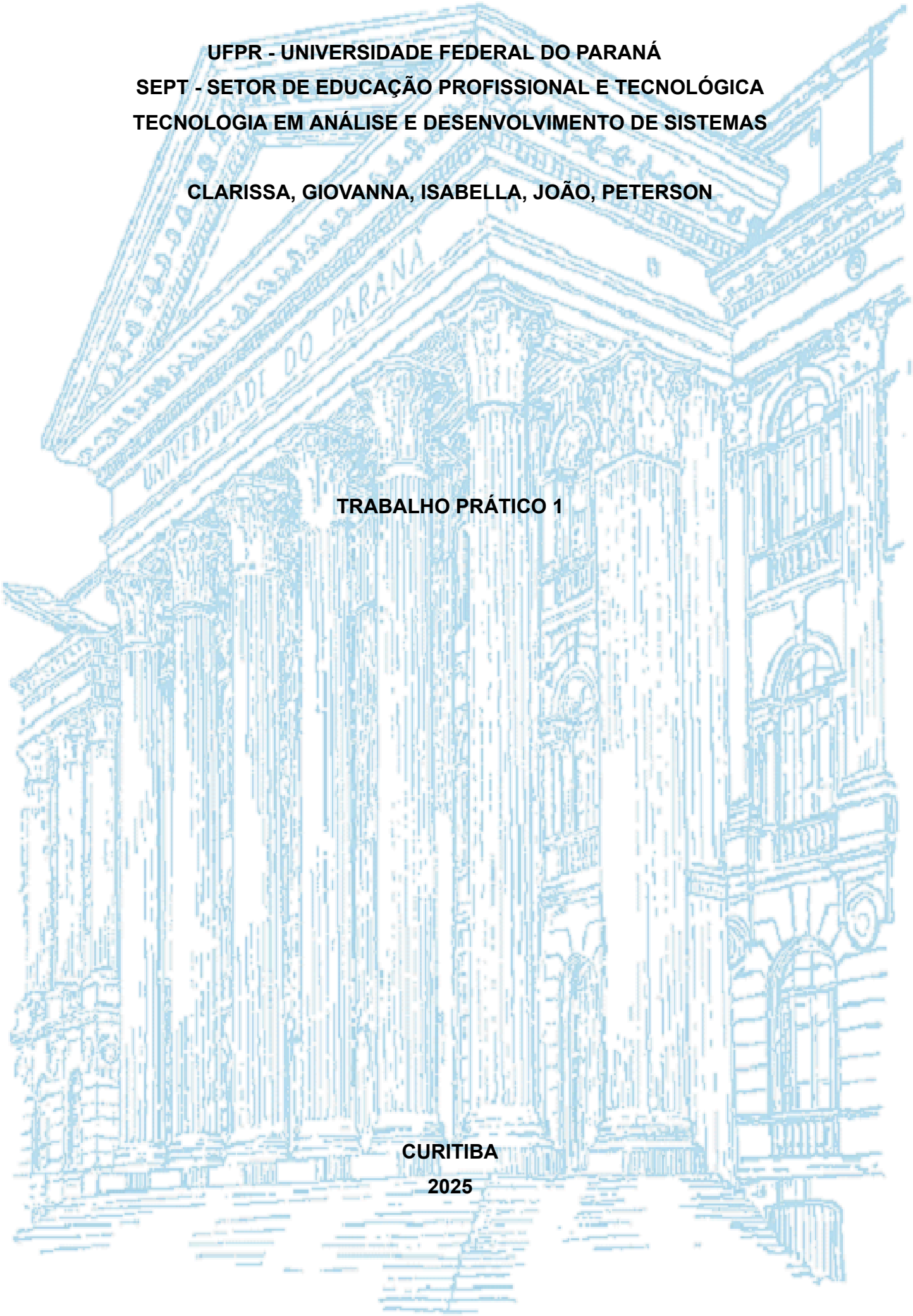
UFPR - UNIVERSIDADE FEDERAL DO PARANÁ
SEPT - SETOR DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

CLARISSA, GIOVANNA, ISABELLA, JOÃO, PETERSON



TRABALHO PRÁTICO 1

CURITIBA
2025

CLARISSA, GIOVANNA, ISABELLA, JOÃO, PETERSON

TRABALHO PRÁTICO 1

Diários de bordo do processo de desenvolvimento do Sistema de Controle de Participação em Eventos Estudantis.

