**FILES**

There are 2 types of files:

* Text files : Stores characters. For example test.c file is a text file.
* Binary files

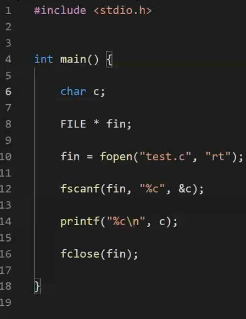
With command “more”, you can read text files.

For example, you can read your .c file with “more test.c”

A file that I have can be accessed by using a set of functions that C provides us mostly in stdio.h.

Why we need files?

* Store our results.
* Read something from them.

C says if you want to store something to file or read from the file or do both, you need a special data type. You need a variable to refer to that type so that C can actually tell “Oh okay, you are working on a file.”. Then that data type can be used for processing those files.

For this thing, C uses “FILE“.

fin is actually a FILE pointer.

Think that file is a cabinet. We trust C if we want to open that cabinet and if we want to open where the cabinet is. We have key on our hand which is our pointer (fin). We say to compiler that “Here is the key, I want to get that file.”

Key in C is actually the name of the file as OS knows it (test.c). C says “I am gonna talk with OS. Give me the name of the file, I will actually get the key from the OS and return that key in the your pointer, fin.”. So fin is key to access to file.

Why do I open a cabinet (file)?

* Take something
* Put something

C doesn’t ask you to give it 1 parameter (just a name) for opening a file, it also asks what are you gonna do with that file.

C wants to know are you opening for:

* Reading, get the content or
* Writing to file

Also C asks what kind of file is this, text or binary? This information is gonna be given with string that has 2 letters in it in 10th line above which is “rt”.

First letter - - - > What purpose are you opening that file for? Reading (r), Writing (w), Updating (a)

Second letter - - - > Indicates if the file is text file (t), or binary file (b).

Now in line 11, I opened the file and I have the key. Key is fin.

You can read from the file with fscanf with your key. Rest of the arguments are same with normal scanf.

Compiler says “I will deal with the OS for you. I will take the key and tell to OS that you want to open this file. Give me the key.”. But OS is very protective. OS says if I give you the key of this file, no one can touch it. File cannot be touched by 2 people at the same time because they can make mess. One of them might be reading from it, one of them might be writing to it and so things can be crashed. Prevent this type of crashes is the thing that OS must prevent. C says “Don’t worry, we are OK. We will keep this key and you can hold it and only the person or the program that has the key can access it.”

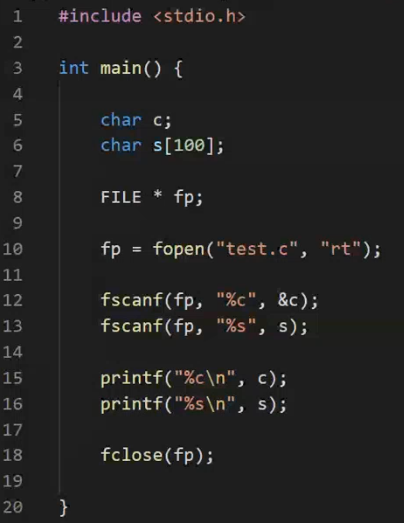
OS also says, “If you are done with this key, return it to me. If you don’t, I will assume that you have the key. I will keep it there and I will not let anybody else to touch that file anymore other than that key.”. So if you open a file with fopen, you need to close it with fclose. Otherwise, OS may never know that key has been returned. With fclose, you give the key back to OS.

If you failed to return the key to the OS, OS can indefinetely hold access to that file and this happens from time to time.

Reading usually is not a big deal. Compiler sometimes handles these types of things and it says “Hey this guy forgot to give the key back, but I can tell to OS that it’s done.”

BUT you have to close every file you open.

