Programação de Computadores

Aula #03

Codificação de Caracteres e *Strings*

O que é um sistema de codificação? O que são *strings*? Quais são as funções de manipulação de *strings*?

Ciência da Computação – BCC e IBM – 2024/01
Prof. Vinícius Fülber Garcia

Codificação de Caracteres

Como já vimos anteriormente, em um nível de programação, a memória pode ser entendida como um conjunto de bytes.

Um byte, no entanto, representa um número entre 0 e 255... Sendo assim, a pergunta que fica é: como caracteres são representados em um computador?

Em termos gerais, podemos dizer que por meio de mapeamentos:

TABELAS QUE RELACIONAM NÚMEROS E CARACTERES

Codificação de Caracteres

Antes de explorarmos as possíveis codificações de caracteres mais a fundo, vamos estabelecer alguns conceitos:

- Caractere: símbolo de uma linguagem (letra, dígito ou sinal)
- Conjunto de caracteres (*charset*): conjunto de todos os caracteres suportados por um sistema ou padrão de codificação
- Codificação: processo de tradução entre caracteres e seu respectivo valor numérico em um sistema ou padrão de codificação

A Codificação ASCII

Para computadores eletrônicos digitais, a codificação de caracteres ainda em uso mais antiga é a chamada ***American Standard Code for Information Interchange*** **(ASCII)**, criada em 1960.

A codificação ASCII contempla o conjunto de caracteres da língua inglesa, sinais gráficos e alguns caracteres de controle (nova linha, tabulação, etc), totalizando 128 caracteres (0~127).

A codificação ASCII é suportada por todos os sistemas computacionais atuais!

A Codificação ASCII

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

A tabela ASCII, exibida ao lado, tem a seguinte organização geral:

- 0~31 e 127: caracteres de controle
- 32-126: caracteres imprimíveis

Páginas de Codificação

Mas há um ponto interessante a ser notado relacionado à codificação ASCII:

ELA DEFINE 128 CARACTERES!

O que isso significa em termos computacionais? Qual é a consequência disso?

Páginas de Codificação

Na época em que a codificação ASCII foi criada, ela foi implementada em sistemas que utilizavam codificação de sete (7) bits e tinham **imensas limitações de memória**!

Depois, com os sistemas de codificação de oito (8) bits, o oitavo bit passou a ser utilizado como um **bit de paridade**.

Porém, existiam muitos símbolos que não eram representados pela tabela ASCII e era muito tentador **utilizar esse oitavo bit para estender a mesma**.

Páginas de Codificação

É nesse contexto que surgiram as **páginas de codificação, que eram totalmente compatíveis com a codificação ASCII** (primeiros 128 símbolos), mas estendiam a tabela com mais 128 símbolos de acordo com uma necessidade específica!

Isso permitiu a representação de caracteres típicos de linguagens latinas (acentuados, por exemplo) e de outras linguagens específicas (como o russo).

Páginas de Codificação

Algumas páginas de codificação bastante proeminentes são:

- CP-437: codificação usada nos primeiros PCs, com caracteres acentuados e gráficos simples (▒╝╦┼)
- Windows-1252: codificação usada em sistemas Windows mais antigos
- KOI8-R: cirílico russo (Код Обмена Информацией, 8 бит)
- BraSCII: português brasileiro, usada nos anos 1980-90
- ISO-8859: codificações da ISO para diversas línguas

Páginas de Codificação

Particularmente, a ISO-8859 prevê uma página de código conveniente para várias línguas e regiões representativas, por exemplo:

- ISO-8859-1: Europa ocidental (francês, espanhol, italiano, alemão, etc)
- ISO-8859-2: Europa central (Bósnio, Polonês, Croata, etc)
- ISO-8859-6: árabe simplificado
- ISO-8859-7: grego
- ISO-8859-15: revisão do ISO-8859-1, contendo o € e outros símbolos

Qual é o tipo de dados adequado para usar esse tipo de codificação?

