Testing Rápido en C++

Introducción al Testing

El testing (pruebas de software) es una práctica fundamental en el desarrollo de software que consiste en verificar que el código funciona como se espera. Las pruebas nos ayudan a:

- **Detectar errores temprano**: Encontrar bugs antes de que lleguen a producción
- **Refactorizar con confianza**: Cambiar el código sabiendo que no rompemos funcionalidades existentes
- **Documentar el comportamiento**: Las pruebas sirven como ejemplos de uso del código
- **Mejorar el diseño**: Escribir código testeable suele resultar en mejor arquitectura

Tipos de Pruebas

1. **Unit Tests (Pruebas Unitarias)**: Prueban funciones o métodos individuales
2. **Integration Tests (Pruebas de Integración)**: Verifican que diferentes componentes trabajen juntos
3. **End-to-End Tests**: Prueban el sistema completo desde la perspectiva del usuario

Frameworks de Testing en C++

1. Catch2 (Recomendado para Principiantes)

Catch2 es un framework moderno, ligero y fácil de usar que solo requiere un archivo header.

Instalación de Catch2

```
# Opción 1: Descargar el header único
wget https://github.com/catchorg/Catch2/releases/download/v2.13.7/catch.hpp

# Opción 2: Usar vcpkg (Windows)
vcpkg install catch2

# Opción 3: Usar conan
conan install catch2/2.13.7@
```

Estructura Básica de un Test

```
#define CATCH_CONFIG_MAIN  // Le dice a Catch que proporcione la función main
#include "catch.hpp"

TEST_CASE("Descripción del test") {
    // Código del test aquí
    REQUIRE(2 + 2 == 4);
}
```

2. Google Test (gtest)

Framework robusto desarrollado por Google, ideal para proyectos más grandes.

```cpp
#include <gtest/gtest.h>

TEST(TestSuiteName, TestName) {
    EXPECT_EQ(2 + 2, 4);
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

3. Doctest

Similar a Catch2 pero más ligero y rápido.

```cpp
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include "doctest.h"

TEST_CASE("testing the factorial function") {
    CHECK(factorial(1) == 1);
    CHECK(factorial(2) == 2);
    CHECK(factorial(3) == 6);
}
```

Testing con Catch2 - Ejemplos Detallados

Ejemplo 1: Testing de Funciones Matemáticas

archivo: matematicas.h

```cpp
#ifndef MATEMATICAS_H
#define MATEMATICAS_H

class Matematicas {
public:
    static int sumar(int a, int b);
    static int restar(int a, int b);
    static int multiplicar(int a, int b);
    static double dividir(double a, double b);
    static long long factorial(int n);
    static bool esPrimo(int n);
};

#endif
```

archivo: matematicas.cpp

```cpp
#include "matematicas.h"
#include <stdexcept>
#include <cmath>

int Matematicas::sumar(int a, int b) {
    return a + b;
}

int Matematicas::restar(int a, int b) {
    return a - b;
}

int Matematicas::multiplicar(int a, int b) {
    return a * b;
}

double Matematicas::dividir(double a, double b) {
    if (b == 0) {
        throw std::invalid_argument("División por cero");
    }
    return a / b;
}

long long Matematicas::factorial(int n) {
    if (n < 0) {
        throw std::invalid_argument("Factorial de número negativo");
    }
    if (n == 0 || n == 1) {
        return 1;
    }

    long long resultado = 1;
    for (int i = 2; i <= n; i++) {
        resultado *= i;
    }
    return resultado;
}

bool Matematicas::esPrimo(int n) {
    if (n <= 1) return false;
    if (n <= 3) return true;
    if (n % 2 == 0 || n % 3 == 0) return false;

