Notas de aula estrutura de dados (Fundamentos em C)

Sumario

Tipos de dados e tipagem

Em C, os tipos de dados e a tipagem são conceitos fundamentais para a definição de variáveis e a manipulação de dados. C é uma linguagem fortemente tipada, o que significa que cada variável deve ser declarada com um tipo específico e os tipos de dados devem ser compatíveis para operações aritméticas e lógicas.

Tipos de Dados em C

C oferece uma variedade de tipos de dados básicos e compostos, que podem ser classificados em:

Tipos Primitivos

1. **Inteiros**:
    - `int`: Tipo usado para representar números inteiros, ou seja, números sem ponto flutuante. O tamanho de `int` pode variar, mas normalmente é de 4 bytes (32 bits) em sistemas modernos.
    - `short`: Tipo inteiro com menor intervalo, geralmente 2 bytes (16 bits).
    - `long`: Tipo inteiro com maior intervalo, geralmente 4 ou 8 bytes.
    - `long long`: Para inteiros muito grandes, geralmente 8 bytes.

    Exemplos:

```c
int idade = 25;
short num_curto = 10;
long saldo = 100000L;
long long distancia = 100000000000LL;
```

2. **Ponto Flutuante**:
    - `float`: Usado para números com precisão simples (normalmente 4 bytes).
    - `double`: Usado para números com maior precisão (normalmente 8 bytes).
    - `long double`: Para maior precisão que o `double`, com tamanho variável.

    Exemplos:

```
float altura = 1.75f;
double temperatura = 36.6;
long double pi = 3.141592653589793L;
```

3. **Caracteres**:
    - `char`: Usado para armazenar um único caractere. Pode ser assinado ou não, geralmente ocupando 1 byte (8 bits).

    Exemplos:

```
char letra = 'A';
```

Tipos Compostos

1. **Arrays**:
   Arrays são coleções de elementos do mesmo tipo. Eles podem ser unidimensionais ou multidimensionais.

   Exemplos:

```
int numeros[5] = {1, 2, 3, 4, 5};
char nome[50] = "Carlos";
```

2. **Estruturas (`struct`)**:
   Estruturas permitem agrupar diferentes tipos de dados sob um único nome, possibilitando a criação de tipos compostos mais complexos.

   Exemplo:

```
struct pessoa {
    char nome[50];
    int idade;
    float altura;
};
```

```
struct pessoa pessoa1 = {"Ana", 30, 1.65};
```

3. **Uniões (`union`)**:
   As uniões permitem armazenar diferentes tipos de dados na mesma posição de memória, mas apenas um tipo pode ser armazenado por vez. Isso economiza memória.

   Exemplo:

```
union dado {
    int inteiro;
    float decimal;
    char caractere;
};

union dado valor;
valor.inteiro = 10;  // Agora 'valor' contém um inteiro
```

Tipos Pointers

Os ponteiros são variáveis que armazenam o endereço de memória de outra variável. São usados para manipular diretamente os dados na memória.

Exemplo:

```
int x = 10;
int *ptr = &x;  // Ponteiro ptr que aponta para o endereço de x

printf("Valor de x: %d\n", *ptr);  // Desreferenciando ptr para acessar o valor de x
```

Tipagem em C

C é uma linguagem com **tipagem estática** e **fortemente tipada**. Isso significa que os tipos das variáveis devem ser conhecidos no momento da compilação, e a conversão implícita de tipos entre variáveis é restrita.

Tipagem Estática

A tipagem estática significa que os tipos das variáveis são definidos no momento da declaração e não podem ser alterados durante a execução do programa.

Exemplo:

```

```

```
int a = 10;  // Tipo estático: 'a' é um inteiro
a = 20;       // Correto, pois 'a' é um inteiro
```

Tipagem Forte

Em C, a tipagem forte impede que variáveis de tipos diferentes sejam misturadas sem a devida conversão explícita.

Exemplo de erro de tipagem forte:

```
int x = 5;
float y = 3.2;
x = y;  // Erro: tipo incompatível, não é permitido atribuir float a int diretamente
```

Conversão de Tipos (Casting)

A conversão de tipos, também conhecida como **casting**, pode ser feita explicitamente ou implicitamente.

1. **Conversão Implícita (Casting Automático)**:
   Quando o compilador converte automaticamente o tipo de uma variável para um tipo mais amplo.

   Exemplo:

   ```
   int a = 5;
   double b = a;  // O valor de 'a' é automaticamente convertido para double
   ```

2. **Conversão Explícita (Casting Manual)**:
   Quando o programador converte um tipo de dados para outro tipo de dados.

   Exemplo:

   ```
   double a = 5.75;
   int b = (int) a;  // Converte explicitamente 'a' para um inteiro, truncando o valor
   printf("%d", b);  // Saída: 5
   ```

Tipos Modificadores

Em C, os modificadores podem ser usados para alterar as características dos tipos de dados, como seu tamanho ou sinal.

1. `signed` e `unsigned`:

- `signed`: Define que uma variável pode armazenar valores negativos e positivos.
- `unsigned`: Define que uma variável pode armazenar apenas valores positivos ou zero.

Exemplos:

```
unsigned int a = 10;
signed char b = -5;
```

2. `short` e `long`:
   Modificadores de tamanho. O uso de `short` e `long` pode alterar a quantidade de memória alocada para uma variável, afetando o intervalo de valores que ela pode armazenar.

   Exemplos:

```
long long int numeroGrande = 1000000000;
short int numeroPequeno = 32767;
```

Exemplos Completos

Aqui estão alguns exemplos combinando diferentes tipos e operações de tipagem:

```
#include <stdio.h>

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

int main() {
    // Tipos primitivos
    int idade = 30;
    float salario = 2500.75;
    char letra = 'A';

    // Tipos compostos (struct)
    struct Pessoa pessoa1;
    pessoa1.idade = 25;
    pessoa1.altura = 1.70;
    snprintf(pessoa1.nome, sizeof(pessoa1.nome), "João");

    // Ponteiro
    int *ptr = &idade;

    // Exibindo valores
    printf("Nome: %s, Idade: %d, Altura: %.2f\n", pessoa1.nome, pessoa1.idade,
pessoa1.altura);
```

```c
    printf("Idade através do ponteiro: %d\n", *ptr);

    // Casting
    double pi = 3.14159;
    int pi_int = (int) pi;  // Casting explícito
    printf("Valor de pi como inteiro: %d\n", pi_int);

    return 0;
}
```

A tipagem em C é fundamental para garantir que os dados sejam manipulados corretamente e de forma eficiente. Compreender os tipos de dados, a conversão de tipos e o uso de ponteiros permite que você aproveite o máximo da linguagem, criando programas robustos e com desempenho otimizado.

