

# Principes & paradigmes

Eric Lecolinet - Télécom Paris – Institut Polytechnique de Paris

<http://www.telecom-paris.fr/~elc>

# Dans ce cours

## Organisation du cours

- principaux **paradigmes et principes** des langages informatiques
- **orienté objet** illustré en **C++** (et comparaison **Java**)
- **autres concepts** et compléments
- **programmation événementielle** et interfaces graphiques **Java Swing**

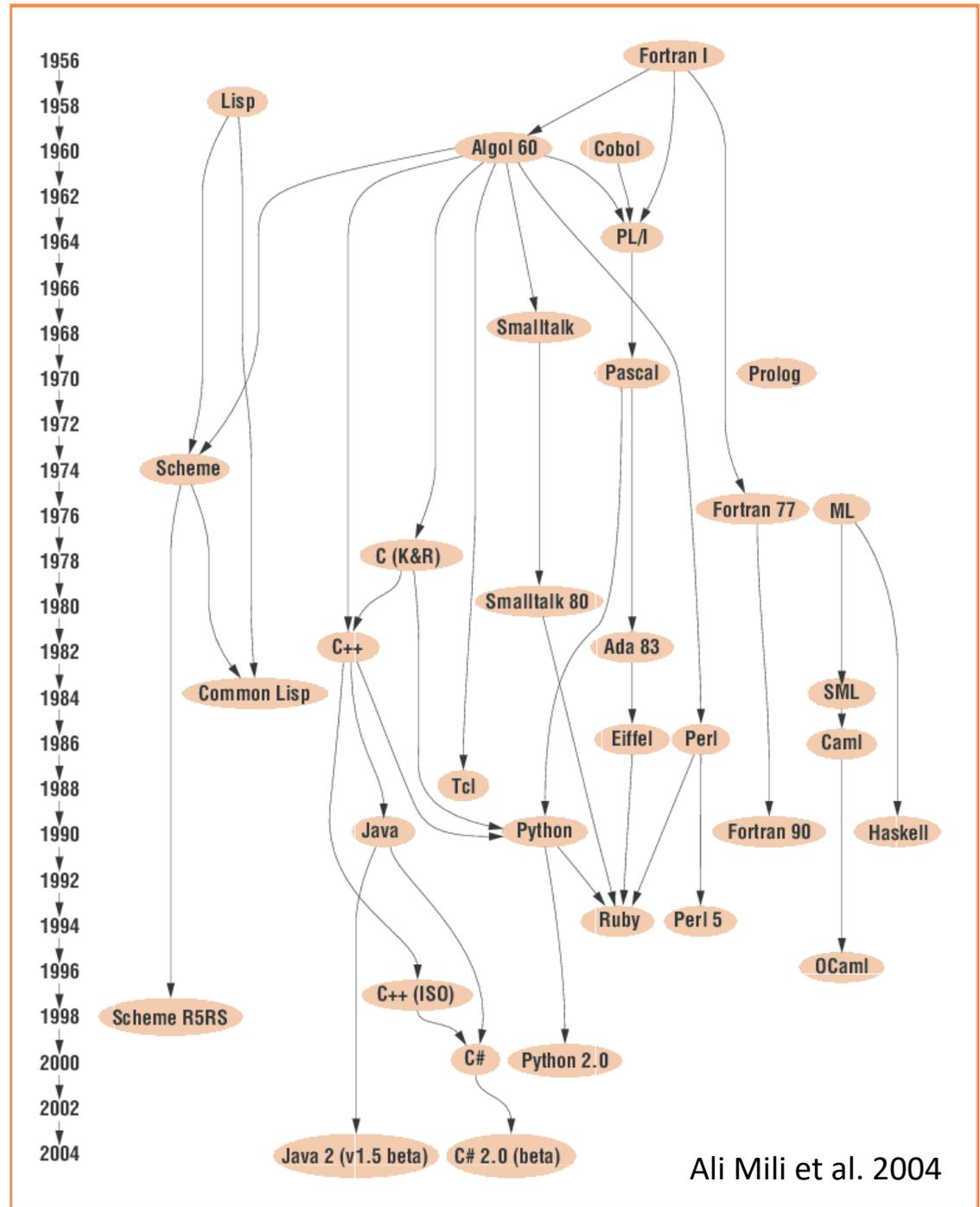
## Liens

- <http://www.telecom-paristech.fr/~elc/>
- <http://www.telecom-paristech.fr/~elc/inf224/> - lien vers page **eCampus**

# Paradigmes

## Paradigme

- approche, philosophie
- la plupart des langages actuels sont **multi-paradigmes**



Ali Mili et al. 2004

# Paradigmes

## Principaux paradigmes

- **Impératif**
  - exécution pas à pas des instructions
  - *Fortran, C, Pascal ...*
- **Fonctionnel**
  - évaluation de formules, lambda-calcul
  - *Lisp, Scheme, Caml, Haskell ...*
- **Orienté objet**
  - modélise des interactions entre des entités
  - *Smalltalk, C++, Python, Java, C#, Swift ...*
- **Logique**
  - recherche via des règles (Prolog)
  - démonstration automatique

# Paradigmes

- **Déclaratif**
  - décrire le "*quoi*" plutôt que le "*comment*"
- **Événementiel**
  - gérer les réponses à des **événements**
  - typique des interfaces graphiques, du Web ...
- **Concurrent**
  - langages orientés concurrence pour gestion multi-tâches (Go, Erlang ...)
  - partiellement intégré dans langages courants (**thread, mutex, future** ...)
- **Langages spécialisés**
  - pour le **Web** (PHP, Perl, JavaScript, TypeScript ...)
  - langages de **scripts** (bash, zsh ...)
  - langages de **simulation** (circuits)
  - langages **réactifs/synchrones** (embarqué), etc.

# Paradigme impératif

## Principe

- un programme est :
  - une suite d'**instructions**
  - à l'intérieur de **structures de contrôle** emboîtées
- exécution **pas à pas** des instructions

```
int vec[100] ;  
...  
int somme = 0 ;  
for (int i=0;i<100;i++)  
    somme += vec[i] ;
```

### Structures de contrôle de l'exécution :

- **if-then-else**, etc.
- boucles **for**, **while**, etc.
- appels de **fonctions**
- **exceptions**, etc.

