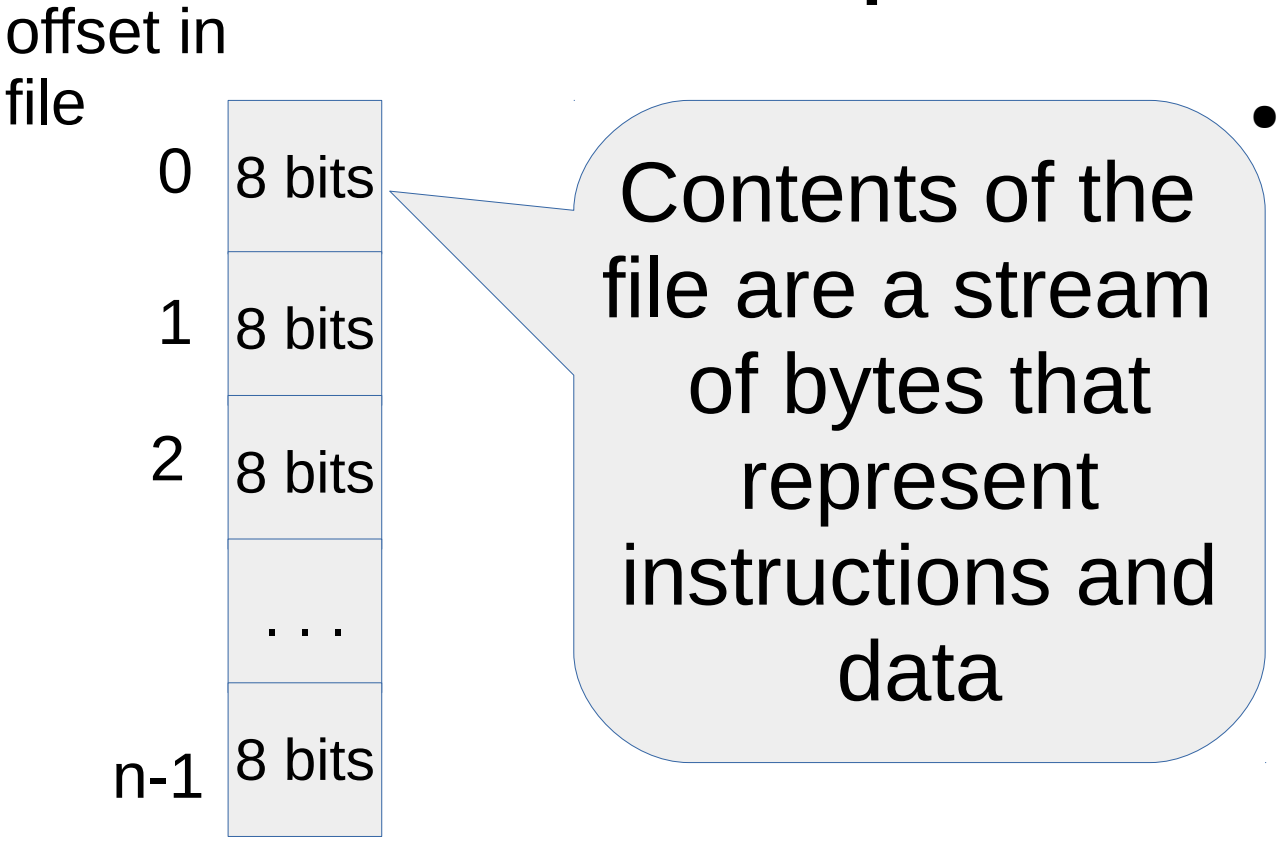
The project

C++ – the interpreter

- Two partner teams
- Send me email with your partner name. Copy your partner on it
- We will implement an interpreter or virtual machine
- It will execute programs that are a string of bytes
  - We'll have test examples available on Piazza

