The Execution Model, the Stack & the Heap

Luca Tornatore - I.N.A.F.

ISTITUTO NAZIONALE DI ASTROFISICA INAF

DATA SCIENCE & ARTIFICIAL INTELLIGENCE

SCIENTIFIC & DATA-INTENSIVE COMPUTING

2023-2024 @ 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).

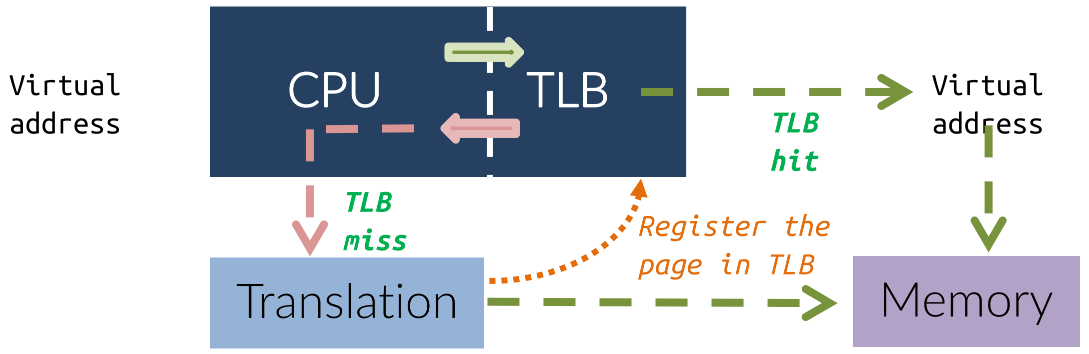
The memory model

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



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, thy usually stays at around ~1-2%

Outline

Stack & Heap

Worked examples

The stack and the Heap

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).

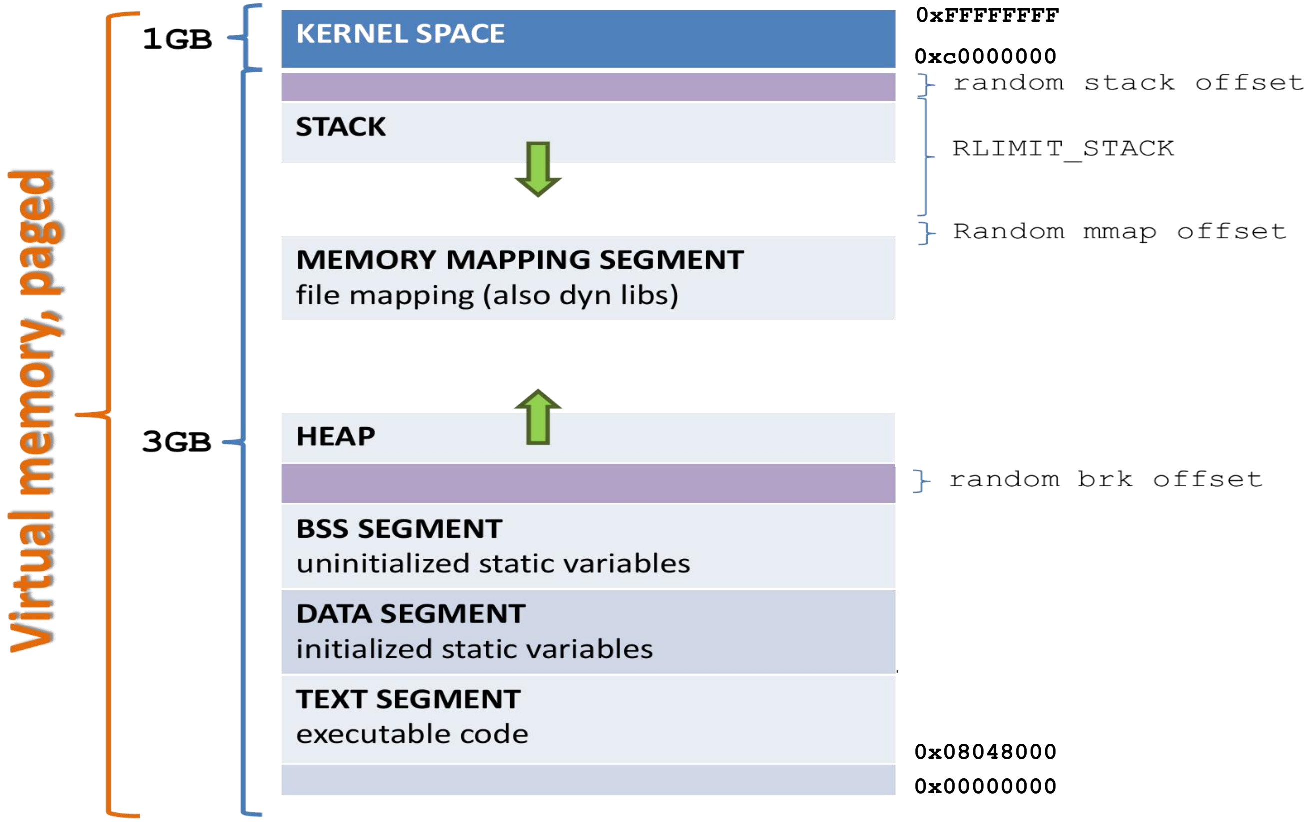
In addition to housing the global variables, its most natural use is to hold big amount of data, whether or not they have a limited scope, or, in any case, the data whose amount is known only at run-time.

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.

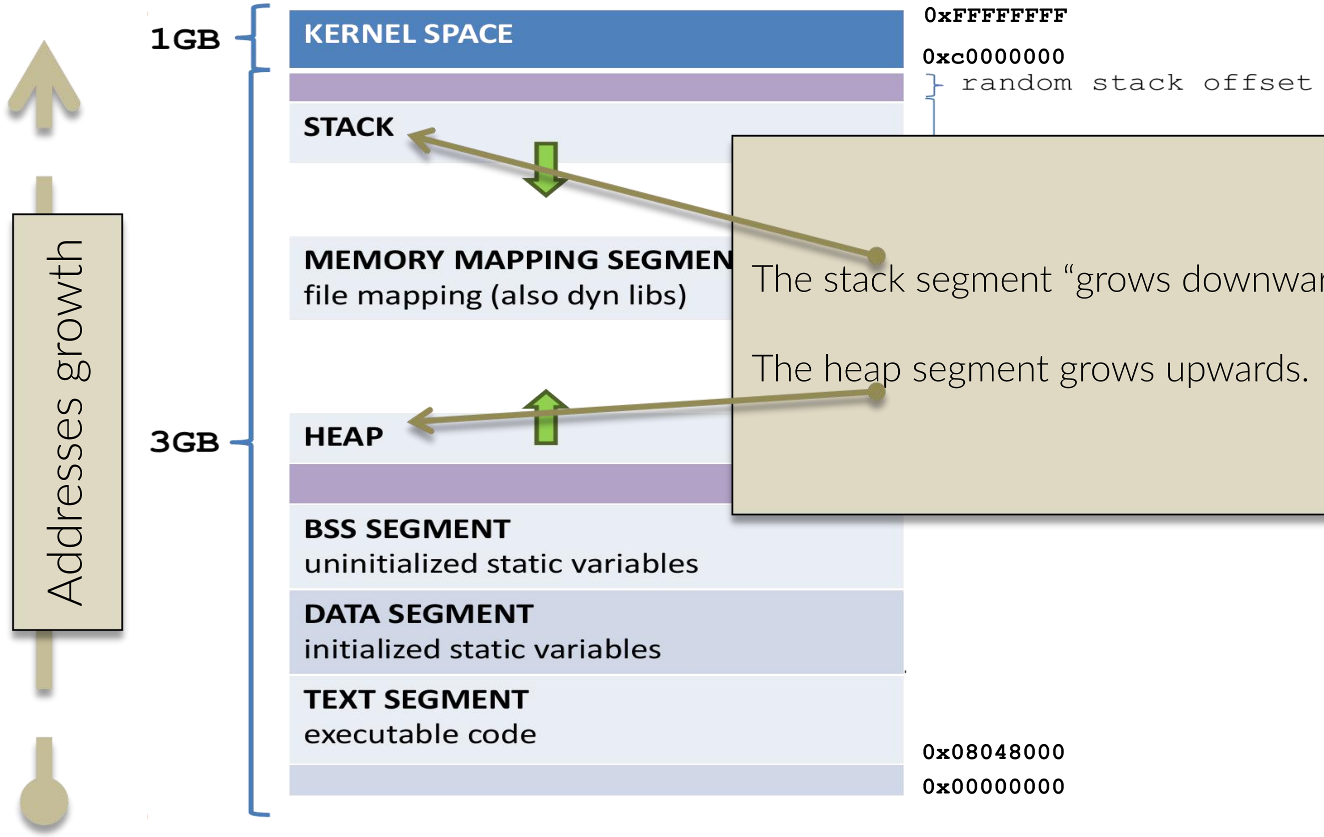
Note:
4GB was the limit in 32bits systems.
In 64bits systems, the total amount of addressable memory is much larger



