C++ et génie logiciel

Martin Quinson

April 29, 2024

Présentation du module de génie logiciel

- Après l'apprentissage du C en vue de comprendre le système, ce cours est une introduction au génie logiciel, c'est à dire l'art de faire de très gros systemes informatiques
- 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.
- 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++
  - L'an passé il y avait un projet mais pas cette année. Il y aura un TP à rendre
  - Examen final: sur table, avec une feuille de pompe manuscrite autorisée

Chapter 1

Le langage C++ (Better C)

I) 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, mais rapide.
- 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 (pour la vitesse; pour la taille, pas sûr)
- 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.
- **Langage qui évolue encore**: des standards fréquents pour améliorer le langage.
  - C++98, C++11, C++14, C++17, C++20.
  - C++11 est une révolution, C++20 apporte des nouveautés sans changer la base; les autres sont des petits incréments pour utilisateurs chevronnés
  - C++23 ajoute des choses compliquées pour la programmation fonctionnelle (genre map et filtrage)
  - Tentatives de maintenir la compatibilité (par opposition au python3). C'est probablement une mauvaise idée.

II) À quoi sert le C++?

- "C++ makes programming more enjoyable for serious programmers", Stroustup 1994.
- Très utilisé: toutes les entreprises connues.
  - TIOBE: 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%
    - 2022: #1 Python 16.4%; #2 C 16.2%; #3 C++ 12.9%; #4 Java 12.2%; #5 C# 5.7%
    - 2024: #1 Python 13.4%; #2 C 11.4%; #3 C++ 9.9%; #4 Java 7.8%; #5 C# 7.1%
    - C se maintient grâce à Linux et à l'IOT; Java décroche enfin.
    - Javascript #7, Rust #18, Perl #19. les dev Rust interrogent moins les moteurs de recherche?
  - 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.
  - Un OS complet qui vaut la peine d'êre visité : SerenityOS (écrit en rehab, contient tout du noyau au navigateur web en passant par l'éditeur en 800k lignes monorepo)

III) 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++.
  - 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, mais pas tant que Rust
- **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. Typage bien plus fort qu'en C, mais moindre qu'en Rust
- **Programming at large**. Il est fait pour les projets qui durent des décennies. Par opposition au programming at small, qui regroupe les scripts one shot écrit en python voire bash
  - Ca reste le langage qui gagne le plus souvent aux compétitions de prog comme le SWERC car une fois maîtrisé (!) ce langage est vraiment top

- **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 8 ans de C++ (et 20 de C), je ne maîtrise tjs pas complètement 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.

IV) C++ en pratique

IV.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 (plus simple, je l'ai utilisé longtemps) ou VS Code/Codium (usine à gaz, mais bcp de gens aiment gacher de l'electricité)
- 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-format (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.

IV.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()`

IV.3) Les flux (stream)

1. 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

2. Lire/écrire dans un fichier
   - Cf le doc joint, garanti sans magie dedans.

3. 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)

4. 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

IV.4) Langage typé

- Le compilateur fait plus qu'en C (où il n'utilise les types que pour connaître la taille mémoire des variables), même si ce n'est pas encore Haskell bien sûr.
- L'objectif est que le typage prenne le meilleur des deux mondes, à vous de voir si c'est une réussite.
  - transtypage est une vraie hisoire par rapport au C (y'a des verif faites)
  - On est loin des fonctionnalités du rust ou Haskell dans le systeme de typage
  - C++23 a des concepts, qui sont des sortes de meta-types de données (copiés des Typeclass Haskell il me semble), mais c'est comme souvent en C++ compliqué pour pas grand chose
  - Pas d'introspection à runtime, c'est triste
- Impacts immédiats sur le langage:
  - Un vrai type booléen au lieu de surcharger les entiers: `bool done = true; // or false`
  - `const` du C++ est bien préférable au `#define`. `const` existe en C, mais le compilo n'insiste pas trop dessus.
    - En C++, le compilateur fait réellement attention au typage des expressions, et aussi à leur const-ness (pour approcher la prog fonctionnelle?)
    - `const int width = 512;` préférable à `#define WIDTH 512` car le compilo veille au typage.
  - la valeur `nullptr` à la place de `NULL` (qui était une affreuse macro définie à 0), pour donner une valeur correctement typée au compilateur.

IV.5) Nouvelles constructions du langage

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

1. 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
2. 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 `<` sont des fonctions définies par défaut...
3. 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
  - Il faut évidement pas que ce soit ambigu avec des fonctions de meme nom et même prototype
- Pas de passage de parametre nommés

4. Boucles étendues et itérateurs
   - On peut écrire des choses comme dans "Vecteurs et boucles étendues" pour parcourir une collection
     - `for (variable : collection)` la variable prend successivement toutes les valeurs de la collection
   - Les itérateurs permettent de parcourir les *collections* de données en C++. Ca a donné les conteneurs en Java
     - Cela permet 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 en fonction des parcours qu'on peut faire (forward: sens unique, random access: accède à l'indice), et de si on peut modifier les cases pendant le parcours
     - Les collections donnent des méthodes pour retrouver des itérateurs de chaque type

IV.6) La bibliothèque standard (STL)

- En C, on n'avait rien. En C++, on a bcp.
  - La STL contient bcp de choses, mais ca reste centralisé (difficile d'ajouter des choses, ca doit être standardisé)
  - Rust et python ont de bien meilleurs mécanismes pour avoir un écosysteme de modules (pip et cargo)
  - Mais c'est mieux que le langage D qui était pas mal, mais est mort d'avoir deux stdlib incompatibles.

1. 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`
2. 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());`
3. 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.
   - Manque encore les zip sur les collections (c'est C++26 mais encore plus compliqué)
   - Pas de module, pas de cmake

IV.7) 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, car le compilateur fait certes quelques vérifications à la déclaration de la template, mais quand on l'instancie, il génère le code correspondant avec les bons types, et compile le code généré. Cela lui permet de faire plus de vérifications.

IV.8) 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 (mais ça aussi c'est permis dans les

versions modernes de C)

Chapter 2

Encapsulation en C++

I) Etude du code

- On observe la classe `point` du document distribué, et on compare à l'implémentation C. `class` est comme `struct` sauf que:
  - On peut mettre des méthodes dedans (par exemple les méthodes x(), y() et move())
  - On peut spécifier ce qui est public et ce qui est privé directement ici
- Les méthodes sont implémentées dans un fichier séparé point.cpp. Elles ont un nom genre Point::move()
- On utilise (dans main.cpp) les objets comme si c'était des structures (ligne 7).
- On peut surcharger les opérateurs + et << facilement, pour faire du beau code. On y revient.

II) Constructeur et destructeur

- La mémoire est allouée de facon transparente
- Création d'un ctor par défaut si on en donne pas
- Les créations d'objets (dans main.cpp) demande des paramètres même pour les objets créés automatiquement (cpp:ligne 5).
- On a demandé le destructeur par défaut (hpp:15) car il suffit. On n'avait même pas besoin de le demander.
- Le constructeur **initialise** les champs et le corps de méthode est vide
  - L'initialisation a lieu avant l'exécution du constructeur, et c'est plus efficace
  - C'est aussi plus lisible: on voit bien que c'est une initialisation préliminaire et non un calcul
  - Attention, l'ordre des initialiseurs doit être le même que celui des champs dans la classe.

III) Propriété des champs

- le champ `count_` est statique : une seule copie est partagée entre toutes les instances de la classe. On l'utilise pour attribuer des numéros uniques à chaque point.
  - Il est interdit d'attribuer une valeur à un champ statique dans la définition de classe, il faut le faire dans un fichier séparé (cpp:3)
- `Point::rank` est constant. Même le constructeur n'a pas le droit de lui affecter une valeur. Il faut donc l'initialiser avant le corps du constructeur

IV) Fonctions d'accès (getters)

- les fonctions `x()` et `y()` donnent accès en lecture seule à des champs privés sinon. Habituellement, on les nomme d'après le champ directement, sans préfixer "get", mais certains suivent les habitudes java.

V) Mot-clé this

- Cela désigne l'objet courant.
- Ce n'est pas indispensable pour accéder aux champs de l'objet (`x_` est défini dans le scope de la méthode `move()`)
- On peut l'utiliser pour désigner l'objet courant, par exemple en passage de paramètres. Il est de type `(Point*)`

VI) Propriété des methodes

- const: la méthode ne modifie pas l'objet (le compilo peut optimiser, et ça donne des garanties à l'utilisateur)
- static: la méthode n'utilise pas d'instance d'objet en particulier. Seulement des champs statiques ou des choses extérieures
- inline: on peut, mais ce n'est plus une bonne idée de nos jours car les compilos sont meilleurs que nous à ce jeu

VII) Visibilité (public/private)

- En fait, `struct` existe encore. Publique par défaut en `struct`, privé par défaut en `class`.
- La privacy se fait par class, pas par objet. Donc on peut voir les champs privés des autres objets de la même classe que soit
- On peut avoir des classes amies qui verront les champs privés. On peut aussi avoir des fonctions amies, ou limiter l'amitié à une méthode de la classe

VIII) Surcharge d'opérateurs

- Le C++ a ceci de jouissif qu'on peut faire en sorte que nos classes deviennent des citoyennes de première classe dans le langage.
- On peut réécrire la plupart des opérateurs du langage pour qu'ils s'appliquent bien à nos classes
  - Les classiques
    - arithmétique: `operator+()` `operator-()` `operator*()` `operator/()` etc;
    - comparateur `operator==()`, `operator!=()` `<` `>` `<=` `>=`
    - incrément `++` `--` en pré et post. On fait la différence car post prend un paramètre entier inutilisé.
  - Les agréables
    - sérialisation/désérialisation << et >>
  - Les plus surprenant
    - déréférencement: `operator*()` et `operator->()` et `operator&()`
    - invocation: `operator()()`
    - subscript `operator[]()`
    - virgule: `operator,()`
    - new, delete
    - conversion: `operator int()` pour convertir en entier; `operator bool()`;
      - s'il y a des ambiguités à la compil sur la conversion à réaliser, alors erreur de compil.
      - On ne peut pas convertir en tableau[] (mais en pointeur sur éléments, ca passe)
  - Celui qui manque: `operator.()` pour ajouter des méthodes dynamiquement aux objets. Tjs cet objectif d'efficacité du C++
- Certains opérateurs peuvent se définir depuis l'intérieur de la classe (+ et == dans l'exemple) ou en dehors, avec une fonction normale comme « ici.
  - Voir la doc pour quels opérateurs sont surchargeables comment.
  - << est forcément sous forme de fonction car on peut pas modifier le receveur de l'appel (`std::cout`) dans notre code

IX) Outillage C++ (le reste de l'exemple fourni)

- On utilise la bibliothèque de tests Catch2, un standard du domaine
- On utilise la partie `ctest` de cmake pour expliciter quels sont les tests

X) Design objet: méthode CRC

- Classes, responsabilités et collaborateurs
- Difficile de définir ce qui constitue un bon design logiciel pour un projet donné, surtout quand on manque d'expérience.

