PROGRAMMING LANGUAGES: RUN-TIME SUPPORT

SO FAR AND FROM PREVIOUS COURSES....

.Programming languages Compilers

- Regular expressions and lexical analysis
- Grammars and Parsing

At runtime, a series of coordinated steps occur to transform your program's code into active execution on a computer.

RUN-TIME

→ Load the program in a segment provided by the OS.

- Program Loading: This process involves mapping code segments, data segments, and other resources into the process's address space.
- Dynamic Linking: The dynamic linker/loader resolves these dependencies, loading the necessary shared libraries and linking them to the program at runtime.
- Initialization: The runtime environment performs various initialization tasks
- Execution of the entry program point:

  The entry point (often the main function) is called. The CPU begins executing instructions from this entry point, following the program's control flow which may involve function calls, loops, conditionals, and so on.

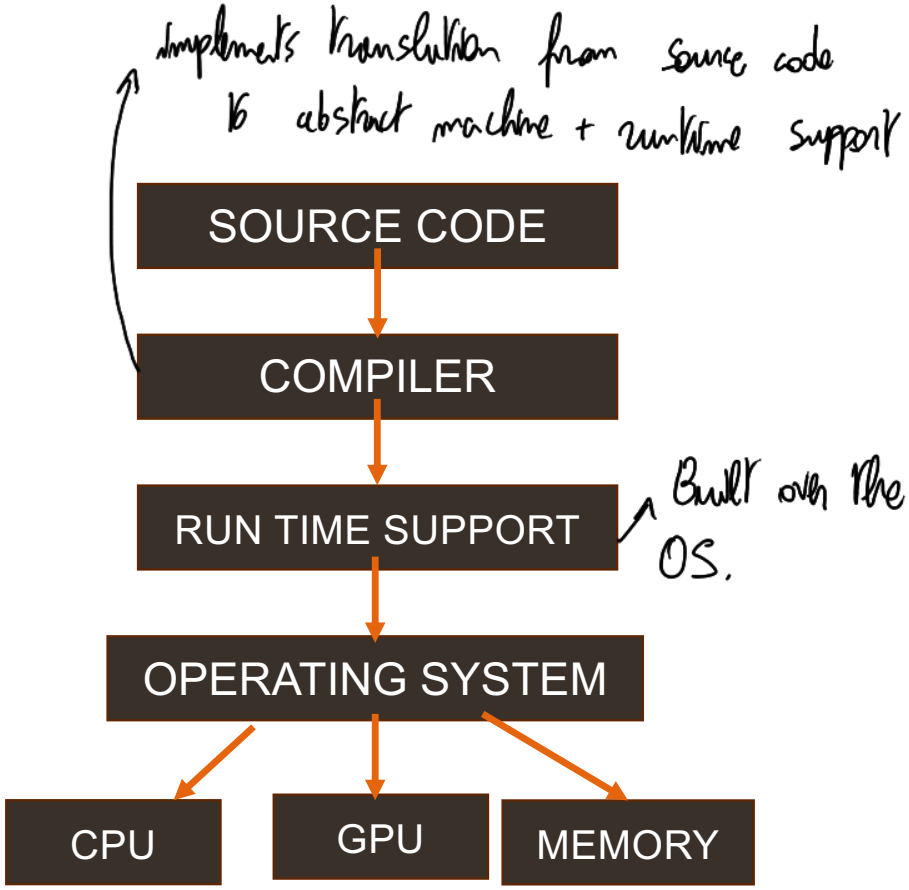
(Java: class with the main method)

RUN-TIME SUPPORT

The instructions generated by
The compiler that adjusts the
Run time structures

- **The stacck**
- **The heap managed by malloc (and garbage collection)**
- **:**

