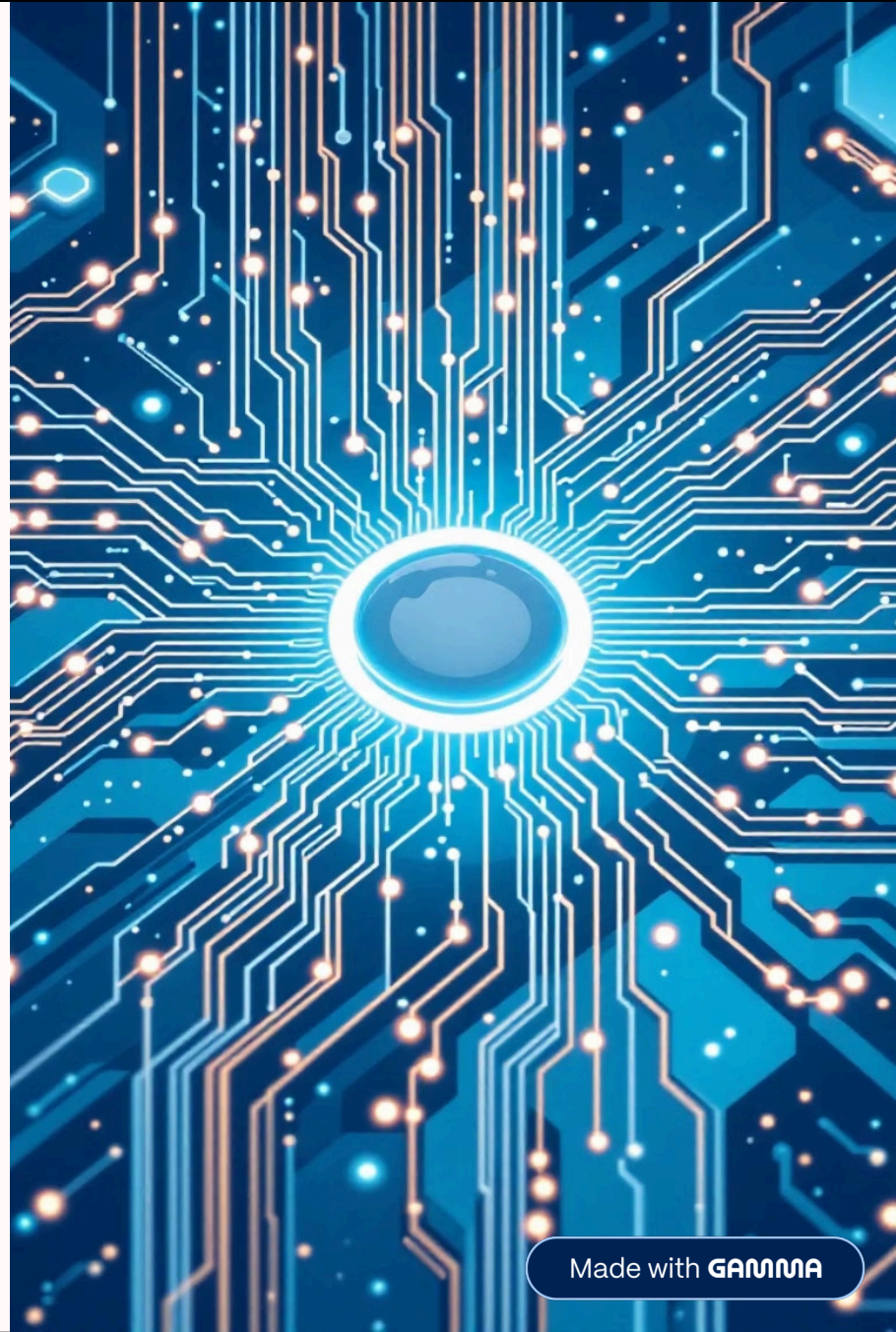
Testes Unitários em C com µTest (utest.h)

Por: Arthur Oliveira

Data: 12/2025



Por Que Testar em C?

