Estruturas de dados: conceitos e representação

Você vai aprender os conceitos fundamentais de estruturas de dados, ponteiros, alocação dinâmica de memória e modularização na linguagem C. Para consolidar esses conhecimentos, você será desafiado a criar a base lógica do clássico jogo WAR, aplicando na prática os conceitos estudados, essenciais para o desenvolvimento de sistemas eficientes e bem estruturados.

Curadoria de TI

1. Itens iniciais

Objetivos

- Desenvolver um programa em C que implemente um sistema em linguagem C para cadastro dos dados (nome, cor e tropas) para estruturar as informações básicas do jogo War utilizando um vetor de structs.

- Implementar a funcionalidade de ataque entre territórios em linguagem C, utilizando alocação dinâmica e ponteiros, para simular batalhas e aproximação do sistema do comportamento estratégico do jogo real.

- Implementar o sistema de missões estratégicas em linguagem C, utilizando ponteiros e alocação dinâmica, para verificar a condição de vitória e conclusão do desenvolvimento do jogo.

Introdução

Boas-vindas ao desafio de programação, em que você construirá uma versão digital do clássico jogo War utilizando a linguagem C!

Imagine: uma empresa inovadora e especializada em jogos estratégicos contratou você para integrar a equipe responsável por transformar esse jogo icônico em uma experiência interativa e moderna. Seu objetivo principal é construir a base lógica do jogo utilizando estruturas de dados, modularização, ponteiros e técnicas de alocação dinâmica de memória para conquistar a Ásia e a América do Sul. Desafiador, não é mesmo?

No mercado de trabalho na área de tecnologia, dominar estruturas de dados, modularização e gerenciamento de memória é essencial para o desenvolvimento de sistemas, jogos, aplicações e soluções que exigem performance e controle sobre os recursos computacionais. Portanto, o desafio lançado vai além da teoria: ele te prepara para a prática profissional com um projeto divertido, estratégico e desafiador.

Neste vídeo, você acompanhará a jornada prática de desenvolvimento do jogo War em linguagem C, aplicando conceitos essenciais como estruturas de dados, uso de structs, modularização, ponteiros, alocação dinâmica de memória e passagem de parâmetros por valor e referência. Dividido em três níveis — novato, aventureiro e mestre —, o desafio simula um projeto real de desenvolvimento, no qual você criará o mapa do jogo, programará a lógica de ataque entre territórios e implementará missões estratégicas. Ao final, você terá construído um sistema modular, escalável e alinhado às exigências do mercado de TI.

> Conteúdo interativo
>
> Acesse a versão digital para assistir ao vídeo.

Antes de navegar e avançar pelos níveis do desafio, você conhecerá o cenário onde tudo acontece e descobrirá qual será a sua missão. Prepare-se!

Cenário

A TechNova, empresa de desenvolvimento de jogos, visionária no mercado de entretenimento digital, decidiu lançar a nova edição do War — agora com muito mais estratégia e controle sobre os recursos computacionais. Contudo, a nova versão exige que a lógica por trás do domínio de territórios seja sólida, escalável e eficiente. Para isso, será necessário implementar o gerenciamento de territórios, ataques, missões e condições de vitória com base em conceitos fundamentais de estrutura de dados e usando a linguagem C.

Sua missão

Você fará parte dessa missão e terá que lidar com desafios similares aos encontrados no desenvolvimento real de softwares: estruturação de dados, organização de código, controle de memória e escalabilidade de funções.

A nova versão do jogo terá uma visão estruturada e sistemática, com foco na organização e manipulação de dados. O projeto deverá contemplar as seguintes funcionalidades:

- Construir a base do jogo com o uso de structs para representar os territórios. Nesse momento, o objetivo será entender como representar e organizar informações em memória com estruturas lineares e não lineares, iniciando o mapeamento dos elementos do jogo de forma lógica e estruturada.
- Implementar a lógica de ataque entre territórios utilizando ponteiros e alocação dinâmica de memória com funções como malloc, calloc e free. Você também explorará ponteiros para funções e começará a modularizar o código para torná-lo mais organizado, reutilizável e pronto para expansão.
- Evoluir para o desenvolvimento para a implementação das missões estratégicas, verificações de condição de vitória e o refinamento da estrutura do código com modularização profissional, passagem de parâmetros por valor e referência, encapsulamento e uso de tipos abstratos de dados. A experiência simulará um desenvolvimento de software robusto e completo, com foco em performance e manutenibilidade.

Chegou a vez de mostrar suas habilidades. Prepare sua lógica, organize a estratégia e lidere suas estruturas rumo à vitória.

Boa sorte na guerra da programação!

Conhecendo as estruturas de dados

Veja, neste vídeo, o que são estruturas de dados, sua importância na programação e como aplicá-las no desenvolvimento de sistemas mais eficientes. Com analogias interessantes, acompanhe os tipos de estruturas de dados e como elas são usadas em aplicativos reais e em diferentes contextos computacionais.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Uma estrutura de dados é um modelo utilizado para organizar, armazenar e manipular informações em um programa de forma eficiente. Ela define o formato e a maneira como os dados são organizados na memória, permitindo acesso rápido e uso otimizado dos recursos computacionais. Lembre-se: o uso correto de estruturas de dados é o que permite que programas resolvam problemas complexos com rapidez e clareza.

Vamos entender melhor! Imagine que você está organizando sua despensa de casa. Você tem pacotes de arroz, feijão, macarrão, além de enlatados e temperos. Se você só colocar tudo em qualquer lugar, vai demorar mais para encontrar quando precisar. Mas se você organizar por categoria, tamanho ou data de validade, tudo fica mais fácil e rápido de encontrar.

A lógica da organização vale também para a programação. Quando criamos um programa, precisamos armazenar, organizar e acessar dados com eficiência — e, para isso, usamos estruturas de dados.

Atenção

Estrutura de dados é uma forma de organizar os dados na memória do computador para que possam ser utilizados de maneira eficiente. Ela define como os dados são armazenados, acessados e manipulados. Em outras palavras: em vez de pensar em dados soltos, como números ou palavras espalhadas, você começa a agrupá-los e estruturá-los, como gavetas em um armário.

A importância das estruturas de dados está em quase tudo que usamos no mundo digital. Aplicativos de redes sociais, sistemas bancários, jogos, sites de compras usam alguma forma de estrutura de dados para funcionar corretamente. Confira na imagem!

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.

Aplicativos de redes sociais, sistemas bancários, jogos e sites de compras vistos a partir das estruturas de dados.

Imagine um aplicativo que mostra os contatos do seu telefone. Eles estão organizados em uma lista com nome, número e, às vezes, até foto. Essa lista é uma estrutura de dados.

Vamos ao exemplo!

Um exemplo comum é a lista de compras, como apresentado na próxima imagem. Ela pode ser feita em papel ou no celular, mas, em ambos os casos, você está organizando itens que deseja adquirir. Você pode usar essa lista para incluir novos itens, retirar outros e até reorganizar a ordem. Isso é exatamente o que estruturas de dados permitem: inserir, remover, buscar e reorganizar informações.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.

Aplicativo de lista de compras.

Tipos de estruturas de dados

São lineares ou não lineares, e isso dependendo do tipo do dado. Veja a seguir cada um dos tipos em detalhes.

Estruturas de dados lineares

Organizam os elementos em sequência, formando uma linha lógica contínua. Cada elemento (com exceção do primeiro e do último) possui exatamente um antecessor e um sucessor, facilitando o acesso sequencial. Vetores (arrays), listas ligadas, pilhas (stacks) e filas (queues) são exemplos clássicos de estruturas lineares. Estruturas lineares são muito utilizadas em tarefas que exigem ordenação ou processamento em série, como manipulação de listas de usuários, execução de tarefas em ordem cronológica ou controle de operações empilhadas em programas.

Estruturas de dados não lineares

Organizam os elementos de forma hierárquica (como organogramas ou sistemas de arquivos) ou em rede complexas (como mapas, redes sociais ou sistemas de transporte), permitindo múltiplas conexões entre os dados. Ao contrário das estruturas lineares, nas não lineares não há uma sequência única de acesso: os elementos podem ter mais de um relacionamento, como ocorre em árvores e grafos. Elas oferecem maior flexibilidade e são fundamentais para resolver problemas mais sofisticados envolvendo caminhos, decisões e conexões.

Tipos de estruturas de dados – lineares

Neste vídeo, aprenda o que são estruturas de dados lineares, como elas funcionam e onde são aplicadas. Com analogias do cotidiano, confira os principais tipos de estruturas lineares e como elas organizam os dados em sequência e facilitam o armazenamento, a busca e o processamento de informações em sistemas reais.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

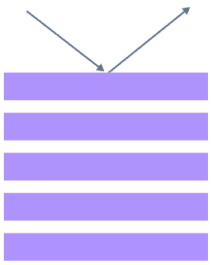
As estruturas de dados lineares organizam os elementos de forma sequencial, isto é, um após o outro em uma única linha lógica. Cada elemento (exceto o primeiro e o último) possui exatamente um antecessor e um sucessor. Isso significa que o acesso aos dados segue uma ordem bem definida.

> ⚠
>
> Atenção
>
> Estrutura de dados lineares são ideais para representar listas, pilhas, filas e outras coleções em que a ordem de entrada, leitura ou remoção dos elementos é importante.

Vamos entender melhor imaginando uma fila no caixa do supermercado. As pessoas entram por um lado e saem pelo outro, respeitando a ordem de chegada.

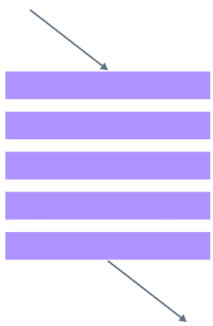
Fila no caixa do supermercado.

Pense agora em uma pilha de pratos limpos: o colocado no alto será o primeiro a ser retirado, certo? Ambas as situações representam duas estruturas de dados lineares muito conhecidas: pilha (stack) e fila (queue), conforme as imagens a seguir.



Pilha

Último que entra, primeiro que sai.



Fila

Primeiro que entra, primeiro que sai.

Trata-se de estruturas fáceis de implementar e entender. Elas aparecem em diversas tarefas, como armazenar nomes em uma lista, processar tarefas em sequência ou empilhar operações a serem desfeitas.

Principais tipos de estruturas de dados lineares

São divididos em:

Vetores (arrays)

Armazenam dados em posições consecutivas de memória. São como caixas numeradas, em que cada uma guarda um valor.

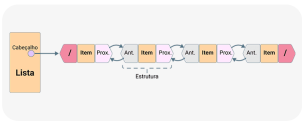
Exemplo: tabela de notas de alunos em um sistema acadêmico. A posição de cada vetor representa a nota de um aluno específico.

Confira a imagem!

Listas lineares (listas ligadas)

São semelhantes a vetores, mas cada elemento aponta para o próximo, como um colar de contas.

