Relatório Técnico - Batalha Naval
Programação Imperativa e Funcional

Professor: João Victor Tinoco de Souza Abreu
Alunos: Heitor Castilhos de Melo, Lucas Chaves de Albuquerque, Luiz Gustavo Gonçalves da Silva.
Turma: Sistemas de Informação 2025.2

---

1. Introdução

Este relatório apresenta, de forma resumida, o desenvolvimento de um jogo de **Batalha Naval em linguagem C**, realizado como parte do Projeto Integrador Final (PIF) do curso de Sistemas de Informação da CESAR School.

O objetivo principal do sistema é implementar o jogo para dois jogadores em modo texto, **demonstrando domínio de**:

- uso de **structs** para modelar entidades como tabuleiro, navio e jogador;
- **ponteiros** para passagem de estruturas por referência;
- **alocação dinâmica de memória** com `malloc`, `realloc` e `free`;
- **modularização** do código em múltiplos arquivos `.c` e `.h`, com separação clara de responsabilidades.

---

2. Arquitetura do Projeto

O projeto foi organizado em módulos, cada um responsável por uma parte da lógica do sistema, seguindo a estrutura sugerida no enunciado.

- **`board.h` / `board.c` – Módulo do Tabuleiro**
  Responsável por representar o tabuleiro do jogo. Define o estado de cada célula, guarda a quantidade de linhas e colunas e oferece funções para inicialização, liberação e acesso seguro às células.

- **`fleet.h` / `fleet.c` – Módulo da Frota**
  Modela os navios individuais e a frota como um todo. Controla nomes, tamanhos, quantidade de acertos e verifica quando todos os navios de um jogador foram afundados.

- **`game.h` / `game.c` – Módulo do Jogo**
  Agrupa as estruturas de **Jogador** e **Jogo**. Implementa a lógica de inicialização dos jogadores, posicionamento manual e automático da frota e o fluxo de um tiro (ataque) em um turno.

- **`io.h` / `io.c` – Entrada e Saída (CLI)**
  Centraliza toda interação com o usuário via console: exibição do tabuleiro, leitura de coordenadas como `"B5"` e impressão de mensagens de erro/sucesso.

- **`rnd.h` / `rnd.c` – Aleatoriedade**
  Isola a inicialização do gerador de números aleatórios (`srand(time(NULL))`) e a geração de números em faixas específicas, usados principalmente no posicionamento automático dos navios.

- **`main.c` – Ponto de Entrada**
  Implementa o **menu inicial** (Novo jogo, Configurações, Sair), lê as configurações do tabuleiro e do modo de posicionamento, cria a estrutura de jogo e controla o loop principal até que haja um vencedor, exibindo as estatísticas ao final.

- **`Makefile` – Automação de Compilação**
  Automatiza o processo de compilação, gerando o executável a partir dos arquivos `.c` usando `gcc` com flags de aviso (`-Wall -Wextra`)..

---

3. Estruturas de Dados

Nesta seção explicamos como as principais estruturas foram projetadas e como se relacionam.

3.1. Tabuleiro e Células

O tabuleiro é representado pela struct `Tabuleiro`, que guarda:

- `linhas` e `colunas`;

- um ponteiro `Celula *celulas`, alocado dinamicamente com `malloc(linhas * colunas * sizeof(Celula))`.

Cada `Celula` contém:

- um `EstadoCelula` (enum) indicando se ali há **água**, **navio**, **acerto** ou **erro**;

- um `id_navio` que armazena o índice do navio na frota daquele jogador, ou `-1` caso não haja navio naquela posição.

Essa combinação permite:

- representar o tabuleiro como **um vetor 1D** em vez de uma matriz 2D, economizando detalhes de sintaxe e simplificando a alocação;
- localizar a célula correspondente a uma coordenada `(linha, coluna)` pela fórmula `indice = linha * colunas + coluna`.

A função `obterCelula(Tabuleiro *t, int linha, int coluna)` encapsula esse cálculo e **faz a validação de limites**, retornando `NULL` se houver tentativa de acesso fora do tabuleiro.

3.2. Frota e Navios

A frota é modelada pela struct `Frota`, que possui:

- um ponteiro `Navio *navios`, que é um vetor dinâmico;
- um inteiro `quantidade`, indicando quantos navios existem.

Cada `Navio` contém:

- `nome` (ex.: `"Porta-avioes"`, `"Cruzador"`);
- `tamanho` (quantas células ocupa);
- `acertos` (quantas vezes já foi atingido).

