Introdução

No desenvolvimento do jogo de Batalha Naval usamos a linguagem C para a codificação. Nessa etapa do processo, usamos toda a estrutura sugerida pelo professor João Tinoco para o desenvolvimento, ou seja, todos os nomes de arquivos, estrutura da modularização, os structs, o design e o fluxo do jogo foram feitos de maneira igual. Com isso, reaproveitamos todos os ponteiros e outros elementos dos structs para fazer a nossa lógica do código e alocação dinâmica de memória.

Arquitetura do Projeto

O código foi dividido em módulos para garantir manter o código mais legível, profissional e mais fácil de encontrar erros.

Módulos do sistema:

Módulo	Função
main.c	O arquivo main controla o menu do jogo e chama as funções de outros módulos para o jogo ser executado.
game.c / game.h	Regras gerais do jogo, controle de turnos, detecção de vitória.
board.c / board.h	Gerenciamento do tabuleiro (criação, prints e acesso a células).
fleet.c / fleet.h	Gerenciamento da frota e posicionamento dos navios.
io.c / io.h	Entrada e saída de dados (menu, leitura de coordenadas, configurações).
rnd.c / rnd.h	Funções auxiliares de sorteio.

Diagrama de fluxos de chamadas

O programa começa em `main.c`.
Ele mostra o menu e quando o usuário escolhe jogar ele chama uma função em `game.c`.

`game.c` controla toda a lógica do jogo.
Para funcionar, ele chama outros módulos quando precisa.

Quando precisa usar navios, `game.c` chama funções de `fleet.c`.

Quando precisa mexer no tabuleiro, tanto `game.c` quanto `fleet.c` chamam funções de `board.c`.

Quando precisa ler dados do usuário ou mostrar informações na tela, `game.c` chama `io.c`.

Quando precisa gerar números aleatórios, `fleet.c` chama funções de `rnd.c`.

Estruturas de Dados

A base do sistema é composta essencialmente por três structs principais:

1 Board

Representa o tabuleiro de cada jogador.

```
typedef struct {
    int rows, cols;
    Cell **cells;
} Board;
```

Cada célula possui um estado (água, navio, acerto ou erro).
O uso de `cells` como matriz dinâmica evita limitar o tamanho fixo do tabuleiro.

2 Ship

Representa cada navio individualmente.

```
typedef struct {
    char *name;
    int length;
    int hits;
    int row, col;
    Orientation orient;
} Ship;
```

Armazena posição inicial, orientação e quantidade de acertos.

3 Fleet

Contém todos os navios do jogador.

```
typedef struct {
    Ship *ships;
    int total_ships;
} Fleet;
```

4 Player

Representa o estado completo de cada jogador:

```
typedef struct {
    char nickname[50];
    Board board;
    Board shots;
    Fleet fleet;
} Player;
```

Cada jogador tem:

- um tabuleiro
- um mapa de tiros
- uma frota completa
- um apelido.

Uso de alocação dinâmica

O projeto exige `malloc()` para:

- criar o tabuleiro de tamanho configurável,
- criar matrizes bidimensionais,
- armazenar frotas.

O uso de alocação dinâmica permite:

- flexibilidade no tamanho do grid,
- independência entre módulos,
- limpeza total via `free()` ao final.

Garantias de segurança

Evitamos acessos indevidos:

- Função `getCell()` valida o acesso centralizado.
- Toda coordenada digitada é conferida antes de acessar o tabuleiro.
- Navios só são posicionados após garantir que o espaço está livre.
- Tiros repetidos não são permitidos.

Fluxo de Execução

O jogo segue um fluxo claro e sequencial:

1 Menu inicial

O usuário escolhe:

1. **Novo jogo**
2. **Configurações**
3. **Sair**

2 Configurações

- Definição do tamanho do tabuleiro.
- Definição se o posicionamento será automático ou manual.

3 Criação dos jogadores

Para cada jogador:

- leitura do apelido,
- criação dos tabuleiros (`board` e `shots`),
- criação da frota,
- posicionamento (manual ou automático).

