# Programmer en C++

Martin Quinson



# Remarque

- On va tenter d'éviter le catalogue de syntaxe, mais il en faut un peu. On verra surtout ça en TP.
- Introduire le fait qu'il y aura des rendus de cours comme au S1 en arcsys.

# I) Introduction au langage C++

- Objectif du cours: apprendre un nouveau langage (C++), très riche et quasi incontournable
  - Pas objectif : apprendre par coeur la syntaxe ou lib standard, donc pas de cours de référence
  - Objectif: vous rendre adaptable à n'importe quelle équipe (pas de former des mozarts du code)
  - Objectif: comprendre les différences entre langages de prog, avec les avantages relatifs de chacun
- Syllabus en pratique (chaque semaine de cours placée sur 2 semaines calendaires)
  - Cette semaine : vous rendre opérationnels pour les TP le plus vite possible, Better C
  - Topic 2: programmation orientée objet
  - Topic 3: héritage et classes
  - Topic 4: Gestion mémoire en C++ (et C++ moderne)

  - Topic 5: programmation fonctionnelle et générique en C++
  - Semaine du 28 mars: bloquée pour un gros projet à faire par 4
  - Examen final: sur table, avec une feuille de pompe manuscrite autorisée

## I.1) Histoire du C++

- L'auteur historique du C++: Bjarne Stroustrup, doctorant danois à l'université de Cambrige (UK).
  - Durant sa thèse (1974-1979), il a écrit un simulateur pour étudier les systèmes distribués.
- Première tentative en Simula. C'est l'un des premiers langages orientés objet.
  - Très agréable : Le code était plaisant à écrire et clair à lire
  - Mais performances déplorables : 80% du temps en garbage collection même si y'avait rien à faire
- Seconde tentative dans un langage rapide de l'époque: BCPL (ancêtre du B, qui est l'ancêtre du C).
  - Horrible, aucune aide du langage, tout à la main. L'enfer.
- Après sa thèse, il a été embauché au Bell Labs pour inventer un nouveau langage "C with classes"
  - Constructions de haut niveau, performances de bas niveau
- Renommé en C++ en 1983; Publication décrivant le langage en 1985, et succès immédiat.
  - On disait que le nombre de programmeurs doublait tous les 7 mois, 3M en 2007.
  - Maintenant, le langage est standardisé par un comité adéquat.

## I.2) Place du C++ dans le monde

- "C++ makes programming more enjoyable for serious programmers", Stroustup 1994.
- Très utilisé: toutes les entreprises connues.
  - TIOBE janvier 2020: Mesure de popularité (hit google, stack overflow, youtube, wikipedia, amazon). Pas mesure quantité code ou qualité langage
    * 2020: #1 Java 16.9% ; #2 C 15.8%; #3 Python 9.7%; #4 C++ 5.6%; #5 C# 5.3%
    * 2021: #1 Python 13.6%; #2 C 12.4%; #3 Java 10.7; #4 C++ 8.3%; #5 C# 5.7%
    * C se maintient grâce à Linux et à l'IOT
  - Tous les browsers: Chromium (et donc Edge)/Firefox/Safari/Opera
  - MS office, libreoffice, JVM, VLC, Windows OS, la plupart des jeux. Le code embarqué dans Curiosity, dans un F35.

## I.3) Caractéristiques du langage

- **Langage généraliste**, par opposition à Matlab dédié aux maths. Il n'offre pas de solution magique pour un domaine donné, mais il n'existe pas de domaine où ce serait une très mauvaise idée de faire du C++. Bon, sauf peut-être là où il faut coder vite fait un ptit machin jetable.
  - De plus, le langage cherche à résoudre les pbs que les programmeurs ont dans leurs vrais projets.
- **Langage compilé** et non interprété. Cela permet au compilo d'optimiser le code produit.
  - De plus, on cherche à détecter un max de problèmes dès la compilation
- **Typé statiquement**: il faut déclarer le type des variables à l'avance, et spécifier explicitement quand on veut convertir. Important pour détecter les problèmes à la compilation
- **Multi-paradigme**: On peut faire ce qu'on veut en C++
  - *Impératif:* sorte de prolongement du C, comblant les défauts du langage de base