OS provides segment of a memory

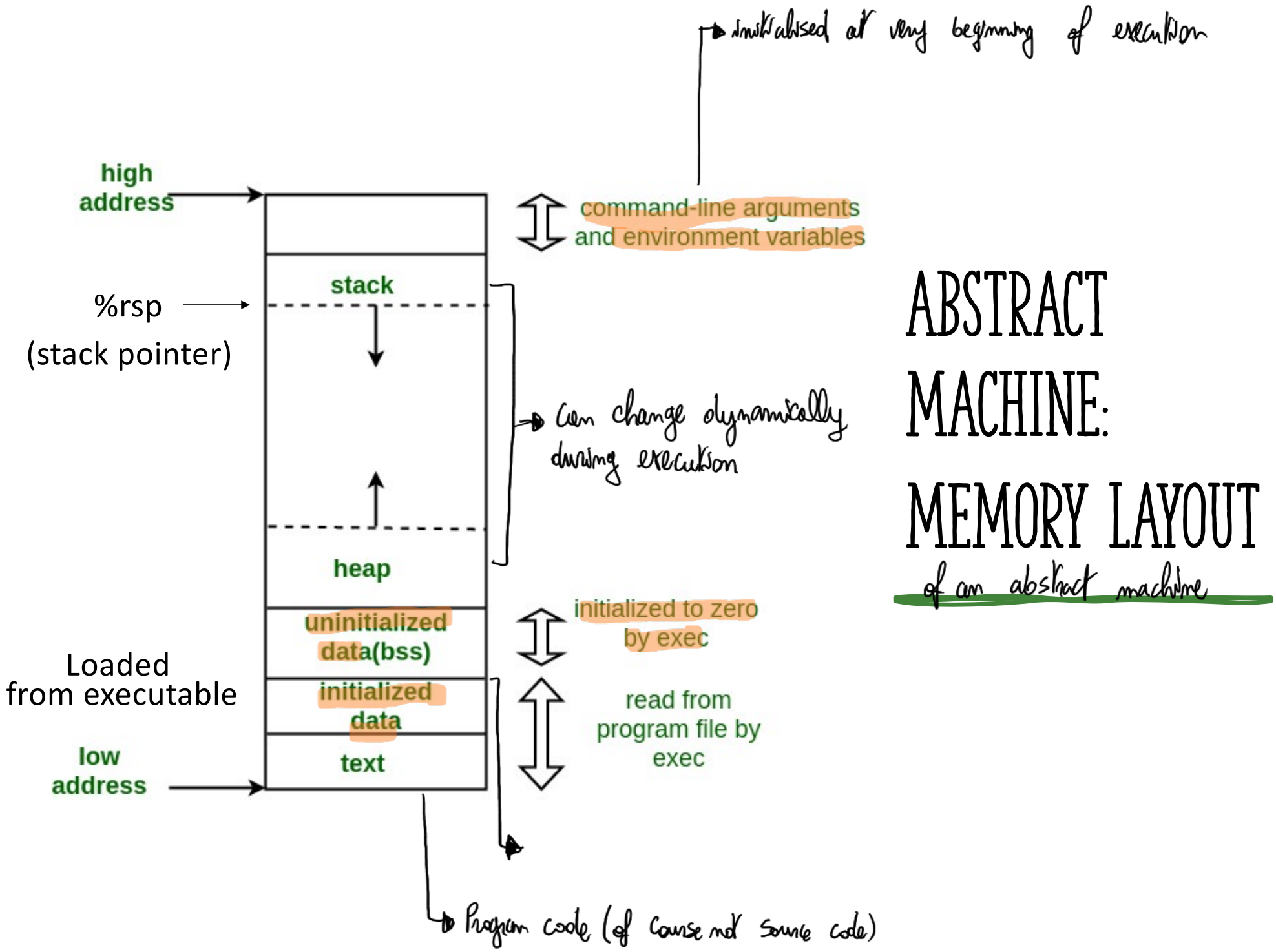


So run time support is all the code added to manage all that stuff?

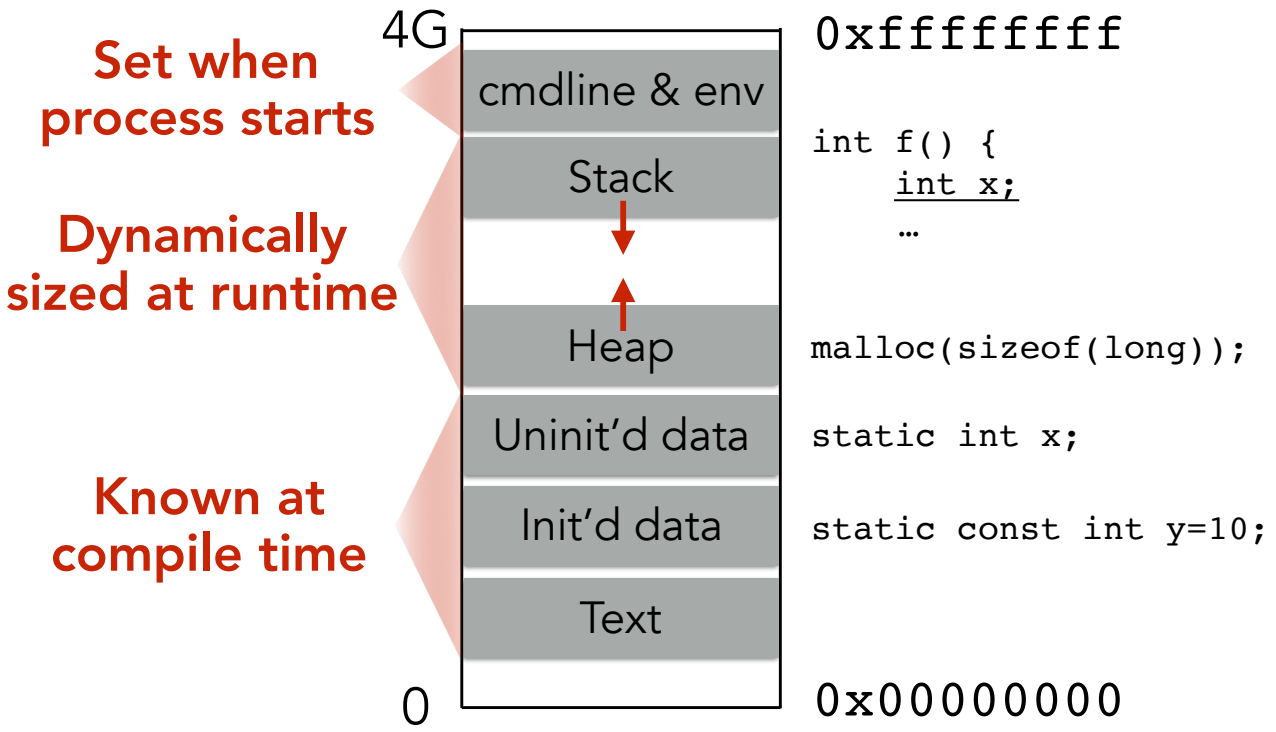
Exactly! **Runtime support** is all the additional code—whether in the form of libraries, built-in functions, or a full runtime environment—that helps manage things like memory, function calls, and error handling while the program is running.

