ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Photo Martin Klimas

Programmation

Introduction
Principes de programmations

ME 2e semestre

Rev. 2015.1

Christophe Salzmann

Plan

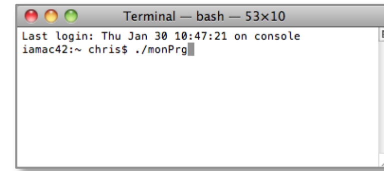
- Rappels 1er semestre
- Principes et paradigmes de programmation
- Variables et expressions
- Mon premier programme en C/C++

Computers follow your orders, not your intentions!

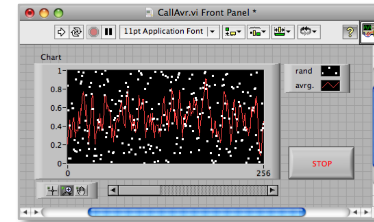
http://melbel.hubpages.com/hub/Funny-Email-Signatures

Types de programmes

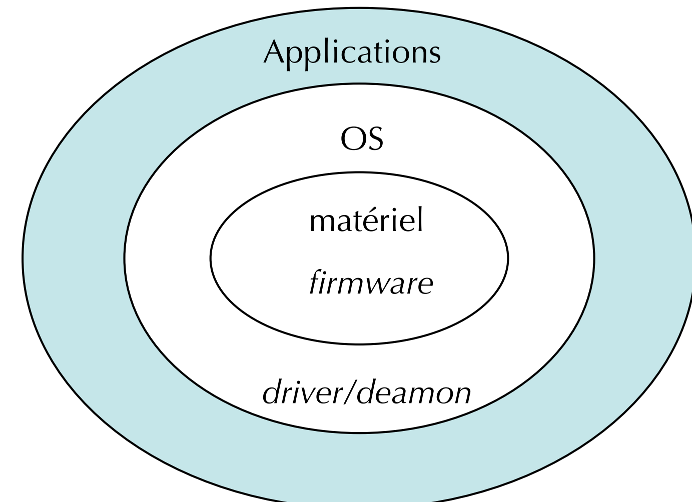
- Applications sans interface utilisateur graphique
  - ligne de commande, ex. `ls`, `dir`



- Applications avec une interface utilisateur graphique
  - ex. Word, browser



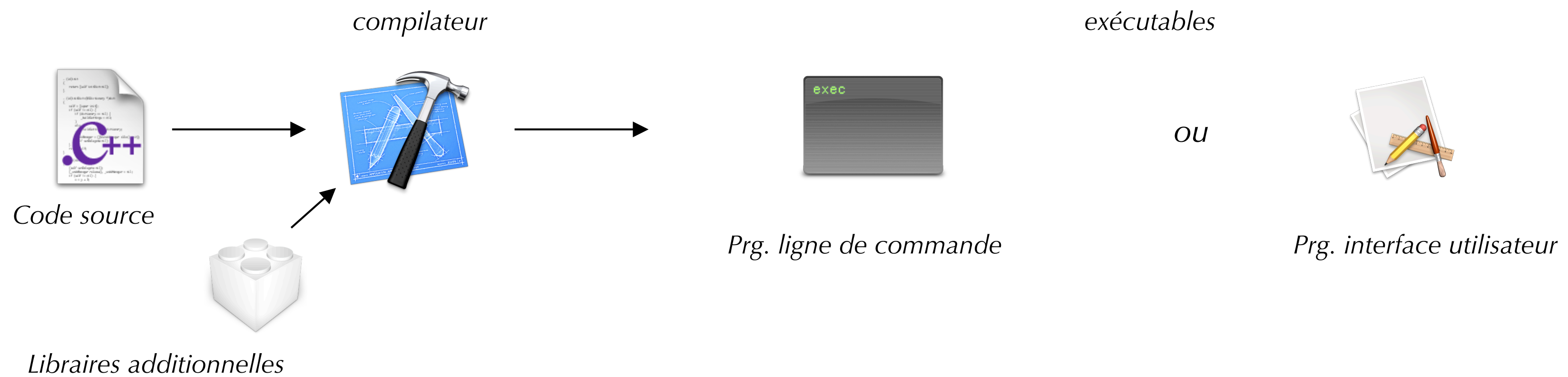
- Daemon (ex. gestionnaire d'impression)
- Driver (ex. accès à la carte réseau)
- OS (ensemble de *drivers* et autres services, esp. Gestionnaire de fichiers (Finder, Explorer,)
- Firmware (micro prg matériel)



Etapes de création d'un programme

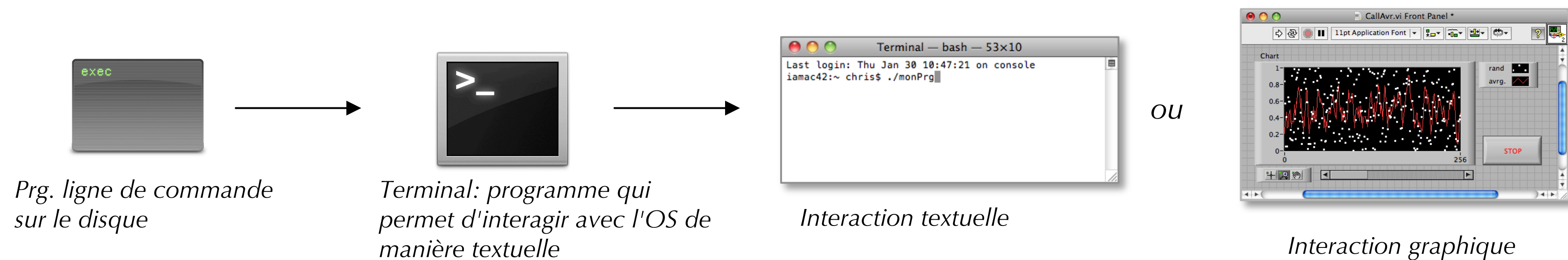
Création

Code source – Compilation – Création de l'exécutable (fichier sur votre disque)