- Pour nous aider, nous allons réaliser des *cartes CRC* (classes, responsabilités et collaborateurs), inspiré de l'article de K. Beck and W. Cunningham décrivant cette approche, disponible sur le site du cours.}.
- On décrit chaque classe dans un format simple sans penser à l'implémentation
- Ca peut se faire en UML (Unified Modeling Language), un "formalisme" dédié, mais on va pas s'embêter pour l'instant
- Les 3 parties d'une carte CRC:
  - Un bon **nom de la classe** est important pour créer un vocabulaire entre les différents humains impliqués dans le projet. Évocateur, et attaché à ce que fait l'objet plutôt qu'à comment il est construit. What plutôt que How.
  - **Des responsabilités** identifient les problèmes qui doivent être résolus par la classe. Deux types:
    * ce que l'objet **doit pouvoir faire** (effectuer un calcul, modifier son état interne, créer d'autres objets, coordonner d'autres objets, etc) pour contribuer à la résolution du problème
    * ce qu'il **doit savoir** (valeurs à conserver) pour pouvoir implémenter les services proposés.
    * On décrit l'interface publique des services offerts sans trop penser à l'implémentation, mais il faut que ça reste implémentable sans magie
  - **Les collaborateurs** sont les classes avec lesquelles les objets de la classe décrite interagieront directement. Il peut s'agir de classes qui serviront d'outils pour la réalisation des responsabilités de la classe décrite, ou au contraire de classes qui utiliseront l'un des services offerts par la classe décrite.
- Exemple d'une carte CRC représentant un dé à jouer.

dé à jouer	
Responsabilities	Collaborators
– Conserve la valeur de chacune des faces du dés. – Conserve quelle est la face actuellement visible du dé. – Permet de consulter la face visible du dé. – Permet de lancer le dé (modification de la face visible).	– Générateur aléatoire.

X.1) Exercice: Gestionnaire de bibliothèque

- Imaginez que vous devez concevoir un système de gestion pour une bibliothèque. Vous allez utiliser la méthodologie CRC pour identifier les classes, leurs responsabilités et leurs collaborations.
- Classe : Livre
  - Resp: Stocker les informations sur un livre (titre, auteur, édition, etc.).
  - Resp: Gérer les copies disponibles et empruntées du livre.
  - Collab: Bibliothèque (pour l'ajout, la suppression et la recherche de livres).
  - Collab: Membre (pour le processus d'emprunt et de retour de livres).

- Classe : Bibliothèque
  - Resp: Stocker la collection de livres.
  - Resp: Gérer les opérations liées aux livres (ajout, suppression, recherche).
  - Collab: Livre (pour l'ajout, la suppression et la recherche de livres).
  - Collab: Membre (pour le processus d'emprunt et de retour de livres).
- Classe : Membre
  - Resp: Enregistrer les informations sur les membres de la bibliothèque (nom, adresse, etc.).
  - Resp: Gérer les emprunts et les retours de livres.
  - Collab: Livre (pour emprunter et retourner des livres).
  - Collab: Bibliothèque (pour vérifier la disponibilité des livres et mettre à jour les emprunts).
- Classe : Emprunt
  - Resp: Suivre les emprunts de livres effectués par les membres.
  - Resp: Gérer les dates de début et de fin de l'emprunt.
  - Collab: Membre (pour associer un emprunt à un membre).
  - Collab: Livre (pour associer un emprunt à un livre).

Chapter 3

Gestion de la mémoire en C++

Différents types d'objets, pour simplifier l'usage. Même si la multiplicité de possibilités complique un peu

- Objet automatique: pour les variables locales. Créés à la demande quand on entre dans le bloc, et détruits en sortant
- Objets statiques: marqué avec le modifieur static, créés avant le main, et détruit automatiquement en fin de programme
- Objets temporaire: si on fait un appel explicite au constructeur dans une expression $\boxed{\text{a = Point(1,4)}}$, le compilo crée un objet temporaire qui sera détruit dès qu'il ne sera plus utile
  - Affectation d'objet temporaire très différente de l'initialisation d'un objet automatique
- Objets dynamiques: avec `new` et `delete` à la place de malloc et free pour mettre sur le tas. Attention à tout bien libérer à la fin.

I) Objets comme champ de classe

- Si l'un des champs est un objet comme ci-dessous (et sur le code fourni), le constructeur de pointcouleur est appellé avant celui de point.

```
class pointcouleur {
 point p;
 int couleur;
 public:
 pointcouleur(double x, double y, int c) : p(x,y), couleur(c) {}
};
```

II) Passages d'objets en paramètre, retour d'objet

- On fait une affectation pour copier.
  - Le compilateur sait faire si la classe est triviale (champs scalaires ou triviaux, pas de constructeur ni méthode virtuelle)

  - Si non, il faut surcharger l'opérateur d'affectation (ou bien on a une erreur de compilation)
    `T& T::operator = (T const& other);`
- On peut aussi interdire explicitement de copier les objets (par exemple s'ils contiennent des ressources difficiles à copier)
  `T& T::operator = (T const&) = delete;`

III) Initialisation et conversion implicite

```
point p = 5; // il faut un constructeur de type int
```

- comme ce genre de conversion est très dangereuse, il est conseillé de marquer le constructeur à un seul argument come `explicit`, indiquant qu'il faudra demander la conversion explicitement

```
class Point {
  explicit Point(int v);
};

 Point p = Point(5);
```

- S'il y a des ambiguités à la compil sur la conversion à réaliser, alors erreur de compil

IV) Tableaux d'objets

- Il faut utiliser `new point[20]` pour allouer la mémoire pour stocker 20 points, et invoquer le constructeur sans paramètre sur chacun. `delete[] variable` ensuite.
- On peut spécifier des paramètres au constructeur avec des {} depuis c++11, cf l'exemple en bas du main.cpp. Les accolades marquent des initialiseurs de liste, que l'on peut imbriquer.

Chapter 4

Héritage

I) Principe

- On a dit la semaine dernière que l'encapsulation était l'un des principes fondamentaux de l'OOP, avec l'héritage et le polymorphisme. Continuons avec l'héritage.
- Objectif: factoriser du code pour réduire sa taille / réutiliser du code existant pour en écrire moins
  - On veut avoir suffisament peu de code pour que tout tienne dans la tête à la fois.
  - Le grand démon est le copie-colle de code: on va vite se perdre dans des kilomètres de code presque pareil, à quelques nuances près. Et il faudra tout maintenir en double: appliquer chaque amélioration/correction à toutes les copies.
- Supposons qu'on veuille modéliser deux especes animales: le pangolin, et le pangolin à longue queue.
  - Un pangolin a un nom, un nombre d'écaille et sait crier.
  - La seule différence d'un pangolin à longue queue est qu'en plus il a une longueur de queue
  - Le code des deux classes s'annonce répétitif si on n'y prend garde.
- Observons le code fourni à la place
  - La classe `PangolinALongueQueue` est une extension / spécialisation de `Pangolin`
  - Ca se voit `PangolinALongueQueue.hpp:3`: le nom de la classe dérivée est écrit après `:`
  - On précise `public` pour signaler que le contenu de `PangolinALongueQueue` voit le contenu de `Pangolin` comme si c'était défini chez lui. On peut aussi hériter sans montrer son implémentation à ses classes filles.
  - En résumé, un héritage permet de faire comme un copie/colle, mais en mieux

II) Vocabulaire

- **Classe mère** ou super-classe. Ici `Pangolin`
- **Classe fille** ou sous-classe. Ici `PangolinALongueQueue`
- On parle de **hiérarchie de classes**

- On dit que la classe fille **spécialise**, ou **dérive** de la classe mère.
- On dit que la classe mère **généralise** la classe fille. `Pangolin` regroupe les caractéristiques communes à tous les sous-types de pangolins.
- On dit aussi que `PangolinALongueQueue` **est-un** `Pangolin`

III) Modalité d'accès

- Les champs privés de la classe mère sont inaccessible à la classe fille. L'encapsulation est respectée, même dans la hiérarchie
- Si `class PangolinALongueQueue : public Pangolin`, les champs publiques de Pangolin sont publiques dans la fille
- Si `class PangolinALongueQueue : private Pangolin`, les champs publiques de la mère deviendraient privés à la fille. La fille les voit mais pas les utilisateurs de la classe fille.
- Nouveau type d'accès: protected en plus de private et public. C'est visible depuis les sous-classes.

IV) Ordre d'appel des ctor, dtor

- Si aucun ctor n'est donné, le compilateur en fait un par défaut, qui donne les valeurs par défaut aux champs. Si les champs sont des objets, il faut que les classes correspondantes aient des constructeurs par défaut elles-même.
- Le constructeur de la fille doit choisir un ctor de la mère dans son initialisation (sauf si y'a des ctor sans paramètre)
- Le constructeur de l'ancêtre le plus haut appelé en premier, puis dans l'ordre. On construit un batiment en partant des fondations de base, puis on monte en abstraction.
- Les destructeurs sont invoqués dans l'ordre contraire

V) Heritage multiple

- On peut donner plus d'un ancêtre quand on défini une classe, mais c'est souvent une mauvaise idée
- Les champs de chaque ancêtre sont disponibles dans la classe fille, MAIS
- Si les deux ancêtres définissent tous deux une methode d'un nom donné, on ne sait pas laquelle sera invoquée et il faut spécifier dans la liste d'initialisation du constructeur
- C'est particulièrement bête si on a un schéma d'héritage en diamant comme dans le code fourni: l'ancêtre commun est dans les deux branches, donc le constructeur de l'ancetre est appelé 2 fois
  - On peut utiliser un héritage virtuel comme suit pour éviter que le ctor soit appellé 2 fois.
  - Mais c'est souvent une meilleure idée de juste pas faire de diamants, puisque d'autres pbs arriveront dès qu'on oublie de spécifier laquelle des classes mères utiliser dans les fctions virtuelles

- Java interdit l'héritage multipe (sauf pour les interfaces qui sont des classes complétement abstraites, puisqu'elles n'ont pas de code)
  - Scala et Rust l'autorisent, mais que pour les traits avec un ordre de résolution explicite des méthodes

Chapter 5

Polymorphisme d'objets

I) Principe de substitution de Liskov

- On peut affecter un objet dans une variable moins spécialisée: `A* a = new B(3)`.
  - L'utilisateur veut un outil; un marteau saura faire tout ce qu'on peut demander à un outil.
  - Attention, `A a = B();` (sans étoile ni new) est un piège du C++, on y revient la semaine prochaine.
- En revanche l'inverse ne peut pas marcher puisque l'objet général affecté dans une variable plus spécialisé ne saura pas faire les choses de spécialité.
  `B* b = new A(3)` interdit: tous les outils ne font pas de bons marteaux.
- La même règle s'applique sur le passage de paramètre
- Voir si deux objets sont de même type: `typeid(a) == typeid(b)`
- Voir si des objets sont compatibles : C++ introduit de nouveaux transtypages, sur lesquels on revient.

II) Redéfinition de méthode

II.1) Principe

- On veut parfois modifier une méthode dans la classe fille
- En java, il suffit de définir une méthode de même signature, éventuellement en ajoutant `@Override`
- En C++, il faut déjà que la mère déclare que cette méthode est redefinissable, avec `virtual`
  - La fille peut ensuite redéfinir, éventuellement en ajoutant `override` (bonne idée de laisser le compilateur vérifier qu'on a pas fait de typo dans le nom ou le type des paramètres)
  - Il est interdit de redéfinir une méthode qui n'est pas marquée `virtual` car le compilateur n'installe pas le mécanisme qui permet la redéfinition (la `vtable` cf juste ci-dessous)

- Au besoin l'implémentation de la fille peut invoquer l'implémentation de la mère avec `mere::methode()` pour l'étendre au lieu de la remplacer
- Une méthode redéfinie (ou dérivée) peut elle-même être redéfinie dans une classe fille (sauf si elle est marquée `final`, qui casse la possibilité de dériver)

II.2) Exemple du code des Marsouins

II.3) Override, overload, overwrite

- Quand deux méthodes ont le même nom, plusieurs cas sont possibles en C++.
- Si c'est dans les classes mère/fille, que la méthode ancêtre est marquée `virtual` et que la signature est inchangée, c'est `override` (redéfinition)
- Si la signature est changée, c'est de la surcharge (overload), c'est à dire que deux méthodes co-existent avec le même nom et des paramètres différents. Cela n'implique pas d'héritage, et c'est ça qu'on fait depuis longtemps pour les opérateurs.
- Mais il y a un twist en C++ : l'overload est forcément au sein de la même méthode, jamais intergénérationel.
  - Si l'override échoue (soit pas de `virtual`, soit pas exactement les mêmes paramètres – on revient sur les règles de covariance la semaine prochaine), alors c'est un ... overwrite.
  - La méthode réécrite dans la classe fille masque celle de la classe mère.
  - C'est assez naturel quand les paramètres sont identiques dans les deux méthodes, et bien plus étrange quand les paramètres sont différents. Mais il faut se souvenir qu'il n'y a jamais d'overload intergénérationel, c'est comme ça.