Variáveis

Em C, uma **variável** é um espaço de armazenamento nomeado, que pode armazenar dados temporariamente durante a execução de um programa. O tipo de dados da variável determina o tipo de valor que ela pode armazenar, como inteiros, números de ponto flutuante, caracteres, etc. Vamos entender mais sobre variáveis, como declará-las e usá-las em C, com exemplos práticos.

1. **Declaração de Variáveis**

Em C, antes de usar uma variável, você precisa declará-la, ou seja, definir seu nome e tipo. A declaração de uma variável em C segue a sintaxe:

```c
tipo nome_da_variavel;
```

Exemplo:

```c
int idade;
float salario;
char letra;
```

Aqui, declaramos:

- `idade` como uma variável do tipo `int` (inteiro),
- `salario` como uma variável do tipo `float` (número de ponto flutuante),
- `letra` como uma variável do tipo `char` (caractere).

2. **Inicialização de Variáveis**

Após declarar uma variável, você pode atribuir um valor a ela. Isso pode ser feito na própria declaração ou em uma linha separada. A atribuição de valor é feita com o operador `=`.

Exemplo:

```c
int idade = 30;
float salario = 5000.75;
char letra = 'A';
```

Agora temos:

- A variável `idade` foi inicializada com 30,
- A variável `salario` foi inicializada com 5000.75,
- A variável `letra` foi inicializada com o caractere 'A'.

3. **Tipos de Variáveis em C**

Existem vários tipos de variáveis em C, e cada tipo ocupa uma quantidade diferente de memória. Aqui estão alguns tipos comuns:

- **int**: armazena números inteiros, como -10, 0, 25.
- **float**: armazena números de ponto flutuante (decimais), como 3.14, 0.01.
- **double**: similar ao `float`, mas com maior precisão.
- **char**: armazena um único caractere, como 'a', 'b', '1', etc.
- **bool** (em C99 e versões posteriores): armazena valores lógicos, `true` ou `false`.

Exemplo de uso de diferentes tipos:

```c
int numero = 100;
float media = 7.8;
double pi = 3.14159265359;
char grade = 'A';
```

4. **Escopo das Variáveis**

O **escopo** de uma variável determina onde ela pode ser acessada no código. Existem dois tipos principais de escopo:

- **Variáveis locais**: São declaradas dentro de uma função e só podem ser usadas dentro dessa função.
- **Variáveis globais**: São declaradas fora de todas as funções, no início do programa, e podem ser acessadas por qualquer função.

Exemplo de variável local:

```c
void exemplo() {
    int x = 5;  // variável local
    printf("%d\n", x);  // válida aqui
}

int main() {
```

```
    exemplo();
    // printf("%d\n", x);  // erro! x não é acessível aqui
    return 0;
}
```

Exemplo de variável global:

```
int x = 10;  // variável global

void exemplo() {
    printf("%d\n", x);  // válida aqui
}

int main() {
    exemplo();  // válida aqui também
    return 0;
}
```

5. **Constantes em C**

Em C, você pode usar a palavra-chave `const` para criar variáveis cujos valores não podem ser alterados após a inicialização. Isso é útil para definir valores fixos no programa, como a constante matemática `PI`.

Exemplo:

```
const float PI = 3.14159;
```

6. **Arrays**

Um **array** é uma estrutura que permite armazenar múltiplos valores do mesmo tipo em uma única variável. Você pode acessar os valores em um array usando índices.

Exemplo de array de inteiros:

```
int numeros[5] = {1, 2, 3, 4, 5};
printf("%d\n", numeros[0]);  // Imprime 1
```

7. **Referências e Ponteiros**

Em C, uma variável pode armazenar o **endereço de memória** de outra variável através de um **ponteiro**. Isso é útil para manipulação direta de dados em memória.

Exemplo de ponteiro:

```c
int num = 10;
int *ptr = &num;  // Ponteiro ptr armazena o endereço de num
printf("%d\n", *ptr);  // Imprime 10, o valor de num
```

8. **Modificando Variáveis**

Você pode modificar o valor de uma variável a qualquer momento, a não ser que ela seja `const`. A modificação é feita diretamente ou através de operações aritméticas.

Exemplo de modificação:

```c
int a = 5;
a = a + 10;  // Agora a vale 15
printf("%d\n", a);  // Imprime 15
```

Variáveis em C são essenciais para armazenar e manipular dados durante a execução do programa. Compreender os tipos, escopos e como usá-las corretamente é fundamental para escrever programas eficientes e corretos.

Conversão Explícita (Casting)

Conversão de Tipos em C

Introdução

A **conversão de tipos** em C refere-se ao processo de transformar um dado de um tipo para outro. Esse processo pode ocorrer de forma **implícita** ou **explícita** e desempenha um papel fundamental na manipulação eficiente de variáveis e operações matemáticas. Segundo Kernighan e Ritchie (1988), a conversão de tipos é essencial para evitar perda de dados e garantir que operações entre diferentes tipos sejam realizadas corretamente.

Tipos de Conversão

1. Conversão Implícita (Type Promotion)

A conversão implícita ocorre automaticamente quando valores de diferentes tipos são usados em expressões. O compilador converte o tipo menor para um tipo maior para evitar perda de precisão.

Exemplo em C:

```c
#include <stdio.h>

int main() {
    int inteiro = 10;
```

```c
    float decimal = inteiro;  // Conversão implícita de int para float

    printf("Valor de decimal: %f\n", decimal);
    return 0;
}
```

2. Conversão Explícita (Type Casting)

A conversão explícita, também chamada de **type casting**, é quando o programador especifica manualmente a conversão de um tipo para outro usando o operador de cast `(tipo) valor`.

Exemplo em C:

```c
#include <stdio.h>

int main() {
    float num = 5.75;
    int inteiro = (int) num;  // Conversão explícita de float para int

    printf("Valor de inteiro: %d\n", inteiro);  // Saída: 5
    return 0;
}
```

Aqui, `num` (float) foi convertido para um inteiro, descartando a parte decimal.

Promoção de Tipos em Expressões

Em expressões matemáticas, os tipos de dados podem ser promovidos para um tipo maior para evitar perda de precisão. A hierarquia de promoção geralmente segue esta ordem:

1. `char`, `short` → `int`
2. `int` → `float`
3. `float` → `double`

Exemplo:

```c
#include <stdio.h>

int main() {
    int a = 5;
    float b = 2.5;
    float resultado = a + b;  // 'a' é promovido para float

    printf("Resultado: %f\n", resultado);
    return 0;
}
```

Possíveis Problemas com Conversão de Tipos

Embora a conversão de tipos seja útil, ela pode causar perda de precisão e comportamento inesperado:

- **Truncamento de dados**: Converter `float` para `int` remove a parte decimal.
- **Overflows**: Converter um valor maior para um tipo menor pode levar a perda de dados.
- **Conversões inseguras**: Usar `unsigned` e `signed` incorretamente pode causar resultados inesperados.

