CSCI 476: Computer Security

Buffer Overflow Attack (Part 1)

The stack, stack frames, function prologue and epilogue

Reese Pearsall

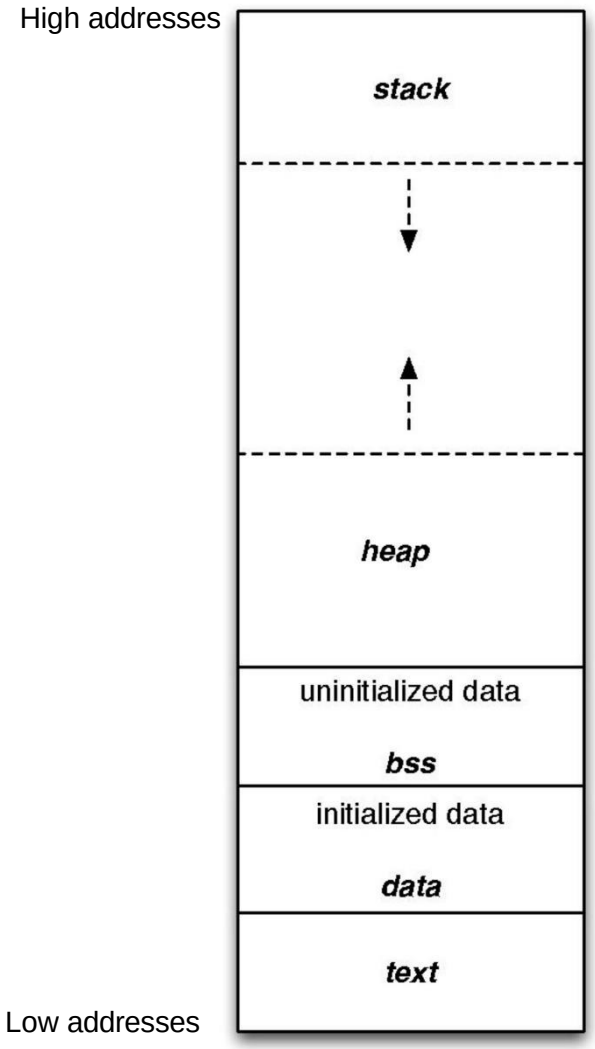
Fall 2024

Announcements

Lab 2 (Shellshock) posted

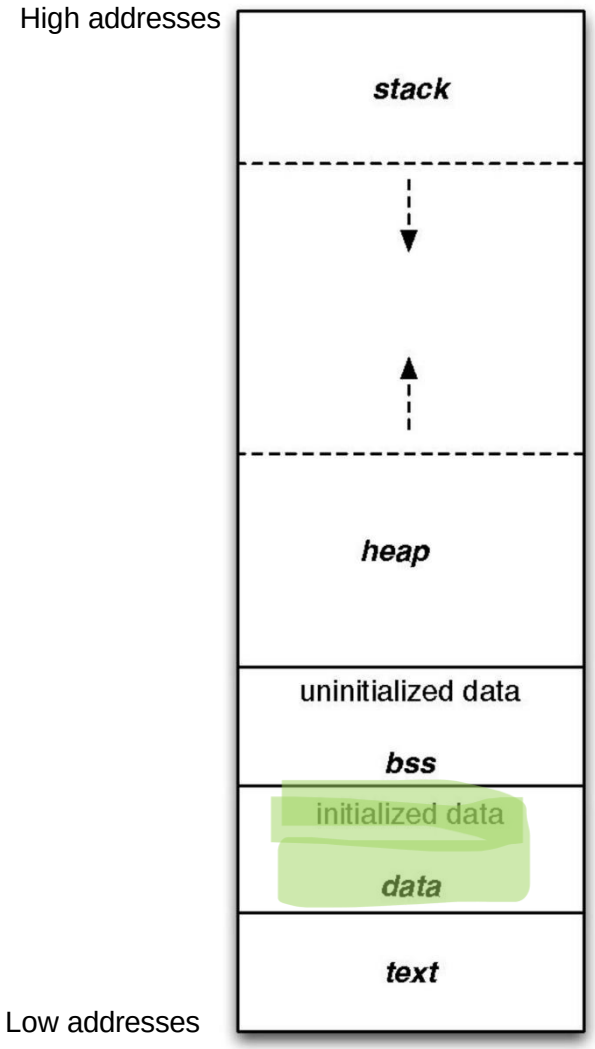
- Still ironing out some issues for the Apple M1/M2 people

Program layout in memory



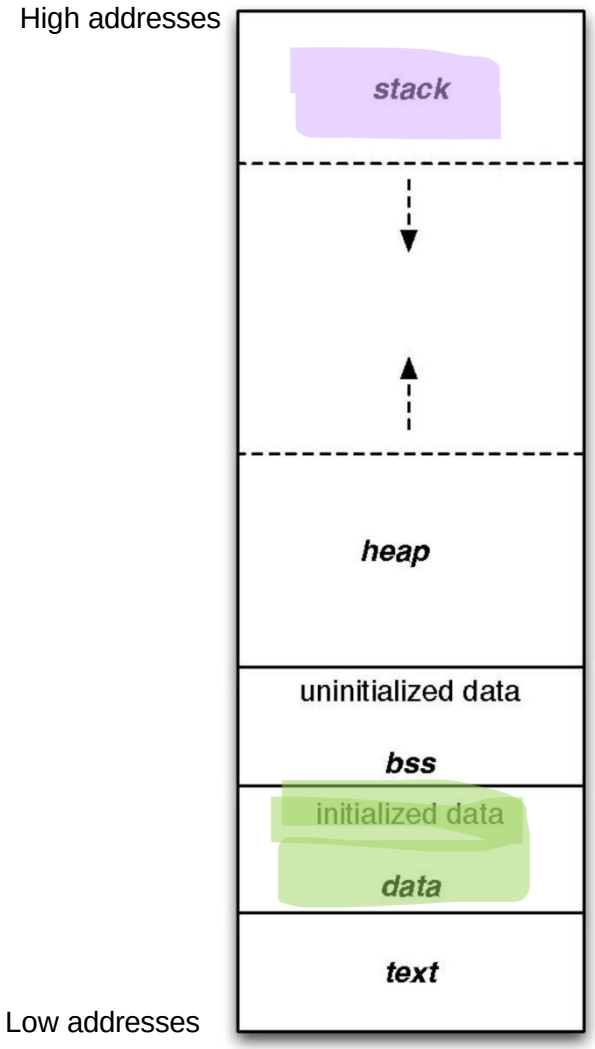

```
int x = 100;
int main()
{
        int a =2;
        float b = 2.5;

        static int y;

        int *ptr = (int *) malloc(2*sizeof(int));

        ptr[0] = 5;
        ptr[1] = 6;

        free(ptr)
        return 1;
}
```

Program layout in memory



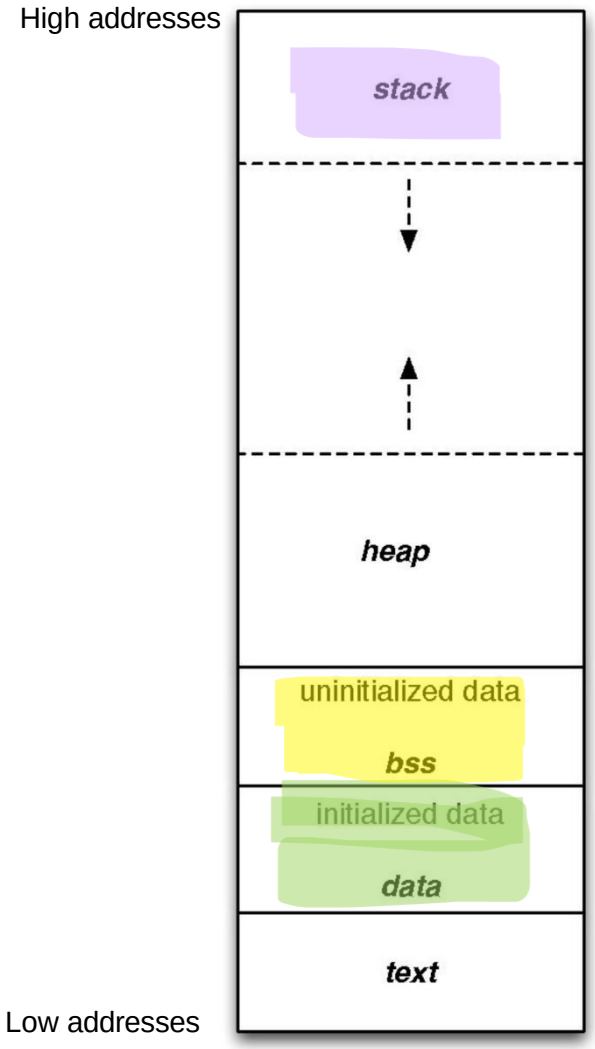

```
int x = 100;
int main()
{
        int a =2;
        float b = 2.5;

        static int y;

        int *ptr = (int *) malloc(2*sizeof(int));

        ptr[0] = 5;
        ptr[1] = 6;

        free(ptr)
        return 1;
}
```

Program layout in memory



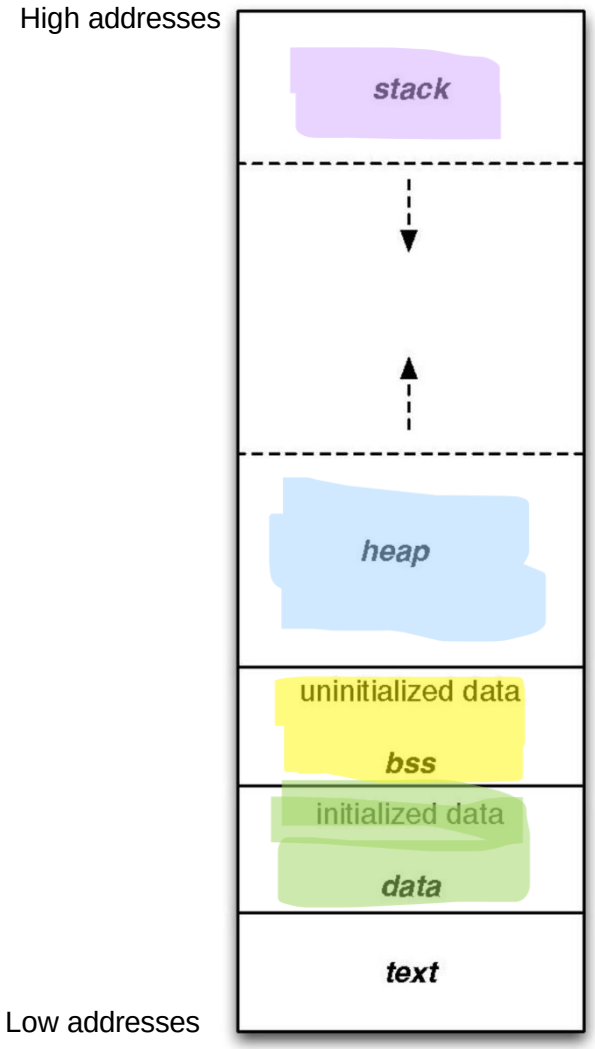

```
int x = 100;
int main()
{
        int a =2;
        float b = 2.5;

        static int y;

        int *ptr = (int *) malloc(2*sizeof(int));

        ptr[0] = 5;
        ptr[1] = 6;

        free(ptr)
        return 1;
}
```

Program layout in memory




```
int x = 100;
int main()
{
        int a =2;
        float b = 2.5;

        static int y;

        int *ptr = (int *) malloc(2*sizeof(int));

        ptr[0] = 5;
        ptr[1] = 6;

        free(ptr)
        return 1;
}
```

Program layout in memory




```
int x = 100;
int main()
{
        int a =2;
        float b = 2.5;

        static int y;

        int *ptr = (int *) malloc(2*sizeof(int));

        ptr[0] = 5;
        ptr[1] = 6;

        free(ptr)
        return 1;
}
```

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

       printf(p);

       return 0;
}
```

Stack and Function Invocation

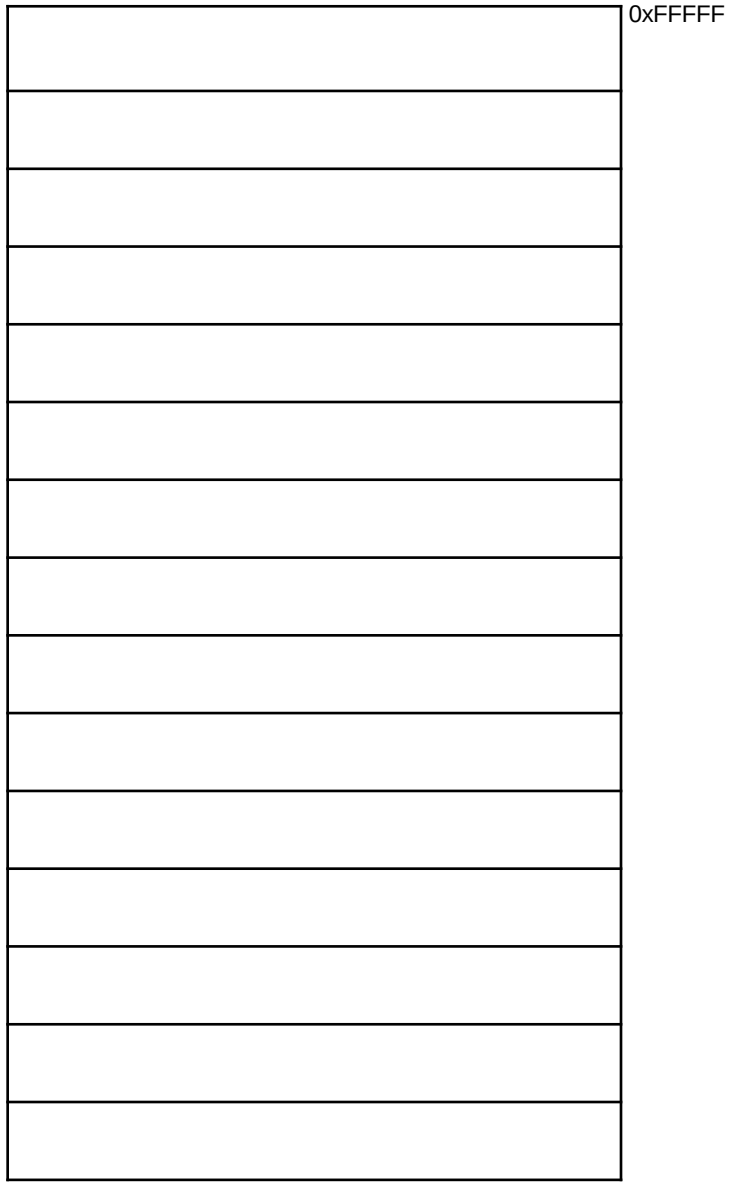
```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Every time a function is called, memory gets allocated on **the stack** to hold function values and information

