Programmation avancée en C++: Gestion de projet, compilation et IDE

Emmanuel Franck

Bureau 225, UFR math info
mail: emmanuel.franck@inria.fr

1. Compilation

Compilation

- **Objectif**: comprendre mieux la compilation et introduction d'outils de compilation et de tests unitaires.
- Suites (ensemble) de compilateurs principales:
  - GCC: gcc/g++/fortran. Libre, très répondu sur Linux.
  - LLVM: clang/clang++/flang. Libre, Plus rapide, moins gourmande que GCC. Fournis avec MAC OS.
  - Intel: icc/icpc/ifort. Propriétaire, Plus performant/plus strict que GCC. Debugging et outils d'analyse de performance.
- Exemple (même que le cours 1):

```
user $ g++ -g first$\_$program.cpp
user $ ./a.out
user $ >>>> Donnez un nombre
user $ 1.5
user $ >>>> Racine carré: 1.22474
```

Compilation: préprocesseur I

- Trois étapes dans la compilation
  - preprocessing
  - compilation
  - édition de lien
- Les **directives de préprcesseur** sont indiquées dans les fichiers sources par ♯.
- La directive la plus classique:

```
1    #include <stdlib.h>
2    #include "coucou.h"
```

- Elle permet d'inclure des fichiers. On utilise "" pour vos fichiers, <> pour les fichiers directement issus de l'installation C++.
- Concrètement, le préprocesseur démarre avant la compilation. Il parcourt vos fichiers. Lorsqu'il rencontre la directive ♯include, il insère littéralement le contenu du fichier indiqué à l'endroit du ♯.

Compilation: préprocesseur II

- Autre directive de préprocesseur:

```
1 #include <stdlib.h>
2 #define XVAR 4
```

- La variable ”XVAR” est une variable globale pour tout le code.

Makefile

```
1 void affichexvar(){
2   cout<< "xvar dans fonction: "<<XVAR<<endl;}
3 int main() {
4   cout<< "xvar dans main: "<<XVAR<<endl;
5   affichexvar();
6   return 0;}
```

```
user $ xvar dans main: 4
user $ xvar dans fonction: 4
```

- Si la variable était déclarée dans la fonction ”main” ou ”affichervar” elle ne serait pas accessible à partir de l’autre fonction.

Compilation: préprocesseur III

- La constante préprocesseur ne prend pas de place en mémoire.
- La directive remplace dans votre code source tous les mots par la valeur correspondante.
- Le ”define” permet de remplacer une expression par un nombre mais par un code entier. On parle de macro.

```
1 #define COUCOU() cout<<"Coucou"<<endl;
2 int main() {
3   COUCOU();
4   return 0;
5 }
```

- On peut faire une macro dépendant d’un paramètre.
- Troisième directive: directive conditionnelle. Exemple:

```
1 #if condition
2     /* Code compilé si la condition est vraie */
3 #elif condition2
4     /* Code compilé si condition2 est vraie */
5 #endif
```

Compilation: préprocesseur IV

- **Utilisation très fréquente**: éviter les inclusions infini des .h/.hpp.
- En général on sépare l'en-tête des fonctions dans les ".h/.hpp" du corps des fonctions.
- Les ".h/.hpp" doivent être inclus dans les fichiers ".c/.cpp" et certains autres fichiers ".h/.hpp".
- **Risque**: On inclut "A.h" dans "B.h" et "B.h" dans "A.h".
- Exemple:

```
1 #ifndef COMPLEXE_HPP // Si la constante n'a pas été définie le fichier
2 // n'a jamais été inclus
3 #define COMPLEXE_HPP // On définit la constante pour que la
4 // prochaine fois le fichier ne soit plus inclus
5
6 #endif
```

- On fait cela dans tout les fichiers .h/.hpp.

