UE 3.1 C++ : Outils de développement

Simon Rohou

2025/2026

Supports de cours disponibles sur www.simon-rohou.fr/cours/c++/
Slides inspirées des cours de Damien Rohmer (École Polytechnique Paris) et Thierry Eude (Université Laval)

ENSTA INSTITUT POLYTECHNIQUE DE PARIS

Compilation

CMake

Tester un projet

Documenter du code avec Doxygen

La gestion de versions sous Git

Utiliser un logiciel d'intégration continue (CI)

Section 1

Compilation

Génération d'un exécutable

Contrairement au Python, le C++ est un **langage de programmation compilé**.

La compilation consiste à transformer un **code source** (lisible par un humain) en un **code objet** (interprétable par une machine) afin de créer un **exécutable**.

Pour la compilation, on retient **les phases principales** suivantes :

1. Prétraitement des fichiers sources
2. Compilation séparée (ou effective)
3. Édition de liens (appelée *linking*)

ENSTA INSTITUT POLYTECHNIQUE DE PARIS

1. Prétraitement des sources

Il s'agit d'une opération effectuée par le **préprocesseur**.
Celui-ci parcourt le code, détecte les instructions qui lui sont destinées (préfixées par #) et réalise des substitutions de texte.

Notamment :

- remplacement de macros
- suppression de texte
- inclusion de fichiers...

Exemple :

```
#ifndef __CELL_H__
#define __CELL_H__

// ...

#endif
```

ENSTA INSTITUT POLYTECHNIQUE DE PARIS

2. Compilation séparée (ou effective)

Elle traduit le langage C++ en **langage machine**.

Le résultat de cette compilation est un **fichier objet** portant l'extension `.o`.

– chaque fichier source (`.cpp`) est compilé indépendamment ;

– résultat en objets (`.o`) : traduction du code assembleur en langage machine ;

– les fichiers objets peuvent être regroupés en bibliothèques statiques, afin de rassembler un certain nombre de fonctionnalités qui seront utilisées ultérieurement.

3. Édition de liens (appelée linking)

Phase de **regroupement des fichiers objets** provenant :

- des sources du programme développé (ex : `main.cpp`) ;
- de la bibliothèque standard (ex : STL) ;
- des bibliothèques utilisateurs éventuelles.

On parle d'**édition de liens** :

- regroupement des fichiers objets
- résolution des références inter-fichiers
- puis création d'un fichier image
  (*ex :* exécutable, ou bibliothèque)

ENSTA | INSTITUT POLYTECHNIQUE DE PARIS

3. Édition de liens (appelée linking)

Code :

```
1  #include "vibes.h"
2  using namespace std;
3
4  int main()
5  {
6    vibes::beginDrawing();
7    return EXIT_SUCCESS;
8  }
```

Compilation :

```
g++ main.cpp -o output
```

Erreur à l'édition des liens :

```
/tmp/cc7VAZ4D.o: In function ‘main’:
main.cpp:(.text+0x5): undefined reference to ‘vibes::beginDrawing()’
collect2: error: ld returned 1 exit status
```

3. Édition de liens (appelée linking)

Code :

```
1  #include "vibes.h"
2  using namespace std;
3
4  int main()
5  {
6    vibes::beginDrawing();
7    return EXIT_SUCCESS;
8  }
```

Compilation (commande correcte) :

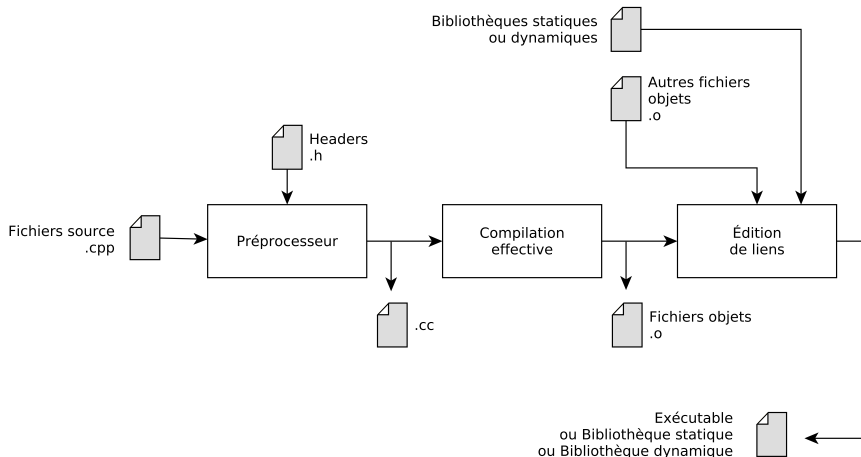
```
g++ main.cpp vibes.cpp -o output
```

