The stack & the Heap

RUN MODELS:What happens to code wh

Luca Tornatore - I.N.A.F.

"**Foundation of HPC" course**

DATA SCIENCE &
SCIENTIFIC COMPUTING

2019-2020 @ Università di Trieste

Why this lecture ..

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" ...

Outline

The execution and running model

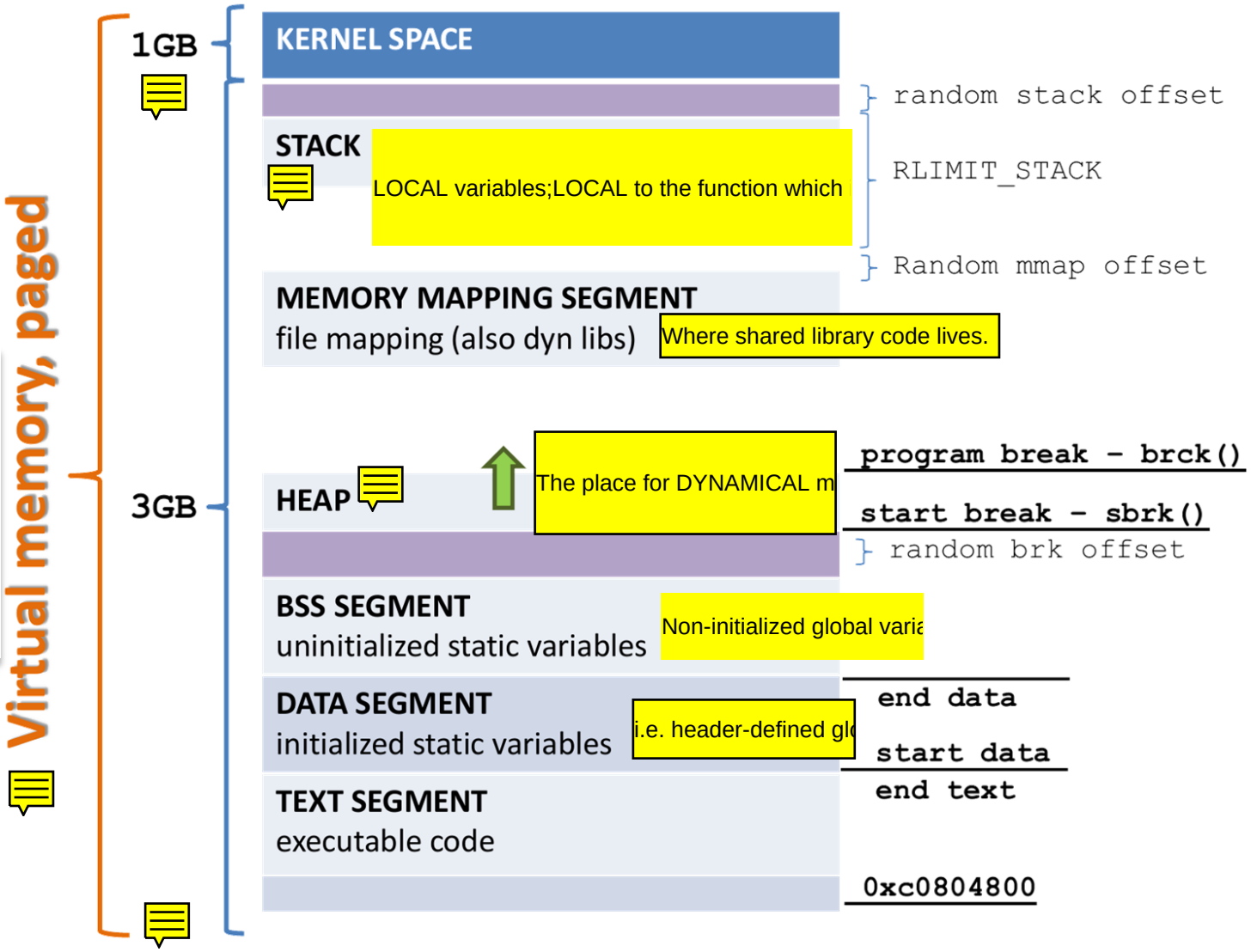
Stack & Heap

Worked examples

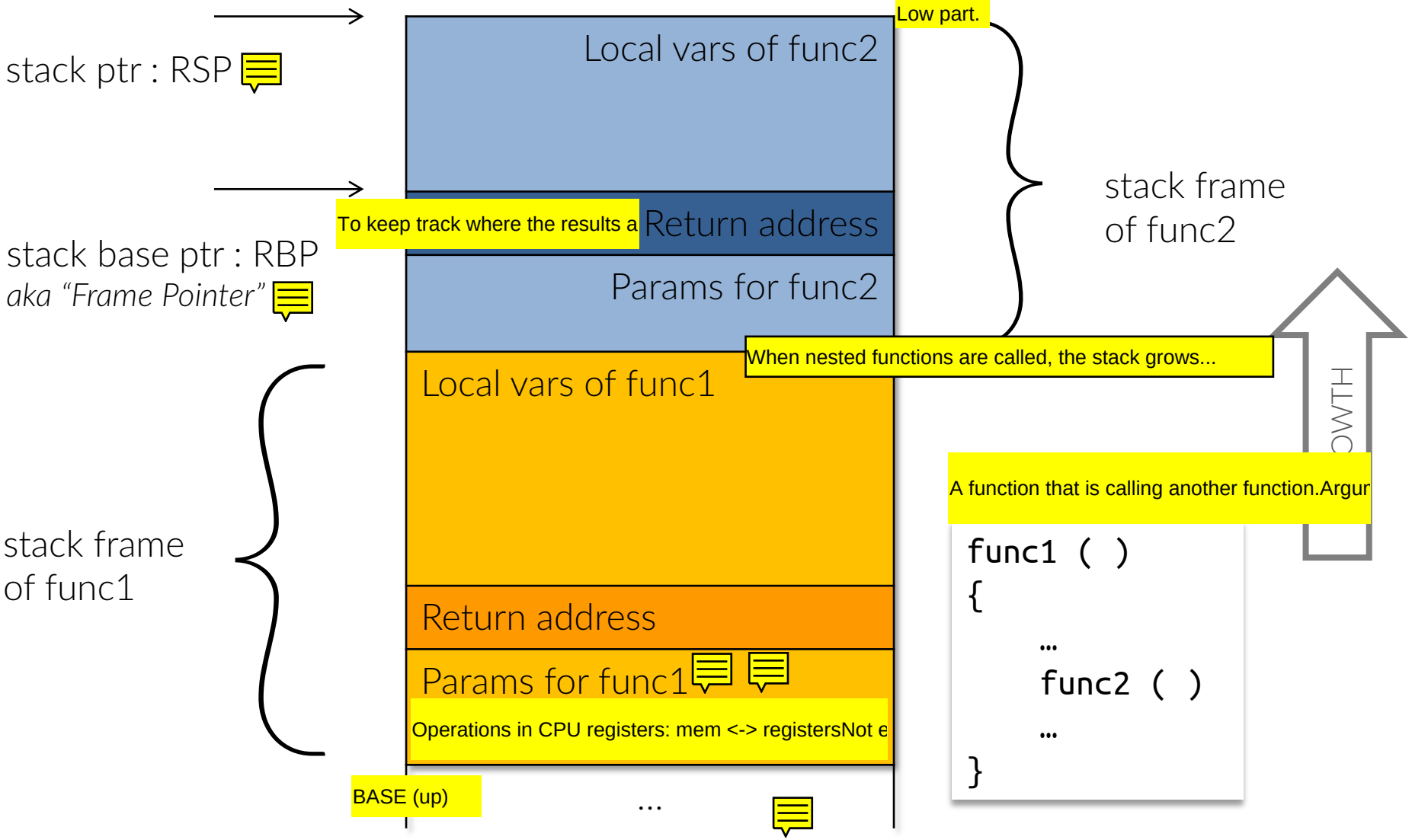
The standard execution model

What does happen when "code" is executed?

