

# TRABALHO DE ESTRUTURAS DE DADOS

Desenvolvimento por:

## JOÃO PEDRO SÁ LEONARDO SILVA

# A Loja Digital do Seu Zé

Desvendando Estruturas de Dados: **Structs, Listas e Filas**



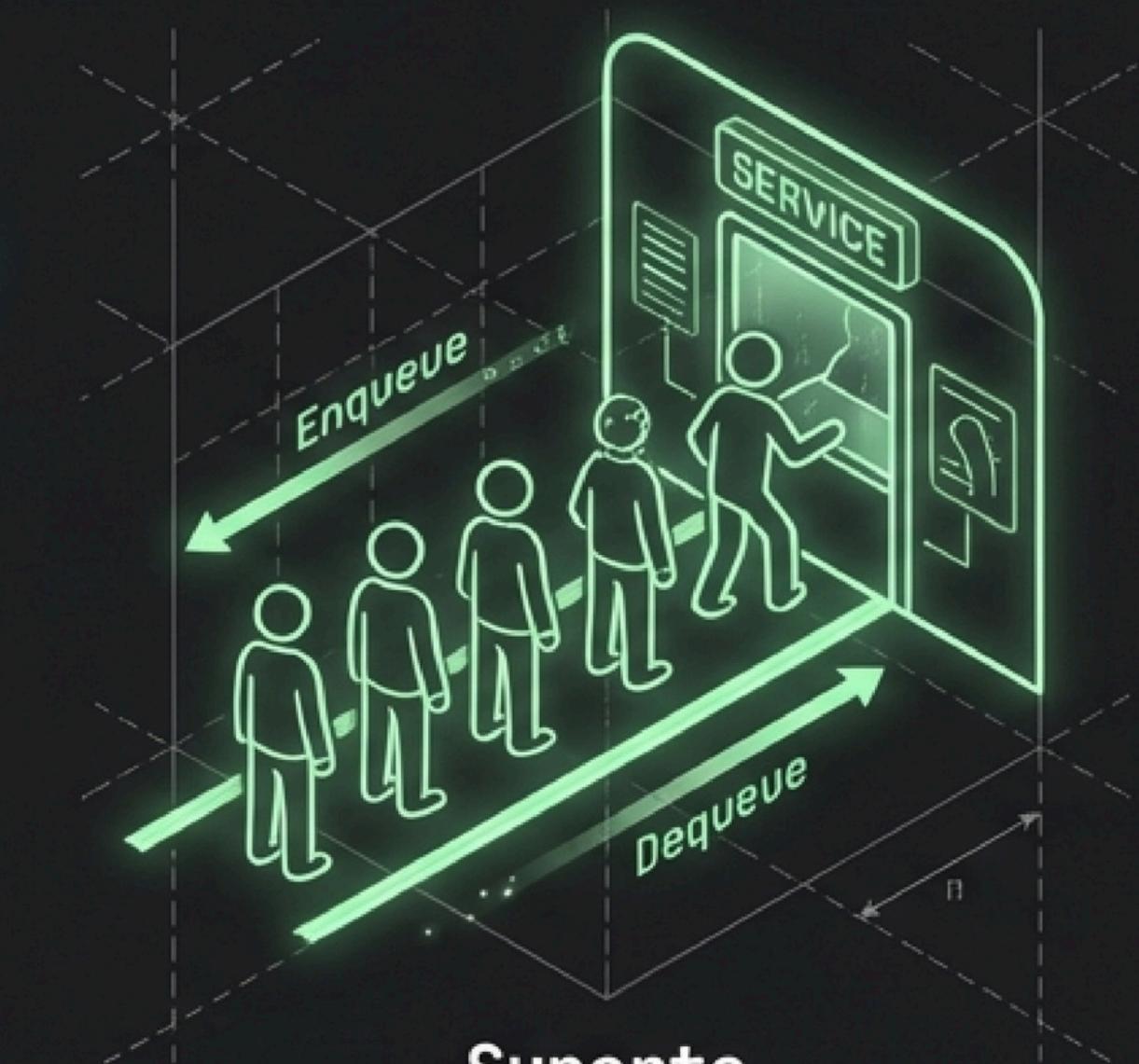
Uma explicação visual da arquitetura do sistema.

# A Missão do Código

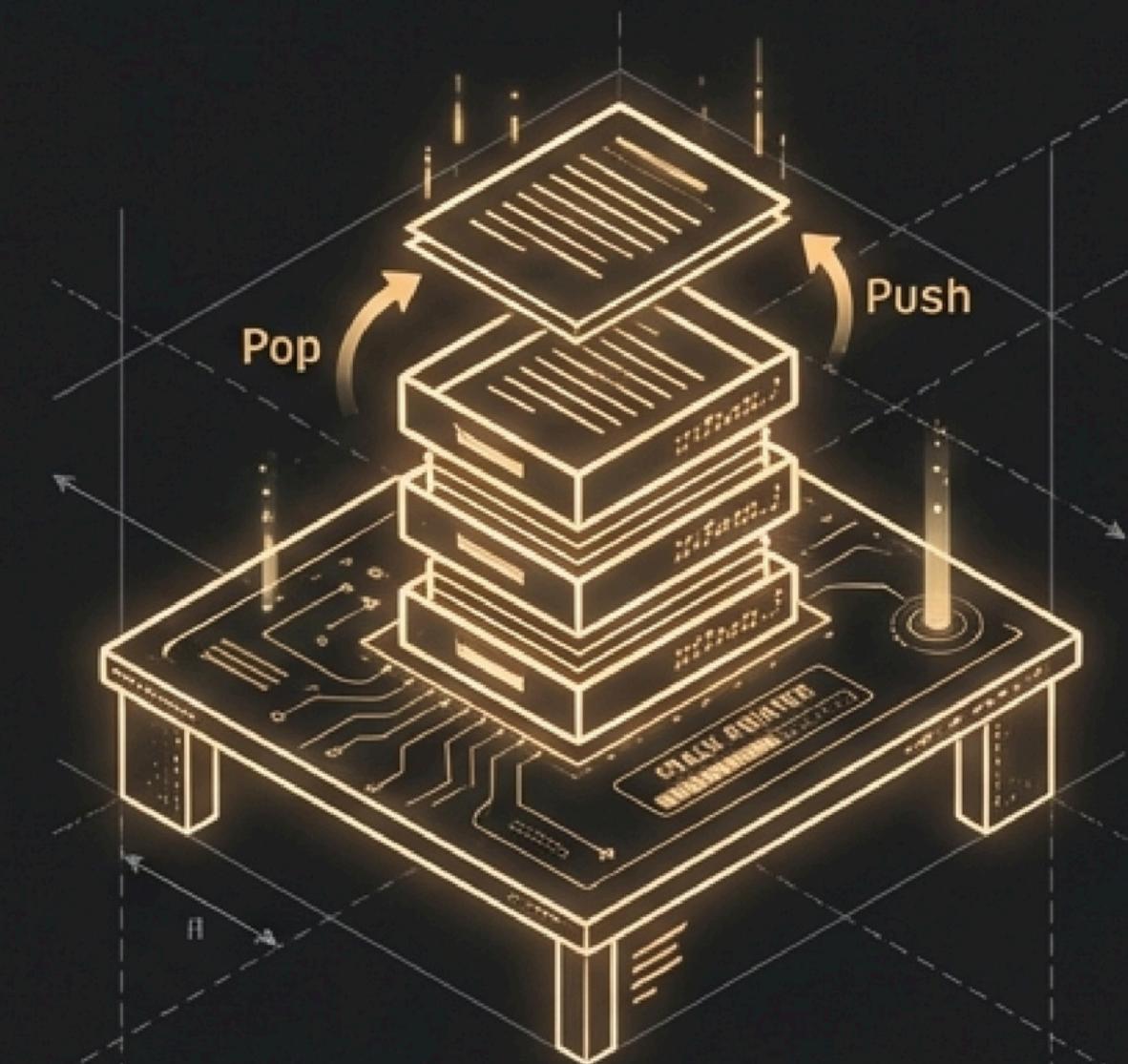
O sistema gerencia três fluxos distintos na memória.



**Estoque**  
(Lista Encadeada)



**Suporte**  
(Fila / Queue)

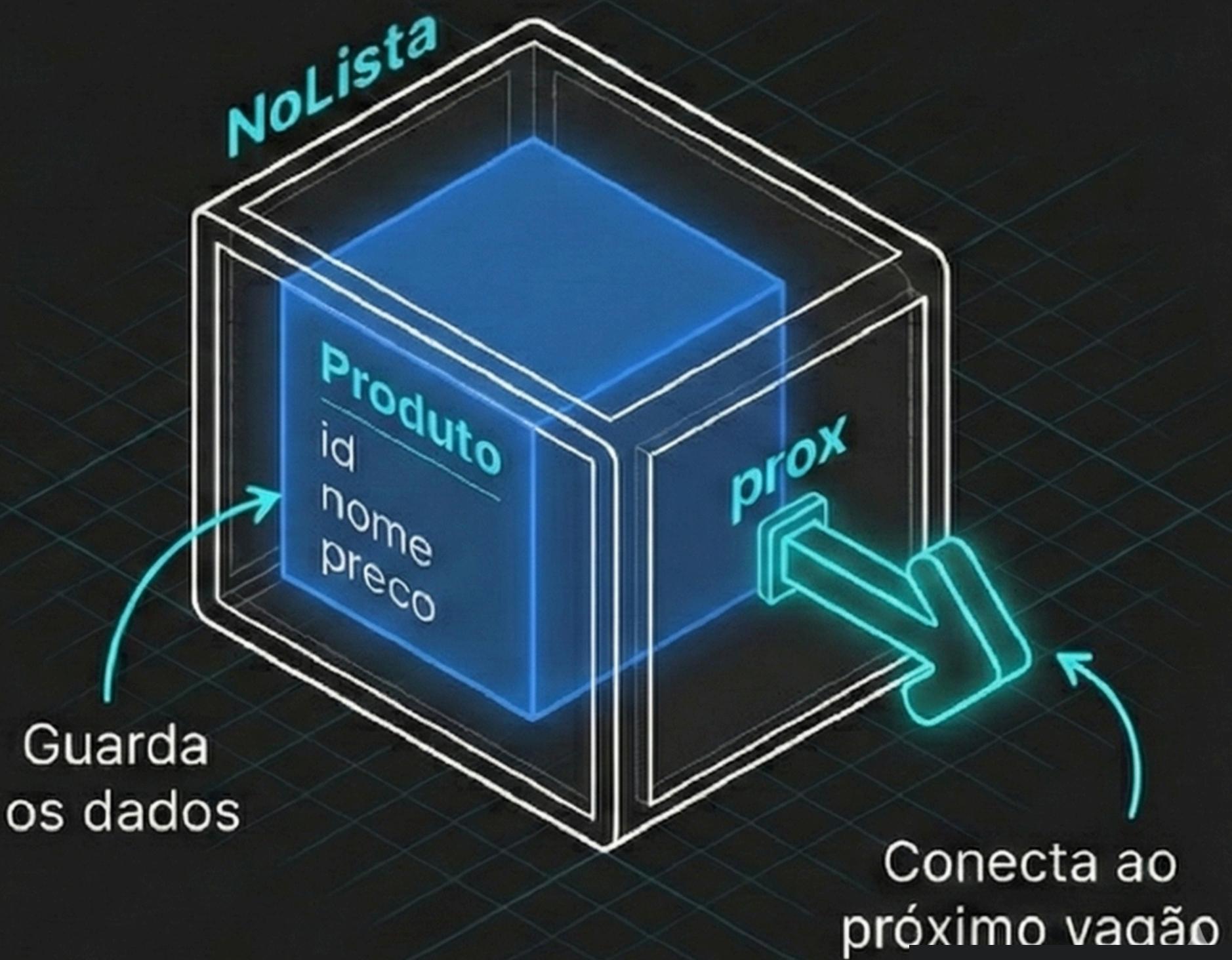


**Histórico**  
(Pilha / Stack)

As estruturas fundamentais (**typedef struct**) definem como esses objetos são organizados na memória do computador.

# Estoque (Lista Encadeada)

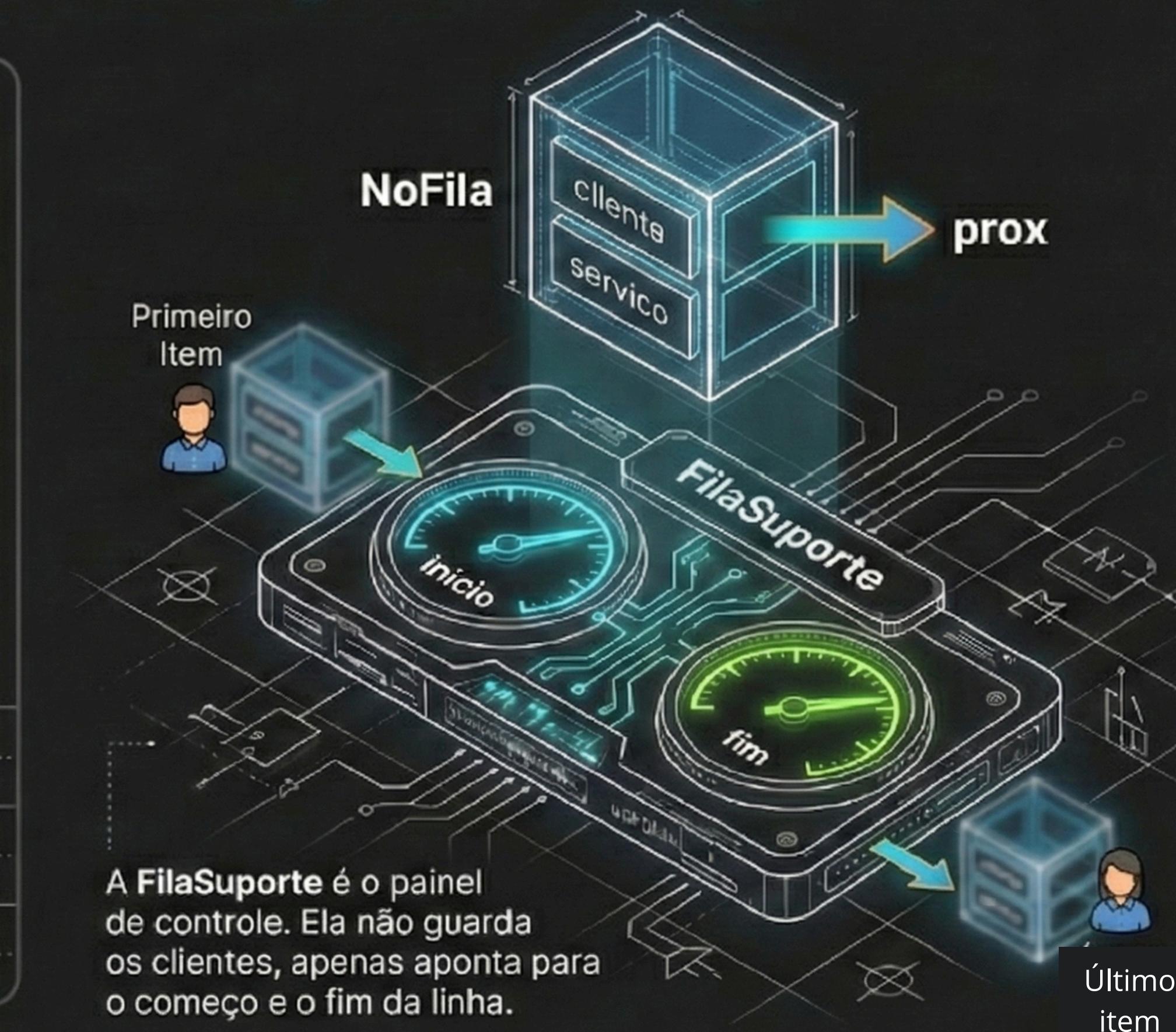
```
typedef struct {  
    int id;  
    char nome[50];  
    float preco;  
} Produto;  
  
typedef struct NoLista {  
    Produto dado;  
    struct NoLista* prox;  
} NoLista;
```