What the input looks like

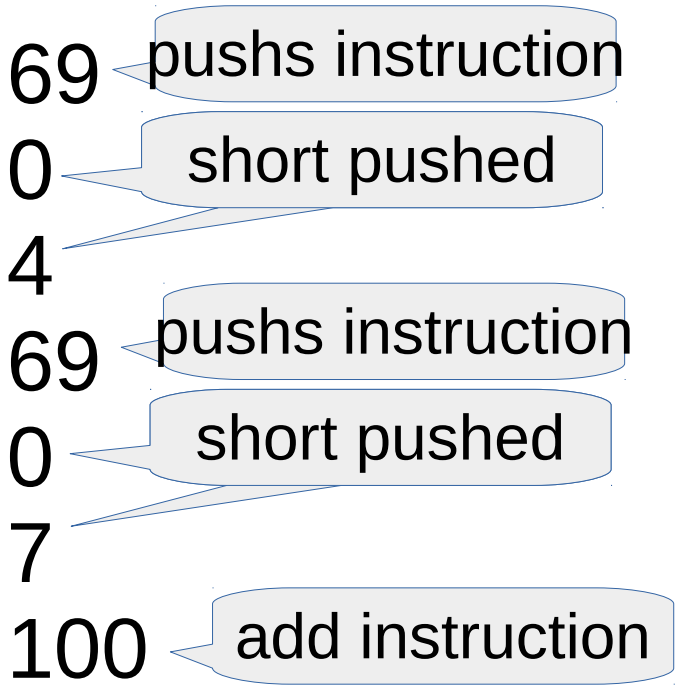


- The input should be read in as a file of bytes

Main data structures in the interpreter – the memory

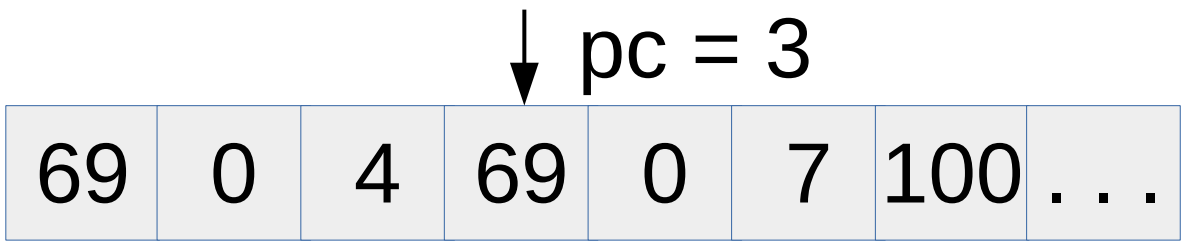
- Memory
  - In the simplest form, this is the array of bytes holding the program
  - The input file is read into memory

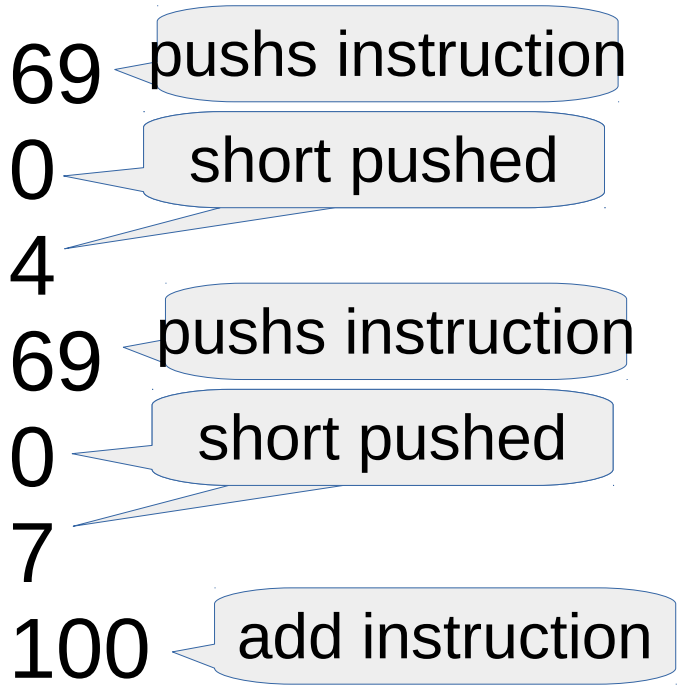
69	0	4	69	0	7	100	. . .



Main data structures in the interpreter – the program counter (pc)

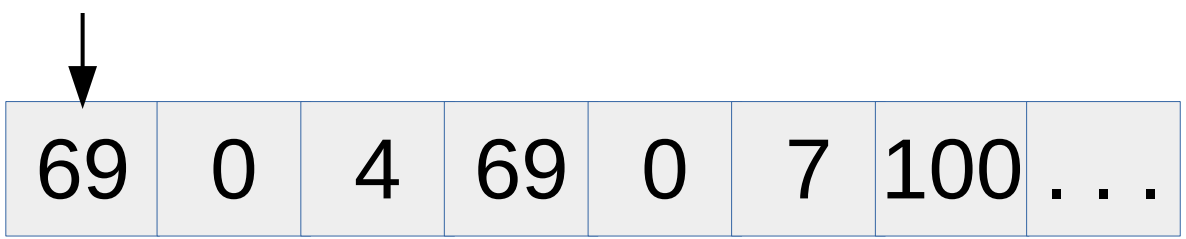
- The program counter
  - This is an index into the memory that gives the next memory location to be accessed
  - This is often the next instruction