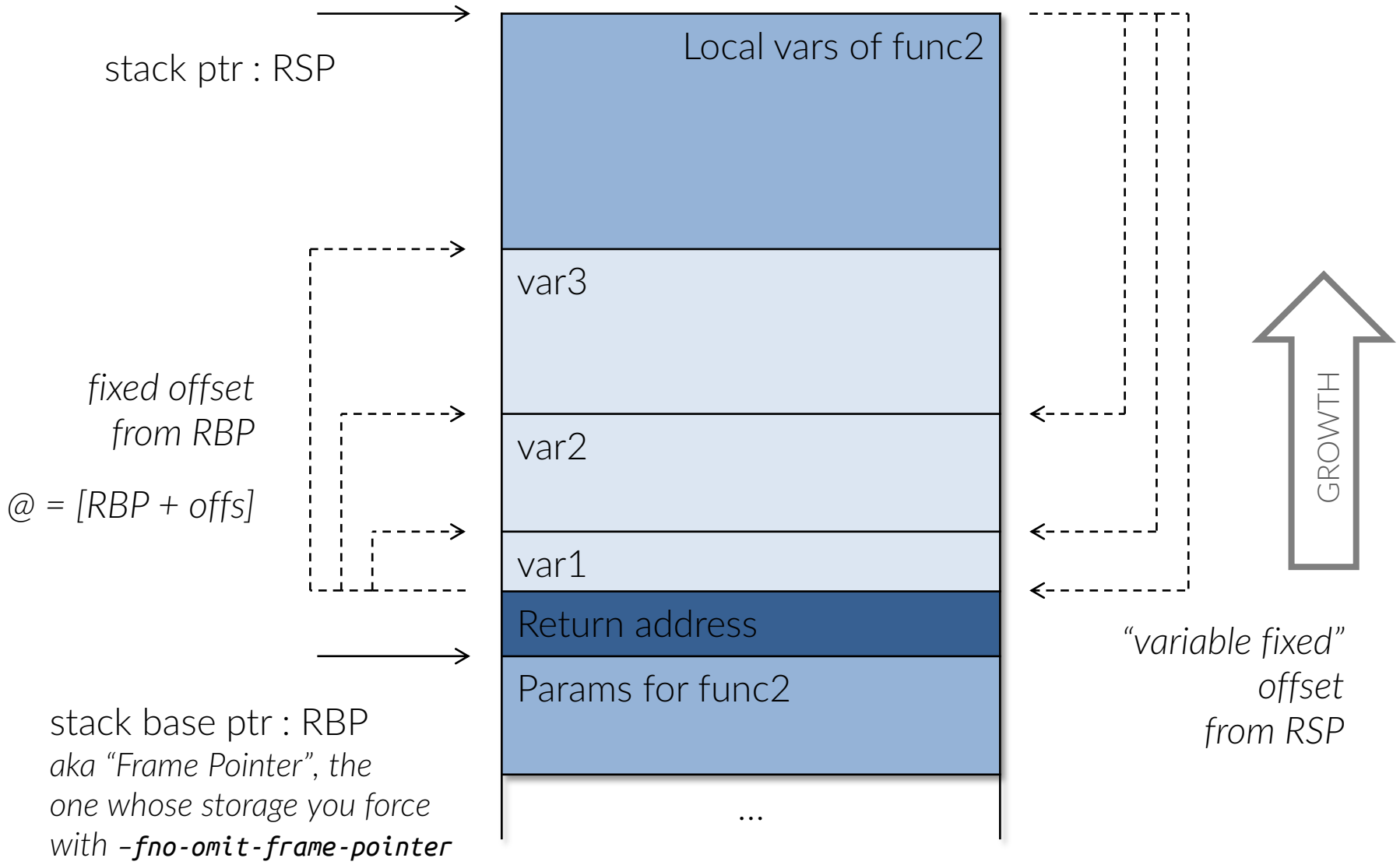
A simplistic but still effective representation of the execution model in most O.S.



Zoom in the stack



Zoom in the stack



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.