Exemplo: sistema de histórico de navegação em um navegador de internet, em que cada página visitada é ligada à próxima.



Pilhas (stacks)

Seguem a lógica LIFO (Last In, First Out), ou seja, o último a entrar é o primeiro a sair.

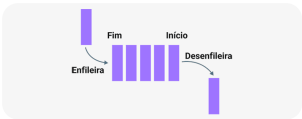
Exemplo: controle de desfazer/refazer (undo/redo) em editores de texto ou imagem.

Entenda melhor na imagem a seguir.

Filas (Queues)

Seguem a lógica FIFO (First In First Out), em que o primeiro a entrar é o primeiro a sair.

Exemplo: gestão de chamadas em um call center, onde os atendimentos são realizados na ordem de chegada.



LIFO X FIFO

O conceito de **LIFO (Last In First Out)** é usado em estruturas de dados chamadas pilhas (stacks). Nessa lógica, o último elemento que foi inserido é o primeiro a ser removido.

As estruturas de pilhas (stacks) são ideais para situações em que precisamos voltar ou retomar ações recentes, como no sistema de desfazer/refazer de editores de texto, ou para controle de chamadas de funções em programas, em que a execução da função mais recente deve ser finalizada antes de retornar à anterior.

Já o conceito de **FIFO (First In First Out)** é aplicado em estruturas chamadas filas (queues). Nesse modelo, o primeiro elemento a entrar é o primeiro a sair.

Exemplo

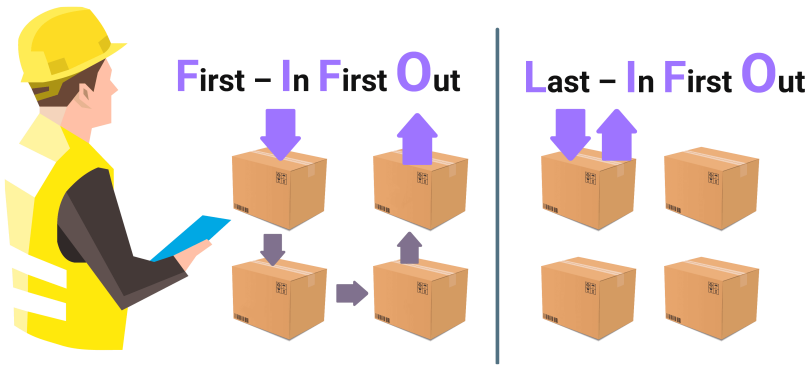
Visualize o que ocorre em uma fila de banco ou de ônibus: quem chega primeiro é logo atendido.

Filas são muito utilizadas em sistemas que exigem ordem de processamento, como impressão de documentos (spooler), filas de atendimento ao cliente ou gerenciamento de tarefas em servidores e sistemas operacionais.

A imagem a seguir ilustra muito bem a diferença entre os dois conceitos que acabamos de estudar. Confira!

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



FIFO X LIFO.

Tipos de estruturas de dados – não lineares

Neste vídeo, explore as estruturas de dados não lineares, que organizam os elementos de forma hierárquica ou em rede, permitindo conexões múltiplas. Confira também os conceitos de árvores e grafos e as aplicações reais dessas estruturas, destacando sua importância em situações que exigem acesso flexível e dinâmico às informações.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Nas estruturas de dados não lineares, os elementos não estão organizados de maneira sequencial. Mas como assim? Em outras palavras, nelas, cada elemento pode estar relacionado a vários outros, formando conexões que se assemelham a ramos de uma árvore ou a nós interligados em uma rede.

Atenção

A estrutura de dados não linear é ideal quando os dados precisam representar relações hierárquicas ou com múltiplas conexões, permitindo maior flexibilidade e complexidade nas operações.

Diferentemente de uma lista ou fila, em que os elementos seguem uma ordem definida de início, meio e fim, as estruturas não-lineares permitem caminhos múltiplos e acessos em diferentes direções e ramificações. Isso as torna muito úteis em situações como:

Hierarquia organizacional

Representar a hierarquia de uma organização (presidente → diretores → gerentes).

Conexões em redes

Modelar as conexões entre páginas da web ou estações de transporte.

Jogos com decisões

Criar jogos com múltiplos caminhos ou tomadas de decisão.

Busca e navegação

Desenvolver mecanismos de busca e algoritmos de navegação.

Para entender melhor, imagine o mapa do metrô. Cada estação está conectada a várias outras. Você pode ir da estação A para B, depois para C, ou direto de A para D. Observe adiante!

Mapa de metrô de Paris.

O tipo de estrutura de conexões múltiplas não pode ser representado de forma sequencial. É aí que entram as estruturas de dados não lineares.

Principais tipos de estruturas de dados não lineares

Os mais usados são:

Árvores

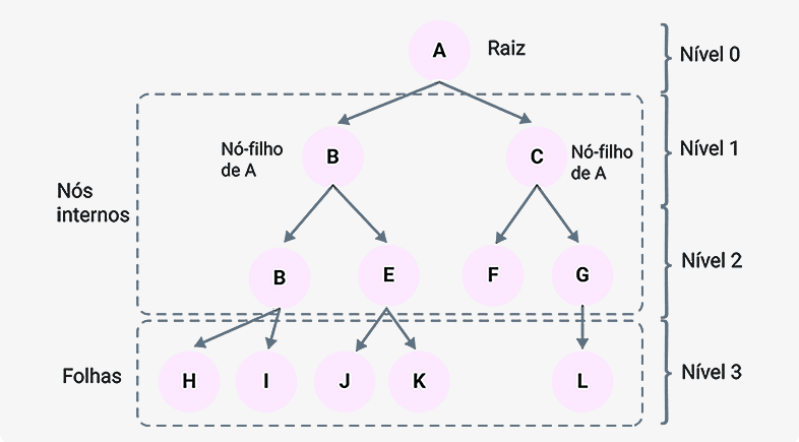
Um bom jeito de explicar esse tipo é pensando que cada elemento (nó) pode ter vários “filhos”, mas apenas um “pai”. Um exemplo clássico é a estrutura de pastas do seu computador: existe uma principal e várias outras dentro dela.

Outro exemplo é a organização da empresa, em que o presidente comanda diretores, que comandam gerentes.

A imagem a seguir nos ajuda a entender por que esse tipo é chamado de árvore: é só pensar em uma árvore genealógica. Veja!

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Árvore.

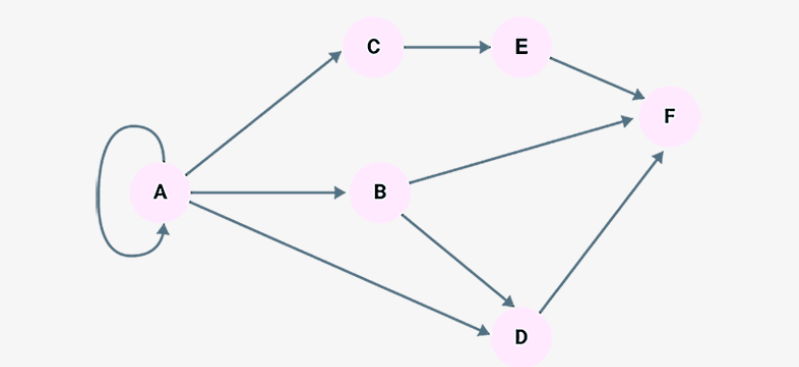
Grafos

São estruturas em que cada nó pode se conectar a vários outros, formando redes — como estradas, redes sociais ou roteadores de internet.

Exemplo: mapa com cidades e estradas conectando-as.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Grafo.

Uso de structs e encapsulamento de dados

Neste vídeo, veja o que é uma struct na linguagem C e como ela permite organizar diferentes tipos de dados relacionados em um único bloco, facilitando o armazenamento e a manipulação de informações. Acompanhe ainda como structs ajudam a criar programas mais organizados, reutilizáveis e escaláveis.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Na programação, lidar com muitos dados separados pode se tornar confuso e propenso a erros. Para resolver esse problema, usamos uma estrutura chamada **struct** em C.

Uma struct permite agrupar diferentes informações (de tipos variados) sobre um mesmo objeto. Mas o que é exatamente uma struct? Também chamada de estrutura, é um tipo de dado composto que permite agrupar diferentes variáveis sob um mesmo nome, facilitando a organização de informações relacionadas. Cada variável em uma struct é chamada de **membro** ou campo, e pode ter um tipo diferente.

Por exemplo, uma estrutura que representa um aluno pode agrupar seu nome, idade e média em um único bloco de dados. Mas

- O campo nome é do tipo char.
- O campo idade é do tipo int.
- O campo media é do tipo float.

Todos os três são campos são tipos de dado diferentes. Acompanhe!

```c
struct Aluno {
    char nome[50];
    int idade;
    float media;
};
```

Com isso, podemos declarar um aluno com:

```c
struct Aluno a1;
```

e acessar seus campos com:

```c
a1.idade = 20;
```

Esse agrupamento é chamado de **encapsulamento de dados**.

Encapsular significa esconder ou proteger detalhes internos, expondo apenas o necessário. Com isso, o código se torna mais organizado, legível e reutilizável. Esse conceito é bastante usado em linguagens orientadas a objetos, mas em C ele já pode ser praticado com o uso de structs e boas práticas de modularização.

Para entendermos melhor, usaremos alguns exemplos. Vamos lá!

Prontuário médico

Um médio anota em seu prontuário todas as informações do mesmo paciente, como nome, idade, tipo sanguíneo, alergias e histórico de doenças, entre outros. Mas imagine ter essas informações espalhadas em folhas soltas: seria difícil controlar, correlacionar e manter consistência, certo?

Portanto, esse cenário nos leva a concluir que o prontuário é a struct que encapsula os dados.

Caixa de ferramentas

Você poderia guardar o martelo em um armário, os pregos em uma gaveta e a chave inglesa em outro canto. Mas é mais prático guardar tudo em um único lugar, como uma caixa de ferramentas.

Assim, a caixa é a struct, e as ferramentas são os atributos encapsulados nela.

Benefícios do encapsulamento com structs

Existem alguns benefícios no conceito de encapsulamento aplicado à structs. São eles:

Menos repetições

Reduz repetições no código.

Dados organizados

Organiza a lógica dos dados.

Reutilização facilitada

Facilidade para reutilizar estruturas em outras partes do programa.

Listas de estruturas

Permite criar vetores ou listas de objetos estruturados.

Exemplo prático em C

Vamos usar typedef para facilitar o uso da struct e criar um aluno com seus dados.