ABSTRACT MACHINE: MEMORY LAYOUT

of an abstract machine



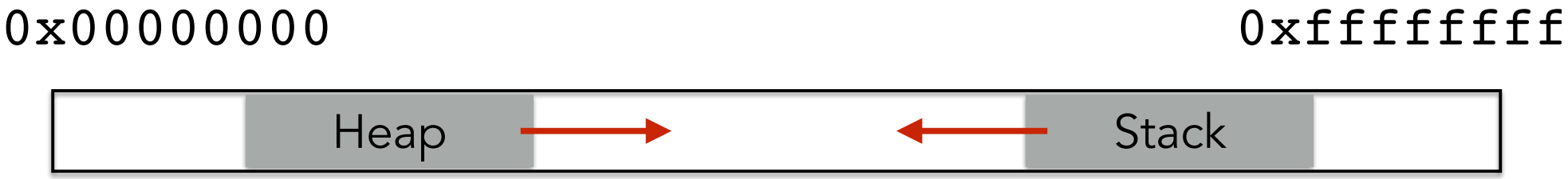
DATA'S LOCATION DEPENDS ON HOW IT'S CREATED



REMARK

Stack and heap grow in opposite directions

Compiler provides instructions that
adjusts the size of the stack at runtime

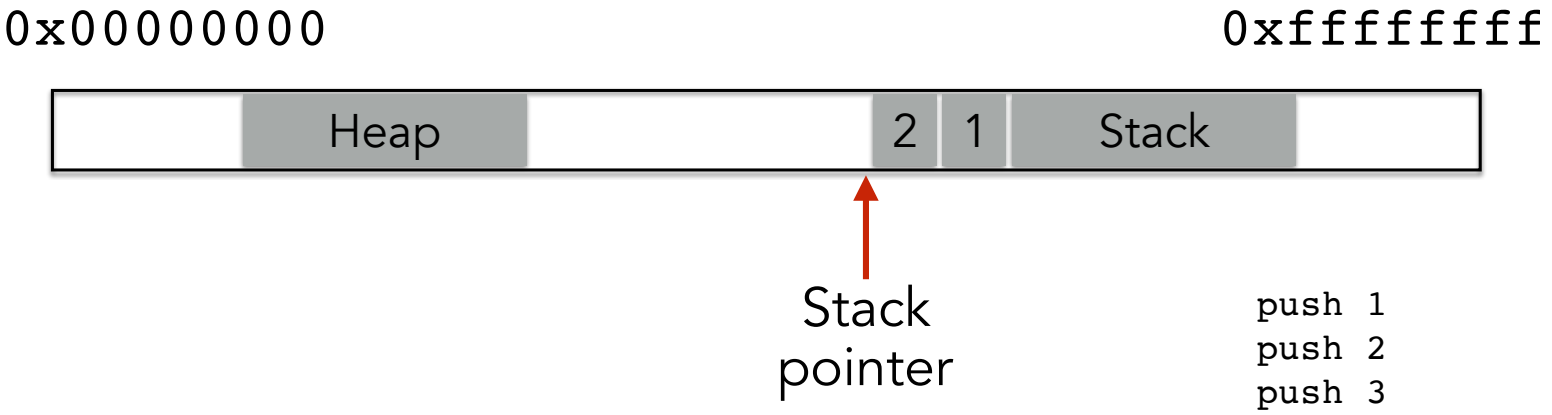


At runtime, we need a pointer, stack pointer, that can be used to perform push, pop, top.