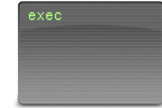


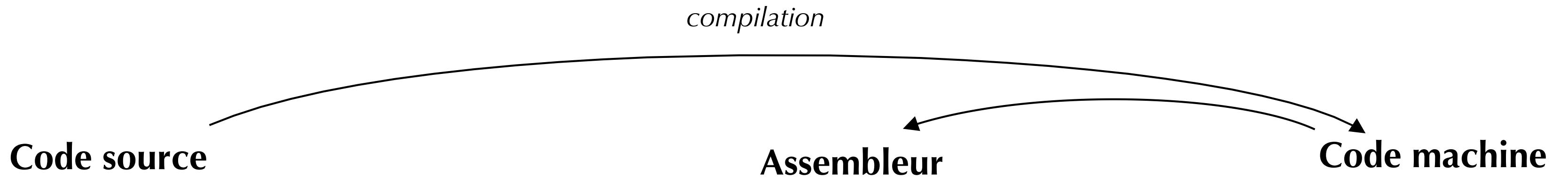
Exécution

Chargement de l'exécutable – Exécution – Interaction/entrée-sortie



Code source -> code machine






```
1
2   int main () {
3       int x,y,z;
4
5       x = 1 ;
6       y = 2 ;
7       z = x + y;
8
9       return 0;
10  }
11
```

asm main 0x0000000100000f10:18:1

```
1   0x0000000100000f10  <+0000>  nop
2   0x0000000100000f11  <+0001>  nop
3   0x0000000100000f12  <+0002>  nop
4   0x0000000100000f13  <+0003>  nop
5   0x0000000100000f14  <+0004>  nop
6   0x0000000100000f15  <+0005>  nop
7   0x0000000100000f16  <+0006>  push   %rbp
8   0x0000000100000f17  <+0007>  mov    %rsp,%rbp
9   0x0000000100000f1a  <+0010>  mov    %edi,-0x14(%rbp)
10  0x0000000100000f1d  <+0013>  mov    %rsi,-0x20(%rbp)
11  0x0000000100000f21  <+0017>  movl   $0x1,-0x4(%rbp)
12  0x0000000100000f28  <+0024>  movl   $0x2,-0x8(%rbp)
13  0x0000000100000f2f  <+0031>  mov    -0x8(%rbp),%eax
14  0x0000000100000f32  <+0034>  add    -0x4(%rbp),%eax
15  0x0000000100000f35  <+0037>  mov    %eax,-0xc(%rbp)
16  0x0000000100000f38  <+0040>  mov    $0x0,%eax
17  0x0000000100000f3d  <+0045>  leaveq
18  0x0000000100000f3e  <+0046>  retq

    0x00007fff5fbff7d4           00000000
    0x00007fff5fbff7d8           02000000
    0x00007fff5fbff7dc           01000000
```

```
90 90
90 90
90 90
55 48
89 e5
c7 45
fc 01
00 00
00 c7
45 f8
02 00
00 00
8b 45
f8 03
45 fc
89 45
f4 b8
00 00
```

Contenu mémoire

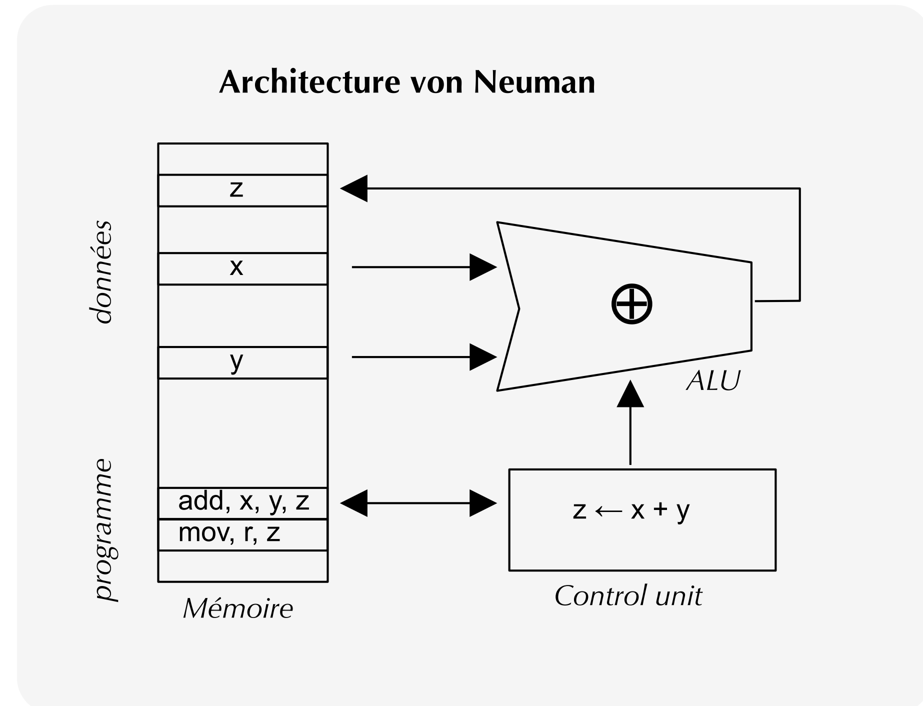
lisible — *"lisible"* — *illisible*

Exécution d'un programme

- **Langage machine**: suite d'instructions binaires compréhensibles par une machine réel ou virtuelle, suit l'architecture von Neuman

- **Assembleur**: langage machine "lisible" par l'homme

- **Code source**: langage haut niveau lisible par l'homme, ex. C/C++, Java, matlab, etc.



La compréhension de ce qui se passe au niveau matériel aide à la compréhension du fonctionnement du programme

Paradigmes de programmation

- Différents paradigmes de programmations:
  - **impérative,** fonctionnel, logique, **dataflow**, etc.

