

Programmation multi-paradigmes en C++

Session #07 : Introduction au C++ moderne

Julien Deantoni, Thomas Soucheyre,
Stephane Janel, Ken-Patrick Lehrmann,
Ludovic Tessier, Leandro Fontoura Cupertino,
Kevin Yeung

Organisation de la seconde partie du cours

Contenu et évaluation

- 5 sessions de cours
- 6 sessions de TDs
- 1 évaluation sur table (1h, à la place du cours)
- 1 des TDs sera noté (lequel ? Surprise !)

Rendre les TDs dans la journée

Échange et communication

- Accès aux supports de cours et TDs: Moodle
<https://lms.univ-cotedazur.fr/2024/course/view.php?id=13795>
- Dépôt des TDs : Moodle
- Toute autre communication sur Slack :
[#prog-multiparadigme-si4](#)

Organisation de la seconde partie du cours

Enseignants: ingénieurs à Amadeus

- Ken-Patrick LEHRMANN
- Thomas SOUCHEYRE
- Stéphane JANEL
- Leandro FONTOURA CUPERTINO

Organisation de la seconde partie du cours

Planning prévisionnel

- Session 7: Introduction au C++ moderne 18 octobre 2024
- Session 8: *Move semantics & Rule of 3/5/0* 25 octobre 2024
- Session 9: Durée de vie des objets (RAII) 8 novembre 2024
- Session 10: Introduction au STL (*Standard Template Library*) 22 novembre 2024
- Session 11: Vers la programmation fonctionnelle en C++ 29 novembre 2024
- Session 12: Performance & Efficacité 6 décembre 2024
- Évaluation sur table 13 décembre 2024

Agenda

Session 7: Introduction au C++ moderne

01 Pourquoi le C++ aujourd’hui ?

02 Rappels sur C++

Pourquoi le C++ aujourd’hui ?

Pourquoi le C++ aujourd'hui ?

C++, langage multi-paradigme, axé sur la performance et la flexibilité

- Au départ, C + orienté objet + typage fort
- Il donne beaucoup de contrôle (ce qui le rend complexe).
- Approx. compatible avec C
- Approx. rétrocompatible (un programme écrit en C++98 devrait être facilement adaptable avec un compilateur C++23)
- C++ est libre de droit. La norme ISO est payante, mais les versions préliminaires sont disponibles gratuitement.
- Beaucoup d'évolutions depuis 10/15 ans, et toujours en cours !

Pourquoi le C++ aujourd'hui ?

Principes de conception du langage

Le langage suppose que le développeur sait ce qu'il fait, et lui laisse la possibilité de faire des erreurs...

... parce que ça lui donne aussi beaucoup de possibilité (de faire quelque chose d'intelligent)

... et aussi pour ne pas casser la rétrocompatibilité...

Pourquoi le C++ aujourd'hui ?

Principes de conception du langage

https://en.cppreference.com/w/cpp/language/Zero-overhead_principle

"Pas de coût additionnel" (Zero-overhead principle)

- Principe de conception du langage C++:
 - Ce qu'on n'utilise pas n'a pas de coût (en performance principalement)
 - Ce qu'on utilise est aussi efficace/performant que ce qu'on pourrait raisonnablement écrire soi-même.
- En gros, on n'ajoute pas une fonctionnalité dans C++ si elle impose un coût supplémentaire, en temps d'exécution ou en mémoire utilisée, plus grand que ce qu'un développeur ferait sans cette fonctionnalité.
- Potentiellement deux exceptions: RTTI et les exceptions...

