



UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO - UFERSA  
DEPARTAMENTO DE ENGENHARIAS E TECNOLOGIA  
DOCENTE: GEORGE FELIPE FERNANDES VIEIRA  
DISCIPLINA: ALGORITMOS E ESTRUTURAS DE DADOS I

IZADORA LOUYZA SILVA FIGUEIREDO  
LÍVIAN MARIA LUCENA GOMES PINHEIRO  
MARIA VITORIA FERNANDES ROCHA  
RENATO VITOR JUVÊNCIO LEITE  
VICTOR HUGO DE OLIVEIRA

RELATÓRIO DE USO DO SISTEMA DE GERENCIAMENTO DE PEDIDOS DE  
RESTAURANTE

PAU DOS FERROS/RN  
MARÇO 2025

## 1. Descrição da Arquitetura do Projeto

O Sistema de Gerenciamento de Pedidos para Restaurante foi desenvolvido em linguagem C, utilizando uma arquitetura modular que organiza o código em funções focadas em cada operação e que realizam ações específicas. As principais funcionalidades implementadas no sistema incluem o manuseio do cardápio (com funções de cadastro, exibição, atualização e remoção de itens) e dos pedidos (criação, gerenciamento, alteração de status e alteração e remoção dos pedidos).

A estrutura do programa também é baseada em recursos da linguagem C, como ponteiros, alocação dinâmica de memória, vetores dinâmicos, registros (structs) e enumerações, o que garante que o sistema consuma memória conforme a necessidade e otimize os seus recursos. A organização dos dados é feita de forma estruturada, permitindo o armazenamento eficiente das informações dos itens do cardápio e dos pedidos. A arquitetura modular e esses recursos utilizados permitem a fácil manutenção e expansão do sistema, bem como a sua eficiência.

## 2. Estruturas Utilizadas

### 2.1. Ponteiros

Os ponteiros são variáveis que armazenam o endereço de memória de outras variáveis. Dizemos que um ponteiro “aponta” para uma variável quando contém o endereço da mesma. Eles são muito úteis quando uma variável precisa ser acessada em qualquer parte do programa, proporcionando maior flexibilidade para quem os manuseia.

### 2.2. Strings com Ponteiros

Ponteiros podem “apontar” para qualquer tipo de variável e *string* é uma delas. No sistema, *strings* são manipuladas utilizando ponteiros, especialmente nas funções que recebem e retornam *strings*. O uso de *fgets* para ler *strings* e *strcspn* para remover quebras de linha são exemplos de operações básicas com *strings* e estão presentes no código.

### 2.3. Alocação Dinâmica de Memória

A alocação dinâmica de memória nos permite alocar espaços na memória do computador durante a execução de um programa e é de grande

utilidade pois o espaço é alocado apenas quando necessário. Além disso, a alocação de memória é bastante flexível, permitindo aumentar ou diminuir a quantidade de memória alocada.

O sistema utiliza alocação dinâmica de memória para gerenciar o armazenamento de itens do cardápio e pedidos. As funções *malloc* e *realloc* são empregadas para alocar e redimensionar a memória conforme a necessidade.

## **2.4. Vetores Dinâmicos**

Um vetor dinâmico é uma estrutura de dados que possui um espaço contínuo na memória, no entanto, quando a capacidade chega em seu limite, o vetor é realocado para um tamanho maior que o anterior em tempo de execução, ou seja, quando o código está sendo executado. No sistema, vetores dinâmicos são utilizados para armazenar listas de itens do cardápio e pedidos, utilizando funções para redimensionar a memória e reorganizar os dados.

## **2.5. Structs**

As *structs* são variáveis especiais que contém outras variáveis de tipos iguais ou diferentes. Podemos associar as *structs* da linguagem C aos denominados registros em outras linguagens de programação. No sistema de gerenciamento de pedidos para restaurante, as *structs* são definidas para representar os itens do cardápio e os pedidos, facilitando a manipulação dos dados. Cada *struct* contém campos que armazenam informações relevantes, como nome, descrição, preço e status.

