Estrutura de Dados 1

Revisão Linguagem C

Prof. Lucas Boaventura
lucas.boaventura@unb.br

Introdução

- Toda informação contida está armazenada na memória do computador

Endereço	Valor
...	...
200	0
204	8355
208	4
212	-1366160
...	...

Introdução

- Para que o nosso programa possa utilizar essa memória, utilizamos variáveis
- No código, declaramos:
- **tipo_da_variável** nome_da_variável;
- O tipo da varíavel deve ser um dos tipos suportados na linguagem e no seu código
- O nome da variável deve seguir as regras de declaração

Regras de nome da variável

- Uma variável pode conter letras, números ou underscore (_)
- Sempre deve iniciar com letras ou underscore
- A linguagem C é case-sensitive, sensível a maiúsculas e minúsculas
- Ou seja, a variável “Soma” e “soma” são variáveis diferentes

Regras de nome da variável

- Além disso, C possui 32 palavras-chave reservadas que não podem ser usadas como nome de variáveis

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

Tipos de variáveis

- Alguns tipos básicos de variáveis:

Tipo	Uso
char	Armazenar letras
int	Armazenar números inteiros
float	Armazenar números com casas decimal
double	Armazenar números com casas decimal

Tipos de variáveis

- Alguns tipos básicos de variáveis:

Tipo	Bits	Valor
char	8	-128 a 127
int	32	-2.147.483.648 a 2.147.483.647
float	32	complexo
double	64	complexo

Int

- Uma variável int armazena um número inteiro, ou seja, sem casas decimais
- **int** n = 10;
- Neste caso, ele irá armazenar um valor decimal

Imprimindo as variáveis

- Cada uma dessas variáveis podem ser utilizadas de diversas formas
- Uma das formas mais básica é imprimir na tela com a função “printf”, por exemplo:
- printf(“%d\n”, i);
- Imprime a variável “i” e uma nova linha na tela

Imprimindo as variáveis

```c
#include <stdio.h>

int main()
{
    int i = 1951;

    printf("%d\n", i);
    return 0;
}
```

- Código imprime 1951 na tela

Float

- Para a usar casas decimais, podemos declarar variáveis float (precisão simples) ou double (precisão dupla)
- float f = 3.14;
- double d = 1.84;
- Float possui 32 bits, double 64 bits. Ou seja, double consegue trabalhar com mais casas decimais. Nenhum deles é igual a número real

Float

- Para imprimir esses tipos, utilizamos “%f” no printf:

```c
#include <stdio.h>

int main()
{
    float f = 3.14;
    double d = 1.81;

    printf("%f %lf\n", f, d);
    return 0;
}
```

Saída: 3.140000 1.810000

Float

- Podemos alterar o número de casas decimais:

```c
#include <stdio.h>

int main()
{
    float f = 3.14;
    double d = 1.81;

    printf("%.5f %.3lf\n", f, d);
    return 0;
}
```

Saída: 3.14000 1.810

Char

- Também queremos representar letras nos programas de computador
- Para isso, utilizamos o tipo “char” para representar um caractere
- As letras são atribuídas com aspas simples

Char

- Exemplo:

```c
#include <stdio.h>

int main()
{
    char c = 'A';

    printf("%c\n", c);
    return 0;
}
```

Saída: A

Char

- Na verdade, as letras são representadas como números em um computador
- Uma tabela padrão conhecida como Tabela ASCII ilustra quais números representam quais caracteres

Char

```
[user@station]$ ascii -d
    0 NUL    16 DLE    32      48 0    64 @    80 P    96 `   112 p
    1 SOH    17 DC1    33 !    49 1    65 A    81 Q    97 a   113 q
    2 STX    18 DC2    34 "    50 2    66 B    82 R    98 b   114 r
    3 ETX    19 DC3    35 #    51 3    67 C    83 S    99 c   115 s
    4 EOT    20 DC4    36 $    52 4    68 D    84 T   100 d   116 t
    5 ENQ    21 NAK    37 %    53 5    69 E    85 U   101 e   117 u
    6 ACK    22 SYN    38 &    54 6    70 F    86 V   102 f   118 v
    7 BEL    23 ETB    39 '    55 7    71 G    87 W   103 g   119 w
    8 BS     24 CAN    40 (    56 8    72 H    88 X   104 h   120 x
    9 HT     25 EM     41 )    57 9    73 I    89 Y   105 i   121 y
   10 LF     26 SUB    42 *    58 :    74 J    90 Z   106 j   122 z
   11 VT     27 ESC    43 +    59 ;    75 K    91 [   107 k   123 {
   12 FF     28 FS     44 ,    60 <    76 L    92 \   108 l   124 |
   13 CR     29 GS     45 -    61 =    77 M    93 ]   109 m   125 }
   14 SO     30 RS     46 .    62 >    78 N    94 ^   110 n   126 ~
   15 SI     31 US     47 /    63 ?    79 O    95 _   111 o   127 DEL
```

