Caesar (César)

Implemente um programa que codifica (criptografa) uma mensagem usando o algoritmo de César.

```
$ ./caesar 13
plaintext:  HELLO
ciphertext: URYYB
```

Supostamente, César (sim, o imperador César) usou para "criptografar" (isto é, esconder uma mensagem de modo a poder recuperá-la posteriormente) mensagens deslocando cada letra por um valor fixo. Por exemplo, escrevendo B em vez de A, C no lugar de B, D em vez de C e assim por diante até que A seja usado no lugar de Z. Assim, a palavra `HELLO` poderia ser escrita como `IFMMP`. Depois de receber uma mensagem "criptografada", bastava deslocar as letras na direção oposta para obter a mensagem original.

O segredo deste "sistema de criptografia" é que apenas César e seu correspondente conheciam o segredo, isto é, eles estavam deslocando cada letra. Este é um sistema que, claramente, não é seguro para os dias atuais, mas se você foi o primeiro a inventar tal sistema ele seria seguro.

Textos não criptografados são geralmente chamados de **texto claro** ou **texto plano** (do inglês, *plaintext*) enquanto textos criptografados são chamados de **texto cifrado** (do inglês, *cyphertext*) e o segredo usado para a codificação é chamado de **chave** (do inglês, *key*).

Vamos deixar mais claro por meio de um exemplo. Ao codificar a palavra `HELLO` com chave `1`, obtemos o texto cifrado `IFMMP`.

texto plano	**H**	**E**	**L**	**L**	**O**
+chave	1	1	1	1	1
=texto cifrado	I	F	M	M	P

Mais formalmente, o algoritmo (cifra) de César codifica uma mensagem por rotacionar cada letra do alfabeto em `k` posições. Ou seja, se `p` é um texto plano (não criptografado), p_i é a i-ésima letra do texto e `k` é a chave. Então cada letra c_i do texto cifrado `c` é dada por

$$c_i = (p_i + k) % 26$$

onde `%` representa o "resto da divisão inteira por 26". Esta fórmula pode parecer mais complicada que que realmente é, mas ela simplesmente reduz a discussão acima a respeito dos deslocamentos das letras quando mapeamos A (ou a) para 0, B (ou b) para 1, ... e Z (ou z) para 25.

Seu objetivo é escrever um código em C chamado `caesar.c`. Seu programa deverá permitir ao usuário usar argumento de linha de comando para definir a chave. Você não precisa supor que o usuário vai digitar um número, mas se ele digitar você pode supor que o número é um inteiro positivo.

Seguem alguns exemplos de como o programa funciona.

Se você usar a chave `1` com a palavra `HELLO`.

```
$ ./caesar 1
plaintext:  HELLO
ciphertext: IFMMP
```

Se você usar a chave `13` com a expressão `hello, world`

```
$ ./caesar 13
plaintext:  hello, world
ciphertext: uryyb, jbeyq
```

Note que nem a vírgula nem o espaço foram "deslocados". Somente caracteres alfabéticos foram alterados.

Mais um exemplo? Aqui é como o programa deveria se comportar com a chave `13` em um texto mais complexo.

```
$ ./caesar 13
plaintext:  be sure to drink your Ovaltine
ciphertext: or fher gb qevax lbhe Binygvar
```

Note que as letras maiúsculas e minúsculas do texto plano permanecem maiúsculas e minúsculas, respectivamente, no texto cifrado.

O que ocorre se o usuário não cooperar?

```
$ ./caesar HELLO
Usage: ./caesar key
```

ou

```
$ ./caesar
Usage: ./caesar key
```

ou ainda

```
$ ./caesar 1 2 3
Usage: ./caesar key
```

Especificação

Projete e implemente um programa, `caesar`, que codifica uma mensagem usando a cifra de César.

- Implemente seu programa em um arquivo chamado `caesar.c` no diretório ~/pset2/caesar.
- Seu programa deverá aceitar um único parâmetro de linha de comando que deve ser um inteiro não-negativo que chamaremos de `k`.
- Se seu programa for executado sem qualquer parâmetro de linha de comando ou com mais de um parâmetro, ele deverá exibir uma mensagem de erro e a função `main` deveria retornar valor `1` (para indicar um erro) imediatamente.
- Se qualquer caractere do argumento da linha de comando não for um dígito decimal, seu programa deveria exibir a mensagem `Usage: ./caesar key` e a função `main` deve retornar `1`.
- Não suponha que `k` é menor ou igual a 26. Seu programa deveria funcionar para todo número não-negativo `k` menor que $2^{31}-26$. Em outras palavras, você não precisa se preocupar do seu programa "quebrar" caso o usuário escolha um número muito grande. Mas se o usuário digitar um número maior que 26, por exemplo, 27, o que deveria ocorrer? Bem, a letra `A` deveria se tornar `B` desde que `B` está 27 posições deslocadas a partir de `A`.
- Seu programa precisa exibir `plaintext:` (sem caractere de nova linha) e então solicitar uma `string` ao usuário (use a função `get_string`).
- Seu programa precisa exibir o `ciphertext`: (sem caractere de nova linha) seguido do texto cifrado correspondente ao texto plano. Cada caractere alfabético deve ser "rotacionado" por `k` e os demais caracteres devem permanecer inalterados.
- Seu programa precisa preservar letras maiúsculas e minúsculas.
- Após a exibição do texto cifrado, seu programa deveria exibir uma nova linha e a função `main` deve retornar valor `0`.

