

CSCI 4907/6545 Software Security

Fall 2026

Instructor: Jie Zhou

Department of Computer Science

George Washington University



Slides materials are partially credited to Gang Tan of PSU.

Outline of Today's Lecture

- A C Program's Life Journey
- Memory
- Buffer overflows: common pitfalls and exploitation

SECOND EDITION

THE



PROGRAMMING
LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

SOFTWARE SERIES

The C Programming Language

- Invented in 1972 by Dennis Ritchie
- Building blocks for our modern digital world.
 - OS, web servers, databases, systems libraries, ...

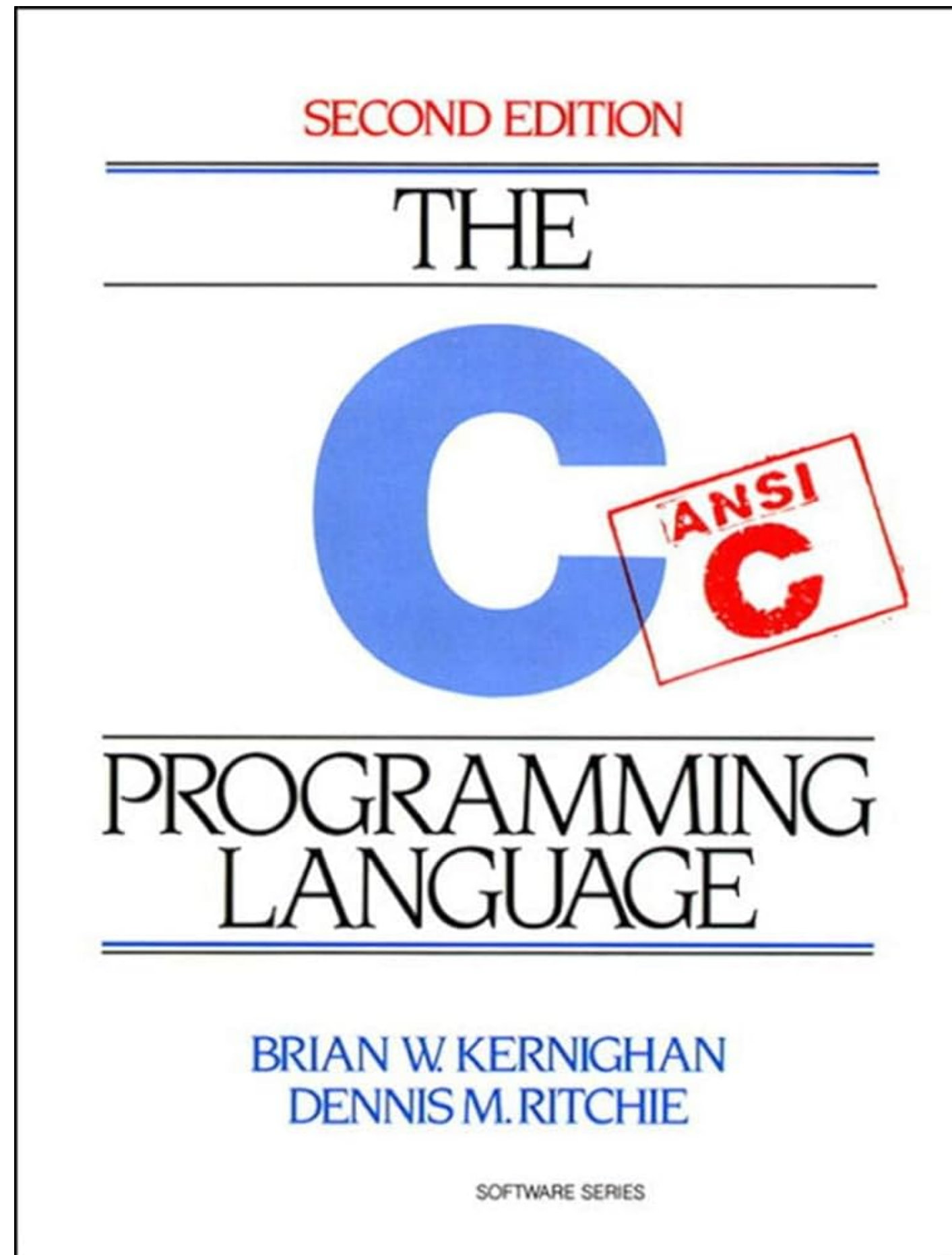


Programming in C is Simple

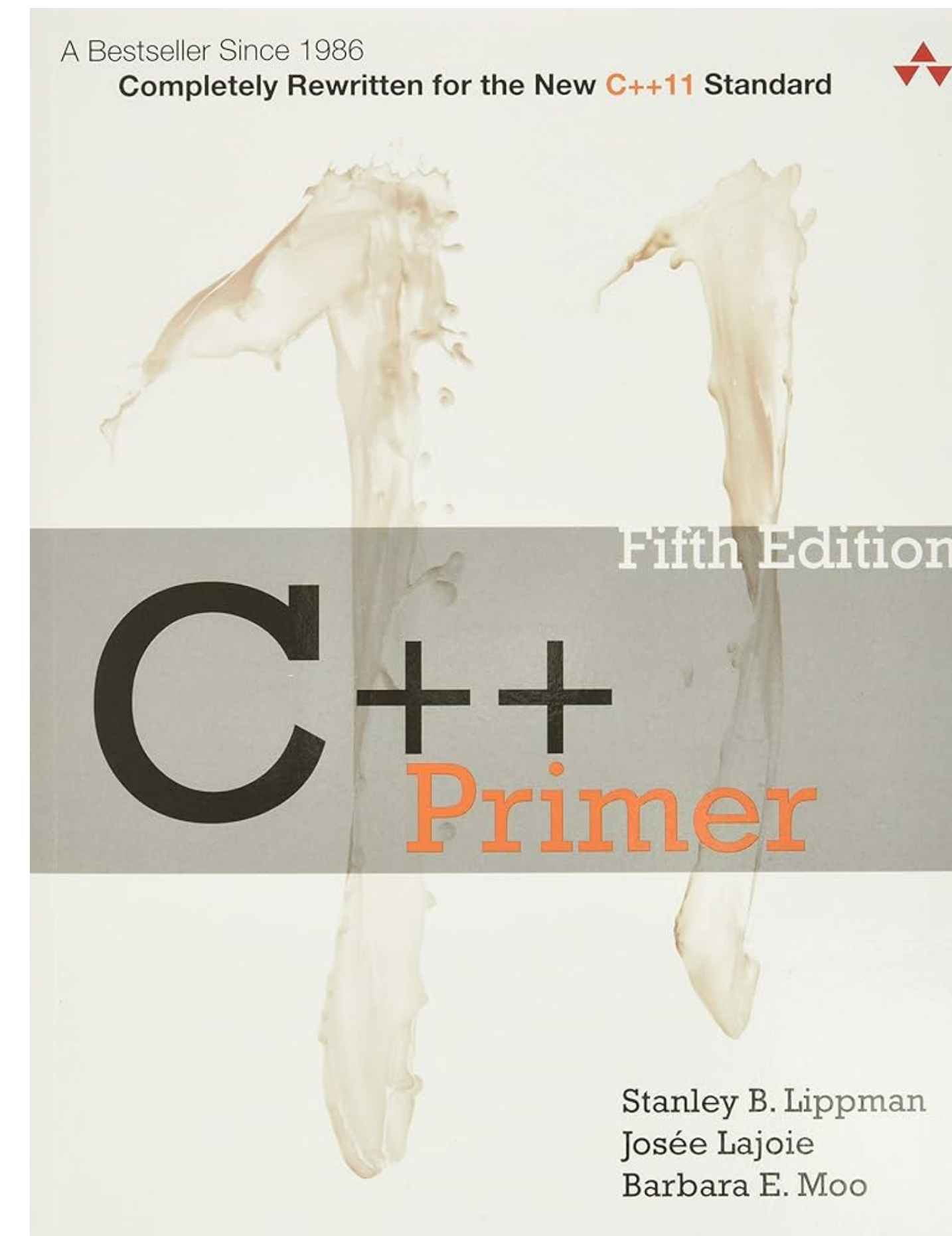
Simple and primitive language features

- Basic data types (char, integer, boolean, etc.)
- struct
- Pointers
- Basic control flow (conditional branches, loops, etc.)

Programming in C is Simple

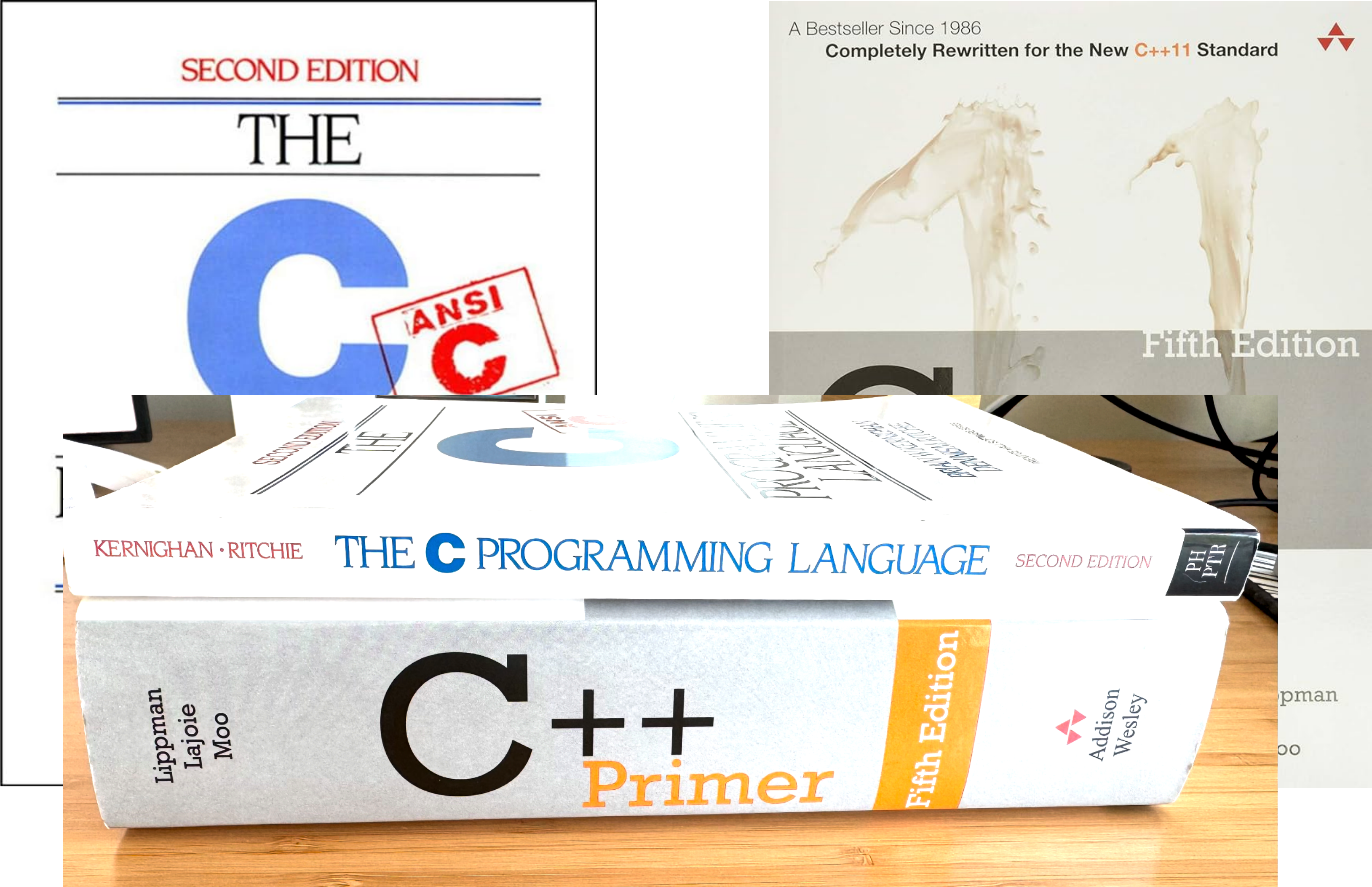


~200 pages



~1,000 pages

Programming in C is Simple



**If so, why do we have so many bugs in
C programs?**

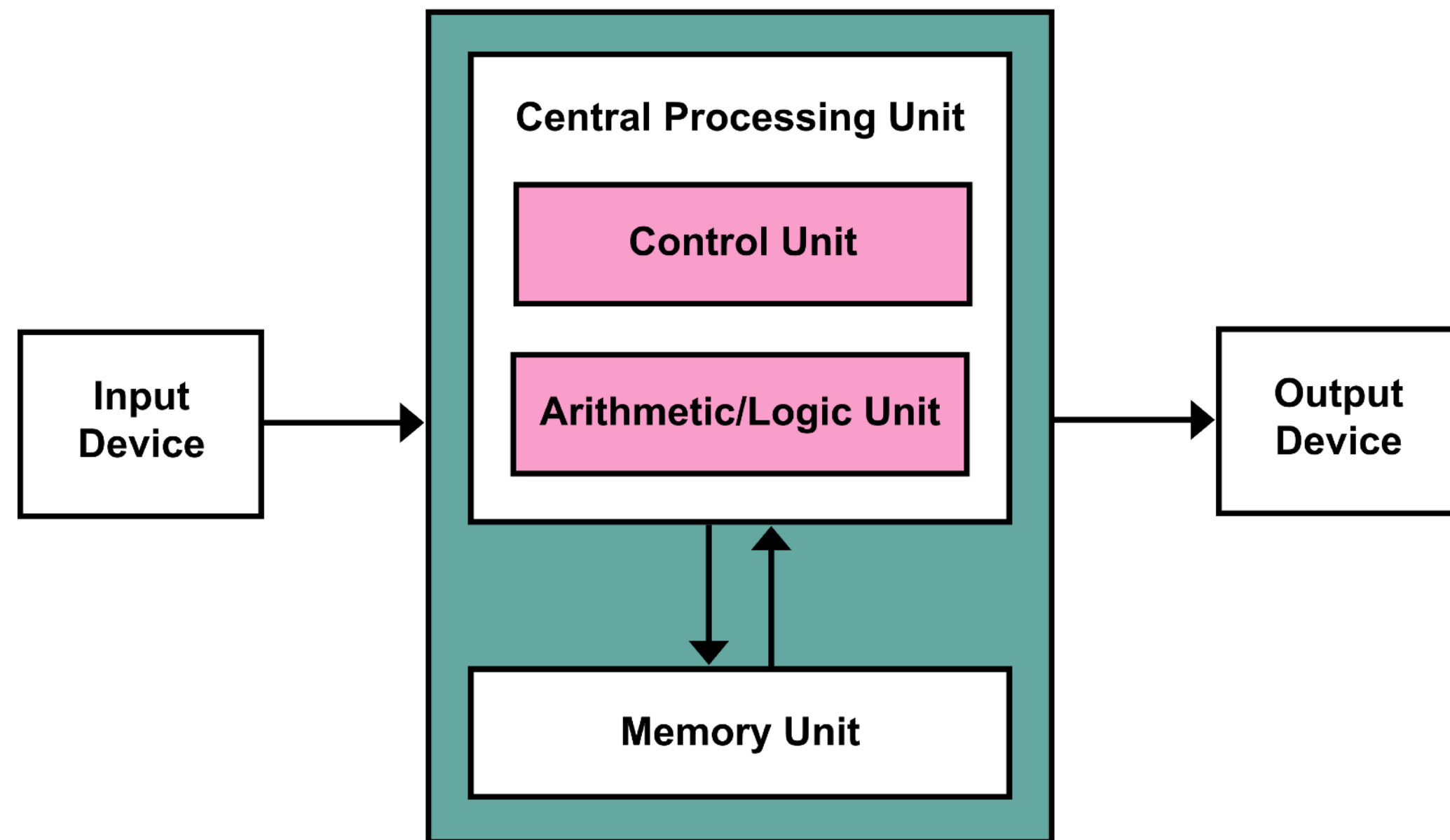
Programming *Correctly* in C is (*Extremely*) Hard

Simple and primitive language features

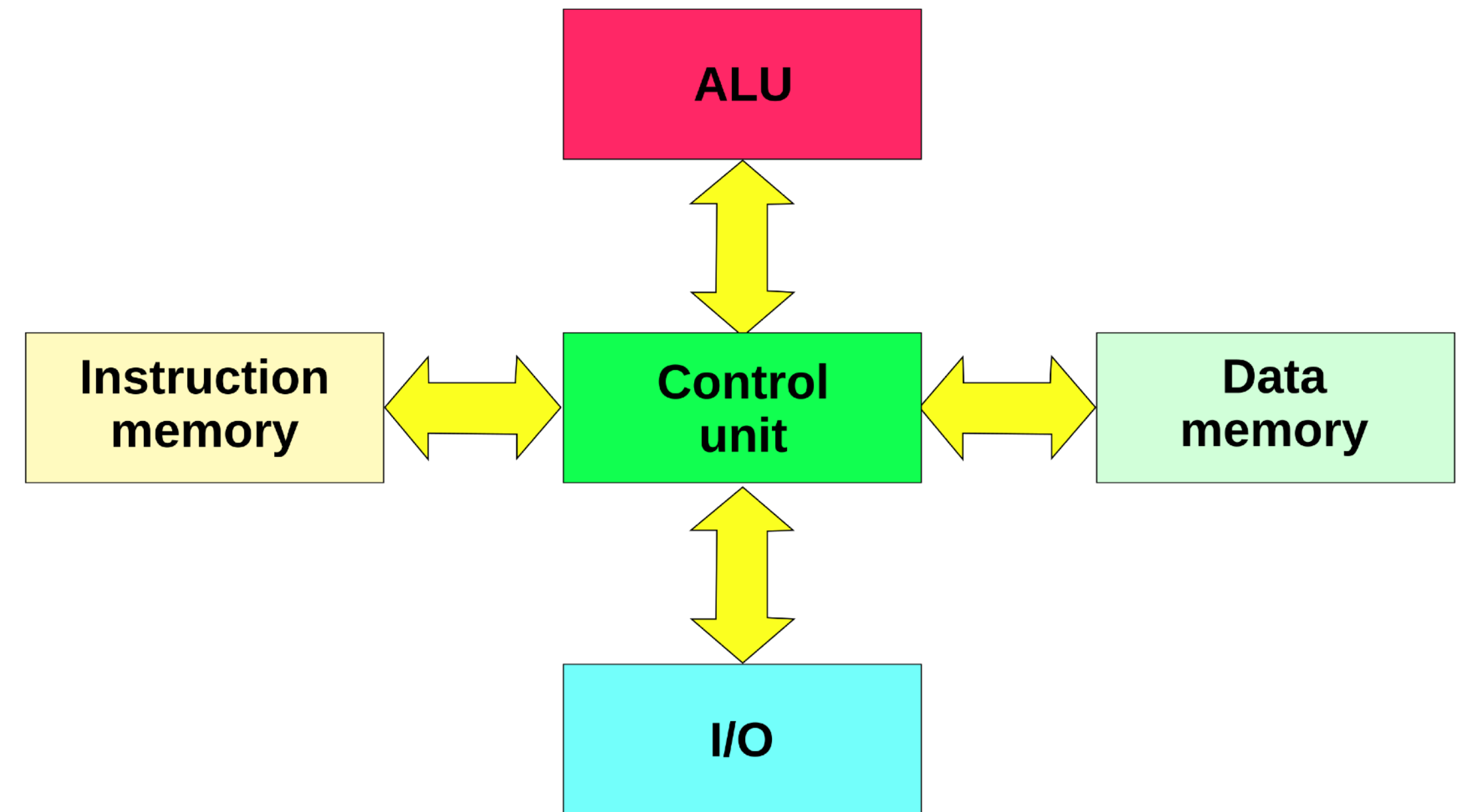
- Basic data types (char, integer, boolean, etc.)
- struct
- **Pointers**
- Basic control flow (conditional branches, loops, etc.)

-  **Pointer:** Capability to manipulate memory.
 - For C, pointer is usually implemented as a virtual address.
 - But this is not the only way to implement pointers.