Erreur à l'édition des liens :

```
/tmp/cc7VAZ4D.o: In function 'main':
main.cpp:(.text+0x5): undefined reference to 'vibes::beginDrawing()'
collect2: error: ld returned 1 exit status
```

En résumé

Les principales étapes du processus de **compilation** :



Bibliothèques (en anglais : libraries)

Il existe **deux types de bibliothèques** :

- les **bibliothèques statiques**
  - le code utilisé est copié dans le binaire à l'édition des liens ;
  - typiquement de la forme `lib***.a` ;
  - *exemple :* `libm.a` est la bibliothèque mathématique (fonctions `sqrt`, `cos`, etc.).
- les **bibliothèques dynamiques**
  - les dépendances sont résolues lors de l'exécution du programme ;
  - la librairie peut être partagée entre plusieurs exécutables ;
  - typiquement de la forme `lib***.so`.

ENSTA INSTITUT POLYTECHNIQUE DE PARIS

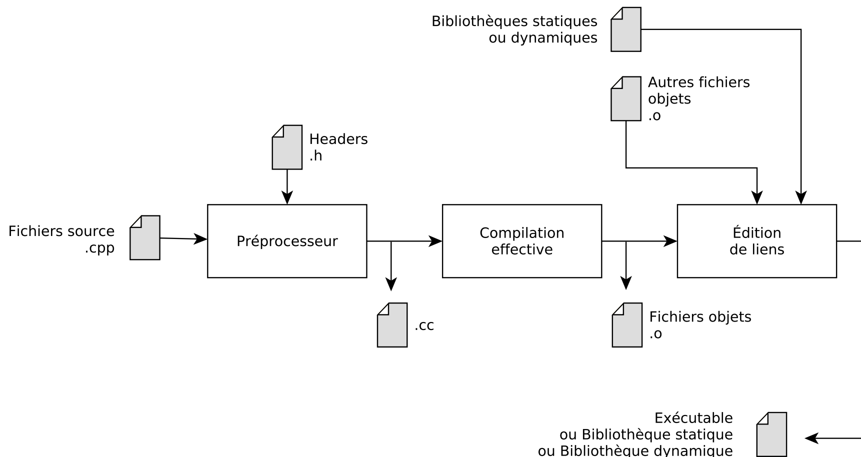
Bibliothèques (en anglais : libraries)

Avantages et inconvénients des deux types :

- les **bibliothèques statiques** :
  - le fichier exécutable qui l'intègre contient, avant l'exécution, tout ce qui lui est nécessaire pour fonctionner ;
  - mais il sera beaucoup plus volumineux que le même programme obtenu par compilation avec une bibliothèque dynamique (car la définition des fonctions se trouve dans le fichier exécutable).
- les **bibliothèques dynamiques** :
  - la mise à jour d'une bibliothèque dynamique ne demande pas de recompilation de l'exécutable ;
  - si elle disparaît ou est considérablement modifiée, les programmes qui en dépendent peuvent devenir inopérants.

La compilation : en résumé

Les principales étapes du processus de **compilation** :



La compilation : en résumé

Toutes ces opérations peuvent être **régroupées en une seule étape** par des outils de compilation.

- les compilateurs appellent généralement le préprocesseur et l'assembleur automatiquement ;
- ils réalisent même souvent l'édition des liens eux-mêmes ;
- il reste généralement possible d'obtenir les fichiers intermédiaires de chaque étape.

