

Linguagens de Programação 1

Aula #2

Message of the Day

"Simplicity is prerequisite for reliability."

"The question of whether a computer can think is no more interesting than the question of whether a submarine can swim."

— Edsger W. Dijkstra

Edsger Dijkstra (1930 – 2002)

Desenvolveu o **Algoritmo de Dijkstra** para encontrar o caminho mais curto num grafo.

Defendeu a **programação estruturada** e as **melhores práticas de engenharia de software**.

Trabalhos fundamentais em **concorrência, exclusão mútua, deadlock**

Vencedor do Prêmio Turing, considerado o "Nobel da Computação"

Conteúdos da Aula

Estrutura de um programa em C

Bibliotecas

Funções

Variáveis e Tipos de Dados

Estrutura de um Programa em C

Componentes principais

- 1 Inclusão de Bibliotecas
- 2 Declaração de Funções
- 3 Função Principal (`main`)
- 4 Definição de Funções

Bibliotecas

São um **conjunto de funções** contidas num único ficheiro.

Permitem **reutilização de código** e **organização eficiente**.

Estrutura das bibliotecas

Cada biblioteca tem um **ficheiro de cabeçalho** (**.h**) que contém:

Declarações de tipos de dados especiais usados nas funções.

Declarações das funções que podem ser usadas por um programa.

Exemplo de inclusão de biblioteca

```
#include <stdio.h> // Inclui a biblioteca padrão de entrada/saída
```

Exemplo: Biblioteca `math.h`

Utilização em um programa

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

int main() {
    double num = 25.0;
    printf("Raiz quadrada de %.2f é %.2f\n", num, sqrt(num));
    return 0;
}
```

✓ **Vantagem:** Não precisamos de reimplementar funções matemáticas!

Bibliotecas Mais Importantes em C

Biblioteca	Descrição
<code>math.h</code>	Funções matemáticas
<code>stdio.h</code>	Operações de entrada e saída
<code>stdlib.h</code>	Funções genéricas (memória, conversões)
<code>string.h</code>	Funções para manipulação de strings
<code>time.h</code>	Funções para data e hora
<code>ctype.h</code>	Manipulação de caracteres (ex: <code>isdigit()</code> , <code>toupper()</code>)
<code>limits.h</code>	Constantes dos limites dos tipos de dados

Função Principal (`main`)

```
int main(void) {  
    int x = 10;  
    int y = 20;  
    int z = min(x, y);  
  
    return 0; // Sucesso  
}
```

- ✓ Ponto de entrada do programa.
- ✓ Declara variáveis e chama funções.
- ✓ O `return 0;` indica **sucesso na execução**. Valores de retorno diferentes de 0 indicam que ocorreu um erro.

Programa completo

```
#include <stdio.h>
int min(int, int);
int max(int, int);

int main(void) {
    int x = 10, y = 20, z;
    z = min(x,y);
    return 0;
}

int min(int a, int b) {
    return a < b ? a : b;
}
int max(int a, int b) {
    return a > b ? a : b;
}
```

← Inclusão de Bibliotecas

← Declaração de Funções

← Função `main()`

← Definição de Funções

Representação da Informação

 Como os computadores armazenam dados?

- ✓ Tudo no computador é representado como **números**.
- ✓ Estes números são armazenados em **base binária (0 e 1)**.

Modelo da Memória

- 📌 A memória RAM armazena dados em endereços numerados sequencialmente.
- ✓ Cada **endereço** contém um **valor armazenado**.
- ✓ As variáveis ocupam **um ou mais bytes** na RAM.
- 📌 Uma variável tem:
 - ✓ **Nome** (Identificador)
 - ✓ **Valor** (Conteúdo armazenado)
 - ✓ **Endereço de memória**

Exemplo de Alocação na RAM

```
char x = 'a';
int N = 10;
```

- ✓ O `char x` ocupa **1 byte**.
- ✓ O `int N` ocupa **4 bytes** (dependendo da arquitetura).

Endereço	Conteúdo	Identificador
...
509	'a'	x
510	0	N
511	0	
512	0	
513	10	
...


Tipos de Dados em C - Tipos Primitivos

O tipo é necessário para determinar o espaço de memória que deve ser reservado para armazenar o valor correspondente

Tipo	Tamanho (bytes)	Descrição
char	1 byte	Caracteres
int	4 bytes	Números inteiros
float	4 bytes	Números reais (ponto flutuante, precisão simples)
double	8 bytes	Números reais (ponto flutuante, precisão dupla)
enum	4 bytes	Lista de valores inteiros nomeados

 **Arquiteturas diferentes podem ter tamanhos diferentes para os tipos de dados!**

Qualificadores de Tipo

 Os qualificadores alteram a forma como os dados são armazenados e interpretados

Qualificador	Aplicável a	Descrição
<code>short</code>	<code>int</code>	Reduz o tamanho do inteiro
<code>long</code>	<code>int</code> , <code>double</code>	Aumenta o tamanho do número
<code>signed</code>	<code>int</code> , <code>char</code>	Permite valores positivos e negativos
<code>unsigned</code>	<code>int</code> , <code>char</code>	Apenas valores positivos



Regras Importantes

- ✓ O `int` pode ser precedido por **todos os qualificadores** (`short`, `long`, `signed`, `unsigned`).
- ✓ O `double` pode apenas ser precedido por **long** (`long double`).
- ✓ Por omissão, **todos os tipos inteiros são `signed`**, exceto se declarado `unsigned`.

Exemplos de Declarações Válidas

