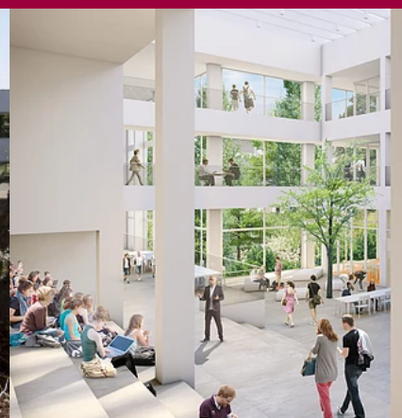
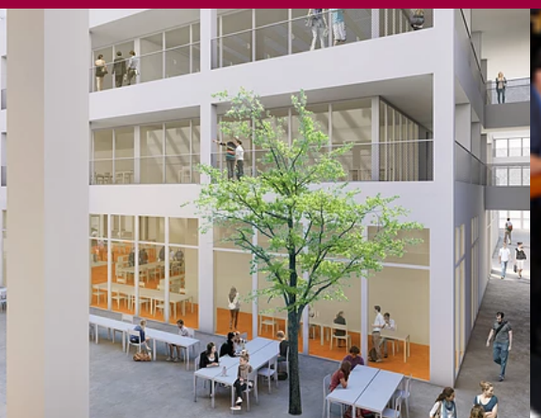
CentraleSupélec

Concepts des langages de programmation
Mise en œuvre en C/C++


Plan

- Rappels, généralités
- Approche impérative
- Approche fonctionnelle
- Approche objet
- Approche générique
- Compléments

Objectifs

- Présenter les différents concepts que l'on trouve dans les langages de programmation de troisième génération
- Comprendre les différences possibles de leur mise en œuvre selon les langages
- Apprendre C++ (et C par la même occasion)
- Être en mesure d'apprendre rapidement un nouveau langage de programmation

Connaissances supposées

- Maitrise, au moins partielle, du langage de programmation Python : SIP, CW, Algo
- Structures de données, algorithmes, complexité : Algo
- Architecture générale d'un ordinateur (processeur, mémoire centrale, périphériques…) : SIP
- Services rendus par un système d'exploitation : SIP
- Bonus
  - Concepts et avantages de l'approche objet, mise en œuvre dans un langage comme Java : GLOO, OOSE
  - Quelques bonnes pratiques de conception (patrons de conception, SOLID) : GLOO, OOSE

Plan

- Rappels, généralités
- Approche impérative
- Approche fonctionnelle
- Approche objet
- Approche générique
- Compléments

Points abordés

- Rappels sur l'exécution d'un programme par un processeur
- Les langages de programmation et leur utilisation
- Comparaison entre Python et C/C++
- Code source, fichiers, exécutable
- Les espaces de noms
- Entrées/sorties simples en C/C++

Machine de “Von Neumann”

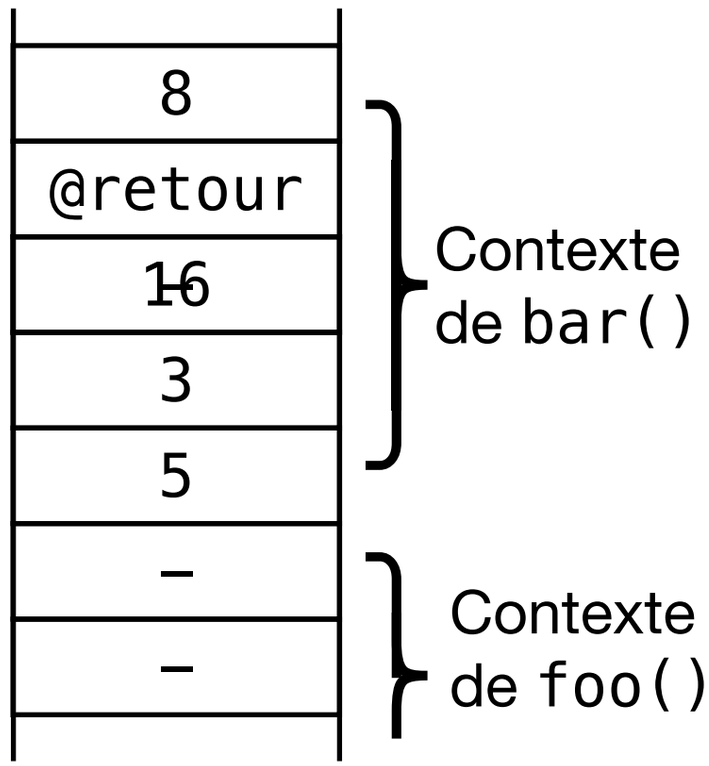
- ***Mémoire*** = tableau de mots (des octets maintenant)
- ***Adresse*** = indice dans le tableau
- Mots banalisés (instructions ou données ou adresses)
- La sémantique du contenu du mot est décidée par ce qu'en fait le processeur (instruction, adresse, nombre entier, nombre réel, caractère…)
- Pour modifier une donnée en mémoire, le processeur la charge dans l'un de ses registres, effectue la modification et remet la donnée modifiée en mémoire (load and store)
- Instruction Set Architecture (ISA) : codage des instructions, modes d’adressage