- Programmation **impérative**:
  - décrit *comment faire*, reflète l'architecture von Newman
  - programmation structurée et/ou encapsulation dans des objets, mêmes concepts de base pour tous les langages de cette catégorie:

> **l'assignation**
>
> **le branchement conditionnel**
>
> ~~le branchement sans condition~~ -> programmation structurée
>
> **le bouclage**

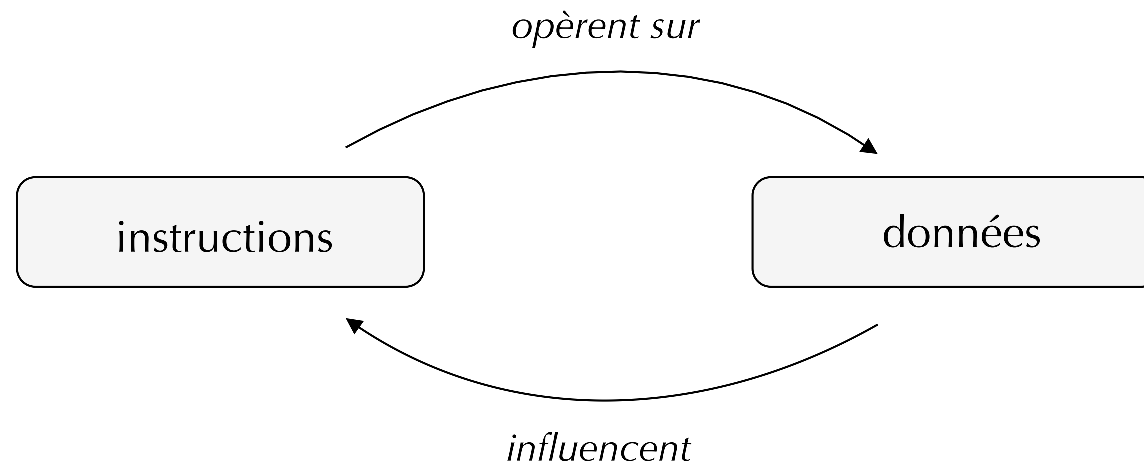
- Différents langages pour des raisons conceptuelles, historiques, commercial, etc.

Programme = Algorithme + Données

Niklaus. Wirth 1985

Programme:

- suite d'instructions (algorithme) **agissant** sur des données
- les données **influences** l'ordre d'exécution des instructions



Difficultés:

- comment formaliser une idée en un algorithme
- comment passer d'une idée (cahier des charges) à une suite d'instructions

  -> par ex. stratégie "Diviser pour Régner"

Algorithme – analogie culinaire

Adapté de http://sweetrandomscience.blogspot.ch/2014/01/quest-ce-quun-algorithme-explication.html

Une recette est un algorithme qui décrit les étapes pour confectionner un plat. Les étapes principales peuvent être décomposées en sous-tâches. Les résultats de ces différentes sous-tâches doivent être stockés dans des conteneurs avant d'être "assemblés" dans le plat final.

Plat:

- Crêpes

Sous-tâches:

- Vérifier que l'on a les ingrédients et les ustensiles nécessaires
- Faire la pate à crêpes (dans le bol)
- Cuire la pate dans une poêle pour en faire des crêpes, mettre les crêpes dans le plat

Conteneurs:

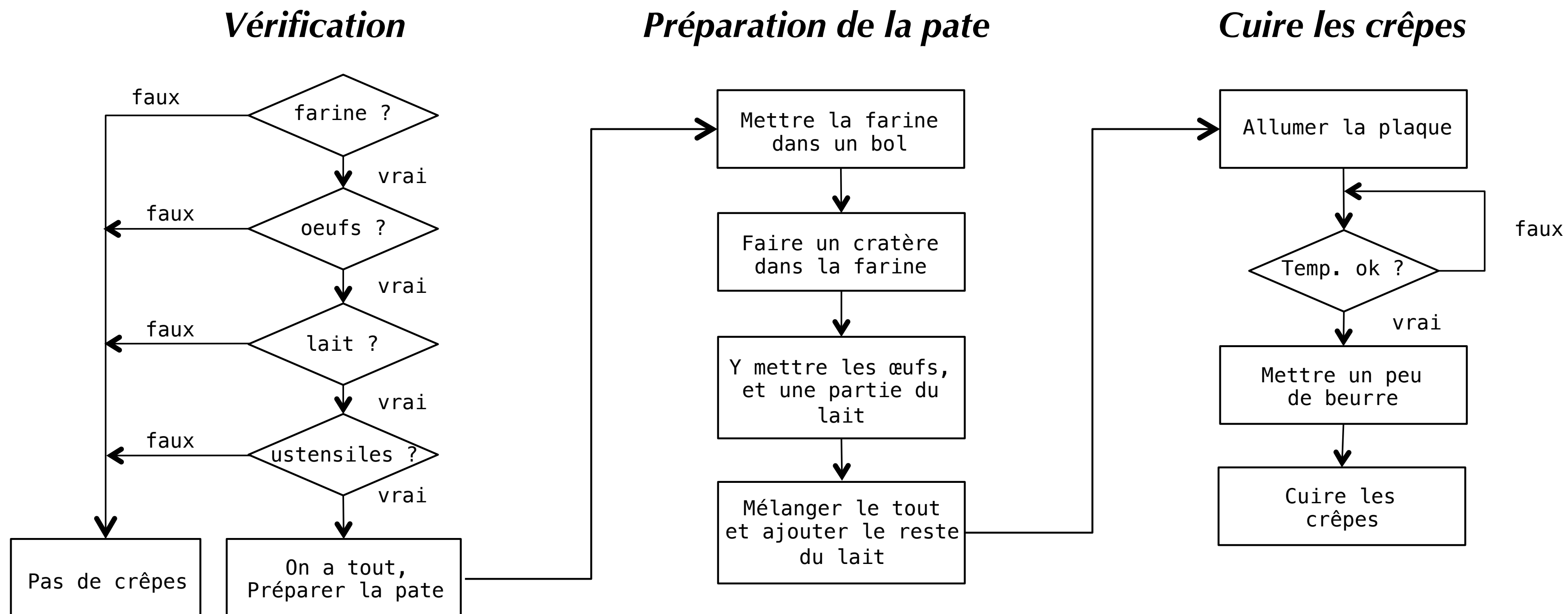
- Bol (vide)
- Poêle (vide)
- Plat (vide)

```
int FaireCrepes(){
  int bol(0);
  int plat(0)

  if (Verification() == OK) {
     FairePate(bol);
     CuirePate(bol,plat);
    }
  else
    afficher("no crêpes :-( ");

  return 0;
}
```

Algorithme – analogie culinaire




```
int Verification (){
  int verifier(pasOK);

  if (farineOK() == OK)
   else if (oeufOK() == OK)
    else if (laitOK() == OK)
     else if (ustensileOK() == OK)
      verifier=OK;
  return verifier;
}
```

```
void FairePate(int& bol){

  mettreFarine(bol);
  faireCratere(bol);
  mettreOeuf(bol);
  melanger(bol);

}
```

```
void CuirePate(int bol, int& plat){
  int poele(0);

  allumerPlaque();
  while(temp <= OK);
  mettreBeurre(poele);
  cuire(bol, poele, plat);

}
```

Algorithme – analogie culinaire

La mise en œuvre de la recette peut être vue comme un programme s'appelant ***Faire des Crêpes***.

