Programmation Orientée Objet : Compilation séparée

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

Objectifs de la leçon d'aujourd'hui

- Présenter le cours et son déroulement
- Brefs rappels
  (**ATTENTION !** à tou(te)s celles et ceux qui n'ont plus fait de programmation depuis 1 an :
  s'y remettre très vite !)
- Compilation séparée
- (namespaces)

Objectifs du cours

1. Consolider votre apprentissage de la programmation du 1er semestre
2. Apprendre la **programmation orientée-objet**
   ☞ concrètement, la pratique se fera en C++

Même fonctionnement qu'au 1er semestre :

- base de cours en vidéo sur le MOOC
  https://www.coursera.org/learn/programmation-orientee-objet-cpp/
- compléments de cours en classe
- quizz + exercices communs MOOC
- forums : (1) commun au MOOC (2) spécifique au cours dans Ed Discussion (via Moodle)

avec en plus ce semestre :

- un projet par binôme à rendre en fin de semestre

Organisation du travail (semaines)

- **AVANT** le cours : voir les vidéos
- avant le cours si possible : faire les quiz
- en cours (jeudis 8h15–9h00) : rappels, approfondissements, questions
- jeudis 9h15–11h00 : séance d'exercices avec assistants
  travail sur le projet (*après* avoir fait des exercices)
- après la séance d'exercices : encore *plus* d'exercices ; finir les quiz

Organisation du travail (semestre)

		MOOC	déc.	cours 1 h Jeudi 8-9	exercices 2 h Jeudi 9-11
1	22.02.24		0	Intro + compil. séparée	
2	29.02.24	1. Intro POO	0	Intro POO	
3	07.03.24	2. Constructeurs/Dest	0	Constructeurs	
4	14.03.24	3. Surcharge des opé	0	Surcharge	
5	21.03.24	4. Héritage	0	Héritage	
6	28.03.24	5. Polymorphisme	0	Polymorphisme 1	
-	11.04.24		-	vacances Pâques	
7	04.04.24		1	Polymorphisme 2 / Collections hétérogènes	
8	18.04.24		-	-	Série notée
9	25.04.24	6. Héritage multiple	2	Héritage multiple	
10	02.05.24	(7. Etude de cas)	-	Templates	
12	16.05.24		-	(Ascension)	
11	09.05.24		-	Structure de données abstraites ; Bibliothèques	
13	23.05.24	(7. Etude de cas)	-	Bibliothèques (fin) + Révisions	
14	30.05.24		-	-	Examen

Notes et examens

La note finale sera calculée de la façon suivante :

- Série notée (1h45) ⇨ coef. 1
- Examen théorique (1h45) ⇨ coef. 2
- Projet (en binôme) ⇨ coef. 3 [1]

Devoirs notés du MOOC :

- sont un très bon entrainement
- n'entrent pas dans le calcul de la note EPFL.

1. mais en aucun cas plus que 1.5 fois la moyenne individuelle (sous forme fractionnaire) ; c.-à-d. ne permet pas de passer si on a une moyenne individuelle hors projet strictement inférieure à 3.0 (= fraction 0.4).

Calcul des notes

Soit p_x le nombre de points obtenus à l'épreuve x sur un total maximal pour cette épreuve de t_x. La note publiée pour cette épreuve est alors l'arrondi suivant :

$$n_x = 1 - 0.25 \left\lfloor -20 \cdot \frac{p_x}{t_x} \right\rfloor$$

(0 en cas d'absence)

La note finale N est ensuite calculée **directement sur les points obtenus** (et non pas les notes intermédiaires) par

$$N = 1 - 0.25 \left\lfloor -20 \cdot \frac{\sum_x \theta_x (p_x / t_x)}{\sum_x \theta_x} \right\rfloor$$

où θ_x est le coefficient de l'épreuve x. , et

$$\frac{p_{\text{projet}}}{36} \leq 1.5 \left(\frac{1}{3} \frac{p_{\text{série notée}}}{t_{\text{série notée}}} + \frac{2}{3} \frac{p_{\text{examen}}}{t_{\text{examen}}} \right)$$

Dates importantes pour le semestre

- Série notée :

  Jeudi 18 avril

- Examen final :

  Jeudi 30 mai

- Projet :

  Inscription :

  au plus tard le jeudi 14 mars

  Rendu :

  au plus tard le dimanche 2 juin 23h59

Qu'avons nous vu en programmation ?

