What is an operating system?

When any user is working on a computer, the operating system is the software that's controlling everything that's happening, in real-time.

Volatile vs non-volatile memory

In order to understand where the operating system is stored on the device, you need to understand the difference between the different kinds of memory, RAM vs hard disk.

Simply put, RAM, (which stands for Random Access Memory) is the temporary (volatile) storage on your computer, while the hard disk is the permanent (non-volatile) memory. Every time you turn off your computer, the RAM will be erased, while the hard disk or hard drive data (HDD will be kept for the long-term.

It's much faster to collect data from RAM, which is why the Central Processing Unit (CPU) relies on RAM to get the information it needs. The CPU will be what performs actions, sends and receives data, and basically controls everything that happens while you are using the machine. Ideally, we would store everything in RAM!

However, the more full your RAM is, the longer it will take to draw information, and there is a finite amount of space in this local memory. That's why the most common IT advice one can get is to give their machines a quick reboot to see if that speeds up processing time, and why we keep larger files and applications on the hard drive.

As RAM is wiped every time the machine shuts off, and you wouldn't want to lose your operating system every time you left your desk for the evening, the answer to where your operating system resides is...in the hard drive.

Can the operating system be stored on RAM?

The operating system is stored on the hard disk, but to speed up the whole process, the OS is copied into RAM on start-up. This is completed by **BIOS** (Basic Input Output System).
Here's how it works: BIOS is in its element for the start-up sequence of your computer, and then once you're all booted up, it actually does very little. Stored on the read-only memory of the computer, (often known as **ROM**) it controls all of your computer hardware, including the hard drive. That means its responsible for loading the operating system from the hard drive when you start up your machine.

As we said, the BIOS is responsible for loading the operating system, but remember the big problem? **Grabbing information from the hard disk is a slow way to work**. To combat this, and speed up the whole start up process, **the BIOS copies the whole operating system to RAM**. Now, the CPU doesn't have to load information from the hard disk (slow), as it can get all the information it needs from RAM (super speedy).

While users are utilising the operating system, all of this information is leveraged from the copy - the one that is stored in RAM. However, the "real" version of the operating system remains on the hard drive, and stays in non-volatile memory that will not be deleted or changed.

Where is Windows 10 stored?

If you're looking for the system files of your Windows OS, you'll probably find them in *C:\Windows*, usually in specific subfolders, like */System32* and */SysWOW64*. Some files may be hidden in your C drive.

TERMINOLOGY EXPLAINED:

✦ **Volatile memory** (or volatile storage) is a type of computer memory that **cannot retain** stored information after power is removed, e.g. random access memory - RAM.

✦ **Non-volatile memory** (or non-volatile storage) is a type of computer memory that **can retain** stored information even after power is removed, e.g. read-only memory - ROM, disk storage, hard disk drives, and generally most types of computer data storage devices.

✦ **ROM** (read-only memory) is a type of non-volatile memory used in electronic devices, and refers to memory that is hardwired. This means data stored in ROM cannot be electronically modified after the manufacture of the memory device. Corrections of errors, or updates to the software, require new devices to me manufactured and to replace the installed device.

➡ Non-volatile computer memory storage media that can be electrically erased and re-programmed do exist. Examples are:

1. **Erasable programmable read-only memory** (EPROM): they have to be erased completely before they can be re-written.

2. **Electrically erasable programmable read-only memory** (EEPROM): memory can be erased, written, or read independently or in blocks.

3. **Flash memory**: a type of floating-gate MOSFET memory, based on EEPROM technology. There are two types of flash memory: NOR flash and NAND flash. Flash memory has fast read access time, but is not as fast as static RAM or ROM.

All of the above fall under the category of floating-gate ROM semiconductor memory. This means they can be erased and re-programmed, but have relatively slow speeds.

ROM is useful for storing software that is rarely changed during the life of the system, a.k.a. firmware (e.g. BIOS, or even complete operating systems for less complicated devices).

The term “ROM” is sometimes used to mean a ROM device containing specific software, or a file with software to be stored in EEPROM or Flash memory.

✦ **BIOS** (Basic Input/Output System) is a firmware used to provide runtime services for operating systems and programs, and to perform hardware initialisations during the booting (power-on startup) process. Originally, BIOS firmware was stored in a ROM chip on the PC motherboard. In later computer systems, the BIOS contents are stored on Flash memory so it can be re-written without removing the chip from the motherboard.

Overview

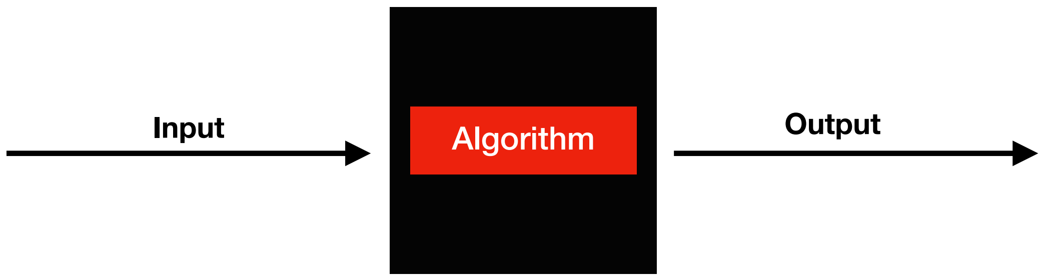
Non-volatile memory is typically used for the task of secondary storage or long-term persistent storage. The most widely used form or primary storage today is a volatile form or random access memory (RAM), meaning that when the computer is shut down, anything in RAM is lost.