	Signature	**Scope**	**Comport**
`override`	Inchangée	Mère-fille	Remplacer
`overload`	Modifiée	Même classe	Co-existen
`overwrite`	Inchangée mais méthode pas `virtual` dans mère ou bien signature modifiée (même si marquée `virtual`)	Mère-fille	Remplacer

II.4) Classe abstraite

- si une méthode est marquée `virtual =0` (méthode virtuelle pure), la classe ne fourni pas d'implémentation de cette méthode
- Cette classe est impossible à instancier (car on ne saurait pas quoi faire si quelqu'un appelle cette méthode sur cet objet). On dit que la classe est **abstraite**
- Les filles doivent implémenter cette méthode, sous peine d'être abstraites elles aussi
- C'est parfois utile pour factoriser du code: les autres méthodes de la classe abstraite peuvent appeler la méthode qui sera définie dans les filles, sans se préoccuper de comment c'est fait.
  - Exemple: définir `operator==(A& other)` en fonction de `operator!=(A& other)`

II.5) Implémentation du C++

- l'édition de liens d'une méthode virtuelle ne peut se faire à compil time ni meme à link time. Il faut attendre runtime car on ne connait pas le type dynamique de l'objet avant ce moment.
  - la classe a un champ magique `vtable` "table des méthodes virtuelles disponibles", et pour chaque méthode virtuelle, le compilateur génère un petit stub qui consulte cette table à l'exécution pour choisir la méthode à invoquer aujourd'hui.
  - le dtor doit être virtuel lui aussi, pour nettoyer la table des méthodes

III) Types objets en C++

III.1) Types statique et dynamique

- Le **type statique** (celui de la variable contenant l'objet) indique la liste des méthodes invocables sur l'objet (liste de signatures)
  - Connu dès la compilation
  - On peut le changer avec un transtypage
- Le **type dynamique** (celui donné à la création de l'objet) détermine la méthode qui sera invoquée
  - Connu uniquement à l'exécution, impossible à déterminer dans le cas général
- Souvent, c'est le même type, mais pas avec `A* a = new B();`
- Il s'agit bien d'une forme d'édition de lien à runtime, appellée **Liaison tardive (late binding)**

III.2) Transtypage

- Le type statique est le type donné à la variable. Le type dynamique est le type de la donnée stockée dans la variable
- `static_cast<A>(b)` équivalent du transtypage C: je te dis que c'est ce type, peu importe ce que tu penses, fais moi confiance
- `dynamic_cast<A>(b)` vérifie que c'est possible, s'il te plait.
  - Si oui, donne moi un objet du nouveau type (en utilisant les liens d'héritage, ou les opérateurs de convertion – cf. code fourni).
  - Si non, donne moi `nullptr`. Erreur à la compilation si c'est impossible d'après le type statique
  - Utilisable uniquement sur les pointeurs vers des objets, donc, pour que nullptr soit une valeur légale. Dans un langage moderne on aurait un type optionnel avec Some/None
  - c'est ce qu'on utilise pour savoir si l'objet contenu dans x est d'un type compatible au sens de Liskov avec la classe A (utilisable comme un A)
    `if (dynamic_cast<A>(x) != nullptr)` ou bien `if (auto* ax = dynamic_cast<A>(x)`
- `const_cast<A>(b)` juste pour changer la const-ness

- reinterpret_cast<A>(b) truc étrange rarement utile, sauf pour convertir entiers en pointeurs (donc, rarement utile)
- Le transtypage C est encore valide. Le compilo tente un `const_cast`, puis `static_cast`, puis `static_cast<const_cast>` avant des choses plus étranges.
- On regarde le code "Opérateurs de convertion" et on complète ce qui se passe dans chaque cas

cf. sur la fiche distribuée

```
struct A {
    A(int) { }      // converting constructor
    A(int, int) { } // converting constructor (C++11)
    operator bool() const { return true; }
};
int main() {
    A a1 = 1;      // copy-initialization -> A::A(int) (car '=', sauf si explicit)
    A a2(2);       // direct-initialization -> A::A(int)
    A a3 {4, 5};   // direct-list-initialization -> A::A(int, int)
    A a4 = {4, 5}; // copy-list-initialization -> A::A(int, int) (sauf si explicit
    A a5 = (A)1;   // explicit cast performs static_cast
    if (a1) ;      // operator bool()
    bool na1 = a1; // copy-initialization -> A::operator bool() (sauf si explicit)
    bool na2 = static_cast<bool>(a1); // static_cast performs direct-initializatio
}
```

- `copy-initialization` n'a pas lieu si le constructeur est marqué `explicit`. C'est souvent mieux car cette conversion magique est dangereuse: on préfère souvent (ou on devrait préférer) que le compilo nous signale du code chelou plutôt que s'imaginer qu'on sait parfaitement ce qu'on fait.

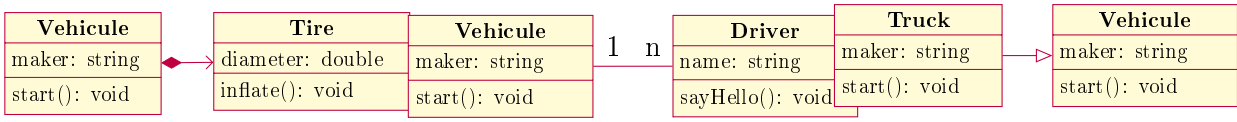
Chapter 6

Design OOP

- Quand je cherche à décomposer un problème de façon OOP, il faut donc que je trouve un ensemble de concepts qui se combinent bien
- Le plan d'ensemble est un schéma UML: des boites avec des flèches entre les boîtes
  - Chaque boîte est un concept (une classe), bien encapsulé
  - Chaque flèche dénote une interaction conceptuelle entre les concept
    - Relation 1: composition (**a-des**) quand on décompose un objet en sous-objets
    - Relation 2: association = composition réciproque: mise en relation de concepts
    - Relation 3: héritage (**est-un**) quand on voit une spécialisation d'un objet

I) Formalisme UML

- Vocabulaire graphique universel en matière de décomposition



- C'est ici une forme extrêmement simplifiée.
  - La visibilité publique/protected/privée peut s'écrire `+/-/\#`
  - La composition rend propriétaire des bouts; l'agregation (losange vide) non
  - `https://programmation-orientee-objet.pages.ensimag.fr/poo/resources/fiches/07-UML/`

II) Programmer proprement

- en général, on fait des sous-classes pour factoriser du code entre des trucs semblables. Mais c'est vite compliqué, les clients ne devraient pas forcément voir l'arborescence.
- Les classes abstraites sont pour proposer une interface à l'utilisateur, qui ne voit pas les implémentations
- L'héritage multiple est très dangereux, à proscrire sauf exception

- La surcharge de méthode et le sous-typage des paramètres ne font pas bon ménage, à proscrire

Chapter 7

Exception

- try/catch comme en Java. Pas de finally
- Les exceptions levées devraient être sous-classes de std::exception, mais ce n'est pas une obligation. On peut même lever des scalaires si on est sans coeur.
- On peut avoir plusieurs catchs d'affilé, le premier qui match est choisi.
- On lève une exception sans new (elle est copiée où il faut, pas besoin de chercher à faire fuiter sa mémoire) throw std::exception("Ouille");
- On catch par référence } catch (std::exception& e) { std::cerr « e.what(); }
- Quelques exceptions standards:
  - `logic_error`
    - `invalid_argument`: stoi("aze")
    - `domain_error`: std::acos(42)
    - `length_error`: on essaye de faire grandir un objet de taille statique
    - `out_of_range`: on lit en dehors
  - `runtime_error`
    - `range_error`: result cannot be represented by the destination type
    - `overflow_error` ; `underflow_error` ; `system_error` (C++11)
  - `bad_alloc`: erreur malloc
  - `bad_typeid`: typeid(nullptr)
  - `bad_cast`: `dynamic_cast` sans de lien d'héritage entre cible et type dynamique
  - `bad_exception`; `bad_weak_ptr`(C++11); `bad_function_call`(C++11)

Chapter 8

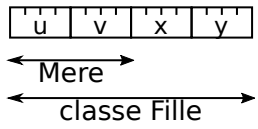
Implémentation de C++

- L'objectif est de comprendre comment plusieurs mécanismes de C++ sont implémentés, pour mieux savoir les utiliser.

I) Héritage d'objets

I.1) Objets triviaux

- Trivial en C++ = des choses facilement copiable, ie non-polymorphique
- On observe le code fourni "héritage en mémoire"
  - On a une classe mère et une classe fille, on initialise des instances de chaque
  - a2 est une instance de mère dans laquelle on range une fille
  - b2 est une instance de fille dans laquelle on range une mère, mais c'est interdit d'après Liskov. C++ permettrait d'avoir un opérateur de conversion Fille -> Mere, mais c'est ce qu'il permet de faire quand il n'y a pas de conversion naturelle (genre Fille -> bool ou int -> Mère)
  - Une classe plus spécialisée peu se faire passer pour une plus générique, mais pas le contraire.
- Qu'est ce qui se passe en mémoire ?
  - Le printf de la ligne 18 affiche `sizeof(a)=8; sizeof(b)=16; sizeof(a2)=8`
  - (sur mon ordi – `sizeof(int)=4` dans g++). Schéma mémoire correspondant:



- La représentation mémoire est extrêmement efficace
  - les champs de la fille sont derrière ceux de la mère
  - C'est pour ça qu'on peut transtyper fille -> mère (avec une perte d'information).
    - Cette perte d'information est exactement comme lorsqu'on converti le double 3.14159 en entier. On change la représentation mémoire et on a une perte d'information. De la même façon, quand on converti un objet de la classe `Fille`

en le type `Mere`, on a une modification de la représentation mémoire (ça prend 2x moins de place en mémoire) et on a une perte d'information irréversible.
  - Cela permet l'héritage sans aucun overhead, ce qui a fait le succès du C++ au début
- Et si héritage multiple ?
  - Les champs sont pris dans l'ordre d'héritage: Si C hérite de A et B, les objets C ont les champs A, les champs B puis les champs C.
- Et les champs statiques ?
  - Placés à coté des globales, dans le segment texte du processus. Comme les locales statiques, finalement.
- Et si on ajoute des méthodes (non virtuelle) ?
  - Elles sont ajoutées au segment texte du processus, comme les fonctions C, avec des règles de mutilation (mangling) pour passer des noms C++ aux identificateurs C.
  - Si on a les mêmes noms de méthodes dans les classes mères et filles, on se retrouve avec de la réécriture (overwrite), ce qui n'est probablement pas ce qu'on veut
    * `a.toto()` -> `Mere`, sans surprise
    * `b.toto()` -> `Fille`, sans surprise
    * `a2.toto()` -> `Mere` bien qu'on ait rentré un objet fille !!
      · C'est le type statique décide de l'appel, il n'y a jamais de polymorphisme en C++ s'il n'y a pas de pointeurs.
      · C'est comme la perte d'information liée au transtypage de double vers entier (3.14 -> 3) évoqué ci-dessus.
    * `a3->toto()` -> `Mere` également!!! On a stocké une fille dans la variable, il y a un pointeur qui pourrait permettre au polymorphisme de s'exprimer (pas de perte d'information lié à un transtypage brutal en mémoire), et malgré tout c'est le type statique de l'objet qui s'exprime. Pour comprendre ça, il faudra la section II de cette séance sur la décoration de nom. En spoiler, c'est parce que comme il n'y a pas `virtual`, tout est décidé à compile-time et le compilateur ne se base que sur le type statique de l'objet, jamais le type dynamique.

