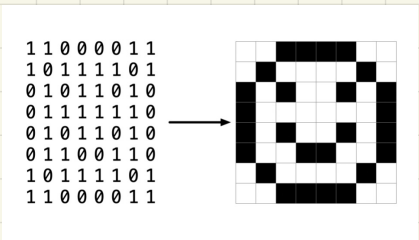
CS50-IV

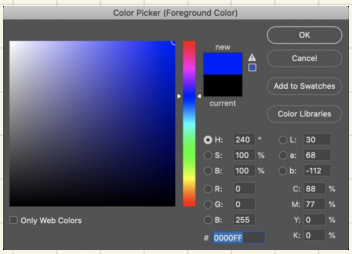
Memory

If we keep zooming into an image, we will see pixels.

We can imagine an image as a map of bits, where zeros represent black and ones represent white.



RGB, or Red, Green, Blue, are numbers that represent the amount of each of this colors.



Hexadecimal

Most western cultures use the decimal system, aka base-10, to represent numeric data.

1	2	3	4	5	6	7	8	9

Computers use the binary system, aka base-2.

1	2

As computer scientists, it's useful to be able to express data the same way the computer does. Trying to parse a huge chain of 0s and 1s can be quite difficult.

Hexadecimal is a system of counting that has 16 counting values.

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

Hexadecimal make this mapping easy because a group of 4 binary digits has 16 different combinations, and each of these combinations map to a single hexadecimal digit.

Hexadecimal is also known as base-16.
When counting in hexadecimal, each column is a power of 16.

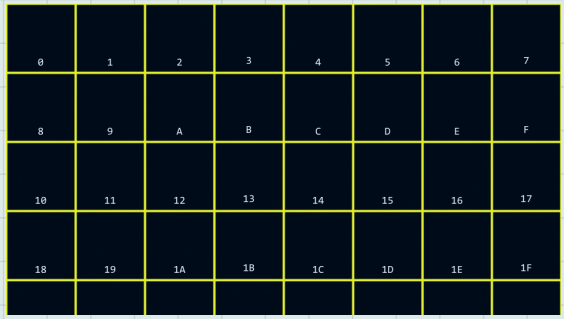
0x	3	7	9
	16^2	16^1	16^0
	256	16	1

The number 255 is represented has FF, because 16 x 15 (or F) is 240. Add 15 more to make 255.
This is the highest number you can count using a two-digit hexadecimal system

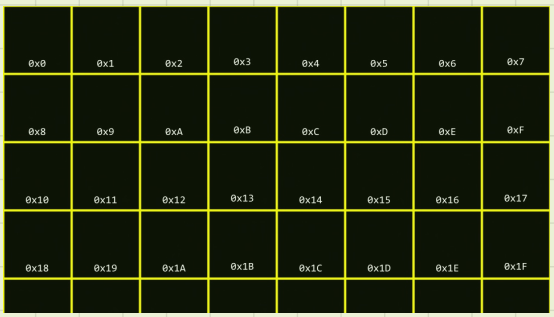
Hexadecimal is useful because it can be represented using fewer digits. Hexadecimal allows us to represent information more succintly.

Addresses

Applying hexadecimal numbering to each blocks of memory, it's possible to visualize these as follows:



We can imagine how there may be confusion regarding wether the 10 block may represent a location in memory or the value 10. Accordingly by convention, all hexadecimal numbers are often represented with the 0x prefix as follows:



The C language has two powerful operators that relate to memory:

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

For `int n = 50;` n returns 50. And &n returs it's address: 0x123

```
int n = 50;
printf("%p\n", &n);
```

→

```
50
0x123
```

%p

%p allows us to view the address of a location in memory.

Pointers

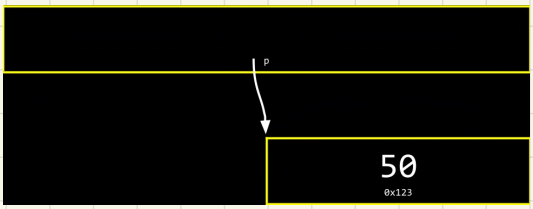
A pointer is a variable that contains the address of some value.
Most succinctly, a pointer is an address in your computer's memory.

```
int n = 50;

int *p = &n;
```

p is a pointer that contains a number that is the address of an integer n.

A pointer is usually stored as an 8-byte value.



We can more accurately visualize a pointer as one address that points to another:

The * is one important part of both the type name and the variable name.

```
int* px, py, pz; ✗
int* pa, *pb, *pc; ✓
```

The dereference operator * returns the value stored in the block of memory pointed by the pointer.