REMARK

Stack and heap grow in opposite directions

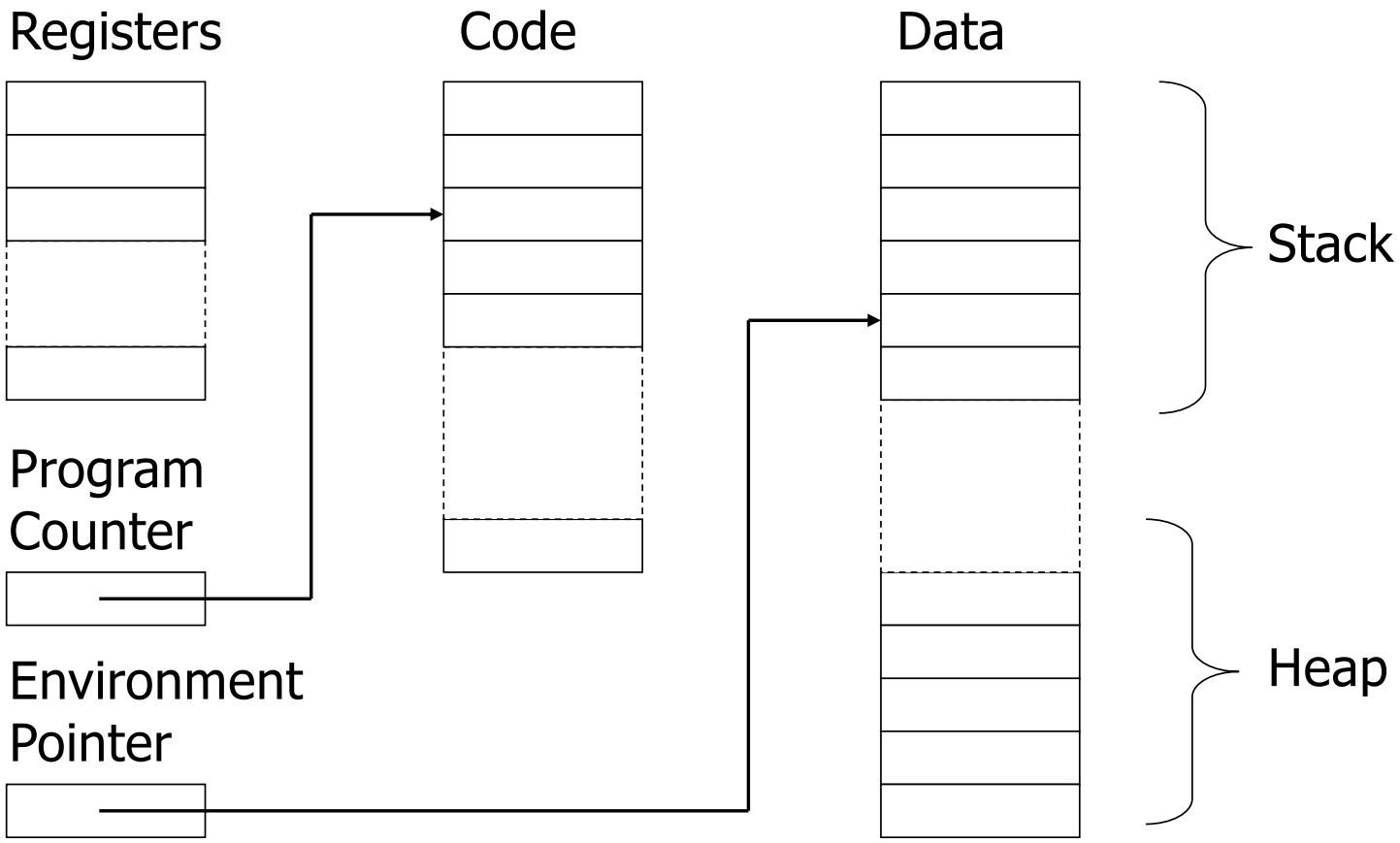
Compiler provides instructions that
adjusts the size of the stack at runtime



In the abstract machine we need two special registers: stack pointer / Environmental pointer and program counter.

The OS provides to the abstract machine this representation

PROGRAMMING LANGUAGES: RUN-TIME STRUCTURE: A SIMPLIFIED VIEW



The flow of control

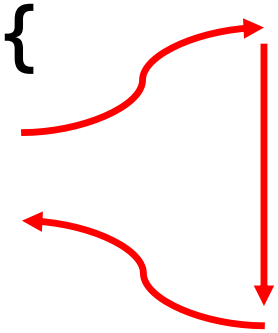
- when the caller calls a function, where do we go?
- when the callee's code is finished, where do we go?

```
void fork() {
    knife();
    oops;
}
```

caller

```
void knife() {
    oops;
}
```

callee



When we call knife(), fork() stops execution. When knife finishes, we return to the next instruction of the caller.

To implement this mechanism (call and return of function) we need both the prog. ctr and typically the stack.

Activation record: represents the space in the stack allocated for a called function.

↳ When a function is called, this is pushed in the stack. When callee terminate

CALLING A FUNCTION: RUN TIME STRUCTURES

- When we call a function f, we push a new *activation record* (also called a *frame*) on the run-time stack, which is particular to the function f.
  - Each frame represents the local environment of the functiopon and can occupy many consecutive bytes in the stack and may not be of a fixed size.
- When the callee function returns to the caller, the activation record of the callee is popped out.
- Example, if the main program calls function P, P calls E, and E calls P, then at the time of the second call to P, there will be 4 activation records in the stack in the following order: main, P, E, P

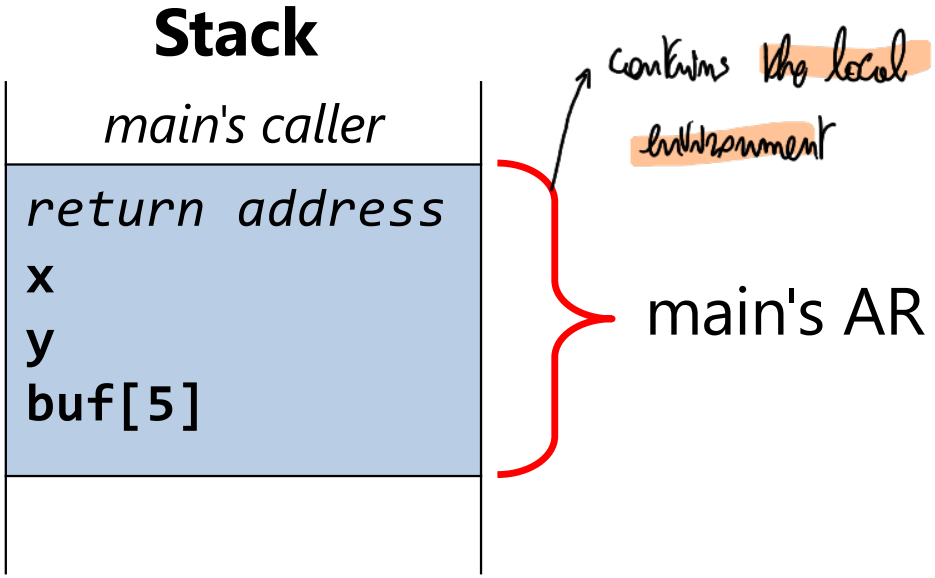
What's the stack

- it's an **area of memory** provided to your program by the OS.
  - when your program starts, it's already there.
- the stack holds **information about function calls.**
- it's not a *strict* stack…
  - you can read and write any part of it.
  - but it **grows and shrinks** like a stack…
  - and we use "push" and "pop" to describe that.
- each program (actually each thread) gets **one** stack!

