Estrutura de Dados

Prof. Maurício Neto

UNIFAVIP
wyden

O que vamos ver?

01

Variáveis e Constantes

02

Variáveis Locais e Globais

03

TAD

UNIFAVIP wyden

01

Variáveis e Constantes

Variáveis e Constantes

Uma variável em C representa uma informação que pode ser utilizada dentro do programa. Essa informação está vinculada a um local específico da **memória**, algo que é determinado pelo compilador. O nome da variável está associado ao endereço de memória onde o dado é armazenado.

Embora o nome da variável e o endereço permaneçam os mesmos, o valor armazenado pode ser alterado, ainda que o tipo de dado continue o mesmo. Cada variável possui um tipo associado, como **int, char** ou **float,** que definem o tipo de dado que ela pode armazenar.

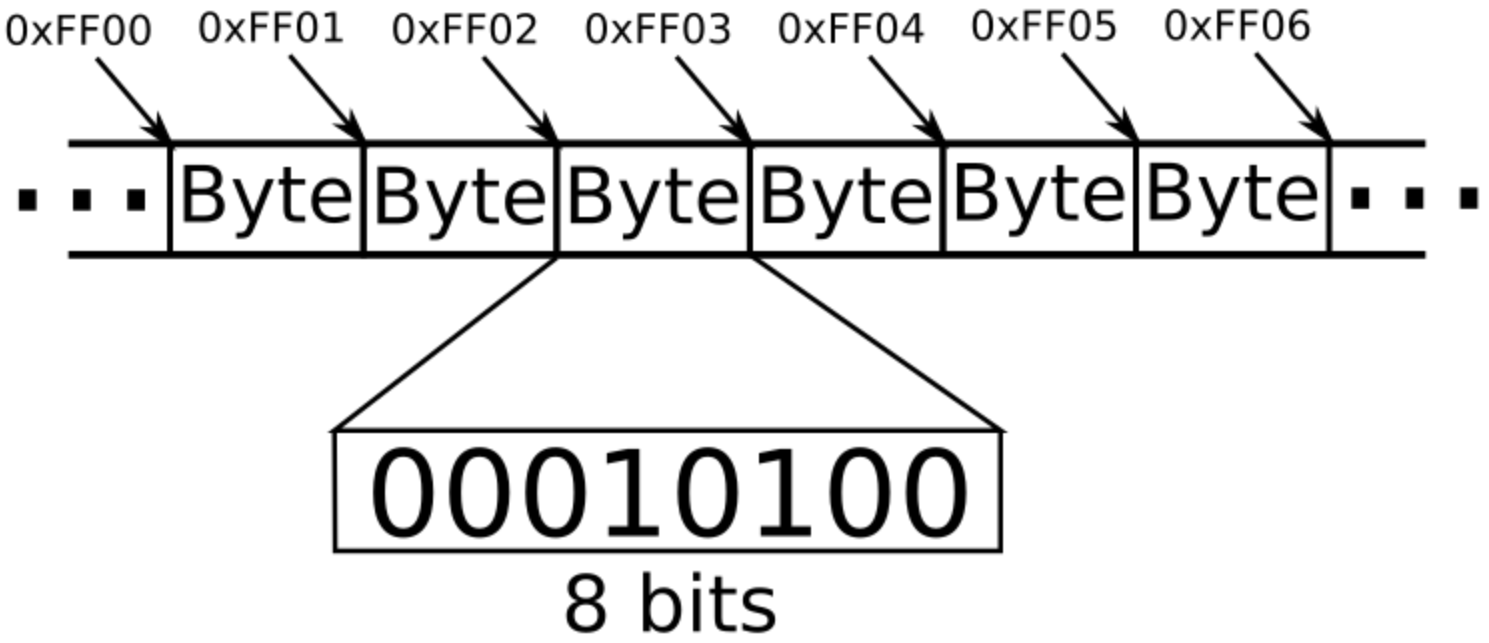
Variáveis e Constantes

O espaço de memória utilizado por cada variável depende de seu tipo. Variáveis do mesmo tipo ocupam a mesma quantidade de bytes, independentemente do valor armazenado.

