SÉANCE 1

Mise à niveau "pro" & Outillage qualité

C Avancé - Master 2

📝 Changelog — V0.0.2

- 01/02/2026 18:03 — Ajout de corrections et clarifications : procédures d'installation (gdb, valgrind, cppcheck), exemples de debug, petites améliorations de mise en page.
- Mise à jour de la version du deck et synchronisation avec les autres slides mis à jour.

📋 Objectifs de la séance

- Renforcer les bases "professionnelles"
  - Build multi-configuration
  - Arborescence projet
  - Tests unitaires & coverage
- Prendre en main les outils de qualité
  - Sanitizers (ASan, UBSan)
  - Valgrind
  - Profiling de base
- **Asseoir une culture d'ingénierie qualité dès le départ**

🎯 Compétences visées

✅ Maîtriser CMake moderne (approche par targets)
✅ Structurer un projet C professionnel
✅ Écrire et exécuter des tests unitaires
✅ Mesurer la couverture de code
✅ Détecter les bugs mémoire et comportements indéfinis
✅ Utiliser Git avec hooks pour automatiser la qualité

📂 Arborescence projet professionnelle

```
mon-projet/
├── CMakeLists.txt
├── README.md
├── .gitignore
├── .git/
│   └── hooks/
├── src/
│   ├── main.c
│   └── module.c
├── include/
│   └── module.h
├── tests/
│   ├── test_module.c
│   └── CMakeLists.txt
└── build/
    ├── debug/
    └── release/
```

🔨 CMake moderne : approche par targets

Ancienne méthode (à éviter)

```
include_directories(${PROJECT_SOURCE_DIR}/include)
add_executable(mon_app main.c module.c)
```

Méthode moderne ✅

```
add_library(mon_module STATIC src/module.c)
target_include_directories(mon_module PUBLIC include)

add_executable(mon_app src/main.c)
target_link_libraries(mon_app PRIVATE mon_module)
```

Avantage : propagation automatique des dépendances

📦 CMakeLists.txt - Structure minimale

```
cmake_minimum_required(VERSION 3.20)
project(MonProjet C)

set(CMAKE_C_STANDARD 11)
set(CMAKE_C_STANDARD_REQUIRED ON)

# Options de compilation
set(CMAKE_C_FLAGS_DEBUG "-g -O0 -Wall -Wextra")
set(CMAKE_C_FLAGS_RELEASE "-O3 -DNDEBUG")

# Bibliothèque
add_library(mon_module STATIC src/module.c)
target_include_directories(mon_module PUBLIC include)

# Exécutable
add_executable(mon_app src/main.c)
target_link_libraries(mon_app PRIVATE mon_module)

# Tests (si activés)
if(BUILD_TESTING)
    enable_testing()
    add_subdirectory(tests)
endif()
```

🏗️ Build multi-configuration

```
# Configuration Debug
cmake -B build/debug -DCMAKE_BUILD_TYPE=Debug
cmake --build build/debug

# Configuration Release
cmake -B build/release -DCMAKE_BUILD_TYPE=Release
cmake --build build/release

# Avec Sanitizers
cmake -B build/asan -DCMAKE_BUILD_TYPE=Debug \
  -DCMAKE_C_FLAGS="-fsanitize=address,undefined"
cmake --build build/asan
```

🧪 Tests unitaires - Framework Unity

Installation Unity (exemple)

```
git clone https://github.com/ThrowTheSwitch/Unity.git extern/Unity
```

Test simple

```c
#include "unity.h"
#include "module.h"

void setUp(void) { /* Init avant chaque test */ }
void tearDown(void) { /* Nettoyage après chaque test */ }

void test_addition(void) {
    TEST_ASSERT_EQUAL_INT(4, add(2, 2));
}

int main(void) {
    UNITY_BEGIN();
    RUN_TEST(test_addition);
    return UNITY_END();
}
```

🧪 Tests unitaires - Framework CUnit

Structure d'un test CUnit

```c
#include <CUnit/CUnit.h>
#include <CUnit/Basic.h>
#include "module.h"

void test_multiplication(void) {
    CU_ASSERT_EQUAL(multiply(3, 4), 12);
    CU_ASSERT_NOT_EQUAL(multiply(2, 2), 5);
}

int main() {
    CU_initialize_registry();

    CU_pSuite suite = CU_add_suite("Suite_Module", NULL, NULL);
    CU_add_test(suite, "test_multiplication", test_multiplication);

    CU_basic_run_tests();
    CU_cleanup_registry();
    return 0;
}
```

📊 Coverage avec gcov/lcov

CMakeLists.txt pour coverage

```
option(ENABLE_COVERAGE "Enable coverage reporting" OFF)

if(ENABLE_COVERAGE)
    set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} --coverage")
    set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} --coverage")
endif()
```