Most forms of non-volatile memory have limitations that make them unsuitable for use as primary storage. Typically, non-volatile memory costs more, provides lower performance, or has a limited lifetime compared to volatile RAM.

Non-volatile data storage can be categorised into electrically addressed systems (read-only memory - ROM), and mechanically addressed systems (hard disks, optical disks etc). Generally speaking, electrically addressed systems are expensive and have limited capacity, but are fast, whereas mechanically addressed systems cost less per bit, but are slower.

Encoding information

Computer science is all about **problem solving**: we get some input data, we process them based on some logic, and we get the desired output.



But, computers are not smart; In fact, they work on a binary basis: they can only understand if a circuit is closed (electric current flows - 1) or not (electric current does not flow - 0). How can we make them understand complicated concepts allowing them to operate similar to a human? For example, a human will typically count in the decimal system using intuitively their finger, each one of them representing a single unit. How can computers do the same using only 0's and 1's? How can a computer count past 1?

First we need to decide in advance **how to represent** these input and outputs we discussed above. Let's see how high we can count using some number of binary digits (a.k.a. bits):

Binary representation	Decimal representation
0	0
1	1

1-bit range

Binary representation	Decimal representation
00	0
01	1
10	2
11	3

2-bits range

Binary representation	Decimal representation
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

3-bits range

The more bits I have available (memory to store these digits), the higher I can count, i.e. represent more information.

One may wonder how did we come up with this particular pattern to represent numbers in the binary system. The truth is that it's the same set of rules we use for our familiar decimal system. Let's take a look at number "123" as an example; each "column" represent something: hundreds, tens etc.

…	10^3	10^2	10^1	10^0
…	#	#	#	#

Example:

…	10^3	10^2	10^1	10^0
…	0	1	2	3

So, the number 123 can be written as:

$$(123)_{10} = (0 \times 10^3) + (1 \times 10^2) + (2 \times 10^1) + (3 \times 10^0)$$

Similar, in the binary system:

…	2^3	2^2	2^1	2^0
…	#	#	#	#

<u>Example</u>:

…	2^3	2^2	2^1	2^0
…	0	1	0	1

So, the number 5 in the binary system would be written as:

$$(101)_2 = (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = (1 \times 4) + 0 + (1 \times 1) = 1 + 4 = (5)_{10}$$

Practice:

Convert $(53)_{10}$ to an 8-bit binary number.

Nowadays, it's standardised that any number will be represented typically using 8 bytes (= 64 bits), but of course we can use more bytes if deemed necessary.

Representation of letters

Hopefully, by now, we got a better understanding of how representation of numbers is done in computers, and hence how number crunching programs, e.g. MS Excel, operate under the hood. But what about letters? How are these represented using only 0's and 1's?

An obvious solution is to assign each character (including lowercase, uppercase, symbols etc) to a unique number and use the representation of the number in binary to refer to that letter, e.g. 1 is a, 2 is b, … 27 is A, 28 is B etc.

It turns out that people did exactly that: they standardised the English alphabet and symbols creating the **ASCII table.** For example, "A" in ASCII is the decimal number 65 or, equivalently, the number 01000001 in binary. So, if that pattern of 0's and 1's appear the computer would perceive it as the capital letter "A".

But now it seems we created a problem: how do we differentiate between the letter "A" and the decimal representation of the number 65? How does the computer know when to use the number 65 for math or its mapped ASCII value?

The answer is that it's context dependent: In the context of a text editor, like MS Word, it could be meaningful this pattern to be perceived as a letter but in MS Excel, the same pattern of 0's and 1's would be perceived as a number, or in Photoshop as a colour even!
That's why the programmer provides some hints to the computer that tells the compiler to interpret something in a specific way.