# Fila de Atendimento (Fila - FIFO)

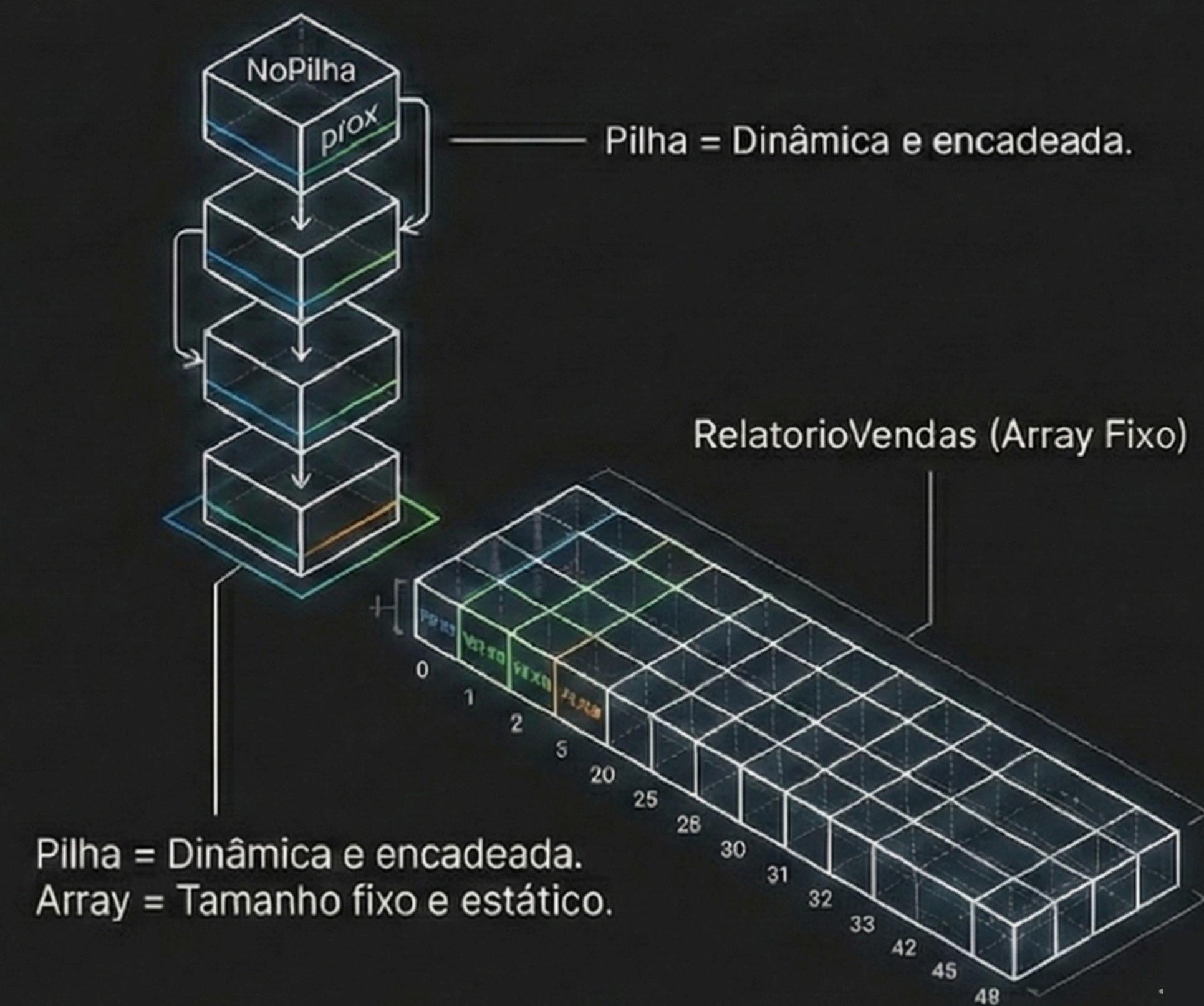
```
typedef struct NoFila {  
    char cliente[50];  
    char servico[50];  
    struct NoFila* prox;  
} NoFila;
```

```
typedef struct {  
    NoFila* inicio;  
    NoFila* fim;  
} FilaSuporte;
```



# Histórico (Pilha - LIFO) e Relatório (Array)

```
typedef struct NoPilha {  
    int id_produto;  
    struct NoPilha* prox;  
} NoPilha;  
  
#define MAX_VENDAS 50  
typedef struct {  
    float valores[MAX_VENDAS];  
    int qtd;  
} RelatorioVendas;
```



# Gerenciando o Estoque: Adicionar Produto

## Passo 1: A Criação (malloc)

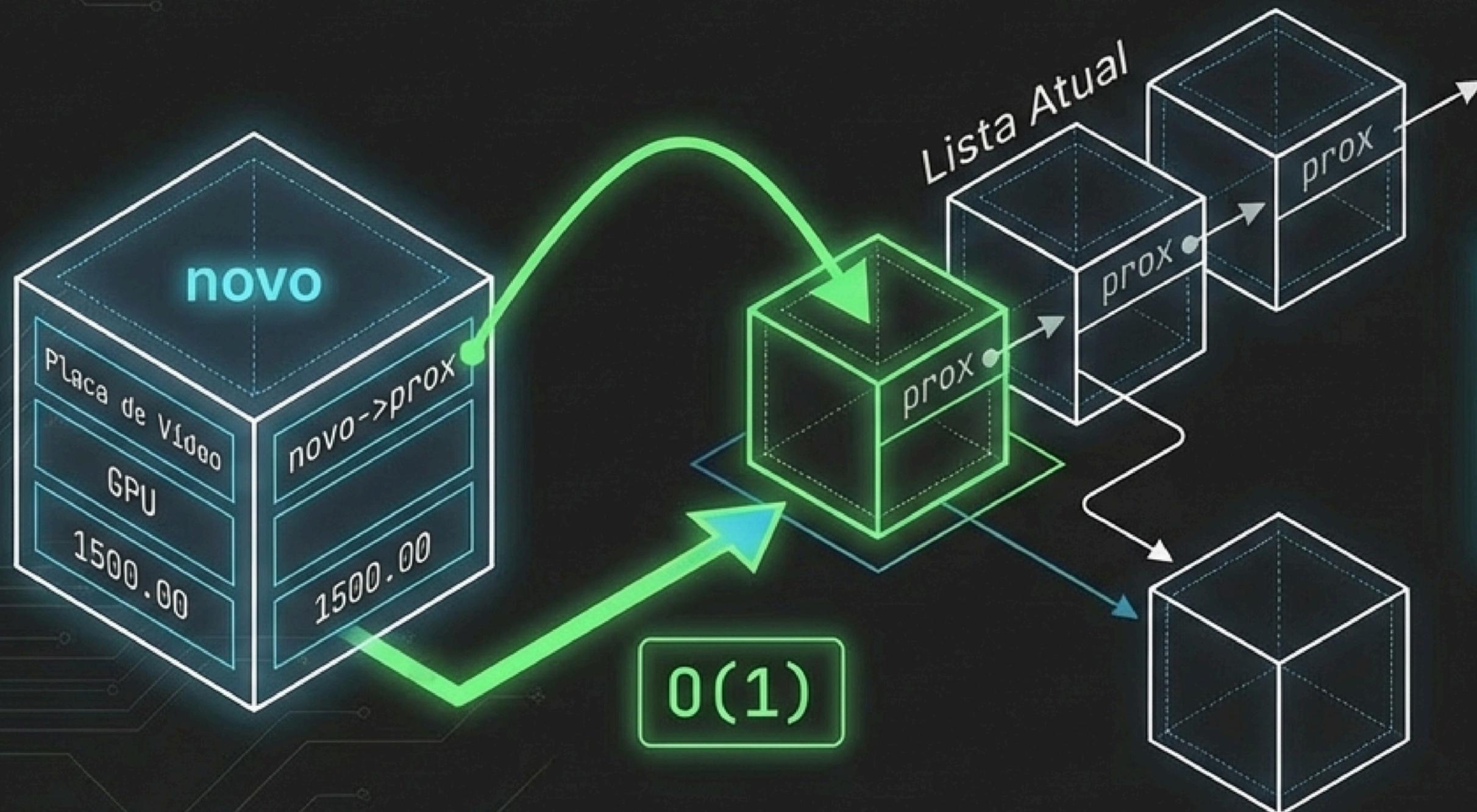
```
NoLista* novo = (NoLista*)  
    malloc(sizeof(NoLista));
```



O comando **malloc** reserva um espaço na memória  
e devolve o endereço para o ponteiro '**novo**'.

# Adicionar Produto: Passo 2

## Furando a Fila (Inserção no Início)



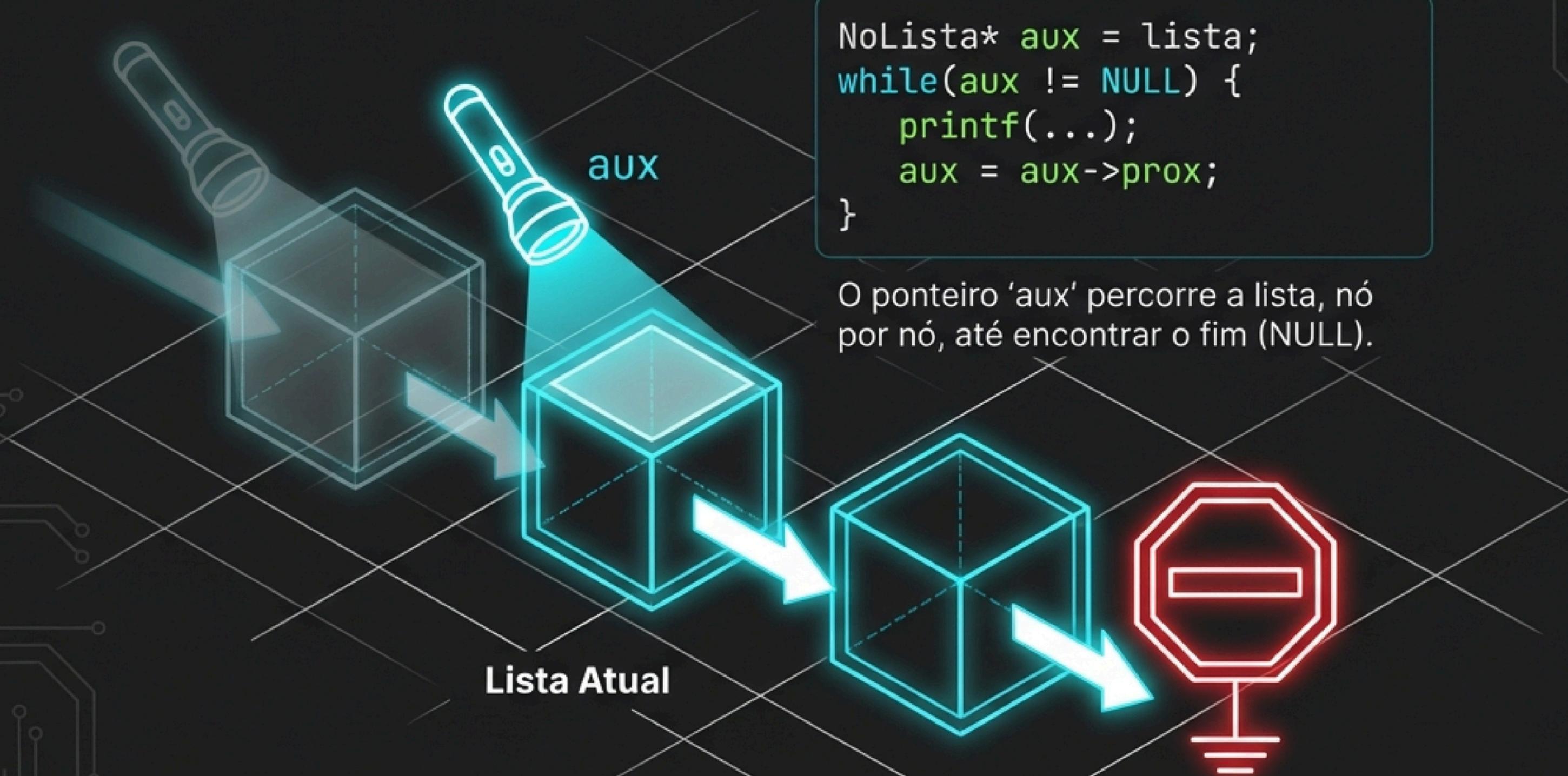
Código:

```
novo->prox = lista;  
return novo;
```

O novo produto aponta para o antigo começo. Ele se torna o novo chefe da lista.