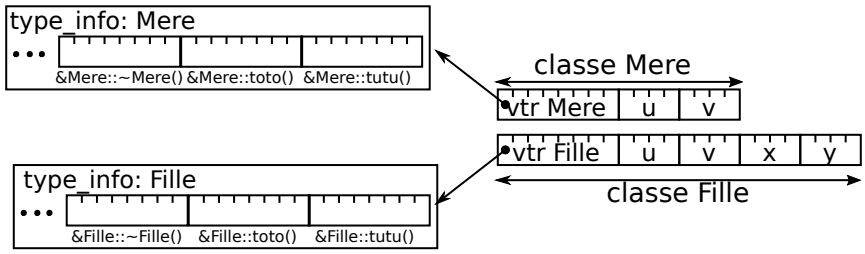
I.2) Objets non-triviaux

- Comment sont gérées les méthodes virtuelles?
- On regarde maintenant le code d'héritage polymorphique.
  - Qu'est ce que la méthode `toto()` affiche pour chaque objet?
    * `a.toto()` -> `Mere`, sans surprise
    * `b.toto()` -> `Fille`, sans surprise
    * `a2.toto()` -> `Mere` bien qu'on ait rentré un objet fille !!
      · C'est le type statique décide de l'appel, il n'y a jamais de polymorphisme en C++ s'il n'y a pas de pointeurs.
      · C'est lié à la perte d'information due au transtypage d'objet brutal. Au moment où j'ai copié l'objet `Fille` dans la case mémoire juste assez grande pour un objet `Mere`, j'ai coupé les champs spécifiques à la classe Fille. Il est donc

peu probable que la méthode `Fille::toto()` fonctionne sans les champs spécifiques à la classe spécialisée. C'est pourquoi le compilateur choisi la méthode `Mere::toto()` ici malgré l'affectation, car l'objet Fille n'est probablement plus fonctionnel après l'amputation de ses champs spécifiques.

  - `a3->toto()` -> `Fille`, comme on s'y attendait au début : le type dynamique peut s'exprimer car il n'y a pas eu de perte d'info lors de l'affectation, vu qu'on restait aux pointeurs
- Quelles sont les tailles des objets?
  - a:16, b:24, a2:16 (sur ma machine).
    - tous ces objets ont grossi de 8 octets par rapport à la version sans méthode virtuelle.
    - Des méthodes non virtuelles, n'auraient pas fait changer la taille: `sizeof(a)=8`
  - a3: 8. Preuve que les pointeurs sont de taille 8 sur mon ordi, et c'est la solution
    - On ajoute un pointeur par classe (pas de différence mère ou fille)

- Virtual method table
  - Pointeurs vers des tables stockées à part (mis en commun entre les objets de la même classe, probablement segment text)
  - Il y a bien une seule table virtuelle (sauf si héritage multiple). D'ailleurs, 24=16+8, pas 16+8*2



- – Il y a d'autres choses dans le `type_info`, dont le `typeid()`
  - Les méthodes sont toujours dans le segment de code du processus, donc les différentes cases de la vtable pointent vers des adresses dans le segment de code.
  - L'appel `a2->toto()` est compris comme `(* (a2->vtr[1]) )(a2)`
    - On cherche un pointeur sur fonction dans la table puis on l'invoque
    - On retrouve le `self` que Python rend explicite dans les méthodes de classe, et que C++ cache lors de la définition de la méthode (`this` est défini implicitement en C++ tandis que `self` est explicite en Python)
- Discussion et comparaison
  - Invoquer une méthdode virtuelle est plus lent qu'invoquer une méthode non virtuelle (un déréférencement de pointeur en plus, et un risque de prédiction de branche ratée), mais ça reste très efficace
    - On peut pas faire mieux sans JIT (prototypes de JIT dans LLVM)
  - Les choses sont un peu plus compliquées en cas d'héritage multiple, mais c'est très comparable
  - Explique que C++ ne permette pas d'ajouter des méthodes à la volée (par opposition

aux langages à Duck-typing comme le Python ou Javascript): taille de vtable doit être connue à la compil
  - C++ ne permet que le simple dispatch (choix de la méthode en fonction du type d'un receveur unique), tandis que d'autres langages permettent le double dispatch (choix de la méthode d'un opérateur en fonction du type des deux opérandes), mais c'était pas implémentable avec une vtable simple comme ça.
- Et dans le cas d'héritage multiple? Il y a plusieurs vtables dans la classe
  - `https://prog.world/c-vtables-part-1-basics-multiple-inheritance/` présente une session gdb pour explorer

II) Décoration de noms

- Pour comprendre ce qui se passe en cas d'overwrite, il est utile de comprendre un autre point du fonctionnement du compilateur C++: comment les noms typiquement C++ sont convertis en noms de symbole C pour permettre l'édition de liens.
- En anglais, ça se dit "name mangling", dont la traduction litérale est plutôt "déformation, broyage". Perso j'aime assez "mutilation de noms"
- À l'origine, les compilateurs C++ étaient des transpilers, des compilateurs source-to-sources ciblant le C. Il fallait donc que les identificateurs respectent les règles C.
- Même ensuite, les éditeurs de lien ne fonctionnaient que pour le C, ce qui fait que le mangling a perduré
  - Les autres langages comme Rust ou Go savent compiler du binaire pour eux, ou pour la compatibilité avec du C. Dans le second cas, il y a des règles de mutilation de noms.
- Le C++ a bcp plus de fonctionnalités que le C (classes, namespace, surcharge, redéfinitions, etc).
- En particulier, on a souvent plusieurs fonctions de même nom en C++, alors que c'est impossible en C
  - L'éditeur de liens a donc seulement le nom des symboles dans sa table, ce qui force la décoration de noms à inscrire des meta-data dans le nom du symbole généré
  - namespace, classe, types des paramètres sont ajoutés autour du nom de méthode
- Il n'y a pas de norme, pour laisser liberté à l'implémenteur ou au cas où l'éditeur de lien comprendrait le C++. La norme incite même à utiliser une décoration différente afin de ne pas mélanger quand le reste de l'ABI ne correspondrait pas (exception handling, vtable layout, stack frame padding).
  - Mais certaines plates-formes ont standardisé l'ABI quand même, pour la compatibilité. En particulier, g++ et clang++ ont la même ABI. Quand elle change, ca complique la vie des distributions linux
  - 1) le nom commence par `_Z` pour dire qu'il est décoré
  - 2) S'il y a des namespace et classes, `Nnombre`. `N` pour signaler le fait, et le nombre pour donner la taille de ce bout. Termine par `E`
    Sinon, le nom de la fonction est seulement préfixé de la taille du nom (sans `N` ni `E`)

  - 3) le type des paramètres, avec `i` pour `int`, `v` pour `void` et `c` pour char
  - 4) modificateurs `const` et autres sur les paramètres
- Chose importante : le type de retour n'est pas dans la décoration, donc on ne différencie pas dessus
- On peut désactiver la décoration en mettant `extern "C" {...}` autour des symboles à ne pas décorer (mais on ne peut utiliser que les choses compatibles avec C++ dans ces blocs)
- On peut maintenant comprendre l'overwriting, comme dans le cas de l'appel `a3->toto()` du cas trivial ci-dessus. Comme on n'a pas de vtable, tout a lieu à la compilation.
  - En pratique, le compilateur remplace la ligne `a3->toto()` par un appel de fonction standard, comme C aurait pu faire, mais avec le nom mutilé construit d'après le type statique de la variable `a3`.
  - La méthode `Mere::toto` devient `_ZN4Mere4totoEv` si on suit les règles de mutilation gcc/clang. Cela a lieu à la compilation.
  - Ce qui reste à l'exécution, c'est l'appel `_ZN4Mere4totoEv(*a3)`. On voit bien que le type dynamique de l'objet a3 n'a aucune chance de s'exprimer pour influer sur le code appelé, vu que c'est à la compilation qu'on a choisi le code à exécuter.

III) Covariance et contravariance

- Quelle est la méthode invoquée, à la fin? On peut répondre maintenant que l'on comprend le layout memoire
- Variance d'un constructeur de type:
  - covariant: accepte des sous-types mais pas de super-types
  - contravariant: accepte des super-types mais pas des sous-types
  - bivariant: accepte à la fois des super-types et des sous-types
  - invariant: n'accepte ni super-types ni sous-types
  - Ca devient drôle quand on combine avec la spécialisation/héritage de collections (mais pas en C++)
- Affectation (paramètre, affectation variable): covariance pour respecter le principe de Liskov. On peut passer une Fille là où une Mère est attendue
  - Si c'est une copie d'objet, il y aura transtypage avec amputation
  - Si c'est un pointeur, on utilisera l'autre vtable
- Redéfinition:
  - Les paramètres sont invariants que ce soit copie d'objet ou passage de pointeur (sinon le mangling ne fonctionne plus)
    * C++ n'autorise pas les redéfinitions à spécialiser les paramètres : le compilo veille
    * Si on défini quand même des méthodes covariantes ou contravariantes, ce sont ne sont pas des redéfinitions mais des réécritures. On ne peut pas utiliser `override` sous peine de compil error. Une réécriture est comme une surcharge intergénérationnelle. Le twist est que la méthode de l'ancêtre est masquée et impossible à

invoquer.

- Le type de retour est covariant si c'est un pointeur, et invariant si c'est une classe
  - Exercice: covariance de la feuille

```
struct A {
   virtual void boom()  { std::cout << "A::boom\n"; }
};
struct B : public A {
   void boom() override { std::cout << "B::boom\n"; }
};

struct Up {
   virtual A bidule()  { return A();     }
   virtual A* machin() { return new A(); }
   virtual void truc(A a)   { a.boom();  }
   virtual void chose(A* a) { a->boom(); }
   virtual void chouette(A a) { a.boom();  }
   virtual void muche(A* a)   { a->boom(); }
};
struct Down : public Up {
   // ERROR: invalid covariant return type for ‘virtual B Down::bidule()’
   //B bidule() override  { return B(); }
   // OK,
   B* machin() override { return new B(); }
   // ERROR: marked ‘override’, but does not override
   //void truc(B b) override { b.boom();  }
   // ERROR: marked ‘override’, but does not override
   //void chose(B* b) override { b->boom(); }

   void chouette(B b) { b.boom();  } // OK, mais c'est un overwrite
   // la méthode de l'ancêtre est masquée, réécrite. overload intergénérationel n
   void muche(B* b) { b->boom(); } // OK

};

int main()
{  Up up; Down down; A a;  B b;

   up  .bidule().boom(); // A::boom
   down.bidule().boom(); // A::boom (pas de surcharge vu que c'est interdit)
   up  .machin()->boom();// A::boom
   down.machin()->boom();// B::boom
   std::cout << "------------\n";
   up.truc(a);   // A::boom
   up.truc(b);   // A::boom (b est amputé lors de l'affectation au paramètre)
   up.chose(&a); // A::boom
   up.chose(&b); // B::boom
   std::cout << "------------\n"; // down tout comme up: pas de surcharge réussie
   down.truc(a);
   down.truc(b);
   down.chose(&a);
   down.chose(&b);
   //down.chouette(a);// Up::chouette(A) masqué par réécriture, Down::chouette(B)
   down.chouette(b);  // Downcasting d'objets interdit en C++
   //down.muche(&a);// Up::muche(A) masquée par réécriture, Down::much(B*) utilis
   down.muche(&b);  // Downcasting de pointeurs toléré mais très mauvaise idée
   return 0;
}
```