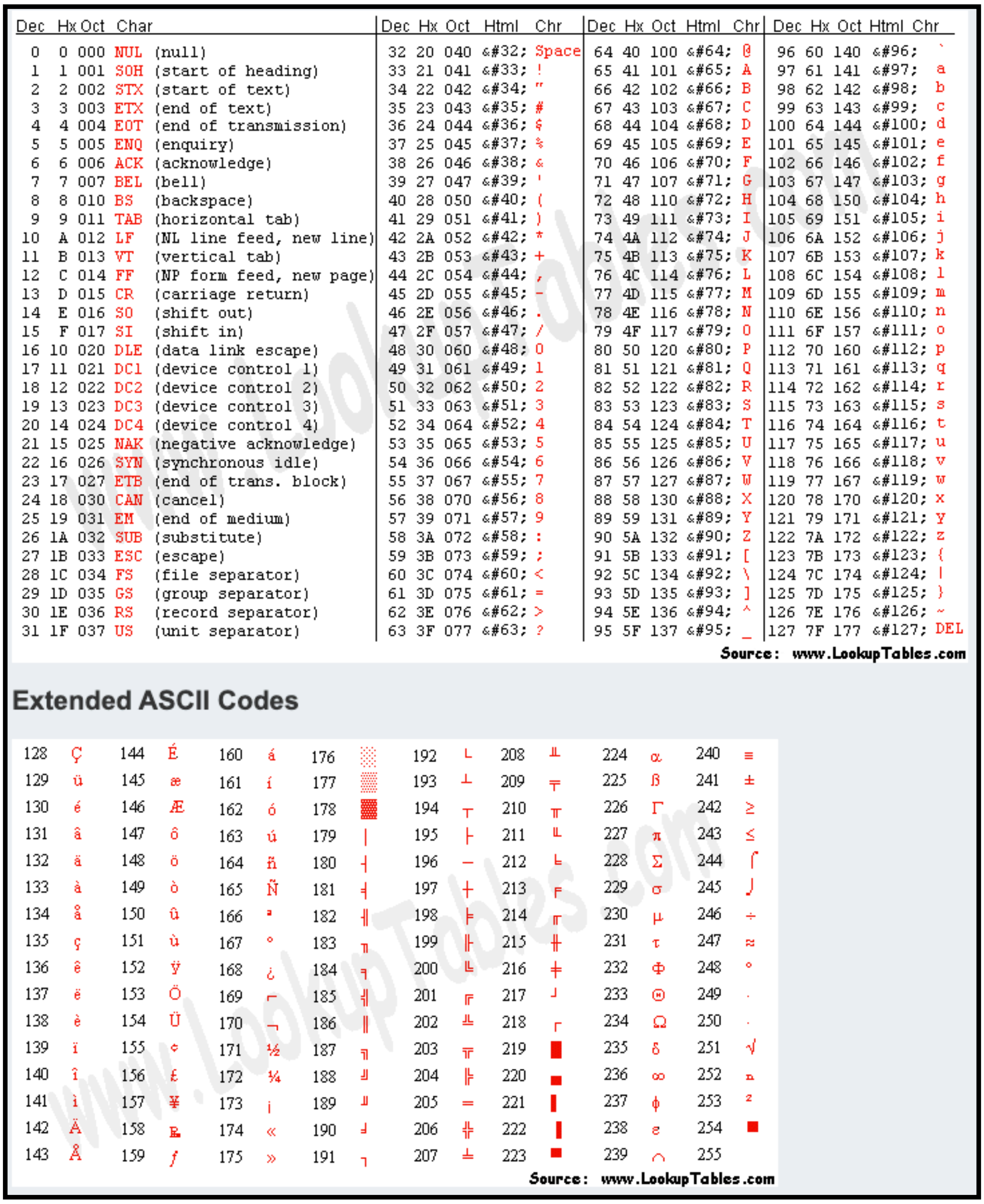
```
int a = 5;
float b = 3.14;
double c = 1234.567;
char d = 'A';
```

So if you receive an SMS that says "Hi!", what is really going on behind the scenes is that it received the sequence of numbers : "72 73 33" that maps to "H I !".

Binary	01001000	01001001	00100001
Decimal	72	73	33
ASCII letter	H	I	!

Notice that even if the numbers 72, 73, 33 do not need 8 bits to be represented in binary, it's standard to come up as packets of 8. That's because the extended ASCII table has 255 mappings (negatives not included) for each character and the number 255 requires exactly 8 bits to be represented in the binary system (i.e. 255 = 11111111). So, this "HI!" Message would be 8 + 8 +8 = 24 bits = 3 bytes.

If we want to create a map for the English language alone, 255 characters (256 if we include zero) will do it but there are many more languages with specific characters, e.g. accents, that need to are represented as well —> we need many more bits!



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Extended ASCII Codes

128	Ç	144	É	160	á	176	░	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	ⁿ
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

Source: www.LookupTables.com

If we use enough bits we can map every possible character in every human language and then some more. For example, emoticons are represented in the same way using a sufficient number of bits. This new standard which is a superset of the ASCII table is called **UNICODE**. Unicode uses 8-bit for backward compatibility, 16-bit or even 32-bits to represent accented letters, emoticons, various symbols etc.

Representation of colours

In order to represent a colour we use a similar approach: we use a scheme, e.g. RGB, and a range within this scheme, e.g. 0 - 255 (8 bits), to "measure" how much red, green, and blue a pixel should have. For example, the decimal numbers 72, 73, 33 were interpreted as the characters "HI!" using the ASCII table as a map. The same pattern, within the RGB scheme, would be translated to a specific colour (R, G, B) = (72, 73, 33) —> yellowish.

So each pixel uses 24 bits (8 for red, 8 for green, and 8 for blue) in some pattern and a program, e.g. Photoshop, interprets this pattern as black, yellow, pink etc.

At the end of the day, a picture, a video, a colour, a music note, or pretty much anything can be represented as a pattern of 1's and 0's using some sort of mapping. From there it's up to a dedicated software, or program, to interpret these patterns correctly.

In modern days, compression and more advanced and complicated math is required to represent the same information in a more elegant way, in order of a file or a program to handle multiple mappings of information. However, the very foundations of how can we encode information are the ones we described above.

Practice:

Write a function called "*upper*" that takes as an argument a character and it checks if this character is lower case. If it is, it returns the upper case version of this character. In any other case it returns the original character.

Test your function by converting the string literal "Hello World!" to all upper case letters.

Pointer fundamentals in C

Physical addresses in computer memory are locations that we do not physically control when they get assigned. It's the computer's operating system that decides what will be stored where.

However, it is possible to store in a physical address if computer memory another memory address. Because this stored variable points to another location within the memory, it is called a **pointer**.

Pointer variables must be declared:

```c
int* a;
float* b;
char* c;
```

The asterisk, **when used in declaration**, tells the compiler that the variable is to be a pointer and the type of data (the value that we get indirectly that the pointer points to).