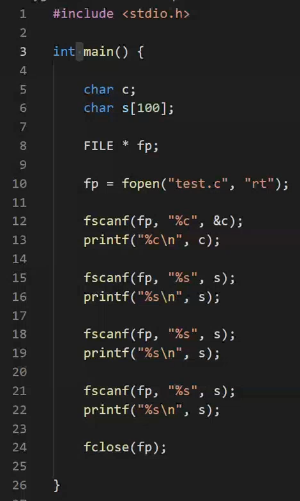
What will be printed?

- - -> #

include

When fscanf reads something, it will read what’s left afterwards when it runs next time.

What will be printed?

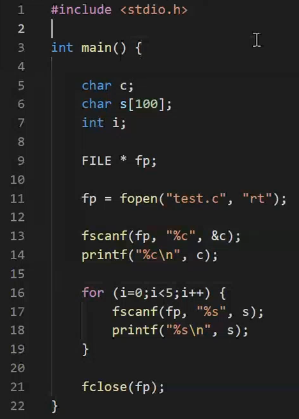
- - - > #

include

<stdio.h>

int

What will be printed?

- - - > #

include

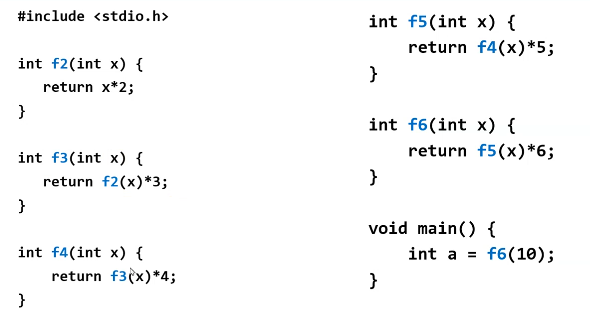
<stdio.h>

int

main()

{

**RECURSION**



Compiler doesn’t forget what’s happening in f6 when it executes the f5. It knows that x is 10.

Callee’s returning point is always known.

IF WE DO LIKE THIS:

int f3(int x){

return f3(x) \* 3;

}

void main(){

int a = f3(10);

}

So every time calling f3,

* reserve memory for x - - - > 4 bytes
* save the return address (where to return) - - - > 8 bytes (for instance)

i need 12 bytes of space. - -> allocation is done from stack part of the memory allocated for the program

f3 will be executed forever.

So there is a limited amount of call that you can make.

When you go further than you should go, there will be an error : stack overflow. You are trying to allocate more memory than your stack allows you to do.

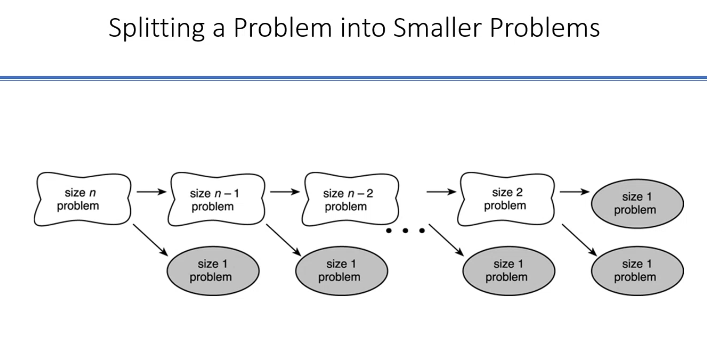
ALL OUR LOCAL VARIABLES, RETURN ADDRESSES FOR FUNCTIONS ARE STORED IN STACK.

Stack is limited (a few hundred megabytes). It is determined by the OS.

We should make a condition for stopping in the function.

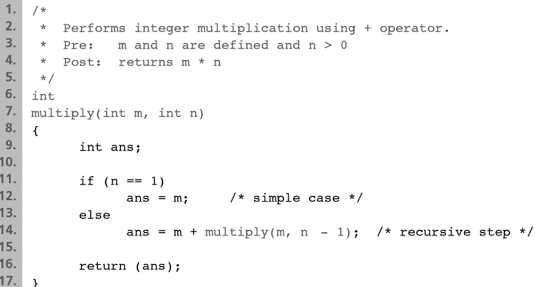
FUNCTIONS CALLS ALWAYS HAPPEN FIRST.