# Paradigme impératif

## Principe

- un programme est :
  - une suite d'**instructions**
  - à l'intérieur de **structures de contrôle** emboîtées
- exécution **pas à pas** des instructions

```
int vec[100] ;  
...  
int somme = 0 ;  
for (int i=0;i<100;i++)  
    somme += vec[i] ;
```

## La gestion des données incombe au programmeur

- qui doit :
  - déclarer les **variables** (et leur **type**)
  - décider de leur **durée de vie** (on y reviendra)
  - décider des **effets de bord** (affectations = ce que les variables contiennent)

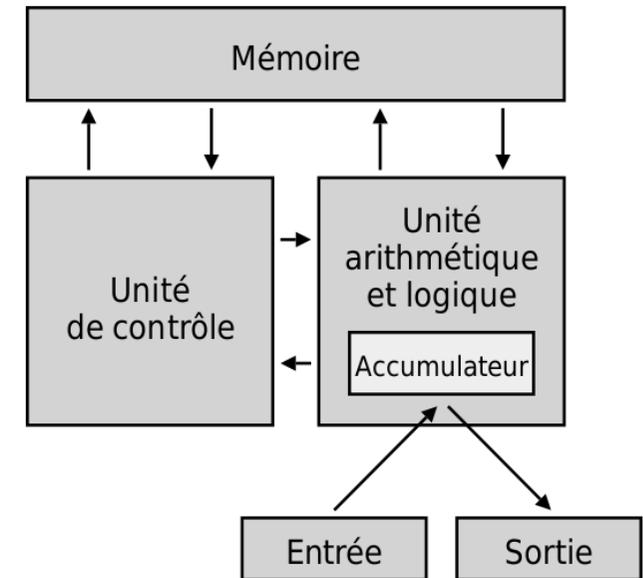
# Origines du paradigme impératif

## Machine de Turing

- modèle **abstrait** permettant des calculs arbitraires

## Machine de von Neumann

- architecture **matérielle** d'un ordinateur (1945)
- instructions numérotées :
  - arithmétiques
  - de transfert de données
  - de rupture de séquence (**goto**)



wikipedia

- l'instruction d'affectation :  
`<variable> := <expression> ;`

- l'instruction de *saut conditionnel* :  
`if <condition> goto <instruction> ;`  
et *inconditionnel* :  
`goto <instruction> ;`

# Exemple : programme d'addition

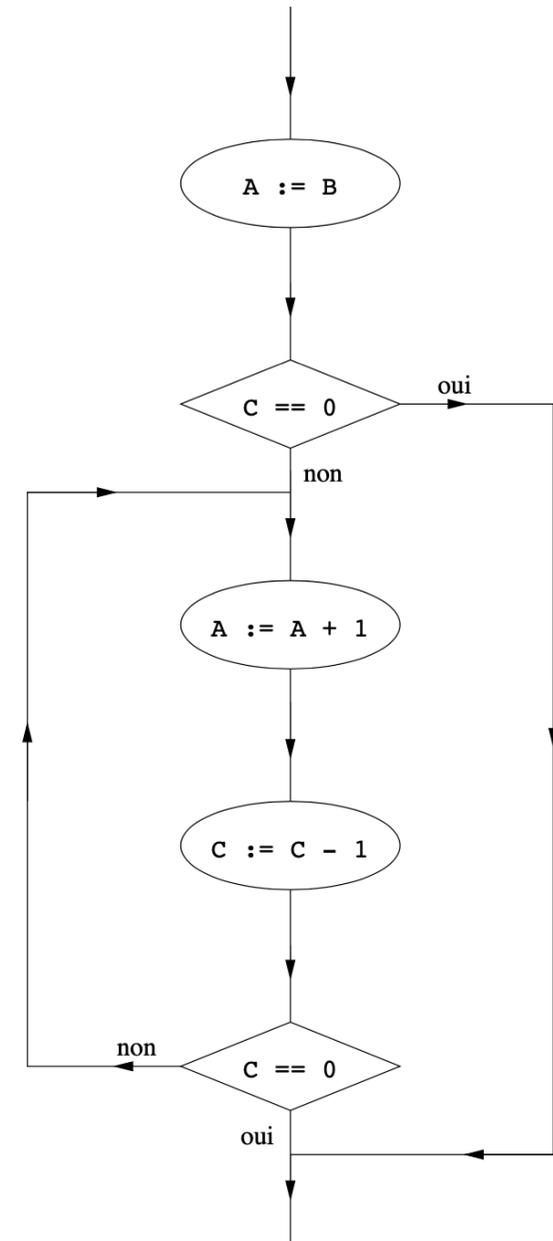
## Pré-conditions

- B et C sont des variables contenant des entiers positifs
- on ne sait faire que +1 et -1

## Post-conditions

- A vaut B + C
- C vaut 0 et B inchangée

```
00 : A := B ;  
01 : if (C == 0) goto 05 ;  
02 : A := A + 1 ;  
03 : C := C - 1 ;  
04 : if (C <> 0) goto 02 ;  
05 : end ;
```



organigramme

# Pré-conditions et post-conditions

## Pré-conditions

- conditions qui doivent être vérifiées **avant** le traitement

## Post-conditions

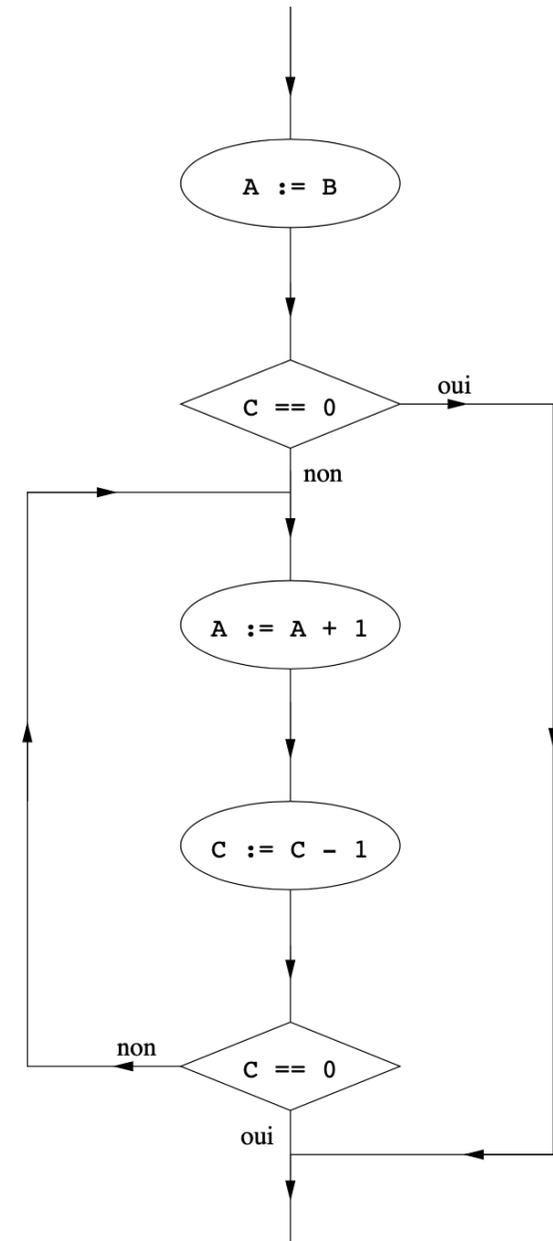
- conditions qui doivent être garanties **après** le traitement