Confiabilidade Inabalável

Garanta que seu código funcione como esperado, mesmo sob pressão, essencial para sistemas críticos.

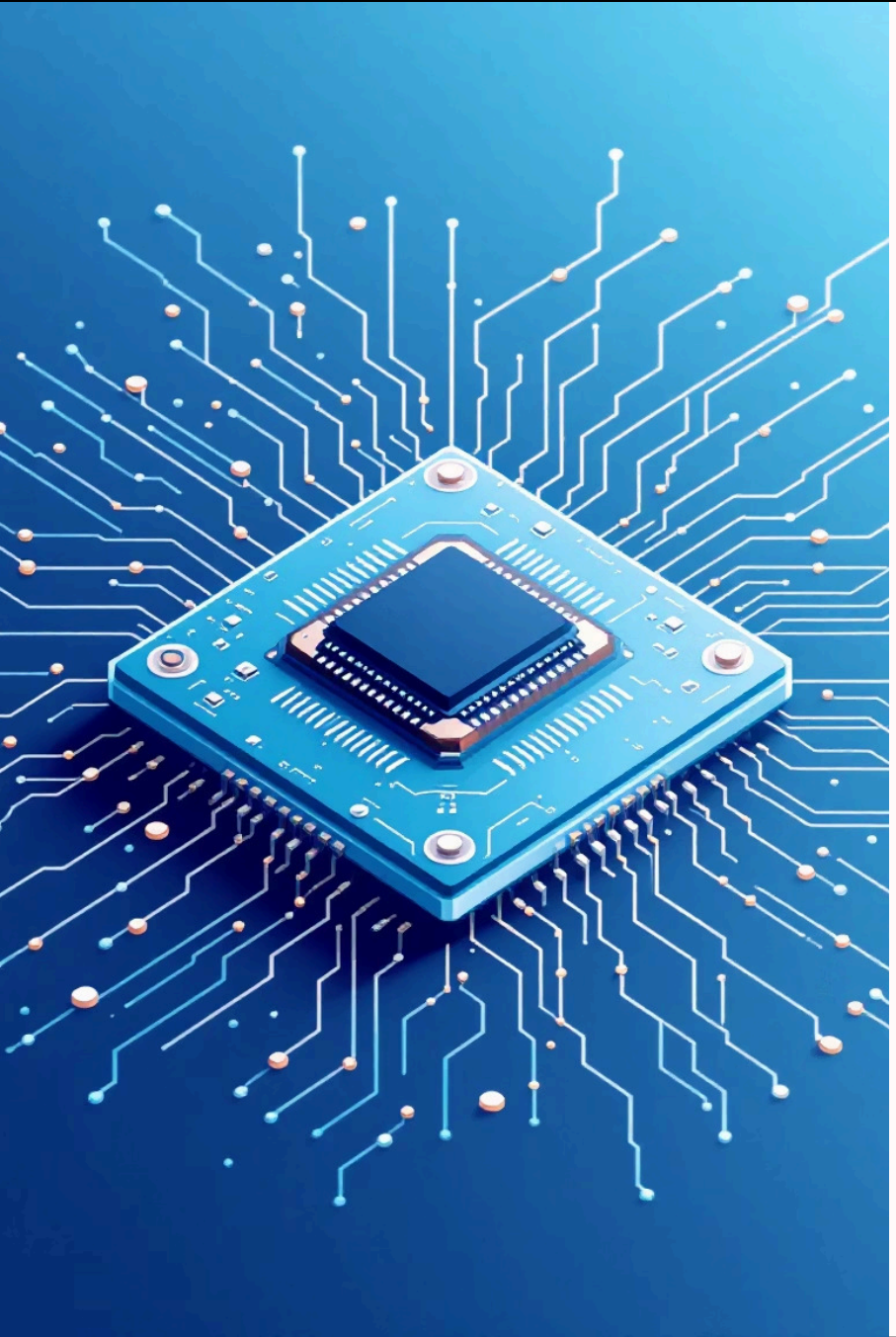
Detecção Precoce de Erros

Identifique falhas no início do desenvolvimento, economizando tempo e recursos valiosos.