Programmer c'est décomposer une **tâche** à automatiser en une séquence d'instructions (**traitements**) et des **données**

Traitements	Données
Algorithmes	S.D.A.
Expressions & Opérateurs Structures de contrôle Fonctions	Variables Portée Chaînes de caractères Tableaux statiques Tableaux dynamiques Structures Pointeurs
Entrées/Sorties	

Fondamentaux

1. **déclarez avant d'utiliser**
   - variables

     ```
     int i;
     vector<double> v;
     ```

   - fonctions ☞ prototype

     ```
     double sin(double x);
     bool cherche_valeur(Listechainee l, Valeur v);
     ```

2. **modularisez** / décomposez / pensez « atomique »
   - 2.1 conception (qu'est ce qu'on veut ?)
   - 2.2 implémentation (comment ça se réalise ?)
   - 2.3 syntaxe (comment ça s'écrit ?)
   - 2.4 tests (où sont mes fautes, comment pourrais-je les tester ?)

☞ **Révisez/Reprennez cela au plus vite !**
Attention ! en particulier à tous ceux qui n'ont plus fait de programmation depuis 1 an ...

Approche modulaire

Jusqu'à maintenant vos programmes étaient écrits en une seule fois, dans un seul fichier.

Cette approche n'est pas réaliste pour des programmes plus conséquents, qui nécessitent partage de composants, maintenance séparée, réutilisation, ...

On préfère une **approche modulaire**,
c'est-à-dire une approche qui
décompose la tâche à résoudre en sous-tâches
implémentées sous la forme de **modules génériques**
(qui pourront être **réutilisés** dans d'autres contextes).

Chaque module correspond alors à une tâche ponctuelle, à un ensemble cohérent de données, à un concept de base, etc.

Utilité

☞ Pourquoi faire cela ?

- Pour rendre **réutilisable** : éviter de réinventer la roue à chaque fois

  La conception d'un programme doit tenir compte de deux aspects importants :
  - la **réutilisation** des objets/fonctions existants : bibliothèques logicielles (« libraries » en anglais) ;
    (les autres/passé ⟶ nous/présent)
  - la **réutilisabilité** des objets/fonctions nouvellement créés.
    (nous/présent ⟶ les autres/futur)
- Pour **maintenir** plus **facilement** : pas besoin de tout recompiler le jour où on corrige une erreur dans une (sous-...-sous-)fonction
- Pour pouvoir **développer** des programmes **indépendamment**, c'est-à-dire même si le code source n'est pas disponible
- **Distribuer des bibliothèques** logicielles (morceaux de code) sans en donner les codes sources (protection intellectuelle).
  Remarque : vous pouvez vous-même créer vos propres bibliothèques.

Conception modulaire

Concrètement, cela signifie que les types, structures de données et fonctions correspondant à un « concept de base » seront **regroupés dans un fichier** qui leur est propre.

Par exemple, on définira la structure `qcm` et ses fonctions dans un fichier, séparé de son utilisation.

☞ séparation des déclarations des objets de leur utilisation effective (dans un `main()`).

Concrètement, cela crée donc plusieurs fichiers séparés qu'il faudra regrouper (« lier ») en un tout pour faire un programme.

Exemple : exercice sur les QCM

Supposons que notre programme utilise trois « concepts » :

```
struct qcm { ... };

void affiche(const qcm& question);
int poser_question(const qcm& question);
...

void affiche(const qcm& question)
{
   ...
}

int poser_question(const qcm& question)
{
   ...
   i = demander_nombre(1, question.nb_reponses);
   ...
}
```

QCMs

```
int demander_nombre(int min, int max);

int demander_nombre(int a, int b) {
  ...
}
```

demander nombre

```
int main()
{
   qcm maquestion;

   ...
   poser_question(maquestion);
}
```

Programme principal

Compilation séparée

Le but est de séparer chacun de ces « concepts » dans un fichier séparé.