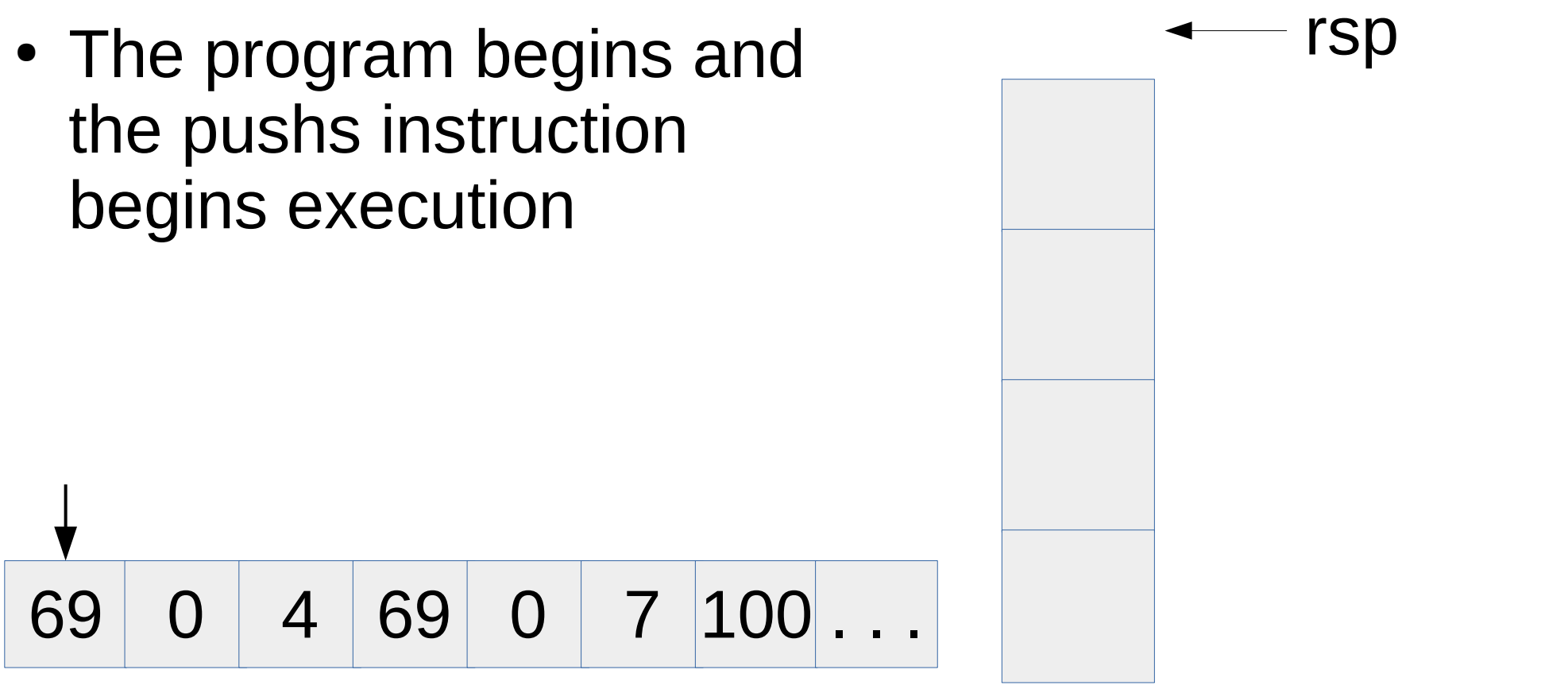
Main data structures in the interpreter – the runtime stack and runtime stack pointer (rsp)

- The runtime stack
  - Holds operands to be acted on
  - holds local variables
  - holds arguments to functions