Manutenção Simplificada

Facilite futuras modificações e refatorações, sabendo que seus testes protegerão a integridade do sistema.

Testes unitários são a base para construir software e firmware robustos, minimizando surpresas indesejadas e elevando a qualidade do produto final.



µTest: Seu Aliado nos Testes em C

O Que é?

Um framework de testes unitários em C, projetado para ser leve e eficiente.

Header-Only

Extremamente portátil e fácil de integrar – apenas um arquivo de cabeçalho! Perfeito para projetos embarcados.

Ideal para Firmware

Sua simplicidade e baixa pegada de memória o tornam a escolha ideal para sistemas embarcados e bibliotecas C.

Ponto a Ponto

Mais simples que frameworks complexos como Google Test, focado na funcionalidade essencial sem sobrecarga.

Nosso Ambiente de Desenvolvimento

Sistema Operacional

Todo o processo será demonstrado no ambiente Windows, comum para muitos desenvolvedores e estudantes.

Editor de Código

Utilizaremos o Visual Studio Code, um editor moderno e versátil, com extensões que facilitam o desenvolvimento em C.

Compilação e Execução

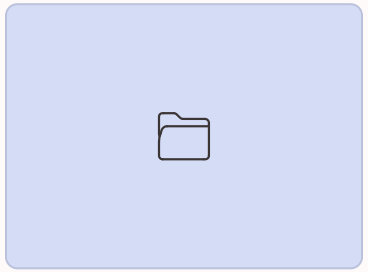
MSYS2 (MINGW64 ou UCRT64) com GCC será nossa ferramenta para compilar e executar os testes.

Por que MSYS2?

Oferece um ambiente Linux-like no Windows, garantindo compatibilidade com ferramentas GNU e um GCC robusto, diferente do terminal integrado do VS Code.

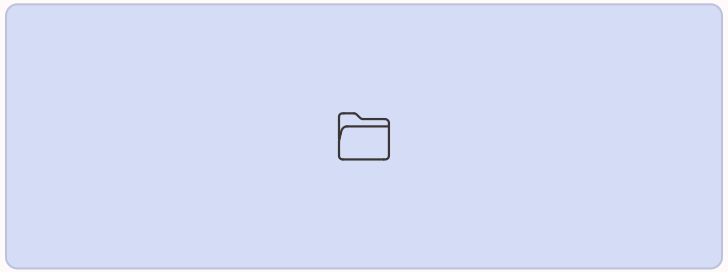
Estrutura do Projeto

Uma organização clara é fundamental para a manutenção e escalabilidade do seu projeto.



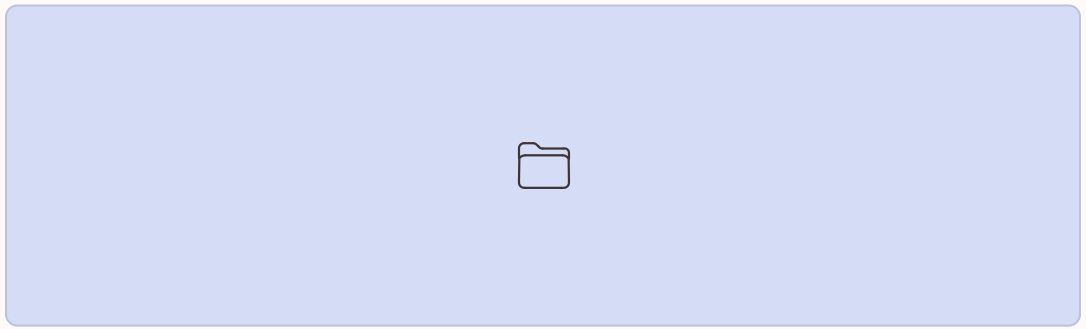
tests/

Contém todos os arquivos de testes unitários (.c), separados do código de produção para evitar dependências cíclicas e manter a modularidade.