A frota é construída em `criarFrotaPadrao`, que utiliza `adicionarNavioNaFrota`. Essa função usa `realloc` para expandir o vetor de navios conforme novos navios são adicionados. Isso torna a solução mais flexível do que um array estático.

A função `frotaAfundou` percorre todos os navios e verifica se `acertos == tamanho` para cada um. Se isso valer para todos, a frota é considerada completamente afundada, e o jogo termina.

3.3. Jogador e Jogo

A struct `Jogador` agrupa tudo o que pertence a um participante:

- `Tabuleiro tabuleiro` – onde seus navios estão;
- `Tabuleiro tiros` – tabuleiro separado para registrar os tiros que ele já deu (radar);
- `Frota frota` – a coleção de navios;
- `apelido[32]` – nome do jogador;
- contadores de estatísticas (total de tiros e acertos).

A struct `Jogo` contém:

- dois `Jogador` (j1 e j2);
- um inteiro `jogador_atual` para controlar de quem é o turno;
- uma flag `jogo_acabou` para encerrar o loop principal;
- As configurações de linhas e colunas do tabuleiro.

---

4. Fluxo de Execução

O fluxo geral do jogo segue a sequência abaixo:

1. **Menu inicial** (`main.c`):
   O usuário escolhe entre **Novo jogo**, **Configurações** ou **Sair**.
2. **Configurações**:
   - Definição do tamanho do tabuleiro (entre 6 e 26);
   - Escolha do modo de posicionamento da frota: **manual** ou **automático**.
3. **Inicialização dos jogadores** (`inicializarJogador`):
   Para cada jogador são criados:
   - o tabuleiro de navios;

- o tabuleiro de tiros;
- a frota com os navios padrão.

4. **Posicionamento dos navios**:
   - **Automático**: `posicionarFrotaAuto` usa números aleatórios para tentar posicionar cada navio em coordenadas válidas, com orientações horizontal ou vertical, sem colisão.
   - **Manual**: `posicionarFrotaManual` solicita coordenadas do tipo `"A1"` e orientação (`H`/`V`), valida as posições com `podeColocarNavio` e só permite concluir o posicionamento se for seguro.
5. **Loop de turnos**:
   Enquanto nenhuma frota estiver completamente afundada:
   - o jogador atual fornece uma coordenada de tiro (`lerCoordenada`);
   - `jogadorAtira` verifica se o tiro é válido, atualiza o tabuleiro do alvo e o tabuleiro de tiros do atirador;
   - as estatísticas de tiros e acertos são atualizadas;
   - é feita a verificação de vitória usando `frotaAfundou`.
6. **Detecção de vitória e encerramento**:
   Quando `frotaAfundou` indicar que todas as embarcações de um jogador foram destruídas, o jogo marca `jogo_acabou` e exibe o vencedor e as estatísticas (total de tiros, acertos e precisão).

---

5. Decisões de Design

Algumas escolhas foram feitas conscientemente para simplificar o código e torná-lo mais seguro:

- **Tabuleiro como vetor 1D**
  Em vez de usar `Cell board[LIN][COL]`, optou-se por um vetor linear (`Celula *celulas`).
  Isso facilita a alocação dinâmica (`malloc(linhas * colunas)`), evita limites fixos e

permite mudar o tamanho do tabuleiro em tempo de execução.

- **Uso de `id_navio` na célula**
  Cada célula guarda o índice do navio na frota. Isso torna simples atualizar os acertos de um navio após um tiro: basta olhar `cel_alvo->id_navio` e incrementar `navios[id].acertos`.
  Evite buscas desnecessárias e deixe a lógica de “navio afundar” mais direta.

- **Orientação horizontal/vertical como `enum`**
  Em vez de usar `0` e `1` soltos, foi definido `Orientacao` com `ORIENTACAO_H` e `ORIENTACAO_V`. Isso torna o código mais legível e reduz a chance de erros ao interpretar os valores.

- **Módulo `rnd.c` separado**
  A geração de números aleatórios foi isolada em `rnd.c` para que o restante do código não dependa diretamente de `rand()` e `srand()`. Assim, se a lógica de aleatoriedade mudar no futuro, basta alterar esse módulo.

- **Centralização de I/O em `io.c`**
  Toda a interação com o usuário (desenhar tabuleiro, ler coordenadas, mostrar mensagens) foi concentrada em um único módulo. Isso evita espalhar `printf`/`scanf` pelo código e facilita possíveis mudanças na interface (por exemplo, mudar o formato de exibição sem alterar a lógica de jogo).