Char

- Exemplo, imprimindo 2 caracteres:

```c
#include <stdio.h>

int main()
{
    char c1 = 'A';
    char c2 = 80;

    printf("%c%c\n", c1, c2);
    return 0;
}
```

Saída: AP

Char

- Nota: Apesar de ser aceito na linguagem C, é preferível que se inicialize caracteres com aspas simples
- Dessa forma, o código fica mais legível para humanos
- Isso é considerado uma “boa prática de programação”

Char

- Em APC, não trabalharemos com caracteres acentuados: ã, ç, ó, etc...
- Recomendação: nunca utilize acentos no código da nossa disciplina. Apenas caracteres ASCII
- Para mais informações, veja seção 2.3 do livro https://riscv-programming.org/book/riscv-book.html#pf1f

Lendo dados do teclado

- A leitura de dados do teclado pode ser realizada de diversas formas
- Para ler apenas um dado do teclado, é possível utilizar a função “getchar”

Lendo dados do teclado

- A função getchar não recebe parâmetros e retornar o caractere lido (ou um erro, que vamos aprender a tratar mais na frente)

```c
#include <stdio.h>

int main()
{
    char c1;

    c1 = getchar();
    printf("%c\n", c1);
    return 0;
}
```

Lendo dados do teclado

- É possível realizar a leitura de dados de uma forma semelhante ao printf, usando a função scanf
- No entanto, as variáveis que são passadas como argumento precisam receber o operador &
- Iremos entender o uso desse operador quando aprenderemos passagem por referência e parâmetro...

Lendo dados do teclado

- Uso de scanf:

```c
#include <stdio.h>

int main()
{
    int i;
    char c;

    scanf("%c", &c);
    scanf("%d", &i);
    printf("%c\n", c);
    printf("%d\n", i);

    return 0;
}
```

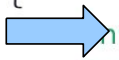
Reutilizando Variáveis

```c
#include <stdio.h>

int main()
{
    int i = 0;
    int y;

    printf("Valor inicial %d\n", i);
    y = i;
    printf("Valor intermediario %d %d\n", i, y);
    i = 6;
    printf("Valor final %d %d\n", i, y);
    return 0;
}
```

Saída:

Reutilizando Variáveis

```c
#include <stdio.h>

int main()
{
    int i = 0;
    int y;

    printf("Valor inicial %d\n", i);
    y = i;
    printf("Valor intermediario %d %d\n", i, y);
    i = 6;
    printf("Valor final %d %d\n", i, y);
    return 0;
}
```

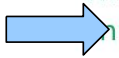


Endereço de Memória	Valor	Variável
0x7f..68	0	i
0x7f..60	LIXO	

Saída:

Reutilizando Variáveis

```c
#include <stdio.h>

int main()
{
    int i = 0;
    int y;

    printf("Valor inicial %d\n", i);
    y = i;
    printf("Valor intermediario %d %d\n", i, y);
    i = 6;
    printf("Valor final %d %d\n", i, y);
    return 0;
}
```



Endereço de Memória	Valor	Variável
0x7f..68	0	i
0x7f..60	LIXO	y

Saída:

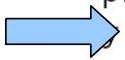
Reutilizando Variáveis

```c
#include <stdio.h>

int main()
{
    int i = 0;
    int y;

    printf("Valor inicial %d\n", i);
    y = i;
    printf("Valor intermediario %d %d\n", i, y);
    i = 6;
    printf("Valor final %d %d\n", i, y);
    return 0;
}
```

Saída:
Valor inicial 0

Reutilizando Variáveis

```c
#include <stdio.h>

int main()
{
    int i = 0;
    int y;

    printf("Valor inicial %d\n", i);
    y = i;
    printf("Valor intermediario %d %d\n", i, y);
    i = 6;
    printf("Valor final %d %d\n", i, y);
    return 0;
}
```