    for (int i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0) {
            return false;
        }
    }
    return true;
}
```

archivo: test_matematicas.cpp

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include "matematicas.h"

TEST_CASE("Pruebas de operaciones básicas", "[matematicas]") {

    SECTION("Suma") {
        REQUIRE(Matematicas::sumar(2, 3) == 5);
        REQUIRE(Matematicas::sumar(-1, 1) == 0);
        REQUIRE(Matematicas::sumar(0, 0) == 0);
        REQUIRE(Matematicas::sumar(-5, -3) == -8);
    }

    SECTION("Resta") {
        REQUIRE(Matematicas::restar(5, 3) == 2);
        REQUIRE(Matematicas::restar(0, 5) == -5);
        REQUIRE(Matematicas::restar(-2, -5) == 3);
    }

    SECTION("Multiplicación") {
        REQUIRE(Matematicas::multiplicar(3, 4) == 12);
        REQUIRE(Matematicas::multiplicar(-2, 3) == -6);
        REQUIRE(Matematicas::multiplicar(0, 100) == 0);
    }
}

TEST_CASE("Pruebas de división", "[division]") {

    SECTION("División normal") {
        REQUIRE(Matematicas::dividir(10.0, 2.0) == Approx(5.0));
        REQUIRE(Matematicas::dividir(7.0, 2.0) == Approx(3.5));
        REQUIRE(Matematicas::dividir(-10.0, 2.0) == Approx(-5.0));
    }

    SECTION("División por cero lanza excepción") {
        REQUIRE_THROWS_AS(Matematicas::dividir(5.0, 0.0), std::invalid_argument);
        REQUIRE_THROWS_WITH(Matematicas::dividir(5.0, 0.0), "División por cero");
    }
}

TEST_CASE("Pruebas de factorial", "[factorial]") {

    SECTION("Factoriales conocidos") {
        REQUIRE(Matematicas::factorial(0) == 1);
        REQUIRE(Matematicas::factorial(1) == 1);
        REQUIRE(Matematicas::factorial(2) == 2);
        REQUIRE(Matematicas::factorial(3) == 6);
        REQUIRE(Matematicas::factorial(4) == 24);
        REQUIRE(Matematicas::factorial(5) == 120);
    }

    SECTION("Factorial de número negativo lanza excepción") {
        REQUIRE_THROWS_AS(Matematicas::factorial(-1), std::invalid_argument);
        REQUIRE_THROWS_WITH(Matematicas::factorial(-5), "Factorial de número negativo")
;
    }
}

TEST_CASE("Pruebas de números primos", "[primos]") {

    SECTION("Números primos conocidos") {
        REQUIRE(Matematicas::esPrimo(2) == true);
```

```
        REQUIRE(Matematicas::esPrimo(3) == true);
        REQUIRE(Matematicas::esPrimo(5) == true);
        REQUIRE(Matematicas::esPrimo(7) == true);
        REQUIRE(Matematicas::esPrimo(11) == true);
        REQUIRE(Matematicas::esPrimo(13) == true);
    }

    SECTION("Números no primos") {
        REQUIRE(Matematicas::esPrimo(1) == false);
        REQUIRE(Matematicas::esPrimo(4) == false);
        REQUIRE(Matematicas::esPrimo(6) == false);
        REQUIRE(Matematicas::esPrimo(8) == false);
        REQUIRE(Matematicas::esPrimo(9) == false);
        REQUIRE(Matematicas::esPrimo(10) == false);
    }

    SECTION("Casos especiales") {
        REQUIRE(Matematicas::esPrimo(0) == false);
        REQUIRE(Matematicas::esPrimo(-1) == false);
        REQUIRE(Matematicas::esPrimo(-5) == false);
    }
}
```

Ejemplo 2: Testing de Clases

archivo: cuenta_bancaria.h

```
#ifndef CUENTA_BANCARIA_H
#define CUENTA_BANCARIA_H

#include <string>
#include <stdexcept>

class CuentaBancaria {
private:
    std::string titular;
    double saldo;

public:
    CuentaBancaria(const std::string& titular, double saldoInicial = 0.0);

    bool depositar(double cantidad);
    bool retirar(double cantidad);
    double getSaldo() const;
    std::string getTitular() const;

    // Operador de comparación para testing
    bool operator==(const CuentaBancaria& otra) const;
};

class SaldoInsuficienteException : public std::runtime_error {
public:
    SaldoInsuficienteException(const std::string& mensaje)
        : std::runtime_error(mensaje) {}
};

#endif
```