Architecture of Modern Computers

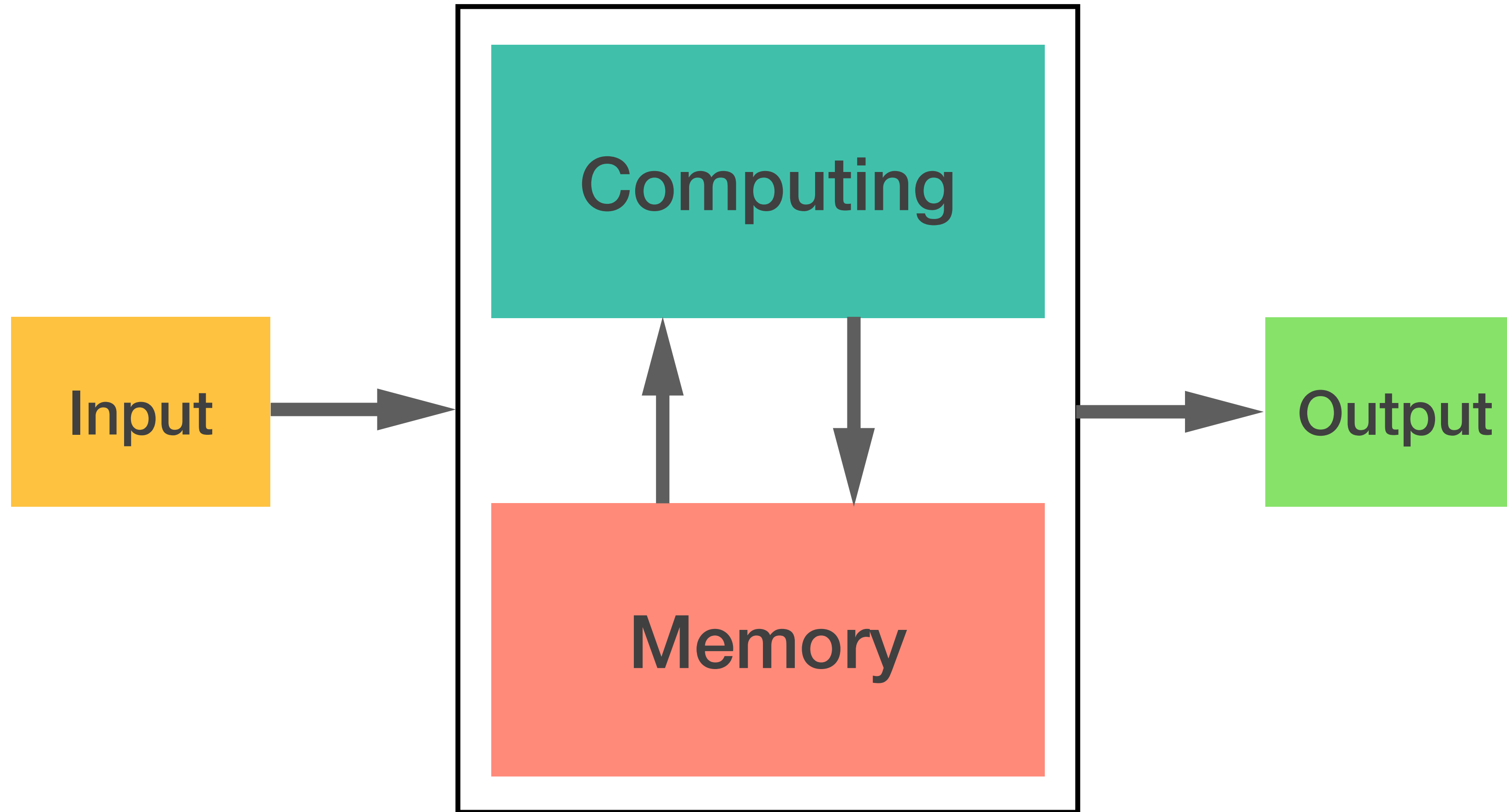


von Neumann Architecture



Harvard Architecture

Architecture of Modern Computers



Programming *Correctly* in C is (Extremely) Hard

Simple and primitive language features

- Basic data types (char, integer, boolean, etc.)
- struct
- **Pointers**
- Basic control flow (conditional branches, loops, etc.)

-  **Pointer:** Capability to manipulate memory.
- For C, pointer is usually implemented as a virtual address.
 - But this is not the only way to implement pointers.

-  **C pointers can do almost arbitrary memory manipulation!**
- The correctness is at the discretion of programmers.

Hello World Program

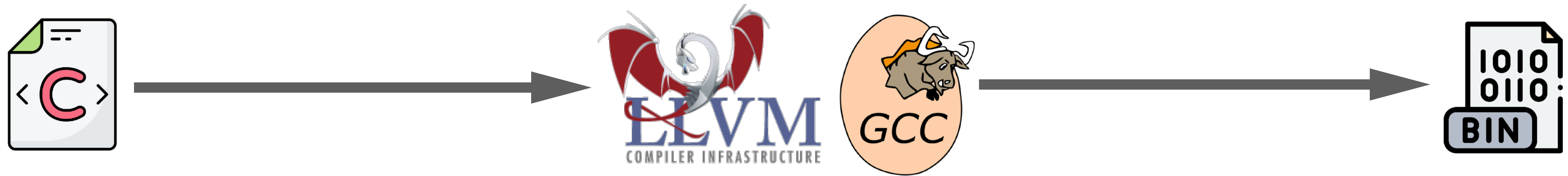
```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     printf("Hello, world!\n");
5
6     return 0;
7 }
```

hello.c

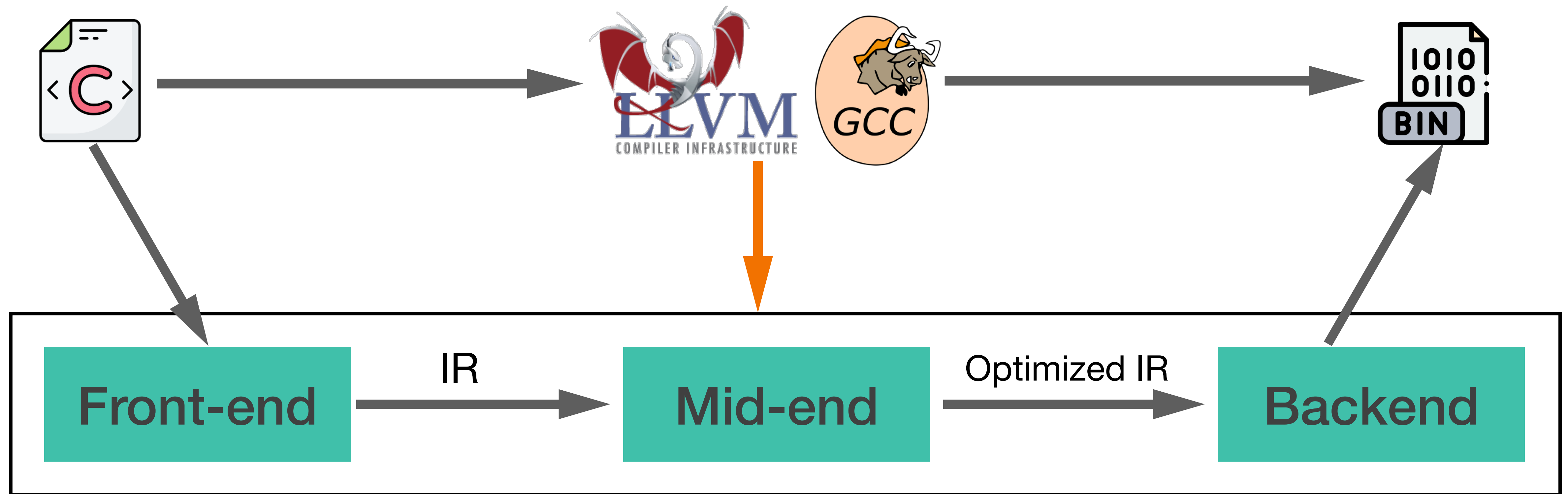
```
$ gcc hello.c -o hello
$ ./hello
```

- Size of this .c file: 98 bytes
 - Source line of code: 7
-
- Size of the hello binary: 17 KB
 - Instructions executed: 657,679

Life of a C Program: Compilation



Life of a C Program: Compilation



- Lexical analysis
- Parsing
- Semantic analysis
- Intermediate Representation (IR) code generation

- IR Optimizations

- Native CodeGen
- Linking

Life of a C Program: Execution



Loading

- Initializing memory layout
- (Optional) Dynamic linking, e.g. `libc`
- Environment initialization, e.g., stack setup
- Setting program counter (PC) to `_start()`

Execution

- `_start()` calls `main()`
- `main()` runs the program

Termination

- `main()` returns,
- `_start()` calls `exit()`
- cleanup and shutdown

CIA Security Triad

- **Confidentiality:** An attacker cannot recover protected data.
- **Integrity:** An attacker cannot modify protected data.
- **Availability:** An attacker cannot stop/hinder computation.



Definition: Threat Model



The abilities and resources of the attacker

- Threat models enable structured reasoning about the attack surface.
- Awareness of entry points (and associated threats) to break into the target.
- Look at systems from an attacker's perspective:
 - Decompose application: **identify structure**
 - Determine and rank threats
 - Determine countermeasures and mitigations

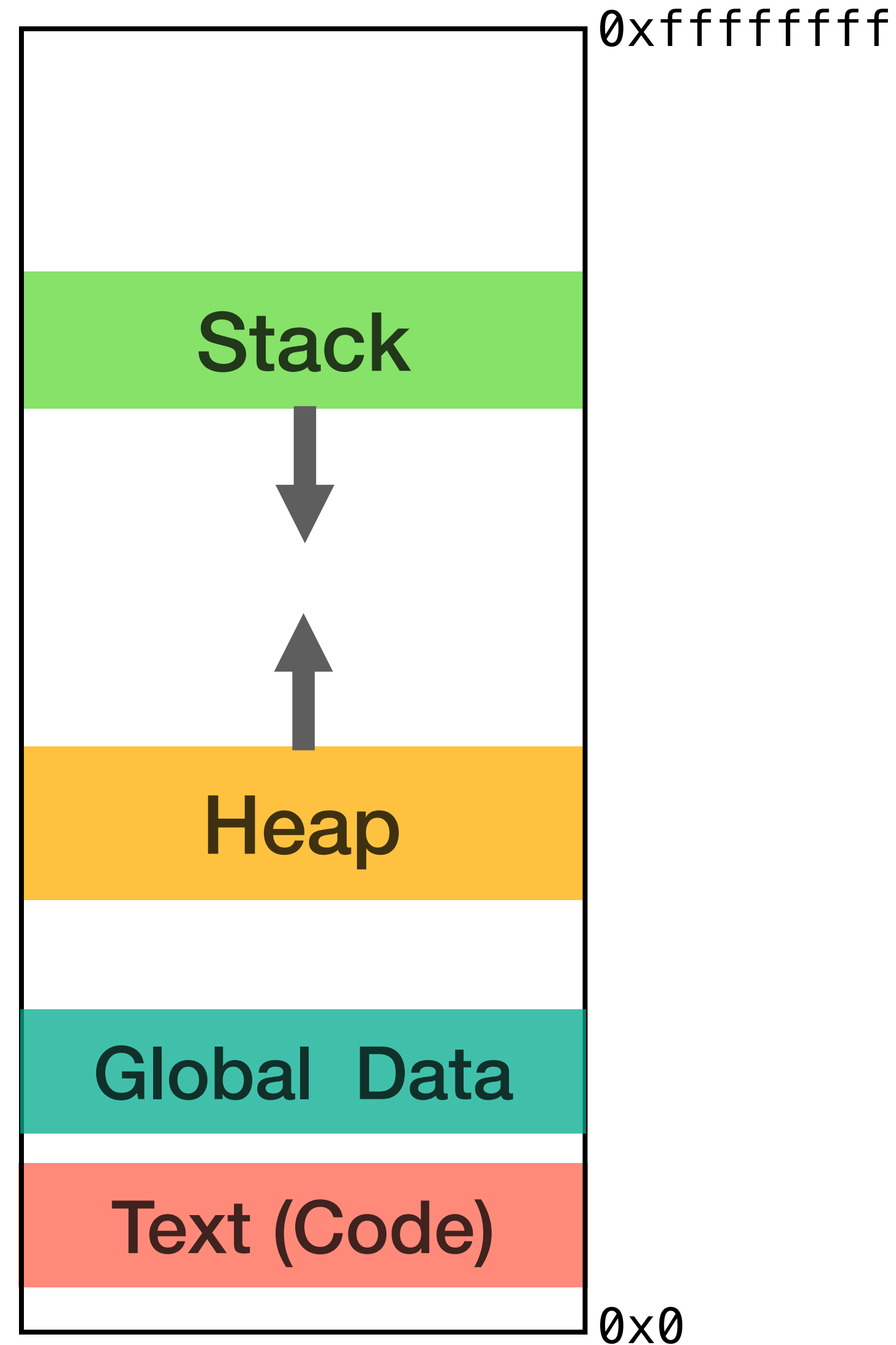
Further reading:

https://owasp.org/www-community/Threat_Modeling

Address Space of a C Program on x86-32

What do programs need in memory?

- Code
- Data
 - Globals
 - Stack for local variables
 - Heap for dynamic memory



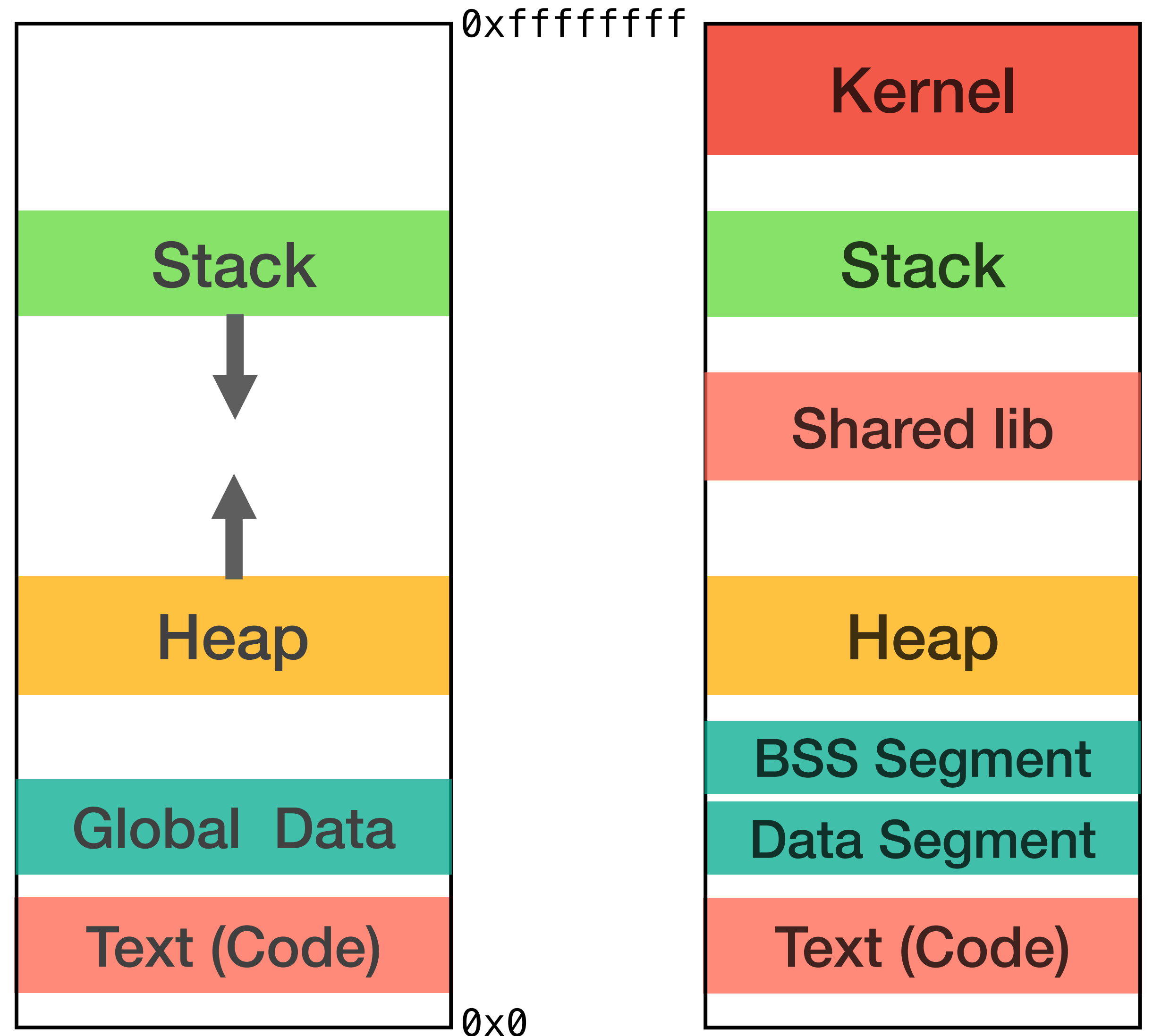
Address Space of a C Program on x86-32

What do programs need in memory?

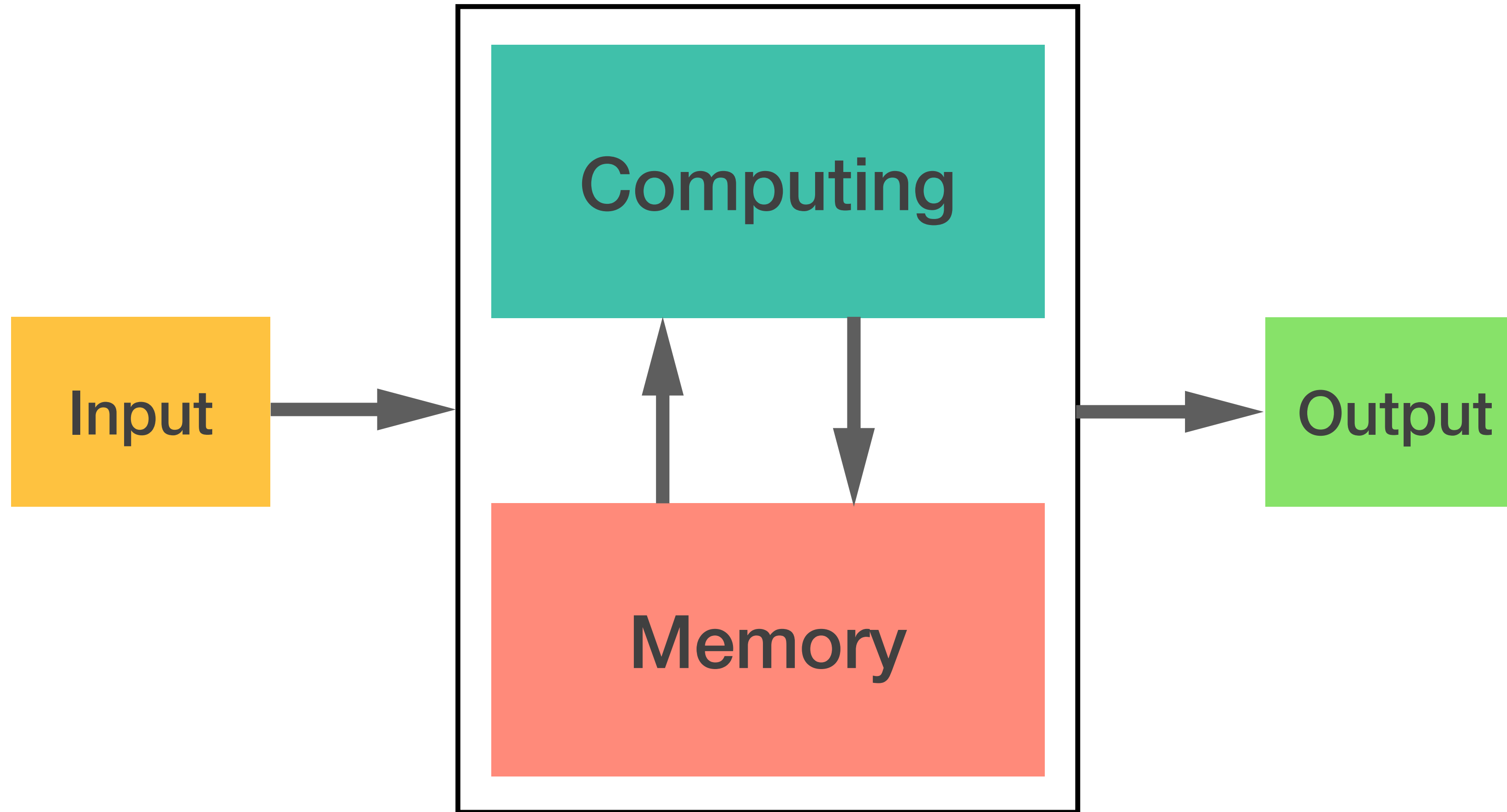
- Code
- Data Segment
 - Initialized global variables
- BSS Segment
 - Uninitialized global data
- Heap
- Shared libraries
- Stack
- Kernel



Check “/proc/pid/maps” to see how memory mapping looks in a real system.



Architecture of Modern Computers



What exactly is memory for programmers?

Usable Memory From a C Programmer's Perspective: Virtual Address Space + Registers

AMD64/x86-64 ISA

- General-purpose registers
 - rax, rbx, rcx, rdx, rsi, rdi, r8–r15
 - rbp, rsp
- Program counter
 - rip
- Segment registers
 - cs, ss, ds, ss, es, fs, gs
- Control registers
 - cr0, cr2, cr3, cr4

What can go wrong in memory?