Usamos typedef na criação de uma struct em C para simplificar a sua utilização, tornando o código mais limpo e legível. Sem typedef, toda vez que quisermos declarar uma variável do tipo struct, precisaremos repetir a palavra-chave struct antes do nome da estrutura. Contudo, com typedef, podemos criar um apelido (alias) para o tipo, eliminando essa necessidade, como podemos ver no código a seguir.

```
c

#include

// Definindo a estrutura do tipo Aluno
typedef struct {
    char nome[50];
    int idade;
    float media;
} Aluno;

int main() {
    // Criando uma variável do tipo Aluno
    Aluno aluno1 = {"João", 20, 8.5};

    // Exibindo os dados do aluno
    printf("Aluno: %s
Idade: %d
Média: %.2f
", aluno1.nome, aluno1.idade, aluno1.media);
    return 0;
}
```

Hora de codar: sistema de biblioteca

Nesta prática, você aplicará os conhecimentos adquiridos no nível novato de estrutura de dados para implementar um sistema de cadastro de livros da biblioteca em linguagem C.

O objetivo é criar uma estrutura (struct) que armazene informações como nome, autor, editora e edição de cada livro. Cada campo da struct deve ter um tipo de dados definido. O sistema solicitará ao usuário que cadastre os dados de vários livros por meio do terminal e, em seguida, exibirá as informações registradas.

No vídeo a seguir, acompanhe a implementação de um sistema de cadastro de livros em linguagem C, utilizando structs para organizar os dados de cada obra — como nome, autor, editora e edição. A prática reforça o uso de estruturas de dados compostas, entrada de dados via terminal e exibição organizada das informações, aplicando na prática os conceitos estudados no nível novato.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Desafio: nível novato

Com este vídeo, dê os primeiros passos na criação do jogo War em linguagem C, implementando um sistema de cadastro de territórios com uso de structs. Para isso, acompanhe os conceitos abordados que reforçam a prática de vetores de structs, entrada de dados via terminal e organização lógica em programação estruturada. Lembre-se: este é o início que evoluirá nos próximos níveis do projeto War estruturado.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Construção dos territórios

Sua missão é construir uma base de dados de territórios utilizando uma estrutura de dados composta.

O que você vai fazer?

Criar uma struct chamada Territorio que armazenará informações como nome, cor do exército e quantidade de tropas. O sistema permitirá o cadastro de 5 territórios e exibirá seus dados logo após o preenchimento.

Requisitos funcionais

Veja os passos para criar e manipular uma estrutura que representa territórios em um sistema simples.

- **Criação da struct:** definir uma struct chamada Territorio com os campos char nome[30], char cor[10] e int tropas.
- **Cadastro dos territórios:** o sistema deve permitir que o usuário cadastre cinco territórios informando o nome, cor do exército e o número de tropas de cada um.
- **Exibição dos dados:** o sistema deve exibir as informações de todos os territórios registrados após o cadastro.

Requisitos não funcionais

Inclua também os seguintes critérios para ter um código funcional, eficiente e fácil de entender:

- **Usabilidade:** a interface de entrada deve ser simples e clara, com mensagens que orientem o usuário sobre o que digitar.
- **Desempenho:** o sistema deve apresentar os dados logo após o cadastro, com tempo de resposta inferior a 2 segundos.
- **Documentação:** o código deve conter comentários explicativos sobre a criação da struct, entrada e exibição de dados.
- **Manutenibilidade:** os nomes das variáveis e funções devem ser claros e representativos, facilitando a leitura e manutenção do código.

Instruções detalhadas

Siga os passos a seguir para implementar o programa de cadastro de territórios:

- **Bibliotecas necessárias:** inclua as bibliotecas stdio.h e string.h.
- **Definição da struct:** crie a struct Territorio com os campos mencionados.
- **Declaração de vetor de structs:** crie um vetor com capacidade para armazenar 5 estruturas do tipo Territorio.
- **Entrada dos dados:** utilize um laço for para preencher os dados dos 5 territórios.
- **Exibição:** percorra, após o cadastro, o vetor e exiba os dados de cada território com formatação clara.

Requisitos técnicos adicionais

Aqui estão algumas orientações adicionais para uma implementação correta e bem documentada:

- Use scanf para ler o nome e o número de tropas.
- Utilize fgets ou scanf("%s", ...) com cuidado para strings.
- Comente seu código explicando a criação e o uso da struct e a lógica do laço de entrada e saída.

Comentários adicionais

Este desafio introduz o conceito de structs como ferramenta para agrupar dados relacionados. Assim, ao final, você entenderá como utilizar estruturas compostas para organizar informações e criar sistemas mais legíveis e escaláveis.

Entregando seu projeto

Vamos ao passo a passo:

1. **Desenvolva seu projeto no GitHub:** continue usando o mesmo repositório do GitHub dos níveis anteriores.

2. **Atualize o arquivo do seu código: atualize** o arquivo super_trunfo.c com o código completo, incluindo as novas funcionalidades.

3. **Compile e teste:** faça isso rigorosamente, de modo que todas as comparações e cálculos estejam corretos.

4. **Faça commit e push:** faça commit das suas alterações e envie (push) para o seu repositório no GitHub.

5. **Envie o link do repositório:** envie o link do seu repositório no GitHub por meio da plataforma SAVA.

   Para concluir o desafio, é necessário que você faça o commit no GitHub do que você desenvolveu. Além disso, certifique-se de organizar e documentar de maneira adequada seu repositório. Essa etapa demonstra seu aprendizado.

   Por fim, cadastre na Guia de trabalhos da SAVA o link do seu repositório.

   Atenção! Caso você atualize o seu repositório, não é necessário alterá-lo na SAVA.

   Por último (mas não menos importante), confira o tutorial!

Tutorial git

Você está prestes a aplicar os conceitos aprendidos para resolver um desafio prático no ambiente do GitHub. Veja as instruções gerais a seguir para acessar, aceitar e executar o desafio, a fim que sua solução esteja bem estruturada e documentada.

Dê o primeiro passo

Acesse o GitHub Classroom. Nesse ambiente, você terá acesso ao repositório padrão do desafio. Caso ainda não tenha uma conta no GitHub, não se preocupe: você pode criar uma gratuitamente, clicando no link.

Aceite o desafio

Tendo acesso ao repositório no GitHub, você, então, encontrará na plataforma o repositório criado para o desenvolvimento do seu desafio.

Acesse o repositório

Clique no link do repositório para abrir o ambiente GitHub com a descrição do desafio e a estrutura modelo de arquivos e pastas que devem ser utilizadas. Lembre-se: é esse o link que você deve enviar no SAVA.

Explore a estrutura do ambiente

Veja a estrutura organizada de pastas e arquivos necessários para o desenvolvimento do desafio. Você encontrará isso no ambiente do GitHub.

Desenvolva o desafio

Utilize o GitHub CodeSpace para editar o arquivo do código-fonte e desenvolver o desafio. Certifique-se de que o código esteja organizado e funcional para resolver o problema proposto.

Entregue o desafio

Forneça o repositório do GitHub com todos os arquivos de código-fonte e conteúdos relacionados ao projeto. Certifique-se de que o repositório esteja bem estruturado, com pastas e arquivos nomeados de maneira clara e coerente. Envie o link para o repositório do seu desafio no GitHub.

Para finalizar, comente todos os arquivos de código-fonte. Os comentários são indispensáveis para demonstrar seu entendimento sobre o funcionamento do código e facilitar a correção. Além disso, eles devem explicar a finalidade das principais seções do código, o funcionamento de algoritmos complexos e o propósito de variáveis e funções utilizadas.

Confira agora o vídeo com dicas e detalhes sobre a entrega do seu projeto no GitHub.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Conceito e manipulação de ponteiros

Neste vídeo, aprenda o conceito de ponteiros na linguagem C, entendendo como eles armazenam o endereço de memória e permitem acessar ou modificar os dados diretamente. Veja também por que e quando utilizar ponteiros em situações como passagem por referência, manipulação de arrays e controle de memória.

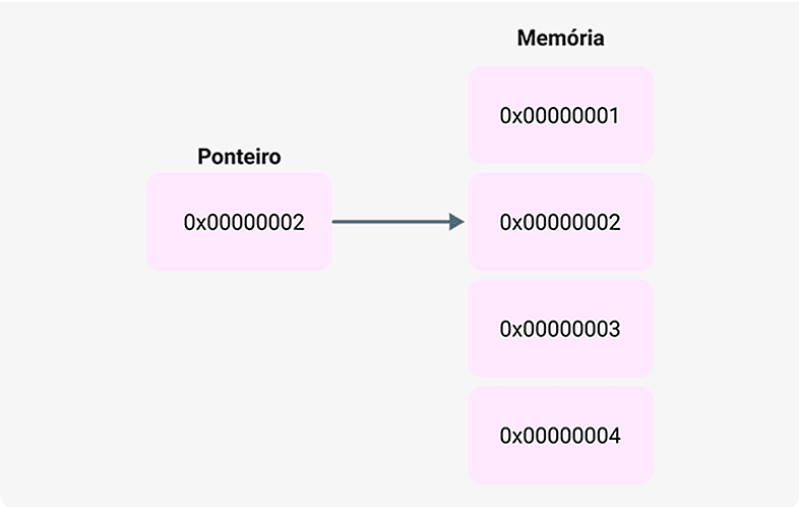
Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Na linguagem C, um ponteiro é uma variável especial que armazena o endereço de memória de outra variável. No entanto, em vez de guardar um valor, como o número 10 ou a letra A, o ponteiro guarda onde esse valor está localizado na memória do computador. Com isso, ele permite que você acesse, modifique e gerencie dados de maneira muito mais flexível e poderosa.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Ponteiro.

Como representar?

Em C, usamos o símbolo * para declarar um ponteiro, e o símbolo & para obter o endereço de uma variável.

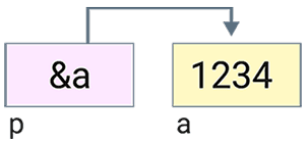
Por exemplo:

```c
int a;
int *p;
p = &a;  // 'p' agora guarda o endereço de 'x'
```

Observe na imagem a seguir a representação do que consta no código visto:

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



O código também poderia ter sido inicializado assim:

```c
int *p = &a;
```

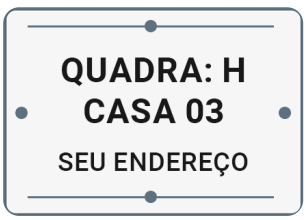
p é um ponteiro para inteiro que aponta para a variável x.

Vamos entender melhor usando alguns exemplos!

Endereço de uma casa

Não é a casa em si, mas indica onde ela está. Logo, com o endereço, você pode ir à casa, ver o que tem dentro ou até fazer reformas. Um ponteiro funciona assim: ele não é o valor, mas sabe onde ele está e pode acessá-lo ou modificá-lo.

Imagine que você quer visitar um amigo. Você não precisa levar a casa do seu amigo até você; basta que ele lhe envie o endereço. Assim, você pode ir lá, bater na porta e entregar algo ou fazer uma visita (e quem sabe, toma até um café). O ponteiro faz exatamente isso com as variáveis: ele não é o valor, mas leva você até onde o valor está guardado.



Cartão de endereço

Pense em um cartão com o número de uma caixa de correio, conforme a imagem.