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



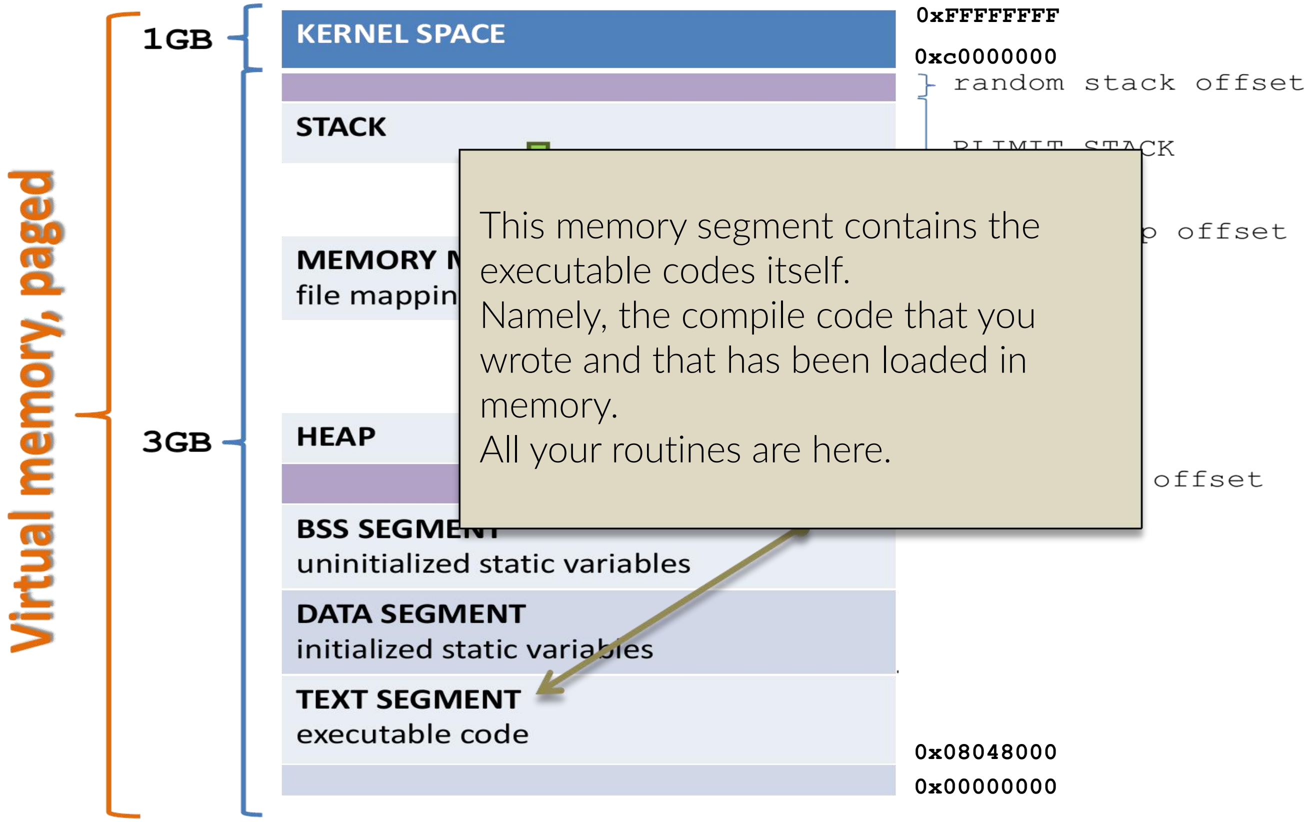
The stack segment "grows downwards".

The heap segment grows upwards.

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



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.

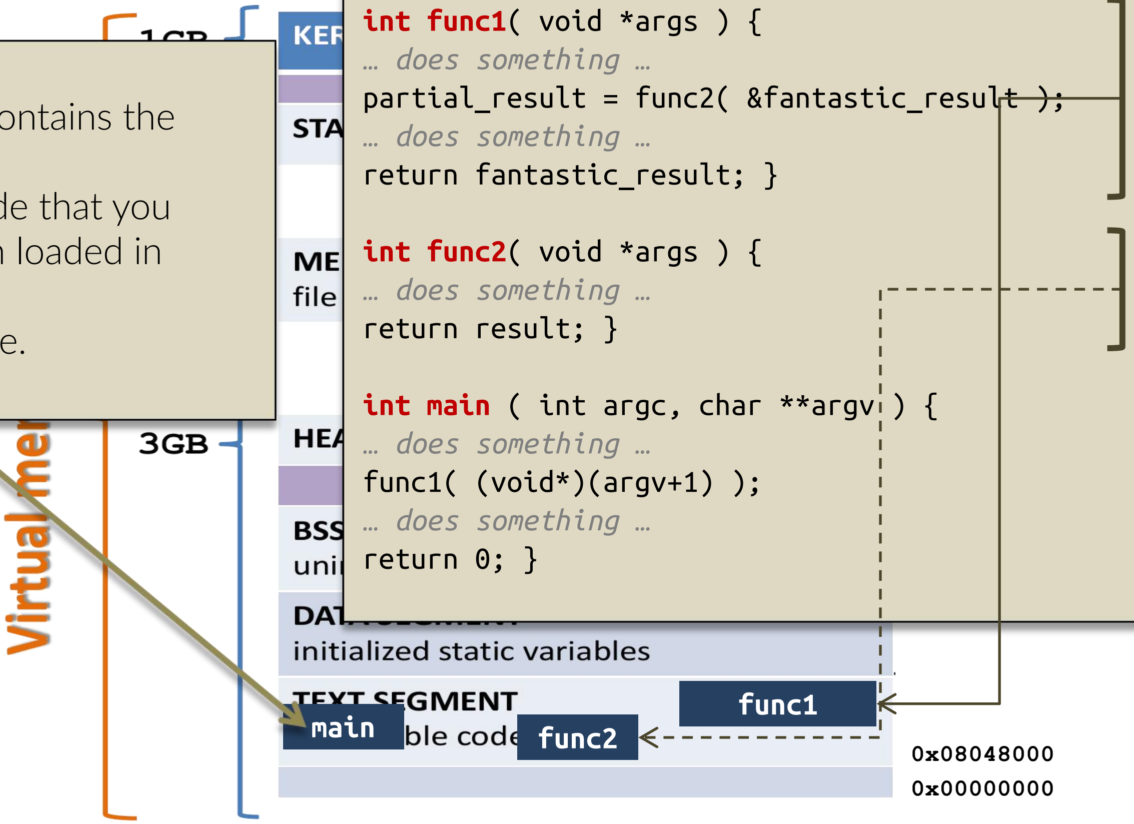
The standa

A
e
re
e
m

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.