## Invariants

- conditions **toujours** vraies

## Il faut :

- les **expliquer** (documentation)
- si possible les **vérifier dans les fonctions**
  - au moins en **phase de développement**
  - pour détecter les **erreurs**



organigramme

# Programmation structurée

## Théorème de Boehm et Jacopini (en substance)

- toute **fonction calculable** peut être calculée en combinant des sous-programmes via **trois structures de contrôle** :
  - **séquencement**
  - **tests** (if-then)
  - **itérations** (boucles)

## Programmation structurée

- dérivée de Boehm et Jacopini
- tout programme peut s'écrire en n'utilisant que **while** et **if-then** (donc sans **goto**)

```
00 : A := B ;  
01 : if (C == 0) goto 05 ;  
02 : A := A + 1 ;  
03 : C := C - 1 ;  
04 : if (C <> 0) goto 02 ;  
05 : end ;
```



```
A := B ;  
while (C <> 0) {  
    A := A + 1 ;  
    C := C - 1 ;  
} ;
```

# Programmation structurée

## Structure **if-then**

```
if <condition> {  
    <instruction 1>  
    ...  
    <instruction k>  
} ;
```

devient :

```
n:      if (not <condition>) goto n+k+1 ;  
n+1:    <instruction 1>  
    ...  
n+k:    <instruction k>  
n+k+1:
```

## Structure **while**

```
while <condition> do {  
    <instruction 1>  
    ...  
    <instruction k>  
} ;
```

devient :

```
n:      if (not <condition>) goto n+k+2 ;  
n+1:    <instruction 1>  
    ...  
n+k:    <instruction k>  
n+k+1:  goto n ;  
n+k+2:
```

# Paradigme fonctionnel

## Origine

- dérivé de la théorie du  **$\lambda$ -calcul** (Church, années 30)
- langage **Lisp** (J. McCarthy, années 50)

## Principe

- repose sur l'**évaluation de fonctions**
- pas d'**effets de bord** (en fonctionnel "pur")
- repose largement sur la **récurtivité** (pas de boucles !)

```
(def fact(n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

```
int fact(int n) {
  return
    (n == 0)
    ? 1
    : (n * fact(n-1)) ;}
```

Factorielle en **Lisp** et en **C**

(notation **préfixe** en Lisp, **infixe** en C)

# Lisp

## Programme = donnée

- **fonctions** de "**première classe**"  
(peuvent être un argument)
- une **fonction** est une **donnée**  
(c'est une liste)

```
? (defun twice(f x) (apply f (apply f x)))  
= twice
```

```
? (defun z(x) (+ x 1))  
= z
```

```
? (twice 'z 1)  
= 3
```

## Lambda = fonction anonyme

- notée ( $\lambda x . x + 1$ )  
dans la théorie du  **$\lambda$ -calcul**
- cette notion :
  - a été reprise dans les **langages courants** (C++, Java, Python ...)
  - où elle est liée à la **capture de variables**

```
? (twice '(lambda (x) (+ x 1)) 1)  
= 3
```

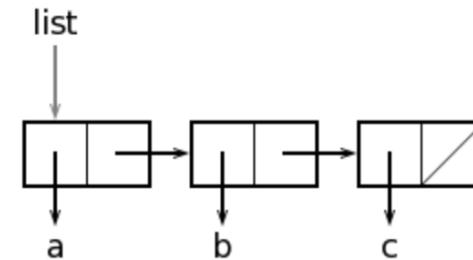
# Lisp

## Tout s'exprime sous forme de listes

- Listes **chaînées** composées de **doublets**
  - **(cons x y)** crée un **doublet**
  - **(car d)** renvoie la composante **gauche**
  - **(cdr d)** renvoie la composante **droite**
  - **nil** ou **()** est la liste **vide**
  - **(null x)** est vrai si x est vide
- Un **programme** est une **liste** !

## Gestion mémoire automatique

- le programmeur **n'a pas à gérer la mémoire** !
  - **ramasse-miettes** (garbage collector)
  - idée reprise dans Java, Python, etc.



```
? (defun last(x)
    (if (null (cdr x))
        (car x)
        (last (cdr x))))
= last
? (last '(a b c d))
= d
```

# Lisp

## Reflexivité

- Lisp est écrit en Lisp (en théorie)
- Lisp est **réflexif**

```
(defun toplevel()  
  (forever  
    (print (eval (read))))))
```

## Reflexion : deux notions

- **1) Introspection** : un programme peut **examiner** son état (ou ses classes)
  - notion la plus courante
  - **métaclasses** de Java, C#, Python, JavaScript ...
    - ex : class **Class** de Java
- **2) Code auto-modifiant** : un programme peut **modifier son comportement**
  - Lisp peut redéfinir **print()** ou **eval()** en cours d'exécution

# Paradigme fonctionnel

## Lisp a été source d'inspiration

- de **langages fonctionnels** plus récents (Scheme, Caml, Erlang, F# ...)
- dans le monde **académique** mais aussi l'**industrie** (Erlang = Ericson language)
- des **langages courants** :
  - C++, Java, C#, Python, etc.
  - récursivité, ramasse-miette, lambdas, réflexion, etc.

## Avantages / inconvénients du fonctionnel

- **plus robuste**, plus facile à **vérifier** et à **maintenir**
  - du fait de l'absence d'états / d'effets de bord
- généralement **moins performant** que l'impératif

# Paradigme logique

## Programmation logique

- issue de la **démonstration automatique**
- repose sur :
  - une base de **faits**
  - une base de **règles** logiques
  - un **moteur d'inférence**
- forme de **programmation déclarative**

## Prolog

- Colmerauer, années 1970, Marseille
- basé sur les **clauses de Horn**

**Clause Prolog :**  $p(x, \dots) \text{ :- } q_1(x, \dots), \dots, q_n(x, \dots).$

**signifie :**  $p(x, \dots)$  si  $q_1(x, \dots)$  et  $\dots$  et  $q_n(x, \dots)$   
(clause de Horn)

# Prolog

**Faits :**

*atomes*      *terme composé*

```
pere(patrick, jerome).
pere(patrick, helene).
pere(patrick, camille).
pere(patrick, daniel).
mere(marianne, jerome).
mere(marianne, helene).
mere(fadila, camille).
mere(lam, daniel).
```

**Règles :**

*variables*

```
parent(X, Y) :- pere(X, Y).
parent(X, Y) :- mere(X, Y).