Cette analogie permet d'illustrer plusieurs concepts de programmation:

- L'**algorithme** définissant la suite d'instructions est représenté par la recette
- Les **variables** (données) contenant les ingrédients sont représentés par les conteneurs
- La **valeur** et le **type** des variables sont représentés par le contenu des conteneurs
- L'**initialisation** des variables est représentée par le fait que les conteneurs doivent être vides (et propres) au début de la recette
- Les **fonctions** sont représentées par les sous-tâches
- Différentes opérations de base sont représentées:
  - L'**assignation**, ex. lorsque l'on met de ingrédient dans un conteneur
  - Le **test** (*if*), par exemple lors de la vérification des ingrédients
  - la **boucle** (*while*) pour l'attente d'une condition à vrai, par ex. attente sur la température du four

Algorithme -> programme

- Comme pour le langage parlé, un langage de programmation utilise une **syntaxe** (règle de construction pour les instructions) et un **vocabulaire** (mot réservé) bien définie

- Un langage de programmation défini des **traitements** mis en œuvre sur des **données**

- Une **donnée** est stockée quelque part (en mémoire). Cet endroit est décrit au niveau du langage par une **variable**. C'est un objet informatique qui pourra être manipulé par le programme

- Les **traitements** sont associés dans le programme à la notion d'**instructions** et d'**expressions**

Dans la première partie de ce cours nous allons explorer les principes de programmation à l'aide du C++ (l'aspect objet du C++ ne sera pas présentée). Le C et le C++ (de base sans les objets) ont une syntaxe très similaire.

Variables

Une variable *vit* dans la mémoire de votre ordinateur à une **adresse** donnée, elle a un **nom** qui permet de la désigner et un **type** qui indique la taille qu'elle occupe dans la mémoire et comment interpréter la **valeur** contenue à cette adresse.

- **nom (identificateur)**, il doit être le plus explicite possible,

  ex. *maSomme*
- **type:** indique comment interpréter la représentation binaire de la valeur en mémoire et quels traitements elle peut (et ne peut pas) subir.

  ex : *double, int, string,*
- **valeur :** séquence de bits, qui pourra être définie sous forme de *valeur littérale*. Par exemple, si la donnée est un nombre, sa valeur pourra être : 123, -18, 3.1415, 2e-13 (= 2 · 10–13), . . .
- **adresse**: ça position en mémoire
- **durée de vie:** ou portée, i.e. son existence dans un bloc de code

Variables

```
int x(67);

double maSomme(16.5);

string myStr("B");
```

adresse | valeur | type | nom

adresse	valeur		nom
0100			
0101	67	10000011	x
0102			
0103			
0104	16.5	10000011	maSomme
0105		00001000	
0106			
0107	B	10000011	myStr
0108			

mémoire

Variable - nom

Nom ou **identificateur: une** séquence composée de lettres, de chiffres ou du caractère '_' et commençant par une lettre ou par '_'.

Les noms doivent être le plus explicite possible.

Un identificateur ne doit pas correspondre à un mot réservé du langage, sinon le compilateur génère une erreur.

Attention, le C/C++ différencie les **MAJUSCULES** des **minuscules.**

Keywords C

```
auto        else        long        switch
break       enum        register    typedef
case        extern      return      union
char        float       short       unsigned
const       for         signed      void
continue    goto        sizeof      volatile
default     if          static      while
do          int         struct      _Packed
double
```

C++ keywords

```
alignas (since C++11)
alignof (since C++11)
and
and_eq
asm
auto(1)
bitand
bitor
bool
break
case
catch
char
char16_t (since C++11)
char32_t (since C++11)
class
compl
const
constexpr (since C++11)
const_cast
continue
decltype (since C++11)
default(1)
delete(1)
do
double
dynamic_cast
else
enum
explicit
export(1)
extern
false
float
for
friend
goto
if
inline
int
long
mutable
namespace
new
noexcept (since C++11)
not
not_eq
nullptr (since C++11)
operator
or
or_eq
private
protected
public
register
reinterpret_cast
return
short
signed
sizeof
static
static_assert (since C++11)
static_cast
struct
switch
template
this
thread_local (since C++11)
throw
true
try
typedef
typeid
typename
union
unsigned
using(1)
virtual
void
volatile
wchar_t
while
xor
xor_eq
```

Variable - type

Types de base:

`int`	nombre entier:	`1, 12, -156`
`double`	nombre réel#:	`0.00025,  2.3e3` (=2300)
`char`	caractère:	`'a', '\32'`
`bool`	(c++) valeur logique:	`true (1, ≠ 0), false (0)`

La taille en mémoire d'une variable dépends de son type. Il existe un nombre fini de valeurs pour un type de variable. Par exemple, une variable stockée sur 8 bits peut valoir $2^8 = 256$ valeurs.

Pour les nombres réels, un point '.' (et non pas une virgule) sépare la partie entière de la partie décimale.

#Les nombres réels (`double`) sont approximés par l'ordinateur. En effet, il n'est pas possible d'encoder une infinité de valeurs sur un nombre fini de bits.

Variable - déclaration et initialisation

Les variables doivent être déclarées avant d'être utilisées dans le programme.

Il est possible (recommandé) d'initialiser la variable à une valeur par défaut lors de sa déclaration.

Attention, si aucune valeur par défaut n'est spécifié, le contenu précédent de la mémoire sera utilisé (-> valeur aléatoire, non reproductible)

```
Type   Identificateur(valeur initialisation)    (c++)
Type   Identificateur = valeur initialisation   (c/c++)
```

La valeur d'initialisation est une constante (*valeur littérale*) ou expression du type choisi.

Ex.

```
int myIndex(0);
int myIdx(myIndex + 1);  // myIdx vaut 1 (0+1)
double quarter(0.25);
char myAnswer('y');
```

Variable - accès

Par défaut les variables sont modifiables.

Si on ne veut pas qu'une variable soit modifiable il faut la définir comme **constante**

La nature **modifiable** ou **non modifiable** d'une variable doit être défini lors de la déclaration par l'indication du mot réservé **`const`**

La variable ne pourra donc plus être modifiée par le programme, toute tentative de modification produira un message d'erreur lors de la compilation.

Ex.