Exemplo de problema:

```c
#include <stdio.h>

int main() {
    unsigned int x = -10;  // Conversão insegura
    printf("Valor de x: %u\n", x);  // Comportamento indefinido
    return 0;
}
```

A conversão de tipos é um mecanismo essencial em C, mas deve ser utilizada com cautela para evitar perda de dados e bugs difíceis de depurar. A conversão implícita pode ser útil em muitas situações, mas a conversão explícita deve ser usada sempre que houver risco de truncamento ou overflow.

Struct

Em C, a palavra-chave `struct` (abreviação de "structure") permite definir um novo tipo de dado composto, que pode agrupar variáveis de diferentes tipos sob um único nome. Essa funcionalidade é essencial para organizar e manipular dados de forma mais estruturada.

Segundo Harbison e Steele (1995), "as estruturas em C oferecem um meio eficaz de agrupar dados relacionados, tornando o código mais organizado".

Definição de `struct`

A sintaxe básica para definir uma estrutura é:

```c
struct NomeDaEstrutura {
    tipo1 nome_variavel1;
    tipo2 nome_variavel2;
    ...
};
```

A `struct` serve como um molde para criar variáveis que armazenam múltiplos dados relacionados.

Exemplo Básico: Estrutura para Representar um Aluno

```c
#include <stdio.h>

// Definição da struct
struct Aluno {
    char nome[50];
    int idade;
    float nota;
};

int main() {
    struct Aluno aluno1; // Declaração de uma variável do tipo struct Aluno

    // Atribuição de valores
    printf("Digite o nome do aluno: ");
    scanf("%49s", aluno1.nome);

    printf("Digite a idade do aluno: ");
    scanf("%d", &aluno1.idade);

    printf("Digite a nota do aluno: ");
    scanf("%f", &aluno1.nota);

    // Exibição dos dados
    printf("\nDados do Aluno:\n");
    printf("Nome: %s\n", aluno1.nome);
    printf("Idade: %d\n", aluno1.idade);
    printf("Nota: %.2f\n", aluno1.nota);

    return 0;
}
```

Uso de `typedef` para Simplificar a Sintaxe

O `typedef` permite definir um alias para uma estrutura, tornando a declaração mais legível:

```c
#include <stdio.h>

typedef struct {
    char nome[50];
    int idade;
    float nota;
} Aluno;

int main() {
    Aluno aluno1 = {"Carlos", 20, 9.5}; // Inicialização direta
```

```
    printf("Nome: %s\n", aluno1.nome);
    printf("Idade: %d\n", aluno1.idade);
    printf("Nota: %.2f\n", aluno1.nota);

    return 0;
}
```

Structs com Ponteiros

Em C, podemos usar ponteiros para acessar e modificar dados dentro de uma estrutura:

```
#include <stdio.h>

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

void modificarIdade(Pessoa *p, int novaIdade) {
    p->idade = novaIdade; // Uso do operador '->' para acessar membros do
ponteiro
}

int main() {
    Pessoa pessoa1 = {"Ana", 25};
    printf("Idade antes: %d\n", pessoa1.idade);

    modificarIdade(&pessoa1, 30);
    printf("Idade depois: %d\n", pessoa1.idade);

    return 0;
}
```

Structs Dentro de Structs (Aninhadas)

Uma `struct` pode conter outra `struct` como membro:

```
#include <stdio.h>

typedef struct {
    char rua[50];
    int numero;
} Endereco;

typedef struct {
    char nome[50];
```

```
    Endereco endereco;
} Pessoa;

int main() {
    Pessoa pessoa1 = {"Roberto", {"Rua A", 123}};

    printf("Nome: %s\n", pessoa1.nome);
    printf("Endereço: %s, %d\n", pessoa1.endereco.rua,
pessoa1.endereco.numero);

    return 0;
}
```

Uso de Structs com Arrays

Podemos criar um array de `structs` para armazenar múltiplas instâncias:

```
#include <stdio.h>

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

int main() {
    Pessoa pessoas[2]; // Array de structs

    for (int i = 0; i < 2; i++) {
        printf("Digite o nome da pessoa %d: ", i + 1);
        scanf("%49s", pessoas[i].nome);
        printf("Digite a idade da pessoa %d: ", i + 1);
        scanf("%d", &pessoas[i].idade);
    }

    printf("\nLista de Pessoas:\n");
    for (int i = 0; i < 2; i++) {
        printf("Nome: %s, Idade: %d\n", pessoas[i].nome, pessoas[i].idade);
    }

    return 0;
}
```

As `structs` são fundamentais para organizar dados de forma eficiente em C. Elas permitem agrupar múltiplos valores em uma única unidade lógica, facilitando a manipulação e o entendimento do código. Seja em aplicações simples ou complexas, o uso adequado de `structs` melhora a organização e a legibilidade dos programas.

Constantes

Em C, **constantes** são valores que não podem ser modificados durante a execução do programa. Elas são úteis para garantir que determinados valores permaneçam inalterados e para tornar o código mais legível e fácil de manter.

1. **Constantes Literais**

São valores fixos diretamente inseridos no código.

Exemplos:

```
int idade = 25;         // 25 é uma constante literal inteira
float pi = 3.14;        // 3.14 é uma constante literal de ponto flutuante
char letra = 'A';       // 'A' é uma constante literal de caractere
```

2. **Constantes `#define` (Macros)**

Utiliza-se a diretiva `#define` para criar constantes antes da compilação.

Exemplo:

```
#include <stdio.h>

#define PI 3.14159
#define TAMANHO 10

int main() {
    printf("O valor de PI é: %f\n", PI);
    printf("Tamanho: %d\n", TAMANHO);
    return 0;
}
```

Aqui, `PI` e `TAMANHO` são substituídos por seus valores durante a pré-processamento.

3. **Constantes `const`**

A palavra-chave `const` define uma variável que não pode ser alterada.

Exemplo:

```
#include <stdio.h>

int main() {
    const int NUMERO = 100; // Constante inteira
    printf("Valor: %d\n", NUMERO);
```

```
    // NUMERO = 200; // Isso geraria um erro, pois NUMERO é constante

    return 0;
}
```

4. **Constantes `enum`**

Uma `enum` cria um conjunto de valores inteiros nomeados.

Exemplo:

```
#include <stdio.h>

enum Dias {DOMINGO, SEGUNDA, TERÇA, QUARTA, QUINTA, SEXTA, SÁBADO};

int main() {
    enum Dias hoje = QUARTA;
    printf("O valor de hoje é: %d\n", hoje);
    return 0;
}
```

Aqui, `DOMINGO` começa com 0, `SEGUNDA` com 1, e assim por diante.

5. **Constantes `const` com Ponteiros**

Se `const` for aplicado a um ponteiro, a restrição pode ser sobre o valor apontado ou o próprio ponteiro.