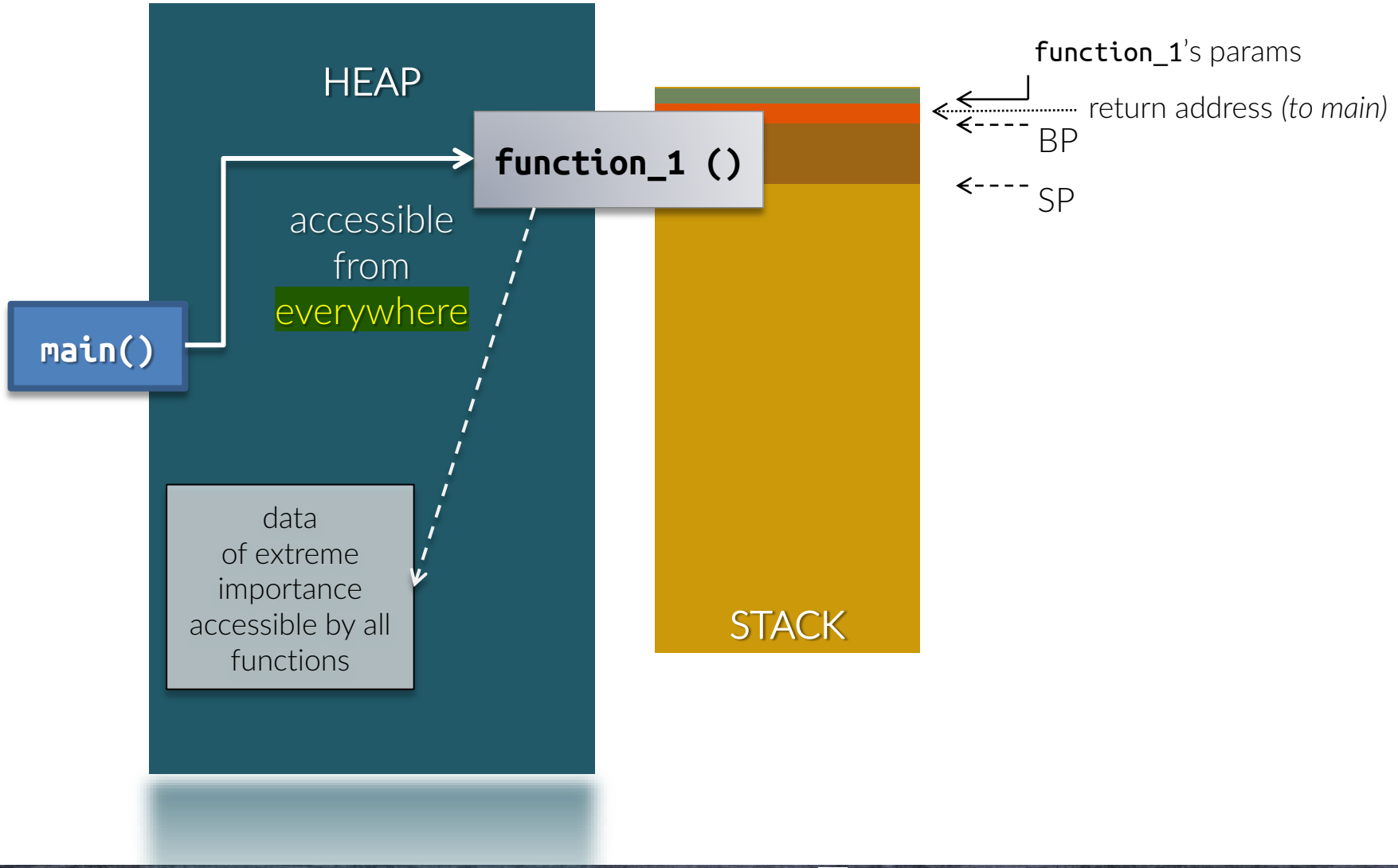
The stack is basically a bunch of memory allocated for you by O.S., to which the CPU refers to by using two dedicated registers. Its most natural use is for **static allocation.**

The **heap** is all your available memory, it 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).

