
    cout << monMax(1, 3) << endl;
    return 0;
}
```

compilation-separee-avant/programme2.cpp

```
#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;
}
```

Problème

Partager une fonction `monMax` entre ces deux programmes ?

Exemple : la bibliothèque max

compilation-separee/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);
```

compilation-separee/max.cpp

```
#include "max.h"

int monMax(int a, int b) {
    if ( a >= b )
        return a;
    else
        return b;
}
```

Deux programmes utilisant cette bibliothèque

compilation-separee/programme1.cpp

```
#include <iostream>
using namespace std;
#include "max.h"

int main() {
    cout << monMax(1, 3) << endl;
    return 0;
}
```

compilation-separee/programme2.cpp

```
#include <iostream>
using namespace std;
#include "max.h"

int main() {
    cout << "Entrez a et b:" << endl;
    int a, b;
    cin >> a >> b;
    cout << "Le maximum est: "
         << monMax(a, b) << endl;
    return 0;
}
```

Exemple : Les tests de la bibliothèque max

compilation-separee/maxTest.cpp

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

#include "max.h"

/** Infrastructure minimale de test **/
#define ASSERT(test) if (!(test)) cout << "Test failed in file " << __FILE__ <<

void monMaxTest() {
    ASSERT( monMax(2,3) == 3 );
    ASSERT( monMax(5,2) == 5 );
    ASSERT( monMax(1,1) == 1 );
}

int main() {
    monMaxTest();
}
```

Déclaration de fonctions

Syntaxe

compilation-separee/max.h

```
int monMax(int a, int b);
```

Sémantique

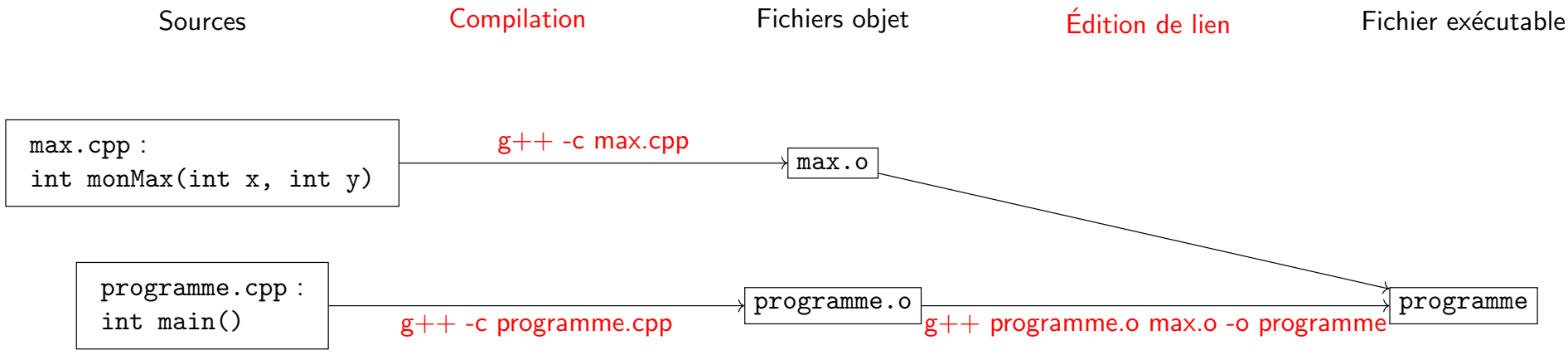
- Le programme **définit** quelque part une fonction `monMax` avec cette **signature** : type des paramètres et type du résultat
- Cette définition n'est pas forcément dans le même fichier
- Si cette définition n'existe pas ou n'est pas unique, une erreur est déclenchée par le compilateur
- Cette erreur est déclenchée au moment où l'on combine les différents fichiers (édition de liens ; voir plus loin)

♣ application

Deux fonctions qui s'appellent réciproquement

Compilation séparée

- 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**



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 contenant la **déclaration** des fonctions **définies** dans le fichier .cpp correspondant

Exemple (Fichier d'entête `max.h`)

compilation-separee/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`, ...

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
```

Digression : surcharge de fonctions ♣

Exemple (la fonction `monMax` : `Exemples/max-surcharge.cpp`)

- Pour les entiers, les réels, les chaînes de caractères, ...

max-surcharge.cpp

```
int monMax(int a, int b) {
```

max-surcharge.cpp

```
string monMax(string a, string b) {
```

- À deux ou trois arguments (et plus ?)

max-surcharge.cpp

```
int monMax(int a, int b, int c) {
```

Surcharge

- 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 (duplication)
- Correctif : les **templates** pour écrire du code **générique**
  La fonction `monMax` suivante est valide pour tout type sur lequel on peut faire des comparaisons :

max-surcharge-templates.cpp

```
template<class T>
T monMax(T a, T b) {
    if ( a >= b )
        return a;
    else
        return b;
}
```

- Ce sera pour un autre cours !

Digression : espaces de noms ♣

Problème

Conflit entre bibliothèques définissant des fonctions avec le même nom et la même signature !

Solution

Isoler chaque bibliothèque dans un **espace de noms**

Exemple

La bibliothèque standard C++ utilise l'espace de nom `std` :

bonjour-std.cpp

```
#include <iostream>
int main() {
    std::cout << "Bonjour !" << std::endl;
    return 0;
}
```

- `using namespace std ;` : raccourcis pour std
- Vous verrez plus tard comment créer vos propres espaces de noms

Digression : débogage, prévention

Développement incrémental

- Toujours être « proche de quelque chose qui marche »
- Gestion de version (`git`, `mercurial`, ...)

Spécifications et tests

- Définir précisément la sémantique des fonctions :
  qu'est-ce qu'elles doivent faire
- Tester que la sémantique est respectée sur des exemples

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**

Résumé de la séance

Programmes et compilation

- 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