## **2.6. Enumerações**

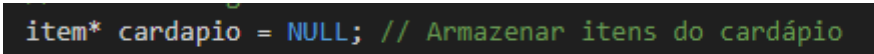
Enum em C é um tipo de dado definido previamente pelo usuário que recebe apenas um conjunto restrito de valores. Ele pode ser usado para definir um conjunto de nomes simbólicos que representam valores inteiros. Essas enumerações são utilizadas no sistema para definir categorias de

pratos e status dos pedidos, garantindo maior clareza e organização do código.

### 3. Aplicação dos Conceitos Obrigatórios

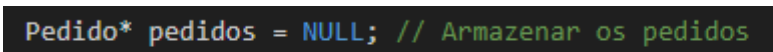
#### 3.1. Ponteiros

No sistema, os ponteiros são amplamente utilizados para gerenciar a memória e manipular dados de maneira otimizada. Por exemplo, eles são usados para gerenciar o cardápio e os pedidos, armazenando e manipulando listas de itens cadastrados. Isso permite que o programa redimensione dinamicamente a memória conforme necessário.



```
item* cardapio = NULL; // Armazenar itens do cardápio
```

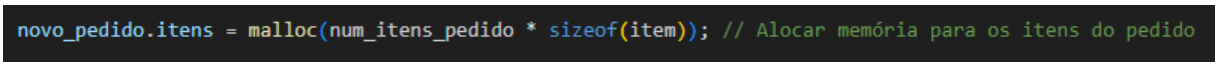
Imagem 1 - Declaração do ponteiro item\* cardapio.



```
Pedido* pedidos = NULL; // Armazenar os pedidos
```

Imagem 2 - Declaração do ponteiro Pedido\* pedidos.

Por exemplo, na imagem 3, a função *malloc* para alocar dinamicamente o número de itens no cardápio, passa como parâmetro um ponteiro item do tipo item\*, que foi declarado dentro da *struct* Pedido (imagem 13).



```
novo_pedido.itens = malloc(num_itens_pedido * sizeof(item)); // Alocar memória para os itens do pedido
```

Imagem 3 - Passagem ponteiro como parâmetro.

#### 3.2. Strings com Ponteiros

*Strings* com ponteiros estão presentes, por exemplo, na função *cadastro\_card*. A função *fgets* foi usada para realizar a leitura do nome do item a ser cadastrado no cardápio. Já a função *strcspn* foi usada, pois após a leitura de uma *string* com *fgets*, nesse caso o nome do item, pode ocorrer que um caractere de nova linha (\n) seja armazenado no final da string caso o usuário tenha pressionado enter, e, para resolver isso, o *strcspn* substitui o caractere de nova linha por um caractere nulo (\0) evitando a quebra de linha (imagem 4).

```
// Função para cadastrar itens no cardápio
void cadastro_card(item *p) { // Cadastrar e atualizar itens do cardápio
    getchar(); // Limpar o buffer
    printf("Nome do item: "); // solicita o nome do item
    fgets(p->nome, sizeof(p->nome), stdin); // recebe o nome do item
    p->nome[strcspn(p->nome, "\n")] = 0; // Remover a quebra de linha

    printf("Descrição [máx.: 100 caracteres]: "); // solicita a descrição do item
    fgets(p->descri, sizeof(p->descri), stdin); // recebe a descrição do item
    p->descri[strcspn(p->descri, "\n")] = 0; // Remover a quebra de linha
}
```

Imagem 4 - Leitura de strings com ponteiros e quebra de linha..

### 3.3. Alocação de Memória

No sistema, a função *malloc* é utilizada, por exemplo, para alocar memória dinamicamente para os itens de um pedido (imagem 5). Quando o usuário decide criar um novo pedido e especifica o número de itens que ele quer no pedido, a memória precisa ser alocada para armazenar esses itens.

```
// Alocar memória inicial para os pedidos
pedidos = malloc(capacidade_pedidos * sizeof(Pedido));
if (pedidos == NULL) {
    printf(">> Erro ao alocar memória para os pedidos.\n");
    free(cardapio); // Liberar memória do cardápio antes de sair
    return 1; // Sair do programa em caso de erro
}
```

Imagem 5 - Alocação de memória para pedidos usando o malloc.

