Este é o CS50

Introdução à Ciência da Computação (CS50)

OpenCourseWare

Doar (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan)
(https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

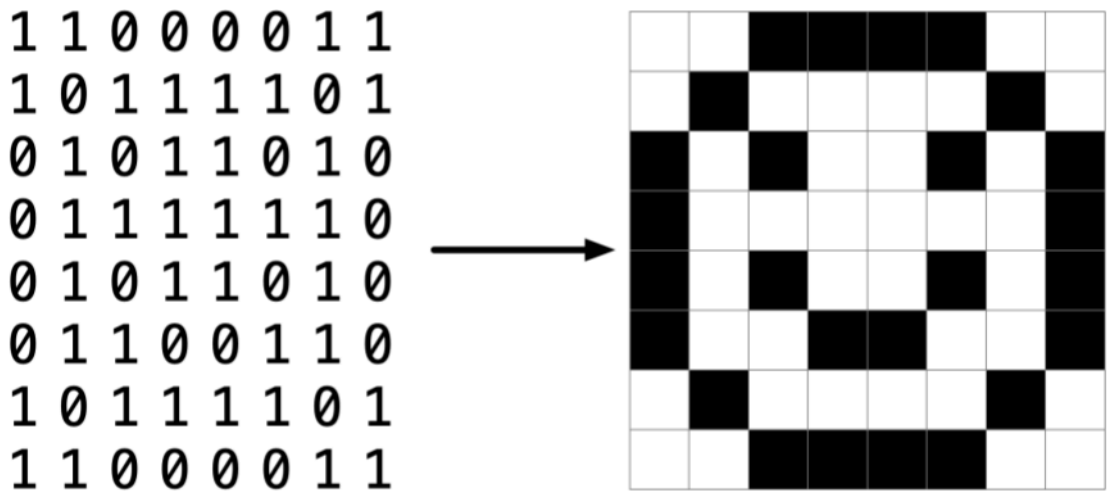
Aula 4

Bem-vindo!

- Nas semanas anteriores, falamos sobre como as imagens são compostas de blocos de construção menores chamados pixels.
- Hoje, vamos nos aprofundar nos zeros e uns que compõem essas imagens. Em particular, vamos explorar os blocos de construção fundamentais que formam os arquivos, incluindo as imagens.
- Além disso, discutiremos como acessar os dados subjacentes armazenados na memória do computador.
- Ao iniciarmos a aula de hoje, tenhamos em mente que os conceitos abordados podem levar algum tempo para serem totalmente *compreendidos* .

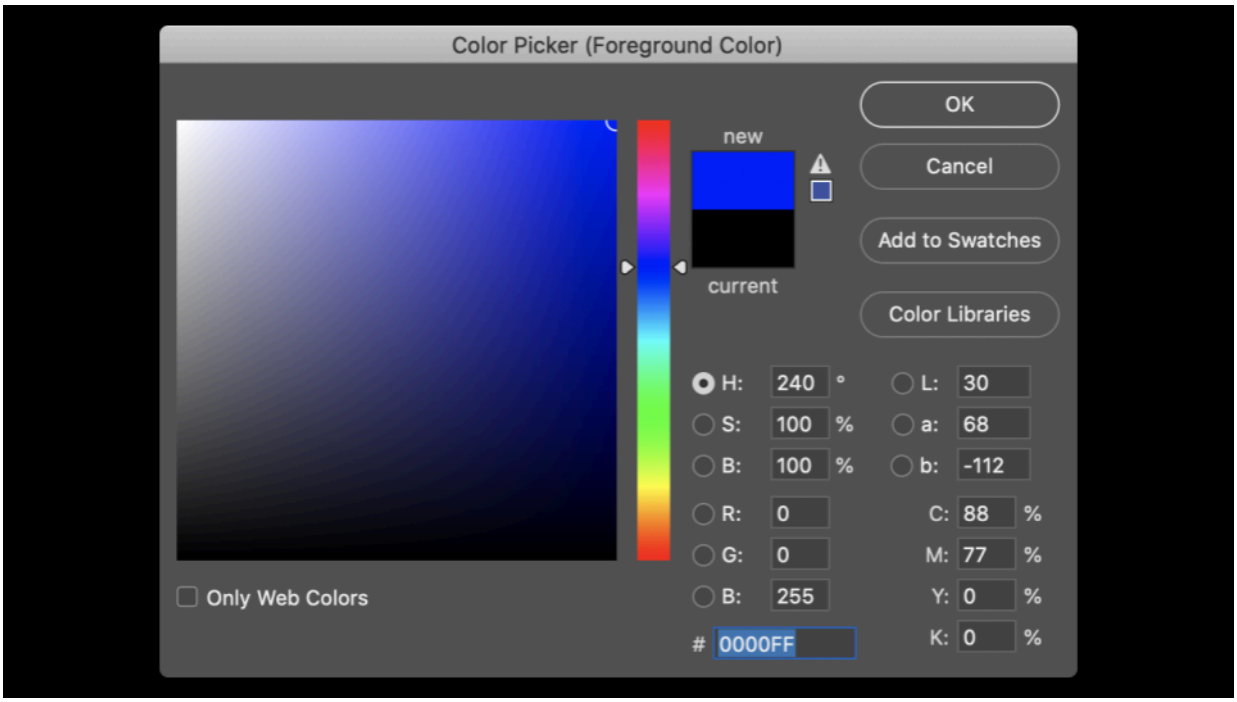
Arte em pixel

- Pixels são quadrados, pontos individuais, de cor que são dispostos em uma grade vertical (de cima para baixo) e horizontal (da esquerda para a direita).
- Você pode imaginar uma imagem como um mapa de bits, onde zeros representam preto e uns representam branco.



Hexadecimal

- *RGB* , ou *vermelho, verde e azul* , são números que representam a quantidade de cada uma dessas cores. No Adobe Photoshop, você pode visualizar essas configurações da seguinte forma:



Observe como a quantidade de vermelho, azul e verde altera a cor selecionada.

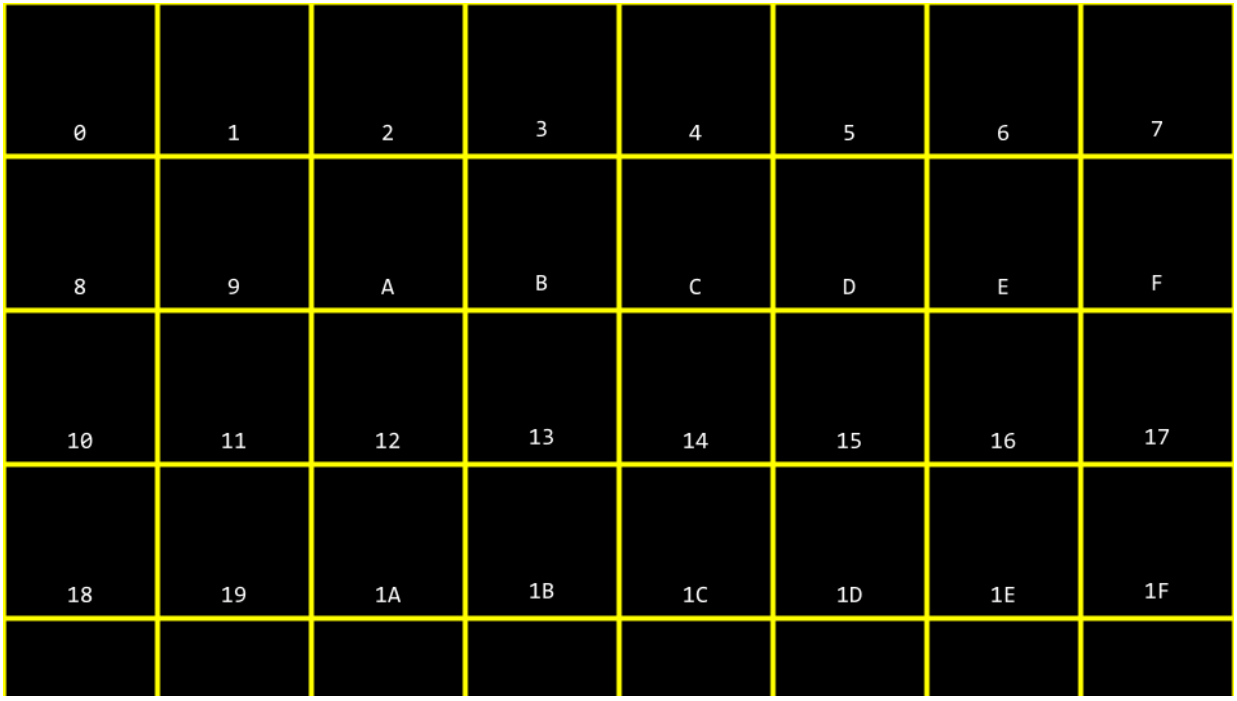
- Como você pode ver na imagem acima, a cor não é representada apenas por três valores. Na parte inferior da janela, há um valor especial composto por números e caracteres, `255` representado por `FF`. Por que isso acontece?
- *O sistema hexadecimal* é um sistema de contagem que possui 16 valores numéricos. São eles:

  ```
  0 1 2 3 4 5 6 7 8 9 A B C D E F
  ```