4 Loop principal de turnos

O jogo alterna entre os jogadores:

1. mostra de quem é o turno,
2. solicita coordenada,
3. valida entrada,
4. aplica acerto ou erro no tabuleiro do alvo,
5. registra no tabuleiro de tiros.

O jogador mantém sua vez até inserir uma coordenada válida (regra implementada no projeto).

5 Detecção de vitória

A cada acerto, verificamos:

- Se o navio afundou,
- Se todos os navios do jogador foram destruídos.

Quando isso ocorre:

- o loop termina,
- o vencedor é anunciado.

Decisões de Design

1 Organização horizontal/vertical

A orientação dos navios é definida por:

```
typedef enum { ORIENT_H, ORIENT_V } Orientation;
```

O uso de `enum` evita “números mágicos” e aumenta a clareza.

2 Validação de coordenadas

A coordenada é composta por:

- letra da coluna (A, B, C…)
- número da linha (1…N)

A validação inclui:

- verificar limites,
- garantir que a pessoa digitou algo no formato correto,
- recusar coordenadas inválidas sem trocar de turno.

5.3 Uso de módulo rnd.c

O módulo `rnd.c` centraliza:

- geração de números aleatórios,
- inicialização de seeds.

Isso impede que cada módulo chame rand() por conta própria.

5.4 Benefícios da modularização

- Permite alterar o tabuleiro sem afetar o resto do sistema.
- Permite trocar a lógica do jogo sem mudar I/O.
- Facilita testes independentes.
- Aumenta legibilidade e reutilização.

6. Gestão de Memória

1 Alocação

O tabuleiro é criado com:

```
cells = malloc(rows * sizeof(Cell*));
```

e cada linha com:

```
cells[i] = malloc(cols * sizeof(Cell));
```

A frota é alocada dinamicamente de acordo com o número de navios.

2 Liberação

Cada módulo possui sua própria função de liberação:

- `liberarBoard()`
- `liberarFrota()`
- `liberarPlayer()`

Apesar de não terem sido liberados na ordem inversa de alocação, não tem vazamento de memória.

3 Prevenção de erros

O sistema garante:

- nenhum acesso fora da matriz,
- campos sempre inicializados,
- todas as áreas alocadas são liberadas.

7. Testes e Validação

Para fazermos o teste rodamos o código várias vezes e forçando entrada de dados inválida para testar o tratamento de erro do programa

1 Testes de posicionamento

- Posicionamento automático múltiplas vezes para verificar colisões.
- Posicionamento manual com validações de limites.

2 Testes de coordenadas

- Letras inválidas.

- Números fora do intervalo.
- Coordenadas repetidas.

3 Testes de jogabilidade

- Sequência completa de partida.
- Teste para garantir que a coordenada inválida não perde turno.
- Teste de afundamento de cada tipo de navio.

8. Conclusão

O que nós aprendemos:

- Aprendemos a modularizar o nosso código separando em arquivos .c e .h com um arquivo main chamando eles. Assim, consequentemente aprendemos a fazer um código mais organizado e legível.
- Uso correto de structs e ponteiros
- Aprendermos algumas boas práticas de alocação dinâmica de memória como atribuir o valor NULL a um ponteiro após dar um free nele para evitar que ele fique com algum endereço de memória nele e evite falhas no programa
- aplicação de lógica de validação
- construção de um jogo funcional com código simples e didático.

O que tivemos dificuldade:

- Fazer o tratamento de erros foi uma dificuldade. Em um momento do desenvolvimento quando o jogador digitava errado alguma célula do tabuleiro ele perdia a vez. Posteriormente, conseguimos mudar para o jogo dar mais chance para o jogador até ele digitar uma célula válida
- implementar posicionamento automático sem sobreposição,
- manter o código legível mesmo com muitos módulos.

Em uma futura versão, possíveis melhorias incluem:

- Fazer algumas funções mais curtas
- Melhorar o design dos barcos do tabuleiro e dos alvos atingidos
- aprimorar o posicionamento manual com interface visual
- criar um modo de jogo contra IA.