O cartão não tem a encomenda, mas indica onde ela está. Com esse número (o endereço), você vai direto ao local certo e pega o pacote. Da mesma forma, o ponteiro aponta onde o dado está.

Exemplo prático em C

p é um ponteiro para x. O operador & obtém o endereço da variável, e o operador $*$ acessa o conteúdo armazenado no endereço.

```
c

#include

int main() {
    int x = 10;
    int* p = &x; // ponteiro para x

    printf("Valor de x: %d\n", x);
    printf("Endereço de x: %p\n", &x);
    printf("Conteúdo de p: %p\n", p);
    printf("Valor apontado por p: %d\n", *p);

    return 0;
}
```

Vamos entender melhor o código!

Inicialmente, declara-se uma variável inteira x com o valor 10. Em seguida, cria-se um ponteiro p que armazena o endereço de memória da variável x. O programa imprime o valor de x, o endereço de memória onde ele está armazenado (&x), o conteúdo do ponteiro p (que é esse mesmo endereço) e, por fim, o valor apontado por p, usando o operador *, que acessa o conteúdo do endereço.

Manipulando valores com ponteiros

Podemos modificar o valor da variável original usando o ponteiro:

```
c

*p = 20;
```

Tal linha modifica diretamente o valor de x, porque p aponta para ele. Pensando no exemplo da caixa do correio, é como se você atualizasse o conteúdo da caixa através do número dela.

Quando e por que usar ponteiros?

O uso de ponteiros em C é adequado nas seguintes situações:

- Modificar variáveis dentro de funções sem retornar valores (passagem por referência).
- Manipular arrays e strings de forma eficiente.
- Construir estruturas complexas, como listas, árvores e grafos.
- Gerenciar a memória de modo dinâmico.

Alocação e desalocação dinâmica de memória (malloc, calloc e free)

Neste vídeo, confira a alocação dinâmica de memória na linguagem C, entendendo quando e por que usar malloc, calloc e free. Veja ainda exemplos práticos de alocação e liberação de memória com ponteiros, além de erros comuns e como evitá-los.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

A memória de um computador é o local onde os dados e as instruções de um programa são armazenados por um tempo durante sua execução. Ao programar em C, precisamos entender como a memória é utilizada, alocada e liberada para que o uso dos recursos do sistema seja eficiente.

Podemos considerar dois tipos de alocação de memória:

Alocação estática ou automática

É o tipo que ocorre quando criamos uma variável comum, como $\text{int } x = 10$. Nesse caso, o compilador reserva de maneira automática um espaço fixo na memória. Essa alocação acontece durante a compilação ou no início da execução do programa, e a quantidade de memória é previamente conhecida.

Alocação dinâmica de memória

É necessária quando não sabemos com antecedência quanto espaço será preciso. Por exemplo, ao criar um sistema que leia uma quantidade de dados definida pelo usuário em tempo de execução. Nesse caso, o programa solicita memória ao sistema operacional durante a execução e é responsável por liberá-la quando não for mais necessária.

Em C, esse controle é feito de forma manual com três funções principais: malloc(), calloc() e free().

Vamos entender melhor! Imagine que você vai a um restaurante self-service. Em vez de receber um prato padrão, você escolhe o tamanho e os itens conforme a sua fome. A alocação dinâmica é assim: o programa “pede” para o sistema exatamente a quantidade de memória que precisa usar.

Alocação e desalocação dinâmica: principais funções

Iremos conhecer agora as principais funções usadas em alocação e desalocação dinâmica na linguagem C!

malloc() – memory allocation

Aloca uma quantidade de memória em bytes, mas não inicializa.

A função malloc aloca um bloco de memória de tamanho específico (em bytes) e retorna um ponteiro para o início desse bloco. Os valores dentro desse espaço não são inicializados.

Suponha que você alugue um armário no shopping sem saber se ele está limpo ou bagunçado. Você só recebe o espaço e depois decide como usá-lo.

Exemplo:

```
c

int* vetor = (int*)malloc(5 * sizeof(int));
```

Estamos alocando espaço para um vetor de 5 inteiros. A função retorna um ponteiro para a primeira posição da memória reservada.

calloc() – clear allocation

Aloca e inicializa todos os bytes com zero.

A função calloc também aloca memória, mas inicializa todos os bytes com zero. Ela recebe dois parâmetros: a quantidade de elementos e o tamanho de cada um.

Para entendermos melhor, podemos pensar em alugar um armário e já o receber limpo e vazio. Ou seja, você não precisa se preocupar com sujeira ou bagunça anterior.

Exemplo:

```
c

int* vetor = (int*)calloc(5, sizeof(int));
```

No caso do calloc, os 5 inteiros do vetor já começam com valor zero.

free()

Libera a memória alocada para evitar desperdício.

Após utilizar a memória alocada dinamicamente, é preciso liberá-la com a função free para evitar desperdício de recursos.

Pensando ainda no exemplo do armário no shopping, podemos comparar essa função a devolver a chave do armário alugado. Se você não devolver, ele continua ocupado — mesmo que você não use mais.

Exemplo:

```
c

free(vetor);
```

Free libera o espaço ocupado por vetor, permitindo que o sistema reutilize essa memória.

Vamos ao exemplo prático em C!

Veja o uso de alocação dinâmica de memória com malloc e o acesso à memória usando ponteiros.

No código, visualizamos o uso das funções malloc e free, além do uso de ponteiros.

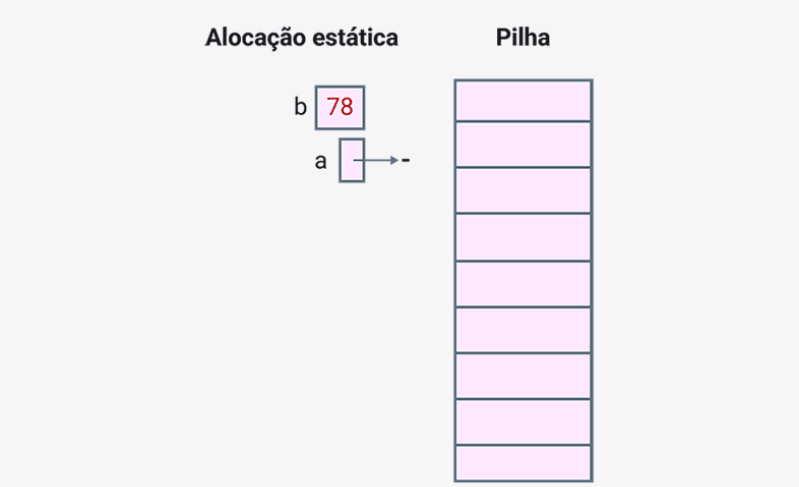
```
c

int  *a, b;
:
:
b = 10;
a = (int*) malloc(sizeof(int));
*a = 20;
a = &b;
free(a);
```

Ao executar a linha 1, "**a**" e "**b**" estarão em um espaço de endereçamento estático, e seu valor será aquele anteriormente armazenando na memória, conforme imagem a seguir.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.

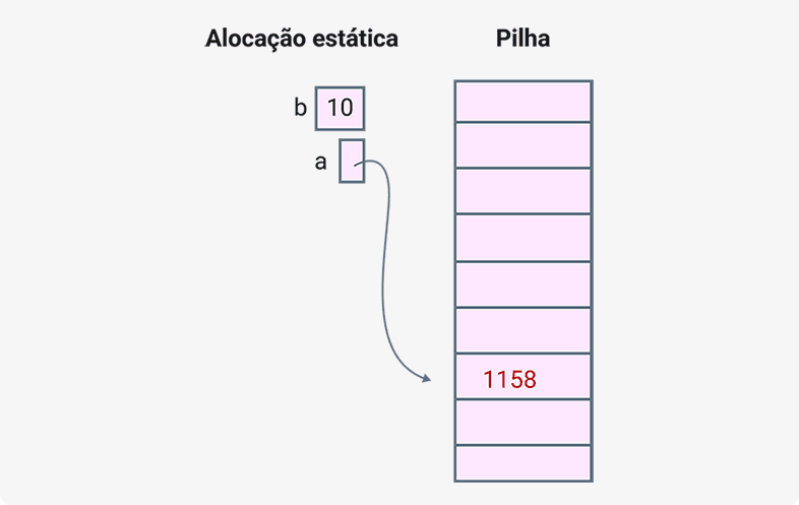


Variáveis a e b na memória após a execução da linha de código 1.

Ao executar as linhas 4 e 5 , o conteúdo de "**b**" passa a ser 10 e apontará para um endereço do espaço dinâmico, que conterá o valor armazenado anteriormente na memória, conforme imagem adiante.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.

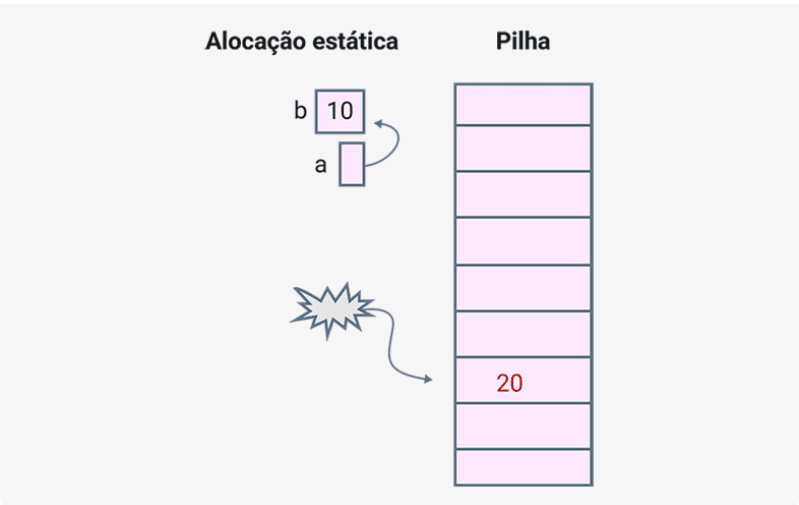


Variáveis a e b na memória após a execução das linhas de código 4 e 5 .

Ao executar a linha 6, o endereço que "**a**" aponta passa a armazenar o valor 20. Na linha 7, "**a**" passa a apontar para "**b**", mas agora ninguém referencia mais o endereço apontado antes por "**a**". A utilização da função free na linha 8 está incorreta. O objetivo seria liberar a memória que foi, a princípio, para "**a**". No entanto, "**a**" não aponta mais para esse endereço de memória, conforme a próxima imagem.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Variáveis a e b na memória após a execução das linhas de código 6.

Para corrigir esse problema, as linhas 7 e 8 deveriam ser invertidas. Primeiro, a memória alocada dinamicamente é desalocada, e depois "**a**" passa a apontar para "**b**".

Quando usar free?