A função *sizeof* determina o número de *bytes* para um determinado tipo de dados. No código, ela retorna o número de *bytes* necessários para armazenar o *array* *nome\_cliente* (imagem 6). Neste caso, *nome\_cliente* é um *array* de 100 caracteres então *sizeof(p->nome\_cliente)* retorna 100 *bytes*. Isso é necessário, porque queremos garantir que a função *fgets* não leia mais do que o tamanho alocado para o *array*, evitando estouros de memória.

```
// Função para criar um novo pedido
void criar_pedido_menu() {
    if (codigo > 0) { // verificar se há itens a oferecer ao cliente
        char nome_cliente[100];
        printf("Nome do cliente: "); // solicita o nome do cliente
        getchar(); // Limpar o buffer
        fgets(nome_cliente, sizeof(nome_cliente), stdin); // recebe o nome do cliente
        nome_cliente[strcspn(nome_cliente, "\n")] = 0; // Remover a quebra de linha
    }
}
```

Imagem 6 - Determinação do tamanho do nome do cliente com o sizeof.

Já a função *realloc* é usada para redimensionar um espaço alocado previamente com *malloc* e o novo tamanho redimensionado pode ser maior ou menor que o tamanho inicialmente alocado. No sistema, o *realloc* é usado para redimensionar a capacidade de pedidos no sistema, quando essa capacidade é alcançada, o *realloc* é chamado para dobrar o espaço de memória (imagem 7). O código original possui uma capacidade inicial de 2 pedidos (imagem 1), e quando mais pedidos são adicionados, a capacidade de memória precisa ser aumentada.

```
Pedido* temp = realloc(pedidos, capacidade_pedidos * sizeof(Pedido)); // Tenta realocar a memória
if (temp == NULL) {
    printf(">> Erro ao redimensionar os pedidos.\n");
    exit(1); // Sair do programa em caso de erro
}
pedidos = temp; // Atualiza o ponteiro dos pedidos
}
```

Imagem 7 - Realocação de memória para pedidos usando o realloc.

E ao final do código, dentro do *main*, temos a função *free*, que libera o espaço de memória alocado, para que ela seja reutilizada, evitando vazamento de memória (imagem 8).

```
free(cardapio); // libera a variável dinâmica cardapio para evitar vazamento de memória
free(pedidos); // libera a variável dinâmica pedidos para evitar vazamento de memória
```

Imagem 8 - Liberação do espaço da memória alocada com o free.

### 3.4. Vetores Dinâmicos

Os vetores dinâmicos são utilizados no sistema para armazenar os itens cadastrados no cardápio, por exemplo. Assim como o número de pedidos, o início do programa, a capacidade inicial para o cardápio é definida para 2 itens (imagem 9). A alocação de memória para o vetor cardápio é feita com *malloc* (imagem 10).

```
// Variáveis globais
item* cardapio = NULL; // Armazenar itens do cardápio
int codigo = 0; // Contador de itens no cardápio
int capacidade_cardapio = 2; // Capacidade inicial do cardápio

Pedido* pedidos = NULL; // Armazenar os pedidos
int num_pedidos = 0; // Contador de pedidos
int capacidade_pedidos = 2; // Capacidade inicial para pedidos
```

Imagem 9 - Definição a capacidade inicial de itens no cardápio e de pedidos.

```
// Alocar memória inicial para o cardápio
cardapio = malloc(capacidade_cardapio * sizeof(item));
if (cardapio == NULL) { // verifica se foi alocado na memória corretamente
    printf(">> Erro ao alocar memória para o cardápio.\n");
    return 1; // Sair do programa em caso de erro
}
```

Imagem 10 - Alocação de memória para itens do cardápio usando o malloc.

Como são dinâmicos, ou seja, podem se expandir em tempo de execução, usamos a função *realloc* para redimensionar o vetor à medida que o número de itens no cardápio aumenta para aumentar seu tamanho e para que ele possa armazenar mais itens (imagem 11).

```
// Função para redimensionar o cardápio
void redimensionar_cardapio() {
    capacidade_cardapio *= 2; // Dobra a capacidade do cardápio
    item* temp = realloc(cardapio, capacidade_cardapio * sizeof(item)); // Tenta realocar a memória
    if (temp == NULL) {
        printf(">> Erro ao redimensionar o cardápio.\n");
        exit(1); // Sair do programa em caso de erro
    }
    cardapio = temp; // Atualiza o ponteiro do cardápio
}
```

Imagem 11 - Realocação de memória para itens do cardápio usando o realloc.