Il existe **plusieurs outils pour compiler** un projet. Par exemple :

- en ligne de commande (g++ ...)
- avec `make` et un fichier Makefile
- à plus haut niveau avec CMake

Un exemple de projet simple à compiler

Fichier lib1.h :

```
#ifndef __LIB1_H__
#define __LIB1_H__
  void ma_procedure();
#endif
```

Fichier lib1.cpp :

```
#include "lib1.h"
#include <iostream>

using namespace std;

void ma_procedure()
{
  cout << "Test" << endl;
}
```

Programme principal : fichier main.cpp :

```
#include "lib1.h"

int main()
{
  ma_procedure();
  return 0;
}
```

En utilisant g++, le compilateur de GNU

GCC (GNU Compiler Collection) propose un ensemble de compilateurs libres du projet GNU.

Pour C++, il met à disposition le compilateur g++.

Exemple en ligne de commande (avec plusieurs options) :

```
g++ -c lib1.cpp -g -Wall -Wextra -std=c++20
g++ -c main.cpp -g -Wall -Wextra -std=c++20
g++ lib1.o main.o -o main_exe
```

L'option `-Wall` permet par exemple d'afficher tous les warnings possibles à la compilation.

Exécution :

```
./main_exe
```

ENSTA INSTITUT POLYTECHNIQUE DE PARIS

En utilisant Make

Le logiciel Make :

- **construit automatiquement** des fichiers exécutables ou des bibliothèques ;
- permet de gérer des dépendances lors de la compilation par la **définition de règles**.

Il se base sur un fichier `Makefile` :

- qui détermine les opérations nécessaires pour compiler un programme ;
- les exécute **si nécessaire**
  (ex : recompilation si modification des sources).

Cette solution était auparavant très répandue. Elle tend à disparaître au profit de CMake. Elle est évoquée dans ce cours à titre informatif.

En utilisant Make

Exemple (très simple !) de `Makefile` :

```
CXXFLAGS=-g -Wall -Wextra -std=c++17
CXX=g++

# executable name
project=main_exe

all:${project}

# linking
${project} : main.o lib1.o
   ${CXX}$^ -o${project}

# compile object files
main.o: main.cpp lib1.h
lib1.o: lib1.cpp lib1.h

clean:
   rm -f *~ *.o${project}
```

Et son utilisation :

```
make
./main_exe
```

Note : il est possible de nettoyer le répertoire des fichiers temporaires et de l'exécutable par l'appel à la commande :

```
make clean
```

Section 2

CMake

La gestion du processus de compilation par CMake

CMake : système de **construction logicielle multiplateforme**,

- plus haut niveau que Make ;
- permet de vérifier les prérequis et les dépendances entre les différents composants d'un projet.

La gestion du processus de compilation par CMake

CMake : système de **construction logicielle multiplateforme**,

- plus haut niveau que Make ;
- permet de vérifier les prérequis et les dépendances entre les différents composants d'un projet.

CMake pour **toutes les plate-formes** :

- CMake : abréviation de *cross platform make* ;
- planifie une construction ordonnée et adaptée à la plateforme ;
- particulièrement adapté à la construction des logiciels destinés à fonctionner sous Linux et Windows.

La gestion du processus de compilation par CMake

CMake : système de **construction logicielle multiplateforme**,

- plus haut niveau que Make ;
- permet de vérifier les prérequis et les dépendances entre les différents composants d'un projet.

CMake pour **toutes les plate-formes** :

- CMake : abréviation de *cross platform make* ;
- planifie une construction ordonnée et adaptée à la plateforme ;
- particulièrement adapté à la construction des logiciels destinés à fonctionner sous Linux et Windows.

Autres intérêts :

- outils d'installation (votre programme dans `sudo apt install ...`) ;
- outils de tests et de documentation.

Utilisation de CMake

En reprenant l'exemple précédent (`main.cpp`).
La configuration se fait dans un fichier `CMakeLists.txt` :

```
cmake_minimum_required(VERSION 3.0)
project(mon_executable)

set(CMAKE_BUILD_TYPE debug)
set(CMAKE_CXX_FLAGS "-Wall -Wextra -std=c++20")

file(
  GLOB_RECURSE
  source_files

  src/main.cpp
  src/vibes.cpp
  # etc.
)

add_executable(
  ${PROJECT_NAME}
  ${source_files}
)
```