Memória

Memória RAM é o principal armazenamento do computador, onde são mantidas as informações (dados e instruções) dos programas que estão sendo executados no momento. Isso inclui, por **exemplo**, o armazenamento das variáveis dos nossos programas durante sua execução.

Podemos imaginar a memória como um grande **array** de endereços. Como a memória é endereçada por byte, cada endereço corresponde a 1 byte (8 bits).



Cada posição de memória armazena **1 byte** e é identificada por um **endereço**. Esse endereço é representado em notação hexadecimal.

UNIFAVIP
wyden

Tamanho da Memória

O tamanho da memória é determinado pelo tamanho dos endereços. No exemplo, um endereço é representado por um número hexadecimal de 4 dígitos:

Aqui, o prefixo "0x" indica que o número está em base **hexadecimal**.

Cada dígito hexadecimal equivale a 4 bits binários. Portanto, nesse exemplo, cada endereço possui 16 bits.

Tamanho da Memória

O tipo de processador define o limite de endereçamento da memória:

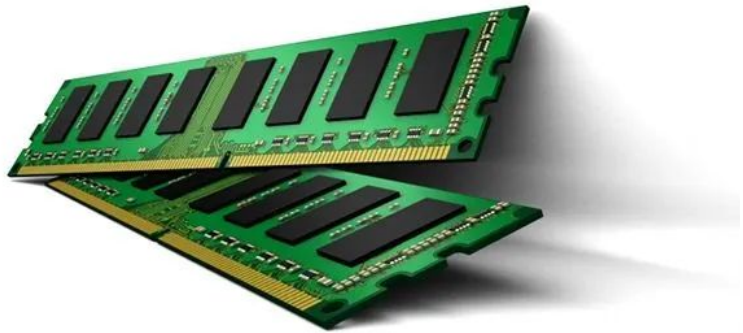
- Um processador x86 utiliza 32 bits para endereçamento, permitindo um máximo de 2^{32} bytes, ou 4 GiB.

- Um processador x64 utiliza 48 bits para endereçamento, permitindo até 2^{48} bytes, ou 256 TiB.

Como usamos a Memória

Alocarmos espaço para variáveis em nossos programas sempre que necessário.

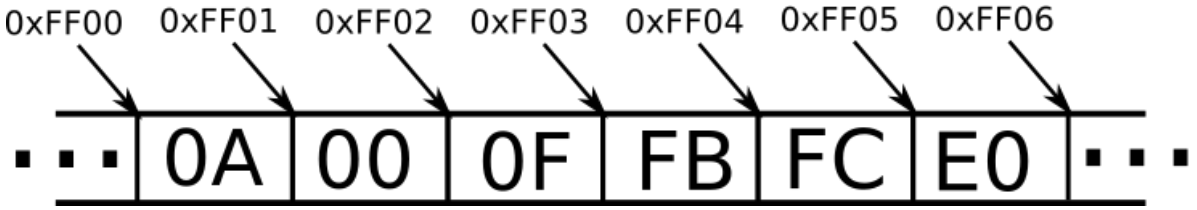
Ler os valores armazenados nessas variáveis e escrever novos valores. Esses valores, tanto os lidos quanto os escritos, são sempre representados em bytes.