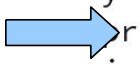


Endereço de Memória	Valor	Variável
0x7f..68	0	i
0x7f..60	0	y

Saída:
Valor inicial 0

Reutilizando Variáveis

```c
#include <stdio.h>

int main()
{
    int i = 0;
    int y;

    printf("Valor inicial %d\n", i);
    y = i;
    printf("Valor intermediario %d %d\n", i, y);
    i = 6;
    printf("Valor final %d %d\n", i, y);
    return 0;
}
```



Endereço de Memória	Valor	Variável
0x7f..68	0	i
0x7f..60	0	y

Saída:
Valor inicial 0
Valor intermediario 0 0

Reutilizando Variáveis

```c
#include <stdio.h>

int main()
{
    int i = 0;
    int y;

    printf("Valor inicial %d\n", i);
    y = i;
    printf("Valor intermediario %d %d\n", i, y);
    i = 6;
    printf("Valor final %d %d\n", i, y);
    return 0;
}
```

Endereço de Memória	Valor	Variável
0x7f..68	6	i
0x7f..60	0	y

Saída:
Valor inicial 0
Valor intermediario 0 0

Reutilizando Variáveis

```c
#include <stdio.h>

int main()
{
    int i = 0;
    int y;

    printf("Valor inicial %d\n", i);
    y = i;
    printf("Valor intermediario %d %d\n", i, y);
    i = 6;
    printf("Valor final %d %d\n", i, y);
    return 0;
}
```

Endereço de Memória	Valor	Variável
0x7f..68	6	i
0x7f..60	0	y

Saída:

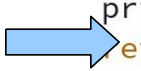
Valor inicial 0

Valor intermediario 0 0

Valor final 6 0

Reutilizando Variáveis

```c
#include <stdio.h>

int main()
{
    int i = 0;
    int y;

    printf("Valor inicial %d\n", i);
    y = i;
    printf("Valor intermediario %d %d\n", i, y);
    i = 6;
    printf("Valor final %d %d\n", i, y);
    return 0;
}
```



Endereço de Memória	Valor	Variável
0x7f..68	6	i
0x7f..60	0	y

Saída:

Valor inicial 0

Valor intermediario 0 0

Valor final 6 0

Operadores Relacionais

- Os operadores relacionais são usados para comparações:

Operador	Significado
>	Maior
>=	Maior ou igual
<	Menor
<=	Menor ou igual
==	Igual
!=	Diferente

Operadores Relacionais

- A linguagem C não possui um “tipo” lógico
  - bool b; //NAO e' um codigo em C
- No C, são utilizados valores inteiros:
  - Valor 0: falso
  - Valor diferente de 0: verdadeiro
- Já os operadores relacionais retornam:
  - Valor 0 para falso
  - Valor 1 para verdadeiro

Se / Então

- Em C, existem algumas formas de tratar as decisões
- A palavra-chave “if” é utilizada para isso:
  - if (verificação)
    {
  - ...
    }

Se / Então

```c
#include <stdio.h>

int main()
{
    int i;

    scanf("%d", &i);

    if (i % 2 == 1)
        printf("Numero impar\n");
    else
        printf("Numero par\n");
    return 0;
}
```

Lembrete: % é o operador de resto

```c
#include <stdio.h>

int main()
{
    int i;

    scanf("%d", &i);

    if (i % 2)
    {
        if (i > 10)
            printf("Numero impar maior que 10\n");
        else
            printf("Numero impar menor ou igual a 10\n");
    }
    else
    {
        printf("Numero par\n");
    }
    return 0;
}
```

Else if

```c
#include <stdio.h>

int main()
{
    int i;

    scanf("%d", &i);

    if (i % 3 == 1)
        printf("Nao divisivel. Resto 1\n");
    else if (i % 3 == 2)
        printf("Nao divisivel. Resto 2\n");
    else
        printf("Divisivel por 3.\n");
    return 0;
}
```

Switch

- Se a operação for analisada na mesma variável, podemos uma outra construção, com as palavras-chave “switch” e “case”
- Dentro do switch, colocamos diversos “case”, um para cada valor desejado

```c
#include <stdio.h>

int main()
{
    int i;

    scanf("%d", &i);
    switch (i % 3)
    {
        case 0:
            printf("Divisivel por 3.\n");
            break;
        case 1:
            printf("Nao divisivel. Resto 1\n");
            break;
        case 2:
            printf("Nao divisivel. Resto 2\n");
            break;
    }

    return 0;
}
```

Repetição

- Vimos que as condicionais são utilizadas para determinar múltiplos fluxos de execução no seu código
- Desta um programa pode determinar se um conjunto de instruções deve ou não ser executado
- Essa estrutura de repetição é básica na programação e chamada de **laço**