Reference operator: &

The ampersand indicates that we're getting the address of a variable, e.g. &variableName.

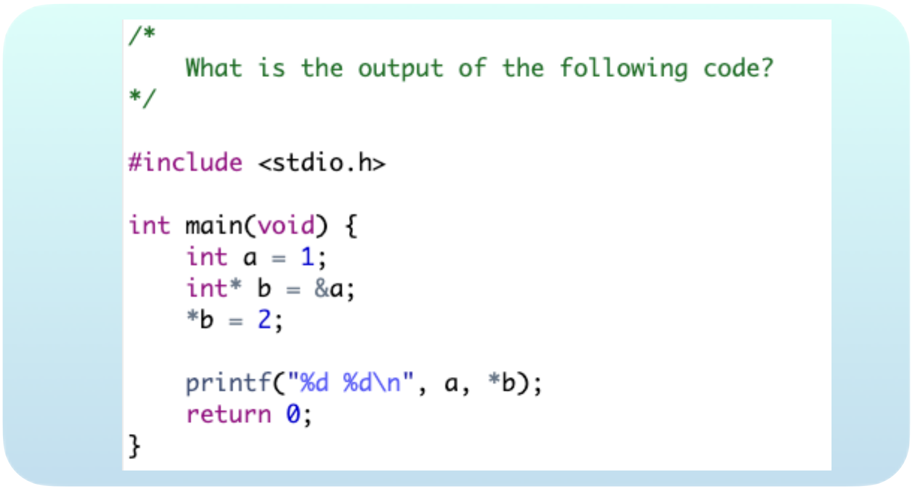
Dereference operator: *

The asterisk has two different meanings:

- When used in the declaration, it tells the compiler that the variable is to be a pointer.

- When *NOT* used in declaration, the asterisk means "get the value that the pointer points to".

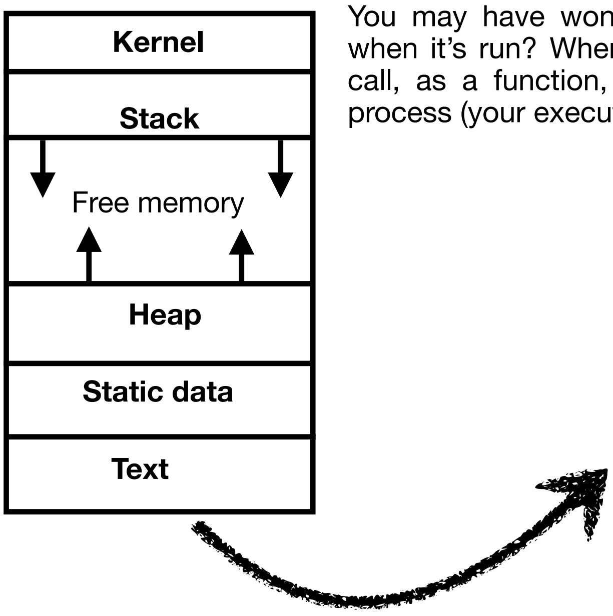
Practice:



```c
/*
    What is the output of the following code?
*/

#include <stdio.h>

int main(void) {
    int a = 1;
    int* b = &a;
    *b = 2;

    printf("%d %d\n", a, *b);
    return 0;
}
```

Memory segments and organisation

Understanding how memory is managed is an essential concept in computing. Here we are going to explore memory management in the context of the C programming language.



You may have wondered: what happens in memory with a program when it's run? When a program is run by the operating system, it will call, as a function, the main method of your code. But your actual process (your executable) will be held in memory in a very specific way.

Block of RAM:

Bottom of memory (small memory address, e.g. 0x000…)

Top of memory (large memory address, e.g. 0xFFF…)

- **Text**: actual code of our program (read-only: cannot be changed during execution) —> machine instructions that we've compiled get loaded in this segment. It is placed in lower memory address so its not overwritten by heap or stack.

- **Static data**: uninitialised and initialised variables get held here, e.g. global variables. The lifetime of these variables is for the entire duration of the program, and the size is fixed/known at compile time. Variables can be changed, but we can also have read-only variables here (constants).

- **Heap (a.k.a. dynamic memory)**: this is where we allocate large things in our memory. Lifetime and size of allocated memory is on the discretion of the programmer. Poor management can lead to heap overflow (memory leak), e.g. when we allocate memory but we are not freeing it after we're done using it. The functions used to allocate memory on heap are: malloc, realloc, calloc, free.

- **Stack**: holds the local variables for each of our functions during function calls. The lifetime of the variables stored here is temporal and the (de)allocation is done automatically. The size of stack memory grows when calling nested functions. Poor management can lead to stack (buffer) overflow, e.g. when using recursion.

- **Kernel**: this segment holds command-line parameters that we pass to our program, environment variables etc.