grand-pere(X, Y) :- pere(X, Z), parent(Z, Y).
grand-mere(X, Y) :- mere(X, Z), parent(Z, Y).
```

```
couleur(voiture(patrick), bleu).
```

## Buts (queries) :

<pre>?- pere(patrick, camille). OK</pre>	<pre>?- pere(patrick, X). X=jerome, OK</pre>	<pre>?- couleur(voiture(X), bleu). X=patrick, OK</pre>
<pre>?- pere(patrick, julie). NO</pre>	<pre>X=helene, OK X=camille, OK</pre>	<pre>?- couleur(voiture(X), rose). NO</pre>

*tester*

*chercher*

# Prolog

## Méthode de résolution SLDNF

(Selection, Linear, Definite, Negation as failure)

### ▪ Unification

- trouver les **valeurs** pour que deux termes soient **identiques**

```
g( f(X, Y), Z, 2, U) et g( f(1, 3), Y, T, V)
=> X = 1, Y = Z = 3, T = 2, U = V
```

### ▪ Résolution d'un but $p(a,b)$ avec une clause $p(x,y) :- q1(...), \dots, qn(...)$ .

- unifier  $p(a,b)$  avec  $p(x,y)$
- si succès, résoudre  $q1(...)$ , etc. dans l'environnement résultant
- et ainsi de suite, récursivement

```
but : parent(X, camille)
```

```
parent(X, Y) :- pere(X, Y).
parent(X, Y) :- mere(X, Y).
```

```
pere(patrick, camille). ←
pere(patrick, daniel).
mere(marianne, jerome).
mere(marianne, helene).
mere(fadila, camille). ←
```

### ▪ Négation par l'échec

- ce qui n'est pas vrai est faux

# Prolog

## En pratique

- la **combinatoire** sous-jacente peut rendre le temps excessif
  - => opérateur de **cut** qui détruit une branche de recherche
- l'ordre des clauses compte !
- peut vite devenir assez complexe ...

## Utilisation, évolutions

- utilisé principalement en **IA** et traitement linguistique
- diverses extensions (équations, arbres infinis, contraintes, etc.)

# Paradigme déclaratif

## Principe

- on décrit le "**quoi**" plutôt que le "**comment**"
- paradigme "transversal" (combiné avec d'autres)

## Exemples

- programmation **logique** ou par contraintes
  - *ex Prolog* : base de faits et de règles
- programmation **fonctionnelle pure** (sans boucles ni affectations)
  - *ex Lisp* : évaluation de fonctions / formules
- langages **descriptifs**
  - *ex HTML* : décrit une page Web (pas comment la construire)
  - *contre-exemple* : programme Javascript / DOM qui crée une page Web

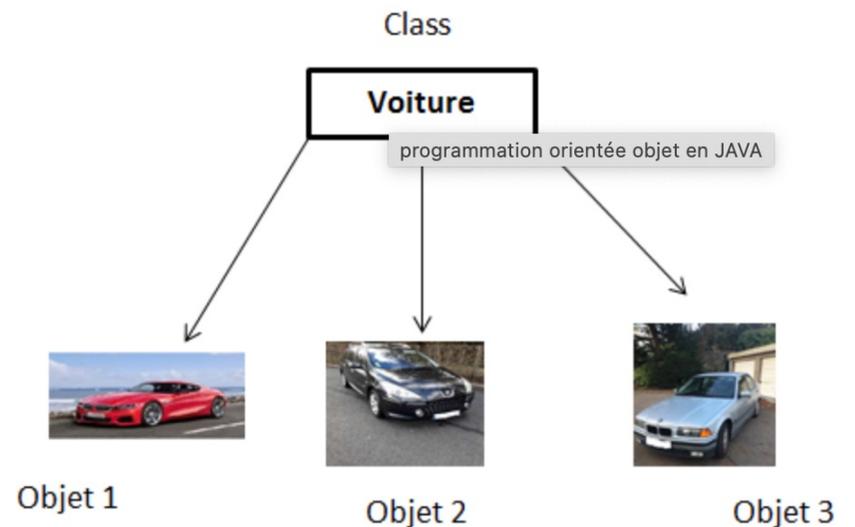
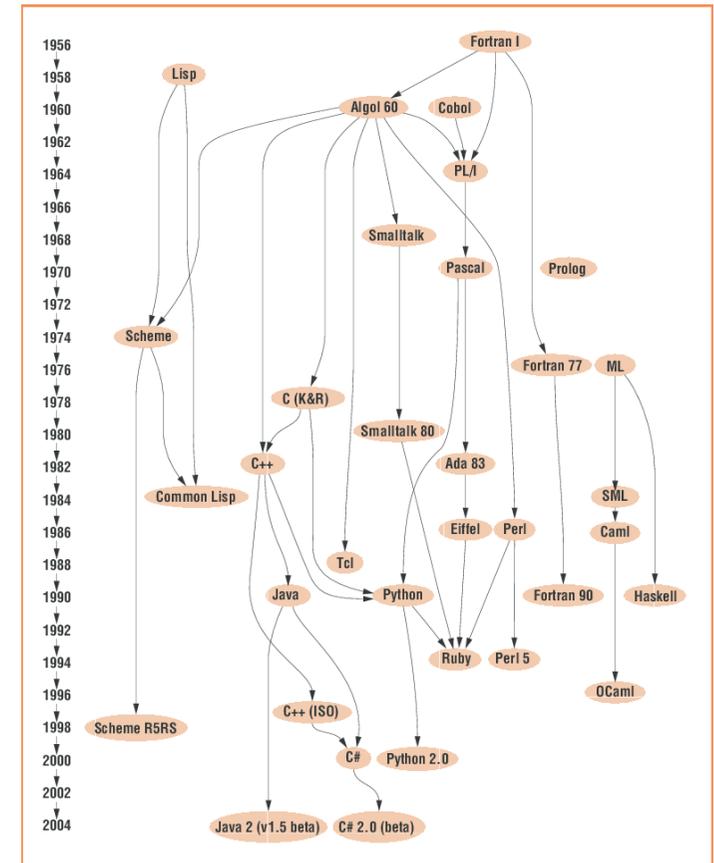
# Paradigme objet

## Historique

- **origine** : Smalltalk, programmation structurée