Caracteres Multibyte

Porém, mesmo com as páginas de codificação, 256 símbolos ainda é um número muito pequeno para representar toda a variedade de caracteres que linguagens e aplicações requisitam.

E se usássemos mais do que um byte para codificar um caractere?!

CARACTERES MULTIBYTE

Caracteres Multibyte

Diversos padrões de codificação foram propostos utilizando caracteres multibyte, empregando de dois (2) a quatro (4) bytes, por exemplo:

- ISO-2022-CJK: chinês, japonês, coreano
- Shift-JIS: japonês (Windows)
- GB 18030: padrão oficial chinês
- Big5: chinês tradicional (Taiwan)
- Unicode: **SIMPLESMENTE INCRÍVEL!**

Caracteres Multibyte

O padrão Unicode é simplesmente um “campeão de vendas”!

Esse padrão tem uma quantidade enorme de caracteres e está muito longe de ser exaurido. Dentre as principais subdivisões desse padrão, temos:

- UTF-8 (*8-bit Unicode Transformation Format*): usa de 1 a 4 bytes por caractere
- UTF-16 (*16-bit Unicode Transformation Format*): usa 2 ou 4 bytes por caractere; muito usado nas APIs dos sistemas Windows
- UTF-32 (*32-bit Unicode Transformation Format*): usa sempre 4 bytes por caractere

Caracteres Multibyte

Mas sempre existe o melhor entre os melhores...

O nosso campeão é o UTF-8!

Mas, por qual motivo?
A grande sacada do UTF-8 é a **codificação de tamanho ajustável**!

A Codificação UTF-8

Nesse modelo, um caractere é sempre codificado utilizando o menor número de bytes possível. Considere os seguintes exemplos:

Caractere	*Code Point*	Binário	Bits	Bytes
A	0x41	100 0001	7	1
ç	0xE7	1110 0111	8	2
©	0xC2 A9	1100 0010 1010 1001	16	3
😀	0x01 F6 00	1 1111 0110 0000 0000	17	4

???????

A Codificação UTF-8

Para indicar a quantidade de bits sendo usada, é necessário utilizar uma máscara de codificação:

Quantidade de bits do caractere	Máscara de codificação	Bytes
1~7	0xxx-xxxx	1
8~11	110x-xxxx 10xx-xxxx	2
12~16	1110-xxxx 10xx-xxxx 10xx-xxxx	3
17~21	1111-0xxx 10xx-xxxx 10xx-xxxx 10xx-xxxx	4

Conceito de *String*

Strings são sequências de caracteres utilizadas para definir informações textuais.

Tecnicamente, em C, *strings* são compreendidas como vetores de caracteres (declarar uma *string* é declarar um vetor de *char*).

Já é bom se preparar:

C é uma linguagem de programação chata para *strings*!

Conceito de *String*

Vamos começar com o básico!

Como identificamos o final de uma *string*?

O final de uma *string* é identificada por um caractere nulo (‘\0’). Esse caractere não é digitável, apenas serve para interromper o fluxo de processamento de uma *string*.

Assim, se a *string* tem 50 caracteres, o vetor deve ter, pelo menos, 51 posições!

Strings Constantes

A *string* mais fácil!

Strings **constantes são fáceis de criar, basta escrever a sequência de caracteres entre aspas (“”)!**

Vocês já fazem muito isso, por exemplo, para utilizar a função *printf*.

```
printf(“Esta é uma string constante!”);
```

Strings Constantes

```
1)   #include <stdio.h>
2)   int main(){
3)     char *possibilidade1  = "PROG2";
4)     char possibilidade2[]  = "PROG2";
5)     char possibilidade3[]  = { 'P', 'R', 'O',
     'G', '2', '\0' };
6)     printf("%s -- %d\n", possibilidade1,
     possibilidade1);
7)     printf("%s -- %d\n", possibilidade2,
     possibilidade2);
8)     printf("%s -- %d\n", possibilidade3,
     possibilidade3);
9)     return 0;
10)  }
```

Porém, há algumas outras maneiras de definir *strings* constantes; por exemplo, podemos usar um ponteiro para referenciar a mesma.