Compilation: préprocesseur V

- **Exemple**: Projet polynôme avec les directives préprocesseurs ou sans
- Avec:

```
user $ c++    -c -o main.o main.cpp
user $ g++ -o complexe.o -c complexe.cpp
user $ g++ -o polynome_1.o -c polynome_1.cpp
user $ g++ -o run main.o complexe.o polynome_1.o
```

- Sans:

```
user $ ./complexe.hpp:8:7: error: redefinition of 'complexe'
user $ class complexe {
user $ main.cpp:3:10: note: './complexe.hpp' included multiple times,
      additional
user $ include site here #include "complexe.hpp"
user $ ./polynome_1.hpp:5:10: note: './complexe.hpp' included
      multiple times,
user $ additional include site here #include "complexe.hpp"
...
```

Compilation

- Après l'étape du préprocesseur vient l'étape de compilation.
- Trois sous étapes:
  - **Analyse lexicale et syntaxique**: construction d'un arbre syntaxique décrivant le code. Noeuds internes pour représenter les fonctions et opérateurs. Feuilles (noeuds externes) pour les variables ou constantes.

```
user $ clang -Xclang -ast-dump -fsyntax-only first_program.cpp
```

- **Génération du code et optimisation**: a partir de l'arbre syntaxique, le compilateur génère et optimise un code intermédiaire.
- Le code intermédiaire est dit linéarisé: suite séquentielle d'instructions.
- **Optimisation**: différents niveaux possibles. Plus on optimise, plus la syntaxe du code doit être stricte.

Optimisation I

- Exemple: fonction qui calcul d^n avec n un nombre relatif.
- On calcul ensuite la série $\sum_{i}^{n} \frac{1}{i^2}$

```
1  using namespace std;
2  double powern(double d,unsigned n) {
3    double x =1.0;
4    unsigned j;
5    for(j =1; j <= n; j++){
6      x *= d;
7      return x;}
8  }
9  int main() {
10   double sum =0.0;
11   unsigned i;
12   for(i =1; i <=200000000; i++) {
13     sum += powern (1.0/i,2);
14   }
15   cout<<"sum ="<<sum<<endl;
16   return 0;
17 }
```

Optimisation II

- Différence entre les options d'optimisation:

```
user $ g++ -g -O0 first_program.cpp
user $ time ./a.out
user $ real    0m9.716s
user $ g++ -g -O1 first_program.cpp
user $ time ./a.out
user $ real    0m4.864s
user $ g++ -g -O2 first_program.cpp
user $ time ./a.out
user $ real    0m2.068s
user $ g++ -g -O3 first_program.cpp
user $ time ./a.out
user $ real    0m2.162s
```

- Ici le code est optimisé au maximum des "O2". Si on ne précise pas c'est "O0" qui est utilisé par défaut.
- Gain important entre "O0" et "O2".
- **Dernière sous étape**: l'assemblage qui traduit le code assembleur produit par l'étape précédente en code machine. **Fichiers obtenus**: les ".o"

Edition de liens

- La dernière étape est l'édition de liens. Elle permet de construire un exécutable à partir de l'ensemble des fichiers objets ".o".
- Tâches:
  - Mettre les fichiers objets les uns derrière les autres.
  - Résout les références symboliques entre ces fichiers et avec les bibliothèques externes.
- Deux types de liens:
  - **Statique**: les liens avec les fichiers objets et les bibliothèques statiques sont directement inclus dans l'exécutable
  - **Dynamique**: les liens avec les bibliothèques dynamiques ne sont pas directement inclus dans l'exécutable. Le chemin de ces bibliothèques est fourni par le système d'exploitation.

Options de compilation I

- **Optimisation**: on a vu précédemment qu'on pouvait choisir plusieurs options pour l'optimisation: de ”O0” à ”03”.
- Il existe beaucoup d'options dont certaines très utiles.
- Compilation en deux temps (compilation puis édition de liens):

```
user $ g++ -g -O0 first_program.cpp
user $ time ./a.out
user $ g++ -c first_program.cpp
user $ g++ -o run first_program.o
```

- L'option ”-c” fait la compilation sans édition de liens.
- En faisant juste ”g++” du fichier ”.o” permet de faire l'édition de liens.
- L'option ”-o run” permet de créer un exécutable de nom ”run”.

Options de compilation II

- **Warning**: signalent des cas de programmation pouvant entraîner des comportements inattendus.
- Options: $-w$ (pas de warning), $-W$ (warning supplémentaire), $-Wall$ (tous les warnings).

```
1  double sum;
2  unsigned i;
3  for(i =1; i <=200000000; i++) {
4    sum += powern (1.0/i,2);
5  }
```

- On initialize pas ”sum” à zéro. Possible mais dangereux.

```
user $ g++ -g -W first_program.cpp
user $ g++ -g -Wall first_program.cpp
user $ first_program.cpp:18:5: warning: variable ’sum’ is
     uninitialized when used here [-Wuninitialized]
    sum += powern (1.0/i,2);
first_program.cpp:15:13: note: initialize the variable ’sum’ to
     silence this warning
  double sum;
```

