***DYNAMIC DATA STRUCTURES***

We use variables to manipulate memory. This is an abstraction that compiler allows us to use.

int a; char abc[100]; complex c[100]; 🡪 This requires the knowledge of all the data/memory needs at compile time, while I am writing my code. So I need to know how much data I am gonna manipulate.

This is a very big restriction. Because in some cases memory need that we have depends on what the user gives us to process.

Dynamic data mechanism is had by C as many other programming languages.

I don’t need to give size at programming time. At run-time, depending on the what user is doing, I can increase or decrease the data I need.

Our data is always managed under variables. This means that I have to have some way of keeping track of dynamic data with my variables.

* Dynamic data structures: expands and contracts as the program executes
  + Decision on space is made during execution.
  + Array: decision is made beforehand (at compile time)
    - Partially filled array is possible but still maximum size needs to be decided before compilation
  + Required dynamic memory allocation
    - Allocate space as necessary during execution
* Linked list: linear sequence of nodes
  + Nodes: a structure that points to another structure
  + Can be used to form lists, stacks, queue

Pointers

* Used extensively for dynamic data structures.
* Pointer review:
  + Reference / Indirect access
* Pointer is a type that has a value (unsigned (long) int for address and also type of the data stored at that address.).
* Value is between 0 and MAX (2^32 for 32 bit system, 2^64 for 64 bit system). MAX is limited by OS which is limited by the hardware (CPU & RAM).

We have also operations that we can do on pointers.

Cell length is given as the number of bytes that is required to hold that value.

address(pointer) + 3 🡪 going three cells. 24x3 bytes for a structure, 4x3 bytes for integer.

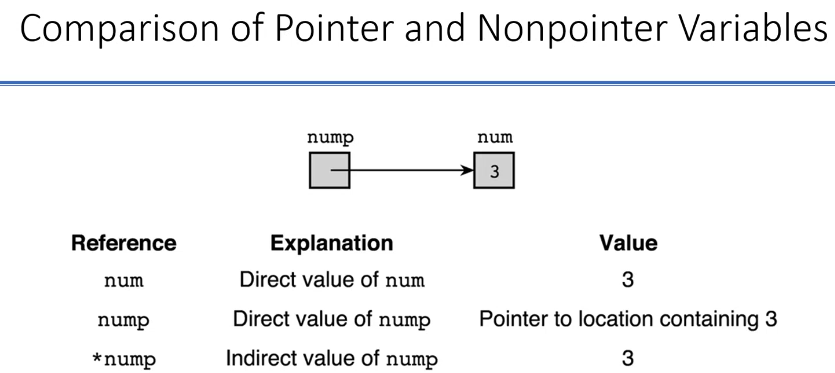
Definition of address+3 depends on what type of value this address is actually keeping.

So pointers are not just address values, they also know type of data that is stored in that address.

Subtraction is very similar, but multiplication and division don’t make sense.

int \* ip;

type

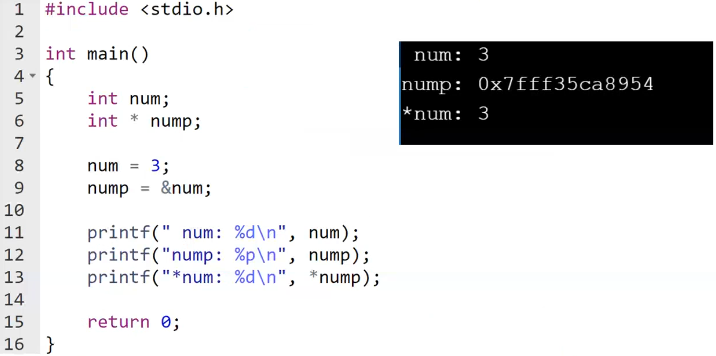


int num;

int \* nump;