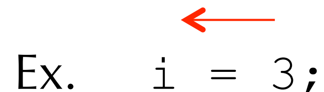
```
const int MaxIteration(2);
const double pi(3.1415);
```

Variable - affectation

L’opération d’affectation (**=**) modifie la valeur d'une variable. La variable doit avoir été déclarée précédemment. En C/C++, la syntaxe d’une affection est :

```
Identificateur = valeur;
```

où `valeur` est une constante ou une expression du **même type** que la variable référencée par `identificateur`.



Ex. `i = 3;`

Attention!

`i = 3;` est une affectation et non pas une vérité mathématique! Ca n'est pas une équation mathématique. Le '=' de C/C++ n’est pas symétrique,

`i = 3;` ☺

`3 = i;` ☹ erreur de compilation, je ne peux pas affecter i à 3

Variable - affectation

L'opération d'affectation est un processus dynamique

Ex. `i = i + 1;`

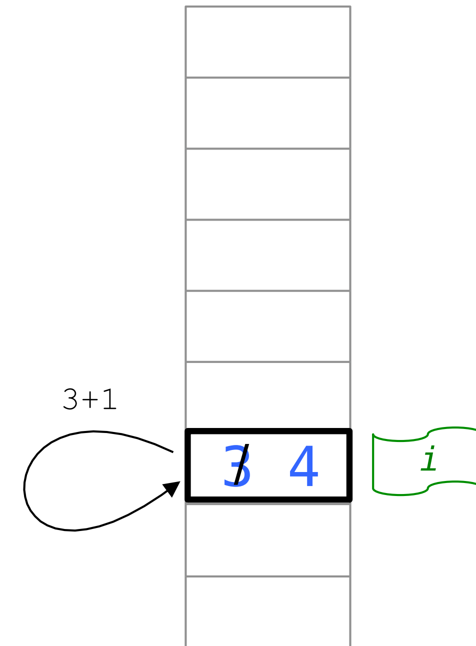
Le compilateur commence par évaluer la partie à droite du signe d'affectation (=), pour cela il va:

- lire le contenu de la variable i
- y ajouter 1

Puis mettre (affecter) le résultat de l'addition dans la variable i

Si la variable `i` valait *3* avant `i = i + 1;`

elle vaudra *4* après celle-ci



Opérateurs et expressions

- Tout langage de programmation fournit des **opérateurs** permettant de manipuler les objets prédéfinis

- Les opérateurs sont associés au type des objets sur lesquels ils peuvent opérer. Ainsi, les opérateurs *arithmétiques* (+, -, *, /, ...) sont définis pour les types numériques (entiers et réels), les opérateurs *logiques* (&&, ||, !, ...) pour les types booléens, etc...

- Les **expressions** sont des séquences combinant des opérateurs et des arguments (variables ou valeurs) suivant la syntaxe du langage

- Exemple d'expression numérique : (2*(13-7)/(1+4))

Opérateurs arithmétiques

+	addition
-	soustraction
*	multiplication
/	division
%	reste division entière (modulo)

Ex.

```
b = (2 + (4 * 3)); // b vaut 14, calculé lors de la compilation
z = (3 * x) / 10;  // z dépend de x, calculé lors de l'exécution
```

En C/C++ il est possible d'abréger des affectations particulières (à éviter au début)

```
x = x + y;    peut aussi s'écrire   x += y;   (idem pour -, *, / et %)
x = x + 1;    peut aussi s'écrire   ++x;      (idem pour - : --x)
```

Types et conversions

Lorsque l'on mélange les types de variables ou lorsque l'on choisit certain opérateurs comme la division, le compilateur effectue (ou non) des conversions de types. Ceci peux amener à des résultats qui ne correspondent pas à ceux attendus! Les compilateurs récents indiquent ces conversions (warning).

Ex.

`5 / 2`	-> 2	entier / entier	=> entier (division entière, tronqué sans arrondis)
`5 % 2`	-> 1	entier % entier	=> entier (%: modulo, reste de la division entière)
`5.0 / 2.0`	-> 2.5	réel / réel	=> réel (division réelle)
`5.0 / 2`	-> 2.5	réel / entier(**->réel**)	=> réel (division réelle)
`5 / 2.0`	-> 2.5	entier(**->réel**) / réel	=> réel (division réelle)

`int x(2.8);` x est initialisé à 2, car le compilateur converti le **réel** -> entier puis instancie la variable

`x = 10.0/6.0;` x = 1, car le compilateur converti le **réel** -> entier

Opérateurs de comparaisons

`==`	égalité
`!=`	différent (non-égalité)
`<`	inférieur
`>`	supérieur
`<=`	inférieur ou égale
`>=`	supérieur ou égale

Ex.

```
x > Y           // vrai si x est plus grand que y, faux sinon
4 == (x + y)    // vrai si x + y vaut 4, faux sinon
```

⚠ **Ne pas confondre** `==` (comparaison) et `=` (affectation)

Pour éviter cette confusion écrire (si possible) `4 == z`
plutôt que `z == 4`

Le compilateur vous indiquera une erreur si vous écrivez `4 = z` pour votre comparaison. L'expression `z = 4` étant valide, il ne vous génèrera pas d'erreur.

Ordre d'évaluation des opérateurs

Précédence des opérateurs, ordre de priorité décroissant:

```
()
!     ++    --
*      /    %
+      -
<     <=    >    >=
==    !=
&&
||
=     +=    -=    *=   /=   %=
```

(plus haute priorité)

(plus basse priorité)

En l'absence de parenthèses, l'ordre de priorité des opérateurs sera appliqué. En cas d'ambiguïté entre opérateurs du même ordre de priorité c'est la règle d'associativité à gauche qui s'applique, ex. a+b+c ≡ (a+b)+c

Pour rendre votre code le plus lisible possible il est recommandé de mettre des parenthèses à vous expressions.

Ordre d'évaluation des opérateurs

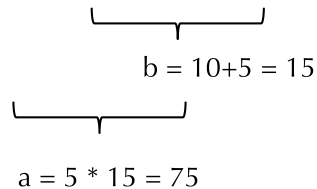
Ex. avec a=5, b=10, c=1

```
x = 2*a + 3*b + 4*c;
```
x = 44

```
x = 2*(a+3)*(b+4)*c;
```
changement de l'ordre de priorité à l'aide des parenthèses, x = 30

Le C/C++ peut être écrit de manière très compact et devenir *illisible*#, exemple d'une expression valide mais *illisible*

```
a *= b += 5;
```
a = 75, b = 15, il y a une double affectation



Ecrit de manière lisible

```
b = b + 5;
a = a * b;
```

Mettez des parenthèses à vos expressions pour les rendre lisibles et pour bien exprimer votre algorithme. Les parenthèses ne *coûtent* rien au niveau de la compilation.