Activation records (ARs)

- when a function is called, a bunch of data (frame) is **pushed** onto the stack
- it contains **local variables** (including arguments) and the **return address**

```
int main() {
    int x, y;
    char buf[5];
    return 0;
}
```



In Java, arrays are placed in the heap. Arrays are objects. Unlike C

NOTE: Size of AR can be evaluated statically.

The low-level layout

- each variable (*including* array variables) gets enough bytes in the AR to hold its value
- *where* the variable is located is up to the compiler

```
int x ;
char arr[3];
short y;
```

`sizeof(int)` == 4, so x gets 4 bytes
`sizeof(arr)` == 3, so it gets 3 bytes
`sizezof(short)` == 2, so..

* 4 bytes for return address
But lot of grey. Why? Compiler does its job.

Addr	Value
d02f	
d02e	
d02d	
d02c	return
d02b	
d02a	
d029	
d028	
d027	
d026	y
d025	
d024	
d023	
d022	
d021	
d020	x
d01f	
d01e	
d01d	arr
d01c	
d01b	
d01a	
d019	
d018	

The low-level layout

- each variable (*including* array variables) gets enough bytes in the AR to hold its value
- *where* the variable is located is up to the compiler

```
int x ;
char arr[3];
short y;
```

`sizeof(int)` == 4, so x gets 4 bytes
`sizeof(arr)` == 3, so it gets 3 bytes
`sizezof(short)` == 2, so..

but what's all that grey space?

The compiler does what it wants

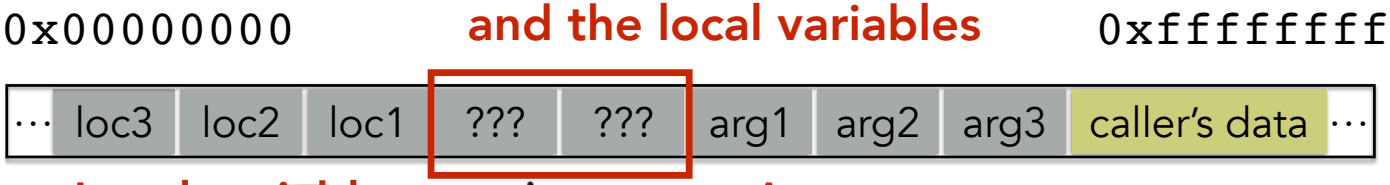
optimization

Addr	Value
d02f	
d02e	
d02d	
d02c	*return*
d02b	
d02a	
d029	
d028	
d027	
d026	y
d025	
d024	
d023	
d022	
d021	
d020	x
d01f	
d01e	
d01d	arr
d01c	
d01b	
d01a	
d019	
d018	

STACK LAYOUT EXAMPLE

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```

Two values between the arguments and the local variables



Local variables pushed in the same order as they appear in the code

Arguments pushed in reverse order of code

Logic of C compilers

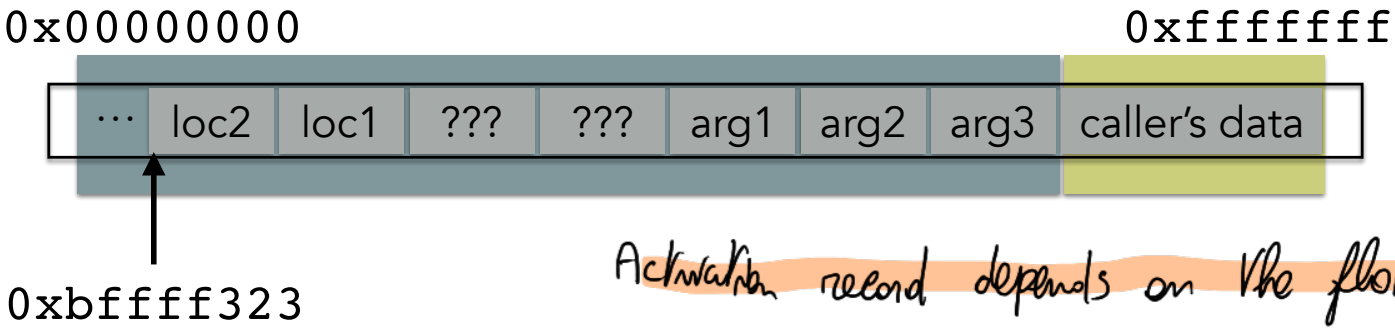
opposite order

Required for bookkeeping. So you don't know how much time you need.

