Aula 01 Exercício 01

Mundo 1-1

Perto do final do Mundo 1-1 no Super Mario Brothers da Nintendo, Mario deve ascender a pirâmide de blocos alinhada à direita, como demonstrado abaixo.

Vamos recriar essa pirâmide em C, ainda que em texto, usando hashes (**#**) para tijolos, como visto a seguir. Cada hash é um pouco mais alto do que largo, então a pirâmide em si também é mais alta do que larga.

```
       #
      ##
     ###
    ####
   #####
  ######
 #######
########
```

Este vídeo irá te ajudar a entender o problema ;)

Atenção: *para adicionar legendas ao vídeo clique no botão CC localizado no Player e selecione a opção "Português (Brasil)". Uma excelente aula para você!*

O programa que escreveremos se chamará **mario**. E vamos permitir que o usuário decida qual deve ser a altura da pirâmide, primeiro solicitando um número inteiro positivo entre, digamos, 1 e 8, inclusive.

Veja como o programa pode funcionar se o usuário inserir **8** quando solicitado:

```
$ ./mario
Tamanho: 8
        #
       ##
      ###
     ####
    #####
   ######
 #######
########
```

Veja como o programa pode funcionar se o usuário inserir **4** quando solicitado:

```
$ ./mario
Tamanho: 4
   #
  ##
 ###
####
```

Veja como o programa pode funcionar se o usuário inserir **2** quando solicitado:

```
$ ./mario
Tamanho: 2
 #
##
```

Veja como o programa pode funcionar se o usuário inserir **1** quando solicitado:

```
$ ./mario
Tamanho: 1
#
```

Se o usuário não inserir, de fato, um número inteiro positivo entre 1 e 8, inclusive, quando solicitado, o programa deve solicitar novamente ao usuário até que ele coopere:

```
$ ./mario
Tamanho: -1
Tamanho: 0
Tamanho: 42
Tamanho: 9
```

```
Tamanho: 4
   #
  ##
 ###
####
```

Quero resolver este exercício agora, clique aqui para ir para o IDE.

Pseudocódigo

Primeiro, crie um novo diretório (ou seja, pasta) chamado mario dentro do seu diretório pset1, executando

```
~/$ mkdir ~/pset1/mario
```

Adicione um novo arquivo chamado **pseudocodigo.txt** dentro do seu diretório mario.

Escreva em **pseudocodigo.txt** algum pseudocódigo que implemente este programa, mesmo que não tenha (ainda!) certeza de como escrevê-lo em código. Não existe uma maneira certa de escrever pseudocódigo, mas frases curtas são suficientes. É provável que seu pseudocódigo use (ou implique o uso!) de uma ou mais funções, condições, expressões booleanas, loops e/ou variáveis.

Clique aqui para ver o Spoiler ;)
Existe mais de uma forma para resolver esse exercício, esse spoiler aqui é apenas uma delas!
1- Peça ao usuário o tamanho da altura.
2- Se o tamanho da altura for menor que 1 ou maior que 8(ou não inteiro), fique nesse passo até que o usuário insira uma entrada válida.
3- Itere a variável i até o tamanho da altura.
    4- Imprima os #. [Não se esqueça da quebra de linha!]

Como testar seu código no IDE do CS50?

Execute o seguinte para avaliar se seu código está correto usando **check50**. Mas certifique-se de compilar e testar você mesmo!

```
check50 cs50/problems/2021/x/mario/less
```
Execute o seguinte para avaliar o style do seu código usando **style50**.

```
style50 mario.c
```

Exercício 2: Mario (desafio)

Mundo 1-1

No início de World 1-1 em Super Mario Brothers, da Nintendo, Mario deve pular pirâmides de blocos adjacentes, conforme mostrado abaixo.

Vamos recriar essas pirâmides em C, ainda que em texto, usando hashes (**#**) para tijolos, a la a seguir. Cada hash é um pouco mais alto do que largo, então as pirâmides em si também são mais altas do que largas.

```
  # #
 ## ##
### ###
```

```
#### ####
```

Este vídeo irá te ajudar a entender o problema ;)

Atenção: *para adicionar legendas ao vídeo clique no botão CC localizado no Player e selecione a opção "Português (Brasil)". Uma excelente aula para você!*

O programa que escreveremos se chamará **mario**. E vamos permitir que o usuário decida a altura das pirâmides, primeiro solicitando um número inteiro positivo entre, digamos, 1 e 8, inclusive.