Veja ao lado algumas alternativas para fazer isso!

Escrita de *Strings*

Tem um ponto interessante no código anterior: a escrita de *strings* na tela!

```
printf("%s\n", pnt_string);
```

Para escrever uma *string* na tela via *printf* devemos:

- Utilizar o indicador de tipo “%s”
- Passar o ponteiro para a *string* desejada como o argumento correspondente

Ocorrerá a escrita de todos os caracteres até que um nulo (\0) seja encontrado.

Escrita de *Strings*

```
1)   #include <stdio.h>
2)   int main(){
3)     char *string = "PROG2";
4)     printf("Alternativa com puts: \n");
5)     puts(string);
6)     printf("\nAlternativa iterativa: \n");
7)     for (int i = 0; string[i]; i++)
     putchar(string[i]);
8)     return 0;
9)   }
```

Existem outras alternativas de escrita de *strings*, algumas delas são:

- Através da função *puts*
- Escrita manual de caracteres

Leitura de *Strings*

A leitura de *strings* é um processo um tanto complicado em C!

De maneira geral, podemos utilizar o *scanf* para realizar esse processo... porém, dependo de como queremos ler a *string* precisamos ajustar a formatação de leitura.

ENTÃO... QUAIS SÃO AS ALTERNATIVAS?

Leitura de *Strings*

```
1)  #include <stdio.h>
2)  int main(){
3)    char string[101];
4)    printf("Digite uma string: ");
5)    scanf("%s", string);
6)    printf("A string digitada foi: %s",
    string);
7)    return 0;
8)  }
```

PRELIMINARES

`char string[101];`

- Qual é o tamanho máximo da *string* suportada pelo vetor declarado?
- Esse vetor pode ser passado por argumento?
- Esse vetor pode ser retornado como resultado de uma função?

Leitura de *Strings*

```
1)  #include <stdio.h>
2)  int main(){
3)    char string[101];
4)    printf("Digite uma string: ");
5)    scanf("%s", string);
6)    printf("A string digitada foi: %s",
    string);
7)    return 0;
8)  }
```

PRIMEIRA ALTERNATIVA

```
scanf("%s", string);
```

A leitura é realizada até que um espaço, tabulação ou fim de linha seja encontrado.

Leitura de *Strings*

```
1)  #include <stdio.h>
2)  int main(){
3)    char string[101];
4)    printf("Digite uma string: ");
5)    scanf("%20s", string);
6)    printf("A string digitada foi: %s",
    string);
7)    return 0;
8)  }
```

SEGUNDA ALTERNATIVA

```
scanf("%20s",
string);
```

A leitura é realizada até que um espaço, tabulação ou fim de linha seja encontrado; a leitura é realizada até vinte (20) caracteres, descartando o excedente.

Leitura de *Strings*

```
1)  #include <stdio.h>
2)  int main(){
3)    char string[101];
4)    printf("Digite uma string: ");
5)    scanf("%[A-Za-z0-9]", string);
6)    printf("A string digitada foi: %s",
    string);
7)    return 0;
8)  }
```

TERCEIRA ALTERNATIVA

```
scanf("%[A-Za-z0-9]",
string);
```

A leitura é realizada até que um espaço, tabulação ou fim de linha seja encontrado; apenas caracteres alfanuméricos são considerados.

Leitura de *Strings*

```
1)  #include <stdio.h>
2)  int main(){
3)    char string[101];
4)    printf("Digite uma string: ");
5)    scanf("%[^\n]", string);
6)    printf("A string digitada foi: %s",
    string);
7)    return 0;
8)  }
```

QUARTA ALTERNATIVA

```
scanf("%[^\n]",
nome);
```

Todos os caracteres que não são quebra de linhas são lidos; a leitura para na quebra de linha (que não é lida).

AQUI TEM UM PROBLEMA!