It is Too Easy to Write Bugs

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     char user_name[32];
5     scanf("%s", user_name);
6     printf("Hello, %s!\n", user_name);
7 }
```



What if user's name is longer than 32 characters?

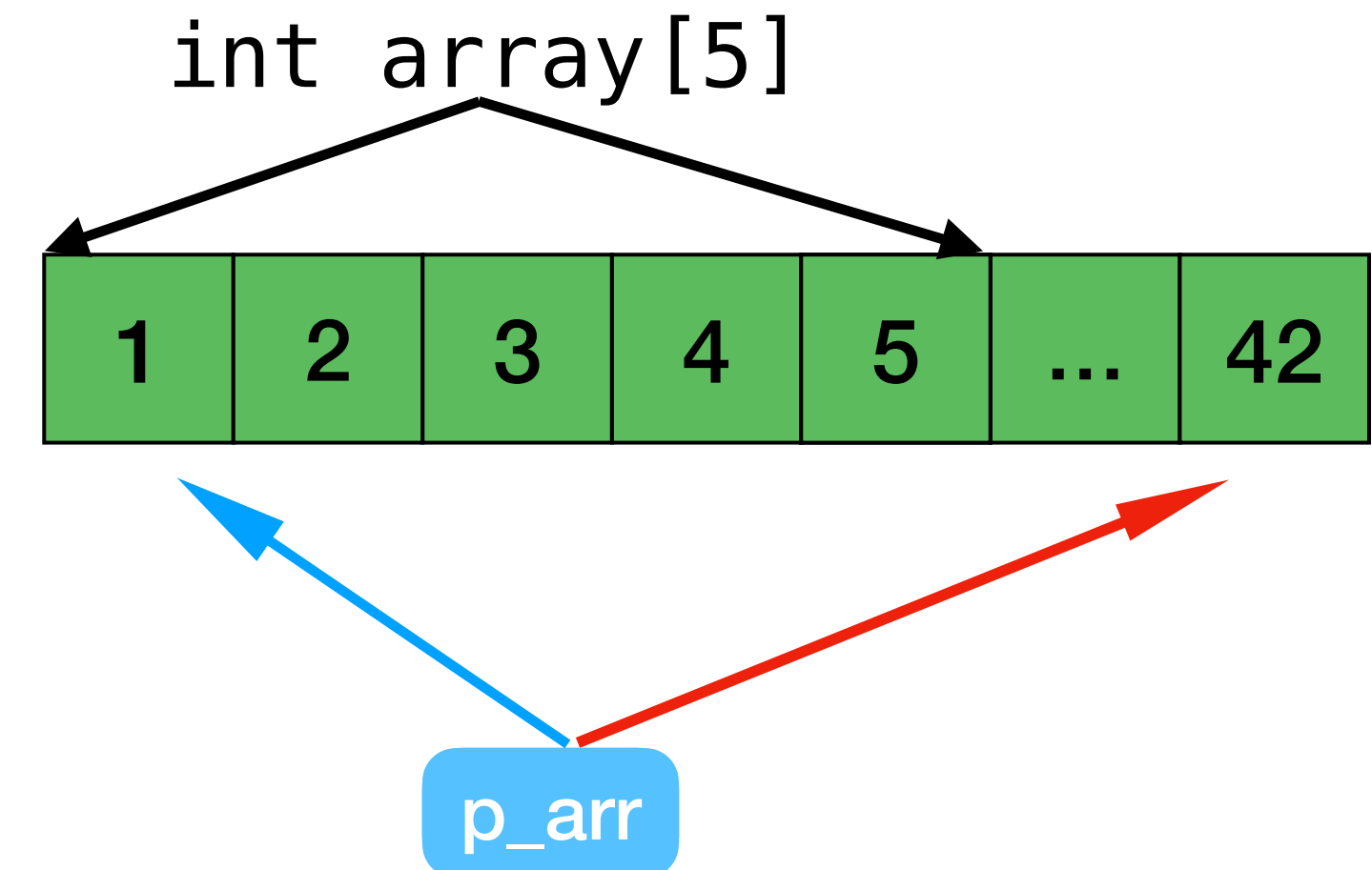


What if the user deliberately/maliciously input something than 32 characters?

Buffer Overflows



Reading/writing a buffer out of its bounds.



- It is C/C++ programmers' job to ensure such errors do not happen.
- In contrast, most modern languages (e.g., Java, Rust, ...) prevent buffer overflows by performing automatic bounds checking.
- The first Internet worm, Morris Worm, and many subsequent ones (CodeRed, Blaster, ...) exploited buffer overflows.
- Buffer overflows are still among the most commonly exploited vulnerabilities.

Buffer Overflows

2025 CWE Top 10 KEV Weaknesses

Top 25 Home | Share via:  | View in table format | KEV Key Insights | KEV Methodology

- 1
- Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
- [CWE-78](#) | CVEs in KEV: 20 | Rank Last Year: 3 (up 2) ▲
- 2
- Use After Free
- [CWE-416](#) | CVEs in KEV: 14 | Rank Last Year: 9 (up 7) ▲
- 3
- Out-of-bounds Write
- [CWE-787](#) | CVEs in KEV: 12 | Rank Last Year: 1 (down 2) ▼
- 4
- Missing Authentication for Critical Function
- [CWE-306](#) | CVEs in KEV: 11 | Rank Last Year: 7 (up 3) ▲
- 5
- Deserialization of Untrusted Data
- [CWE-502](#) | CVEs in KEV: 11 | Rank Last Year: 5
- 6
- Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
- [CWE-22](#) | CVEs in KEV: 10 | Rank Last Year: 6
- 7
- Improper Control of Generation of Code ('Code Injection')
- [CWE-94](#) | CVEs in KEV: 7 | Rank Last Year: 4 (down 3) ▼
- 8
- Authentication Bypass Using an Alternate Path or Channel
- [CWE-288](#) | CVEs in KEV: 6 | Rank Last Year: 14 (up 6) ▲
- 9
- Heap-based Buffer Overflow
- [CWE-122](#) | CVEs in KEV: 6 | Rank Last Year: 11 (up 2) ▲
- 10
- Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- [CWE-79](#) | CVEs in KEV: 7 | Rank Last Year: 21 (up 11) ▲

 **Common Weakness Enumeration**
A community-developed list of SW & HW weaknesses that can become vulnerabilities

Home > CWE Top 25 > 2024 CWE Top 10 KEV Weaknesses

Home | About ▼ | Learn ▼ | Access Content ▼ | Community ▼ | Search ▼

Top 25

2024 CWE Top 10 KEV Weaknesses

Top 25 Home | Share via:  | View in table format | KEV Key Insights | KEV Methodology

- 1
- Out-of-bounds Write
- [CWE-787](#) | CVEs in KEV: 18 | Rank Last Year: 3 (up 2) ▲
- 2
- Access of Resource Using Incompatible Type ('Type Confusion')
- [CWE-843](#) | CVEs in KEV: 6 | Rank Last Year: 8 (up 6) ▲
- 3
- Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
- [CWE-78](#) | CVEs in KEV: 6 | Rank Last Year: 5 (up 2) ▲
- 4
- Improper Control of Generation of Code ('Code Injection')
- [CWE-94](#) | CVEs in KEV: 7 | Rank Last Year: 33 (up 29) ▲
- 5
- Deserialization of Untrusted Data
- [CWE-502](#) | CVEs in KEV: 5 | Rank Last Year: 6 (up 1) ▲
- 6
- Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
- [CWE-22](#) | CVEs in KEV: 5 | Rank Last Year: 9 (up 3) ▲
- 7
- Missing Authentication for Critical Function
- [CWE-306](#) | CVEs in KEV: 6 | Rank Last Year: 10 (up 3) ▲
- 8
- Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
- [CWE-89](#) | CVEs in KEV: 4 | Rank Last Year: 11 (up 3) ▲
- 9
- Use After Free
- [CWE-416](#) | CVEs in KEV: 5 | Rank Last Year: 1 (down 8) ▼
- 10
- Improper Neutralization of Special Elements used in a Command ('Command Injection')
- [CWE-77](#) | CVEs in KEV: 4 | Rank Last Year: 15 (up 5) ▲

One Common Source of Pitfalls: C String Manipulation

Using Strings in C

- C provides many string functions in its libraries (`libc`)
- For example, we use the `strcpy` function to copy one string to another:

```
#include <string.h>

char string1[] = "Hello, world!";
char string2[20];
strcpy(string2, string1);
```

Using Strings in C

- Another lets us compare strings:

```
char string3[] = "this is";
char string4[] = "a test";
if(strcmp(string3, string4) == 0) {
    printf("strings are equal\n");
} else {
    printf("strings are different\n");
}
```

- This code fragment will print "strings are different". Notice that `strcmp` does not return a boolean result.

Note: Use the “man page” to check how to use `libc` functions, e.g., “man `strcmp`”

Other Common String Functions

- `strlen`: Get the length of a string
- `strcat/strncat`: String concatenation
- `gets/fgets`: Receive inputs to a string
- `strdup`: Duplicate a string
- `strstr`: Locate a substring
- ...

Common String Manipulation Errors

- Buffer overflows
- Null-termination errors
- Off-by-one errors
- ...

Unbounded String Copies

```
char *gets(char *s);
```

- Get a string from standard input to the destination buffer
- Does not restrict the size of the input
- Can overflow the destination fixed-size buffer

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     char user_name[32];
5     scanf("%s", user_name);
6     printf("Hello, %s!\n", user_name);
7 }
```

gets: Unbounded String Copies

```
char *gets(char *s);
```

- Get a string from standard input to the destination buffer
- Does not restrict the size of the input
- Can overflow the destination fixed-size buffer

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     char user_name[32];
5     gets(user_name);
6     printf("Hello, %s!\n", user_name);
7 }
```

strcpy and strcat

```
char *strcpy(char *dest, const char *src);  
char *strcat(char *dest, const char *src);
```

- Copy/Concatenate a string to another
- Do not consider the size of the destination buffer
- Can overflow the destination fixed-size buffer

```
int main(int argc, char *argv[]) {  
    char name[2048];  
    strcpy(name, argv[1]);  
    strcat(name, " = ");  
    strcat(name, argv[2]);  
    ...  
}
```

Better String Library Functions

- Functions that restrict the number of bytes are recommended.
- Never use `gets(char *s)`
 - Use `fgets(char *s, int size, FILE *stream)` instead

From gets to fgets

```
char *fgets(char *s, int size, FILE *stream);
```

“fgets reads in at most one less than size characters from stream and stores them into the buffer pointed to by s. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer.”

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     char user_name[32]; 34
5     fgets(user_name, 32, stdin);
6     printf("Hello, %s!\n", user_name);
7 }
```

Better String Library Functions

- Functions that restrict the number of bytes are recommended.
- Never use `gets(char *s)`
 - Use `fgets(char *s, int size, FILE *stream)` instead
- `gets()` has been deprecated since 2007.

Better String Library Functions

- Instead of `strcpy()`, use `strncpy()`
- Instead of `strcat()`, use `strncat()`
- Instead of `sprintf()`, use `snprintf()`

But Still Need Care

```
char *strncpy(char *dest, const char *src, size_t n);
```

- Copy *at most* `n` char from `src` to `dest`. Stop at `n`th char or `'\0'`.
- What happens if the size of `src` is `n` or greater?
 - Only the first `n` char will get copied.
 - `dest` may not be null-terminated!

Any more problems?

- What happens if `dest`'s buffer is smaller than `n`?
 - We may have a buffer overflow bug!

**C Strings Are Assumed/Expected
to Be Null-terminated.**

Null-termination Errors

```
int main(int argc, char* argv[]) {  
    char a[16], b[16];  
    strncpy(a, "0123456789abcdef", sizeof(a));  
    printf("%s", a);  
    strcpy(b, a);  
}
```



What will be printed out?

- a[] not properly terminated
 - Undefined behaviors, e.g., segmentation fault if printf is executed.

```
jie@gwsyssec: /tmp  
$ clang copy.c -o copy  
jie@gwsyssec: /tmp  
$ ./copy  
0123456789abcdef??  
??jie@gwsyssec: /tmp
```

Null-termination Errors

```
int main(int argc, char* argv[]) {  
    char a[16], b[16];  
    strncpy(a, "0123456789abcdef", sizeof(a));  
    printf("%s", a);  
    strcpy(b, a);  
}
```



What will be printed out?

- a[] not properly terminated
 - Undefined behaviors, e.g., segmentation fault if printf is executed.

How to fix it?

```
$ clang copy.c -o copy  
jie@gwsyssec: /tmp  
$ ./copy  
0123456789abcdef??  
??jie@gwsyssec: /tmp  
$ ./copy  
0123456789abcdef(??jie@gwsyssec: /tmp  
$ ./copy  
0123456789abcdef?86?jie@gwsyssec: /tmp
```

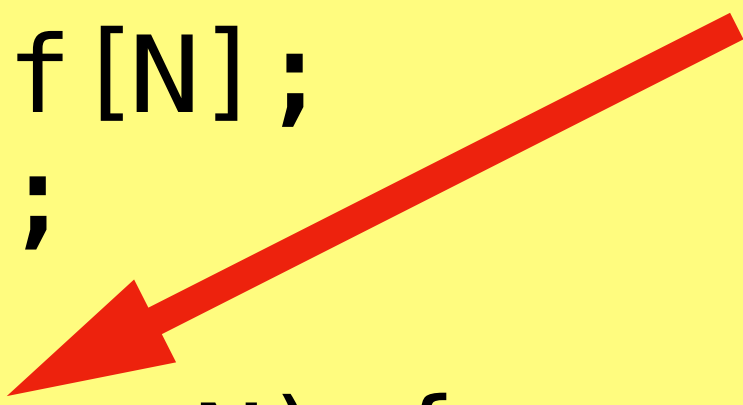
strcpy to strncpy

- Do not replace `strcpy(dest, src)` by
 `strncpy(dest, src, sizeof(dest))`
but by
 `strncpy(dest, src, sizeof(dest) - 1);`
 `dest[sizeof(dest) - 1] = '\0';`
if `dest` should be null-terminated.
- You never have this headache in memory-safe languages (e.g., Rust).
- Further, `strncpy` has big performance penalty vs. `strcpy`.
 - It NIL-fills the remainder of the destination

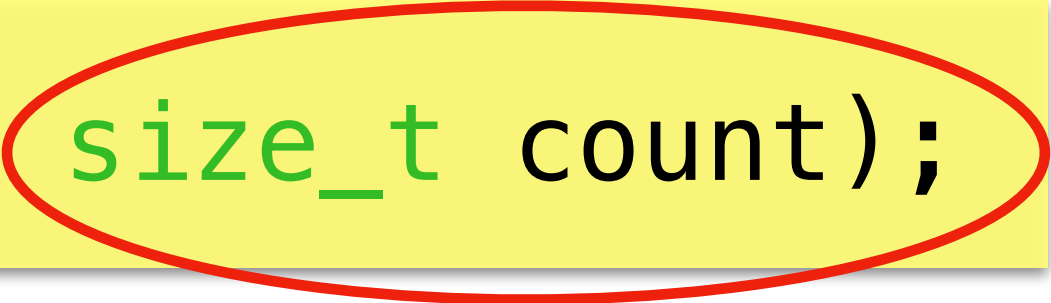
Signed vs. Unsigned Numbers

```
char buf[N];
int len;
...
if (len > N) {
    error("Invalid length");
    return;
}
read(fd, buf, len);
```

What if *len* is negative?



```
ssize_t read(int fd, void *buf, size_t count);
```



len will be cast to unsigned int and negative length overflows,
e.g., -1 -> $2^{32} - 1 = 4294967295$

Checking for Negative Lengths

```
char buf[N];  
int len;  
...  
if (len < 0 || len > N) {  
    error("Invalid length");  
    return;  
}  
read(fd, buf, len);
```

Any other problems?

However, it still has a problem if the buf is going to be treated as a C string.

A Good Version

```
char buf[N];
int len;
...
if (len < 0 || len > N) {
    error("Invliad length");
    return;
}
read(fd, buf, len);
buf[len] = '\0'; // null terminate buf
```

Is it really a good version?

A Good Version

```
char buf[N];  
int len;  
...  
if (len < 0 || len > N - 1) {  
    error("Invalid length");  
    return;  
}  
read(fd, buf, len);  
buf[len] = '\0'; // null terminate buf
```

Exploiting Buffer Overflows

How Can Buffer Overflow *Bugs* Lead to *Vulnerabilities*?

- All the examples look like simple programming bugs.
- How can they possibly enable attackers to do bad things?

Bugs vs. Vulnerabilities



Wikipedia: “A software *bug* is a bug in computer software.”

Wikipedia: “In engineering, a bug is a design **defect** in an engineered system that causes an undesired result.”



Wikipedia: “Vulnerabilities are **flaws** in a computer system that weaken the overall security of the system.”

Vulnerabilities -> Exploitable Bugs

[https://en.wikipedia.org/wiki/Vulnerability_\(computer_security\)](https://en.wikipedia.org/wiki/Vulnerability_(computer_security))

How Can Buffer Overflow *Bugs* Lead to *Vulnerabilities*?

- All the examples look like simple programming bugs.
- How can they possibly enable attackers to do bad things?

Definition: Threat Model



The abilities and resources of the attacker

- Threat models enable structured reasoning about the attack surface.
- Awareness of **entry points** (and associated threats) to break into the target.
- Look at systems from an attacker's perspective:
 - Decompose application: **identify structure**
 - Determine and rank threats
 - Determine countermeasures and mitigations

Further reading:

https://owasp.org/www-community/Threat_Modeling

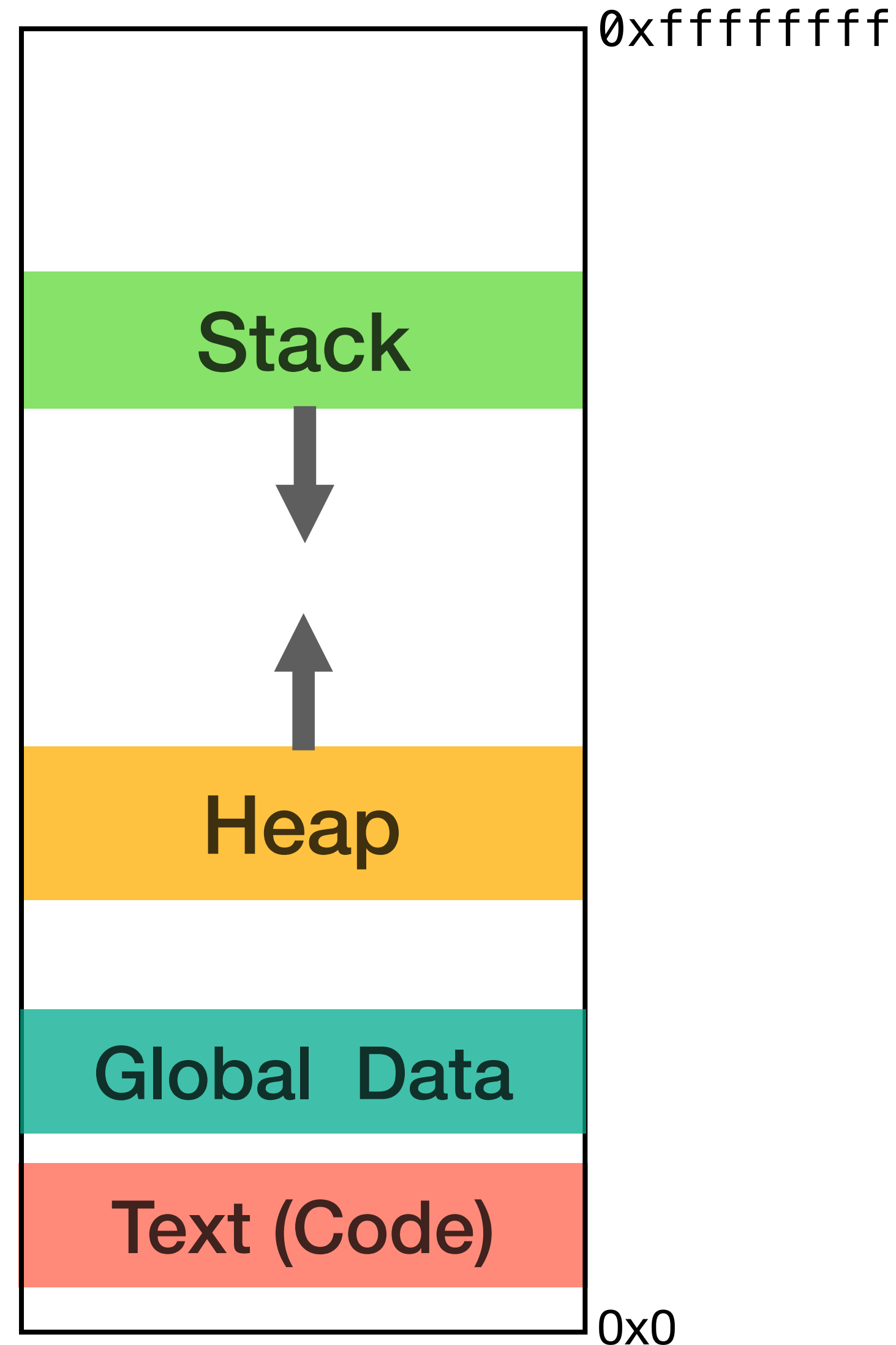
How Can Buffer Overflow *Bugs* Lead to *Vulnerabilities*?