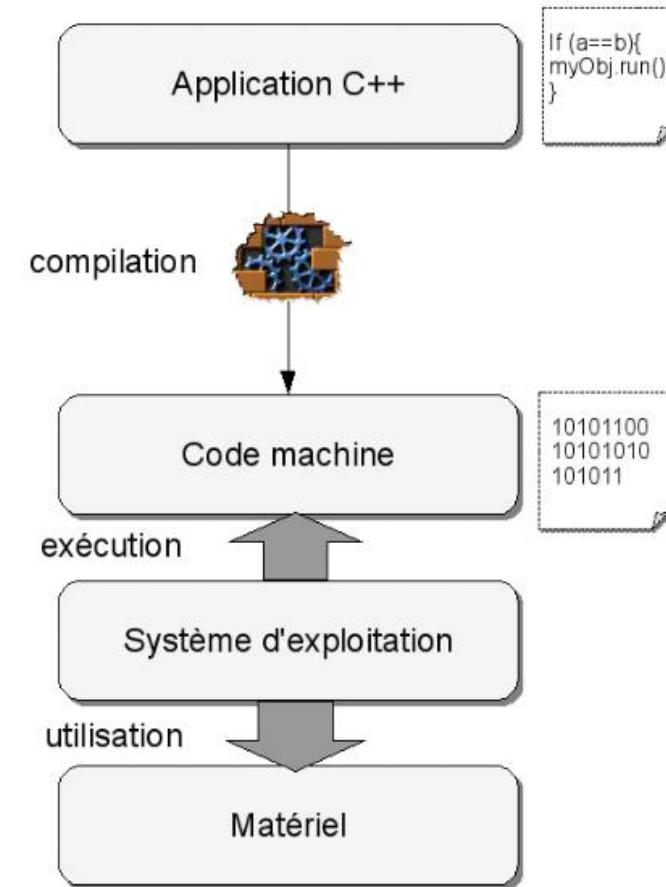
Pourquoi le C++ aujourd'hui ?

C++ est compilé

Plusieurs compilateurs (libres ou proprio)

- GCC (g++)
- LLVM/Clang
- Microsoft Visual C++
- Intel C++ compiler (maintenant dérivé de LLVM)
- ...

Les options de compilation ont un impact sur la performance, le débogage, etc.

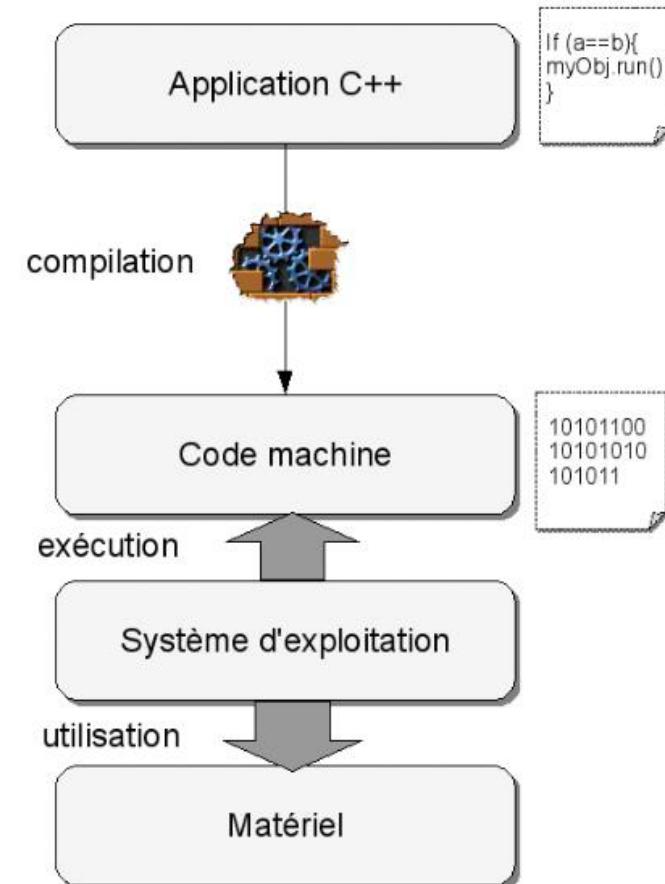


Pourquoi le C++ aujourd'hui ?

C++ est compilé

Des outils "moteur de production" (build automation/build system) automatisent ça.

- Make (et ses variantes)
- Autotools (vieillissant mais encore très utilisé)
- CMake (~ standard de fait)
- Meson (future standard de fait ?)
- (Buck, Bazel)



Pourquoi le C++ aujourd’hui ?