Quando a quantidade de dados não é conhecida antes da execução e em estruturas como listas encadeadas, filas, árvores e vetores redimensionáveis. O uso consciente de malloc, calloc e free ajuda a evitar vazamentos de memória que podem deixar o sistema mais lento ou instável.

Ponteiros para funções e estruturas

Neste vídeo, explore como utilizar ponteiros para acessar funções e estruturas na linguagem C. Veja como eles permitem chamadas indiretas e como ponteiros para structs facilitam o uso de dados alocados dinamicamente. Com exemplos práticos, entenda como esses conceitos ampliam a flexibilidade e reutilização do código.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Antes de explorarmos os ponteiros aplicados a funções e estruturas, é importante compreendermos o que são esses dois elementos fundamentais em C.

Função

É um bloco de código nomeado que realiza uma tarefa específica e pode ser reutilizado ao longo do programa.

Estrutura (ou struct)

É uma forma de agrupar variáveis de diferentes tipos sob um mesmo nome, representando um único conceito — como um aluno, um livro ou um funcionário.

Ponteiros podem ser usados para referenciar tanto funções quanto estruturas, permitindo acesso indireto, flexível e eficiente.

Ponteiros para funções

Armazenam o endereço de uma função, possibilitando chamá-la de forma indireta. Isso é útil quando queremos criar programas mais genéricos, que mudam de comportamento dependendo da função associada a determinado momento, como em menus ou filtros.

Vamos entender melhor comparando com a ação de salvar o número de um contato no celular. Você não precisa discar o número toda vez — basta clicar no nome do contato e a chamada será feita. Portanto, o ponteiro armazena o caminho até a ação (função), e executá-la fica simples.

Os ponteiros para funções podem ser passados para funções, retornados de funções, armazenados em vetores (arrays) e atribuídos a outros ponteiros para funções.

Como representar um ponteiro para função em C?

Utilizamos uma sintaxe específica que indica que o ponteiro armazena o endereço de uma função, permitindo chamá-la indiretamente.

Agora, confira a forma geral da declaração em duas situações.

Funções sem parâmetros:

```
c

tipo_de_retorno (*nome_do_ponteiro)();
```

Funções que recebem argumentos (com parâmetros):

```c
(*nome_do_ponteiro)(declaração_de_parâmetro);
```

Por exemplo, se você quiser declarar um ponteiro para uma função que recebe dois int e retorna um float, deverá usar:

```c
float (*func_ptr)(int, int);
```

Trata-se de um tipo de construção útil quando se deseja passar funções como argumentos para outras funções, criar vetores de funções ou implementar **callbacks**, proporcionando maior flexibilidade e modularidade ao código.

Vamos ao exemplo prático em C

A variável ptr armazena o endereço da função saudacao, e a chamada ptr(); executa a função por meio do ponteiro. Acompanhe!

Callbacks

Um callback em C é um ponteiro para função que é fornecido a uma função ou biblioteca, permitindo que o código do usuário seja chamado ("call back") em determinado momento. A ideia central é permitir que uma função chame uma outra função cujo comportamento pode ser definido em tempo de execução, promovendo flexibilidade e reutilização de código.

```c
#include

void saudacao() {
    printf("Olá, mundo!
");
}

int main() {
    void (*ptr)() = saudacao; // ponteiro para função
    ptr(); // chamada indireta
    return 0;
}
```

Ponteiros para estruturas

Permitem acessar os campos de uma struct de forma indireta, o que é bastante útil em alocações dinâmicas com malloc, quando não sabemos quantos registros teremos de antemão.

Exemplo

Imagine uma ficha de cadastro guardada em uma pasta. Em vez de carregar a ficha inteira, você anota o número da gaveta em que ela está guardada. Assim, sempre que precisar, você pode consultar o conteúdo acessando aquele local.

Exemplo prático em C

Usamos "→" para acessar os campos da estrutura Pessoa por meio do ponteiro "***p***", que direciona para um espaço de memória alocado de modo dinâmico. Confira!

```
c

#include
#include

typedef struct {
    char nome[50];
    int idade;
} Pessoa;

int main() {
    Pessoa* p = (Pessoa*)malloc(sizeof(Pessoa));
    if (p == NULL) return 1;

    printf("Digite o nome: ");
    scanf("%s", p->nome);
    printf("Digite a idade: ");
    scanf("%d", &p->idade);

    printf("%s tem %d anos.
", p->nome, p->idade);
    free(p);
    return 0;
}
```

Quando usar?

Ponteiro é um tipo especial de variável da linguagem C, sendo muito útil nas seguintes situações:

- Quando funções precisam ser passadas como argumentos para outras funções (como callbacks).
- Para trabalhar com dados alocados dinamicamente (listas e árvores).
- Para construir programas adaptáveis, modulares e reutilizáveis.

Ponteiros para funções e estruturas tornam o código mais poderoso e flexível, de modo que as ações e os dados sejam manipulados de forma indireta, promovendo modularidade e reutilização em aplicações mais complexas. Em C, além de apontar para variáveis comuns, ponteiros também podem direcionar para funções e

estruturas. Isso aumenta a flexibilidade do programa, permitindo comportamentos mais genéricos e reutilizáveis.

Hora de codar: sistema de biblioteca

Nesta prática, você aplicará os conhecimentos adquiridos no nível aventureiro de estrutura de dados para implementar a evolução do sistema de uma biblioteca para incluir uma nova funcionalidade: o empréstimo de livros.

Além de registrar informações básicas de cada livro (nome, autor, editora e edição), o sistema também permitirá que usuários realizem e consultem empréstimos. Para isso, você utilizará ponteiros, alocação dinâmica de memória com malloc e calloc e gerenciará a memória com free.

Confira este vídeo, no qual você dará continuidade ao desenvolvimento do sistema da biblioteca, adicionando a funcionalidade de empréstimo de livros. Serão utilizados ponteiros e alocação dinâmica de memória com malloc e calloc, além de free para liberar recursos. O sistema permitirá o cadastro de livros e a realização e consulta de empréstimos feitos por usuários. imperdível!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Desafio: nível aventureiro

Neste vídeo, confira o desenvolvimento do jogo War estruturado, implementando a funcionalidade de ataque entre territórios. Veja também funções para simular batalhas, atualizar territórios e modularizar o código, aproximando o sistema do comportamento estratégico do jogo real.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Interatividade no WAR estruturado

Agora, você dará um grande passo no desenvolvimento do War estruturado, adicionando a funcionalidade de ataque entre os territórios. Continue praticando e se preparando para o desafio final!

O que você vai fazer?

Primeiro, você receberá a missão de implementar a funcionalidade de ataque entre territórios. Com base nos territórios já cadastrados, o sistema deverá permitir que um jogador selecione um território como atacante, e outro como defensor. O ataque será resolvido por meio de uma simulação com dados aleatórios (como rolagem de dados), e o resultado alterará o controle e as tropas do território atacado.

Lembre-se: essa etapa deve utilizar ponteiros para manipular os dados dos territórios e a alocação dinâmica de memória para armazenar os territórios cadastrados, fornecendo maior flexibilidade ao sistema.

Requisitos funcionais

Veja as funcionalidades essenciais para a implementação de uma simulação de batalha entre territórios.

- **Alocação dinâmica de territórios:** utilizar calloc ou malloc para alocar memória para um vetor de struct Territorio com tamanho informado pelo usuário.
- **Simulação de ataques:** criar uma função void atacar(Territorio* atacante, Territorio* defensor) que simula um ataque, utilizando números aleatórios como se fossem dados de batalha.
- **Atualização de dados:** o território defensor deve mudar de dono (cor do exército) se o atacante vencer, e suas tropas devem ser atualizadas.
- **Exibição pós-ataque:** o sistema deve exibir os dados atualizados dos territórios após cada ataque.

Requisitos não funcionais

A implementação deve seguir boas práticas de organização, uso eficiente da memória e interação clara com o usuário. Acompanhe!

- **Modularização:** o código deve estar organizado com funções distintas para cadastro, exibição, ataque e liberação de memória.
- **Uso de ponteiros:** todos os acessos e modificações aos dados dos territórios devem ser feitos por ponteiros.
- **Gerenciamento de memória:** toda memória alocada dinamicamente deve ser liberada ao final do programa utilizando free.
- **Interface amigável:** o terminal deve orientar o jogador sobre quais territórios podem ser usados para atacar e defender, com mensagens claras.

Instruções detalhadas

Para que o programa funcione da maneira esperada, alguns elementos e etapas de implementação devem ser seguidos conforme descrito a seguir.

- **Bibliotecas necessárias:** inclua stdio.h, stdlib.h, string.h e time.h.
- **Struct atualizada:** utilize a struct Territorio com os campos char nome[30], char cor[10], int tropas.
- **Alocação de memória:** peça ao usuário o número total de territórios e use calloc ou malloc para alocar esse vetor de forma dinâmica.
- **Função de ataque:** implemente void atacar(Territorio* atacante, Territorio* defensor) que utilize rand() para simular um dado de ataque (1 a 6) para cada lado.
- **Atualização dos campos:** transfira a cor e metade das tropas para o território defensor se o atacante vencer. Se perder, o atacante perde uma tropa.
- **Liberação de memória:** crie uma função void liberarMemoria(Territorio* mapa) para liberar o espaço alocado.

Requisitos técnicos adicionais

Alguns cuidados extras proporcionam o bom funcionamento e a legibilidade do programa. São eles:

- Utilizar srand(time(NULL)) para garantir aleatoriedade nos dados de ataque.
- Validar as escolhas de ataque para que o jogador não ataque um território da própria cor.
- Usar free corretamente ao final do programa.
- Comentar o código explicando cada função e trecho importante.

Comentários adicionais

O desafio do nível aventureiro representa um passo importante na evolução do sistema do War estruturado. A funcionalidade de ataque aproxima o sistema do comportamento real do jogo e reforça habilidades fundamentais para a construção de programas interativos e escaláveis em C.

Entregando seu projeto

Vamos ao passo a passo:

1. **Desenvolva seu projeto no GitHub:** use o mesmo repositório do GitHub dos níveis anteriores.
2. **Atualize o arquivo do seu código:** atualize o arquivo super_trunfo.c com o código completo, incluindo as novas funcionalidades.
3. **Compile e teste seu programa:** faça isso com rigor para que todas as comparações e cálculos estejam corretos.
4. **Faça commit e push:** realize commit das suas alterações e envie (push) para o seu repositório no GitHub.
5. **Envie o link do repositório no GitHub:** faça isso por meio da plataforma SAVA.

   Para concluir o desafio, é necessário que você faça o commit no GitHub do que você desenvolveu. Certifique-se de organizar e documentar seu repositório como esperado, pois essa etapa demonstra o seu aprendizado.

   Por fim, cadastre o link do seu repositório na Guia de trabalhos da SAVA.

   Atenção! Caso você atualize o seu repositório, não é necessário alterá-lo na SAVA.