Disciplina: Estrutura de Dados 1
Prof. Helcio Soares Padilha Junior

CURITIBA

2025

Diário de bordo - Clarissa Eri Morita, GRR20245515

Sistema de Controle de Participação em Eventos Estudantis

Responsabilidades/atividades realizadas:

Implementei as funções que controlam a **lista encadeada de participantes de eventos** (lista.h e lista.c), utilizando uma **lista encadeada com cabeçalho**. No arquivo lista.h desenvolvi a seguinte função:

- **apagarListaParticipantes( ):** função para apagar uma lista de um evento, libera toda a memória da lista de participantes

```
void apagarListaParticipantes(ListaParticipantes*lista);

//Declaração da função que recebe um ponteiro para uma lista de
participantes. Essa função apaga todos os elementos (nós) dessa lista
```

- **inicializarLista( ):**

```
ListaParticipantes* inicializarLista(ListaParticipantes*lista);
//inicializa a lista
```

E no arquivo lista.c fiz a implementação dessas:

```
//Procedimento para inicializar lista de participantes, feito por Clarissa
Morita
ListaParticipantes* inicializarLista(){
    ListaParticipantes* lista = (ListaParticipantes*)
malloc(sizeof(ListaParticipantes));//aloca mem para struct
ListaParticipantes
    NodeParticipante* inicio = (NodeParticipante*)
malloc(sizeof(NodeParticipante)); //aloca mem  para a struct
NodePaticipante(nó-cabeçalho)

    lista->head = inicio; //head aponta para o nó-cabeçalho
    lista->head->proximo = NULL; //o campo "proximo" do nó-cabeçalho
aponta para NULL(lista vazia)
    lista->quantidade = 0; //inicia a contagem com 0
```

```
    return lista; //retorna a lista pronta para ser usada
}
```

```
//implementação da função apagarListaParticipantes

void apagarListaParticipantes(ListaParticipantes* lista) {
    if (lista == NULL) return;   //verifica se a lista esta vazia, se sim
então retorna fazendo nada

    NodeParticipante* atual = lista->head;  //caso nao, cria um ponteiro
para começar pelo primeiro nó(head)
    NodeParticipante* proximo; // Cria outro ponteiro para armazenar
temporariamente o próximo nó antes de liberar o atual

    while (atual != NULL) {   //loop para percorrer todos os elementos
        proximo = atual->proximo; //Armazena o próximo nó antes de liberar
o atual.
        free(atual); //libera a memória para o nó atual
        atual = proximo;// vai para o próximo nó
    }

    lista->head = NULL; // quando terminar o loop(liberou a memoria de
todos os nós), a lista é sinalizade como vazia
    lista->quantidade = 0;  //lista sem nenhum elemento, atualiza o campo
quantidade
}
```

Durante o desenvolvimento a primeira dificuldade foi evitar o vazamento de memória. Como a lista é encadeada dinamicamente, era essencial percorrer corretamente todos os nós e liberá-los um por um, por isso adotei o uso de um ponteiro auxiliar (proximo) para guardar atual->proximo antes de liberar a memória do nó atual. Dessa forma, consegui liberar toda a memória da lista de forma correta e ordenada.

Outra dificuldade importante foi lidar com os ponteiros, especialmente o head da lista. Um descuido na manipulação desses ponteiros pode causar erros. Assim, foi necessário entender bem como a estrutura da lista estava organizada e tomar cuidado para alterar apenas os campos necessários após a liberação dos nós.

Também tive que me preocupar em como a função poderia ser chamada com uma lista nula (por erro de lógica em outros pontos do programa), sendo assim, incluí uma verificação no início da função para garantir que lista != NULL antes de qualquer operação.

Após a liberação da memória, foi necessário redefinir corretamente os campos da estrutura da lista. Mesmo que todos os nós tenham sido liberados, deixar o ponteiro head apontando para a memória já liberada ou manter a contagem de elementos diferente de zero poderia causar comportamento inesperado em outras funções. Por isso, após o loop de liberação, defini lista->head = NULL e lista->quantidade = 0, sinalizando que a lista está devidamente esvaziada.

Por fim, outro ponto importante foi lembrar que o sistema possui várias listas, cada uma associada a um evento. A função foi desenvolvida para apagar apenas uma dessas listas por vez, de acordo com o evento selecionado, evitando qualquer risco de apagar participantes de eventos diferentes.

Diário de Bordo – Giovanna Dornelles

Responsabilidades/Atividades Realizadas:

● Fui responsável pela criação de funções de controle relacionadas à estrutura de “ListadeParticipantes”.

○**lista_participante_h, lista_participante.c:**
● Após a criação das estruturas essenciais do sistema (GerenciadorEventos, Eventos, ListaParticipante e Participante) pude começar a criar “rascunhos” das funções que seriam implementadas no arquivo.