Utilisation de CMake

En reprenant l'exemple précédent (`main.cpp`).
La configuration se fait dans un fichier `CMakeLists.txt` :

```
cmake_minimum_required(VERSION 3.0) # compatibilités de CMake
project(mon_executable) # nom du binaire : "mon_executable"

set(CMAKE_BUILD_TYPE debug) # compilation en mode debug
set(CMAKE_CXX_FLAGS "-Wall -Wextra -std=c++20") # options

file( # liste des fichiers à compiler
  GLOB_RECURSE # recherche récursive
  source_files # les fichiers suivants seront listés dans la variable source_files
  # Les fichiers .cpp sont listés ci-après :
  src/main.cpp
  src/vibes.cpp
  # etc.
)

add_executable( # création de l'exécutable binaire du projet
  ${PROJECT_NAME} # contient le nom du binaire (i.e. "mon_executable")
  ${source_files} # liste des fichiers compilés pour le linking
)
```

Utilisation de CMake

L'exemple de `CMakeLists.txt` précédent est **très générique** et peux être recopié tel quel pour compiler de nombreux projets.

CMake génère un `Makefile` ainsi que de nombreux fichiers temporaires.
Il convient de laisser cette génération se faire dans un **dossier temporaire**.
Ainsi, pour compiler notre projet, on fera :

```
mkdir build # 'build' contiendra tous les éléments temporaires de la compilation
cd build
cmake .. # on renseigne à CMake l'emplacement relatif du fichier CMakeLists.txt (ici : le
         répertoire parent)
make # après configuration avec cmake, compilation avec make
```

L'exécutable `mon_executable` est généré dans le dossier `build` :

```
./build/mon_executable
```

Note : l'utilisation de la commande `make`, pour lancer la compilation, n'est pas portable et peut ne pas fonctionner sur votre système. L'objectif de CMake étant d'être portable, il convient d'utiliser :

```
# après configuration avec cmake, compilation avec la commande portable :
cmake --build .
```

Exemple de structure d'un projet

Il convient d'organiser/séparer les différents fichiers du projet :

- les sources (`.cpp`, `.h`)
- la documentation (par exemple : `.md`, `.html`)
- les fichiers temporaires de compilation (`Makefile`, `.o`)

Exemple de structure :

```
mon_projet
 |__build/     > contient les fichiers de compilation
 |__src/       > contient les sources C++ (.cpp/.h)
 |__tests/     > contient des tests unitaires
 |__doc/       > contient de la documentation
 |__README.md
 |__CMakeLists.txt
```

Section 3

Tester un projet

ENSTA | INSTITUT POLYTECHNIQUE DE PARIS

Écrire des tests unitaires

Les **tests unitaires** :

- sont des programmes différents du code ;
- peuvent être écrits dans un autre langage ;
- servent à **vérifier** et à **valider** le code.

Également :

- il est préférable de les écrire :
  - avant le développement du code lui-même ;
  - par un autre développeur
    (qui aura un usage différent de vos développements)
- ils peuvent avoir plusieurs objectifs :
  - valider la conformité d'un résultat ;
  - vérifier la vitesse d'exécution d'un programme, etc.

ENSTA | INSTITUT POLYTECHNIQUE DE PARIS

Pourquoi écrire des tests unitaires ?

Imaginez : votre projet fait plusieurs dizaines de milliers de lignes de code. Il est composé de fonctions qui interagissent les unes avec les autres. Enfin, vous l'avez diffusé à une centaine d'utilisateurs qui comptent sur sa fiabilité, ou le déploient sur des robots autonomes.

Vous souhaitez optimiser votre projet en changeant l'implémentation d'une de vos fonctions.

Comment s'assurer que cette nouvelle implémentation est viable ?
→ on ré-exécute les **tests unitaires** de cette fonction.

Comment s'assurer que les fonctions qui en dépendent fonctionnent toujours ? Que le projet ne sera pas en bug dans un cas particulier ?
→ on implémente des **tests d'intégration** pour valider les dépendances entre les fonctions.

ENSTA | INSTITUT POLYTECHNIQUE DE PARIS

Pourquoi écrire des tests unitaires ?

Imaginez : votre projet fait plusieurs dizaines de milliers de lignes de code. Il est composé de fonctions qui interagissent les unes avec les autres. Enfin, vous l'avez diffusé à une centaine d'utilisateurs qui comptent sur sa fiabilité, ou le déploient sur des robots autonomes.

Vous souhaitez optimiser votre projet en changeant l'implémentation d'une de vos fonctions.

