MC-202
Curso de C — Parte 3

Lehilton Pedrosa
lehilton@ic.unicamp.br

Universidade Estadual de Campinas

Segundo semestre de 2024

Cifra de César

A Cifra de César é uma das formas mais simples de criptografia

Cifra de César

A Cifra de César é uma das formas mais simples de criptografia

- E uma das mais fáceis de quebrar...

Cifra de César

A Cifra de César é uma das formas mais simples de criptografia

- E uma das mais fáceis de quebrar...
- Dado um parâmetro inteiro k

Cifra de César

A Cifra de César é uma das formas mais simples de criptografia

- E uma das mais fáceis de quebrar...
- Dado um parâmetro inteiro k
- cada letra é trocada pela k-ésima letra após ela

Cifra de César

A Cifra de César é uma das formas mais simples de criptografia

- E uma das mais fáceis de quebrar...
- Dado um parâmetro inteiro k
- cada letra é trocada pela k-ésima letra após ela
  - Se $k = 1$, a é trocada por b, b por c, c por d, etc

Cifra de César

A Cifra de César é uma das formas mais simples de criptografia

- E uma das mais fáceis de quebrar...
- Dado um parâmetro inteiro k
- cada letra é trocada pela k-ésima letra após ela
  - Se $k = 1$, a é trocada por b, b por c, c por d, etc
  - Se $k = 2$, a é trocada por c, b por d, c por e, etc

Cifra de César

A Cifra de César é uma das formas mais simples de criptografia

- E uma das mais fáceis de quebrar...
- Dado um parâmetro inteiro k
- cada letra é trocada pela k-ésima letra após ela
  - Se $k = 1$, a é trocada por b, b por c, c por d, etc
  - Se $k = 2$, a é trocada por c, b por d, c por e, etc
- ao chegar no final do alfabeto, nós voltamos para o início

Cifra de César

A Cifra de César é uma das formas mais simples de criptografia

- E uma das mais fáceis de quebrar...
- Dado um parâmetro inteiro k
- cada letra é trocada pela k-ésima letra após ela
  - Se $k = 1$, a é trocada por b, b por c, c por d, etc
  - Se $k = 2$, a é trocada por c, b por d, c por e, etc
- ao chegar no final do alfabeto, nós voltamos para o início

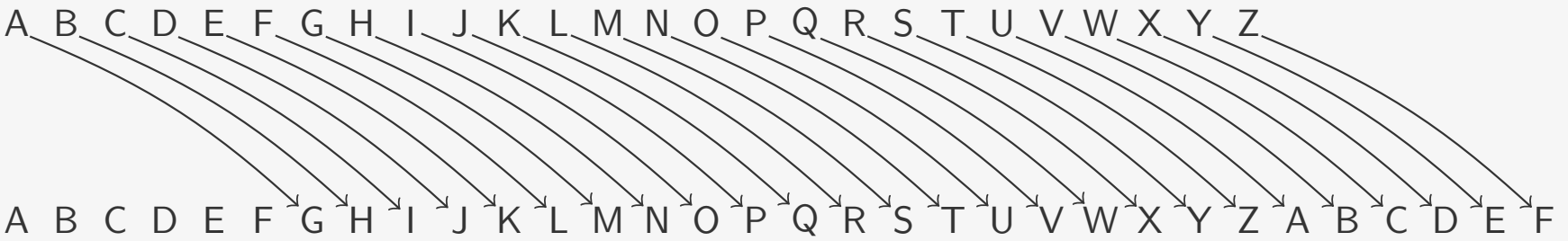
Cifra de César para $k = 6$:

Cifra de César

A Cifra de César é uma das formas mais simples de criptografia

- E uma das mais fáceis de quebrar...
- Dado um parâmetro inteiro k
- cada letra é trocada pela k-ésima letra após ela
  - Se $k = 1$, a é trocada por b, b por c, c por d, etc
  - Se $k = 2$, a é trocada por c, b por d, c por e, etc
- ao chegar no final do alfabeto, nós voltamos para o início

Cifra de César para $k = 6$:

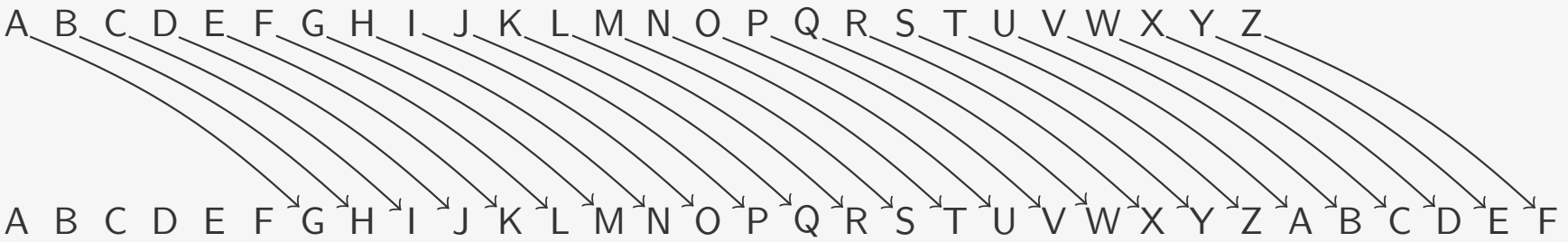


Cifra de César

A Cifra de César é uma das formas mais simples de criptografia

- E uma das mais fáceis de quebrar...
- Dado um parâmetro inteiro k
- cada letra é trocada pela k-ésima letra após ela
  - Se $k = 1$, a é trocada por b, b por c, c por d, etc
  - Se $k = 2$, a é trocada por c, b por d, c por e, etc
- ao chegar no final do alfabeto, nós voltamos para o início

Cifra de César para $k = 6$:



Para desencriptar, basta fazer o mesmo processo para $26 - k$

Cifra de César em C

Vamos fazer um programa que encripta uma sequência de letras usando a cifra de César

Cifra de César em C

Vamos fazer um programa que encripta uma sequência de letras usando a cifra de César

Para isso precisamos:

Cifra de César em C

Vamos fazer um programa que encripta uma sequência de letras usando a cifra de César

Para isso precisamos:

- Saber como representar letras no C

Cifra de César em C

Vamos fazer um programa que encripta uma sequência de letras usando a cifra de César

Para isso precisamos:

- Saber como representar letras no C
- Como ler e imprimir letras no C

Cifra de César em C

Vamos fazer um programa que encripta uma sequência de letras usando a cifra de César

Para isso precisamos:

- Saber como representar letras no C
- Como ler e imprimir letras no C
- Como converter as letras de uma maneira prática

O tipo `char`

Uma letra ou caractere em C é representado pelo tipo `char`

O tipo `char`

Uma letra ou caractere em C é representado pelo tipo `char`

- é um número inteiro

O tipo `char`

Uma letra ou caractere em C é representado pelo tipo `char`

- é um número inteiro
  - normalmente tem 8 bits (está entre -128 e 127)

O tipo `char`

Uma letra ou caractere em C é representado pelo tipo `char`

- é um número inteiro
  - normalmente tem 8 bits (está entre -128 e 127)
  - podemos somar, subtrair, multiplicar, dividir, etc

O tipo `char`

Uma letra ou caractere em C é representado pelo tipo `char`

- é um número inteiro
  - normalmente tem 8 bits (está entre -128 e 127)
  - podemos somar, subtrair, multiplicar, dividir, etc
  - como se fosse um `int` mas com menos valores válidos

O tipo `char`

Uma letra ou caractere em C é representado pelo tipo `char`

- é um número inteiro
  - normalmente tem 8 bits (está entre -128 e 127)
  - podemos somar, subtrair, multiplicar, dividir, etc
  - como se fosse um `int` mas com menos valores válidos
- representa caracteres usando a tabela ASCII

O tipo `char`

Uma letra ou caractere em C é representado pelo tipo `char`

- é um número inteiro
  - normalmente tem 8 bits (está entre -128 e 127)
  - podemos somar, subtrair, multiplicar, dividir, etc
  - como se fosse um `int` mas com menos valores válidos
- representa caracteres usando a tabela ASCII
  - cada número representa um caractere

O tipo `char`

Uma letra ou caractere em C é representado pelo tipo `char`

- é um número inteiro
  - normalmente tem 8 bits (está entre -128 e 127)
  - podemos somar, subtrair, multiplicar, dividir, etc
  - como se fosse um `int` mas com menos valores válidos
- representa caracteres usando a tabela ASCII
  - cada número representa um caractere
- representamos constantes usando aspas simples

O tipo `char`

Uma letra ou caractere em C é representado pelo tipo `char`

- é um número inteiro
  - normalmente tem 8 bits (está entre -128 e 127)
  - podemos somar, subtrair, multiplicar, dividir, etc
  - como se fosse um `int` mas com menos valores válidos
- representa caracteres usando a tabela ASCII
  - cada número representa um caractere
- representamos constantes usando aspas simples
  - ex: `'a'`, `'b'`, `'c'`, `'\n'`, etc...