Génération du rapport

```
# Build avec coverage
cmake -B build/coverage -DENABLE_COVERAGE=ON
cmake --build build/coverage

# Exécuter les tests
./build/coverage/tests/test_module

# Générer rapport HTML
lcov --capture --directory build/coverage --output-file coverage.info
genhtml coverage.info --output-directory coverage_html
```

🔍 AddressSanitizer (ASan)

Détecte :

- Débordements de buffer (heap, stack, global)
- Use-after-free
- Use-after-return
- Fuites mémoire (avec `ASAN_OPTIONS=detect_leaks=1`)

Activation

```
cmake -B build/asan -DCMAKE_BUILD_TYPE=Debug \
  -DCMAKE_C_FLAGS="-fsanitize=address -fno-omit-frame-pointer -g"
cmake --build build/asan
```

Exemple de détection

```
int* ptr = malloc(sizeof(int) * 10);
free(ptr);
ptr[0] = 42;  // ❌ ASan détecte use-after-free
```

🔍 UndefinedBehaviorSanitizer (UBSan)

Détecte :

- Débordements d'entiers signés
- Division par zéro
- Décalages invalides
- Conversions invalides
- Déréférencement de pointeur null

Activation

```
cmake -B build/ubsan -DCMAKE_BUILD_TYPE=Debug \
  -DCMAKE_C_FLAGS="-fsanitize=undefined -fno-omit-frame-pointer -g"
cmake --build build/ubsan
```

Exemple

```
int x = INT_MAX;
x = x + 1;  // ❌ UBSan détecte le débordement
```

🔍 Combinaison de Sanitizers

```
# ASan + UBSan ensemble
cmake -B build/sanitizers -DCMAKE_BUILD_TYPE=Debug \
  -DCMAKE_C_FLAGS="-fsanitize=address,undefined -fno-omit-frame-pointer -g"
cmake --build build/sanitizers

# Exécution avec options
ASAN_OPTIONS=detect_leaks=1:halt_on_error=0 \
UBSAN_OPTIONS=print_stacktrace=1 \
  ./build/sanitizers/mon_app
```

⚠️ **Note** : ThreadSanitizer (TSan) incompatible avec ASan/MSan

🔧 Valgrind - Memcheck

Détecte :

- Fuites mémoire
- Accès à mémoire non initialisée
- Débordements de buffer
- Double free

Utilisation basique

```
valgrind --leak-check=full \
         --show-leak-kinds=all \
         --track-origins=yes \
         --verbose \
         ./mon_app
```

Avantages vs ASan

- Pas de recompilation nécessaire
- Plus lent mais plus exhaustif
- Meilleure détection de mémoire non initialisée

📊 Profiling de base - gprof

Compilation avec profiling

```
gcc -pg -O2 src/main.c src/module.c -o mon_app
```

Exécution et analyse

```
# Exécuter le programme (génère gmon.out)
./mon_app

# Analyser les résultats
gprof mon_app gmon.out > analysis.txt

# Vue condensée
gprof -b mon_app gmon.out
```

Alternative moderne : `perf` (Linux), Instruments (macOS)

📊 Profiling avec perf (Linux)

```
# Enregistrer l'exécution
perf record -g ./mon_app

# Rapport interactif
perf report

# Rapport simple
perf report --stdio

# Statistiques
perf stat ./mon_app
```

Métriques : cycles CPU, cache misses, branch mispredictions, etc.

🔄 Git - Hooks pour automatisation

pre-commit hook

```
#!/bin/bash
# .git/hooks/pre-commit

# Vérifier le formatage
clang-format --dry-run --Werror src/*.c include/*.h || exit 1

# Lancer les tests
cmake --build build/debug
./build/debug/tests/test_module || exit 1

echo "✅ Pre-commit checks passed"
```

Installation

```
chmod +x .git/hooks/pre-commit
```

🔄 Git - Hook pre-push

```
#!/bin/bash
# .git/hooks/pre-push

# Build release
cmake -B build/release -DCMAKE_BUILD_TYPE=Release
cmake --build build/release || exit 1

# Tests avec sanitizers
cmake -B build/asan -DCMAKE_BUILD_TYPE=Debug \
  -DCMAKE_C_FLAGS="-fsanitize=address,undefined"
cmake --build build/asan || exit 1
./build/asan/tests/test_module || exit 1

# Vérifier coverage minimale
./scripts/check_coverage.sh 80 || exit 1

echo "✅ Pre-push checks passed"
```

🛠️ TP - Partie 1 : Initialisation

Tâches :

1. Créer l'arborescence projet (src, include, tests)
2. Initialiser un dépôt Git
3. Créer `.gitignore` approprié
4. Écrire un `CMakeLists.txt` minimal avec targets
5. Implémenter une fonction simple dans `module.c`
6. Créer un `main.c` qui utilise cette fonction

Livrable : Projet qui compile en Debug et Release

🛠️ TP - Partie 2 : Tests & Coverage

Tâches :