● Decidimos em grupo que as funções necessárias do arquivo eram:

●inicializarLista(): função que receberia código do evento e a lista de eventos e iniciaria uma lista de participantes para o evento;
●imprimirLista(): função que receberia o código do evento, lista de participantes e a lista de eventos e imprimiria todos os participantes;
●selecionarLista(): função que receberia o código do evento e a lista de eventos e retornaria a lista de participantes.
●apagarLista(): função que receberia a lista de participantes e a apagaria;

● Como eu e a Clarissa ficamos responsáveis pelo arquivo lista_participantes, decidimos que ela implementaria inicializarLista() e apagarLista() e eu implementaria imprimirLista() e selecionarLista().

●Enquanto trabalhava na função imprimirLista, percebi que:

1. Não seria necessário a implementação de selecionarLista(), já que toda vez que precisarmos da lista de participantes, teríamos que imprimi-la.
2. É possível acessar a lista de participantes de um evento com apenas um parâmetro (evento).

● A função selecionarLista foi então apagada e mudei os parâmetros da função imprimirLista().

○**Trecho de código explicado:**

```
void imprimirLista(Evento* evento){
    // Evento* evento = buscarEvento(listaEventos, codigoEvento);
    ListaParticipantes* listaEvento = evento->inscritos;

    if(evento->inscritos->head->proximo == NULL){
        printf("\nLista de inscritos vazia!\n");
        printf("\n---------------------------------\n");
        return;
```

```
    }

    NodeParticipante* participante = listaEvento->head->proximo;

    int i = 1;
    printf("\nEVENTO : %s, NUM DE PARTICIPANTES: %d\n\n\tLISTA DE
PARTICIPANTES:", evento->nome, listaEvento->quantidade);
    while(participante != NULL){
        printf("\n%d. Nome: %s, RA: %s", i,
participante->dadosParticipante.nome,
participante->dadosParticipante.ra);
        participante = participante->proximo;
        i++;
    }
    if(participante == NULL){
        printf("\n------------Fim-da-lista-------------\n");
    }
}
```

●A função imprimir Lista() recebe como parâmetro apenas uma variável do tipo Evento, esse parâmetro não vai ser fornecido pelo usuário, mas sim por outras funções que chamam a função imprimirLista();
●É criada uma variável do tipo ListaParticipantes* que vai receber a lista de participantes do evento se o topo da lista for nulo, imprime “Lista de inscritos vazia!”;
●É criada uma variável do tipo NodeParticipante que vai receber o topo da lista de participantes, e outra variável int que vai contabilizar os participantes.
●É impressa uma mensagem com dados do evento;
●Enquanto a variável participante não for nula, imprime o nome e ra do participante, participante recebe o próximo participante e o contabilizador é incrementado;
●Quando o participante recebido for nulo, chegou ao fim da lista então imprime “Fim da lista”.

Dificuldades Encontradas e Como Foram Resolvidas:
●Uma das primeiras dificuldades foi na implementação de imprimirLista(), essa função inicialmente necessitava de muitos parâmetros, isso acontecia porque as funções de outros arquivos ainda não haviam sido implementadas. Então precisei copiar o trabalho para outra pasta, excluir as funções de outros arquivos para verificar se, com apenas a estrutura, a lógica e os ponteiros da função estavam funcionando corretamente. Depois da implementação de funções essenciais como buscarEvento() e inserirParticipante(), decidimos em grupo que a entrada de usuário ficaria separada no arquivo main.c

●Outra dificuldade encontrada foi os conflitos de merge na branch feat-lista_participantes. A branch estava muitos commits pra trás da main, e arquivos haviam sido excluídos e renomeados, isso causou um conflito que não era possível resolver no merge editor, então foi necessário usar comandos para realizar um merge “forçado” (que não é recomendável), para favorecer as mudanças na main e desfavorecer os arquivos da branch.

Diário de Bordo – Isabella Vicente

Função removerParticipanteDeEvento

O objetivo principal da função removerParticipanteDeEvento é localizar um participante pelo seu RA (Registro Acadêmico) e removê-lo da lista de participantes de um evento.

Separei em passo a passo:

1. Declaração das varáveis que irei usar

2. Usei buscarParticipante( ) dentro da função remover, pegando o retorno do nó para comparar e depois remover

3. Comparação do que eu quero procurar com cada elemento da lista: como queria procurar um RA específico para remover, coloquei a condição que continuasse procurando enquanto a lista não chegasse ao final e enquanto não achasse o RA que eu quero.