Historique très rapide

1979: Bjarne Stroustrup (at AT&T), C with Classes (C avec classes)

- Classes, classes dérivées, typage fort, inlining

1985: Première implémentation commerciale de C++

- Fonctions virtuelles, surcharge, références, allocation typée (new/delete)

1998: Première norme ISO (C++98)

- Largement compatible avec C
- Templates (généricité)
- STL

2003: C++03

- Bug fix release

Pourquoi le C++ aujourd'hui ?

Historique très rapide

C++ moderne

2011: C++11 (aka C++0x)

- rvalue reference et move constructor
- constexpr
- auto (inférence de type)
- Range-based for loop
- Lambda
- static_assert
- STL: threads, tuples, unique_ptr, small string optimization, plein d'algorithmes

2014: C++14

- auto
- constexpr

2017: C++17

- auto
- constexpr
- Structured binding declaration (auto [a, b] = getTwoReturnValues();)
- STL: filesystem, variant (type somme)

Pourquoi le C++ aujourd’hui ?

Historique très rapide

C++ moderne

2020: C++20

- Ranges
- `<=>`
- concept
- module
- span

2023: C++23

- `std::print`
- `std::expected`
- `std::mdspan`
- `constexpr`

Pourquoi le C++ aujourd'hui ?

C++ moderne

Depuis C++11.

- Ajout de nombreuses abstractions pour remplacer, clarifier et rendre plus sûres les constructions héritées de C (avec un coût nul à l'exécution)
- Utiliser le typage fort de C++, sans l'abuser ("ne pas faire de *cast*"). Le système de type apporte des garanties vérifiées à la compilation.

Par exemple:

- manipulation de pointeur -> smart ptr, containers, moins de comportements indéfinis (« *out of bound* »), voire impossible si on utilise les bonnes interfaces (at())
- Macro -> fonction *inline* (type safe), *template*, *meta-programming*
- Boucles explicites -> *range-based for loop*, *algorithm* (pas d'*off-by-one*, l'intention du programmeur est plus claire)

Pourquoi le C++ aujourd'hui ?

C++ moderne

Dans le même temps, les outils et la communauté se sont beaucoup améliorés.

- Trois gros compilateurs: GCC, LLVM/clang, Microsoft Visual C++. Il y a en d'autres, souvent dérivés de *clang*.
- Les compilateurs fournissent de bien meilleurs diagnostics (message d'erreur ou d'avertissement).
- De nombreux outils pour aider les développeurs:
 - Les sanitizers permettent de détecter quasiment tous les bugs à l'exécution, sans faux positif
 - Des outils de formatage de code (*clang-format*)
 - Des outils d'analyse statique (*clang-tidy*, *cppcheck*, *sonarqube*)
 - Compiler explorer <https://godbolt.org/>

Pourquoi le C++ aujourd’hui ?

C++ moderne

De nombreuses ressources sont accessibles en ligne.

- C++ core guidelines (collections de bonnes pratiques et conseils à suivre)
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>
- documentation du langage et de la bibliothèque standard: « C++ reference »
<https://en.cppreference.com/w/cpp>
- vidéos des conférences CppCon (et autres) disponibles sur Youtube

Pourquoi le C++ aujourd’hui ?

TIOBE Index for October 2024 – TOP 10

<https://www.tiobe.com/tiobe-index/>

Oct 2024	Oct 2023	Change	Programming Language	Ratings	Change
1	1		 Python	21.90%	+7.08%
2	3		 C++	11.60%	+0.93%
3	4		 Java	10.51%	+1.59%
4	2		 C	8.38%	-3.70%
5	5		 C#	5.62%	-2.09%
6	6		 JavaScript	3.54%	+0.64%
7	7		 Visual Basic	2.35%	+0.22%
8	11		 Go	2.02%	+0.65%
9	16		 Fortran	1.80%	+0.78%
10	13		 Delphi/Object Pascal	1.68%	+0.38%

Très utilisé

Libre

Performant

Pourquoi le C++ aujourd’hui ?