src/

Aqui residem os arquivos de código-fonte da sua aplicação ou biblioteca (.c), ou seja, a lógica de negócio que será testada.



inc/

Armazena os arquivos de cabeçalho (.h) da sua aplicação e o próprio `utest.h`, garantindo fácil acesso às definições necessárias.

Exemplo Prático: Uma Calculadora Simples

Biblioteca C

Desenvolveremos uma biblioteca de calculadora com operações básicas para ilustrar os conceitos.

Retorno de Status

As funções retornarão códigos de status (ex: 0 para sucesso, -1 para erro) para indicar o resultado da operação.

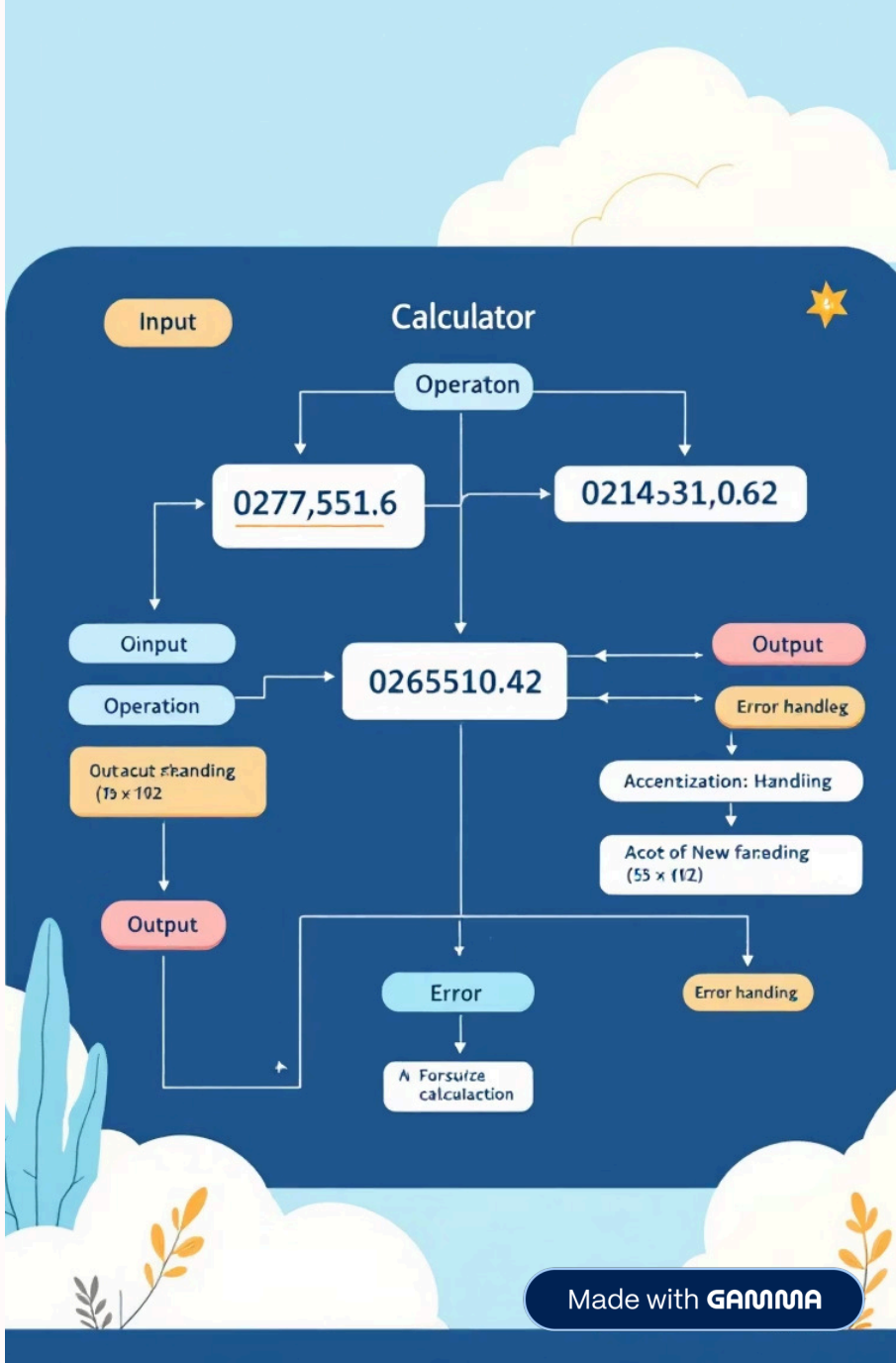
Resultado via Ponteiro

O resultado da operação será passado por ponteiro, permitindo que a função retorne o status.

Tratamento de Erros

Incluiremos tratamento de casos de erro, como ponteiro nulo para o resultado ou divisão por zero, e testaremos esses cenários.

Este exemplo nos permitirá explorar diversos aspectos dos testes unitários em C de forma controlada e didática.



ASSERT vs. EXPECT no µTest

Entender a diferença entre ASSERT e EXPECT é crucial para um controle preciso do fluxo de seus testes unitários.

ASSERT: Interrupção Imediata

Interrompe o teste no primeiro erro, indicando uma falha crítica que inviabiliza a continuação das verificações subsequentes.

EXPECT: Continuidade do Teste

Registra a falha, mas permite que o teste prossiga, útil para verificar múltiplos problemas ou condições menos críticas em um único cenário.

Use **ASSERT** para condições fatais e **EXPECT** para falhas não fatais.

Escrevendo Testes com µTest

`UTEST(grupo, nome)`

