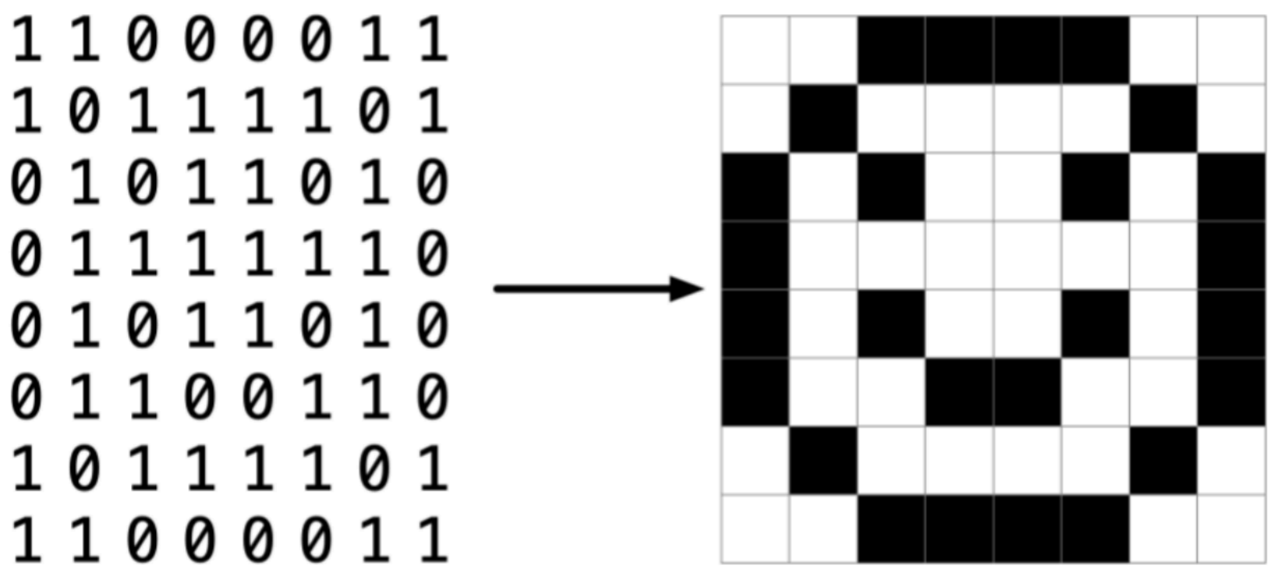
Lecture 4_Memory

Pixel Art

1. Pixels are squares, individual dots, of color that are arranged on an up-down, left-right grid.
2. You can imagine an image as a map of bits, where zeros represent black and ones represent white.



3. *RGB*, or *red, green, blue*, are numbers that represent the amount of each of these colors.

Memory

1. We can interpret the whole Memory as concurrent blocks. And each smallest block represents a byte (8 bits).
2. The C language has two powerful operators that relate to memory:

```
& Provides the address of something stored in memory.
  &n can be literally translated as "the address of n"
```

- Instructs the compiler to go to a location in memory.
  *n as "the value stored in the address that n stores"

```
Example:
```

```c
// Prints an integer's address
#include <stdio.h>

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

The `%p` allows us to view the address of a location in memory.
Here, it tells the printf to go to the address of n, and then print out the address.

Pointer

1. A *pointer* is a variable that stores the address of something. In other words, a pointer is an address in the computer's memory.
2. A pointer is usually stored as an 8-byte value

Use of &

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

```
Here, p is a pointer that points to an int, so p stores the address of the int
(p is the address of the int), and the int is n. The right side means the
address of n. So the expression means, p is the address(points to the address)
of int n.
```

Now leverage p, we can change the code into as followed:

```c
// Stores and prints an integer's address
#include <stdio.h>
int main(void)
{
    int n = 50;
    int *p = &n;
    printf("%p\n", p);
}
```

```
Here, %p requires a data type of address, and p as a pointer satisfies. So no
```

```
more & is needed in indicating the variable is an address.
```

Use of *

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

#include <stdio.h>

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

```
Here, p pertains to be the pointer pointing to an int, and *p means go to the
address and find the value, i.e. go to the address of n and find the value
stored in that address,that is n. so *p is the value stored in the address
that p stores.
```

Strings

1. A string is an array of characters ended with $\backslash 0$
2. In C, there is no data type called string, instead, there is $char*$, which means a string is literally a pointer that points to a character. This make sense because a pointer tells the compiler where the first byte of the string exists in memory.
3. A string of characters is simply an array of characters identified by the location of its first byte.

