

# The Execution Model, the Stack & the Heap

Luca Tornatore - I.N.A.F.



DATA SCIENCE &  
ARTIFICIAL INTELLIGENCE



SCIENTIFIC &  
DATA-INTENSIVE COMPUTING

2024-2025 @ Università di Trieste

# Aim of this Lecture



What happens when you execute a program on a machine, either your laptop or a large tier-0 HPC facility ?

How does the code – “the program” – interact with the Operating System and how can it find the way to memory and cpu ?

After all, if a “program” is just the code that has been compiled into machine code, to run it needs more than what meets the eye.

It needs, at least, the memory where the code itself and the data must be uploaded.

It needs access to other resources, the cpu but also I/O and what else it needs

In this lecture we draft what is the execution model focused on \*nix systems.

We choose \*nix systems because they are the *de facto* standard in HPC and on all HPC facility: however, since we are not going into much detail, the big picture is still valid for other O.S.

We also clarify the difference between the stack and the heap, which are the two fundamental memory regions you will deal with.

# Outline



The execution  
model



Stack & Heap



Worked  
examples



# The memory model

When you execute your code, the O.S. provides a sort of “memory sandbox” in which your code will run and that offers a *virtual address space* that appear to be homogeneous to your program (this is to hide the hw details of memory to the applications, enhancing the ease of programming and the portability).

The amount of memory that can be addressed in this virtual box depends on the machine you’re running on and on the operating system:

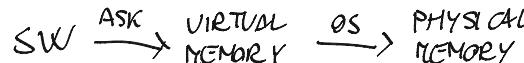
32bits		4GB
64bits	48bits used	256TB
	56bits used	65536TB
	64bits used	16EB

In the very moment it is created, not the whole memory is obviously at hand for the program: that aforementioned is just the addressing capability and the possible maximum size of memory available (which actually is the physical one you have).



# The memory model

The virtual memory inside the sandbox must be related to the physical memory where the data actually live: when the cpu executes a *read* or *write* instruction, it translates the virtual address to a physical address that is then given to the memory controller which deals with the physical memory.



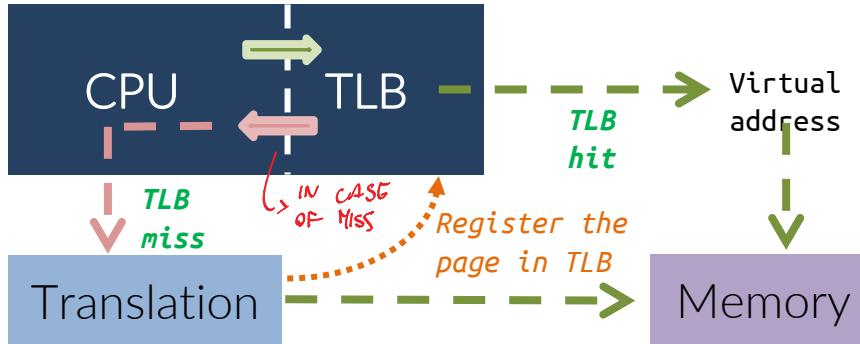
The translation from virtual addresses to physical addresses is done with the paging mechanism. Basically, the physical memory is seen as split in pages, whose size is specific to each architecture (normally is 4KB for RAM of few total GB; it can be 2MB or even 1GB in systems with a much larger RAM). Each physical page can be translated in more than one virtual page (for instance, a shared library will be addressed by more than one process, and so it will be mapped on different virtual spaces).

The mappings is described in a hierarchical table that the kernel keeps in memory, containing a Page Table Entry (PTE) for each page addressed.

Since the translation of virtual addresses to physical addressed can take some time, to make it faster a special cache memory has been introduced, namely the Translation Lookaside Buffer (TLB).



# The memory model

Virtual  
address

The approximate cycle for retrieving a physical address, upon either a TLB hit or a TLB miss.  
The translation is way slower than the looking-up into the TLB.

Typically a TLB can have 12bits of addressing, meaning the storage room for 4096 entries, and a hit time of 1 cycle whereas the miss penalty can be huge, about 100 cycles.

That is why TLB miss have a high cost in terms of performance. However, they usually stays at around ~1-2%

# Outline



## Stack & Heap



## Worked examples



The **stack** is a bunch of **LOCAL MEMORY** that is meant to contain **LOCAL VARIABLES** of each function.

“Local variables” basically are all the variables used to serve the workflow: counters, pointers used locally, strings, initialized local data, .. etc.

The SCOPE of the stack is very limited: **only the current function, or its callees, can access the stack of that same function.**

The stack is basically a bunch of memory allocated for you by O.S., to which the CPU refers by using two dedicated registers (BP and SP). Its most natural use is for **static allocation**.

The **heap** is meant to host the mare magnum of your data and global variables. “Global” data and variables are those that must be **accessible from all your functions in all your code units** (provided that you included the concerning headers).

In addition to housing the global variables, its most natural use is to hold big amount of data, whether or not they have a limited scope, or, in any case, the data whose amount is known only at run-time.

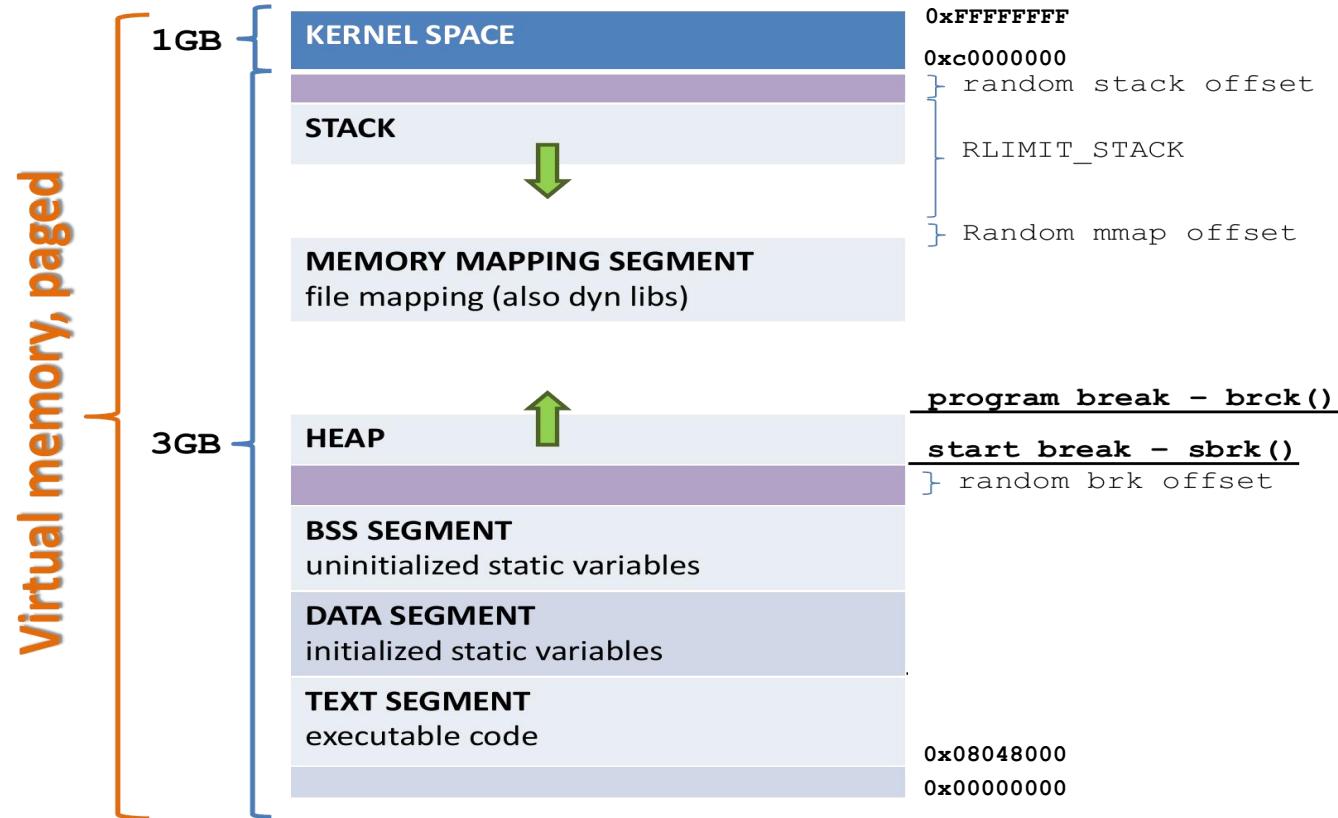
Its most natural use is then for **dynamic allocation**.