Appel de sous-programme

- Utilisation d'une ***pile***
  - arguments
  - place pour la valeur de retour
  - adresse de retour
- La pile stocke aussi les variables locales
  - Les registres servent à faire les calculs et à optimiser le temps d'exécution
- Les programmes doivent s'accorder sur une ABI (***Application Binary Interface***) pour fonctionner ensemble sur le système

```
def bar(a, b):
  c = a + b
  return c * 2

def foo():
  x = 3
  print(bar(x, x + 2))
  return bar(7, x + 5)
```

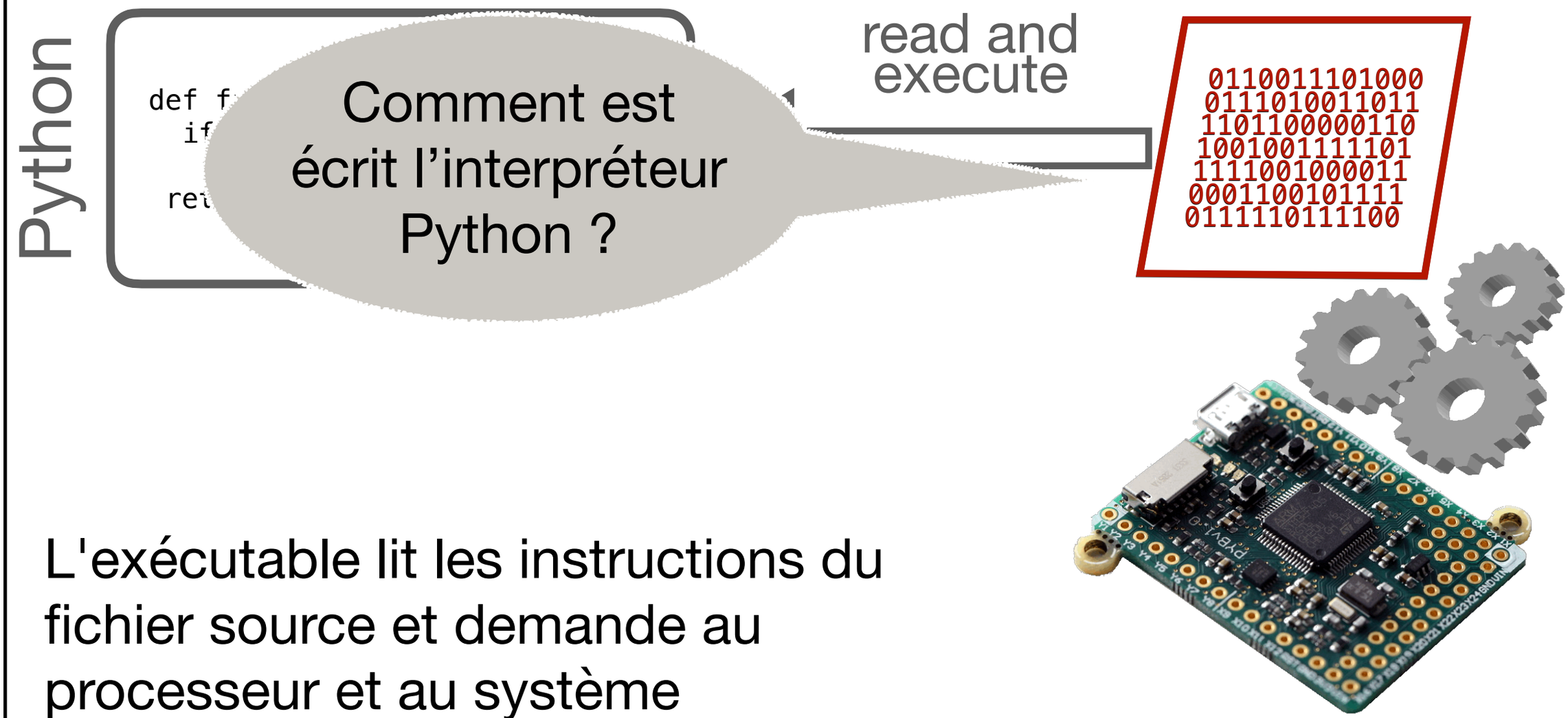


Exécution des langages

- Pas de règles générales
- C++ s'appuie sur un modèle théorique d'ordinateur pour définir sa ***sémantique*** :
  - *The semantic descriptions [...] define a parameterized nondeterministic abstract machine. This International Standard places no requirement on the structure of conforming implementations. [...] Rather, conforming implementations are required to emulate (only) the observable behavior of the abstract machine [...].*
- Contrairement à beaucoup de langages, certains comportements sont explicitement décrits comme *spécifiés par l'implémentation* (exemple : `sizeof( int )`) ou *non spécifiés* ou *non définis*.