Veja como o programa pode funcionar se o usuário inserir **8** quando solicitado:

```
$ ./mario

Altura: 8

         # #

        ## ##

       ### ###

      #### ####

     ##### #####
```

```
   ###### ######

 ####### #######

######## ########
```

Veja como o programa pode funcionar se o usuário inserir **4** quando solicitado:

```
$ ./mario

Altura: 4

     # #

    ## ##

   ### ###

#### ####
```

Veja como o programa pode funcionar se o usuário inserir **2** quando solicitado:

```
$ ./mario

Altura: 8

  # #

## ##
```

Veja como o programa pode funcionar se o usuário inserir **1** quando solicitado:

```
$ ./mario

Altura: 8

# #
```

Se o usuário não inserir, de fato, um número inteiro positivo entre 1 e 8, inclusive, quando solicitado, o programa deve solicitar novamente ao usuário até que ele escreva o valor correto:

```
$ ./mario

Altura: -1

Altura: 0

Altura: 32

Altura: 10

Altura: 4

   # #

  ## ##

 ### ###
```

```
#### ####
```

Observe que a largura da “lacuna” entre as pirâmides adjacentes é igual à largura de dois hashes, independentemente da altura das pirâmides. Crie um novo diretório (ou seja, pasta) chamado **mario** dentro do seu diretório **pset1** , executando:

```
~/$ mkdir ~/pset1/mario
```

Crie um novo arquivo chamado mario.c dentro do seu diretório **mario**. Modifique mario.c de forma que implemente este programa conforme descrito!

Como testar seu código no IDE do CS50?

Seu código funciona conforme prescrito quando você insere:

- -1 (ou outros números negativos)?
- 0 ?
- 1 a 8 ?
- 9 ou outros números positivos?
- letras ou palavras?
- nenhuma entrada, quando você apenas pressiona Enter?

Execute o seguinte para avaliar se seu código está correto usando **check50**. Mas certifique-se de compilar e testar você mesmo!

```
check50 cs50/problems/2021/x/mario/more
```

Execute o seguinte para avaliar o style do seu código usando **style50**.

```
style50 mario.c
```

Exercício 3: Dinheiro (versão fácil)

Algoritmos Gulosos ou Algoritmos Ambiciosos

Ao dar o troco, é provável que você queira minimizar o número de moedas que está distribuindo para cada cliente, para não acabar com o estoque (ou irritar o cliente!). Felizmente, a ciência da computação deu aos caixas em todos os lugares maneiras de minimizar o número de moedas devidas: algoritmos ambiciosos, também conhecidos como gulosos ou gananciosos.

De acordo com o Instituto Nacional de Padrões e Tecnologia (NIST), um algoritmo ambicioso é aquele “que sempre pega a melhor solução imediata, ou local, enquanto encontra uma resposta. Algoritmos ambiciosos encontram a solução geral ou globalmente ideal para alguns problemas de otimização, mas podem encontrar soluções menos do que ideais para algumas instâncias de outros problemas.”

O que tudo isso significa? Bem, suponha que um caixa deva a um cliente algum troco e na gaveta desse caixa estejam moedas de 25, 10, 5 e 1 centavo(s). O problema a ser resolvido é decidir quais moedas e quantas de cada uma entregar ao cliente. Pense em um caixa “ganancioso” como alguém que quer tirar o maior proveito possível desse problema com cada moeda que tira da gaveta. Por exemplo, se algum cliente deve pagar 41 centavos, a maior “mordida”(ou seja, melhor “mordida” imediata ou local) que pode ser feita é 25 centavos. (Essa mordida é "melhor" na medida em que nos deixa mais perto de 0 ¢ mais rápido do que qualquer outra moeda faria.) Observe que uma mordida desse tamanho reduziria o que era um problema de 41 ¢ a um problema de 16 ¢, já que 41 - 25 = 16. Ou seja, o restante é um problema semelhante, mas menor. Desnecessário dizer que outra mordida de 25 centavos seria muito grande (supondo que o caixa prefere não perder dinheiro), e assim nosso caixa ganancioso mudaria para uma mordida de 10 centavos, deixando-o com um problema de 6 centavos. Nesse ponto, a ganância pede uma mordida de 5 centavos seguida de uma mordida de 1 centavo, ponto em que o problema é resolvido. O cliente recebe um quarto, um centavo, um centavo e um centavo: quatro moedas no total. Acontece que essa abordagem gananciosa (do algoritmo) não é apenas ótima localmente, mas também globalmente para a moeda dos Estados Unidos (e também da União Europeia). Ou seja, desde que o caixa tenha o suficiente de cada moeda, essa abordagem do maior para o menor renderá o menor número possível de moedas. Quão menor? Bem, diga-nos você!