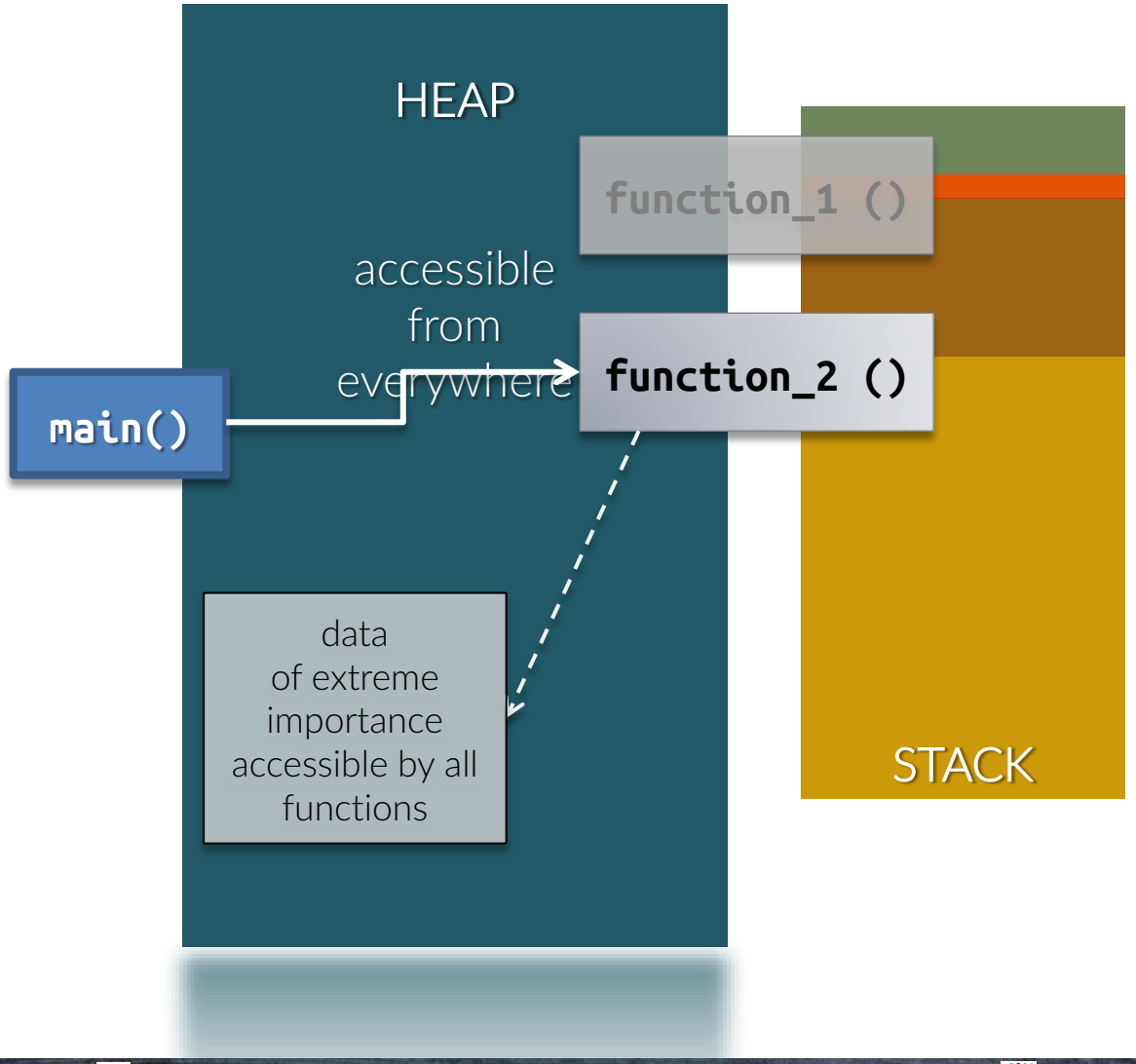
In addition to housing the global variables, its most natural use is to hold big amount of data or, in any case, the data whose amount is known only at run-time.