```
short int x;           // Inteiro curto
long double y;         // Número de precisão extra
unsigned int z;        // Apenas valores positivos
signed char letra;     // Pode armazenar valores negativos
```

Limites dos Tipos de Dados em C

-  O número de bytes associado a cada tipo depende da arquitetura do processador.
-  Os limites influenciam o número e a magnitude dos valores possíveis.
- ✓ **Arquiteturas modernas** geralmente seguem estas regras, mas podem variar!



Tamanhos e Limites dos Tipos Inteiros


Tipo	Bytes	Mínimo	Máximo
char	1	-128	127
short int	2	-32,768	32,767
int	4	-2,147,483,648	2,147,483,647
enum	4	-2,147,483,648	2,147,483,647
long int	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

Tamanhos e Limites dos Tipos Unsigned

 Os tipos **unsigned** não permitem valores negativos, mas dobram o limite superior.

Tipo	Bytes	Mínimo	Máximo
<code>unsigned char</code>	1	0	255
<code>unsigned short int</code>	2	0	65,535
<code>unsigned int</code>	4	0	4,294,967,295
<code>unsigned long int</code>	8	0	18,446,744,073,709,551,615

Tamanhos e Limites dos Tipos de Ponto Flutuante


 Os tipos de ponto flutuante armazenam valores decimais com precisão variável.


Tipo	Bytes	Mínimo	Máximo
<code>float</code>	4	-3.4028E+38	3.4028E+38
<code>double</code>	8	-1.7977E+308	1.7977E+308
<code>long double</code>	16	-1.1897E+4932	1.1897E+4932

✓ Quanto maior o tamanho, maior a precisão e o intervalo de valores!

α Variáveis

Identificadores de Variáveis

 O identificador (ou nome) de uma variável permite ao programador referenciá-la sem precisar saber seu endereço na memória.

- ✓ Um identificador pode conter **letras, dígitos e underscores** ().
- ✓ Deve ter **pelo menos um caractere**.
- ✓ **Não pode começar com um dígito!**

Exemplos de Identificadores

Válidos:

```
x  
times10  
get_next_char  
_b1  
Xbto
```



Inválidos:

```
1a      // Começa por um dígito  
café    // Contém caractere acentuado  
get-next // Contém hífen (-)
```

Boa prática: Usar nomes descritivos!

```
int total_alunos; // Melhor que apenas 't'
```

O que é o Endereço de uma Variável?

-  O **endereço da variável** indica onde ela está armazenada na memória RAM.
-  O programador **não controla diretamente os endereços**, pois eles são atribuídos pelo **sistema operativo (SO)** quando o programa é executado.

```
int x = 10;
int y = 50;
```

Endereço	Conteúdo	Identificador
...
600	10	x
604	50	y
...

Como Obter o Endereço de uma Variável

 Podemos obter o endereço de uma variável usando o operador `&`.

```
int x = 10;
int y = 50;

printf("Valor de x: %d\n", x);
printf("Endereço de x: %p\n", &x);
```

Endereço	Conteúdo	Identificador
...
600	10	x
604	50	y
...

- ✓ O `%p` imprime o **endereço de memória**.
- ✓ Cada vez que o programa executa, o endereço pode ser diferente!

Declaração de Variáveis

 O que acontece ao declarar uma variável?

- ✓ Reserva memória de acordo com seu tipo de dados.
- ✓ Atribui um nome (identificador) à variável.
- ✓ Opcionalmente, pode atribuir um valor inicial.
- ✓ As variáveis devem ser declaradas antes de serem utilizadas.