Detalhes de Implementação

Este vídeo irá te ajudar a entender o problema ;)

Atenção: *para adicionar legendas ao vídeo clique no botão CC localizado no Player e selecione a opção "Português (Brasil)".*

Implemente, em um arquivo chamado **cash.c** em um diretório **~/pset1/cash**, um programa que primeiro pergunta ao usuário quanto dinheiro é devido e depois imprime o número mínimo de moedas com as quais essa mudança pode ser feita.

Use **get_float** para obter a entrada do usuário e **printf** para gerar sua resposta. Suponha que as únicas moedas disponíveis sejam de 25, 10, 5 e 1 centavo(s).

- Pedimos que você use **get_float** para que possa lidar com reais e centavos, embora sem o cifrão. Em outras palavras, se algum cliente deve R$9.75 (como no caso em que um jornal custa 25 centavos, mas o cliente paga com uma nota de R$10), suponha que a entrada de seu programa será de **9.75** e não de **R$9.75** ou **975** . No entanto, se algum cliente deve exatamente R$9, suponha que a entrada de seu programa será **9.00** ou apenas **9**, mas, novamente, não **R$9** ou **900** . É claro que, pela natureza dos valores de ponto flutuante, seu programa provavelmente funcionará com entradas como 9.0 e 9.000 também; você não precisa se preocupar em verificar se a entrada do usuário está "formatada" como o dinheiro deveria estar.

Você não precisa tentar verificar se a entrada de um usuário é muito grande para caber em um **float**. Usar **get_float** sozinho garantirá que a entrada do usuário seja realmente um valor de ponto flutuante (ou integral), mas não que seja não negativo.

Se o usuário não fornecer um valor não negativo, seu programa deve solicitar novamente ao usuário uma quantia válida até que o usuário concorde.

Para que possamos automatizar alguns testes do seu código, certifique-se de que a última linha de outpt do seu programa seja apenas o número mínimo de moedas possível: um inteiro seguido por **\n**.

Cuidado com a imprecisão inerente aos valores de ponto flutuante. Lembre do **floats.c** da aula, em que, se **x** é 2 , e **y** é 10 , **x / y** não é precisamente dois décimos! E assim, antes de fazer a alteração, você provavelmente desejará converter os dólares inseridos pelo usuário em centavos (ou seja, de um **float** para um **int**) para evitar pequenos erros que poderiam se acumular!

Tome cuidado para arredondar seus centavos até o último centavo mais próximo, por exemplo usando o **round**, que é declarado na **math.h**. Por exemplo, se o real é um **float** com input do usuário (por exemplo, **0.20**), então uma linha como:

```
int centavos = round(reais * 100);
```

irá converter com segurança **0.20** (ou mesmo 0.200000298023223876953125) em 20.

Utilize o ponto final ao invés de vírgula!!

Seu programa deve se comportar de acordo com os exemplos abaixo.

```
$ ./cash

Troca devida: 0.41

4
```

```
 $ ./cash

Troca devida: -0.41
```

```
Troca devida: foo

Troca devida: 0.41

4
```

Como testar seu código no IDE do CS50?

Seu código funciona conforme prescrito quando você insere:

- **-1.00** (ou outros números negativos)?
- **0.00** ?
- **0.01** (ou outros números positivos)?
- letras ou palavras?
- nenhuma entrada, quando você apenas pressiona Enter?

Execute o seguinte para avaliar se seu código está correto usando **check50**. Mas certifique-se de compilar e testar você mesmo!

```
check50 cs50/problems/2021/x/cash
```

Execute o seguinte para avaliar o style do seu código usando **style50**.

```
style50 cash.c
```

Exercício 4: Crédito (desafio))

Um cartão de crédito (ou débito), é claro, é um cartão de plástico com o qual você pode pagar por bens e serviços. Impresso nesse cartão está um número que também é armazenado em um banco de dados em algum lugar, de modo que, quando o cartão for usado para comprar algo, o credor saiba a quem cobrar. Há muitas pessoas com cartões de crédito no mundo, então esses números são bem longos: American Express usa números de 15 dígitos, MasterCard usa números de 16 dígitos e Visa usa números de 13 e 16 dígitos. E esses são números decimais (0 a 9), não binários, o que significa, por exemplo, que a American Express poderia imprimir até 10 ^ 15 = 1.000.000.000.000.000 de cartões exclusivos! (Isso é, hum, um quatrilhão.)