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

Different in scope

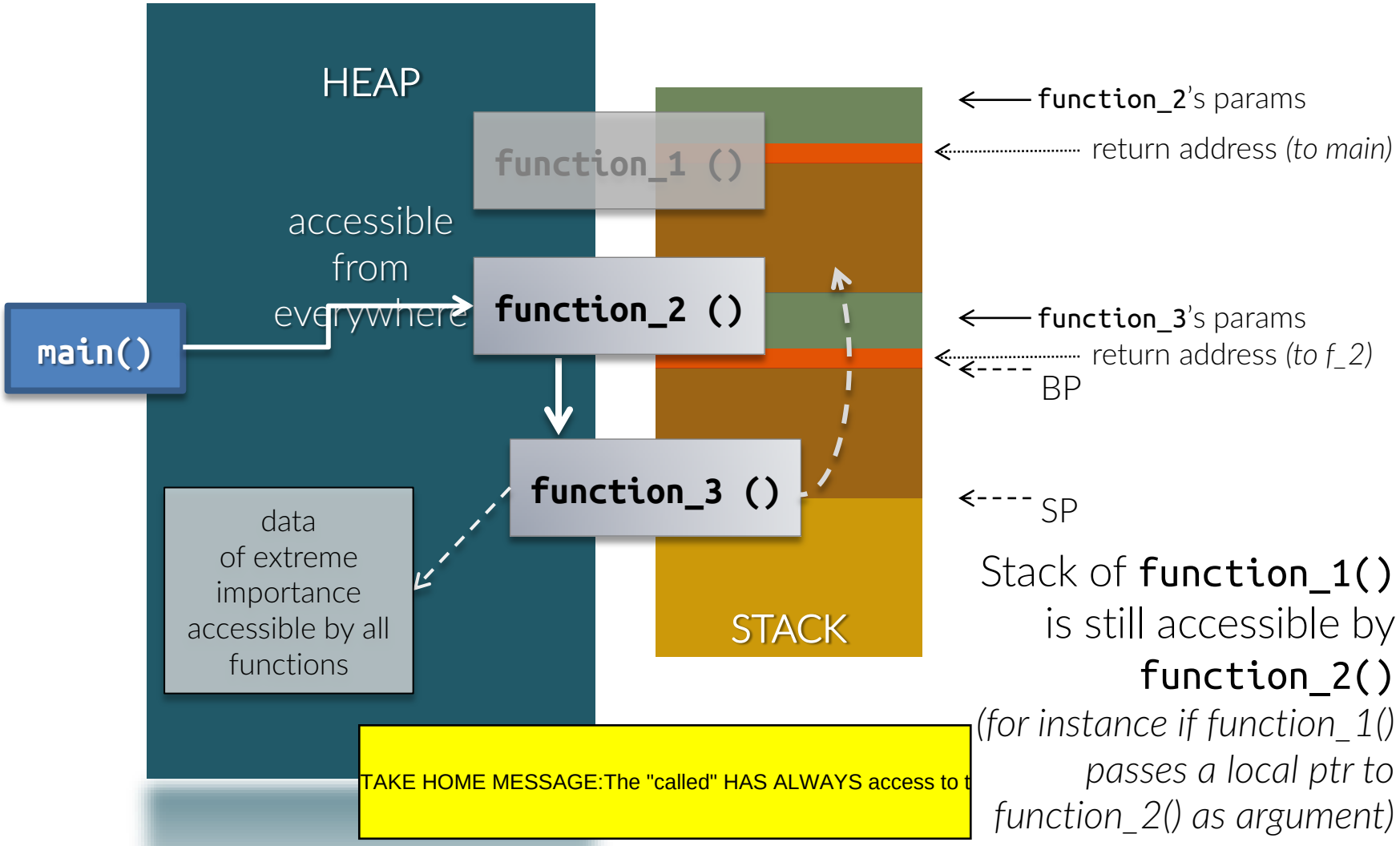


Different in scope



Now the stack of **`function_1()`** has disappeared, and its data are not accessible by **`function_2()`** *(actually they have been even overwritten by function_2's ones)*

Different in scope