Leitura de *Strings*

```
1)  #include <stdio.h>
2)  int main(){
3)    char string[101];
4)    printf("Digite uma string: ");
5)    scanf("%[^\n]", string);
6)    getchar();
7)    printf("A string digitada foi: %s",
    string);
8)    return 0;
9)  }
```

QUARTA ALTERNATIVA

```
scanf("%[^\n]",
nome);
getchar();
```

A função *getchar* faz a limpeza do *buffer*, removendo a quebra de linha do mesmo.

Manipulação de *Strings*

Apesar das *strings* "não serem o forte" da linguagem C, existem bibliotecas para manipular às mesmas. A principal biblioteca de *strings* é a chamada *string.h* que fornece funções como:

strlen, *strcpy*, *strncpy*, *strcat*, *strncat*, *strchr*, *strrchr*, *strstr*, *strdup e strtok*

VAMOS ANALISAR ESSAS FUNÇÕES E VER ALGUNS EXEMPLOS DE USO

Manipulação de *Strings*

```
1)   #include <stdio.h>
2)   #include <string.h>
3)   int main(){
4)       char string[101];
5)       printf("Digite uma string: " );
6)       scanf("%s", string);
7)       printf("A string digitada tem  %d
     caracteres!", strlen(string));
8)       return 0;
9)   }
```

STRLEN

Recebe o ponteiro para uma *string* como argumento; retorna a quantidade de caracteres na mesma (sem contar o '\0').

Manipulação de *Strings*

```
1)   #include <stdio.h>
2)   #include <string.h>
3)   int main(){
4)       char string[101], copia[101];
5)       printf("Digite uma string: ");
6)       scanf("%s", string);
7)       strcpy(copia, string);
8)       printf("A string digitada foi: %s",
     copia);
9)       return 0;
10)  }
```

STRCPY

Recebe, em ordem, o ponteiro para onde a *string* base deve ser copiada (**um vetor de *char* com tamanho suficiente**) e o ponteiro para a *string* base; retorna o ponteiro para a *string* copiada.

Manipulação de *Strings*

```
1)   #include <stdio.h>
2)   #include <string.h>
3)   int main(){
4)       char string[101], copia[101];
5)       printf("Digite uma string: ");
6)       scanf("%s", string);
7)       strncpy(copia, string, 4);
8)       printf("A string digitada foi: %s",
     copia);
9)       return 0;
10)  }
```

STRNCPY

Mesmo funcionamento da *strcpy*. Porém, recebe um terceiro argumento inteiro indicando quantos caracteres devem ser copiados, no máximo (excluindo o '\0').

Manipulação de *Strings*

```
1)   #include <stdio.h>
2)   #include <string.h>
3)   int main(){
4)       char string01[101], string02[101];
5)       printf("Digite uma string: " );
6)       scanf("%s", string01);
7)       printf("Digite outra string: " );
8)       scanf("%s", string02);
9)       if ((strlen(string01) <= 50) &&
     (strlen(string02) <= 50)){
10)          strcat(string01, string02);
11)          printf("A concatenacao das strings
     e: %s", string01);
12)      }
13)      return 0;
14)  }
```

STRCAT

Recebe dois ponteiros para *strings*, sendo que a segunda *string* deve ser concatenada na primeira (a primeira string deve ter espaço previamente alocado); retorna o ponteiro para a *string* copiada.

Manipulação de *Strings*

```
1)   #include <stdio.h>
2)   #include <string.h>
3)   int main(){
4)       char string01[101], string02[101];
5)       printf("Digite uma string: " );
6)       scanf("%s", string01);
7)       printf("Digite outra string: " );
8)       scanf("%s", string02);
9)       if (strlen(string01) <= 50){
10)          strncat(string01, string02, 50);
11)          printf("A concatenacao das strings
     e: %s", string01);
12)      }
13)      return 0;
14)  }
```

STRNCAT

Mesmo funcionamento da *strcat*. Porém, recebe um terceiro argumento inteiro indicando quantos caracteres devem ser concatenados, no máximo (excluindo o '\0').