# The standard execution model

A simplistic but still effective representation of the execution model in most O.S.

Note:  
4GB was the limit in 32bits systems.  
In 64bits systems, the total amount of addressable memory is much larger

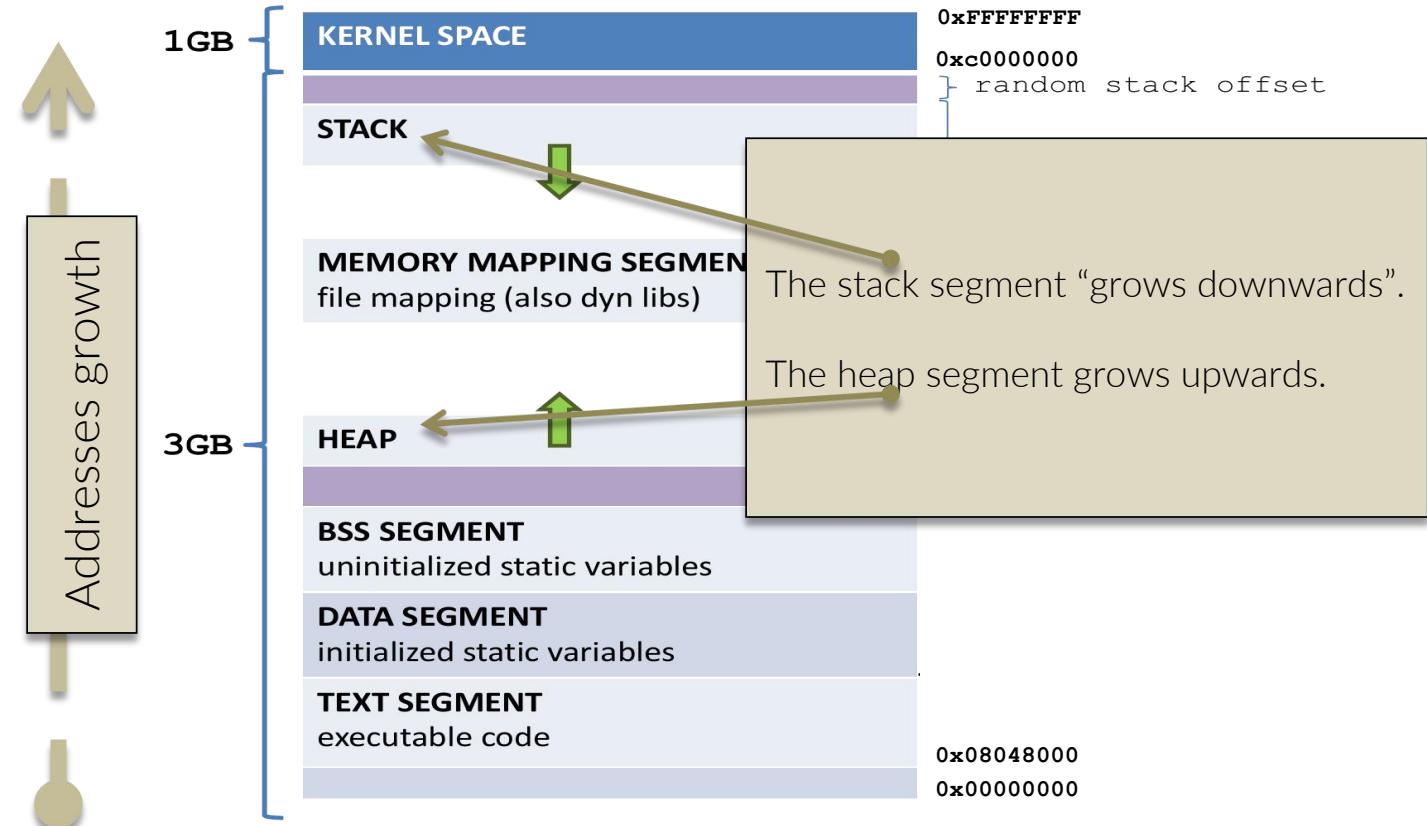




# The standard execution model

A simplistic but still effective representation of the execution model in most O.S.

Note:  
4GB was the limit in 32bits systems.  
In 64bits systems, the total amount of addressable memory is much larger

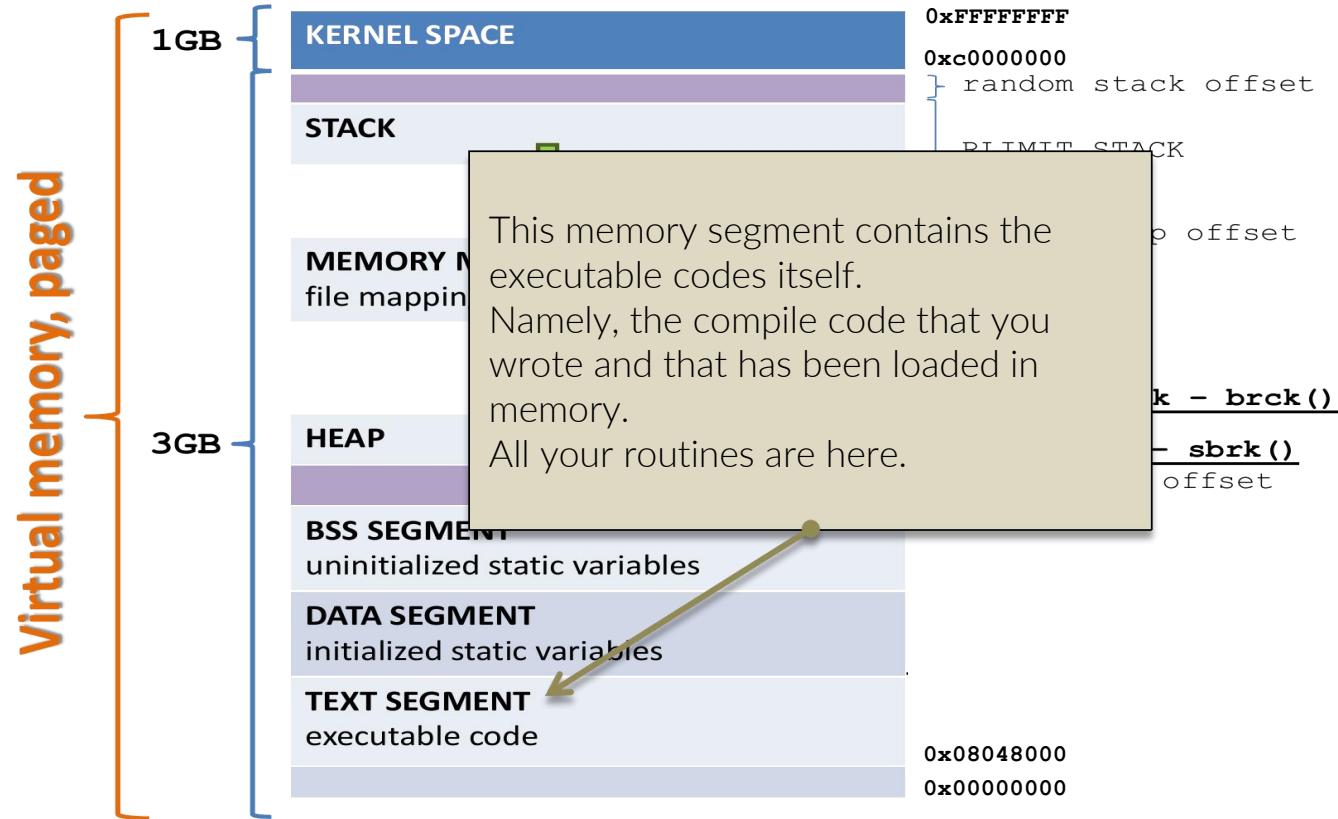




# The standard execution model

A simplistic but still effective representation of the execution model in most O.S.