archivo: cuenta_bancaria.cpp

```cpp
#include "cuenta_bancaria.h"

CuentaBancaria::CuentaBancaria(const std::string& titular, double saldoInicial)
    : titular(titular), saldo(saldoInicial) {
    if (saldoInicial < 0) {
        throw std::invalid_argument("El saldo inicial no puede ser negativo");
    }
}

bool CuentaBancaria::depositar(double cantidad) {
    if (cantidad <= 0) {
        return false;
    }
    saldo += cantidad;
    return true;
}

bool CuentaBancaria::retirar(double cantidad) {
    if (cantidad <= 0) {
        return false;
    }
    if (cantidad > saldo) {
        throw SaldoInsuficienteException("Saldo insuficiente para realizar el retiro");
    }
    saldo -= cantidad;
    return true;
}

double CuentaBancaria::getSaldo() const {
    return saldo;
}

std::string CuentaBancaria::getTitular() const {
    return titular;
}

bool CuentaBancaria::operator==(const CuentaBancaria& otra) const {
    return titular == otra.titular && saldo == otra.saldo;
}
```

archivo: test_cuenta_bancaria.cpp

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include "cuenta_bancaria.h"

TEST_CASE("Construcción de cuenta bancaria", "[constructor]") {

    SECTION("Constructor con saldo inicial válido") {
        CuentaBancaria cuenta("Juan Pérez", 1000.0);
        REQUIRE(cuenta.getTitular() == "Juan Pérez");
        REQUIRE(cuenta.getSaldo() == Approx(1000.0));
    }

    SECTION("Constructor con saldo inicial cero") {
        CuentaBancaria cuenta("María García");
        REQUIRE(cuenta.getTitular() == "María García");
        REQUIRE(cuenta.getSaldo() == Approx(0.0));
    }

    SECTION("Constructor con saldo inicial negativo lanza excepción") {
        REQUIRE_THROWS_AS(CuentaBancaria("Error", -100.0), std::invalid_argument);
    }
}

TEST_CASE("Operaciones de depósito", "[deposito]") {
    CuentaBancaria cuenta("Test User", 100.0);

    SECTION("Depósito válido") {
        REQUIRE(cuenta.depositar(50.0) == true);
        REQUIRE(cuenta.getSaldo() == Approx(150.0));
    }

    SECTION("Múltiples depósitos") {
        cuenta.depositar(25.0);
        cuenta.depositar(25.0);
        REQUIRE(cuenta.getSaldo() == Approx(150.0));
    }

    SECTION("Depósito de cantidad negativa") {
        double saldoInicial = cuenta.getSaldo();
        REQUIRE(cuenta.depositar(-10.0) == false);
        REQUIRE(cuenta.getSaldo() == Approx(saldoInicial));
    }

    SECTION("Depósito de cero") {
        double saldoInicial = cuenta.getSaldo();
        REQUIRE(cuenta.depositar(0.0) == false);
        REQUIRE(cuenta.getSaldo() == Approx(saldoInicial));
    }
}

TEST_CASE("Operaciones de retiro", "[retiro]") {
    CuentaBancaria cuenta("Test User", 100.0);

    SECTION("Retiro válido") {
        REQUIRE(cuenta.retirar(30.0) == true);
        REQUIRE(cuenta.getSaldo() == Approx(70.0));
    }

    SECTION("Retiro que deja saldo cero") {
        REQUIRE(cuenta.retirar(100.0) == true);
        REQUIRE(cuenta.getSaldo() == Approx(0.0));
    }
```

```
    SECTION("Retiro con saldo insuficiente lanza excepción") {
        REQUIRE_THROWS_AS(cuenta.retirar(150.0), SaldoInsuficienteException);
        REQUIRE_THROWS_WITH(cuenta.retirar(150.0),
"Saldo insuficiente para realizar el retiro");
    }

    SECTION("Retiro de cantidad negativa") {
        double saldoInicial = cuenta.getSaldo();
        REQUIRE(cuenta.retirar(-10.0) == false);
        REQUIRE(cuenta.getSaldo() == Approx(saldoInicial));
    }

    SECTION("Retiro de cero") {
        double saldoInicial = cuenta.getSaldo();
        REQUIRE(cuenta.retirar(0.0) == false);
        REQUIRE(cuenta.getSaldo() == Approx(saldoInicial));
    }
}