The Stack



Stack and Function Invocation

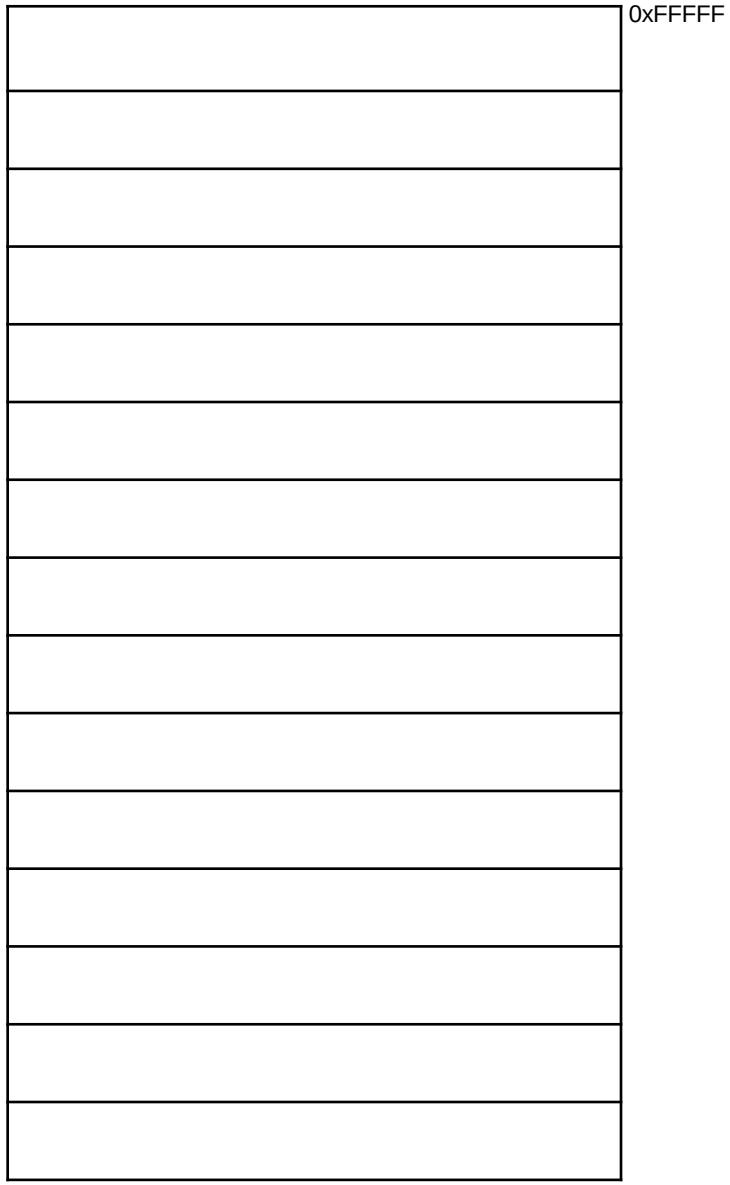
```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

This memory on the stack is called a **stack frame**

Every time a function is called, memory gets allocated on **the stack** to hold function values and information



Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

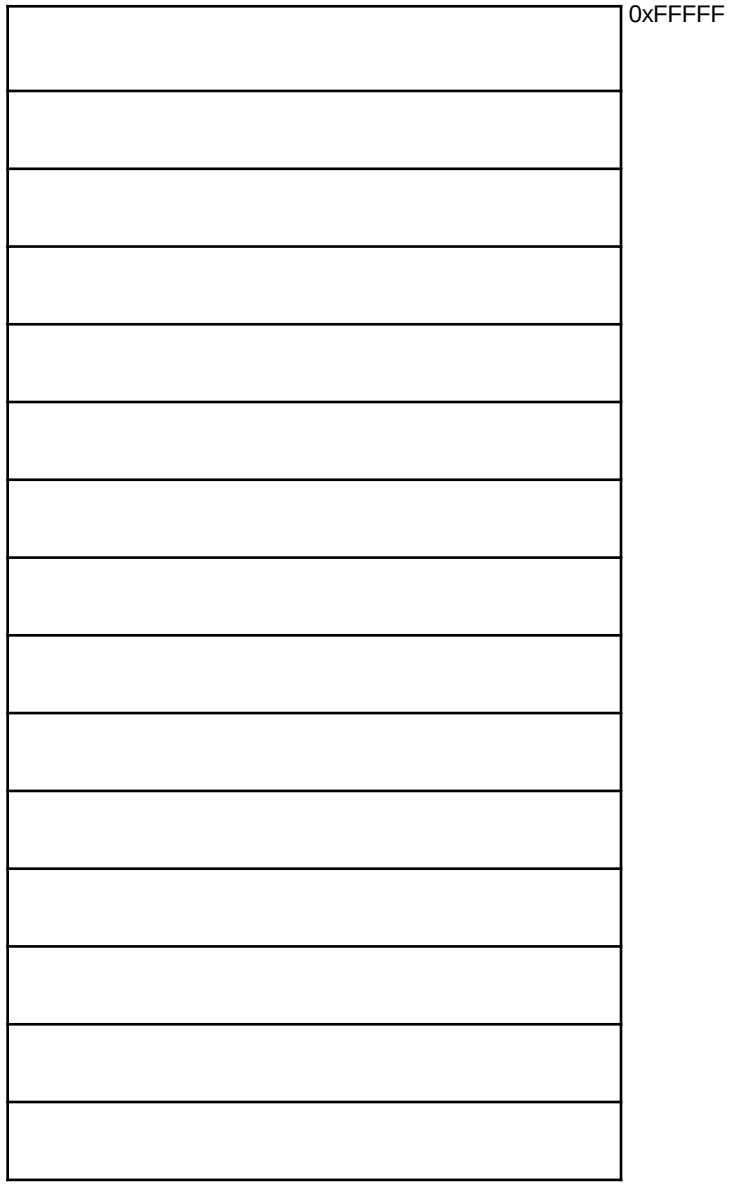
```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

The stack frame consists of local variables, function arguments, and addresses

The Stack

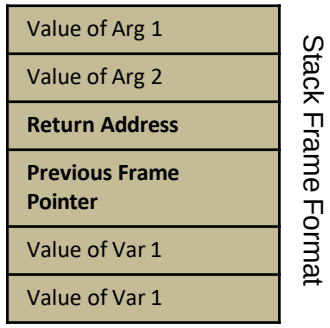


Stack and Function Invocation

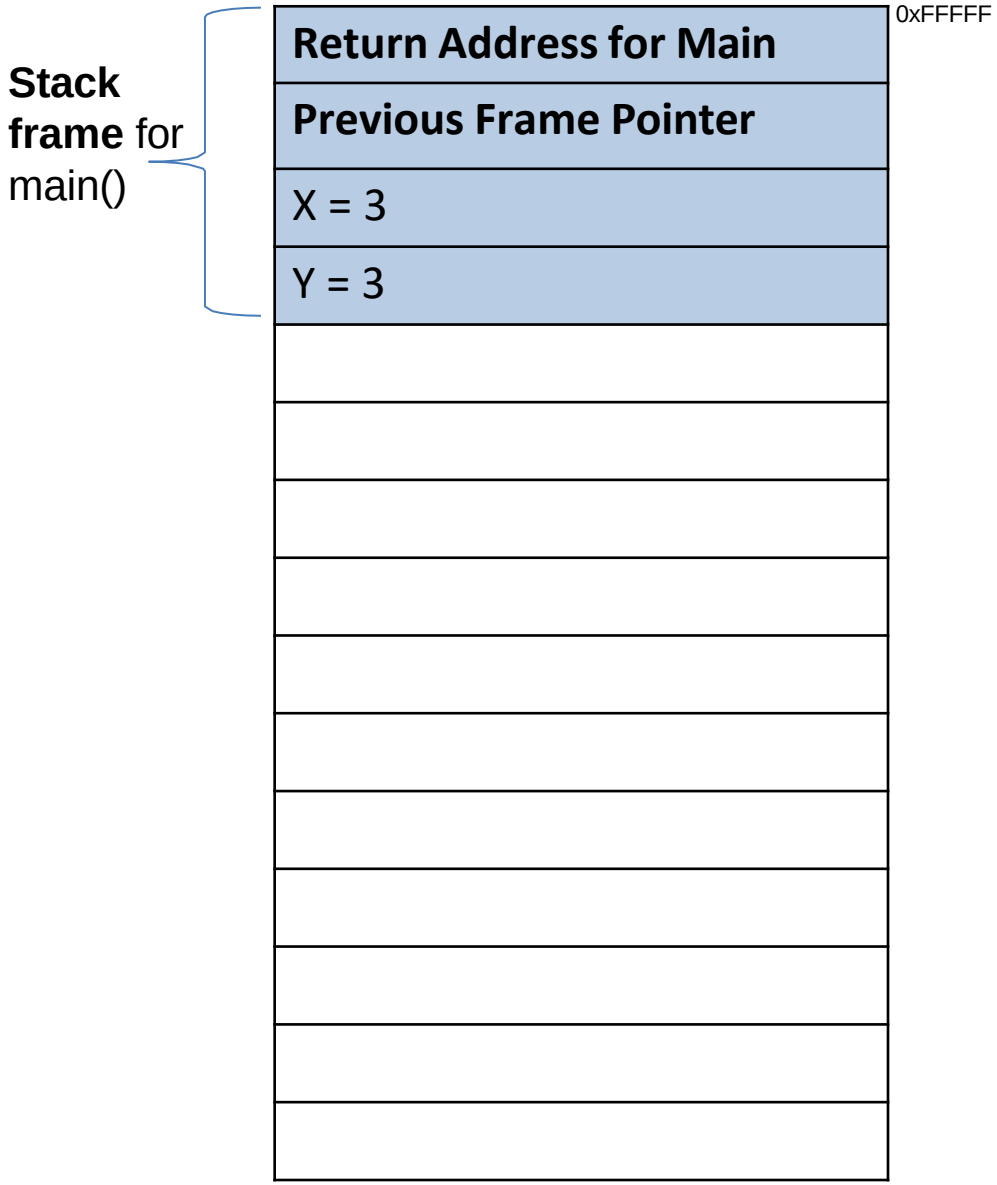
```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```



The Stack



Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

Stack Frame Format

The Stack

Stack frame for main()

0xFFFFF

Return Address for Main
Previous Frame Pointer
X = 3
Y = 3

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

Stack Frame Format

Stack frame for foo()

X = 3
Y = 3
Return Address for foo()
Previous Frame Pointer
Z = 1

We need to know where to return to when this function finishes

The Stack

Stack frame for main()

Return Address for Main
Previous Frame Pointer
X = 3
Y = 3

0xFFFFF

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

Stack Frame Format

Stack frame for foo()

X = 3
Y = 3
Return Address for foo()
Previous Frame Pointer
Z = 1

We need to know where to return to when this function finishes

The Stack

Stack frame for main()

0xFFFFF

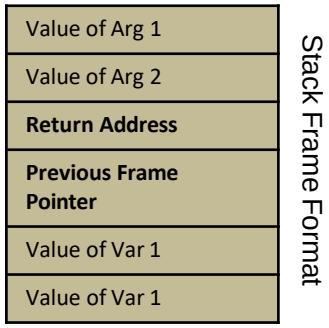
Return Address for Main
Previous Frame Pointer
X = 3
Y = 3

Stack and Function Invocation

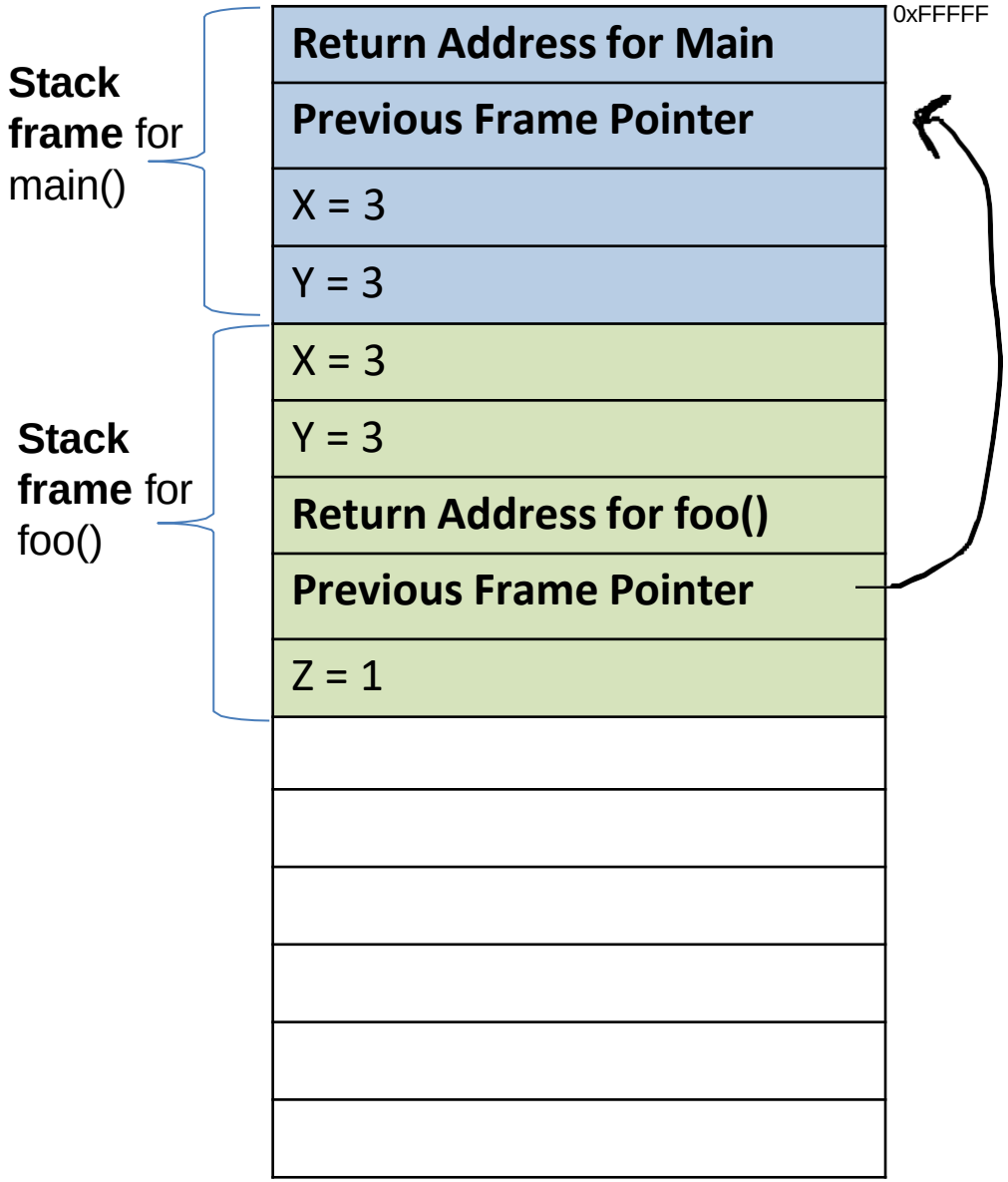
```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```