Note:  
4GB was the limit in 32bits systems.  
In 64bits systems,  
the total amount of addressable memory  
is much larger

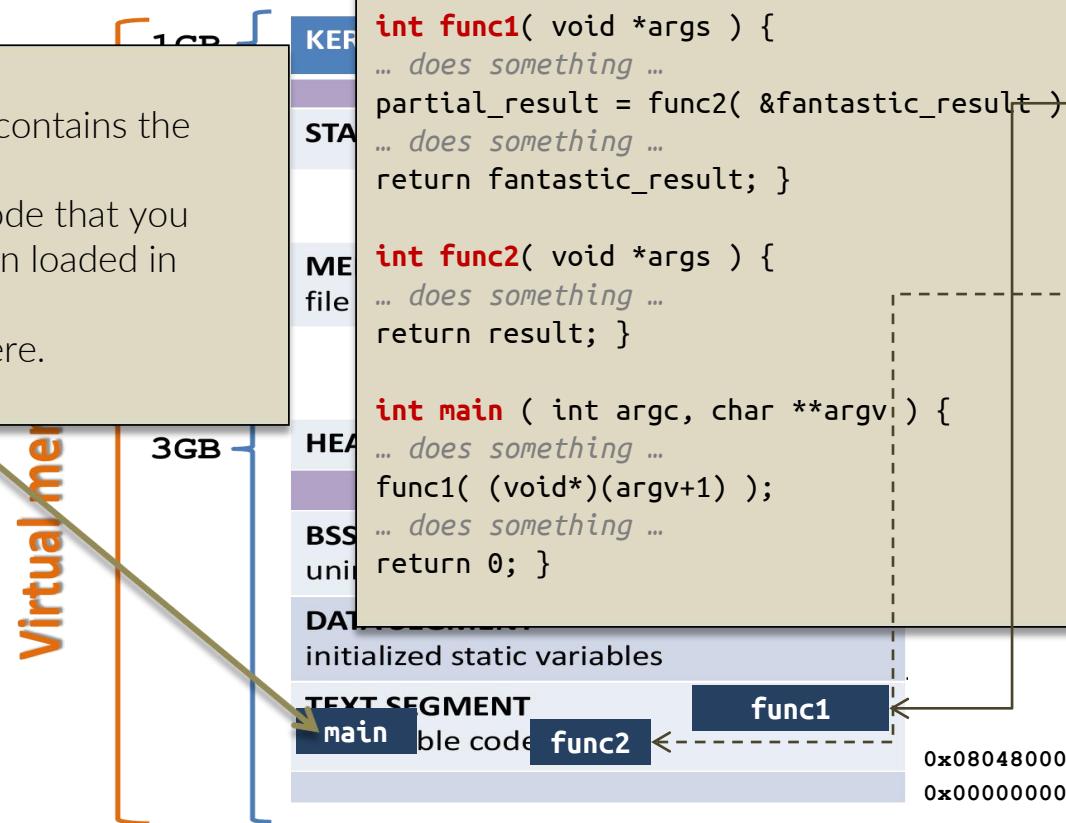




# The standard memory model

This memory segment contains the executable codes itself.  
Namely, the compile code that you wrote and that has been loaded in memory.  
All your routines are here.

Now...  
4GB was the limit in 32bits systems.  
In 64bits systems, the total amount of addressable memory is much larger

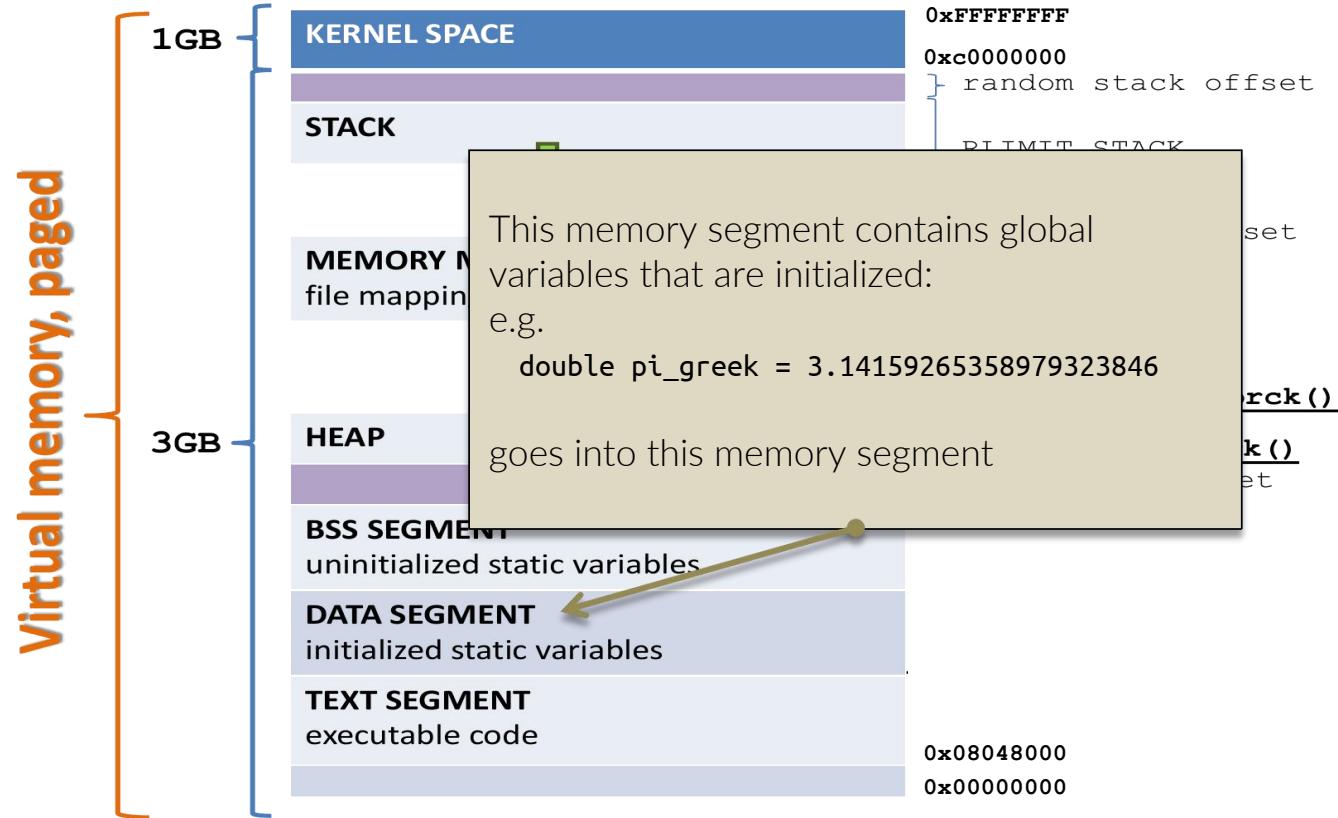




# The standard execution model

A simplistic but still effective representation of the execution model in most O.S.

Note:  
4GB was the limit in 32bits systems.  
In 64bits systems, the total amount of addressable memory is much larger