O tipo `char`

Uma letra ou caractere em C é representado pelo tipo `char`

- é um número inteiro
  - normalmente tem 8 bits (está entre -128 e 127)
  - podemos somar, subtrair, multiplicar, dividir, etc
  - como se fosse um `int` mas com menos valores válidos
- representa caracteres usando a tabela ASCII
  - cada número representa um caractere
- representamos constantes usando aspas simples
  - ex: `'a'`, `'b'`, `'c'`, `'\n'`, etc...
  - `'a'` significa o número do caractere `a` na tabela ASCII

O tipo `char`

Uma letra ou caractere em C é representado pelo tipo `char`

- é um número inteiro
  - normalmente tem 8 bits (está entre -128 e 127)
  - podemos somar, subtrair, multiplicar, dividir, etc
  - como se fosse um `int` mas com menos valores válidos
- representa caracteres usando a tabela ASCII
  - cada número representa um caractere
- representamos constantes usando aspas simples
  - ex: `'a'`, `'b'`, `'c'`, `'\n'`, etc...
  - `'a'` significa o número do caractere `a` na tabela ASCII
  - não precisamos saber qual é esse número exatamente...

O tipo `char`

Uma letra ou caractere em C é representado pelo tipo `char`

- é um número inteiro
  - normalmente tem 8 bits (está entre -128 e 127)
  - podemos somar, subtrair, multiplicar, dividir, etc
  - como se fosse um `int` mas com menos valores válidos
- representa caracteres usando a tabela ASCII
  - cada número representa um caractere
- representamos constantes usando aspas simples
  - ex: `'a'`, `'b'`, `'c'`, `'\n'`, etc...
  - `'a'` significa o número do caractere `a` na tabela ASCII
  - não precisamos saber qual é esse número exatamente...
- para ler e imprimir usamos `%c`

O tipo `char`

Uma letra ou caractere em C é representado pelo tipo `char`

- é um número inteiro
  - normalmente tem 8 bits (está entre -128 e 127)
  - podemos somar, subtrair, multiplicar, dividir, etc
  - como se fosse um `int` mas com menos valores válidos
- representa caracteres usando a tabela ASCII
  - cada número representa um caractere
- representamos constantes usando aspas simples
  - ex: `'a'`, `'b'`, `'c'`, `'\n'`, etc...
  - `'a'` significa o número do caractere `a` na tabela ASCII
  - não precisamos saber qual é esse número exatamente...
- para ler e imprimir usamos `%c`
  - quando queremos o caractere em si

O tipo `char`

Uma letra ou caractere em C é representado pelo tipo `char`

- é um número inteiro
  - normalmente tem 8 bits (está entre -128 e 127)
  - podemos somar, subtrair, multiplicar, dividir, etc
  - como se fosse um `int` mas com menos valores válidos
- representa caracteres usando a tabela ASCII
  - cada número representa um caractere
- representamos constantes usando aspas simples
  - ex: `'a'`, `'b'`, `'c'`, `'\n'`, etc...
  - `'a'` significa o número do caractere `a` na tabela ASCII
  - não precisamos saber qual é esse número exatamente...
- para ler e imprimir usamos `%c`
  - quando queremos o caractere em si
  - ex: `printf("letra: %c, código: %d", 'a', 'a');`

O tipo `char`

Uma letra ou caractere em C é representado pelo tipo `char`

- é um número inteiro
  - normalmente tem 8 bits (está entre -128 e 127)
  - podemos somar, subtrair, multiplicar, dividir, etc
  - como se fosse um `int` mas com menos valores válidos
- representa caracteres usando a tabela ASCII
  - cada número representa um caractere
- representamos constantes usando aspas simples
  - ex: `'a'`, `'b'`, `'c'`, `'\n'`, etc...
  - `'a'` significa o número do caractere `a` na tabela ASCII
  - não precisamos saber qual é esse número exatamente...
- para ler e imprimir usamos `%c`
  - quando queremos o caractere em si
  - ex: `printf("letra: %c, código: %d", 'a', 'a');`
    - imprime `letra: a, código: 97`

Tabela ASCII

32	(espaço)	51	3	70	F	89	Y	108	l
33	!	52	4	71	G	90	Z	109	m
34	”	53	5	72	H	91	[	110	n
35	#	54	6	73	I	92	\	111	o
36	$	55	7	74	J	93	]	112	p
37	%	56	8	75	K	94	^	113	q
38	&	57	9	76	L	95	_	114	r
39	’	58	:	77	M	96	‘	115	s
40	(	59	;	78	N	97	a	116	t
41	)	60	<	79	O	98	b	117	u
42	*	61	=	80	P	99	c	118	v
43	+	62	>	81	Q	100	d	119	w
44	,	63	?	82	R	101	e	120	x
45	-	64	@	83	S	102	f	121	y
46	.	65	A	84	T	103	g	122	z
47	/	66	B	85	U	104	h	123	{
48	0	67	C	86	V	105	i	124	|
49	1	68	D	87	W	106	j	125	}
50	2	69	E	88	X	107	k	126	~

Tabela ASCII

32	(espaço)	51	3	70	F	89	Y	108	l
33	!	52	4	71	G	90	Z	109	m
34	”	53	5	72	H	91	[	110	n
35	#	54	6	73	I	92	\	111	o
36	$	55	7	74	J	93	]	112	p
37	%	56	8	75	K	94	^	113	q
38	&	57	9	76	L	95	_	114	r
39	’	58	:	77	M	96	‘	115	s
40	(	59	;	78	N	97	a	116	t
41	)	60	<	79	O	98	b	117	u
42	*	61	=	80	P	99	c	118	v
43	+	62	>	81	Q	100	d	119	w
44	,	63	?	82	R	101	e	120	x
45	-	64	@	83	S	102	f	121	y
46	.	65	A	84	T	103	g	122	z
47	/	66	B	85	U	104	h	123	{
48	0	67	C	86	V	105	i	124	|
49	1	68	D	87	W	106	j	125	}
50	2	69	E	88	X	107	k	126	~

Existem também `\t` (9 - tab) e `\n` (12 - quebra de linha)

Tabela ASCII

32	(espaço)	51	3	70	F	89	Y	108	l
33	!	52	4	71	G	90	Z	109	m
34	"	53	5	72	H	91	[	110	n
35	#	54	6	73	I	92	\	111	o
36	$	55	7	74	J	93	]	112	p
37	%	56	8	75	K	94	^	113	q
38	&	57	9	76	L	95	_	114	r
39	'	58	:	77	M	96	`	115	s
40	(	59	;	78	N	97	a	116	t
41	)	60	<	79	O	98	b	117	u
42	*	61	=	80	P	99	c	118	v
43	+	62	>	81	Q	100	d	119	w
44	,	63	?	82	R	101	e	120	x
45	-	64	@	83	S	102	f	121	y
46	.	65	A	84	T	103	g	122	z
47	/	66	B	85	U	104	h	123	{
48	0	67	C	86	V	105	i	124	|
49	1	68	D	87	W	106	j	125	}
50	2	69	E	88	X	107	k	126	~