If we try to dereference a pointer whose value is NULL, we get a **segmentation fault**.

Memory

Every file on the computer lives on the disk drive; hard disk drive (HDD) or solid-state drive (SSD).

Disk drives are just storage space; we can't directly work there. Manipulation and use of data can only take place in RAM, so we have to move data there.

Memory is usually a huge array of 8-bit wide bytes.
512 MB, 1 GB, 2GB, 4 GB...

Pointers are just addresses

A Pointer, then, is a data item whose;
Value is a memory address.
Type describes the data located at the memory address.

As such, pointers allow data structures and/or variables to be shared among functions.

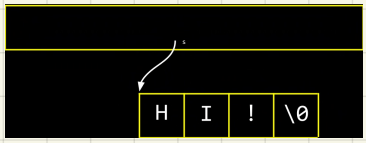
The address extraction operator &, extracts the address of an already existing variable

The simplest pointer available to use in C is the NULL pointer.
This pointer points to nothing. It's possible to check wether a pointer is NULL using the == operator.

Strings

A string is simply an array of characters.

for a string s = "HI!", s is a pointer pointing to the first byte in memory.



Pointer Arithmetic

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

⟺

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

If you attempted to access something at location s+50, the program will likely quit as a safety precaution.

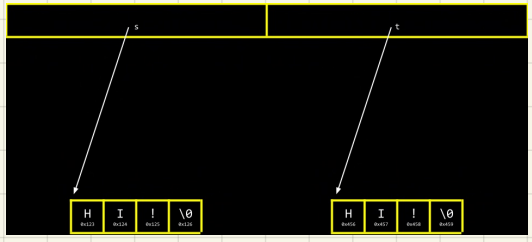
Comparing Strings

A string of characters is simply an array of characters identified by its first byte.

s == t : false

Using the == operator in an attempt to compare strings will attempt to compare the memory locations of the strings instead of the characters therein.

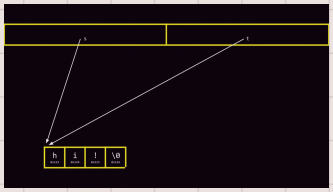
The following pointers point to two separate locations in memory.



Copying

A common need in programming is to copy one string to another.

String s = String t



that s and t are pointing at the same block of memory.
This is not an authentic copy of a string.
Instead, there are two pointers pointing at the same string.

To be able to make an authentic copy of a string,
we need two new building blocks;

Malloc; allows to allocate a block of specific size of memory.

Free; allows to tell the compiler to free up that block of memory you previously allocated.

```
// 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];
}
```

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.

The C language has a built-in function to copy strings named strcpy(char *str1, char *str2)

malloc returns NULL, a special value in memory, in the event that something goes wrong.

Valgrind

Valgrind is a tool that can check to see if there are memory-related issues with a program wherein malloc is used.
Specifically, it checks to see if all the memory allocated has been freed.

make program_name, followed by valgrind ./program_name, will report where memory has been lost as a result of the program.
will return a report from valgrind that

Garbage Values

When asking the compiler for a block of memory, there is no guarantee that this memory will be empty.

It's very possible that this memory was previously utilized by the computer.
Accordingly, you may see junk or garbage values.
This is a result of getting a block of memory but not initializing it.

Swap

A common need in programming is to swap two values.
Naturally, it's hard to swap two variables without a temporary holding space.

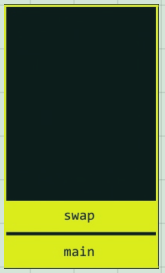
```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

While this code runs, it does not work.
The values do not swap.

When values are passe to functions, only copies are actually provided. (Scope)

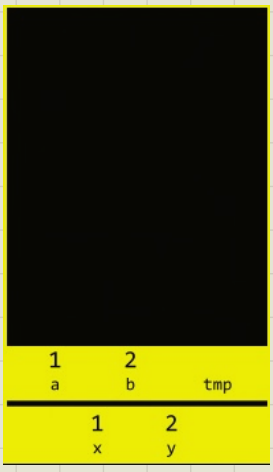
Global variables live in one place in memory.
Various functions are stored in the stack.

The main and swap have two seperate frames.
Therefore, we cannot simply pass the values from one function to another to change them.




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

The variables are not passed by value, but by reference.
The addresses of a and b are provided.



Call Stacks

When calling a function, the system sets aside space in memory for it.
Such chunks of memory are frequently called **stack frames** or **function frames**.

More than one function's stack frame may exist in memory at a given time.

The frames are arranged in a **stack**.
The frame for the most recently called function is always on the top.

When a new function is called, a new frame is **pushed** onto the top of the stack and becomes the active frame.

When a function finishes its work, its frame is **popped off**.
The frame below becomes the new active one.

Overflow

A **heap overflow** is when you overflow the heap,
touching areas of the memory you are not supposed to.
A **stack overflow** is when too many functions are called,
overflowing the amount of memory available.
Both of these are considered **buffer overflows**.

scanf

scanf is a built-in function that can **get user input**.

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

The value of x is stored at the location of x.

To read a string we need to pre-allocate the amount of memory required.
An input larger would create an error.