2. Gestion de projet: Cmake et Ctest

Gestion de projet

- **Make**: Pour des projets avec plusieurs fichiers, des bibliothéques etc on utilise make pour gérer la compilation (cours 1).
- Les ”Makefile” sont parfois difficile à écrire.
- **Outil intéressant**: cmake permet de simplifier la compilation des gros projets.
- Comment on organise un projet de taille moyenne:
  - Un dossier **src** avec les sources (.cpp en général).
  - Un dossier **include** avec les en-têtes (.hpp en général).
  - Un dossier **test** avec les tests unitaires (.cpp en général).
  - Un dossier **build** dans le lequel on compile voir execute le code.
- **Exemple**: polynôme complexe.
- Pour utiliser cmake on écrit un ficihier CMakeLists.txt contenant les informations nécessaires à la compilation et permettant de générer un Makefile.

CmakeList I

- On va construire un CMakeList pour le projet: polynôme.

```
# Version minimal de Cmake
cmake_minimum_required(VERSION 3.3)

# Nom du projet
project(first_project)

# On indique où se trouvent les fichiers d'en-tête
include_directories(include)

# On construit la liste des fichiers sources à partir du contenu de src/
file(GLOB_RECURSE SOURCE_FILES src/*.cpp)
```

- On nomme le projet à l'aide de la seconde commande (essentiel pour la suite).
- On inclut le dossier contenant les fichiers d'en-tête puis on construit la liste des fichiers sources stockée dans "SOURCE_FILES".

CmakeList II

- Etape 2: on spécifie les options de compilation.
- En général deux types de compilation:
  - Debug: on ne cherche pas la rapidité mais a voir tous les défauts possibles.
  - Release: pour exécuter un code déjà débuggé. Objectif: rapidité et efficacité.

```
# On fixe un type de compilation par défaut
if(NOT CMAKE_BUILD_TYPE)
  set(CMAKE_BUILD_TYPE Release)
endif()
# On indique les options de compilations
set(CMAKE_CXX_FLAGS "-std=c++11")
# Options de compilation pour le type Release
set(CMAKE_CXX_FLAGS_RELEASE "-O3")
# Options de compilation pour le type Debug
set(CMAKE_CXX_FLAGS_DEBUG "-O0 -g -Wall")
```

- Pour la version **debug**: on n'optimise pas et on ajoute tout les warnings par exemple.
- Si on ne spécifie pas on utilise par défaut la version **release**.

CmakeList III

- Dans le fichier ”Cmakelist” on peut:
  - afficher des messages lors de la compilation,
  - donner des instructions conditionnelles.

```
message(STATUS "Build type: ${CMAKE_BUILD_TYPE}")
message("   Compiler CXX:          ${CMAKE_CXX_COMPILER}")
message("   Compiler CXX flags:         ${CMAKE_CXX_FLAGS}")
if(CMAKE_BUILD_TYPE MATCHES "DEBUG")
message("   Compiler CXX debug flags:   ${CMAKE_CXX_FLAGS_DEBUG}")
endif(CMAKE_BUILD_TYPE MATCHES "DEBUG")
if(CMAKE_BUILD_TYPE MATCHES "RELEASE")
message("   Compiler CXX release flags: ${CMAKE_CXX_FLAGS_RELEASE}")
endif(CMAKE_BUILD_TYPE MATCHES "RELEASE")
```

- On affiche ici: le type de compilation, le compilateur et les options.
- De plus en fonction du type de compilation on affiche les options associées.
- Dans le dossier build:

```
user $ cmake -DCMAKE_BUILD_TYPE=RELEASE ..
user $ Build type: RELEASE
user $   Compiler CXX flags:         -std=c++11
user $   Compiler CXX release flags: -O3
```

CmakeList IV

- Pour finir :
  - On crée une librairie à partir des fichiers sources.
  - On crée un executable à partir du fichier ”main” qui s’appelera ”poly.e”

```
# On crée la bibliothèque libpolynom.a
add_library(first_project ${SOURCE_FILES})

# On génère l'exécutable poly.e à partir du fichier src/main.cpp
add_executable(poly.e src/main.cpp)

# On lie l'exécutable avec la bibliothèque poly.a
target_link_libraries(poly.e first_project)
```