Existem também `\t` (9 - tab) e `\n` (12 - quebra de linha)

- Outros códigos não-negativos não são imprimíveis

Tabela ASCII

32	(espaço)	51	3	70	F	89	Y	108	l
33	!	52	4	71	G	90	Z	109	m
34	"	53	5	72	H	91	[	110	n
35	#	54	6	73	I	92	\	111	o
36	$	55	7	74	J	93	]	112	p
37	%	56	8	75	K	94	^	113	q
38	&	57	9	76	L	95	_	114	r
39	'	58	:	77	M	96	`	115	s
40	(	59	;	78	N	97	a	116	t
41	)	60	<	79	O	98	b	117	u
42	*	61	=	80	P	99	c	118	v
43	+	62	>	81	Q	100	d	119	w
44	,	63	?	82	R	101	e	120	x
45	-	64	@	83	S	102	f	121	y
46	.	65	A	84	T	103	g	122	z
47	/	66	B	85	U	104	h	123	{
48	0	67	C	86	V	105	i	124	|
49	1	68	D	87	W	106	j	125	}
50	2	69	E	88	X	107	k	126	~

Existem também `\t` (9 - tab) e `\n` (12 - quebra de linha)

- Outros códigos não-negativos não são imprimíveis
- Códigos negativos são usados em outras tabelas

O programa

```c
#include <stdio.h>

int main() {
  int k;
  char original, encriptado, pos_original, pos_encriptado;
  scanf("%d ", &k);
  scanf("%c", &original);
  while (original != '#') {
    pos_original = original - 'A';
    pos_encriptado = (pos_original + k) % 26;
    encriptado = 'A' + pos_encriptado;
    printf("%c", encriptado);
    scanf("%c", &original);
  }
  printf("\n");
  return 0;
}
```

O programa

```c
#include <stdio.h>

int main() {
  int k;
  char original, encriptado, pos_original, pos_encriptado;
  scanf("%d ", &k);
  scanf("%c", &original);
  while (original != '#') {
    pos_original = original - 'A';
    pos_encriptado = (pos_original + k) % 26;
    encriptado = 'A' + pos_encriptado;
    printf("%c", encriptado);
    scanf("%c", &original);
  }
  printf("\n");
  return 0;
}
```

Detalhes:

O programa

```c
#include <stdio.h>

int main() {
  int k;
  char original, encriptado, pos_original, pos_encriptado;
  scanf("%d ", &k);
  scanf("%c", &original);
  while (original != '#') {
    pos_original = original - 'A';
    pos_encriptado = (pos_original + k) % 26;
    encriptado = 'A' + pos_encriptado;
    printf("%c", encriptado);
    scanf("%c", &original);
  }
  printf("\n");
  return 0;
}
```

Detalhes:

- Há um espaço após o %d

O programa

```c
#include <stdio.h>

int main() {
  int k;
  char original, encriptado, pos_original, pos_encriptado;
  scanf("%d ", &k);
  scanf("%c", &original);
  while (original != '#') {
    pos_original = original - 'A';
    pos_encriptado = (pos_original + k) % 26;
    encriptado = 'A' + pos_encriptado;
    printf("%c", encriptado);
    scanf("%c", &original);
  }
  printf("\n");
  return 0;
}
```

Detalhes:

- Há um espaço após o `%d`
  - consome os próximos caracteres brancos: espaço, `\n` e `\t`

O programa

```c
#include <stdio.h>

int main() {
  int k;
  char original, encriptado, pos_original, pos_encriptado;
  scanf("%d ", &k);
  scanf("%c", &original);
  while (original != '#') {
    pos_original = original - 'A';
    pos_encriptado = (pos_original + k) % 26;
    encriptado = 'A' + pos_encriptado;
    printf("%c", encriptado);
    scanf("%c", &original);
  }
  printf("\n");
  return 0;
}
```

Detalhes:

- Há um espaço após o `%d`
  - consome os próximos caracteres brancos: espaço, `\n` e `\t`
  - sem isso, o `scanf` leria um `\n`

O programa

```c
#include <stdio.h>

int main() {
  int k;
  char original, encriptado, pos_original, pos_encriptado;
  scanf("%d ", &k);
  scanf("%c", &original);
  while (original != '#') {
    pos_original = original - 'A';
    pos_encriptado = (pos_original + k) % 26;
    encriptado = 'A' + pos_encriptado;
    printf("%c", encriptado);
    scanf("%c", &original);
  }
  printf("\n");
  return 0;
}
```

Detalhes:

- Há um espaço após o `%d`
  - consome os próximos caracteres brancos: espaço, `\n` e `\t`
  - sem isso, o `scanf` leria um `\n`
  - cuidado, o C é chato na leitura de caracteres...

Comparações e Operadores Lógicos em C

Como no Python, os operadores de comparação a seguir:

Comparações e Operadores Lógicos em C

Como no Python, os operadores de comparação a seguir:

- `<`, `<=`, `>`, `>=`, `==` e `!=`

Comparações e Operadores Lógicos em C

Como no Python, os operadores de comparação a seguir:

- `<`, `<=`, `>`, `>=`, `==` e `!=`
- mas não temos o operador `is`

Comparações e Operadores Lógicos em C

Como no Python, os operadores de comparação a seguir:

- `<`, `<=`, `>`, `>=`, `==` e `!=`
- mas não temos o operador `is`

Em C, não temos o tipo `bool`

Comparações e Operadores Lógicos em C

Como no Python, os operadores de comparação a seguir:

- `<`, `<=`, `>`, `>=`, `==` e `!=`
- mas não temos o operador `is`

Em C, não temos o tipo `bool`

- O C considera o valor `0` como falso

Comparações e Operadores Lógicos em C

Como no Python, os operadores de comparação a seguir:

- `<`, `<=`, `>`, `>=`, `==` e `!=`
- mas não temos o operador `is`

Em C, não temos o tipo `bool`

- O C considera o valor `0` como falso
- E valores diferentes de `0` como verdadeiro

Comparações e Operadores Lógicos em C

Como no Python, os operadores de comparação a seguir:

- `<`, `<=`, `>`, `>=`, `==` e `!=`
- mas não temos o operador `is`

Em C, não temos o tipo `bool`

- O C considera o valor `0` como falso
- E valores diferentes de `0` como verdadeiro

Os operadores lógicos são diferentes em C:

Comparações e Operadores Lógicos em C

Como no Python, os operadores de comparação a seguir:

- `<`, `<=`, `>`, `>=`, `==` e `!=`
- mas não temos o operador `is`

Em C, não temos o tipo `bool`

- O C considera o valor `0` como falso
- E valores diferentes de `0` como verdadeiro

Os operadores lógicos são diferentes em C:

	Python	C
E	`and`	`&&`
Ou	`or`	`||`
Não	`not`	`!`

Busca em um texto

Queremos buscar por um padrão em um texto

1

Busca em um texto

Queremos buscar por um padrão em um texto

- Um símbolo $*$ representa um caractere coringa

1

Busca em um texto

Queremos buscar por um padrão em um texto

- Um símbolo `*` representa um caractere coringa

Por exemplo, se procurarmos por `*os` no seguinte texto:

1

Busca em um texto

Queremos buscar por um padrão em um texto

- Um símbolo `*` representa um caractere coringa

Por exemplo, se procurarmos por `*os` no seguinte texto:

> *Muito além, nos confins inexplorados da região mais brega da Borda Ocidental desta Galáxia, há um pequeno sol amarelo e esquecido.*[1]

[1] Douglas Adams, O Guia do Mochileiro das Galáxias, Editora Arquiteto, 2004

Busca em um texto

Queremos buscar por um padrão em um texto

- Um símbolo `*` representa um caractere coringa

Por exemplo, se procurarmos por `*os` no seguinte texto:

> *Muito além, nos confins inexplorados da região mais brega da Borda Ocidental desta Galáxia, há um pequeno sol amarelo e esquecido.*[1]

encontraremos `nos` e `dos`

[1] Douglas Adams, O Guia do Mochileiro das Galáxias, Editora Arquiteto, 2004

Ideia do algoritmo

Para cada posição do texto, verifique se o padrão começa ali

Ideia do algoritmo

Para cada posição do texto, verifique se o padrão começa ali

- Existem algoritmos melhores do que esse

Ideia do algoritmo

Para cada posição do texto, verifique se o padrão começa ali

- Existem algoritmos melhores do que esse
- Vamos trabalhar com strings sem acentos

Ideia do algoritmo

Para cada posição do texto, verifique se o padrão começa ali

- Existem algoritmos melhores do que esse
- Vamos trabalhar com strings sem acentos

De novo, vamos listar as tarefas de que precisamos:

Ideia do algoritmo

Para cada posição do texto, verifique se o padrão começa ali

- Existem algoritmos melhores do que esse
- Vamos trabalhar com strings sem acentos

De novo, vamos listar as tarefas de que precisamos:

- verificar se o padrão ocorre em uma posição do texto:

Ideia do algoritmo

Para cada posição do texto, verifique se o padrão começa ali

- Existem algoritmos melhores do que esse
- Vamos trabalhar com strings sem acentos

De novo, vamos listar as tarefas de que precisamos:

- verificar se o padrão ocorre em uma posição do texto:
  - `int ocorre(char texto[], int pos, char padrao[])`

Ideia do algoritmo

Para cada posição do texto, verifique se o padrão começa ali

- Existem algoritmos melhores do que esse
- Vamos trabalhar com strings sem acentos

De novo, vamos listar as tarefas de que precisamos:

- verificar se o padrão ocorre em uma posição do texto:
  - `int ocorre(char texto[], int pos, char padrao[])`
- imprimir um trecho do texto:

Ideia do algoritmo

Para cada posição do texto, verifique se o padrão começa ali

- Existem algoritmos melhores do que esse
- Vamos trabalhar com strings sem acentos

De novo, vamos listar as tarefas de que precisamos:

- verificar se o padrão ocorre em uma posição do texto:
  - `int ocorre(char texto[], int pos, char padrao[])`
- imprimir um trecho do texto:
  - `void imprime_trecho(char texto[], int ini, int tam)`

Ideia do algoritmo

Para cada posição do texto, verifique se o padrão começa ali

- Existem algoritmos melhores do que esse
- Vamos trabalhar com strings sem acentos

De novo, vamos listar as tarefas de que precisamos:

- verificar se o padrão ocorre em uma posição do texto:
  - `int ocorre(char texto[], int pos, char padrao[])`
- imprimir um trecho do texto:
  - `void imprime_trecho(char texto[], int ini, int tam)`
- medir o tamanho de uma string:

Ideia do algoritmo

Para cada posição do texto, verifique se o padrão começa ali

- Existem algoritmos melhores do que esse
- Vamos trabalhar com strings sem acentos

De novo, vamos listar as tarefas de que precisamos:

- verificar se o padrão ocorre em uma posição do texto:
  - `int ocorre(char texto[], int pos, char padrao[])`
- imprimir um trecho do texto:
  - `void imprime_trecho(char texto[], int ini, int tam)`
- medir o tamanho de uma string:
  - `int tamanho(char string[])`

String em C

Strings em C são vetores de `char` terminados com `'\0'`

String em C

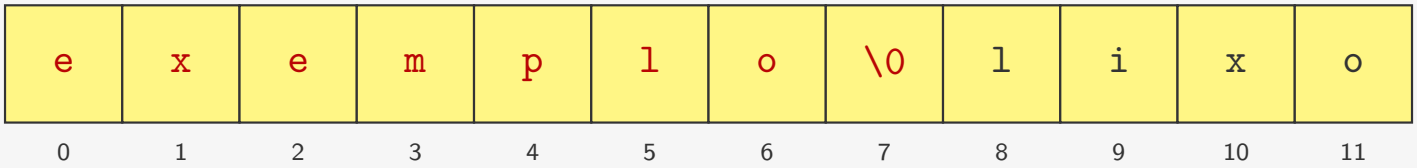
Strings em C são vetores de `char` terminados com `'\0'`

- Por exemplo, podemos ter um vetor de `char` com 12 posições mas a string ter apenas 7 caracteres

String em C

Strings em C são vetores de `char` terminados com `'\0'`

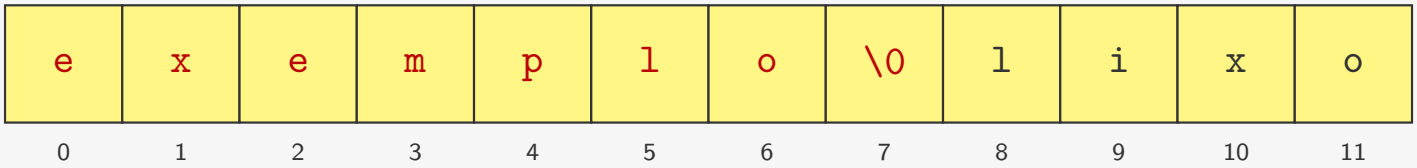
- Por exemplo, podemos ter um vetor de `char` com 12 posições mas a string ter apenas 7 caracteres



String em C

Strings em C são vetores de `char` terminados com `'\0'`

- Por exemplo, podemos ter um vetor de `char` com 12 posições mas a string ter apenas 7 caracteres

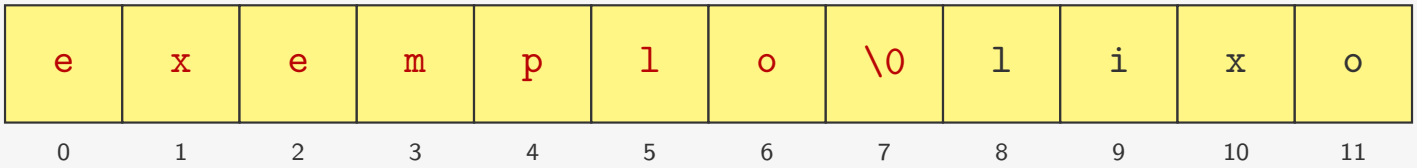


O tamanho da string é o número de caracteres antes do `'\0'`

String em C

Strings em C são vetores de `char` terminados com `'\0'`

- Por exemplo, podemos ter um vetor de `char` com 12 posições mas a string ter apenas 7 caracteres



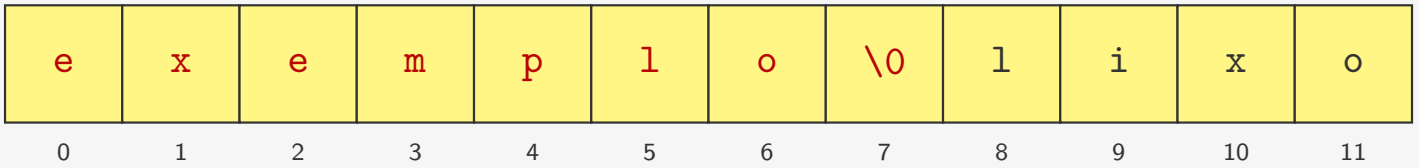
O tamanho da string é o número de caracteres antes do `'\0'`

```
1 int tamanho(char string[]) {
2   int i;
3   for (i = 0; string[i] != '\0'; i++) ;
4   return i;
5 }
```

String em C

Strings em C são vetores de `char` terminados com `'\0'`

- Por exemplo, podemos ter um vetor de `char` com 12 posições mas a string ter apenas 7 caracteres



O tamanho da string é o número de caracteres antes do `'\0'`

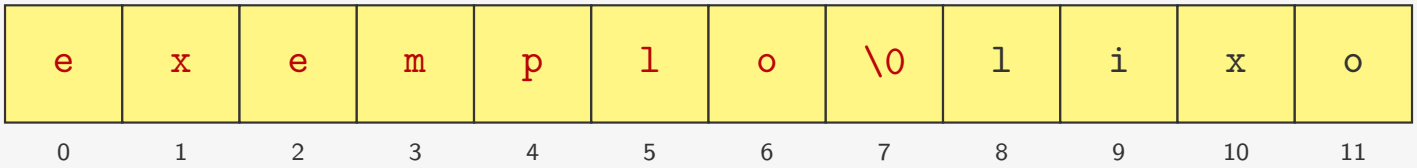
```
1 int tamanho(char string[]) {
2   int i;
3   for (i = 0; string[i] != '\0'; i++) ;
4   return i;
5 }
```

