Variáveis

Em C, como em outras linguagens, podemos criar variáveis. A primeira coisa que você precisa saber é o escopo dessas variáveis, ou onde são válidas e qual é o seu tempo de vida.

Escopo de variáveis

Escopo de bloco

Variáveis declaradas dentro de um bloco de código (delimitado por {}) têm escopo de bloco. Essas variáveis são acessíveis apenas dentro do bloco em que foram declaradas. E são recriadas cada vez que o bloco é executado.

```
void funcao() {
    int x = 10; // 'x' tem escopo de função
    if (x > 5) {
        int y = 20; // 'y' tem escopo de bloco, acessível apenas dentro
deste 'if'
    }
    // 'y' não é acessível aqui
}
```

Escopo local

Variáveis declaradas dentro de uma função, mas fora de qualquer bloco interno, têm escopo de função. Essas variáveis são acessíveis em qualquer ponto da função após sua declaração. E são recriadas a cada invocação da função.

```c
void funcao() {
    int x = 10; // 'x' tem escopo de função
    // 'x' é acessível em toda a função
}

void outraFuncao() {
    // 'x' não é acessível aqui.
}
```

Escopo de arquivo (global)

Variáveis declaradas fora de todas as funções têm escopo de arquivo. Essas variáveis são acessíveis de qualquer ponto no arquivo após sua declaração. Elas não são **globais** com relação ao projeto, ou seja, outros arquivos não podem acessá-las.

```c
int x = 10; // 'x' tem escopo de arquivo
void funcao1() {
    x = 20; // 'x' é acessível aqui
}
void funcao2() {
    x = 30; // 'x' também é acessível aqui
}
```

Escopo estático

Cuidado com essas variáveis, pois elas retém estado entre várias chamadas à função e podem complicar a sua lógica. Variáveis declaradas com a palavra-chave static dentro de uma função têm escopo de bloco composto.Essas variáveis mantêm seu valor entre chamadas da função e são acessíveis apenas dentro da função onde foram declaradas.

```c
void funcao() {
    static int x = 0; // 'x' tem escopo de bloco composto
    x++;
    printf("%d\n", x); // 'x' manterá seu valor entre chamadas da função
}
```

Escopo externo

Variáveis declaradas com a palavra-chave extern têm escopo externo ou de ligação. Essas variáveis são definidas em um arquivo, mas podem ser referenciadas em outros arquivos usando a palavra-chave extern.

```c
// arquivo1.c
int x = 10; // 'x' é definido aqui

// arquivo2.c
```

```c
extern int x; // 'x' é referenciado aqui
void funcao() {
    x = 20; // 'x' é acessível aqui
}
```

Área da memória ocupada pelas variáveis

Esse é um assunto pouco discutido nos cursos de linguagens de programação, mas em C é importante sabermos. Onde você acha que o C Runtime aloca suas variáveis?

No C Runtime, as variáveis são alocadas em diferentes áreas de memória dependendo do tipo de variável e de como ela é declarada. As principais áreas de memória são:

1. **Stack (Pilha):**
    - Utilizada para armazenar variáveis locais (não estáticas) e parâmetros de função.
    - A alocação e desalocação na pilha seguem uma ordem LIFO (Last In, First Out).
    - Cada chamada de função cria um novo quadro de pilha (stack frame) para suas variáveis locais e parâmetros.
    - A memória é automaticamente desalocada quando a função retorna.
    - Exemplos: Variáveis locais dentro de funções.

    ```c
    void func() {
        int x; // Alocado na pilha
    }
    ```

2. **Heap:**
    - Utilizada para alocação dinâmica de memória.
    - A memória é alocada e desalocada manualmente pelo programador usando funções como `malloc`, `calloc`, `realloc` e `free`.
    - A vida útil da memória alocada no heap é controlada pelo programador, não seguindo a ordem LIFO.
    - Exemplos: Memória alocada dinamicamente.