The Stack



Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

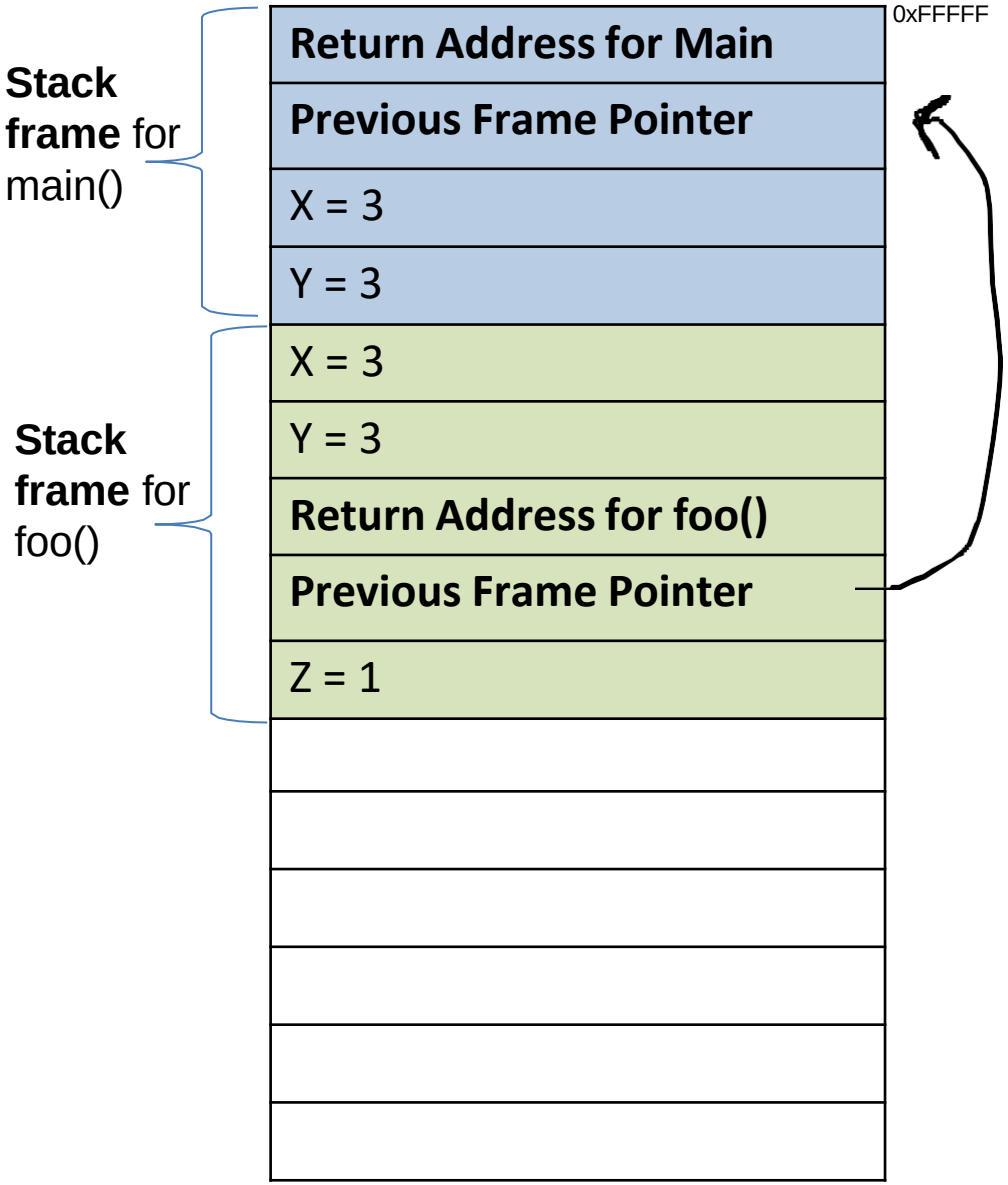
Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

Stack Frame Format

Stack frame for foo2()

p = 1
Return Address for foo2
Previous Frame Pointer

The Stack



Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

Stack Frame Format

Stack frame for foo2()

p = 1
Return Address for foo2
Previous Frame Pointer

The Stack

0xFFFFF

Stack frame for main()

Stack frame for foo()

Return Address for Main
Previous Frame Pointer
X = 3
Y = 3
X = 3
Y = 3
Return Address for foo()
Previous Frame Pointer
Z = 1

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

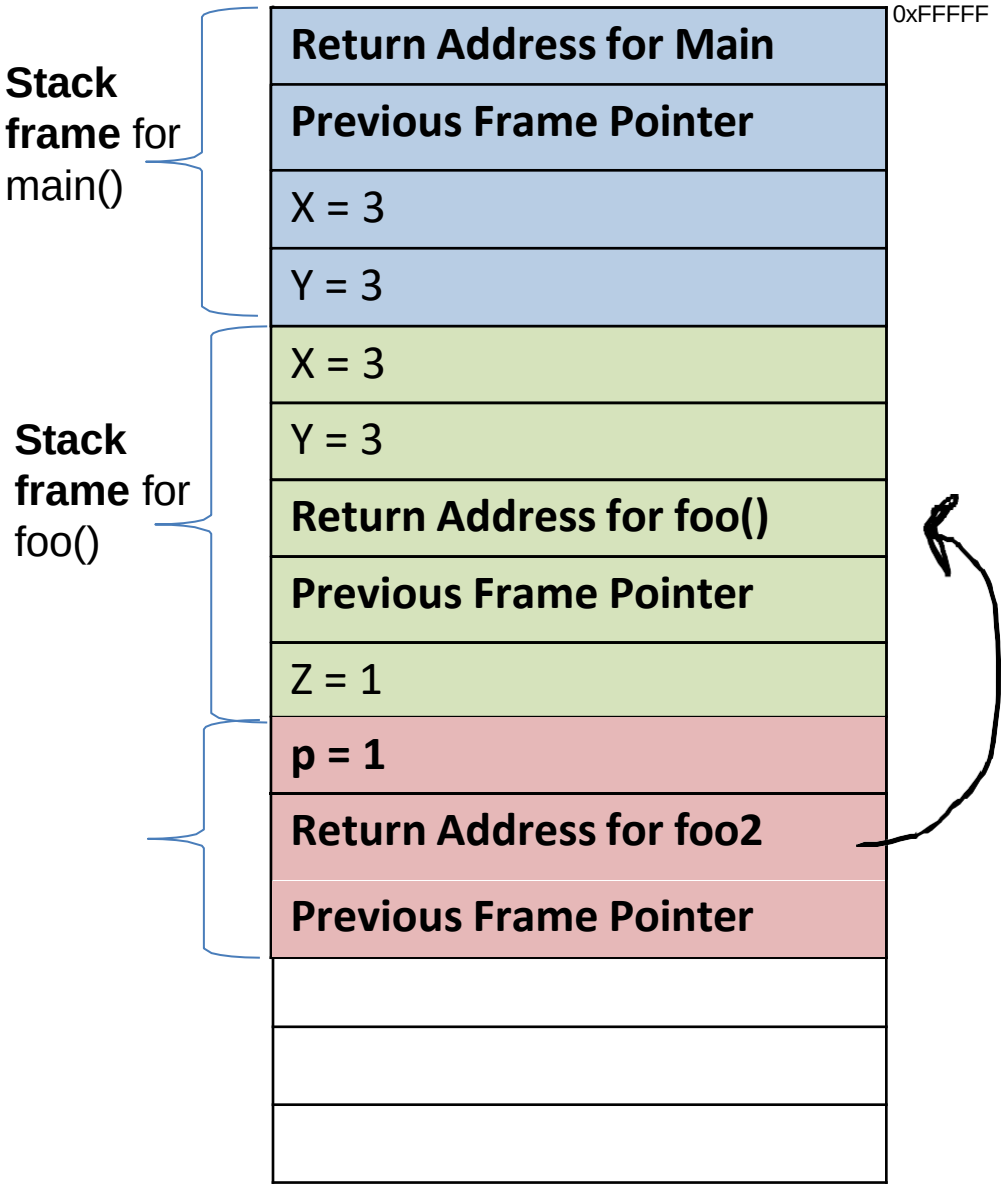
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

Stack Frame Format

The Stack



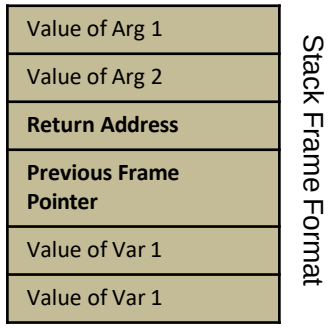
Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

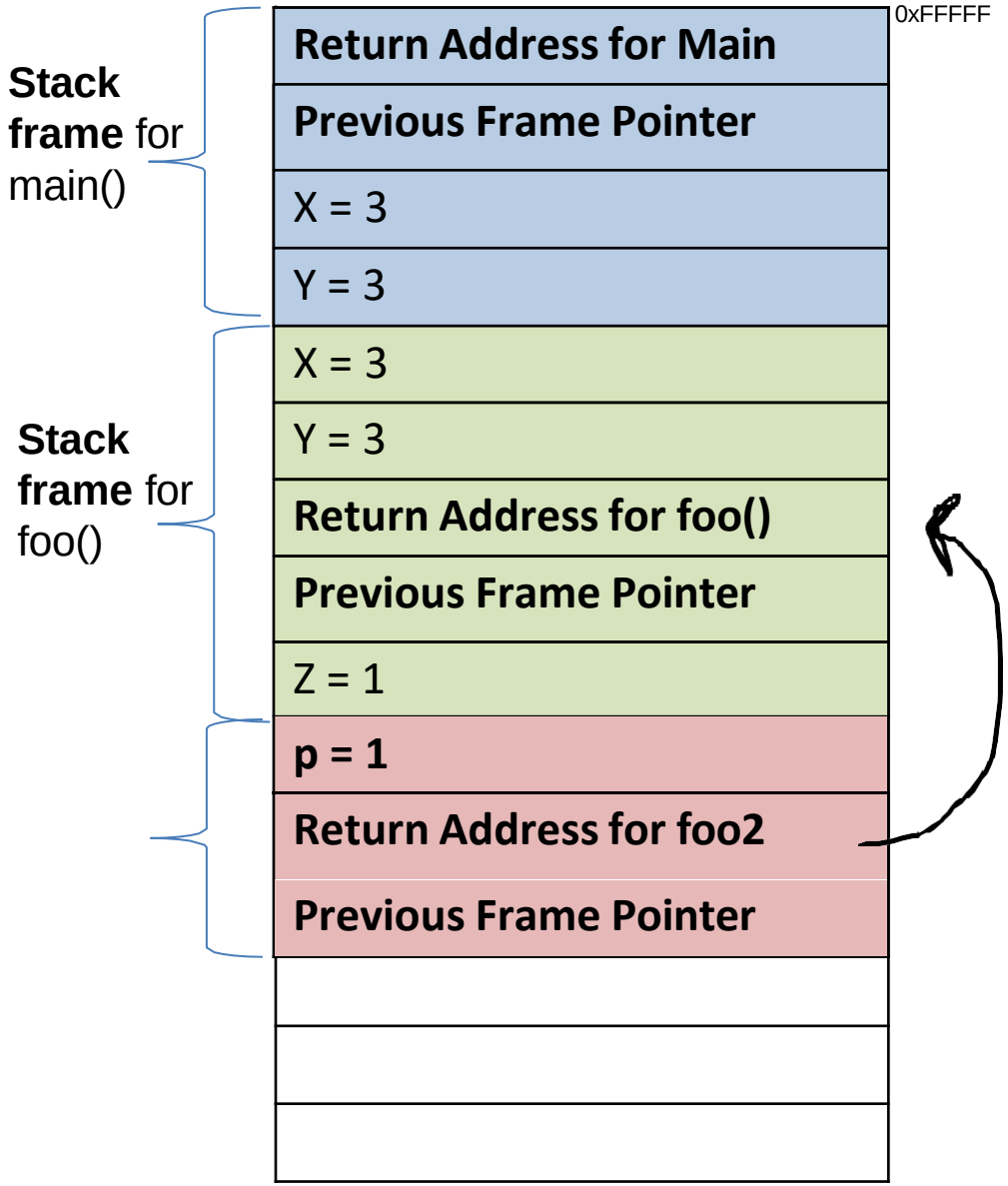
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

       printf(p);