# Lendo o Estoque: O Loop While

## A Lanterna (Ponteiro Auxiliar)



# Abrindo o Suporte: Iniciar Fila Estado Zero

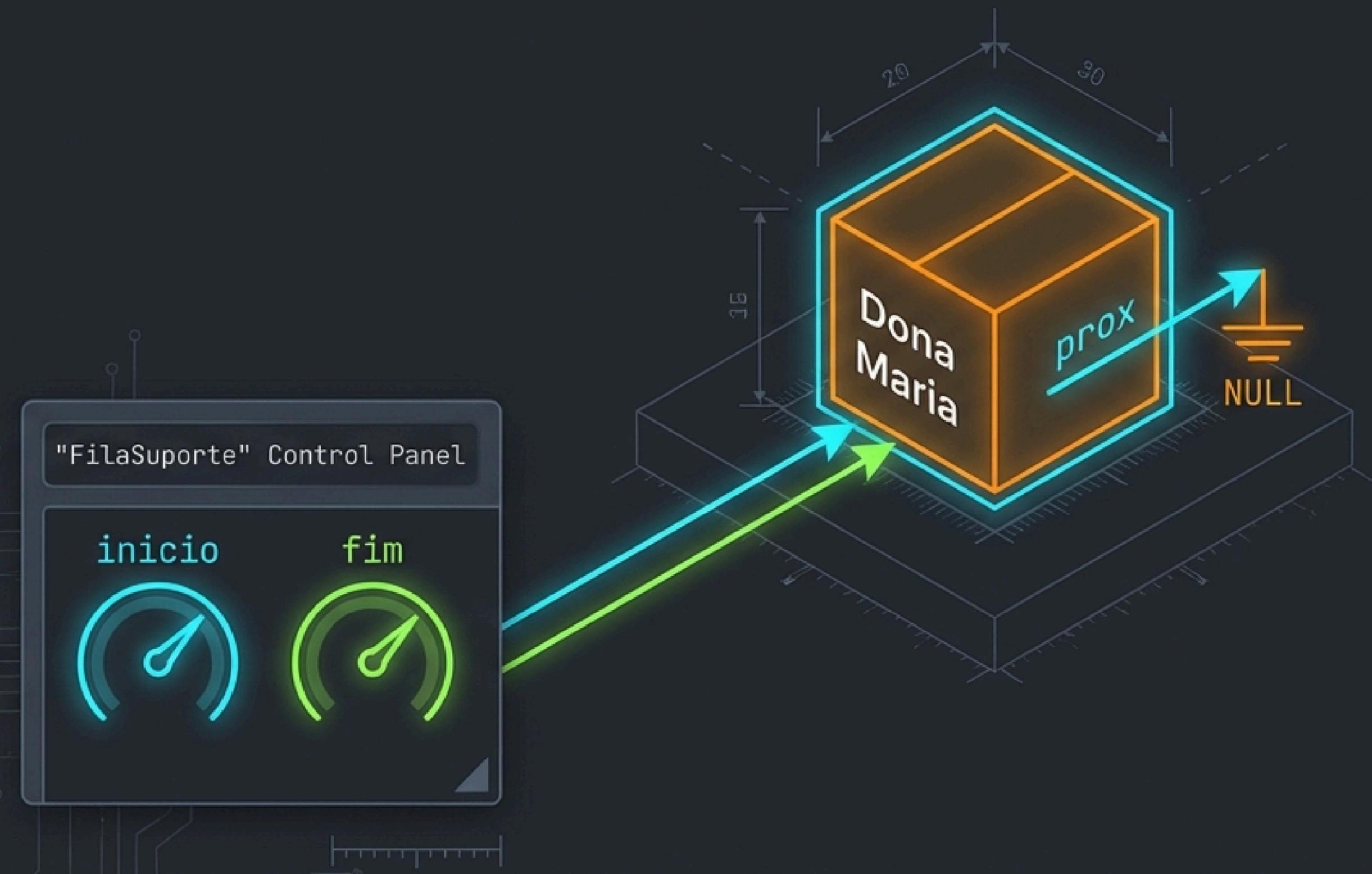


```
void iniciar_fila(FilaSuporte* f) {  
    f->inicio = NULL;  
    f->fim = NULL;  
}
```

Antes da loja abrir, a fila não existe. Ponteiros apontam para NULL para evitar erros.

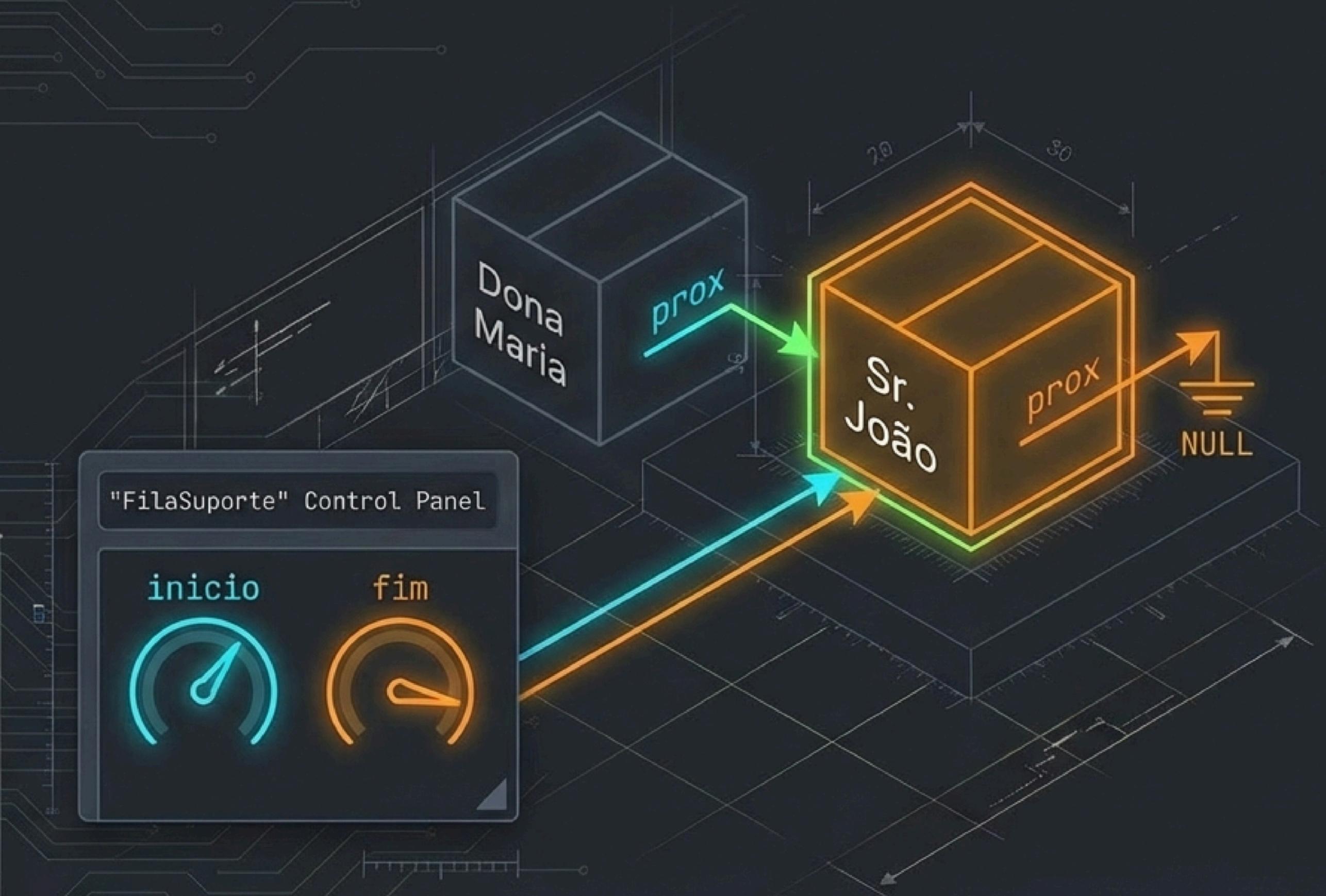
# Chega um Cliente: Entrar na Fila (Vazia)

## (Estado Zero)



Primeiro cliente:  
Ele é o início  
E o fim da fila  
simultaneamente.

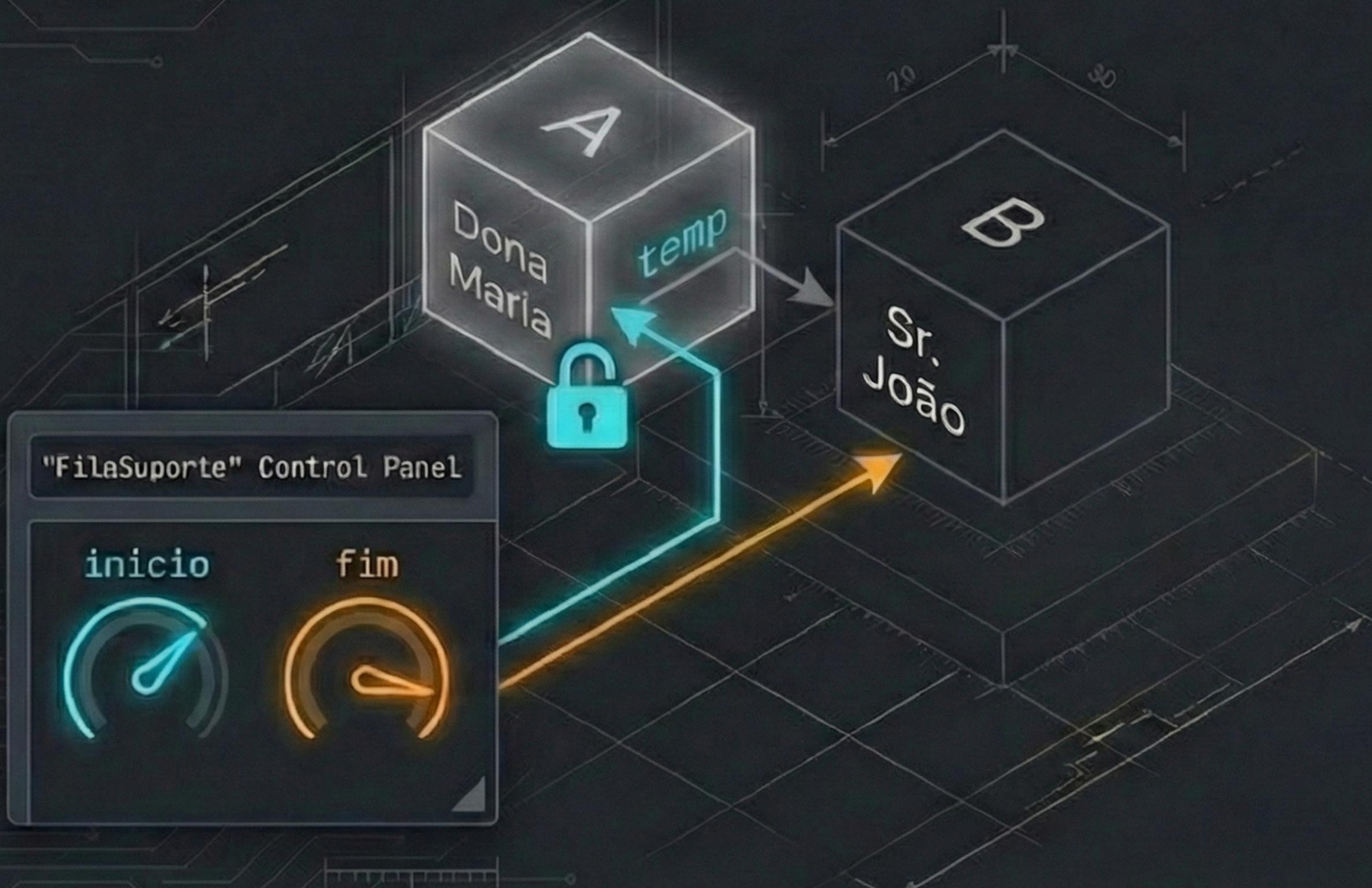
# Entrar na Fila (Ocupada)



```
else {  
    f->fim->prox = novo;  
    f->fim = novo;  
}
```

A Conexão: O último da fila dá a mão para o novo. A placa de 'Fim' é atualizada.

# Atender Cliente: Sair da Fila



```
NoFila* temp =  
f->inicio;  
f->inicio = temp;  
f->inicio->prox;
```

Passando a Vez: O ponteiro de início avança para o segundo da fila. O primeiro é segurado pelo 'temp'.