  - *Orienté objet:* (comme simula) élaboration de hiérarchies de classes complexes, cachées derrière des interfaces simples
  - *Programmation générique:* écriture de code réutilisable dans de nombreux contextes, grâce aux templates qui sont une amélioration des macro préprocesseur
  - support limité pour la *programmation d'ordre supérieur:* construction et manipulation d'autres fonctions au runtime
  - C'est une richesse: ne pas forcer une façon de faire aux utilisateurs est un objectif fort du langage
  - Mais ça rend le langage complexe, et donc plus complexe à apprendre. Perso, après 5 ans de C++ (et 20 de C), je ne pense tjs pas maîtriser complétement le langage (et encore moins la lib standard)
- **Langage de niveau intermédiaire**: combine les perfs des langages bas niveau avec les constructions des langages haut niveau. Twists résultants:
  - temps de compilation assez important
  - message d'erreur ... absolument incroyables parfois.
- **Langage qui évolue**: des standards fréquents pour améliorer le langage.
  - C++98, C++11, C++14, C++17, C++20.
  - C++11 est une révolution, les autres sont des changements plus avancés (pas pour débutants).
  - Tentatives de maintenir la compatibilité (par opposition au python3). Est-ce une bonne idée ?

## I.4) Le C++ par rapport à d'autres langages

- Une famille de langages de prog règne sur le monde: ASM -> C -> C++ -> Java -> C# -> Rust
  - Python vient de Java/C# mais aussi de langages de script comme Perl
- Il est intéressant de regarder les fonctionnalités de ces langages qui se sont propagées et celles qui ont été arrêtées / introduites.
- Ca fait que chaque langage a une sorte de personnalité propre, qu'il faut comprendre pour espérer le maîtriser

### I.4.a) Avantages et limites du C

- Inventé en 1972 comme une sorte d'assembleur portable: Rapide, simple, portable.
- Mais la simplicité est une force comme une limite: Pas de lib standard, pas d'aide à l'abstraction.
- Le préprocesseur est une bonne idée, mais piégeux car pas inclu dans le langage. Comme un langage dans le langage.
- C++ est (presque) un sur-ensemble du C.
  - Rares sont les bouts de C qui ne compilent pas en C++. Souvent, c'était une mauvaise idée de l'écrire comme ça.
  - Mais il y a souvent une façon C et une façon C++. Par exemple, il ne reste que ifdef dans le préprocesseur
- Point commun : le compilateur fait confiance à l'humain, et produit des messages d'erreurs abscons

### I.4.b) Avantages et limites du Java

- Java inventé pour résoudre les pbs de portabilité et de sécurité (à l'époque pour le web)
- Très fortement inspiré du C++ que les développeurs connaissaient à l'époque, pour prendre le marché. Et les dev de la JVM l'ont fait en C++, aussi.

- C++ a confiance en vous et vous donne les pleins pouvoirs. Ca rend le langage plus compliqué à bien utiliser.
  - Souvent C++ offre des constructions par défaut pour les choix par défaut, mais le programmeur peut prendre le contrôle des choses. Par exemple, gestion mémoire (allocators) dans les collections d'objets
- Java ne fait pas confiance au programmeur: pose des limites sur ce qu'on peut écrire, et masque le matériel
  - Plus de pointeurs pour ne plus avoir de buffer overflows
  - Très peu de "undefined behavior" => facile à apprendre
  - Compilateur parano => code dangereux ou surprenant ne compile pas (dead code)

# II) C++ en pratique

## II.1) Outils à installer en pratique

(pour la prochaine fois):

- Un compilateur, de préférence clang++ de la famille LLVM. g++ peut faire l'affaire, mais la famille LLVM semble être l'avenir de l'humanité.
- Un IDE: QtCreator (celui que j'utilise, perso) ou VS Code/Codium (celui qu'utilise Solène en TP)
- Un système de construction: cmake. On va l'utiliser pour exprimer les fichiers composant le projet, afin qu'ils soient bien recompilés.
- Des outils de debug: gdb et valgrind.
- Des outils d'analyse de la qualité du code: clang-tidy et cpplint (analyse du code), clang-tidy (reindent), scan-build (analyse statique)
- Le site `cppreference.com`: site de référence contenant toute la doc du langage. Incontournable même si c'est assez indigeste au début, malgré les qques exemples.