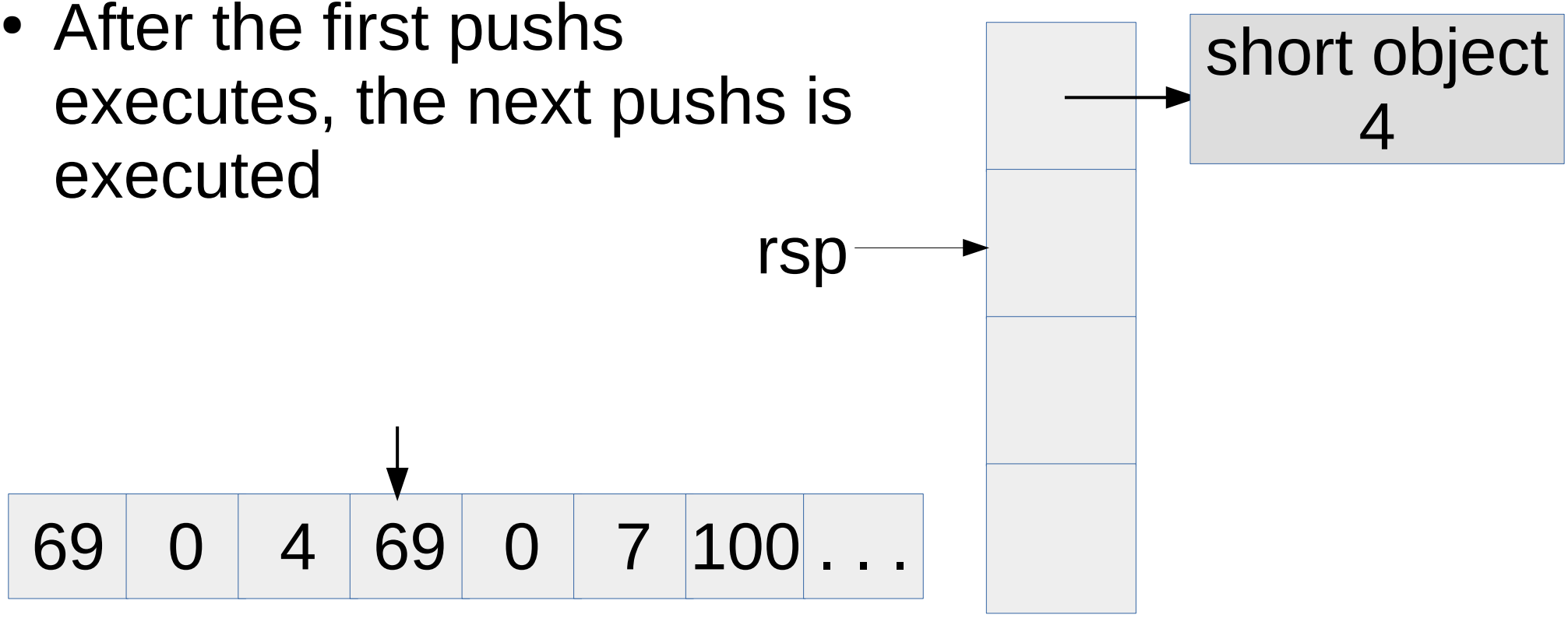
Main data structures in the interpreter – the runtime stack and runtime stack pointer (rsp)