Stack of **function_1()** is still accessible by **function_2()**

(for instance if function_1() passes a local ptr to function_2() as argument)

Different in scope

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.

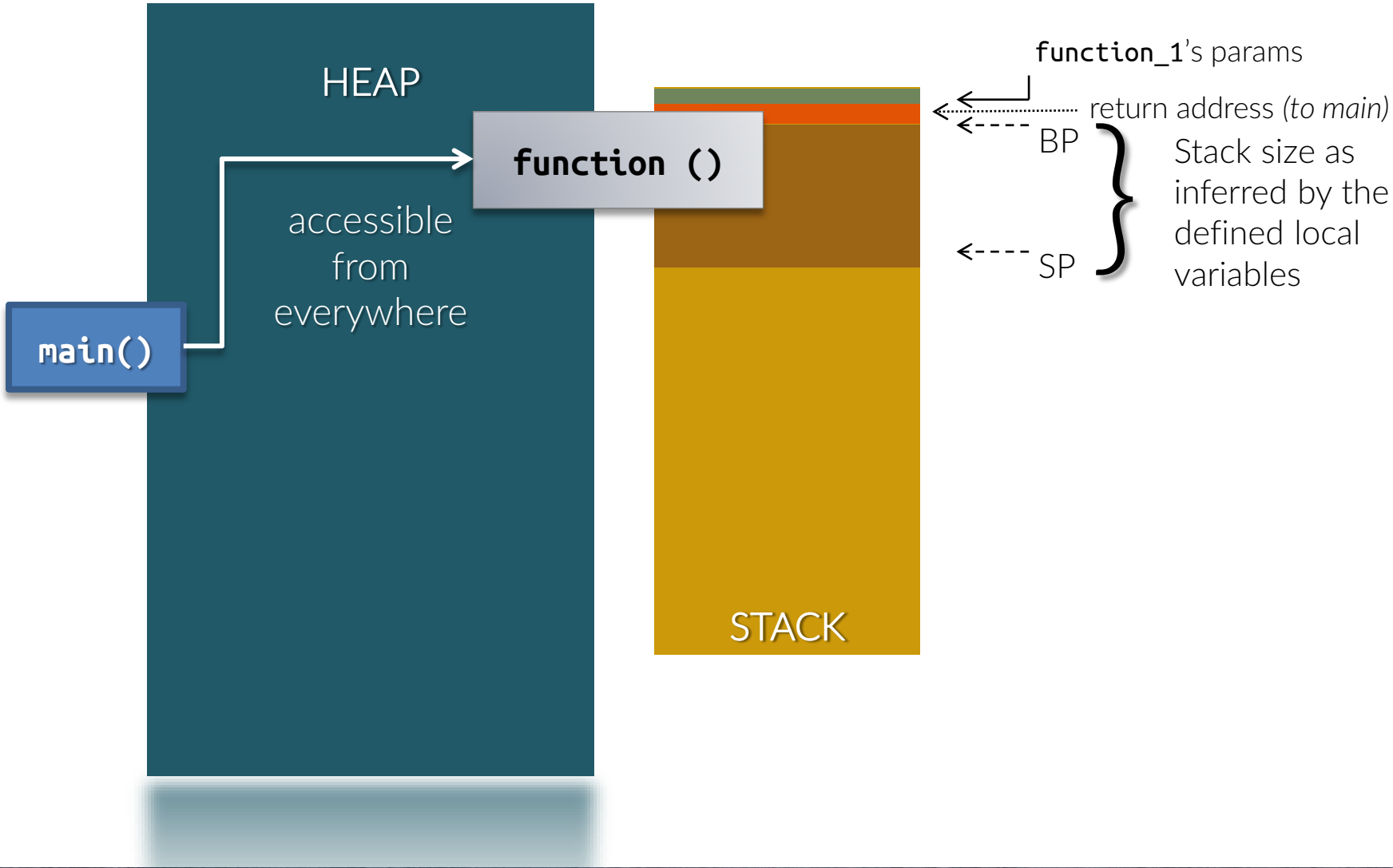
Note: its size is meant to allow a deep call stack (for instance, for **recursion**)