    ```c
    int* ptr = (int*)malloc(sizeof(int)); // Alocado no heap
    free(ptr); // Desalocado do heap
    ```

3. **Data Segment (Segmento de Dados):**
    - Dividido em duas subáreas:
        - **Initialized Data Segment:** Armazena variáveis globais e estáticas que são inicializadas com valores explícitos.
        - **Uninitialized Data Segment (BSS - Block Started by Symbol):** Armazena variáveis globais e estáticas que não são inicializadas explicitamente.
    - Essas variáveis têm duração durante toda a execução do programa.

- Exemplos:

```
int globalVar = 10; // Inicialized Data Segment
static int staticVar; // BSS
```

4. **Text Segment (Segmento de Texto):**

- Contém o código executável do programa.
- Esta área de memória é geralmente marcada como somente leitura para evitar que o código seja alterado durante a execução.
- Exemplos: Instruções do programa.

```
void func() {
    // O código da função está no segmento de texto
}
```

Essas áreas de memória são gerenciadas pelo C Runtime para garantir a correta alocação e desalocação de recursos durante a execução de um programa em C.

Tipos de dados C

Os tipos de dados em C ANSI podem ser divididos em várias categorias, e o número de bits que eles ocupam pode variar dependendo da implementação do compilador e da arquitetura da máquina (por exemplo, 16 bits, 32 bits, 64 bits). No entanto, abaixo estão os tamanhos típicos para uma arquitetura de 32 bits. Para uma arquitetura de 64 bits, os tamanhos podem variar, especialmente para ponteiros e tipos longos.

Tipos de Dados Inteiros

1. **char**

- Tamanho: 8 bits (1 byte)
- Intervalo: -128 a 127 (signed char) ou 0 a 255 (unsigned char)

2. **short**

- Tamanho: 16 bits (2 bytes)
- Intervalo: -32,768 a 32,767 (signed short) ou 0 a 65,535 (unsigned short)

3. **int**

- Tamanho: 32 bits (4 bytes)
- Intervalo: -2,147,483,648 a 2,147,483,647 (signed int) ou 0 a 4,294,967,295 (unsigned int)

4. **long**

- Tamanho: 32 bits (4 bytes)
- Intervalo: -2,147,483,648 a 2,147,483,647 (signed long) ou 0 a 4,294,967,295 (unsigned long)

5. **long long**
    - Tamanho: 64 bits (8 bytes)
    - Intervalo: -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807 (signed long long) ou 0 a 18,446,744,073,709,551,615 (unsigned long long)

Tipos de Dados de Ponto Flutuante

1. **float**
    - Tamanho: 32 bits (4 bytes)
    - Precisão: Aproximadamente 6-7 dígitos decimais
2. **double**
    - Tamanho: 64 bits (8 bytes)
    - Precisão: Aproximadamente 15-16 dígitos decimais
3. **long double**
    - Tamanho: Dependente da implementação (geralmente 80 bits, 96 bits ou 128 bits)
    - Precisão: Maior que a do `double`

Outros Tipos de Dados

1. **void**
    - Não ocupa espaço de armazenamento, utilizado para indicar a ausência de tipo, especialmente em ponteiros.
2. **Ponteiros**
    - Tamanho: Depende da arquitetura (geralmente 32 bits em sistemas de 32 bits e 64 bits em sistemas de 64 bits)

Resumo dos Tamanhos de Tipos de Dados (em uma arquitetura de 32 bits)

Tipo	Tamanho (bits)	Tamanho (bytes)
char	8	1
signed char	8	1
unsigned char	8	1
short	16	2
signed short	16	2
unsigned short	16	2
int	32	4
signed int	32	4

Tipo	Tamanho (bits)	Tamanho (bytes)
unsigned int	32	4
long	32	4
signed long	32	4
unsigned long	32	4
long long	64	8
signed long long	64	8
unsigned long long	64	8
float	32	4
double	64	8
long double	80/96/128	10/12/16
void	0	0
ponteiros	32/64	4/8

> Não existe **string**! Em C utilizamos um vetor de char para representar texto.