- fort développement dans les années 80
- devient **dominant** dans les années 90
- *ex* : C++, Eiffel, Java, C#, Python, Swift ...

## Principe

- modélise des **interactions** entre des **entités** appelées **objets**
- les **objets** sont des **instances** des **classes**
  - exception : langages de **prototypes** :
    - pas de classes mais des **objets** qui servent de **modèles**
    - *ex* : *Javascript* (initialement)



# Paradigme objet

## Point de vue

- modèle "social" d'**interaction** entre **acteurs**



### **Alice** (manager)

- **ne fait pas le rapport** (ce n'est pas son **rôle**)
- **ne "farfouille" pas** dans les affaires de Bob !

### **Bob** (ingé thermique)

- **est responsable** du rapport **thermique** (c'est sa spécialité)

# Paradigme objet

L'orienté objet c'est pareil !



## L'objet Alice

- ne fait pas le rapport => **envoie un message**
- ne "farfouille" pas => **les données de Bob sont privées**

## L'objet Bob

- **est responsable** du rapport => **décide des actions à exécuter**

# Paradigme objet

L'orienté objet c'est pareil !



Alice



*bob.performReport()*



Bob

## L'objet Alice

- ne fait pas le rapport => **envoie un message**
- ne "farfouille" pas => **les données de Bob sont privées**

## L'objet Bob

- est responsable => **décide des actions à exécuter**

principes de l'OO  
qu'on verra plus loin

appel de méthode

attributs  
encapsulation  
droits d'accès

encapsulation  
abstraction

# Paradigme objet

## Polymorphisme d'héritage



Alice

*bob.performReport()*



Bob

*mallory.performReport()*



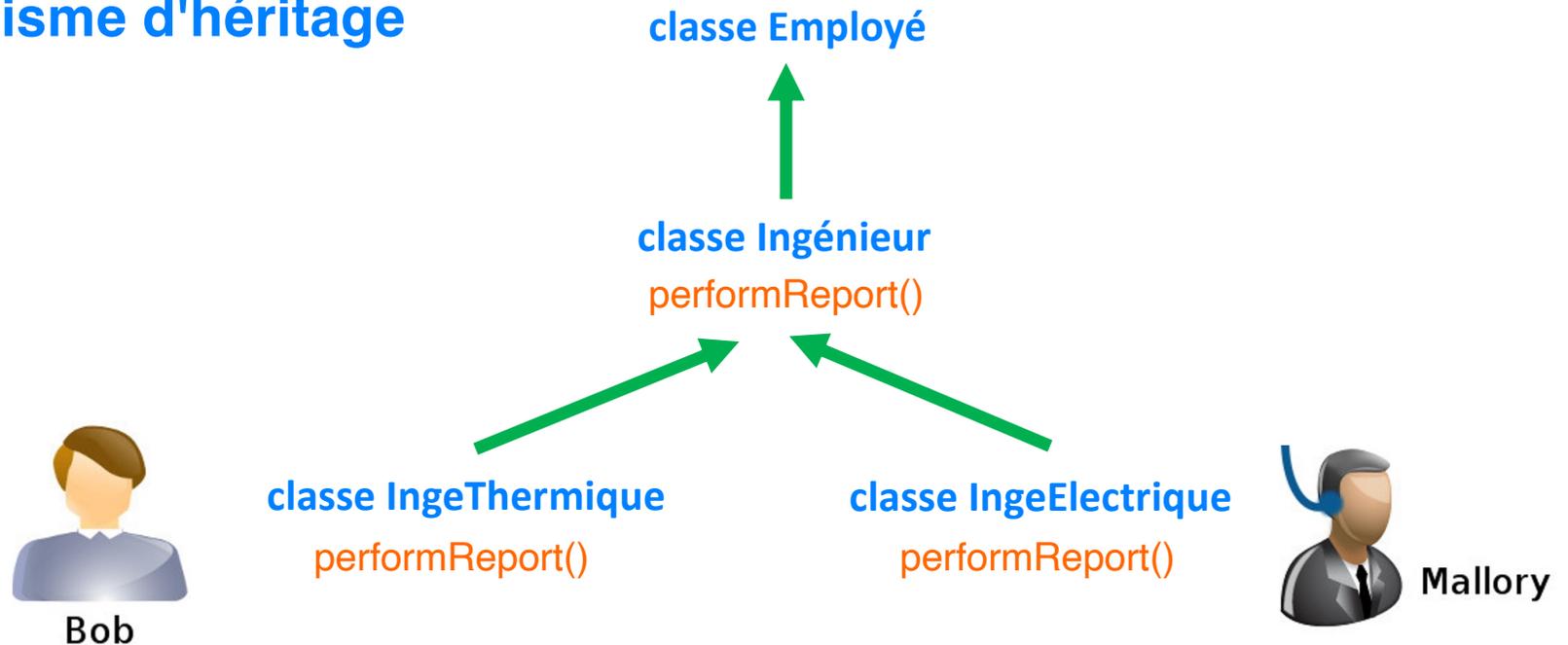
Mallory

### Même message, effet différent !

- **Bob** = expert thermique  
=> fait le rapport **thermique**
- **Mallory** = expert électricité  
=> fait le rapport **électrique**

# Paradigme objet

## Polymorphisme d'héritage



## Polymorphisme: même message, effet différent

- **Bob** = instance de **IngeThermique** => **performReport()** fait le rapport **thermique**
- **Mallory** = instance de **IngeElectrique** => **performReport()** fait le rapport **électrique**

# Compléments, outils

# Types

## Type

- Définit ce qu'une variable **peut faire**

## Typage dynamique

- type déterminé à l'**exécution**
- il peut **changer**