# Finalizando e Limpando Memória

`free(temp)`



Limpeza de Memória

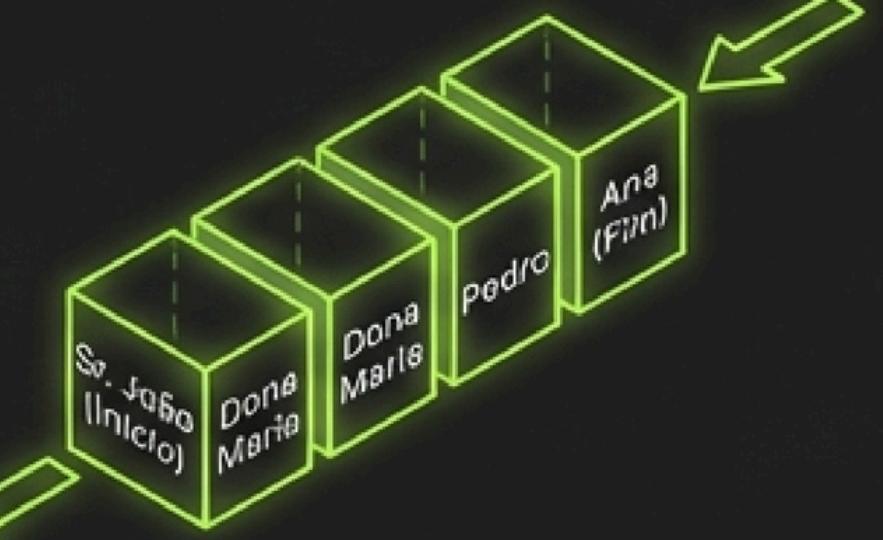


Caso de Fila Vazia

```
if (f->inicio == NULL)
    f->fim = NULL;
    free(temp);
}
```

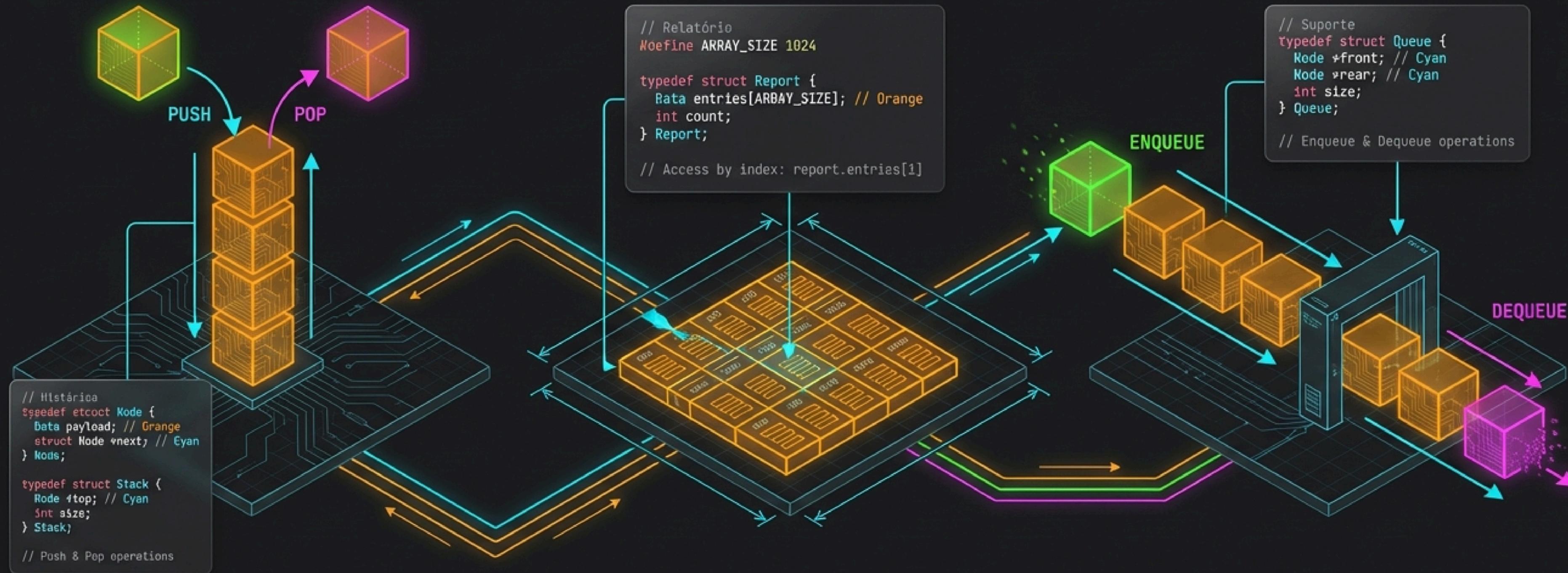
Não deixe pontas soltas! Se a fila esvaziar, o ponteiro 'fim' também deve ser resetado para NULL.

# Resumo da Operação do Seu Zé

	Estoque (Lista)	Suporte (Fila)
Ação:	Adicionar	Entrar / Atender
Lógica:	Entra no Início (LIFO)	Entra no Fim / Sai do Início (FIFO)
		

**Lista** = Rapidez na inserção. **Fila** = Justiça no atendimento.

# A Arquitetura da Memória



**Histórico (Pilha)**

Estrutura dinâmica.  
LIFO (Last In, First Out).

**Relatório (Array)**

Estrutura estática.  
Acesso direto e tamanho fixo.

**Suporte (Fila)**

Estrutura de limpeza.  
FIFO (First In, First Out).

As estruturas fundamentais (`typedef struct`) definem como esses objetos são organizados na memória do computador.

# Navegando no Passado

## Operação: Mostrar Histórico

```
void mostrar_historico(NoPilha* topo) {  
    NoPilha* aux = topo;  
    while(aux != NULL) {  
        printf("ID: %d\n", aux->id_produto);  
        aux = aux->prox;  
    }  
}
```

O ponteiro 'topo' fica imóvel para não pertermos a pilha. Usamos uma cópia ('aux') para percorrer os dados.



# Limpeza de Memória (Pilha)

Operação: Liberar Pilha

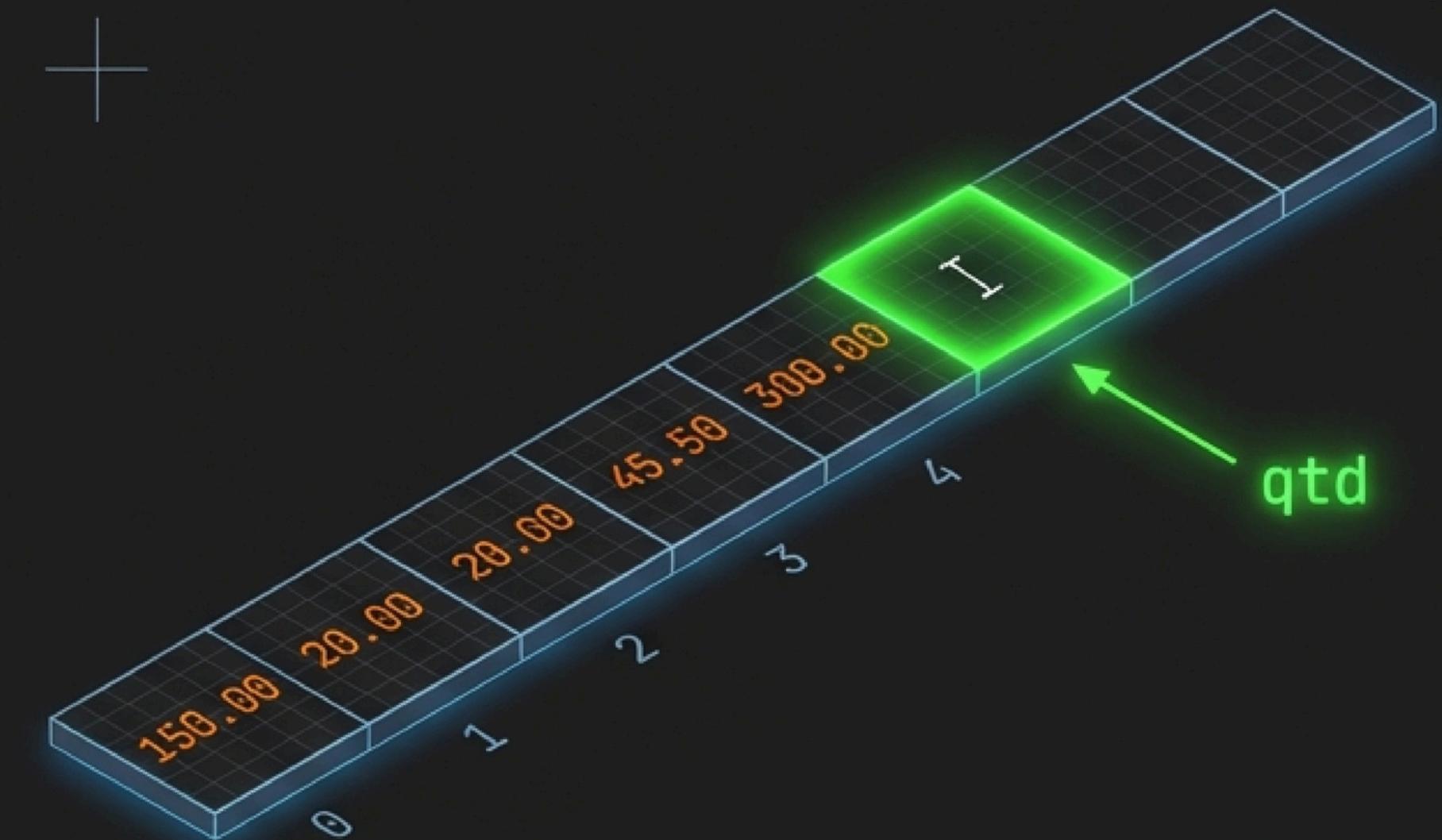


**Cuidado:** Se der `free(atual)` sem salvar o próximo, você quebra a corrente!

# Módulo 2: O Relatório (Array Estático)

Operação: Registrar Venda

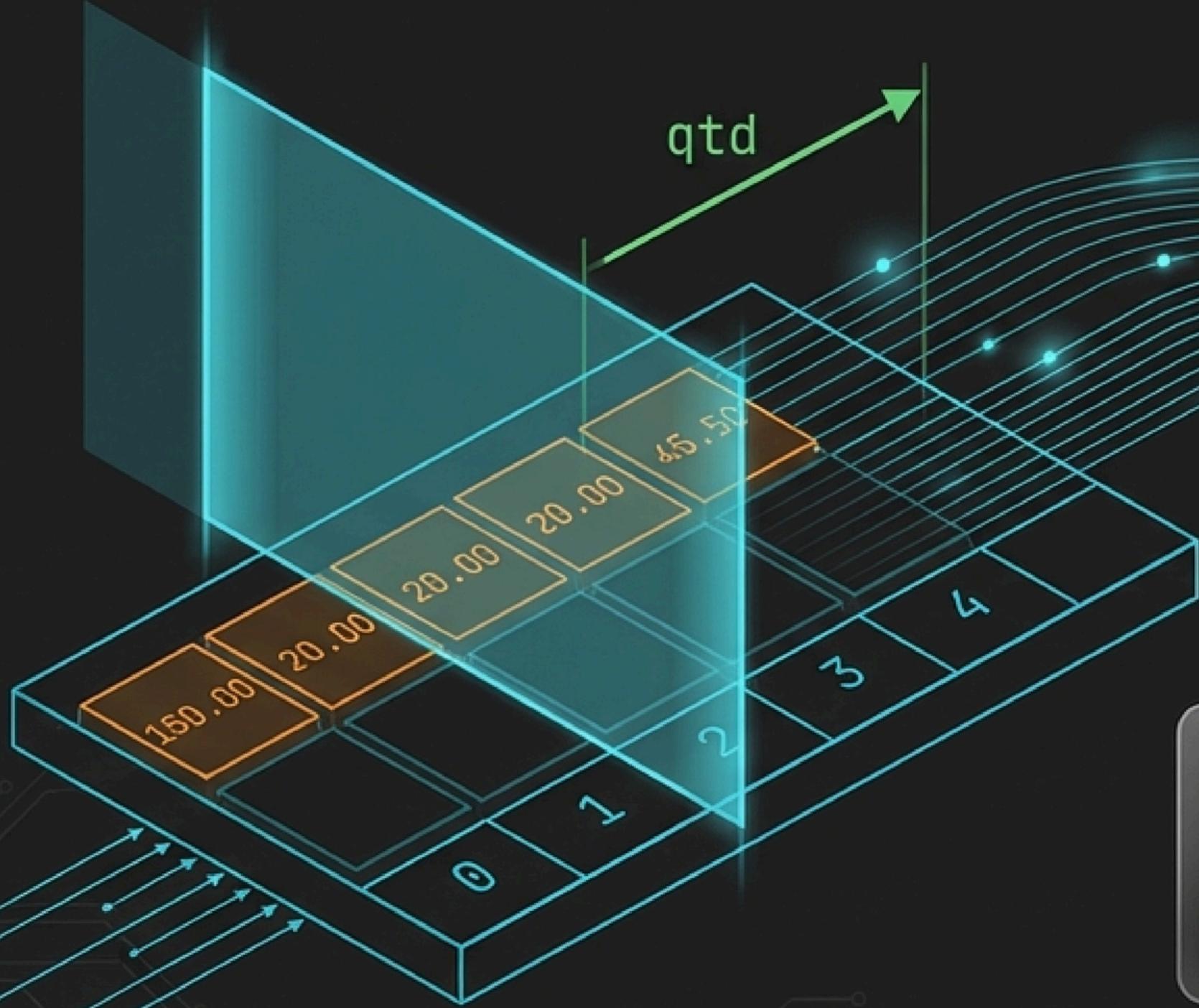
```
void registrar_venda(RelatorioVendas* r,  
                      float valor) {  
    if (r->qtd < MAX_VENDAS) {  
        r->valores[r->qtd] = valor;  
        r->qtd++;  
    }  
}
```



Acesso Direto: Sem ponteiros. Usamos a variável 'qtd' como índice para encontrar a primeira página em branco do caderno.