- **Valor constante:**

```
const int x = 10; // x não pode ser modificado
const int *ptr = &x; // ptr pode apontar para outro endereço, mas o valor não pode ser modificado
```

- **Ponteiro constante:**

```
int y = 20;
int *const ptr2 = &y; // ptr2 não pode mudar de endereço, mas o valor pode ser alterado
```

- **Ponteiro e valor constantes:**

```
const int z = 30;
const int *const ptr3 = &z; // Nem o endereço nem o valor podem ser modificados
```

Constantes são fundamentais para manter a integridade dos dados e melhorar a legibilidade do código. Elas podem ser implementadas de diversas formas, como `#define`, `const`, e `enum`, dependendo da necessidade do projeto.

Booleano

Em C, os tipos booleanos não são nativos, como em algumas linguagens de programação modernas (como Python ou JavaScript). Contudo, o conceito de valores booleanos (verdadeiro e falso) pode ser implementado utilizando tipos de dados inteiros.

1. Definição e Conceito

Em C, os valores booleanos podem ser representados utilizando o tipo `int`. Embora o C não tenha um tipo explícito para booleanos, qualquer valor diferente de zero é tratado como **verdadeiro** (true), enquanto o valor **0** é tratado como **falso** (false). Isso significa que:

- **0** → Falso
- **Qualquer número diferente de 0** → Verdadeiro

A partir do C99, foi introduzido o cabeçalho `<stdbool.h>`, que oferece a possibilidade de usar as palavras-chave `true` e `false` de maneira mais legível e compreensível.

2. Usando o Tipo `bool` com o `<stdbool.h>`

O cabeçalho `<stdbool.h>` define um tipo `bool`, que é essencialmente um `int`, e os valores `true` e `false` são definidos como 1 e 0, respectivamente.

Exemplo:

```c
#include <stdio.h>
#include <stdbool.h>

int main() {
    bool isTrue = true;
    bool isFalse = false;

    if (isTrue) {
        printf("A variável isTrue é verdadeira!\n");
    }

    if (!isFalse) {
        printf("A variável isFalse é falsa!\n");
    }

    return 0;
}
```

3. Operações Booleanas

As operações booleanas em C são feitas com os operadores lógicos, como:

- `&&` (E lógico)
- `||` (OU lógico)
- `!` (NEGAÇÃO)

Exemplos:

1. **Operador `&&` (E lógico):**
   - Retorna verdadeiro apenas se **ambos os operandos forem verdadeiros**.

```c
#include <stdio.h>
#include <stdbool.h>

int main() {
    bool a = true;
    bool b = false;

    if (a && b) {
        printf("Ambos são verdadeiros.\n");
    } else {
        printf("Pelo menos um é falso.\n");
    }

    return 0;
}
```

2. **Operador `||` (OU lógico):**
   - Retorna verdadeiro se **pelo menos um dos operandos for verdadeiro**.

```c
#include <stdio.h>
#include <stdbool.h>

int main() {
    bool a = true;
    bool b = false;

    if (a || b) {
        printf("Pelo menos um é verdadeiro.\n");
    }

    return 0;
}
```

3. **Operador `!` (NEGAÇÃO):**

- Retorna verdadeiro se o valor for falso, e vice-versa.

```c
#include <stdio.h>
#include <stdbool.h>

int main() {
    bool a = false;

    if (!a) {
        printf("A variável a é falsa, então a negação é verdadeira.\n");
    }

    return 0;
}
```

4. Comparações Booleanas

Em C, comparações geralmente envolvem operadores relacionais como:

- `==` (igual a)
- `!=` (diferente de)
- `>` (maior que)
- `<` (menor que)
- `>=` (maior ou igual a)
- `<=` (menor ou igual a)

Esses operadores podem ser usados para gerar valores booleanos (verdadeiro ou falso).

Exemplo:

```c
#include <stdio.h>
#include <stdbool.h>

int main() {
    int x = 10;
    int y = 5;

    bool result = (x > y);  // Comparação x > y, que retorna verdadeiro

    if (result) {
        printf("x é maior que y.\n");
    } else {
        printf("x não é maior que y.\n");
    }

    return 0;
}
```

5. Resumo

- **Tipo `bool`:** Usado para representar valores booleanos em C, definido em `<stdbool.h>`.
- **Valores Booleanos:** `true` (1) e `false` (0).
- **Operadores Lógicos:**
  - `&&` (E lógico)
  - `||` (OU lógico)
  - `!` (NEGAÇÃO)
- **Operadores Relacionais:** Usados para gerar valores booleanos a partir de comparações.

Essa abordagem facilita a manipulação de condições lógicas em programas C, permitindo o controle de fluxo por meio de valores booleanos.

Variáveis Estáticas

Variáveis `static` podem ser **globais** (acessíveis dentro do arquivo) ou **locais** (persistem entre chamadas da função).

Exemplo de variável `static` local:

```
void contador() {
    static int count = 0;
    count++;
    printf("%d\n", count);
}
```

Cada chamada da função preserva o valor de `count`.

Segundo Harbison e Steele (1995), "o uso de variáveis estáticas reduz o uso de memória e melhora a modularidade do código".

Escopo

Escopo de Variáveis em C

Introdução

O escopo de uma variável em C determina onde essa variável pode ser acessada dentro do programa. Entender os diferentes tipos de escopo é essencial para evitar erros e melhorar a eficiência do código. Segundo Kernighan e Ritchie (1988), o escopo e o tempo de vida de variáveis são aspectos fundamentais na estrutura de um programa em C.

Tipos de Escopo

Em C, as variáveis podem ter diferentes tipos de escopo:

1. Escopo de Bloco (Local)

Variáveis declaradas dentro de um bloco {} são acessíveis apenas dentro desse bloco. Esse é o escopo mais comum e ajuda a evitar conflitos de nome.

Exemplo:

```c
#include <stdio.h>

void funcao() {
    int x = 10; // Variável local
    printf("Dentro da função: %d\n", x);
}

int main() {
    funcao();
    // printf("Fora da função: %d\n", x); // Erro: x não é visível aqui
    return 0;
}
```

2. Escopo de Arquivo (Global)

Variáveis declaradas fora de qualquer função possuem escopo global e podem ser acessadas por qualquer parte do código.

Exemplo:

```c
#include <stdio.h>

int global = 100; // Variável global

void funcao() {
    printf("Dentro da função: %d\n", global);
}

int main() {
    funcao();
    printf("No main: %d\n", global);
    return 0;
}
```

Embora variáveis globais sejam úteis, seu uso excessivo pode dificultar a manutenção e depuração do código.

3. Escopo de Função (Parâmetros)

Os parâmetros de uma função possuem escopo local à própria função, garantindo que seus valores não sejam alterados por outras partes do programa.