- Les templates: il faut tout faire soit-même
  - Invariant par défaut: les bouts de code générés n'ont pas de relation d'héritage entre eux
  - Mais on peut le faire en surchargeant `operator=()`
  - Les containers STL (vecteur, map, set) sont invariants, cf exemple
  - `std::pair<T *, U *>` et `std::tuple<T *, U *>` sont covariants
  - `std::shared_ptr` `std::unique_ptr` (sur lesquels on revient bientôt) sont covariants
  - `std::function<R *(T *)>` est covariant sur le return, et contravariant sur les paramètres...
  - `http://cpptruths.blogspot.com/2015/11/covariance-and-contravariance-in-c.html`
- On en sait maintenant assez pour comprendre le casse-tête de Beugnard
  - En C++, que des erreurs de compilations, rien au runtime. Autre histoire avec des `dynamic_cast`
  - Erreurs de compilations quand on généralise les paramètres vis-à-vis du type statique du receveur
    - `d.cov(top)` $\rightsquigarrow$ `Down::cov()` demande un `Middle*` et on tente une généralisation `Top*` $\Rightarrow$ error
    - `?.inv(top=` $\rightsquigarrow$ idem: `Middle*` attendu donc généralisation `Top*` refusée
    - `{u/ud}.contra( {top/mid} )` $\rightsquigarrow$ Même genre. `Bottom*` attendu, toute généralisation refusée
    - `d.contra(top)` $\rightsquigarrow$ `Down::contra()` demande `Middle*`, tentative `Top*`
  - Les colonnes `u` et `d` sont assez simples car type statique = type dynamique, donc liaison à la compil
  - Comment expliquer la colonne `ud` ?
    - Si le type dynamique s'exprimait seul, ça ferait `Down` tout le temps
    - Si le type statique s'exprimait seul, ça ferait `Up` tout le temps.
    - Pour comprendre la réponse, il faut se demander si les méthodes de `Down` sont `override` ou pas
    - Comme C++ n'autorise pas la variance, seule `inv()` accepte `override` quand tout est `virtual`
    - Conclusion: à la compilation, on détermine la méthode cible (son mangled name), et à l'exécution on cherche une méthode de cette signature là dans la vtable
    - au passage, si on retire "virtual" de `Up::inv`, le type dynamique ne peut pas s'exprimer car la méthode n'est plus dans la vtable, donc la méthode non virtuelle de `Up` s'applique même sur un type dynamique `Down`.

	u.?(?)	**d**.?(?)	**ud**.?(?)
?.**cov(top)**	Up	Compil error	Up
?.**cov(mid)**	Up	Down	Up
?.**cov(bot)**	Up	Down	Up
?.**inv(top)**	Compil error	Compil error	Compil error
?.**inv(mid)**	Up	Down	Down
?.**inv(bot)**	Up	Down	Down
?.**contra(top)**	Compil error	Compil error	Compil error
?.**contra(mid)**	Compil error	Down	Compil error
?.**contra(bot)**	Up	Down	Up

- Discussion
  - Il y a beaucoup de différences entre les langages OO : voir la page d'Antoine Beugnard pour les détails.
  - La sémantique C++ est faite pour permettre une implémentation aussi efficace que possible (invariance paramètres)
  - D'autres langages sont plus permissifs
    - Certains sont même contravariants: Eiffel et Dart autorisent à passer plus générique lors d'une affectation (sens inverse de Liskov), même si ça risque de poser pb si on utilise les champs spécialisés sur la généralisation passée à la fin. Ces langages lèvent une erreur runtime si le cas se présente

Chapter 9

Gestion automatique de la mémoire

- Les pointeurs C sont un outil aussi puissant que dangereux à utiliser. On peut se tromper de 1000 façons avec les pointeurs, et le langage C++ apporte plusieurs mécanismes pour réduire les risques.
- On a déjà vu les références, qui sont des pointeurs dont on sait qu'ils sont valides (une référence ne peut pointer que sur un objet donné). En échange, on ne peut pas faire d'arithmetique des références comme on fait de l'arithmetique des pointeurs. On ne peut donc pas les prendre pour des tableaux.
- On va voir deux mécanismes pour éviter les fuites mémoire : RAII et pointeurs intelligents

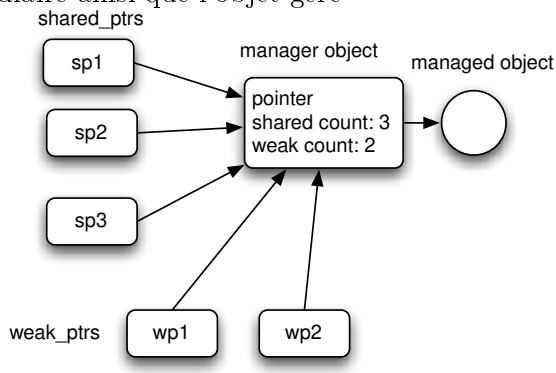
I) RAII (Resource Acquisition Is Initialisation)

- Il s'agit d'utiliser les objets automatiques pour ne pas avoir à gérer la mémoire à la main. Le compilateur invoque automatiquement le constructeur des **objets automatiques** quand ils rentrent dans le scope, et leur destructeur quand les objets sortent du scope. Par construction, ils sont vivants tant que la pile d'appel est dans ou au delà du cadre de pile où ils se trouvent. C'est très pratique
- Le principe du RAII est de créer des classes d'objet qui gèrent seules les ressources dans les constructeurs/destructeur pour ne plus avoir à s'en occuper.
- On regarde le motivating example, **code sans RAII** de la feuille d'exemples de code.
  - Le pointeur p est visiblement un espace de travail de cette fonction, alloué au début de la fonction, et libéré quand on sort. Ok, ça devrait marcher, mais on peut faire mieux.
  - (1) c'est pas si simple si y'a des exceptions. On peut tout mettre dans un gros try/catch, mais c'est affreux à lire et en plus ça devient très pénible s'il y a plusieurs ressources à éventuellement libérer.
  - (2) la dernière ligne est tellement obvious qu'on voudrait pouvoir s'en passer si possible.
- Le second exemple: **code avec RAII** fait ça. On n'a plus rien à faire à part créer la variable au début puis l'utiliser

  - Pour que ça marche, il faut un peu de magie C++, avec trois opérateurs dont un de conversion et deux pour "forwarder" les appels.
  - On est d'accord, cet exemple est sans intéret puisque `std::vector` ferait tout ça et bien plus. Mais bon.
- On peut faire du RAII pour la mémoire, mais pas seulement. Les fichiers vont se refermer tout seul de la même façon.

II) Smart pointers

- ce sont des objets prêts à l'emploi pour faire du RAII sur la gestion mémoire. Ils se chargent de faire new/delete automatiquement quand on n'en a plus besoin. Il en existe 3 sortes.
- `std::unique_ptr` c'est exactement l'implémentation du RAII vu avant. On peut le mettre en haut de méthode, où il fait new. On pourra pas déplacer son contenu facilement, et il fera delete à la fin. C'est super, mais on ne peut pas les déplacer ou les retourner à l'appelant. La ligne 15 du code d'exemple lève une erreur de compilation.
  - D'ailleurs, comment est-ce réalisé ? Simple: l'opérateur d'affectation est marqué `delete`
- `std::shared_ptr` c'est la même chose mais on peut avoir plusieurs "références intelligentes" sur le même objet, qui n'est détruit que quand la dernière référence est détruite.
  - Comment ça marche? Simple refcounting.
  - Il y a un objet intermédiaire qui (1) garde la référence à l'objet géré (2) compte le nombre de références.
  - Les références modifient l'objet intermédiaire dans leur ctor/dtor, et la dernière ref détruit l'objet intermédiaire ainsi que l'objet géré



  - Bien evidement, si on fait n'imp (comme retourner le pointeur nu dans l'objet), le smart pointer ne peut plus nous protéger
  - En fait si on fait `make_shared`, on n'a qu'un seul malloc, assez grand pour contenir à la fois le gestionnaire et l'objet géré
  - Les opérateurs `*` et `->` sont bien définis pour qu'on puisse utiliser le smart ptr comme une référence. (CODE: heritage et smart pointers)
    * Les opérateurs `==` et `!=` sont surchargés pour tester le pointeur géré

  - On peut faire un alias de type pour ne pas écrire `std::shared_ptr<MyType>` à tout bout de champ: `using MyTypePtr = std::shared_ptr<MyType>;`
    * On pourrait aussi faire un typedef, mais les alias sont conseillées en C++ car certaines choses avancées sont possibles avec using mais pas avec typedef (quand on fait du templating avancé)
- **Héritage et smart pointers**
  - Si l'affectation entre objets est OK, alors l'affectation entre smarty l'est (ie, les smart pointers sont covariants)
  - Les transtypages nécessitent un outil: `std::static_pointer_cast`, qui est une template de plus fournie par la STL
- `std::weak_ptr` pour lutter contre les références cycliques. Elles pointent sur l'objet pour retrouver l'objet géré, mais ne participent pas au refcounting. En pratique, c'est pas très pratique non plus. Il faut juste savoir que ça existe.
- Conclusion: avec les smart pointers, on n'écrit plus jamais new/delete soit-même, et les fuites mémoires sont un souvenir ancien.
  - Note: `std::make_unique()` dont y'a besoin pour ne même pas écrire `new` à l'initialisation est C++14, mais il devient très difficile de trouver un compilateur qui ne comprenne pas le C++14, de nos jours.

III) Déplacement mémoire

- On veut parfois déplacer un `unique_ptr`, ou bien passer l'ownership d'un `shared_pointer` sans jouer le jeu du refcounting (pour gagner du temps)
- `std::move()` permet de faire juste ça. Après le move, le pointeur original est mis à `nullptr` (comme s'il avait fait release)
  - D'ailleurs ce n'est pas limité au smart pointer, mais ça permet d'informer le compilateur du flot de durée de vie des objets afin qu'il puisse optimiser et éviter des copies.
  - Quand une fonction crée une chaine avant de la renvoyer, la copie est une perte de temps puisque l'original va être détruite.
  - C'est un sujet complexe, et je ne suis pas sûr de tout comprendre moi-même

IV) Référence rvalue

- Une référence rvalue (de type `T&&`), c'est comme une référence habituelle (dite lvalue) dont on ne peut pas changer la valeur.
  - En pratique, c'est une référence dont le propriétaire me donne l'ownership
  - C'est donc pas une qestion de const ou non, mais plutot de mouvement mémoire. On fait ça quand on veut informer le compilo pour qu'il optimise les copies mémoires inutiles.

V) Règles informelles pour mieux programmer en C++

V.1) Utiliser le compilateur à son avantage

- Activer un max d'options de warning, et compiler en `-Werror` (comme en C, quoi)
- Utiliser des outils d'analyse statique comme `clang-tidy`, le Clang static analyzer, Sonar ou LGTM (les deux derniers sont des programmes privateurs).