4. Atribuição de atual para anterior para anterior armazenar a informação de atual, e de atual para atual->proximo para atual armazenar a proxima informação, andando na lista.

5. Verificar se atual chegou ao final da lista, caso tenha chegado no final da lista, ele para de procurar, pois nao achou o RA.

6. anterior->proximo=atual->proximo serve para atual passar para anterior o proximo valor sem que atual seja liberado e a informação do proximo se perca.

Função buscarParticipante

Ela é bem parecida com a remover, mas pode ser usada em outras funções para evitar duplicação de dados. Essa função percorre a lista linearmente, comparando cada RA com o procurado. Se encontrar, retorna o ponteiro para o nó correspondente; caso contrário, retorna NULL.

Ela evita duplicação de código e separa a responsabilidade da busca da responsabilidade de remoção.

Como a lista foi usada

Foi utilizada uma lista encadeada com nó cabeçalho, o que traz algumas vantagens:

- Facilidade nas operações de remoção e inserção, especialmente no início da lista.
- O uso do head como nó fictício evita verificações extras para remover o primeiro elemento real da lista.
- Cada nó (NodeParticipante) aponta para o próximo, permitindo percorrer sequencialmente a lista.

Por que essa estrutura foi adequada

A lista encadeada com nó cabeçalho foi a melhor escolha para esse cenário porque:

- O número de participantes pode variar dinamicamente.
- Não há necessidade de alocar espaço fixo como em um vetor.
- A remoção é eficiente (O(n)), e o código é mais limpo com o uso do nó cabeçalho.
- Evita tratamentos especiais para a remoção do primeiro nó real, pois sempre há um nó anterior (head).

Dificuldades encontradas

1. Entender quais funções teriam que ser escritas e delegadas.
2. Manipulação de ponteiros: No início, entender o uso correto de anterior e atual foi um desafio, principalmente como manter o controle dos nós durante a remoção.
3. Evitar perder nós: Houve cuidado para garantir que, ao remover um nó, o restante da lista permanecesse corretamente encadeado.
4. Uso da função de busca separada: Garantir que a lógica de busca e remoção estivesse bem desacoplada exigiu organização mental e clareza nos retornos da função buscarParticipante.

Diário de Bordo - João Victor -> Semana: *25/05/2025 a 06/06/2025*

Responsabilidades e Atividades Desempenhadas:

Fui responsável pela concepção e implementação das funcionalidades centrais de gerenciamento de participantes, criando a estrutura inicial e a lógica nos arquivos participantes.c e participantes.h.

Desenvolvi a função inscreverParticipanteEmEvento, que permite adicionar um participante a um evento específico, incluindo o tratamento de erros para evitar a duplicação de inscrições (mesmo RA no mesmo evento).

Implementei a função emitirRelatorioIndividual, projetada para buscar e listar todos os eventos nos quais um único participante, identificado pelo seu RA, está inscrito.

Integrei as funcionalidades de inserirParticipante e emitirRelatorioIndividual ao menu principal de interação com o usuário no arquivo main.c, permitindo que fossem acessadas através das opções do sistema.

Colaborei ativamente na refatoração da arquitetura do projeto, auxiliando na migração da lógica de interação com o usuário para o novo módulo controller, o que ajudou a melhorar a organização e a manutenção do código.

Participei do processo de depuração (debugging), ajudando a identificar e a solucionar erros lógicos que surgiam na interação entre as diferentes partes do sistema, especialmente na manipulação de dados do usuário.

Trechos de Código Explicados:

1. Função `inscreverParticipanteEmEvento` (em `participantes.c`):

```
bool inscreverParticipanteEmEvento(const char* nome, const char* ra, Evento*
evento) {
    // Verificando se o participante já está inscrito no evento
    NodeParticipante* atual = evento->inscritos->head->proximo;
    while (atual != NULL) {
        if (strcmp(atual->dadosParticipante.ra, ra) == 0) {
            printf("Participante com RA '%s' já está inscrito no evento
'%s'.\n", ra, evento->nome);
            return false;
        }
        atual = atual->proximo;
    }

    // Criando novo nó para o participante
    NodeParticipante* novoParticipante =
```

```
(NodeParticipante*)malloc(sizeof(NodeParticipante));

    // Preenche os dados do participante
    strncpy(novoParticipante->dadosParticipante.nome, nome,
sizeof(novoParticipante->dadosParticipante.nome) - 1);

novoParticipante->dadosParticipante.nome[sizeof(novoParticipante->dadosParticipa
nte.nome) - 1] = '\0';

    // [...] (cópia do RA)

    // Inserindo o participante na lista de inscritos do evento
    NodeParticipante* ultimo = evento->inscritos->head;
    while (ultimo->proximo != NULL) { // Encontra o último nó real
        ultimo = ultimo->proximo;
    }
    ultimo->proximo = novoParticipante; // adiciona o novo participante

    evento->inscritos->quantidade++;
    return true; // Inscrição de participante foi bem-sucedida
}
```