Comment s'assurer que cette nouvelle implémentation est viable ?
→ on ré-exécute les **tests unitaires** de cette fonction.

Comment s'assurer que les fonctions qui en dépendent fonctionnent toujours ? Que le projet ne sera pas en bug dans un cas particulier ?
→ on implémente des **tests d'intégration** pour valider les dépendances entre les fonctions.

Tester son code, c'est pénible,
mais c'est éviter de *régresser* dans son développement !

Écrire des tests unitaires

Différents outils déjà existants. Par exemple :

- CTest proposé avec CMake
- cppunit
- Catch2, bibliothèque *header only*

Exemple avec Catch2 :

```
TEST_CASE("Test case 1") {
  SECTION("Test 1") {
      // ... quelques initialisations ici ...
    CHECK(robot.pos_x() == 3.);
    robot.move_x(5.);
    CHECK(robot.pos_x() == 8.);
  }
}
```

Un exemple de tests unitaires sur un projet développé à l'ENSTA :

https://github.com/codac-team/codac/blob/master/tests/core/tests_values.cpp

Section 4

Documenter du code avec Doxygen

ENSTA | INSTITUT POLYTECHNIQUE DE PARIS

Doxygen : générateur de documentation libre

De même que pour les tests, il est nécessaire de documenter son code : expliquer aux utilisateurs et co-développeurs le rôle de chaque fonction, classe, fichier.

Doxygen : générateur de documentation libre

De même que pour les tests, il est nécessaire de documenter son code : expliquer aux utilisateurs et co-développeurs le rôle de chaque fonction, classe, fichier.

Doxygen permet de créer des **documentations techniques**, notamment pour le C++.

La documentation des sources passe par l'écriture de **balises-commentaires** qui définissent des éléments de documentation.

Un programme extérieur vient ensuite analyser les sources commentées et **génère une documentation** sous différents formats : HTML, XML, PDF...

Exemple de code commenté

```
/**
 * \class  Car
 * \brief  A non-autonomous robot car on a circular road.
 * \author Simon Rohou
 * \date   2024
 */
class Car
{
  /**
   * \brief Creates a car
   * \param road a pointer to the Road object
   * \param x    the initial position of the car on the road
   * \param v    the initial speed of the car (0 by default)
   */
  Car(const Road *road, float x, float v = 0.);

  /**
   * \brief Stops the vehicle
   * \note Its speed is set to 0.
   */
  void stop();
```

Génération de la documentation technique

Il est nécessaire d'installer les outils suivants :

- `doxygen`
- `graphviz`
  (pour générer des diagrammes de dépendances et d'héritages)

La documentation technique peut ensuite être **générée** dans un sous-répertoire `doc` de votre projet :

```
cd doc
doxygen Doxyfile
cd ..
```

Le fichier `Doxyfile` permet de **configurer** la génération de la documentation (fichiers à traiter, style de rendu, etc.).

Section 5

La gestion de versions sous Git

ENSTA INSTITUT POLYTECHNIQUE DE PARIS

Qu'est-ce que Git ?

Git est un **système de contrôle de version**. Il permet de :

- mémoriser et retrouver différentes versions d'un projet ;
- faciliter le travail entre différents développeurs.

ENSTA | INSTITUT POLYTECHNIQUE DE PARIS

Qu'est-ce que Git ?

Git est un **système de contrôle de version**. Il permet de :

- mémoriser et retrouver différentes versions d'un projet ;
- faciliter le travail entre différents développeurs.

Git est un système :

- initialement créé par Linus Torvalds pour faciliter le développement de Linux ;
- gratuit et open-source ;
- disponible sur toutes les plate-formes.

ENSTA | INSTITUT POLYTECHNIQUE DE PARIS

Qu'est-ce que Git ?

Git est un **système de contrôle de version**. Il permet de :

- mémoriser et retrouver différentes versions d'un projet ;
- faciliter le travail entre différents développeurs.

Git est un système :

- initialement créé par Linus Torvalds pour faciliter le développement de Linux ;
- gratuit et open-source ;
- disponible sur toutes les plate-formes.

Exemple de documents à versionner sous Git :

- codes C++ ;
- articles ou rapports (écrits par exemple en LaTeX) ;
- tout autre document en format ASCII (on évitera les binaires).