Manipulação de *Strings*

```
1)   #include <stdio.h>
2)   #include <string.h>
3)   int main(){
4)       char string[101], *busca;
5)       printf("Digite uma string: " );
6)       scanf("%s", string);
7)       busca = strchr(string, 'a');
8)       if (busca != 0)
9)           printf("O primeiro 'a' encontrado
     esta no indice %d", busca-string);
10)      else
11)          printf("Nao existe a letra 'a' na
     string!");
12)      return 0;
13)  }
```

STRCHR

Recebe um ponteiro para *string* e um caractere qualquer; retorna o ponteiro para a primeira ocorrência do caractere na *string* fornecida. Se não houver ocorrências, retorna *NULL* (0).

Manipulação de *Strings*

```
1)   #include <stdio.h>
2)   #include <string.h>
3)   int main(){
4)       char string[101], *busca;
5)       printf("Digite uma string: " );
6)       scanf("%s", string);
7)       busca = strrchr(string, 'a');
8)       if (busca != 0)
9)           printf("O ultimo 'a' encontrado
     está no indice %d", busca-string);
10)      else
11)          printf("Nao existe a letra 'a' na
     string!");
12)      return 0;
13)  }
```

STRRCHR

Recebe um ponteiro para *string* e um caractere qualquer; retorna o ponteiro para a última ocorrência do caractere na *string* fornecida. Se não houver ocorrências, retorna *NULL* (0).

Manipulação de *Strings*

```
1)   #include <stdio.h>
2)   #include <string.h>
3)   int main(){
4)       char string[101], *busca;
5)       printf("Digite uma string: " );
6)       scanf("%s", string);
7)       busca = strstr(string, "aba");
8)       if (busca != 0)
9)           printf("A substring esta a partir
     do indice %d", busca-string);
10)      else
11)          printf("Nao existe a substring
     'aba' na string!" );
12)      return 0;
13)  }
```

STRSTR

Recebe um ponteiro para *string* base e um para uma *substring*; retorna o ponteiro para a primeira ocorrência da *substring* na *string* fornecida. Se não houver ocorrências, retorna *NULL* (0).

Manipulação de *Strings*

```
1)   #include <stdio.h>
2)   #include <string.h>
3)   int main(){
4)       char string01[101], *string02;
5)       printf("Digite uma string: " );
6)       scanf("%s", string01);
7)       string02  = strdup(string01);
8)       printf("A string digitada foi:  %s",
     string02);
9)       return 0;
10)  }
```

STRDUP

Recebe um ponteiro para uma *string* como argumento e duplica a mesma, alocando a memória necessária; retorna o ponteiro para a *string* duplicada.

Manipulação de *Strings*

```
1)   #include <stdio.h>
2)   #include <string.h>
3)   int main(){
4)       char string[101], separador[] = " - ",
     *token;
5)       printf("Digite uma string: ");
6)       scanf("%[^\n]", string);
7)       getchar();;
8)       token = strtok(string, separador);
9)       do {
10)          printf("O token encontrado foi:
     %s\n", token);
11)          token = strtok(NULL, separador);
12)      } while(token);
13)      return 0;
14)  }
```

STRTOK

Recebe um ponteiro para *string* base e um para uma *substring* separadora; retorna o ponteiro para a primeira ocorrência de um caractere na *string* base antes/após a *substring* fornecida. Se não houver ocorrências, retorna *NULL* (0).

Exercício #3

Você já ouviu falar em CSV?

O formato CSV organiza dados de maneira tabular em texto simples, onde cada linha de dados é uma *string* e a separação entre uma coluna e outra é demarcada por um símbolo de vírgula (,).

Neste momento, para simplificar, vamos considerar que dados de uma coluna não podem conter o símbolo de vírgula (,).

Faça um programa que recebe uma *string* de uma linha CSV e apresente o valor de cada coluna separadamente. Numere as colunas por ordem de leitura.

Obrigado!

Vinícius Fülber Garcia
inf.ufpr.br/vinicius
vinicius@inf.ufpr.br