# Calculando o Lucro

Operação: Exibir Relatório



## TOTAL

```
total = 0  
total += 150.00  
total += 20.00  
total += 20.00  
total += 45.50
```

R\$ 235.50

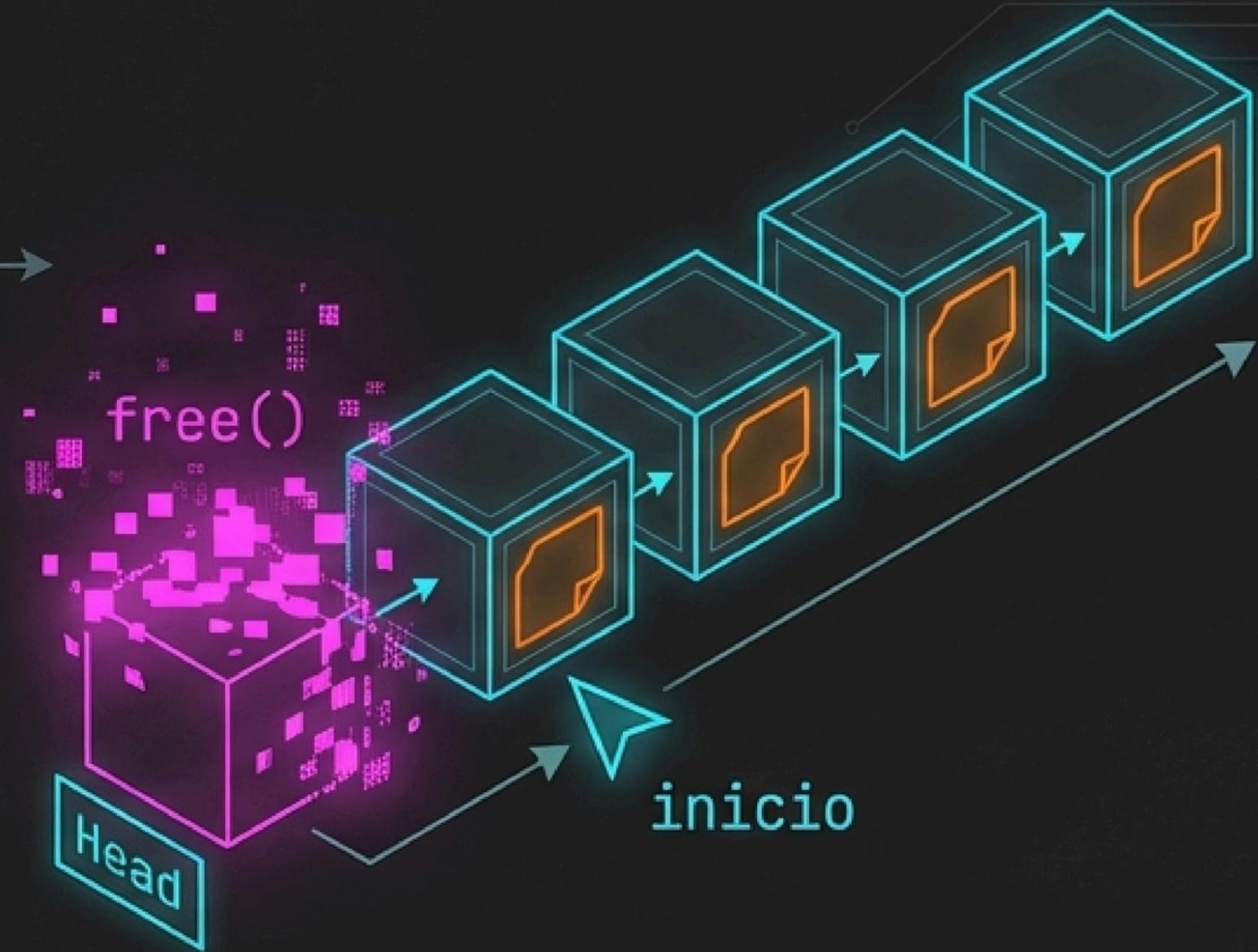
```
for(int i = 0; i < r->qtd; i++) {  
    printf("Venda %d: R$ %.2f\n", i+1, r->valores[i]);  
    total += r->valores[i];  
}
```

Eficiência: O loop percorre estritamente as vendas realizadas (0 até `qtd`), ignorando o espaço vazio restante da memória.

# Módulo 3: O Suporte (Fila)

Operação: Liberar Fila (Fechamento)

```
void liberar_fila(FilaSuporte* f) {  
    while(f->inicio != NULL) {  
        atender_cliente(f); // Limpa atendendo  
    }  
}
```



**Reaproveitamento de Lógica:** Para fechar a loja, simplesmente "atendemos" (removemos) todos da fila rapidamente até que o ponteiro `inicio` aponte para NULL.

# Resumo das Operações

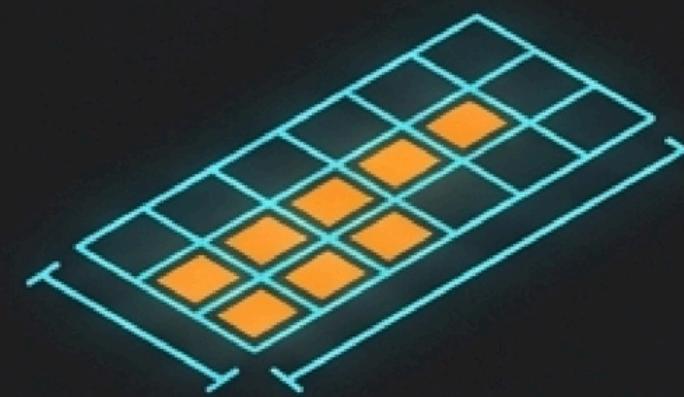
Pilha (Stack)



Dinâmica.  
Cresce para cima.

```
novo->prox = topo;
```

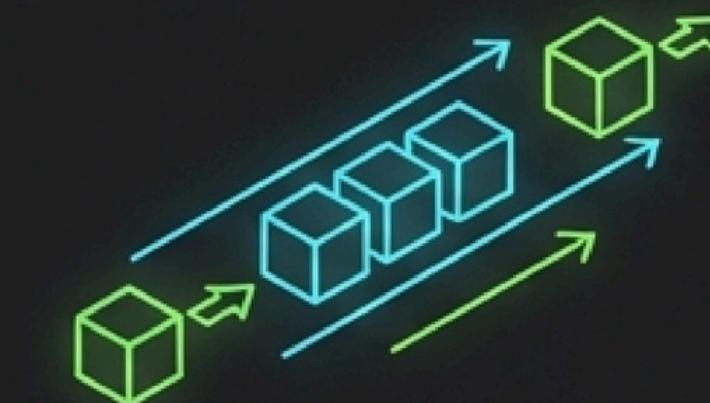
Array (Relatório)



Estática.  
Tamanho fixo.

```
valores[qtd];
```

Fila (Queue)



Dinâmica.  
Cresce para o lado.

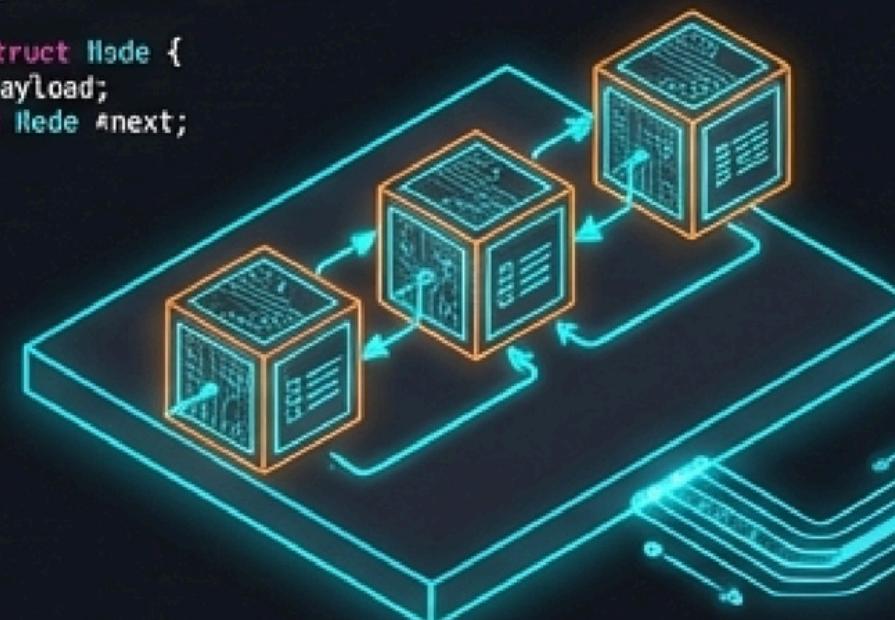
```
inicio = inicio->prox;
```

# O Cérebro da Operação: A Função Main

Orquestrando a Lógica da Loja do Seu Zé

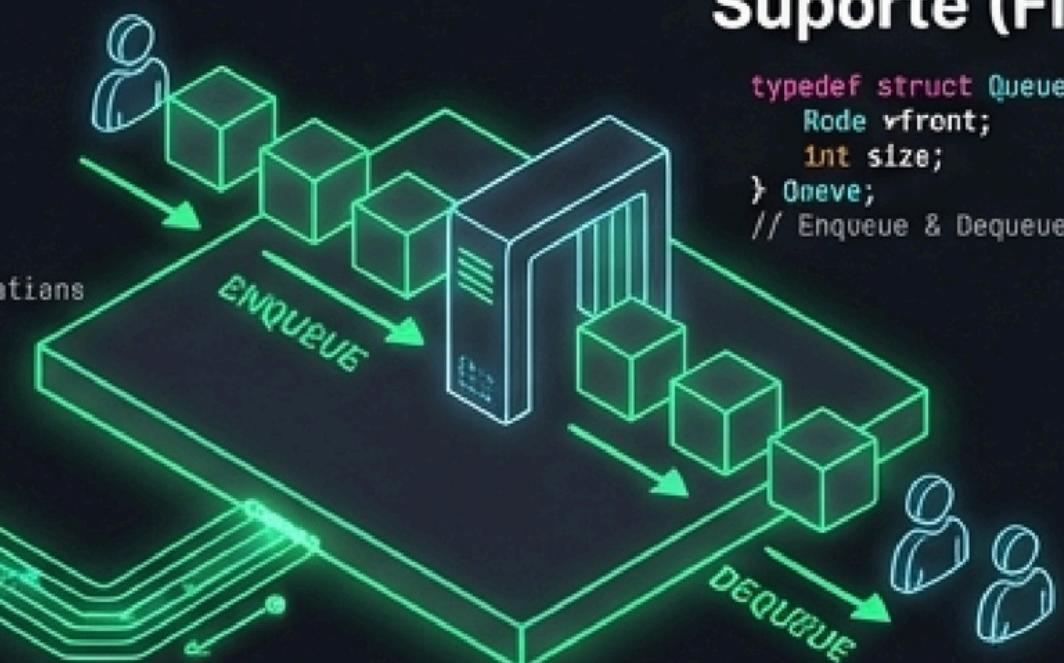
## Estoque (Lista)

```
typedef struct Node {  
    lista payload;  
    struct Node *next;  
} Node;
```



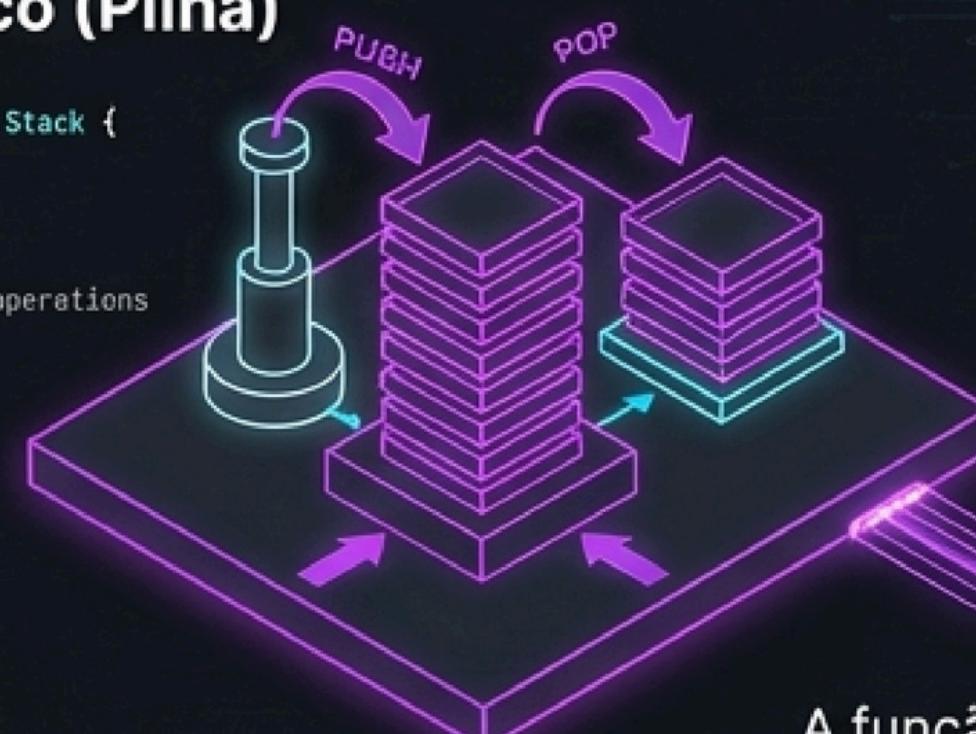
## Supporte (Fila)

```
typedef struct Queue {  
    Node *front;  
    Node *rear;  
    int size;  
} Queue;  
  
// Enqueue & Dequeue operations
```



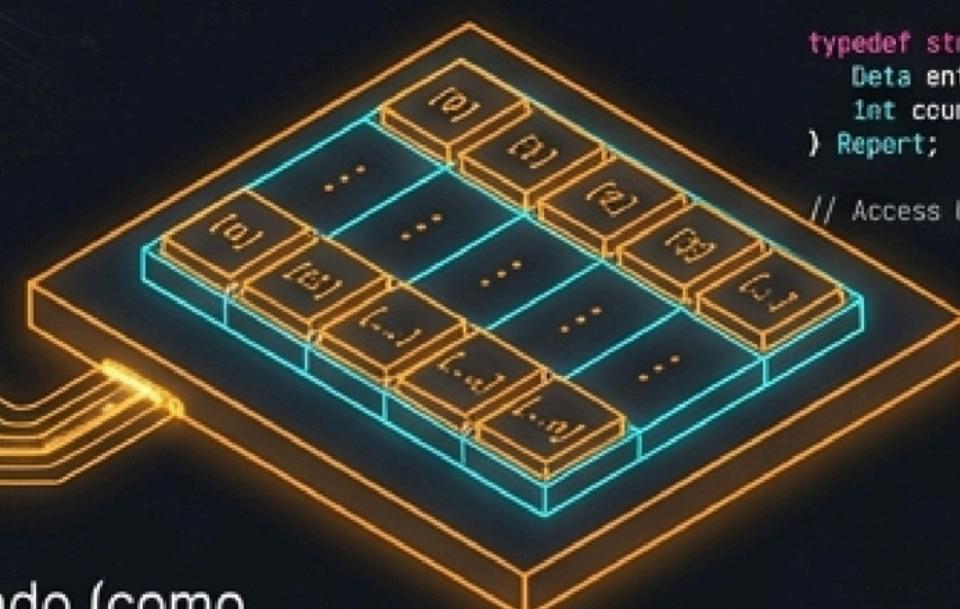
## Histórico (Pilha)

```
typedef struct Stack {  
    Node *top;  
    int size;  
} Stack;  
  
// Push & Pop operations
```



## Relatório (Array)

```
#define ARRAY_SIZE 1824  
  
typedef struct Report {  
    Data entries[ARRAY_SIZE];  
    int count;  
} Report;  
  
// Access by index: report.entries[i]
```



A função 'main' é o gerente. Ela não faz o trabalho pesado (como empilhar ou enfileirar), mas decide quem deve trabalhar e quando. É aqui que a memória é iniciada, o loop de atendimento acontece e as ordens são distribuídas.