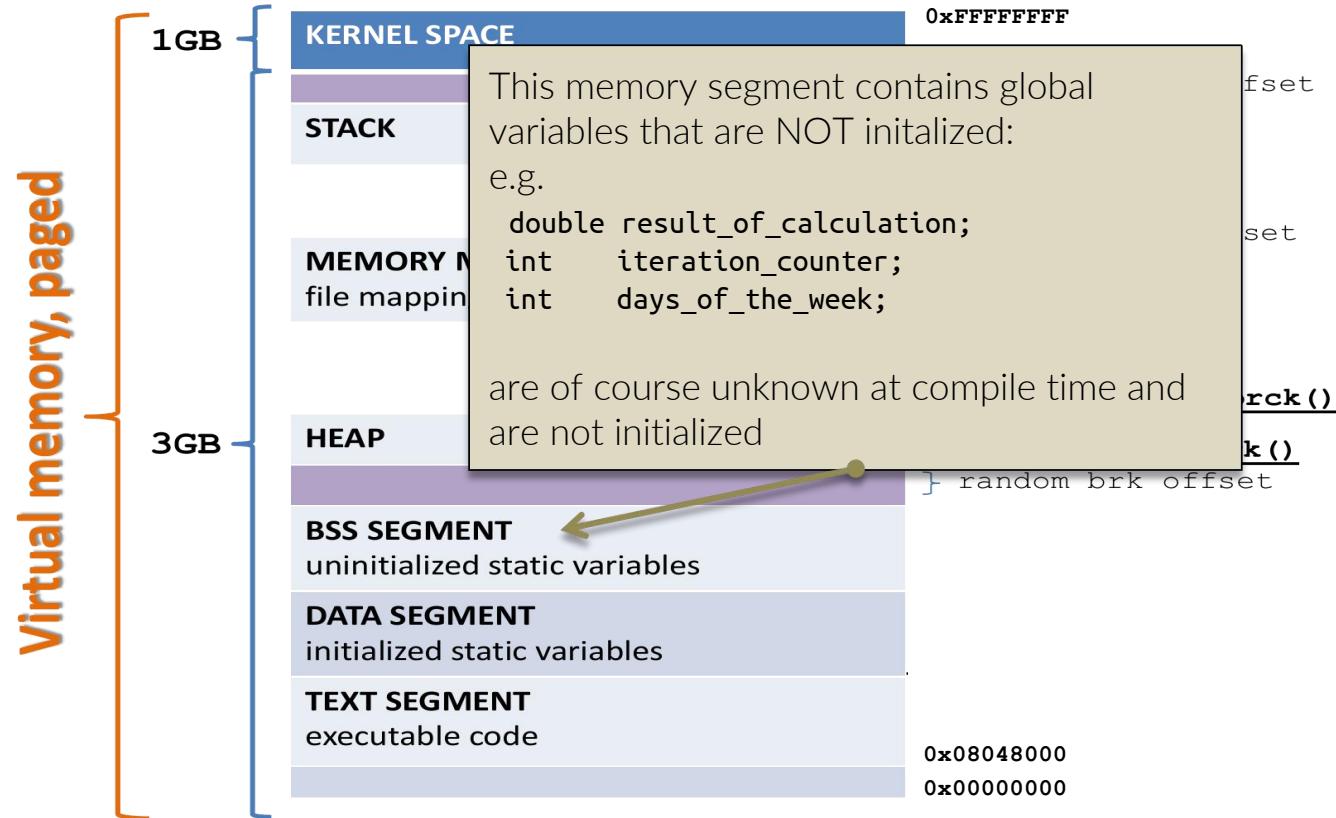




# The standard execution model

A simplistic but still effective representation of the execution model in most O.S.

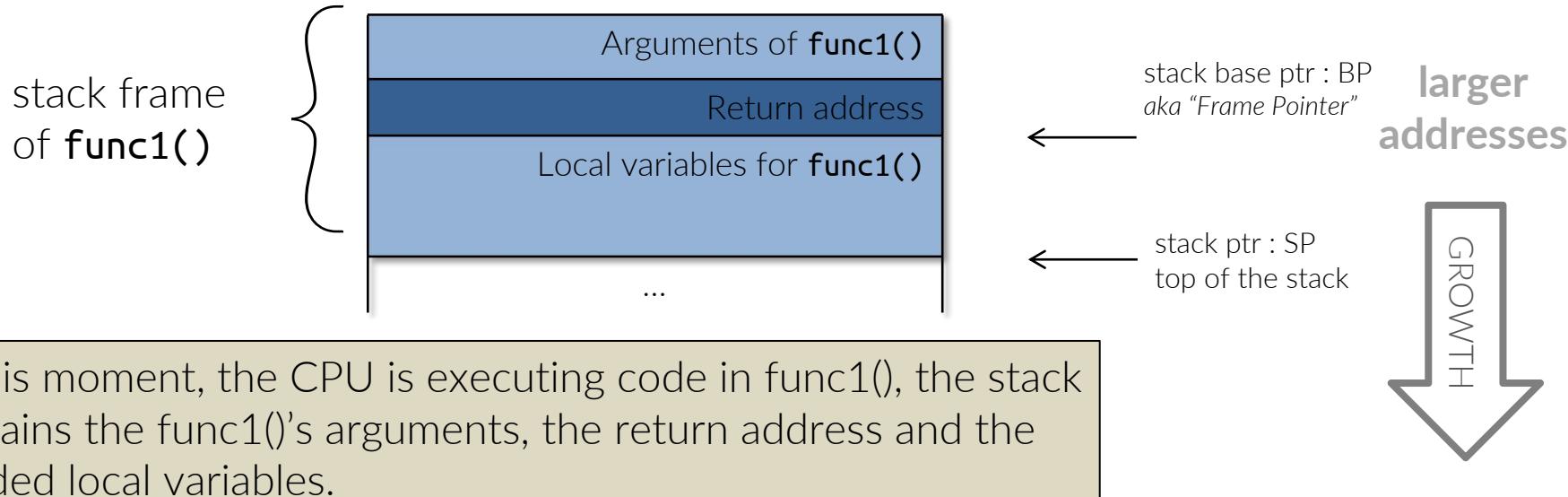
Note:  
4GB was the limit in 32bits systems.  
In 64bits systems, the total amount of addressable memory is much larger





Let's examine in the following slides a simple, common case: a function that calls another function:

```
func1 ( )  
{  
    ...  
    func2 ( )  
    ...  
}
```



**The register SP points at the “top” of the stack, whereas the register BP points at the beginning of the local variables right after the return address address.**



## Zoom in the stack details

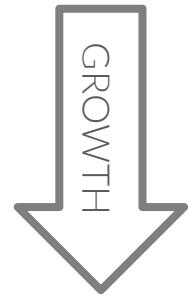
Actually the first 6 arguments that fits in 64bits (numerical values and pointers) are passed in registers.

Arguments that exceed 64bits or beyond the 6th arguments are passed through the stack

stack base ptr : BP  
aka "Frame Pointer"

stack ptr : SP  
top of the stack

larger  
addresses



In this moment, the CPU is executing code in func1(), the stack contains the func1()'s arguments, the return address and the needed local variables.

**The register SP points at the “top” of the stack, whereas the register BP points at the beginning of the local variables right after the return address address.**

smaller  
addresses



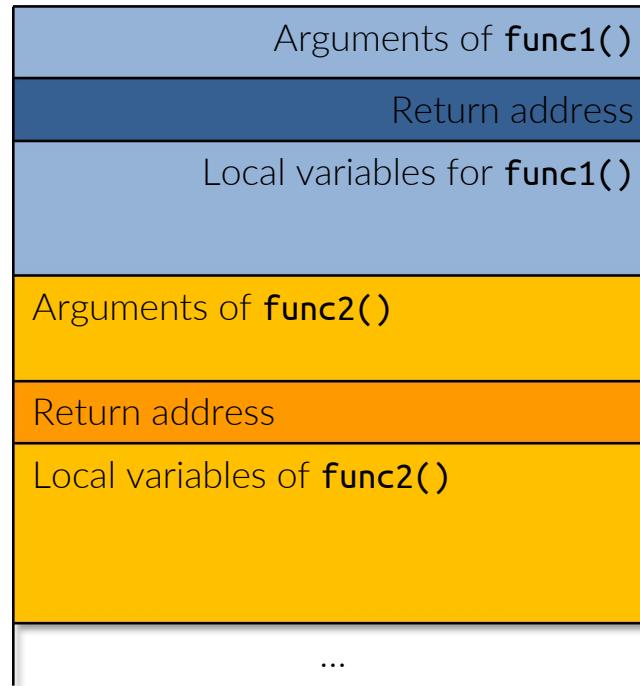
# Zoom in the stack details

Example:

func1() calls func2()

```
func1 ( )  
{  
    ...  
    func2 ( )  
    ...  
}
```

stack frame  
of **func2()**

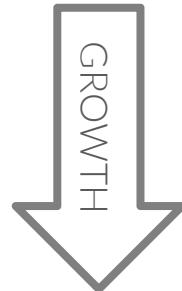


stack frame  
of **func1()**

larger  
addresses

←  
stack base ptr : RBP  
aka "Frame Pointer"

←  
stack ptr : SP  
top of the stack



smaller  
addresses



Example:

```
func1() {
```

Then, **func1()** calls **func2()**.

```
}
```

The stack grows downwards (meaning: SP and BP are decreased as much as needed to host arguments, ret address and local variables)

stack frame  
of **func2()**

Arguments of **func2()**

Return address

Local variables of **func2()**

...

Arguments of **func1()**

Return address

Variables for **func1()**

stack frame  
of **func1()**



stack base ptr : RBP  
aka "Frame Pointer"



stack ptr : SP  
top of the stack

larger  
addresses



smaller  
addresses



Let's now understand how data are accessed in the stack.

Basically, you know the *relative position* of each variable with respect to either SP or BP: for instance,

```
void function( void ) { int i; i = 1; return; }
```

the `i` variable, that occupies 4 bytes, will be at BP or at SP+4 (SP points at the end of the stack which contains only `i`).