Note que esse `for` tem um bloco vazio

String em C

Strings em C são vetores de `char` terminados com `'\0'`

- Por exemplo, podemos ter um vetor de `char` com 12 posições mas a string ter apenas 7 caracteres



O tamanho da string é o número de caracteres antes do `'\0'`

```
1 int tamanho(char string[]) {
2   int i;
3   for (i = 0; string[i] != '\0'; i++) ;
4   return i;
5 }
```

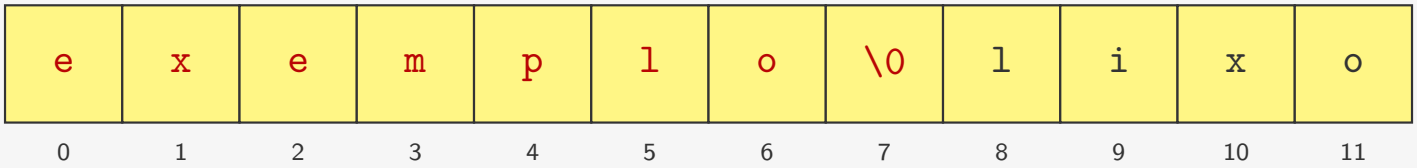
Note que esse `for` tem um bloco vazio

- é raro usarmos isso (e algumas pessoas não gostam)

String em C

Strings em C são vetores de `char` terminados com `'\0'`

- Por exemplo, podemos ter um vetor de `char` com 12 posições mas a string ter apenas 7 caracteres



O tamanho da string é o número de caracteres antes do `'\0'`

```
1 int tamanho(char string[]) {
2   int i;
3   for (i = 0; string[i] != '\0'; i++) ;
4   return i;
5 }
```

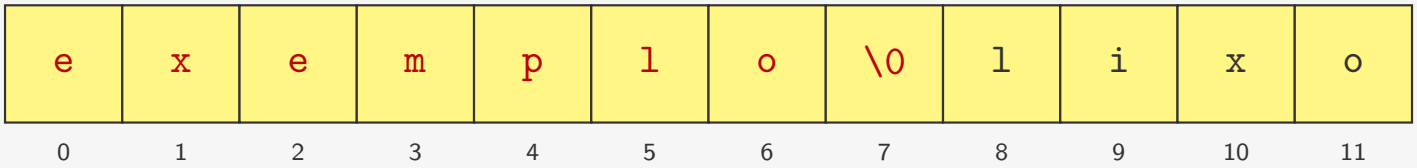
Note que esse `for` tem um bloco vazio

- é raro usarmos isso (e algumas pessoas não gostam)
- poderia ser trocado por um while (exercício)

String em C

Strings em C são vetores de `char` terminados com `'\0'`

- Por exemplo, podemos ter um vetor de `char` com 12 posições mas a string ter apenas 7 caracteres



O tamanho da string é o número de caracteres antes do `'\0'`

```
1 int tamanho(char string[]) {
2   int i;
3   for (i = 0; string[i] != '\0'; i++) ;
4   return i;
5 }
```

Note que esse `for` tem um bloco vazio

- é raro usarmos isso (e algumas pessoas não gostam)
- poderia ser trocado por um while (exercício)
- um `for` desses pode ser um bug no seu programa

Imprimindo uma substring

Queremos uma função que imprima um trecho de um texto

Imprimindo uma substring

Queremos uma função que imprima um trecho de um texto

- imprimiremos o pedaço correspondente ao padrão

Imprimindo uma substring

Queremos uma função que imprima um trecho de um texto

- imprimiremos o pedaço correspondente ao padrão

```
1 void imprime_trecho(char texto[], int ini, int tam) {
2   int j;
3   printf("%d: ", ini);
4   for (j = 0; j < tam; j++)
5     printf("%c", texto[ini + j]);
6   printf("\n");
7 }
```

Imprimindo uma substring

Queremos uma função que imprima um trecho de um texto

- imprimiremos o pedaço correspondente ao padrão

```
1 void imprime_trecho(char texto[], int ini, int tam) {
2   int j;
3   printf("%d: ", ini);
4   for (j = 0; j < tam; j++)
5     printf("%c", texto[ini + j]);
6   printf("\n");
7 }
```

Um bug:

Imprimindo uma substring

Queremos uma função que imprima um trecho de um texto

- imprimiremos o pedaço correspondente ao padrão

```
1 void imprime_trecho(char texto[], int ini, int tam) {
2   int j;
3   printf("%d: ", ini);
4   for (j = 0; j < tam; j++)
5     printf("%c", texto[ini + j]);
6   printf("\n");
7 }
```

Um bug:

- pode ser que `j` ultrapasse a última letra da string

Imprimindo uma substring

Queremos uma função que imprima um trecho de um texto

- imprimiremos o pedaço correspondente ao padrão

```
1 void imprime_trecho(char texto[], int ini, int tam) {
2   int j;
3   printf("%d: ", ini);
4   for (j = 0; j < tam; j++)
5     printf("%c", texto[ini + j]);
6   printf("\n");
7 }
```

Um bug:

- pode ser que `j` ultrapasse a última letra da string
- poderíamos parar antes se encontrarmos o `'\0'`

Imprimindo uma substring

Queremos uma função que imprima um trecho de um texto

- imprimiremos o pedaço correspondente ao padrão

```
1 void imprime_trecho(char texto[], int ini, int tam) {
2   int j;
3   printf("%d: ", ini);
4   for (j = 0; j < tam; j++)
5     printf("%c", texto[ini + j]);
6   printf("\n");
7 }
```

Um bug:

- pode ser que `j` ultrapasse a última letra da string
- poderíamos parar antes se encontrarmos o `'\0'`

Aqui imprimimos a string `char` a `char`

Imprimindo uma substring

Queremos uma função que imprima um trecho de um texto

- imprimiremos o pedaço correspondente ao padrão

```
1 void imprime_trecho(char texto[], int ini, int tam) {
2   int j;
3   printf("%d: ", ini);
4   for (j = 0; j < tam; j++)
5     printf("%c", texto[ini + j]);
6   printf("\n");
7 }
```

Um bug:

- pode ser que `j` ultrapasse a última letra da string
- poderíamos parar antes se encontrarmos o `'\0'`

Aqui imprimimos a string `char` a `char`

- mas veremos uma forma mais fácil

Verificando se o padrão está na posição

Queremos ver se `padrao` ocorre na posição `pos` do `texto`

Verificando se o padrão está na posição

Queremos ver se `padrao` ocorre na posição `pos` do `texto`

- função devolve `0` se não ocorre

Verificando se o padrão está na posição

Queremos ver se `padrao` ocorre na posição `pos` do `texto`

- função devolve `0` se não ocorre
- função devolve diferente de `0` caso contrário

Verificando se o padrão está na posição

Queremos ver se `padrao` ocorre na posição `pos` do `texto`

- função devolve `0` se não ocorre
- função devolve diferente de `0` caso contrário

```
1 int ocorre(char texto[], int pos, char padrao[]) {
2   int j;
3   for (j = 0; padrao[j] != '\0'; j++)
4     if (texto[pos + j] == '\0' ||
5         (texto[pos + j] != padrao[j] && padrao[j] != '*'))
6       return 0;
7   return 1;
8 }
```

Verificando se o padrão está na posição

Queremos ver se `padrao` ocorre na posição `pos` do `texto`

- função devolve `0` se não ocorre
- função devolve diferente de `0` caso contrário

```
1 int ocorre(char texto[], int pos, char padrao[]) {
2   int j;
3   for (j = 0; padrao[j] != '\0'; j++)
4     if (texto[pos + j] == '\0' ||
5         (texto[pos + j] != padrao[j] && padrao[j] != '*'))
6       return 0;
7   return 1;
8 }
```