ENSTA INSTITUT POLYTECHNIQUE DE PARIS

La différence entre Git et GitHub

GitHub est un **service web d'hébergement** et de gestion de développement de logiciels versionnés sous Git.

GitHub permet notamment :

- d'héberger les sources d'un projet sur un serveur ;
- de gérer des droits d'accès de développeurs ;
- de lister et suivre des *issues* (bug, suggestions, etc.) ;
- de créer une documentation en ligne ;
- de tester et déployer automatiquement un logiciel,
  → par exemple avec l'outil Travis CI.

Git : principes généraux d'un commit

Git permet donc de gérer **les versions d'un projet**.
Il enregistre un historique de ses évolutions :

- chaque évolution est matérialisée par un `commit`
- les `commit` sont marqués à l'initiative de l'utilisateur
- chaque `commit` est un *snapshot* du projet
  → *i.e.* un instantané du contenu de l'espace de travail, des fichiers
- on pourra par la suite revenir à un `commit` pour remonter le temps
- un `commit` est créé par différentiation avec l'état précédent du projet
  → on n'enregistre que les changements de caractères, ce qui explique pourquoi Git n'est pas adapté pour versionner du binaire
- les `commit` sont enregistrés en local sur notre propre machine
  → les opérations de versionnage sont beaucoup plus rapides que sur un serveur distant

Git : les états des fichiers

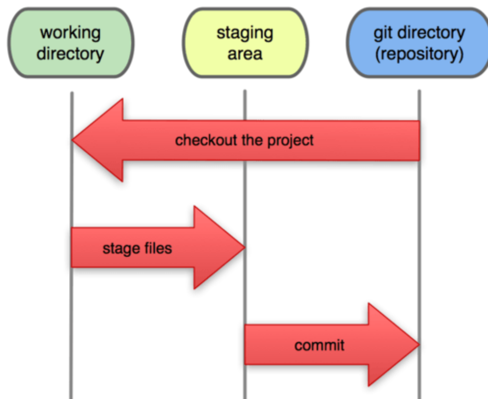
Un fichier peut avoir deux grands états dans Git :

- être sous suivi de version
  → s'il appartient déjà à un `commit` précédent
- être non suivi
  → s'il n'appartient pas au dernier `commit` et n'a pas été **indexé**

Ensuite, chaque **fichier suivi** peut avoir l'un de ces trois états :
Modifié (`modified`) Indexé (`staged`) Validé (`committed`)

1. Un fichier qu'on a modifié depuis le dernier `commit` est considéré comme `modified` par Git tant qu'il n'a pas été indexé.
2. Lorsqu'on signale à Git que le fichier qu'on vient de modifier doit faire partie du prochain `commit`, il devient indexé (`staged`).
3. Enfin, en créant le `commit`, on enregistre une version du projet avec l'ensemble des fichiers indexés et suivis mais non modifiés. Les fichiers du `commit` deviennent "validés" (`committed`) et on peut s'intéresser au `commit` suivant.

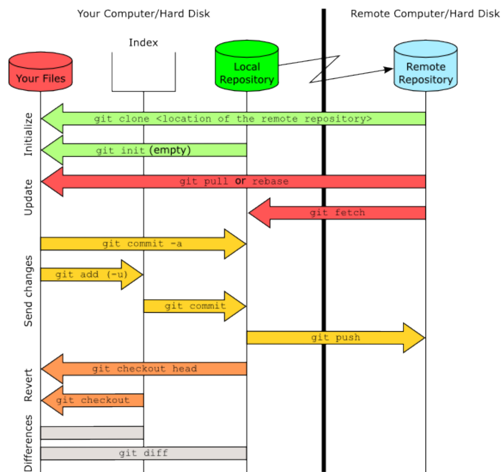
Git : les états des fichiers



Issu de `http://liris.cnrs.fr/~pchampin`

ENSTA INSTITUT POLYTECHNIQUE DE PARIS

Git : principales commandes



Issu de `http://www.it.uc3m.es`

Git : les branches

Il arrive que l'on souhaite dupliquer virtuellement notre projet pour y faire de nouveaux développements (*i.e.* de nouveaux `commit`), sans pour autant impacter le projet de base.