N

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

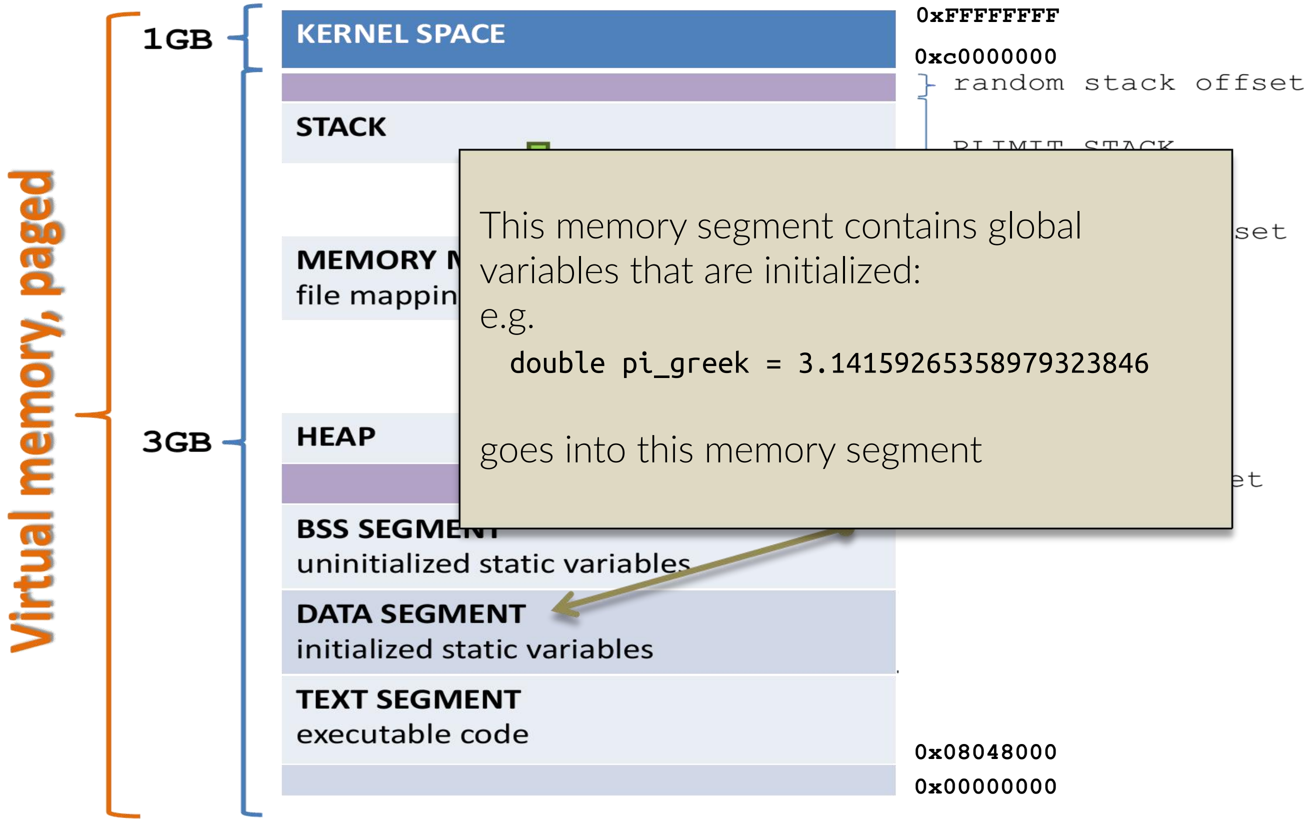


```
#include <stdlib.h>
#include …

int func1( void *args ) {
… does something …
partial_result = func2( &fantastic_result );
… does something …
return fantastic_result; }

int func2( void *args ) {
… does something …
return result; }

int main ( int argc, char **argv ) {
… does something …
func1( (void*)(argv+1) );
… does something …
return 0; }
```

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



This memory segment contains global variables that are initialized:
e.g.

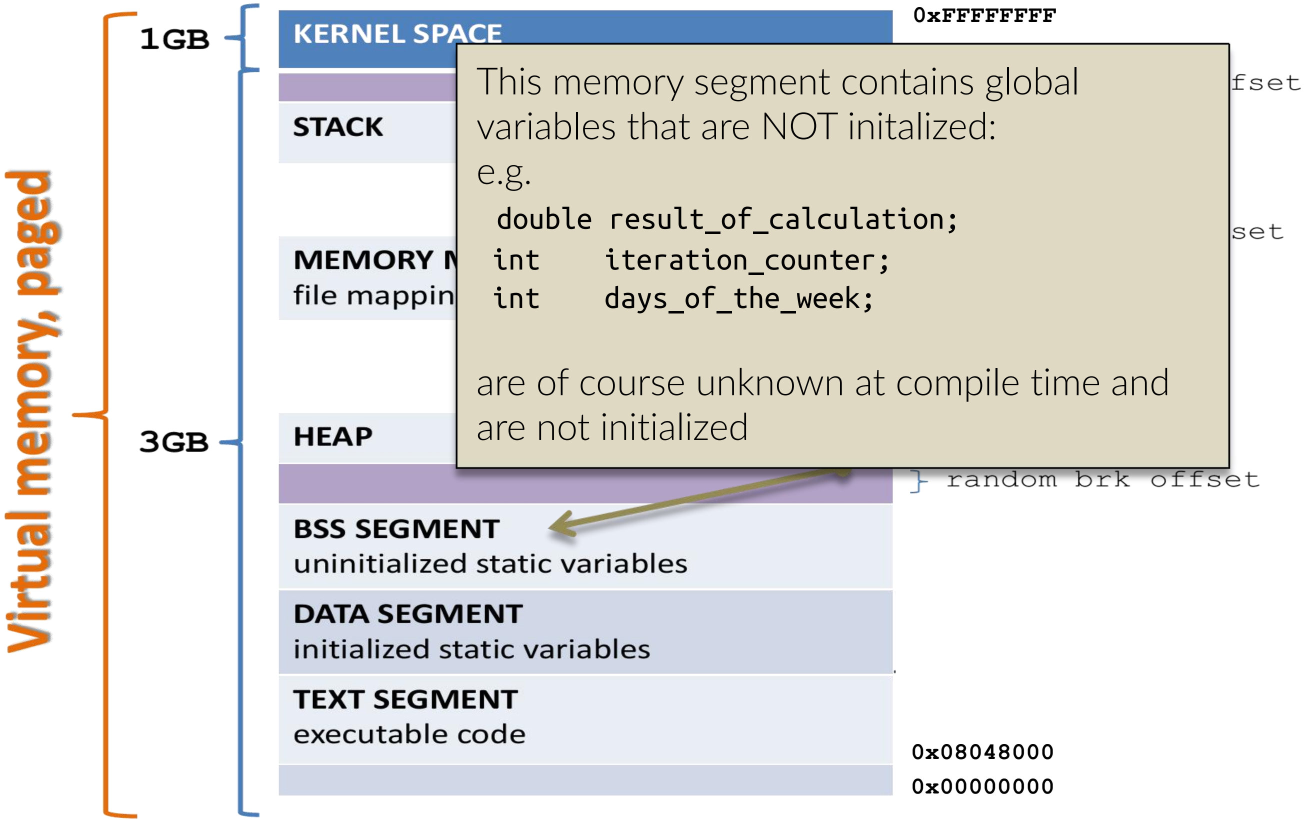
```
double pi_greek = 3.14159265358979323846
```

goes into this memory segment

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



This memory segment contains global variables that are NOT initalized:
e.g.

```
double result_of_calculation;
int    iteration_counter;
int    days_of_the_week;
```