- **Adoção de Nomenclatura em Português** Embora o enunciado do projeto utilizasse termos em inglês (como `Board`, `Fleet`, `Cell`) para exemplificar as estruturas, o grupo optou por traduzir identificadores de funções, variáveis e *structs* para o português (ex.: `Tabuleiro`, `Frota`, `Celula`). Essa decisão visou manter a **consistência semântica** do projeto, alinhando o código-fonte com a interface de usuário (CLI) e os comentários explicativos, que já seriam em português. Além disso, essa escolha facilitou a comunicação interna da equipe e o raciocínio sobre as regras de negócio durante a implementação das lógicas de ponteiros e memória.

---

6. Gestão de Memória

Como o projeto exige uso explícito de `malloc` e `realloc`, foi necessário planejar com cuidado a alocação e liberação de memória.

- **Alocação do tabuleiro**
  Em `inicializarTabuleiro`, o ponteiro `t->celulas` recebe memória via `malloc(total * sizeof(Celula))`.
  Em seguida, o código verifica se o retorno é `NULL` e encerra o programa com uma mensagem de erro em caso de falha, evitando acessar memória inválida.

- **Alocação da frota**
  A função `adicionarNavioNaFrota` usa `realloc` para aumentar o vetor de navios conforme novos navios são inseridos. Se `realloc` falhar, o programa imprime uma mensagem de erro e encerra, garantindo que o ponteiro antigo não é perdido sem controle.

- **Liberação de memória**

  - `liberarTabuleiro` verifica se `t->celulas != NULL`, chama `free(t->celulas)` e então seta o ponteiro para `NULL`, além de zerar `linhas` e `colunas`. Isso evita **dangling pointers** (ponteiros apontando para memória já liberada).

  - `liberarFrota` faz o mesmo para o vetor de navios: chama `free(f->navios)`, zera o ponteiro e a quantidade.

- **Evitar acesso fora dos limites**
  A função `obterCelula` checa explicitamente se `linha` e `coluna` estão dentro de `[0, linhas)` e `[0, colunas)`. Se não estiverem, retorna `NULL`.
  Funções como `podeColocarNavio` e `posicionamento` usam essas validações para garantir que não há acesso fora do vetor `celulas`.

---

7. Testes e Validação

Para garantir que o jogo funcionasse sem “quebrar”, foram feitos diversos testes manuais:

- **Testes de posicionamento**

  - Tabuleiros com tamanhos diferentes (6x6, 10x10, valores próximos de 26);

  - Tentativas de posicionar navios nas bordas, para verificar se o código recusa corretamente navios que sairiam do tabuleiro;

    - Tentativas de sobrepor navios, garantindo que `podeColocarNavio` detecta colisões.

- **Testes de entrada de coordenadas**
    - Coordenadas válidas (`A1`, `B5`, etc.);
    - Coordenadas inválidas (letras fora do intervalo, números maiores que o tamanho do tabuleiro, strings estranhas), verificando se a função `lerCoordenada` recusa a jogada.
- **Testes de loop de jogo**
    - Partidas completas com diferentes tamanhos de tabuleiro;
    - Testes de tiros repetidos na mesma posição, para certificar que o jogo não permite jogar duas vezes no mesmo lugar;
    - Verificação se, ao afundar todos os navios de um jogador, a função `frotaAfundou` é chamada corretamente e o jogo termina.
- **Testes de memória**
  Embora não tenha sido usado um analisador externo (como Valgrind), o grupo revisou o código para garantir que todo `malloc`/`realloc` tivesse um `free` correspondente e que nenhum ponteiro fosse utilizado após a liberação.

---

8. Conclusão

O desenvolvimento deste projeto de Batalha Naval em C permitiu ao grupo praticar, em um contexto concreto, conceitos que muitas vezes ficam abstratos em exemplos menores: **ponteiros**, **alocação dinâmica**, **structs**, **enums**, **modularização** e **organização de código em múltiplos arquivos**.

A divisão de responsabilidades por módulo (tabuleiro e frota, lógica do jogo, entrada/saída, aleatoriedade, função principal e Makefile) ajudou a organizar o trabalho e simulou um cenário real de desenvolvimento em equipe, em que cada integrante cuida de partes diferentes de um mesmo sistema.

Entre os principais aprendizados estão:

- maior segurança ao trabalhar com memória dinâmica e **ponteiros**;

- melhor compreensão de como organizar projetos em C usando cabeçalhos e múltiplos arquivos;

- experiência prática em pensar primeiro na **arquitetura e nas estruturas de dados**, e só depois no código.

Em uma próxima versão, seria possível evoluir o projeto adicionando, por exemplo, diferentes modos de jogo, níveis de dificuldade ou até uma camada de interface gráfica, mantendo a mesma base de lógica e estruturas definida neste trabalho.