       return 0;
}
```

This function is finished, so we need to determine where the next instruction of the program is



The Stack

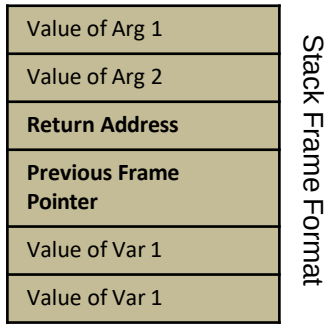


Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

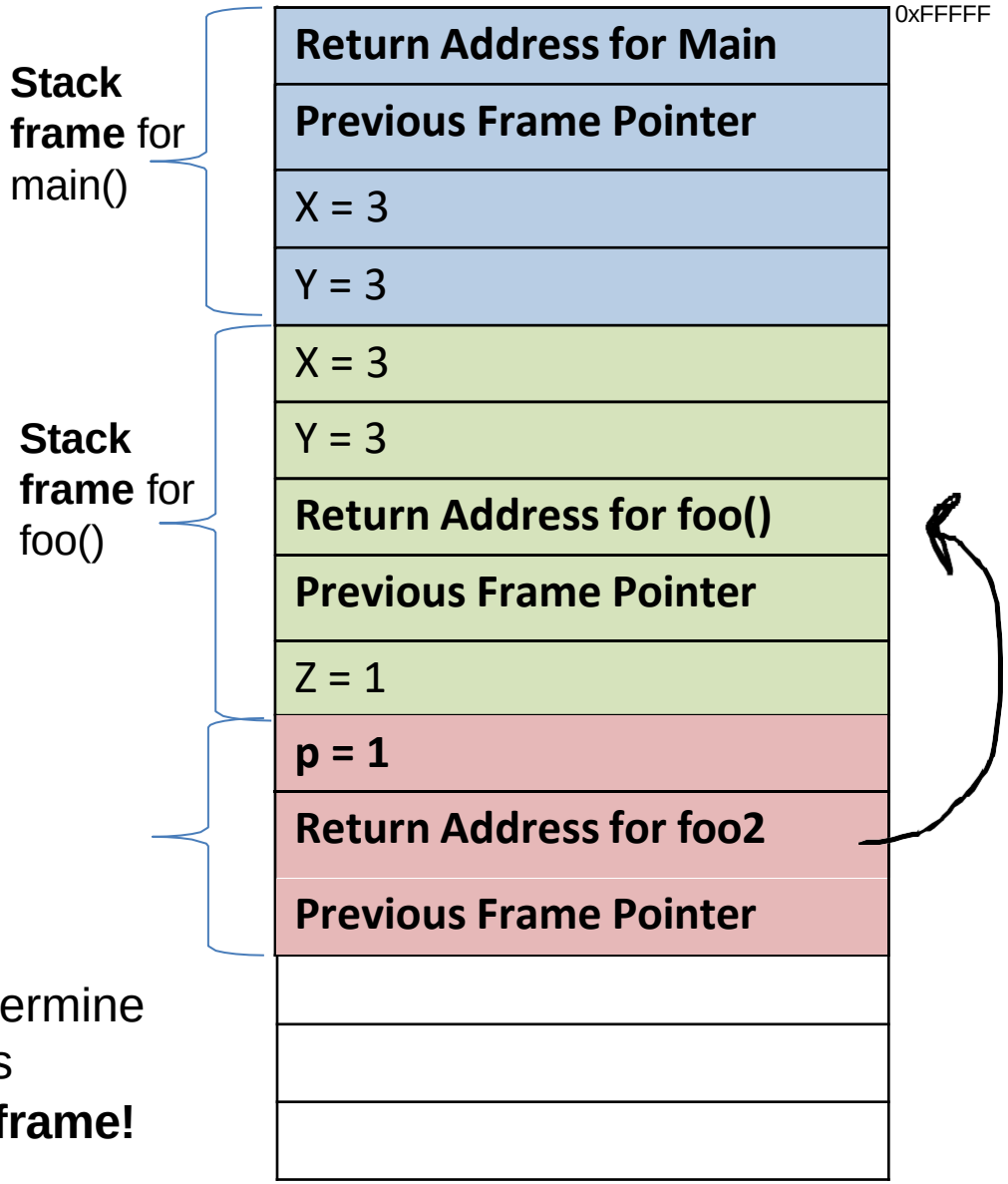
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

       printf(p);

       return 0;
}
```



This function is finished, so we need to determine where the next instruction of the program is

Look at the return address in the stack frame!



Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

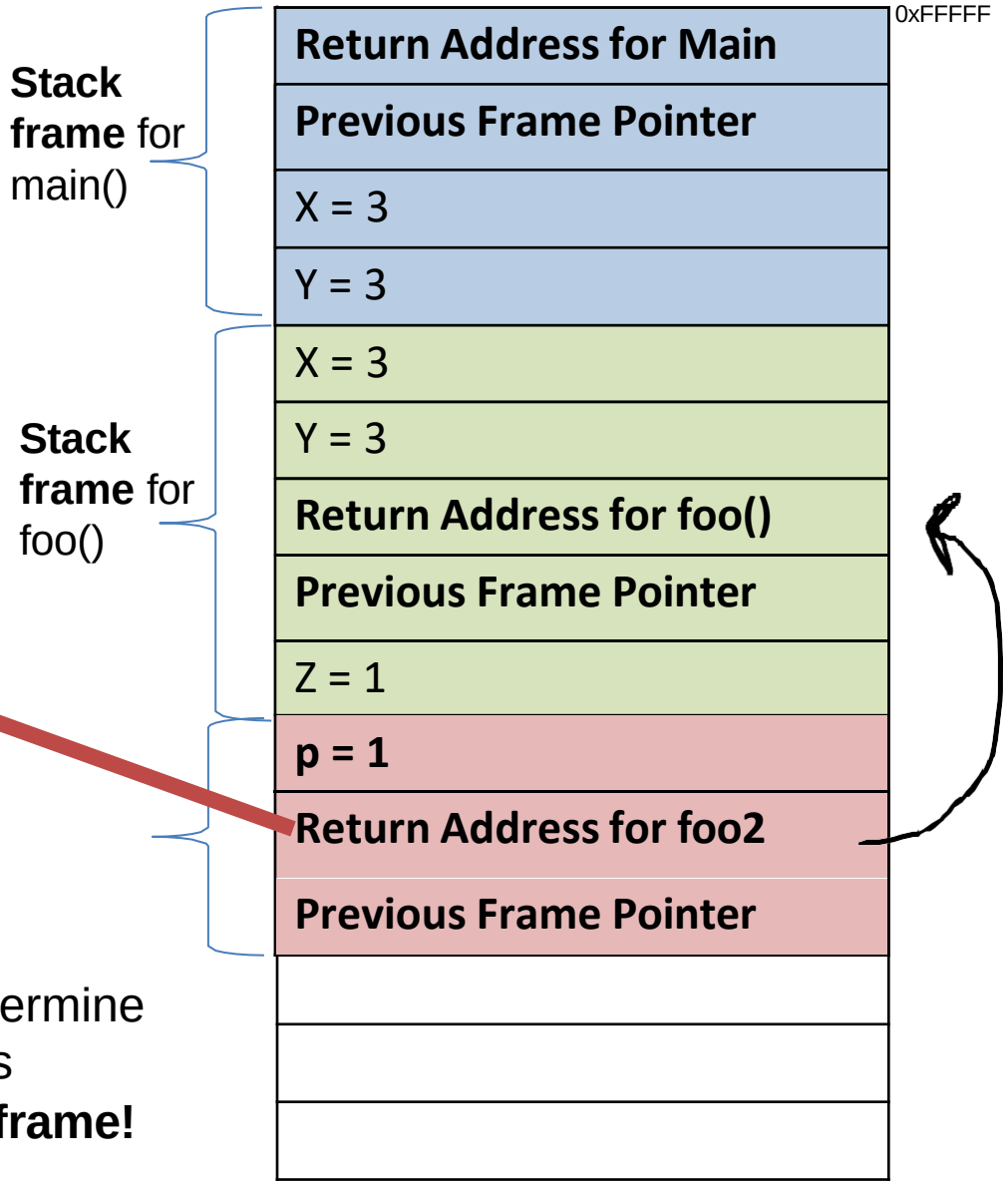
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

Stack Frame Format

The Stack



Return back to foo()

This function is finished, so we need to determine where the next instruction of the program is

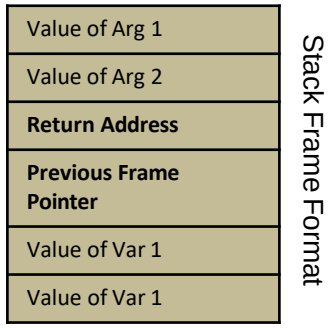
Look at the return address in the stack frame!

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

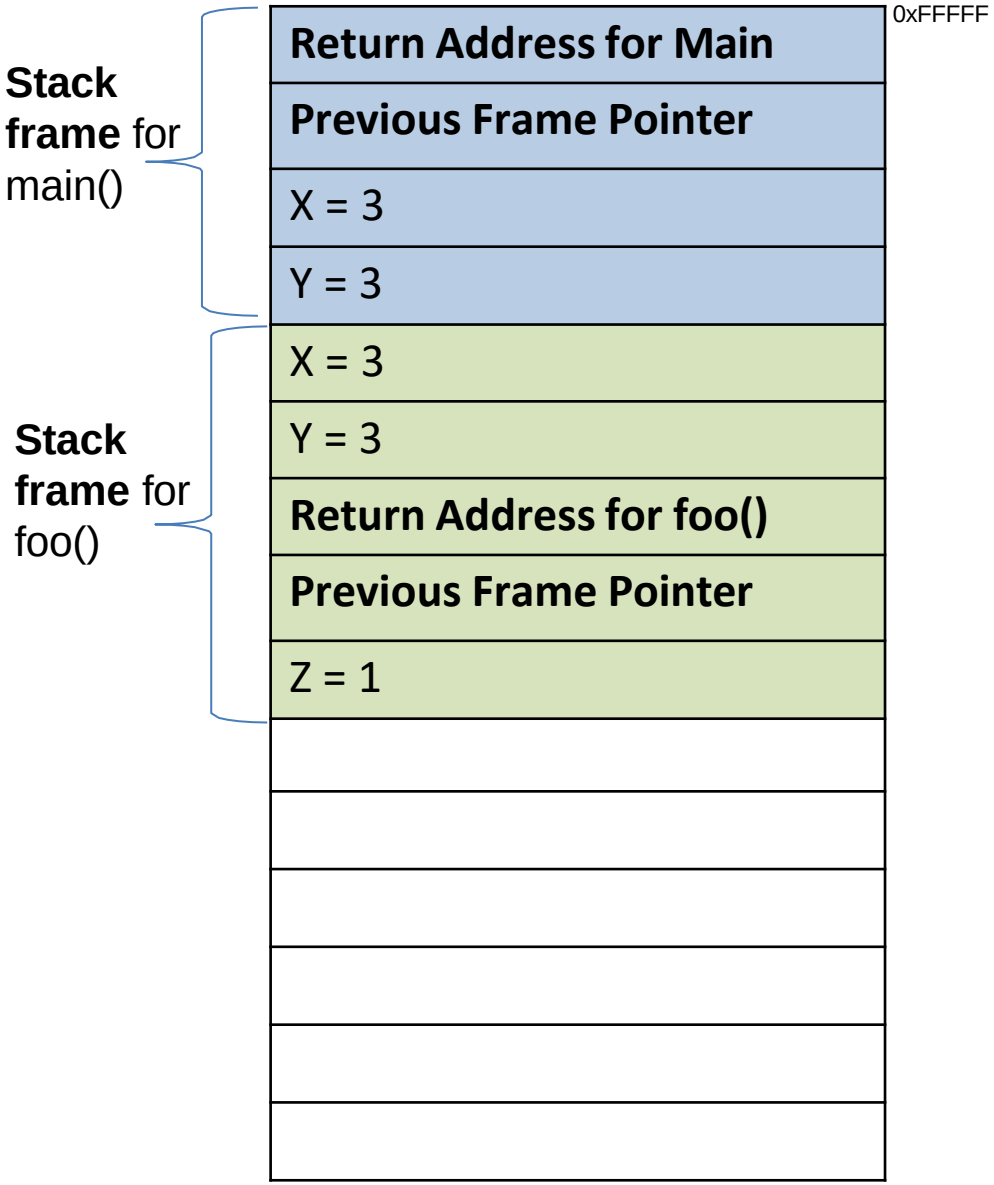
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```



`foo2()` is finished, so we can remove their information from the stack

The Stack



Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

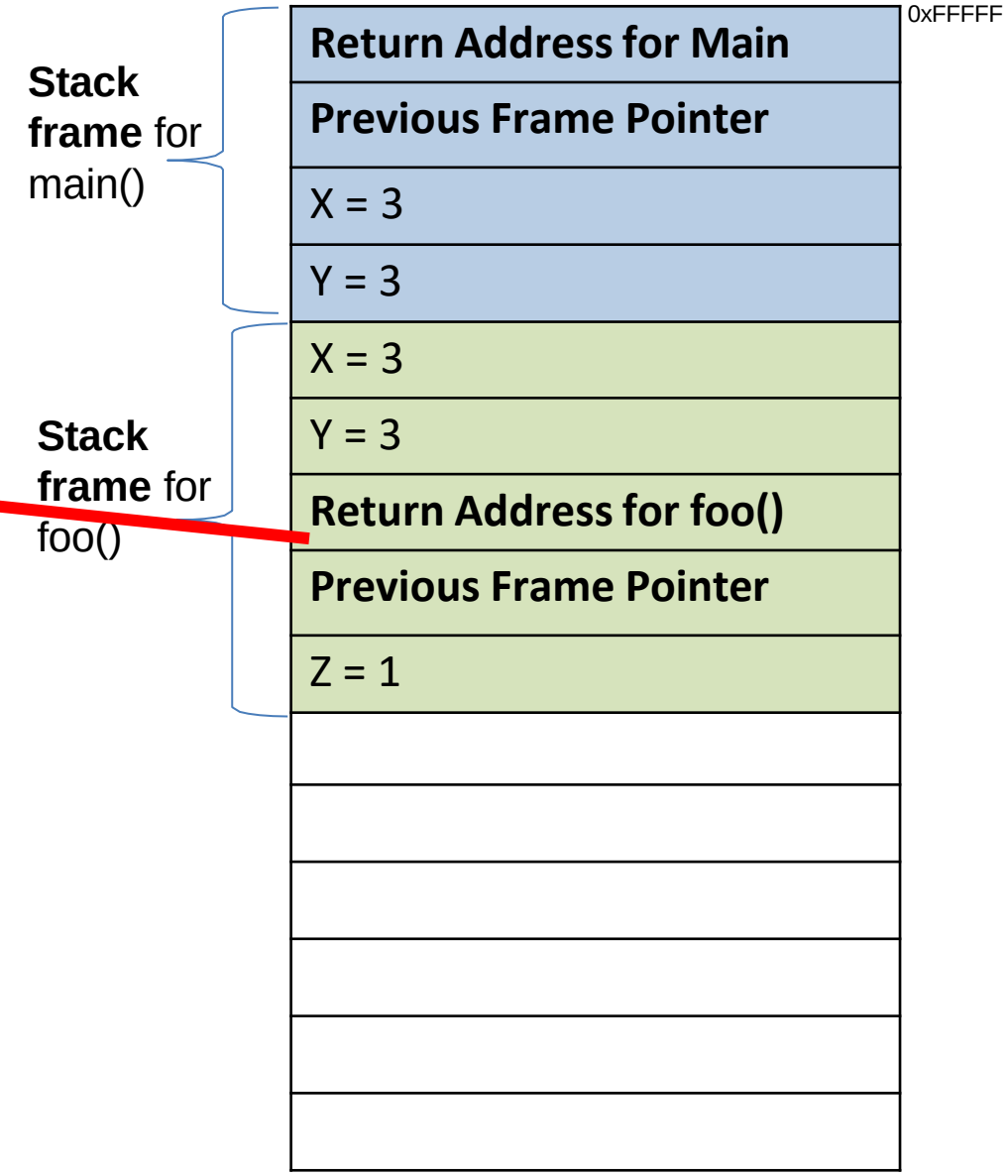
```
int foo2(p){

        printf(p);

        return 0;
}
```

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

Stack Frame Format

`foo()` is done, we now need to return back to main!