- All the examples look like simple programming bugs.
- How can they possibly enable attackers to do bad things?
- We start with some background.
 - Program stack management
 - AMD64/x86-64

Address Space of a C Program on x86-32

What do programs need in memory?

- Code
- Data
 - Globals
 - Stack for local variables
 - Heap for dynamic memory

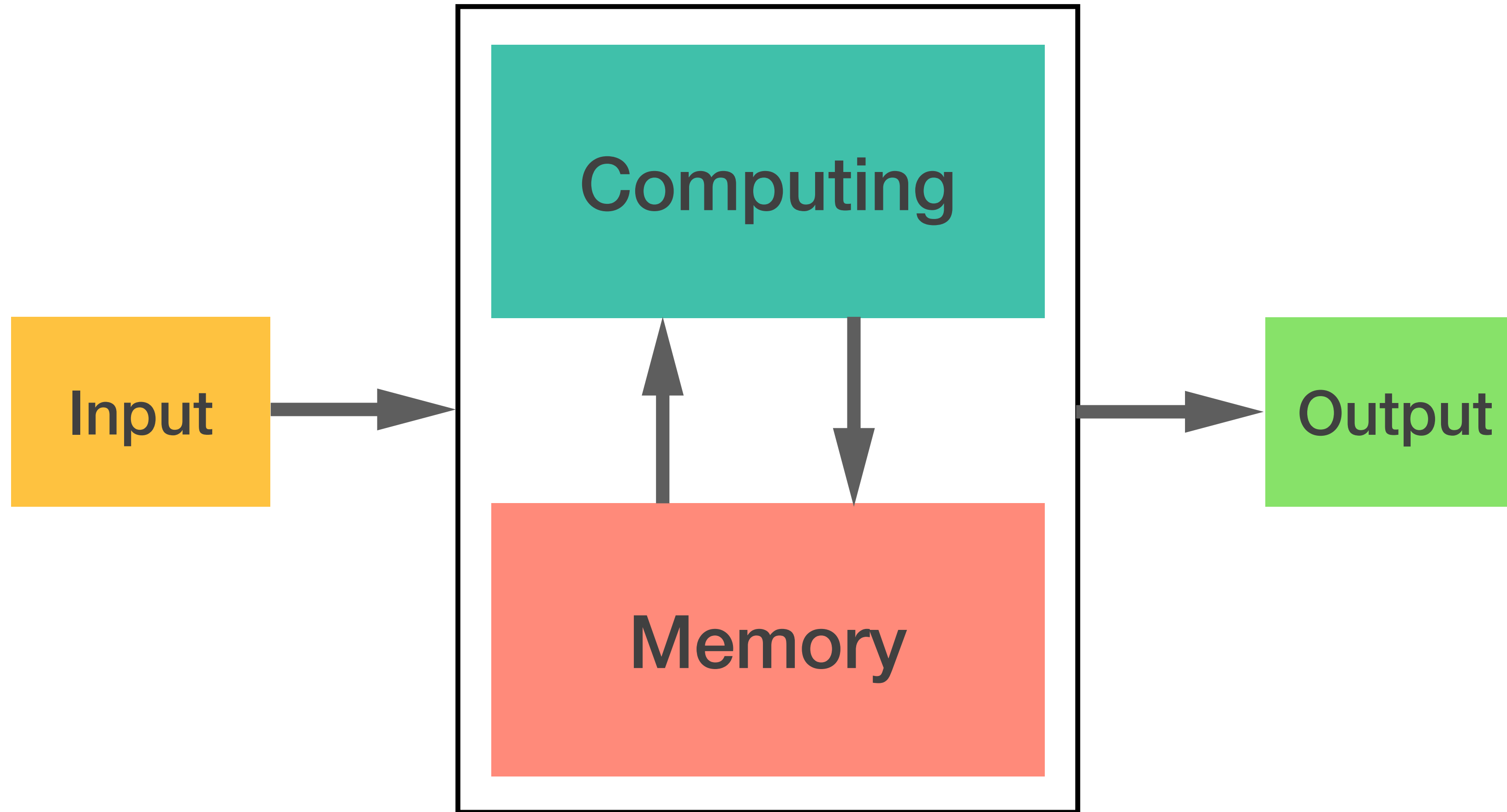


Program Stack

- For implementing function calls and returns

Why do we need functions?

Architecture of Modern Computers



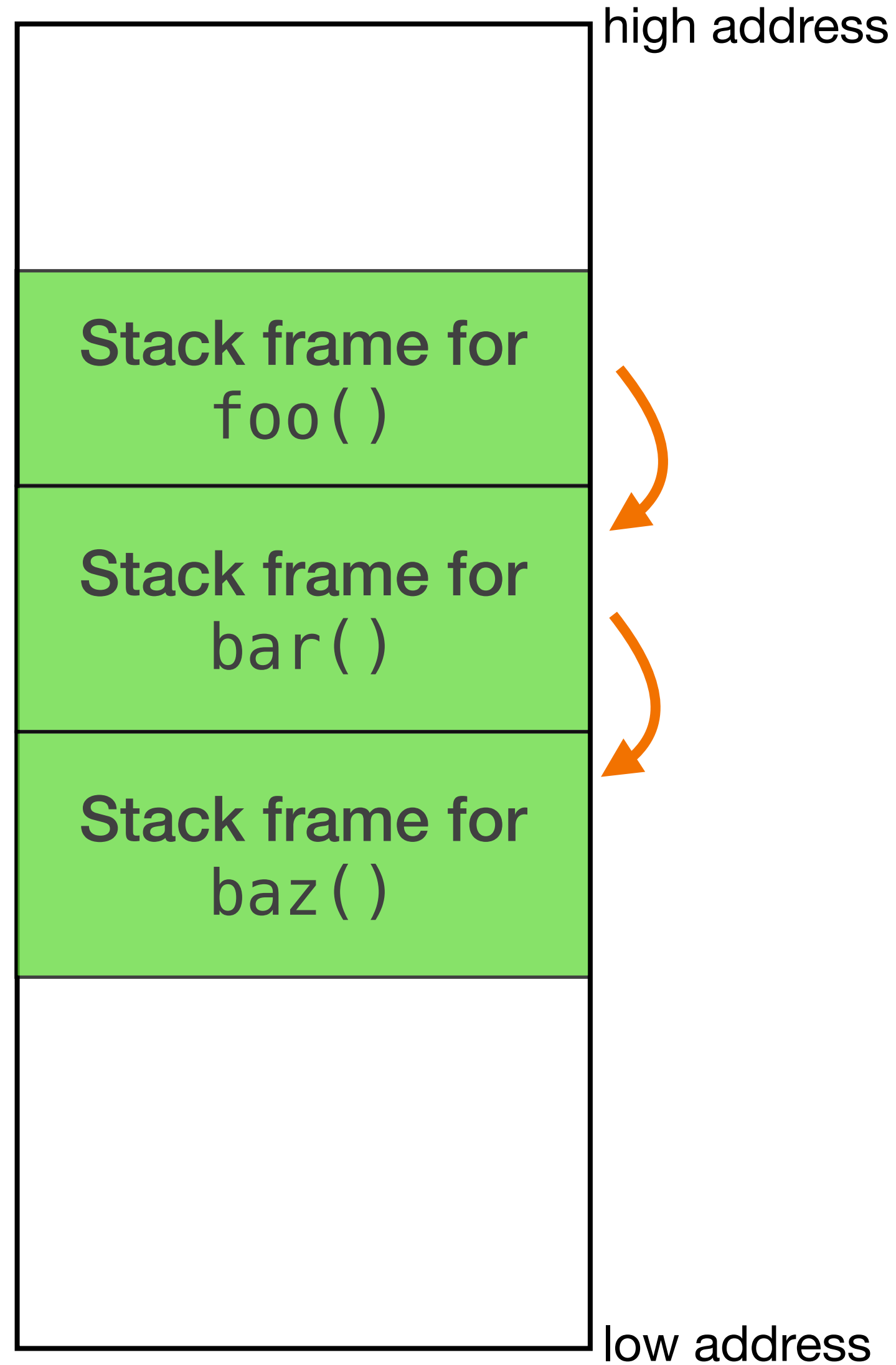
Program Stack

- For implementing function calls and returns
- A stack frame is created for the called function (i.e., the callee)
 - Whenever the caller function calls the callee
- The frame keeps track of program execution state by storing
 - Local variables
 - Some arguments to the callee
 - Depending on the calling convention
 - Return address of the calling function (caller)
 - Other things, such as saved registers and padding for alignment.

Program Stack

```
... foo(...) {  
    ...  
    bar(...);  
    ...  
}  
  
... bar(...) {  
    baz(...);  
    ...  
}  
  
... baz(...) {  
    ...  
}
```

Stack



Stack Frames

- Stack grows from high memory address to low memory address.
- The stack pointer points to the top of the stack.
 - RSP in AMD64/x86-64
- The frame pointer points to the base of the current frame.
 - also called the base pointer
 - RBP in AMD64/x86-64
- The stack is modified during
 - function calls, by the caller
 - function initialization, by the callee
 - function execution, by the callee
 - returning from a function, by the callee

Calling Convention



How functions call and return at the machine-code level.

- How arguments are passed
- How return values are passed
- Which registers are caller-saved / callee-saved

```
void foo() {  
    ...  
    bar(a, b, c, d, e, f, g, h);  
    ...  
}  
  
long bar(long a, long b, long c, long d,  
         long e, long f, long g, long h) {  
    long xx = a * b * c * d * e * f * g * h;  
    long yy = a + b + c + d + e + f + g + h;  
    long zz = utilfunc(xx, yy, xx % yy);  
    return zz + 20;  
}
```

- Where to put all the arguments?
- Where to put the return value?

Usable Memory From a C Programmer's Perspective: Virtual Address Space + Registers

Background: AMD64/x86-64

- Pointers and long integer are 64-bit long.
 - Integer arithmetic operations support 8, 16, 32, and 64 bits.
- 16 general-purpose registers; each 64-bit long

AMD64/x86-64 ISA

- General-purpose registers
 - rax, rbx, rcx, rdx, rsi, rdi, r8–r15
 - rbp, rsp
- Program counter
 - rip
- Segment registers
 - cs, ss, ds, ss, es, fs, gs
- Control registers
 - cr0, cr2, cr3, cr4

Background: AMD64/x86-64

- Pointers and long integer are 64-bit long
 - Integer arithmetic operations support 8, 16, 32, and 64 bits
- 16 general-purpose registers; each 64-bit long
- Calling conventions pass arguments first in registers, then via stack.
 - System V AMD 64 ABI: Pass the first 6 arguments in registers
 - UNIX-like operating systems (e.g. Linux) use this calling convention.
 - Microsoft has its own calling convention.
 - As a result, some procedures do not need to access the stack at all.

Calling Convention



How functions call and return at the machine-code level.

- How arguments are passed
- How return values are passed
- Which registers are caller-saved / callee-saved

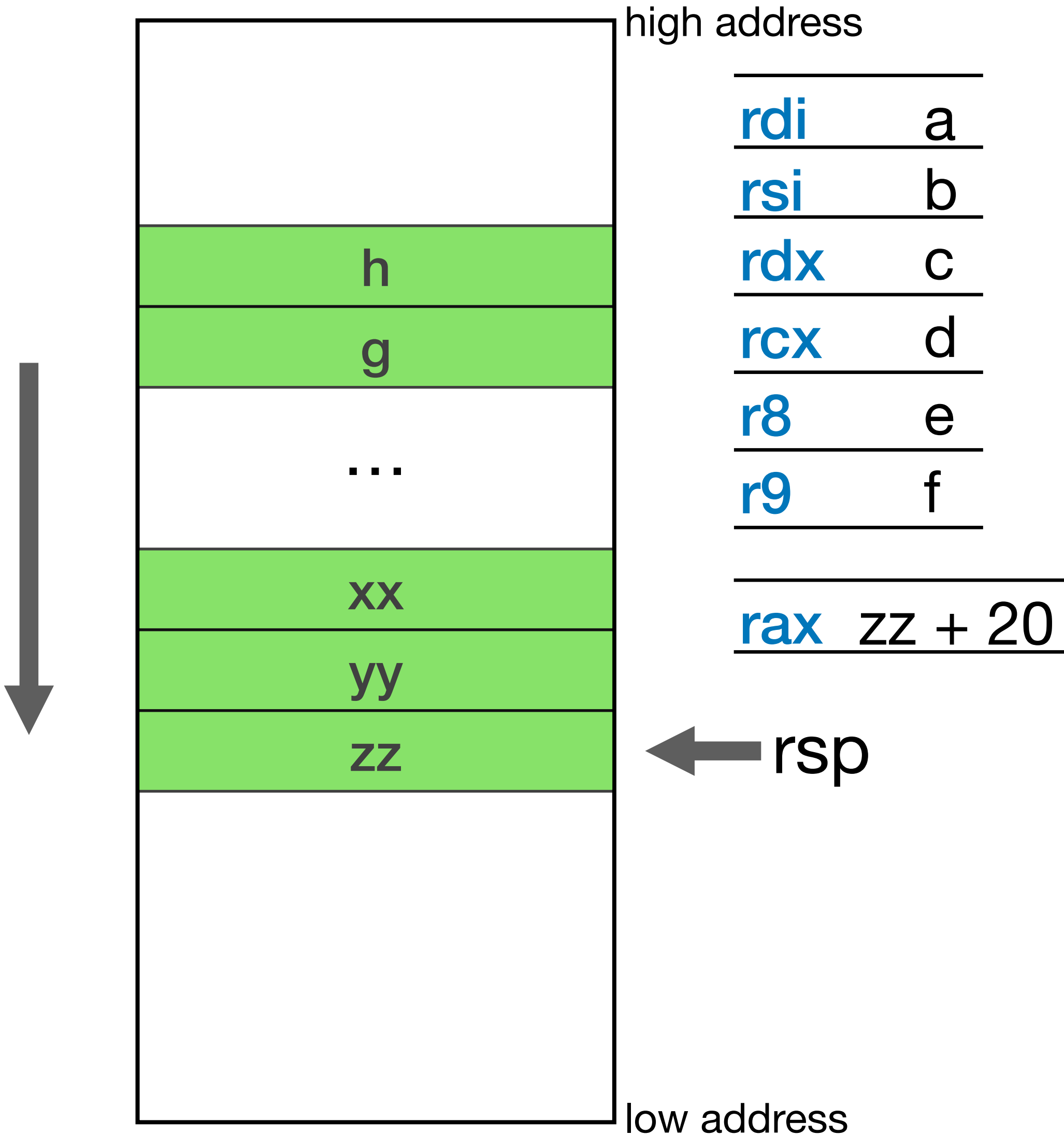
```
void foo() {  
    ...  
    bar(a, b, c, d, e, f, g, h);  
    ...  
}  
  
long bar(long a, long b, long c, long d,  
         long e, long f, long g, long h) {  
    long xx = a * b * c * d * e * f * g * h;  
    long yy = a + b + c + d + e + f + g + h;  
    long zz = utilfunc(xx, yy, xx % yy);  
    return zz + 20;  
}
```

- Where to put all the arguments?
- Where to put the return value?
- Arguments are passed
 - in registers: rdi, rsi, rdx, rcx, r8, r9
 - then via stack
- Return value is passed via
 - in registers: rax, rdx
 - then via stack

System V AMD64 Calling Convention

```
void foo() {
    ...
    bar(a, b, c, d, e, f, g, h);
    ...
}

long bar(long a, long b, long c, long d,
        long e, long f, long g, long h) {
    long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = utilfunc(xx, yy, xx % yy);
    return zz + 20;
}
```

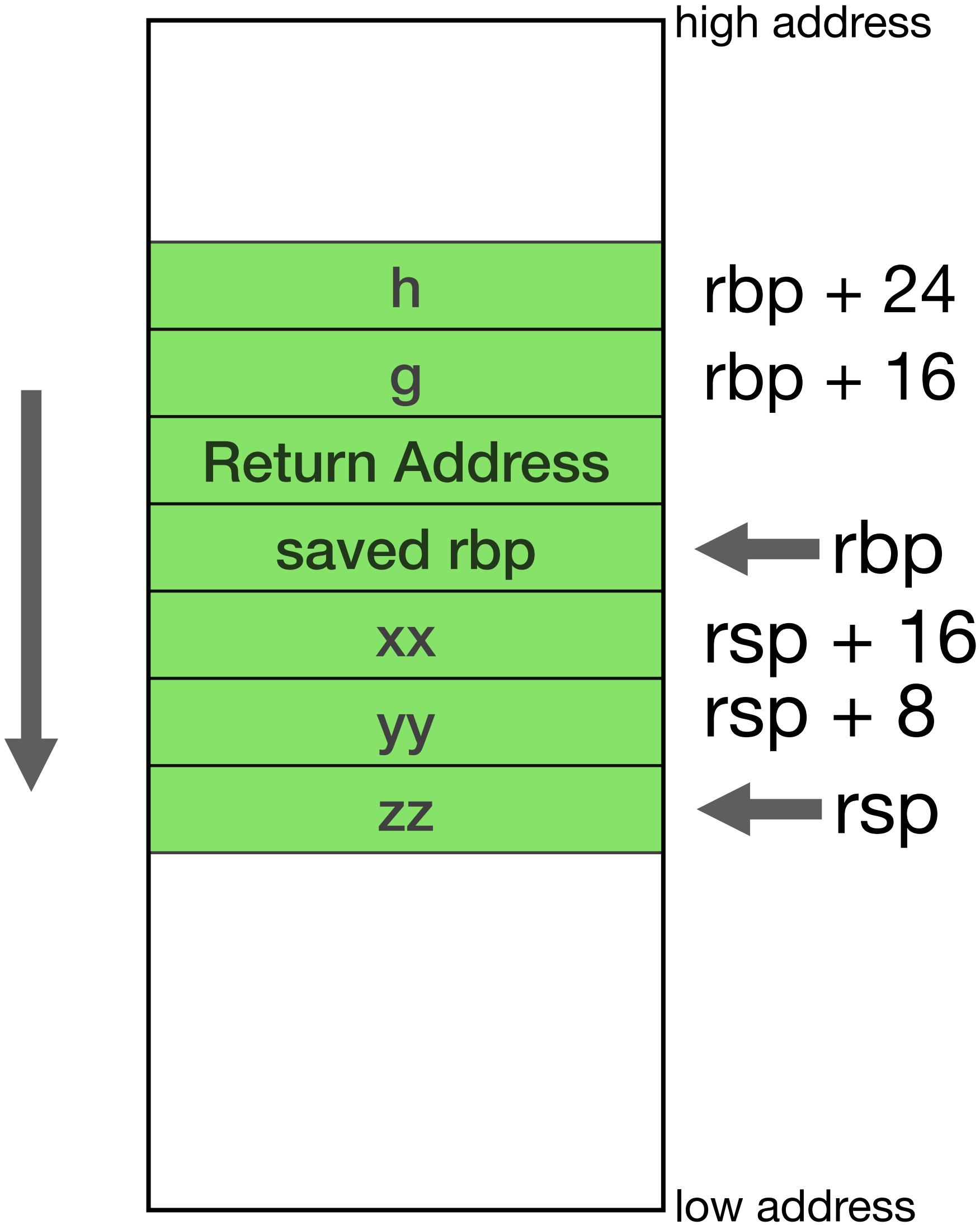


- 💡 What is missing in the frame?
- There is only one `rbp` & `rsp`.
 - How to return to the caller?

System V AMD64 Calling Convention

```
void foo() {
    ...
    bar(a, b, c, d, e, f, g, h);
    ...
}

long bar(long a, long b, long c, long d,
        long e, long f, long g, long h) {
    long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = utilfunc(xx, yy, xx % yy);
    return zz + 20;
}
```



💡 What is missing in the frame?

- There is only one rbp & rsp.
- How to return to the caller?

How are function frames set up?