```
int idade = 25; // Declaração com inicialização  
float temperatura; // Declaração sem inicialização
```


12 **34** Tipos Reais

📌 Os tipos reais representam números de ponto flutuante e podem armazenar valores positivos e negativos.

📌 Exemplo de declaração e inicialização:

```
float pi = 3.14F;  
double area = 5.0E-1; // Notação científica  
long double constante = 6.00000023L;
```

- ✓ Por **omissão**, um literal decimal como `3.14` é do tipo `double`.
- ✓ Para definir um `float`, usamos **F** (`3.14F`).
- ✓ Para definir um `long double`, usamos **L** (`3.14L`).

Imprecisão dos números de vígula flutuante

A precisão é limitada pelo número de bits disponíveis.

Nem todos os números decimais podem ser representados exatamente.

Pequenos erros de arredondamento podem acumular-se em cálculos sucessivos.

Exemplo de erro de precisão:

```
int main() {  
    float x = 0.1f + 0.2f;  
    printf("Resultado: %.10f\n", x); // Pode não ser exatamente 0.3  
    return 0;  
}
```

Resultado: 0.3000000119

Por que ocorre a imprecisão?

Os números de vígula flutuante são armazenados usando a **representação IEEE 754**.

Um número é representado por:

Sinal (1 bit)

Expoente (8 bits para `float`, 11 para `double`)

Mantissa (23 bits para `float`, 52 para `double`)

Como resultado, apenas frações binárias exatas podem ser representadas.

Alguns números decimais, como **0.1** e **0.3**, não têm representação exata.

1²₃₄ Representação binária de 0.1 e 0.3

Exemplo: 0.1 em IEEE 754 (32 bits)

A conversão de `0.1` para binário resulta em uma fração periódica infinita:

```
0.00011001100110011001100110011001100110... (binário infinito)
```

Que é truncada no formato `float` e é armazenado como:

```
IEEE 754 (32 bits): 0x3DCCCCCD
```

Isso causa pequenos erros quando somamos valores como `0.1f + 0.2f`.

Como lidar com a imprecisão?

- ✓ Use `double` em vez de `float` para maior precisão.
- ✓ Evite comparar números de ponto flutuante diretamente.
- ✓ Utilize tolerância (`epsilon`) para comparações:

```
#include <math.h>
if (fabs(a - b) < 1e-6) {
    printf("Os valores são aproximadamente iguais.\n");
}
```

Valores Lógicos (Booleanos)

 Em C, valores booleanos não possuem um tipo nativo.

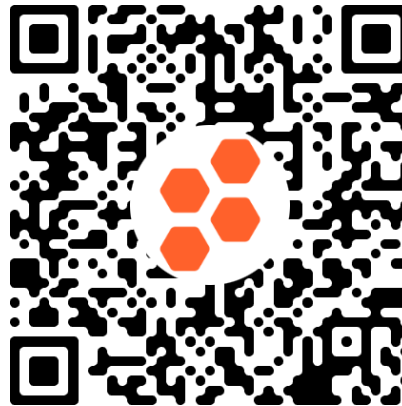
✓ Tudo que é `0` é considerado **falso** (`false`).

✓ Qualquer valor diferente de `0` é considerado **verdadeiro** (`true`).

```
int main(void) {  
    int a = 10, b = 0;  
  
    if (a) printf("a é verdadeiro!\n"); //  Será impresso  
    if (b) printf("b é verdadeiro!\n"); //  Não será impresso  
  
    return 0;  
}
```

 Valores negativos também são considerados verdadeiros!

? Quizz - Declaração de Variáveis



Conversão de Tipos

📌 A conversão de tipos ocorre automaticamente em algumas situações, mas pode ser forçada usando `casting`.

✓ Conversão automática:

```
int a = 5;
float b = a; // `a` é automaticamente convertido para float (5.0)
```

✓ Conversão explícita (`casting`):

```
int x = 5, y = 2;
float resultado = (float)x / y; // Sem casting: 5/2 = 2 (int)
                                // Com casting: 5.0/2 = 2.5 (float)
```


✓ Regras gerais de conversão:

`char` e `short` são convertidos para `int` .

`float` é convertido para `double` .

O **tipo menor** é convertido para o **tipo maior** automaticamente.



Exercício

? Identifique quais são válidas e inválidas:

```
int x, X;  
char c, double d;  
char char;
```

Exercício (Solução)

? Identifique quais são válidas e inválidas:

```
int x, X; // Válido  
char c, double d; // Inválido  
char char; // Inválido
```

 Dicas:

- ✓ Variáveis não podem ter nomes reservados (`char char;` é inválido)
- ✓ Devem ser declaradas antes de serem usadas

? Exercício

Qual o valor gravado na variável `c` em cada linha do seguinte código

```
int a = 5, b = 1;  
float d = 1.0;  
float c;  
  
c = d / a;  
  
c = b / a;  
  
c = (float) b / a;
```

? Exercício (solução)

Qual o valor gravado na variável `c` em cada linha do seguinte código

```
int a = 5, b = 1;
float d = 1.0;
float c;

c = d / a; // c = (float)1.0 / (int)5 = 0.2

c = b / a; // c = (int)1 / (int)5 = 0

c = (float) b / a; // c = (float)1 / (int)5 = 0.2
```

O Tipo `void`

O que é o `void` ?

- ✓ Representa um **tipo de dados vazio**.
- ✓ Usado para funções que **não retornam valores**.
- ✓ Pode ser usado para **funções sem parâmetros**.