The static memory

Global variables and the static keyword:

```c
#include <stdio.h>

int globalVariable = 5; // Initialized global globalVariable
                        // This is equivalent to: static int globalVariable = 5;
                        // All global variables are static, so in this case
                        // the static keyword is optional

int main(void) {
    static int a = 10; // Indicate that this local variable should be stored
                       // in static memory
    return 0;
}
```

<u>Example</u>:

```c
#include <stdio.h>

int globalVariable = 5; // Global variable stored by definition in static memory
const double phi = 1.61803398875; // Global read-only variable stored in static

void foo() {
    static int a = 1; // Local variable stored in static memory
    printf("a = %d\n", a);
    a++;
}

int main(void) {
    static const float pi = 3.14; // Read-only local variable stored in static
    printf("pi = %.2f\n", pi);
    printf("phi = %.3lf\n", phi);

    foo(); // First call will print out "a = 1"
    foo(); // Second call will print out "a = 2"
    foo(); // Third call will print out "a = 3"
    foo(); // Fourth call will print out "a = 4"

    return 0;
}
```

Notice that a variable stored in static memory has exactly one copy in memory address. So, each time the foo function is called, it operates on the same memory address, incrementing its content by one.

The stack memory

One of the main advantages of the stack memory is that allocation and deallocation of memory is done automatically. Naturally, this is easier for the programmer. But the same feature can be perceived as a disadvantage as well. That's because lack of memory control is not always what we want. Many times is crucial to be able to directly manage the memory.

Let's see an example of how the stack memory operates differently than the static memory. The code segment below is exactly the same as the example we presented previously in the static memory section. The only difference is that we omit the static keyword, so all local variables are stored in stack memory instead.

```c
#include <stdio.h>

void foo() {
    int a = 1; // Local variable stored in stack memory
    printf("a = %d\n", a);
    a++;
}

int main(void) {

    foo(); // First call will print out "a = 1"
    foo(); // Second call will print out "a = 1"
    foo(); // Third call will print out "a = 1"
    foo(); // Fourth call will print out "a = 1"

    return 0;
}
```

Notice that local variable "a" is automatically created and destroyed with each call of the foo function. In other words, when a function returns, it will automatically deallocate all its local variables that were stored in the stack memory.

We do not have one underlying copy of the variable: we are constantly creating and destroying the variable with each function call.

Understanding how stack memory operates

```c
#include <stdio.h>

void foo() {
    printf("I am the foo function!\n");
}

void foo2() {
    printf("I am the foo2 function!\n");
    foo();
}

void foo3() {
    printf("I am the foo3 function!\n");
    foo2();
}

int main(void) {

    foo3();

    return 0;
}
```

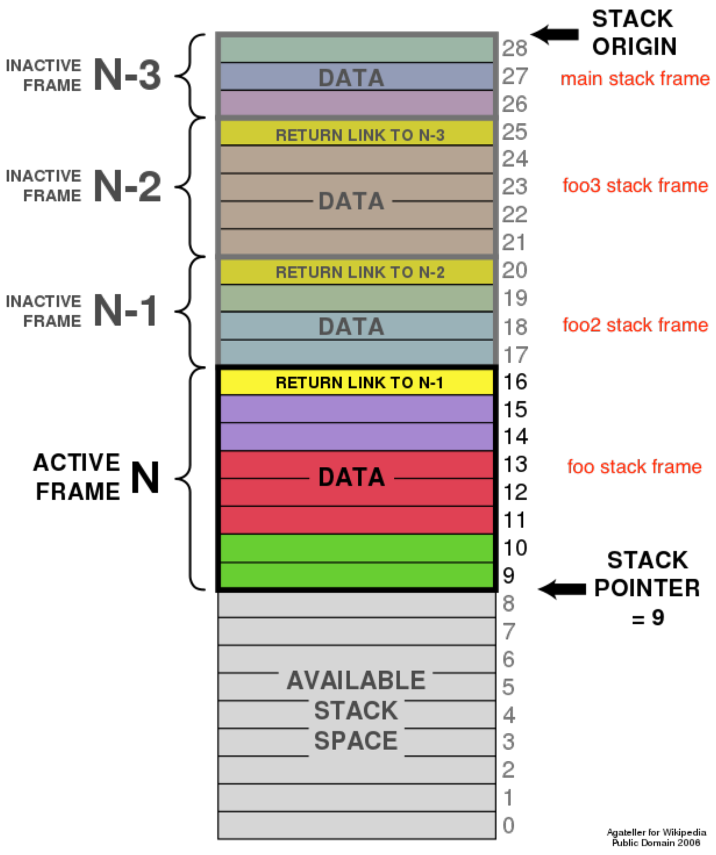
Stacks are organised in a Last-In, First-Out (LIFO) order. That means when something is added to the stack the last thing that was pushed in it will be the first thing that will be able to get out (or popped).

As an example, we see on the left a code segment that is basically a series of nested function calls. When the program is executed, the main function is called. This creates a "**stack frame**" that contains all the local variables and data related to the main function. Since the main function is the first function to be called, the main stack frame is pushed onto the stack memory first.

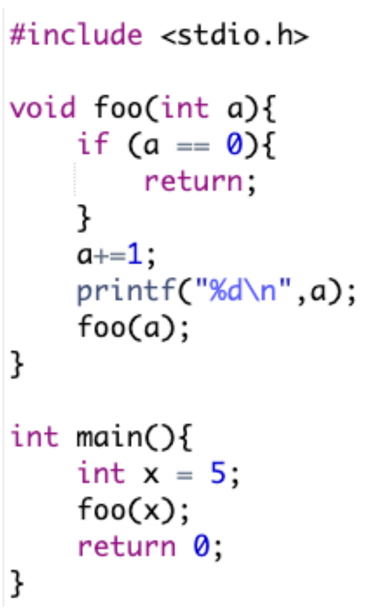
Then the function foo3 is called, and all the local variables and data related to that function is pushed onto the stack next. After that it’s foo2’s turn to be pushed onto the stack, and finally foo.