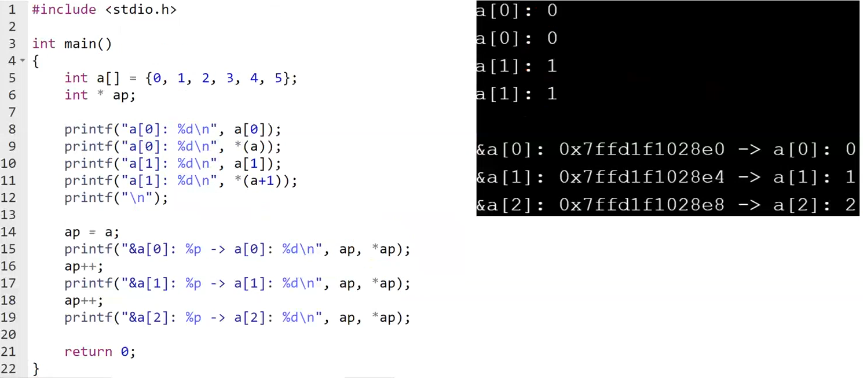
num = 3;

nump = &num;



(\*num should be \*nump)

you can say “%x” (unsigned hexadecimal number) instead of “%p” but you have to do typecast



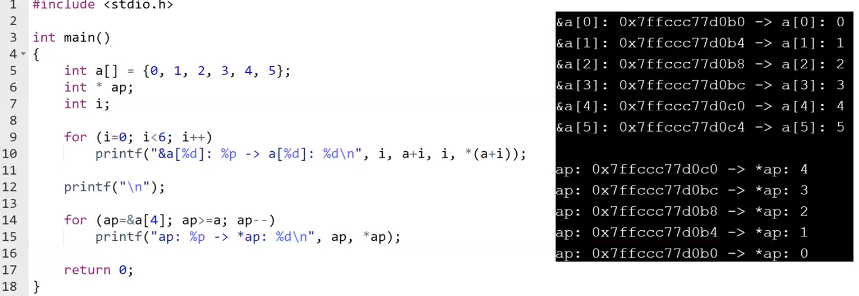
C treats array variables as pointers.

a 🡪 001000

a+1 🡪 001004

a is somewhere in the local memory of the function. So “ap = &a” doesn’t have a meaning.

(pointer)++; 🡪 look at the amount of the pointer and increment by the amount of bytes that type requires



&ap 🡪 int \* \*

Pointer review (continue):

* Function parameters
  + Output parameter
  + Input parameter
* Representing arrays and strings
* Pointers to structures

Operations with pointers:

* Indirection
* Assignment
* Equality operators (==, !=)
* Increment, decrement

***Dynamic Memory Allocation***

Pointer declaration does not allocate memory for values.

double \* nump;

Use function malloc (Memory ALLOCation) to allocate memory

void \* malloc(…); 🡪 definition of malloc

* + - idk what the type of my pointer is, that’s why we call it void
    - this will return something of undefined type but depending on the number of bytes, I can interpret it anyway I want
    - If I say malloc(8) : 8 byte is enough for a double. malloc(8) is gonna allocate 8 bytes in the memory, return the address to me but address type is not known, it is void pointer. I need to recast it to a double pointer to store an double pointer variable. We can do this because void pointer, int pointer, double pointer or planet\_t pointer all have the same range of values, they all are same thing, they are addresses.

All in all nump only holds address which is double pointer.