TEST_CASE("Escenarios complejos", "[integracion]") {

    SECTION("Múltiples operaciones") {
        CuentaBancaria cuenta("Usuario Test", 1000.0);

        // Serie de operaciones
        cuenta.depositar(200.0);  // Saldo: 1200.0
        cuenta.retirar(300.0);    // Saldo: 900.0
        cuenta.depositar(50.0);   // Saldo: 950.0
        cuenta.retirar(150.0);    // Saldo: 800.0

        REQUIRE(cuenta.getSaldo() == Approx(800.0));
    }

    SECTION("Operaciones hasta agotar saldo") {
        CuentaBancaria cuenta("Usuario Test", 100.0);

        cuenta.retirar(25.0);  // Saldo: 75.0
        cuenta.retirar(25.0);  // Saldo: 50.0
        cuenta.retirar(50.0);  // Saldo: 0.0

        REQUIRE(cuenta.getSaldo() == Approx(0.0));

        // Intentar retirar de cuenta vacía
        REQUIRE_THROWS_AS(cuenta.retirar(1.0), SaldoInsuficienteException);
    }
}

// Test parametrizado usando GENERATE
TEST_CASE("Testing parametrizado de depósitos", "[parametrized]") {
    CuentaBancaria cuenta("Test", 0.0);

    double cantidad = GENERATE(10.0, 25.5, 100.0, 1000.0);

    SECTION("Depósito de cantidad positiva") {
        REQUIRE(cuenta.depositar(cantidad) == true);
        REQUIRE(cuenta.getSaldo() == Approx(cantidad));
    }
}
```

Compilación y Ejecución de Tests

Compilar con g++

```
# Compilar el código y los tests
g++ -std=c++11 -I. matematicas.cpp test_matematicas.cpp -o test_matematicas

# Ejecutar los tests
./test_matematicas

# Ejecutar tests con más verbosidad
./test_matematicas -s

# Ejecutar solo tests con tag específico
./test_matematicas "[matematicas]"
```

Usando CMake

archivo: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(TestingProject)

set(CMAKE_CXX_STANDARD 11)

# Encontrar Catch2
find_package(Catch2 REQUIRED)

# Crear ejecutable de tests
add_executable(tests
    matematicas.cpp
    cuenta_bancaria.cpp
    test_matematicas.cpp
    test_cuenta_bancaria.cpp
)

# Enlazar con Catch2
target_link_libraries(tests Catch2::Catch2)

# Habilitar CTest
enable_testing()
add_test(NAME unit_tests COMMAND tests)
```

Mejores Prácticas de Testing

1. Principio AAA (Arrange, Act, Assert)

```
TEST_CASE("Test siguiendo patrón AAA") {
    // Arrange (Organizar): Configurar el escenario
    CuentaBancaria cuenta("Test", 100.0);
    double cantidadDeposito = 50.0;

    // Act (Actuar): Ejecutar la acción
    bool resultado = cuenta.depositar(cantidadDeposito);

    // Assert (Afirmar): Verificar los resultados
    REQUIRE(resultado == true);
    REQUIRE(cuenta.getSaldo() == Approx(150.0));
}
```

2. Tests Independientes

Cada test debe ser independiente y poder ejecutarse en cualquier orden.

```
// MAL: Tests dependientes
TEST_CASE("Test que modifica estado global") {
    static int contador = 0;
    contador++;
    REQUIRE(contador == 1); // Falla si se ejecuta múltiples veces
}

// BIEN: Tests independientes
TEST_CASE("Test independiente") {
    int contador = 0; // Variable local
    contador++;
    REQUIRE(contador == 1); // Siempre funciona
}
```

3. Nombres Descriptivos

```
// MAL: Nombre poco descriptivo
TEST_CASE("test1") { /* ... */ }

// BIEN: Nombre descriptivo
TEST_CASE("debería lanzar excepción cuando se intenta retirar más dinero del dispon-
ible") {
    /* ... */
}
```

4. Testing de Casos Límite

```
TEST_CASE("Testing de casos límite en factorial") {
    // Casos límite
    REQUIRE(Matematicas::factorial(0) == 1);
    REQUIRE(Matematicas::factorial(1) == 1);

    // Valores extremos
    REQUIRE_THROWS_AS(Matematicas::factorial(-1), std::invalid_argument);
}
```