Function Frame Setup

1. Callsite
2. Function Initialization
3. Function Return

Example Illustrating Stack Buffer Overflows

```
void foo(int a, int b) {  
    char buffer[12];  
    gets(buffer);  
    return;  
}  
  
int main() {  
    int x;  
    x = 0;  
    foo(1,2);  
    x = 1;  
    printf("%d\n",x);  
    return 0;  
}
```

(gdb) disassemble main

Dump of assembler code for function main:

0x0000000000001170	<+0>:	push	%rbp
0x0000000000001171	<+1>:	mov	%rsp,%rbp
0x0000000000001174	<+4>:	sub	\$0x10,%rsp
0x0000000000001178	<+8>:	movl	\$0x0,-0x4(%rbp)
0x000000000000117f	<+15>:	movl	\$0x0,-0x8(%rbp)
0x0000000000001186	<+22>:	mov	\$0x1,%edi
0x000000000000118b	<+27>:	mov	\$0x2,%esi
0x0000000000001190	<+32>:	call	0x1150 <foo>
0x0000000000001195	<+37>:	movl	\$0x1,-0x8(%rbp)
0x000000000000119c	<+44>:	mov	-0x8(%rbp),%esi
0x000000000000119f	<+47>:	lea	0xe5e(%rip),%rdi
0x00000000000011a6	<+54>:	mov	\$0x0,%al
0x00000000000011a8	<+56>:	call	0x1030 <printf@plt>
0x00000000000011ad	<+61>:	xor	%eax,%eax
0x00000000000011af	<+63>:	add	\$0x10,%rsp
0x00000000000011b3	<+67>:	pop	%rbp
0x00000000000011b4	<+68>:	ret	

Compiled by clang-14 on Linux/AMD64

Function Calls

foo(1, 2);

```
0x00000000000001186 <+22>:    mov    $0x1,%edi
0x0000000000000118b <+27>:    mov    $0x2,%esi
0x00000000000001190 <+32>:    call   0x1150 <foo>
```

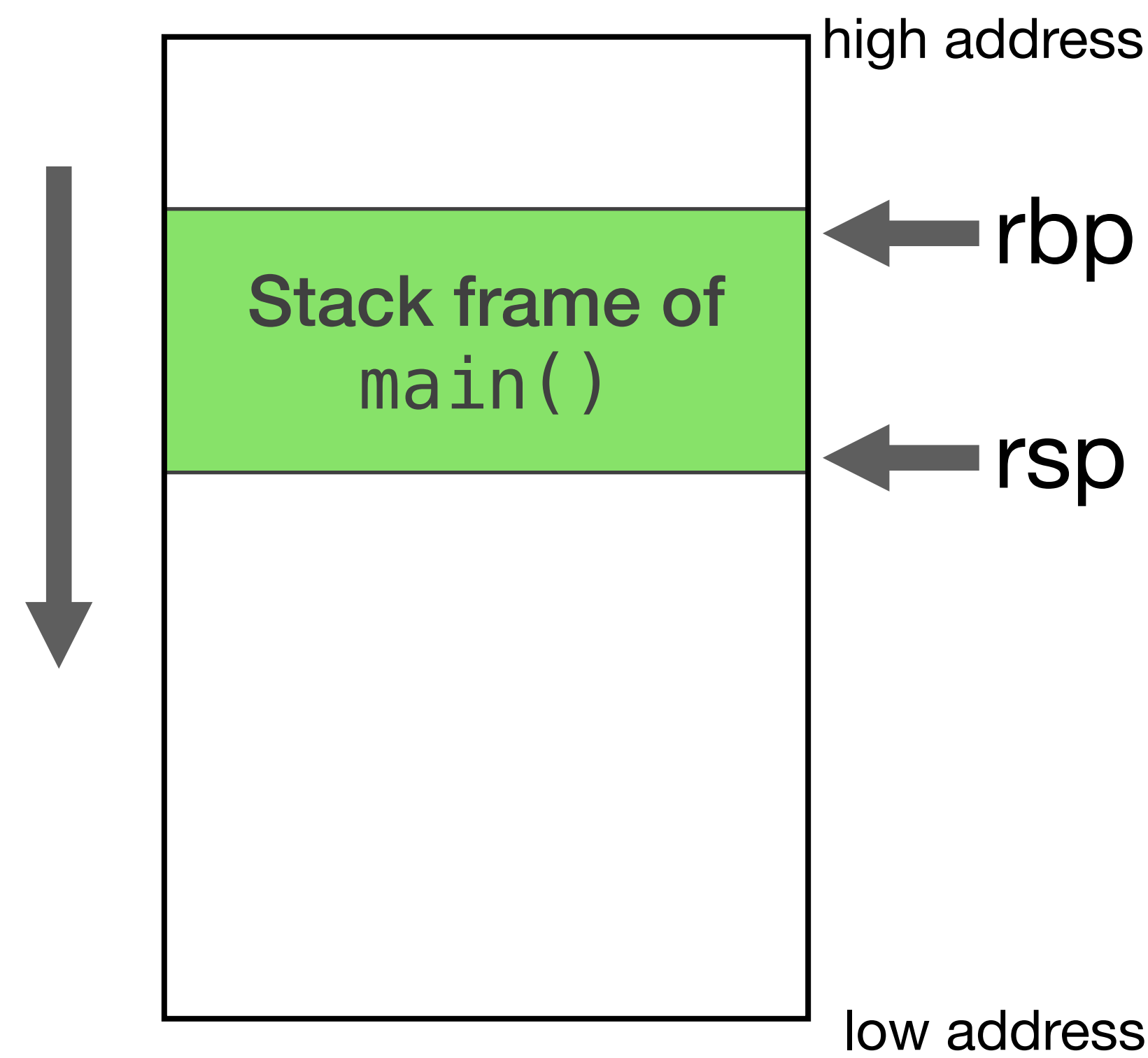
- Arguments are passed
 - in registers: rdi, rsi, rdx, rcx, r8, r9, then via stack

- Pass the 1st argument to edi (the lower half of rdi)
- Pass the 2nd argument to esi (the lower half of rsi)
- Push the return address onto the stack, and jump to the callee function

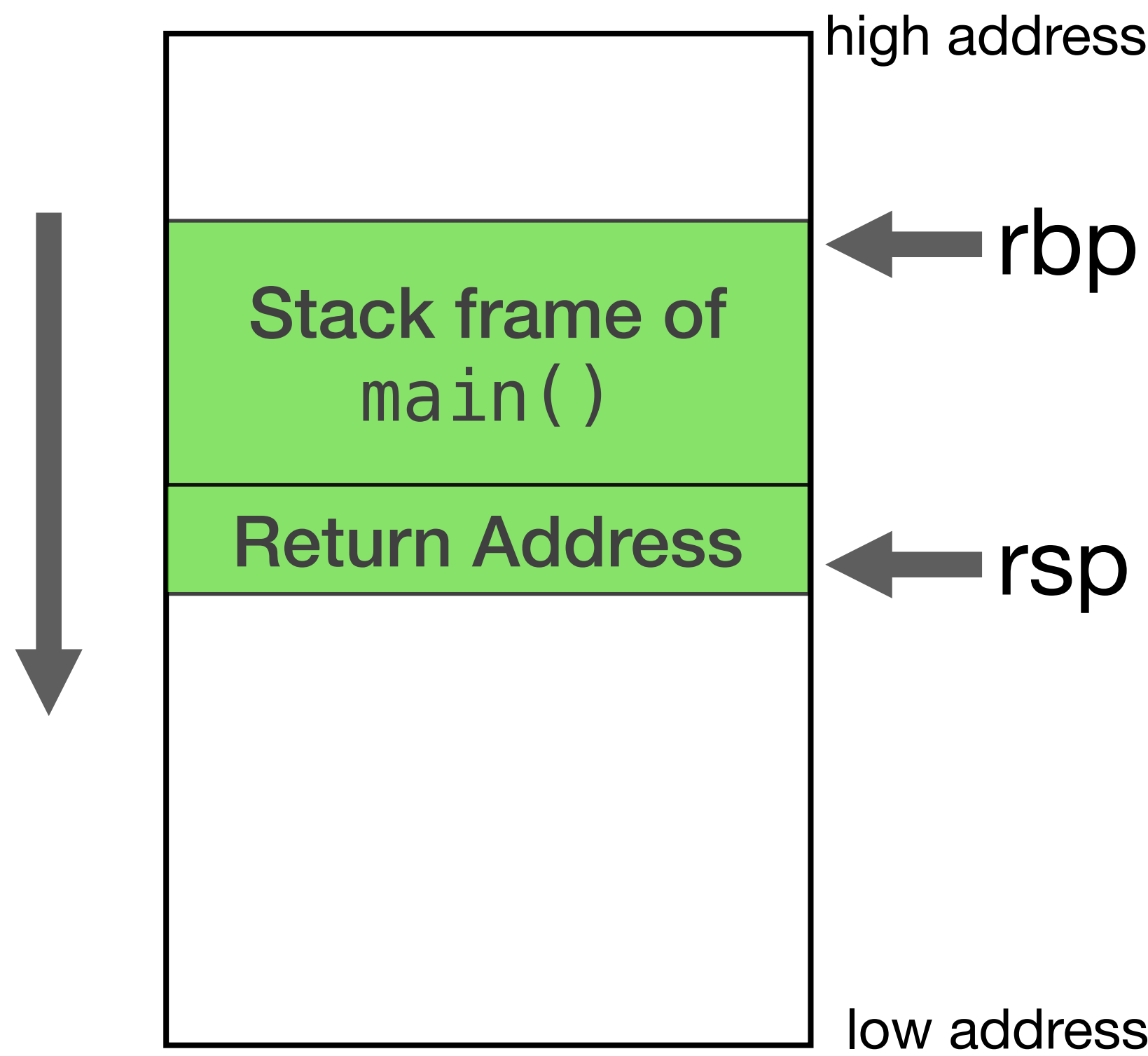
Function Calls: Stack

```
0x00000000000001186 <+22>:  mov    $0x1,%edi
0x0000000000000118b <+27>:  mov    $0x2,%esi
0x00000000000001190 <+32>:  call   0x1150 <foo>
```

Before the call



After the call



Function Initialization

```
void foo(int a, int b) {  
    char buffer[12];  
    gets(buffer);  
    return;  
}
```

Dump of assembler code for function `foo`:

0x0000000000001150	<+0>:	push	%rbp
0x0000000000001151	<+1>:	mov	%rsp,%rbp
0x0000000000001154	<+4>:	sub	\$0x20,%rsp
0x0000000000001158	<+8>:	mov	%edi,-0x4(%rbp)
0x000000000000115b	<+11>:	mov	%esi,-0x8(%rbp)
0x000000000000115e	<+14>:	lea	-0x14(%rbp),%rdi
0x0000000000001162	<+18>:	mov	\$0x0,%al
0x0000000000001164	<+20>:	call	0x1040 <gets@plt>
0x0000000000001169	<+25>:	add	\$0x20,%rsp
0x000000000000116d	<+29>:	pop	%rbp
0x000000000000116e	<+30>:	ret	

Function Initialization

```
void foo(int a, int b) {  
    char buffer[12];  
    gets(buffer);  
    return;  
}
```

0x0000000000000150	<+0>:	push	%rbp
0x0000000000000151	<+1>:	mov	%rsp,%rbp
0x0000000000000154	<+4>:	sub	\$0x20,%rsp

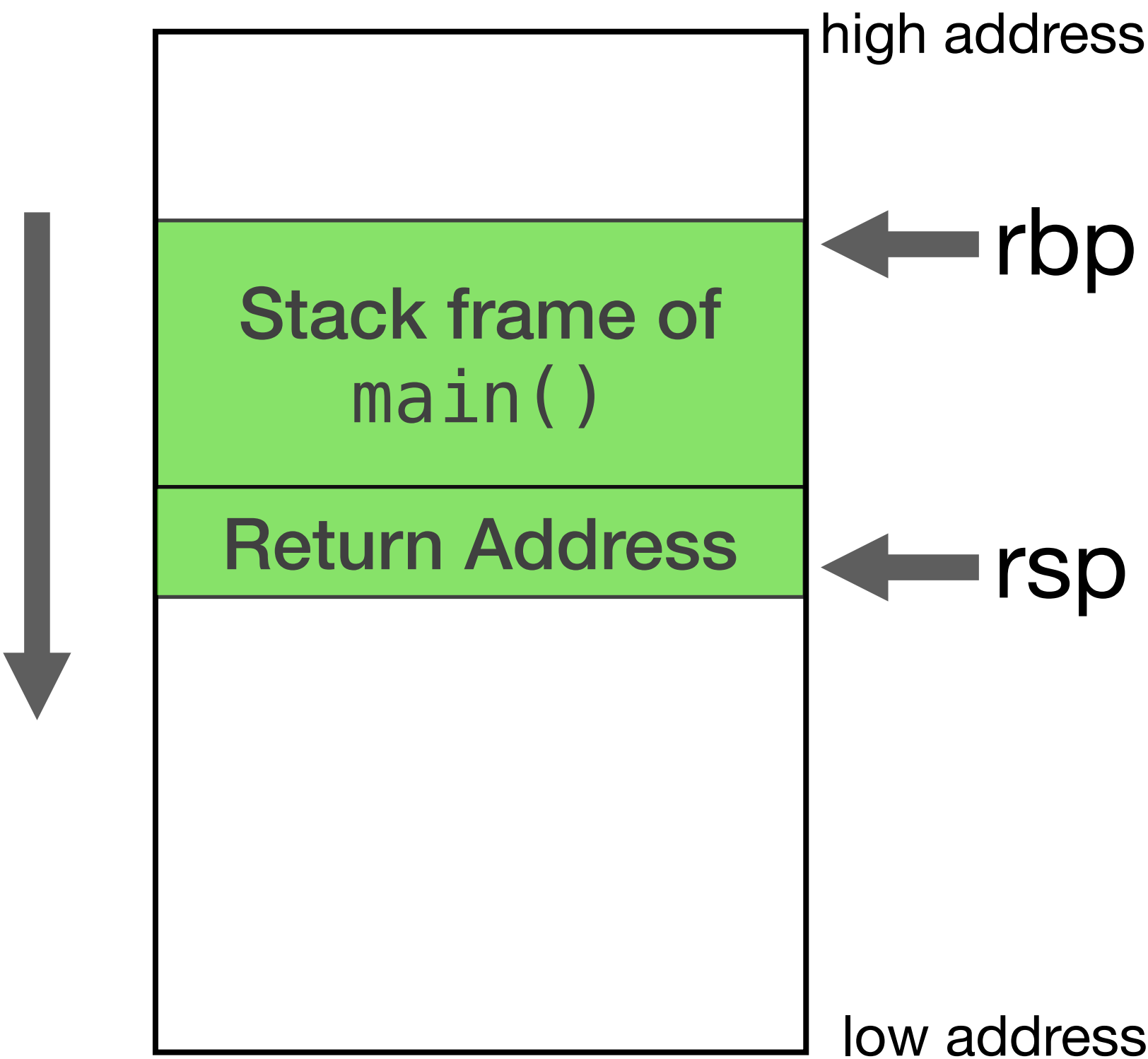
- Save the old frame pointer
- Set the new frame pointer
- Allocate space for local variables

Function Initialization: Stack

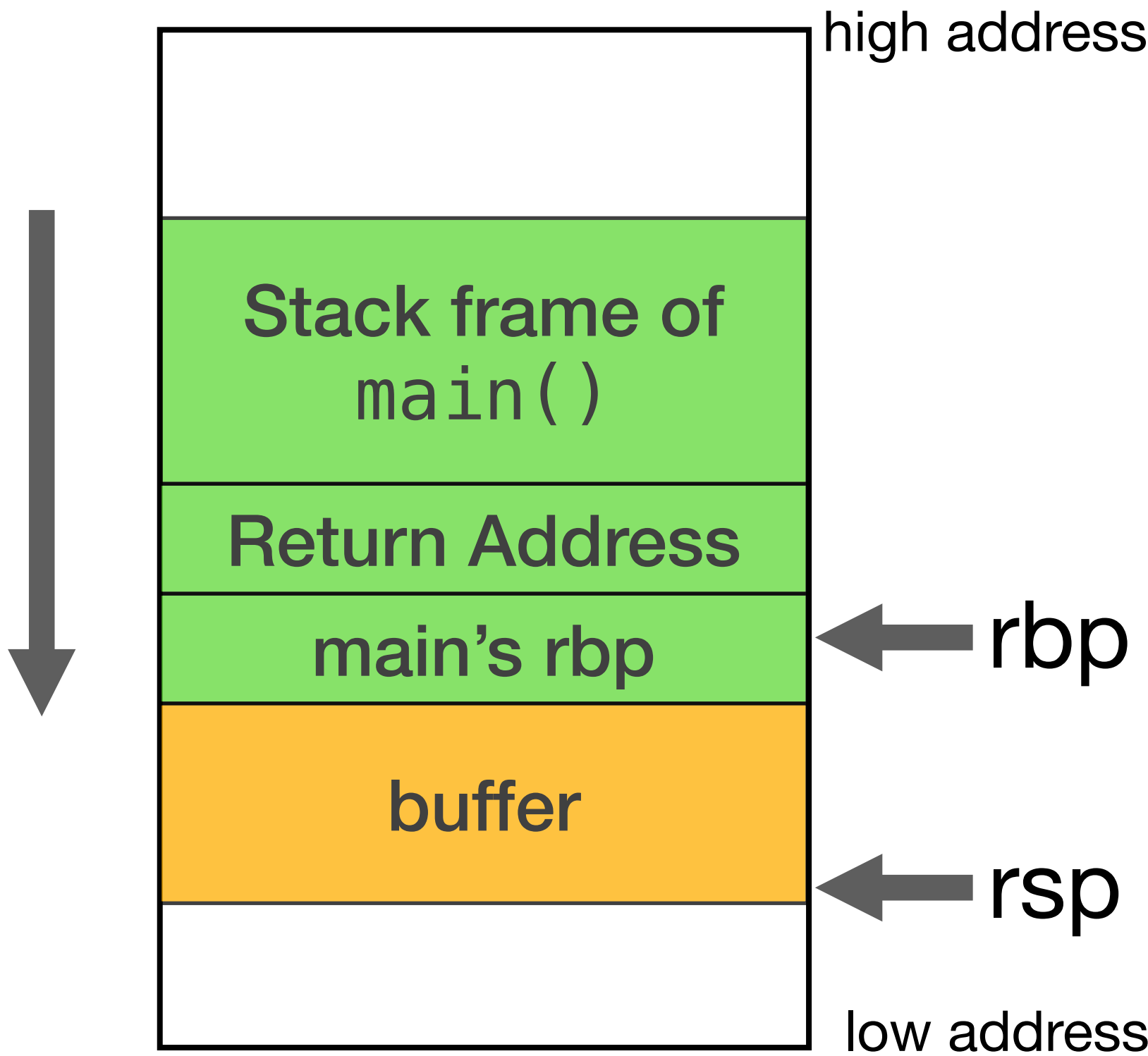
```
void foo(int a, int b) {  
    char buffer[12];  
    gets(buffer);  
    return;  
}
```

0x0000000000000150	<+0>:	push	%rbp
0x0000000000000151	<+1>:	mov	%rsp,%rbp
0x0000000000000154	<+4>:	sub	\$0x20,%rsp

Before the call



After the call



Anything unexpected?