Define um novo teste unitário. grupo agrupa testes relacionados e nome descreve o cenário específico.

`ASSERT_EQ(esperado, obtido)`

Verifica se o valor esperado é **igual** ao valor obtido. Fundamental para a maioria das verificações.

`ASSERT_NEAR(esperado, obtido, precisão)`

Ideal para comparar números de ponto flutuante, onde uma **precisão** é aceitável devido a imprecisões de representação.

`UTEST_MAIN()`

Macro que gera a função `main` para executar todos os testes definidos. É o ponto de entrada do executável de testes.

Com essas ferramentas, podemos criar testes robustos para garantir o comportamento correto de cada função.

Comparando Valores

O µTest oferece um conjunto robusto de macros para verificar valores, garantindo que seu código se comporte exatamente como esperado.

Igualdade e Diferença

Verifique se dois valores são iguais ou diferentes. Essencial para códigos de retorno e validação de estado.

- `ASSERT_EQ(x, y)`: Interrompe o teste se `x != y`.
- `EXPECT_NE(x, y)`: Continua o teste mesmo se `x == y`.
- Versões `_MSG` disponíveis para mensagens personalizadas.

Uso Comum

Ideal para:

- Códigos de retorno de funções (`CALC_OK`, `CALC_ERROR`)
- Validação do estado de variáveis ou objetos após operações
- Verificação de cenários de erro esperados

Comparações Relacionais

Teste limites, faixas e comportamentos com macros para maior ou menor que.

- `EXPECT_LT(x, y)` / `ASSERT_LT(x, y)`: Verifica se `x < y`.
- `EXPECT_LE(x, y)` / `ASSERT_LE(x, y)`: Verifica se `x <= y`.
- `EXPECT_GT(x, y)` / `ASSERT_GT(x, y)`: Verifica se `x > y`.
- `EXPECT_GE(x, y)` / `ASSERT_GE(x, y)`: Verifica se `x >= y`.