Note o uso de `||` e `&&`:

Verificando se o padrão está na posição

Queremos ver se `padrao` ocorre na posição `pos` do `texto`

- função devolve `0` se não ocorre
- função devolve diferente de `0` caso contrário

```
1 int ocorre(char texto[], int pos, char padrao[]) {
2   int j;
3   for (j = 0; padrao[j] != '\0'; j++)
4     if (texto[pos + j] == '\0' ||
5         (texto[pos + j] != padrao[j] && padrao[j] != '*'))
6       return 0;
7   return 1;
8 }
```

Note o uso de `||` e `&&`:

- `&&` precede `||`

Verificando se o padrão está na posição

Queremos ver se `padrao` ocorre na posição `pos` do `texto`

- função devolve `0` se não ocorre
- função devolve diferente de `0` caso contrário

```
1 int ocorre(char texto[], int pos, char padrao[]) {
2   int j;
3   for (j = 0; padrao[j] != '\0'; j++)
4     if (texto[pos + j] == '\0' ||
5         (texto[pos + j] != padrao[j] && padrao[j] != '*'))
6       return 0;
7   return 1;
8 }
```

Note o uso de `||` e `&&`:

- `&&` precede `||`
- mas os parênteses deixam clara a ordem de precedência

Função main

```
1  int main() {
2    int i;
3    char texto[MAX], padrao[MAX];
4    scanf("%s ", padrao);
5    fgets(texto, MAX, stdin);
6    printf("Procurando por %s no texto: %s\n", padrao, texto);
7    for (i = 0; texto[i] != '\0'; i++)
8      if (ocorre(texto, i, padrao))
9        imprime_trecho(texto, i, tamanho(padrao));
10   return 0;
11 }
```

Função main

```
1  int main() {
2    int i;
3    char texto[MAX], padrao[MAX];
4    scanf("%s ", padrao);
5    fgets(texto, MAX, stdin);
6    printf("Procurando por %s no texto: %s\n", padrao, texto);
7    for (i = 0; texto[i] != '\0'; i++)
8      if (ocorre(texto, i, padrao))
9        imprime_trecho(texto, i, tamanho(padrao));
10   return 0;
11 }
```

Imprimimos strings usando %s

Função main

```
1  int main() {
2    int i;
3    char texto[MAX], padrao[MAX];
4    scanf("%s ", padrao);
5    fgets(texto, MAX, stdin);
6    printf("Procurando por %s no texto: %s\n", padrao, texto);
7    for (i = 0; texto[i] != '\0'; i++)
8      if (ocorre(texto, i, padrao))
9        imprime_trecho(texto, i, tamanho(padrao));
10   return 0;
11 }
```

Imprimimos strings usando `%s`

Lemos strings sem espaço usando `%s`:

Função main

```
1  int main() {
2    int i;
3    char texto[MAX], padrao[MAX];
4    scanf("%s ", padrao);
5    fgets(texto, MAX, stdin);
6    printf("Procurando por %s no texto: %s\n", padrao, texto);
7    for (i = 0; texto[i] != '\0'; i++)
8      if (ocorre(texto, i, padrao))
9        imprime_trecho(texto, i, tamanho(padrao));
10   return 0;
11 }
```

Imprimimos strings usando `%s`

Lemos strings sem espaço usando `%s`:

- isto é, lê até o primeiro espaço, `'\n'` ou `'\t'`

Função main

```
1  int main() {
2    int i;
3    char texto[MAX], padrao[MAX];
4    scanf("%s ", padrao);
5    fgets(texto, MAX, stdin);
6    printf("Procurando por %s no texto: %s\n", padrao, texto);
7    for (i = 0; texto[i] != '\0'; i++)
8      if (ocorre(texto, i, padrao))
9        imprime_trecho(texto, i, tamanho(padrao));
10   return 0;
11 }
```

Imprimimos strings usando `%s`

Lemos strings sem espaço usando `%s`:

- isto é, lê até o primeiro espaço, `'\n'` ou `'\t'`
- não colocamos o `&` antes do nome da variável

Função main

```
1  int main() {
2    int i;
3    char texto[MAX], padrao[MAX];
4    scanf("%s ", padrao);
5    fgets(texto, MAX, stdin);
6    printf("Procurando por %s no texto: %s\n", padrao, texto);
7    for (i = 0; texto[i] != '\0'; i++)
8      if (ocorre(texto, i, padrao))
9        imprime_trecho(texto, i, tamanho(padrao));
10   return 0;
11 }
```

Função main

```
1  int main() {
2    int i;
3    char texto[MAX], padrao[MAX];
4    scanf("%s ", padrao);
5    fgets(texto, MAX, stdin);
6    printf("Procurando por %s no texto: %s\n", padrao, texto);
7    for (i = 0; texto[i] != '\0'; i++)
8      if (ocorre(texto, i, padrao))
9        imprime_trecho(texto, i, tamanho(padrao));
10   return 0;
11 }
```

Lemos strings com espaços usando a função `fgets`:

Função main

```
1  int main() {
2    int i;
3    char texto[MAX], padrao[MAX];
4    scanf("%s ", padrao);
5    fgets(texto, MAX, stdin);
6    printf("Procurando por %s no texto: %s\n", padrao, texto);
7    for (i = 0; texto[i] != '\0'; i++)
8      if (ocorre(texto, i, padrao))
9        imprime_trecho(texto, i, tamanho(padrao));
10   return 0;
11 }
```

Lemos strings com espaços usando a função `fgets`:

- primeiro parâmetro: nome da variável

Função main

```
1  int main() {
2    int i;
3    char texto[MAX], padrao[MAX];
4    scanf("%s ", padrao);
5    fgets(texto, MAX, stdin);
6    printf("Procurando por %s no texto: %s\n", padrao, texto);
7    for (i = 0; texto[i] != '\0'; i++)
8      if (ocorre(texto, i, padrao))
9        imprime_trecho(texto, i, tamanho(padrao));
10   return 0;
11 }
```

Lemos strings com espaços usando a função `fgets`:

- primeiro parâmetro: nome da variável
- segundo parâmetro: tamanho máximo da string

Função main

```
1  int main() {
2    int i;
3    char texto[MAX], padrao[MAX];
4    scanf("%s ", padrao);
5    fgets(texto, MAX, stdin);
6    printf("Procurando por %s no texto: %s\n", padrao, texto);
7    for (i = 0; texto[i] != '\0'; i++)
8      if (ocorre(texto, i, padrao))
9        imprime_trecho(texto, i, tamanho(padrao));
10   return 0;
11 }
```

Lemos strings com espaços usando a função `fgets`:

- primeiro parâmetro: nome da variável
- segundo parâmetro: tamanho máximo da string
  - contando o `'\0'`

Função main

```
1  int main() {
2    int i;
3    char texto[MAX], padrao[MAX];
4    scanf("%s ", padrao);
5    fgets(texto, MAX, stdin);
6    printf("Procurando por %s no texto: %s\n", padrao, texto);
7    for (i = 0; texto[i] != '\0'; i++)
8      if (ocorre(texto, i, padrao))
9        imprime_trecho(texto, i, tamanho(padrao));
10   return 0;
11 }
```

Lemos strings com espaços usando a função `fgets`:

- primeiro parâmetro: nome da variável
- segundo parâmetro: tamanho máximo da string
  - contando o `'\0'`
- terceiro parâmetro: de qual arquivo devemos ler

Função main

```
1  int main() {
2    int i;
3    char texto[MAX], padrao[MAX];
4    scanf("%s ", padrao);
5    fgets(texto, MAX, stdin);
6    printf("Procurando por %s no texto: %s\n", padrao, texto);
7    for (i = 0; texto[i] != '\0'; i++)
8      if (ocorre(texto, i, padrao))
9        imprime_trecho(texto, i, tamanho(padrao));
10   return 0;
11 }
```

Lemos strings com espaços usando a função `fgets`:

- primeiro parâmetro: nome da variável
- segundo parâmetro: tamanho máximo da string
  - contando o `'\0'`
- terceiro parâmetro: de qual arquivo devemos ler
  - estamos lendo da entrada padrão, por isso passamos `stdin`

Função main

```
1  int main() {
2    int i;
3    char texto[MAX], padrao[MAX];
4    scanf("%s ", padrao);
5    fgets(texto, MAX, stdin);
6    printf("Procurando por %s no texto: %s\n", padrao, texto);
7    for (i = 0; texto[i] != '\0'; i++)
8      if (ocorre(texto, i, padrao))
9        imprime_trecho(texto, i, tamanho(padrao));
10   return 0;
11 }
```

Lemos strings com espaços usando a função `fgets`:

- primeiro parâmetro: nome da variável
- segundo parâmetro: tamanho máximo da string
  - contando o `'\0'`
- terceiro parâmetro: de qual arquivo devemos ler
  - estamos lendo da entrada padrão, por isso passamos `stdin`

O fgets lê apenas até o primeiro `'\n'`

Função main

```
1  int main() {
2    int i;
3    char texto[MAX], padrao[MAX];
4    scanf("%s ", padrao);
5    fgets(texto, MAX, stdin);
6    printf("Procurando por %s no texto: %s\n", padrao, texto);
7    for (i = 0; texto[i] != '\0'; i++)
8      if (ocorre(texto, i, padrao))
9        imprime_trecho(texto, i, tamanho(padrao));
10   return 0;
11 }
```

Lemos strings com espaços usando a função `fgets`:

- primeiro parâmetro: nome da variável
- segundo parâmetro: tamanho máximo da string
  - contando o `'\0'`
- terceiro parâmetro: de qual arquivo devemos ler
  - estamos lendo da entrada padrão, por isso passamos `stdin`

O fgets lê apenas até o primeiro `'\n'`

- e pode incluir o `'\n'` na string

Função main

```
1  int main() {
2    int i;
3    char texto[MAX], padrao[MAX];
4    scanf("%s ", padrao);
5    fgets(texto, MAX, stdin);
6    printf("Procurando por %s no texto: %s\n", padrao, texto);
7    for (i = 0; texto[i] != '\0'; i++)
8      if (ocorre(texto, i, padrao))
9        imprime_trecho(texto, i, tamanho(padrao));
10   return 0;
11 }
```

Função main

```
1  int main() {
2    int i;
3    char texto[MAX], padrao[MAX];
4    scanf("%s ", padrao);
5    fgets(texto, MAX, stdin);
6    printf("Procurando por %s no texto: %s\n", padrao, texto);
7    for (i = 0; texto[i] != '\0'; i++)
8      if (ocorre(texto, i, padrao))
9        imprime_trecho(texto, i, tamanho(padrao));
10   return 0;
11 }
```

Por que colocamos o espaço após o `%s` na linha 4?

Função main

```
1  int main() {
2    int i;
3    char texto[MAX], padrao[MAX];
4    scanf("%s ", padrao);
5    fgets(texto, MAX, stdin);
6    printf("Procurando por %s no texto: %s\n", padrao, texto);
7    for (i = 0; texto[i] != '\0'; i++)
8      if (ocorre(texto, i, padrao))
9        imprime_trecho(texto, i, tamanho(padrao));
10   return 0;
11 }
```

Por que colocamos o espaço após o `%s` na linha 4?

- para consumir os espaços em branco depois da string...

Função main

```
1  int main() {
2    int i;
3    char texto[MAX], padrao[MAX];
4    scanf("%s ", padrao);
5    fgets(texto, MAX, stdin);
6    printf("Procurando por %s no texto: %s\n", padrao, texto);
7    for (i = 0; texto[i] != '\0'; i++)
8      if (ocorre(texto, i, padrao))
9        imprime_trecho(texto, i, tamanho(padrao));
10   return 0;
11 }
```

Por que colocamos o espaço após o `%s` na linha 4?

- para consumir os espaços em branco depois da string...

Caso contrário, o `fgets` poderia ler apenas o `\n` após o padrão

A biblioteca `string.h`

A biblioteca `string.h` tem várias funções úteis:

A biblioteca `string.h`

A biblioteca `string.h` tem várias funções úteis:

`strlen` devolve o tamanho da string

A biblioteca `string.h`

A biblioteca `string.h` tem várias funções úteis:

- `strlen` devolve o tamanho da string
- `strcmp` compara duas strings já que não podemos usar `<`, `<=`, `>`, `>=`, `==` e `!=`

A biblioteca `string.h`

A biblioteca `string.h` tem várias funções úteis:

- `strlen` devolve o tamanho da string
- `strcmp` compara duas strings já que não podemos usar `<`, `<=`, `>`, `>=`, `==` e `!=`
- `strcpy` copia uma string

A biblioteca `string.h`

A biblioteca `string.h` tem várias funções úteis:

- `strlen` devolve o tamanho da string
- `strcmp` compara duas strings já que não podemos usar `<`, `<=`, `>`, `>=`, `==` e `!=`
- `strcpy` copia uma string
- `strcat` concatena duas strings

A biblioteca `string.h`

A biblioteca `string.h` tem várias funções úteis:

- `strlen` devolve o tamanho da string
- `strcmp` compara duas strings já que não podemos usar `<`, `<=`, `>`, `>=`, `==` e `!=`
- `strcpy` copia uma string
- `strcat` concatena duas strings

entre outras...

A biblioteca `string.h`

A biblioteca `string.h` tem várias funções úteis:

- `strlen` devolve o tamanho da string
- `strcmp` compara duas strings já que não podemos usar `<`, `<=`, `>`, `>=`, `==` e `!=`
- `strcpy` copia uma string
- `strcat` concatena duas strings

entre outras...

Veja o manual para a documentação

A biblioteca `string.h`

A biblioteca `string.h` tem várias funções úteis:

- `strlen` devolve o tamanho da string
- `strcmp` compara duas strings já que não podemos usar `<`, `<=`, `>`, `>=`, `==` e `!=`
- `strcpy` copia uma string
- `strcat` concatena duas strings

entre outras...

Veja o manual para a documentação

- exemplo: `man strlen`

A biblioteca `string.h`

A biblioteca `string.h` tem várias funções úteis:

- `strlen` devolve o tamanho da string
- `strcmp` compara duas strings já que não podemos usar `<`, `<=`, `>`, `>=`, `==` e `!=`
- `strcpy` copia uma string
- `strcat` concatena duas strings

entre outras...

Veja o manual para a documentação

- exemplo: `man strlen`

Não confunda com a biblioteca `strings.h`

Tipos mais comuns do C

dado	tipo	formato	ex. de constante
inteiros	`int`	`%d`	`10`

ponto flutuante	`float`	`%f`
		`%g`
		`%e`
			`10.0f`
			`2e-3f`

ponto flutuante (precisão dupla)	`double`	`%lf`
		`%lg`
		`%le`
			`10.0`
			`2e-3`

caractere	`char`	`%c`	`'c'`
string	`char []`	`%s`	`"string"`

Lembrando que `%s` lê strings sem espaço

Outros tipos

Temos variações de tamanho para `int`:

Outros tipos

Temos variações de tamanho para `int`:

- `short` ou `short int` — `%hi`

Outros tipos

Temos variações de tamanho para `int`:

- `short` ou `short int` — `%hi`
  - pelo menos 16 bits

Outros tipos

Temos variações de tamanho para `int`:

- `short` ou `short int` — `%hi`
  - pelo menos 16 bits
- `int` — `%d` ou `%i`

Outros tipos

Temos variações de tamanho para `int`:

- `short` ou `short int` — `%hi`
  - pelo menos 16 bits
- `int` — `%d` ou `%i`
  - pelo menos 16 bits

Outros tipos

Temos variações de tamanho para `int`:

- `short` ou `short int` — `%hi`
  - pelo menos 16 bits
- `int` — `%d` ou `%i`
  - pelo menos 16 bits
- `long` ou `long int` — `%li`