The Stack

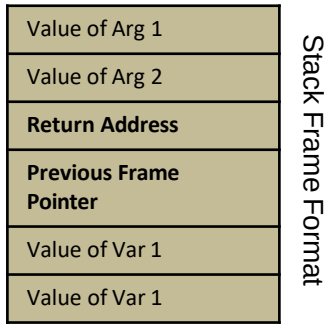


Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

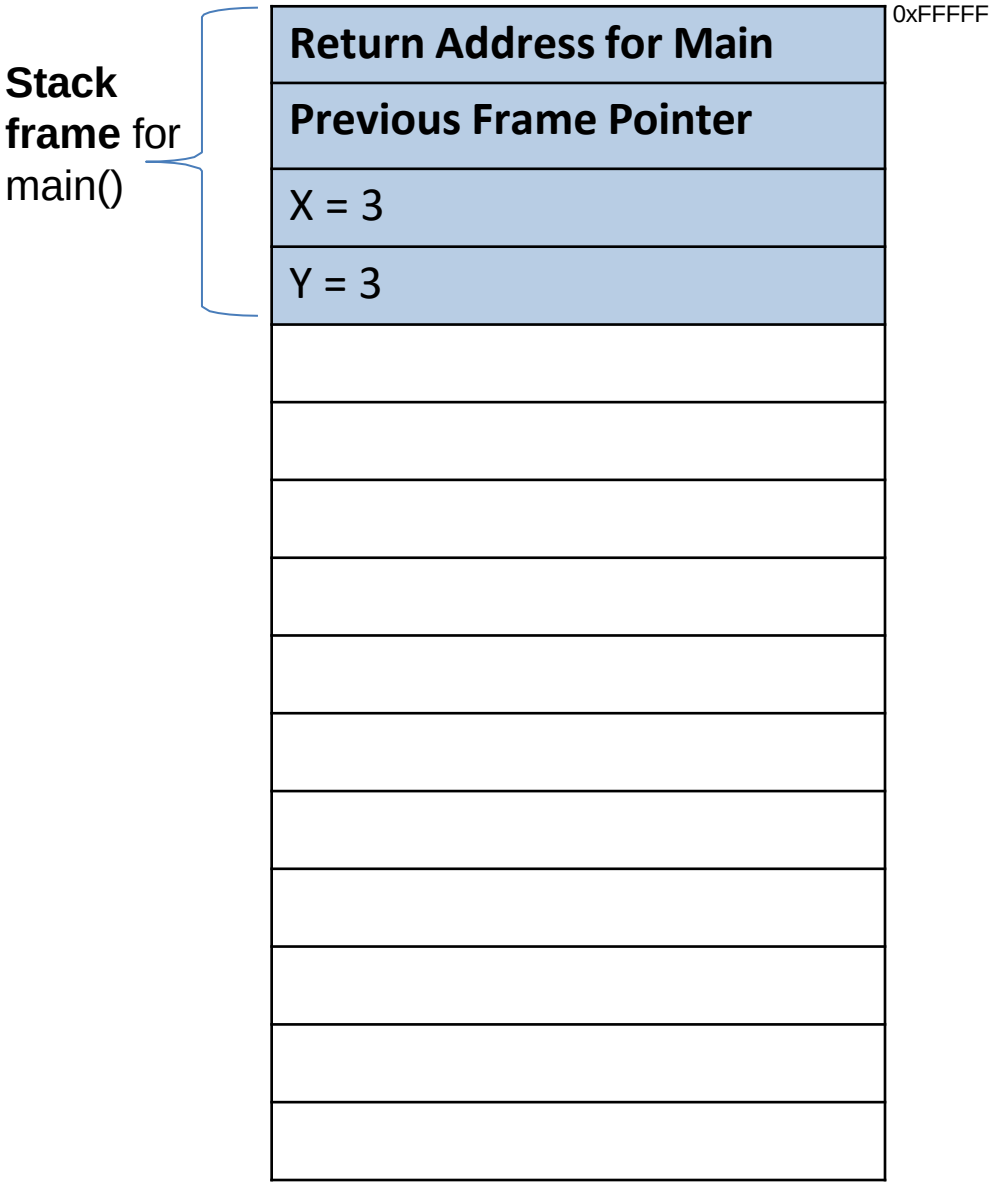
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```



`foo()` is done, we now need to return back to main!

The Stack

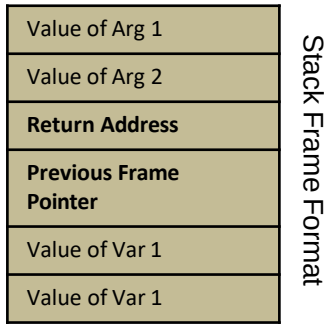


Stack and Function Invocation

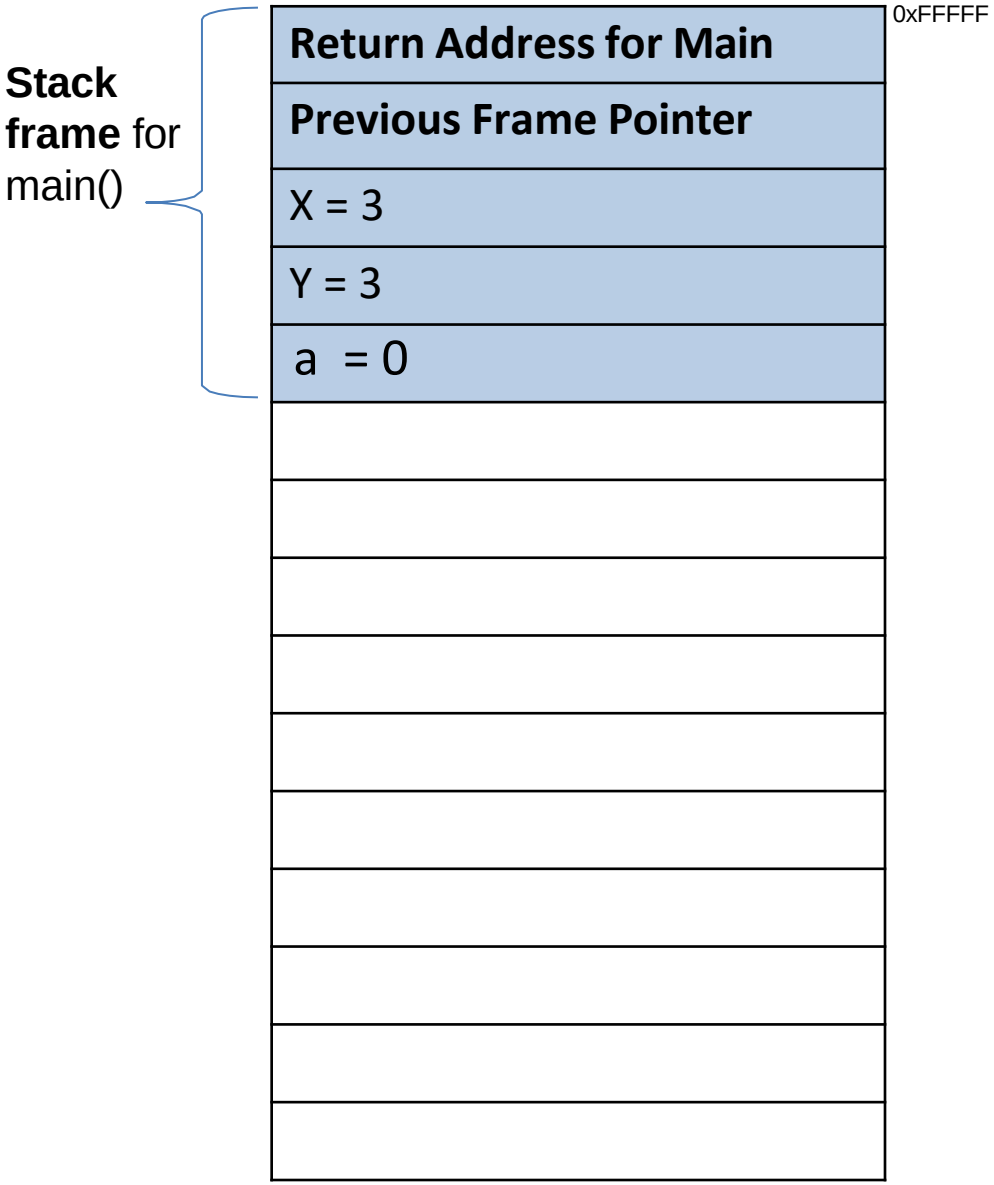
```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

       printf(p);

       return 0;
}
```



The Stack

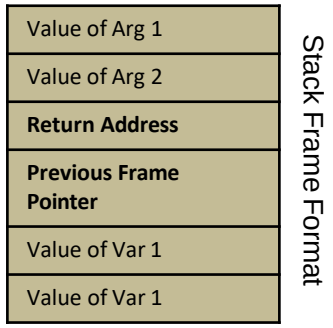


Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

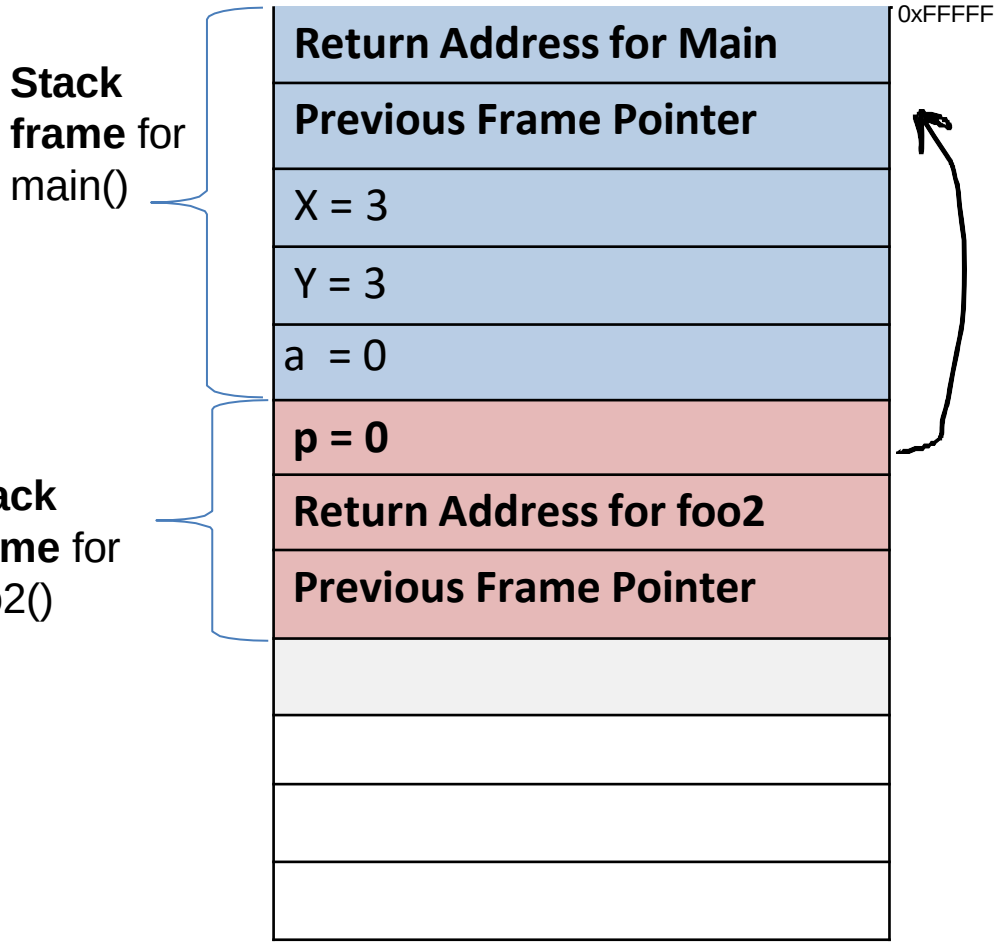
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```