# O "Setup": Preparando o Balcão



```
NoLista* estoque = NULL;  
FilaSuporte fila;  
iniciar_fila(&fila);  
NoPilha* historico = NULL;  
RelatorioVendas relatorio = { .qtd = 0 };
```

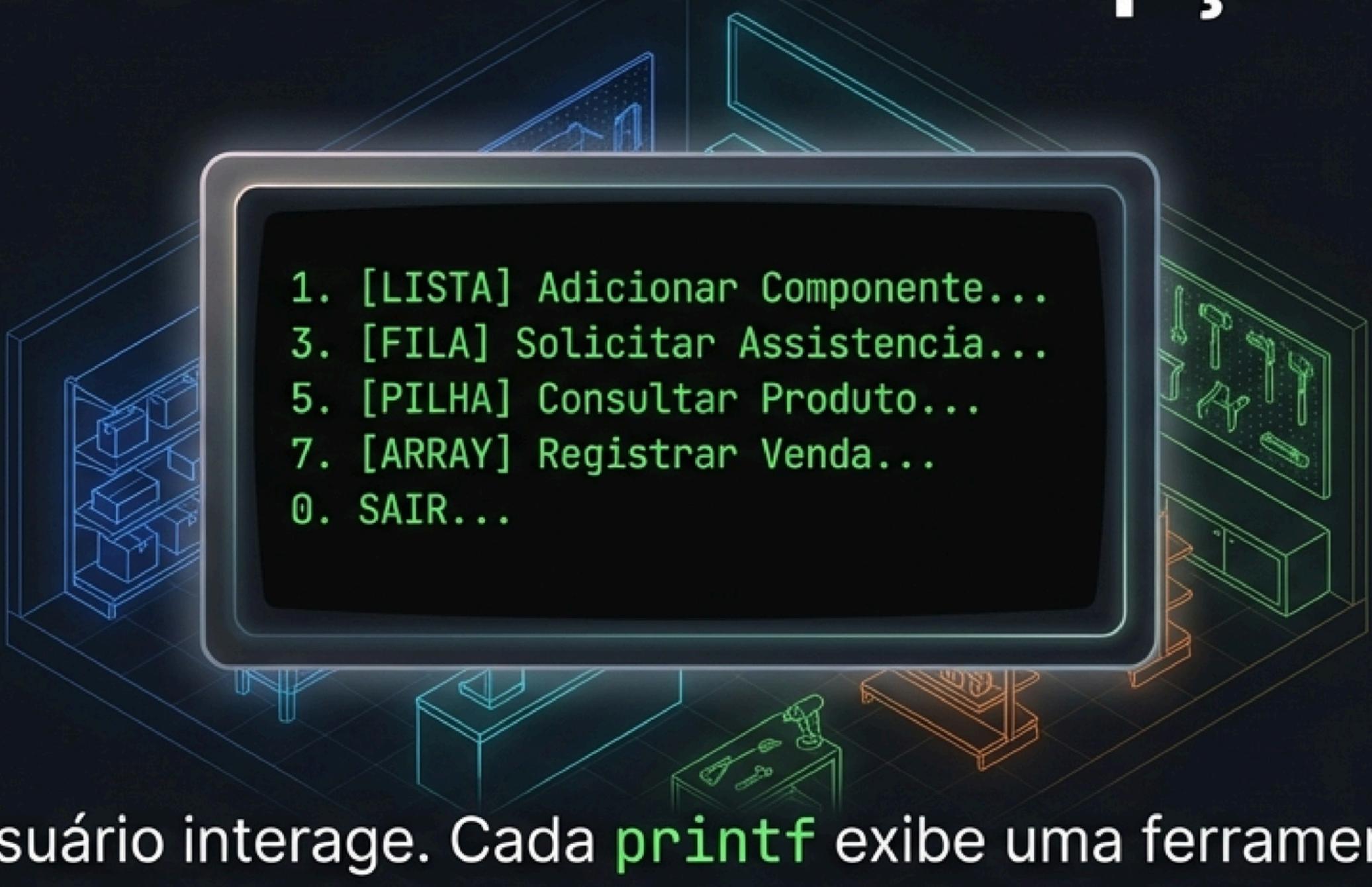
Antes de abrir a loja, precisamos de caixas vazias. Definimos os ponteiros como **NULL** e zeramos os contadores. Isso evita que o sistema tente acessar "lixo" de memória antiga.

# O Ciclo Vital: Loop Infinito



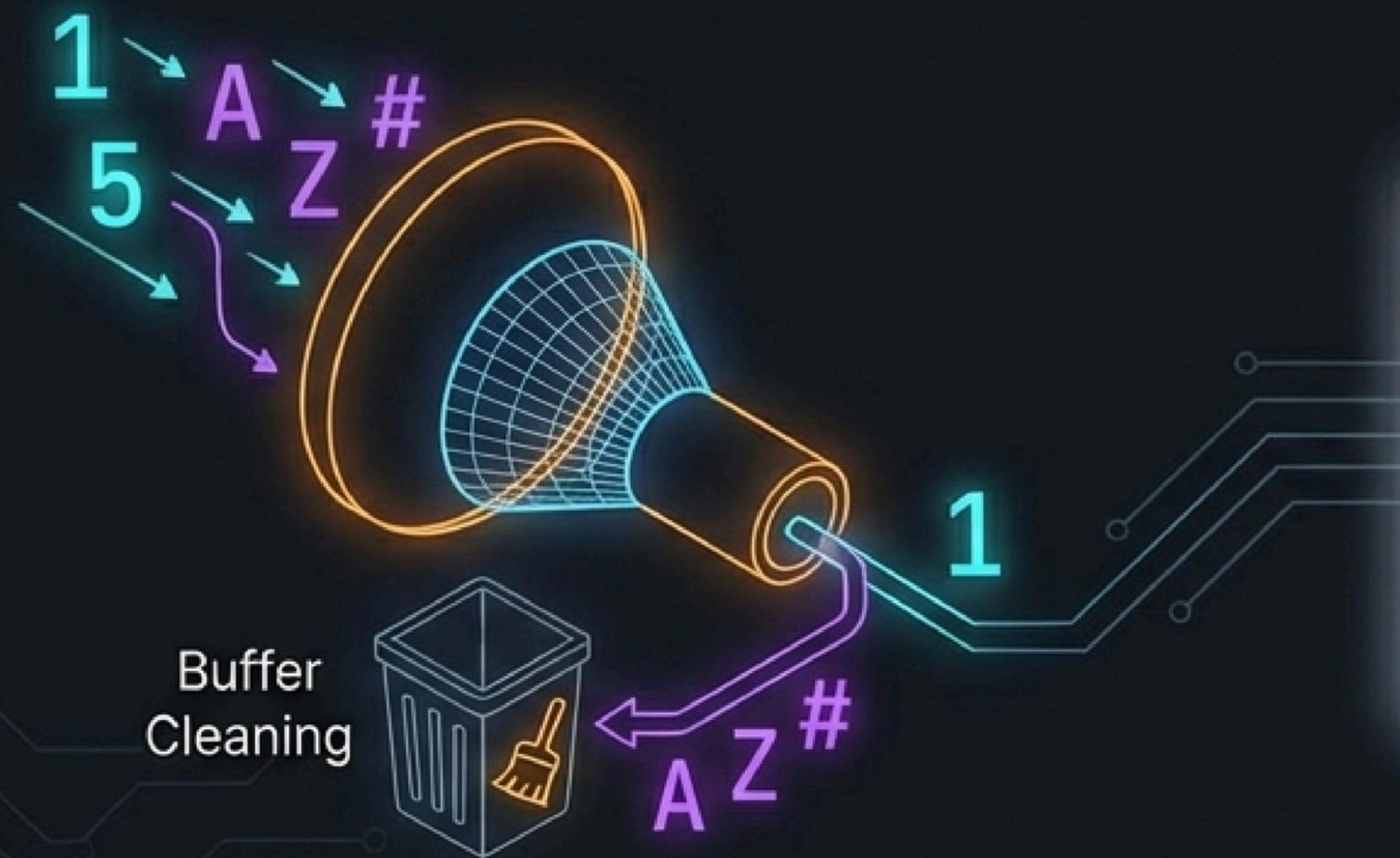
A loja não fecha após atender um cliente. O comando '**do-while**' mantém o programa vivo, repetindo o menu indefinidamente até que a variável 'opcao' seja igual a 0 (o comando de fechar as portas).

# A Vitrine: O Menu de Opções

- 
1. [LISTA] Adicionar Componente...
  3. [FILA] Solicitar Assistencia...
  5. [PILHA] Consultar Produto...
  7. [ARRAY] Registrar Venda...
  0. SAIR...

É aqui que o usuário interage. Cada `printf` exibe uma ferramenta disponível no sistema. Note que cada estrutura de dados (`Lista`, `Fila`, `Pilha`, `Array`) tem sua opção dedicada.

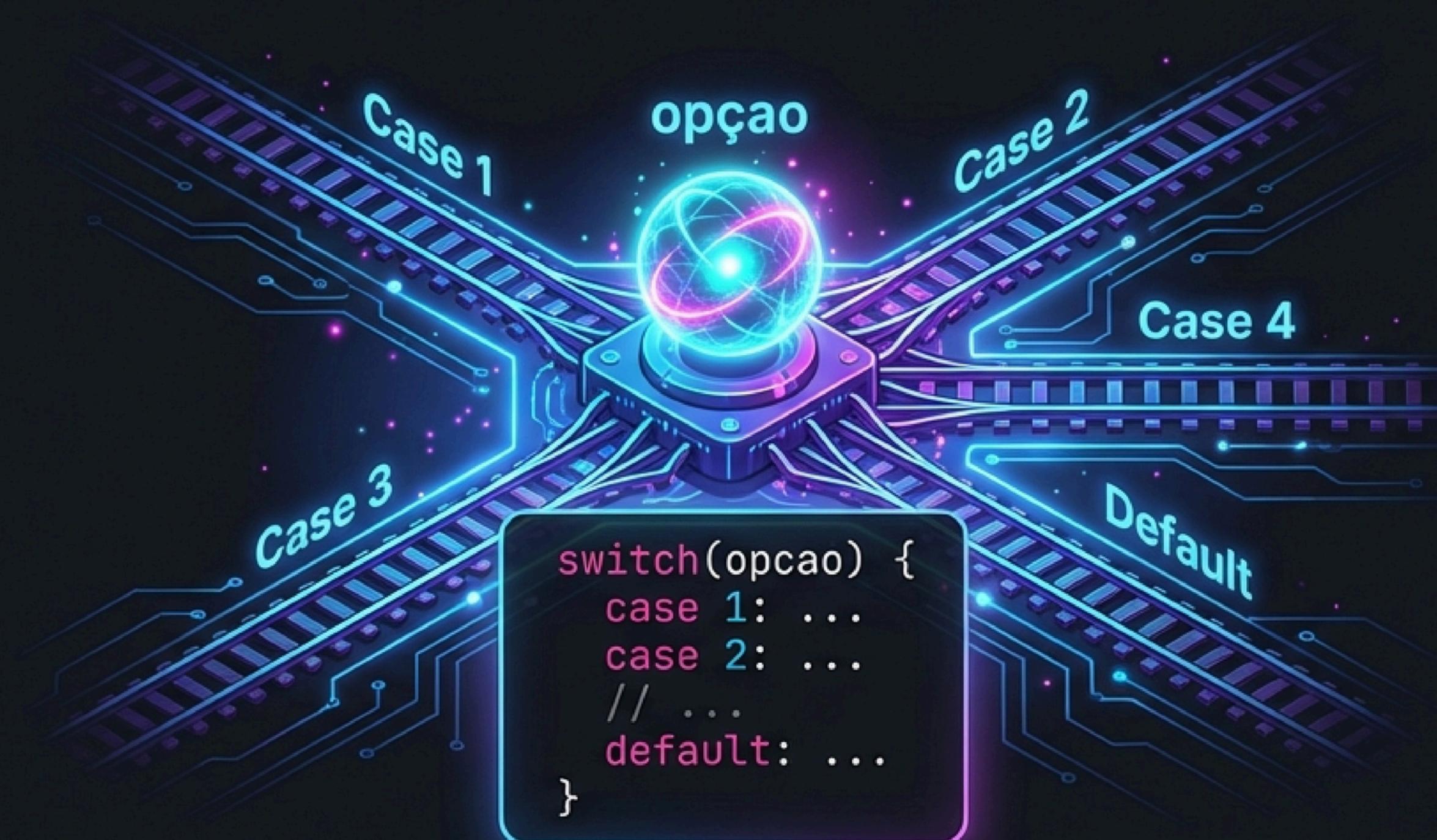
# A Entrada: Escutando o Pedido



```
printf('Escolha: ');
if (scanf('%d', &opcao) != 1) {
    while(getchar() != '\n'); // Limpa buffer
    opcao = -1; // Força erro
}
```

O scanf aguarda o comando. O código extra: `while(getchar() != '\n')` é a proteção do Seu Zé. Se o usuário digitar letras em vez de números, isso limpa a sujeira do 'buffer' para o programa não travar.