On peut alors créer une nouvelle **branche** (par exemple : `develop`) dans laquelle ces développements se feront, en marge de la branche principale :

Issu de `https://www.atlassian.com`

Lorsque ces développements seront concluants et que le projet sera de nouveau stable, la branche `develop` pourra être fusionnée avec la branche `master`.

Cela facilite le travail collaboratif sur un même projet.

Git en lignes de commande

Démarrer un dépôt depuis un répertoire distant :

La commande `git clone` télécharge un répertoire projet depuis un serveur distant (le *remote*) et le configure comme dépôt Git en local sur votre machine.

```
git clone http://projet.com/repertoire.git mon_projet
cd ./mon_projet
```

Les fichiers sources du projet sont ensuite librement modifiables.

Les `commit` peuvent être retournés au serveur distant avec `git push`.

ENSTA INSTITUT POLYTECHNIQUE DE PARIS

Git : commandes usuelles

- `git log`
  liste les derniers *commit* (révisions) de la branche courante
- `git status`
  état de la branche courante avec liste des fichiers suivis et non suivis
- `git add <file1> <file2> ...`
  ajoute/indexe des fichiers dans la zone de transit (*staging area*)
- `git commit -m "titre de commit"`
  propage la *staging area* dans le dépôt, avec un identifiant et un nom
- `git pull <branch>`
  récupère les dernières modifications sur le dépôt distant
- `git push <branch>`
  publie les nouvelles révisions vers le dépôt distant (*remote*)
- `git merge <branch>`
  fusionne une branche dans la branche courante

ENSTA INSTITUT POLYTECHNIQUE DE PARIS

Git : éviter l'indexation de certains types de fichiers

Il suffit d'ajouter un fichier `.gitignore` à la racine du projet :

```
# Compiled Object files
  *.slo
  *.lo
  *.o
  *.obj

# Compiled libraries
  *.so
  *.dll
  *.a
  *.lib

# Executables
  *.exe
  *.out
  *.app

# hidden sources
  *._cpp
  *._h
  *._py
  *~

# build directory
  make
  build
```

Git : un exemple

Un exemple de projet Git : le dépôt de VIBes.

Par exemple, l'ensemble des `commit` de la branche `master` du projet est visible sur : `https://github.com/ENSTABretagneRobotics/VIBES/commits/master`

En cliquant sur chaque commit, on retrouve chaque modification faite par l'un des développeurs du projet.

Section 6

Utiliser un logiciel d'intégration continue (CI)

Travis CI

Travis CI est un logiciel libre d'intégration continue. Il s'interface facilement avec le service d'hébergement GitHub.

Il permet :

- de compiler automatiquement votre projet sur un serveur distant, dans un environnement vierge qu'on peut configurer
- de le tester sur ce même serveur après compilation
- éventuellement de *déployer* les binaires générés (les rendre disponibles au téléchargement à d'autres utilisateurs)

Différentes configurations d'environnements sont possibles dans Travis CI, ce qui permet de tester votre projet avec différents **langages**, différents **compilateurs**, ou **différents systèmes d'exploitation** (Windows, Linux, Mac OS).

Travis pour tester la compilation d'un projet C++

Travis CI est payant mais propose une solution gratuite à tous les projets open source.
Cette version est disponible sur travis-ci.org.
(vous pouvez vous identifier sur Travis avec votre compte GitHub)

Pour configurer Travis CI sur GitHub, il suffit d'ajouter un fichier `.travis.yml` à la racine du projet.

Les instructions sont détaillées ici :

`https://docs.travis-ci.com/user/languages/cpp/`

Une fois configuré, votre projet sera compilé par Travis à chaque commit sur GitHub !

Travis pour tester la compilation d'un projet C++

Exemple de fichier `.travis.yml` :
(installation d'un g++ et compilation/exécution d'un fichier `main.cpp`)

```
1  language: cpp
2  compiler: gcc
3  dist: trusty
4
5  before_install:
6  - sudo add-apt-repository -y ppa:ubuntu-toolchain-r/test
7  - sudo apt-get update -qq
8
9  install:
10 - sudo apt-get install -qq g++-6
11
12 script:
13 - g++ main.cpp -std=c++17 -o main_exe
14 - ./main_exe
```