In general it will be:

```
address = BP - offset - sizeof( variable )
```

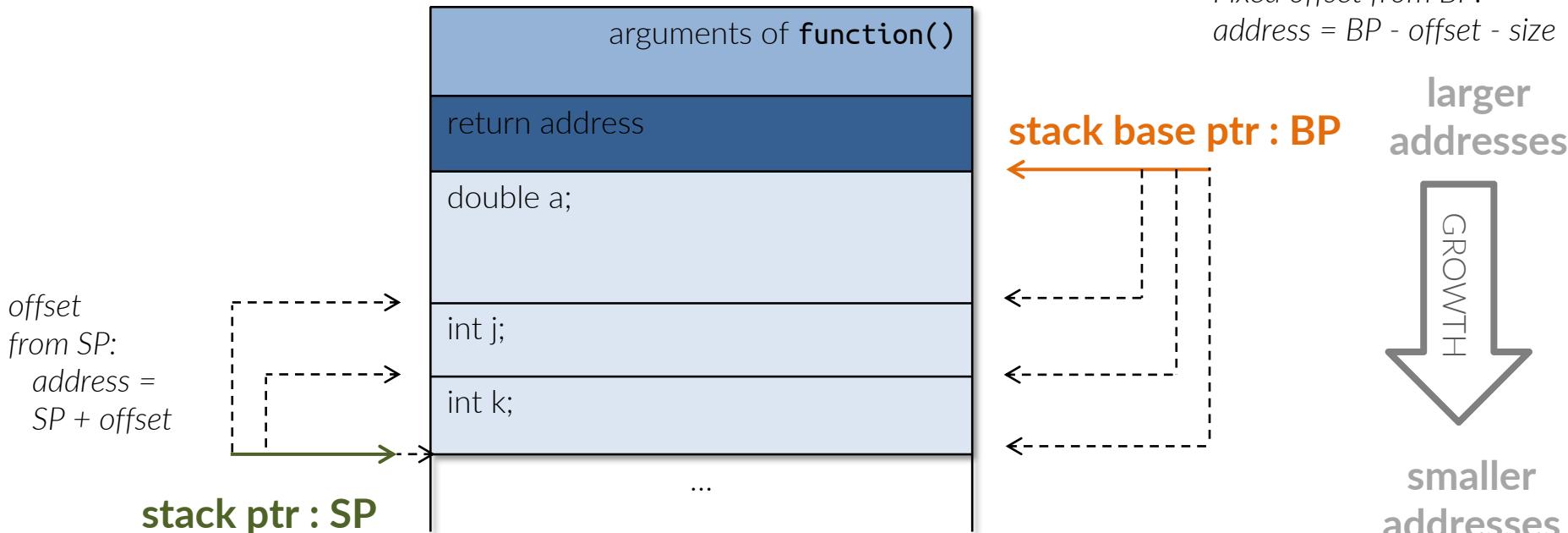
or

```
address = SP + offset
```

where **offset** accounts for all the other data



# Zoom in the stack details



Note: **offset** accounts for all the data from either `SP` or `BP` and the starting of the variable. For instance, the offset of `k` from `SP` is 0, the offset of `j` from `SP` is `sizeof(k)`. Whereas, the offset of `k` from `BP` is `sizeof(a)+sizeof(j)`



# Where does a variable start ?



As you know, any given “variable” is nothing but a number of contiguous bytes that reside somewhere in memory, starting at the variable’s address.

Quite naturally, the variable’s address (i.e. its starting point) is at the lowest memory address, so that the variable’s occupancy grows towards largest address.

Running the simple code snippet here on the right will convince you that the first and last bytes of `i` are at the lowest and highest addresses respectively.

*hint: have a look at the note  
[AOB/Endianism/A\\_note\\_on\\_Endiansim.pdf](#)*

```
int i = 1;
for( int j = 0; j < 4; j++ )
{
    printf("i = %d\n"
           "%p: %d -- %p: %d -- %p: %d -- %p: %d\n",
           i,
           (char*)&i, * (char*)&i,
           ((char*)&i+1), * ((char*)&i+1),
           ((char*)&i+2), * ((char*)&i+2),
           ((char*)&i+3), * ((char*)&i+3) );
    i <= 8;
}
```



Let's scrutinize in more detail the scope of the stack.

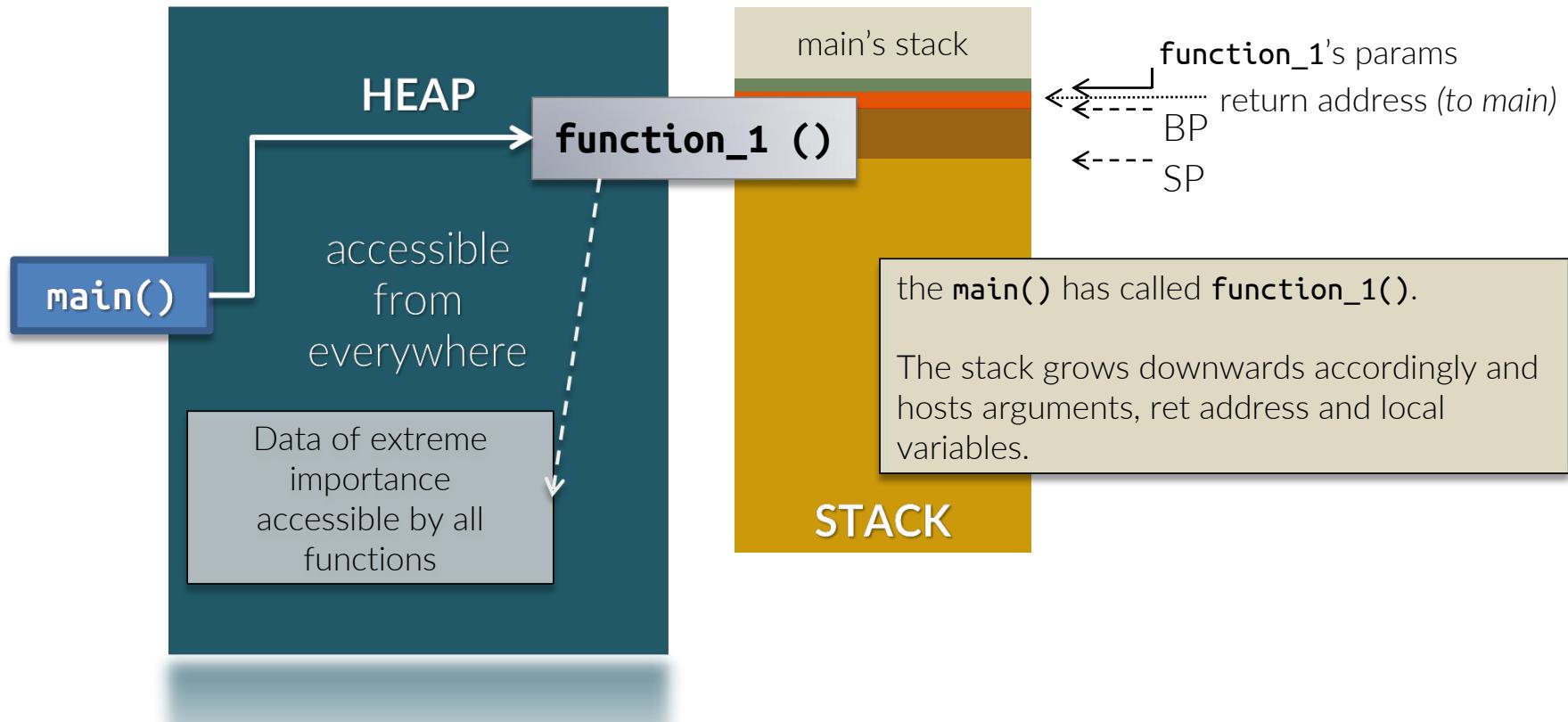
We'll consider the following case: a `main()` that calls `function_1()` and, afterwards, calls `function_2()` which in turns calls `function_3()`.

```
function_1 ( ) { ... }
```

```
function_3 ( ) { ... }
```

```
function_2 ( ) { ...; function_3 ( ); ... }
```

```
int main( ) { function_1(); function_2(); ...; return 0;}
```