Tutorial git

Você está prestes a aplicar os conceitos aprendidos para resolver um desafio prático no ambiente do GitHub. A seguir, veja as instruções gerais para acessar, aceitar e executar o desafio, a fim de que sua solução esteja bem estruturada e documentada.

Dê o primeiro passo

Acesse o GitHub Classroom. Nesse ambiente, você terá acesso ao repositório padrão do desafio. Caso ainda não tenha uma conta no GitHub, não se preocupe: você pode criar uma grátis, clicando no link.

Aceite o desafio
Encontre o repositório criado no GitHub para desenvolver o seu desafio.

Acesse o repositório
Clique no link do repositório para abrir o ambiente GitHub com a descrição do desafio e a estrutura modelo de arquivos e pastas que devem ser utilizadas. Lembre-se: é este o link que você deve enviar através do SAVA.

Explore a estrutura do ambiente
Veja, no ambiente do GitHub, a estrutura organizada de pastas e arquivos necessários para o desenvolvimento do desafio.

Desenvolva o desafio
Utilize o GitHub CodeSpace para editar o arquivo do código-fonte e desenvolver o desafio. Certifique-se de que o código esteja organizado e funcional para resolver o problema proposto.

Entregue o desafio
Forneça o repositório do GitHub com todos os arquivos de código-fonte e conteúdos relacionados ao projeto. Certifique-se de que o repositório esteja bem estruturado, com pastas e arquivos nomeados de maneira clara e coerente. Envie o link para o repositório do seu desafio no GitHub.

Por fim, **comente todos os arquivos de código-fonte,** pois isso demonstra o quanto você sabe sobre o funcionamento do código e facilita a correção por terceiros. Seus comentários devem explicar a finalidade das principais seções do código, o funcionamento de algoritmos complexos e o propósito de variáveis e funções utilizadas.

Confira o vídeo a seguir, que tem dicas e orientação sobre a entrega do seu projeto.

Conteúdo interativo
Acesse a versão digital para assistir ao vídeo.

Modularização – dividir para conquistar

Neste vídeo, aprenda o conceito de modularização na linguagem C, compreendendo como dividir um programa em funções e arquivos com responsabilidades específicas. Veja ainda os conceitos de funções, arquivos .h e abstração de dados.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Em uma grande empresa, há diferentes setores, certo? Acompanhe alguns exemplos!

Gerência

Marketing

Recursos humanos

Finanças

Cada setor tem uma função específica, mas todos colaboram para o funcionamento da organização.

Assim como em um restaurante bem organizado. Na cozinha, o chef não faz tudo sozinho: há uma cozinheira só para os grelhados, outro para os molhos e outro para as sobremesas. Cada um cuida de uma parte, mas, juntos, produzem um prato perfeito (não estranhe se, aparentemente do nada, sentir fome ou vontade de comer!).

Restaurante bem-organizado.

Podemos chamar a divisão de tarefas de **modularização**: quebrar um sistema grande em partes menores e especializadas, chamadas de **módulos**. Vamos conferir mais detalhes!

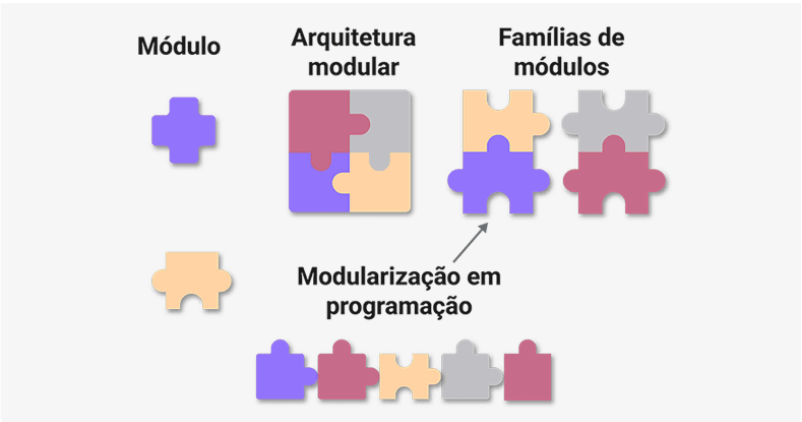
Conceito de modularização

Podemos entender a modularização como o processo de dividir um programa em partes menores e independentes, chamadas módulos. Cada uma é responsável por uma função ou tarefa específica, conforme imagem adiante.

Essa técnica permite organizar, fazer a manutenção e reutilizar o código, de modo que desenvolvedores trabalhem em partes isoladas do sistema sem afetar o restante. Ela também facilita o entendimento do programa na totalidade, pois cada módulo pode ser testado, desenvolvido e atualizado de forma separada, promovendo uma abordagem mais limpa e escalável na construção de softwares.

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Modularização.

Em programação, usamos a modularização para que o programador faça a divisão do programa em módulos. Cada um dos módulos tem uma responsabilidade bem definida, sem precisar escrever um código gigantesco e difícil de entender.

Modularizar um programa significa organizar o código em **funções** e **arquivos** com responsabilidades bem definidas. Isso torna o código mais fácil de corrigir, entender e expandir. Além disso, ainda permite o **reaproveitamento de código**, já que uma função pode ser usada em diferentes contextos.

Exemplo

Modularizar é como preparar uma receita complexa em etapas: primeiro você pica os ingredientes, depois refoga, cozinha, e, por fim, tempera. Se fosse tudo misturado de uma vez, o prato não teria qualidade. O mesmo acontece com o código: módulos claros tornam o programa mais organizado, reutilizável e fácil de manter.

Na prática, modularizar significa separar o código em funções e arquivos diferentes. Cada função executa uma tarefa específica e pode ser usada por outras partes do programa, como um bloco de Lego que se encaixa em vários contextos. A imagem a seguir nos ajuda a entender melhor esse conceito!

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.

Modularização em blocos de Lego.

Conceito de função e arquivos em C

Em programação na linguagem C, uma **função** é um bloco de código que realiza uma tarefa específica e pode ser chamada a partir de diferentes pontos do programa, promovendo reutilização e organização. **Funções**, por sua vez, permitem dividir o programa em partes lógicas, facilitando o desenvolvimento, a leitura e a manutenção.

Já um arquivo em C refere-se a uma unidade de código-fonte, em geral, com extensão .h, que armazena funções, declarações, definições e variáveis.

Os arquivos em C com extensão .h são também chamados de **arquivos de cabeçalho (.h)**. Existem dois tipos principais de arquivos .h. Veja!

Arquivos de cabeçalho da biblioteca padrão

São fornecidos pela linguagem C. São eles:

Arquivo .h	Finalidade
<stdio.h>	Entrada e saída padrão (ex: printf, scanf)
<stdlib.h>	Funções utilitárias (ex: malloc, free, exit)
<string.h>	Manipulação de strings (ex: strlen, strcpy, strcmp)
<math.h>	Funções matemáticas (ex: sqrt, pow, sin)
<time.h>	Controle de tempo e data (ex: time, clock, difftime)
<ctype.h>	Testes de caracteres (ex: isalpha, isdigit, toupper)
<stdbool.h>	Permite usar o tipo bool com true e false
<limits.h>	Define limites de tipos primitivos (INT_MAX, CHAR_MIN, etc.)
<float.h>	Define limites para tipos de ponto flutuante (FLT_MAX, etc.)

Tabela: Exemplo de arquivos de cabeçalho da biblioteca padrão.
Daisy Cristine Albuquerque da Silva.

Arquivos de cabeçalho personalizados

São criados pelo próprio programador para organizar o código em módulos. Veja alguns casos:

Arquivo .h	Finalidade
"funcoes.h"	Declara funções utilitárias do projeto
"aluno.h"	Define a estrutura Aluno e funções relacionadas
"jogo.h"	Define estruturas e funções do jogo (como WAR, batalha, ranking etc.)
"menu.h"	Contém a interface dos menus do sistema
"usuario.h"	Gerencia estruturas e operações com usuários

Tabela: Exemplos de arquivos de cabeçalho personalizado.
Daisy Cristine Albuquerque da Silva.

Conceito de abstração de dados

Significa esconder os detalhes de implementação e mostrar apenas o necessário. Voltando ao exemplo do restaurante: o cliente só vê o prato servido, não precisa saber como o molho foi feito ou como o bife foi grelhado, certo?

Logo, em programação, o usuário de uma função deve saber apenas como a usar, sem se preocupar com o que acontece por dentro.

Vamos ao exemplo!

Pense em um aplicativo de delivery. A função calcular frete está separada da função mostrar os restaurantes. Isso é modularização. Se você mudar a regra do frete, não precisa reescrever o código do menu!

Por último, modularizar ajuda a manter o foco em resolver partes menores do problema, o que facilita testar e encontrar erros. Além disso, também permite que equipes trabalhem em paralelo: enquanto uma pessoa cuida do módulo de pagamentos, outra cuida do cadastro de usuários.

Aplicativo de delivery.

Escopo de variáveis — o que é escopo local e global

Neste vídeo, acompanhe o conceito de escopo em C, que define onde e como variáveis podem ser acessadas no programa. Com analogias práticas, como o funcionamento de um hotel, diferencie escopo local e global, sabendo como cada tipo influencia na organização e segurança do código.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Em um hotel, cada hóspede tem uma chave que abre apenas a porta do seu quarto. Isso é um **acesso local**. Já o gerente tem um cartão que abre todos os quartos; isso é um **acesso global.**

Atenção

Em programação, chamamos isto de escopo: o local do programa onde uma variável pode ser acessada. Em linguagem C, escopo refere-se à região do programa onde uma variável é declarada e pode ser acessada. Ele determina a visibilidade e o tempo de vida de variáveis e funções dentro do código. Entender o escopo evita conflitos de nomes, controla o uso da memória e garante que as variáveis sejam manipuladas apenas onde for apropriado.

Tipos de escopo

Vamos conhecer melhor os dois tipos que começamos a estudar!

Local

Uma variável tem escopo local quando é declarada **dentro de uma função**. Ela só existe ali. É como um copo d'água servido durante uma refeição: serve apenas para aquela ocasião.

Global

Uma variável tem escopo global quando é declarada **fora de qualquer função**. Ela pode ser usada por qualquer parte do programa. Mas atenção! Se cair em mãos erradas, pode causar problemas.

Vamos ao exemplo!

Se você escreve sua lista de compras em um papel guardado no seu bolso, só você pode acessá-la: isso é escopo local. Mas se escreve em um quadro na cozinha, todos da casa podem ver e alterar: isso é escopo global.

O escopo local favorece o encapsulamento, pois impede que outras partes do programa alterem informações de modo indevido. Já o escopo global deve ser usado com cuidado, apenas para dados que, de fato, precisam estar visíveis para todos os módulos.

Vamos ao código!