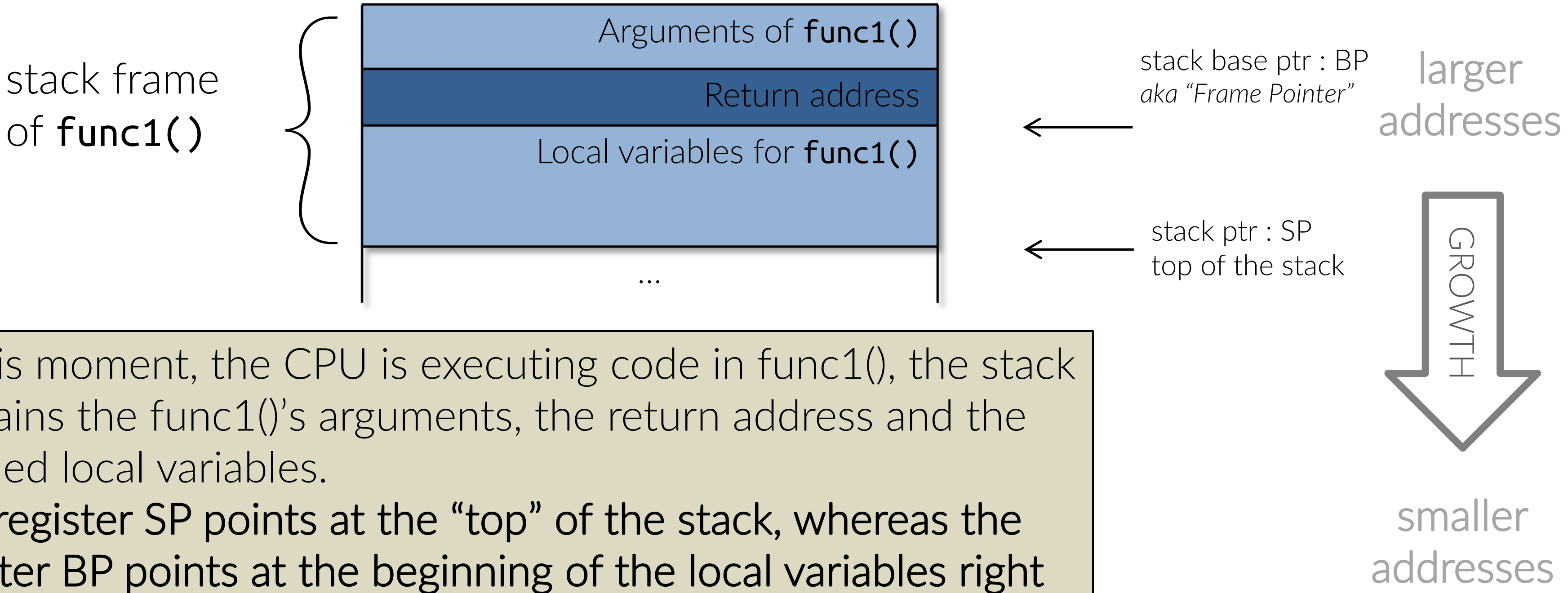
are of course unknown at compile time and are not initialized

Zoom in the stack details

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

```
func1 ( )
{
    …
    func2 ( )
    …
}
```

Zoom in the stack details

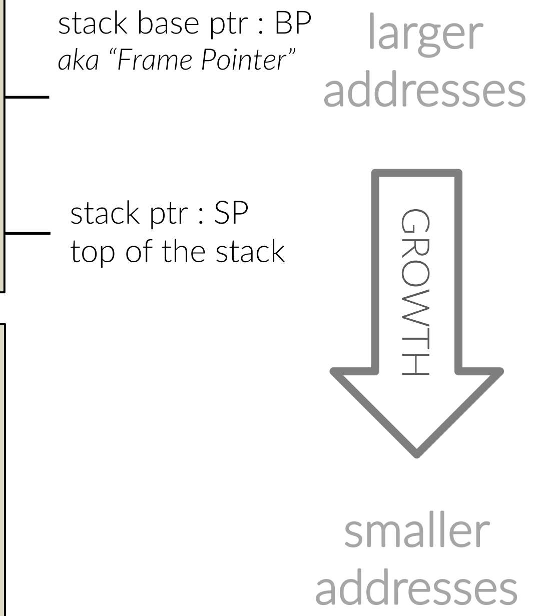


In this moment, the CPU is executing code in func1(), the stack contains the func1()'s arguments, the return address and the needed local variables.
The register SP points at the "top" of the stack, whereas the register BP points at the beginning of the local variables right after the return address address.

Zoom in the stack details

Actually the first *N* arguments that fits in 64bits (numerical values and pointers) are passed in registers. *N* is system-dependent, it is generally of the order of 6 to 8.
Arguments that exceed 64bits or beyond the *N*th arguments are passed through the stack

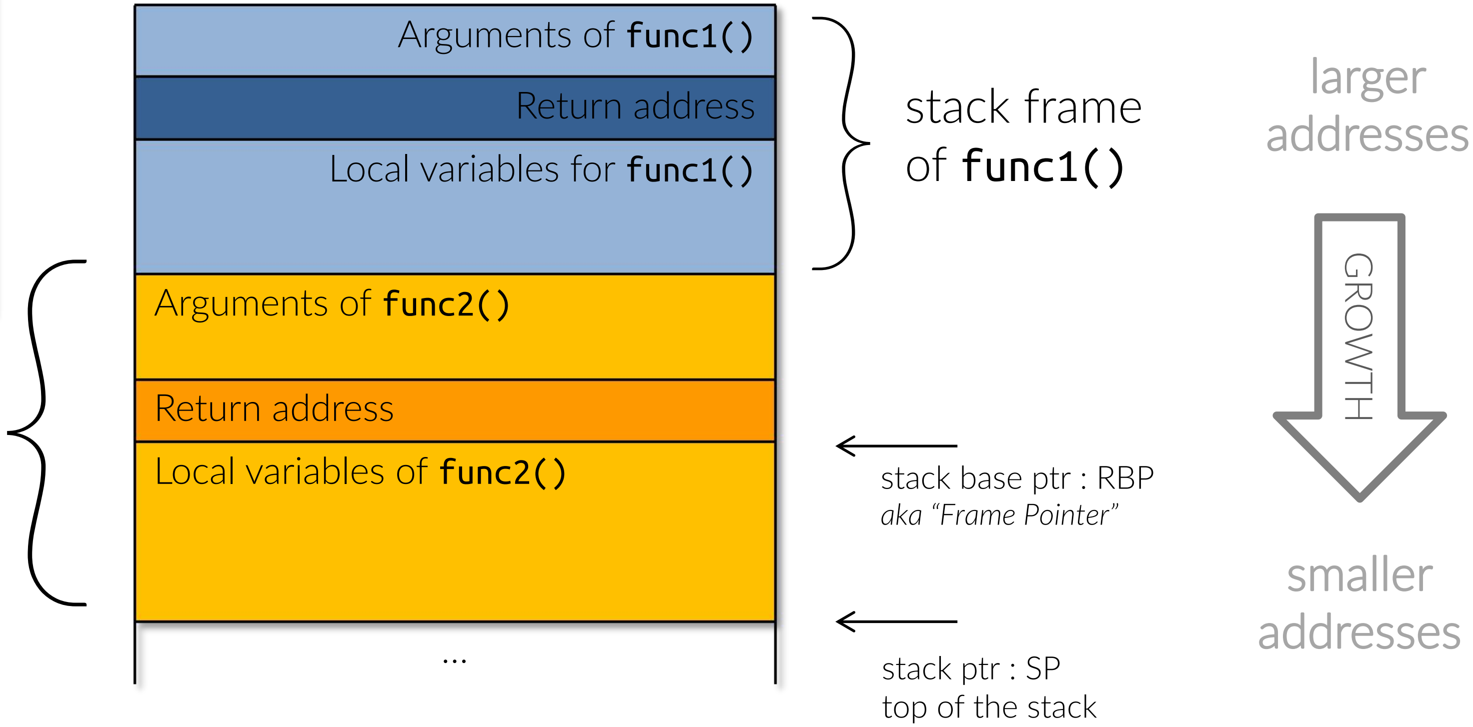
In this moment, the CPU is executing code in func1(), the stack contains the func1()'s arguments, the return address and the needed local variables.
The register SP points at the "top" of the stack, whereas the register BP points at the beginning of the local variables right after the return address address.