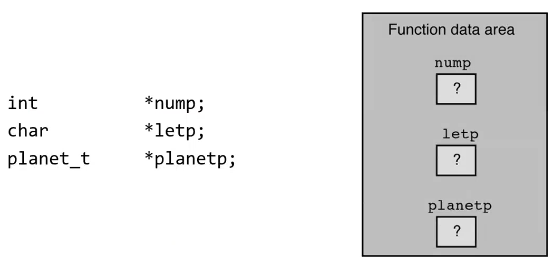
nump = (double \*)malloc(8);

malloc(sizeof(double))

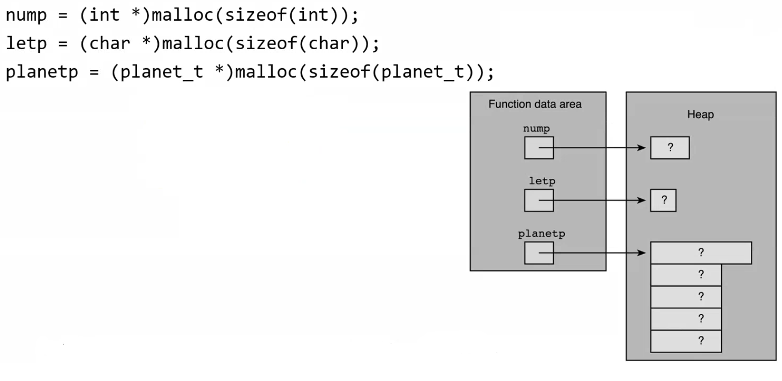
sizeof 🡪 tell you how many bytes C needs for the parameter you gave

* Allocates number of bytes defined by the parameter
* Memory allocated in the heap ( not stack )
* Returns a pointer to the block allocated
* Memory allocated by malloc could be used to store any value
* What should be the return type? (type of the pointer)

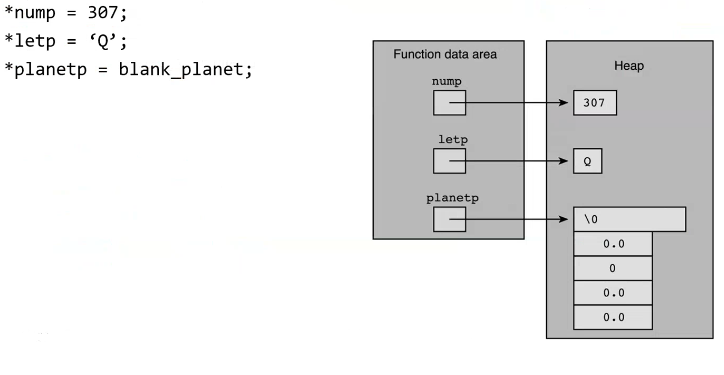
double \* nump = (double\*)malloc(sizeof(double));



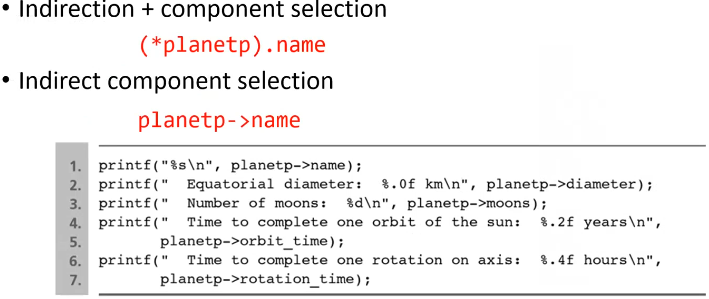
(STACK)

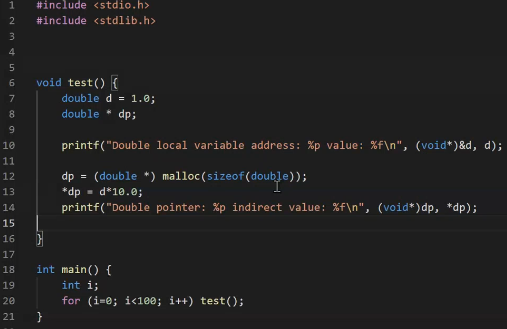


Different area of the memory, managed by the run-time engine



planet must be “planet\_t \*” type





printf doesn’t care about type of your pointer, it cares only the address so cast it to void\*.

Each time you call the function, same portion of the stack is allocated for the function. So address of d will be same every time.

