Jenga2 – Unitest

Unitest est le framework de tests unitaires **intégré** à Jenga2.
Écrit en C++ moderne, il fournit un ensemble complet de macros d'assertion, de gestion de cas de test, de benchmarking et de profilage.
Il est conçu pour être **léger**, **rapide** et **facilement intégrable** dans tout projet C++ géré par Jenga2.

Sommaire

Philosophie

Unitest est né du besoin d'un framework de test **interne** à Jenga2, utilisable à la fois par les développeurs de Jenga et par les projets clients.
Ses principes directeurs :

- **Sans dépendance externe** (hormis la bibliothèque standard C++).
- **Macros simples et expressives** pour réduire la verbosité.
- **Intégration transparente** avec le DSL Jenga2 via le contexte `unitest`.
- **Support du benchmarking et du profilage** pour les tests de performance.

Architecture du framework

© 2025 Rihen — Tous droits réservés

rihen.universe@gmail.com • teuguiasederis@gmail.com
(+237) 693-761-773
CEO – RIHEN

Le code source d'Unitest se trouve dans `src/Unitest/`. Il est organisé comme suit :

```
src/Unitest/
├── AutoMain.h              # Générateur de main() automatique
├── Benchmark.cpp/h         # Système de benchmarking
├── ConsoleReport.cpp/h     # Rapport console
├── IPerformanceReporter.h  # Interface pour rapports de perf
├── PerformanceReporter.cpp/h # Implémentation du reporting
├── Profiler.cpp/h          # Profilage (flamegraph, statistiques)
├── TestAggregator.cpp/h    # Agrégateur de résultats
├── TestAssert.cpp/h        # Assertions centralisées
├── TestCase.cpp/h          # Classe de base d'un cas de test
├── TestCaseRegistrar_impl.h # Enregistrement automatique des tests
├── TestConfig.h            # Configuration globale
├── TestConfiguration.h     # Paramètres d'exécution
├── TestLauncher.cpp/h      # Lanceur de tests
├── TestMacro.h             # Macros publiques (inclus par l'utilisateur)
├── TestReporter.cpp/h      # Reporters de résultats
├── TestRunner.cpp/h        # Exécuteur de tests
├── Unitest.cpp/h           # Point d'entrée principal
└── UnitTestData.h          # Structures de données partagées
```

L'entrée utilisateur se fait via le fichier `TestMacro.h` qui définit toutes les macros publiques (`TEST_CASE`, `ASSERT_TRUE`, `BENCHMARK`, ...).
Le framework utilise l'enregistrement statique : les cas de test sont automatiquement enregistrés via des variables globales avant `main()`, puis exécutés par `TestRunner`.

🔧 Utilisation dans un projet Jenga

Pour utiliser Unitest dans un projet Jenga2, deux modes sont disponibles.

Mode précompilé (par défaut)

```
with workspace("MonProjet"):
    with unitest() as u:
        u.Precompiled()   # Utilise la version précompilée d'Unitest

    with project("MaBibliotheque"):
        staticlib()
        files(["src/**.cpp"])
        includedirs(["include"])

        with test():
            testfiles(["tests/**.cpp"])
```

- Le projet `__Unitest__` est créé automatiquement.

- Les chemins d'include et de lib sont résolus via les variables `%{Unitest.include}` et `%{Unitest.libdir}`.
- **Aucune compilation d'Unitest** n'est effectuée.

Mode compilation (sources personnalisées)

```
with workspace("MonProjet"):
    with unitest() as u:
        u.Compile(
            kind="STATIC_LIB",
            objDir="Build/Obj/Unitest",
            targetDir="Libs",
            targetName="Unitest",
            cxxflags=["-O2", "-DNDEBUG"]
        )
```

- Unitest est **compilé** à partir de ses sources (situées dans `%{Jenga.Unitest.Source}`).
- Vous pouvez personnaliser les flags, le répertoire de sortie, etc.
- Le projet `__Unitest__` est créé et sera lié à vos tests.

✍️ Écrire des tests

`TEST_CASE` et `TEST`

```
#include <Unitest/TestMacro.h>

TEST_CASE(MonGroupe, MonTest) {
    int a = 1, b = 2;
    ASSERT_EQUAL(3, a + b);
}

// Raccourci sans groupe
TEST(TestSimple) {
    ASSERT_TRUE(true);
}
```

- `TEST_CASE(ClassName, TestName)` → crée une classe nommée `ClassName##TestName##TestCase` et enregistre le test.
- `TEST(TestName)` → équivalent à `TEST_CASE(Default, TestName)`.

Test Fixtures

```
class MonFixture : public nkentseu::test::TestCase {
public:
    MonFixture(const std::string& name) : TestCase(name) {}
```

```
    void SetUp() override { /* initialisation */ }
    void TearDown() override { /* nettoyage */ }
};

TEST_FIXTURE(MonFixture, MonTestAvecFixture) {
    // utilise les membres du fixture
    ASSERT_TRUE(/* ... */);
}
```

Assertions

Macro	Description
`ASSERT_EQUAL(expected, actual)`	Égalité entre deux valeurs
`ASSERT_NOT_EQUAL`	Inégalité
`ASSERT_TRUE(cond)`	Condition vraie
`ASSERT_FALSE(cond)`	Condition fausse
`ASSERT_NULL(ptr)`	Pointeur nul
`ASSERT_NOT_NULL(ptr)`	Pointeur non nul
`ASSERT_LESS(left, right)`	left < right
`ASSERT_LESS_EQUAL`	left ≤ right
`ASSERT_GREATER`	left > right
`ASSERT_GREATER_EQUAL`	left ≥ right
`ASSERT_NEAR(val, ref, eps)`	égalité à epsilon près
`ASSERT_THROWS(exc, expr)`	expression lance une exception donnée
`ASSERT_NO_THROW(expr)`	expression ne lance pas
`ASSERT_CONTAINS(cont, val)`	conteneur contient la valeur
`ASSERT_NOT_CONTAINS`	conteneur ne contient pas la valeur

Toutes ces macros existent également avec le suffixe `_MSG` pour ajouter un message personnalisé :

```
ASSERT_EQUAL_MSG(42, answer, "La réponse à la vie n'est pas bonne !");
```

⏱ Benchmarks

Unitest intègre un système de benchmarking simple mais puissant.