5. Uso de Fixtures para Configuración Común

```
class CuentaBancariaFixture {
protected:
    CuentaBancaria* cuenta;

    CuentaBancariaFixture() {
        cuenta = new CuentaBancaria("Test User", 1000.0);
    }

    ~CuentaBancariaFixture() {
        delete cuenta;
    }
};

TEST_CASE_METHOD(CuentaBancariaFixture, "Test usando fixture") {
    // 'cuenta' ya está disponible y configurada
    REQUIRE(cuenta->getSaldo() == Approx(1000.0));
}
```

Testing Avanzado

Mocking y Stubs

Para testing de componentes que dependen de otros servicios:

```
// Interface a mockear
class IBaseDatos {
public:
    virtual ~IBaseDatos() = default;
    virtual bool guardarUsuario(const std::string& nombre) = 0;
};

// Mock implementation
class MockBaseDatos : public IBaseDatos {
public:
    bool guardarUsuarioLlamado = false;
    std::string ultimoNombre;
    bool retornoGuardar = true;

    bool guardarUsuario(const std::string& nombre) override {
        guardarUsuarioLlamado = true;
        ultimoNombre = nombre;
        return retornoGuardar;
    }
};

TEST_CASE("Test con mock") {
    MockBaseDatos mockDb;
    ServicioUsuario servicio(&mockDb);

    servicio.registrarUsuario("Juan");

    REQUIRE(mockDb.guardarUsuarioLlamado == true);
    REQUIRE(mockDb.ultimoNombre == "Juan");
}
```

Property-Based Testing

Testing usando propiedades que siempre deben ser verdaderas:

```
TEST_CASE("Propiedades de ordenamiento") {
    std::vector<int> datos = GENERATE(take(10, chunk(100, random(1, 1000))));
    std::vector<int> ordenado = datos;
    std::sort(ordenado.begin(), ordenado.end());

    // Propiedad: El array ordenado debe tener el mismo tamaño
    REQUIRE(ordenado.size() == datos.size());

    // Propiedad: El array debe estar ordenado
    REQUIRE(std::is_sorted(ordenado.begin(), ordenado.end()));

    // Propiedad: Debe contener los mismos elementos
    std::sort(datos.begin(), datos.end());
    REQUIRE(datos == ordenado);
}
```

Integración con CI/CD

GitHub Actions

archivo: .github/workflows/ci.yml

```yaml
name: CI

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2

    - name: Install dependencies
      run: |
        sudo apt-get update
        sudo apt-get install -y cmake g++

    - name: Build
      run: |
        mkdir build
        cd build
        cmake ..
        make

    - name: Run tests
      run: |
        cd build
        ./tests
```

Herramientas Útiles

Coverage (Cobertura de Código)

```
# Compilar con información de coverage
g++ -std=c++11 --coverage matematicas.cpp test_matematicas.cpp -o test_matematicas

# Ejecutar tests
./test_matematicas

# Generar reporte de cobertura
gcov matematicas.cpp
lcov --capture --directory . --output-file coverage.info
genhtml coverage.info --output-directory coverage_report
```

Sanitizers para Detectar Errores

```
# AddressSanitizer (detecta memory leaks, buffer overflows)
g++ -fsanitize=address -g test_matematicas.cpp -o test_matematicas

# UndefinedBehaviorSanitizer
g++ -fsanitize=undefined -g test_matematicas.cpp -o test_matematicas

# ThreadSanitizer (para código multi-threaded)
g++ -fsanitize=thread -g test_matematicas.cpp -o test_matematicas
```

Conclusión

El testing es una habilidad esencial para cualquier programador de C++. Te permite:

- Escribir código más confiable
- Refactorizar sin miedo
- Documentar el comportamiento esperado
- Detectar regresiones temprano

Consejos finales:

1. Empieza simple con Catch2
2. Escribe tests para casos comunes y límite
3. Mantén los tests simples y enfocados
4. Ejecuta los tests frecuentemente
5. Trata los tests como código de primera clase

¡El testing es una inversión que siempre vale la pena!