Outros tipos

Temos variações de tamanho para `int`:

- `short` ou `short int` — `%hi`
  - pelo menos 16 bits
- `int` — `%d` ou `%i`
  - pelo menos 16 bits
- `long` ou `long int` — `%li`
  - pelo menos 32 bits

Outros tipos

Temos variações de tamanho para `int`:

- `short` ou `short int` — `%hi`
  - pelo menos 16 bits
- `int` — `%d` ou `%i`
  - pelo menos 16 bits
- `long` ou `long int` — `%li`
  - pelo menos 32 bits
- `long long` ou `long long int` — `%lli`

Outros tipos

Temos variações de tamanho para `int`:

- `short` ou `short int` — `%hi`
  - pelo menos 16 bits
- `int` — `%d` ou `%i`
  - pelo menos 16 bits
- `long` ou `long int` — `%li`
  - pelo menos 32 bits
- `long long` ou `long long int` — `%lli`
  - pelo menos 64 bits

Outros tipos

Temos variações de tamanho para `int`:

- `short` ou `short int` — `%hi`
  - pelo menos 16 bits
- `int` — `%d` ou `%i`
  - pelo menos 16 bits
- `long` ou `long int` — `%li`
  - pelo menos 32 bits
- `long long` ou `long long int` — `%lli`
  - pelo menos 64 bits

A quantidade de bits pode variar de acordo com a plataforma

Outros tipos

Temos variações de tamanho para `int`:

- `short` ou `short int` — `%hi`
  - pelo menos 16 bits
- `int` — `%d` ou `%i`
  - pelo menos 16 bits
- `long` ou `long int` — `%li`
  - pelo menos 32 bits
- `long long` ou `long long int` — `%lli`
  - pelo menos 64 bits

A quantidade de bits pode variar de acordo com a plataforma

- por exemplo, `int` em geral tem 32 bits

Outros tipos

Temos variações de tamanho para `int`:

- `short` ou `short int` — `%hi`
  - pelo menos 16 bits
- `int` — `%d` ou `%i`
  - pelo menos 16 bits
- `long` ou `long int` — `%li`
  - pelo menos 32 bits
- `long long` ou `long long int` — `%lli`
  - pelo menos 64 bits

A quantidade de bits pode variar de acordo com a plataforma

- por exemplo, `int` em geral tem 32 bits
- mas a especificação diz pelo menos 16 bits

Outros tipos

Temos variações de tamanho para `int`:

- `short` ou `short int` — `%hi`
  - pelo menos 16 bits
- `int` — `%d` ou `%i`
  - pelo menos 16 bits
- `long` ou `long int` — `%li`
  - pelo menos 32 bits
- `long long` ou `long long int` — `%lli`
  - pelo menos 64 bits

A quantidade de bits pode variar de acordo com a plataforma

- por exemplo, `int` em geral tem 32 bits
- mas a especificação diz pelo menos 16 bits

A vantagem é poder escolher entre economizar memória ou representar mais números

Outros tipos

Temos também as versões sem sinal (`unsigned`):

Outros tipos

Temos também as versões sem sinal (`unsigned`):

- `unsigned char` — (`%c`)

Outros tipos

Temos também as versões sem sinal (`unsigned`):

- `unsigned char` — (`%c`)
- `unsigned short` ou `unsigned short int` — (`%hu`)

Outros tipos

Temos também as versões sem sinal (`unsigned`):

- `unsigned char` — (`%c`)
- `unsigned short` ou `unsigned short int` — (`%hu`)
- `unsigned` ou `unsigned int` — (`%u`)

Outros tipos

Temos também as versões sem sinal (`unsigned`):

- `unsigned char` — (`%c`)
- `unsigned short` ou `unsigned short int` — (`%hu`)
- `unsigned` ou `unsigned int` — (`%u`)
- `unsigned long` ou `unsigned long int` — (`%lu`)

Outros tipos

Temos também as versões sem sinal (`unsigned`):

- `unsigned char` — (`%c`)
- `unsigned short` ou `unsigned short int` — (`%hu`)
- `unsigned` ou `unsigned int` — (`%u`)
- `unsigned long` ou `unsigned long int` — (`%lu`)
- `unsigned long long` ou `unsigned long long int` — (`%llu`)

Outros tipos

Temos também as versões sem sinal (`unsigned`):

- `unsigned char` — (`%c`)
- `unsigned short` ou `unsigned short int` — (`%hu`)
- `unsigned` ou `unsigned int` — (`%u`)
- `unsigned long` ou `unsigned long int` — (`%lu`)
- `unsigned long long` ou `unsigned long long int` — (`%llu`)

A vantagem do `unsigned`:

Outros tipos

Temos também as versões sem sinal (`unsigned`):

- `unsigned char` — (`%c`)
- `unsigned short` ou `unsigned short int` — (`%hu`)
- `unsigned` ou `unsigned int` — (`%u`)
- `unsigned long` ou `unsigned long int` — (`%lu`)
- `unsigned long long` ou `unsigned long long int` — (`%llu`)

A vantagem do `unsigned`:

- se você for trabalhar apenas com números não-negativos, você consegue representar mais números...

Outros tipos

Temos também as versões sem sinal (`unsigned`):

- `unsigned char` — (`%c`)
- `unsigned short` ou `unsigned short int` — (`%hu`)
- `unsigned` ou `unsigned int` — (`%u`)
- `unsigned long` ou `unsigned long int` — (`%lu`)
- `unsigned long long` ou `unsigned long long int` — (`%llu`)

A vantagem do `unsigned`:

- se você for trabalhar apenas com números não-negativos, você consegue representar mais números...

Em geral, trabalhamos apenas com os tipos básicos

Outros tipos

Temos também as versões sem sinal (`unsigned`):

- `unsigned char` — (`%c`)
- `unsigned short` ou `unsigned short int` — (`%hu`)
- `unsigned` ou `unsigned int` — (`%u`)
- `unsigned long` ou `unsigned long int` — (`%lu`)
- `unsigned long long` ou `unsigned long long int` — (`%llu`)

A vantagem do `unsigned`:

- se você for trabalhar apenas com números não-negativos, você consegue representar mais números...

Em geral, trabalhamos apenas com os tipos básicos

- `int`, `double` e `char`

Exercício

Faça uma função `void copia(char str1[], char str2[])` que copia o conteúdo de `str1` para `str2`.

Solução

```
1 void copia(char str1[], char str2[]) {
2     int i;
3     for(i = 0; str1[i] != '\0'; i++)
4         str2[i] = str1[i];
5     str2[i] = '\0';
6 }
```

Exercício

Faça uma função `void reverte(char str[])` que reverte o conteúdo de `str`.

Exemplo: Se a string era `"MC202"`, a string deve passar a ser `"202CM"`.

Solução

```
1  int tamanho(char str[]) {
2      int i;
3      for(i = 0; str[i] != '\0'; i++);
4      return i;
5  }
6
7  void reverte(char str[]) {
8      int tam = tamanho(str);
9      for (int i = 0; i < tam / 2; i++) {
10         char temp = str[i];
11         str[i] = str[tam - i - 1];
12         str[tam - i - 1] = temp;
13     }
14 }
15
16 // Outra versão, só para mostrar um for mais complicado
17 void reverte_v2(char str[]) {
18     int tam = tamanho(str);
19     for (int i = 0, j = tam - 1; i < j; i++, j--) {
20         char temp = str[i];
21         str[i] = str[j];
22         str[j] = temp;
23     }
24 }
```

Exercício

Faça uma função `int compara(char str1[], char str2[])` que

- devolve `0` se as strings são iguais
- devolve um número menor do que zero se `str1` é lexicograficamente menor do que `str2`
- devolve um número maior do que zero caso contrário

Solução

```
1 int compara(char str1[], char str2[]) {
2     int i;
3     for(i = 0; str1[i] == str2[i]; i++)
4         if (str1[i] == '\0')
5             return 0; // strings são iguais
6     return str1[i] - str2[i]; // comparação lexicográfica
7 }
```

Dúvidas?