## II.2) Le type std::string

- Les chaînes étaient particulièrement pénibles en C avec `char*`, c'est fini en C++
- Voir le code sur le document associé pour la création, l'assignement et le retour au C
- Opérations faciles: concaténation, empty(), test d'égalité entre chaines (!), comparaison avec < (ordre ascii)
- Conversion string->nombre: `s1.stoi()` `stol()` `stof()`; Conversion nombre->string: `std::to_string(32)`
- Bcp d'autres opérations: `s3.substr(pos, count)`, `push_back()`, `c_str()`

## II.3) Les flux (stream)

### II.3.a) Adieu printf et scanf

- Cf le code du document associé
- Les flux font (1) conversion type variable <-> chaîne caractères (2) interaction avec le clavier ou l'écran
  - On peut ajouter des convertisseurs, comme si on ajoutait %Q dans printf pour une nouvelle structure
- Il y a aussi `std::cerr` pour stderr et `std::clog` pour la sortie d'erreur bufferisée
- En pratique, on peut écrire `"\n"` à place de `std::endl`, mais il y a beaucoup d'autres modificateurs de flux pour faire des sorties formatées.

  - std::setw(14) précise un champ de largeur fixe (14 caractères)
  - std::left/right précisent l'alignement
  - setfill(0) précise de compléter avec des 0
  - boolalpha: écrit le booleen qui vient ensuite sous la forme "true" ou "false"
  - hex, oct, dec: change la base des entiers (en in ou out)
- En lecture, ca tokenize sur un ou plusieurs espaces blancs

### II.3.b) Lire/écrire dans un fichier

- Cf le doc joint, garanti sans magie dedans.

### II.3.c) Lire/écrire dans une chaine de caractères

- `include <sstream>`, `std::istringstream` en input, `std::istringstream` en output.
- Contenu initial en paramètre à la création (du constructeur)

### II.3.d) Flux et gestion d'erreurs

- La gestion des erreurs est assez complexe et error prone. Le plus simple est d'en rester à `std::getline` comme dans l'exemple. Détail pas vu en cours:
  - `good()` si prêt pour la lecture; `bad()` problème insurmontable; `eof()`; `fail()` s'il y a eu une erreur (par exemple fin de fichier)
  - contre-intuitif: G+B = problème de typage; G+F = EOF. En pratique, on regarde rarement autre chose que F et E. Le mieux est de juste utiliser la valeur booleen de stream: `if (not stream) { Erreur }`
  - Attention, si y'a eu une erreur, le contenu du buffer est assez difficile à prédire / corriger. `std::getline` est plus sûr (mais faut pas mélanger lectures directes et getline car lectures directes passent les espaces avant la donnée donc elles laissent le whitespace terminant leur donnée dans le flux)
- Il y a énormément de méthodes définies sur les différents types de flux, cf cppreference en cas de besoin

## II.4) Nouvelles constructions du langage

- pas mal de choses qu'on ne verra pas tout de suite (programmation OOP, programmation générique)

### II.4.a) Les namespaces

- Une bonne façon de ranger ses identifiants pour éviter les name clashes.
- Par exemple, std:: est le namespace
- Déclaration avec `namespace blabla { ... }` et on peut imbriquer autant qu'on veut.
- On peut charger un namespace dans l'espace global avec `using namespace blabla;` en haut du code, mais c'est pas forcément conseillé car ça casse la séparation

### II.4.b) Surcharge de fonctions: plusieurs de même nom

- On a le droit d'avoir plusieurs fonctions de même nom, différenciées par le type des paramètres
- On peut aussi surcharger les opérateurs, puisque `+` ou `<` dont des fonctions définies par défaut...

### II.4.c) Passage de paramètres

- Comme en C, on peut faire du passage de paramètre **par valeur** ou **par adresse** (cf. l'exemple)
  - Si on veut, on peut préciser que le **pointeur est constant**: on passe par adresse mais le receveur n'a pas le droit de modifier (erreur de compil)

```
void square (const int* x) {
  *x = (*x) * (*x);  /* compile-time error */
}
```

- C++ introduit une 3ieme façon: passage de **paramètre par référence**. Comme utiliser un pointeur, mais sans le sucre syntaxique
  - Cf l'exemple: dans la fonction on l'utilise comme un truc copié, sauf qu'en fait ce n'est pas copié en dessous
- **Paramètres par défaut**: valeur donnée aux derniers paramètres de la fonction. Si omis à l'appel, la valeur par défaut est utilisée
  - Faut évidement pas que ce soit ambigu avec des fonctions de meme nom et même prototpye

### II.4.d) Itérateurs

- Notion un peu abstraite pour l'instant de *collection* de données en C++. Ca a donné les conteneurs en Java
  - Les itérateurs permettent d'écrire des algorithmes génériques (qui fonctionnent pour les double, les int ou les structures de l'utilisateur)
- Notion d'itérateur très complete et donc assez complexe en C++
  - Différents types d'itérateurs (forward, random access), et chaque collection en défini certains.