Exécution par interprétation

Interpréteur



L'exécutable lit les instructions du fichier source et demande au processeur et au système d'exploitation de faire l'équivalent (y compris les entrées/sorties)

Exécution par compilation

Compilateur

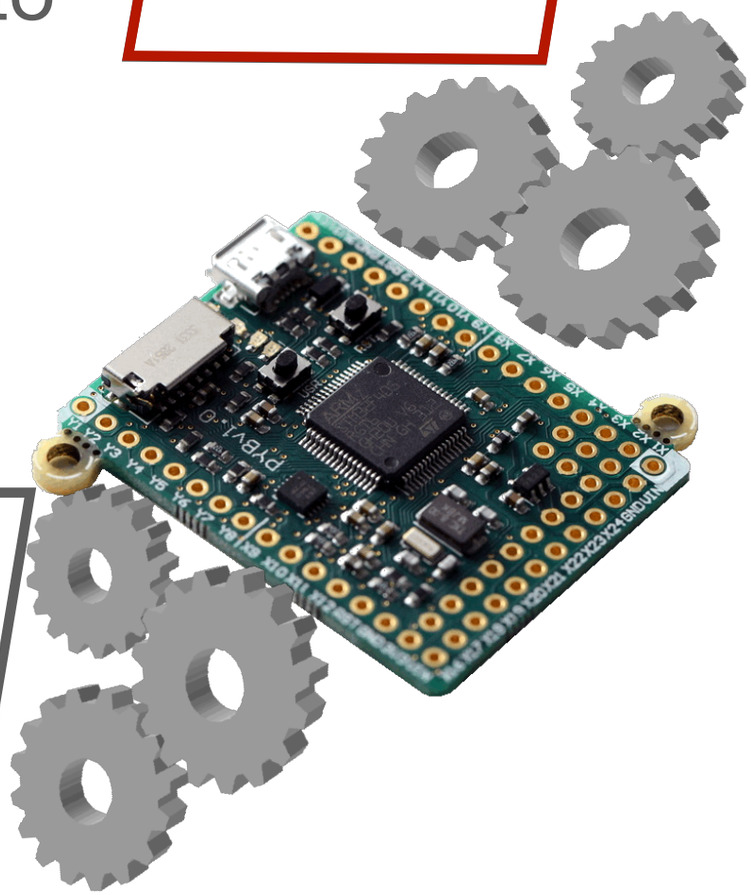
C, C++

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

compile to binary

```
0110011101000
0111010011011
1101100000110
1001001111101
1111001000011
0001100101111
0111110111100
```

```
1100110000110
1111000101010
0011000001101
0000111111111
0000110011111
0001100101111
1100101010101
```



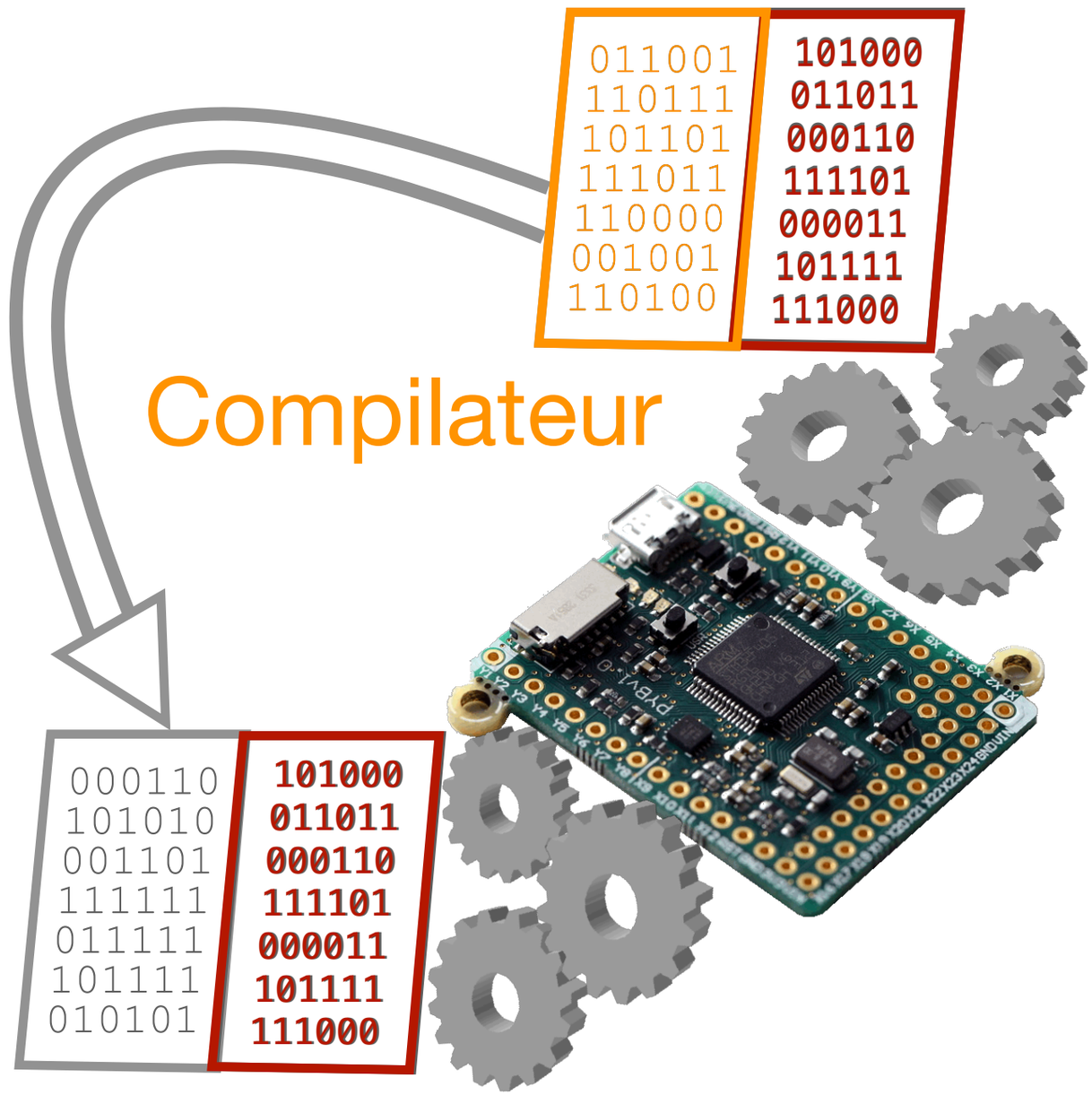
L'exécutable lit les instructions du fichier source et les traduit en code binaire équivalent qui peut ensuite être exécuté sur le processeur