- Une fois cmake initialisé. On peut compiler:

```
user $ cmake -DCMAKE_BUILD_TYPE=RELEASE ..
user $ make
```

```
Scanning dependencies of target first_project
[ 16%] Building CXX object CMakeFiles/first_project.dir/src/complexe.cpp.o
[ 33%] Building CXX object CMakeFiles/first_project.dir/src/main.cpp.o
[ 50%] Building CXX object CMakeFiles/first_project.dir/src/polynome_1.cpp.o
[ 66%] Linking CXX static library libfirst_project.a
```

CmakeList V

```
user $ cmake -DCMAKE_BUILD_TYPE=DEBUG ..
user $ make
```

```
[ 16%] Building CXX object CMakeFiles/first_project.dir/src/complexe.cpp.o
/Users/franck/Desktop/projets_git/Projects/cours_cpp/2020/cours4/cmake/src/complexe.cpp:1
  double res=0;
         ^
1 warning generated.
[ 33%] Building CXX object CMakeFiles/first_project.dir/src/main.cpp.o
[ 50%] Building CXX object CMakeFiles/first_project.dir/src/polynome_1.cpp.o
[ 66%] Linking CXX static library libfirst_project.a
[ 66%] Built target first_project
[ 83%] Building CXX object CMakeFiles/poly.e.dir/src/main.cpp.o
[100%] Linking CXX executable poly.e
[100%] Built target poly.e
irma-dhcp-14:build franck$
```

- On voit qu'en mode debug on a compilé avec l'option d'affichage de tous les warnings.
- Plusieurs remarques:
  - A l'aide de make clean on nettoye les fichiers issus de compilation.
  - A l'aide de make -j X on compile en parallèle sur X processus (utile pour les gros projets).

CmakeList VI

- **Variables CMake**:
  - Comme dit précédemment on note les variables CMAKE_XXX.
  - La commande ”set(..)” permet de donner une valeur aux variables.
  - Au moment de lancer ”cmake” on peut specifier la valeur de la variable:

```
user $ cmake -DNAME_OF_VARIABLE=XXX ..
```

  - C’est donc la commande ”-D” qui permet de spécifier une valeur.
- **Conclusion**: Cmake permet de simplifier la compilation en donnant un outil puissant pour générer un makefile (tache souvent compliquée) et gérer le projet associé.

CTest I

- **Cmake** permet, afin de mieux gérer votre projet, d'automatiser la compilation et l'exécution de tests unitaires.
- Ici: un dossier ”test” et deux fichiers dans ce dossier:
  - test_complexe.cpp (tests unitaires pour la classe complexe),
  - test_polynome.cpp (tests unitaires pour la classe polynôme).

```
# On ajoute les test unitaires
enable_testing()

# On construit la liste des fichiers de test
message(STATUS "Ajout des tests :")
file(GLOB_RECURSE TEST_FILES test/*.cpp)
foreach(TEST_FILE ${TEST_FILES})
  # On récupère le nom sans l'extension
  get_filename_component(TEST_NAME ${TEST_FILE} NAME_WE)
  # On crée l'exécutable test_toto.e à partir du fichier test_toto.cpp
  add_executable(${TEST_NAME}.e ${TEST_FILE})
  target_link_libraries(${TEST_NAME}.e first_project)
  # On ajoute le test "test_toto" qui utilise l'exécutable "test_toto.e"
  add_test(${TEST_NAME} ${TEST_NAME}.e)
  message("     - ${TEST_NAME}")
endforeach()
```

CTest II

- Deux opérations séparées dans les commandes précédentes:
  - On scan les fichiers de ”/test et on crée un exécutable pour chaque test.
  - On crée une liste de test contenant les exécutables de chaque test.
- On peut lancer tout les tests grace à ”ctest”

```
user $ ctest
user $ Test project /Users/....../build
user $ Start 1: test_complexe
user $ 1/2 Test #1: test_complexe ....................   Passed
     0.00 sec
user $ Start 2: test_polynome
user $ 2/2 Test #2: test_polynome ....................   Passed
     0.00 sec
user $ 100 pourcent tests passed, 0 tests failed out of 2
user $ Total Test time (real) =   0.01 sec
```

- La commande ”ctest -V” permet d’afficher les sorties de chaque test.