Zoom in the stack details

Example:
func1() calls func2()

```
func1 ( )
{
    …
    func2 ( )
    …
}
```



Zoom in the stack details

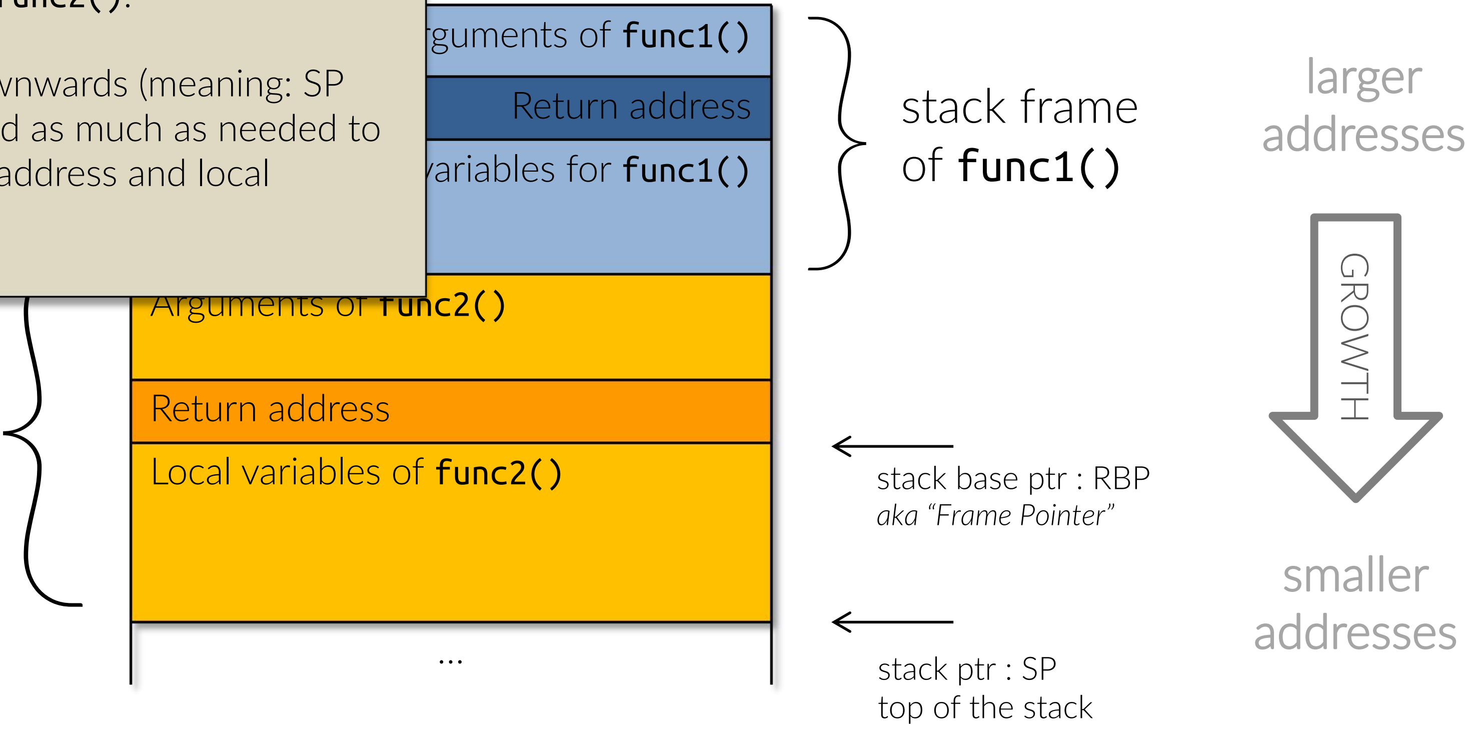
Example:

fu

```
f
{


}
```

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)



Zoom in the stack details

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,

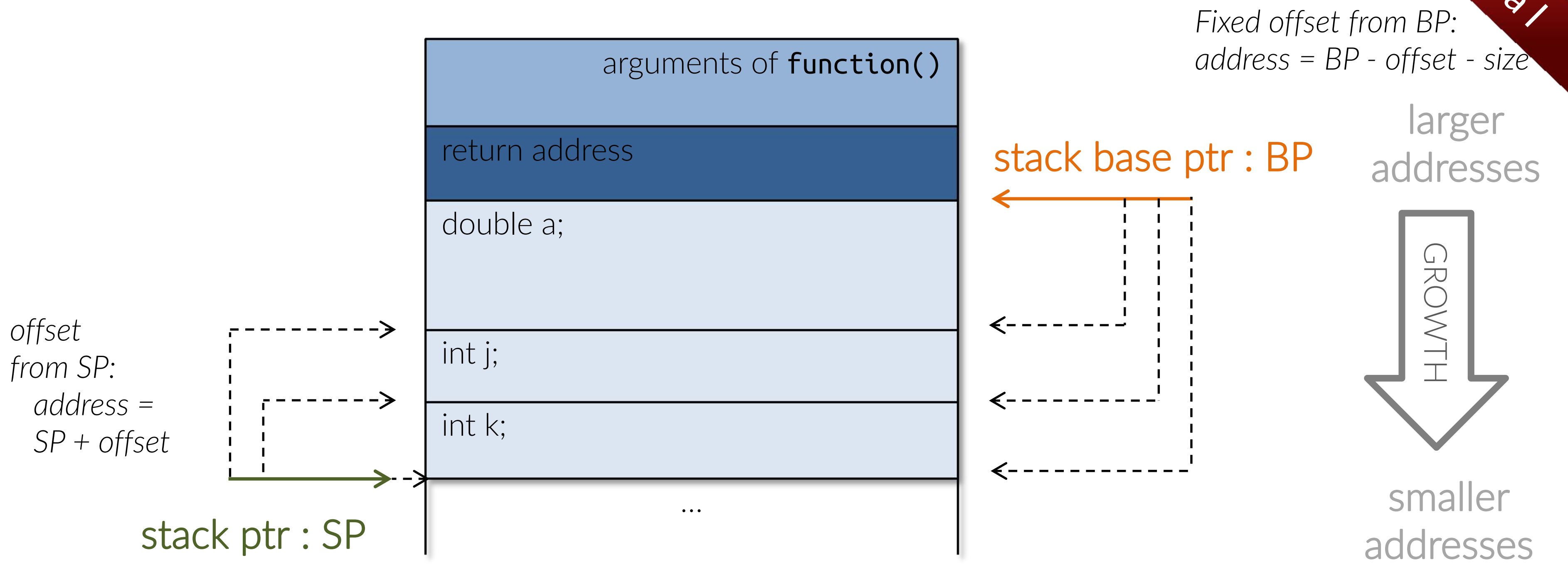
```
void function( void ) { int i; i = 1; return; }
```

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 ?

Optional

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 varable'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 *`Materials/A_note_on_Endiansim.pdf`*