## HEAP

**main()**

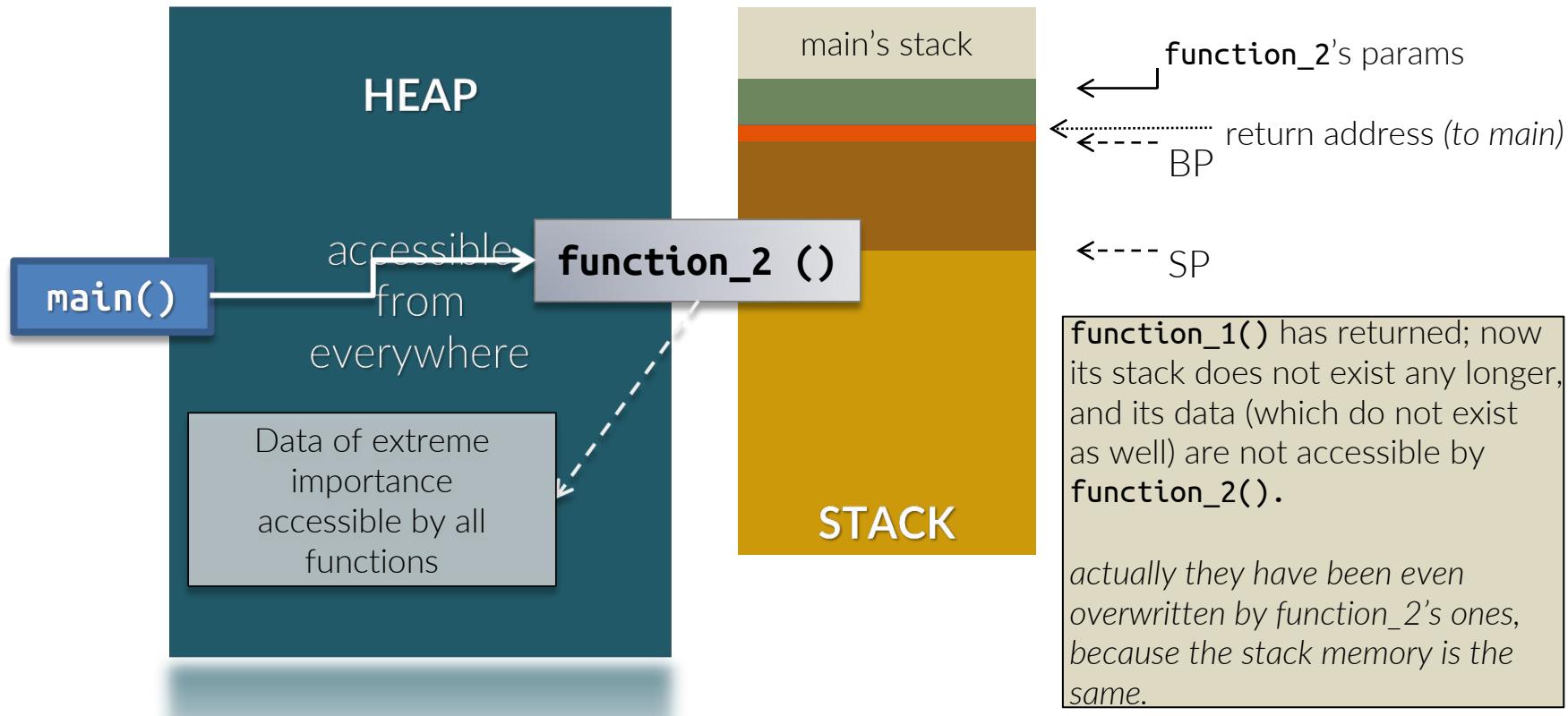
accessible  
from  
everywhere

Data of extreme  
importance  
accessible by all  
functions

main's stack

## STACK

Here `function_1()` has returned to `main()`.



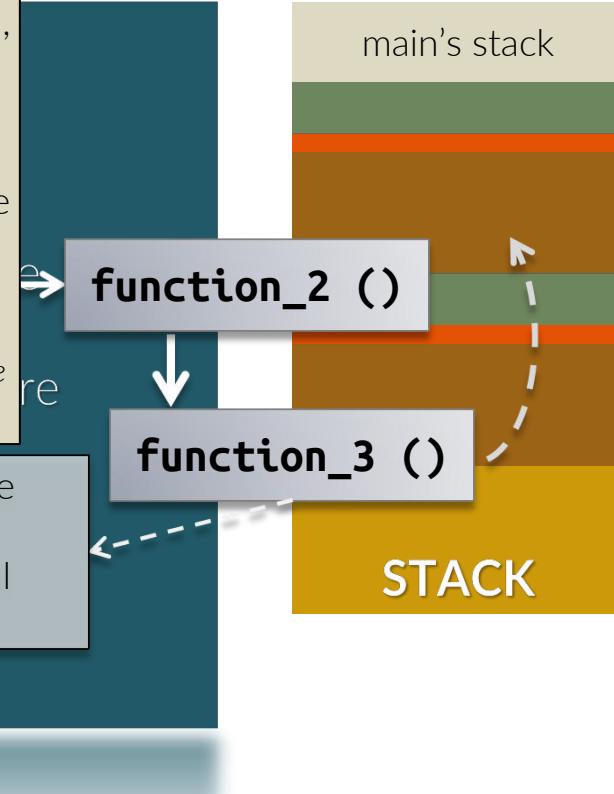


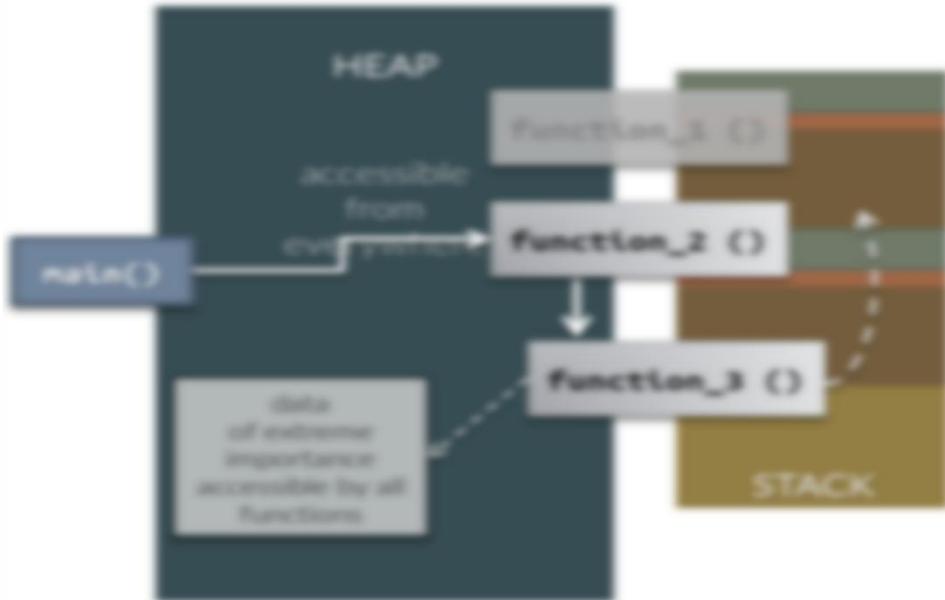
**function\_2()** has called **function\_3()**, and the stack has grown to make room for **function\_3()** context.

Stack of **function\_2()** is still accessible by **function\_3()**

(for instance if **function\_2()** passes a local ptr to **function\_3()** as argument, **function\_3()** could use it to access the stack of **function\_2()**)

Data of extreme importance accessible by all functions





Because of the very nature of the stack, it is obvious that using it as a general storage amounts to not having global variables.

Moreover, among the main advantages of the stack is being small, which means that local variables are likely to fit in the cache.

Wildly widening it is a kind of non sense.