However malloc part changes.

When I am out of the function, dp and d (local variables) are gone, I don’t have access to them anymore. However I actually stored my allocated memory in one of those variables. When I am out of the function, that variable’s value is gone but address is there, address is allocated. Compiler thinks that “I have that memory used by me.” but I have no longer access to that memory because I am out of the function. Either I need to “free” it.

free(dp); 🡪 I no longer need it, it is yours, do whatever you wanna do with it. If I don’t do this, compiler would think I am using it but I cannot use it because I forgot the address of it when I am out of the function.

Or either I return the address in dp so it becomes useful for someone else.

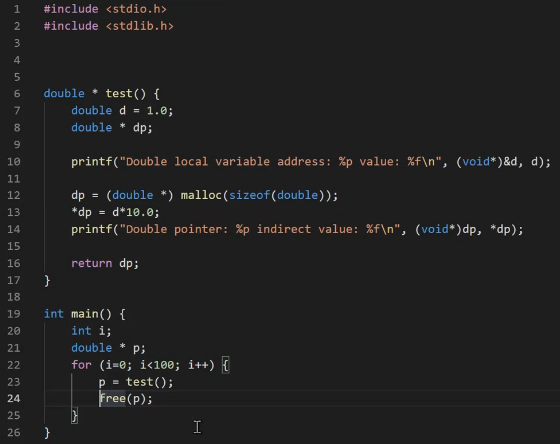
C says that memory allocation is done, your program has that memory.

free 🡪 I needed this dynamic local memory, I have done my job with it, I no longer need it. I am returning that memory back to the compiler.

You should use “ free(dp);” in the line 15.

Using free doesn’t guarantee that you always get different address for dp.

You can also do like this:



What happens if there are no more memory available in the heap?

Is the heap area of the memory allocated for the program a limited resource?

* + It is limited because our memory is limited. There is a limitation also by the machinery, OS and so forth, it can only support finite amount of memory.
  + If the heap is limited, there is a possibility that I can ask for the compiler to give all the memory it has for the heap.
  + Assume that I have 2 GB of heap and I do this:

for (i = 0; i < 1000000; i++)

malloc(2500);

* + What is gonna be done when there isn’t any memory in the heap to allocate?

malloc returns a warning. It returns a value that doesn’t mean a valid address, 0.

So malloc will return 0 if there is no more memory to give. So I should do sth like this:

planet = (planet\_t \*)malloc(sizeof(planet\_t));

if (planet == 0) /\* I couldn’t get memory for this thing \*/

… /\* I need to do sth specific \*/

int \* ip;

ip = (int \*)malloc(2 \* sizeof(int));

ip points to here, beginning of the address

this is what malloc reserves

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

\*ip = 10; 🡪 You just use first 4 bytes.

\*(ip + 1) = 11; 🡪 You use second 4 bytes.

You can do ip[0], ip[1] but not ip[2]

Actually when you define an int pointer (int \* ip), you can access it in 2 ways:

(\*ip) = … (prefer this if you are not using an array)

ip[0] = …

So any pointer can be thought of as an array.

\*(ip + i) === ip[i]

You can treat this as an array but this is not a very good way of doing this. C says if you want to use this as an array (to get an array), use calloc.

* malloc: to allocate single memory block
* calloc: to dynamically create array of elements
  + Elements of any type (built-in or user defined)

void \* malloc (unsigned int size);

void \* calloc (unsigned int num\_elements, unsigned int size\_of\_each\_element);

* calloc: 2 arguments
  1. Number of elements
  2. Size of one element

int \* ap = (int \*)calloc(2, sizeof(int));

(ap: array pointer which has 2 entries)

* Calloc allocates the memory and initializes everything to zero (**bits** will be all 0)

int \* a = calloc(10, …); 🡪 guarantees that a[0], a[1], a[2], …, a[9] all are zero

* Returns a pointer

Suppose that you ask to user how many numbers he/she will give.