UNIFAVIP
wyden

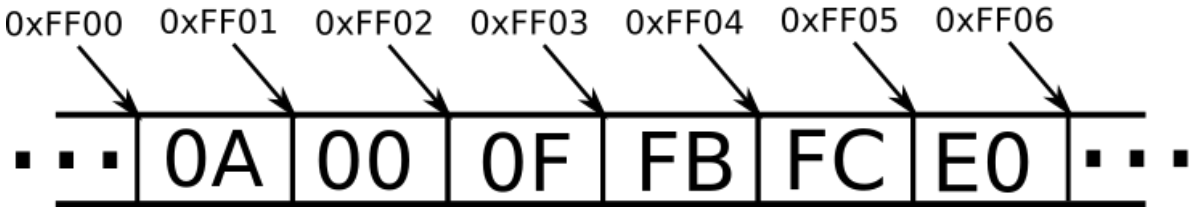
“Dentro” da memória

Do ponto de vista da **máquina**, a memória armazena apenas bits. No exemplo, cada byte é representado em notação hexadecimal.



Para o **programador**, o que está armazenado na memória depende do tipo de variável associado ao endereço acessado.

“Dentro” da memória



- Se interpretarmos o byte no endereço **0xFF00** como um **char**, ele será avaliado como '**\n**'.
- Se interpretarmos o byte no endereço **0xFF03** como um **char**, ele será avaliado como '**û**'.
- Interpretando a sequência de 4 bytes a partir do endereço **0xFF01** como um **int**, ela será avaliada como **1047548**.
- Interpretando a sequência de 4 bytes a partir do endereço **0xFF00** como um float, ela será avaliada como **6.16598e-33.**

UNIFAVIP wyden

Decimal	Binário	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Definição Variável em C

Ao utilizar variáveis em um programa, é necessário defini-las, o que envolve especificar o tipo da variável e atribuir um nome a ela. As regras para formar nomes de variáveis em C são as seguintes:

Podem ser compostos por qualquer sequência de letras, dígitos e ‘_’, mas devem iniciar com uma letra ou com '_'.

Exemplos como **hora_inicio**, **tempo** e **var1** são válidos, enquanto 3horas, total$ e azul-claro não são permitidos.

Maiúsculas são diferentes de minúsculas.
Não são permitidos nomes ou palavras reservadas da linguagem.

Definição Variável em C

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Definição Variável em C

Tipo	Tamanho (em bits)	Intervalo
Char	8	-128 a 127
Int	16	-32768 a 32767
Float	32	3,4E-38 a 3,4E+38
double	64	1,7E-308 a 1,7E+308
void	0	sem valor

```c
#include <stdio.h>

int main() {

    int idade;
    char qualidade;
    float peso;

    idade = 10;
    qualidade = 'C';
    peso = 0.78;

}
```

UNIFAVIP wyden

Constantes

Em C, além de variáveis, podemos utilizar números ou caracteres cujos valores permanecem **inalterados**, denominados **constantes**. Ao contrário das variáveis, as constantes não estão associadas a locais específicos na memória.

Assim como as variáveis, as constantes também possuem tipos, podendo ser do tipo **int, char, entre outros**. Não é necessário declarar constantes explicitamente; elas podem ser utilizadas diretamente, sendo o tipo reconhecido pelo compilador com base na forma como são escritas.
Por exemplo, o valor 2 é do tipo int, enquanto 2.0 é do tipo double. Por convenção, todas as constantes reais são tratadas como sendo do tipo double.

```c
#include <stdio.h>

int main() {

    const int NUMERO = 42;          // Uma constante do tipo int
    const char CARACTERE = 'A';     // Uma constante do tipo char
    const double PI = 3.14159;      // Uma constante do tipo double

    // Utilizando as constantes
    printf("Numero: %d\n", NUMERO);
    printf("Caractere: %c\n", CARACTERE);
    printf("Valor de PI: %f\n", PI);

    return 0;
}
```

```c
#include <stdio.h>

int main() {
    int idade = 25;
    float altura = 1.75;

    const double PI = 3.14159;

    idade = 26;

    return 0;
}
```

VARIÁVEIS E CONSTANTES

[LÓGICA DE PROGRAMAÇÃO]

Atividade

Celsius para Fahrenheit.
No programa, defina uma **constante** para o fator de conversão (9.0/5.0) e **outra constante** para o valor de ajuste **(+32)**. O usuário vai digitar o valor em celsius e vai ser impresso o valor convertido.

```
Digite a temperatura em Celsius: 29
A temperatura de 29.0°C em Fahrenheit é 84.2.
```

02

Variáveis Locais e Globais

Variáveis Locais

Variáveis definidas dentro de uma função são conhecidas como **variáveis locais** dessa função. Isso inclui os parâmetros formais da função, que também são variáveis locais.