Exécution sur une machine virtuelle

Java

```
public class Math {
  public static int factorial( int n ) {
    if( n == 0 )
      return 1;
    return n * factorial( n-1 );
  }
}
```

Machine virtuelle (interpréteur)



Les instructions du fichier source sont traduites en « bytecode » équivalent qui peut ensuite être interprété par la machine virtuelle

Compilations

- ***Compilation séparée*** : les différents fichiers sources sont compilés indépendamment

- ***Compilation croisée*** : le langage machine cible n'est pas le même que celui du compilateur

- ***Compilation à la volée*** (Just-In-Time), ***compilation anticipée*** (Ahead-Of-Time) : amélioration des performances de l'interprétation en traduisant, au moment ou avant l'exécution, le code source (ou le bytecode) vers le langage machine

Langages de programmation

- Les « ancêtres »
  - Fortran (impératif, 1954, John Backus)
  - LISP (fonctionnel, 1958, John McCarthy)
  - Algol (impératif, 1958, John Backus et Peter Naur)
  - Cobol (impératif, 1959)
- Les « remarquables »
  - Simula (objet, 1962, Kristen Nygaard)
  - Smalltalk (objet, dynamique, machine virtuelle, 1971, Alan Kay)
  - Haskell (fonctionnel, 1990)
- Liste d'environ 750 langages, un schéma des principaux langages, un autre plus détaillé

Utilisation des langages

- Tiobe index
- PYPL PopularitY of Programming Language
- GitHub Language Stats
- Stack Overflow Trends
- The RedMonk Programming Language Rankings
- IEEE Spectrum "The Top Programming Languages"
- Top 8 Most Demanded Programming Languages in 2022

Python

- Implémentation à partir de décembre 1989 par Guido van Rossum (« dictateur bienveillant à vie » du langage jusqu'en 2018)
- Initialement conçu pour l'utilisation d'un système d'exploitation distribué (Amoeba)
- 0.9.0 (1991), 1.0.0 (1994), 2.0.0 (10/2000), 3.0 (12/2008), 3.10.7 (09/2022)
- Implémentation de référence : CPython
- Multi-paradigme
- Typage dynamique
- Gestion automatique de la mémoire
- Python Software Foundation License

C

- Dennis Ritchie, 1972 (successeur du langage B, Ken Thomson)
- Associé à Unix, Bell Labs
- Programmation système (*assembleur de haut niveau*)
- K&R 1978, ANSI 1989, ISO 1990, 1999, 2011, 2018, 2023
- Impératif, modulaire, fonctionnel
- Préprocesseur, compilateur et éditeur de liens

C++

Pay Only For What You Use

- Bjarne Stroustrup
- Sur-ensemble du langage C
- Impératif, fonctionnel, objet, modulaire, génératif
- Préprocesseur, compilateur et éditeur de liens
- Initialement : compilateur AT&T de C++ vers C
- ISO : C++98, C++03, C++11, C++14, C++17, C++20
- https://github.com/cplusplus : ISO C++ Standards Committee (C++ Standard Draft Sources...)