```
a = 1  
a = 'toto'  
print(a)
```

Python affiche toto

## Typage statique

- type déterminé à la **compilation**
- il peut **ne peut pas** changer

```
int i = 1;  
i = "toto";
```

Java, C++  
ne compile pas !

## Inférence de type

- type déterminé **automatiquement**
- dynamiquement (Python) ou statiquement (C++, Java)

```
a = 1;           Python  
auto i = 1;     C++  
var i = 1;      Java
```

# Typage dynamique vs. statique

## Avantages / inconvénients

- typage **dynamique** : plus **pratique**, facilite l'**écriture**
  - typage **statique** : **moins d'erreurs** à l'exécution, moins de **débogage**
- => c'est à la fois un des principaux avantages et inconvénients de *Python* !

```
def foo(val):  
    if val == 0:  
        print(val + 'toto')  
    else:  
        print(val)
```

Python

ne détecte pas d'erreur et pourtant ...

```
foo(1)
```

OK

```
foo(0)
```

Erreur à l'exécution !

# Polymorphisme

## Polymorphisme = plusieurs formes

- même **interface** pour différents **types**

## Polymorphisme d'héritage

- la **même** méthode a un **effet différent** suivant la sous-classe (cf. précédemment)  
=> abstraction, spécialisation
- souvent appelé "**polymorphisme**" (sans préciser "d'héritage")

## Polymorphisme paramétré

- les types sont des **paramètres** d'autres types
- ex : **Generics** de Java, **Templates** de C++

```
var table = new ArrayList<String>();      Generics Java  
auto table = new vector<string>();      Templates C++
```

# Interprétation vs compilation

## Code interprété

- décodé et exécuté **au fil de l'eau**
  - ⇒ **pratique** (résultat visible immédiatement)
  - ⇒ **performance** —
- *Python (en apparence)*

```
>>> a = 1
>>> a = 'toto'
>>> print(a)
toto
```

interpréteur Python

## Code compilé

- traduit en un **exécutable** par un **compilateur**
- **exécutable** = **code machine**
  - propre à un type de machine
    - ⇒ **non portable**
    - ⇒ **performance +++**
- *C, C++*

# Interprétation vs compilation

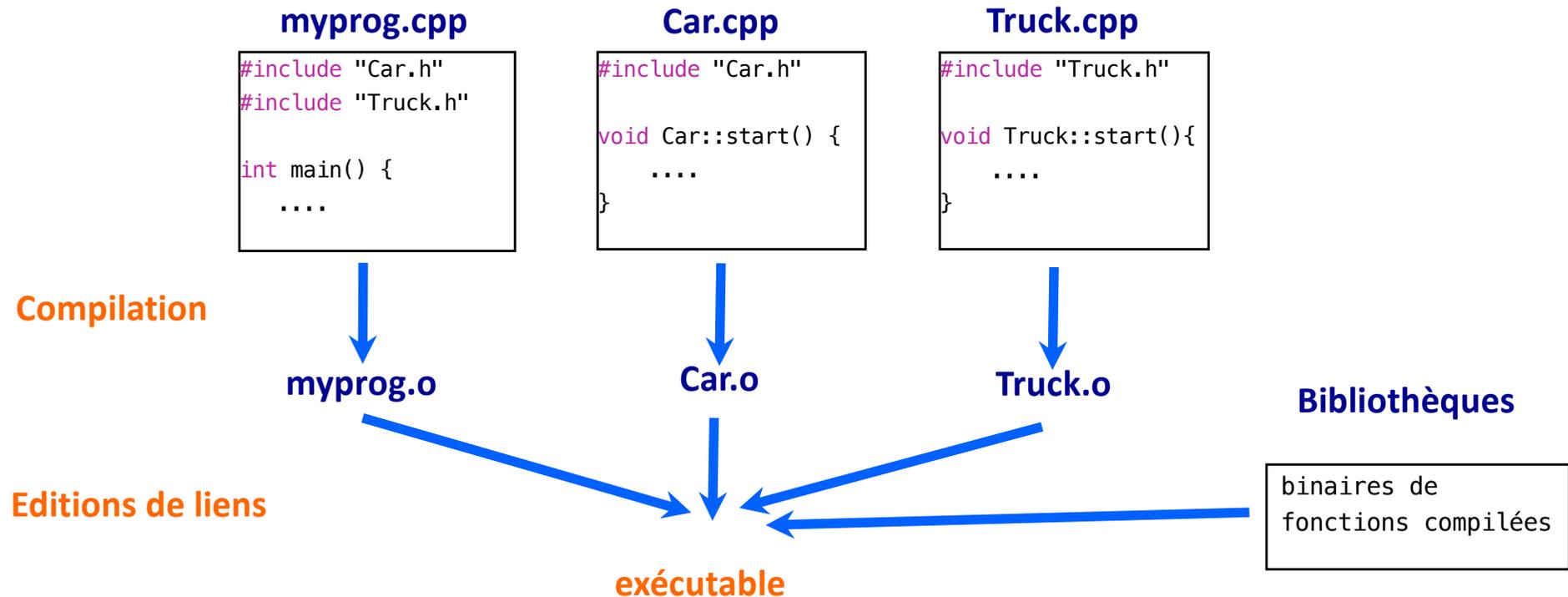
## Bytecode

- Approche mixte :
  - 1) le programme est **compilé** en **bytecode**
  - 2) le **bytecode** est **interprété** par une **machine virtuelle**
    - => portable**
    - => performance +**
- *Java + JVM (Java Virtual Machine)*
- *Python (en réalité)*

## Compilation JIT (juste-à-temps)

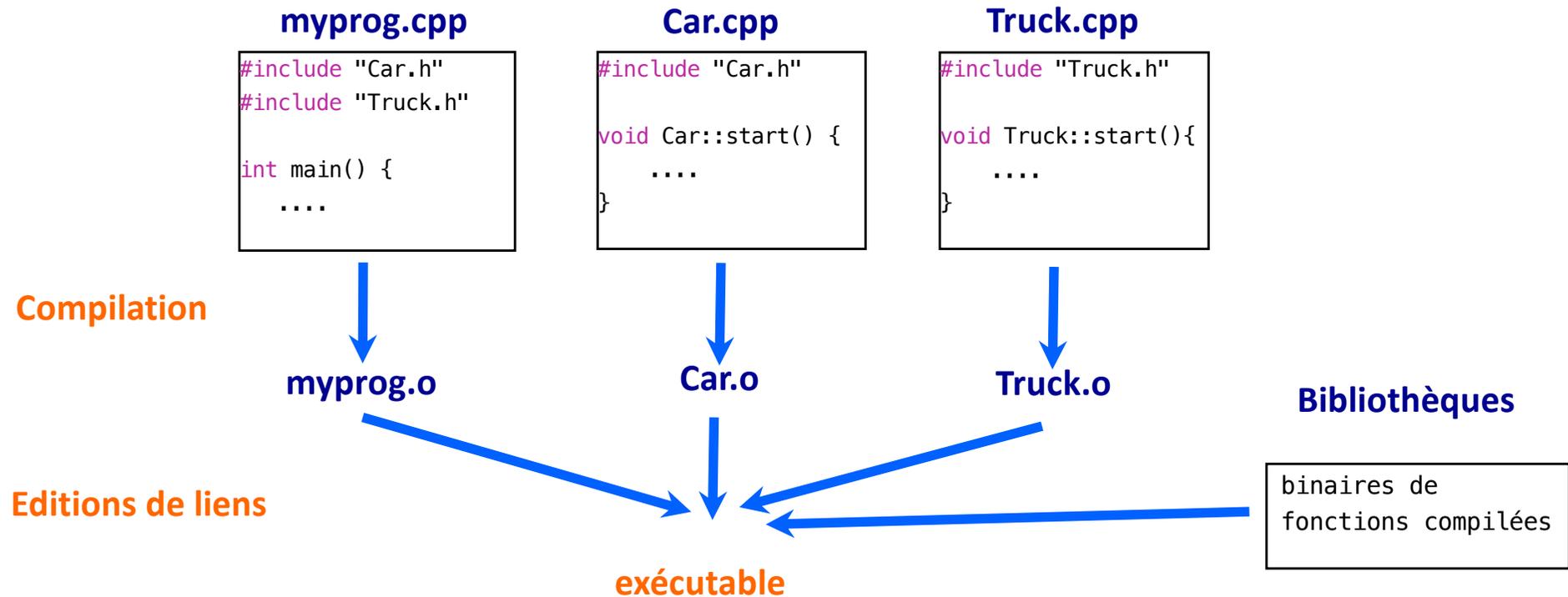
- une partie du **bytecode** est **compilé** en **code machine** pendant l'**exécution**
  - => performance ++**
- *Java, JavaScript*

# Compilation séparée



- les fichiers sources sont compilés **indépendamment**
- l'**éditeur de liens (linker)** :
  - resout les **symboles** des fichiers générés
  - les lie aux **bibliothèques**
  - génère un **exécutable**

# Librairies statiques et dynamiques



- **Bibliothèques statiques**
  - leur code binaire est **inclus** dans l'exécutable
- **Bibliothèques dynamiques**
  - les liens sont faits à **l'exécution**
  - **avantage** : programmes (beaucoup) moins gros et plus performants

# Performance

## Critères

- 1. **temps** d'exécution
- 2. consommation **énergétique**
- 3. occupation **mémoire**
- 4. (éventuellement) **temps de compilation**

# Performance

## Ranking Programming Languages by Energy Efficiency

Rui Pereira et al.

in *Science of Computer Programming*, Elsevier

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01
(i) Perl	6.62
(i) Lua	6.72
(v) Erlang	7.20
(i) Dart	8.64
(i) Jruby	19.84

# Performance

## Conséquences

- ne pas négliger l'**impact énergétique** (et le CO2 induit !)
- utiliser / **combiner** au mieux les langages
  - ex : du *Python* qui appelle des routines *C/C++*
- **optimiser l'exécutable**
  - cf. mode "Debug" vs. "Release"
- **optimiser le code source**
  - jamais "au juger" => **inefficace** et source de **bugs**
  - utiliser des outils de **profilage**

# Optimiser l'exécutable

## Mode développement (Debug)

- exécutable **non optimisé**
- options de **compilation (g++)** :
  - **-g** : pour utiliser un **débogueur**

## Mode déploiement (Release)

- exécutable **optimisé**
- options :
  - **-O2** : optimisé en **vitesse**
  - **-O4 / -Ofast** : super-optimisé (parfois faux!)
  - **-Os / -Oz** : optimisé en **taille**

## Notes

- **-DXYZ** : pour tests / messages de mise au point
- **-DNDEBUG** : ignore `assert()`

- les options dépendent des **compilateurs**
- les **modes** des **IDEs** affectent ces options