Variáveis locais são específicas para a função em que são definidas, ou seja, apenas essa função pode acessá-las, conhecer seu endereço e modificar seu conteúdo.

Variáveis Locais

Outras funções não podem acessar essas variáveis locais a menos que o endereço da variável seja passado como argumento.

O fato de cada função manter suas variáveis locais "**escondidas**" ajuda a escrever programas mais estruturados e modulares. Ao criar uma função, podemos nomear suas variáveis locais como desejamos e não precisamos nos preocupar se outras funções podem acessar ou alterar essas variáveis.

```c
#include <stdio.h>


int quadrado(int numero) {
    int resultado; // Variável local para armazenar o resultado

    resultado = numero * numero;
    return resultado;
}

int main() {
    int valor = 5; // Variável local na função main
    int quadradoDoValor; // Variável local para armazenar o resultado do quadrado

    quadradoDoValor = quadrado(valor);

    printf("O quadrado de %d é %d.\n", valor, quadradoDoValor);

    return 0;
}
```

Variáveis Locais

Em geral, uma **variável local** existe apenas durante a execução da função na qual foi definida. Isso significa que as variáveis locais permanecem ativas desde o momento em que a função é chamada até a conclusão da função. Essas variáveis são conhecidas como **automáticas**.

Em C, também é possível definir uma variável como **static**. Nesse caso, a variável local continua não sendo visível fora do corpo da função, mas, diferentemente das variáveis automáticas, não é destruída ao final da execução da função. Cada vez que a função é chamada, a variável **static** retém o valor da última chamada, ao contrário das variáveis automáticas, que são reinicializadas a cada chamada.

```c
#include <stdio.h>

void contagem() {
    int contagemAutomatica = 0; // Variável automática
    static int contagemEstatica = 0; // Variável static

    contagemAutomatica++; // Incrementa a variável automática
    contagemEstatica++; // Incrementa a variável static

    printf("Variável automática: %d\n", contagemAutomatica);
    printf("Variável static: %d\n", contagemEstatica);
}

int main() {

    contagem(); // Saída: Variável automática: 1, Variável static: 1
    contagem(); // Saída: Variável automática: 1, Variável static: 2
    contagem(); // Saída: Variável automática: 1, Variável static: 3

    return 0;
}
```

Variáveis Globais

Até este momento, todas as variáveis abordadas foram definidas dentro de funções (seja no corpo da função ou como parâmetros formais).
No entanto, é possível também definir variáveis fora das funções. Essas variáveis são chamadas de **variáveis globais** ou **externas**. A definição de variáveis globais segue o mesmo formato das variáveis locais, com a diferença de que são definidas fora de qualquer função. Ao contrário das variáveis locais, as variáveis globais podem ser acessadas por todas as funções definidas após a sua declaração.

```c
#include <stdio.h>

// Definição de uma variável global
int contador = 0;

void incrementar() {
    contador++;
}

void exibir() {
    printf("Valor do contador: %d\n", contador);
}

int main() {

    exibir(); // Saída: Valor do contador: 0

    incrementar();
    incrementar();

    exibir(); // Saída: Valor do contador: 2

    return 0;
}
```

Variáveis Globais

Até este momento, todas as variáveis abordadas foram definidas dentro de funções (seja no corpo da função ou como parâmetros formais).
No entanto, é possível também definir variáveis fora das funções. Essas variáveis são chamadas de **variáveis globais** ou **externas**. A definição de variáveis globais segue o mesmo formato das variáveis locais, com a diferença de que são definidas fora de qualquer função. Ao contrário das variáveis locais, as variáveis globais podem ser acessadas por todas as funções definidas após a sua declaração.