Provided what we have just said, it may be useful to dynamically allocate on the stack some array that has a local scope, being useful *hinc et nunc* but whose size is not predictable at compile time:

```
int some_function_whatever ( int n, double *, ... )
{
    // n is supposed to be not that big

    double *my_local_temporary =
        (double*) alloca ( n * sizeof(double) );

    < ... bunch of ops on my_local_temporary ... >

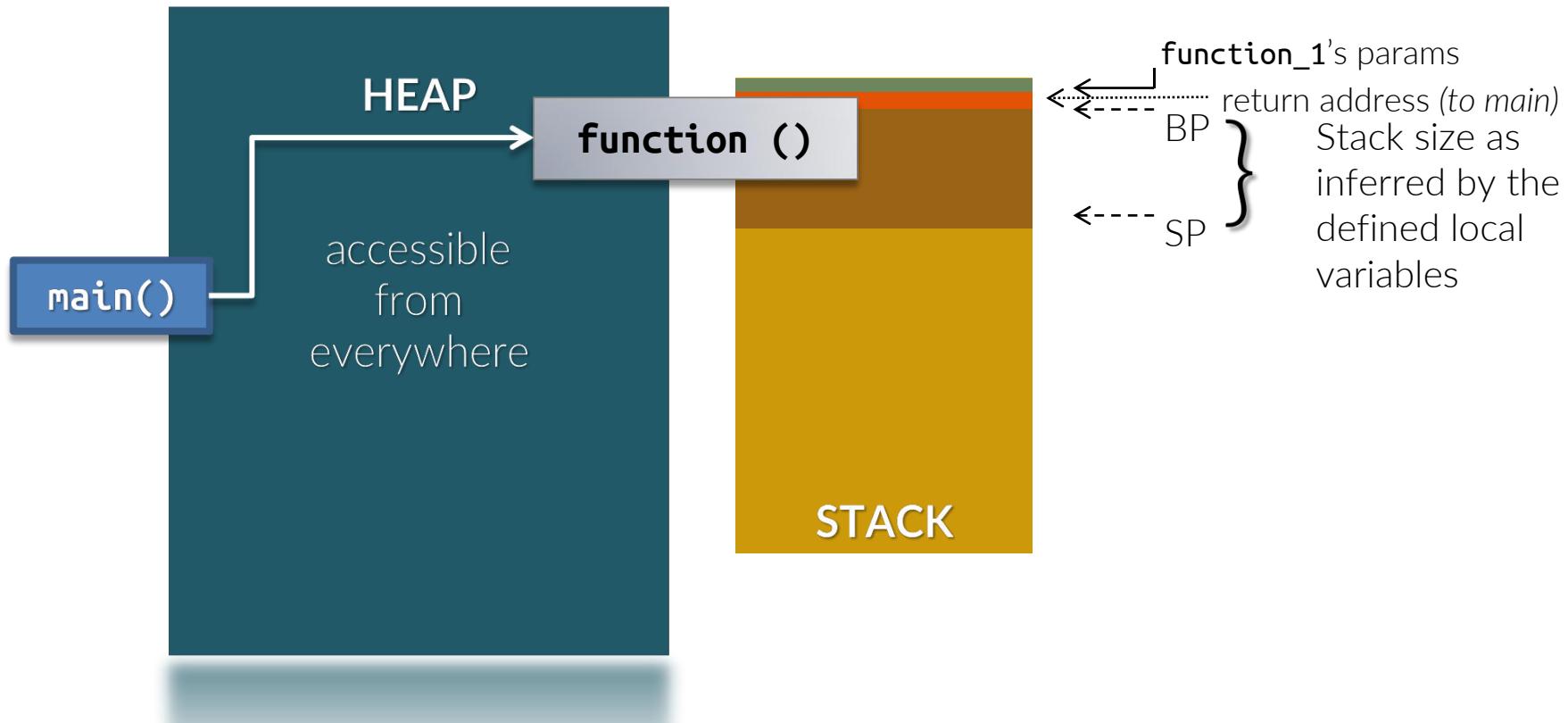
    // you do not need to free my_local_temporary;
    // actually a free would raise an exception
    return result;
}
```

**alloca(...)** allows the dynamic allocation on the stack.

Have a look at **man alloca** for more details.

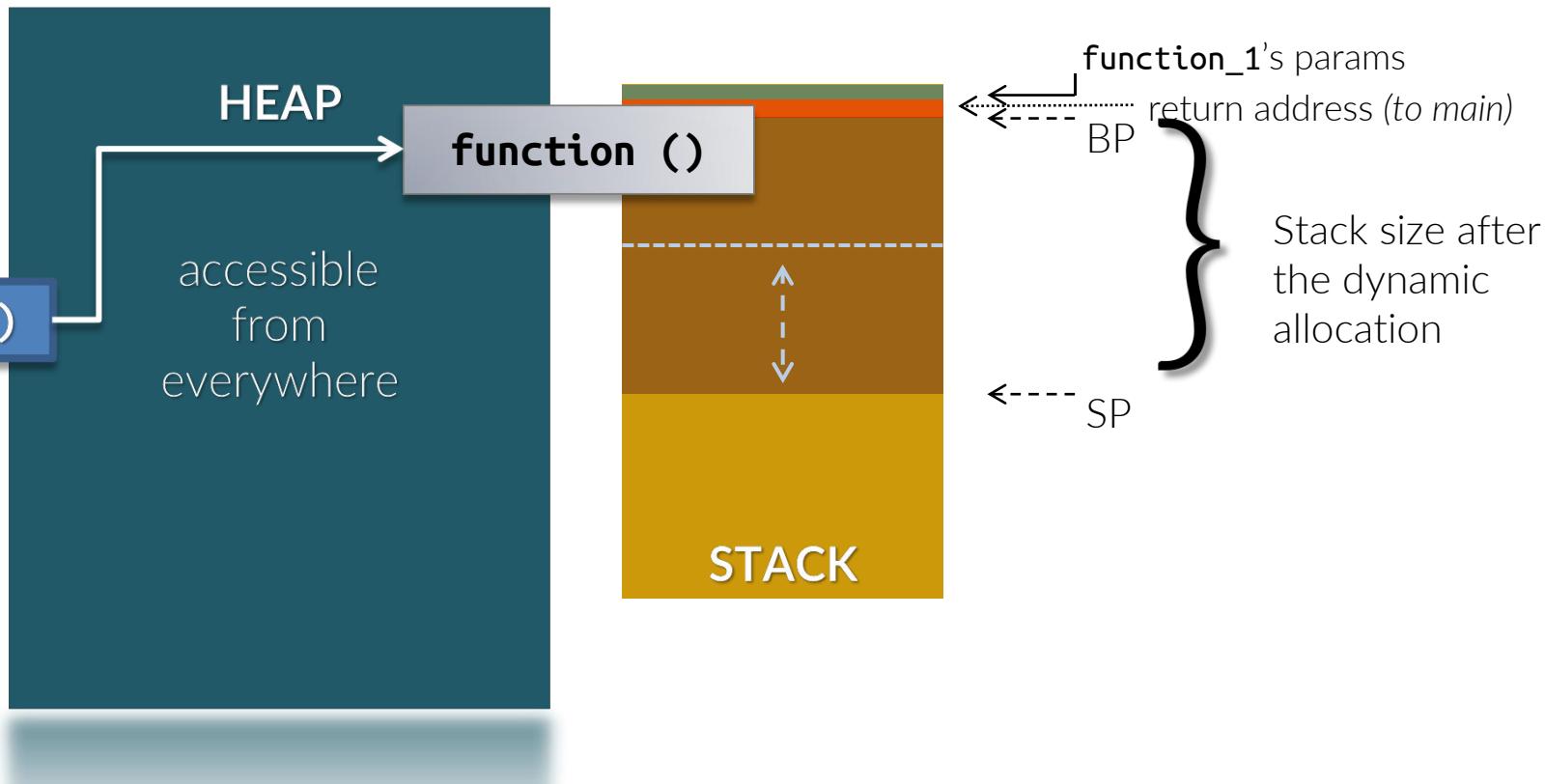


# Dynamic allocation on the stack





# Dynamic allocation on the stack



# Outline



The execution  
and running  
model



Stack & Heap



Worked  
examples



Let the O.S. tell you:

```
> ulimit -s
```

gives you back the stack size in KBs.

Typical values are around 8192 (8 MBs).

This size is the so called *soft limit*, because you can vary it, by using

```
> ulimit -s value
```

The O.S. also sets an *hard limit*, i.e. a limit beyond which you (the user) can not set your stack size.

```
> ulimit -H -s ( or -Hs )
```

gives you back the hard limit, it may well be “unlimited” or “-1”  
*(question: why -1 is a handy value to mean “unlimited”? )*



Compile and run **stacklimit.c** (if needed, depending on what ulimit -s told you, modify the quantity STACKSMASH at the very top)

```
> gcc -o stacklimit stacklimit.c
> ./stacklimit
```

That small code tries to allocate on the stack the maximum amount of bytes available and obviously fails because there are already some local variables already at the bottom of the stack.