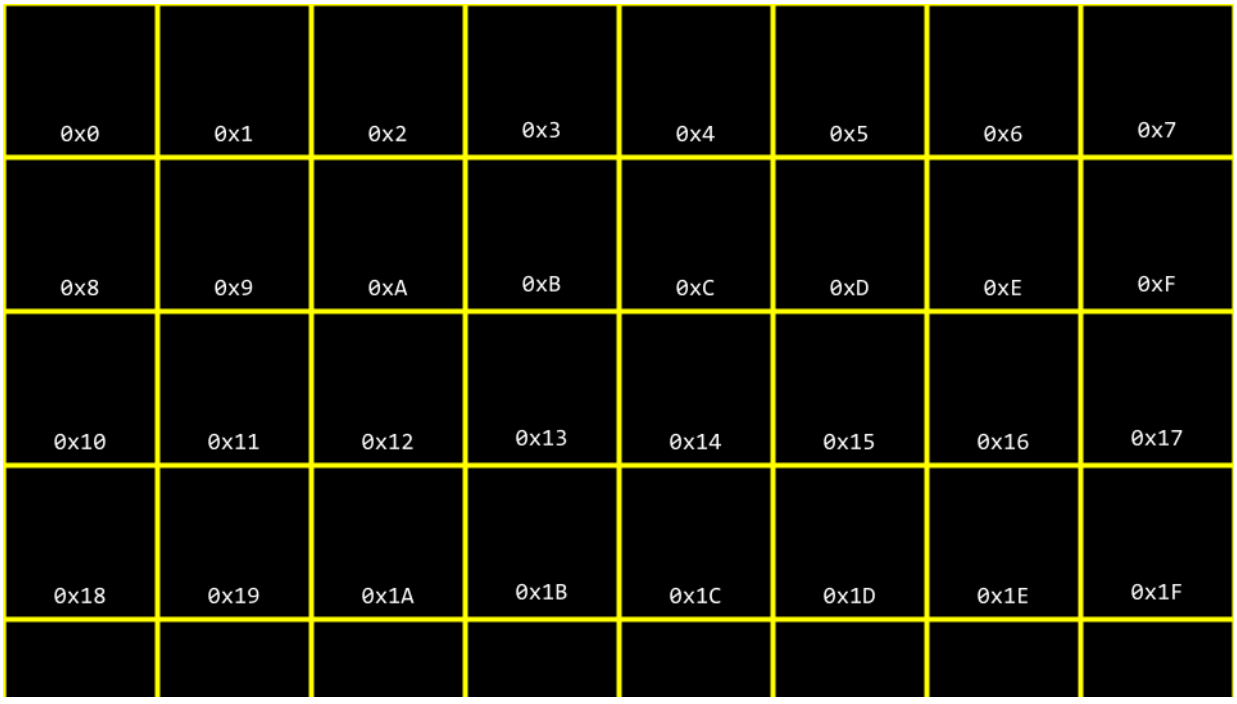
  Observe que `F` representa `15`.
- O sistema hexadecimal também é conhecido como *base 16* .
- Ao contar em hexadecimal, cada coluna representa uma potência de 16.
- O número `0` é representado como `00`.
- O número `1` é representado como `01`.
- O número `9` é representado por `09`.
- O número `10` é representado como `0A`.
- O número `15` é representado como `0F`.
- O número `16` é representado como `10`.
- O número `255` é representado como `FF`, porque 16 x 15 (ou `F`) é igual a 240. Adicionando mais 15, obtemos 255. Este é o maior número que você pode contar usando um sistema hexadecimal de dois dígitos.
- O sistema hexadecimal é útil porque permite representar informações com menos dígitos. O hexadecimal nos permite representar informações de forma mais concisa.

Memória

- Nas últimas semanas, você deve se lembrar da nossa representação artística de blocos de memória simultâneos. Aplicando a numeração hexadecimal a cada um desses blocos de memória, você pode visualizá-los da seguinte forma:



- É fácil imaginar a confusão que pode surgir sobre se o `10` bloco acima representa um endereço na memória ou o valor `10`. Por convenção, todos os números hexadecimais são frequentemente representados com o `0x` prefixo da seguinte forma:



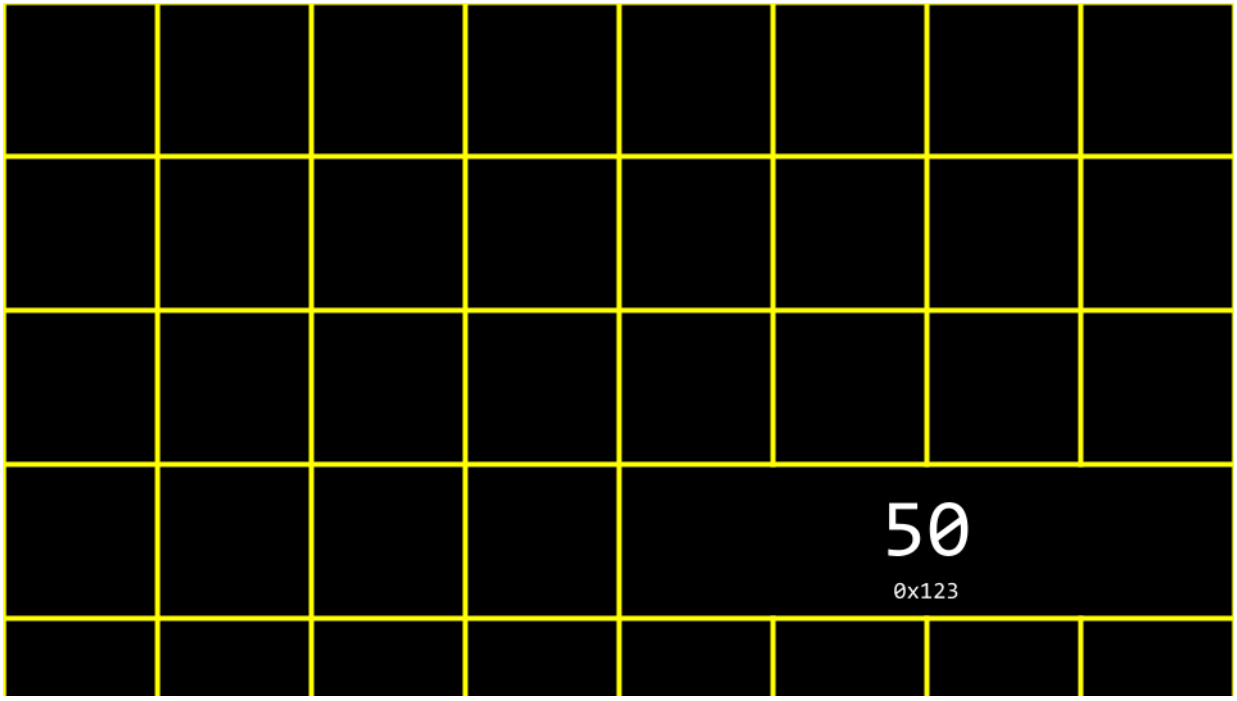
- Na janela do terminal, digite `code addresses.c` e escreva o código da seguinte forma:

```c
// Prints an integer

#include <stdio.h>

int main(void)
{
    int n = 50;
```

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

Observe como `n` é armazenado na memória com o valor `50`.