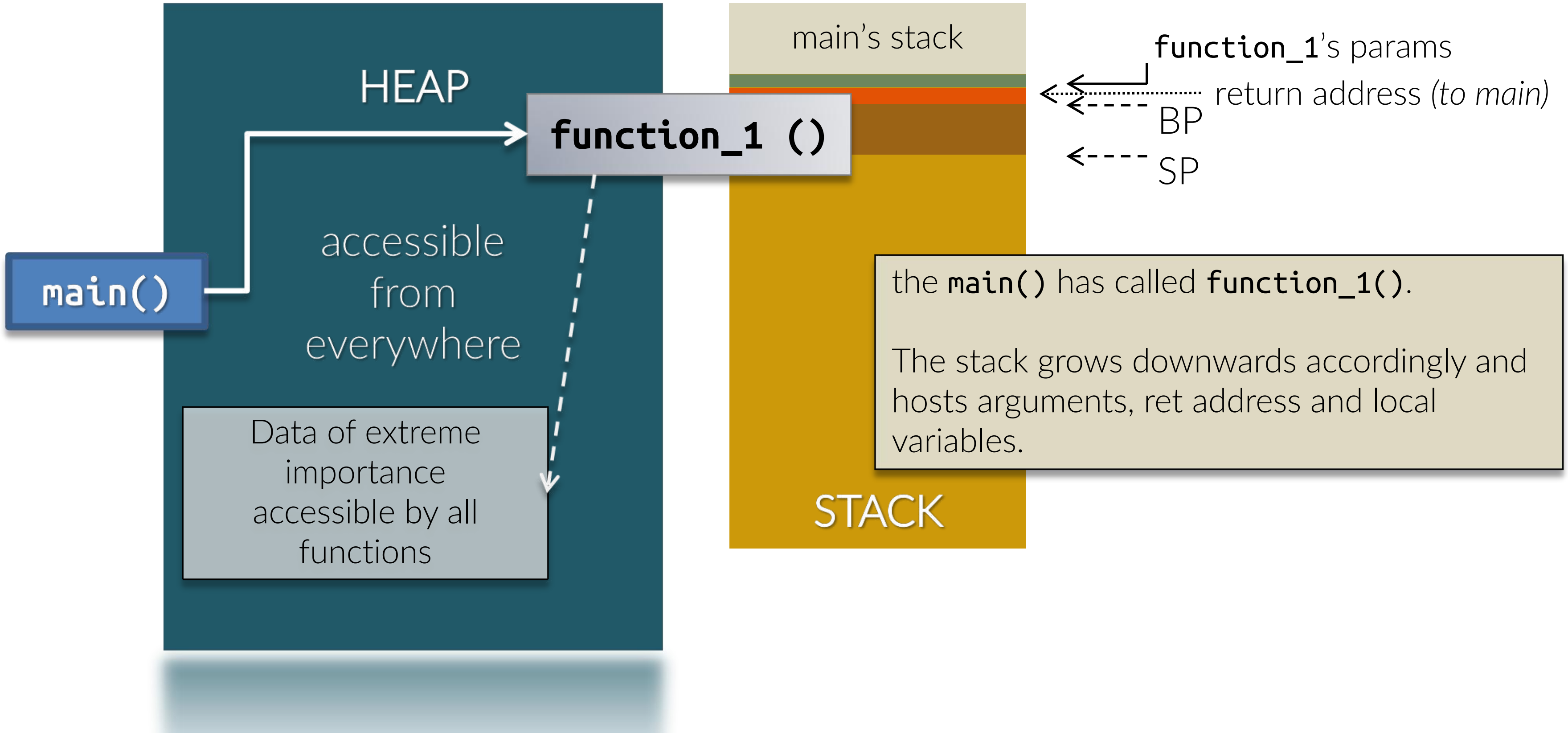
```
int i = 1;
for( int j = 0; j < 4; j++ )
    {
      printf("i = %d\n"
             "%p: %d -- %p: %d -- %p: %d -- %p: %d\n",
             i,
             (char*)&i, *(char*)&i,
             ((char*)&i+1), *((char*)&i+1),
             ((char*)&i+2), *((char*)&i+2),
             ((char*)&i+3), *((char*)&i+3)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

Different in scope

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;}
```

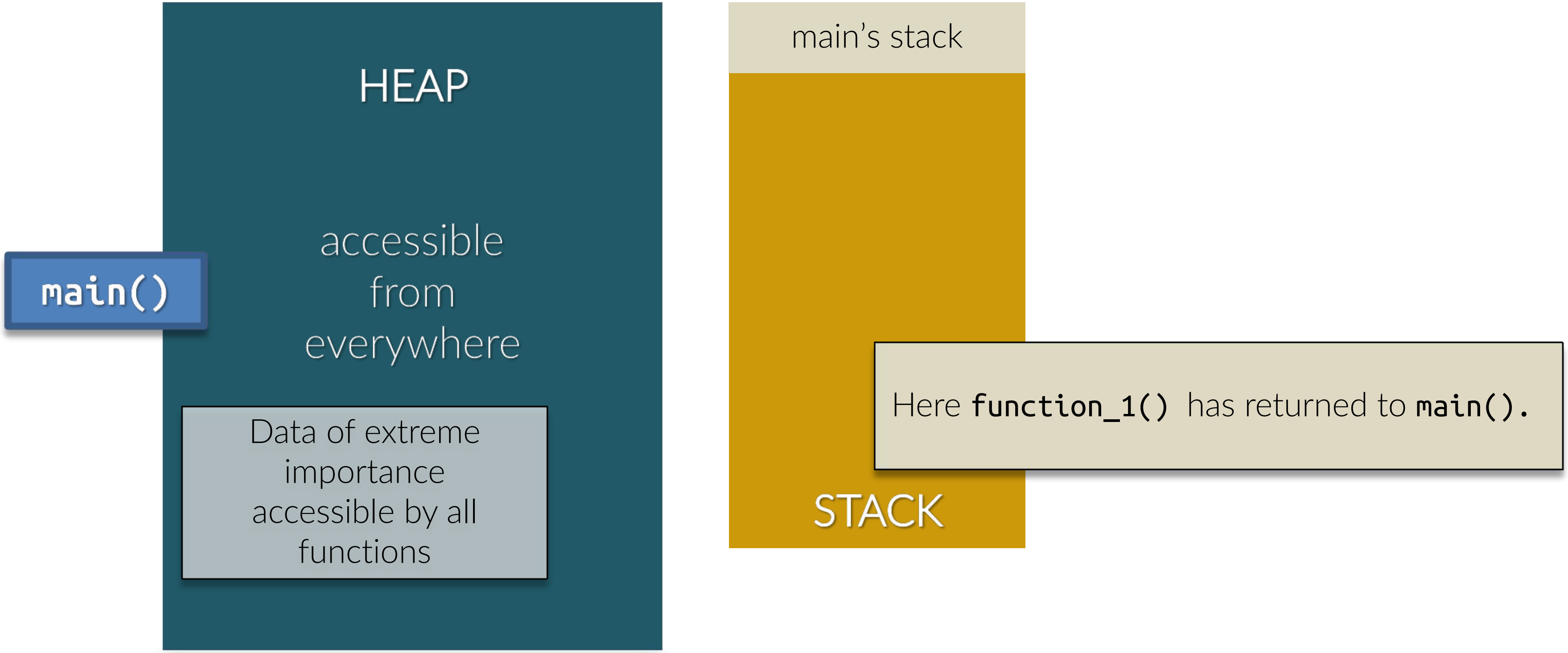
Different in scope



the **main()** has called **function_1()**.

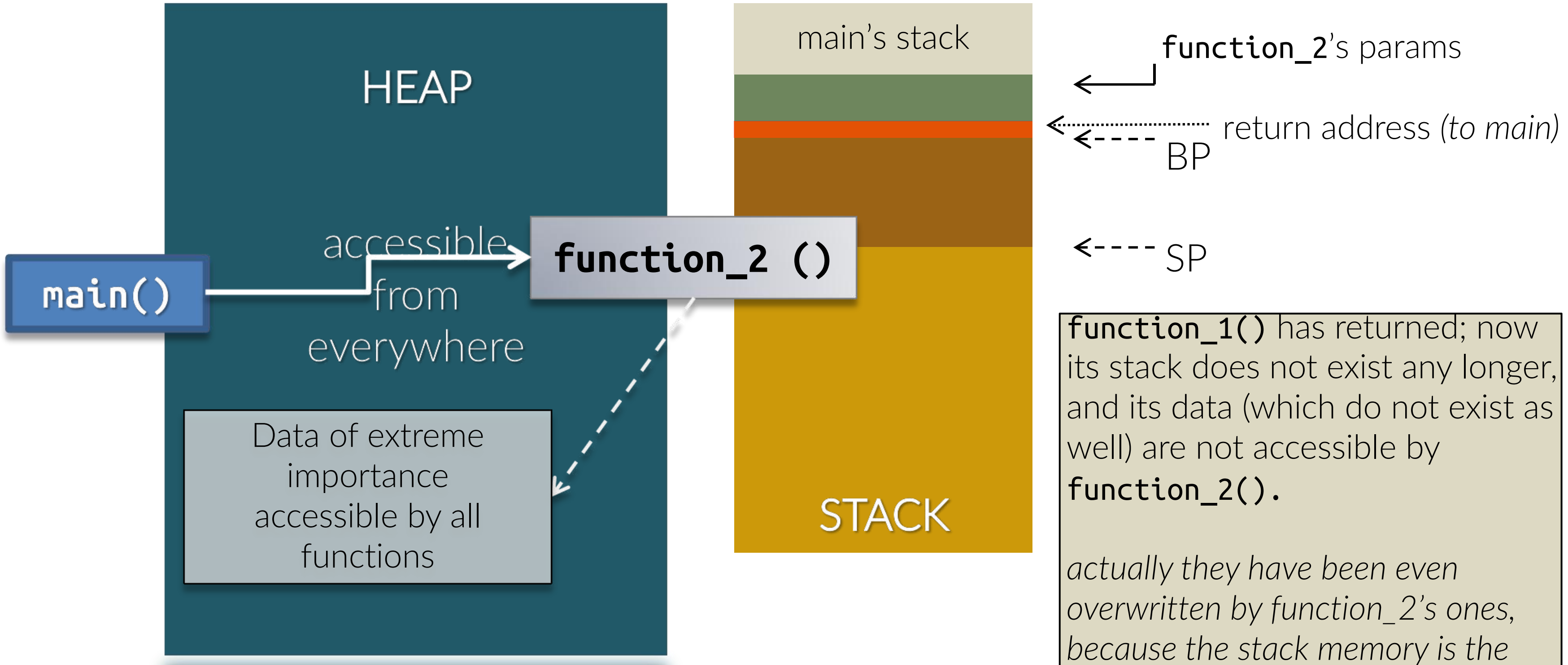
The stack grows downwards accordingly and hosts arguments, ret address and local variables.