V.2) Autres conseils divers

- préférer les initialisations avec des accolades
- marquer "override" les fonctions redéfinies
- Google: utiliser les références qu'en const pour éviter la copie mémoire, mais préférer les pointeurs pour ce qui doit être modifié (afin de mieux voir ce qui peut être changé coté appelant)
- La plupart des conseils du livre de Scott Meyer (Effective Modern C++) s'y prete bien

V.3) Règles de 3, de 5 et de 0

- C'est un truc de programmeur C++98 comme le DRY/SPOT pour éviter les erreurs de conceptions sachant que le compilo crée des méthodes spéciales quand ça lui chante
- La règle est : si une classe définie l'une des trois méthodes suivantes, elle doit définir les trois: destructor, copy constructor, copy assignment operator
  - On a besoin de ces méthodes quand y'a un état non scalaire, comme des choses sur le tas ou des fichiers ouverts.
- Avec le C++11, on complète la règle avec deux méthodes supplémentaires: move constructor, move assignment operator qui ne font pas une copie profonde, mais détruisent l'objet de départ
- La règle de 0 dit que les classes qui ont l'une des 5 méthodes ne devraient avoir que celles-ci. Separation of concern, elles ne s'occupent que de la gestion de la mémoire.
  - Les autres méthodes ne devraient avoir aucune des 5 méthodes (elles sont marquées `=delete`).
  - Mais bon, c'est un mantra de programmeur, faut pas en faire une religion.

Chapter 10

Programmation fonctionnelle

- C'est le troisième paradigme de programmation accessible en C++: on a déjà parlé au cours 2:
  - Procédural C: découpe en procédure ayant des effets de bord; les données sont des globales
  - Orienté Objet: découpe en entités regroupant une partie de l'état global et les traitements qui s'y appliquent
  - Fonctionnel: déclaratif centré sur les traitements (inspiré des maths) Evaluation d'expressions plutôt qu'exécution de commandes
- Chaque approche a des avantages et des défauts:
  - Procédural proche de la réalité de l'ordinateur (M99), mais peu d'abstraction et pas de séparation
  - OO plus facile à concevoir pour certain (des choses font des actions), mais séparation de surface seulement (catastrophe en multithread). Tendance au bloat.
  - FP donne des codes plus courts et épurés, mais plus dur à écrire, et tendance au code golf impénétrable.
- Quelques composants qu'on retrouve souvent en FP
  - Fonction de premier ordre : manipuler des données de type "fonction", et currification
  - Programmation d'ordre supérieur: on passe des fonctions en paramètres d'autre fonctions
  - Données immutables pour moins de problèmes (surtout en matière de concurrence)
  - Fonctions pures (sans effet de bord, comme les fonctions maths, pour donner tjs le meme résultat quand on leur donne les même données)
  - Type algébriques et pattern matching (ça c'est très dur en C++)
- Nous allons voir comment ces choses sont possibles en c++ moderne

I) Programmation d'ordre supérieure

- En C, il faut faire des pointeurs sur fonction, et c'est très facile de faire n'importe quoi car le compilateur n'aide aucunement

- En C++, on a un type `std::function` qui permet de manipuler des choses invocable comme une fonction.
  - Toute classe dotée d'un opérateur d'invocation `ret operator()()` peut être utilisé comme une fonction.
  - Le type (pour une variable ou un paramètre) est comme `std::function<bool(int, Mytype)>`
- Cela se combine bien avec les algorithmes de la STL, comme `std::for_each` qui correspond à un map (cf l'exemple de code).
  - Il vaut mieux prendre l'habitude d'utiliser `std::begin(vect)` que `vect.begin()` car le premier fonctionne également avec les tableaux

I.1) Lambda

- Avantages des lambdas : moins de sucre syntaxique à écrire; pas de fonction à usage unique dont le nom pollue l'espace de nommage
- `[capture list] (parameter list) {function body}` : la tête, les paramètres, et le corps
  - La liste des captures donne les variables du contexte où se trouve la def de la lambda qui doivent être rendues disponibles à l'endroit où la lambda sera utilisée
  - On peut capturer par valeur, comme ici, ou par référence avec `[&i, &d]`.
  - On peut aussi capturer toutes les variables locales par copie avec `[=]`, ou capturer toutes les variables locales par référence avec `[&]`
  - Si on est dans une méthode de classe, on a probablement envie de capturer `this` aussi, car sinon la lambda ne s'exécute pas dans le contexte de la classe.
- Le mécanisme de capture n'est pas magique, le compilateur génère juste une classe à usage unique. Les variables capturées sont des champs de la classe (initialisés correctement par le constructeur) et donc accessible par l'opérateur d'évaluation `operator()()`.
  - Cela donne l'impression que les symboles utilisés par le corps de lambda **sont** les variables capturées (même nom et tout), mais en fait c'est une copie.
  - D'ailleurs, si la valeur des variables capturées change entre temps, la lambda ne le verra pas lors de son exécution
  - Si on veut que la lambda modifie les paramètres capturés, il faut le dire avec `mutable` à écrire juste avant le corps de fonction (là où dans d'autres contextes, on écrit `const`)
- Il faut parfois préciser le type de retour quand le compilateur n'arrive pas à déduire les types statiques `[capture list] (parameter list) -> type_retour {function body}`
  - On peut tjs post-fixer ainsi le type de retour (`auto fun() -> type`), mais je trouve que ça alourdit
- En C++14, on peut faire des lambda généralisées, c'est à dire des fonctions qui s'applique à tous les objets sans savoir leur type. Cf l'exemple de code.
- Au final, c'est très pratique de pouvoir stocker des fonctions dans des variables. Trop dommage que les lambda python soient limitées à une seule ligne à cause du manque d'accolades.

I.2) Currification

- Création d'une nouvelle fonction par application partielle des premiers paramètres d'une fonction.
  - C'est une idée de théoricien qui s'avère parfois utile, et c'est surtout très facile à faire en C++.
  - Soit on fait la lambda à la main, soit on utilise `std::bind`
- On peut aussi changer l'ordre des paramètres au passage avec les `_1` qui sont des placeholders des différents paramètres reçus par `std::bind`
- L'exemple classique est le générateur aléatoire, présenté sur la feuille, mais ça marche aussi avec le contexte d'exécution d'une requête sur une base de données ou autre

II) Données immutables et fonctions pures

- `const` indique qu'une variable est constante, ou qu'une fonction ne modifie pas ses paramètres.
  - Informer le compilateur lui permet d'optimiser et surtout de nous aider à chasser nos erreurs
- On dit qu'une telle fonction est **référentiellement transparente** puisque l'environnement ne voit pas la différence quand on remplace l'invocation de la méthode par la valeur qu'elle va retourner.
  - Il n'y a donc pas d'effet de bord (affichage, modification de l'état global, écriture sur disque)
  - Pour qu'un programme soit intégralement référentiellement transparent, il faut que les variables soient constantes après affectation. On peut alors remplacer leur nom par leur valeur.
- **Expression constante**: `constexpr`
  - ce décorateur sur une variable -> valeur connue à la compil
  - sur une fonction -> valeur connue à la compil SI les inputs le sont. Sinon c'est juste une const
- Programmer sans aucun effet de bord est très limité. Le programme ne peut pas être interactif.
  - Il faut quand même réduire le nb d'effets de bord pour simplifier la compréhension du prog
  - On peut faire un jeu complet du même genre que le projet, mais avec une seule variable mutable : l'état du monde. A chaque itération, on remplace le monde actuel par le suivant, qui est une copie: `world = update(world)`. On peut même le faire efficacement, et ça garantie l'absence de plusieurs catégories de bugs.

III) Types algébriques

- c'est pour définir des types sommes: soit ça, soit ça. Comme par exemple soit une référence vers un objet valide, soit le pointeur NULL. Ou encore une valeur python, qui

est soit un entier, soit une chaine de caractères soit un nombre à virgule.
  - En C, on va faire des unions mais c'est très bas niveau et sans aucun garde fou. Si on a un entier et une chaine; on modifie l'entier, quelle proba pour que la chaine soit valide? Ca serait pire en C++: si j'ai deux champs dont un seul est valide à la fois, que faire dans le destructeur?
- `std::optional` (C++17): Une valeur valide, ou rien de bon (cf exemple)
  - Le constructeur sans paramètre fait une valeur invalide, que l'on peut créer explicitement avec `std::nullopt`
  - Constructeur avec paramètre fait une valeur valide, qui peut rendre le type du contenu déductible
  - `make_optional` permet d'utiliser auto ou de le passer en paramètre
  - Il y a une conversion implicite du type contenu vers l'optionnel valide
  - Les opérateurs de déréférencement sont invalides si la valeur l'est ! Il faut uitliser `value()` ou `value_or()` pour faire la vérification de validité
    * Damn besoin d'optimisation du C++
  - `https://www.bfilipek.com/2018/05/using-optional.html`
- `std::variant` (C++17):
  - C'est une généralisation où on peut faire la somme de plusieurs types valides.
  - Assez facile à créer, pas forcément pratique à utiliser car `get_if` travaille sur des pointeurs
  - On peut même faire la forme simplifiée du pattern matching (branchement structurel sur le type, sans considération des valeurs) avec cet objet et des fonctions lambda. Mais on va pas se mentir, la syntaxe est moche comme du C++. Il y a des propositions pour faire du branchement sur les valeurs, mais les syntaxes deviennent trop vilaines pour rentrer dans le standard C++ (c'est dire)

IV) Les itérateurs

- On les utilise depuis longtemps de façon implicite pour parcourir les conteneurs de la STL. On va écrire un itérateur simple en TP.
- A l'usage, on demande au conteneur pour obtenir un itérateur. Il y a plusieurs types (selon qu'ils permettent de modifier la collection ou juste de lire, qu'il sont à usage unique, et le degré de liberté dans l'ordre de parcours), et chaque collection de la STL en offre plusieurs:
  - begin/end: début et fin
  - cbegin/cend: idem mais ce sont des itérateurs constants que je ne peux pas modifier
  - rbegin/rend: itérateurs en sens inverse, ie en commençant par la fin
  - crbegin/crend: constant et sens inverse
- Attention, certaines opérations invalident les itérateurs (entre autres toutes les modifications de la collection). Cf la doc.
- Simple à implémenter avec les opérateurs:
  - `operator*` retourne l'objet derrière l'itérateur en ce moment

  - `operator++` (et éventuellement `operator--`) décalent l'itérateur d'un élément
  - `operator==` et `operator!=` comparent les opérateurs entre eux (pour savoir si le parcours est terminé). Pour comparer les éléménts, faut d'abord déréférencer
  - `operator=` décale l'opérateur. Pour assigner une nouvelle valeur à l'emplacement pointé par l'itérateur, faut dérérencer.
- Il faut apporter des informations en plus pour pouvoir les utiliser dans la STL (par exemple expliquer comment ajouter des choses à la collection à l'endroit de l'itérateur). L'implémentation n'est ni compliquée ni très élégante. Vous irez chercher sur internet quand vous aurez besoin, mais ça ne sera pas pour ce module.
  - `using iterator_category = std::forward_iterator_tag;`
  - `using difference_type = std::ptrdiff_t;`
  - `https://internalpointers.com/post/writing-custom-iterators-modern-cpp`
  - `https://users.cs.northwestern.edu/~riesbeck/programming/c++/stl-iterator-defi` `html`

V) Ranges et views

- La STL offre des algorithmes sur les conteneurs, ce qui était très en avance sur son temps
  - map -> `std::for_each(std::begin(v), std::end(v), OP)`
  - filter -> `std::remove_if`
  - reduce/foldLeft -> `std::accumulate(std::next(std::begin(v)), std::end(v), v[0], OP)`
  - foldRight -> `std::accumulate(std::next(v.rbegin()), v.rend(), v.back(), OP)`
- Mais la syntaxe est assez pénible (il faut donner deux itérateurs – début et fin), et on peut pas combiner, chaîner les opérations comme en shell
- Les ranges et les vues de C++20 permettent de faire ça très simplement
  - Un range est simplement un itérateur que l'on peut parcourir
  - Une vue est une opération sur un range. La vue ne possède pas les données, qui restent propriété du range. C'est assez classique, conceptuellement
  - On peut combiner très simplement, et on peut même utiliser le fait que les vues sont paresseuses à l'évaluation. Cf le code proposé.
  - `std::views::iota(N)` retourne l'infinité des nombres à partir de `N`
  - `std::views::take(N)` ne garde que les `N` premiers éléments de la vue
  - Quelques vues définies:
    * `all()` (tous les éléments)
    * `filter(pred)` (ce qui vérifie un prédicat booleen)
    * `transform(lambda)` (pour faire un map)
    * `take(N)` (prend que les N premiers), `take_while(pred)` (prend tand que le prédicat dit vrai)

  - `drop(N)` (jette les N premiers) `drop_while(pred)` (jette tand que le prédicat dit vrai)
  - `join()`, `split`, `common` (filtrage inter-vues), `reverse`
  - `element(i)` (les Iiemes elements des tuples), `keys()`, `values()`
- Le gros défaut est que cette partie du standard C++20 n'est pas encore inclu dans les compilateurs en ce début d'année 2021.
  - Il faut donc utiliser la bibliothèque `range-v3`, qui est inclu dans toute les bonnes distributions
  - Il faut aussi compiler avec `--std=c++20`
  - En plus, le zip n'est pas dans C++20 pour de sombres raisons techniques et il faudra attendre C++23 pour l'avoir par défaut. `range-v3` a de beaux jours devant elle.