- The program begins and the pushs instruction begins execution



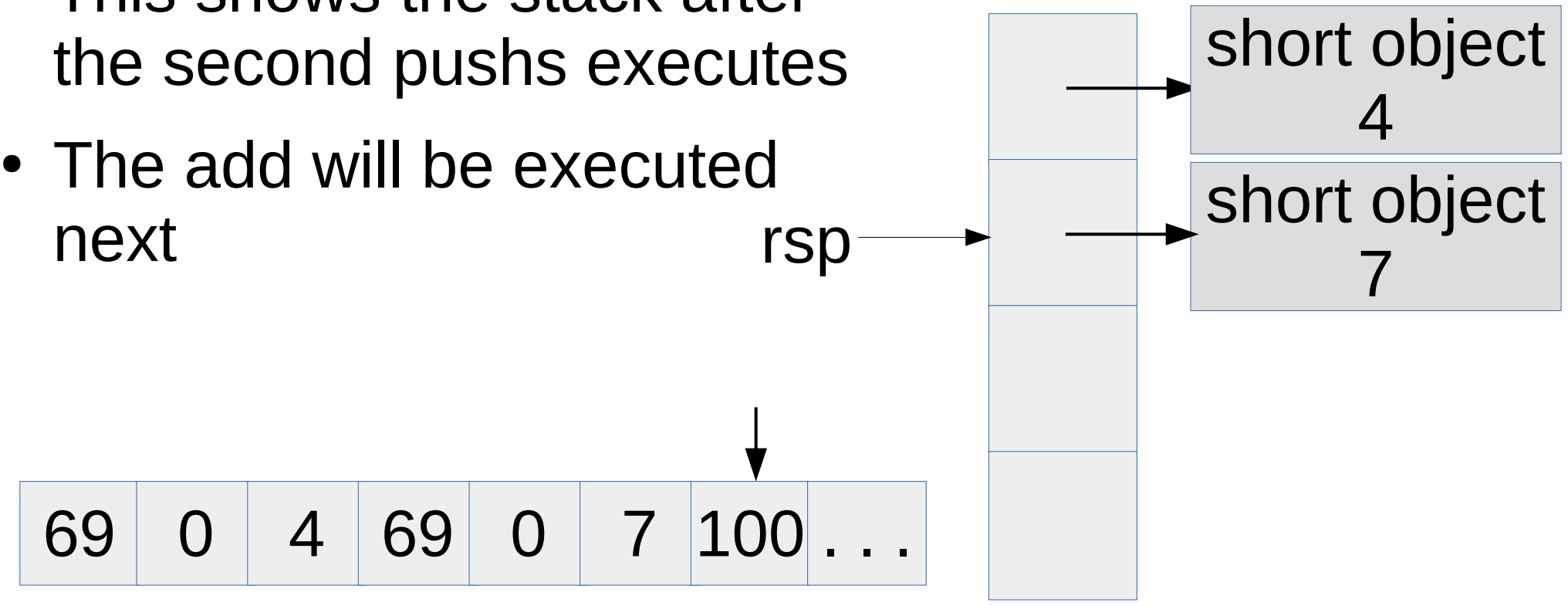
Main data structures in the interpreter – the runtime stack and runtime stack pointer (rsp)

- After the first pushs executes, the next pushs is executed



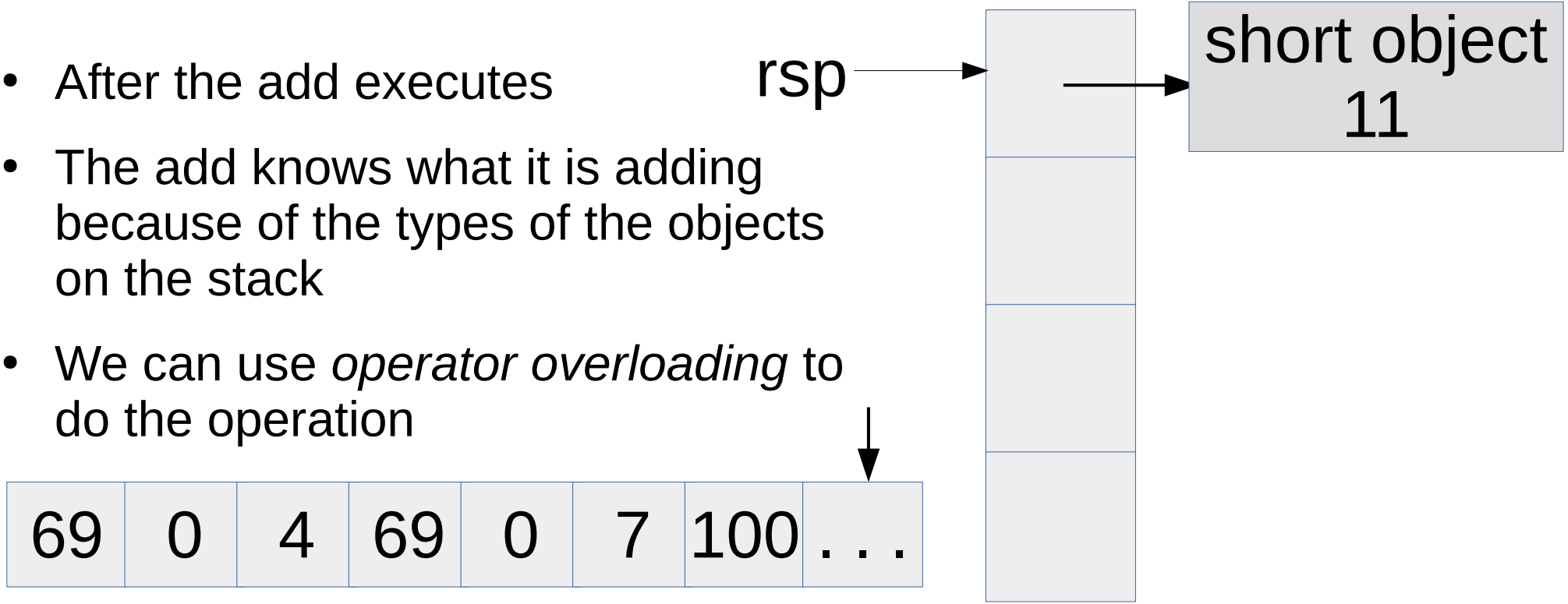
Main data structures in the interpreter – the runtime stack and runtime stack pointer (rsp)