Na linguagem C, variáveis declaradas dentro de funções são locais. Por exemplo:

```
c

void exemplo() {
    int x = 5; // local
}
```

E variáveis globais são declaradas fora de qualquer função:

```
c

int contador = 0; // global
```

Agora vamos ao exemplo na prática em C:

```
c

#include

int global = 10; // escopo global

void mostrarGlobal() {
    printf("Global: %d\n", global);
}

int main() {
    int local = 5; // escopo local
    printf("Local: %d\n", local);
    mostrarGlobal();
    return 0;
}
```

No exemplo, a variável global pode ser usada em qualquer função. Já a local só pode ser usada dentro da função main().

Passagem de parâmetros por valor

Neste vídeo, aprenda sobre a passagem de parâmetros por valor na linguagem C e como funções recebem cópias das variáveis originais, sem alterar seus valores. Entenda também quando e por que utilizar esse tipo de passagem para manter o programa seguro e previsível.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Quando criamos funções em programação, é comum precisarmos enviar dados para essas funções realizarem alguma tarefa. Esse envio de dados é chamado de **passagem de parâmetros.** Em outras palavras: passar um parâmetro é como dar uma informação para a função trabalhar.

Existem diferentes formas de passar parâmetros, e uma das mais comuns é a passagem por valor.

Atenção

Na passagem por valor, a função recebe uma cópia da variável original. Isso significa que a função pode modificar essa cópia à vontade, mas a variável original — aquela que está no main() ou em outro lugar do programa — não será alterada. A ideia é simples: você entrega uma cópia para a função usar, como um documento escaneado. Ela pode riscar, dobrar ou jogar fora a cópia, mas sua versão original continua segura e intacta.

Vamos entender melhor. Imagine que você tem uma receita de brigadeiro escrita em um caderno. Você tira uma cópia para dar ao seu amigo. Mas ele resolve adicionar canela na receita, riscar alguns ingredientes e escrever observações. Apesar disso, a sua receita original no caderno continua do jeitinho que você escreveu. Isso é o que acontece na passagem de parâmetros por valor.

Mas quando usar?

Esse tipo de passagem é ideal quando você quer apenas usar os dados para um cálculo ou exibição, mas não precisa alterá-los para sempre. Isso dá segurança ao programa e evita efeitos colaterais, ou seja, mudanças inesperadas em outras partes do código.

Na linguagem C, quando passamos um parâmetro por valor, estamos dizendo: “toma essa informação, mas não mude a original”. A função recebe uma cópia e trabalha com ela.

Vamos ao exemplo!

Agora, vamos dobrar o número informado por meio de uma chamada à função e verificar se o valor armazenado na memória será alterado.

O objetivo do código é demonstrar o funcionamento da passagem de parâmetros por valor em funções na linguagem C. Ao chamar a função dobrar, uma cópia do valor da variável numero é enviada, o que significa que qualquer modificação realizada dentro da função não vai alterar o valor original da variável. Isso é comprovado ao observarmos que, mesmo após o valor ser dobrado dentro da função, o valor impresso fora dela permanece inalterado. Esse comportamento ajuda a entender a diferença entre modificar uma cópia e acessar diretamente o dado original, o que só seria possível com a utilização de ponteiros.

- A função dobrar(int x) recebe um número e dobra seu valor, mas a modificação acontece apenas dentro da função.

- No main(), a variável numero é inicializada com o valor 5 e passada para a função dobrar(numero).
- Dentro da função, x recebe uma cópia do valor de numero. Quando x é dobrado, a alteração não afeta a variável original numero.
- Ao imprimir dentro da função, vemos o valor dobrado (10), mas fora da função, numero continua com o valor original (5).

Confira o código a seguir.

```
c

include

void dobrar(int x) {
    x = x * 2;
    printf("Dentro da função: %d
", x);
}

int main() {
    int numero = 5;
    dobrar(numero);
    printf("Fora da função: %d
", numero );
    return 0;
}
```

Visualize o que será retornado pelo código ao final:

Terminal

Saída esperada: Dentro da função: 10 Fora da função: 5

Tal comportamento ajuda a manter o programa previsível e confiável, em especial, em aplicações maiores. É comum que funções que consultam dados, fazem verificações e exibem resultados ou calculam valores utilizem passagem por valor. Afinal, elas não precisam modificar os dados originais, apenas trabalhar com uma versão temporária.

Passar por valor é a forma padrão de enviar dados para funções em C. Lembre-se: quando não há necessidade de alterar os dados na função, essa é a forma mais clara, segura e direta de programar.

Passagem de parâmetros por referência

Neste vídeo, veja a passagem de parâmetros por referência na linguagem C, o conceito com analogias e a sintaxe correta com * e &. Por fim, aplique o que você aprendeu em um exemplo prático usando a função dobrar().

Conteúdo interativo

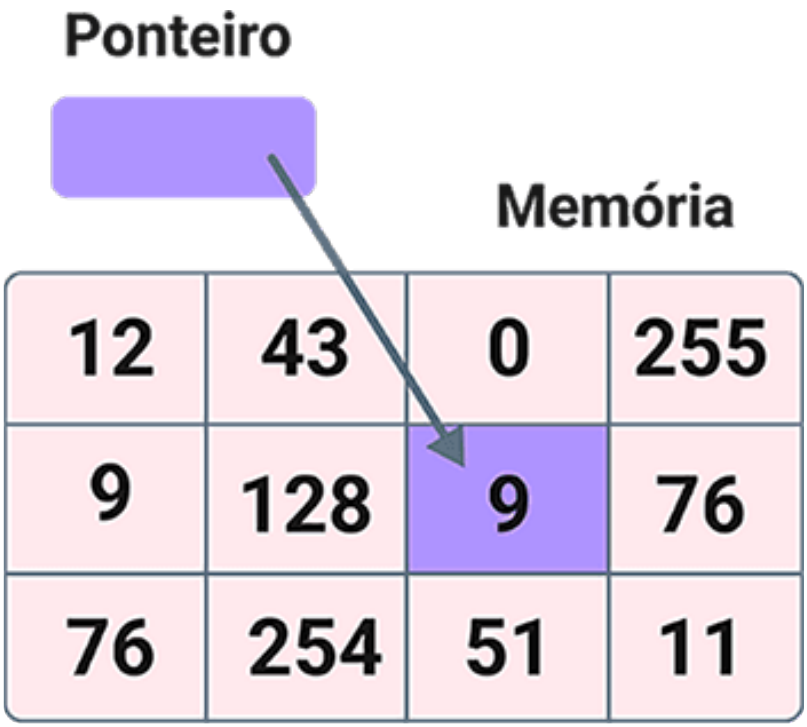
Acesse a versão digital para assistir ao vídeo.

Na programação, quando queremos que uma função modifique diretamente o valor de uma variável, usamos a técnica chamada **passagem de parâmetros por referência**. Isso significa que, em vez de enviar uma cópia da variável para a função (como na passagem por valor), enviamos o endereço de memória no qual a variável está armazenada. Com esse endereço, a função pode acessar e alterar o conteúdo original.

Relembrando o caso do seu amigo e a receita de brigadeiro, pense que, em vez de entregar a cópia da receita, você entrega o original. Seu amigo adiciona canela, risca ingredientes e escreve observações diretamente na folha. Quando você pega o caderno de volta, percebe que a receita foi alterada, de modo que as mudanças feitas agora estão no original. Isso é o que acontece na passagem de parâmetros por referência: a função recebe acesso direto à variável original, podendo modificá-la para sempre.

Mas o que é esse tal de endereço de memória?

Toda variável em C ocupa um espaço na memória do computador. Cada espaço tem um “número”, como se fosse o número de uma casa. Trata-se do endereço, e podemos acessá-lo usando um ponteiro. Observe a imagem a seguir.



Acesso à memória com ponteiros.

A imagem representa o conceito de ponteiros em programação, utilizando uma analogia visual com uma tabela de memória. Cada célula da tabela contém um valor, simbolizando uma posição de memória. Um retângulo vermelho rotulado como PONTEIRO está ligado por uma seta a uma das células da tabela que contém o valor 9, destacada também em vermelho.

Vemos, então, que um ponteiro não guarda diretamente um valor, mas o endereço (ou a posição) na memória em que esse valor está armazenado. Assim, o ponteiro aponta para uma célula específica da memória, permitindo acessar ou manipular o valor nela contido de forma indireta.

Mas o que são ponteiros?

São variáveis especiais que armazenam endereços de memória. Ou seja, enquanto uma variável comum armazena um valor (como um número ou uma letra), o ponteiro armazena o local onde esse valor está guardado.

> Em C, usamos o símbolo * para declarar um ponteiro e & para acessar o endereço de uma variável.

Imagine agora a seguinte situação: você quer entregar um presente para um amigo. Você pode fazer isso de duas formas: dando o presente em mãos ou entregando o endereço da casa dele e pedindo para alguém levá-lo até lá.

Na programação, um ponteiro funciona como esse endereço. Ou seja, em vez de trabalhar com o valor da variável (como o presente em mãos), você trabalha com o endereço de memória no qual esse valor está guardado, isto é, você aponta para o local onde o valor está.

Ainda com dúvida?

Pense em uma carta com instruções (a variável) guardada em uma gaveta (o endereço). Você pode ler a carta (usar a variável) ou entregar para alguém o endereço da gaveta. Assim, essa pessoa pode abrir a gaveta e alterar o conteúdo original da carta. Esse é o poder dos ponteiros: permitir o acesso direto à fonte da informação.

Vamos ao exemplo!

O objetivo da implementação é demonstrar o uso da passagem de parâmetros por referência na linguagem C, utilizando ponteiros. Diferentemente da passagem por valor, aqui a função dobrar recebe o endereço de memória da variável numero, permitindo que a modificação feita dentro da função afete o valor original. Ao acessar e alterar o conteúdo da variável por meio do ponteiro (*x), o valor de numero, de fato, dobra, e essa mudança é refletida ao final da execução do programa.

- int* x declara que a função dobrar recebe um ponteiro para inteiro.
- &numero passa o endereço da variável numero.
- *x acessa o conteúdo no endereço e o modifica.

  Veja o código adiante.

```
c

#include

void dobrar(int* x) {
    *x = (*x) * 2;
}

int main() {
    int numero = 5;
    dobrar(&numero);
    printf("Número dobrado: %d
", numero);
    return 0;
}
```

Veja o que será exibido ao final do código:

Terminal

Saída esperada: Número dobrado: 10

A variável original foi alterada porque a função operou sobre o seu endereço de memória. Isso mostra como a passagem por referência permite que mudanças dentro da função se reflitam fora dela.

Quando usar?

Utilizamos essa abordagem quando queremos que a função modifique os dados originais. Por exemplo, para atualizar saldos bancários, alterar valores em um vetor ou matriz, preencher dados em uma struct e trocar valores entre duas variáveis.

Atenção

Cuidados ao usar ponteiros! Como eles lidam com endereços, é preciso muito cuidado para não acessar áreas inválidas da memória. Um erro comum é esquecer de inicializar o ponteiro antes de usá-lo, o que pode causar falhas no programa.