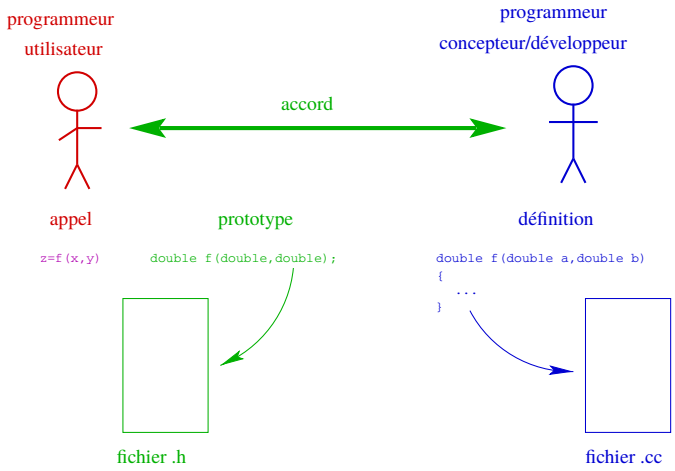
☞ Mais comment alors faire un tout (un programme complet) ?
Comment `main()` connait-il le reste ?
Comment les QCMs connaissent-ils `demander_nombre()` ?

La partie déclaration est la partie **visible** du module que l'on écrit, qui va permettre son utilisation (et donc sa réutilisation).

C'est elle qui est utile aux autres fichiers pour utiliser les objets déclarés.

La partie définition est l'implémentation du code correspondant et n'est pas directement nécessaire pour l'utilisateur du module.
Elle peut être **cachée** (aux autres).

Compilation séparée



Compilation séparée

De ce fait, il est nécessaire (en conception modulaire) de séparer *chacune* de ces parties en deux fichiers :

- les fichiers de déclaration (fichiers « *headers* »), avec une extension `.h`.
  Ce sont ces fichiers qu'on inclut en début de programme par la commande `#include`
- les fichiers de définitions (fichiers sources, avec une extension `.cc`)
  Ce sont ces fichiers que l'on compile pour créer du code exécutable.

A quoi sert donc un fichier `.h` ?

☞ A ce que les *autres* fichiers `.cc`, qui utilisent ce module, puissent compiler.

EPFL

Compilation séparée : exemple

```
struct qcm { ... };

void affiche(const qcm& question);
int poser_question(const qcm& question);
...

void affiche(const qcm& question)
{
   ...
}

int poser_question(const qcm& question)
{
   ...
   i = demander_nombre(1, question.nb_reponses);
   ...
}
```

```
#include "qcm.h"
#include "demander_nombre.h"
void affiche(const qcm& question)
{
   ...
}

int poser_question(const qcm& question)
{
   ...
   i = demander_nombre(1, question.nb_reponses);
   ...
}
```

qcm.cc

```
#pragma once

struct qcm { ... };

void affiche(const qcm& question);
int poser_question(const qcm& question);
...
```

qcm.h

EPFL

Compilation séparée (2)

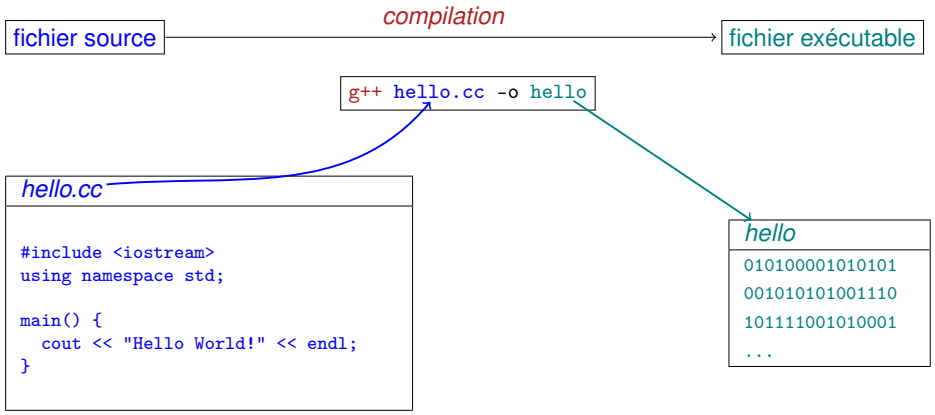
La séparation des parties déclaration et définition en deux fichiers permet une **compilation séparée** du programme complet :