Python vs C/C++ (1)

- Un commentaire ligne commence par `#`
- Une chaîne peut être utilisée pour les commentaires bloc

```
''' (exemple de
    commentaire
    inutile) '''
# on incrémente i
i = i + 1
```

- Un commentaire ligne commence par `//`
- Un commentaire bloc commence par `/*` et se termine par `*/`

```
/* (exemple de
    commentaire
    inutile) */
// on incrémente i
i = i + 1;
```

Python vs C/C++ (2)

- Une instruction est terminée par la ***fin de ligne***

```
r = 10
c = 2 * math.pi * r
s = math.pi * r ** 2
```

- Une instruction est terminée par le **;**

```
auto r = 10;
auto c =
  2 * M_PI * r;
auto s = M_PI
   * std::pow(r, 2);
```

auto uniquement C++

Python vs C/C++ (3)

- Utilisation du **:** pour introduire les instructions dépendantes

```
while n != 1 :
  if n % 2 == 0 :
    n = n / 2
  else :
    n = ( n * 3 ) + 1
```

- Utilisation des ***parenthèses*** autour des conditions

```
while( n != 1 )
  if( n % 2 == 0 )
    n = n / 2;
  else
    n = ( n * 3 ) + 1;
```

Python vs C/C++ (4)

- L'***indentation*** délimite les blocs (importance de l'indentation pour la correction)

```
if a > b :
  tmp = a
  a = b
  b = tmp
```

- Les ***accolades*** délimitent les blocs (importance de l'indentation pour la lisibilité)

```
if( a > b ) {
  tmp = a;
  a = b;
  b = tmp;
}
```

Python vs C/C++ (5)

- Les variables sont typées ***dynamiquement*** (à l'exécution)

```
x = 3
x = -2.8
x = "Bonjour"
x = True
```

- Les variables sont typées ***statiquement*** (dans le code source)

```
int    x  = 3;
double x1 = -2.8;
string x2 = "Bonjour";
auto   x3 = true;
```

`string` et **`auto`** uniquement C++

Python vs C/C++ (6)

- Les entiers sont de taille ***quelconque***, les flottants sont ***uniquement*** en double précision

```
i = 12345678901
```

- Les entiers sont de taille ***8***, ***16***, ***32*** ou ***64*** bits, les flottants sont sur ***32*** ou ***64*** bits

```
// Error : too large
int  i = 12345678901;
long l = 12345678901l;
```

« Hello World » en Python

Fichier source

```
# HelloSayer.py
import sys

def hello( world ):
  print( "Hello " + world );

if __name__ == "__main__":
  hello( sys.argv[1] )
```

Shell Unix

```
% python HelloSayer.py World
Hello World
%
```

« Hello World » en C

Fichier source

```c
// HelloSayer.c
#include <stdio.h>
void hello( char world[] ) {
  printf( "Hello %s\n", world );
}
int main( int argc, char * argv[] ) {
  hello( argv[1] );
  return 0;
}
```

Shell Unix

```
% cc HelloSayer.c -o HelloSayer
% ./HelloSayer World
Hello World
%
```

« Hello World » en C++

Fichier source

```
// HelloSayer.cpp
#include <iostream>
#include <string>
void hello( std::string world ) {
  std::cout << "Hello " << world << "\n";
}
int main( int argc, char * argv[] ) {
  hello( argv[1] );
  return 0;
}
```

Shell Unix

```
% c++ -std=c++20 HelloSayer.cpp -o HelloSayer
% ./HelloSayer World
Hello World
%
```