```
void foo(int i) {  
    assert(i > 0);  
  
    #ifdef XYZ  
        .....  
    #endif  
}
```

# Optimiser le code

## Profileurs

- affichent le **temps passé** dans chaque fonction : **indispensables** pour optimiser !

The screenshot displays the Xcode Instruments interface for a process named 'guitdemo'. The top section shows a timeline with a CPU usage graph. Below the timeline, a table lists the profile's components, including the main thread and various system and application functions. On the right, a 'Heaviest Stack Trace' shows the call stack for the selected function.

Weight	Self Weight	Symbol Name
4.60 s 100.0%	0 s	guitdemo (2631)
3.69 s 80.2%	0 s	Main Thread 0xdc9c
3.68 s 79.9%	0 s	start dyld
3.57 s 77.6%	0 s	main guitdemo
3.57 s 77.6%	0 s	guit::GCore::start(guit::GWindow*) guitdemo
3.52 s 76.5%	0 s	guit::GNatContext::startLoop() guitdemo
1.88 s 40.8%	0 s	guit::GNatContext::waitEvent(SDL_Event*) [inlined] guitdemo
1.86 s 40.4%	0 s	SDL_PumpEventsInternal libSDL2-2.0.0.dylib
17.00 ms 0.3%	1.00 ms	SDL_Delay_REAL libSDL2-2.0.0.dylib
1.00 ms 0.0%	0 s	SDL_PeepEventsInternal libSDL2-2.0.0.dylib
1.11 s 24.2%	1.00 ms	guit::GCoreImpl::paintAllDamaged() guitdemo
978.00 ms 21.2%	0 s	guit::GCoreImpl::paintWindow(guit::GWindow*) guitdemo
915.00 ms 19.8%	0 s	guit::GBox::doPaint(guit::GGraphics&, guit::GRender*) guitdemo
914.00 ms 19.8%	0 s	guit::GRender::drawBox(guit::GGraphics&, guit::GBox&) guitdemo
865.00 ms 18.8%	0 s	guit::GBox::doPaint(guit::GGraphics&, guit::GRender*) guitdemo
865.00 ms 18.8%	0 s	guit::GRender::drawBox(guit::GGraphics&, guit::GBox&) guitdemo
863.00 ms 18.7%	0 s	guit::GBox::doPaint(guit::GGraphics&, guit::GRender*) guitdemo
861.00 ms 18.7%	0 s	guit::GRender::drawBox(guit::GGraphics&, guit::GBox&) guitdemo
758.00 ms 16.4%	0 s	guit::GBox::doPaint(guit::GGraphics&, guit::GRender*) guitdemo
758.00 ms 16.4%	0 s	guit::GRender::drawBox(guit::GGraphics&, guit::GBox&) guitdemo
612.00 ms 13.3%	0 s	guit::GBox::doPaint(guit::GGraphics&, guit::GRender*) guitdemo
610.00 ms 13.2%	0 s	guit::GRender::drawBox(guit::GGraphics&, guit::GBox&) guitdemo
353.00 ms 7.6%	0 s	guit::GBox::doPaint(guit::GGraphics&, guit::GRender*) guitdemo
352.00 ms 7.6%	0 s	guit::GRender::drawBox(guit::GGraphics&, guit::GBox&) guitdemo
1.00 ms 0.0%	0 s	void* std::__1::__libcpp_operator_new<unsigned long>(unsigned long) [inlined] guitdemo

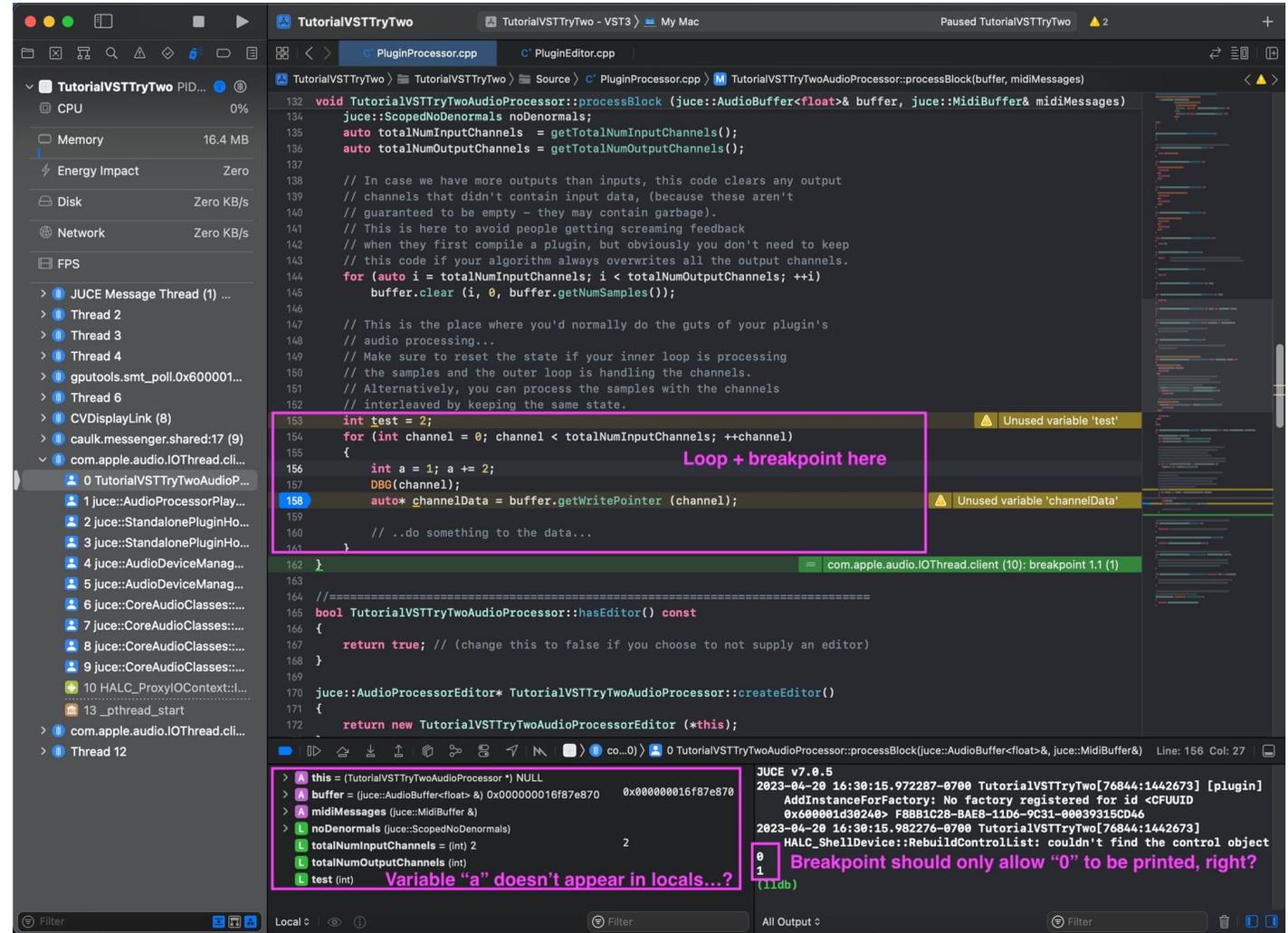
**Heaviest Stack Trace**

- 4601 guitdemo (2631)
- 3694 Main Thread 0xdc9c
- 3680 start
- 3572 main
- 3571 guit::GCore::start(guit::GWin...