Energy efficiency of Programming Languages

Table 4. Normalized global results for Energy, Time, and Memory

Total		
	Energy	Time
	Mb	
(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	
(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	
(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	

Efficacité énergétique

Rapidité

Efficacité de l'utilisation Mémoire

SLE'17, October 23–24, 2017, Vancouver, Canada:

<https://sites.google.com/view/energy-efficiency-languages/results?authuser=0>

Original research paper:

<https://greenlab.di.uminho.pt/wp-content/uploads/2017/10/sleFinal.pdf>

2.

Rappels sur C++

Rappel: Définition/Déclaration

Un programme C++ est constitué de différentes entités:

- variables, fonctions, types, espaces de noms...
- .

Elles doivent être déclarées avant d'être utilisées.
Chaque sorte d'entité est déclarée d'une manière spécifique

// Exemples:

```
int f(int);  
  
extern int a;  
  
class C;
```

Une définition est une déclaration qui implémente complètement une entité, typiquement accompagnée d'une assignation ou d'un corps

// Exemples:

```
int f() { return 4; }  
  
int a = 42;  
  
class C {};
```

<https://en.cppreference.com/w/cpp/language/declarations>

Rappel: One definition rule

Les classes/structures et fonctions non-*inline* ne peuvent pas avoir plus d'une définition dans un programme (sous peine de comportement indéfini).

https://en.wikipedia.org/wiki/One_Definition_Rule

<https://en.cppreference.com/w/cpp/language/definition>



**Attention aux définitions dans les fichiers
d'entête (.h/.hpp/.hxx/.hh/.H/...)**

Rappel: Classes en C++

Classes

```
class C{
public:
    C();
    explicit C(int n) : _n(n) {}

    int compute() const;

private:
    int _n;
    int privateFunction();

protected:
    bool operator==(const C&) const = default;
};
```

Rappel: Classes en C++

Destructeur

Objectif: libérer les ressources que la classe a pu acquérir pendant sa durée de vie.

Il est appelé automatiquement à la fin de la vie d'un objet.

```
class C{
public:
    C();
    ~C(); // Destructeur

private:
    int *_n;
    int privateFunction();
};
```

Rappel: Classes en C++

Copie

Constructeur par copie&opérateur d'assignation
par copie : copie l'argument sans le modifier

```
class C{
public:
    C(const C &other) : _n(other._n) {}
    C &operator=(const C &other) {
        if (this != &other) {
            _n = other._n;
        }
        return *this;
    }

private:
    int _n;
};
```

Rappel: Classes en C++

Définitions implicites

La plupart du temps, le compilateur génère une version implicite des destructeurs, constructeurs et opérateurs par copie, avec le bon comportement si la classe ne gère pas de ressource manuellement.

```
class C{  
public:  
  
private:  
    int _n = 0;  
};
```

Rappel: Classes en C++

Définitions implicites



**Si la classe gère manuellement des ressources,
le comportement est mauvais !**

```
class C{
public:
    C() : _n(new int(42)) {}

private:
    int *_n;
};
```

```
C c1;
// imaginons que C alloue une zone mémoire dans _n;
// la mémoire n'est pas libérée par le destructeur par défaut
```

```
C c2 = c1; // simple copie de pointeur pour _n, probablement pas le comportement voulu
```

Rappel: Classes en C++

Exceptions



Gestions des erreurs **inattendues**
(pas pour autre chose !)

- Une exception interrompt l'exécution normale du programme,
- et remonte les appels de fonctions jusqu'à être attrapée dans un « catch ».

Pas d'exception dans les destructeurs (si ça arrive: std::terminate)
(en revanche c'est une bonne pratique dans les constructeurs)

```
int foo(int a) {
    if (a == 42) {
        throw std::invalid_argument("blabla");
    }
}
int main() {
    try {
        foo(42);
    } catch (std::invalid_argument &e) {
    }
}
```

- Oblige à gérer les erreurs (si pas attrapée: std::terminate)
- Permet de gérer les erreurs à un endroit où ça a du sens
- Sépare la gestion des erreurs

Fin de la session