- phase 1 (compilation) : production de fichiers binaires (appelés **fichiers objets**) correspondant à la compilation des fichiers sources (`.cc`) contenant les parties définitions (et dans lesquels on inclut (`#include`) les fichiers « *headers* » (`.h`) nécessaires) ;
- phase 2 (édition de liens) : production du fichier exécutable final à partir des fichiers objets et des éventuelles bibliothèques.

Note : pour un programme en N parties (`.cc`), on fait N fois la phase de compilation et 1 seule fois la phase d'édition de liens.

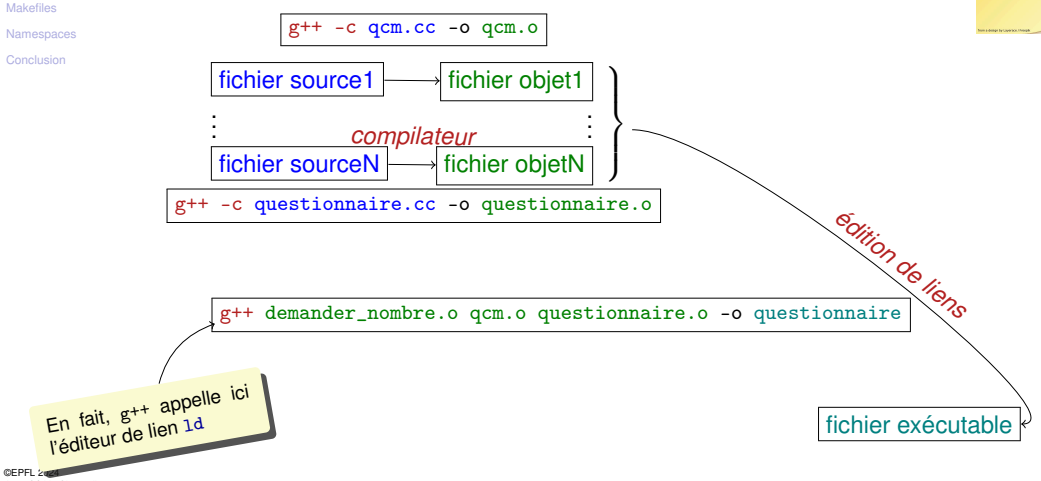
EPFL

RAPPEL : Compilation d'un programme C++



EPFL

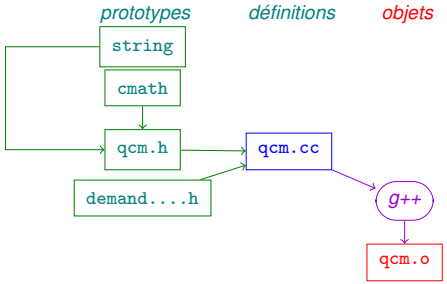
Compilation séparée d'un programme C++



Compilation séparée : phase 1

Pour créer un **fichier objet** (identifié par une extension `.o`), on utilise une commande de compilation du type :

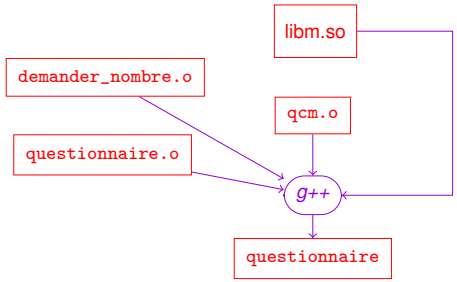
```
g++ qcm.cc -c -o qcm.o
```



Compilation séparée : phase 2

Le fichier exécutable est produit par une compilation qui **intègre** (« lie ») les fichiers objets nécessaires :

```
g++ demander_nombre.o qcm.o questionnaire.o -lm -o questionnaire
```



Règles de bonnes pratiques pour les `.h` et les `.cc`

1. Pour chaque fichier (`.cc` ou `.h`), pris/considéré *indépendemment* (= « pour lui-même ») :
   y mettre **tous** les `#include` dont *ce* fichier a besoin,
   et **uniquement** ceux dont *il* a besoin !