- 3521 guit::GNatContext::startLoop()
- 1881 guit::GNatContext::waitEvent...
- 1863 SDL\_PumpEventsInternal
- 1836 Cocoa\_PumpEvents
- 1831 Cocoa\_PumpEventsUntilDate
- 1650 -[NSApplication(NSEvent)\_n...
- 1488 \_DPSNextEvent
- 1482 \_BlockUntilNextEventMatchi...
- 1482 ReceiveNextEventCommon
- 1468 RunCurrentEventLoopInMode
- 1357 CFRRunLoopRunSpecific
- 867 \_\_CFRunLoopDoObservers
- 863 \_\_CFRUNLOOP\_IS\_CALLING\_...
- 863 CA::Transaction::flush\_as\_ru...

# Debugage

## Debugueurs

- exécution pas à pas
- points d'arrêt
- contexte en cas de plantage



## Notes

- compiler avec option `-g` (ou équivalent)
- exécutable plus gros !

# Debogage

## Analyseurs statiques

- détection des **erreurs formelles** de programmation ou de conception

The screenshot shows a C++ IDE with a static analysis tool. The file being analyzed is `gstring.cpp`. The error message is: "8. Dereference of null pointer (loaded from variable 'pend')".

```
462 return num;
463 }
464 }*/
465
466 double str::toDouble(GString const& from, bool* ok) {
467     if (from.empty()
468         || (from.size()==1 && (from=="-" || from=="+" || from=="." || from==""))
469         || (from.size()==2 && (from=="-." || from=="+." || from=="+", " || from=="-,")))
470     {
471         if (ok) *ok = true;
472         return 0.f;
473     }
474
475     char point = std::use_facet<std::num_punct<char>>(std::cout.getloc()).decimal_point();
476     GString str(from.data());
477     for (auto& ch : str) {
478         if (ch == '.' || ch == ',') ch = point;
479     }
480
481     char* pend{};
482     double val = std::strtod(str.data(), nullptr);
483
484     if (ok) *ok = (pend != str.data() && *pend == 0);
485     return val;
486 }
```

Callout box content:

1. Assuming the condition is false
2. Assuming the condition is false
3. Assuming the condition...
4. Loop body skipped when range is empty
5. 'pend' initialized to a null pointer value
8. Dereference of null pointer (loaded from variable 'pend')

# Popularité

Sept  
2024

Sep 2024	Sep 2023	Change	Programming Language		Ratings	Change
1	1			Python	20.17%	+6.01%
2	3	↑		C++	10.75%	+0.09%
3	4	↑		Java	9.45%	-0.04%
4	2	↓		C	8.89%	-2.38%
5	5			C#	6.08%	-1.22%
6	6			JavaScript	3.92%	+0.62%
7	7			Visual Basic	2.70%	+0.48%
8	12	↑↑		Go	2.35%	+1.16%
9	10	↑		SQL	1.94%	+0.50%
10	11	↑		Fortran	1.78%	+0.49%
11	15	↑↑		Delphi/Object Pascal	1.77%	+0.75%
12	13	↑		MATLAB	1.47%	+0.28%
13	8	↓↓		PHP	1.46%	-0.09%
14	17	↑		Rust	1.32%	+0.35%
15	18	↑		R	1.20%	+0.23%
16	19	↑		Ruby	1.13%	+0.18%
17	14	↓		Scratch	1.11%	+0.03%
18	20	↑		Kotlin	1.10%	+0.20%