```c
// Declares a string with CS50 Library

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

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

is equal to

```c
// Declares a string without CS50 Library

#include <stdio.h>

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

This is what happens under the hood:
The cs50 library includes a struct as follows: `typedef char *string`
4. One cannot compare two strings using the `==` operator, since it is comparing the starting address of two strings, which are apparently different.

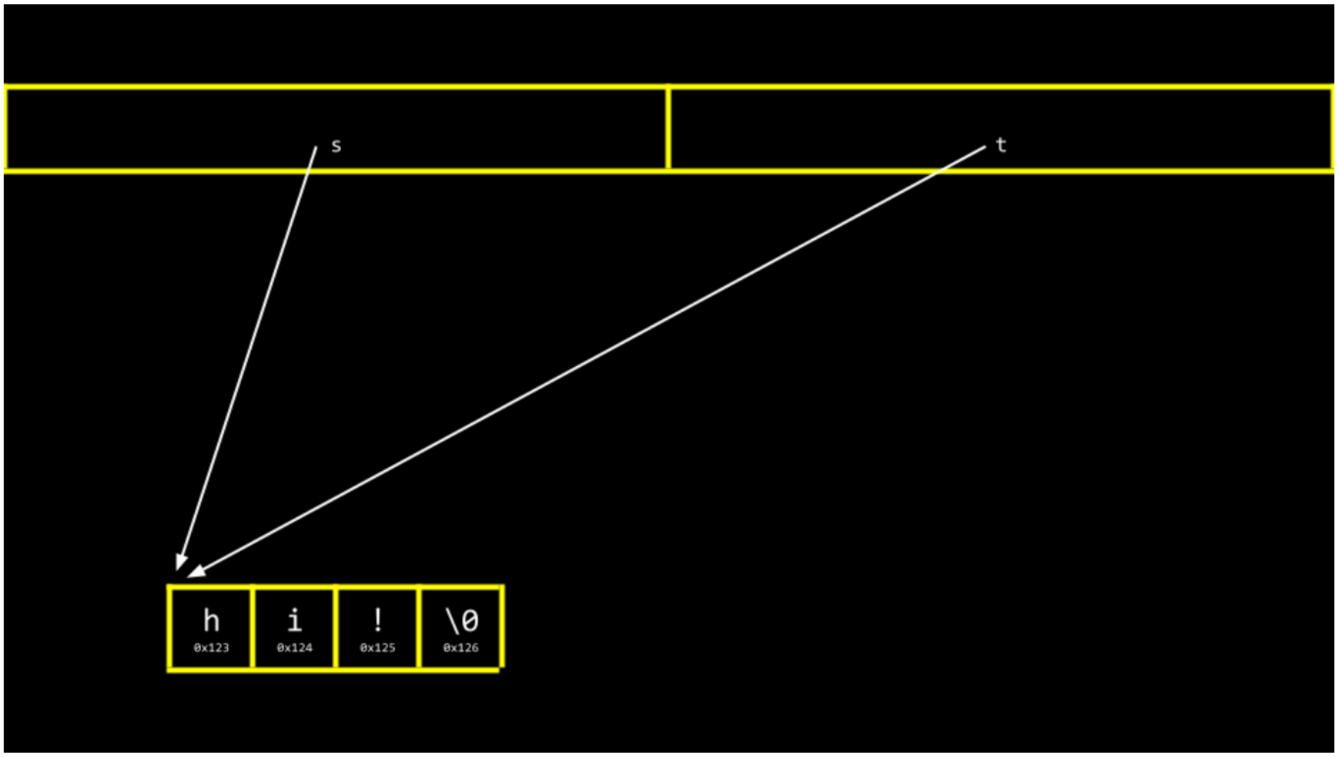
Copying and Malloc

String Copying

```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);
}
```

```
Even if we only change the first character of t, the first letter of s and t
will both be capitalized, given that s and t points to the same string address
because no memory is allocated to t
```

You can visualize the above code as follows:



Malloc and Free

1. `Malloc` allows you to allocate a block of a specific size of memory.
2. `Free` allows you to tell the compiler to *free up* that block of memory you previously allocated.
3. One way to copy an authentic copy of a string is as followed:

```
// 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);
}
```

```
Notice that malloc(strlen(s) + 1) creates a block of memory that is the length
of the string s plus one. This allows for the inclusion of
the null \0 character in our final copied string.
One way to optimize the code is to define n = strlen(s) in the left-hand side
of the for loop, so that it won't calculate strlen(s) in every loop.
```