- Você pode visualizar como este programa armazena esse valor da seguinte forma:



Dicas

- A linguagem C possui dois operadores poderosos relacionados à memória:

```
  & Provides the address of something stored in memory.
  * Instructs the compiler to go to a location in memory.
```

- Podemos aproveitar esse conhecimento modificando nosso código da seguinte forma:

```c
// Prints an integer's address

#include <stdio.h>

int main(void)
{
    int n = 50;
    printf("%p\n", &n);
}
```

Observe o `%p`, que nos permite visualizar o endereço de um local na memória. `&n` pode ser traduzido literalmente como "o endereço de `n`". Executar este código retornará um endereço de memória começando com `0x`.

- Um *ponteiro* é uma variável que armazena o endereço de algo. De forma mais sucinta, um ponteiro é um endereço na memória do seu computador.
- Considere o seguinte código:

```
int n = 50;
int *p = &n;
```

Observe que `p` é um ponteiro que contém o endereço de um número inteiro `n`.

- Modifique seu código da seguinte forma:

```
// Stores and prints an integer's address

#include <stdio.h>

int main(void)
{
    int n = 50;
    int *p = &n;
    printf("%p\n", p);
}
```

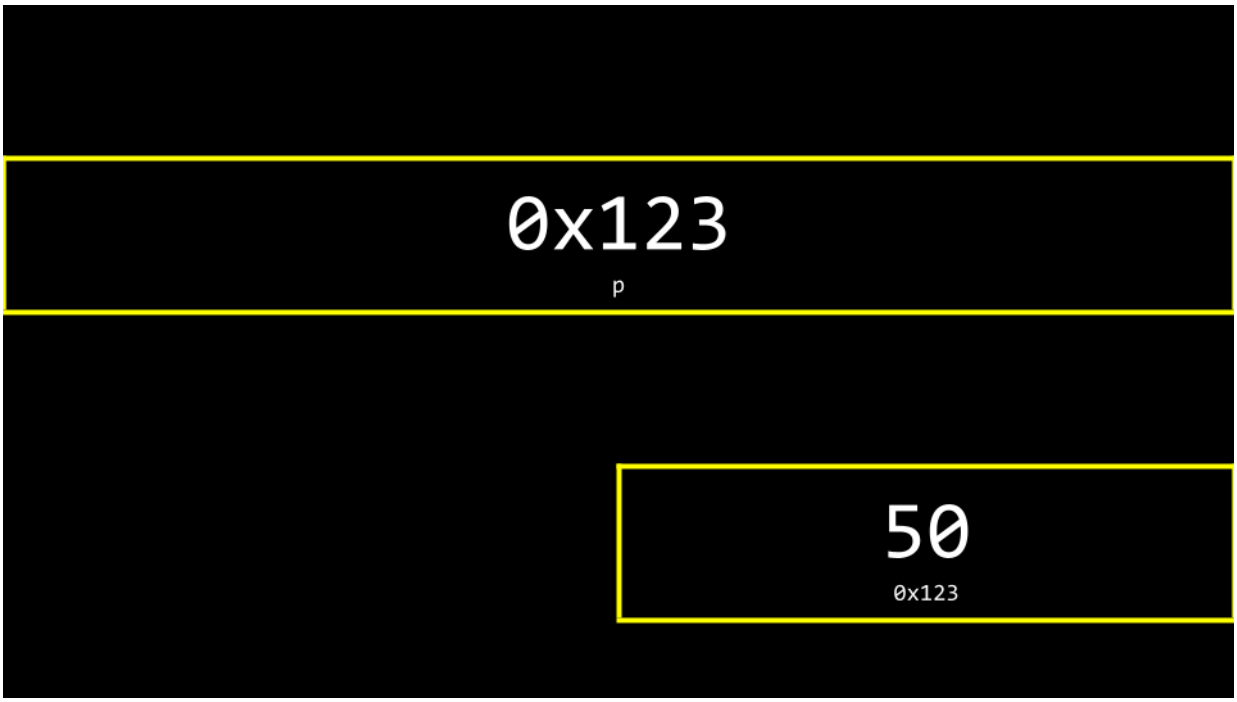
Observe que este código tem o mesmo efeito que o nosso código anterior. Simplesmente aproveitamos o nosso novo conhecimento dos operadores ` `&` and` `*`.

- Para ilustrar o uso do `*` operador, considere o seguinte:

```
// Stores and prints an integer via its address

#include <stdio.h>

int main(void)
{
    int n = 50;
    int *p = &n;
    printf("%i\n", *p);
}
```

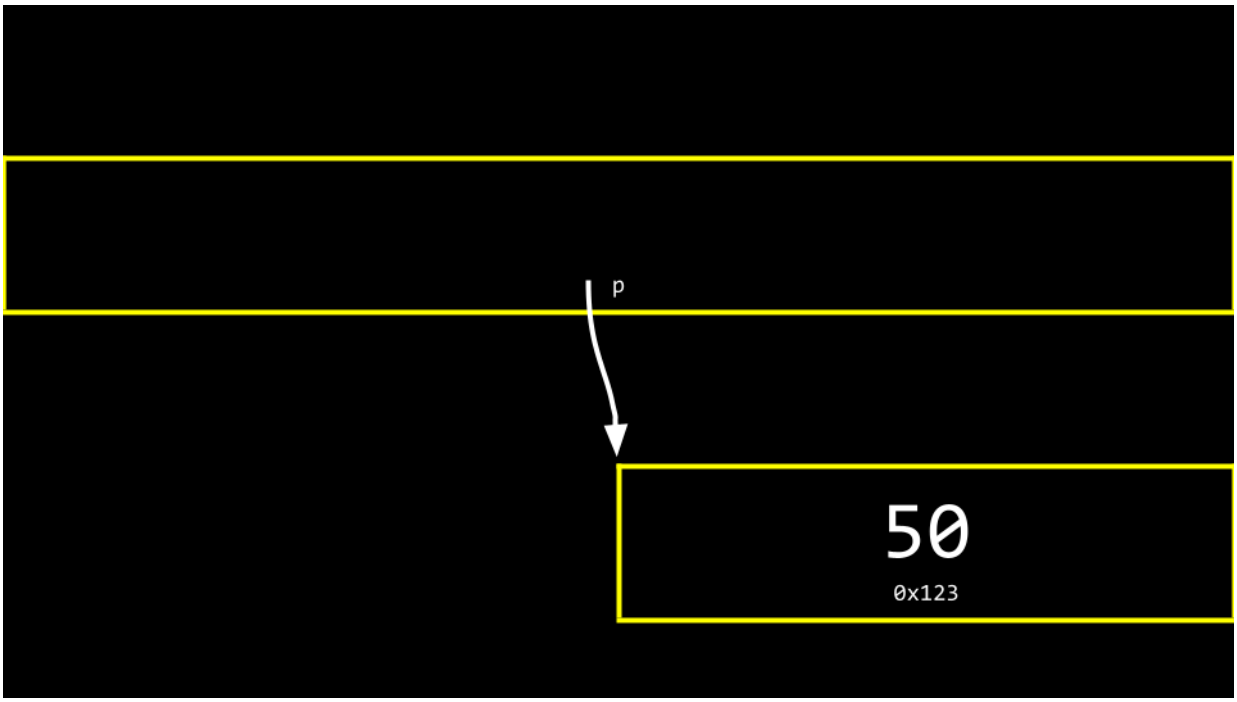
Observe que a `printf` linha imprime o número inteiro na posição de `p`. `int *p` cria um ponteiro cuja função é armazenar o endereço de memória de um número inteiro.

- Você pode visualizar nosso código da seguinte forma:



Note que o ponteiro parece bastante grande. De fato, um ponteiro geralmente é armazenado como um valor de 8 bytes. `p` Ele está armazenando o endereço do `50`.