`foo2()` is called again, so a new stack frame is created and put onto the stack

The Stack

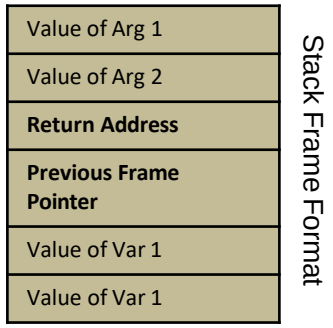


Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

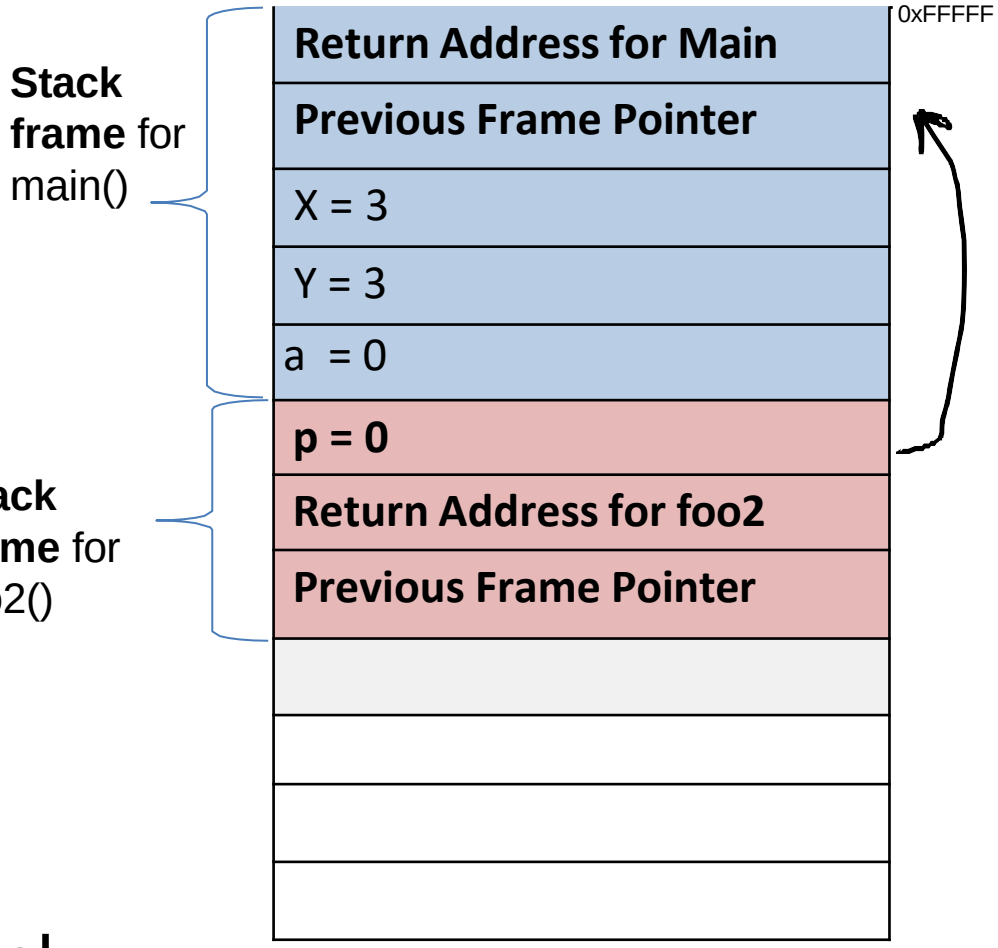
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```



When `foo2()` is finished, it will return back to `main()`

The Stack



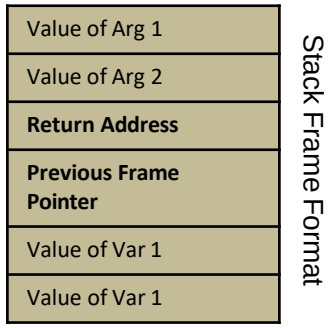
Stack and Function Invocation

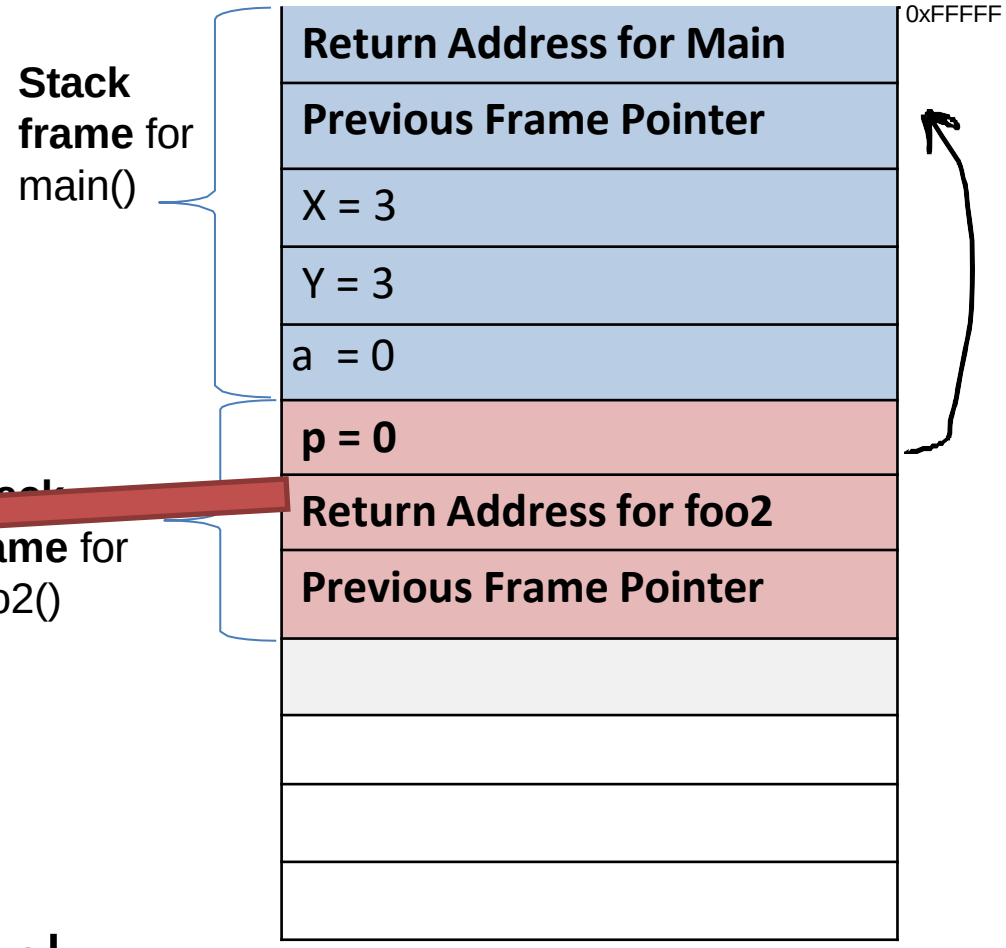
```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

When `foo2()` is finished, it will return back to `main()`



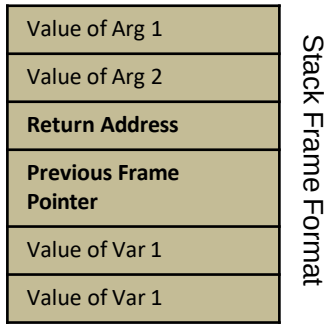


Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

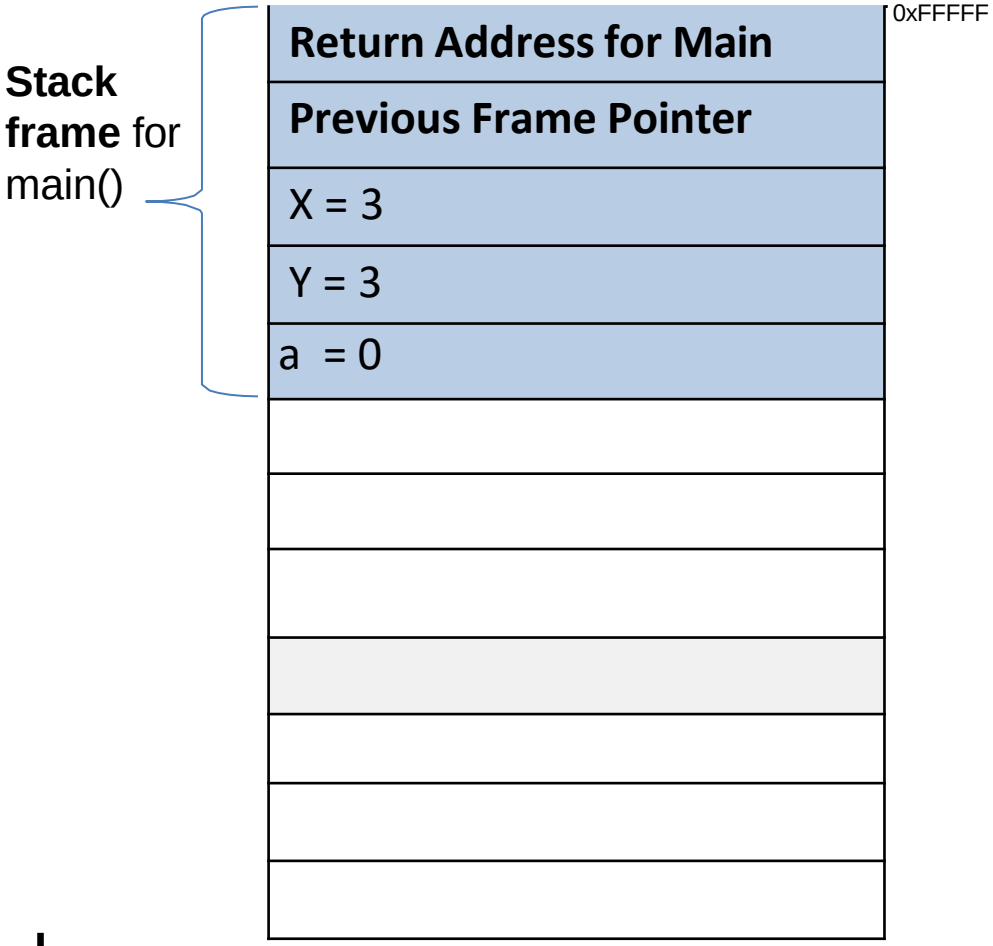
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```



When `foo2()` is finished, it will return back to `main()`

The Stack

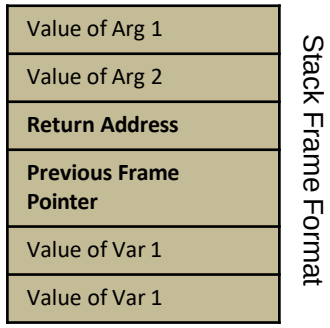


Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

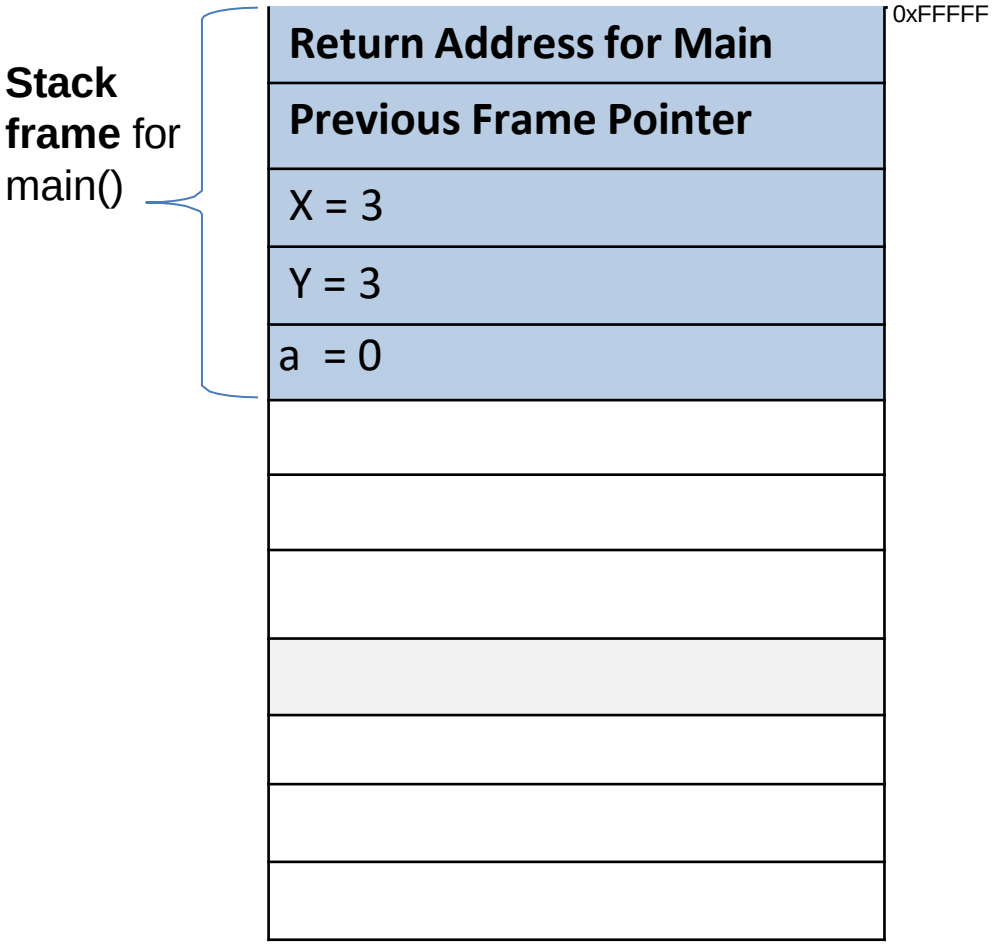
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```



Program done!

The Stack

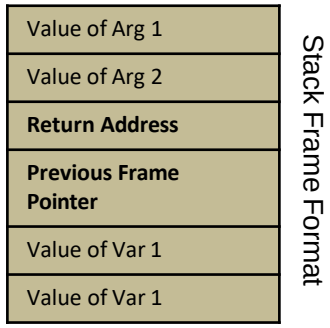


Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

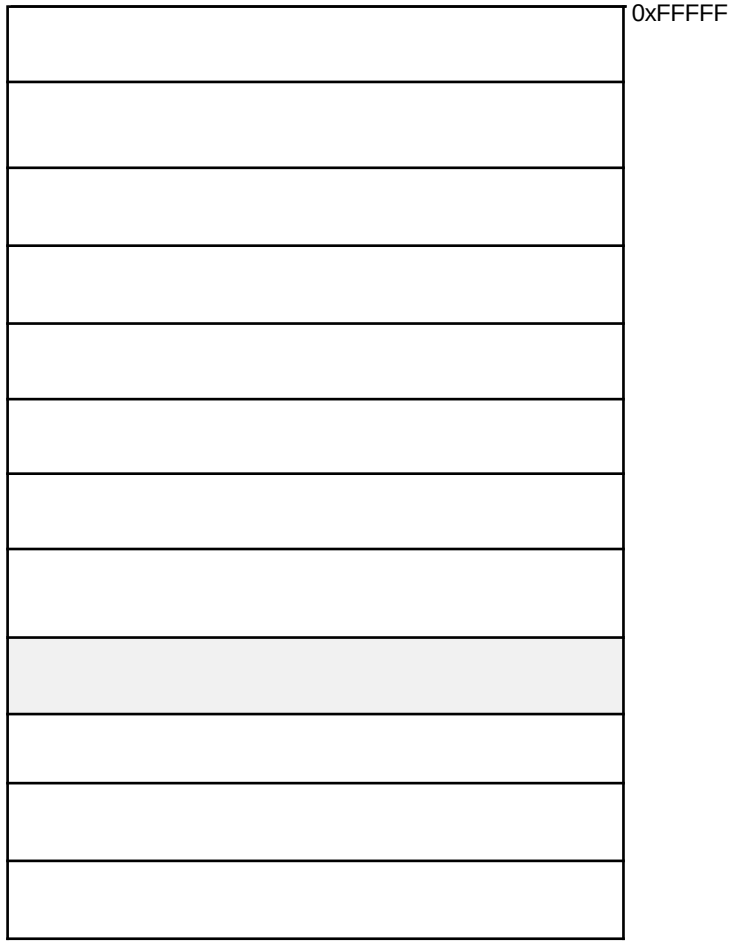
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```



Program done!

The Stack



Demo

is_c_good.c

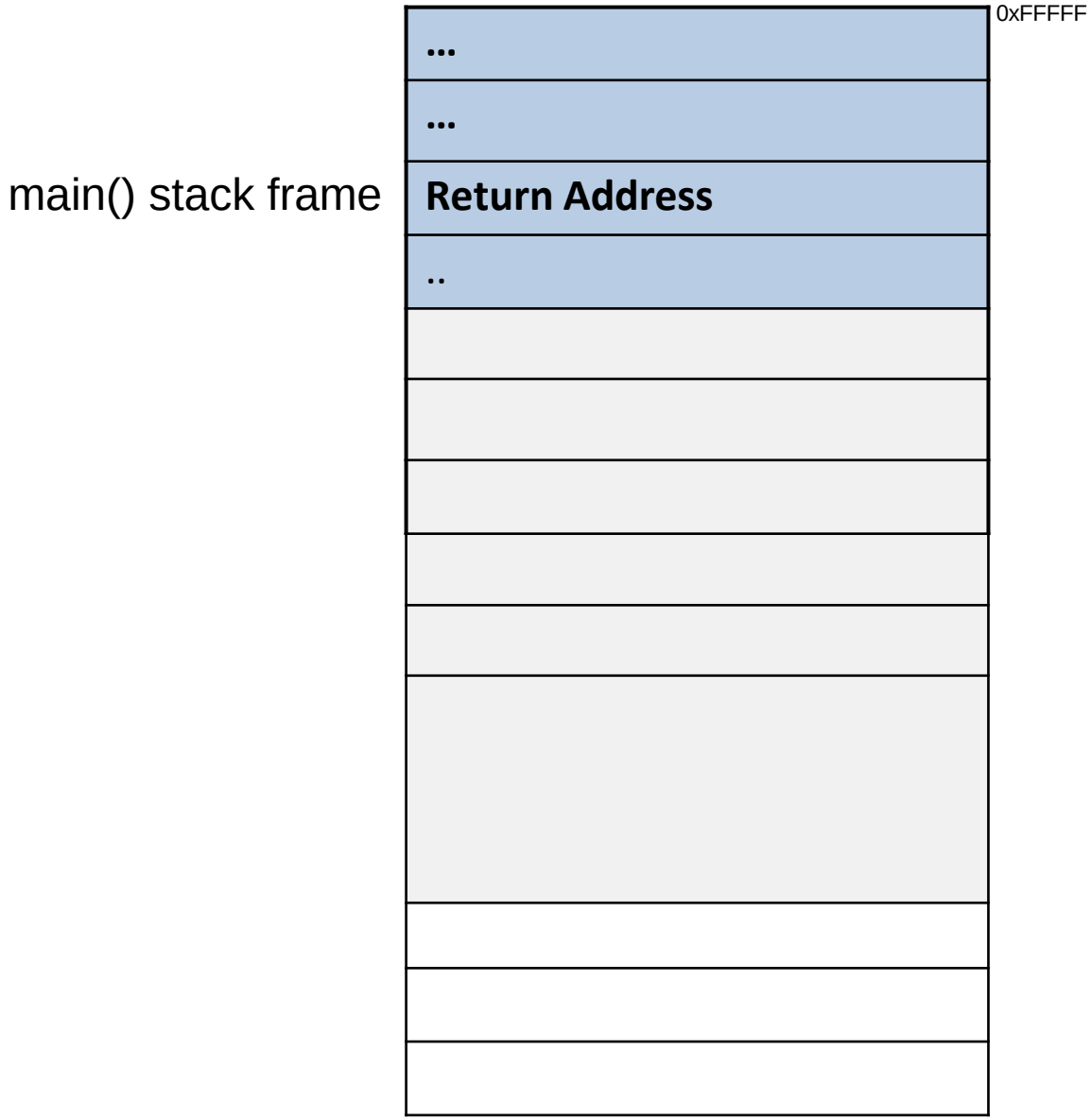
```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
        char buffer[10];
        strcpy(buffer, str);