Aplicações Práticas

Perfeito para:

- Testar limites de valores de entrada ou saída
- Validar faixas de dados numéricos
- Prevenir overflow ou underflow em operações

A escolha entre `ASSERT` e `EXPECT` depende se você deseja interromper a execução do teste imediatamente ou apenas registrar a falha e continuar.

O que acontece internamente: UTEST_COND

Por trás de cada macro de verificação como `ASSERT_EQ` ou `EXPECT_NE`, o µTest utiliza uma macro fundamental chamada `UTEST_COND`. Ela é o coração da avaliação de condições, encapsulando a lógica de comparação e determinando o comportamento do teste em caso de falha. Compreender seu funcionamento conceitual ajuda a dominar o framework.

```
UTEST_COND(x, y, operador_str, operador_func, mensagem, fatal)
```

Os parâmetros mais relevantes para entender seu comportamento são:

- `operador_str`: A representação textual do operador de comparação, como `"=="`, `"!="`, `"<"`, etc. Essencial para mensagens de erro claras.
- `operador_func`: A função interna que efetivamente realiza a comparação entre `x` e `y`.
- `fatal`: Um flag booleano que define a gravidade da falha:
  - Se `fatal = 1`, a condição falhou e o teste é **interrompido imediatamente** (comportamento de `ASSERT`).
  - Se `fatal = 0`, a condição falhou, mas o teste **continua sua execução**, apenas registrando a falha (comportamento de `EXPECT`).

Quando usar ASSERT ou EXPECT?

A escolha entre `ASSERT` e `EXPECT` é uma decisão estratégica que afeta a granularidade e a robustez dos seus testes. Utilize esta orientação prática:

Validação crítica, pré-condição essencial

`ASSERT`

Verificar múltiplos resultados ou efeitos colaterais

`EXPECT`

Checagem de código de retorno de função

`ASSERT_EQ`

Teste de limites (maior que, menor que)

`EXPECT_LT` / `EXPECT_GT`

Debug detalhado com mensagens personalizadas

`*_MSG` (ambos `ASSERT` e `EXPECT`)

Compilação e Execução

Vamos ver como transformar nosso código e testes em um executável funcional.

01

Compilando com GCC

Utilizamos o comando `gcc` para compilar os arquivos `.c` do nosso código de produção e dos testes.

02

Inclusão dos Fontes

É crucial incluir tanto os arquivos de `src/` quanto os de `tests/` na linha de comando do compilador.