```
char s[4];
printf("s: ");
scanf("%s", s);
```

File Pointers

The ability to read data from and write data to files is the primary means of storing persistent data.

The abstraction of files that C provides is implemented in a data structure known as FILE. Almost universally, when working with files, pointers to them are used; FILE*

The file manipulation functions all live in stdio.h.

- All of them accept FILE* as one of their parameters, except for the function fopen(), which is used to get a file pointer in the first place.

Some of the most common file input/output (1/0) functions that we'll be working with are:

fopen()	fclose()	fgetc()	fputc()	fread()	fwrite()

Function	Description
fgets()	Reads a full string from a file.
fputs()	Writes a full string to a file.
fprintf()	Writes a formatted string to a file.
fseek()	Allows you rewind or fast-forward within a file.
ftell()	Tells you at what (byte) position you are at within a file.
feof()	Tells you whether you've read to the end of a file.
ferror()	Indicates whether an error has occurred in working with a file.

fopen()

- Opens a file and returns a file pointer to it.
- Always check the return value to make sure you don't get back NULL

```
FILE* ptr = fopen(<filename>, <operation>);
```

operation: r=read, w=write, a=append

fclose()

- Close the file pointed by the given file pointer.

```
fclose(<file pointer>);
```

fgetc()

- Reads and return the next character from the file pointed to.
- Note: the operation of the file pointer passed in as a parameter must be "r" for read, or you will suffer an error.

```
char ch = fgetc(<file pointer>);
```

EOF: End Of File

fputc()

- Writes or appends the specified character to the pointed file.
- Note: the operation of the file pointer passed in as a parameter must be "w" for write or "a" for append, or you will suffer an error.

```
fputc(<character>,<file pointer>);
```

fread()

- Reads <qty> units of size <size> from the file pointed to and stores them in memory in a buffer (usually an array) pointed to by <buffer>.
- Note: the operation of the file pointer passed in as a parameter must be "r" for read, or you will suffer an error.

```
fputc(<buffer>,<size>,<qty>,<file pointer>);
```

fwrite()

- Writes <qty> units of size <size> from the file pointed to by reading them from a buffer (usually an array) pointed to by <buffer>.
- Note: the operation of the file pointer passed in as a parameter must be "w" for write or "a" for append, or you will suffer an error.

```
fwrite(<buffer>,<size>,<qty>,<file pointer>);
```

Custom Types

The C keyboard `typedef` provides a way to create a shorthand or rewritten name for data types.

`typedef <old name> <new name>`

The basic idea is to first define a type in the normal way, then alias it to something else.

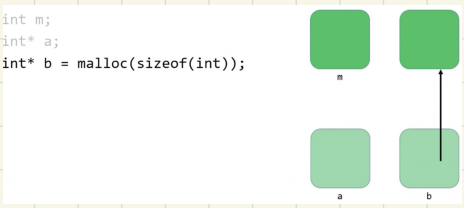
```
typedef struct car
{
  ...
} car_t;
```

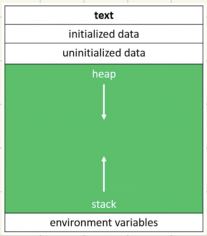
We can typedef a struct car and rename it car_t. So instead of defining a struct car card, we can use car card.

Dynamic Memory Allocation

Pointers can be used to get access to a block of dynamically allocated memory at runtime.

Dynamically allocated memory comes from the heap.





We get this dynamically-allocated memory by making a call to the C standard library function `malloc()`, passing as its parameter the number of bytes requested.
If it can obtain memory, malloc will return a pointer.
Otherwise, it will return NULL.

Dynamically-allocated memory is not automatically returned to the system for later use when the function in which it's created finishes execution.

Failing to return memory back to the system when you're finished with it results in a memory leak which can compromise your system's performance.

Common memory errors

Failing to free every block of memory which we've malloc'd.

Failing to fclose every file we've fopened.

Using more memory than we've allocated.