Démo

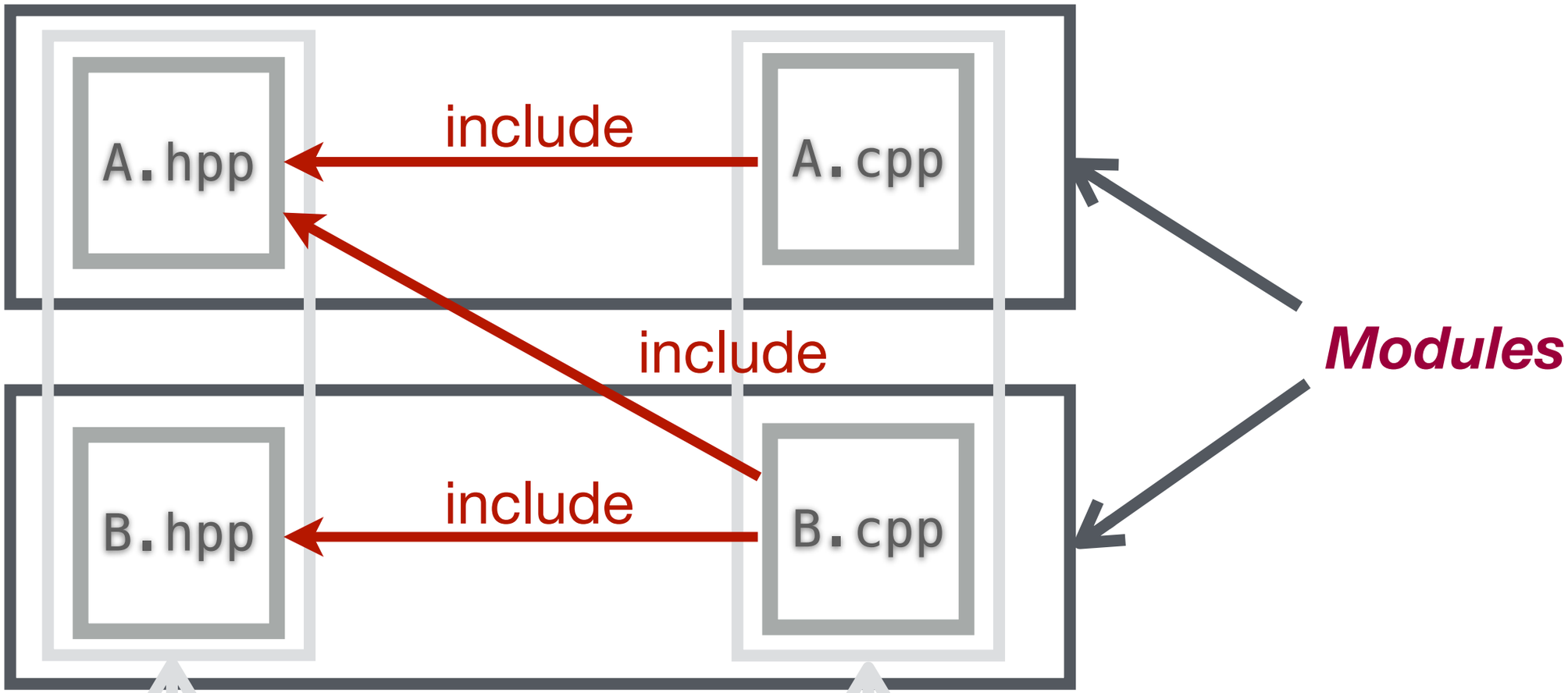
GitLab

cours/introduction/demo_1

Modules et paquetages

- Ces termes ont des sens différents selon les langages
  - Java
    - un ***module*** est une collection de paquetages (`requires`, `exports`)
    - un ***paquetage*** est un conteneur hiérarchique et un espace de noms
  - Python
    - un ***module*** est un conteneur d'éléments Python et un espace de noms
    - un ***paquet*** est un conteneur hiérarchique de modules
- Dans tous les cas, liens avec le système de fichiers

Les modules en C, C++



Fichiers d'entête

(définition des types,
déclaration des services)

Fichiers d'implémentation

(définition des services)

Préprocesseur (C, C++) (1)