ACCESSING VARIABLES

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    loc2++;
}
```

Q: Where is (this) loc2?



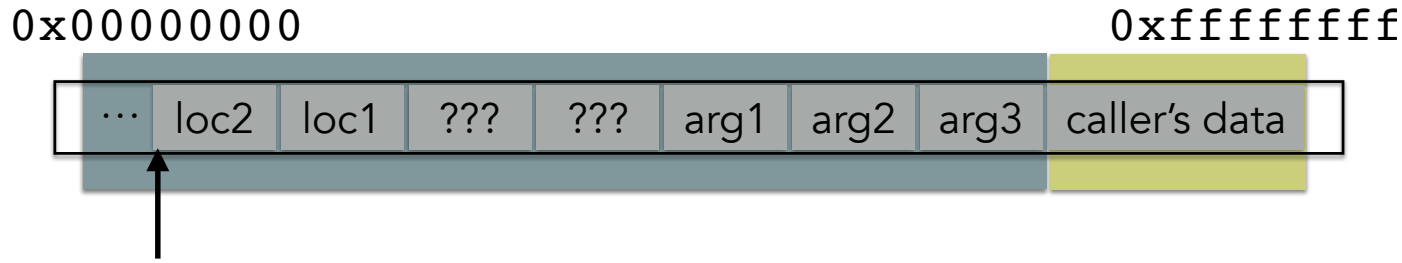
Activation record depends on the flow of execution.

Position in the stack depends from what and from how many AR I have when I call func.

ACCESSING VARIABLES

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    loc2++;
}
```

Q: Where is (this) loc2?

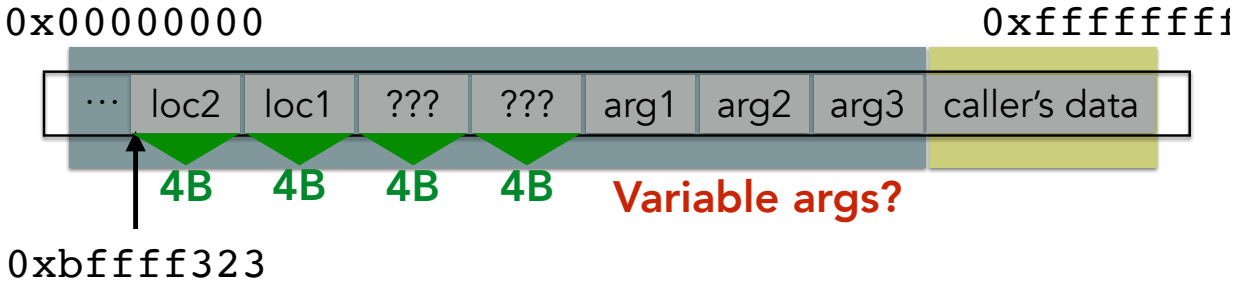


Undecidable at compile time

ACCESSING VARIABLES

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    loc2++;
}
```

Q: Where is (this) loc2?



Undecidable at compile time

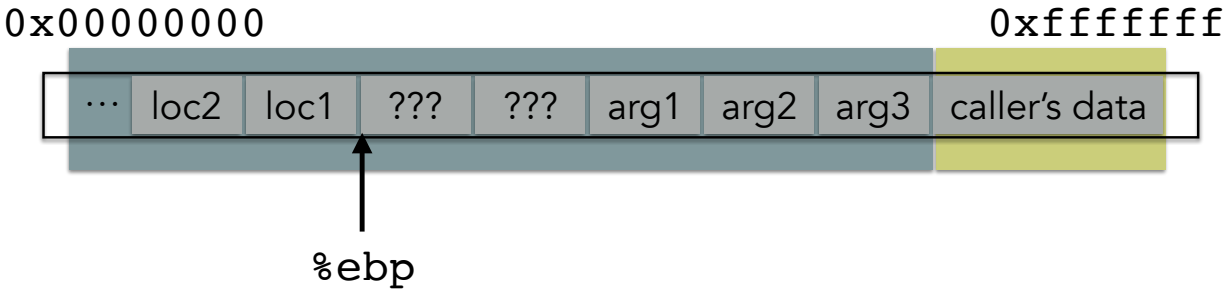
- I don't know where loc2 is,
- and I don't know how many args
- *but* loc2 is *always* 8B before "???"s

This is what I know because the compiler is the one that decides how AR are constructed.

ACCESSING VARIABLES

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    loc2++;
}
```

Q: Where is (this) loc2?
A: -8(%ebp)



Frame pointer

- I don't know where loc2 is,
- and I don't know how many args
- *but* loc2 is *always* 8B before "???"s

↳ This points inside the AR. I can always address loc2 because I know from my AR I know how many bytes I have to move from the FP. We subtract 8 bytes.

Frame pointer

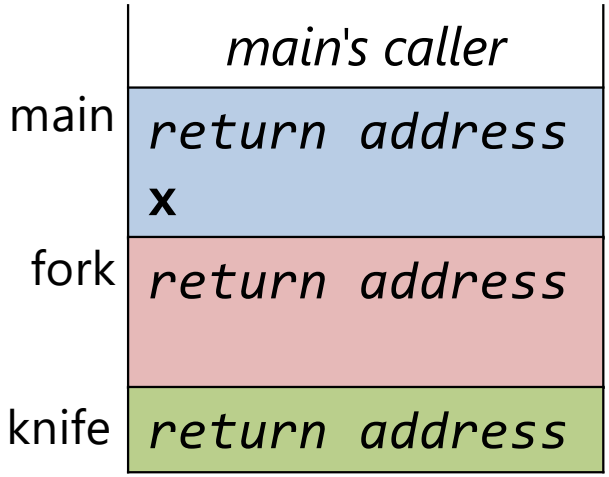
- **The frame pointer**
  - dedicated register that marks the stack frame, providing a stable reference point for accessing function parameters, local variables, and saved registers during runtime.
- **Stable Reference for Variable Access**
  - Variables can be reliably accessed via fixed offsets
  - Negative offsets of the frame pointer (move towards lower memory)
  - Positive offsets of of the frame pointer will allow us to read parameters.
- **Simplifies Code Generation and Debugging**:
  - By using the frame pointer, compilers generate code with predictable memory access patterns. This consistency is particularly useful for debugging, as it allows debuggers to traverse the call stack, reconstruct function frames, and locate variables accurately. Without a frame pointer, variable addresses would be harder to predict, complicating both debugging and exception handling.