Repetição

- Durante a construção de programas é comum repetirmos as instruções que serão precisam ser executadas
- Por exemplo: dado um número N, imprimir todos os números de 0 até N

While

- Para construir esse código em C, podemos utilizar a palavra-chave: **while** (enquanto)
- Sintaxe:
  - while (condição) { ... }
- As instruções { ... } serão executadas enquanto a condição for verdadeira

```c
#include <stdio.h>

int main()
{
    int N;
    int contador = 0;

    scanf("%d", &N);

    while (contador <= N)
    {
        printf("%d\n", contador);
        contador = contador + 1;
    }
    return 0;
}
```

Do / While

- Também é possível usar a estrutura **do/while**
- Sintaxe:
  - do { ... } while (condição)
- As instruções { ... } serão executadas enquanto a condição for verdadeira
- Qual seria a diferença para o **while**?

Comparação while <-> do / while

  - while (condição) { ... }
- Verifica a condição -> Executa instruções
  - do { ... } while (condição)
- Executa instruções -> Verifica condição
- O do/while executa pelo menos uma vez as instruções, mesmo que a (condição) seja sempre falsa!

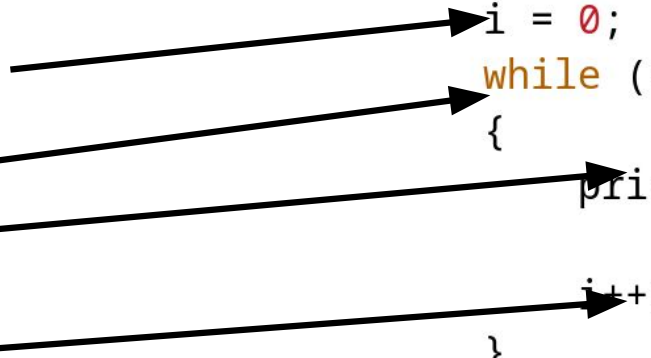
Comparação while <-> do / while

```c
int main()
{
    int N;
    int contador = 0;

    scanf("%d", &N);

    do
    {
        printf("%d\n", contador);
        contador = contador + 1;
    }
    while (contador <= N);

    return 0;
}
```

For

- Outra palavra chave muito importante é o **for** (para cada)
- Ele é especialmente utilizado para contadores, mas pode ser usado genericamente para vários laços complexos

- Esse código pode ser dividido nas partes:
  - Inicialização
  - Condição
  - Laço
  - Incremento



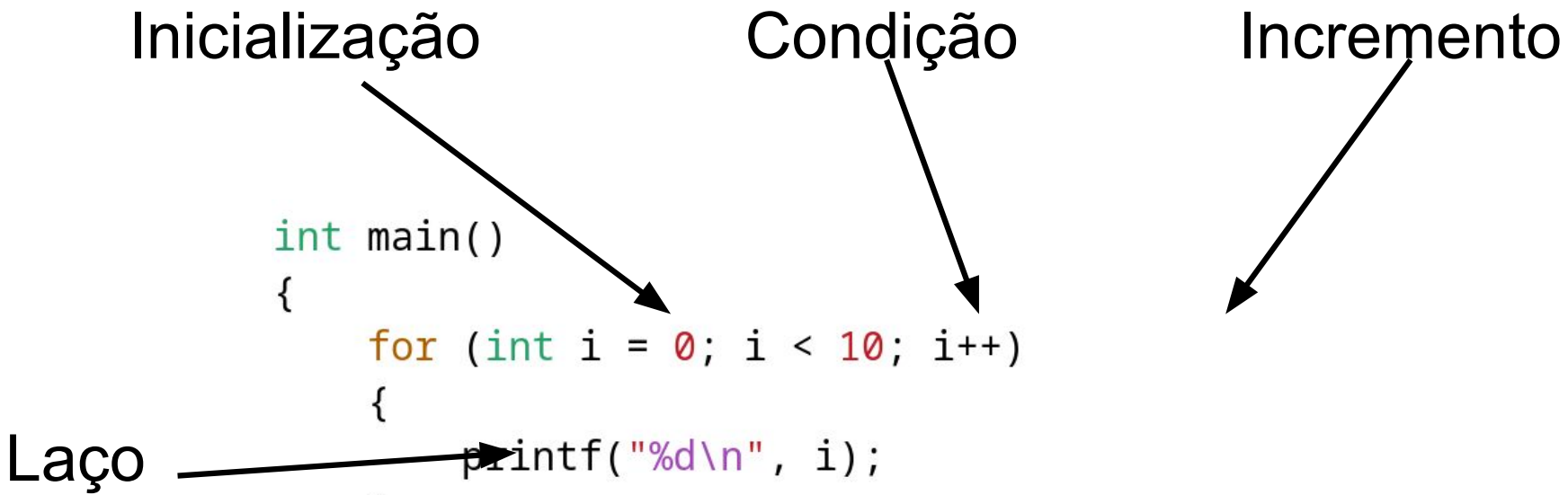
```c
#include <stdio.h>

int main()
{
    int i;

    i = 0;
    while (i < 10)
    {
        printf("%d\n", i);

        i++;
    }

    return 0;
}
```