- Étape de précompilation purement syntaxique
- Inclusion de fichiers d'entête

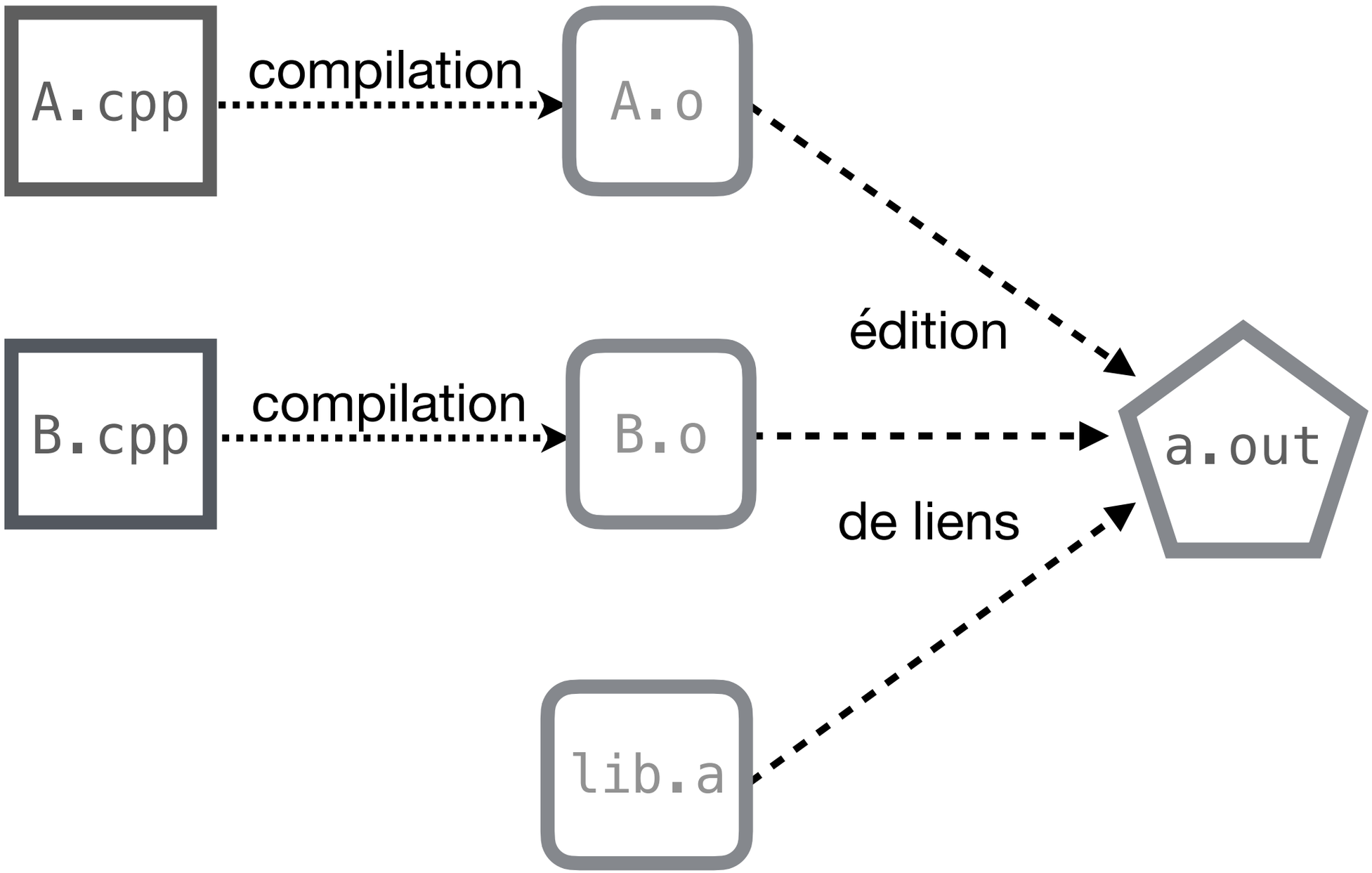
```
#include "A.hpp"

#include <stdio.h>
```

- Définition de constantes nommées en C (en C++, on utilise **const**)

```
#define TAILLE 10
```