   Ni plus, ni moins !

2. **JAMAIS** de « `using namespace ...;` » dans un fichier `.h` !

3. Afin d'éviter les inclusions multiples, faire commencer (toute 1re ligne) chaque fichier `.h` par :
   `#pragma once`

Makefile (introduction)

Mais quand on a un grand nombre de modules, cela devient vite **fastidieux** de faire toutes ces compilations et ces liens...

...pour cela il y a des **moyens plus pratiques** dont les **Makefile**

Un `Makefile` est un fichier qui permet de construire facilement un projet en indiquant les composants et leurs dépendances.

(« `Makefile` » est vraiment le nom de ce fichier, sans extension `.qqchose` ; c'est juste un fichier texte.)

Une fois un `Makefile` constitué, pour réaliser l'exécutable correspondant au projet, il suffit de taper simplement `make`.

ou alors pour construire un programme particulier `cible` :
`make cible`.

Makefile (bases)

Un `Makefile` a une structure très simple : il est constitué d'un ensemble de règles décrivant les différents modules à faire et de quoi ils dépendent (« *liste de dépendances* »).

Une règle s'écrit :

```
but: liste de dépendances
```

Exemple :

```
questionnaire: demander_nombre.o qcm.o questionnaire.o
```

La première règle écrite dans le fichier `Makefile` permet de donner la liste de tous les exécutables que l'on veut créer ; par exemple :

```
all: questionnaire
```

Si on a des librairies système à utiliser, il faut les ajouter dans la variable `LDLIBS` au début du `Makefile` :

```
LDLIBS = -lm
```


Makefile (exemple simple)

Exemple (simple) complet (pour C++11 sur les VMs de l'Ecole) :

```
CXX = g++
CC = $(CXX)
CXXFLAGS = -std=c++11 -Wall
LDLIBS = -lm

all: questionnaire

questionnaire: demander_nombre.o qcm.o questionnaire.o

questionnaire.o: questionnaire.cc qcm.h
qcm.o: qcm.cc qcm.h demander_nombre.h
```


Makefile : remarques complémentaires

1. On peut ajouter d'autres options au compilateur avec la variable `CXXFLAGS`.
   Par exemple :
   `CXXFLAGS += -g`
2. On peut obtenir automatiquement les dépendances de compilation (c.-à-d. les dépendances des fichiers `.cc`) à l'aide de la commande :
   `g++ -MM *.cc`
3. `make` utilise des **règles implicites**.
   On peut donc exprimer encore beaucoup plus de choses dans un `Makefile`.
   Pour ceux qui veulent aller plus loin : voir la mini-référence.

Alternatives aux Makefiles

Il existe plusieurs outils pour générer automatiquement les `Makefiles` en fonction de la configuration de la machine.

Par exemple :

- les outils intégrés de développement de projets (IDE) : Code::Blocks, KDevelop, Anjuta, NetBeans, Eclipse CDT, ...
- CMake, `http://www.cmake.org/`,
- SCons, `http://www.scons.org/`, `http://progsv.epfl.ch/www/miniref/miniref-scons.html`
- gyp, `https://code.google.com/p/gyp/`,
- ninja, `http://martine.github.io/ninja/`,
- Jam (BJam, KJam, ...),
- the GNU Build Tools, alias « autotools » (`automake`, `autoconf` and `libtool`), cf `http://sourceware.org/autobook/`, `http://autotoolset.sourceforge.net/tutorial.html`

Conception modulaire

Compilation modulaire
⇒ séparation des prototypes (dans les fichier `.h`) des définitions (dans les fichiers `.cc`)

⇒ compilation séparée

1. Inclusion des prototypes nécessaires dans le code :
   `#include "header.h"`
2. Compilation vers un fichier « objet » (`.o`) : `g++ -c prog.cc`
3. Lien entre plusieurs objets :
   `g++ prog1.o prog2.o prog3.o -o monprog`

`Makefile` :
moyen utile pour décrire les dépendances entre modules d'un projet (et compiler automatiquement le projet)

Syntaxe : `cible: dependences`

Exemple :

```
examen: examen.o qcm.o
```


Espaces de noms

(Rappel) **Portée** d'un objet = région du programme où l'objet peut être utilisé

Exemples de portées : un bloc, le corps de fonction, tout le programme (variable globale), ...