Hora de codar: Sistema de biblioteca

Nesta prática, você aplicará os conhecimentos adquiridos no nível mestre de estrutura de dados para implementar a evolução do sistema da biblioteca já desenvolvido.

Para isso, você vai modularizar o sistema separando o código em funções específicas com responsabilidades bem definidas. Assim, você aplicará os conceitos de passagem por valor para exibir dados e passagem por referência para atualizar os registros de empréstimos. A prática também reforça o uso de ponteiros e alocação dinâmica de memória com malloc, calloc e free.

Neste vídeo, você dará continuidade ao sistema da biblioteca, mas agora no nível mestre, aplicando modularização do código com funções especializadas. Serão utilizados ponteiros, alocação dinâmica com malloc, calloc e free, além da passagem por valor para exibição de dados e por referência para atualizar empréstimos, consolidando boas práticas de estruturação e manipulação de memória em C.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Desafio: nível mestre

Neste vídeo, conclua o desenvolvimento do jogo War estruturado implementando a funcionalidade de missões estratégicas. Para isso, serão usados ponteiros, alocação dinâmica com malloc, modularização e passagem de parâmetros por valor e referência.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Implementando a funcionalidade de missão estratégica

Este é o desafio final do War estruturado! Coloque em prática tudo o que você aprendeu e mostre suas habilidades de programação!

O que você vai fazer?

Você receberá a missão de implementar a funcionalidade de missões estratégicas individuais para cada jogador, que deverá receber, no início do jogo, uma missão sorteada de forma automática entre diversas descrições pré-definidas, armazenadas em um vetor de strings. Essa missão será consultada durante o jogo para verificar se a condição de vitória foi atingida. A nova camada de estratégia exige organização modular do código, uso de ponteiros, passagem de parâmetros por valor e referência e gerenciamento adequado da memória.

Requisitos funcionais

A seguir, são apresentadas as etapas para implementar o sistema de missões dos jogadores, desde a criação até a verificação de objetivos.

- **Criação do vetor de missões:** declarar um vetor de strings contendo ao menos cinco descrições diferentes de missões estratégicas (ex: Conquistar 3 territórios seguidos, Eliminar todas as tropas da cor vermelha etc.).
- **Sorteio da missão:** implementar a função void atribuirMissao(char* destino, char* missoes[], int totalMissoes) que sorteia uma missão e copia para a variável de missão do jogador usando strcpy.
- **Armazenamento e acesso:** a missão de cada jogador deve ser armazenada dinamicamente utilizando malloc.
- **Verificação da missão:** implementar a função int verificarMissao(char* missao, Territorio* mapa, int tamanho), que avalia se a missão do jogador foi cumprida (crie uma lógica simples inicial para verificação).
- **Exibição condicional:** o sistema deve verificar, ao final de cada turno, se algum jogador cumpriu sua missão e declarar o vencedor.

Requisitos não funcionais

Para garantir organização e clareza na execução das missões, algumas práticas e estruturas devem ser seguidas durante a implementação. Vamos lá!

- **Modularização:** o código deve estar dividido em funções específicas, como atribuirMissao, verificarMissao, exibirMissao, atacar, exibirMapa, liberarMemoria, e a main.
- **Uso de ponteiros:** as missões dos jogadores devem ser manipuladas por meio de ponteiros.
- **Passagem por valor e referência:** a missão deve ser passada por valor para exibição e por referência para atribuição e verificação.
- **Interface intuitiva:** o sistema deve exibir a missão ao jogador apenas uma vez (no início) e verificar silenciosamente se ela foi cumprida ao longo da execução.

Instruções detalhadas

Os elementos a seguir ajudam a estruturar corretamente o funcionamento do jogo e o gerenciamento da memória. Veja!

- **Bibliotecas necessárias:** inclua stdio.h, stdlib.h, string.h e time.h.
- **Estrutura dos territórios:** utilize a struct Territorio com os campos char nome[30], char cor[10], int tropas.
- **Alocação de memória:** use calloc ou malloc para alocar os vetores de territórios e armazenar a missão de cada jogador.
- **Função de ataque:** implemente void atacar(Territorio* atacante, Territorio* defensor) usando rand() para simular uma rolagem de dados (valores entre 1 e 6).
- **Atualização de campos:** transfira a cor e metade das tropas para o território defensor se o atacante vencer. Caso contrário, o atacante perde uma tropa.
- **Função de liberação:** implemente void liberarMemoria(...) para liberar toda a memória alocada dinamicamente (territórios e missões).

Requisitos técnicos adicionais

Para finalizar a implementação com boas práticas, siga os cuidados técnicos adiante:

- Use srand(time(NULL)) para gerar números aleatórios.
- Valide os ataques para que o jogador só possa atacar territórios inimigos.
- Utilize free() ao final para evitar vazamentos de memória.
- Comente o código explicando o papel de cada função e lógica importante.

Comentários adicionais

Com a introdução das missões estratégicas, este desafio aproxima ainda mais o sistema War estruturado da lógica original do jogo. Ele promove a prática de modularização, uso de ponteiros, alocação dinâmica de memória e simulação de regras de vitória baseadas em condições específicas. Trata-se de um excelente exercício para integrar diversos conceitos fundamentais da linguagem C em um projeto envolvente e gamificado.

Entregando seu projeto

Vamos ao passo a passo:

1. **Desenvolva seu projeto no GitHub:** use o mesmo repositório do GitHub dos níveis anteriores.
2. **Atualize o arquivo do seu código:** atualize o arquivo super_trunfo.c com o código completo, incluindo as novas funcionalidades.
3. **Compile e teste:** faça isso com rigor, de modo que todas as comparações e cálculos estejam corretos.
4. **Faça commit e push:** Faça commit das suas alterações e envie (push) para o seu repositório no GitHub.
5. **Envie o link do repositório:** faça isso por meio da plataforma SAVA.

   Para concluir o desafio, é necessário que você faça o commit no GitHub do que você desenvolveu. Certifique-se de organizar e documentar seu repositório como o esperado, pois essa etapa demonstra

seu aprendizado.

Por fim, cadastre na Guia de trabalhos da SAVA o link do seu repositório.

Atenção! Caso você atualize o seu repositório, não é necessário alterá-lo na SAVA.

Tutorial git

Você está prestes a aplicar os conceitos aprendidos para resolver um desafio prático no ambiente do GitHub. Veja as instruções gerais a seguir para acessar, aceitar e executar o desafio, de modo que sua solução esteja bem estruturada e documentada.

Dê o primeiro passo

Acesse o GitHub Classroom. Nesse ambiente, você terá acesso ao repositório padrão do desafio. Caso ainda não tenha uma conta no GitHub, não se preocupe: você pode criar uma grátis clicando no link.

Aceite o desafio

Encontre o repositório criado para o desenvolvimento do seu desafio no GitHub.

Acesse o repositório

Clique no link do repositório para abrir o ambiente GitHub com a descrição do desafio e a estrutura modelo de arquivos e pastas que deve ser utilizada. É esse link que você deve enviar no SAVA.

Explore a estrutura do ambiente

Veja, no ambiente do GitHub, a estrutura organizada de pastas e arquivos necessários para o desenvolvimento do desafio.

Desenvolva o desafio

Utilize o GitHub CodeSpace para editar o arquivo do código-fonte e desenvolver o desafio. Certifique-se de que o código esteja organizado e funcional para resolver o problema proposto.

Entregue o desafio

Forneça o repositório do GitHub que contém todos os arquivos de código-fonte e conteúdos relacionados ao projeto. Certifique-se de que o repositório esteja bem estruturado, com pastas e arquivos nomeados de maneira clara e coerente. Envie o link para o repositório do seu desafio no GitHub.

Lembre-se de **comente todos os arquivos de código-fonte,** pois isso é indispensável para demonstrar seu entendimento sobre o funcionamento do código e facilitar a correção por terceiros. Os comentários devem explicar a finalidade das principais seções do código, o funcionamento de algoritmos complexos e o propósito de variáveis e funções utilizadas.

Finalizando nosso estudo, assista ao vídeo, a seguir, com dicas e detalhes sobre a entrega do seu projeto.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Considerações finais

Parabéns pela conquista!

Você chegou ao final de uma jornada intensa, estratégica e muito relevante para a sua formação. Ao longo dos três níveis — novato, aventureiro e mestre — você não construiu um jogo funcional, é certo, mas também consolidou conhecimentos fundamentais de estruturas de dados, modularização de código, uso de structs, ponteiros, alocação dinâmica de memória e passagem de parâmetros por valor e referência.

No nível novato, você aprendeu a importância de organizar dados com vetores e estruturas lineares, criando um mapa básico com territórios. Já no nível aventureiro, você aplicou conceitos avançados(como ponteiros, malloc, calloc e free), implementando a lógica de ataque entre territórios.

Por fim, no nível mestre, você implementou a lógica de missões estratégicas, tornando o jogo dinâmico e desafiador, modularizando o código, organizando melhor as funcionalidades e implementando o papel das funções bem definidas e do escopo.

Agora que você domina esses conceitos, aqui vão algumas dicas para continuar evoluindo:

• **Pratique com frequência:** programação é como treino. Quanto mais você pratica, mais fluente e seguro se torna.

• **Desafie-se:** crie seus próprios projetos, melhore o que foi feito aqui e teste outras funcionalidades do jogo War.

• **Comente e refatore seu código:** melhore sua lógica e sua comunicação com outros desenvolvedores.

• **Estude problemas do mundo real:** aplicações de estruturas de dados aparecem em jogos, apps, bancos, redes e muito mais.

• **Continue explorando, codando e aprendendo — o mercado precisa de profissionais como você:** desenvolva lógica, organização e paixão por resolver problemas.

Até o próximo desafio!

Continue praticando

A prática constante é o que transforma conhecimento em habilidade, e nada melhor do que desenvolver novos projetos para consolidar o aprendizado. Veja algumas ideias de projetos para continuar evoluindo:

• Sistema de gerenciamento de estudantes.

Crie um sistema em C que permita cadastrar, listar, atualizar e remover dados de estudantes utilizando structs, alocação dinâmica de memória e ponteiros. Adicione funcionalidades como cálculo de média, verificação de aprovação e exportação de relatórios. Esse projeto ajuda a reforçar o uso de structs compostas, modularização e passagem de parâmetros por referência — habilidades perfeitas para o desenvolvimento de sistemas mais robustos.

Referências

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos**: teoria e prática. 3. ed. Rio de Janeiro: Elsevier, 2012.

DEITEL, P. J.; DEITEL, H. M. **C**: Como programar. 6. ed. São Paulo: Pearson Prentice Hall, 2011.

BACKES, A. R. **Algoritmos e estruturas de dados em linguagem C**. Rio de Janeiro: LTC, 2012.

TENENBAUM, A. M. *et al.* **Estruturas de dados usando C**. São Paulo: Makron Books, 1995.