Function Return

```
void foo(int a, int b) {  
    char buffer[12];  
    gets(buffer);  
    return;  
}
```

```
push    %rbp  
mov     %rsp,%rbp  
sub     $0x20,%rsp
```

Function Initialization

```
0x00000000000001169 <+25>:    add     $0x20,%rsp  
0x0000000000000116d <+29>:    pop     %rbp  
0x0000000000000116e <+30>:    ret
```

- Deallocate the space for local data
- Restore the old frame pointer
- Get the return address and jump to it

Function Return: Using the `leave` Instruction

```
void foo(int a, int b) {  
    char buffer[12];  
    gets(buffer);  
    return;  
}
```

```
push    %rbp  
mov     %rsp,%rbp  
sub     $0x20,%rsp
```

Function Initialization

```
0x00000000000001d0 <+71>:    leave  
0x00000000000001d1 <+72>:    ret
```

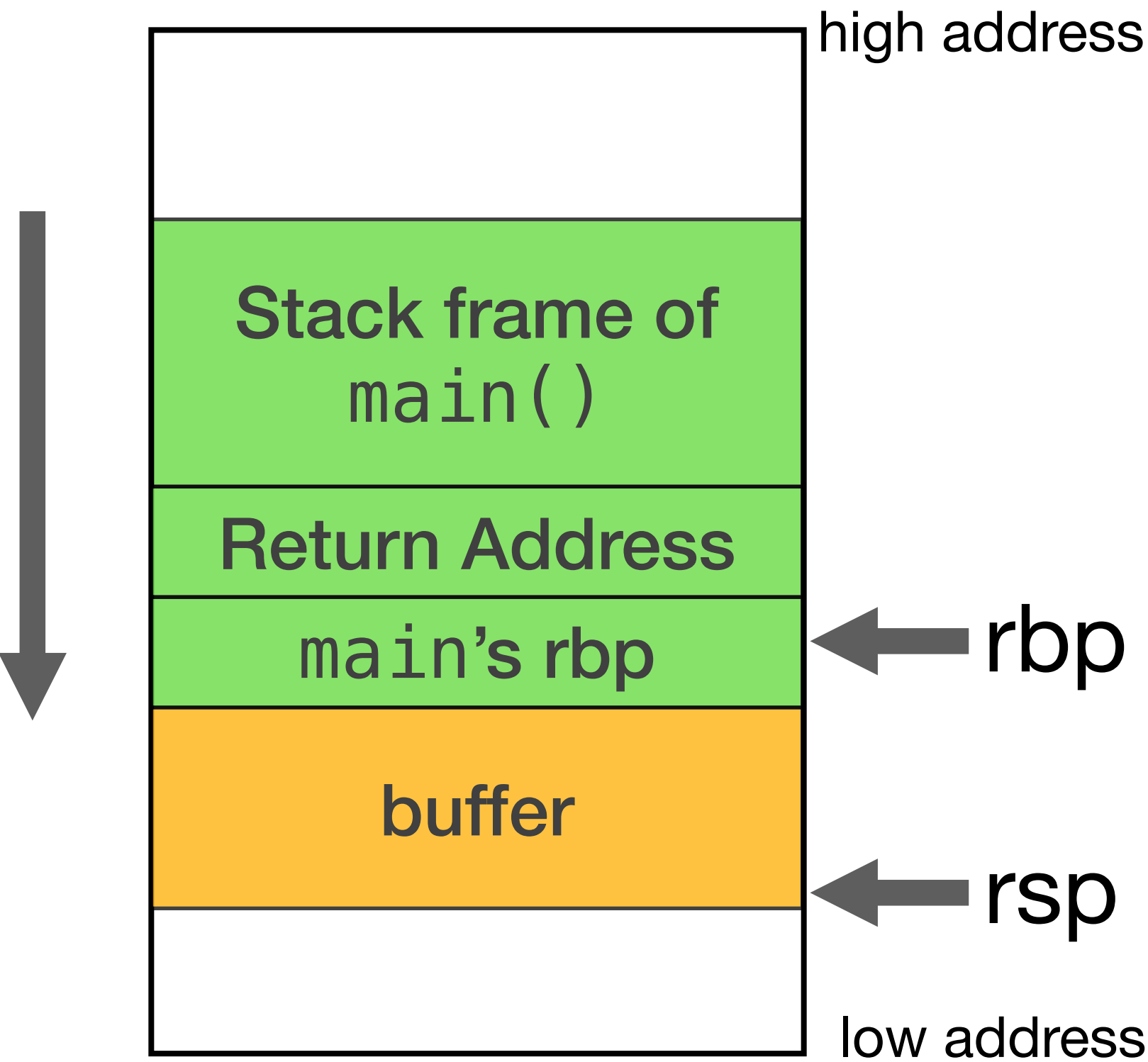
- `leave`: Shorthand for two instructions
 - ▶ `mov %rbp, %rsp`
 - ▶ `pop %rbp`
- Deallocate the space for local data
- Restore the old frame pointer
- Get the return address and jump to it

Function Return: Stack

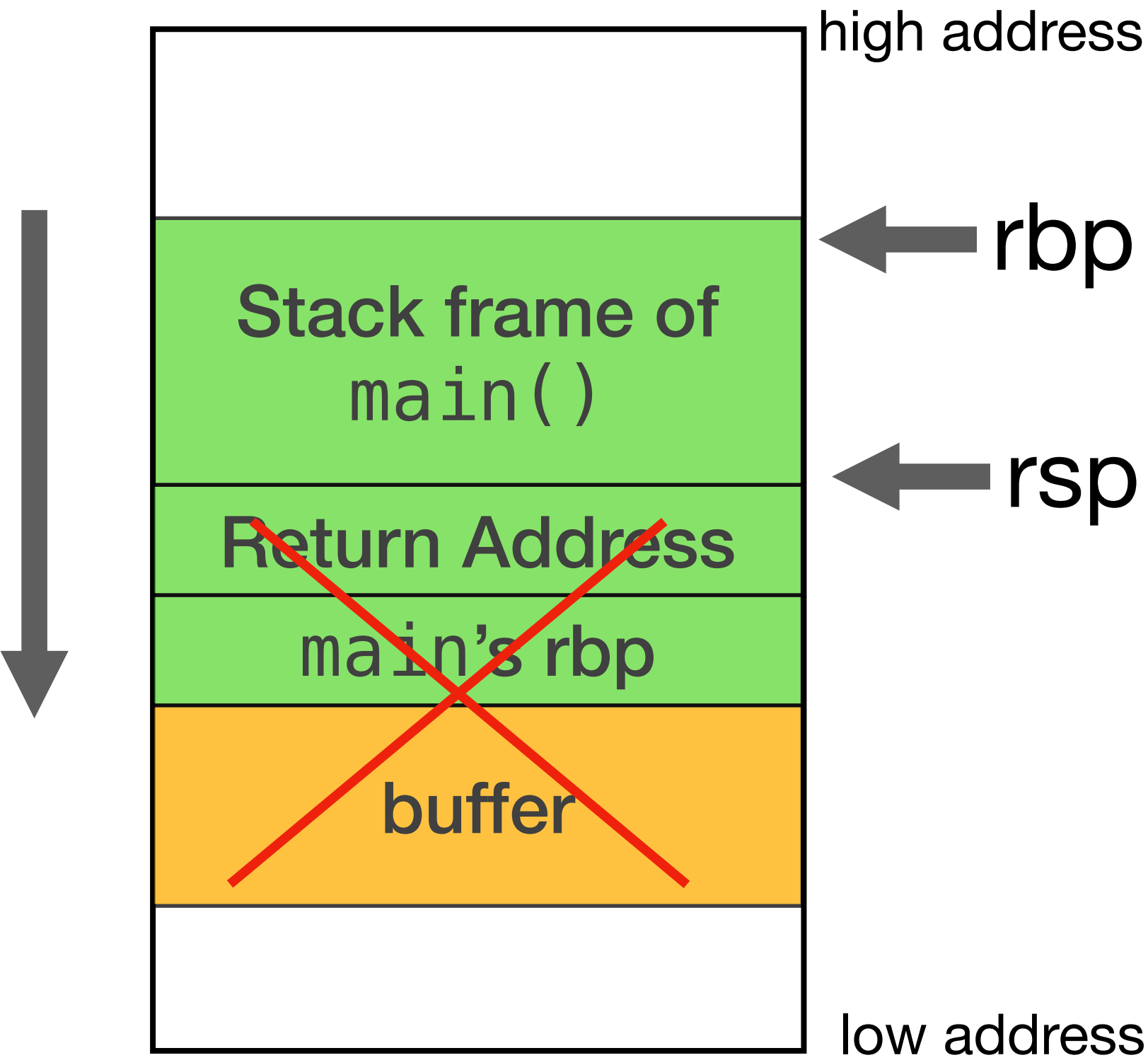
```
void foo(int a, int b) {  
    char buffer[12];  
    gets(buffer);  
    return;  
}
```

```
0x0000000000000169 <+25>:  add    $0x20,%rsp  
0x000000000000016d <+29>:  pop     %rbp  
0x000000000000016e <+30>:  ret
```

Before the call



After the call



What could attackers do?

Definition: Threat Model



The abilities and resources of the attacker

- Threat models enable structured reasoning about the attack surface.
- Awareness of entry points (and associated threats) to break into the target.
- Look at systems from an attacker's perspective:
 - Decompose application: **identify structure**
 - Determine and rank threats
 - Determine countermeasures and mitigations

Further reading:

https://owasp.org/www-community/Threat_Modeling

Definition: Threat Model



The abilities and resources of the attacker

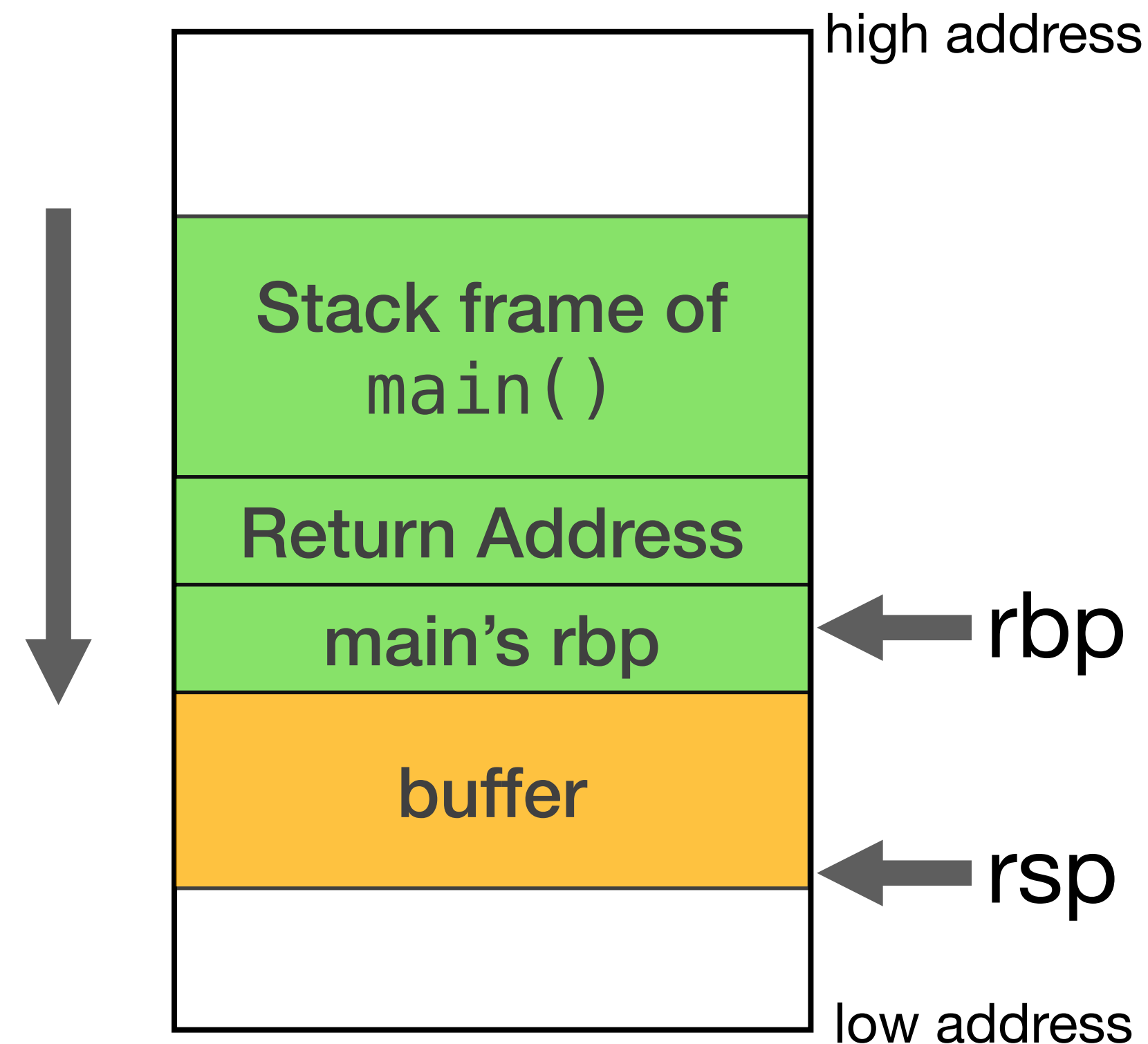
- Threat models enable structured reasoning about the attack surface.
- Awareness of **entry points** (and associated threats) to break into the target.
- Look at systems from an attacker's perspective:
 - Decompose application: **identify structure**
 - Determine and rank threats
 - Determine countermeasures and mitigations

Further reading:

https://owasp.org/www-community/Threat_Modeling

Exploiting Buffer Overflows

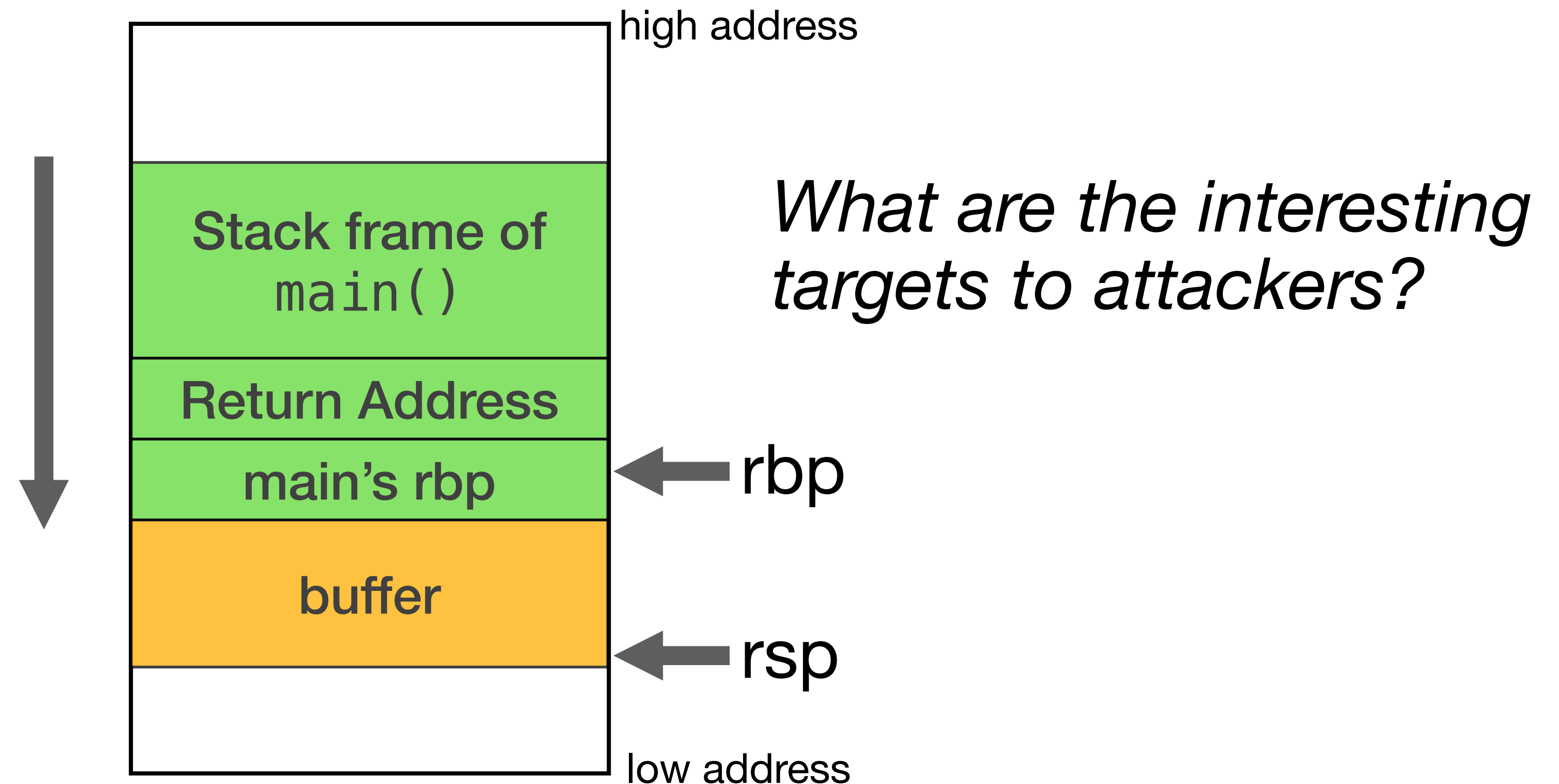
```
void foo(int a, int b) {  
    char buffer[12];  
    gets(buffer);  
    return;  
}  
  
int main() {  
    int x;  
    x = 0;  
    foo(1,2);  
    x = 1;  
    printf("%d\n",x);  
    return 0;  
}
```



If attackers can control the input, then they can overflow the buffer and overwrite the stack!

Exploiting Buffer Overflows

```
void foo(int a, int b) {  
    char buffer[12];  
    gets(buffer);  
    return;  
}  
  
int main() {  
    int x;  
    x = 0;  
    foo(1,2);  
    x = 1;  
    printf("%d\n",x);  
    return 0;  
}
```



If attackers can control the input, then they can overflow the buffer and overwrite the stack!

Smashing the Stack

- Occurs when a buffer overflow overwrites data in the program stack.
- Successful exploits can overwrite the return address on the stack.
 - Could lead to arbitrary code execution on the target machine

Smashing the Stack

```
void foo(int a, int b) {
    char buffer[12];
    gets(buffer);
    return;
}

int main() {
    int x;
    x = 0;
    foo(1, 2);
    x = 1;
    printf("%d\n", x);
    return 0;
}
```

What happens if we input a large string?

```
./demo ffffffffffffffffffffffffff...fffff
```

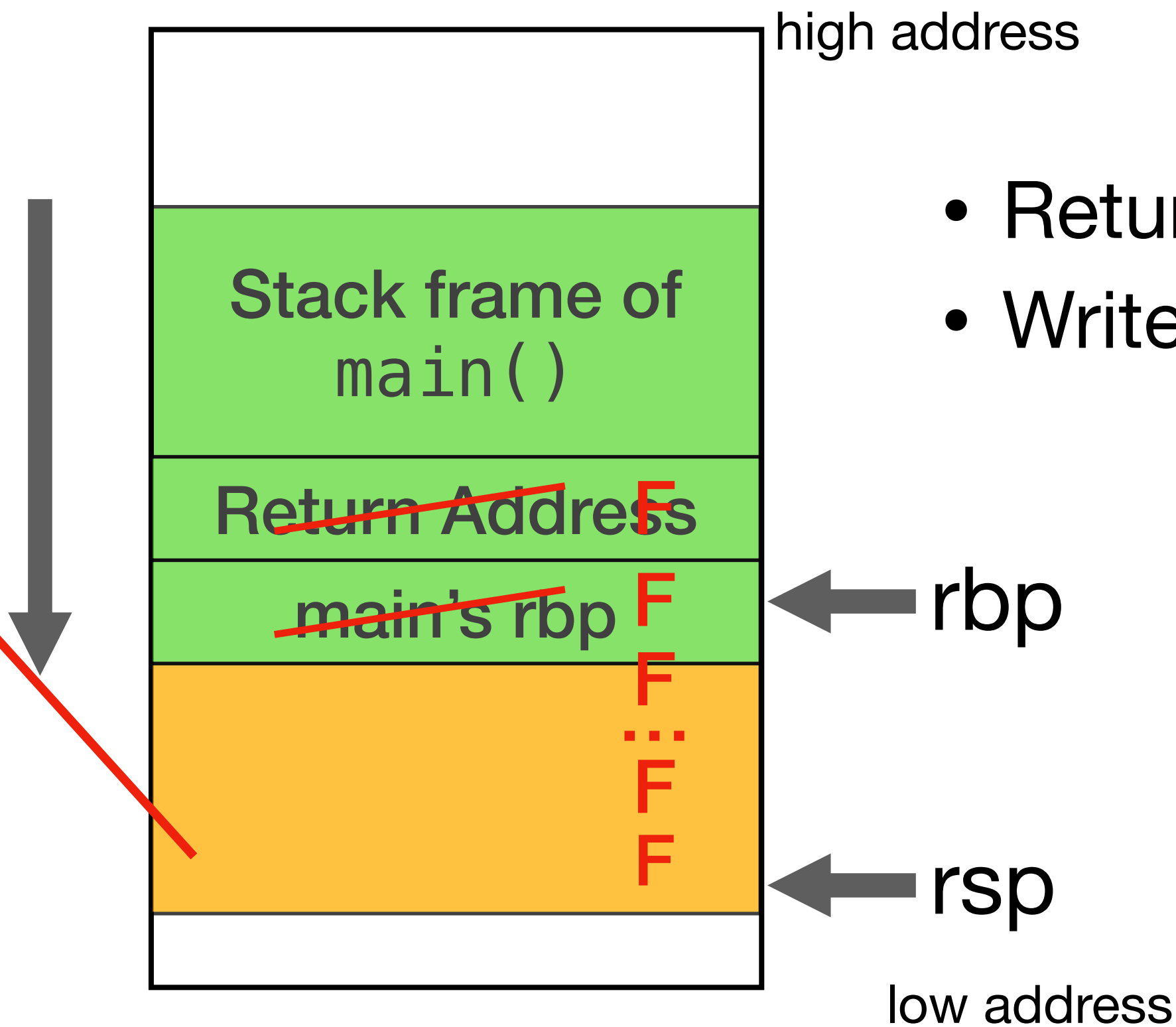
Segmentation fault

Smashing the Stack: What Happened?

```
void foo(int a, int b) {  
    char buffer[12];  
    gets(buffer);  
    return;  
}
```

What happens if we input a large string?