Por onde começar? Vamos a uma abordar um passo por vez.

Pseudocódigo

Você pode querer escrever primeiramente algum pseudocódigo para lhe auxiliar na elaboração do programa. Não existe "um jeito certo" de executar esta tarefa, apenas escreva algumas instruções em português informal que lhe ajutem a pensar na sequência que levará ao resultado.

Exemplo

1. Verificar se o programa roda com exatamente um parâmetro de linha de comando.
2. Iterar em todos os caracteres do argumento passado para verificar que são dígitos numéricos.
3. Converter o parâmetro de linha de comando de `string` para `int`.
4. Solicitar que o usuário digite o `paintext`.
5. Iterar sobre cada um dos caracteres do `plaintext`:
    1. Se é uma letra maiúscula, rotacionar e exibir a nova letra maiúscula correspondente.
    2. Se é uma letra minúscula, rotacionar e exibir a nova letra minúscula correspondente.
    3. Caso contrário, exibir o caractere como ele foi digitado pelo usuário.
6. Exibir uma nova linha

Provavelmente você vai querer revisar o próprio pseudocódigo

Contando argumento de linha de comandos

Independente do seu pseudocódigo, você vai precisar escrever seu código real em C que verifica quantos argumentos de linha de comando foram passados.

Especificamente, escreva um arquivo chamado `caesar.c` contendo seu código em que: se o usuário informar exatamente um parâmetro, exibe a mensagem `success`. Se o usuário fornecer zero argumento ou mais de um argumento, seu programa deveria exibir `Usage: ./caesar key`. Lembre, desde que a `chave` está sendo passada via linha de comando e não via `get_string`, você não tem como solicitar novamente ao usuário sem que ele execute o programa novamente. Veja alguns exemplos abaixo:

```
$ ./caesar 20
Success
```

ou

```
$ ./caesar
Usage: ./caesar key
```

ou

```
$ ./caesar 1 2 3
Usage: ./caesar key
```

Dica

- Lembre de compilar seu código usando `make`.
- Você pode usar a função `printf`.
- `argc` e `argv` lhe dá informações sobre argumentos de linha de comando.
- Lembre que o nome do programa em si (neste caso, `./caesar`) está em `argv[0]`.

Acessando a chave

Agora seu programa (eu espero que sim) está aceitando entradas via linha de comando. Nós precisamos garantir que os dados fornecidos pelo usuário são válidos. Por exmeplo, a entrada abaixo é inválida

```
$ ./caesar xyz
```

Antes de iniciar a analizar a `chave`, você precisa "ler" a chave. Então, modifique seu programa para, além de exibir a mensagem `Success`, exibir também o valor da chave. Seu programa deveria se comportar como abaixo:

```
$ ./caesar 20
Success
20
```

Dica

- Lembre-se que `argc` e `argv` lhe dão informações sobre os parâmetros de linha de comando.
- `argv` é um *array* (vetor) de *strings*.
- Você pode usar a função `printf` com a opção `%s` como **lugar reservado** para *strings*.
- Lembre que cientistas da computação iniciam contagem em 0 (zero).
- Você pode acessar cada elemento individual de um *array* usando o nome da variável, por exemplo, `argv` e o índice entre colchetes como em `argv[0]`.

Validando a chave

Agora que você está lendo a chave, vamos avaliá-la. Modifique seu `caesar.c` de modo que em vez de exibir o argumento fornecido pela linha de comando, ele verifique se cada caractere do parâmetro é um dígito numérico. Se todos forem dígitos, converta o parâmetro para inteiro (`int`) e use a função `printf` com a opção `%i%` como lugar reservado. Seu programa deveria se comportar como abaixo

```
$ ./caesar 20
Success
20
```

O que parece não diferir do exemplo anterior. Mas se pelo menos um dos caracteres do parâmetro não for dígito numérico, seu programa deveria se comportar coom abaixo:

```
$ ./caesar 20x
Usage: ./caesar key
```

Dica

- Lembre que `argv` é um vetor de *strings*.
- Lembre que uma *string* é um vetor de caracteres.
- Lembre que a biblioteca `string.h` contém funções úteis para lidar com *strings*.
- Você pode usar um laço para verificar cada caractere do parâmetro.
- A biblioteca `ctype.h` contém várias funções úteis para trabalhar com caracteres.
- Use `return` com um valor diferente de zero para indicar que seu programa terminou com erro.
- Use `printf` com `%i` como lugar reservado para inteiros.
- A função `atoi` converte *string* para inteiro (curiosidade: `atof` converte *string* para `float`).

Como testar seu código

Execute o comando abaixo para verificar a **corretude** do seu código. Mas tente compilar e testar antes de executar o comando

```
check50 cs50/problems/2020/x/caesar
```

Execute o comando abaixo para garantir a **estilização** do código

```
style50 caesar.c
```

Enviando seu código

Execute o comando abaixo, logando com seu **nome de usuário** do GitHub, para enviar seu código. Por questões de segurança, asteríscos serão exibidos em vez dos caracteres da sua senha

```
submit50 cs50/problems/2020/x/caesar
```