- Você pode visualizar um ponteiro com mais precisão como um endereço que aponta para outro:



Cordas

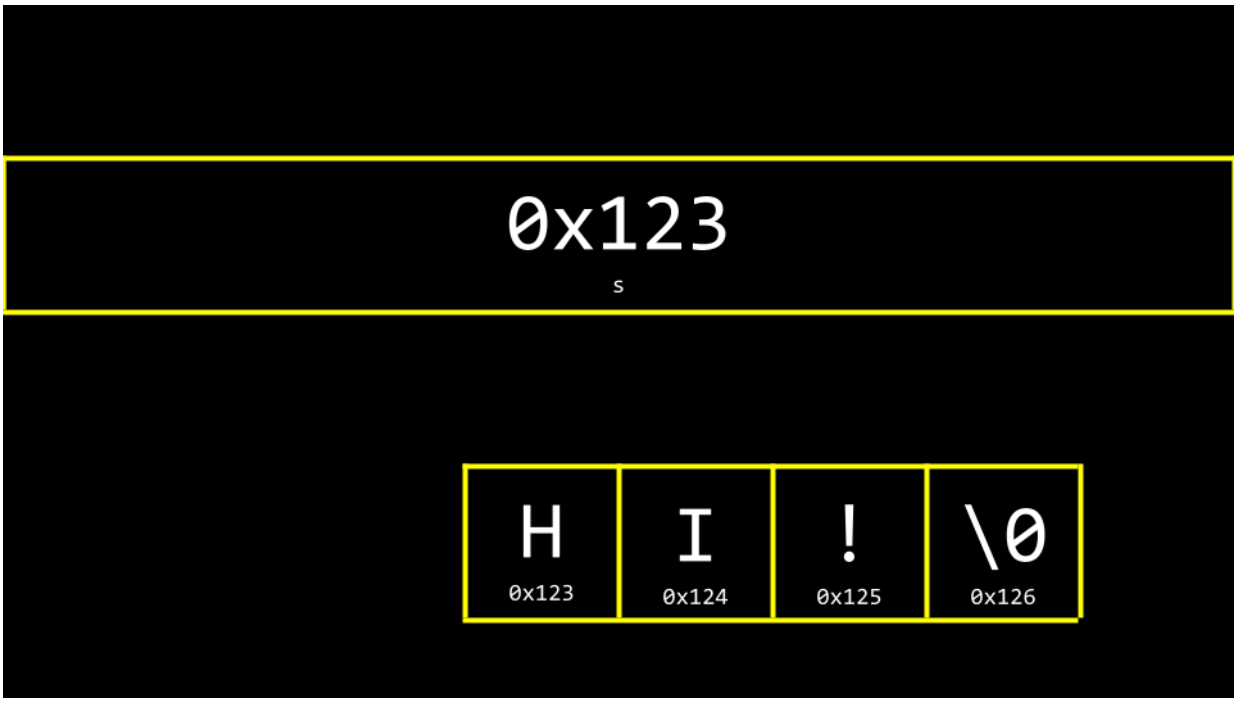
- Agora que temos um modelo mental para ponteiros, podemos simplificar um pouco o que foi apresentado anteriormente neste curso.
- Modifique seu código da seguinte forma:

```
// Prints a string
```

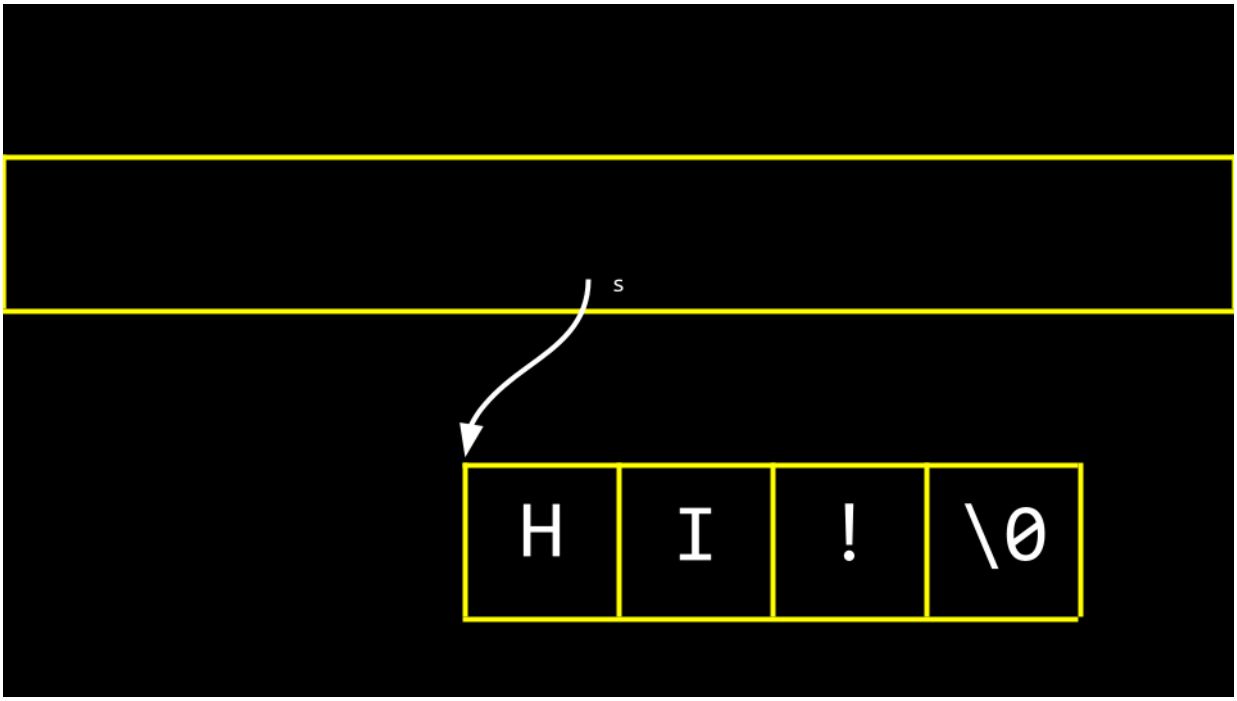
```c
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string s = "HI!";
    printf("%s\n", s);
}
```

Observe que uma sequência de caracteres `s` foi impressa.

- Lembre-se de que uma string é simplesmente uma matriz de caracteres. Por exemplo, `string s = "HI!"` pode ser representada da seguinte forma:



- Mas, afinal, o que é `s` isso? Onde é `s` armazenado na memória? Como você pode imaginar, `s` precisa ser armazenado em algum lugar. Você pode visualizar a relação de `s` com a string da seguinte forma:



Observe como um ponteiro chamado `is` `s` informa ao compilador onde o primeiro byte da string existe na memória.

- Modifique seu código da seguinte forma:

```
// Prints a string's address as well the addresses of its chars

#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string s = "HI!";
    printf("%p\n", s);
    printf("%p\n", &s[0]);
    printf("%p\n", &s[1]);
    printf("%p\n", &s[2]);
    printf("%p\n", &s[3]);
}
```

Observe que o código acima imprime os endereços de memória de cada caractere na string `s`. O `&` símbolo é usado para mostrar o endereço de cada elemento da string. Ao executar este código, observe que os elementos `0`, `1`, `2`, e `3` estão lado a lado na memória.

- Da mesma forma, você pode modificar seu código da seguinte maneira:

```
// Declares a string with CS50 Library

#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string s = "HI!";
    printf("%s\n", s);
}
```

Observe que este código apresentará a string que começa na posição de `s`. Este código efetivamente remove as "rodinhas de treinamento" do `string` tipo de dados oferecido por `cs50.h`. Este é um código C puro, sem a estrutura da biblioteca cs50.

- Ao remover as rodinhas de apoio, você pode modificar seu código novamente:

```
// Declares a string without CS50 Library

#include <stdio.h>

int main(void)
{
    char *s = "HI!";
    printf("%s\n", s);
}
```

Observe que `cs50.h` foi removido. Uma string é implementada como um `char *`.

- Você pode imaginar como uma string, enquanto tipo de dado, é criada.

- Na semana passada, aprendemos como criar seu próprio tipo de dados como uma estrutura (struct).
- A biblioteca cs50 inclui uma estrutura da seguinte forma: `typedef char *string`
- Essa estrutura, ao usar a biblioteca cs50, permite usar um tipo de dados personalizado chamado `string`.

Aritmética de ponteiros

- A aritmética de ponteiros é a capacidade de realizar cálculos matemáticos em endereços de memória.
- Você pode modificar seu código para imprimir cada endereço de memória na string da seguinte forma:

```
// Prints a string's chars

#include <stdio.h>

int main(void)
{
    char *s = "HI!";
    printf("%c\n", s[0]);
    printf("%c\n", s[1]);
    printf("%c\n", s[2]);
}
```

  Observe que estamos imprimindo cada caractere na posição de `s`.

- Além disso, você pode modificar seu código da seguinte forma:

```
// Prints a string's chars via pointer arithmetic

#include <stdio.h>

int main(void)
{
    char *s = "HI!";
    printf("%c\n", *s);
    printf("%c\n", *(s + 1));
    printf("%c\n", *(s + 2));
}
```

  Observe que o primeiro caractere na posição de `s` é impresso. Em seguida, o caractere na posição de `s + 1` é impresso, e assim por diante.