Since foo is the last function that was pushed onto the stack, the code contained within it is the code that is currently running, hence the stack frame of the foo function is the **active frame**. All other variables that are not in foo’s scope are “buried” being further up the stack and their local variables are not available (inactive frames).

When the foo function returns, the associated stack frame is popped off the stack and that part of memory is freed. Similarly, the stack frames of foo2, foo3, and finally main functions are popped. When the main stack frame is popped the program is done executing.



Take away message: every function call results in a new stack frame in memory containing local variables. When the function returns, the stack frame is popped off the memory and all the data, i.e. local variables, are lost.



```
#include <stdio.h>

void foo(int a){
    if (a == 0){
        return;
    }
    a+=1;
    printf("%d\n",a);
    foo(a);
}

int main(){
    int x = 5;
    foo(x);
    return 0;
}
```

Stack overflow in programming

To understand how a stack overflow error occurs let’s take a look at the code on the left. This is an example of **recursion** where a function calls itself. Recursion is a very powerful and useful tool but needs caution.
The recursive code on the left demonstrates why: by initialising the variable x to a positive integer, the if conditional within the foo function is never evaluated to true. This causes the function to call itself again and again; each call creates a new stack frame and continues until the stack memory is exhausted. Eventually, the program will fail because of stack overflow error. Notice that the code would work fine if we had initialised the x variable to a negative number.

The heap memory

The advantage of heap, or dynamic memory, is the direct control of memory. The obvious disadvantage is that it forces the programmer to handle manually the memory, and explicitly allocate/deallocate memory. It is also slower than stack memory.

Example:

```c
#include <stdio.h>

int main(void) {

    // allocation of memory
    int* p = (int*) malloc(sizeof(int) * 1024);

    // Use that memory for something

    // Deallocate the memory when you're done using it
    free(p);

    return 0;
}
```

Notes:

sizeof() —> gives the number of bytes of its argument.

malloc() —> returns a memory address of a specific size.

Example:

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

int* foo() {
    int* ptr = (int*) malloc(sizeof(int));
    *ptr = 5;
    return ptr;
}

int main(void) {
    int* ptr2 = foo();
    printf("%d\n", *ptr2); // This will output 5
    return 0;
}
```

Notice that the address returned by malloc is available outside the function scope because that space is available until the programmer decides to free it.

So, what happens is when the foo function returns, the ptr local variable is destroyed, but ptr2 is initialised to point to the same location in memory where ptr was pointing to.

Here we didn't explicitly freed the heap memory when we were finished using it. Normally, we should have included a "free(ptr2)" statement right after printing out the desired result. In this particular program will not cause an issues because when a program terminates, all heap memory is automatically deallocated but one should not rely on that!

Heap overflow in programming

Memory leaks caused by heap overflow error occur when the dynamic memory allocation overflows the heap memory layout created by the operating system. The two possible scenarios for heap overflow are:

1. If we continuously allocate memory without deallocating previously allocated memory space.

2. Explicit allocation of large number of variables.

Below we provide two code examples of these two scenarios. ATTENTION: DO NOT RUN THESE CODE SEGMENTS IN YOUR SYSTEM!

Heap overflow: scenario 1

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

int main(void) {
    while (true) {
        // Keep allocating memory and never freeing it
        int* ptr = (int*) malloc(sizeof(int));
    }
    return 0;
}
```

Heap overflow: scenario 2

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

int main(void) {
    int* ptr = (int*) malloc(sizeof(int) * 10000000000000000);
    return 0;
}
```

Practice:

In section "Encoding information" you wrote a function called "upper" that was checking if a character is lower case, in which case it returned the corresponding upper case character.

Re-write the "upper" function but this time it should take as an argument a string and will return a **new** upper case string. **Make use of the heap memory for this purpose**.

Write a function that will append a number to the end of an integer array of a certain size:

- Using only the stack memory
- Using only the heap memory

Functions and Scope

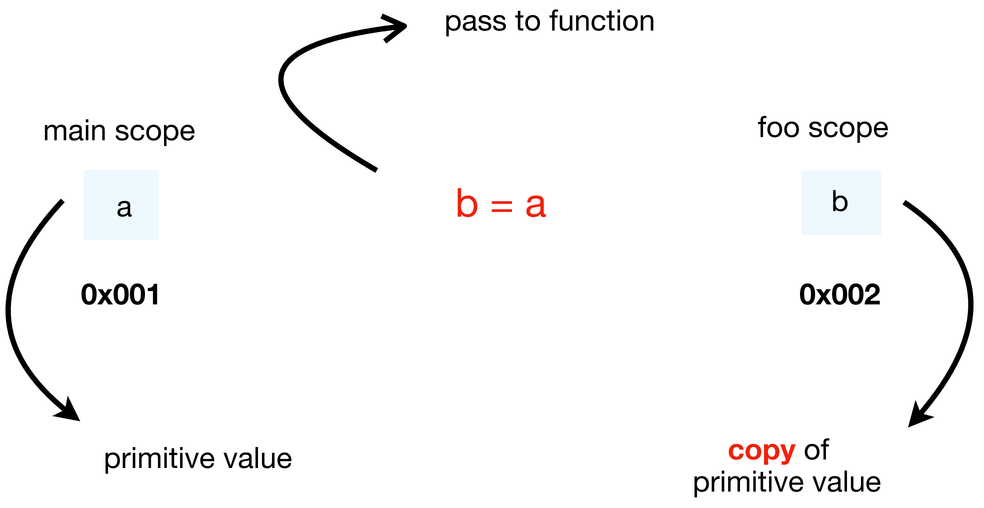
There are two primary ways to send information into functions:

1. Pass by value
2. Pass by pointer (a.k.a. pass by reference)

Pass by Value

Let's see the example below:

```c
void foo(int b) {
    //
}

int main(void) {
    int a = 5;
    foo(a);
    return 0;
}
```

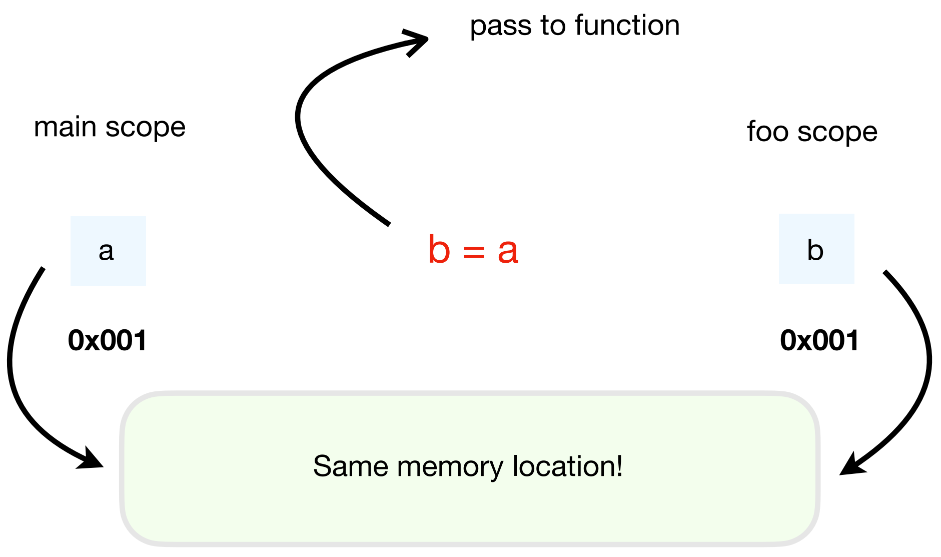


When passing by value it makes a **copy** of the original variable, that is stored in a **different** memory location. The key is that we have two different locations where the value is stored! So, if we change something, or modify in any way the variable "b", the variable "a" will not be affected because it's stored in a different location.

Pass by Pointer

Similarly,

```c
void foo(int* b) {
    //
}

int main(void) {
    int value = 5;
    int* a = &value;
    foo(a);
    return 0;
}
```



The key here is that we have the same memory location! That means if the value of the variable that “b” pointer points to is somehow modified, do does the value pointer “a” points to, because both “a” and “b” point to the same location.

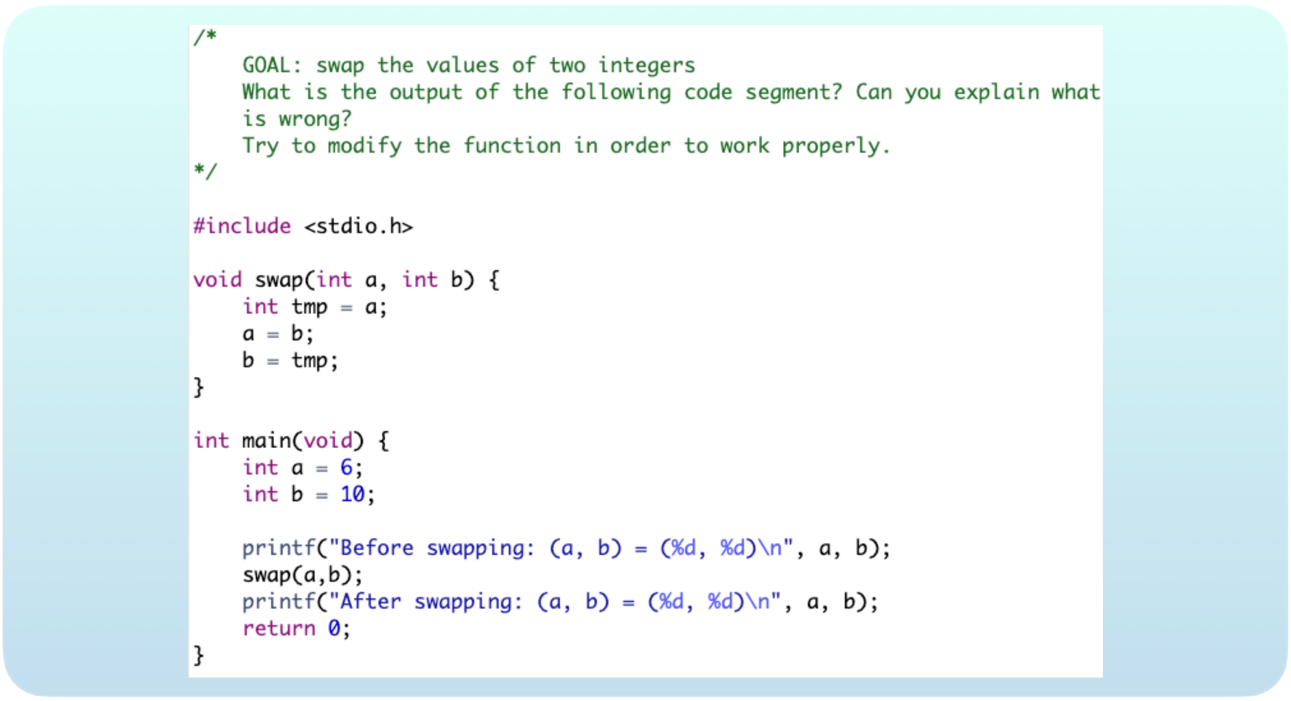
Example:

```
void addOne(int* x) {
    *x = *x + 1; // Dereferencing pointer x --> add 1 to the value ponter x
                 // points to
}

int main(void) {
    int x = 5;
    addOne(&x);
    printf("x is: %d\n", x); // Prints out 6 in spite that x here is in the
                             // main scope and not in addOne scope.
    return 0;
}
```

In the example above, when the function addOne() is called, it operates on the x value (defined in the main scope) that is stored in memory.

Practice:

```
/*
    GOAL: return the address (i.e. a pointer) of the largest integer
    If you try to execute the code below you will get a run-time/compile error.
    Why?
*/

#include <stdio.h>

int *largest(int a, int b) {
    int max = a;
    if (b > a) {
        max = b;
    }
    return &max;
}

int main(void) {
    int a = 1;
    int b = 5;
    int* ptr = largest(a, b);
    printf("%p\n", (int*) ptr);
    return 0;
}
```



```c
/*
    GOAL: swap the values of two integers
    What is the output of the following code segment? Can you explain what
    is wrong?
    Try to modify the function in order to work properly.
*/

#include <stdio.h>

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

int main(void) {
    int a = 6;
    int b = 10;

    printf("Before swapping: (a, b) = (%d, %d)\n", a, b);
    swap(a,b);
    printf("After swapping: (a, b) = (%d, %d)\n", a, b);
    return 0;
}
```

Arrays and Strings in C

SPOILER: Arrays are pointers!

Arrays and strings (a string is nothing more than an array of characters that includes a null terminator, **'\0'**, signifying the end of a string) **are always passed-by-pointer** in a function. This is due to the enormous size arrays can reach, becoming cumbersome to copy (it takes a lot of time) and store a copy of the original array (it takes a lot of space) in order to be passed-by-value.

Pass-by-pointer allows changing the value of a variable inside the function scope —> needs caution!

Arrays as function parameters

Let's take the example of a function that finds the largest integer within an array of numbers:

```
int findMax(const int* array, int size, int key);
```

Key points:

- The "const" modifier forces the array to be read-only inside the function scope. This prevents ambiguous modifications of the array with the function.

- Always pass the *size* for arrays (unless array is a string, in which case is not necessary). The reason is we're just passing a pointer that points to location in memory. So, we know where the array starts in memory, but not where it stops (in strings we do know because of the null terminator).

It is not trivial to make a function that returns an array! That's because **the function is returning an address of a local variable**, i.e. a chunk of memory that is no longer available once the function returns (assuming the returning array was declared locally within the function scope).
Possible solutions to this problem are:

✳ **Change original input**, i.e. the function will not return a new, locally defined, array but it will modify in-place the existing original array that accepted as argument. Notice that this array has a different scope (possibly the main scope, if it's define within the main function).

✳ **Pass in, as arguments, two arrays.** We can define in the main scope a new array (the one we want to store the output of the function) and pass it as an extra argument to the function. In this case, the original array will be read-only (using the const modifier) and the output array will be modified in-place, within the function scope.

✳ **Use heap memory region**.

Remember: When you pass in an array as an argument to a function, you pass in the **address** of the **first** element of the array!

Q: What happens if you try to iterate over the end of an array?

A: Sometimes it will throw an error of the sort "Error: Out of Bounds". But sometimes it will overwrite other data that happen to be stored in close proximity to the array, without throwing an error! The latter is known as **(stack) buffer overflow** and is a common way to attack systems and take control of the process (see buffer overflow attack).

Structures and Enumerations

✳ **Structures**: essentially they're custom data types, called "composite" or "aggregate" data types, because are built to aggregate in a whole bunch of other data types. In the example below, we define a new data type called "Person". A person here is defined by its first and last name, age, and gender.

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

typedef struct Person{
    char firstName[255];
    char lastName[255];
    int age;
    char gender[255];
}Person;

int main(void){
    Person testPerson; // Declare a variable testPerson of data type Person
    // Initialize testPerson
    strcpy(testPerson.firstName, "Joe");
    strcpy(testPerson.lastName, "Doe");
    strcpy(testPerson.gender, "Male");
    testPerson.age = 28;

    // Print out data
    printf("Name: %s %s\nAge: %d\nGender: %s\n", testPerson.firstName, \
    testPerson.lastName, testPerson.age, testPerson.gender);
    return 0;
}
```

✳ **Enumerations**: they are used to restrict the values a variable can take. A common example is the following:

```c
#include <stdio.h>

typedef enum Day{
    Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
}Day;

int main(void){
    Day currentDay = Tuesday;

    // Print out data
    printf("Current day: %d\n", currentDay);
    return 0;
}
```

Notice that the variable "currentDay" can only take a value that is defined within the enumeration. The output of this code will be: "Current day: 1". That's because, the first element of the enumeration is equal to 0, the second element is equal to 1 etc, hence the elements are enumerated (as the name suggests).