Édition de liens



Démo

GitLab

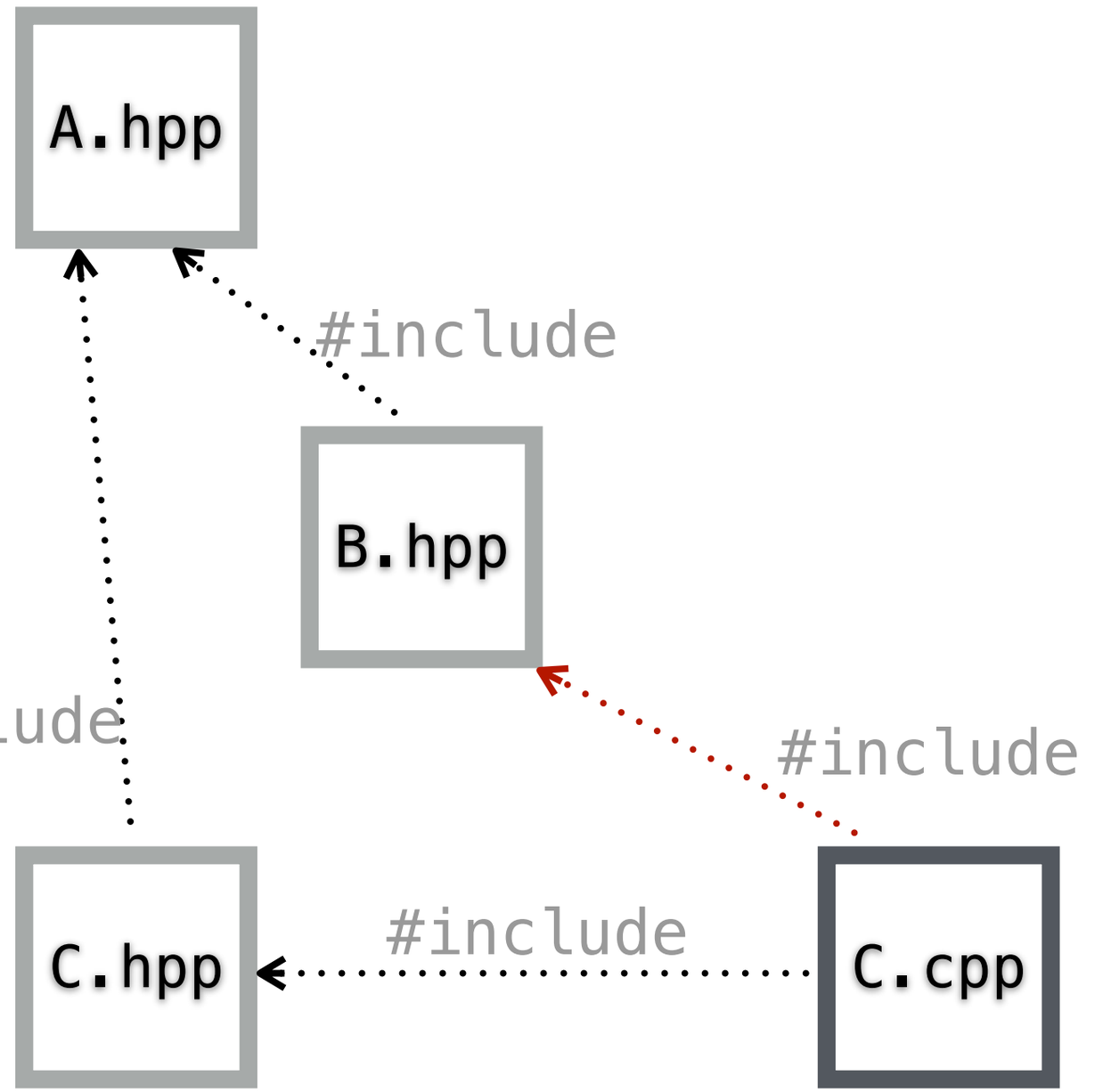
cours/introduction/demo_2

Problème de l'inclusion multiple

```
class A { };
```

```
class B { A a; };
```

```
class C { A a };
```




```
B b;
```

Préprocesseur (C, C++) (2)

- Protection contre l'inclusion multiple

```
// Fichier A.hpp
#ifndef A_HPP_INCLUDED
#define A_HPP_INCLUDED
// ...
#endif
```

- Compilation conditionnelle

```
#ifdef DEBUG
// -DDEBUG sur la ligne de commande
// pour activer cette partie du fichier
// ...
#endif
```

Démo

GitLab

cours/introduction/demo_3

Bonne pratique

- Un fichier d'entête doit toujours être protégé contre l'inclusion multiple
  - soit avec la méthode définissant un symbole du préprocesseur
  - soit en utilisant le pragma

Préprocesseur (C, C++) (3)

- Définition de ***macros*** (en C++, il est préférable d'utiliser la généricité)

```
#define MAX(a,b) ((a)>(b)?(a):(b))
```

- Opérateurs spécifiques

```
#define CONCAT(a,b) a##b
#define AS_STRING(x) #x
```

- Suppression d'une définition

```
#undef MAX
```

- Constantes prédéfinies

```
__FILE__
__LINE__
__DATE__
__TIME__
```

Espace de noms en C++ (1)

- Mot-clef **`namespace`**

```
// Avant C++17
// namespace fr { namespace centralesupelec {
namespace fr::centralesupelec {
    class Etudiant { /* ... */ };
}
```

- Pas de contrainte d'organisation des fichiers (directive `#include` du préprocesseur)

Espace de noms en C++ (2)

- Accès qualifié

```
fr::centralesupelec::Etudiant e;
```

- Par une instruction **using** sur le nom de l'espace

```
using namespace fr::centralesupelec;
Etudiant e;
```

- Par une instruction **using** sur un identificateur dans l'espace

```
using fr::centralesupelec::Etudiant;
Etudiant e;
```

- Possibilité d'alias

```
namespace cs = fr::centralesupelec;
cs::Etudiant e;
```

Espace de noms en C++ (3)

- Espace **std** réservé à la bibliothèque standard

```
#include <cstdio>

int main() {
  std::printf( "hello world\n" );
}
```

- Il est interdit de rajouter des définitions à l'espace **std** (mais on peut ajouter des spécialisations de fonctions pour de nouveaux types).
- C++20 ajoute le support des modules (**import**, **export**)

Bonne pratique

- Dans un fichier d'entête, au niveau principal :
  - Ne pas importer un espace de noms
  - Ne pas définir d'alias
- Peut s'envisager dans un contexte limité

- Plus de liberté dans un fichier d'implementation
  - Pour autant, je ne conseille pas un
    ```
    using namespace std;
    ```
    global

Entrées/sorties en C

- Fonctions de la bibliothèque standard

```c
#include <stdio.h>

int main() {
  int n;
  int nb_parsed = fscanf( stdin, "%d", &n );
  if( nb_parsed == 1 ) {
    fprintf( stdout,
             "carré de %d = %d\n",
             n, n * n );
  }
  return 0;
}
```

Entrées/sorties en C++

- Fonctions et objets de la bibliothèque standard

```
#include <iostream>

int main() {
    int n;
    std::cin >> n;
    if( std::cin.good() ) {
      std::cout << "carré de " << n
                << " = " << n * n
                << std::endl;
    }
    return 0;
}
```

- Utilisation de la ***surcharge des opérateurs*** << et >>
- Extensible aux types utilisateurs

Démo

GitLab

cours/introduction/demo_4