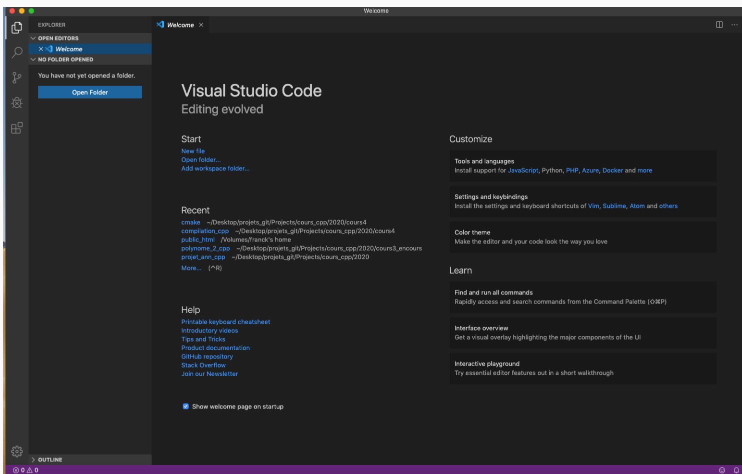
3. Environnement de développement

Environnement de programmation

- Il s'agit d'un ensemble d'outils permettant de faciliter: la compilation, l'écriture, le debugging et le profiling de code.
- **Debugging**: recherche et correction des erreurs (bugs).
- **Profiling**: estimation de temps de calcul/ consommation mémoire de chaque partie du code afin d'optimisation.
- **Exemple d'IDE**: Eclipse, Qtcreator, Visual studio.
- **Visual Studio Code**: Editeur de code multi-plateforme, open source et gratuit (développé par Microsoft) permettant de traiter une dizaine de languages.
- Exemple: C/C++, Java/javascript, Julia, fortran, Python, Visual Basic, HTML.
- **Suite**: gestion du projet redpolynôme.

VSC I

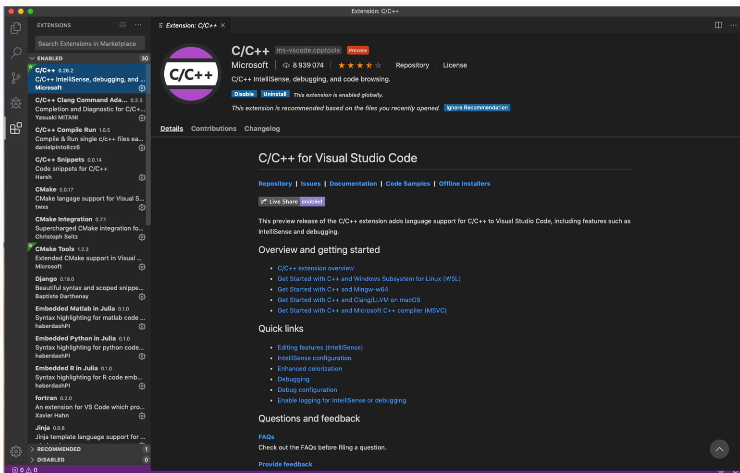
- Page acceuil du logiciel



- **Icones de gauche**: explorer les fichiers, recherche, information sur les dépôts git ouvert,débugger et liste/installation des paquets.

VSC I

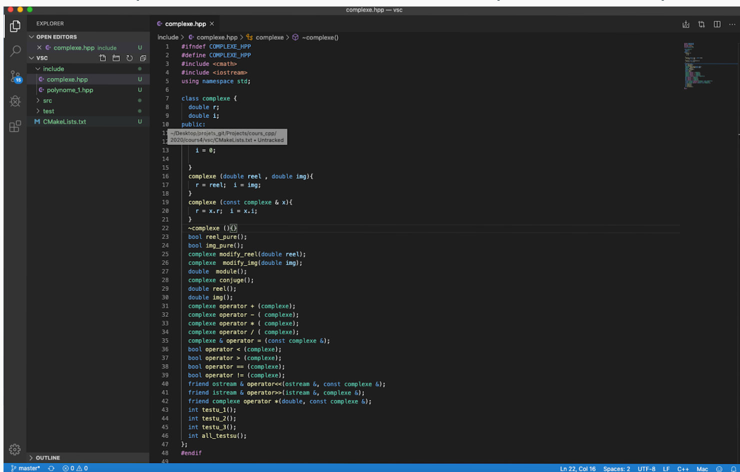
- Paquets:



- Pour utiliser un langage (comme le C ou le Python) ou un outil (comme cmake) il faut installer le paquet associé.

VSC II

- On ouvre le projet ”Polynome” avec VSC.
- Comme beaucoup d’éditeurs VSC colore le code pour le rendre plus lisible.