#c'est même un jeu pour les signatures emails de geeks,
pour des exemples voir http://www.iwriteiam.nl/SigProgC.html

Opérateurs logiques

`&&`	et
`||`	ou
`!`	négation

⚠ **A ne pas confondre** avec (`&`, `|`, `^`, …) opérateurs logiques travaillant sur les bits

Ex.

```
(x > 0) && (x ≤ 5)      // vrai si x ∈ ]0..5], faux sinon
(x > 0) || (x ≤ 5)      // toujours vrai
```

Table de vérité

a	b	a && b	a || b	!a
F	F	F	F	V
F	V	F	V	V
V	F	F	V	F
V	V	V	V	F

Evaluation *paresseuse* des opérateurs logiques

Pour les opérations logiques && et ||, l'évaluation des arguments se fait de la gauche vers la droite et seuls les arguments strictement nécessaires à la détermination de la valeur logique sont évalués. L'évaluation de l'expressions s'arrête# au premier argument faux pour && ou vrai pour ||.

évaluation →

Ex.

faux — jamais évalué

```
(1 == 2) && (a + 3 / b == c),
```

$1 \neq 2$ donc 1e parenthèse toujours faux => arrêt de l'évaluation car faux && (…), donnera toujours faux

vrai — jamais évalué

```
(1 == 1) || (a + 3 / b == c),
```

1=1 donc 1e parenthèse toujours vrai => arrêt de l'évaluation car vrai || (…), donnera toujours vrai-> arrêt de l'évaluation

c'est pour cela qu'elle se nomme *paresseuse*

Conversion *true, false*

En C/C++, n'importe quelle expression de **n'importe quel type** peut être considérée comme une expression logique. Si l'évaluation de l'expression conditionnelle est une *valeur nulle (0)*, alors la condition sera dite **false** (faux), sinon (≠ 0) elle sera dite **true** (vrai).

ex. d'expressions vraies:	true || false	0.5 + 0.33
ex. d'expressions fausses:	true && false	16 % 2

Pour éviter toutes confusion écrivez explicitement vos expressions logiques, et utilisez autant de parenthèses que nécessaire

ex.

```
if (x) {…}              peu clair
if (0 != x) {…}         explicite et comparaison valide
```

Les booleans convertis en un entier donnent **0** pour **false** et **1** pour **true.**

ex.

```
x = (1==1) + 1;
```

(1==1) -> vrai -> 1 => x -> 1 + 1 = 2

Premier programme qui ne fait rien

```
// demo C++

int main ()
{



}
```

Commentaires: texte ignoré par le compilateur

nom du programme principale, obligatoirement **main ()**

'{' début du bloc d'instructions/opérations

'}' fin du bloc d'instructions/opérations

Cet exemple contient les instructions minimum pour que votre compilateur accepte votre programme

Deuxième programme

```
// demo C++

int main ()
{
    int x(67);
    x = x + 3;

    return 0;
}
```

Initialise la **variable** nommée **x** à 67

Instruction pour additionner 3 à **x** et mettre le résultat dans **x**

Indique une exécution correcte

Troisième programme qui affiche quelque chose

```
// demo C++

#include <iostream>
using namespace std;


int main ()
{   int x(67);
    x=x+3;
    cout << "x: "<< (x);
    return 0;
}
```

Commentaires: texte ignoré par le compilateur

Libraires externe pour afficher du texte

nom du programme principale, obligatoirement **main ()**

'{' début du bloc d'instructions/opérations

Affiche *x: 70* à l'écran via **cout<<**, **';'** fin de l'instruction courante

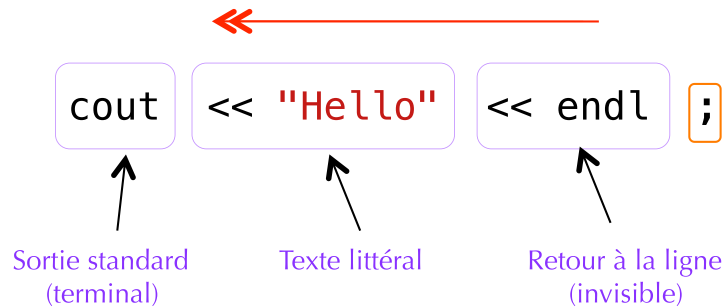
'}' fin du bloc d'instructions/opérations

Affichage sur le terminal

C++

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

Libraire pour l'affichage (ex. cout, <<)
Évite de mettre std:: devant tous les appels de la librairie