03

Execução do Binário

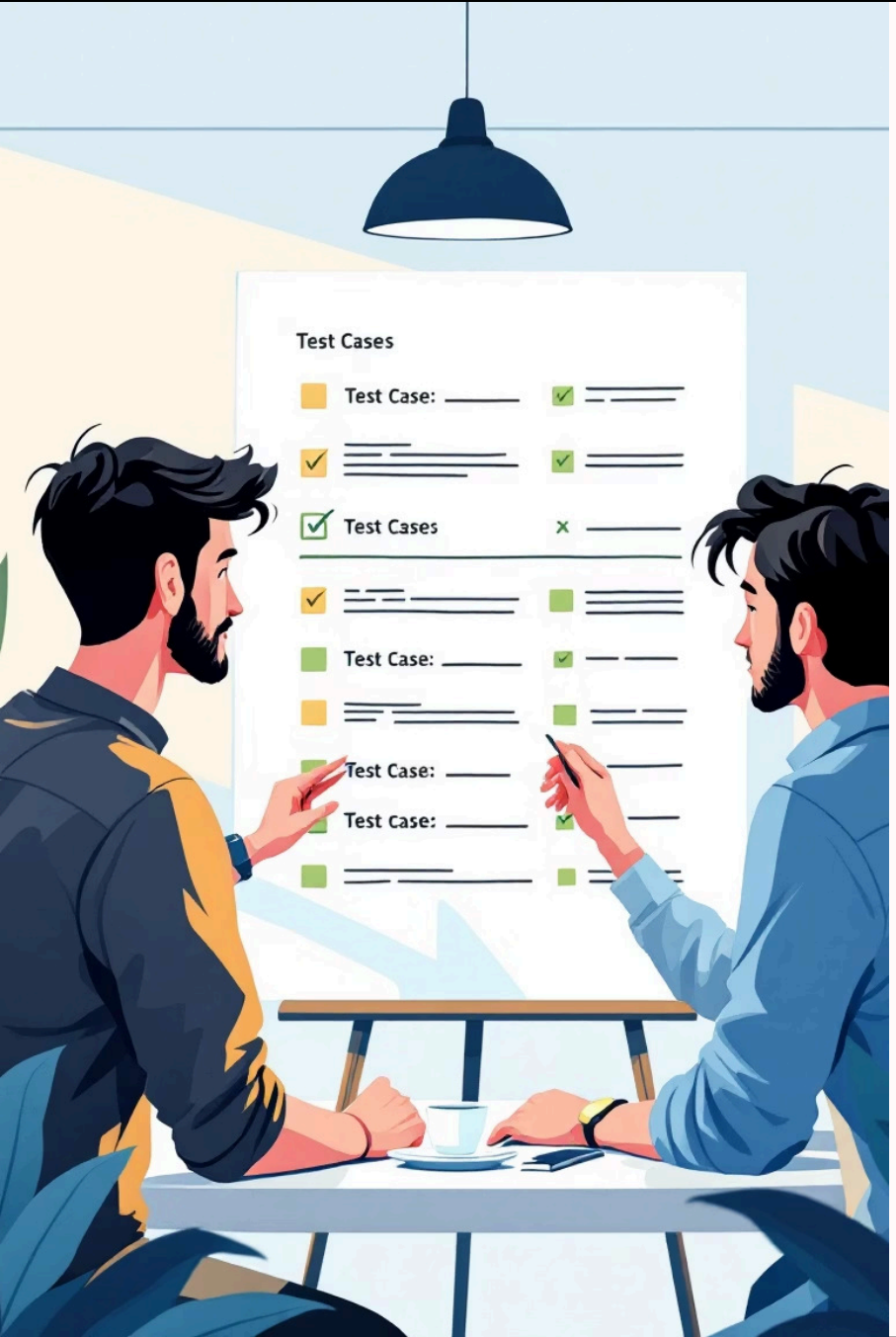
Após a compilação, basta executar o arquivo binário gerado. Ele rodará todos os testes e exibirá os resultados.

04

Saída do µTest

O µTest fornecerá um resumo claro, indicando quantos testes passaram, quantos falharam e o tempo total de execução.

Exemplo de comando: gcc -Iinc src/*.c tests/*.c -o run_tests



Boas Práticas e Dicas de Ouro

- Cada módulo de código (e.g., `calculadora.c`) deve ter seu próprio arquivo de teste (e.g., `test_calculadora.c`).
- Mantenha seus testes pequenos, focados e independentes. Um teste deve verificar apenas uma funcionalidade ou cenário.
- Use nomes claros e descritivos para grupos de testes e para os testes individuais. Isso facilita a compreensão e depuração.
- Não teste apenas o caminho feliz! Inclua testes para cenários de erro, valores limite e entradas inválidas para garantir a robustez do seu código.

Conclusão: O Futuro do Seu Código

Quando Usar?

Sempre que a confiabilidade for crítica: firmware, drivers, bibliotecas e qualquer software em C.

Vantagens

Leveza, portabilidade e facilidade de uso, sem abrir mão da robustez nos testes.

Repositório Oficial

Encontre o µTest e contribua em: **github.com/sheredom/utest.h**

Adote os testes unitários! Eles são um investimento que se paga com código mais seguro, menos bugs e maior confiança nos seus projetos acadêmicos e profissionais.