Dynamic allocation on the stack

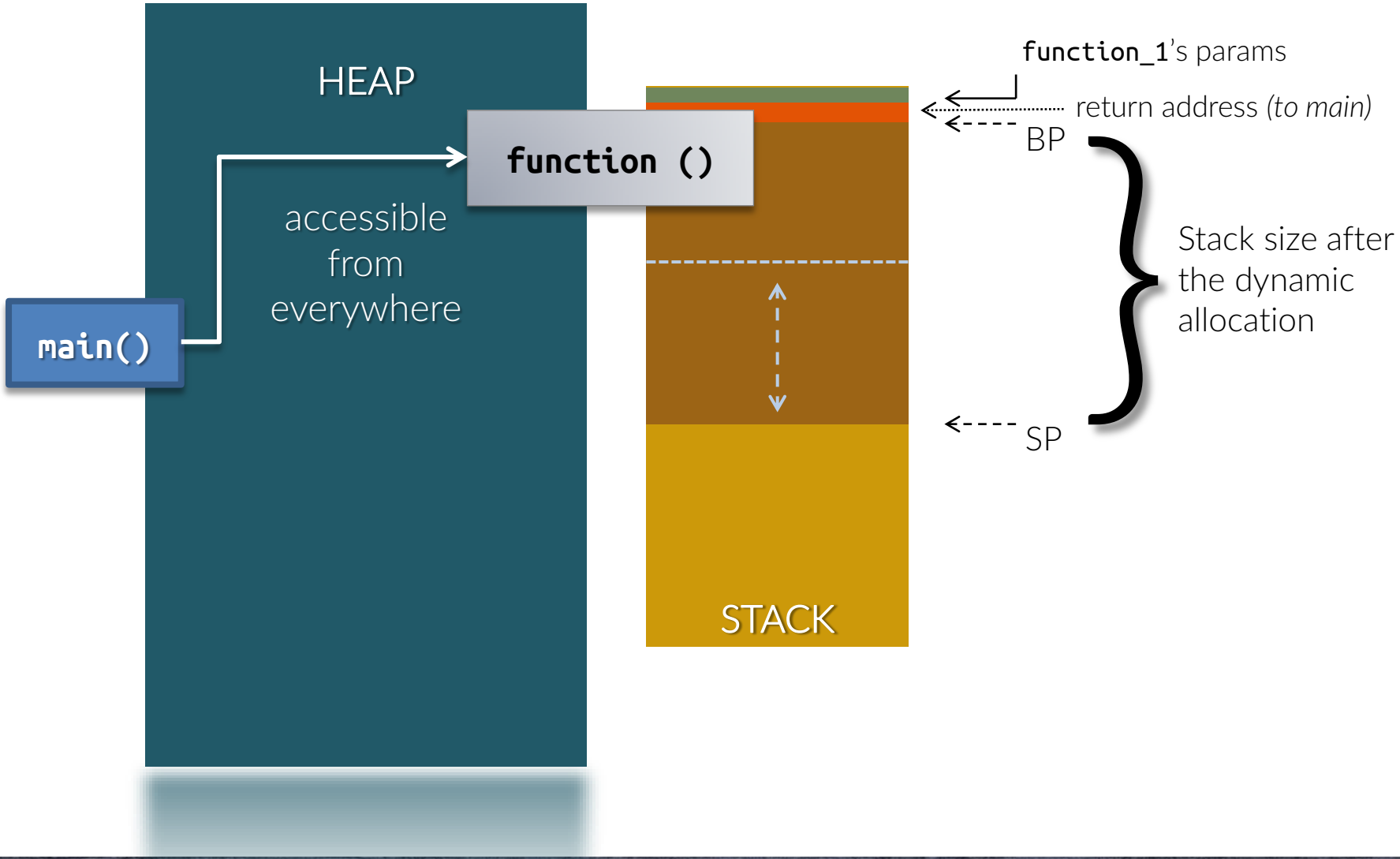
Provided what we have just said, it may be useful to dynamically allocate on the stack some array that has a local scope, being useful *hinc et nunc* but whose size is not predictable at compile time:

```
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;
   }
```