VSC III

- Plusieurs possibilités offertes par VSC:
  - Il scan le code et détecte un certain nombre de bugs possibles. Exemples: type manquant ou non existant, ";" ou accolade de fin manquante etc.
  - On peut accéder à la déclaration, définition des fonctions en un simple clic.

VSC III

- Plusieurs possibilités offertes par VSC:
  - Il scan le code et détecte un certain nombre de bugs possibles. Exemples: type manquant ou non existant, ";" ou accolade de fin manquante etc.
  - On peut aussi trouver toutes les références à une fonction ou un constructeur.

polynome_1.cpp — vsc

complexe.hpp | polynome_1.cpp | polynome_1.hpp

src > polynome_1.cpp > polynome::derivative()

```
150 polynome polynome::conjuge(){
151   polynome res(deg);
152   for(int i=0;i<deg+1;i++){
153     res.coefs[i]=coefs[i].conjuge();
154   }
155   return res;
156 }
157
158 polynome polynome::derivative(){
```

polynome_1.cpp ~/Desktop/projets_git/Projects/cours_cpp/2020/cours4/vsc/src - References (6)

```
151   polynome res(deg);
152   for(int i=0;i<deg+1;i++){
153     res.coefs[i]=coefs[i].conjuge();
154   }
155   return res;
156 }
157
158 polynome polynome::derivative(){
159   polynome res(deg-1,a,b);
160   for(int i =1;i<deg+1;i++){
161     res.coefs[i-1]= i*coefs[i];
162   }
163   return res;
164 };
165
```

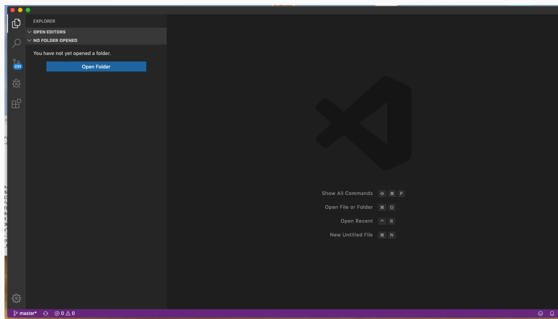
- polynome_1.hpp include 1
- main.cpp src 1
- polynome_1.cpp src 4
  - polynome::derivative(){
  - this->derivative();
  - this->derivative();
  - this->derivative();

```
159   polynome res(deg-1,a,b);
160   for(int i =1;i<deg+1;i++){
```

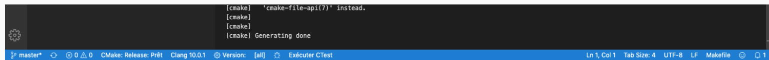
PROBLEMS 1 | OUTPUT | DEBUG CONSOLE | TERMINAL

VSC aide I

- L'interface de VSC permet de simplifier la compilation du programme:
- 1) On ouvre le projet:



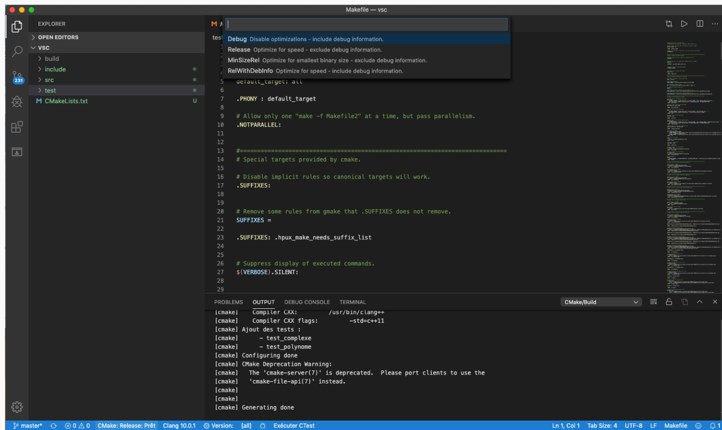
- La barre en bas est actuellement vide.
- Après ouverture du projet:



- **Remarque**: VSC analyse le code, détecte le language et la présence de cmake, propose un compilateur (Clang).