Na verdade, isso é um pouco exagerado, porque os números de cartão de crédito têm alguma estrutura. Todos os números American Express começam com 34 ou 37; a maioria dos números do MasterCard começa com 51, 52, 53, 54 ou 55 (eles também têm alguns outros números iniciais potenciais com os quais não nos preocupamos neste problema); e todos os números Visa começam com 4. Mas os números de cartão de crédito também têm um “checksum” embutido, uma relação matemática entre pelo menos um número e outros. Essa soma de verificação permite que os computadores (ou humanos que gostam de matemática) detectem erros de digitação (por exemplo, transposições), se não números fraudulentos, sem ter que consultar um banco de dados, que pode ser lento. É claro que um matemático desonesto certamente poderia criar um número falso que, no entanto, respeite a restrição matemática, portanto, uma pesquisa no banco de dados ainda é necessária para verificações mais rigorosas.

Algoritmo de Luhn

Então, qual é a fórmula secreta? Bem, a maioria dos cartões usa um algoritmo inventado por Hans Peter Luhn, da IBM. De acordo com o algoritmo de Luhn, você pode determinar se um número de cartão de crédito é (sintaticamente) válido da seguinte maneira:

1. Multiplique cada segundo digito por 2, começando com o penúltimo dígito do número e, em seguida, some os dígitos desses produtos.
2. Adicione essa soma à soma dos dígitos que não foram multiplicados por 2.
3. Se o último dígito do total for 0 (ou, mais formalmente, se o módulo total 10 for congruente com 0), o número é válido!

Isso é meio confuso, então vamos tentar um exemplo com o cartão Visa do David: 4003600000000014.

1- Para fins de discussão, vamos primeiro sublinhar todos os outros dígitos, começando com o penúltimo dígito do número:

4003600000000014

Ok, vamos multiplicar cada um dos dígitos sublinhados por 2:

1 • 2 + 0 • 2 + 0 • 2 + 0 • 2 + 0 • 2 + 6 • 2 + 0 • 2 + 4 • 2

Isso nos dá:

2 + 0 + 0 + 0 + 0 + 12 + 0 + 8

Agora vamos adicionar os dígitos desses produtos (ou seja, não os próprios produtos):

2 + 0 + 0 + 0 + 0 + 1 + 2 + 0 + 8 = 13

2- Agora vamos adicionar essa soma (13) à soma dos dígitos que não foram multiplicados por 2 (começando do final):

13 + 4 + 0 + 0 + 0 + 0 + 0 + 3 + 0 = 20

3- Sim, o último dígito dessa soma (20) é 0, então o cartão de David é legítimo!

Portanto, validar números de cartão de crédito não é difícil, mas se torna um pouco tedioso manualmente. Vamos escrever um programa.

Detalhes de Implementação

Este vídeo irá te ajudar a entender o problema ;)

Atenção: *para adicionar legendas ao vídeo clique no botão CC localizado no Player e selecione a opção "Português (Brasil)".*

Em um arquivo chamado **credit.c** em um diretório **~/pset1/credit/**, escreva um programa que solicite ao usuário um número de cartão de crédito e, em seguida, informe (via **printf**) se é um número de cartão American Express, MasterCard ou Visa válido , de acordo com as definições de formato de cada um neste documento. Para que possamos automatizar alguns testes do seu código, pedimos que a última linha de saída do seu programa seja **AMEX\n** ou **MASTERCARD\n** ou **VISA\n** ou **INVALID\n** , nada mais, nada menos. Para simplificar, você pode assumir que o input do usuário será inteiramente numérica (ou seja, sem hífens, como pode ser impresso em um cartão real). Mas não presuma que o input do usuário caberá em um **int** ! Melhor usar **get_long** da biblioteca do CS50 para obter o input dos usuários. (Por que?)

Considere o seguinte exemplo de como seu próprio programa deve se comportar quando um número de cartão de crédito válido é fornecido (sem hífens).

```
$ ./credit

Número: 4003600000000014

VISA
```

Agora, **get_long** em si rejeitará hífens (e mais) de qualquer maneira:

```
$ ./credit

Número: 4003-6000-0000-0014

Número: foo

Número: 4003600000000014

VISA
```

Mas depende de você pegar entradas que não sejam números de cartão de crédito (por exemplo, um número de telefone), mesmo que sejam numéricos:

```
$ ./credit

Número: 6176292929

INVALID
```

Teste seu programa com um monte de entradas, válidas e inválidas. (Certamente o faremos!) Aqui estão alguns números de cartão que o PayPal recomenda para teste.

Se o seu programa se comporta incorretamente com alguns inputs(ou não compila), é hora de depurar!

Como testar seu código no IDE do CS50?