`TEST_BENCHMARK_SIMPLE`

```
TEST_BENCHMARK_SIMPLE(MonBench, "BenchmarkMonAlgo", [](){
    // code à mesurer
}, 1000);
```

- Exécute le code `1000` fois et enregistre le temps moyen.
- Les résultats sont envoyés au `PerformanceReporter` (affichage console, export JSON possible).

Comparaison de benchmarks

```
COMPARE_BENCHMARKS(Comparaison, "AlgoA", algoA, "AlgoB", algoB, 1000, 1.2);
```

- Vérifie que `AlgoA` n'est pas plus de 1.2× plus lent que `AlgoB`.
- Échoue le test si la régression est trop importante.

📊 Profilage

Unitest peut capturer des traces d'exécution et générer des **flamegraphs**.

```
PROFILE_TEST_SCOPE(MonProfilage, {
    // code à profiler
    for (int i = 0; i < 1000000; ++i)
        computation();
});
```

- Le profileur enregistre les temps d'entrée/sortie des fonctions marquées avec `PROFILE_SCOPE`.
- À la fin du test, un fichier JSON contenant les données de flamegraph est généré (ex: `MonProfilage_flamegraph.json`).

⚙️ Configuration avancée

- `TestConfig.h` : permet de définir des macros globales (ex: `UNITEST_MAX_ASSERTIONS`).
- `TestConfiguration` : paramètres passés au lanceur (filtrage de tests, répétitions, ...).

Exemple d'exécution avec filtrage :

```
auto& runner = nkentseu::test::TestRunner::GetInstance();
runner.SetFilter("GroupeA*");
runner.Run();
```

💡 Intégration avec Jenga2

Le contexte `test` dans le DSL Jenga2 automatise la création du projet de test et le lien avec Unitest.

```
with project("Moteur"):
    staticlib()
    files(["src/**.cpp"])

    with test():
        testfiles(["tests/**.cpp"])
        testmainfile("src/main.cpp")   # exclut le main parent
```

- Le projet de test dépend automatiquement de `__Unitest__` et du projet parent.
- En mode précompilé, les chemins `%{Unitest.include}` et `%{Unitest.libdir}` sont résolus.
- L'exécutable de test est placé dans `%{wks.location}/Build/Tests/%{cfg.buildcfg}`.

🗀 Structure du dossier

```
Unitest/
├── __init__.py                 # Marqueur de package (Python)
├── bin/                        # (optionnel) binaires précompilés
├── libs/                       # (optionnel) bibliothèques précompilées
├── Entry/                      # Exemple de fichier main.cpp
│   └── Entry.cpp
└── src/Unitest/               # **Sources C++ du framework**
        ├── AutoMain.h
        ├── Benchmark.cpp/h
        ├── ConsoleReport.cpp/h
        ├── IPerformanceReporter.h
        ├── PerformanceReporter.cpp/h
        ├── Profiler.cpp/h
        ├── TestAggregator.cpp/h
        ├── TestAssert.cpp/h
        ├── TestCase.cpp/h
        ├── TestCaseRegistrar_impl.h
        ├── TestConfig.h
        ├── TestConfiguration.h
        ├── TestLauncher.cpp/h
        ├── TestMacro.h
        ├── TestReporter.cpp/h
        ├── TestRunner.cpp/h
        ├── Unitest.cpp/h
        └── UnitTestData.h
```

🔨 Compiler Unitest soi-même

Si vous préférez compiler Unitest dans votre projet (mode `Compile`), Jenga2 utilisera les sources situées dans `%{Jenga.Unitest.Source}` (qui pointe vers ce dossier).
Aucune action manuelle n'est nécessaire – le projet `__Unitest__` est configuré automatiquement.

🐞 Dépannage

Problème	Cause probable	Solution
`Unitest is not configured`	Bloc `unitest()` manquant dans le workspace	Ajouter `with unitest(): u.Precompiled()`
`__Unitest__ not found`	Échec de création automatique du projet	Vérifier que `unitest` est bien utilisé avant les `test`
`undefined reference to nkentseu::test::...`	Mode précompilé mais lib manquante	S'assurer que `%{Unitest.libdir}` est accessible (binaire précompilé)
Le test ne s'exécute pas	Filtre actif ou `main()` déjà défini	Utiliser `testmainfile()` pour exclure le `main` parent
`PROFILE_TEST_SCOPE` ne génère pas de fichier	Profiler non initialisé	Ajouter `#include <Unitest/Profiler.h>` et appeler `BEGIN_PROFILING_SESSION`

🔗 Liens connexes

- Documentation Commands – voir `jenga2 test`
- API Jenga2 – contexte unitest – implémentation DSL
- Utilitaires Jenga2 – pour l'affichage des rapports
- Exemples de projets (à créer)

Unitest est un projet open-source interne à Jenga2. Les contributions sont les bienvenues.