```
void mensagem(void) {  
    printf("Olá, mundo!\n");  
}
```

Uso comum:

Como **tipo de retorno** para funções sem retorno.

Como **parâmetro** para indicar que a função não recebe argumentos.

Definição de Constantes

- 📌 Constantes são valores fixos que não podem ser modificados.
- ✓ Melhoram a legibilidade e segurança do código.
- ✓ São avaliadas em **tempo de compilação**.

```
#define TAMANHO 50 // Definição de constante via pré-processador  
const float PI = 3.1415; // Definição com `const`
```

- ✓ `#define` substitui todas as ocorrências do nome pela constante.
- ✓ `const` cria uma variável de leitura apenas, evitando mudanças acidentais.

Diferença entre `#define` e `const`

Característica	<code>#define</code>	<code>const</code>
Tipo de Constante	Substituição de texto	Variável com valor fixo
Avaliação	Pré-processador	Em tempo de execução
Usa Tipo de Dados?	✗ Não	✓ Sim
Ocupa Memória?	✗ Não	✓ Sim

Quando usar qual?

- ✓ Use `#define` para constantes simples (exemplo: tamanhos de arrays).
- ✓ Use `const` para variáveis que devem ter um tipo definido.

Funções

O que é uma Função?

Bloco de código reutilizável que realiza uma tarefa específica

Permite **organizar** o código em partes menores

Ajuda na **manutenção** e reutilização do código

Deve ser **independente** e sem efeitos colaterais

 **Exemplo:**

```
int soma(int a, int b) {  
    return a + b;  
}
```

Estrutura de uma Função

```
int max(int a, int b) {  
    if (a > b)  
        return a;  
    return b;  
}
```

Componentes principais

Tipo de retorno (`int`)

Nome da função (`max`)

Parâmetros (`int a, int b`)

Corpo com instruções - delimitado por `{ }`

Declaração de uma Função

```
int max(int a, int b);
```

- ✓ Diz ao compilador que a função existe.
- ✓ Evita erros se a definição da função vier depois da `main()`.
- ✓ Normalmente ficam antes da `main()` ou em arquivos `.h`.

Definição de uma Função

```
int max(int a, int b)
{
    return (a > b) ? a : b;
}
```

✓ Contém o código que executa a função

Funções sem Retorno (**void**)

Funções podem **não retornar um valor**, utilizando **void** como tipo de retorno

 **Exemplo:**

```
void imprimeMensagem() {  
    printf("Olá, Mundo!\n");  
}
```

 **Chamada da função:**

```
imprimeMensagem();
```

✓ Útil para funções que apenas executam ações sem retornar valores

Funções com Retorno

Funções podem **devolver um valor** utilizando `return`

O tipo de retorno deve ser **declarado corretamente**

 **Exemplo:**

```
int quadrado(int x) {  
    return x * x;  
}
```

 **Uso da função:**

```
int resultado = quadrado(4);  
printf("%d", resultado); // 16
```

✓ O tipo do retorno **deve corresponder** ao tipo declarado

Parâmetros e Argumentos

Funções podem **receber dados de entrada** chamados **parâmetros**

Os valores passados para a função são chamados **argumentos**

Exemplo de função com parâmetros:

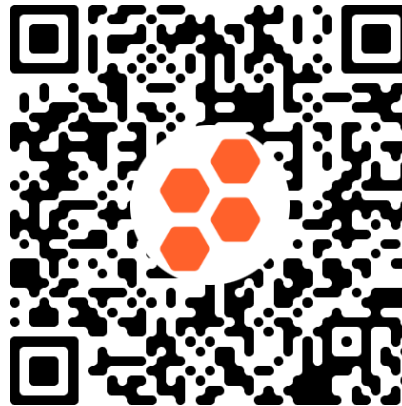
```
int multiplica(int a, int b) {  
    return a * b;  
}
```

Chamada da função:

```
int resultado = multiplica(3, 5); // Retorna 15
```

✓ Os parâmetros são passados na mesma ordem da definição!

? Quizz - Funções



 **Q&A**

 **Dúvidas?**