Explicação:

Para implementar a inscrição, primeiro a função percorre a lista de participantes do evento para verificar se o RA informado já existe, prevenindo que um mesmo aluno se inscreva duas vezes. Se o RA não for encontrado, um novo NodeParticipante é alocado na memória. Utilizei strncpy para copiar o nome e o RA para a estrutura do novo participante, garantindo que não ocorram *buffer overflows* caso a entrada seja maior que o esperado. Por fim, a função avança até o final da lista de inscritos do evento e anexa o novo participante, incrementando o contador de inscritos do evento. Essa abordagem garante tanto a integridade dos dados quanto a segurança da memória.

2. Função `emitirRelatorioIndividual` (em `participantes.c`):

```
void emitirRelatorioIndividual(const char* RA, GerenciadorEventos* ge) {

    printf("\nRelatório Individual para RA %s: \n", RA);

    NodeEvento* eventoAtualNode = ge->head->proximo;
    bool participanteEncontrado = false;

    while (eventoAtualNode != NULL) {
        if (eventoAtualNode->evento != NULL) {
            NodeParticipante* participanteNode =
eventoAtualNode->evento->inscritos->head->proximo;
```

```
            while (participanteNode != NULL) {
                if (strcmp(participanteNode->dadosParticipante.ra, RA) == 0) {
                    printf(" - Evento: %s (Código: %d)\n",
eventoAtualNode->evento->nome, eventoAtualNode->evento->codigo);
                    participanteEncontrado = true;
                }
                participanteNode = participanteNode->proximo;
            }
        }
        eventoAtualNode = eventoAtualNode->proximo;
    }

    if (!participanteEncontrado) {
        printf("Participante de RA '%s' não encontrado em nenhum evento.\n",
RA);
    }
    //...
}
```

Explicação:

Esta função foi criada para gerar um relatório de todos os eventos em que um aluno está inscrito. A lógica consiste em dois loops aninhados: o primeiro percorre a lista principal de todos os eventos (GerenciadorEventos) e, para cada evento, o segundo loop percorre a lista de participantes daquele evento específico. Dentro do segundo loop, comparo o RA de cada participante com o RA fornecido para a busca. Se houver uma correspondência, o nome e o código do evento são impressos na tela. Uma variável booleana, participanteEncontrado, é usada para rastrear se encontramos pelo menos uma inscrição. Caso, ao final de toda a busca, essa variável continue como false, o sistema informa ao usuário que o participante não foi encontrado em nenhum evento.

Dificuldades Encontradas e Como Foram Resolvidas:

Problemas de Compilação com Múltiplos Módulos.

- **Desafio:** No início da integração, ao juntar as diferentes partes do projeto desenvolvidas pela equipe, enfrentei diversos erros de compilação. Os mais comuns eram erros de "tipo incompleto" ou "struct undefined", pois cada módulo (eventos.c, participantes.c) dependia de estruturas definidas em outros arquivos de cabeçalho (.h), gerando confusão e dependências circulares.

- **Solução:** A solução foi alcançada em conjunto com a equipe através de uma grande reestruturação. Criamos um arquivo central, estruturas.h, que passou a conter a definição de TODAS as structs do sistema (como Evento, Participante, NodeEvento, etc.). Com isso, os demais arquivos (.h) passaram

a incluir apenas o estruturas.h e os protótipos de suas próprias funções. Adicionalmente, a criação do módulo controller.c ajudou a separar a lógica de interação com o usuário das regras de negócio, o que simplificou as inclusões e eliminou os erros de "multiple definition".

Dificuldade: Manipulação de Ponteiros e Lógica de Listas Encadeadas.

- **Desafio:** Entender e aplicar corretamente a lógica de ponteiros para percorrer e manipular as listas encadeadas foi um desafio. Em especial, acertar a forma como a função emitirRelatorioIndividual deveria navegar na lista de eventos e, para cada evento, acessar a sua respectiva lista de participantes (evento->inscritos->head) demandou tempo e depuração.