Call = push, return = pop

```
int main() {
  int x;
  fork(&x);
  return 0;
}
```

```
void fork(int* ptr) {
  *ptr = knife();
}

int knife() {
  return 10;
}
```

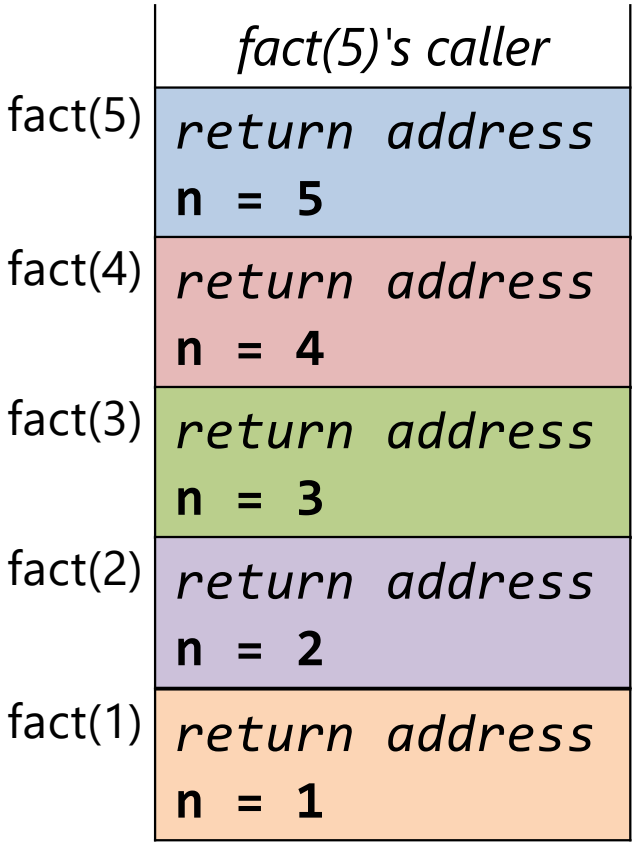


the stack **grows** when we call a function and **shrinks** when it exits

Recursive functions (animated)

- recursive functions *work* by *using the call stack as an implicit stack data structure*

```
int fact(int n) {
  if(n <= 1) {
     return 1;
  } else {
     return n*fact(n-1);
  }
}
```



But they don't *really* go away (animated)

- the stack is used *constantly*
  - it needs to be *really fast*
- implemented as a **pointer**
  - the **stack pointer** (sp)

We do not delete stuff from memory

push AR: `sp -= (size of AR)`

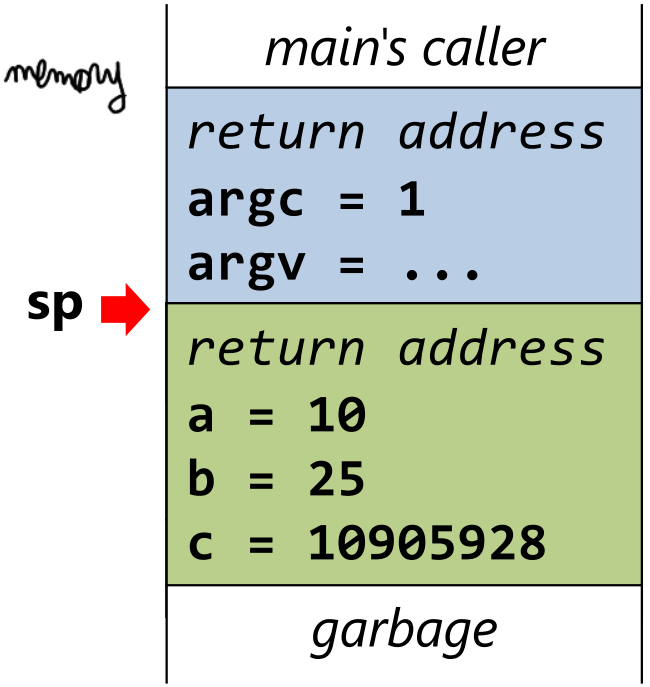
pop AR: `sp += (size of AR)`

but the AR's memory is still there.

so if we call another function...

this is where that garbage in uninitialized variables comes from!

No default value but old value



Quiz

```
int main() {
  char* str = func();
  printf("%s\n", str);
  return 0;
}
char* func() {
  char str[10] =
    "hi there";
  char* dummy = str;
  return dummy;
}
```