☞ Qu'en est-il en cas de compilation séparée ?

Les portées locales restent inchangées (puisqu'elles sont « locales » par définition !)

Un **espace de noms** est justement un moyen de faire un regroupement logique de divers objets (variables, fonctions, ...)

Cela permet de partager des objets tout en **évitant les conflits** au niveau des noms...

...et donc de distinguer clairement deux objets portant le même nom, mais n'étant pas dans le même « espace de noms »

Espaces de noms (2)

Un espace de noms est simplement le nom donné à une portée : c'est l'espace regroupant tous les noms des objets dans cette portée.

On distingue :

- l'espace de noms global (qui a un nom vide) :
  c'est celui qui regroupe tous les objets déclaré en dehors de tout autre espace de noms
  les **variables globales** appartiennent par exemple à cet espace de noms
- les espaces de noms explicitement nommés
- les espaces de noms non nommés (ils n'ont pas de nom, même pas un nom vide !)
  Par exemple les blocs dans votre code (par exemple sous un `if`)

Définition d'un espace de noms nommé

```
namespace nom {
  ... corps de l'espace de noms ...
}
```

Exemple :

```
namespace outils {
  int compteur;
  double moyenne;

  int fonction(double x);
}
```

Note : un espace de noms **n'**est **pas** une structure, un type ou un objet quelconque, c'est juste un **nom de regroupement**, une « *étiquette* », un « nom de famille ».

L'objet `compteur` existe (il est déclaré) et sa **portée** s'appelle `outils`, mais `outils` n'est pas un objet manipulable en soi.
C'est **juste un nom**.

Utilisation des objets appartenant à un espace de noms nommé

Pour référencer explicitement un objet *X* d'un espace de noms *nom*, on écrit : *nom*`::`*X*.

Exemple : `++(outils::compteur);`

Si l'on veut utiliser plus librement tous les noms d'un espace de noms :

`using namespace` *nom*`;`

Exemple :
```
using namespace outils;
compteur += 3;
```

On peut aussi expliciter un objet particulier, ce qui évite de spécifier l'espace de noms à chacune de ses utilisations, mais n'autorise pas l'utilisation des autres objets du même espace de noms.

`using` *nom*`::`*X*`;`

Utilisation des objets appartenant à un espace de noms nommé (2)

Attention ! L'utilisation des namespaces **ne change pas** les règles de résolutions de portée...

...en cas d'ambiguité, c'est toujours la variable « *la plus proche* » qui est choisie.

Conseil : Ne **jamais** mettre de
`using namespace` *nom*`;`
dans des fichiers « *headers* » (`.h`) !

Exemple complet

```
#include <iostream>
using namespace std; // utilisation des objets standards (std)

namespace test {
  int i;  // ceci sont des variables utilisables
  int j;  // uniquement dans la portee nommee "test"
}

int i(3);   // ceci est une variable globale

int main() {
  int i(1); // voici une variable locale

  test::i = 5; // utilisation des variables de l'espace
  test::j = 6; // de noms "test"

  cout << i << ' ' << ::i << ' ' << test::i << endl;  // 1 3 5
  // cout << j << endl;   // ERREUR: j undeclared

  using test::j;
  cout << j << endl; // signifie test::j

  // using test::i;   // ERREUR: redefinition of i (i local)

  using namespace test;
  cout << i << endl; // c'est quand même le i local !
  cout << j << endl; // c'est test::j
}
```

Ce qu'il faut retenir du cours d'aujourd'hui

- se remettre très rapidement à la programmation (réviser)
- il va y avoir un projet : bien le gérer (pas de retard ; savoir travailler raisonnablement à son niveau)
- les principes et la mise en œuvre de la compilation séparée :
  - rôle des fichiers `.h`
  - utilisation simple des `Makefile`