- **Solução:** A resolução veio através da colaboração e comunicação com a equipe. Conversei bastante com o Peterson para entender a lógica que ele havia pensado para a estrutura geral dos dados, principalmente como cada Evento conteria sua própria ListaParticipantes. Essa troca de conhecimento foi fundamental para que eu pudesse desenvolver minhas funções de forma compatível com o resto do sistema. Revisar o código em conjunto e desenhar a lógica dos ponteiros no papel ajudou a visualizar o fluxo de dados e a corrigir os erros.

Diário de Bordo – Peterson Fontinhas

Semana: [25/05/2025 a 01/06/2025]

Responsabilidades/Atividades Realizadas:

- Fui responsável pela concepção e criação da estrutura inicial de arquivos e diretórios do projeto, visando uma organização modular conforme as especificações **(`main.c`, `eventos.c/.h`, `participantes.c/.h`, `lista.c/.h`**).
- Defini as estruturas de dados (**`structs`**) centrais que servirão de base para todo o "Sistema de Controle de Participação em Eventos Estudantis". Esta etapa envolveu a criação dos seguintes arquivos de cabeçalho e suas respectivas **`structs`**, ainda sem a implementação das funções:
    - **`participantes.h`**:
        - **`Participante`**: Estrutura para armazenar os dados de cada participante (RA e nome).
    - **`lista.h`** (para a lista de participantes de um evento):
        - **`NodeParticipante`**: Nó da lista encadeada que armazena um **`Participante`** e o ponteiro para o próximo nó.
        - **`ListaParticipantes`**: Estrutura de controle (cabeçalho) para a lista de participantes de um evento específico, contendo o ponteiro **`head`** e a **`quantidade`** de inscritos.
    - **`eventos.h`** (para o gerenciamento de eventos):
        - **`DataEvento`**: Estrutura auxiliar para armazenar data e hora de forma organizada.
        - **`Evento`**: Estrutura principal para os dados de um evento (código, nome, data, local) e, crucialmente, um campo **`ListaParticipantes inscritos`** para conectar cada evento à sua própria lista de participantes.
        - **`NodeEvento`**: Nó da lista encadeada que armazenará um ponteiro para um `Evento` e o ponteiro para o próximo evento na lista geral.
        - **`GerenciadorEventos`**: Estrutura de controle (cabeçalho) para a lista principal de todos os eventos do sistema, contendo o **`head`** para esta lista e a **`quantidadeEventos`**.
- O objetivo desta fase foi estabelecer um alicerce robusto e bem definido para as estruturas de dados, facilitando a subsequente implementação das funcionalidades e a integração do trabalho em equipe.

Trechos de Código Explicados:

(Em `eventos.h`) A `struct Evento`:

```
typedef struct {
   int codigo;
   char nome[100];
   DataEvento dataEvento;
   char localEvento[100];
   ListaParticipantes inscritos; // cada evento tem uma ListaParticipantes
} Evento;
```

- *Explicação:* "Esta **`struct`** é o coração do sistema no que tange a um evento individual. Ela agrupa todas as informações pertinentes a um evento, como seu **`codigo`** identificador, **`nome`**, **`dataEvento`** (que é outra **`struct`** para melhor organização da data e hora), e **`localEvento`**. O campo mais importante para a dinâmica do sistema é **`inscritos`**, do tipo **`ListaParticipantes`** (definida em **`lista.h`**). Isso significa que cada instância de **`Evento`** carregará consigo a cabeça da lista de todos os participantes inscritos especificamente nele, permitindo o gerenciamento individualizado dos inscritos por evento."

(Em `eventos.h`) A `struct GerenciadorEventos`:

```
typedef struct{
   NodeEvento* head; // NodeEvento aponta para um Evento
   int quantidadeEventos;
} GerenciadorEventos;
```

- *Explicação:* "Para gerenciar todos os diversos eventos que o sistema pode ter, criei a **`GerenciadorEventos`**. Ela atua como um cabeçalho para a lista mestra de eventos. O ponteiro **`head`** aponta para o primeiro **`NodeEvento`** (nó da lista de eventos), e **`quantidadeEventos`** rastreia o número total de eventos cadastrados. Esta estrutura será fundamental para funções como cadastrar um novo evento no sistema ou buscar por um evento existente."

Dificuldades Encontradas e Como Foram Resolvidas:

- "Uma consideração inicial foi como melhor representar a relação entre um evento e sua lista de participantes. Decidi por incluir a **`struct ListaParticipantes`** diretamente dentro da **`struct Evento`** (em vez de apenas um ponteiro para ela que precisaria ser alocado separadamente) para simplificar a gestão da memória de cada evento e garantir que toda **`Evento`** já 'nasça' com sua estrutura de lista de participantes pronta para ser inicializada."