- Da mesma forma, considere o seguinte:

```
// Prints substrings via pointer arithmetic

#include <stdio.h>

int main(void)
{
    char *s = "HI!";
```

```
    printf("%s\n", s);
    printf("%s\n", s + 1);
    printf("%s\n", s + 2);
}
```

Observe que este código imprime os valores armazenados em vários locais de memória, começando com `s`.

Comparação de strings

- Uma sequência de caracteres é simplesmente uma matriz de caracteres identificada pela posição do seu primeiro byte.
- Anteriormente no curso, consideramos a comparação de números inteiros. Poderíamos representar isso em código digitando `code compare.c` o seguinte no terminal:

```
// Compares two integers

#include <cs50.h>
#include <stdio.h>

int main(void)
{
    // Get two integers
    int i = get_int("i: ");
    int j = get_int("j: ");

    // Compare integers
    if (i == j)
    {
        printf("Same\n");
    }
    else
    {
        printf("Different\n");
    }
}
```

Observe que este código recebe dois números inteiros do usuário e os compara.

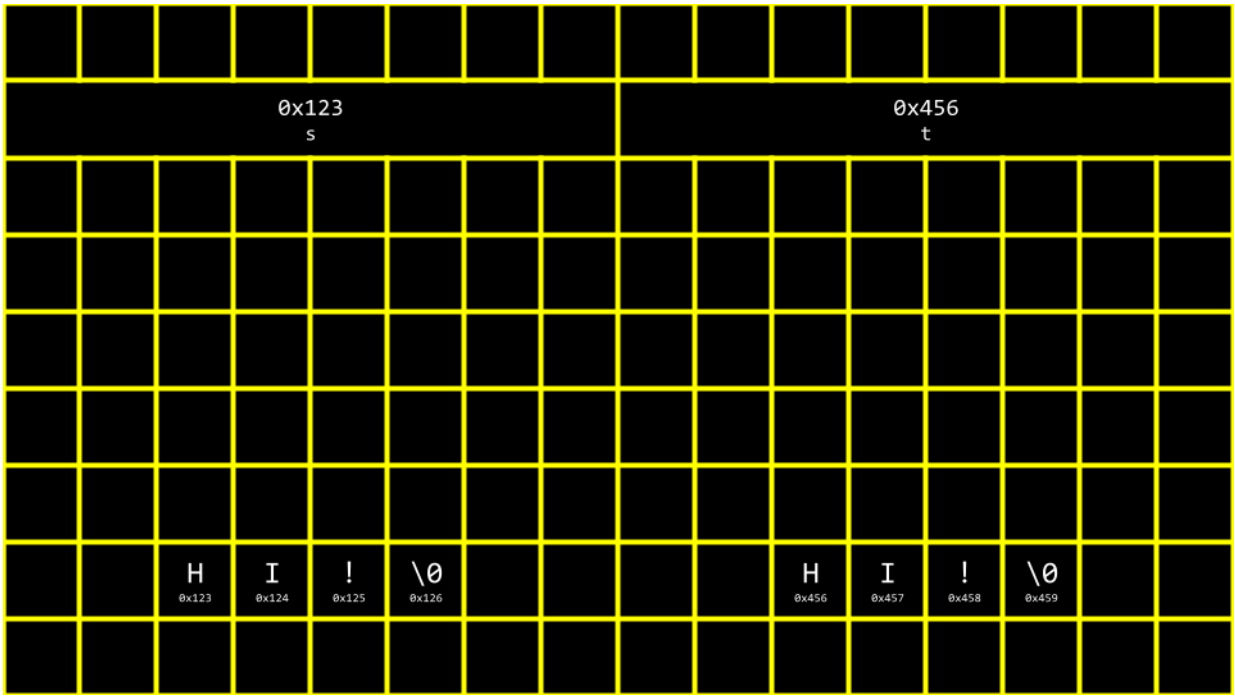
- No caso de strings, porém, não é possível comparar duas strings usando o `==` operador.
- Utilizar o `==` operador `&` numa tentativa de comparar strings resultará na comparação das posições de memória das strings em vez dos caracteres nelas contidos. Consequentemente, recomendamos a utilização de `&` `strcmp`.
- Para ilustrar isso, modifique seu código da seguinte forma:

```
// Compares two strings' addresses

#include <cs50.h>
#include <stdio.h>

int main(void)
{
    // Get two strings
```

```
    char *s = get_string("s: ");
    char *t = get_string("t: ");

    // Compare strings' addresses
    if (s == t)
    {
        printf("Same\n");
    }
    else
    {
        printf("Different\n");
    }
}
```

Percebendo que digitar `HI!` para ambas as strings ainda resulta na saída de `Different`.

- Por que essas sequências parecem diferentes? Você pode usar o seguinte para visualizar o motivo:



- Portanto, o código `compare.c` acima está, na verdade, tentando verificar se os endereços de memória são diferentes, e não as strings em si.
- Utilizando essa ferramenta `strcmp`, podemos corrigir nosso código:

```
// Compares two strings using strcmp

#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    // Get two strings
    char *s = get_string("s: ");
    char *t = get_string("t: ");

    // Compare strings
    if (strcmp(s, t) == 0)
    {
```

```
        printf("Same\n");
    }
    else
    {
        printf("Different\n");
    }
}
```

Observe que `strcmp` pode retornar `0` se as strings forem iguais.

- Para ilustrar melhor como essas duas strings estão em locais diferentes, modifique seu código da seguinte forma:

```
// Prints two strings

#include <cs50.h>
#include <stdio.h>

int main(void)
{
    // Get two strings
    char *s = get_string("s: ");
    char *t = get_string("t: ");

    // Print strings
    printf("%s\n", s);
    printf("%s\n", t);
}
```

Observe como agora temos duas strings separadas armazenadas, provavelmente em dois locais diferentes.

- Com uma pequena modificação, você pode visualizar a localização dessas duas strings armazenadas:

```
// Prints two strings' addresses

#include <cs50.h>
#include <stdio.h>

int main(void)
{
    // Get two strings
    char *s = get_string("s: ");
    char *t = get_string("t: ");

    // Print strings' addresses
    printf("%p\n", s);
    printf("%p\n", t);
}
```

Observe que o " `%s` has" foi alterado para " `%p` in the print statement".

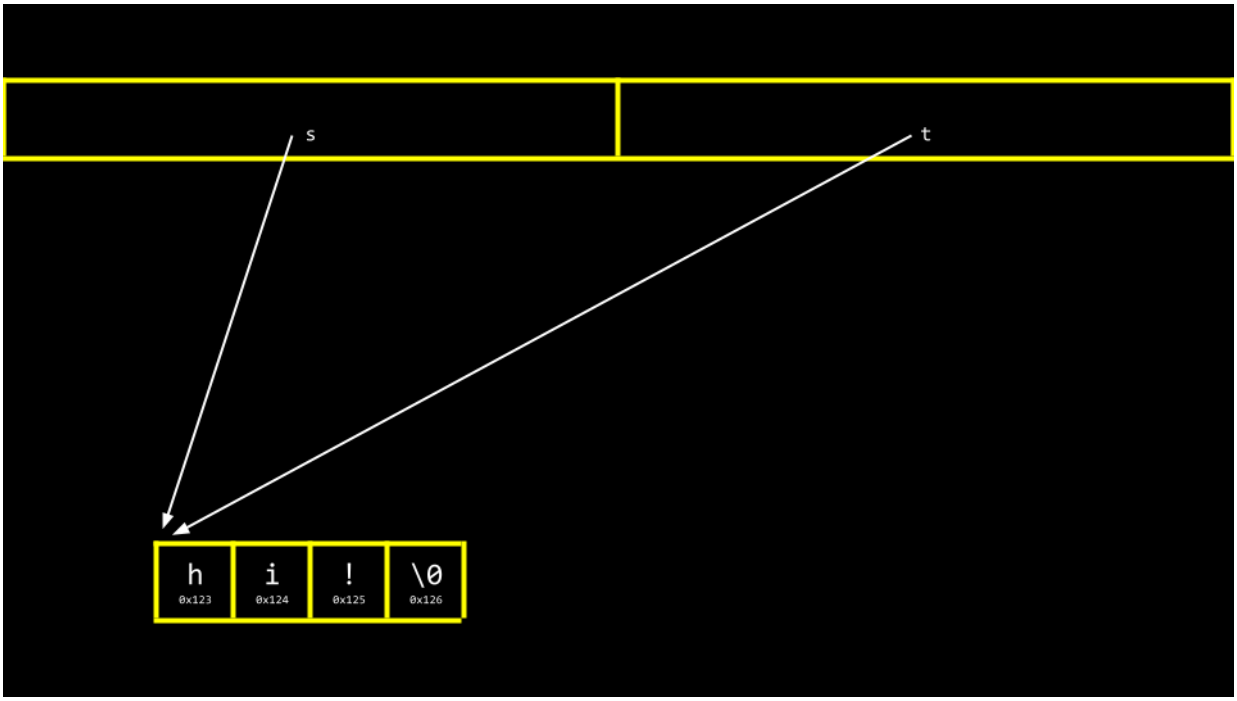
Copiar e alocar espaço em disco

- Uma necessidade comum em programação é copiar uma string para outra.
- Na janela do terminal, digite `code copy.c` e escreva o código da seguinte forma:

```c
// Capitalizes a string

#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    // Get a string
    string s = get_string("s: ");

    // Copy string's address
    string t = s;

    // Capitalize first letter in string
    t[0] = toupper(t[0]);

    // Print string twice
    printf("s: %s\n", s);
    printf("t: %s\n", t);
}
```