}


int main(int argc, char *argv[])
{
        foo(argv[1]);
        printf("Returned Properly\n");
        return 0;

}
```

The Stack

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
        char buffer[10];
        strcpy(buffer, str);

}


int main(int argc, char *argv[])
{
        foo(argv[1]);
        printf("Returned Properly\n");
        return 0;

}
```

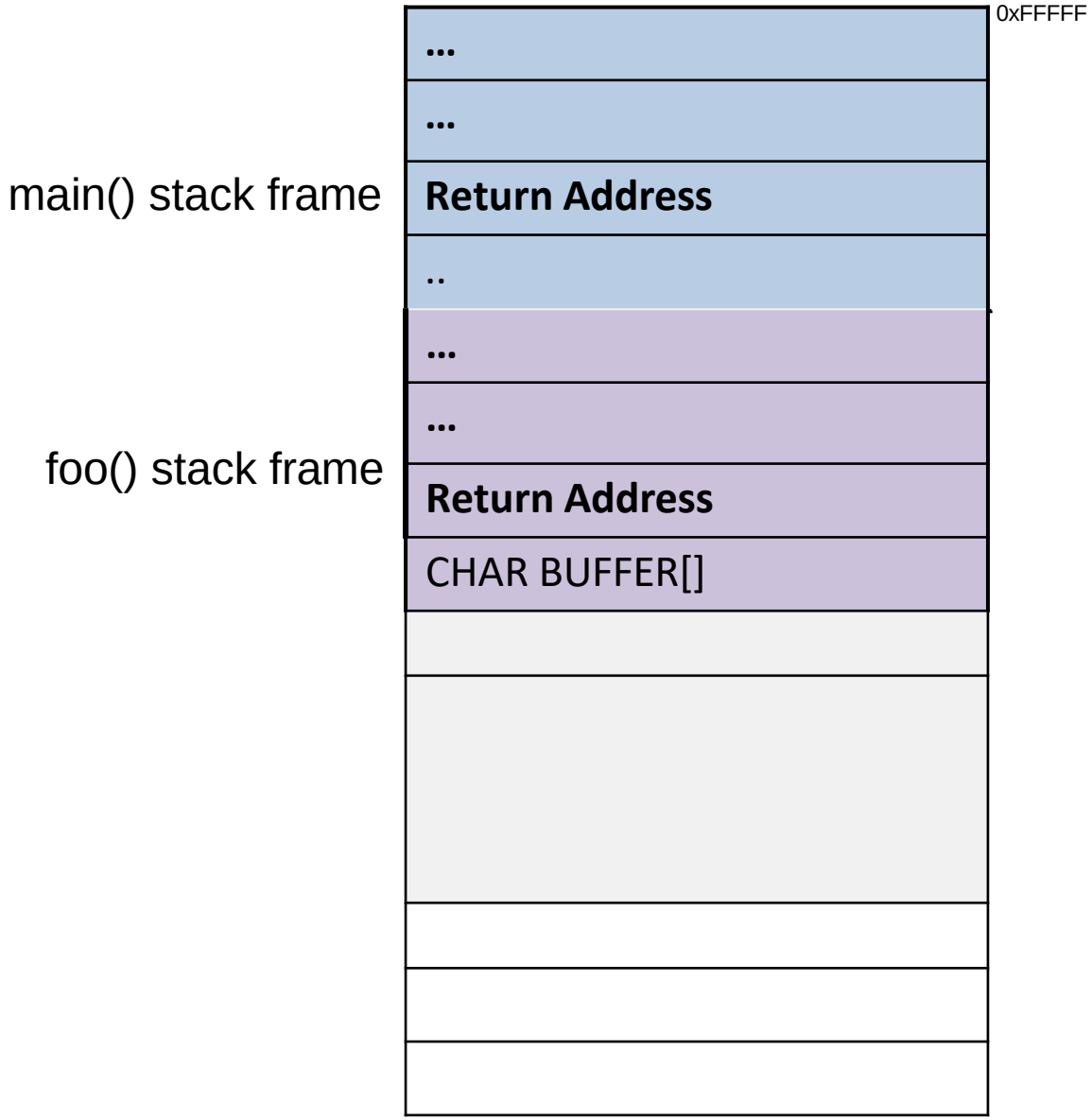
The Stack

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
        char buffer[10];
        strcpy(buffer, str);

}


int main(int argc, char *argv[])
{
        foo(argv[1]);
        printf("Returned Properly\n");
        return 0;

}
```



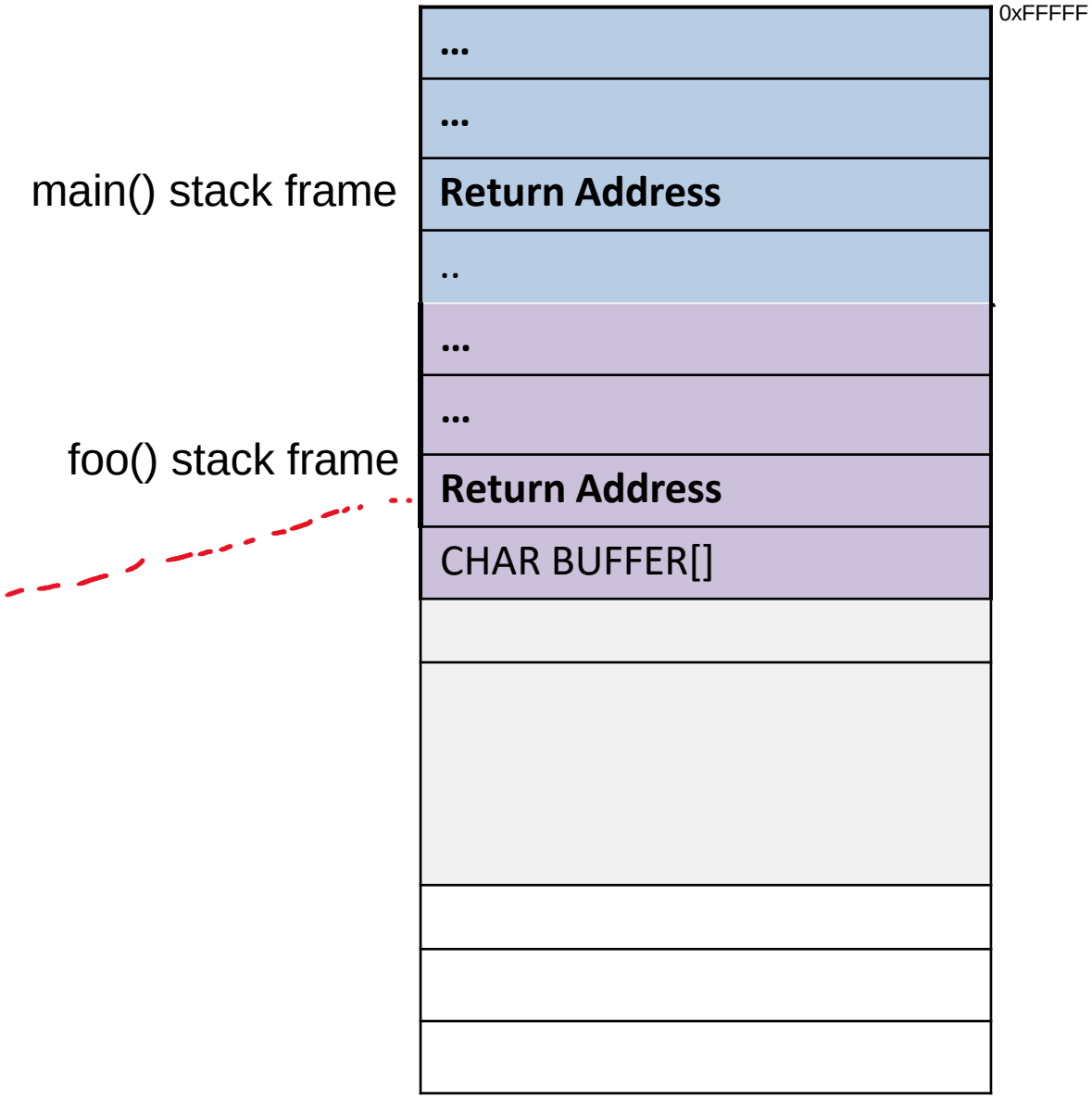
The Stack

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
        char buffer[10];
        strcpy(buffer, str);

}


int main(int argc, char *argv[])
{
        foo(argv[1]);
        printf("Returned Properly\n");
        return 0;

}
```