Próximos Passos (Planejados para a próxima semana/período):

- Concentrar na implementação das funções de manipulação em **`eventos.c`**, começando pelas funcionalidades de **`inicializarGerenciadorEventos`**, **`cadastrarNovoEvento`** (incluindo a correta inicialização da **`ListaParticipantes`** interna do novo evento) e **`buscarEventoPorCodigo`**.

/----------/—---------/—---------/—---------/—---------/—---------/

Semana: [01/06/2025 a 06/06/2025]

Responsabilidades/Atividades realizadas:

- Fui responsável pela criação das implementações relacionadas à feature de Eventos, sendo elas
  - Cadastrar um evento
  - Buscar evento por código
  - Cancelar evento
  - Destruir toda a lista de eventos
- Criei também funções auxiliares, importantes para controle e veracidade dos dados, sendo elas:
  - Uma função para verificar se o evento é válido
    - Analisa a data e local do evento, e se já houver um evento cadastrado nesse local e nessa data, o evento não pode ser cadastrado
  - Uma função para validar a data
    - Verifica se a data condiz com o calendário gregoriano
  - Uma função para verificar se o ano é bissexto
    - Verifica se o ano é bissexto, para adicionar um dia a mais a fevereiro, podendo assim ter eventos cadastrados nos dias 29 de fevereiro
- Foquei na refatoração da estrutura do projeto para melhorar a modularidade, movendo toda a lógica de interação com o usuário (funções de menu como **`criarEvento`**, **`inserirParticipante`**, etc.) do **`main.`**c para um novo módulo **`controller`**.
- Implementei e refinei as funções no **`controller.c`**, incluindo a criação de uma função robusta e reutilizável (**`lerInteiroValidado`**) para tratar a entrada numérica do usuário.
- Trabalhei na criação do **`Makefile`** para facilitar a compilação do projeto, que envolvia muitos passos, agora sendo executados com apenas um comando **make** , e organizar os arquivos de compilação (**`.o`**) em um diretório separado (**`obj/`**), mantendo o projeto mais limpo.

- Ajudei na depuração e correção de bugs lógicos nas funções de manipulação de listas, como a de impressão e cancelamento de eventos.

Trechos de código explicados:

1. Atribuição de uma string para uma variável

```
    // ATRIBUINDO O NOME AO EVENTO
    strncpy(evento->nome, nome, sizeof(evento->nome) - 1);
    evento->nome[sizeof(evento->nome) - 1] = '\0'; // Garante
terminação nula
```

Explicação**:* "Para popular os campos de texto do evento, como **`evento->nome`** e **`evento->localEvento`,** optei por utilizar a função **`strncpy`** em vez de **`strcpy`**. A principal motivação para essa escolha foi a **segurança contra *buffer overflows. O **`strncpy`** permite especificar o número máximo de caracteres a serem copiados (**`sizeof(array) - 1`**), o que previne que uma string de entrada muito longa sobrescreva áreas de memória adjacentes. A linha **`evento->nome[sizeof(evento->nome) - 1] = '\0';`** (e a análoga para **`localEvento`**) foi adicionada como uma **medida de robustez adicional**. Ela assegura que a string no campo da **`struct`** seja **sempre terminada com um caractere nulo (`\0`)**, mesmo que a string original fornecida seja exatamente do tamanho do buffer ou maior. Isso é crucial porque **`strncpy`** não garante a terminação nula nesse cenário específico (quando o tamanho da string de origem é maior ou igual ao limite n e o **`\0`** não é copiado). Garantir a terminação nula é essencial para o correto funcionamento e segurança de subsequentes operações com essas strings utilizando funções padrão da biblioteca C."

2. Função de Leitura e Validação de Inteiros (em `controller.c`):

```
int lerInteiroValidado(const char* prompt) {
    char buffer[20]; // Um buffer para ler a entrada como
string, 20 é suficiente para um int
    bool entradaValida = false;

    do {
        printf("%s", prompt); // Mostra a mensagem para o
usuário
```

```
        // Lê a entrada do usuário de forma segura
        if (fgets(buffer, sizeof(buffer), stdin) == NULL) {
            printf("Erro ao ler a entrada. Saindo.\n");
            return -1; // Retorna um valor de erro
        }

        // Se a entrada foi muito longa, o '\n' não estará no
buffer.
        // Neste caso, precisamos limpar o resto da linha do
buffer de entrada.
        if (strchr(buffer, '\n') == NULL) {
            int c;
            while ((c = getchar()) != '\n' && c != EOF);
        }

        // Remove o caractere de nova linha '\n' do buffer, se
ele existir
        buffer[strcspn(buffer, "\n")] = 0;

        // Usa a nossa função auxiliar para validar se o
conteúdo é numérico e não vazio
        entradaValida = validarInputNumerico(buffer);

        if (!entradaValida) {
            printf("Entrada inválida. Por favor, digite apenas
números.\n");
        }

    } while (!entradaValida);

    // Converte a string validada para inteiro e a retorna
    return atoi(buffer);
}
```