In the past you would create an array (int a[1000];) and suppose user will not exceed 1000.

With this way I allocate 1000 entries and use just a portion of it.

But now you can do this:

int \* a;

int n;

scanf(“%d”, &n);

a = (int \*)calloc(n, sizeof(int));

if (a == 0) {

printf(“There isn’t that much memory.”);

exit(-1); }

free(a);

With this way, my program will use only as many space as needed. There is no waste.

Also user can enter a number bigger than 1000. My code will still work.

**Returning Memory**

* free : returns memory cells to heap
  + Allocated by calloc or malloc
  + Returned memory can be allocated later, else cannot be allocated
  + Parameter should be a valid address, pointer

free(nump);

free(array\_of\_planets);

A function reads number of entries (integers) from the user, sorts them, prints them and returns.

void f(){

int \* a;

int n;

scanf(“%d”, &n);

a = (int \*)calloc(n, sizeof(n));

if (a != NULL){

sort(a, n);

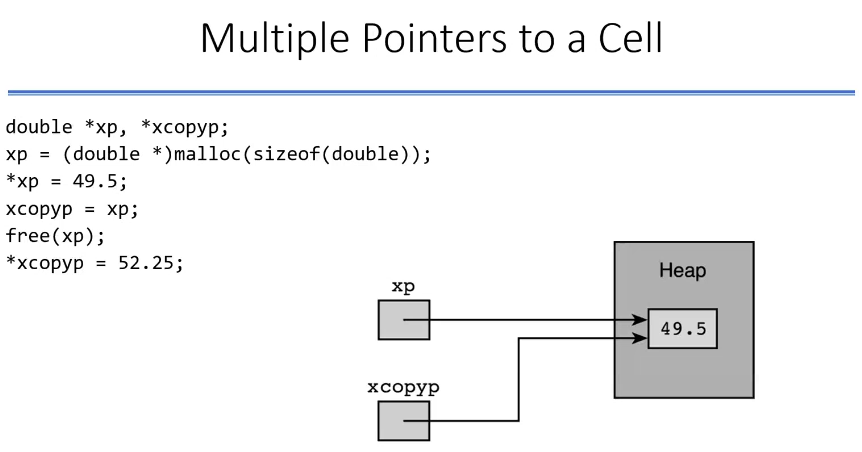
print(a, n);

}

free(a); 🡪 a may be null or not, this causes problem because I cannot free a null address

}

free must be inside the if, after the “print(a);”



We have learnt 3 functions to manage heap memory allocation:

* malloc
* calloc
* free

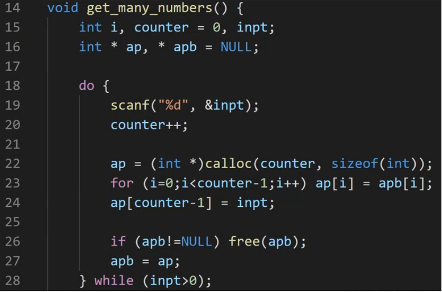
In the beginning they all know memory is in the beginning empty.

They have a list of allocated memory.

When you call malloc, they add an entry to this list. List says “I have allocated this much memory (amount) at this location (address).”.

If you do another allocation (malloc - calloc), same thing will happen.

When you call free, it’s gonna look at this list, and look at the addresses there. It’s gonna match one of those addresses and release it.



counter: number of entries

for loop is for copying older array to new array.

apb is NULL so I don’t enter the if part.

Firstly, let’s assume user enters 5. We won’t enter the for part either because counter-1 and i are equal.