03

TAD

UNIFAVIP
wyden

Tipos Abstratos de Dados

Um TAD é uma maneira de organizar e estruturar dados e operações em um programa de forma que a implementação específica dos dados e operações fique oculta para o usuário do TAD.

Em outras palavras, o TAD define a interface e a funcionalidade sem revelar a implementação detalhada.

```c
#include <stdio.h>

#define MAX_SIZE 100

typedef struct {
    int itens[MAX_SIZE];
    int tamanho;
} Lista;

void inicializar_lista(Lista* lista) {
    lista->tamanho = 0;
}

void adicionar_elemento(Lista* lista, int valor) {
    if (lista->tamanho < MAX_SIZE) {
        lista->itens[lista->tamanho] = valor;
        lista->tamanho++;
    } else {
        printf("Lista cheia!\n");
    }
}

void imprimir_lista(Lista* lista) {
    for (int i = 0; i < lista->tamanho; i++) {
        printf("%d ", lista->itens[i]);
    }
    printf("\n");
}

int main() {
    Lista lista;
    inicializar_lista(&lista);

    adicionar_elemento(&lista, 10);
    adicionar_elemento(&lista, 20);
    adicionar_elemento(&lista, 30);

    printf("Lista: ");
    imprimir_lista(&lista);

    return 0;
}
```

```c
#include <stdio.h>
#include "lista.h"

int main() {
    No* lista = criar_lista();

    adicionar_no_inicio(&lista, 10);
    adicionar_no_inicio(&lista, 20);
    adicionar_no_inicio(&lista, 30);

    printf("Lista: ");
    imprimir_lista(lista);

    liberar_lista(lista);

    return 0;
}
```

UNIFAVIP wyden

Tipos Abstratos de Dados

- Assim, o usuário pode se concentrar na interface e ignorar os detalhes da implementação específica.

- Qualquer alteração na implementação interna é confinada ao Tipo Abstrato de Dados (TAD).

- A escolha da representação interna é majoritariamente guiada pelas operações que precisam ser realizadas.

Tipos Abstratos de Dados

Em linguagens orientadas a objeto **(como C++ e Java)**, a implementação de um Tipo Abstrato de Dados (TAD) é realizada por meio de classes.
As classes permitem **encapsular** dados e comportamentos relacionados, fornecendo uma estrutura que define tanto os atributos quanto os métodos que operam sobre esses atributos. Essa abordagem promove o **encapsulamento, a herança e o polimorfismo**, facilitando a organização e reutilização do código. As classes abstraem a complexidade ao permitir que o usuário interaja com objetos através de interfaces bem definidas, sem precisar conhecer os detalhes internos da implementação.

Tipos Abstratos de Dados

Em linguagens estruturadas **(como C e Pascal)**, a implementação de um TAD é feita através da definição de tipos e da implementação de funções.
Em C, isso é alcançado utilizando **typedefs e structs.** Um typedef permite criar um novo nome para um tipo existente, enquanto uma struct define uma estrutura de dados composta por diferentes tipos de variáveis agrupadas sob um único nome.
Funções são então implementadas para operar sobre essas estruturas, proporcionando a interface para interação com os dados. Esta abordagem, embora **menos orientada a objetos**, ainda permite a criação de abstrações e modularização através de um design cuidadoso e organização do código.

Tipos Abstratos de Dados

Conceitos de C: Em C, o **typedef** é utilizado para criar aliases para tipos de dados, simplificando o código e tornando-o mais legível.

A **struct**, por sua vez, é uma construção fundamental para agrupar variáveis relacionadas em um único bloco de memória, permitindo a criação de tipos de dados compostos. Juntas, essas ferramentas ajudam a implementar TADs de maneira estruturada e eficiente.