  Observe que `string t = s` o endereço de é copiado `s` para `t`. Isso não atinge o objetivo desejado. A string não é copiada – apenas o endereço é. Além disso, observe a inclusão de `ctype.h`.

- Você pode visualizar o código acima da seguinte forma:



  Note que ` `s`a` e ` `t`b` ainda apontam para os mesmos blocos de memória. Isso não é uma cópia autêntica de uma string. Em vez disso, são dois ponteiros apontando para a mesma string.

- Antes de abordarmos esse desafio, é importante garantir que não ocorra uma *falha de segmentação* em nosso código, onde tentamos copiar `string s` para `string t` um local onde esse local `string t` não existe. Podemos usar a `strlen` função da seguinte forma para evitar isso:

```
// Capitalizes a string, checking length first

#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    // Get a string
    string s = get_string("s: ");

    // Copy string's address
    string t = s;

    // Capitalize first letter in string
    if (strlen(t) > 0)
    {
        t[0] = toupper(t[0]);
    }

    // Print string twice
    printf("s: %s\n", s);
    printf("t: %s\n", t);
}
```

  Note que isso `strlen` é usado para garantir `string t` que exista. Se não existir, nada será copiado.

- Para podermos fazer uma cópia autêntica da string, precisaremos introduzir dois novos blocos de construção. Primeiro, `malloc` permite que você, o programador, aloque um bloco de memória de tamanho específico. Segundo, `free` permite que você instrua o compilador a *liberar* esse bloco de memória que você alocou anteriormente.

- Podemos modificar nosso código para criar uma cópia autêntica da nossa string da seguinte forma:

```
// Capitalizes a copy of a string

#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    // Get a string
    char *s = get_string("s: ");

    // Allocate memory for another string
```

```
    char *t = malloc(strlen(s) + 1);

    // Copy string into memory, including '\0'
    for (int i = 0; i <= strlen(s); i++)
    {
        t[i] = s[i];
    }

    // Capitalize copy
    t[0] = toupper(t[0]);

    // Print strings
    printf("s: %s\n", s);
    printf("t: %s\n", t);
}
```

Observe que isso `malloc(strlen(s) + 1)` cria um bloco de memória com o comprimento da string `s` mais um. Isso permite a inclusão do caractere *nulo* na nossa string final copiada. Em seguida, o loop percorre a string e atribui cada valor àquela mesma posição na string . `\0` `for` `s` `t`

- Descobrimos que nosso código é ineficiente. Modifique seu código da seguinte forma:

```
// Capitalizes a copy of a string, defining n in loop too

#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    // Get a string
    char *s = get_string("s: ");

    // Allocate memory for another string
    char *t = malloc(strlen(s) + 1);

    // Copy string into memory, including '\0'
    for (int i = 0, n = strlen(s); i <= n; i++)
    {
        t[i] = s[i];
    }

    // Capitalize copy
    t[0] = toupper(t[0]);

    // Print strings
    printf("s: %s\n", s);
    printf("t: %s\n", t);
}
```

Observe que `n = strlen(s)` agora a função está definida no lado esquerdo do loop `for loop`. É melhor não chamar funções desnecessárias na condição intermediária do `for` loop, pois isso fará com que ele seja executado repetidamente. Ao passar `n = strlen(s)` para o lado esquerdo, a função `strlen` é executada apenas uma vez.

- A `C` linguagem possui uma função integrada para copiar strings chamada `strcpy`. Ela pode ser implementada da seguinte forma:

```
// Capitalizes a copy of a string using strcpy

#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    // Get a string
    char *s = get_string("s: ");

    // Allocate memory for another string
    char *t = malloc(strlen(s) + 1);

    // Copy string into memory
    strcpy(t, s);

    // Capitalize copy
    t[0] = toupper(t[0]);

    // Print strings
    printf("s: %s\n", s);
    printf("t: %s\n", t);
}
```

  Observe que isso `strcpy` executa o mesmo trabalho que nosso `for` loop fazia anteriormente.

- Ambas `get_string` as funções `malloc` retornam `NULL` um valor especial na memória, caso algo dê errado. Você pode escrever um código que verifique essa `NULL` condição da seguinte forma:

```
// Capitalizes a copy of a string without memory errors

#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    // Get a string
    char *s = get_string("s: ");
    if (s == NULL)
    {
        return 1;
    }

    // Allocate memory for another string
    char *t = malloc(strlen(s) + 1);
    if (t == NULL)
```

```
    {
        return 1;
    }

    // Copy string into memory
    strcpy(t, s);

    // Capitalize copy
    if (strlen(t) > 0)
    {
        t[0] = toupper(t[0]);
    }

    // Print strings
    printf("s: %s\n", s);
    printf("t: %s\n", t);

    // Free memory
    free(t);
    return 0;
}
```

Observe que, se a string obtida tiver o comprimento especificado `0` ou se a alocação de memória falhar, `NULL` será retornado. Além disso, observe que isso `free` informa ao computador que você terminou de usar o bloco de memória criado por meio de `malloc` `malloc`.

Valgrind

- *O Valgrind* é uma ferramenta que verifica se há problemas relacionados à memória nos seus programas `malloc`, especificamente se você está utilizando `free` toda a memória alocada.
- Considere o seguinte código para `memory.c`:

```
// Demonstrates memory errors via valgrind

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *x = malloc(3 * sizeof(int));
    x[1] = 72;
    x[2] = 73;
    x[3] = 33;
}
```

Observe que a execução deste programa não causa nenhum erro. Embora `malloc` seja usada para alocar memória suficiente para um array, o código falha ao liberar `free` essa memória alocada.

- Se você digitar ` valgrind` `make memory` seguido de `--` `valgrind ./memory`, você receberá um relatório do Valgrind que indicará onde a memória foi perdida como resultado do seu

programa. Um erro que o Valgrind revela é que tentamos atribuir o valor de `x` `33` na quarta posição do array, quando alocamos apenas um array de tamanho `x` `3`. Outro erro é que nunca liberamos a memória alocada `x`.

- Você pode modificar seu código para liberar a memória da `x` seguinte forma:

```
// Demonstrates memory errors via valgrind

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *x = malloc(3 * sizeof(int));
    x[1] = 72;
    x[2] = 73;
    x[3] = 33;
    free(x);
}
```

Note que executar o valgrind novamente agora não resulta em vazamentos de memória.