Estes tamanhos são típicos, mas podem variar com base no compilador e na arquitetura da máquina. Para precisão absoluta, é sempre bom verificar a documentação específica do compilador ou usar a função `sizeof` no C para determinar o tamanho dos tipos em bytes.

Declaração e Inicialização de Strings

Em C ANSI, uma string é representada como um array de caracteres (`char`). As strings em C são terminadas por um caractere nulo (`'\0'`), que indica o final da string. Aqui estão alguns exemplos de como você pode representar e manipular strings em C ANSI:

1. **Declaração e Inicialização Direta:**

```c
char str1[] = "Hello, World!";
```

   - O compilador automaticamente adiciona o caractere nulo (`'\0'`) no final da string.
   - O tamanho do array `str1` será 14, pois a string "Hello, World!" tem 13 caracteres, mais o caractere nulo.

2. **Declaração com Tamanho Específico:**

```c
char str2[20] = "Hello, World!";
```

- O array `str2` tem espaço para 20 caracteres, mas apenas os primeiros 14 (incluindo o caractere nulo) são inicializados.

3. **Declaração e Inicialização Caractere por Caractere:**

```
char str3[] = {'H', 'e', 'l', 'l', 'o', ',', ' ', 'W', 'o', 'r', 'l',
'd', '!', '\0'};
```

> **A comparação entre strings** requer um cuidado. Como em **Java**, não podemos simplesmente comparar duas variáveis `char*` com o operador **==**! Isso compararia se os dois ponteiros apontam para o mesmo string na memória. Precisamos utilizar a função `strcmp(string1, string2)` da biblioteca **string.h**.

Comparando dois **strings**:

```
#include <string.h>
...
    if(strcmp(operador,"^") == 0) {
        return 3;
    } else if(strcmp(operador,"*") == 0 || strcmp(operador,"/") == 0) {
        return 2;
    } else return 1;
}
...
```

A função `strcmp()` recebe dois parâmetros: String1 e String2. Se forem iguais, retorna zero, caso contrário:

- **Número positivo**: String1 > String2. O primeiro caracter diferente de String1 é maior na tabela ASCII que o primeiro caracter diferente de String2.
- **Número negativo**: String1 < String2. O primeiro caracter diferente de String1 é menor na tabela ASCII que o primeiro caracter diferente de String2.

Comparação entre `long long` e `int64_t`

Existem dois tipos para declarar inteiros de 64 bits: `long long` e `int64_t`. Vou explicar a diferença e dizer qual é a recomendação de uso.

`long long`

- **Introduzido**: Em C99.
- **Tamanho**: Pelo menos 64 bits.
- **Intervalo**: Pelo menos -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807.
- **Portabilidade**: O tamanho pode variar dependendo do compilador e da plataforma, mas sempre será pelo menos 64 bits. Não garante um tamanho exato em todos os sistemas.
- **Uso**: Utilizado quando se precisa de um tipo de dado inteiro que é garantido ser pelo menos 64 bits.

`int64_t`

- **Introduzido**: Em C99 (como parte do cabeçalho `<stdint.h>`).
- **Tamanho**: Exatamente 64 bits.
- **Intervalo**: -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807.
- **Portabilidade**: Garante um tamanho exato de 64 bits em todas as plataformas que suportam C99 e `<stdint.h>`.
- **Uso**: Utilizado quando se precisa de um tipo de dado inteiro que é garantido ser exatamente 64 bits, independentemente da plataforma.

Recomendação de Uso

Portabilidade

- `int64_t` é preferido em termos de portabilidade. Como o `int64_t` é definido para ter exatamente 64 bits em todas as plataformas compatíveis com C99, ele garante que seu código será portátil e se comportará da mesma maneira em diferentes sistemas.
- `long long` pode variar em tamanho em diferentes sistemas (embora seja pelo menos 64 bits), o que pode levar a inconsistências ao mover o código entre plataformas diferentes.