# O Direcionador: Switch Case

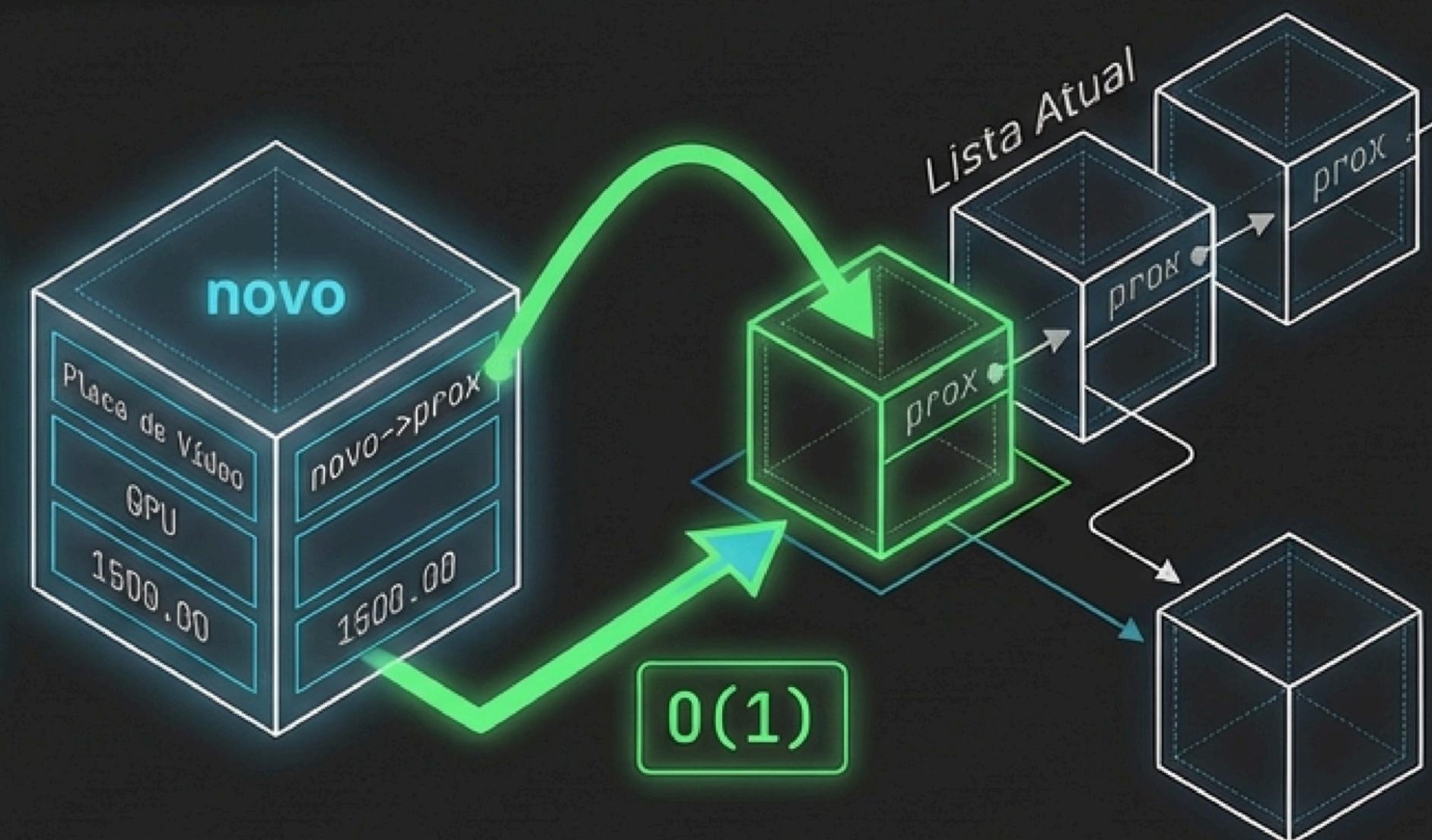


O coração da decisão. O `switch` pega o número da `opcao` e 'troca o trilho', enviando o fluxo de execução para o bloco de código correspondente (o `case`).

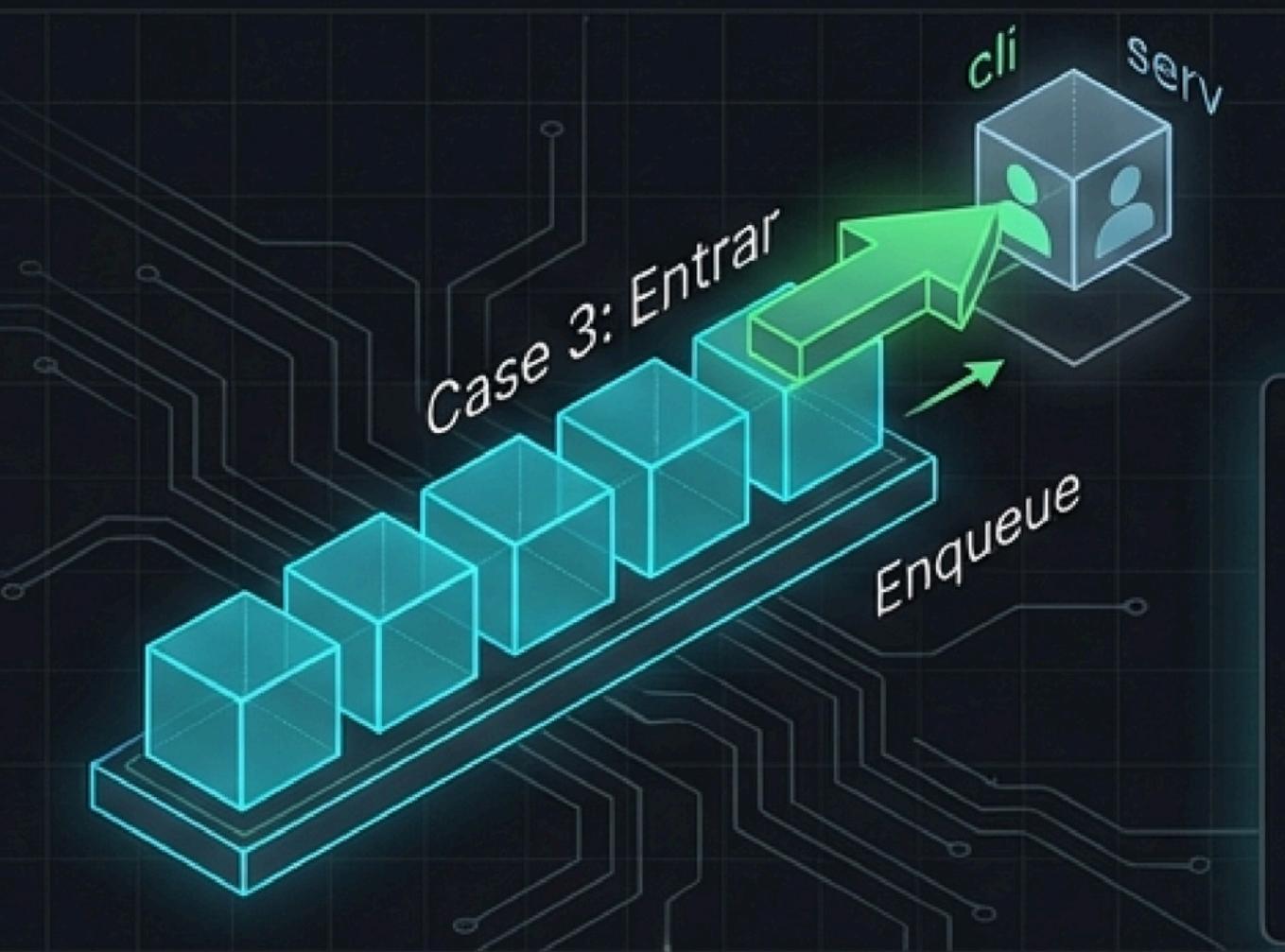
# Rota 1: Gerenciando o Estoque (Lista)

O main coleta os dados brutos (ID, nome, preço) e chama o especialista **adicionar\_produto**. Note que atualizamos o ponteiro estoque com o retorno da função, mantendo a referência da lista atualizada.

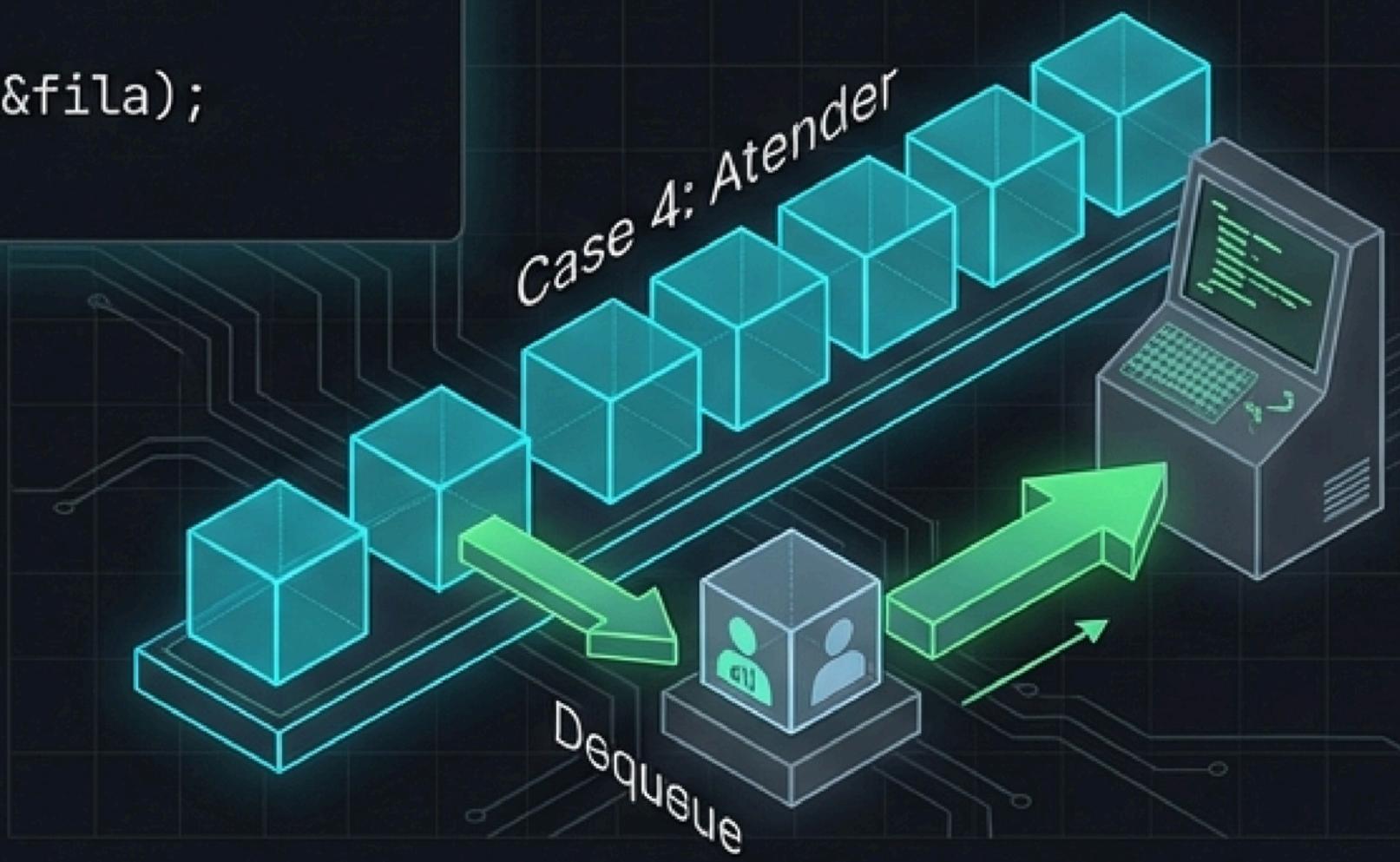
```
case 1:  
    printf('ID/Nome/Preco: ');  
    scanf(...); // Lê dados  
    estoque = adicionar_produto(estoque, ...);  
    break;
```



# Rota 2: O Suporte Técnico (Fila)



```
case 3: // Entrar  
    entrar_fila(&fila, cli, serv);  
    break;  
case 4: // Atender  
    atender_cliente(&fila);  
    break;
```



Aqui gerenciamos a Fila de Espera. O main apenas envia o endereço da fila (`&fila`) e os dados do cliente. A lógica de "quem entra" e "quem sai" (FIFO) é resolvida dentro das funções auxiliares.

# Rota 3: O Histórico (Pilha)

Toda vez que consultamos um produto, o main chama **empilhar**. O ID consultado é colocado no topo da pilha (LIFO), criando um rastro de navegação (“Voltar” do navegador).

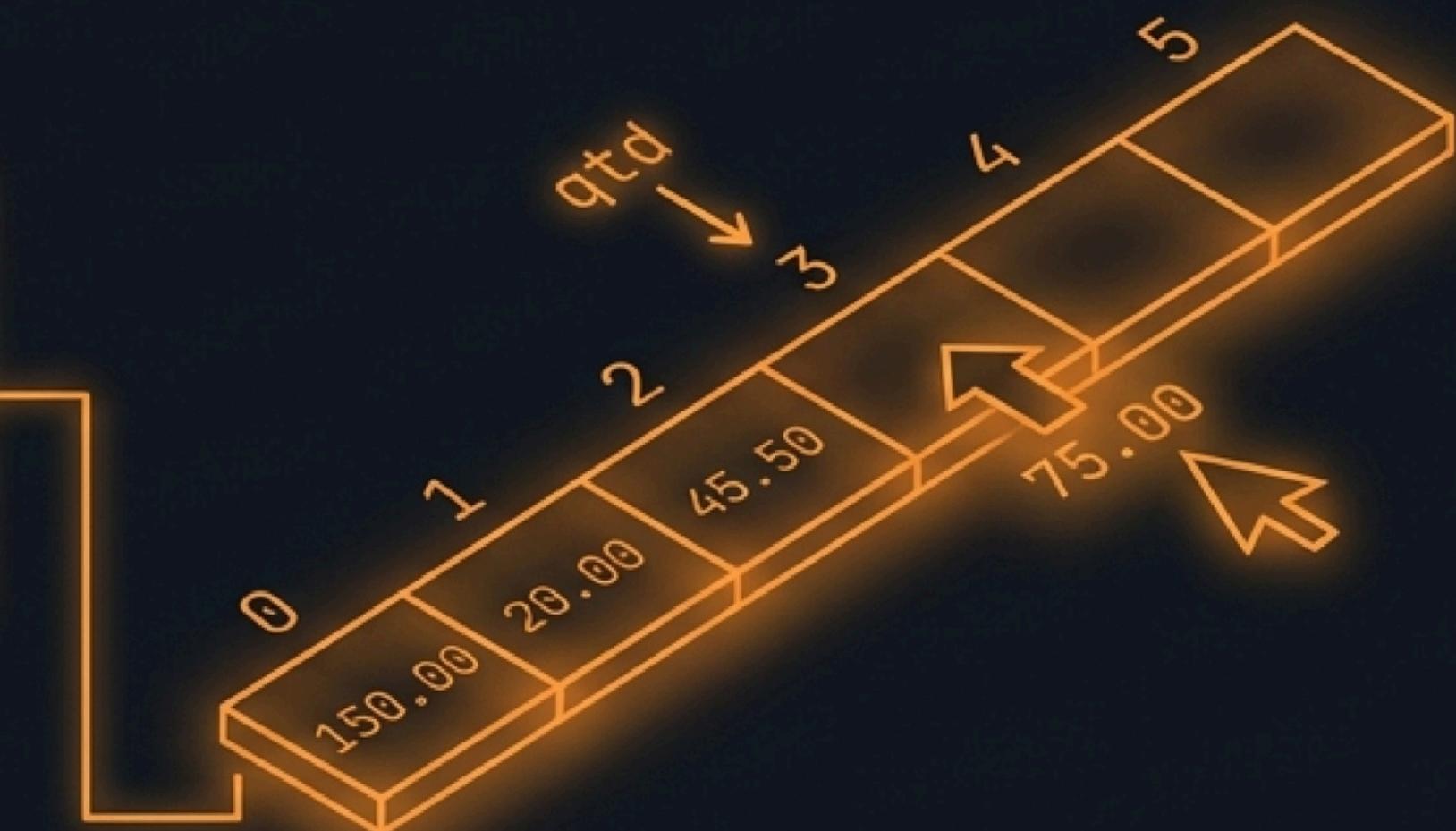
```
case 5:  
    scanf('%d', &id);  
    // Simula busca e salva no histórico  
    historico = empilhar(historico, id);  
    break;
```



# Rota 4: O Financeiro (Array)

Para o dinheiro, usamos algo fixo e rápido. O main passa o endereço do relatorio e o valor. A função preenche o vetor estático sequencialmente.

```
case 7:  
    scanf( "%f", &preco);  
    registrar_venda(&relatorio, preco);  
    break;
```



# O Caso Padrão: Tratamento de Erro

Se o cliente pedir pizza numa loja de hardware (opção 9, por exemplo), caímos no default. O sistema avisa o erro, o break acontece, e o do-while joga o usuário de volta para o menu principal.