```
    for (int i = 0, n = strlen(s); i <= n; i++)
    {
        t[i] = s[i];
    }
```

```
Good news is that C has a built-in function to copy strings called strcpy, so
we can directly malloc memory, then call the strcpy function.
```

```
// 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);
}
```

4. If something goes wrong in `Malloc` , it returns `NULL` , which can be used to check the validation of Malloc operation

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

Valgrind

1. *Valgrind* is a tool that can check to see if there are memory-related issues with programs that utilizes `malloc` . Specifically, it checks to see if you `free` all the memory you allocated.
2. To use this tool, type the following into the terminal

```
    make program_name
    valgrind ./program_name
```

Garbage Values

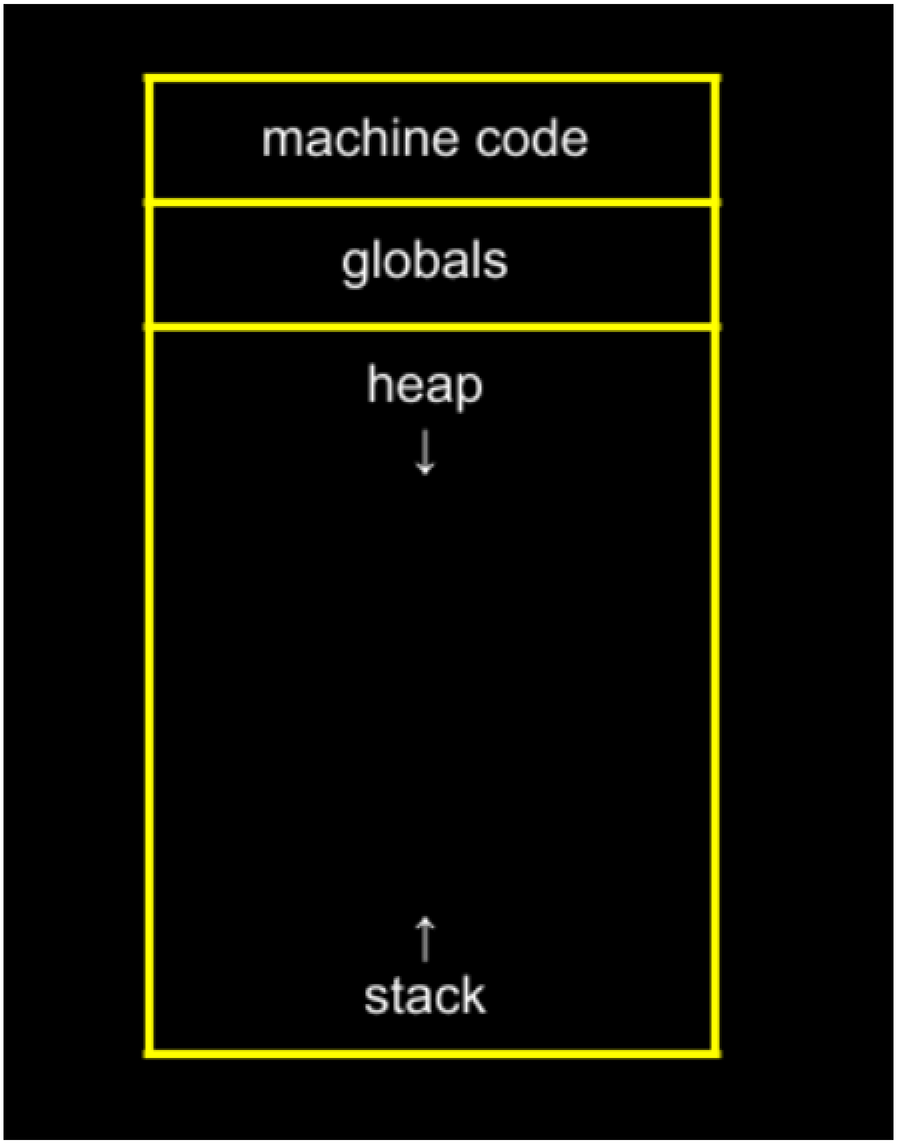
1. When asking the compiler for a block of memory, it is very likely that the memory has previously been utilized, so we might see *junk* or *garbage values*, as a result of asking for memory without initializing it.

Swapping

1. If we are calling `swap` function as followed, it fails to actually swap the value, because when passing values to a function, you are only providing copies. The *scope* of `x` and `y` is limited to the `main` function. `x` and `y` are being passed by *value*.

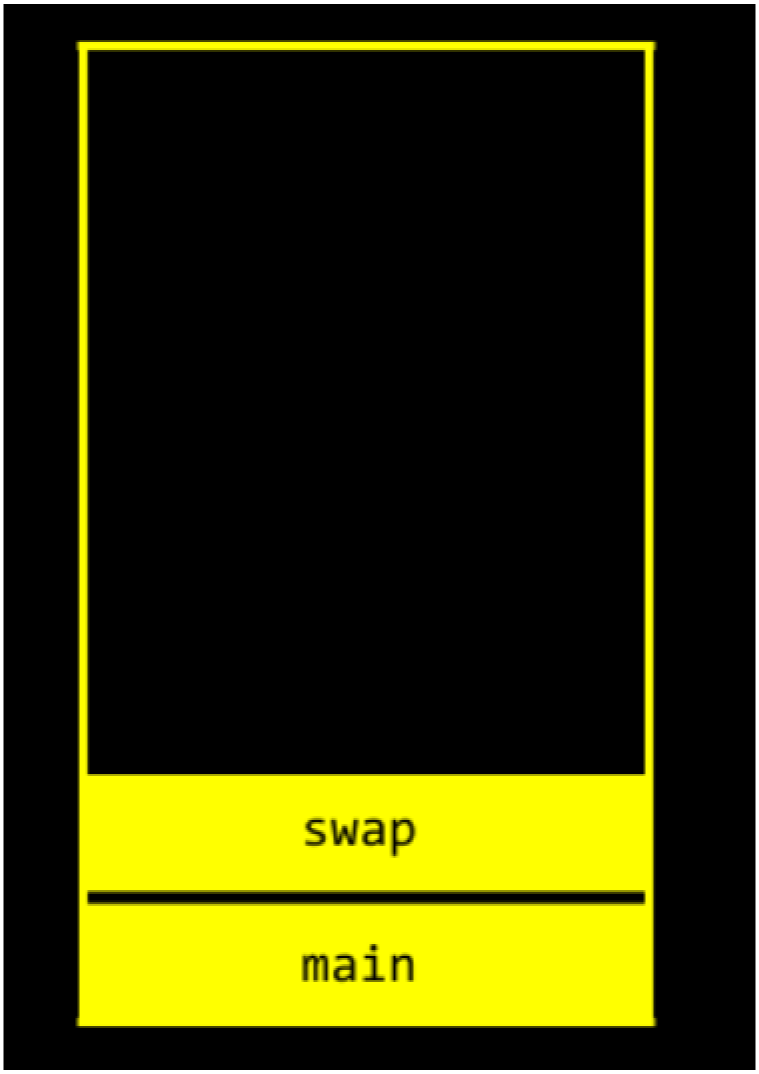
```c
// Fails to swap two integers