- This shows the stack after the second pushs executes
- The add will be executed next



Main data structures in the interpreter – the runtime stack and runtime stack pointer (rsp)

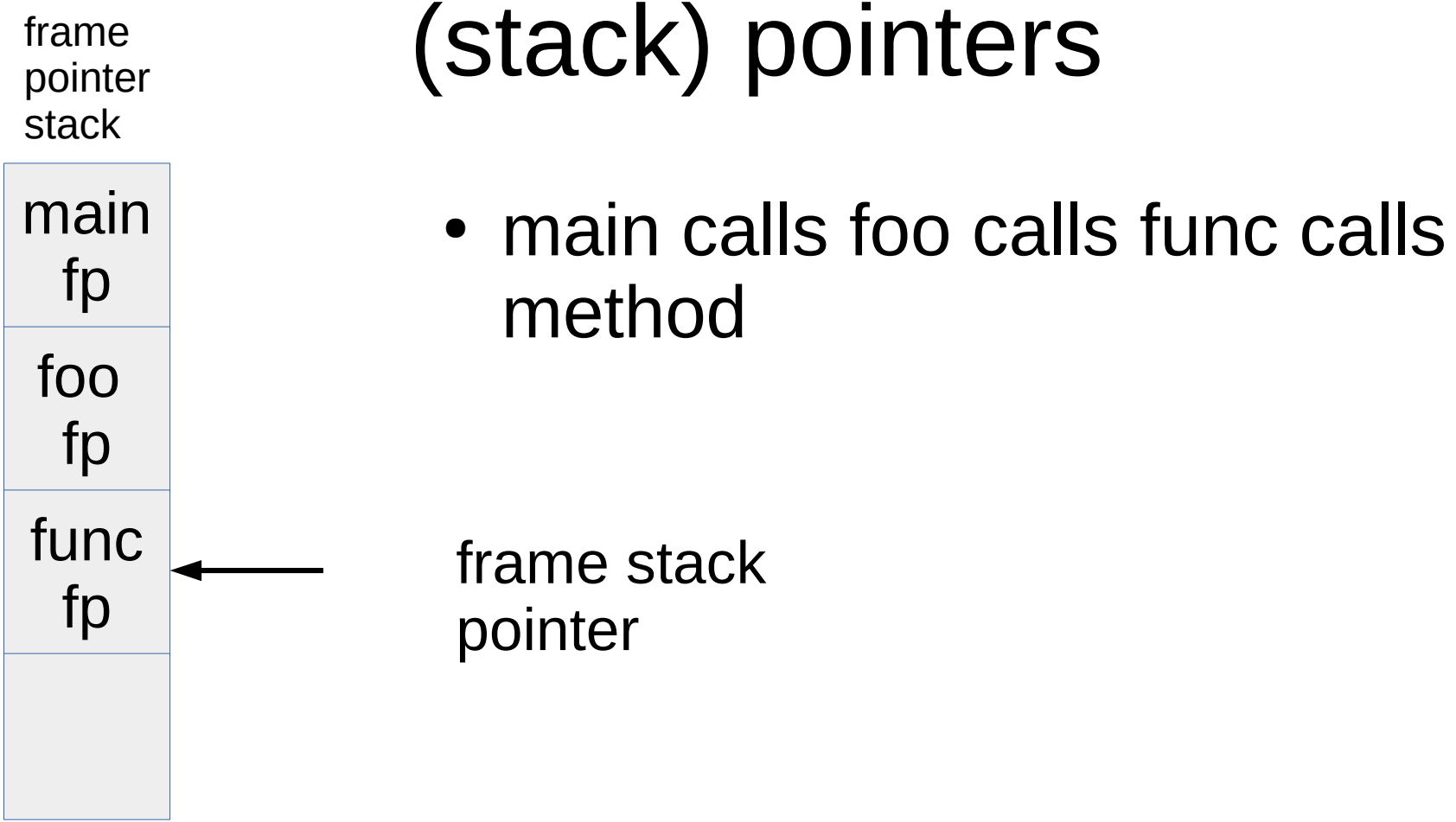
- After the add executes
- The add knows what it is adding because of the types of the objects on the stack
- We can use *operator overloading* to do the operation