Tipos Abstratos de Dados

Para implementar um **Tipo Abstrato de Dados (TAD)** em C, utiliza-se a definição de tipos junto com a implementação de funções que operam sobre esses tipos. É uma boa prática de programação evitar o acesso direto aos dados, realizando operações apenas por meio das funções fornecidas.

Tipos Abstratos de Dados

Uma abordagem recomendada é organizar a implementação dos TADs em arquivos separados do código principal. Normalmente, isso envolve dividir a declaração e a implementação do TAD em dois arquivos distintos:

NomeDoTAD.h: Contém as declarações do TAD, como as definições dos tipos e protótipos das funções.

NomeDoTAD.c (ou **NomeDoTAD.cpp**): Contém a implementação das funções associadas ao TAD.

conta_bancaria.h

```c
#ifndef CONTA_BANCARIA_H
#define CONTA_BANCARIA_H

typedef struct {
    int numeroConta;
    double saldo;
} ContaBancaria;

void inicializarConta(ContaBancaria *conta, int numero, double saldoInicial);
void depositar(ContaBancaria *conta, double valor);
void sacar(ContaBancaria *conta, double valor);
void mostrarSaldo(const ContaBancaria *conta);

#endif
```

conta_bancaria.c

```c
#include <stdio.h>
#include "conta_bancaria.h"

void inicializarConta(ContaBancaria *conta, int numero, double saldoInicial) {
    conta->numeroConta = numero;
    conta->saldo = saldoInicial;
}

void depositar(ContaBancaria *conta, double valor) {
    if (valor > 0) {
        conta->saldo += valor;
        printf("Depósito de %.2f realizado com sucesso.\n", valor);
    } else {
        printf("Valor de depósito inválido.\n");
    }
}

void sacar(ContaBancaria *conta, double valor) {
    if (valor > 0 && valor <= conta->saldo) {
        conta->saldo -= valor;
        printf("Saque de %.2f realizado com sucesso.\n", valor);
    } else {
        printf("Valor de saque inválido ou saldo insuficiente.\n");
    }
}

void mostrarSaldo(const ContaBancaria *conta) {
    printf("Número da Conta: %d\n", conta->numeroConta);
    printf("Saldo Atual: %.2f\n", conta->saldo);
}
```

main.c

```c
#include <stdio.h>
#include "conta_bancaria.h"

int main() {
    ContaBancaria minhaConta;

    inicializarConta(&minhaConta, 12345, 1000.00);

    mostrarSaldo(&minhaConta);

    depositar(&minhaConta, 500.00);
    sacar(&minhaConta, 200.00);

    mostrarSaldo(&minhaConta);

    return 0;
}
```

Conta Bancaria

Declaração (conta_bancaria.h): Define a estrutura ContaBancaria com dois campos: numeroConta e saldo. Inclui declarações para as funções que inicializam a conta, depositam e sacam dinheiro, e mostram o saldo.

Implementação (conta_bancaria.c): Implementa as funções declaradas no cabeçalho. As funções permitem inicializar a conta, depositar e sacar valores, e exibir o saldo da conta.

Principal (main.c): Demonstra o uso do TAD ContaBancaria. Inicializa uma conta com um saldo inicial, realiza depósitos e saques, e mostra o saldo da conta antes e depois das operações.

Atividade

Faça um programa de “gerenciamento de estoque”, utilizando variáveis globais, locais e constantes.
Variáveis Globais: Crie uma variável global para armazenar a quantidade total em estoque.
Constantes: Define uma constante para o valor inicial do estoque.
Variáveis Locais: Dentro da função main, utilize variáveis locais para armazenar a quantidade de itens adicionados e removidos.

```
Estoque inicial: 100
Digite a quantidade de itens para adicionar ao estoque: 50
Estoque após adição: 150
Digite a quantidade de itens para remover do estoque: 20
Estoque após remoção: 130
```

UNIFAVIP wyden