Dynamic allocation on the stack



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”?)

Not smashing the stack..

..can be done.

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
```

STACKSMASH at

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

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

Not smashing 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;
getrlimit ( RLIMIT_STACK, &newstack_limits );
```

Not smashing 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
```

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.

Compare stack and heap access

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

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;
        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;
        return 0; }
```

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 resident in a register.

examples

```
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

-O0

```
          -024[rbp] is an int counter
MOV       eax, DWORD PTR -024[rbp]        #mov QW from mem to reg
CDQE
          -1024[rbp] is the base of an array
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 that:

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

-O0

```
MOV     eax, DWORD PTR -148[rbp]  #load the offset i in reg eax
CDQE
LEA     rdx, 0[0+rax*4]           #put the offset in byte in rdx
MOV     rax, QWORD PTR -36[rbp]   #load array address in rax
ADD     rax, rdx                  #offset array to array+i
MOVSS   DWORD PTR [rax], xmm0     #mv xmm0 to array[i]
                                  #note: let's say xmm0 here
                                   contains 0
```

Is the stack faster than the heap?

However, if you access the dynamically allocated memory in a different way, you may have a different workflow:

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

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

-O0

```
MOV     rbx, QWORD PTR -96[rbp]          #load address pointed by array
                                          (-96[RBP]) into ptr (rbx)
..
MOVSS   DWORD PTR [rbx], xmm0            #mv xmm0 (0, for instance) to
                                          *ptr (i.e. [rbx])
ADD     rbx, 4                           #if type size of array is 4
```

Is the stack faster than the heap?

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
```

-O0

Accessing an array on the heap:

```
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
```

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,-ahldn 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.

How to generate assembler

Generate the most basic assembler listing

```
%> 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 assemmbler
%> 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)
```

Outline

The execution and running model

Stack & Heap

Worked examples

Addendum for your fun and profit

.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r00t, and Underground.Org
bring you

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

by Aleph One
aleph1@underground.org

`smash the stack` [C programming] n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind. Variants include trash the stack, scribble the stack, mangle the stack; the term mung the stack is not used, as this is never done intentionally. See spam; see also alias bug, fandango on core, memory leak, precedence lossage, overrun screw.

Introduction

Over the last few months there has been a large increase of buffer overflow vulnerabilities being both discovered and exploited. Examples of these are syslog, splitvt, sendmail 8.7.5, Linux/FreeBSD mount, Xt library, at, etc. This paper attempts to explain what buffer overflows are, and how their exploits work.

run again with a larger memory space. New memory is added between the data and stack segments.

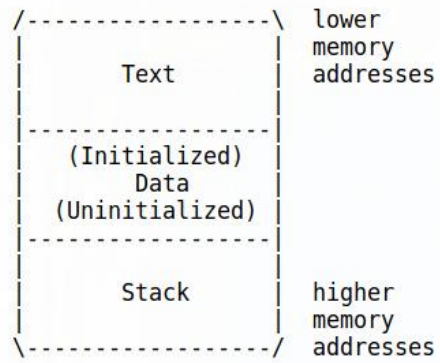


Fig. 1 Process Memory Regions

What Is A Stack?

A stack is an abstract data type frequently used in computer science. A stack of objects has the property that the last object placed on the stack will be the first object removed. This property is commonly referred to as last in, first out queue, or a LIFO.

Several operations are defined on stacks. Two of the most important are PUSH and POP. PUSH adds an element at the top of the stack. POP, in contrast, reduces the stack size by one by removing the last element at the top of the stack.

Why Do We Use A Stack?

Modern computers are designed with the need of high-level languages in mind. The most important technique for structuring programs introduced by