HEAP MEMORY

i

counter

inpt

ap

apb

|  |
| --- |
|  |
| 1 |
| 5 |
|  |
|  |

|  |
| --- |
| 5 |

Secondly, let’s assume user enters 7.

|  |
| --- |
| 5 |

HEAP MEMORY

i

counter

inpt

ap

apb

|  |
| --- |
|  |
| 2 |
| 7 |
|  |
|  |

|  |
| --- |
| 5 (thanks to for loop) |
| 7 |

After the if part, where address in apb is pointing at will be released. So 5 will be freed and address in apb will be same with address in ap.

|  |
| --- |
| 5 (thanks to for loop) |
| 7 |

HEAP MEMORY

i

counter

inpt

ap

apb

|  |
| --- |
|  |
| 2 |
| 7 |
|  |
|  |

Thirdly, let’s assume user enters 1.

|  |
| --- |
| 5 (thanks to for loop) |
| 7 |

HEAP MEMORY

i

counter

inpt

ap

apb

|  |
| --- |
|  |
| 3 |
| 1 |
|  |
|  |

|  |
| --- |
| 5 (thanks to for loop) |
| 7 (thanks to for loop) |
| 1 |

After freeing apb and assign apb to ap.

|  |
| --- |
| 5 (thanks to for loop) |
| 7 (thanks to for loop) |
| 1 |

HEAP MEMORY

i

counter

inpt

ap

apb

|  |
| --- |
|  |
| 3 |
| 1 |
|  |
|  |

Lastly, let’s assume user enters -1.

|  |
| --- |
| 5 (thanks to for loop) |
| 7 (thanks to for loop) |
| 1 |

HEAP MEMORY

i

counter

inpt

ap

apb

|  |
| --- |
|  |
| 4 |
| -1 |
|  |
|  |

|  |
| --- |
| 5 (thanks to for loop) |
| 7 (thanks to for loop) |
| 1 (thanks to for loop) |
| -1 |

After freeing apb and assign ap to apb.

|  |
| --- |
| 5 (thanks to for loop) |
| 7 (thanks to for loop) |
| 1 (thanks to for loop) |
| -1 |

HEAP MEMORY

i

counter

inpt

ap

apb

|  |
| --- |
|  |
| 4 |
| -1 |
|  |
|  |

At the end, I have this array (apb and ap):

|  |
| --- |
| 5 |
| 7 |
| 1 |
| -1 |

|  |
| --- |
| 5 |
| 7 |
| 1 |
| -1 |

|  |
| --- |
| 5 |
| 7 |
| 1 |

|  |
| --- |
| 5 |
| 7 |

If we don’t use free, at the end we would have 4 arrays in the heap:

|  |
| --- |
| 5 |

First 3 arrays would no longer be accessible by anybody.

We call first 3 arrays as zombie (memory leak). calloc-malloc-free thinks that these arrays are used but my program has no way of accessing these particular memories. calloc-malloc-free thinks that these memories are alive but actually they aren’t, I can’t reach them.

Actually what we did with function that managing memory for a dynamic array is not very efficient at all. Because at each entry:

* need to allocate n+1 space
* copy n entries to the new array
* delete the old array

Instead of implementing function in this way, I can grow the array not 1 at a time but 5 at a time.

With 5 at a time, I am doing allocation, copying and freeing at every 5 inputs.

If the person gives up after 1 entry, waste will only be 4 entries.

|  |
| --- |
| a |
| b |

realloc

Let’s say you have 2 entries in the memory.

realloc says “If you have these entries, I can get you some more entry incremental on top of these.”

But array has to be continuous in space (each of entries should be one after another). For realloc to work, if you have 2 entries already and you wanna add 1 more entry, realloc needs to have consecutive memory place after b is empty for you. If it is empty, realloc can add one more or so. You can expand an array using realloc as long as your next entry is available. Otherwise it’s not gonna work.

realloc( {pointer that you allocated before}, {memory size});

You should prefer allocate your memory in heap instead of stack because:

stack memory size << heap memory size

Implementing Stack

#define MAX\_STACK\_SIZE 1000

typedef struct{

int \* a;

int counter;