VI) Éléments FP absents du C++

- Évaluation paresseuse: l'évaluation C++ est stricte (=non paresseuse), bien sûr, mais on peut facilement implémenter des systèmes de memoisation comme on verra en TP.
- le pattern matching structurel ne fait pas partie du langage. Il y a des tentatives de qqch, mais il faut savoir être raisonnable et passer au Rust après un moment
- Le système de type est mieux que le C, mais c'est encore loin de l'isomorphisme de Curry–Howard (qui dit qu'un prog bien typé en Coq est isomorphe à une preuve). Là encore, Rust est l'étape suivante, même si je ne sais pas si c'est la dernière étape du chemin.
- En CC, on peut faire beaucoup (mais pas tout) le FP en C++. Mais la syntaxe est parfois rebutante. Et encore, vous avez pas vu les messages d'erreur que ça produit, punaise.

Chapter 11

Programmation générique

- La programmation générique est quand on écrit des fonctions qui peuvent s'appliquer à tout type de données, indifféremment.
- Habituellement, c'est limité aux langages interprétés, mais C++ permet de faire ça.
- On a déjà croisé des templates de méthode au premier cours, pour écrire une fonction `min` utilisable pour tous les types ayant `operator<()`
- On peut comprendre ces choses comme une extension du préprocesseur, où du code est produit en cas de besoin lors de la compilation.
- Le compilo réalise là où la template est utilisée qu'il lui faut générer du code supplémentaire.
  - C'est assez troublant car ça veut dire qu'on a parfois un code qui fonctionne, puis quand l'appelant fait une instanciation qui n'était pas testée avant, ca fait une nouvelle erreur sur le code de la template.
  - Mais cela permet de belles choses, y compris la STL qui veut dire Standard Template Library.
- STL = Standard Template Library, par Alexander Stepanov qui s'intéresse à la prog générique dès 1979, intégré au standard C++ dès l'origine. Un peu par hasard.
  - decltype
  - Sémantique des types de retour
  - Templates récursives: juste dire que c'est possible et que ça ouvre la porte

I) Patrons de classe

- On a un début d'exemple sur la feuille. C'est assez simple à comprendre : on a pas précisé le type des champs stockés, ce qu'on fera à l'usage
- On peut avoir plusieurs paramètres
- On utilise des patrons de classe depuis longtemps. Ex: `std::vector<int>`
- ultra puissant, mais source de messages d'erreurs incompréhensibles

II) Concepts C++20

- Pour l'instant, on ne contrôle pas trop ce qui rentre dans les templates. On peut faire des `const_assert` sur les traits des types comme dans le code présenté, mais c'est assez limité. Les `const_assert` sont très bien en général, mais pas adapté à ce cas.
- Avec C++20, on pourra poser des contraintes telles que "Comparable" ou "Decrementable" sur les types, et même déclarer ses propres contraintes en imposant aux types d'avoir des fonctions du nom+profil souhaité.

Chapter 12

Extra: Paradigmes de programmation

- C++ a été inventé pour apporter la POO au C très procédural dans l'âme (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 la FP en C++ reste un peu capilotractée.
- Mais de quoi on parle en fait ? Retour sur les paradigmes de programmation classiques
  - Les différences portent sur la façon de découper les données et opérations, leur organisation, ce sur quoi on se focalise, etc.
  - 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.

I) Programmation impérative vs. programmation déclarative/logique

- Différence fondamentale de comment on utilise l'ordinateur: soit on lui dit comment faire, soit on lui dit seulement quoi faire.

I.1) Exemple de langage impératif: Langage C

- impératif = on détaille les traitements pas à pas, l'ordinateur n'a aucune capacité d'initiative. Modèle assez basique.
- Avantages: proche de la machine donc potentiellement efficace (quand le programmeur est bon) et populaire.
- Désavantages:
  - La structure évolue vite vers un vilain code spagetti impossible à faire évoluer si on n'y prend pas garde.

  - L'intention du programme est implicite, il faut abstraire l'écrit pour le trouver, ce qui est hors de portée d'un outil d'analyse
  - Les preuves et debug très difficiles sur ce genre de programme.

I.2) Exemple de langage déclaratif: Prolog ou TLA$^+$

- En impérative, on dit donc quoi faire à l'ordi, pas à pas; En déclarative, on donne des éléments de réponses et un objectif, puis l'ordi se débrouille.
- 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

II) Procédural vs. POO vs. Prog Fonctionnelle

- La différence porte sur l'organisation des opérations
- **Programmation procédurale**: impératif où les traitements sont découpés en procédures ayant des effets de bord sur 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
  - Difficile d'abstraire quoi que ce soit: puisque tout le monde peut modifier les globales, il faut considérer chaque bout à chaque instant et l'ordre des opérations est important.
  - On a arrêté les GOTO dans les années 2000, il faudrait arrêter ça aussi maintenant.
- **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)
- **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
- **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 décennies 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

III) Principes de la POO

III.1) Vocabulaire: classe vs. instance

- En C++, les objets sont des instances de classes prédéfinies.
  - En SmallTalk, tout est objet (même le nombre 42), tout est dynamique. On peut modifier des objets à la volée et on crée les nouveaux en clonant les objets existants
  - Mais C++ vise l'efficacité: le compilateur doit connaître les classes avant la compilation pour optimiser et on instancie depuis les classes
- Vocabulaire
  - Les fonctions associées à un objet s'appellent *méthodes*
  - En smalltalk, on dit qu'on *envoie un message* à un objet pour dire qu'on appelle l'une de ses méthodes. Ca doit être l'influence du Simula, le premier langage OOP, qui était destiné à simuler des systèmes distribués.
  - Le code appelant d'un objet est parfois appelé *code client* (sans doute car SmallTalk pousse l'idée de programmation par échange de messages assez loin)
  - L'objet n'est pas appelé serveur mais plutôt *receveur*, pour une raison qui m'échappe.

III.2) Premier principe de POO: encapsulation (pour ranger ses affaires)

- Les données sont cachées, accessibles uniquement à travers une interface publique controlée
- Découpe du problème: spécification de différents objets qui collaborent entre eux, répartition des responsabilités entre eux
- Contrairement à la croyance populaire, pouvoir modifier l'implem sans changer les clients est un avantage secondaire. Cela permet de réduire le **couplage** entre les bouts de code.
- On veut pouvoir réfléchir à chaque partie séparément (implem, test à fond), puis de réfléchir aux interactions en oubliant complétement les implémentations (ce sont de bons outils conceptuels).
  - Réduire la charge mentale quand on considère le projet dans sa globalité
  - Permet de plus de découper le travail entre plusieurs équipes (ça marche rarement)
- **Adhérence** entre modules = résistance des autres modules à la modification d'un module donné.
- **Cohésion** d'un module = auto-suffisant et atomique

III.3) Deuxième principe: héritage (pour factoriser du code):

- On peut spécialiser une classe pour rafiner son implémentation.
- Par exemple, une voiture est un véhicule à moteur avec 5 places, et un vélo est un véhicule manuel à une place.
  - Si j'arrive à trouver des éléments en commun à tous les véhicules, la notion d'héritage me permettera de n'écrire le code correspondant qu'une seule fois
- Il y a souvent plusieurs solutions possibles, plus ou moins pratiques à mettre en oeuvre

- Sol 1: outil -> [pour tourner -> [visseuse / tournevis manuel] / pour taper [marteau / percuteur]]
- Sol 2: outil -> [manuel -> [tournevis / marteau] / éléctrique -> [visseuse / percuteur]]
- Il n'y a pas de réponse juste et fausse en matière de design, c'est souvent une question de goût

III.4) Troisième principe: polymorphisme/liaison dynamique (simplification de l'usage)

- La méthode utilisée dépend du type du receveur
- En tant qu'utilisateur, peu importe comment l'objet fait ce que je lui demande.
- Boite noire = réduction de la complexité par abstraction des détails
- En python, c'est du *duck typing*. Pour savoir si c'est de type "canard", je regarde s'il sait faire "quack" comme un canard, c'est tout.

IV) Conclusion sur l'OOP pour l'instant

- C'est avant tout une façon de ranger son code orienté sur la façon de ranger ses données. C'est pour ça que c'est troublant pour les prépa qui n'ont vu aucune structure de données avant la spé (sauf erreur de ma part). Il faut aussi reconnaitre qu'il y a un peu de blabla voire de religiosité dans les discussions autour du design logiciel, mais faut pas se priver de cet outil bien pratique pour autant.
- Ce qui est au coeur du problème, ce sont les façons de (1) décomposer les problèmes (2) exprimer l'extensibilité du code
  - Impératif/procédural vs. fonctionnelle porte sur la décomposition
  - Dans la POO, l'encapsulation est une façon de décomposer tandis que l'héritage et le polymorphisme sont des façons d'exprimer l'extensibilité.
  - Parmi les autres façons d'exprimer l'extensibilité, citons les typeclass de Haskell, qui permettent de poser des contraintes sur un type, genre "on sait multiplier deux éléments avec l'opérateur * et ca fait un nouvel élément de ce type là, et on sait aussi appliquer l'opérateur < à deux éléments pour un résultat booléen".
    - Les typeclass sont moins error-prone car ils utilisent le système de types pour vérifier l'absence d'erreurs.
    - Y'en a en C++, on en parlera peut-être à la fin. Rust implémente une forme réduite des typeclasses sous le nom de Trait. Y'en a pas en CAM
- La suite du programme, c'est
  - Une vision de comment utiliser l'encapsulation en pratique en C++ (syntaxe, langage, idiomatismes)
  - Un petit tuto sur comment découper son code en objets en n'utilisant que l'encapsulation (fin du cours et TD)
  - Un TDP sur l'encapsulation et le découpage de code
  - La semaine suivante: on revient sur l'héritage et des notions avancées d'OOP