Exemplo:

```c
#include <stdio.h>

void soma(int a, int b) { // a e b têm escopo de função
    printf("Soma: %d\n", a + b);
}

int main() {
    soma(5, 3);
    // printf("%d", a); // Erro: a não está acessível aqui
    return 0;
}
```

4. Escopo de Bloco com `static`

Variáveis `static` dentro de funções mantêm seu valor entre chamadas da função, mas ainda possuem escopo local.

Exemplo:

```c
#include <stdio.h>

void contador() {
    static int count = 0; // Mantém o valor entre chamadas
    count++;
    printf("Contador: %d\n", count);
}

int main() {
    contador();
    contador();
    contador();
    return 0;
}
```

5. Escopo de Arquivo com `static`

Variáveis `static` globais são limitadas ao arquivo onde foram declaradas, evitando conflitos de nomes com outras partes do programa.

Exemplo:

```c
#include <stdio.h>

static int restrita = 42; // Só pode ser acessada neste arquivo
```

```c
void mostrar() {
    printf("Valor: %d\n", restrita);
}

int main() {
    mostrar();
    return 0;
}
```

Considerações Finais

Compreender o escopo de variáveis em C é crucial para evitar erros de acesso, conflitos de nomes e vazamento de memória. Segundo Kernighan e Ritchie (1988), o uso disciplinado do escopo melhora a modularidade do código e reduz efeitos colaterais inesperados.

Operadores em C

Os operadores em C são símbolos que instruem o compilador a realizar operações matemáticas, lógicas ou de manipulação de bits.

Operadores Aritméticos

Executam operações matemáticas básicas:

```c
int soma = 10 + 5; // Adição
int subtracao = 10 - 5; // Subtração
int multiplicacao = 10 * 5; // Multiplicação
int divisao = 10 / 5; // Divisão
int resto = 10 % 3; // Módulo
```

Operadores Relacionais

Comparam valores e retornam verdadeiro (`1`) ou falso (`0`):

```c
if (a == b) // Igual
if (a != b) // Diferente
if (a > b)  // Maior que
if (a < b)  // Menor que
if (a >= b) // Maior ou igual
if (a <= b) // Menor ou igual
```

Operadores Lógicos

Usados para expressões condicionais:

```c
if (a > 5 && b < 10) // AND lógico
if (a > 5 || b < 10) // OR lógico
if (!(a > 5))        // NOT lógico
```

Operadores Bitwise

Operam diretamente nos bits dos números:

```c
int resultado = a & b;  // AND bitwise
int resultado = a | b;  // OR bitwise
int resultado = a ^ b;  // XOR bitwise
int resultado = ~a;     // NOT bitwise
int resultado = a << 2; // Shift left
int resultado = a >> 2; // Shift right
```

Operadores de Atribuição

Atribuem valores a variáveis:

```c
a += 5; // Equivalente a: a = a + 5;
a -= 5; // Equivalente a: a = a - 5;
a *= 5; // Equivalente a: a = a * 5;
a /= 5; // Equivalente a: a = a / 5;
a %= 5; // Equivalente a: a = a % 5;
```

De acordo com Deitel e Deitel (2016), "os operadores em C são fundamentais para manipulação de dados e controle de fluxo do programa".

Estruturas de Controle de Fluxo e Laços de Repetição em C

As **estruturas de controle de fluxo** e **laços de repetição** são fundamentais na programação, pois permitem a tomada de decisões e a execução repetitiva de blocos de código. Segundo Aho, Hopcroft e Ullman (1983), o controle eficiente do fluxo de execução é essencial para a construção de algoritmos otimizados e legíveis.

Estruturas Condicionais

O **controle condicional** em C é um dos pilares fundamentais para a tomada de decisões dentro de um programa. Ele permite que diferentes blocos de código sejam executados dependendo de certas condições. As estruturas de controle condicional mais comuns são o **if**, o **else** e o **switch**.

1. **Estrutura `if`**

A estrutura `if` é usada para executar um bloco de código apenas se uma condição for verdadeira.

Sintaxe:

```
if (condição) {
    // Bloco de código a ser executado se a condição for verdadeira
}
```

Exemplo:

```
#include <stdio.h>

int main() {
    int idade = 18;

    if (idade >= 18) {
        printf("Você é maior de idade.\n");
    }

    return 0;
}
```

Explicação: O programa verifica se a variável `idade` é maior ou igual a 18 e, se for, imprime "Você é maior de idade.".

2. **Estrutura `if-else`**

A estrutura `if-else` é usada quando você precisa definir dois caminhos de execução: um para o caso de a condição ser verdadeira e outro para quando ela for falsa.

Sintaxe:

```
if (condição) {
    // Bloco de código a ser executado se a condição for verdadeira
} else {
    // Bloco de código a ser executado se a condição for falsa
}
```

Exemplo:

```
#include <stdio.h>

int main() {
    int idade = 16;
```

```
    if (idade >= 18) {
        printf("Você é maior de idade.\n");
    } else {
        printf("Você é menor de idade.\n");
    }

    return 0;
}
```

Explicação: O programa verifica se a idade é maior ou igual a 18. Se for, imprime "Você é maior de idade", caso contrário, imprime "Você é menor de idade".

3. **Estrutura `if-else if-else`**

O `if-else if-else` permite verificar múltiplas condições. Ele é útil quando você tem várias opções e deseja verificar várias condições sequenciais.

Sintaxe:

```
if (condição1) {
    // Bloco de código para condição1
} else if (condição2) {
    // Bloco de código para condição2
} else {
    // Bloco de código para quando todas as condições forem falsas
}
```

Exemplo:

```
#include <stdio.h>

int main() {
    int nota = 75;

    if (nota >= 90) {
        printf("Aprovado com Distinção.\n");
    } else if (nota >= 70) {
        printf("Aprovado.\n");
    } else {
        printf("Reprovado.\n");
    }

    return 0;
}
```

Explicação: O programa verifica a nota do aluno e imprime "Aprovado com Distinção", "Aprovado" ou "Reprovado" de acordo com a faixa da nota.

4. **Estrutura `switch`**

O `switch` é uma estrutura de controle que permite comparar uma variável com vários valores possíveis. Ele é ideal quando você tem muitas opções diferentes para um mesmo valor, mas não deseja usar múltiplos `if-else`.

Sintaxe:

```
switch (expressão) {
    case valor1:
        // Bloco de código para valor1
        break;
    case valor2:
        // Bloco de código para valor2
        break;
    default:
        // Bloco de código para quando nenhum valor for correspondido
}
```

Exemplo:

```
#include <stdio.h>

int main() {
    int dia = 3;

    switch (dia) {
        case 1:
            printf("Domingo\n");
            break;
        case 2:
            printf("Segunda-feira\n");
            break;
        case 3:
            printf("Terça-feira\n");
            break;
        default:
            printf("Dia inválido\n");
            break;
    }

    return 0;
}
```

Explicação: O programa verifica o valor de `dia` e imprime o dia da semana correspondente. Se não encontrar nenhuma correspondência, imprime "Dia inválido".