Function call instructions

- When a function is called (not necessarily in this order)
  - the stack pointer for the calling function is saved so that it can be restored when the function returns to its caller
  - arguments are pushed onto the stack
  - the program counter to the instruction after the function call code is put on the stack so the called function can jump back to it
  - the code at the start of the function is jumped to

We have a stack to save frame (stack) pointers



- main calls foo calls func calls method

Some instructions

add: 100, or 01100100
rstack[sp-1] = rstack[sp-1] + rstack[sp]
sp--;

printi, 150, or 10010110. Print the integer at the top of the stack
System.out.println(rstack[sp--]);

Some instructions

jmp: 36, or 00100100

meaning: jump to the location at the top of the runtime stack.

```
pc = rstack[sp]
sp = sp-1;
```

jmpc: 40, or 00101000

meaning: jump to the location at the top of the runtime stack is the next

```
if (rstack[sp-1] pc = rstack[sp]
sp = sp-2
```

Some instructions

call: 44, or 00101100

meaning: save the stack pointer for the current frame in the fpstack (frame pointer stack). Jump to the location of the function being called, whose address is on the top of the runtime stack.

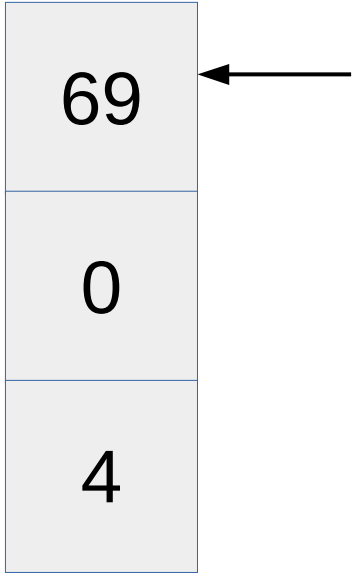
```
fpstack[++fpsp] = sp - rstack[sp]; // subtract off argument stack
                                   // entries
sp--;
pc = rstack[sp—] // set the PC to the address of the label to be
                 // jumped to
```

Some alternatives to implementing this

- Simply have the memory be an array of bytes that are accessed directly (worst)
- Have memory be part of an object that returns objects that correspond to the type of value stored in a memory location, i.e., a pushs byte code, a short value (good, good performance)
- Have memory be an array of pointers to objects representing the value in a memory location (good, good performance)

Some alternatives to implementing this

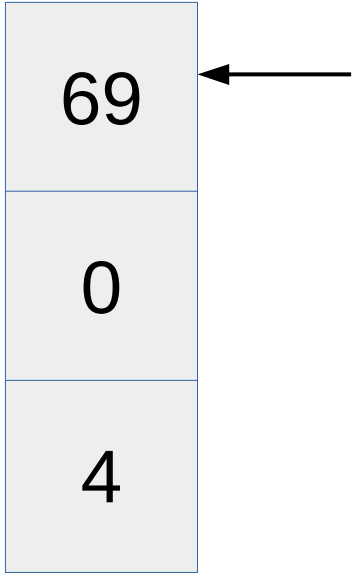
- Have memory be part of an object that returns objects that correspond to the type of value stored in a memory location, i.e., a pushs byte code, a short value (good, good performance)
  - different kinds of byte codes should inherit from a base (abstract) byte code class, which in turn inherits from an (abstract) memory object class




```
Bytecode* bc = getMemory(pc++);
Short* s = getMemory(pc++);
pc++;
// execute a push with s
```