Chapter 13

Conclusion sur le module

- L'objectif est de vous apprendre à faire des programmes non triviaux
  - Dans le projet, l'objectif était de réécrire le code à chaque question pour l'adapter aux nouvelles demandes
  - On ne va pas faire de vous des ingénieurs, mais les meilleurs chercheurs en biologie sont capables de remplacer leurs laborantins en cas de besoin.
  - Rapport à la recherche: usage. Je ne pense pas qu'il y ait des chercheurs en C++
- On a choisi le C++ pour cet objectif car c'est le langage roi
  - Il offre tous les paradigmes, avec des performances difficilement égalables
  - C'est le plus utilisé pour les tâches d'ingénierie compliquées
  - Il a un caractère de cochon. Quand on sait l'utiliser, on sait utiliser les autres.
- Place de ce module dans le cursus:
  - Prérequis de ce module: Le C pour les aspects pratiques, la théorie des langages de prog pour les réflexions sur les paradigmes
  - Suite du module: sorte de prérequis pour les modules utilisant Java comme langage (Jézéquel en M1)
  - Ce qui manque: méthodes de conception (merise/TDD), écrire des tests, programmation concurrente
- L'examen:
  - Feuille de pompe A4 recto verso de votre main
  - Très proche de ce qu'on a fait en TD/TP/Projet (écriture de petits codes sur papier, lecture de code fourni pour faire des schémas UML de l'organisation et/ou le schéma mémoire du tas et de la pile)

.1) 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

1. 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 mauvaise idée: attirant mais piégeux car pas inclu dans le langage. Comme un langage dans le langage.
   - Syntaxiquement, 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 sémantiquement, les deux langages sont très différents. On dirait que le compilo C++ sait compiler du C *en plus* de compiler du C++, pour simplifier l'adoption à l'époque.
     - Un programme idiomatique du C++ est à des années lumieres du C, car C++ apporte énormément d'autres constructions au langage.
   - Différence philosophique entre les langages : face à un nouveau pb, C ajoute du code utilisateur; C++ ajoute des fonctionnalités au langage. Le C est plus stable et il va rester encore longtemps, tandis qu'on se demande tjs quand C++ va s'écrouler sous son poids.
   - Point commun : le compilateur fait confiance à l'humain, et produit des messages d'erreurs abscons
     - Non seulement C++ a confiance en vous, mais il vous donne les pleins pouvoirs en vous permettant de changer les comportements par défaut.
     - Par exemple, gestion mémoire (allocators) dans les collections d'objets
     - Ca rend le langage encore plus compliqué à bien utiliser.

2. Java is a better C++
   - 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.
   - 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)

3. Rust is a better C++
   - Borrow checker permet d'éliminer les 3/4 des pbs de mémoire en interdisant les use after free, les double free et les memleaks (C++ sait aussi résoudre les memleaks,

cf topic 4)

- Systeme de typage vraiment utile, comme en haskell and co. Les types linéaires permettent de finaliser la gestion de la mémoire sans garbage collection ni ref-counting. (mais complique compilo)
- Tout ceci donne des garanties formelles sur les fuites mémoire sans GC, plus de dandling pointers
- Un peu dur à maitriser au premier abord. D'ailleurs c'est le langage préféré de tout le monde, mais sans qu'on observe une déferlante d'usages, au final.

4. D'autres langages dans les cartons
   - La plupart des langages cités ci dessous sont issus de la réalité terrain: soit les grosses boites, soit les hobbyists éclairés.
   - Je ne parle pas de la multitude de langage issus de la recherche car il y en a trop, et parce que faire émerger un langage utilisé en pratique demande plus d'efforts que ce que peut faire une équipe de recherche.
     - Mais il y a des exceptions: CAML sort des labos français, ainsi que Pharo qui mérite d'être connu (descendant de SmallTalk et autres langages OO de qualité).
     - Cyclone de Cornell a introduit les belles idées qu'on apprécie en Rust.
5. C3 is a better C
   - Rester proche du C en apportant un peu de modernité, mais sans "grande idée". Compatible ABI C
   - Modules, génerics, struct subtyping
   - highlevel containers, string, LLVM
   - Zero cost error handling
6. Zig is a better C
   - pas de flot de contrôle caché, ni malloc caché.
   - Pas de preprocesseur ni templates (version C++ des macros). Metaprogrammation = exécution de code à compil time et manipulation des types comme first class
   - Simplicité d'écriture, tests intégrés, makefile intégré, Erreurs intégrées sans cacher de control flow (donc pas d'exception)
   - Des objets, mais pas d'héritage. On ne peut pas étendre les objets ni définir d'interface. A la place, les fonctions prennent des paramètres sans type ('anytype') et font du ducktyping sur le paramètre. C'est à dire qu'elles invoquent la méthode de leur choix sur l'objet. Si le compilateur arrive à satisfaire ça avec le type effectif passé, c'est bon. Si non, erreur de compilation. C'est une forme de programmation générique (qui demande de la monomorphisation ie de la génération de code spécifique comme font les templates) assez originale. Il est rare qu'un langage fasse autant d'effort à faire disparaître les types. À part Python, Perl et d'autres scripts, ca doit venir de l'héritage Lisp. Pas sûr, je connais mal.
   - Autohébergé (compilo écrit en zig) depuis peu.

  - Bootstrapping inventif grâce à WASM: le compilo sait générer du code pour une version simpifiée de la VM WASM nécessitant 5k lignes de C, qui est donc la base à porter sur une nouvelle archi pour porter l'ensemble du compilo. Il faut aussi générer du code pour la nouvelle cible, bien sûr.
  - Compilo de complexité bien moindre que Rust: 250k lignes.

7. Hare is a better C
   - Langage minimaliste visant la simplicité et la durabilité. L'objectif est que le compilateur Hare 1.0 compile du code écrit dans 50 ou 100 ans. Plus fort que la rétrocompatibilité
   - On y parvient en simplifiant les choses, en réduisant la taille de la spécification. D'ailleurs, il y a une spec pour espérer des réimplémentations, ce qui est une bonne stratégie pour simplifier et clarifier ce qu'on peut attendre du langage.
   - Malloc explicites (pour l'instant, ils espèrent ajouter un borrow checker), blocs defer pour le nettoyage
     - gestion des erreurs transparentes avec possibilité de renvoyer un type somme "résultat ou erreur" et obligation de gérer/propager explicitement les erreurs des fonctions appelées (! au type de retour pour dire qu'il peut y avoir une erreur, ? pour invoquer une fonction en propageant l'erreur, et vérification par le compilo que rien n'est oublié)
     - Sous-typage de structure pour définir des interfaces; Unions typées incluses dans le langage; matching un peu plus puissant que le switch C mais moins qu'en Rust.
   - Hare fait confiance à l'humain, mais pas par défaut. Les pointeurs nuls sont interdits sans le mot-clé "nullable", les tableaux sont bound-checked sauf si on demande à ne pas le faire, etc.
   - C'est un langage pragmatique et simple pour implémenter un OS ambitieux (micronoyau Sel4-like où tout est capability; userland prévu inspiré de Plan9)
     - Les mauvaises langues diront que c'est comme Rust en plus moche, mais je dirais que c'est comme Rust qui ne risque pas de s'écrouler sous son poids dans 15 ans.

8. Carbon is a better C++
   - C'est un langage poussé par google pour remplacer/corriger C++. Mais c'est du vaporware pour l'instant

9. Swift is a better C++
   - Langage Apple que certains rapprochent du Rust. J'ai pas regardé de près, encore.

.2) Conclusion

- Tout ca pour insister sur le coté central des langages C et C++ dans le paysage informatique. Souvent challengés, parfois dépassés, jamais remplacés.

Notes d'animation

Séance 1: Programmer en C++ / Better C

- **Exemple de code à imprimer**
- **Programme du jour:** (à utiliser séance suivante en rappel)
  - Histoire du C++
    - Inventé pour être un meilleur C, et pour permettre la programmation orientée objets (entre autres)
    - Langage qui évolue encore (C++14/17 dans le cadre de ce cours, voire C++20 par moments)
  - En pratique
    - `std::string`
    - les flux à la place de printf
    - ajouts du langage: namespaces, surcharge fonctions et opérateurs, références vs. pointeurs, itérateurs et boucles étendues
    - La bibliothèque standard: conteneurs, itérateurs et algorithmes
    - Programmation générique avec les templates qui sont une forme évoluée du préprocesseur (irk)
  - La programmation orientée objet est une façon de ranger son code
    - On s'invente des outils pour décomposer le pb. Outil = objet qui connaît des choses et sait faire des choses
    - Par opposition à l'autre paradigme de programmation dominant de nos jours : le fonctionnel, qui met l'accent sur les verbes et non sur les noms.
- **TODO:**
  - Il serait bon de terminer la partie sur la programmation en C++ le premier jour, donc faut faire plus court sur la partie culture G et comparaison des langages.

Séance 2: POO: encapsulation

- **Exemple de code à imprimer**
- **Programme du jour:**
  - L'encapsulation C++ en pratique
    - classe, constructeur et destructeur. Initialisation de champs

    * champs statiques, champs constants. Méthode statique, constante.
    * Visibilité: private/public
    * Surcharge d'opérateurs
  - Gestion de la mémoire en C++
    * Objets statiques, dynamiques, automatiques et temporaires
    * Opérateur de copie
    * Initialisation et copie implicite
    * Tableaux d'objets
- **TODO**
  - Voir aussi: `https://programmation-orientee-objet.pages.ensimag.fr/poo/resources fiches/05-Polymorphisme_liaisonDynamique/`
  - TODO: ajouter quelques mots sur la représentation mémoire de l'héritage (p452 du reader de Stanford). Ca explique bien pourquoi on peut pas spécialiser une affectation sans pointeurs.

Séance 3: POO: Héritage et polymorphisme

- **Exemple de code à imprimer**
- **Programme du jour**
  - Héritage
    * Principe: factoriser du code, comme du copy/colle en mieux
    * Cacher l'arbre d'héritage est une bonne idée
    * Vocabulaire: mere/fille, spécialise/dérive/généralise
    * transtypage: surtout any `static_cast` et any `dynamic_cast`
  - Polymorphisme
    * Principe de substitution de Liskov
    * Redéfinition méthode (override) dans fille != surcharge (overload) autre prototype
    * Liaison dynamique (type statique, type dynamique)
    * classe abstraite
    * Héritage multiple
  - Design OOP, et UML
    * Composition (a des): flèche à tête de coté propriétaire
    * association (réciproque): trait avec la quantité en légende
    * Héritage (est un): flèche fille->mère
  - Implémentation de l'héritage en mémoire
  - Exceptions
    * Pas de new, pas de finally, on catch par référence

Séance 4: Modern C++

- **Exemple de code à imprimer**
- **Programme du jour**
  - Implémentation C++
    * Représentation mémoire des objets triviaux: comme des structures C
    * Objets non triviaux = présence d'une vtable pour les méthodes virtuelles
    * Transtypage: Assez similaire au C
      · Objet (comme scalaires): perte d'information (zone mémoire plus courte, l'autre vtable est utilisée) donc pas de upcasting
      · Pointeurs d'objet: ne change que la façon d'interpréter la zone pointée
    * Mangling de symboles: règles de mutilation pour encoder les namespaces et types de paramètres dans le nom de symbole C
    * Invariance du C++: les redéfinitions ne peuvent pas raffiner le type des params ou retour
  - Gestion automatique de la mémoire:
    * RAII
    * smart pointers: un objet englobant gère la mémoire d'un pointeur protégé.
      · `uniq_ptr` n'a pas d'opérateur de copie: `operator=(T t)=delete;`
      · `shared_ptr` fait le refcounting dans son opérateur de copie et son destructeur

Séance 5:

- **Exemple de code à imprimer**
- **Programme du jour**
  –
- **TODO:** conseils à placer sur une feuille de TD
  – préférer les initialisations avec des accolades
  – marquer "override" les fonctions redéfinies
  – La plupart des conseils du livre de Scott Meyer s'y prete bien