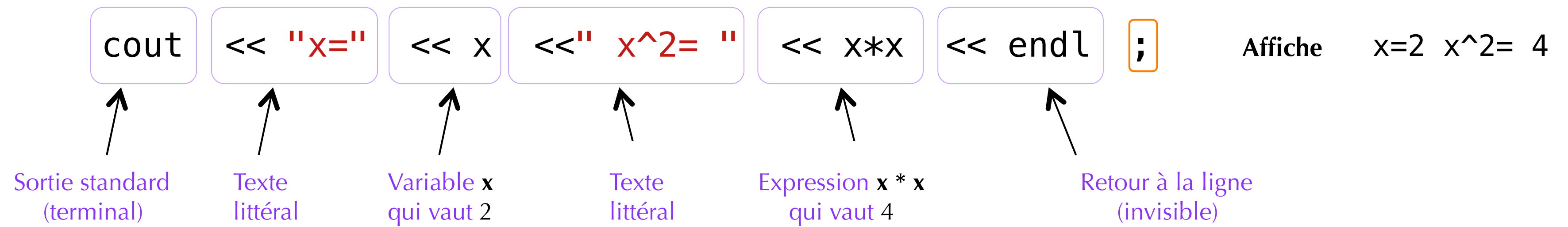
Affichage sur le terminal

C++

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

int x(2);
```

Libraire pour l'affichage (ex. cout, <<)
Évite de mettre std:: devant tous les appels de la librairie



Ce que j'ai appris

- Etapes du développement d'un programme
- Programmation **impérative**:
  - reflète l'architecture von Newman
  - permet d'exprimé un algorithme à l'aide d'un petit nombre de **structures de contrôle**: séquences, conditions, boucles et de **structures de donnée:** constantes, variables, tableaux, récursives (listes, arbres, graphes)
- Variable: permet de stocker définie par un **nom**, un **type**, une **valeur**
- Initialisation et affectation des variables
- Opérations sur les variables, ordre d'évaluation, évaluation paresseuse
- Mon premier programme
- Affichage simple

En informatique, il faut toujours être prêt à tout remettre en question !

Slides supplémentaires

Bonnes pratiques - documentation

Votre code doit pouvoir être repris par quelqu'un d'autre ou vous-même dans 1 mois ou 10 ans…

En plus d'un algorithme correct et valide, votre code doit être:

- Documenté
- Lisible
- Facilement compréhensible
- Assorti de commentaires dans la partie compliquée
- Compliqué ↗ -> commentaires ↗↗

Documentation

En-tête du fichier

Nom du fichier

Description courte (1 ligne)

Description complète contenant les spécifications, algorithmes, etc.

Dépendance

Auteurs/copyright/licence

Version

Révision

```
/**
    MyMathfile.cpp

    Fonctions mathematiques specifiques pour MyProjet1

    Contains a set of functions not available in <math.h>. Data are sorted
    using the less efficient buble sort (n^2) which has the advantage of being
    faster to implement. Will be used in a DLL thus does not use exception.



    Dependancy
      DisplayMyError.framework

    © Me, 2012, GNU licence

    Version 1827365.2

    Revision
           20.12.2011, ChS, initial release
            5.01.2012, ChS, now returns an error and add an option to display it
                              via a dialog
    */
```

Note: idéalement, les commentaires et la documentation sont en anglais. Si vous employez une autre langue avec des caractères accentués, il peut s'avérer judicieux de ne pas employer les caractères accentués, p. ex. é,è -> e.

Documentation

En-tête de fonction

Nom de la fonction

Description courte

Description complète contenant les spécifications, algorithmes, etc.

Entrées/sorties/retour

```
/**
   SumAndMax

   Compute the sum of all coefficiants of the input vector
   Returns the max value and it position in the input vector

   If there are more than 1 max value, returns the first found.
   If the input vector is empty, i.e. if Size is 0 returns pos = -1
   Assume 'Vect' is valid!


   Input
     Size:   number of coefficiants
     Vect[]: coefficiants
   Ouptuts
     Sum: the sum of all coefficiants
     Max: the maximum value
     Pos: the position of the max value, if Size = 0 (empty) pos = -1
   Return
     return: error code, kErrVectEmpty, kErrNoErr

   */
```

Codage – lisibilité

Dans code de la fonction

Convention pour les noms, `SumAndxx`

Utilisation des variables

Convention de noms, `kErr_xx`

Commentaires lorsque c'est nécessaire!

Indentation

Commentaires inutiles

```
int SumAndMax(int Size, int Vect[], int *Sum, int *Max, int *Pos)
   {

     int i;   // index when exploring Vect

    // returns an error if Vect[] is empty
     if (Size==0) {
             *Pos=-1;
             return kErr_ArrayEmpty;
             }

     *Sum = 0;
     *Max = 0;

    // explore Vect from the last position to first one to return
    //    the *first* Max pos, i.e. the last one found

    for (i=Size-1; i=0; i++) {
             *Sum += Vect[i];
             if (Vect[i] >= *Max) {
                          *Max=Vect[i];