5. **Operador Ternário**

Embora não seja uma estrutura condicional completa, o operador ternário (`?:`) permite uma forma compacta de realizar uma verificação condicional, retornando um valor com base em uma condição.

Sintaxe:

```
condição ? valor_se_verdadeiro : valor_se_falso;
```

Exemplo:

```c
#include <stdio.h>

int main() {
    int idade = 18;

    printf("%s\n", idade >= 18 ? "Maior de idade" : "Menor de idade");

    return 0;
}
```

Explicação: O operador ternário verifica se a idade é maior ou igual a 18 e imprime a mensagem correspondente.

O controle condicional em C é essencial para que o programa tome decisões baseadas em dados variáveis. O `if`, `else`, `switch` e o operador ternário são ferramentas poderosas para controlar o fluxo de execução de um programa, tornando-o mais flexível e dinâmico.

Laços de Repetição

Os **laços de repetição** (também chamados de **estruturas de repetição**) em C são usados para executar um bloco de código várias vezes, com base em uma condição. Eles são fundamentais para a automação de tarefas repetitivas, como percorrer listas ou realizar cálculos múltiplos. Em C, temos três tipos principais de laços de repetição: **for**, **while** e **do-while**.

1. **Laço `for`**

O laço `for` é o mais comum quando você sabe de antemão o número de repetições. Ele tem a seguinte estrutura:

```c
for (inicialização; condição; atualização) {
    // Bloco de código
```

```
}
```

- **Inicialização**: Define a variável de controle e a inicializa.
- **Condição**: Define a condição de continuidade. O laço continua enquanto esta condição for verdadeira.
- **Atualização**: Define como a variável de controle é alterada a cada iteração.

Exemplo de `for`:

```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 5; i++) {
        printf("Valor de i: %d\n", i);
    }
    return 0;
}
```

Saída:

```
Valor de i: 1
Valor de i: 2
Valor de i: 3
Valor de i: 4
Valor de i: 5
```

Neste exemplo, o laço `for` inicia com `i = 1` e continua até que `i` seja maior que 5. A cada iteração, `i` é incrementado em 1.

2. Laço `while`

O laço `while` é usado quando não se sabe exatamente quantas vezes o bloco de código será executado, mas se tem uma condição que precisa ser verificada antes de cada execução.

A estrutura básica do `while` é:

```
while (condição) {
    // Bloco de código
}
```

- A condição é verificada antes de cada execução do bloco. Se for verdadeira, o código dentro do laço é executado.
- Se a condição for falsa logo de início, o código dentro do laço nunca será executado.

Exemplo de `while`:

```c
#include <stdio.h>

int main() {
    int i = 1;
    while (i <= 5) {
        printf("Valor de i: %d\n", i);
        i++;  // Incrementa i
    }
    return 0;
}
```

Saída:

```
Valor de i: 1
Valor de i: 2
Valor de i: 3
Valor de i: 4
Valor de i: 5
```

Neste exemplo, o laço `while` repete o bloco de código enquanto `i` for menor ou igual a 5. A cada iteração, o valor de `i` é incrementado.

3. **Laço `do-while`**

O laço `do-while` é semelhante ao `while`, mas a diferença é que ele executa o bloco de código **pelo menos uma vez**, mesmo que a condição inicial já seja falsa.

A estrutura básica do `do-while` é:

```c
do {
    // Bloco de código
} while (condição);
```

- O código dentro do laço é executado primeiro e a condição é verificada **após** a execução.
- Se a condição for verdadeira, o laço continua. Se for falsa, o laço termina.

Exemplo de `do-while`:

```c
#include <stdio.h>

int main() {
    int i = 1;
```

```c
    do {
        printf("Valor de i: %d\n", i);
        i++;  // Incrementa i
    } while (i <= 5);
    return 0;
}
```

Saída:

```
Valor de i: 1
Valor de i: 2
Valor de i: 3
Valor de i: 4
Valor de i: 5
```

Aqui, o laço `do-while` também imprime os valores de `i` de 1 a 5, mas a diferença é que o bloco de código é executado **pelo menos uma vez**.

4. **Laços de repetição aninhados**

Você pode usar laços de repetição dentro de outros laços. Isso é útil quando você precisa realizar uma operação em uma matriz ou em um conjunto de dados bidimensionais.

Exemplo de laços aninhados:

```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 3; i++) {
        for (int j = 1; j <= 3; j++) {
            printf("i = %d, j = %d\n", i, j);
        }
    }
    return 0;
}
```

Saída:

```
i = 1, j = 1
i = 1, j = 2
i = 1, j = 3
i = 2, j = 1
i = 2, j = 2
i = 2, j = 3
i = 3, j = 1
```

```
i = 3, j = 2
i = 3, j = 3
```

Aqui, temos um laço `for` dentro de outro. O laço externo percorre os valores de `i` e, para cada valor de `i`, o laço interno percorre os valores de `j`.

5. **Comandos de controle dentro dos laços**

Além da estrutura básica de um laço, existem dois comandos importantes para controlar o fluxo de execução:

- `break`: Interrompe imediatamente o laço.
- `continue`: Faz com que a próxima iteração do laço seja iniciada, pulando o restante do bloco de código.

Exemplo com `break` e `continue`:

```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            break;  // Interrompe o laço quando i for 3
        }
        if (i == 2) {
            continue;  // Pula o restante do código para i igual a 2
        }
        printf("Valor de i: %d\n", i);
    }
    return 0;
}
```

Saída:

```
Valor de i: 1
Valor de i: 4
Valor de i: 5
```

Neste exemplo, quando `i` é igual a 2, o comando `continue` faz com que o laço pule a impressão. Quando `i` chega a 3, o comando `break` interrompe o laço completamente.

Os laços de repetição são uma das ferramentas mais poderosas da programação em C. Eles permitem a execução repetitiva de blocos de código com base em uma condição, ajudando a resolver problemas que envolvem iteração, como processamento de listas, arrays, ou qualquer situação em que uma tarefa precise ser repetida múltiplas vezes. Conhecer os três tipos de laços (`for`, `while`, `do-while`) e como usá-los de maneira eficaz é essencial para qualquer programador.

Estruturas de Saltos na Programação em C

As **estruturas de saltos** são mecanismos utilizados para alterar o fluxo normal de execução de um programa, permitindo que ele continue de um ponto diferente no código. Essas estruturas são essenciais para a implementação de controle de fluxo avançado e são amplamente utilizadas em linguagens de programação para manipulação eficiente de loops e tomadas de decisão. Segundo Aho, Hopcroft e Ullman (1983), o controle eficiente do fluxo de execução é crucial para a otimização de algoritmos.