}stack;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void push (stack \*s, int x){

s -> a[s -> counter] = x;

s -> counter++;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int pop (stack \*s){

int r = -1;

if (s -> counter > 0){

r = s -> a[s -> counter-1];

/\* Assume counter is showing the very last entry, which is not entered yet \*/

s -> counter--; }

return r;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void create(stack \*s){

s -> a = calloc(MAX\_STACK\_SIZE, sizeof(int));

s -> counter = 0;

}

Implementing Stack Dynamically

void create(stack \*s){

s -> a = NULL;

s-> counter = 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void push(stack \*s, int x){

int \* temp, i;

temp = calloc(s -> counter + 1, sizeof(int));

for(i = 0; i < s -> counter; i++) temp[i] = s -> a[i]; /\*copying\*/

temp[counter+1] = x;

if (s->a != NULL) free(s -> a);

s -> a = temp;

s -> counter++;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int pop(struct \*s){

int \* temp, i;

temp = calloc(s -> counter - 1, sizeof(int));

...

}

typedef struct engine{

char name[20];

int year;

float volume;

enum {v4 = 4, v5 = 5, v6 = 6, v8 = 8} cylinders;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void test(){

engine e1 = {“Alfa Romeo”, 1979, 2.5, v6}, e2, \*e3;

engine\_print(e1);

e2 = engine\_read(“engine1.txt”); /\* returning struct \*/

engine\_print(e2);

e3 = engine\_read\_dynamic(“engine2.txt”); /\* returning pointer \*/

engine\_print(\*e3);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void engine\_print(engine e){

printf(“Engine: ‘%s’ %d %.1fL v%d\n”,

e.name, e.year, e.volume, e.cylinders);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

engine engine\_read(const char \* filename){

/\* also can be const char filename[] \*/

engine e;

FILE \*fin;

fin = fopen(filename, “rt”);

fgets(e.name, 20, fin); /\*reads into string as number of chars\*/

/\* You can also use this instead of fgets:

fscanf(fin, “%[^\n]s”, e.name); \*/

e.name[strlen(e.name)-1] = 0;

fscanf(fin, “%d %f %d”, &(e.year), &(e.volume), &(e.cylinders));

fclose(fin);

return e;

/\* FILE EXAMPLE

Ferrari F154

2014

2.9

8 \*/

}

/\* makes a copy operation, involves exactly 32 bytes without padding \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

engine \* engine\_read\_dynamic(char \* filename){

engine e;

FILE \* fin;

fin = fopen(filename, “rt”);

fgets(e.name, 20, fin);

e.name[strlen(e.name)-1] = 0;

fscanf(fin, “%d %f %d”, &(e.year), &(e.volume), &(e.cylinders));

fclose(fin);

return &e;

}

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

/\* copy only 8 bytes for address \*/

|  |
| --- |
|  |
|  |
|  |
|  |

e

e1

e2

e3

\*\*\*Every rectangle is 8 byte.\*\*\*

As we know all the local variables are gone when we return from the function. So what happens to e3?

This is kind of an another version of zombie.

In this version, there is something that is not alive, but they are there. I have an address in e3 but that address is no longer valid. I can not use that address anymore.

So this function is wrong. We can implement the function as this:

engine \* engine\_read\_dynamic(char \* filename){

engine \* e;

FILE \* fin;

fin = fopen(filename, “rt”);

e = (engine \*)malloc(sizeof(engine));

fgets(e -> name, 20, fin);

e -> name[strlen(e -> name)-1] = 0;

fscanf(fin, “%d %f %d”, &(e->year),&(e->volume),&(e->cylinders));

fclose(fin);

return e;

}

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

|  |
| --- |
|  |
|  |
|  |
|  |

HEAP

e1

e2

e3

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Let’s put the 3 engines in an array

1. You can define an fixed size engine array. If I don’t use 10 engines, there are wastes. Also if I need more than 10 engines, this will cause problem. (SPACE IS THE PROBLEM)

engine ea[10];