Valores de lixo

- Quando você solicita um bloco de memória ao compilador, não há garantia de que essa memória estará vazia.
- É muito possível que a memória alocada já tenha sido utilizada pelo computador. Consequentemente, você pode ver *valores inválidos* ou inconsistentes . Isso ocorre porque você recebeu um bloco de memória, mas não o inicializou. Por exemplo, considere o seguinte código para : `garbage.c`

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int scores[1024];
    for (int i = 0; i < 1024; i++)
    {
        printf("%i\n", scores[i]);
    }
}
```

Observe que a execução deste código alocará `1024` espaços na memória para seu array, mas o `for` loop provavelmente mostrará que nem todos os valores ali contidos são nulos `0`. É sempre uma boa prática estar ciente da possibilidade de valores inválidos quando você não inicializa blocos de memória com algum outro valor, como zero ou outro tipo.

Diversão com o ponteiro e a chupeta

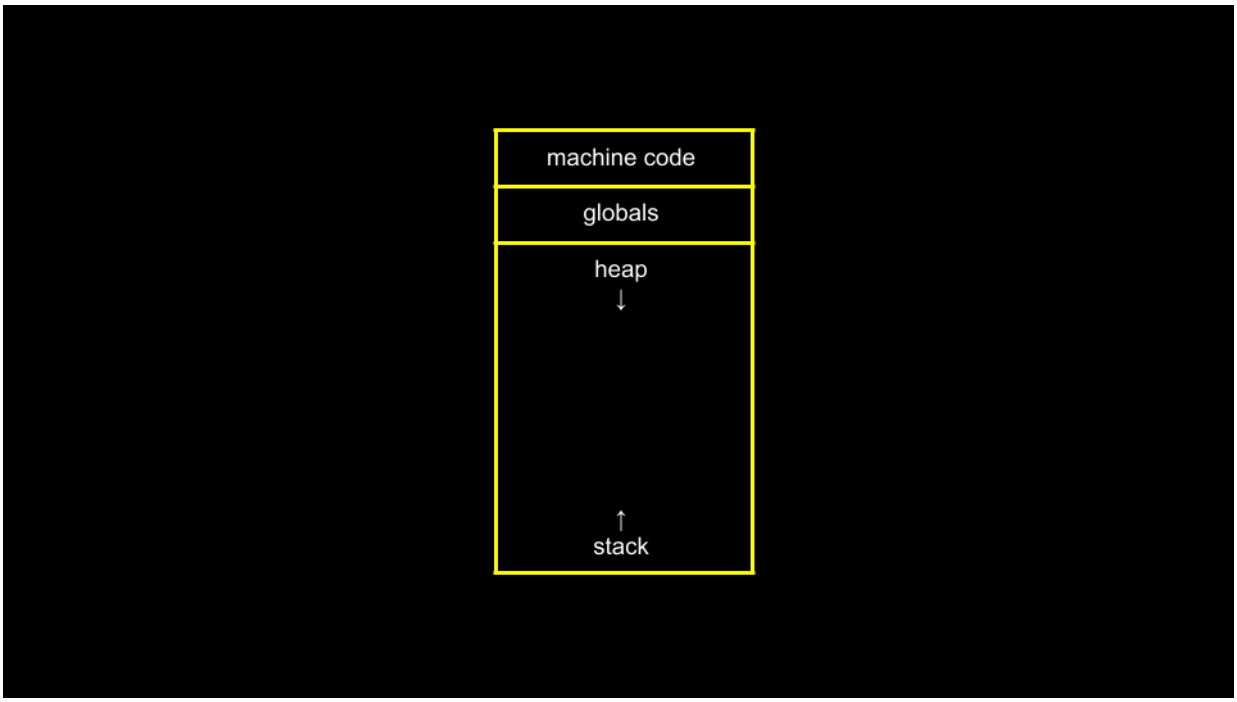
- Assistimos a um vídeo da Universidade de Stanford (https://www.youtube.com/watch?v=5VnDaHBi8dM) que nos ajudou a visualizar e entender os ponteiros.

Troca

- No mundo real, uma necessidade comum em programação é trocar dois valores. Naturalmente, é difícil trocar duas variáveis sem um espaço temporário para armazená-las. Na prática, você pode digitar `code swap.c` e escrever o código a seguir para ver isso em ação:

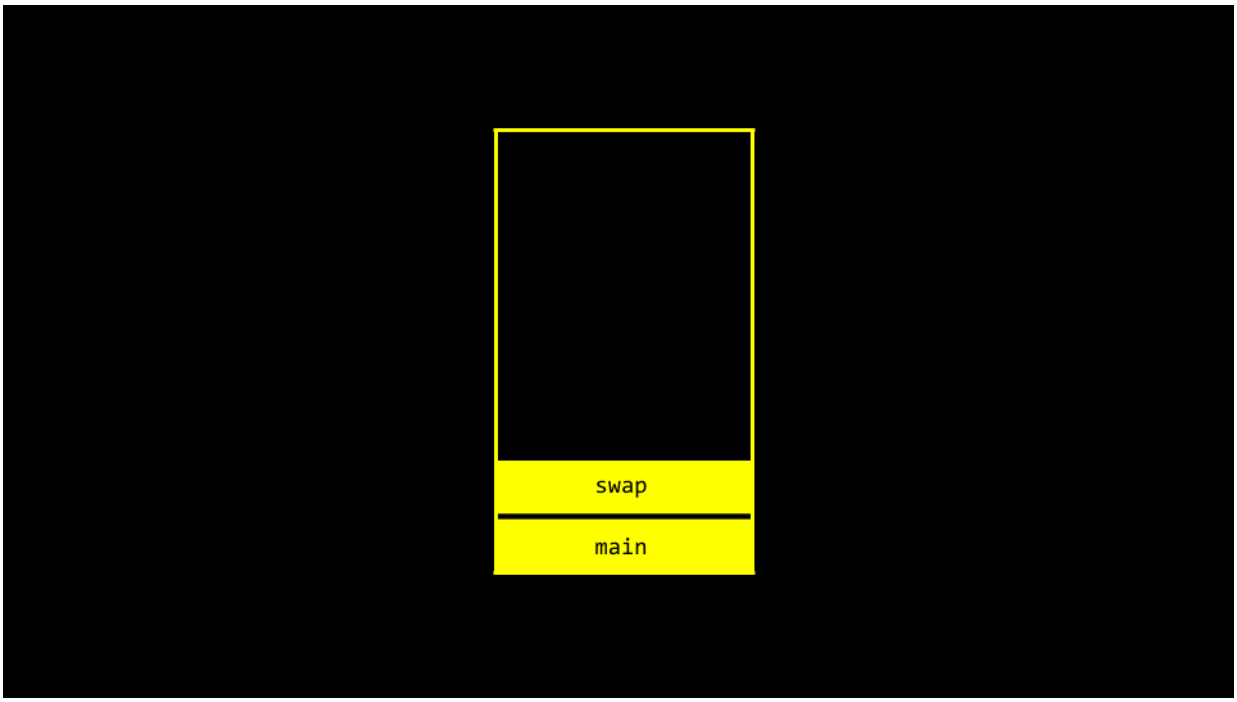
```
// Fails to swap two integers

#include <stdio.h>

void swap(int a, int b);

int main(void)
{
    int x = 1;
    int y = 2;

    printf("x is %i, y is %i\n", x, y);
    swap(x, y);
    printf("x is %i, y is %i\n", x, y);
}

void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

  Note que, embora este código seja executado, ele não funciona. Os valores, mesmo depois de serem enviados para a `swap` função, não são trocados. Por quê?
- Ao passar valores para uma função, você está fornecendo apenas cópias. O *escopo* de `x` `x` e `y` `y` é limitado à função principal, conforme o código está escrito atualmente. Ou seja, os valores de `x` `x` e `y` `y` criados dentro das `{}` chaves da `main` função têm escopo apenas dentro da `main` função. No código acima, ` `x` x` e `y` `y` estão sendo passados por *valor* .
- Considere a seguinte imagem:



Observe que as variáveis *globais* , que não utilizamos neste curso, residem em um local específico da memória. Diversas funções são armazenadas em `stack` outra área da memória.

- Agora, considere a seguinte imagem:



Note que ` `main` x` e `y` `swap` possuem dois *quadros* ou áreas de memória separados. Portanto, não podemos simplesmente passar os valores de uma função para outra para alterá-los.

