Trabalho Prático 1

1 Desafios Encontrados

- Duplo envio
- Comunicação entre servidor e cliente
- Função `send`
- Lógica de movimentação

1.1 Duplo envio

Sem dúvidas, o maior problema na construção deste trabalho foi um pequeno erro de código que me fez passar horas à procura de inconsistências e instabilidades que não faziam sentido ou que tinham explicações estranhas. Ao fazer o loop de resposta do servidor, utilizei a função `recv()` para captar o comando enviado pelo cliente e, em seguida, entrar em um `switch(data.type)` que verificava o recebido e tomava as devidas ações. Porém, ao final do `while`, em um lugar remoto, havia uma função `send()` além das que já estavam dentro de cada `case` do `switch`. Assim, todas as ações do servidor eram enviadas duas vezes.

Para resolver esse problema, debuguei o código e descartei cada possibilidade de erro, desde a lógica até a sintaxe.

1.2 Comunicação entre servidor e cliente

Durante a primeira parte do desenvolvimento do código, o primeiro problema encontrado foi realizar a comunicação contínua entre o servidor e o cliente, de forma que trocassem dados inúmeras vezes, dependendo apenas de o cliente enviar alguma mensagem. Além disso, implementei uma mensagem que realiza a finalização do servidor.

Para resolver essa questão, observei atentamente o funcionamento do código, tentando reunir as funções e ações em uma função que enviasse e recebesse os dados de forma independente. Dei a ela o nome de `RecebeDados()`.

Porém, embora essa função funcionasse, começou a se tornar estranho utilizá-la juntamente com o resto da lógica. Então, tomei a decisão de reescrevê-la no `main` como um loop de recebimento e envio contínuo das requisições de atualização.

```c
unsigned char * serialize_int(unsigned char *buffer, int value)
{
  /* Write big-endian int value into buffer; assumes 32-bit int and 8-bit char. */
  buffer[0] = value >> 24;
  buffer[1] = value >> 16;
  buffer[2] = value >> 8;
  buffer[3] = value;
  return buffer + 4;
}

unsigned char * serialize_char(unsigned char *buffer, char value)
{
  buffer[0] = value;
  return buffer + 1;
}

unsigned char * serialize_temp(unsigned char *buffer, struct temp *value)
{
  buffer = serialize_int(buffer, value->a);
  buffer = serialize_char(buffer, value->b);
  return buffer;
}

unsigned char * deserialize_int(unsigned char *buffer, int *value);
```

Figure 1: Funções para serializar uma struct em um buffer

1.3 Função `send`

Durante o processo de adaptação do código base para formar uma comunicação eficiente entre servidor e cliente, surgiu como única possibilidade o envio dos dados das matrizes por meio de um buffer que seria serializado pelo servidor antes de usar a função `send()` e deserializado ao chegar no cliente. Essa foi uma percepção errada, pois criou problemas desnecessários e complicou algo simples. Inclusive, tentei formas de código encontradas em fóruns, como na figura a seguir:

Somente depois de muito quebra-cabeça, percebi que, de forma incrivelmente simples, a função `send()` podia receber uma `struct` e enviá-la eficientemente.

1.4 Lógica de movimentação

A movimentação do jogador no tabuleiro exigiu um pouco de reflexão, pois esta deveria ser constantemente atualizada e saber exatamente onde o jogador se encontrava. Minhas primeiras implementações das funções que cumpriam esse objetivo apresentaram alguns erros de resultado, principalmente para respeitar todas as regras de locomoção no labirinto e limitações da matriz.

Após muitos testes, cheguei às versões finais das funções:

⟶ `Obter_Valor()` - responsável por obter o elemento de uma determinada posição do labirinto, adaptada para ser uma função simples.

⟶ `Movimentos_Possiveis()` - responsável por verificar se as posições acima, abaixo, à direita e à esquerda são caminhos livres ou não, retornando um vetor de 4 posições.

⟶ `Anda_Jogador()` - responsável por manipular a matriz do labirinto quando um movimento é solicitado.

2 Conclusão

Ao concluir este trabalho, aprendi muito sobre a importância de estar atento aos protocolos de transmissão de dados. Também pude aprimorar minhas habilidades com a linguagem C, relembrando conhecimentos aprendidos nas disciplinas PDS1 e PDS2 (Programação e Desenvolvimento de Software 1 e 2).

Outro ponto importante foi perceber como a organização e estruturação do código são essenciais, pois uma parte considerável dos problemas enfrentados decorreu de pequenos erros que poderiam ter sido evitados com um código mais claro desde o início.