./demo ffffffffffffffffffffffffff...ffffff



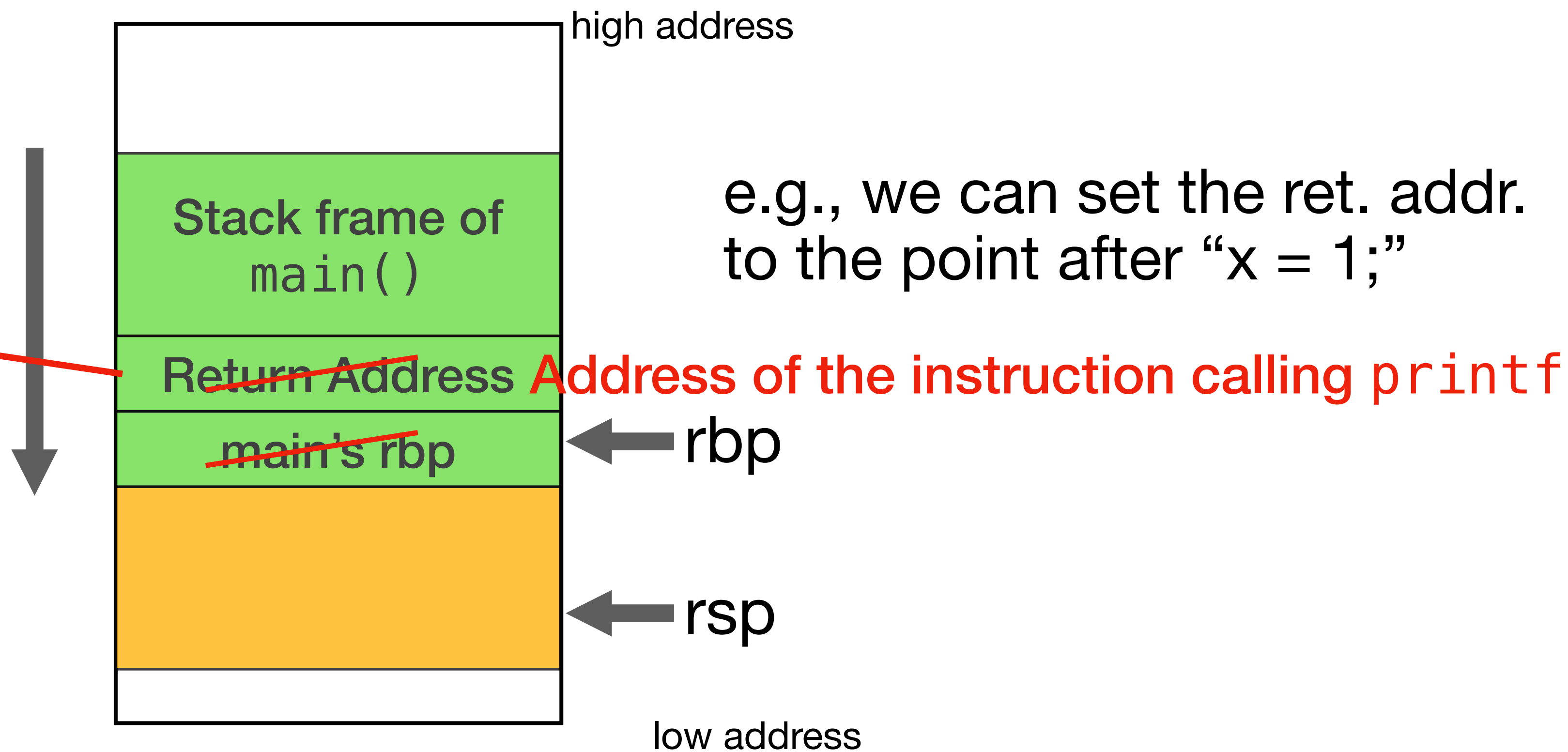
- Return to an invalid address
- Write to an unwritable address

Smashing the Stack: Figure out a Nasty Input

./demo ? What to input?

```
void foo(int a, int b) {
    char buffer[12];
    gets(buffer);
    return;
}

int main() {
    int x;
    x = 0;
    foo(1,2);
    x = 1;
    printf("%d\n", x);
    return 0;
}
```



Definition: Software Security



Allow *intended* use of software and prevent *unintended* use that may cause harm

Goal: Prevent information “mishaps”, but don’t stop good things from happening

- Good things include functionality or legal information access.
- Tradeoff between functionality and security is the key.



E-Voting

Good things: convenience of voting; fast tallying; voting for the disabled; ...

The convenience comes with risks

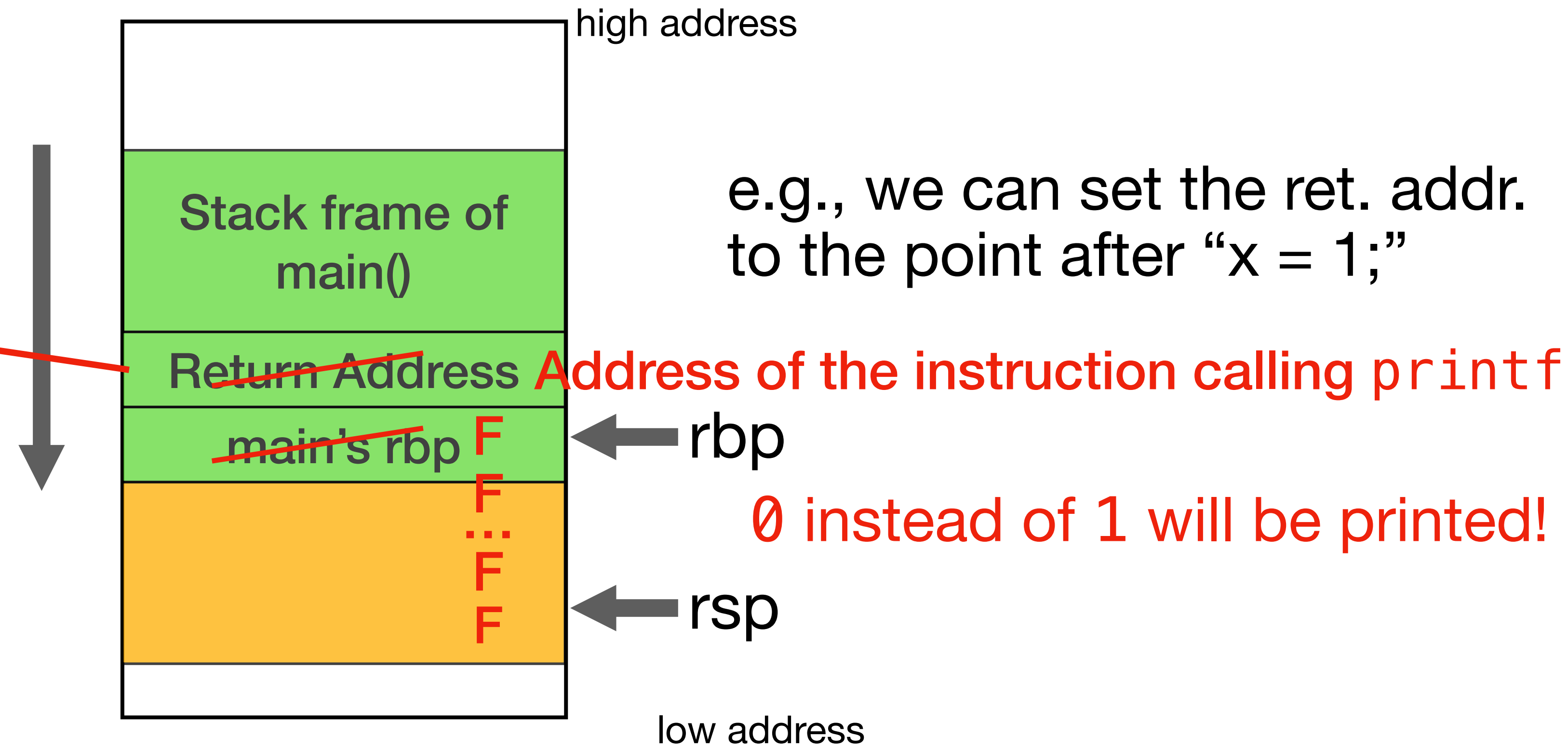
- Buggy voting software/hardware
- Changed e-voting software by insiders
- ...

Smashing the Stack: Figure out a Nasty Input

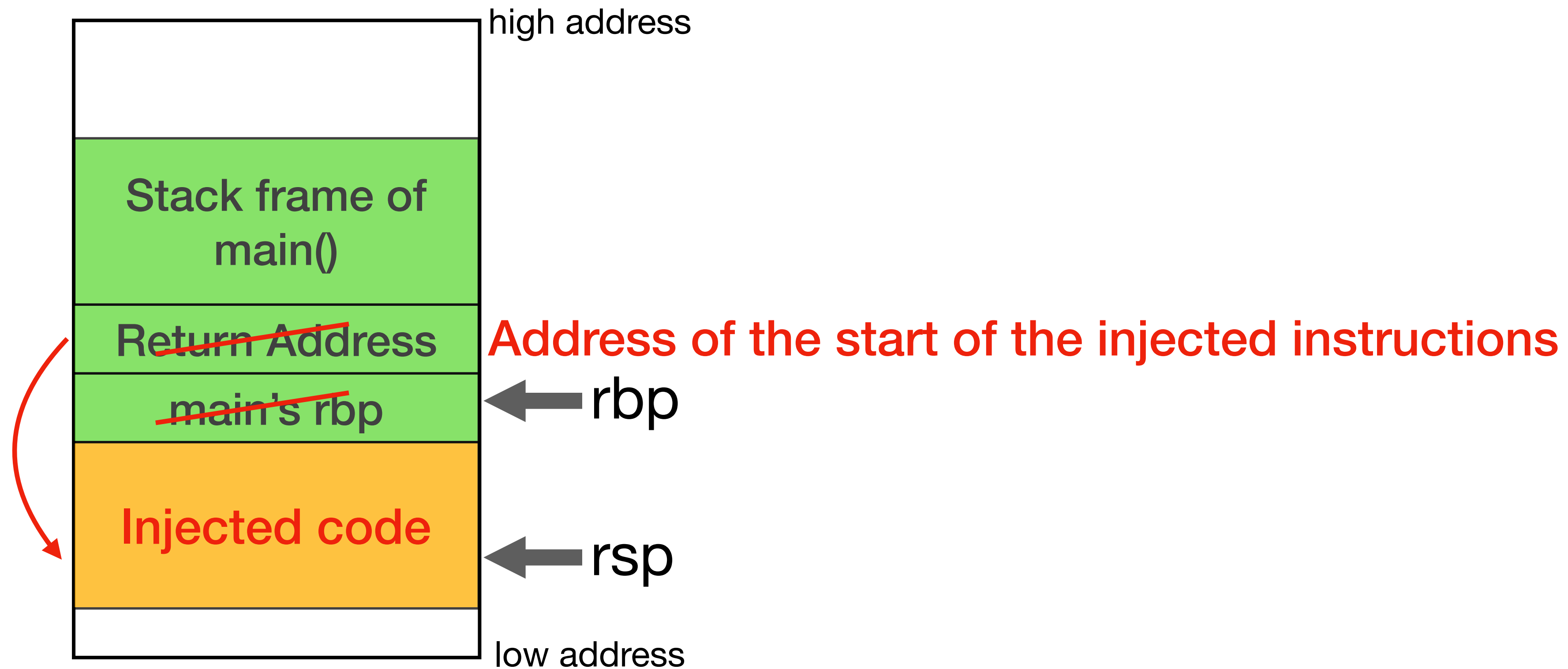
```
void foo(int a, int b) {
    char buffer[12];
    gets(buffer);
    return;
}

int main() {
    int x;
    x = 0;
    foo(1,2);
    x = 1;
    printf("%d\n", x);
    return 0;
}
```

./demo ? What to input?



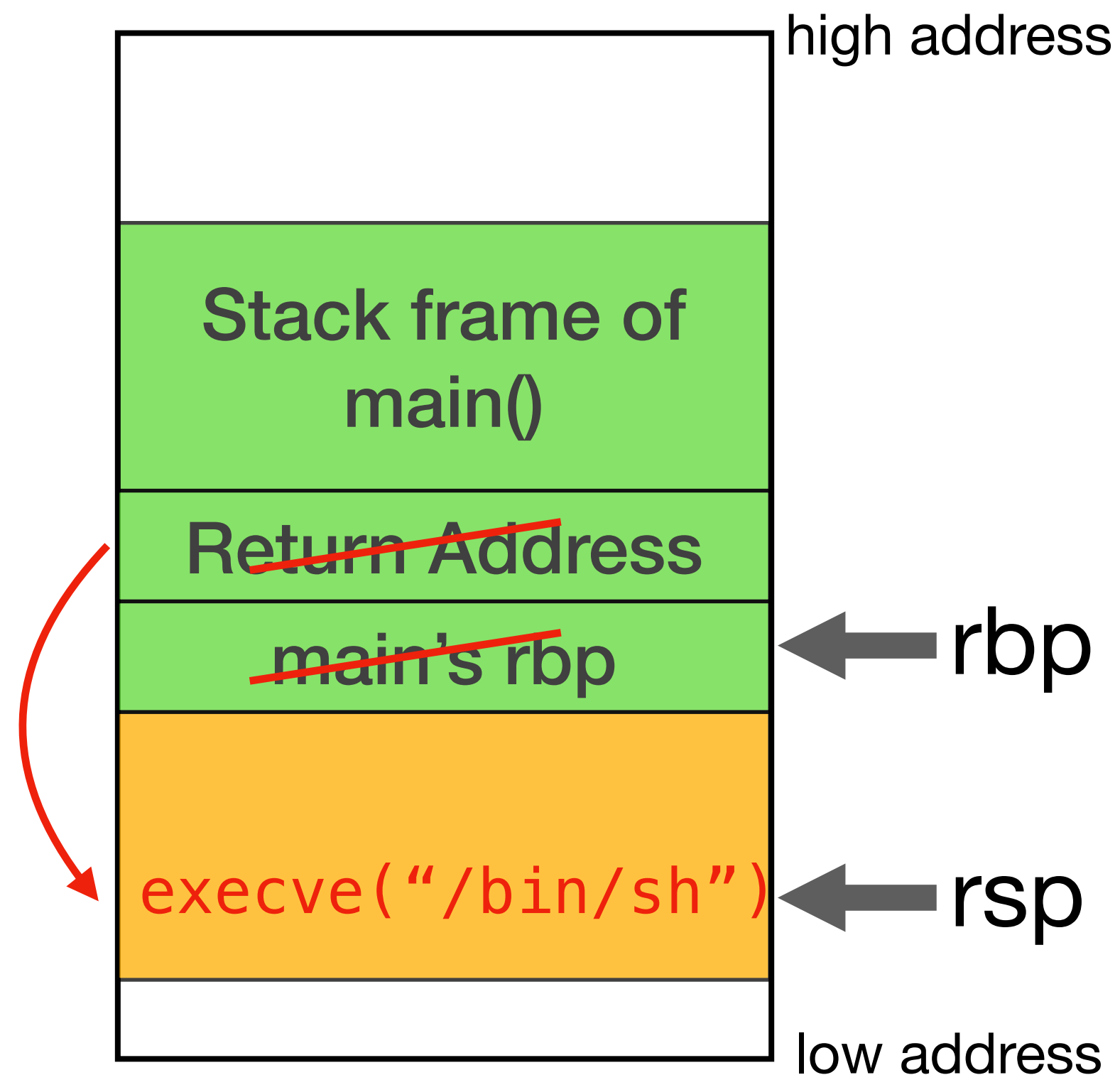
Smashing the Stack: Code Injection



Code Injection

- Attacker creates a malicious input—a specially crafted input that contains a pointer to malicious code included in the input.
- When the function returns, control is transferred to the malicious code.
 - Injected code runs with the permission of the vulnerable program when the function returns.
 - Programs running with root or other elevated privileges are normally targeted.

Smashing the Stack: Injecting Shell Code



- This brings up a shell.
- Attackers can execute *any* command in the shell.
- The shell has the same privilege as the process.



- Good news:

- C/C++ stack is not executable by default.

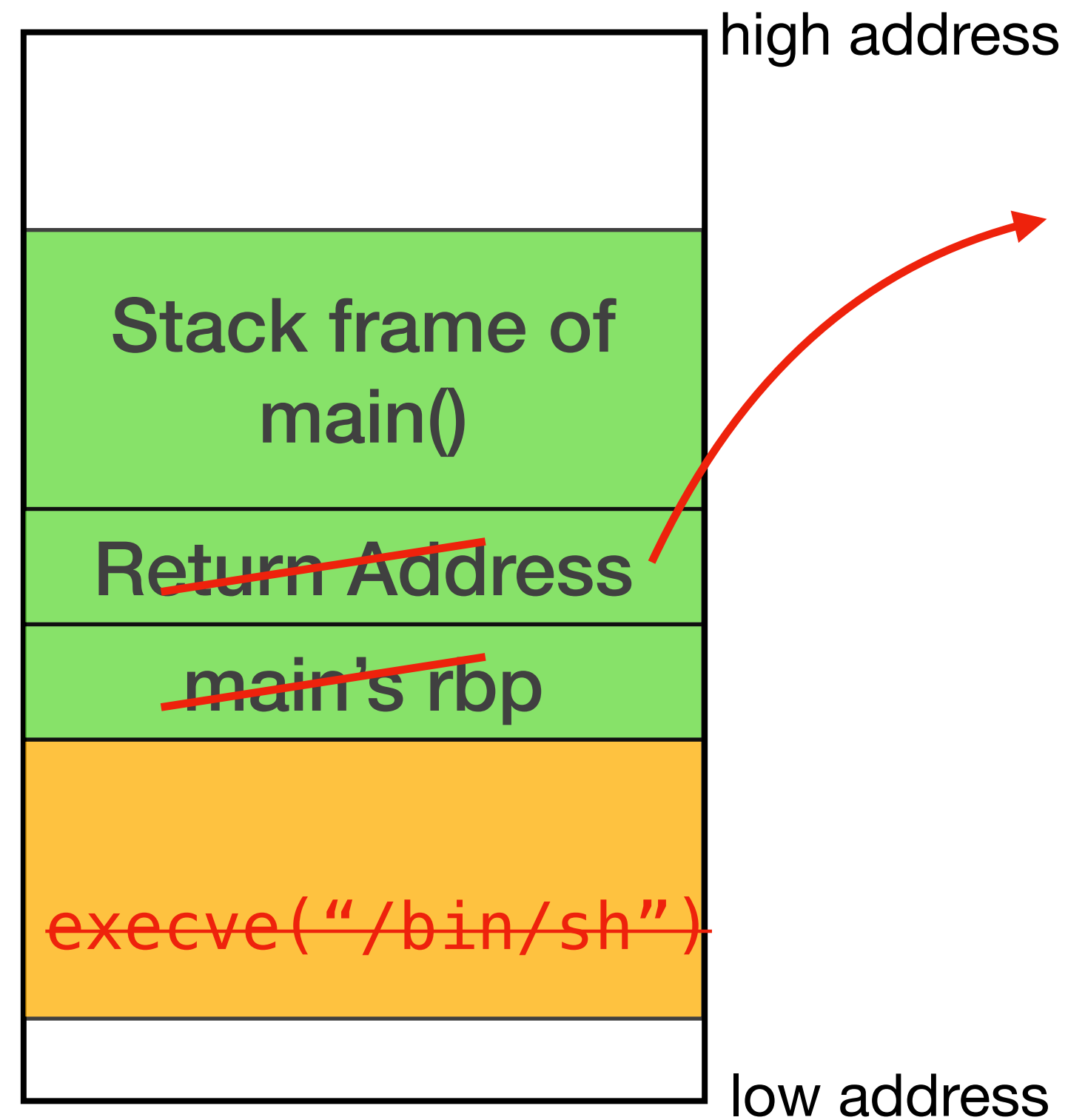


- Bad news:

- Code injection works in other cases, e.g. JIT, certain embedded systems, etc.

**How to circumvent this
non-executable-stack restriction?**

Exploiting Existing and Executable Code



How about "returning" to some existing code?