Different in scope


HEAP
accessible from everywhere
Data of extreme importance accessible by all functions
main()
main's stack
STACK

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

Different in scope



`function_1()` has returned; now its stack does not exist any longer, and its data (which do not exist as well) are not accessible by `function_2()`.

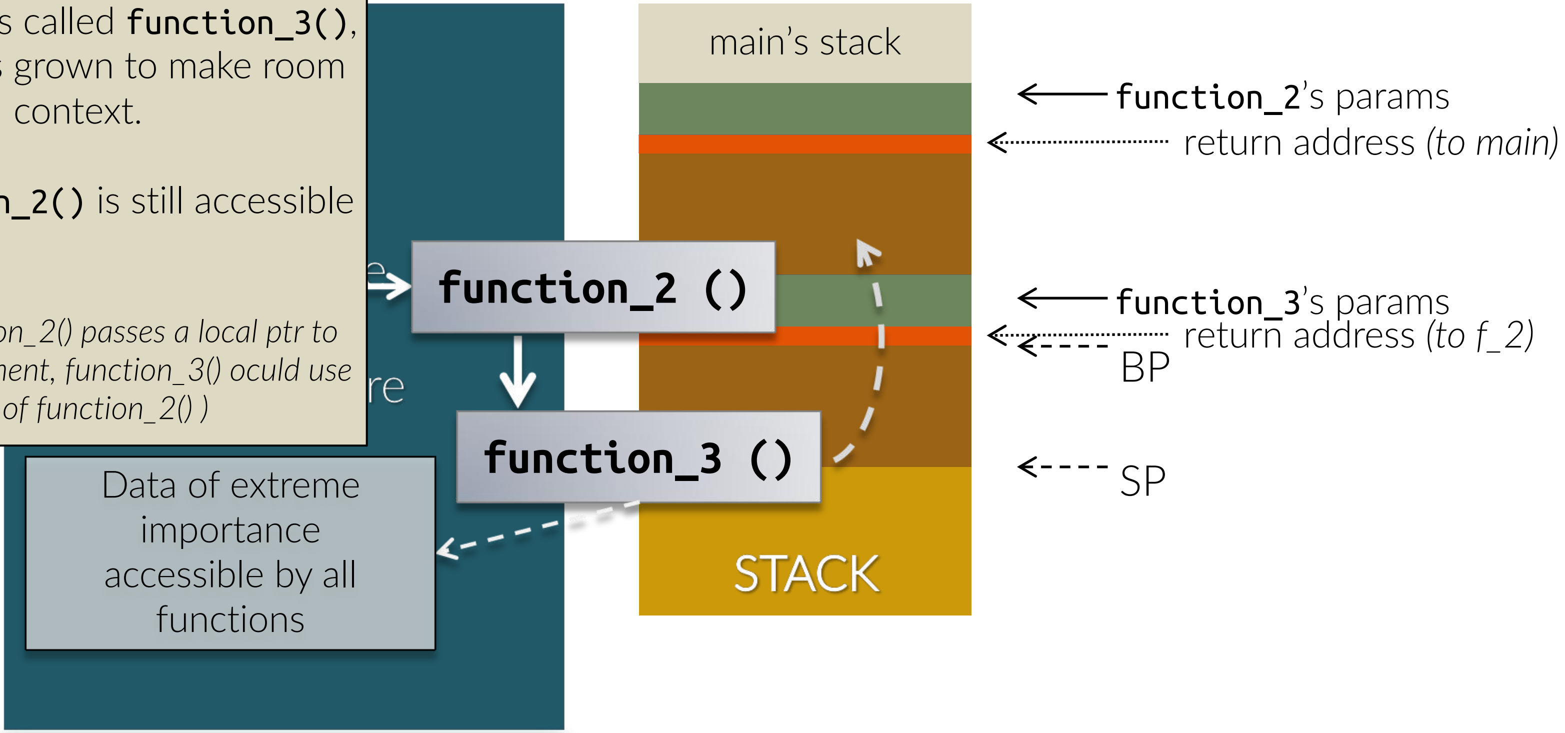
actually they have been even overwritten by function_2's ones, because the stack memory is the same.

Different in scope

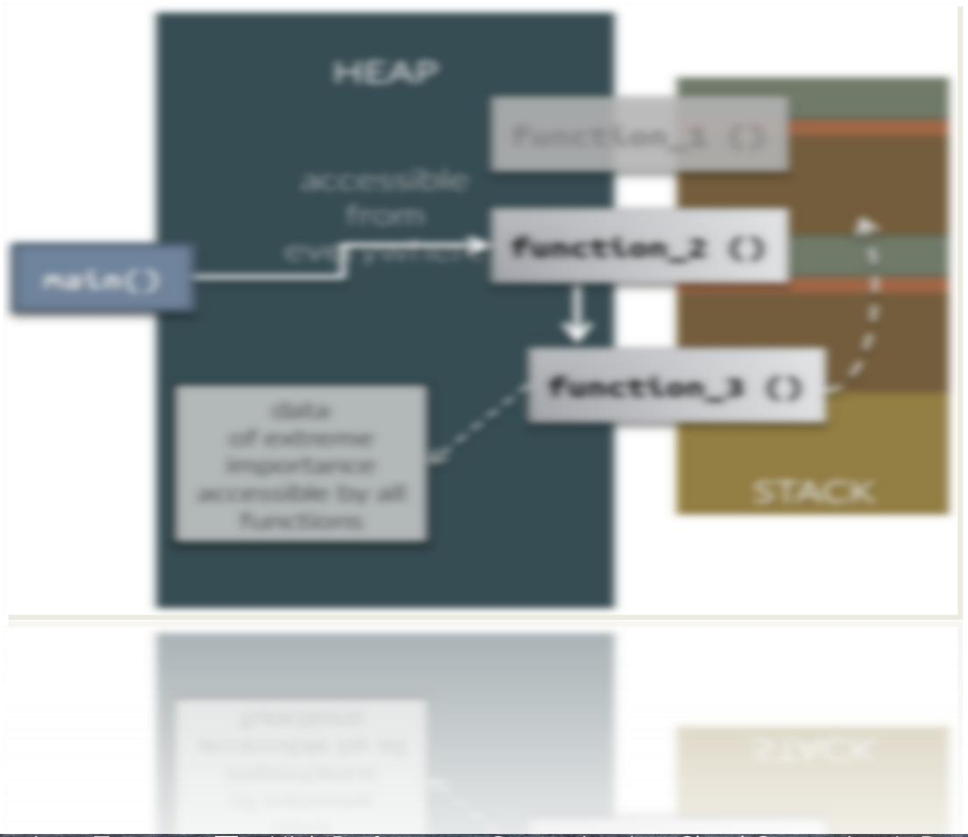
function_2() has called **function_3()**, and the stack has grown to make room for **function_3()** context.

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() oculd use it to access the stack of function_2())



Different in scope


HEAP
function_1 ()
accessible from everywhere
main()
function_2 ()
function_3 ()
data of extreme importance accessible by all functions
STACK

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.