Tipos de Estruturas de Saltos

As estruturas de saltos mais comuns em C incluem `break`, `continue`, `return`, e `goto`. Cada uma tem uma função específica e seu uso pode impactar a legibilidade e eficiência do código.

1. `break`

A instrução `break` interrompe a execução de um laço de repetição (`for`, `while` ou `do-while`) antes que a condição de término seja atingida.

Exemplo em C:

```c
#include <stdio.h>

int main() {
    for (int i = 0; i < 10; i++) {
        if (i == 5) {
            break; // Interrompe o laço quando i for 5
        }
        printf("%d\n", i);
    }
    return 0;
}
```

O `break` é útil para sair de loops quando uma condição específica é satisfeita, evitando execuções desnecessárias.

2. `continue`

A instrução `continue` faz com que a execução do loop salte para a próxima iteração, ignorando as instruções subsequentes dentro do bloco do laço para aquela iteração específica.

Exemplo em C:

```c
#include <stdio.h>

int main() {
    for (int i = 0; i < 10; i++) {
        if (i % 2 == 0) {
```

```
            continue; // Pula a iteração para números pares
        }
        printf("%d\n", i);
    }
    return 0;
}
```

Isso é útil quando se deseja ignorar certos valores sem interromper completamente o loop.

3. `return`

A instrução `return` é utilizada dentro de funções para encerrar sua execução e, opcionalmente, retornar um valor ao chamador.

Exemplo em C:

```
#include <stdio.h>

int quadrado(int x) {
    return x * x; // Retorna o quadrado de x
}

int main() {
    int resultado = quadrado(4);
    printf("%d\n", resultado); // Saída: 16
    return 0;
}
```

Usar `return` melhora a modularidade do código, permitindo a reutilização eficiente de funções.

4. `goto`

O `goto` permite saltos arbitrários para outras partes do código, mas seu uso é desencorajado devido à criação do chamado "código espaguete", tornando o programa difícil de ler e manter.

Exemplo em C:

```
#include <stdio.h>

int main() {
    int x = 0;

    inicio:
        printf("Valor de x: %d\n", x);
        x++;
        if (x < 5) goto inicio;  // Salta para o rótulo 'inicio'
```

```
    return 0;
}
```

Apesar de sua flexibilidade, `goto` raramente é recomendado em linguagens modernas devido à dificuldade de depuração.

Ponteiros em C

Ponteiros são uma das características mais poderosas e complexas da linguagem C. Eles são variáveis que armazenam o endereço de memória de outra variável. Ao invés de armazenar um valor diretamente, um ponteiro contém a localização na memória onde esse valor está guardado.

1. Declaração de Ponteiros

Um ponteiro é declarado usando o asterisco (`*`). O tipo do ponteiro deve corresponder ao tipo da variável que ele aponta, ou seja, um ponteiro para um `int` deve ser do tipo `int*`.

Exemplo de declaração de ponteiro:

```
int *ptr;  // Ponteiro para inteiro
```

2. Inicialização de Ponteiros

Para inicializar um ponteiro, usamos o operador de endereço (`&`). Esse operador retorna o endereço de memória de uma variável.

Exemplo:

```
int x = 10;
int *ptr = &x;  // Ponteiro ptr armazena o endereço de memória de x
```

Aqui, `ptr` aponta para a variável `x`, ou seja, ele armazena o endereço de memória onde `x` está localizado.

3. Acessando o Valor Apontado

Para acessar o valor armazenado no endereço de memória apontado pelo ponteiro, usamos o operador de desreferência (`*`). O operador `*` permite acessar o valor da variável à qual o ponteiro se refere.

Exemplo:

```
#include <stdio.h>

int main() {
    int x = 10;
```

```c
    int *ptr = &x;

    printf("Endereço de x: %p\n", ptr);   // Exibe o endereço de memória
    printf("Valor de x: %d\n", *ptr);     // Exibe o valor de x, que é 10

    return 0;
}
```

Neste exemplo, `*ptr` desreferencia o ponteiro `ptr`, retornando o valor armazenado em `x`, que é `10`.

4. Ponteiros e Arrays

Em C, o nome de um array é, na verdade, um ponteiro para o seu primeiro elemento. Isso significa que você pode acessar elementos do array usando ponteiros.

Exemplo com array:

```c
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int *ptr = arr;  // O nome do array é um ponteiro para o primeiro elemento

    printf("Primeiro valor: %d\n", *ptr);      // Exibe 1
    printf("Segundo valor: %d\n", *(ptr + 1)); // Exibe 2

    return 0;
}
```

Aqui, `ptr + 1` aponta para o segundo elemento do array, e o operador `*` desreferencia esse ponteiro, retornando o valor 2.

5. Ponteiros para Ponteiros

Você pode ter ponteiros para ponteiros, ou seja, ponteiros que armazenam o endereço de outro ponteiro. Isso é útil quando você precisa manipular ponteiros em funções, por exemplo.

Exemplo:

```c
#include <stdio.h>

int main() {
    int x = 10;
    int *ptr1 = &x;
    int **ptr2 = &ptr1;  // Ponteiro para ponteiro

    printf("Valor de x: %d\n", **ptr2);  // Desreferencia ptr2 duas vezes para
acessar x
```

```
    return 0;
}
```

Aqui, `**ptr2` primeiro desreferencia `ptr2` para obter `ptr1` e depois desreferencia `ptr1` para acessar o valor de `x`.

6. Ponteiros e Funções

Ponteiros são frequentemente usados para passar argumentos por referência para funções, o que permite que a função altere o valor da variável original.

Exemplo de função que usa ponteiros:

```
#include <stdio.h>

void alterarValor(int *ptr) {
    *ptr = 20;  // Modifica o valor de x diretamente
}

int main() {
    int x = 10;
    printf("Valor antes: %d\n", x);

    alterarValor(&x);  // Passa o endereço de x para a função

    printf("Valor depois: %d\n", x);  // Exibe 20

    return 0;
}
```

Aqui, a função `alterarValor` modifica diretamente o valor de `x` através do ponteiro passado como argumento.

7. Ponteiros Nulos

Um ponteiro nulo é um ponteiro que não aponta para nenhuma localização válida de memória. Em C, o valor `NULL` é usado para inicializar ponteiros que não apontam para nada.

Exemplo:

```
#include <stdio.h>

int main() {
    int *ptr = NULL;  // Ponteiro nulo

    if (ptr == NULL) {
```

```
        printf("O ponteiro não aponta para nenhuma memória.\n");
    }

    return 0;
}
```

8. Ponteiros e Alocação Dinâmica

Em C, você pode usar ponteiros para alocar memória dinamicamente durante a execução do programa, utilizando as funções `malloc`, `calloc`, `realloc` e `free`.