Another way to implementing this

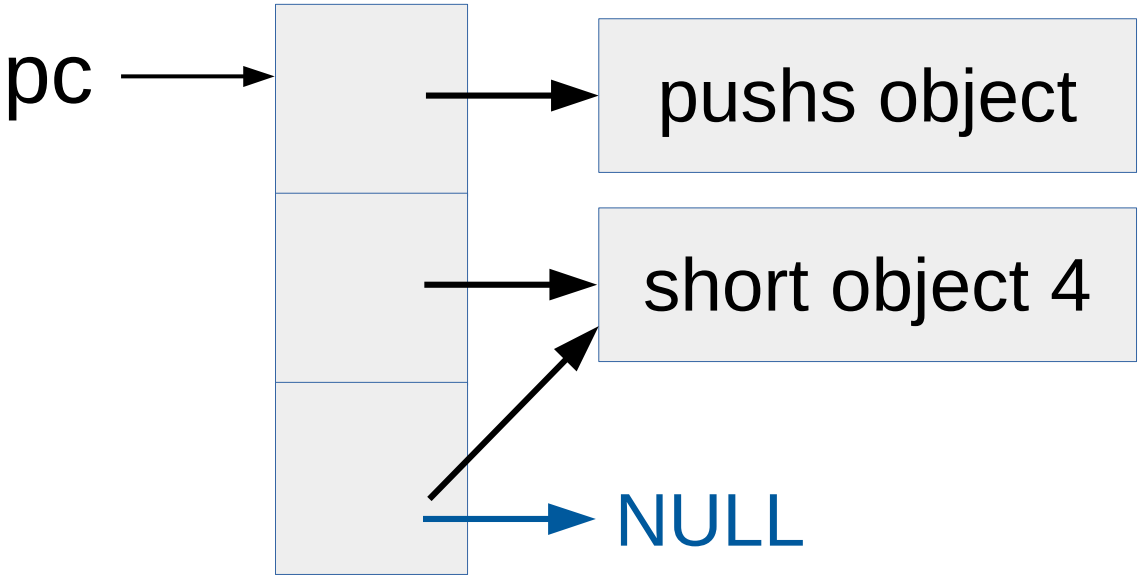
- Have memory be part of an object that returns objects that correspond to the type of value stored in a memory location, i.e., a pushs byte code, a short value (good, good performance)
  - different kinds of byte codes should inherit from a base (abstract) byte code class, which in turn inherits from an (abstract) memory object class




```
Bytecode* bc = getMemory(pc++);
bc->execute( ); // fetches the short
                // from memory and
                // push it onto the
                // stack
```

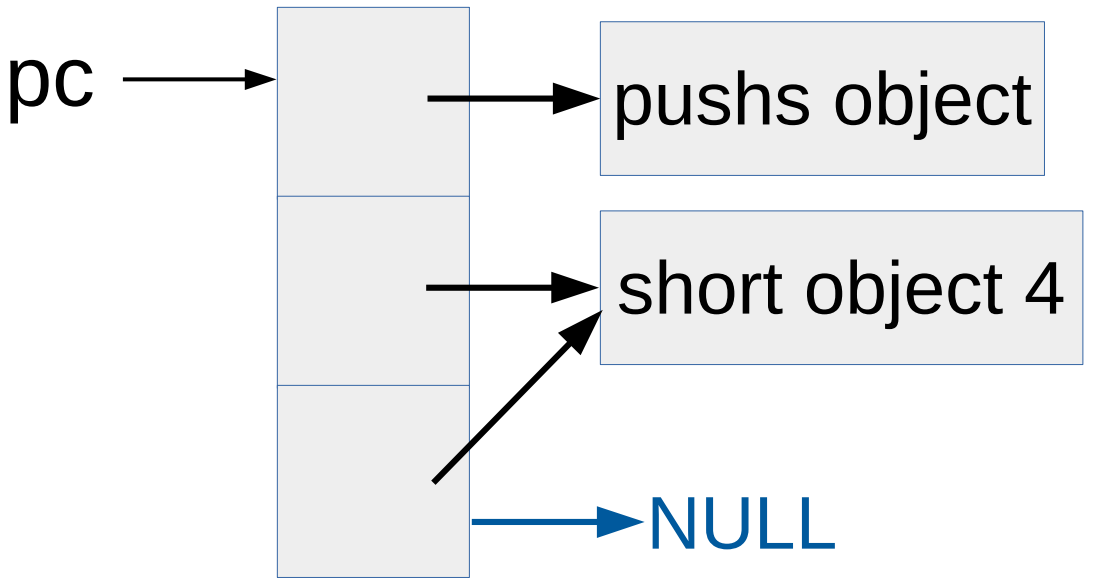
Some alternatives to implementing this

- Have memory be an array of pointers to objects representing the value in a memory location (good, good performance)



Some alternatives to implementing this

- Have memory be an array of pointers to objects representing the value in a memory location (good, good performance)




```
bool cont = true;
while (cont) {
    cont = memory[pc]->execute( );
}
```