Keeping STACKSMASH at its value, slightly enlarge the stack limit using **ulimit** and try it again.





# Not exceeding the stack..

..can be done not so awkwardly than manually setting a variable.

You can get the stack limit from inside your C code, by calling

```
#include <sys/resource.h>
struct rlimit stack_limits;
getrlimit ( RLIMIT_STACK, &stack_limits );
```

and then you can set a different limit (if `stack_limits.rlim_max` is < 0 or larger than the new limit you need) by calling

```
newstack_limits      = stack_limits;
newstack_limits.rlim_cur = new_limit;
setrlimit ( RLIMIT_STACK, &newstack_limits );
```



# Not exceeding the stack..

Let's try by compiling and running `stacklimit_setlimit.c`.

As before, set `STACKSMASH` to the stack size, then compile and run:

```
> gcc -o stacklimit_setlimit stacklimit_setlimit.c  
. /stacklimit_setlimit
```



[SCO/examples\\_on\\_stack\\_and\\_heap/  
stacklimit\\_setlimit.c](#)

Now you should get no more seg fault signals.

Now you should also ask yourself why you do *really* need to enlarge the stack, and about the probability that your design is simply wrong.



Now let's use the third code snippet that you find on github, `stack_use.c`.

Here we set up and access an array in order to measure the time spent on this when :

1. it resides on the stack;
2. it resides on the heap, and we access it through array notation;
3. it resides on the heap, and we access with direct use of pointers.

# Why the next slides ..

Not so long time ago, on a web page not so far away....



*Rumors floated around about why in the hell one should use the heap when the stack is so faster and more optimized..*

*Well, from that question some misunderstanding about the true nature of memory, stack and heap pops out.*

*Of course, thou shall not use neither the stack instead of the heap, nor the other way round.*

*Unless you know what you do.*

*But if you knew, you would not do it, because knowledge is power and “from great powers comes great responsibility” ...*

**Advanced**

...

*And, in any case, the simple codes that follow  
are a fairly good occasion to get acquainted  
with addressing, pointers arithmetic*

*and some basic assembler  
  
for those of you that have never seen such things before*



# Compare stack and heap access

1) 

```
int use_stack ( void ) {
    float on_stack [ N ];
    for (int i = 0; i < N; i++ )
        on_stack[i] = 0;
    return 0; }
```

2) 

```
int use_heap ( void ) {
    float *on_heap = (float*)calloc( N, sizeof(float) );
    for (int i = 0; i < N; i++ )
        on_heap[i] = 0;
    free(on_heap); return 0; }
```

3) 

```
int use_heap_2 ( void ) {
    float *on_heap = (float*)calloc( N, sizeof(float) );
    float *ptr = on_heap-1, *stop = on_heap + N;
    for ( ; ++ptr < stop; )
        *ptr = 0;
    free(on_heap); return 0; }
```

*STILL ON HEAP  
BUT MORE OPTIMIZED*



# Is the stack faster than the heap?

The correct answer is: “it depends”. On you, mostly.

The fundamental difference between the stack and the heap is that the former is accessed through the SP/BP registers, i.e. by static offsetting an address that is *already* resident in a register.

In the following slides we will have a look on how the compiler translates the previous loops in machine instructions.

Examples of some common assembly instructions

CMP	DWORD PTR -168[rbp], 0	#cmp DW with 0
MOVSD	QWORD PTR -156[rbp], xmm0	#mov QW from reg xmm0 to mem
MOV	rax, QWORD PTR -048[rbp]	#mov QW from mem to reg



access an array on the stack

Note:

- $-024[rbp]$  is the position on the stack of the loop counter  $i$ , expressed relatively to  $rbp$  (24 bytes before it in memory)
- $-1024[rbp]$  is the position on the stack of the beginning of the array, expressed relatively to  $rbp$
- $-1024[rbp+rax*4]$  is an element of the array (integers are 4bytes long)

```
MOV      eax, DWORD PTR -024[rbp]          # mov the value if i in a register
CDQE
MOVSS    DWORD PTR -1024[rbp+rax*4], 0     #mov 0 to the i-th entry of array
```



# Is the stack faster than the heap?

To access a dynamically allocated array through pointer-offsetting things look like -00 that:

```
array = (..*)calloc( .., ..);
for ( i = 0; i < N; i++ )
    array[i] = 0;
```

MOV	eax, DWORD PTR -148[rbp]	#load the counter i in reg eax
CDQE		
LEA	rdx, 0[0+rax*4]	#calculate the position in bytes # of the i-th element
MOV	rax, QWORD PTR -36[rbp]	#load array address in rax
ADD	rax, rdx	#add the position to the begin # of the array
MOVSS	DWORD PTR [rax], xmm0	#mv 0 to array[i] # note: xmm0 is a register that # was previously initialized to 0



# Is the stack faster than the heap?

However, if you access the dynamically allocated memory in a different way, you may generate much less instructions:

-00

```
array = (..*)calloc( .., ..);
.. register *ptr = array;
.. register *stop = array + N;

for ( ; ++ptr < stop; )
    *ptr = 0;
```

MOV	rbx, QWORD PTR -96[rbp]	#load the starting point of the array # into rbx (basically rbx behaves as a pointer)
..		
MOVSS	DWORD PTR [rbx], xmm0	#mv 0 directly to the memory pointed by rbx
ADD	rbx, 4	#directly increase the pointer

This is the actual loop





So, let's resume.

Accessing an array on the stack:

```
MOV      eax, DWORD PTR -024[rbp]
CDQE
MOVSS   DWORD PTR -1024[rbp+rax*4], 0
ADD     DWORD PTR -024[rbp], 1
```

Accessing an array on the heap in the fastest way:

```
MOVSS   DWORD PTR [rbx], xmm0
ADD    rbx, 4
```

It seems that the heap can be as fast as the stack. It depends on how things are set-up.



# Compare stack and heap access

Let's discuss what happens compiling(\*) and executing `stack_use.c`.

```
> gcc -o stack_use stack_use.c  
./stack_use
```



SCO/examples\_on\_stack\_and\_heap/  
`stack_use.c`

Try with and without compiler's optimization.

Note that `function_3()` is accessing the stack of `function_2()`.

The bravest among you may want to inspect assembly outputs:

```
> gcc -g -S -fverbose-asm [-O2] -o stack_use.s stack_use.c
```

or

```
> gcc -g [-O2] -Wa,-ahld -fverbose-asm stack_use.c > stack_use.s
```

(\*) the compiler will warn you about "function returns address of local variable". That's on purpose: check `function_1()` to understand the issue. In the light of what has been said before, you should immediately catch the point.



Without compiler's optimizations, you should find that

$$T_{\text{STACK}} < T_{\text{HEAP}} < T_{\text{HEAP-FAST}}$$

Whereas, with compilers' optimization you should find that

$$T_{\text{STACK}}^{O3} = T_{\text{HEAP}}^{O3} < T_{\text{HEAP-FAST}}^{O3} < T_{\text{HEAP-FAST}}$$

Actually, the compiler in O3 case generates basically the code that we wrote for the heap-fast case.

Note: the fact that  $T_{\text{HEAP-FAST}}^{O3}$  is slower than the other two options is due to the initialization phase that is before the loop itself. You'll learn more about this when we'll talk about memory aliasing.



Generate the most basic assembler listing (creates `foo.s`)

```
%> gcc foo.c -S
```

Add some useful details

```
%> gcc foo.c -S -fverbose-asm
```

A more detailed view:

```
%> gcc foo.c -fverbose-asm -Wa,-adhln=foo.detailed.s
```

`-Wa` *passes options to assembler*

`%> as -help`

`-a[sub-options]` *turn on listings*

c	omit false conditionals
d	omit debugging directives
g	include general info
h	include high-level source
l	include assembly
m	include macro expansions
n	omit forms processing
s	include symbols
=FILE	list to FILE (must be last suboption)



The following slides contain some comments about the code that you find in the github repository `day5/examples_on_stack_and_heap/` folder:

`stacksmah_simple.c`

> Very simple example of stack smashing

`stacklimit.c`

> Trying to use too much space in the stack

`stacklimit_setlimit.c`

> Adapting the stack from within the code

`stack_use.c`

> Different ways to access the memory in the stack and the heap + accessing the stack of callers functions

`understanding_the_stack.c`

> Some additional examples about the stack in functions

that's all, have fun

"So long  
and thanks  
forall the fish"