Recursion is more like a fern than tree.



1. First you should think “Can I simplify this problem?”.
2. If I knew the solution for smaller size problem, I can solve the problem.
3. If splitting problem happens in the same way with size n and size n-1 and size n-2 and …. you can use recursion.

m \* n

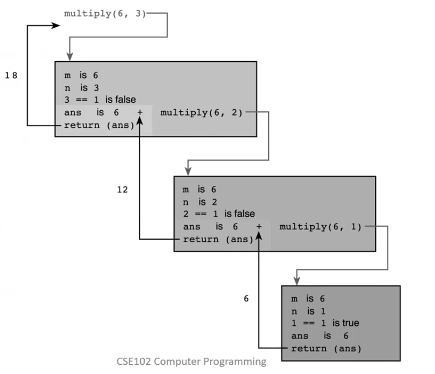


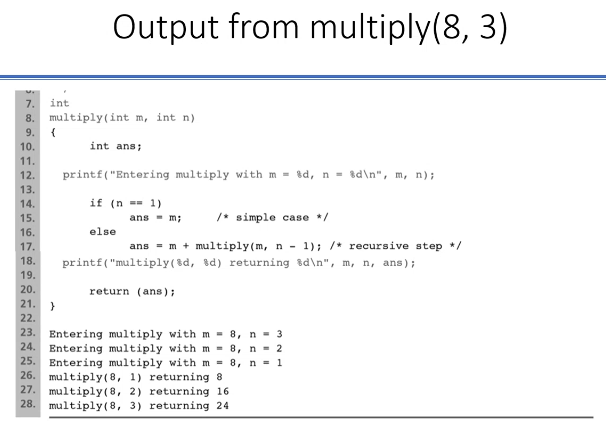
If I know m \* (n - 1), I can add m to it and I have the solution.