Execute o seguinte para avaliar se seu código está correto usando **check50**. Mas certifique-se de compilar e testar você mesmo!

```
check50 cs50/problems/2021/x/credit
```

Execute o seguinte para avaliar o style do seu código usando **style50**.

```
style50 credit.c
```

Lab 1: População

Laboratório 1: crescimento populacional

Determine quanto tempo leva para uma população atingir um determinado tamanho.

```
$ ./population

Start size: 100

End size: 200

Years: 9
```

Background

Digamos que temos uma população de n lhamas. A cada ano, nascem n / 3 novas lhamas e n / 4 morrem.

Por exemplo, se começarmos com **n = 1.200** lhamas, no primeiro ano, **1.200 / 3 = 400** novas lhamas nascerão e **1.200 / 4 = 300** lhamas morrerão. No final daquele ano, teríamos **1.200 + 400 - 300 = 1.300** lhamas.

Para tentar outro exemplo, se começarmos com **n = 1000** lhamas, no final do ano teremos **1000/3 = 333,33** novas lhamas. Não podemos ter uma parte decimal de uma lhama, entretanto, vamos truncar o decimal para que **333** novas lhamas nasçam. **1000/4 = 250** lhamas passarão, então terminaremos com um total de **1000 + 333 - 250 = 1083** lhamas no final do ano.

Começando

Copie o “código de distribuição” (ou seja, código inicial) a seguir em um novo arquivo em seu IDE chamado population.c .

```
#include

#include



int main(void)

{

    // TODO: Solicite o valor inicial ao usuário



    // TODO: Solicite o valor final ao usuário



    // TODO: Calcule o número de anos até o limite



    // TODO: Imprima o número de anos



}
```

Detalhes de Implementação

Conclua a implementação de population.c , de forma que calcule o número de anos necessários para que a população cresça do tamanho inicial ao tamanho final.

Seu programa deve primeiro solicitar ao usuário um tamanho inicial da população.

Se o usuário inserir um número menor que 9 (o tamanho mínimo permitido da população), o usuário deve ser solicitado novamente a inserir um tamanho inicial da população até inserir um número maior ou igual a 9. (Se começarmos com menos de 9 lhamas, a população de lhamas ficará estagnada rapidamente!)

Seu programa deve então solicitar ao usuário o tamanho final da população.

Se o usuário inserir um número menor que o tamanho da população inicial, ele deverá ser solicitado novamente a inserir um tamanho da população final até inserir um número que seja maior ou igual ao tamanho da população inicial. (Afinal, queremos que a população de lhamas cresça!)

Seu programa deve então calcular o número (inteiro) de anos necessários para que a população atinja pelo menos o tamanho do valor final.

Finalmente, seu programa deve imprimir o número de anos necessários para que a população de lhama alcance esse tamanho final, como ao imprimir no terminal **Years: n** , onde **n** é o número de anos.

Dicas

Se você deseja solicitar repetidamente ao usuário o valor de uma variável até que alguma condição seja atendida, você pode usar um loop do ... while. Por exemplo, recupere o seguinte código da palestra, que avisa o usuário repetidamente até que ele insira um número inteiro positivo.

```
int n;

do

{

    n = get_int("Inteiro positivo: ");

}

while (n < 1);
```

Como você pode adaptar este código para garantir um tamanho inicial de pelo menos 9, alem de um tamanho final que seja pelo menos o tamanho inicial?

Para declarar uma nova variável, certifique-se de especificar seu tipo de dado, um nome para a variável e (opcionalmente) qual deve ser seu valor inicial.

Por exemplo, você pode querer criar uma variável para controlar quantos anos se passaram.

Para calcular quantos anos a população levará para atingir o tamanho final, outro ciclo pode ser útil! Dentro do loop, você provavelmente desejará atualizar o tamanho da população de acordo com a fórmula em Background e atualizar o número de anos que se passaram.

Para imprimir um inteiro n no terminal, lembre-se de que você pode usar uma linha de código como

```
printf("O número é %i\n", n);
```

para especificar que a variável **n** deve ser preenchida para o espaço reservado **%i** .

Como testar seu código

Seu código deve resolver os seguintes casos de teste:

```
$ ./population
Start size: 1200
End size: 1300
```

- Years: 1

```
$ ./population
Start size: -5
Start size: 3
Start size: 9
End size: 5
End size: 18
```

- Years: 8

```
$ ./population
Start size: 20
End size: 1
End size: 10
```

```
End size: 100
```

- Years: 20

```
$ ./population
Start size: 100
End size: 1000000
```

- Years: 115

Execute o seguinte comando para avaliar a exatidão do seu código usando **check50** . Mas certifique-se de compilar e testar você mesmo!

```
check50 cs50/labs/2021/x/population
```

Execute o seguinte comando para avaliar o estilo do seu código usando style50 .

```
style50 population.c
```