Quiz

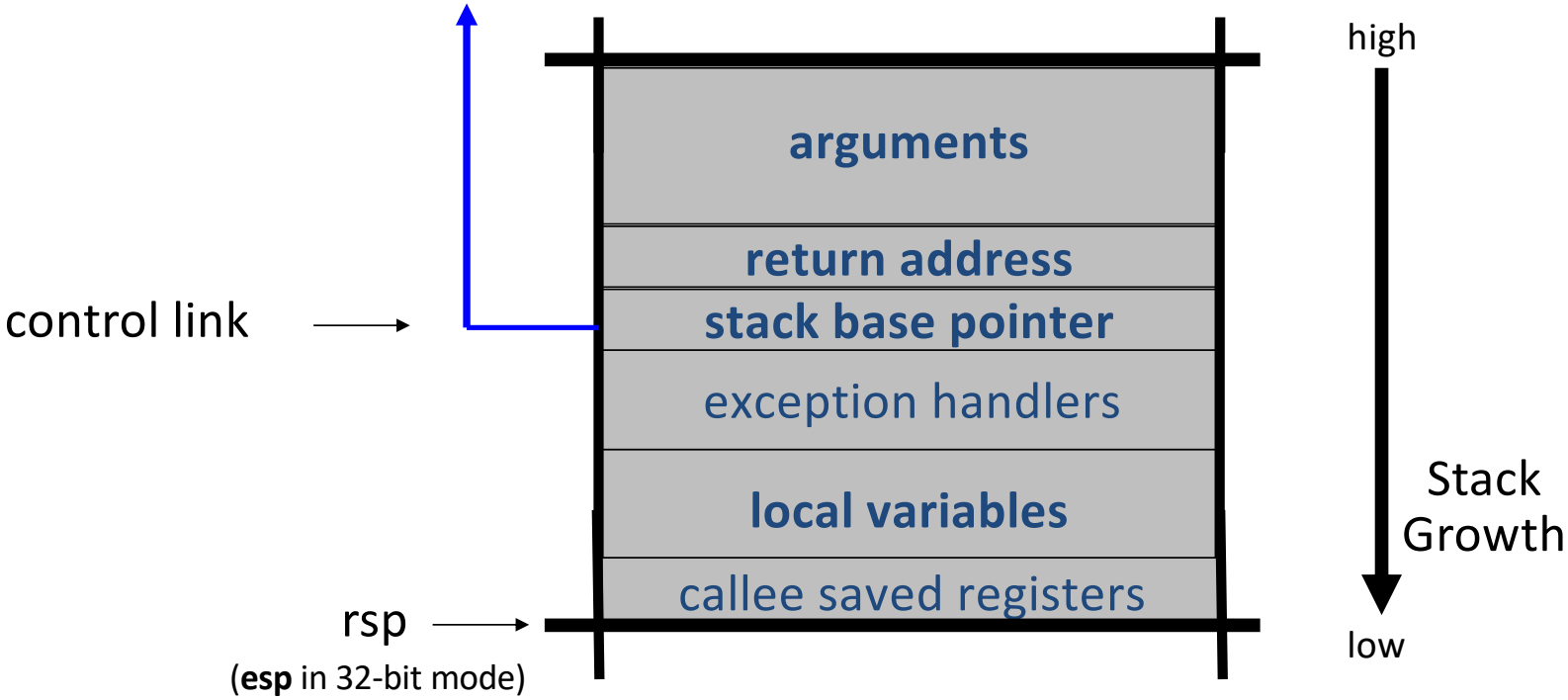
"function returns address of local variable"

- when func() returns, what happened to its variables?
  - ***you have no idea***
- you now have an **invalid pointer**
  - it points to who-knows-what
    - it might crash
      - it might expose secrets
        - » who knows!!

```
int main() {
  char* str = func();
  printf("%s\n", str);
  return 0;
}
char* func() {
  char str[10] =
    "hi there";
  char* dummy = str;
  return dummy;
}
```

SW Vulnerability

C Activation Record



SAVE ALL THE REGISTERS USED BY THE ABSTRACT MACHINE

THE CALLING CONVENTION

To call f with arguments a1, ..., an:

1. Save **caller-save** registers
   - The caller wants to preserve their values, caller must save them
2. Parameter passing
3. The return address is store in the activation record
4. Execute code of function f and returns to return address
5. Pop arguments off stack; restore caller-save registers

Save all the registers of a caller. We save it in the activation record of caller.

NOT IN SOURCE CODE, COMPILER HAS TO PRODUCE THEM

THE CALLING CONVENTION

To call f with arguments a1, ..., an:

STEPS 1-3 are called Prologue

1. **Save caller-save registers**
   - **The caller wants to preserve their values, caller must save them**
2. **Parameter passing**
3. **The return address is store in the activation record**
4. Execute code of function f and returns to return address
5. **Pop arguments off stack; restore caller-save registers**

STEPS 4 is called Epilogue

NOTE: Do we have in compiled language the name of a variable? Compiler statically puts offset from frame pointer to reach the variable. All variables disappear. But what about static variables? Position of static variables are known: they have base pointer for the static variables. Base + offset are a good way to manage this.

CALLING CONVENTION

- Now we understand the calling convention, how do we generator code for functions?
- The compiler must generate prologue & epilogue
- Need to know how much space activation record occupies
  - Space to save callee-save registers and local variables (including parameters and result)
- In the compilation each variable is transformed into its offset from the top of the activation record
  - Access variables relative to the stack pointer.
- When we encounter a return, move the result in the activation of the caller and jump to the return address (stored in the activation record of the callee)
- The epilogue restores the callee-save registers and and the stack pointer

SUMMARY

- Call Stack: Fundamental to managing runtime function calls.
  - Stack Frames: Encapsulate function execution context.
- Calling Conventions: Define how data is transferred and cleaned up.
- Implications for Language Implementation:
  - Direct impact on performance and security
  - Influences compiler design and interoperability between languages
  - Critical for debugging, optimization, and ensuring correct program behavior
- Closing :Mastery of these concepts is key to understanding and advancing programming language implementation.

EVERYTHING IS CLEAR, BUT ....

- Our question: what are the consequences of programming language implementation techniques on software security?
- Several answers!!!