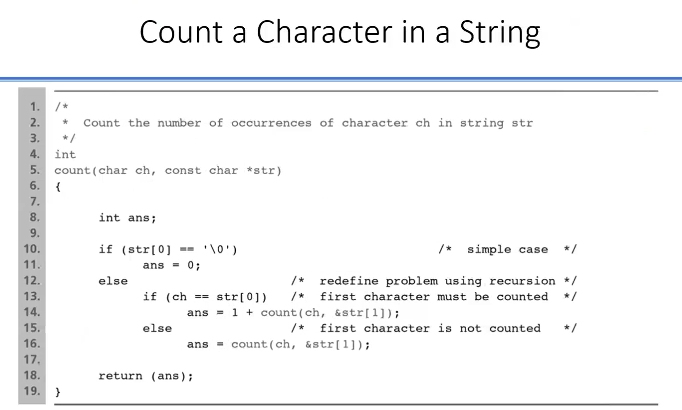
m , if n= 1

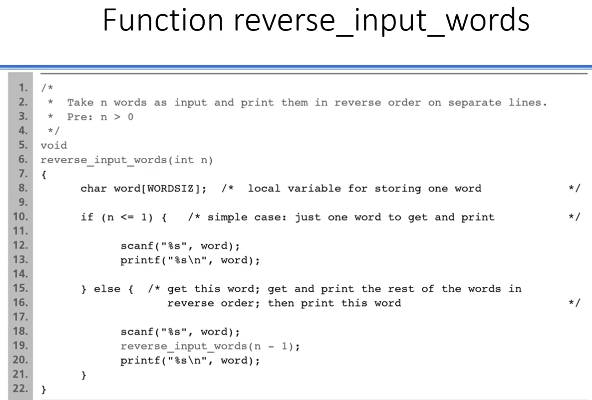
f(m, n) =

m + f(m, n-1) , if n > 1









$ ./a.out

word1

word2

word3 user input

…

wordn

wordn

wordn-1

… my output

word2

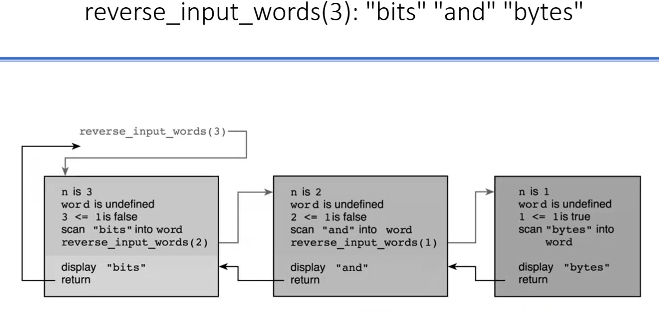
word1

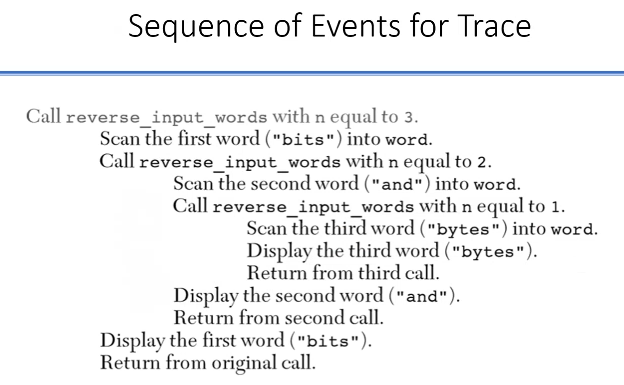
LIFO - - - > LAST IN FIRST OUT 🡪 Then recursion is for you

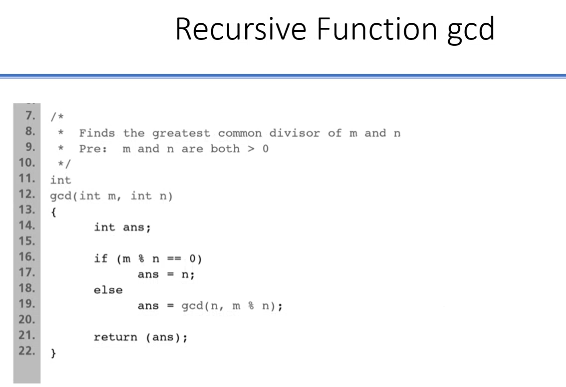
Recursions can be less efficient in terms of run-time efficiency. It may running slower. Because you are storing address (where to return) everytime you call a function.

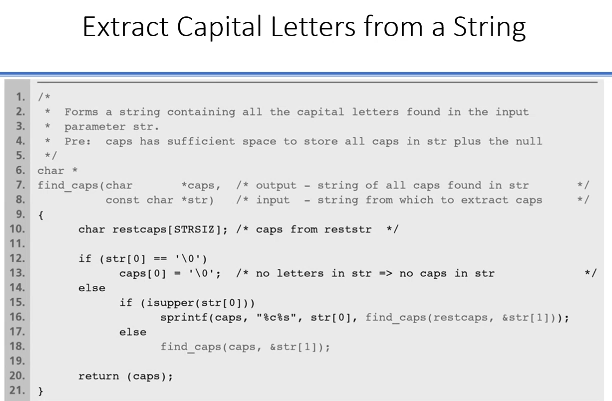
For memory and time, recursive is not that good.