Leitura e Manutenção

- Usar tipos explícitos como `int64_t` torna o código mais claro e explícito sobre o tamanho esperado das variáveis. Isso pode facilitar a leitura e a manutenção do código, especialmente em projetos grandes ou colaborativos.
- Por outro lado, `long long` pode ser menos explícito em termos de tamanho, o que pode causar confusão sobre as garantias de tamanho que ele fornece.

Exemplos de Uso

Usando `long long`

```c
#include <stdio.h>

void example() {
    long long x = 9223372036854775807LL;
    printf("Value of x: %lld\n", x);
}
```

Usando `int64_t`

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

void example() {
    int64_t x = 92233720368547758O7LL;
    printf("Value of x: %" PRId64 "\n", x);
}
```

Sempre que possível, utilize `int64_t` pois tem um tamanho específico e funciona em qualquer plataforma compatível com **c99**.

Vetores

Em C, vetores (ou arrays) são uma estrutura de dados que permite armazenar uma coleção de elementos do mesmo tipo. Eles podem ser unidimensionais ou bidimensionais, dependendo do número de índices usados para acessá-los. Vamos explorar ambos os tipos:

Vetores

Um vetor unidimensional é uma lista linear de elementos. Aqui estão alguns conceitos básicos e exemplos sobre vetores unidimensionais em C:

Declaração e Inicialização

1. **Declaração Simples:**

```
int vetor[10]; // Declara um vetor de 10 inteiros
```

2. **Inicialização no Momento da Declaração:**

```
int vetor[5] = {1, 2, 3, 4, 5}; // Declara e inicializa um vetor de 5
inteiros
```

3. **Inicialização Parcial:**

```
int vetor[5] = {1, 2}; // Os elementos não inicializados são definidos
como 0
// vetor = {1, 2, 0, 0, 0}
```

4. **Sem Especificar o Tamanho:**

```
int vetor[] = {1, 2, 3, 4, 5}; // O compilador determina o tamanho
automaticamente
// vetor é de tamanho 5
```

Acesso aos Elementos

Os elementos de um vetor unidimensional são acessados usando índices, que começam em 0.

```c
#include <stdio.h>

int main() {
    int vetor[5] = {10, 20, 30, 40, 50};

    // Acessar e exibir elementos do vetor
    for (int i = 0; i < 5; i++) {
        printf("vetor[%d] = %d\n", i, vetor[i]);
    }

    return 0;
}
```

Matrizes (Bidimensionais)

Um vetor bidimensional é uma tabela ou matriz de elementos. Ele é declarado e inicializado com duas dimensões.

Declaração e Inicialização

1. **Declaração Simples:**

```c
int matriz[3][4]; // Declara uma matriz de 3 linhas e 4 colunas
```

2. **Inicialização no Momento da Declaração:**

```c
int matriz[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
}; // Declara e inicializa uma matriz de 2x3
```

3. **Inicialização Parcial:**

```c
int matriz[2][3] = {
    {1, 2},
    {4}
}; // Os elementos não inicializados são definidos como 0
// matriz = {{1, 2, 0}, {4, 0, 0}}
```

Acesso aos Elementos

Os elementos de um vetor bidimensional são acessados usando dois índices: o primeiro para a linha e o segundo para a coluna.

```c
#include <stdio.h>

int main() {
    int matriz[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };

    // Acessar e exibir elementos da matriz
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            printf("matriz[%d][%d] = %d\n", i, j, matriz[i][j]);
        }
    }

    return 0;
}
```

Confira o exemplo de código nos anexos da aula

Neste exemplo, que utilizaremos na aula seguinte, é possível identificar as variáveis, seu tipo, seu escopo e sua unicidade ou não (se são vetores).