                          *Pos=i; // met i dans *Pos  <- inutile!
                          }
             }
     }
```

La plupart des éditeurs/environnements proposent des aides à l'édition, notamment *auto indentation, syntax coloring, code completion, code folding*

Stratégies d’implémentation d’algorithme

Comment faire si je n’ai pas l’algorithme dont j’ai besoin ?

Pas de recette miracle, mais du bon sens!

- Modifier un algorithme existant et/ou la structure des données associée
- Combiner des algorithmes existants
- Décomposer le problème en sous-problèmes jusqu’à ce que le problème soit assez simple pour être résolu par une solution **ad hoc**, c'est l'approche *Divide and Conquer*. Eviter de calculer la même chose plusieurs fois.

- Mélanger les solutions ci-dessus

Stratégies d'implémentation d'algorithme

- ***Brute force***: l'algorithme est souvent simple mais fonctionnel, il est suffisant pour un jeu de données restreint. C'est souvent la première étape pour élaborer si nécessaire un algorithme plus efficace et plus complexe.
- **Diviser pour Régner**: consiste à diviser le problème en sous problème de tailles plus petites jusqu'au pouvoir traiter le problème au cas par cas. En générale en deux temps: premièrement diviser, puis consolider (fusionner) pour trouver une solution globale. Similaire à une approche *top-down* ou *bottom-up*.
- **Recherche exhaustive ou combinatoire** (souvent similaire à *Brute force*) utilise la puissance à disposition (machine parallèle) pour explorer **l'ensemble** des solutions.
- **Aléatoire/probabiliste**: certains algorithmes utilisent des recherches aléatoires, ou par approches successives, donnant de meilleurs résultats (en moyenne) que des recherches directes ou explicites. Ex. algorithme du simplexe (optimisation)
- **Pré/post traitement**: le traitement préalable des données (ex. tri) peut permettre d'accélérer un algorithme ou de réduire la taille des données à traiter.
- **Approximation**: le problème à traiter peut être approximé (itérativement) par une solution plus rapide qu'une solution exacte. Ex. algorithme de Newton pour l'approximation des racines d'une fonction.
- **Heuristique**: tire parti des connaissances à priori, par exemple l'emplois d'un dictionnaire pour *cracker* un mot de passe.

Stratégies d’implémentation

Brute force: l’algorithme est souvent simple mais fonctionnel, il est suffisant pour un jeu de données restreint. C’est souvent la première étape pour élaborer si nécessaire un algorithme plus efficace et plus complexe.

Ex. "Trouver un code secret formé de 6 lettres"

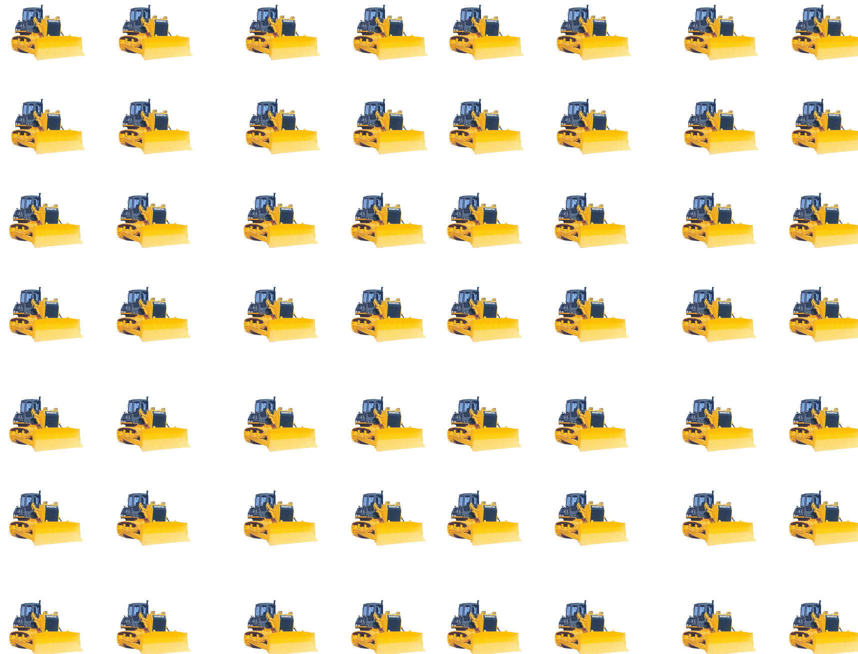
La méthode **brute force** teste systématiquement les codes "aaaaaa" à "zzzzzz" soit 26^6 (~300M) codes différents.

aaaaaa,
aaaaab,
aaaaac,
...
zzzzzx,
zzzzzy,
zzzzzz.

Stratégies d’implémentation

Recherche exhaustive ou combinatoire (souvent similaire à *Brute force*) utilise la puissance à disposition (machine parallèle) pour explorer **l’ensemble** des solutions.

Ex. "Aux échecs, pour toutes les pièces sur l'échiquier recherche de tous les coups possibles à m tours à l'avance"



Stratégies d’implémentation

Une **heuristique** ou une connaissance à priori permet de réduire la taille de l'ensemble des solutions.

Ex. "Trouver un code secret formé de 6 lettres"

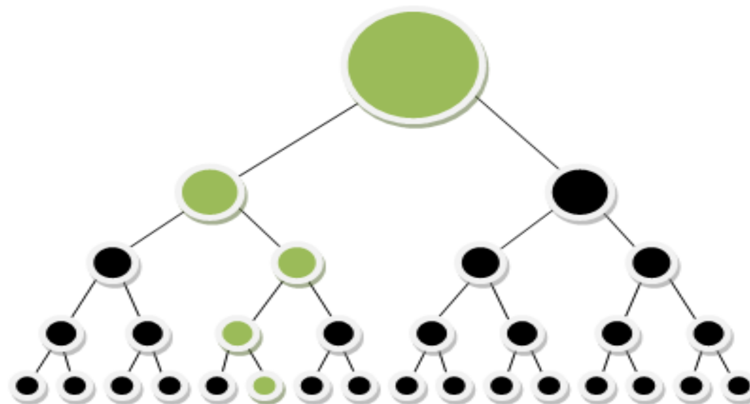
Nous savons que le code appartient à la langue française. Au lieu de tester toutes les solutions possibles, uniquement les mots contenus dans un dictionnaires seront testés. Selon `http://www.liste-de-mots.com/mots-nombre-lettre/6/` il existe 15913 mots de 6 lettres en français

abaque,
abatte,
abattu,
…
zézaya,
zézaye,
zézayé.

Stratégies d’implémentation

Diviser pour Régner: consiste à diviser le problème en sous problème de tailles plus petites jusqu’au pouvoir traiter le problème au cas par cas. En générale en deux temps: premièrement diviser, puis consolider (fusionner) pour trouver une solution globale. Similaire à une approche *top-down* ou *bottom-up*.

Ex. "recherche dichotomique, l'ensemble des solutions possibles est réduit (÷ 2) à chaque itérations"



Stratégies d’implémentation

Pré/post traitement: le traitement préalable des données peut permettre d’accélérer un algorithme ou de réduire la taille des données à traiter ou de réduire la taille de l'ensemble des solutions.

Ex.1. "Le tri des éléments d'un tableau permet d'accélérer la recherche d'appartenance d'un élément au tableau", voir Appartient2()

Ex.2. "Calcul du nombre de cellules dans une boîte de Pétri "

La première étape d'un algorithme de reconnaissance de particules est la conversion de l'image couleur (1 pixel = 24 bit) en une image noir-blanc (1 pixel = 1 bit).

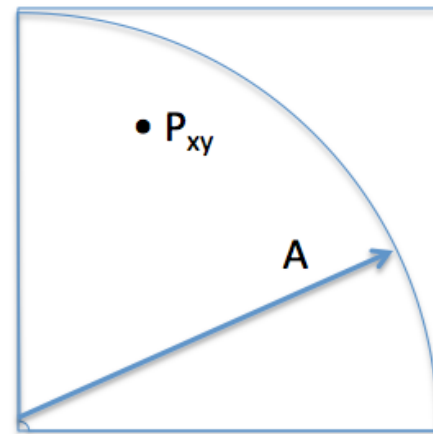
Stratégies d'implémentation

Aléatoire/probabiliste: certains algorithmes utilisent des recherches aléatoires, ou par approches successives, donnant de meilleurs résultats (en moyenne) que des recherches directes ou explicites.

Ex. "Calculer π par approximation successives"

On tire aléatoirement les valeurs de *x* et *y* entre 0..A. Le point P_{xy} appartient au disque de centre (0,0) de rayon A si et seulement si $x^2 + y^2 \leq A$. La probabilité que le point *P* appartienne au disque est $\pi/4$.

En faisant le rapport du nombre de points dans le disque au nombre de tirages, on obtient une approximation du nombre $\pi/4$ si le nombre de tirages est grand.



Stratégies d'implémentation

Approximation: le problème à traiter peut être approximé (itérativement) par une solution plus rapide qu'une solution exacte.

Ex. "Algorithme de Newton pour l'approximation des racines d'une fonction."