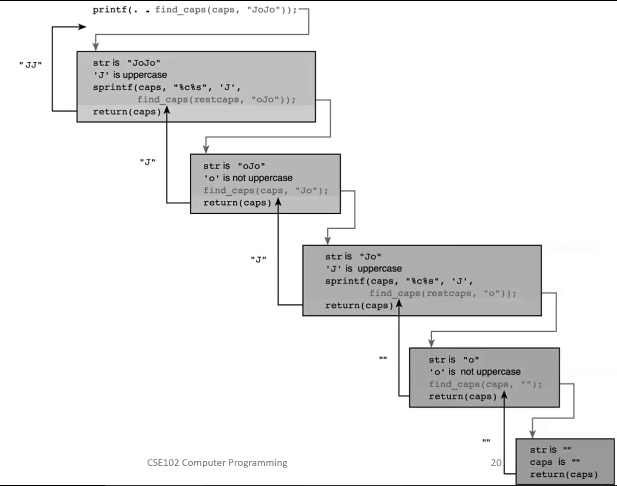
In terms of simplicity, recursive is good.

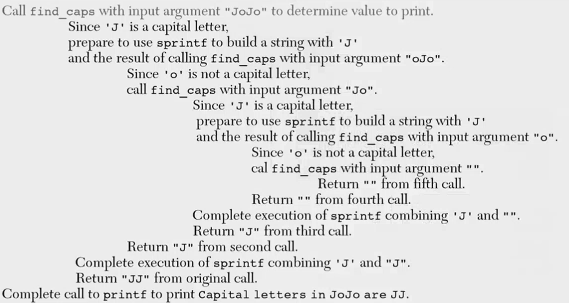


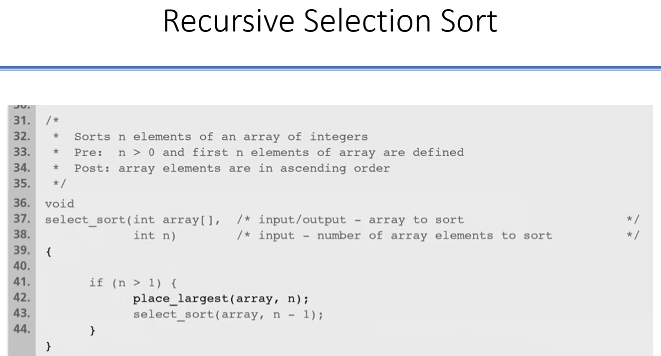


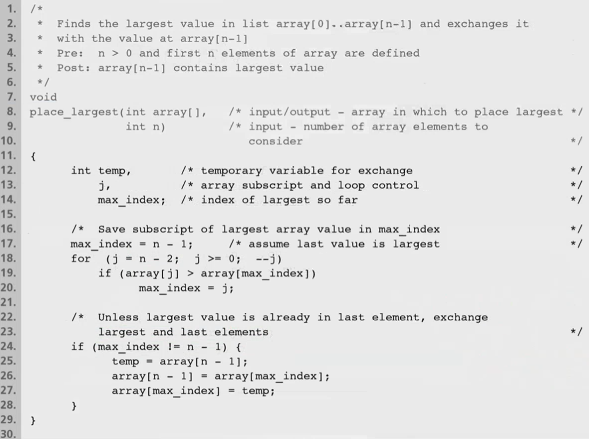


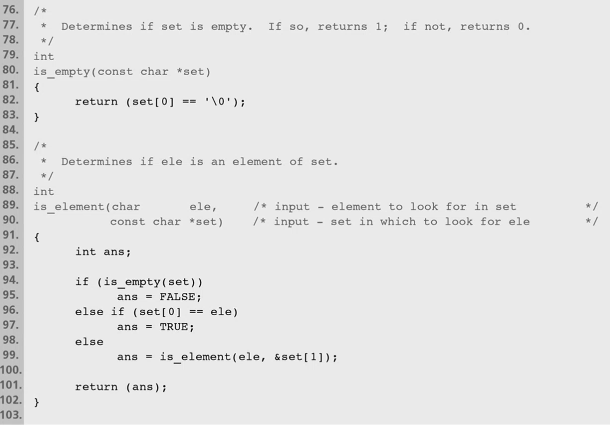


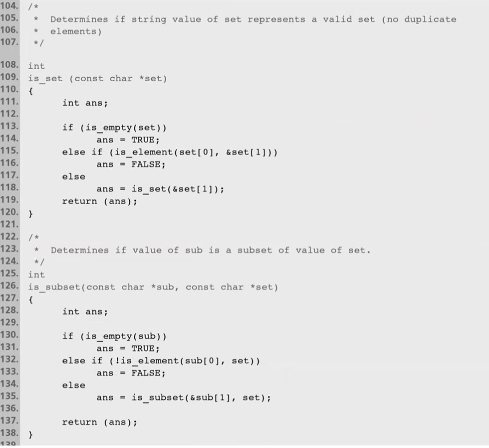












int is\_substring(char \*s1, char \* s2){

int ans = 0;

if (strlen(s1) > strlen(s2))

ans = 0;

else if (strncmp(s1, s2, strlen(s1)) == 0)

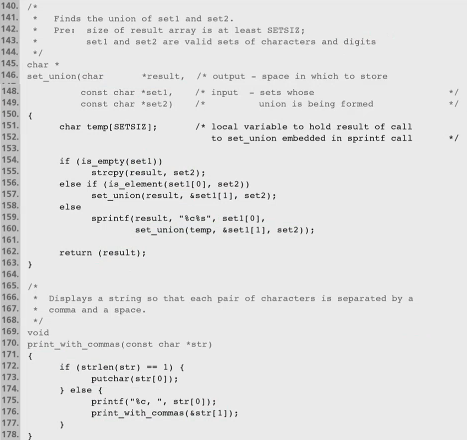
ans = 1;

else

ans = subs(s1, &s2[1]);

return ans;

}



**If I write “int a[5];” how much memory do I use?**

|  |
| --- |
| … |
| a[4] |
| a[3] |
| a[2] |