1. Other possibility is fixed size dynamically allocated array. When you return from the function, you will still have access to the array. This is the advantage. (SPACE IS THE PROBLEM)

engine \* ea = calloc(10, sizeof(engine));

1. You can also make incrementally growing array. But with this option, you have to do a lot of copying as you can see in page 12. (SPEED IS THE PROBLEM)
2. Other possibility is using some kind of stack.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\* DELETING AN ENTRY SPECIFIED BY YEAR FROM THE ENGINE ARRAY \*\*\*/

void test(){

engine ea[10];

int counter = 0; /\* entries that I have in my array \*/

ea[counter++] = engine\_read(“engine1.txt”);

ea[counter++] = engine\_read(“engine2.txt”);

ea[counter++] = \*engine\_read\_dynamic(“engine3.txt”);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\* Assume counter=5 (so I have 5 engines) \*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* We are gonna find the first engine in array built in the given year \*/

If the found entry to be deleted/removed from the array

void find\_and\_remove(engine a[], int year, int \*n){

int i, j;

for (i = 0; i < n; i++)

if (a[i].year == year){

-----------------------------------------------

/\* if order is not important, we can put last

entry to the removed entry in the array and

discard/don’t worry the last entry \*/

ea[i] = ea[\*n - 1];

\*n--;

-----------------------------------------------

/\* if order is important \*/

for (j = i+1; j < \*n; j++)

ea[j-1] = ea[j];

\*n--;

}

}

Memory For The Program

void main(){

int x;

double y;

f(x, y);

}

void f(int a, double b){

int c;

malloc(...);

calloc(...);

}

C code comes in, goes to the compiler, then goes to the linker, then goes to the loading process, then

|  |
| --- |
| Code (machine/assembly)  {from 0th address to some address}  That’s why 0th address means nothing while allocating some memory from the heap |
| {x} |
| {y} |
| Calling the function  (something will be put in here)  function knows where to go back when function ends |
| {a} |
| {b} |
| {c} |
| UNUSED PART |
|  |
| {malloc} |
| {calloc} |

will grow this way

STACK

Program Memory

will grow this way

HEAP

After we done with the function:

|  |
| --- |
| Code (machine/assembly) |
| {x} |
| {y} |
|  |
|  |
|  |
| {malloc} |
| {calloc} |

STACK

Program Memory

HEAP

Compiler didn’t erase a, b, c… It just forgot them.

As you call a function, usage of stack will grow; as you return from a function, usage of stack will shrink.

During this growing and shrinking, values that you previously had (a, b, c) stay there. That’s why when I call the function next time, you might have different value for c because then you are using someone else’s remains in that place.

But when you get out from the function, nothing happens to heap.

Code part of the memory is fixed because code doesn’t change.

When some memory is given to your function, stack part is fixed, compiler says “You are gonna have this much stack.”. There will be a predefined amount of stack that you can use. There is a parameter that you can actually give to the compiler saying that “I wanna use not the default value of the amount of stack I need, but I wanna use larger stack.”. That is a compiler flag that you can use as a parameter to change that stack size but once you compiled to executable, then your stack size is fixed.

However heap part is not necessarily fixed but there is an upper limit which is defined by the OS. There is a limit that you can use but when you look at task manager or some sort of a monitoring program that looks at your program and says how much memory your program is using, you will see that with the use of the heap you are gonna have shrinking or growing memory.

This is something to do with the OS, not necessarily related to C.

That’s why when you have recursive function, and you call it infinitely many time, stack will grow grow grow and it will hit its maximum and you will see a stackoverflow error. OS is gonna kill your program and say “You are out of your stack, do it right or I will not run it.”. This is OS job to monitor and police things around.

Stack is not under your control, it is under compiler’s control. So you can’t decide what’s gonna be released and what’s gonna be kept in stack.