Exemplo de alocação dinâmica com `malloc`:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *ptr = (int *)malloc(sizeof(int));  // Aloca memória para um inteiro

    if (ptr == NULL) {
        printf("Erro de alocação de memória!\n");
        return 1;
    }

    *ptr = 30;
    printf("Valor alocado: %d\n", *ptr);

    free(ptr);  // Libera a memória alocada
    return 0;
}
```

Neste exemplo, a memória para um inteiro é alocada dinamicamente e o valor `30` é armazenado nessa posição de memória.

Ponteiros são uma parte fundamental da linguagem C, permitindo manipulação direta de memória, otimização de desempenho e interação com funções de alocação dinâmica. No entanto, o uso de ponteiros exige cuidados, especialmente no que diz respeito ao gerenciamento de memória e ao acesso a locais inválidos, para evitar erros como falhas de segmentação.

String em C

Em C, **strings** são sequências de caracteres armazenadas em um array de caracteres. Diferente de outras linguagens, C não tem um tipo específico para strings, então, elas são tratadas como um array de `char`, finalizadas com o caractere especial nulo `'\0'`, que indica o fim da string.

Definindo uma String

Uma string é, na verdade, um array de caracteres. Para declarar uma string, você pode fazer o seguinte:

```
char nome[10];
```

Neste exemplo, `nome` é um array de 10 caracteres. Para armazenar uma string em um array de caracteres, você pode fazer da seguinte maneira:

```
char nome[10] = "Carlos";
```

A string `"Carlos"` ocupa 6 caracteres (C, a, r, l, o, s) mais o caractere nulo `'\0'`, que indica o fim da string. Portanto, o array `nome` ocupa 7 posições (6 caracteres + 1 para o terminador `'\0'`).

String com Ponteiro

Em C, as strings também podem ser representadas como ponteiros para o primeiro caractere do array:

```
char *nome = "Carlos";
```

Aqui, o ponteiro `nome` aponta para o primeiro caractere da string `"Carlos"`, que é armazenada como um array de caracteres na memória.

Atribuição e Manipulação de Strings

A atribuição direta de strings para arrays após a declaração não é permitida em C. Em vez disso, você pode usar a função `strcpy` para copiar uma string para outra:

```
#include <stdio.h>
#include <string.h>

int main() {
    char nome[10];
    strcpy(nome, "Carlos");
    printf("Nome: %s\n", nome);
    return 0;
}
```

Isso irá copiar a string `"Carlos"` para o array `nome` e imprimi-la.

Funções Comuns para Manipulação de Strings

Existem várias funções da biblioteca padrão de C (`<string.h>`) que ajudam a manipular strings:

1. **`strlen`**: Retorna o comprimento de uma string (sem contar o terminador `'\0'`).

```c
int tamanho = strlen(nome);  // Retorna 6 para "Carlos"
```

2. `strcpy`: Copia uma string para outra.

```c
char destino[10];
strcpy(destino, "Olá");
```

3. `strcat`: Concatena duas strings.

```c
char saudacao[20] = "Olá, ";
strcat(saudacao, "Carlos!");
printf("%s\n", saudacao);  // Imprime "Olá, Carlos!"
```

4. `strcmp`: Compara duas strings lexicograficamente. Retorna 0 se forem iguais, um valor negativo se a primeira for menor, e um valor positivo se a primeira for maior.

```c
if (strcmp("abc", "def") < 0)
    printf("abc é menor que def\n");
```

5. `strchr`: Encontra a primeira ocorrência de um caractere em uma string.

```c
char *p = strchr("Carlos", 'r');
printf("%s\n", p);  // Imprime "ros"
```

6. `strstr`: Encontra a primeira ocorrência de uma substring dentro de uma string.

```c
char *p = strstr("Carlos", "ar");
printf("%s\n", p);  // Imprime "arlos"
```

Modificando Strings

Como as strings em C são arrays de caracteres, você pode acessar e modificar individualmente cada caractere. Isso permite fazer alterações na string diretamente, mas você deve ter cuidado com o limite do array para não sobrescrever a memória.

```c
#include <stdio.h>

int main() {
```

```
    char nome[10] = "Carlos";
    nome[0] = 'M';  // Modifica a string para "Marlos"
    printf("Nome modificado: %s\n", nome);
    return 0;
}
```

Strings Imutáveis

Embora a string literal em C seja constante, a string que você manipula em um array pode ser alterada. Para evitar alterações acidentais, você pode definir a string como `const`:

```
const char *nome = "Carlos";
```

Neste caso, você não pode modificar os caracteres da string, pois ela está armazenada em um espaço de memória somente leitura.

Exemplo Completo

Aqui está um exemplo completo que demonstra o uso de várias funções de manipulação de strings:

```
#include <stdio.h>
#include <string.h>

int main() {
    char nome[20] = "Carlos";
    char saudacao[50] = "Olá, ";

    // Exibir o comprimento da string
    printf("Comprimento do nome: %lu\n", strlen(nome));

    // Copiar uma string
    strcpy(nome, "Maria");
    printf("Nome após strcpy: %s\n", nome);

    // Concatenar strings
    strcat(saudacao, nome);
    printf("Saudação concatenada: %s\n", saudacao);

    // Comparar strings
    if (strcmp(nome, "Maria") == 0) {
        printf("O nome é Maria!\n");
    }

    // Encontrar um caractere
    char *p = strchr(nome, 'a');
    printf("Primeira ocorrência de 'a' em nome: %s\n", p);
```

```
    return 0;
}
```

Neste exemplo, usamos funções como `strlen`, `strcpy`, `strcat`, `strcmp` e `strchr` para manipular a string `nome` e a string `saudacao`.

Em C, as strings são tratadas como arrays de caracteres, e a manipulação delas depende de funções específicas fornecidas pela biblioteca padrão `<string.h>`. A precisão ao lidar com strings é essencial, já que C não realiza verificações automáticas de limites de arrays, o que pode resultar em erros se não forem tomados os devidos cuidados com o tamanho dos buffers.

Referências

- AHU, Aho, A. V.; Hopcroft, J. E.; Ullman, J. D. **Data Structures and Algorithms**. Addison-Wesley, 1983.
- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms**. MIT Press, 2009.
- KNUTH, Donald E. **The Art of Computer Programming**. Addison-Wesley, 1997.
- Kernighan, B. W., & Ritchie, D. M. (1978). *The C Programming Language*.
- Deitel, P., & Deitel, H. (2016). *C How to Program*.
- Harbison, S. P., & Steele, G. L. (1995). *C: A Reference Manual*.
- KERNIGHAN, Brian W.; RITCHIE, Dennis M. **The C Programming Language**. Prentice Hall, 1988.
- TANENBAUM, A. S. **Structured Computer Organization**. 6ª ed., Pearson, 2016.