| a[1] |
| a[0] |
| a |
| … |

I will have 4 (byte number for 1 integer) x 5 = 20 bytes + 8 bytes for address pointer = 28 bytes.

What is happening to a?

a is a pointer. It is 8 bytes for a 64 bit machine. So a variable will hold 8 bytes, an address.

During the assembly code, variables will be converted to values. They kept in the memory.

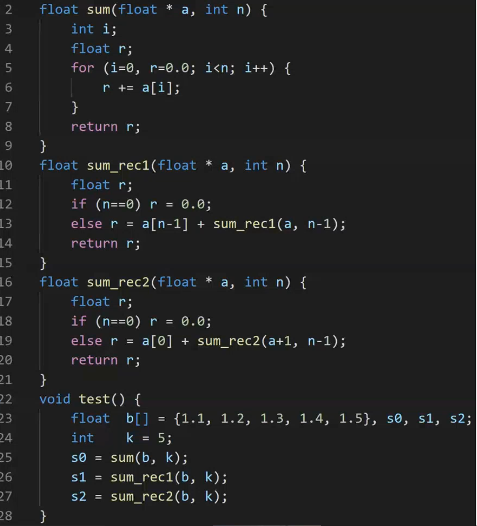
char s[] = “This is the only string”; - - - > s has a value that points to T. s has address of T.

Pointer is variable that holds address.

If I want to get “is the” what should I do?:

&s[5] - - -> treat this as a character array and give the size - - - > (&s[5], 6)