- *Explicação:* "Decidi criar esta função para centralizar e resolver os problemas de leitura de números do usuário. Ela usa **`fgets`** para ler a entrada como

texto, o que é mais seguro que **`scanf`**. A parte chave é a verificação **`if (strchr(buffer, '\n') == NULL)`**, que detecta se o usuário digitou mais caracteres do que o buffer suporta. Se isso acontece, um loop **`while`** com **`getchar()`** é usado para consumir e descartar o resto da entrada, limpando o buffer e evitando bugs nas leituras seguintes. Após garantir que o buffer está limpo e que a string contém apenas dígitos (usando a função auxiliar **`validarInputNumerico`**), ela é convertida para inteiro com **`atoi`**. Essa abordagem tornou o código principal muito mais robusto e legível."

Dificuldades encontradas e como foram resolvidas:

1. **Dificuldade: Caos de Inclusões e "Multiple Definition"**
    - **Problema:** Ao tentar compilar o projeto com os módulos separados (**`eventos`**, **`participantes`**, etc.), enfrentei uma grande quantidade de erros de "struct não definida", "tipo incompleto" e, finalmente, "multiple definition". A tentativa de usar declarações antecipadas (**`forward declarations`**) estava se mostrando confusa e gerando novos erros.
    - **Solução:** Após análise, percebemos que o problema principal era uma mistura de dependências circulares nos arquivos **`.h`** e a duplicação de definições de funções (que estavam tanto em **`main.c`** quanto no novo **`controller.c`**). A solução adotada foi uma reestruturação em duas frentes:
        - **Código:** Foi criado o módulo **`controller`** para centralizar as funções de interface com o usuário, e o **`main.c`** foi limpo para conter apenas o loop principal e as chamadas para o **`controller`**. Isso resolveu os erros de "multiple definition".
        - **Headers:** Para resolver os problemas de tipo, adotei a estratégia de criar um arquivo central **`estruturas.h`** que contém a definição de TODAS as **`structs`** do projeto. Os outros headers (**`eventos.h`**, **`participantes.h`**, etc.) agora incluem **`estruturas.h`** e contêm apenas os protótipos de suas funções, eliminando a teia de includes confusos e os erros de compilação relacionados.
2. **Dificuldade: Validação de Input Frágil e Bugs de Buffer**
    - **Problema:** A leitura de dados numéricos do usuário (como o código de um evento) estava causando bugs. Se o usuário digitasse uma entrada

muito longa (ex: "1234" para um buffer de 4 caracteres) ou texto em vez de números, o **`\n`** (Enter) ou outros caracteres ficavam "presos" no buffer de entrada (**`stdin`**), fazendo com que as leituras seguintes (**`fgets`**) fossem puladas ou se comportassem de maneira inesperada.
    - **Solução:** Implementei uma função auxiliar robusta, **`lerInteiroValidado(const char* prompt)`**. Essa função encapsula toda a lógica complexa: lê a entrada como uma string, verifica se a entrada não foi longa demais (e limpa o buffer se foi), garante que a string contém apenas dígitos e não está vazia, e só então a converte para um inteiro. Isso tornou o resto do código (como em **`criarEvento`**) muito mais limpo e seguro.
3. **Dificuldade: Lógica Incorreta em Funções de Lista Encadeada**

    - **Problema:** Mesmo após cadastrar um participante, a função de imprimir a lista não o exibia. Na função **`cancelarEvento`**, a lógica de travessia para encontrar o nó anterior ao que seria deletado estava incorreta.
    - **Solução:** Revisei a lógica dos loops. O problema na função de impressão estava na condição **`while (atual->proximo != NULL)`**, que fazia o loop parar um elemento antes do fim. A correção foi mudar para **`while (atual != NULL)`**. Na função de cancelamento, ajustei a lógica para **`anterior = atual; atual = atual->proximo;`**, garantindo que o ponteiro para o nó anterior fosse corretamente atualizado a cada passo.

Próximos passos:

- Criar funções para fins estatísticos ou demonstrativos
- Refinar mais as funções para deixar mais intuitivo para o usuário