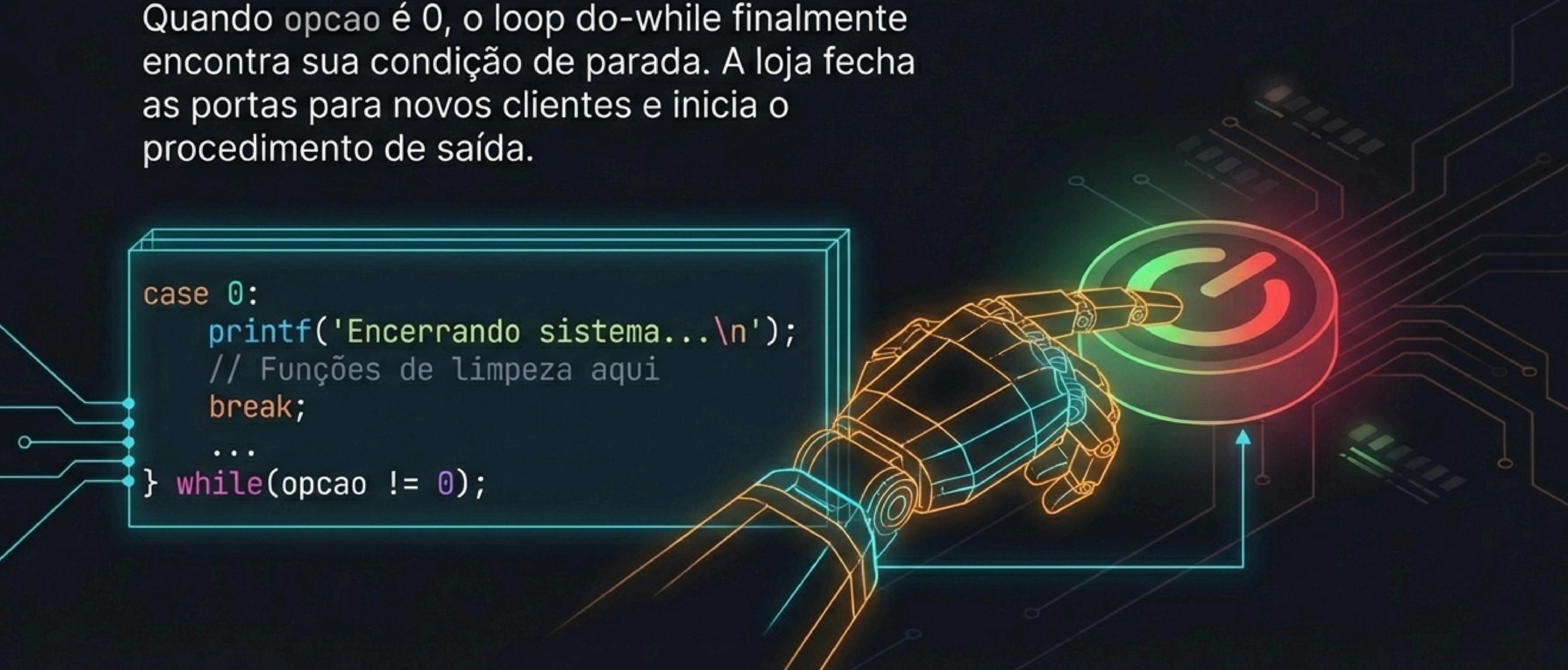
```
default:  
    printf('Opcao invalida!\n');
```



# O Encerramento: Case 0

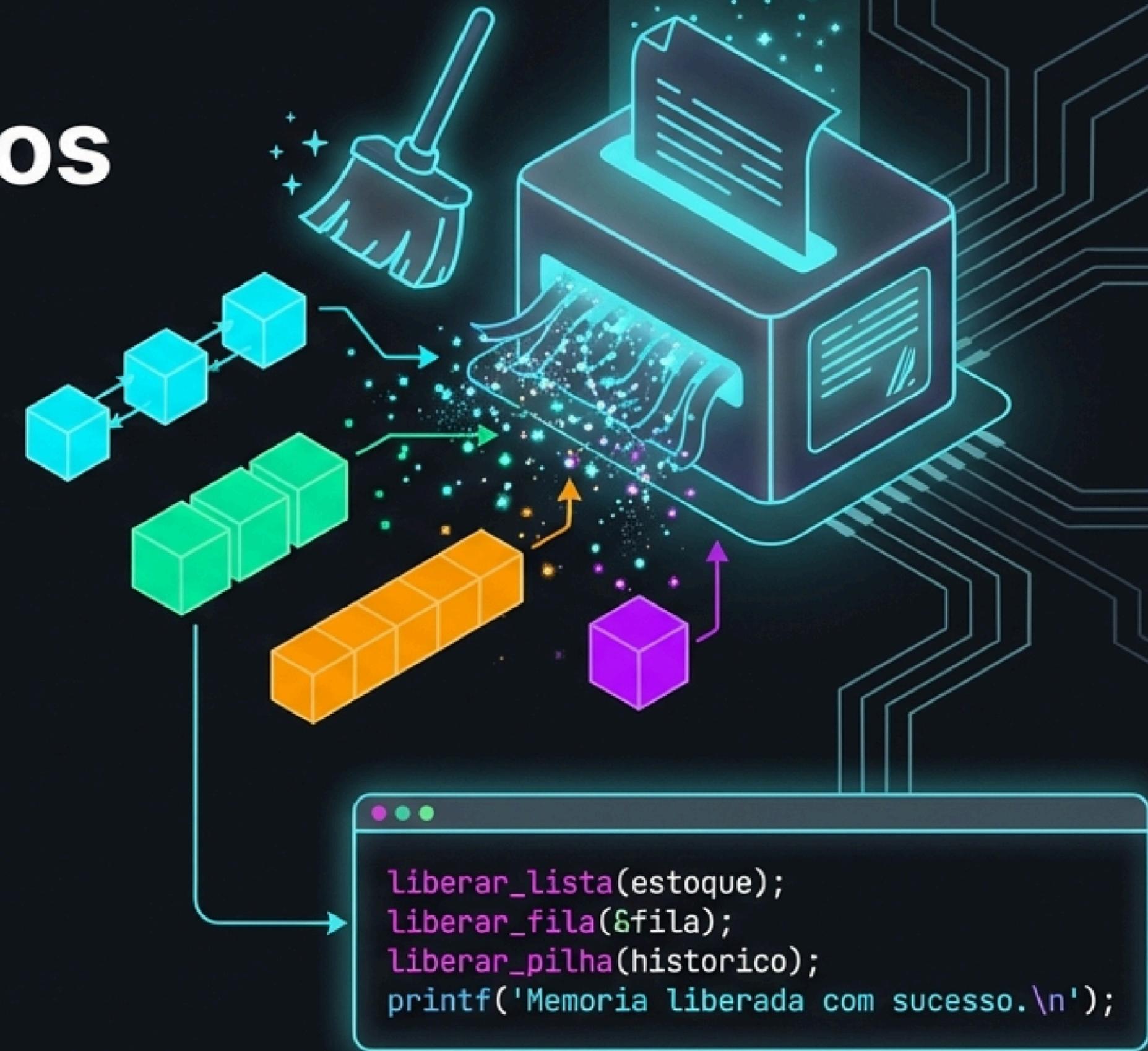
Quando opcao é 0, o loop do-while finalmente encontra sua condição de parada. A loja fecha as portas para novos clientes e inicia o procedimento de saída.

```
case 0:  
    printf('Encerrando sistema...\n');  
    // Funções de limpeza aqui  
    break;  
...  
} while(opcao != 0);
```



# A Limpeza Final: Evitando Vazamentos

Um bom programador sempre limpa a bagunça. Chamamos as funções liberar para devolver a memória RAM (mallocs) ao sistema operacional, evitando 'Memory Leaks'.



# Return 0: Missão Cumprida

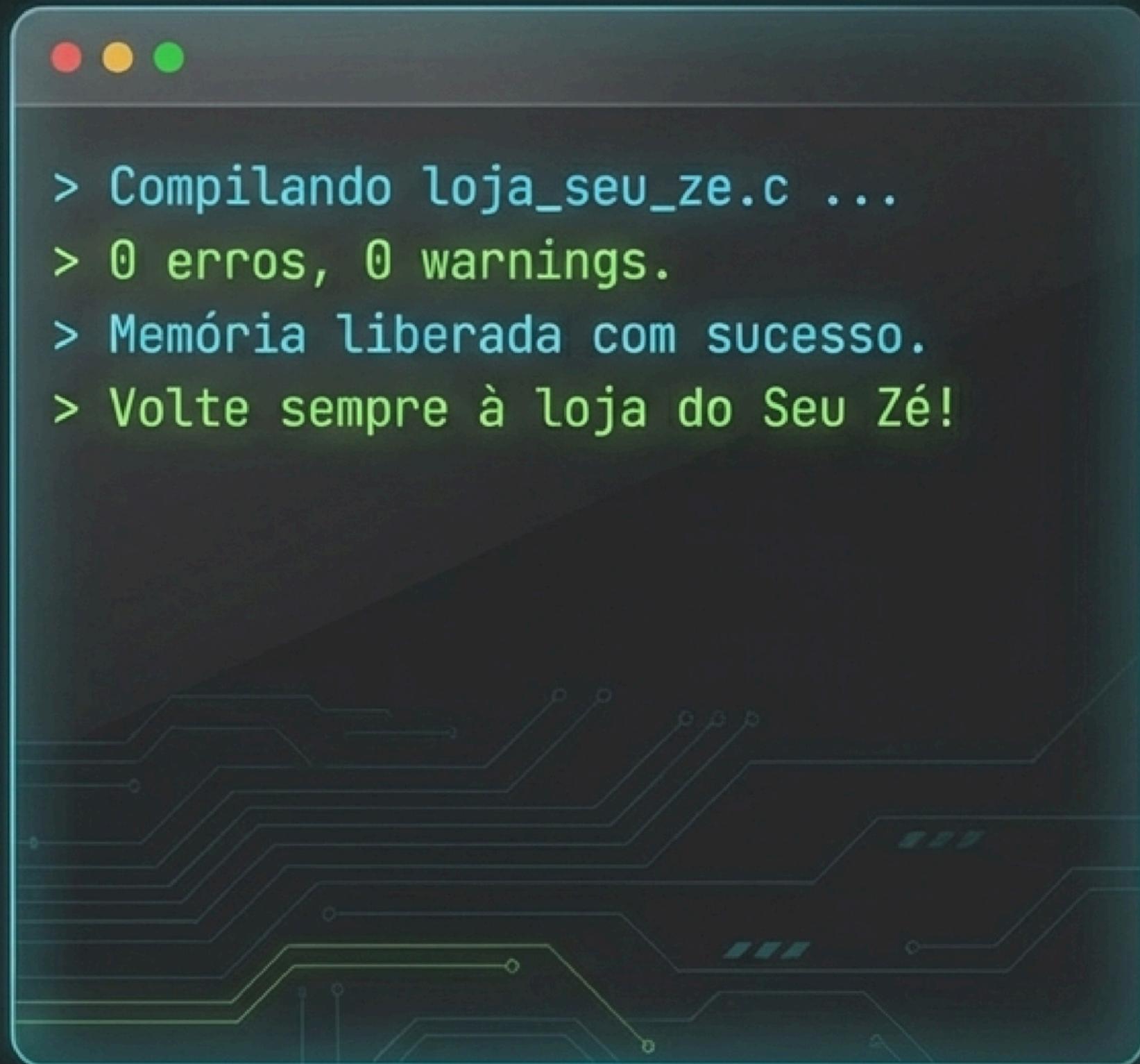
O return 0 é a mensagem final do Seu Zé para o Windows/Linux: "Tudo correu bem, terminamos sem erros". O programa é encerrado.

```
> Compilando loja_seu_ze.c ...
> 0 erros, 0 warnings.
> Memória liberada com sucesso.
> Volte sempre à loja do Seu Zé!
```

**EXIT\_SUCCESS**

```
return 0;
} // Fim do main
```

# Compilação Concluída



Entender estruturas de dados é entender organização. O código em C é apenas a ferramenta que torna essa lógica possível.

# Arquitetura Completa

O main não armazena dados complexos, ele gerencia fluxos. Ele conecta a intenção do usuário (Menu) com as estruturas de dados especializadas (Structs). Entender o main é entender como as peças se encaixam.