Hello World Program

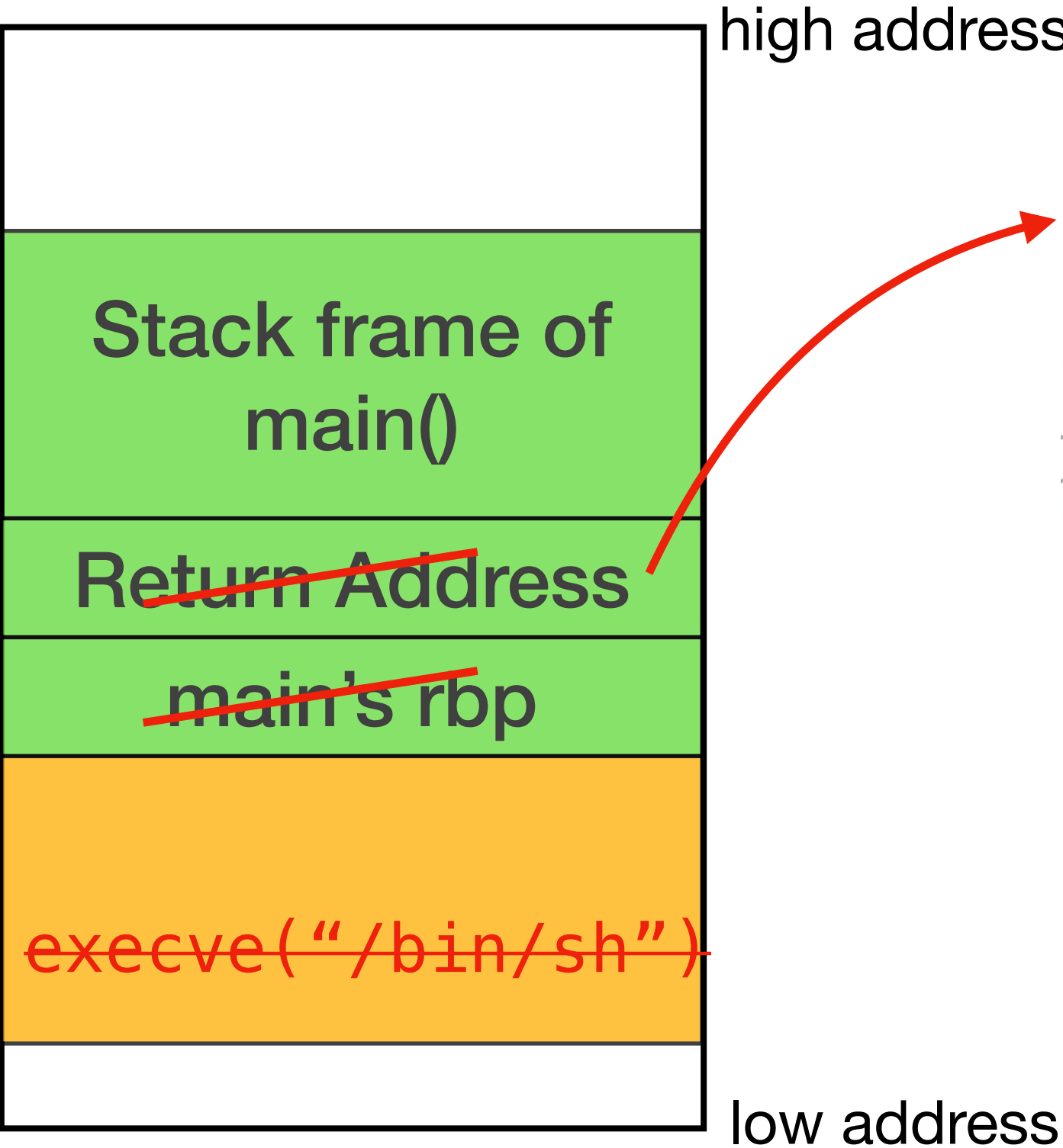
```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     printf("Hello, world!\n");
5
6     return 0;
7 }
```

hello.c

```
$ gcc hello.c -o hello
$ ./hello
```

- Size of this .c file: 98 bytes
 - Source line of code: 7
-
- Size of the hello binary: 17 KB
 - Instructions executed: 657,679

Exploiting Existing and Executable Code



How about “returning” to some existing code?

```
jie@gwsyssec: ~/courses/csci6545/lectures
$ ldd demo
linux-vdso.so.1 (0x00007ffffadfd000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f48a2c00000)
/lib64/ld-linux-x86-64.so.2 (0x00007f48a2efc000)
```

[(gdb) info proc mappings
process 74581
Mapped address spaces:

Start Addr	End Addr	Size	Offset	Perms	objfile
0x555555554000	0x555555555000	0x1000	0x0	r--p	/home/jie/courses/csci6545/lectures/demo
0x555555555000	0x555555556000	0x1000	0x1000	r-xp	/home/jie/courses/csci6545/lectures/demo
0x555555556000	0x555555557000	0x1000	0x2000	r--p	/home/jie/courses/csci6545/lectures/demo
0x555555557000	0x555555558000	0x1000	0x2000	r--p	/home/jie/courses/csci6545/lectures/demo
0x555555558000	0x555555559000	0x1000	0x3000	rw-p	/home/jie/courses/csci6545/lectures/demo
0x7ffff7c00000	0x7ffff7c28000	0x28000	0x0	r--p	/usr/lib/x86_64-linux-gnu/libc.so.6
0x7ffff7c28000	0x7ffff7dbd000	0x195000	0x28000	r-xp	/usr/lib/x86_64-linux-gnu/libc.so.6
0x7ffff7dbd000	0x7ffff7e15000	0x58000	0x1bd000	r--p	/usr/lib/x86_64-linux-gnu/libc.so.6
0x7ffff7e15000	0x7ffff7e16000	0x1000	0x215000	---	/usr/lib/x86_64-linux-gnu/libc.so.6
0x7ffff7e16000	0x7ffff7e1a000	0x4000	0x215000	r--p	/usr/lib/x86_64-linux-gnu/libc.so.6
0x7ffff7e1a000	0x7ffff7e1c000	0x2000	0x219000	rw-p	/usr/lib/x86_64-linux-gnu/libc.so.6
0x7ffff7e1c000	0x7ffff7e29000	0xd000	0x0	rw-p	
0x7ffff7fa6000	0x7ffff7fa9000	0x3000	0x0	rw-p	
0x7ffff7fbb000	0x7ffff7fbd000	0x2000	0x0	rw-p	
0x7ffff7fbd000	0x7ffff7fc1000	0x4000	0x0	r--p	[vvar]
0x7ffff7fc1000	0x7ffff7fc3000	0x2000	0x0	r-xp	[vdso]
0x7ffff7fc3000	0x7ffff7fc5000	0x2000	0x0	r--p	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x7ffff7fc5000	0x7ffff7fef000	0x2a000	0x2000	r-xp	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x7ffff7fef000	0x7ffff7ffa000	0xb000	0x2c000	r--p	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x7ffff7ffb000	0x7ffff7ffd000	0x2000	0x37000	r--p	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x7ffff7ffd000	0x7ffff7fff000	0x2000	0x39000	rw-p	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x7ffff7ffde000	0x7ffff7fff000	0x21000	0x0	rw-p	[stack]
0xffffffff600000	0xffffffff601000	0x1000	0x0	---	[vsyscall]

Exploiting Existing and Executable Code

- `system()` libc function

NAME [top](#)

`system` – execute a shell command

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdlib.h>
```

```
int system(const char *command);
```

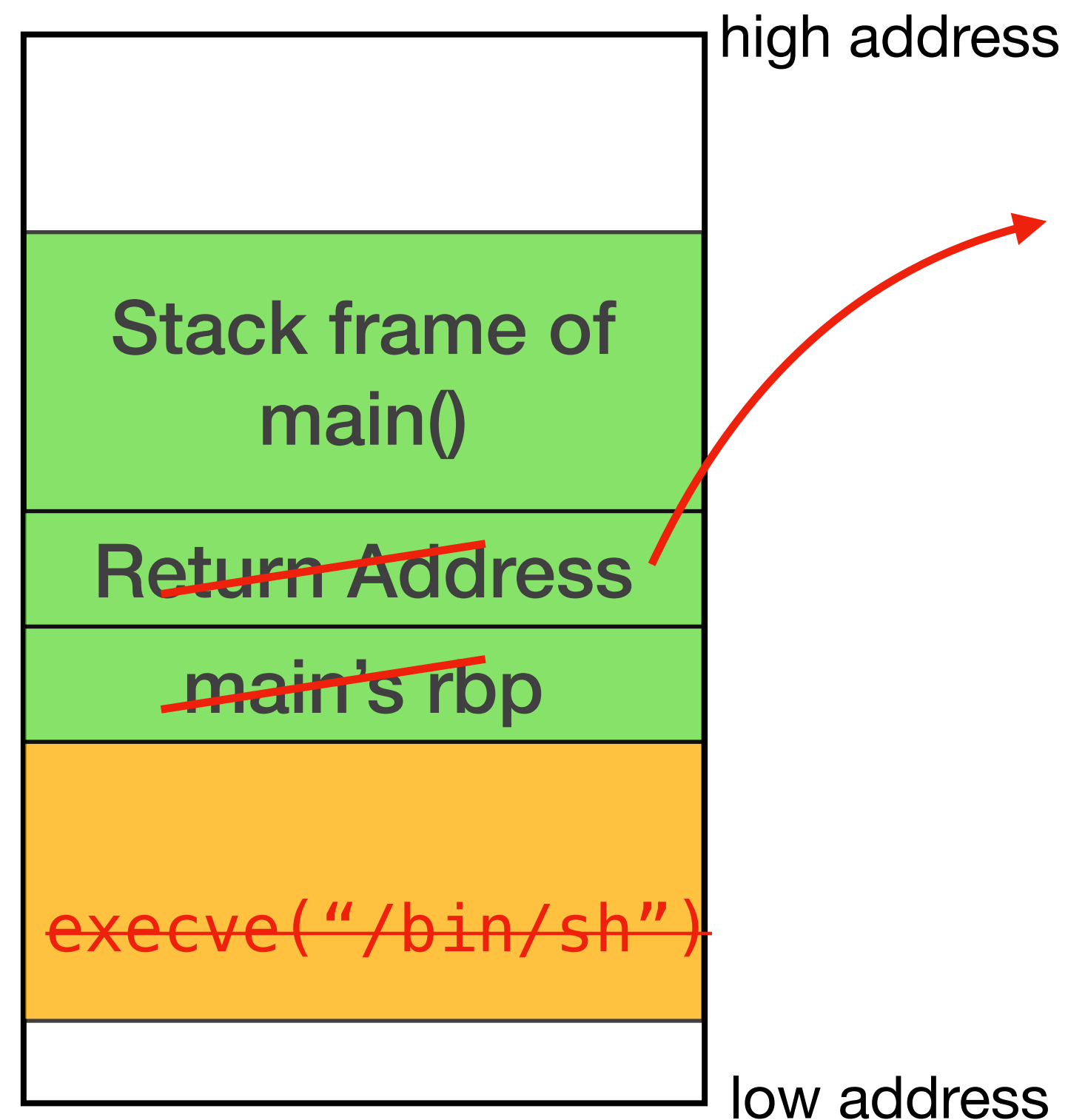
DESCRIPTION [top](#)

The **system()** library function behaves as if it used `fork(2)` to create a child process that executed the shell command specified in *command* using `exec(3)` as follows:

```
execl("/bin/sh", "sh", "-c", command, (char *) NULL);
```

system() returns after the command has been completed.

Exploiting Existing and Executable Code



“return” to system()

(gdb) disassemble system

Dump of assembler code for function system:

```
0x00007ffff7c50d70 <+0>:    endbr64
0x00007ffff7c50d74 <+4>:    test    %rdi,%rdi
0x00007ffff7c50d77 <+7>:    je      0x7ffff7c50d80 <system+16>
0x00007ffff7c50d79 <+9>:    jmp     0x7ffff7c50900
0x00007ffff7c50d7e <+14>:   xchg    %ax,%ax
0x00007ffff7c50d80 <+16>:   sub     $0x8,%rsp
0x00007ffff7c50d84 <+20>:   lea     0x1878f5(%rip),%rdi
0x00007ffff7c50d8b <+27>:   call    0x7ffff7c50900
0x00007ffff7c50d90 <+32>:   test    %eax,%eax
0x00007ffff7c50d92 <+34>:   sete    %al
0x00007ffff7c50d95 <+37>:   add     $0x8,%rsp
0x00007ffff7c50d99 <+41>:   movzbl  %al,%eax
0x00007ffff7c50d9c <+44>:   ret
```

Return-to-libc(ret2libc) Attack: Exploiting system()

- system() libc function

NAME [top](#)

system – execute a shell command

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS [top](#)

```
#include <stdlib.h>
```

```
int system(const char *command);
```

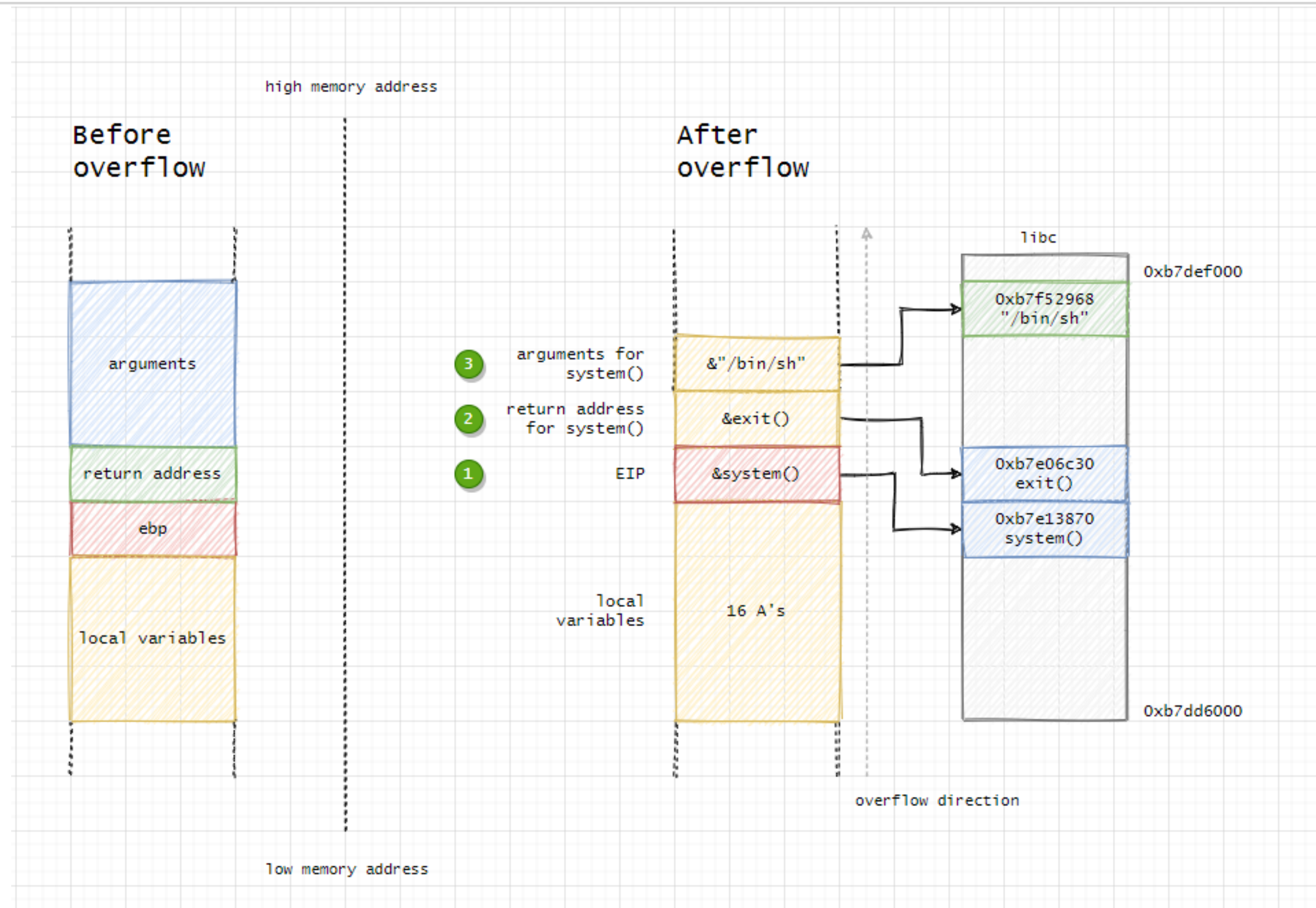
DESCRIPTION [top](#)

The **system()** library function behaves as if it used [fork\(2\)](#) to create a child process that executed the shell command specified in *command* using [execl\(3\)](#) as follows:

```
execl("/bin/sh", "sh", "-c", command, (char *) NULL);
```

system() returns after the command has been completed.

Exploiting ret2libc on x86-32



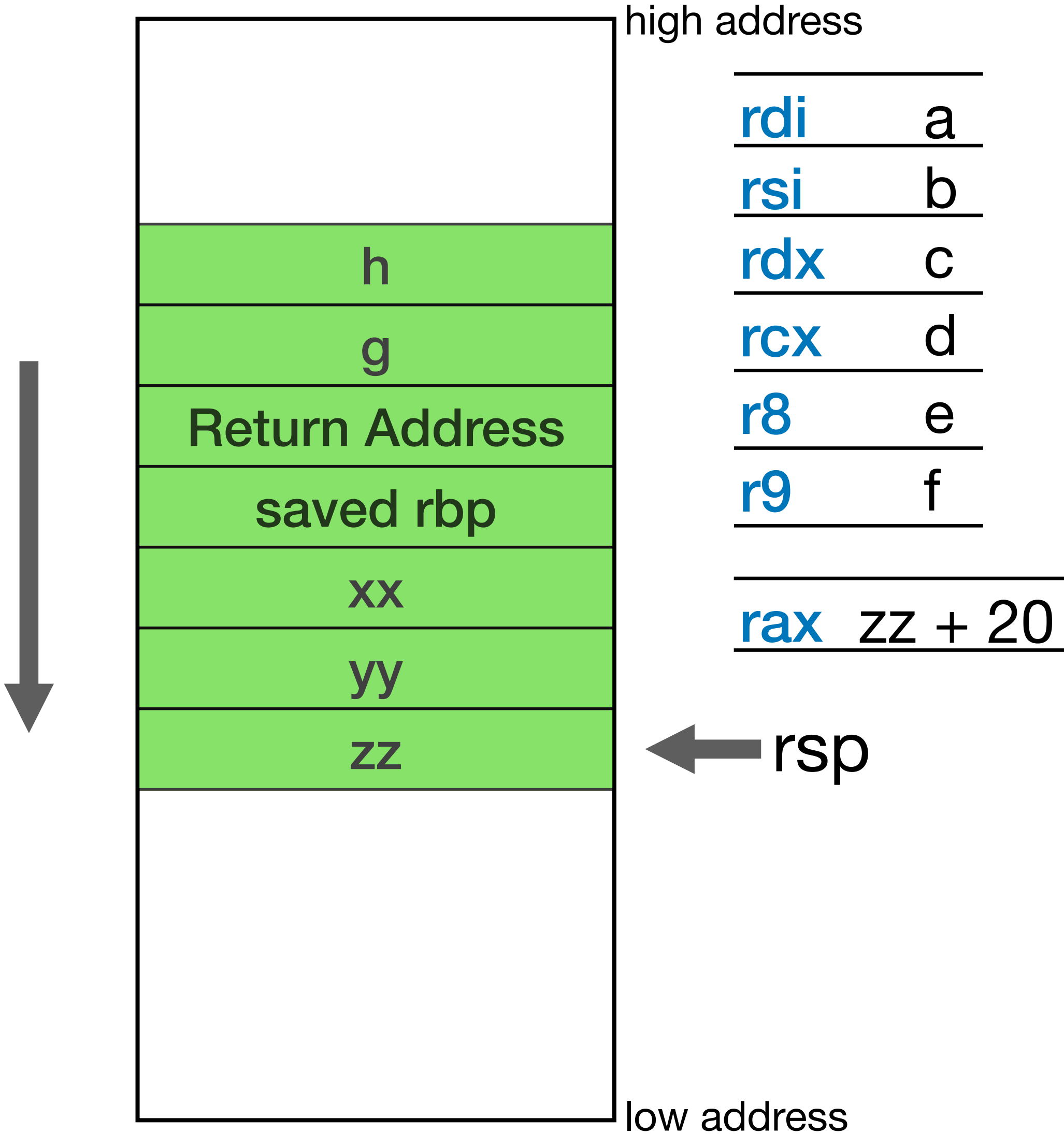
Stack memory layout of a 32-bit vulnerable program

<https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/return-to-libc-ret2libc>

System V ADM64 Calling Convention

```
void foo() {
    ...
    bar(a, b, c, d, e, f, g, h);
    ...
}

long bar(long a, long b, long c, long d,
        long e, long f, long g, long h) {
    long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = utilfunc(xx, yy, xx % yy);
    return zz + 20;
}
```



**How to put malicious data
in target registers?**

Come to the Next Lecture!

GDB (GNU Debugger)

- 
- Debugger: A program that debuggs (examines) other programs.
- See what status a running/crashed program is in.
 - Inspect virtual addresses and registers

GDB basics:

<https://medium.com/@amit.kulkarni/gdb-basics-bf3407593285>

GDB (GNU Debugger)

- Examine code (source and assembly)
- Control execution
 - Break point (where to stop)
 - Next line/instruction/next function/break point
- Examine memory/register
 - Variable's value
 - Value in register
 - Value in a virtual address
- Powerful commands/techniques
 - `info`
 - `define hook-stop`
 - `help + command name`

Other Tools for Studying Binaries

- `objdump`
- `strings`
- `readelf`
- `nm`
- `hexdump`
- `ldd`