### II.4.e) Boucles étendues

- En pratique, on peut écrire des choses comme la ligne 7 de l'exemple "getline sur les fichiers"
  - `for (variable : collection)` la variable prend successivement toutes les valeurs de la collection

### II.4.f) `const` comme un meilleur `#define`

- `const` existe en C, mais le compilo n'insiste pas trop dessus. En C++, c'est plus rigide (pour s'approcher de la prog fonctionnelle?)
- `const int width = 512;` préférable à `#define WIDTH 512` car le compilo est au courant

## II.5) La bibliothèque standard (STL)

- En C, on n'avait rien. En C++, on a tout.

### II.5.a) Conteneurs

- Conteneurs
  - Sequence: vector: vecteur contigu dynamique; deque: liste doublement chainée; ...
  - Conteneurs associatifs: set: ensemble d'éléments; map: clé -> valeur
  - Conteneurs non-triés: `unordered_set`; `unordered_map`

- Voir l'exemple de la feuille sur le vecteur et la boucle étendue
- Opérations vector: `empty()`, `size()`, `push_back()`, `pop_back()`, `clear()`, `at(i)` vérifie les bornes
- La bible est `http://cppreference.com`

### II.5.b) Itérateurs et algorithmes

- la STL a été pensée pour rendre possible l'écriture d'algorithms génériques
- Toutes les collections définissent des itérateurs qui rendent applicables des algo génériques (Standard Templating Library)
  - D'où la facon de décrire les collections, la notion d'itérateur, leur usage généralisé
- Algorithmes définis par la STL: `std::sort(vect.begin(), vect.end());`

### II.5.c) Le reste de la STL

- Et bien plus encore (allocateurs, foncteurs, threads et concurrence). Et je ne parle pas de C++20 pour l'instant.
- Et si la STL ne suffit pas, il y a boost. Sorte d'incubateur d'idées à standardiser. L'API change parfois.

## II.6) Programmation générique

- Mot clé `auto` pour quand le compilateur connait le type `for (auto i: vect)`. Faut que le type soit trivialement déductible: interdit dans les paramètres de fonction.
- Pour faire des fonctions génériques, il faut utiliser des templates, qui sont des macro préprocesseurs en mieux (car connues du compilo).
- Ca reste quand même du cherche/remplace assez troublant. Il est possible que la template compile, mais pas son instanciation...

```
template <typename T>
T min(T a, T b) {
  return (a < b) ? a : b;
}

int a = 3, b = 6, c = min<int>(a, b);
```

## II.7) Autres petites nouveautés du langage

- Commentaires avec *// en plus de /* ... ** (le premier n'était pas compris dans les vieux C), et plus de liberté sur où déclarer les variables
- Les booléens: `bool done = true; // or false`
- la valeur nullptr à la place de NULL (qui était une macro définie à 0), pour donner une valeur correctement typée au compilateur.

## II.8) Idée de projet

- jeu snake dans le livre de stanford

# III) Paradigmes de programmation en C++

- C++ a été inventé pour apporter la POO au C très procédural dans l'ame (la première version de C++ était nommée "C with class")
  - les versions modernes de C++ (C++11, et surtout C++20) ont ajouté un peu de programmation fonctionnelle au C++
- Mais de quoi on parle en fait ? Retour sur les paradigmes de programmation classiques
  - C'est un sujet de troll sans fin, et les définitions qui suivent sont discutables ... mais on est lundi.
  - Formellement, il n'y a pas de paradigme plus expressif, et c'est une question de style, de goûts.
  - Les différences portent sur la façon de découper les données et opérations, leur organisation, ce sur quoi on se focalise, etc.