### 3.5. Structs

No código, as *structs* são usadas para representar os itens individualmente no cardápio e os pedidos feitos pelos clientes, facilitando a organização dos dados de cada item do cardápio e de cada pedido realizado.

Cada item tem um nome, uma descrição, um preço e uma categoria (imagem 12) e cada pedido possui um código de pedido, o nome do cliente,

os itens do pedido, a quantidade de itens do pedido e o status do pedido (imagem 13), todos esses atributos são chamados de membros da *struct*.

```
// Struct para representar um item do cardápio
typedef struct {
    char nome[100]; // nome do item
    char descri[100]; // descrição do item
    float preco; // preço em R$ do item
    categoria catego; // Categoria do item
} item;
```

Imagem 12 - Representação em structs dos itens do cardápio.

```
// Struct para representar um pedido
typedef struct {
    int cod_pedido; // Identificador do pedido
    char nome_cliente[100]; // Nome do cliente
    item* itens; // Armazenar os itens do pedido
    int num_itens; // Número de itens no pedido
    StatusPedido status; // Status do pedido
} Pedido;
```

Imagem 13 - Representação em structs dos pedidos.

Os membros da struct que representam os itens do cardápio podem ser definidos como:

- **nome[100]:** Um vetor de caracteres (string) para armazenar o nome do item.
- **descri[100]:** Um vetor de caracteres para armazenar a descrição do item.
- **preco:** Um número de ponto flutuante (float) para armazenar o preço do item.
- **catego:** Um valor do tipo categoria (uma enumeração), que indica a categoria do item (entrada, principal, sobremesa ou bebida).

Já os membros da struct que representa os pedidos feitos pelos clientes podem ser definidos como:

- **cod\_pedido:** Um número inteiro para identificar de forma única o pedido.



- **nome\_cliente[100]:** Um vetor de caracteres para armazenar o nome do cliente.
- **itens:** Um ponteiro para um vetor dinâmico de item, ou seja, o vetor de itens que o cliente selecionou para o pedido. Como o número de itens pode variar, este campo é um ponteiro para um vetor de item, o que permite que ele seja alocado dinamicamente durante a execução do programa.
- **num\_itens:** Um número inteiro que armazena a quantidade de itens no pedido.
- **status:** Um valor do tipo StatusPedido (uma enumeração) que armazena o status do pedido (como Pendente, Em\_Preparo, Pronto ou Entregue).

### 3.6. Enumerações

No sistema foram criadas duas enumerações: a enumeração categoria e a enumeração StatusPedido. A enumeração categoria (imagem 14) define as possíveis categorias para os itens do cardápio (entrada, principal, sobremesa ou bebida), permitindo que cada item seja identificado facilmente.

```
// Enumeração de categorias do cardápio
typedef enum {entrada, principal, sobremesa, bebida} categoria;
```

Imagem 14 - Definição de enumeração de categorias do cardápio.

Já a enumeração StatusPedido (imagem 15) define os possíveis status em que pode estar o pedido feito pelo cliente (Pendente, Em\_Preparo, Pronto ou Entregue), permitindo a melhor comunicação entre o cliente e organização entre os funcionários do restaurante.

```
// Enum para status dos pedidos
typedef enum {PENDENTE, EM_PREPARO, PRONTO, ENTREGUE} StatusPedido;
```

Imagem 15 - Definição de enumeração de status dos pedidos.

#### **4. Conclusão**

O sistema de gerenciamento de restaurante foi projetado de forma eficiente e modular, utilizando eficientemente os conceitos fundamentais da linguagem C, como combinação de ponteiros, alocação dinâmica de memória, vetores dinâmicos, *structs* e enumerações. O uso de ponteiros e alocação dinâmica permite o gerenciamento flexível de memória, enquanto as *structs* e vetores dinâmicos organizam os dados de forma eficiente. Essa arquitetura permite que o sistema seja facilmente expandido e adaptado a novas funcionalidades.