#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;
}
```

2. We can use the Memory Image to make this clearer:

   Global variables live in one place in memory.
   Various functions are stored in the `stack` in another area of memory.
3. The `main` function and the `swap` function is stored in the stack. They have two separate *frames* or areas of memory. Therefore, we cannot simply pass the values from one

function to another to change them.



4. So we use pointer to make the swap:

```
// Swaps two integers using pointers

#include <stdio.h>

void swap(int *a, int *b);// pass two pointers to the swap function, which are
the addresses of a and b

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

    printf("x is %i, y is %i\n", x, y);
    swap(&x, &y);//& is "the address of the variable"
    printf("x is %i, y is %i\n", x, y);
}
```

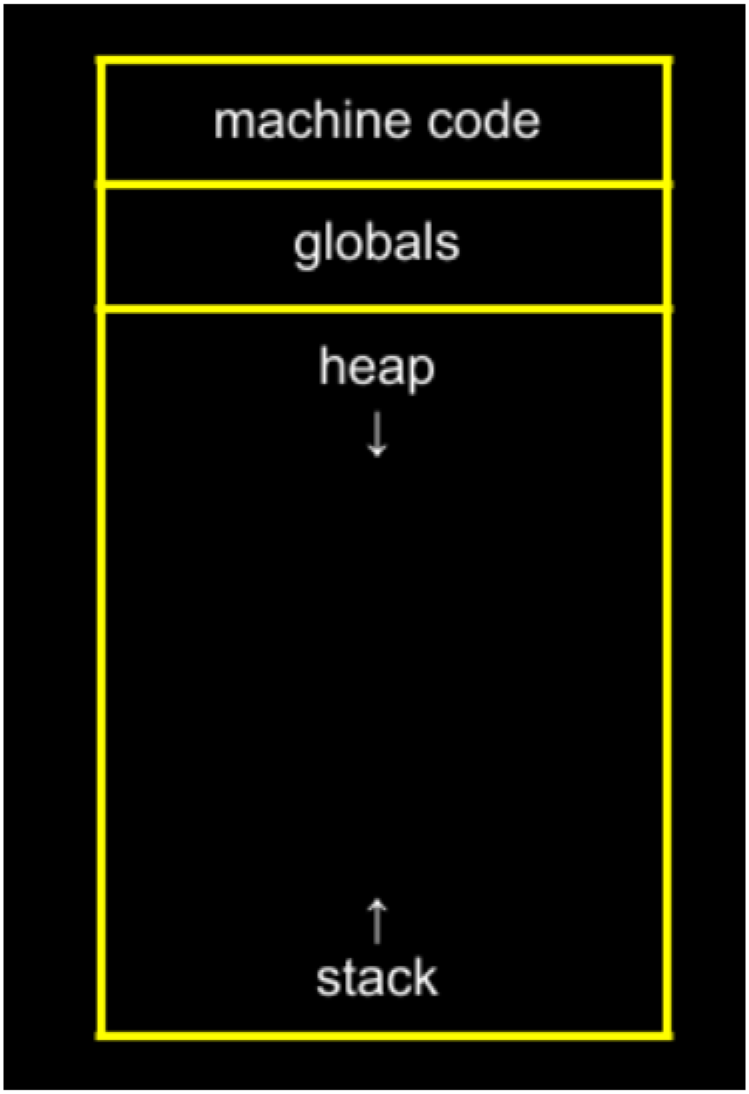
```c
void swap(int *a, int *b)
{
    int tmp = *a;//create a tmp value that now has the value of
x(dereferencing a means the value of the address that a stores, in other
words, * means go to a and get the value, then pass it to tmp)
    *a = *b;// go to the address of a, find the value, replace the value of
y(*b)
    *b = tmp;//go to the address of b, find the value, replace the value of
x(tmp)
    // now the address of x contains the the value of y, vice versa, swap ends
}
```

```
Now variables are not passed by value but by reference. That is,the addresses
of a and b are provided to the swap function.
```

Overflow

1. A *heap overflow* is when you overflow the heap, touching areas of memory you are not supposed to.
2. A *stack overflow* is when too many functions are called, overflowing the amount of memory available.

3. Both of these are considered *buffer overflows*.



Scanf

1. Replace get_int with scanf

```
    printf("n: ");
    scanf("%i", &n);
```

```
where %i specifies the type of variable as input, &n specifies the address
that the value is stored in
```

2. Replace get_string with scanf
   The problem is that we need to pre-allocate memory for the string to store it.

File I/O

1. Like in mps, sometimes we need to access information in another file like a `txt` file, so we need the ability to open a file in a `c` program.
2. CSV: comma-separated values
3. `fopen`

```
FILE *fopen(const char *filename, const char *mode);//prototype for fopen
```

```
Some modes:
    "a": append mode: if the file exists, add the content at the end, else,
create the file
    "r": read mode: read the file with the file pointer initially pointing to
the beginning of the file, if it DNE, fails and return \NULL
An example:
```

```
// 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");
    // FILE is a special data type, which is a file
    // The pointer named "file" points to a file
    // The fopen function opens a file named "phonebook.csv" in append mode
    if (!file)
    //if file is NULL, !file will be 1, if(1) will execute the return 1 inside
    {
        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);
    // name and number will be written into where file points to as strings

    // Close file
```

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

```
Another example for copying a file to another file
```

```
// Copies a file

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

typedef uint8_t BYTE;

int main(int argc, char *argv[])
// take in command-line argument, which contains names of the source file and
destination file
{
    FILE *src = fopen(argv[1], "rb");
    // open the source file specified by the first command-line argument
    // in binary read mode
    FILE *dst = fopen(argv[2], "wb");
    // open the destination file specified by the second command-line argument
    // in binary write mode

    BYTE b;
    // the copy process hope to copy byte by byte, which is why we use rb and
wb

    while (fread(&b, sizeof(b), 1, src) != 0)
    // read from file src, read 1 sizeof(b) and stores in b every time
    {
        fwrite(&b, sizeof(b), 1, dst);
        // write 1 sizeof(b) from b into dst file every time
    }

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

Takeaways

1. One thing about array is that the name of the array itself is the address of the first entry of the array, i.e.

```
char s[4];//s=&s[0]
```