Note: its size is meant to allow a **deep call stack** (for instance, for **recursion**)

Dynamic allocation on the stack

Optional

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:

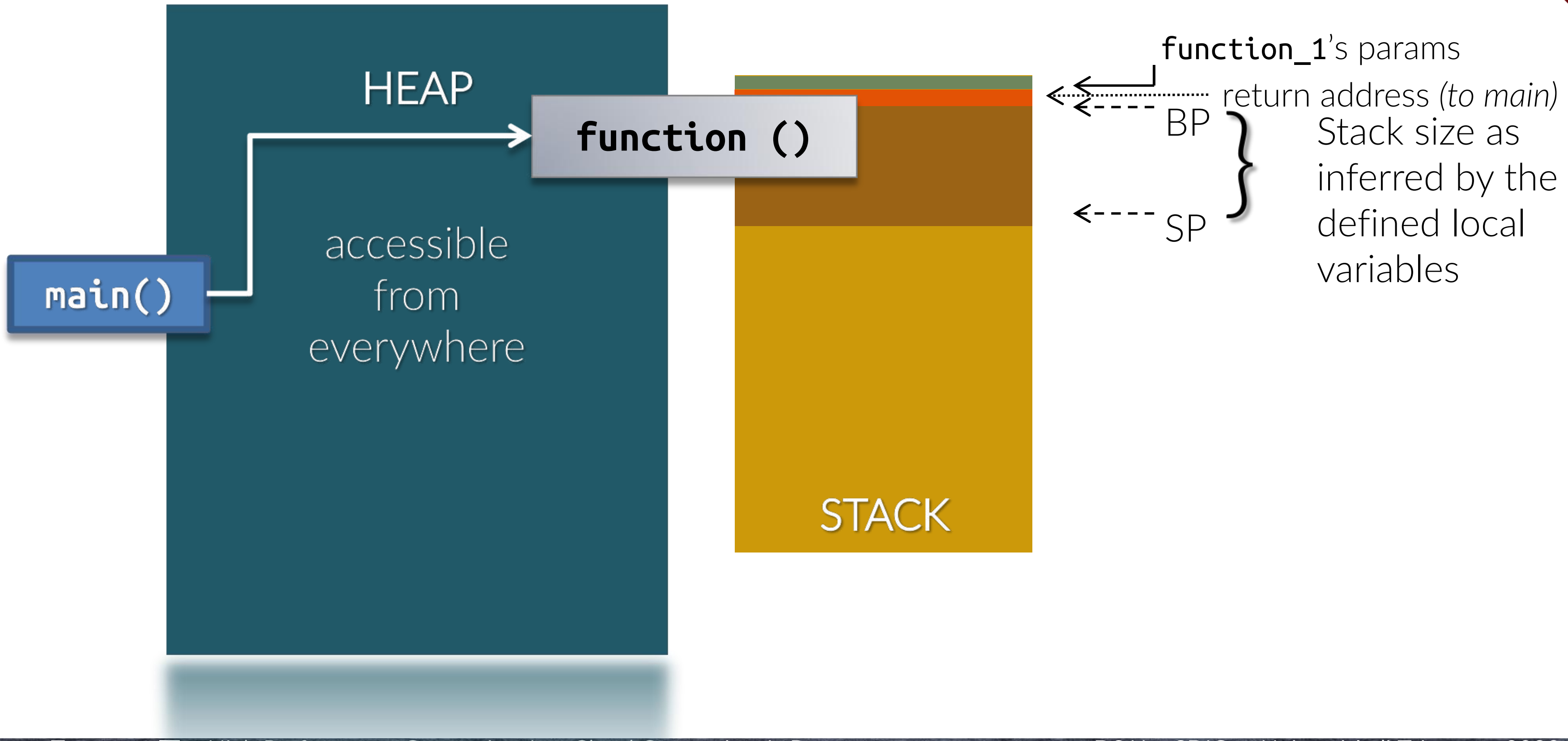
```
int some_function_whatsoever ( int n, double *, … )
   {
       // n is supposed to be not that  big

       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.

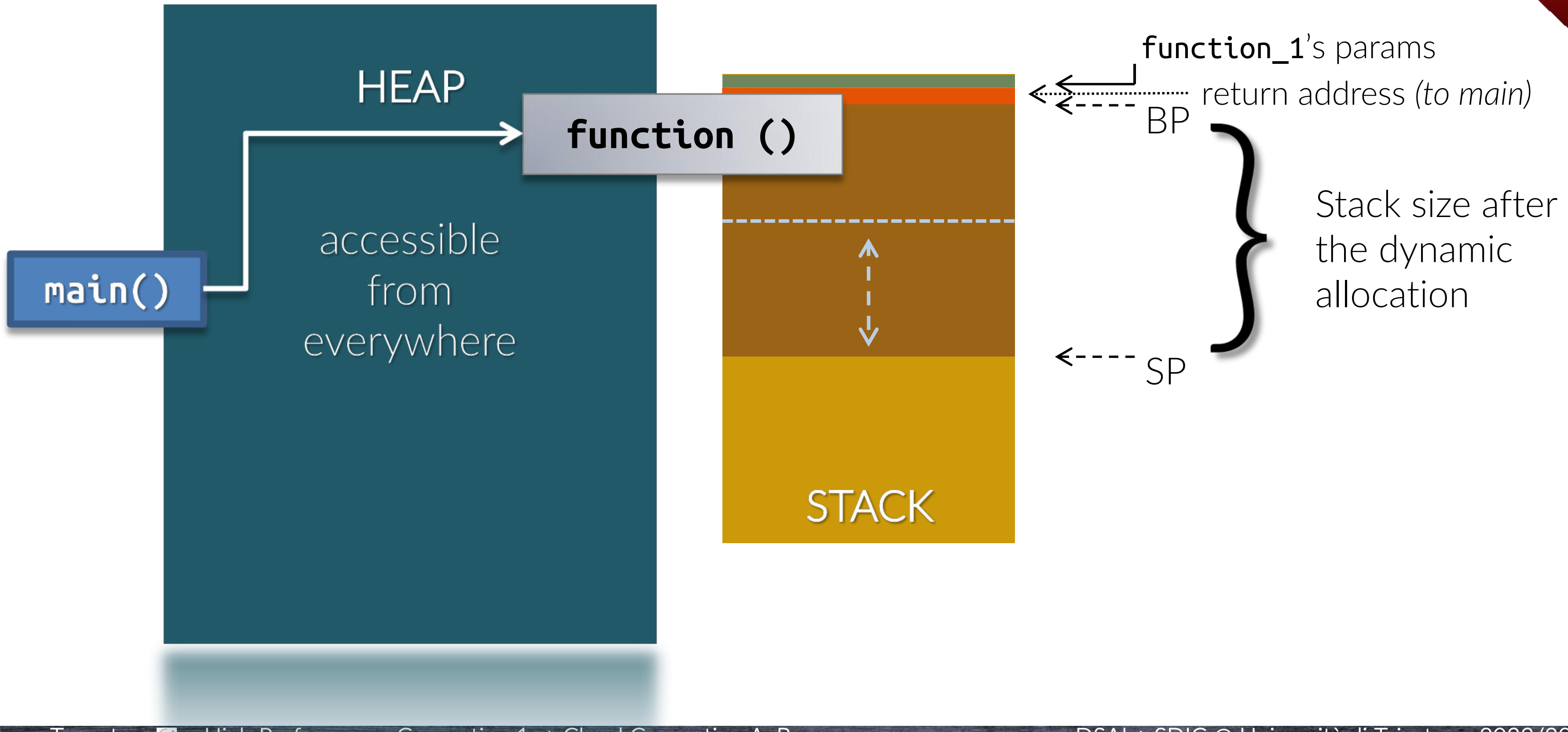
Have a look at **`man alloca`** for more details.

Dynamic allocation on the stack

Optional


HEAP
function ()
main()
accessible from everywhere
STACK
function_1's params
return address (to main)
BP
SP
Stack size as inferred by the defined local variables

Dynamic allocation on the stack



Outline

The execution and running model

Stack & Heap

Worked examples

How large is the stack ?

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 you stack size.

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

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

Exceeding the stack..

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.

`SCO/examples_on_stack_and_heap/stacklimit.c`

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.

that's all, have fun