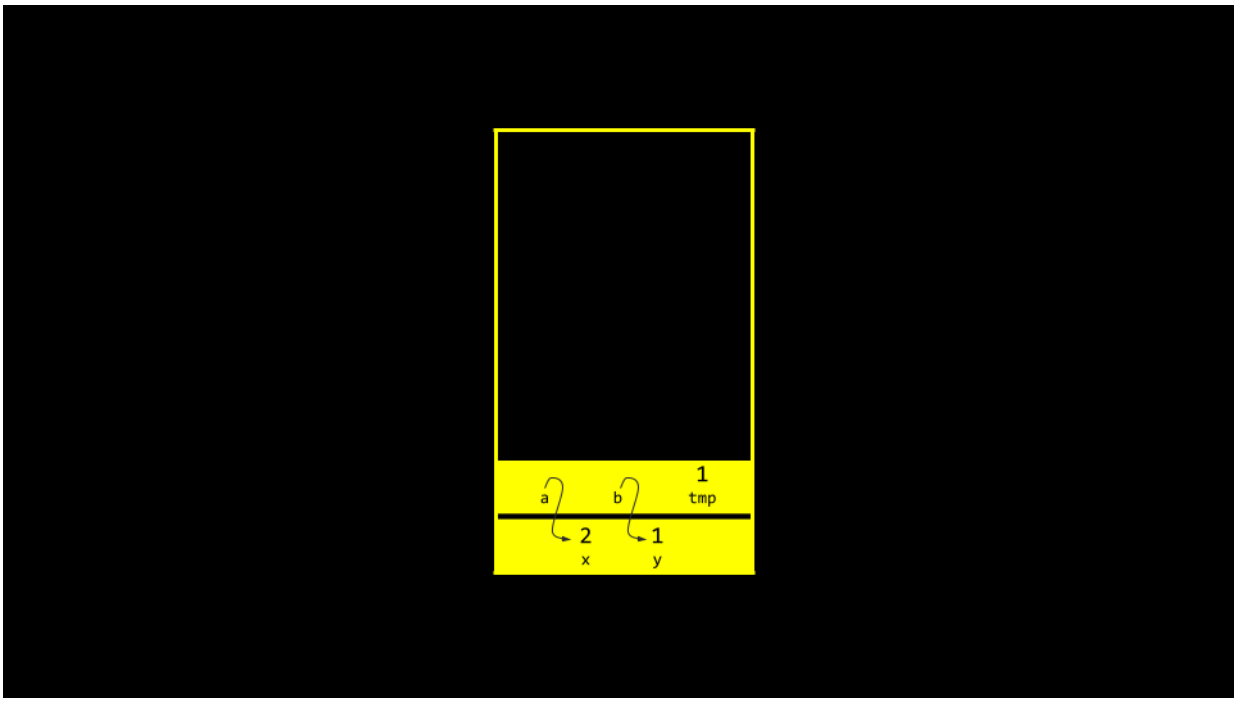
- Modifique seu código da seguinte forma:

```
// Swaps two integers using pointers

#include <stdio.h>

void swap(int *a, int *b);
```

```c
int main(void)
{
    int x = 1;
    int y = 2;

    printf("x is %i, y is %i\n", x, y);
    swap(&x, &y);
    printf("x is %i, y is %i\n", x, y);
}

void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Note que as variáveis não são passadas por *valor* , mas por *referência* . Ou seja, os endereços de ` a` a` e `b` b` são fornecidos à função. Portanto, a `swap` função pode saber onde fazer alterações nos valores reais de `a` `a` e `b` `b` a partir da função principal.

- Você pode visualizar isso da seguinte forma:



Transbordamento

- Um *estouro de heap* ocorre quando você ultrapassa o limite da heap, acessando áreas da memória que não deveria.
- Um *estouro de pilha* ocorre quando muitas funções são chamadas simultaneamente, excedendo a quantidade de memória disponível.
- Ambos são considerados *estouros de buffer* .

`scanf`

- Em CS50, criamos funções `get_int` para simplificar o processo de obtenção de dados do usuário.
- `scanf` É uma função integrada que pode receber entrada do usuário.
- Podemos reimplementar `get_int` com bastante facilidade usando `scanf` o seguinte método:

```
// Gets an int from user using scanf

#include <stdio.h>

int main(void)
{
    int n;
    printf("n: ");
    scanf("%i", &n);
    printf("n: %i\n", n);
}
```

  Observe que o valor de `n` é armazenado na localização de `n` na linha `scanf("%i", &n)`.

- No entanto, tentar reimplementar `get_string` não é fácil. Considere o seguinte:

```
// Dangerously gets a string from user using scanf with array

#include <stdio.h>

int main(void)
{
    char s[4];
    printf("s: ");
    scanf("%s", s);
    printf("s: %s\n", s);
}
```

  Note que não `&` é necessário especificar o tamanho da string, pois ela é um tipo especial de dado. Mesmo assim, este programa não funcionará corretamente todas as vezes que for executado. Em nenhum momento alocamos a quantidade de memória necessária para a string. Aliás, não sabemos qual o tamanho da string que o usuário poderá inserir! Além disso, não sabemos quais valores inválidos podem existir no endereço de memória alocado.

- Além disso, seu código poderia ser modificado da seguinte forma. No entanto, precisamos pré-alocar uma certa quantidade de memória para a string:

```
// Using malloc

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *s = malloc(4);
    if (s == NULL)
    {
```

```
        return 1;
    }
    printf("s: ");
    scanf("%s", s);
    printf("s: %s\n", s);
    free(s);
    return 0;
}
```

Observe que, se for fornecida uma string de quatro bytes, você *poderá* receber um erro.

- Simplificando nosso código da seguinte forma, podemos entender melhor esse problema essencial da pré-alocação:

```
#include <stdio.h>

int main(void)
{
    char s[4];
    printf("s: ");
    scanf("%s", s);
    printf("s: %s\n", s);
}
```

Observe que, se pré-alocarmos um array de tamanho `4`, podemos digitar `cat` e o programa funciona. No entanto, uma string maior que isso *pode* gerar um erro.

- Às vezes, o compilador ou o sistema que o executa pode alocar mais memória do que indicamos. Fundamentalmente, porém, o código acima é inseguro. Não podemos confiar que o usuário inserirá uma string que caiba na memória pré-alocada.

Entrada/Saída de Arquivos

- Você pode ler e manipular arquivos. Embora esse tópico seja abordado com mais detalhes em uma semana futura, considere o seguinte código para `phonebook.c`:

```
// Saves names and numbers to a CSV file

#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    // Open CSV file
    FILE *file = fopen("phonebook.csv", "a");

    // Get name and number
    char *name = get_string("Name: ");
    char *number = get_string("Number: ");

    // Print to file
    fprintf(file, "%s,%s\n", name, number);

    // Close file
```

```
    fclose(file);
}
```

Observe que este código usa ponteiros para acessar o arquivo.

- Você pode criar um arquivo chamado `phonebook.csv`phonebook.csv` antes de executar o código acima ou baixar o arquivo `phonebook.csv` (https://cdn.cs50.net/2024/fall/lectures/4/src4/phonebook.csv?download) . Após executar o programa e inserir um nome e um número de telefone, você perceberá que esses dados serão salvos no seu arquivo CSV.
- Se quisermos garantir que isso `phonebook.csv` exista antes de executar o programa, podemos modificar nosso código da seguinte forma:

```
// Saves names and numbers to a CSV file

#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    // Open CSV file
    FILE *file = fopen("phonebook.csv", "a");
    if (!file)
    {
        return 1;
    }

    // Get name and number
    char *name = get_string("Name: ");
    char *number = get_string("Number: ");

    // Print to file
    fprintf(file, "%s,%s\n", name, number);

    // Close file
    fclose(file);
}
```

Observe que este programa se protege contra um `NULL` ponteiro invocando `return 1`.

- Podemos implementar nosso próprio programa de cópia digitando `code cp.c` e escrevendo o código da seguinte forma:

```
// Copies a file

#include <stdio.h>
#include <stdint.h>

typedef uint8_t BYTE;

int main(int argc, char *argv[])
{
    FILE *src = fopen(argv[1], "rb");
    FILE *dst = fopen(argv[2], "wb");
```

```
    BYTE b;

    while (fread(&b, sizeof(b), 1, src) != 0)
    {
        fwrite(&b, sizeof(b), 1, dst);
    }

    fclose(dst);
    fclose(src);
}
```

Observe que este arquivo cria nosso próprio tipo de dados chamado BYTE, que tem o tamanho de um uint8_t. Em seguida, o arquivo lê um valor `BYTE` e o grava em outro arquivo.

- Os BMPs também são conjuntos de dados que podemos examinar e manipular. Esta semana, vocês farão exatamente isso em suas listas de exercícios!

Resumindo

Nesta lição, você aprendeu sobre ponteiros, que permitem acessar e manipular dados em locais específicos da memória. Especificamente, exploramos...

- Arte em pixel
- Hexadecimal
- Memória
- Dicas
- Cordas
- Aritmética de ponteiros
- Comparação de strings
- Copiando
- malloc e Valgrind
- Valores de lixo
- Troca
- Transbordamento
- `scanf`
- Entrada/Saída de Arquivos

Até a próxima!