For

- Essa estrutura de repetição se repete diversas vezes, que pode ser simplificado utilizando um laço **for**

```c
int main()
{
    for (int i = 0; i < 10; i++)
    {
        printf("%d\n", i);
    }
    return 0;
}
```

For


Inicialização
Condição
Incremento
Laço
int main()
{
for (int i = 0; i < 10; i++)
{
printf("%d\n", i);
}
return 0;
}

For

- No **for**, é opcional fazer a declaração da variável no laço, ou utilizar uma que já existe

```c
int main()
{
    for (int i = 0; i < 10; i++)
    {
        printf("%d\n", i);
    }
    return 0;
}
```

```c
int main()
{
    int i;
    for (i = 0; i < 10; i++)
    {
        printf("%d\n", i);
    }
    return 0;
}
```

For

- Podemos utilizar alguns membros vazios
- Para ser vazio, continuamos usando “;” dentro do for, mas não colocamos nada

```c
int main()
{
    int i;

    scanf("%d", &i);
    for ( ; i < 10; i++)
    {
        printf("%d\n", i);
    }
    return 0;
}
```

For

- Também é possível utilizar múltiplos campos, separando-os por vírgulas:

```c
int main()
{
    for (int i = 0, j = 10 ; i < 10; i++, j --)
    {
        printf("%d - %d\n", i, j);
    }
    return 0;
}
```

Saída do código:

0 - 10

1 - 9

...

9 - 1

Vetores

- Toda informação contida está armazenada na memória do computador

Endereço	Valor
...	...
0x7ffcc203070c	0
0x7ffcc2030708	8355
0x7ffcc2030704	4
0x7ffcc2030700	-1366160
...	...

Vetores

- As variáveis são mapeadas pelo compilador para determinada(s) região(ões)

Endereço	Valor	Variável
...	...	
0x7ffcc203070c	0	i
0x7ffcc2030708	8355	
0x7ffcc2030704	4	num
0x7ffcc2030700	-1366160	
...	...	

Vetores

- num = 5;

Endereço	Valor	Variável
...	...	
0x7ffcc203070c	0	i
0x7ffcc2030708	8355	
0x7ffcc2030704	5	num
0x7ffcc2030700	-1366160	
...	...	

Vetores

- Desta forma, o conteúdo das variáveis são sempre únicos, cada variável pode armazenar apenas um valor por vez
- Operações subsequentes nas variáveis sobrescrevem os valores antigos: eles são perdidos “para sempre”!

Vetores

- Para resolver esse problema, precisamos utilizar vetores de variáveis
- Utiliza-se colchetes [] junto à declaração da variável para determinar o tamanho de vetor desejado
- Declaração de um vetor de tamanho 10
  - int vet[10];

Vetores

- Depois da declaração, também podemos usar os colchetes vet[n] para acessar o *n*-ésimo elemento do vetor vet
- Atribuir o valor 5 para o membro número 2:
  - vet[2] = 5;

Vetores

```c
#include <stdio.h>

int main()
{
    int vet[10];

    vet[0] = 2;
    vet[5] = 10;
    vet[3] = -1;
    return 0;
}
```

Inicialização usando for

```c
#include <stdio.h>

int main()
{
    int vet[10];

    for (int i = 0; i < 10; i++)
    {
        scanf("%d", &vet[i]);
    }
    return 0;
}
```

Lixo de Memória

- É muito importante inicializarmos as variáveis antes de utilizá-las. Em C, não existe inicialização implícita das variáveis
- O que acontece quando lemos uma variável não inicializada?

```c
int main()
{
    int i;
    printf("%d\n", i);
    return 0;
}
```

Dúvidas?

- lucas.boaventura@unb.br