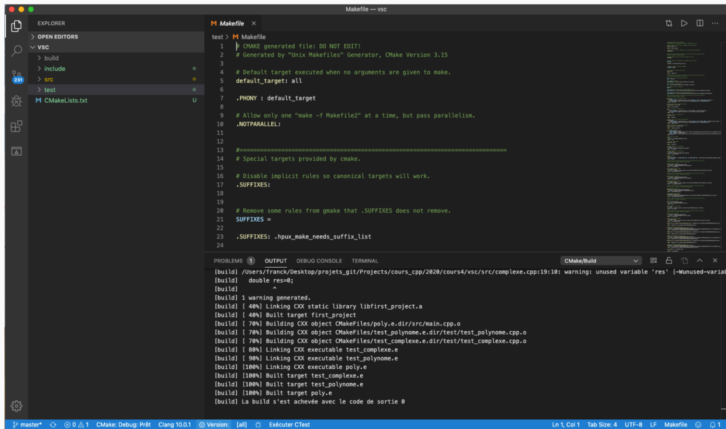
VSC aide II

- VSC génère aussi un Makelife en utilisant le générateur Ninja à la place du générateur Make.
- Par défaut, VSC fait la construction dans le répertoire ”build”.
- 2) On clic sur Cmake pour choisir la version: debug ou release.



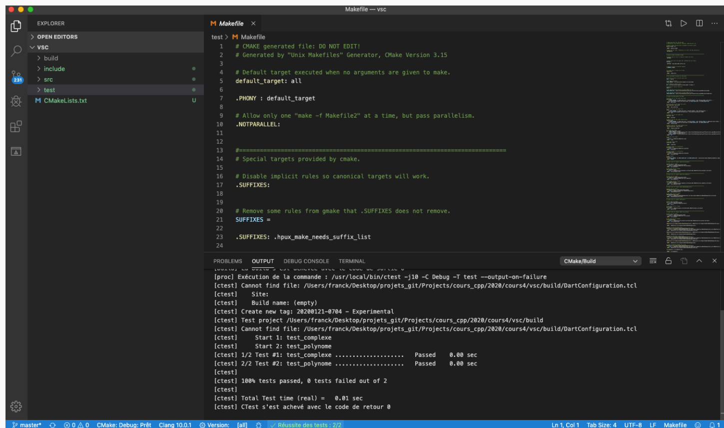
VSC aide II

- VSC génère aussi un Makelife en utilisant le générateur Ninja à la place du générateur Make.
- Par défaut, VSC fait la construction dans le répertoire ”build”.
- En cliquant sur version on lance une compilation:



VSC aide II

- VSC génère aussi un Makelife en utilisant le générateur Ninja à la place du générateur Make.
- Par défaut, VSC fait la construction dans le répertoire ”build”.
- En cliquant sur Execution Ctest on lance les tests:



VSC aide III

- VSC permet aussi de compiler et faire les tests à partir d’un terminal.

- Dans ce cas, comme dans l’autre, lorsqu’il y a une erreur de compilation, VSC fournit un lien vers la ligne correspondante.

Conclusion

Conclusion

- Les gestionnaires de projets comme Cmake, Ctest permettent de simplifier la compilation, vérification de projets lourds (pas forcément évident).
- Les IDE comme VSC permettent de simplifier l'écriture, de debugging et l'utilisation de programmes C++ ou d'autres langages.
- Lien: https://code.visualstudio.com/

Gestionnaire de version I

- **Outil important** pour travailler à plusieurs: Gestionnaire de version
- Exemple: Git, GitLab etc
- **Principe**: On stocke un projet en ligne accessible à plusieurs. A chaque modification: vous l'envoyez vers le serveur, le gestionnaire de version compare à la version anterieur. Si y a des ”conflits” vous devez les gérer. Sinon la nouvelle version remplace l'ancienne.
- **Important**: toutes les versions sont conservées et potentiellement accessible.
- **Utilisation**: il faut installer ”GIT” https://git-scm.com/downloads et un dépôt git: https://github.com/ (par exemple).

Gestionnaire de version I

- **Outil important** pour travailler à plusieurs: Gestionnaire de version
- Exemple: Git, GitLab etc
- **Principe**: On stocke un projet en ligne accessible à plusieurs. A chaque modification: vous l'envoyez vers le serveur, le gestionnaire de version compare à la version anterieur. Si y a des ”conflits” vous devez les gérer. Sinon la nouvelle version remplace l'ancienne.
- **Important**: toutes les versions sont conservées et potentiellement accessible.
- **Utilisation**: il faut installer ”GIT” https://git-scm.com/downloads et un dépôt git: https://github.com/ (par exemple).