1. Ajouter Unity ou CUnit au projet
2. Créer `tests/CMakeLists.txt`
3. Écrire 3 tests unitaires minimum
4. Configurer la coverage (gcov/lcov)
5. Générer un rapport HTML de coverage
6. Atteindre au moins 80% de couverture

Livrable : Rapport coverage_html avec ≥80%

🛠️ TP - Partie 3 : Sanitizers

Tâches :

1. Créer un programme avec bugs volontaires :
    - Buffer overflow (stack)
    - Use-after-free
    - Integer overflow signé
    - Division par zéro
2. Compiler avec ASan et UBSan
3. Détecter et corriger chaque bug
4. Documenter les corrections

Livrable : Code bugué + version corrigée + rapport

🛠️ TP - Partie 4 : Valgrind

Tâches :

1. Créer un programme avec fuites mémoire variées
2. Analyser avec Valgrind
3. Corriger toutes les fuites
4. Vérifier avec `--leak-check=full`

Bonus : Comparer les résultats ASan vs Valgrind

Livrable : Sortie Valgrind sans erreur

🛠️ TP - Partie 5 : Git Hooks

Tâches :

1. Créer un hook `pre-commit` qui :
    - Vérifie le formatage (optionnel : clang-format)
    - Lance les tests unitaires
    - Bloque le commit si échec
2. Créer un hook `pre-push` qui :
    - Build en release
    - Lance tests avec ASan/UBSan
    - Vérifie la coverage minimale

Livrable : Hooks fonctionnels et testés

📝 Évaluation

Critères

✅ **Repository Git** (20%)

- Structure propre, .gitignore, commits cohérents

✅ **Build & Configuration** (20%)

- CMake targets, multi-config (Debug/Release)

✅ **Tests & Coverage** (25%)

- Tests unitaires fonctionnels, coverage ≥80%

✅ **Sanitizers & Valgrind** (25%)

- Détection et correction des bugs

✅ **QCM + Revue de code** (10%)

- Connaissances théoriques, pair review

📝 QCM - Exemples de questions

1. Quelle option CMake active les symboles de debug ?
   - A) `-DCMAKE_DEBUG=ON`
   - B) `-DCMAKE_BUILD_TYPE=Debug` ✅
   - C) `-DDEBUG_MODE=1`
2. Quel sanitizer détecte les débordements d'entiers signés ?
   - A) AddressSanitizer
   - B) UndefinedBehaviorSanitizer ✅
   - C) ThreadSanitizer
3. Quelle commande génère un rapport de coverage HTML ?
   - A) `gcov --html`
   - B) `genhtml coverage.info` ✅
   - C) `lcov --output-html`

📝 Revue de code par les pairs

Process

1. **Pull Request interne** : chaque binôme crée une PR
2. **Review** : un autre binôme fait la revue
3. **Critères de review** :
    - Lisibilité du code
    - Tests pertinents
    - Gestion mémoire correcte
    - Structure CMake propre
4. **Approbation** : corrections puis merge

Objectif : Simuler un workflow professionnel

🛠️ Matériel nécessaire

Logiciels requis

- **Compilateur** : GCC ≥9 ou Clang ≥10
- **Build system** : CMake ≥3.20
- **Contrôle version** : Git
- **Coverage** : gcov, lcov
- **Analyse mémoire** : Valgrind
- **IDE** : VS Code (recommandé)

Extensions VS Code utiles

- C/C++ (Microsoft)
- CMake Tools
- GitLens
- Test Explorer

💡 Bonnes pratiques à retenir

1. **Séparez les configurations** : Debug, Release, ASan, Coverage
2. **Testez tôt, testez souvent** : TDD ou au moins tests réguliers
3. **Automatisez** : Hooks Git, CI/CD
4. **Mesurez** : Coverage, profiling, Valgrind
5. **Documentez** : README, commentaires, rapports
6. **Revoyez** : Code review systématique

Culture qualité = Gain de temps à long terme

📚 Ressources complémentaires

Documentation officielle

- CMake : https://cmake.org/cmake/help/latest/
- Unity : https://github.com/ThrowTheSwitch/Unity
- Sanitizers : https://github.com/google/sanitizers
- Valgrind : https://valgrind.org/docs/manual/

Tutoriels

- Modern CMake : https://cliutils.gitlab.io/modern-cmake/
- GDB/Valgrind : https://sourceware.org/gdb/documentation/

Livres

- "Test Driven Development for Embedded C" - James Grenning

🎯 Objectifs pour la prochaine séance

- Maîtrise complète de la chaîne qualité
- Tests automatisés fonctionnels
- Détection proactive des bugs
- Workflow Git professionnel

Préparez vos questions !

❓ Questions ?

N'hésitez pas à demander des clarifications

📧 Contact enseignant
🔗 Dépôt du cours : [à compléter]
💬 Forum / Discord : [à compléter]

Merci !

Bon courage pour le TP ! 🚀

Rappel : La qualité du code se construit dès le premier commit.