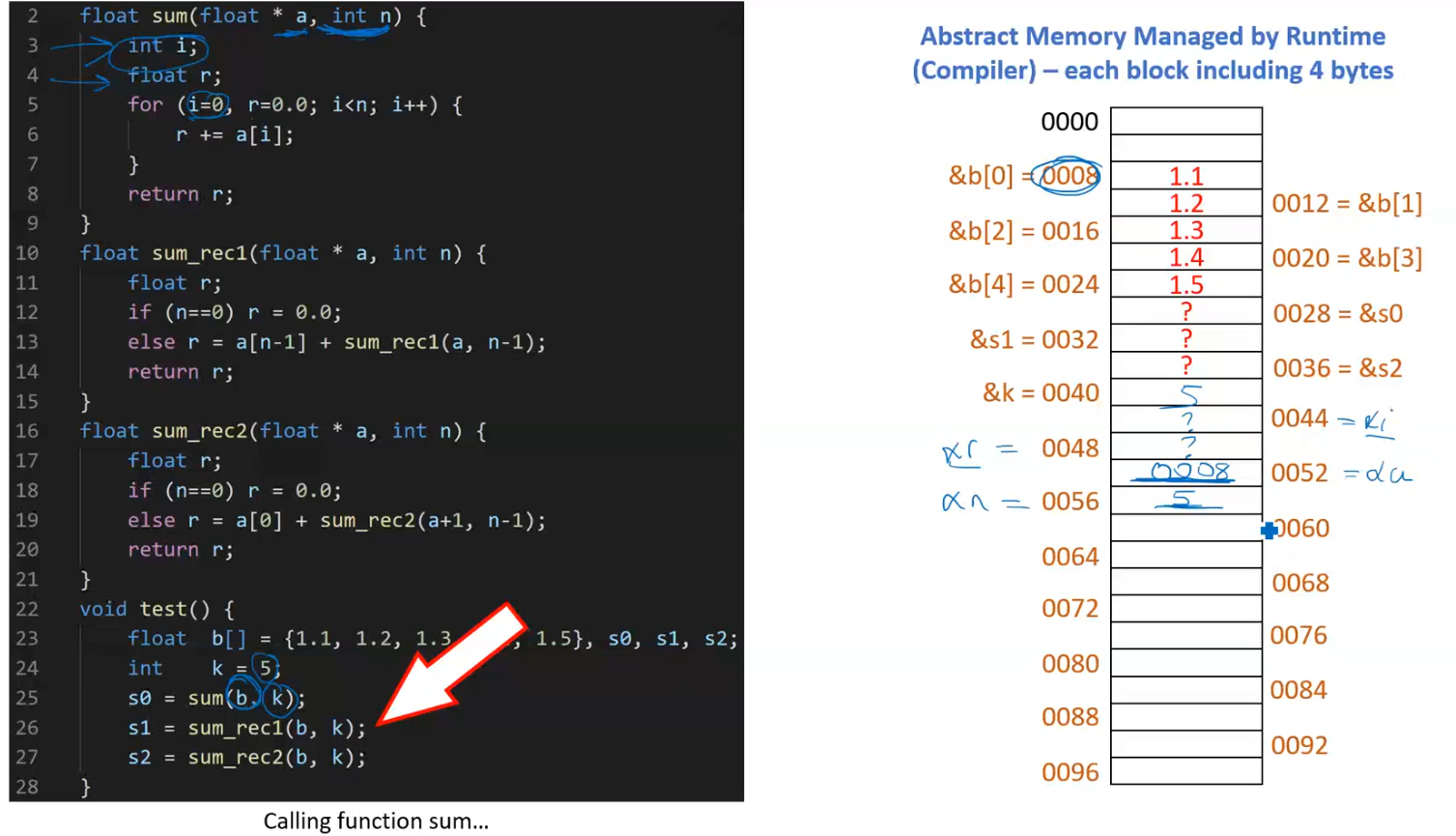
Or you can send beginning and ending - - - > (&s[5], &s[10])



In 19th line, you can write &a[1] instead of a+1.

a+1 - - -> This increments the pointer by 1 entry long. If pointer a is pointing to a character increment should be 1 byte (+1), if pointing to a floating point number increment should be 4 bytes, if pointing to a double number increment should be 8 bytes.

3 functions do the same thing. rec : recursion

a is floating pointer. It is pointing to an array. Since it is pointing to an array, I can index it.

We access to b indirectly in function sum.

**TOWERS OF HANOI**