## III.1) Paradigmes de programmation

- **Programmation impérative vs. déclarative**: Organisation des opérations
  - En impérative, on dit quoi faire à l'ordi, pas à pas
  - En déclarative, on donne des éléments de réponses à l'ordi, puis on lui donne un objectif et il se débrouille
- **Programmation procédurale** (comme en C par exemple)
  - Modèle impératif où les traitements sont découpés en procédures ayant des effets de bord. Les données sont des globales modifiées de partout.
  - Modèle assez basique. Souvent obtenu quand on programme sans faire attention, et ça évolue vers un vilain code spagetti impossible à faire évoluer
  - Avantage: proche de la machine donc efficace, populaire.
  - Désavantage:
    - Difficile d'abstraire quoi que ce soit: puisque tout le monde peut modifier les globales, il faut considérer chaque bout à chaque instant
    - Effets de bord => Preuves et debug très difficiles; ordre des opérations important
- **Programmation logique**: l'autre extrêmité.
  - Modèle déclaratif où l'on donne des faits de base, des règles de déduction et on donne un objectif à l'ordinateur, qui doit le montrer.
  - Exemple en Prolog (sur la feuille imprimée): la famille
    - On donne des faits de base, des règles de déduction, puis on interroge la base de savoir.
    - C'est la base des systèmes experts (version fin 20ieme siècle), même si maintenant le deep learning nous épargne de comprendre pourquoi (pun intended)
    - Cela peut faire de vrais systèmes (comme les premières versions de l'interpréteur Erlang)
  - Exemple en TLA+: les récipients
    - On donne des variables avec leur ensemble de valeurs; un état initial et un ensemble de transitions possibles; des propriétés, et le système cherche si un contre-exemple est atteignable
    - C'est un langage logique, pas un langage de programmation
    - Amazon maintient des spécifications TLA+ de ses principaux modules, et trouve des bugs
  - *Inconvénients:* Difficile à utiliser, surtout pour écrire des programmes génériques
  - *Avantages:* La preuve est triviale
- **Programation fonctionnelle**: Déclaratif inspiré des fonctions mathématiques (focus sur traitements)

  - Déclaratif car on ne liste pas les opérations à réaliser. On défini le résultat comme une expression, que l'ordi évalue.
  - Les fonctions sont comme en maths: pas d'effet de bord. Fonctions passées en paramètres
  - Les données sont juste des paramètres, pas d'état global, ni de variable mutable (seulement des constantes calculées à la volée)
  - *Avantages:* bon niveau d'abstraction; ordre sans importance (multithread sans mal); plus facile à prouver correct (transparence référentielle: une expression peut être remplacée par sa valeur)
  - *Inconvénients:* moins naturel par rapport au fonctionnement d'un ordi (mais l'esprit matheux aide?)
    * code golf: on peut exprimer la solution en peu de caractères, mais c'est dur quand on n'a pas l'habitude (même juste pour relire)
    * pas pratique si le problème a beaucoup de variables et/ou de traitements séquentiels
- **Programmation orientée objets**: Impératif, mais en rangeant les données (focus sur les données)
  - Le problème est découpé en entitées indépendantes: des objets. Regroupent une partie de l'état global, et les méthodes qui peuvent s'appliquer à ces données
  - Données encapsulées, protégées: accessibles seulement au travers des méthodes définies dans l'objet
    * Objet = boite noire ⇒ facile d'abstraire l'implém, usage comme un outil de complexité moindre
  - C'est impératif car les méthodes des objets donnent le détail des traitements
  - *Avantages:* bon niveau d'abstraction, assez naturel à expliquer
    * L'héritage permet de factoriser du code en spécialisant du code générique
  - *Inconvénients:* Difficile à prouver; ordre important (cauchemard multithread);
    * code bloat: code vite gros et lourd. En java, la lourdeur du langage nécessite des éditeurs comme Eclipse pour assister l'écriture (mais ces outils ont la grâce et la légereté d'un tractopelle)
- **POO vs. Fonctionnel**
  - Vous préférez les noms ou les verbes pour écrire un texte ?
  - Étude du code "a cat catches a bird and eats it"
    * En OOP, on se focalise sur les deux noms: cat et bird, et on les dote de méthodes permettant d'agir dessus (catch et eat)
    * En FP, on se focalise sur les deux verbes: catch et eat, et on en fait des fonctions pures s'appliquant à des constantes typées
      · (comme ces objets n'ont pas vocation à persister, on les crée sur la pile en tant qu'objets temporaires – on y revient)
  - Les puristes disent depuis des décénies que l'un des paradigmes va gagner entre FP et OOP.
    * Mais en fait, les langages mainstream deviennent multiparadigmes.
    * Ça montre bien que le *sweet spot* est souvent un mélange des deux approches.
  - C++ initialement pensé comme langage POO, mais concepteur STL était adepte de programmation générique et très critique de l'OOP