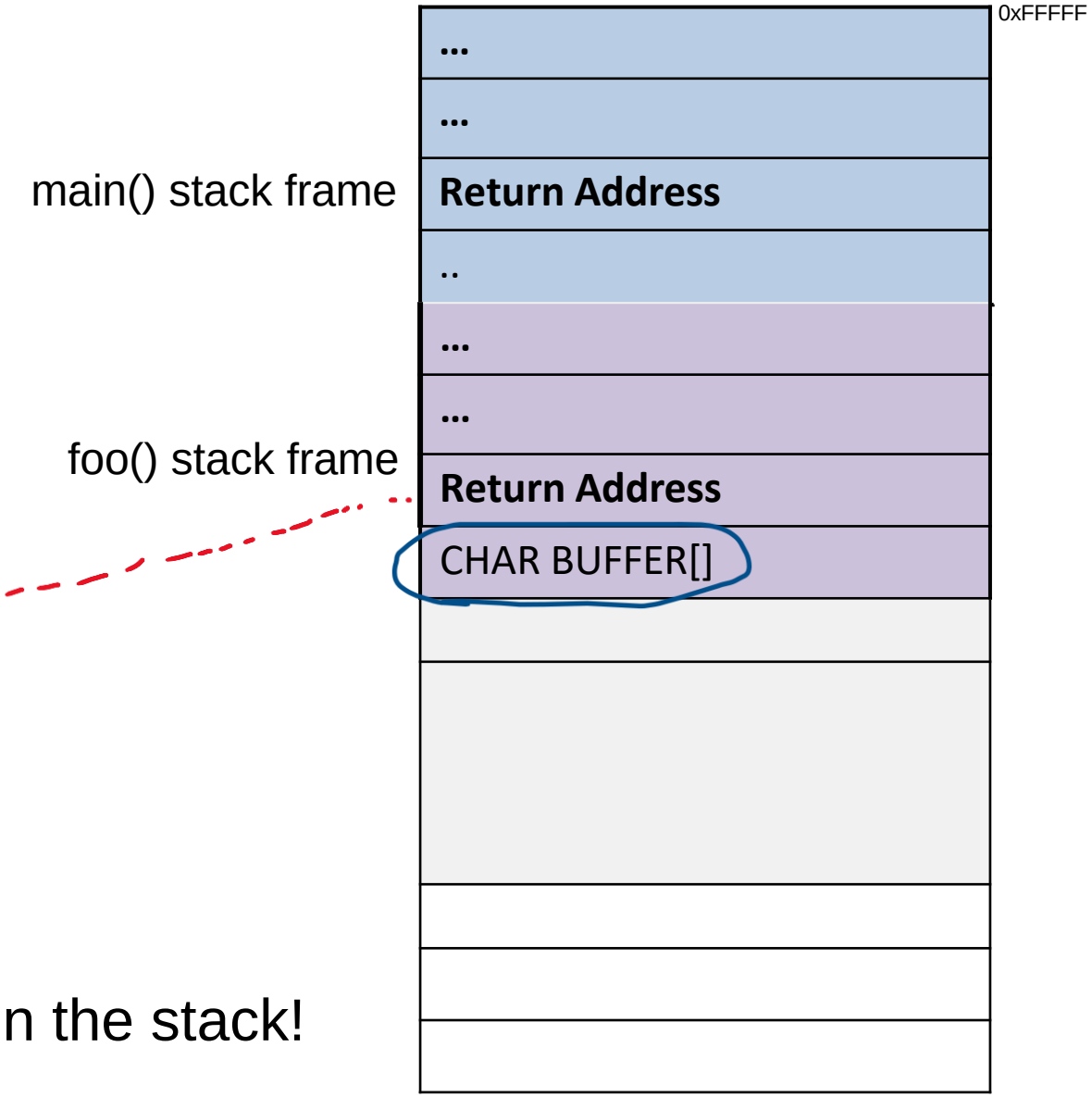
The Stack

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
        char buffer[10];
        strcpy(buffer, str);

}


int main(int argc, char *argv[])
{
        foo(argv[1]);
        printf("Returned Properly\n");
        return 0;

}
```



The Stack

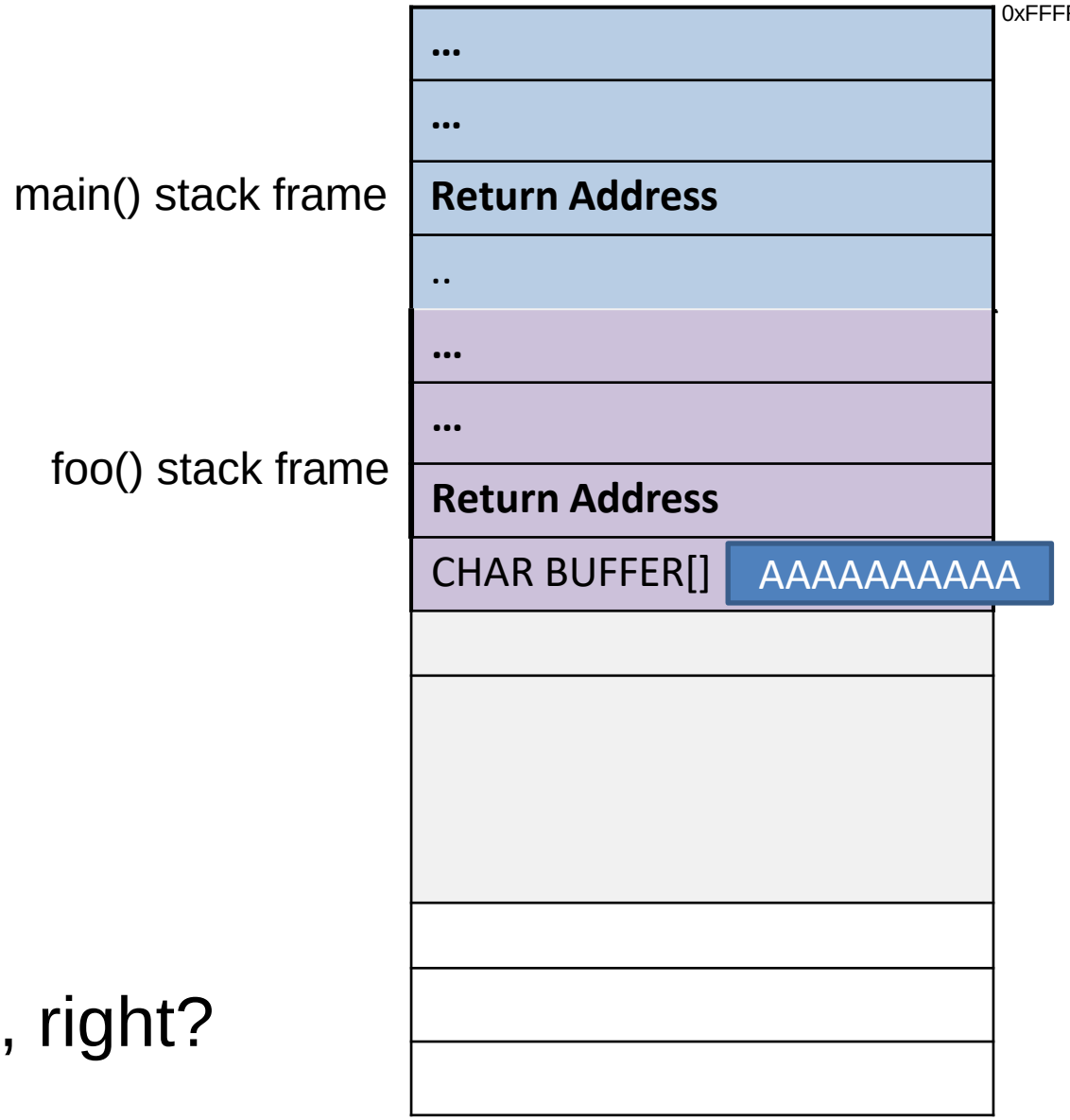
```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
        char buffer[10];
        strcpy(buffer, str);

}


int main(int argc, char *argv[])
{
        foo(argv[1]);
        printf("Returned Properly\n");
        return 0;

}
```



The input of this program eventually gets put on the stack!

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
        char buffer[10];
        strcpy(buffer, str);

}


int main(int argc, char *argv[])
{
        foo(argv[1]);
        printf("Returned Properly\n");
        return 0;

}
```

The Stack

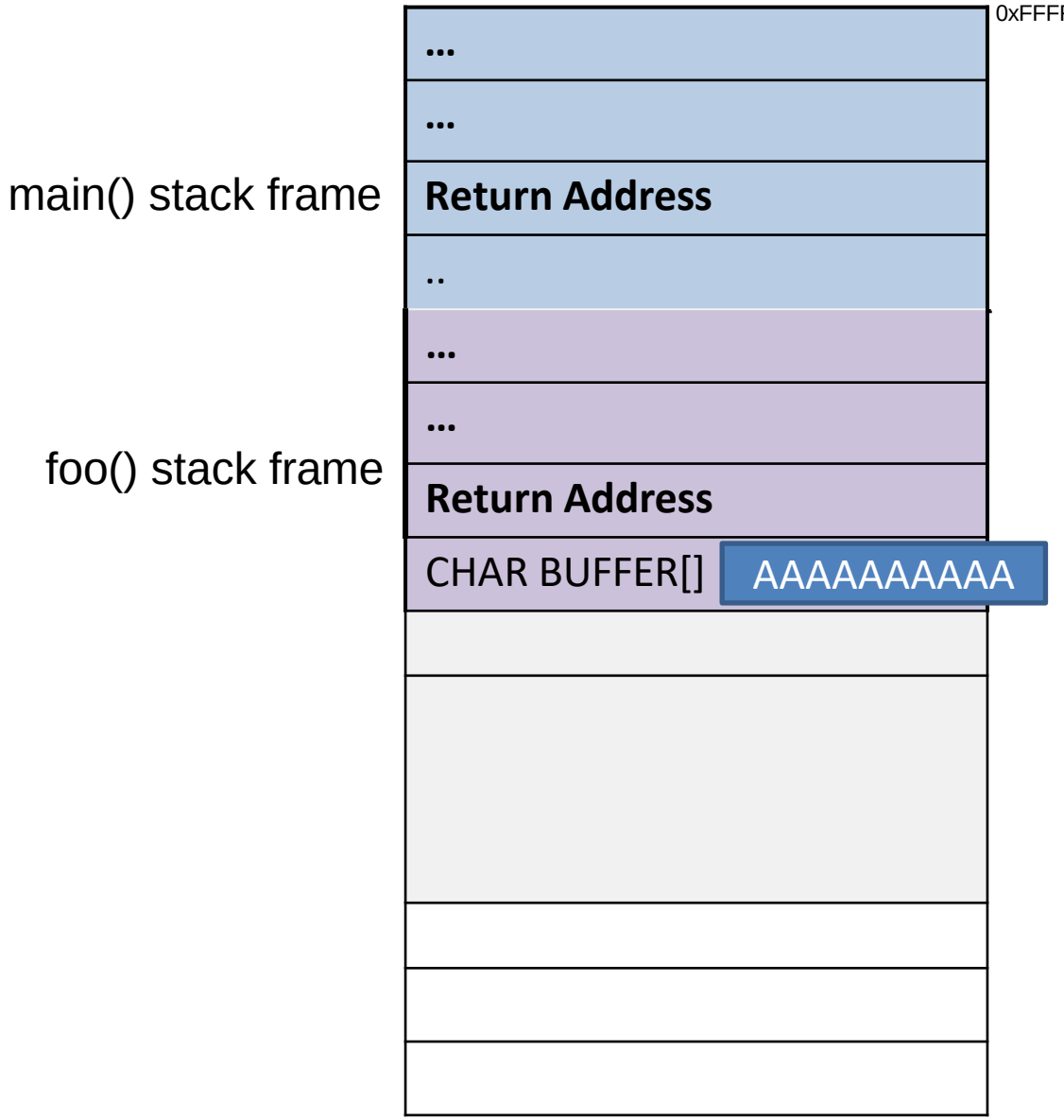


`buffer[]` can only hold 10 characters, right?

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
        char buffer[10];
        strcpy(buffer, str);

}


int main(int argc, char *argv[])
{
        foo(argv[1]);
        printf("Returned Properly\n");
        return 0;

}
```

C doesn't care.

The Stack



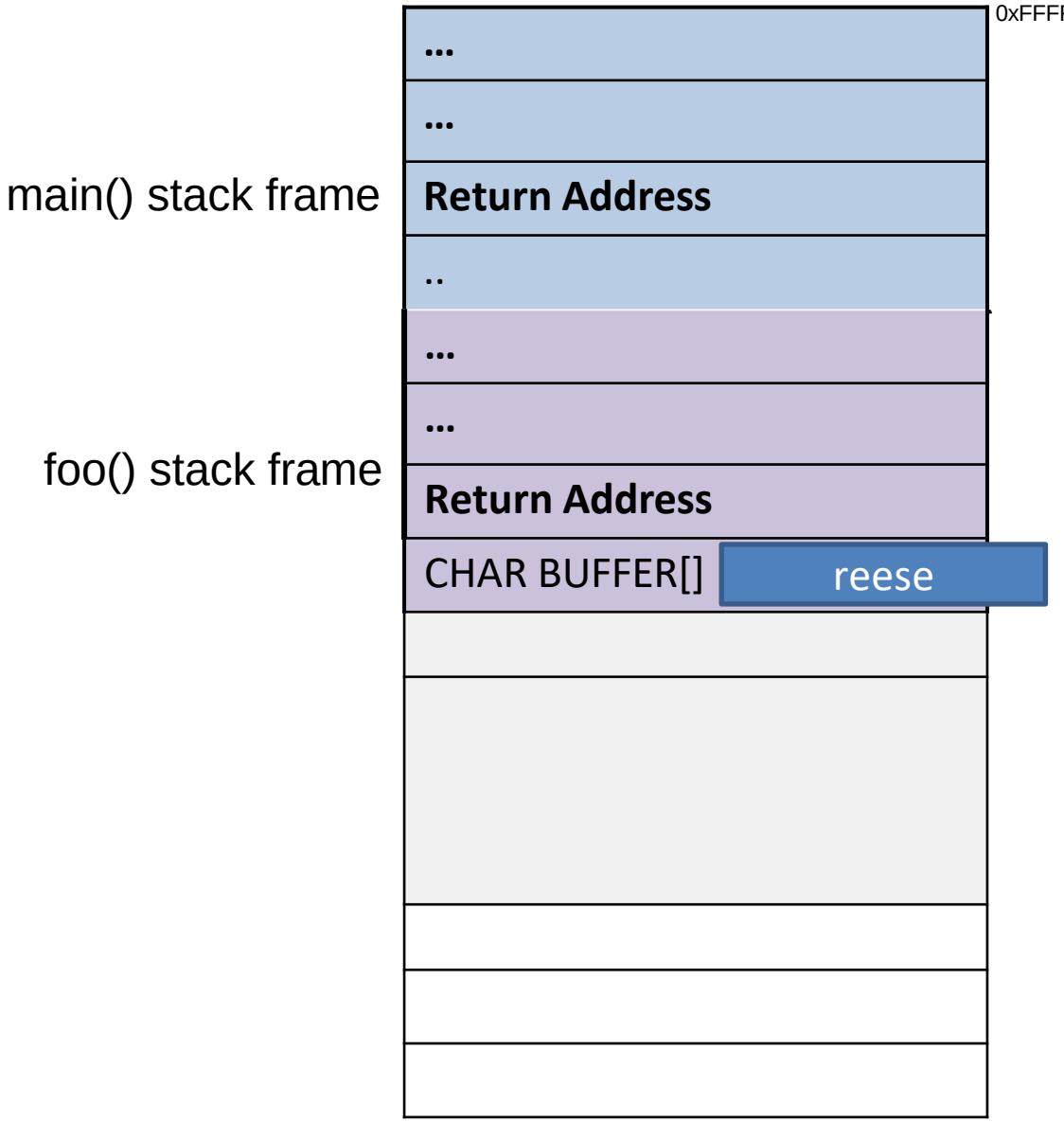
```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
        char buffer[10];
        strcpy(buffer, str);

}


int main(int argc, char *argv[])
{
        foo(argv[1]);
        printf("Returned Properly\n");
        return 0;

}
```

C doesn't care.

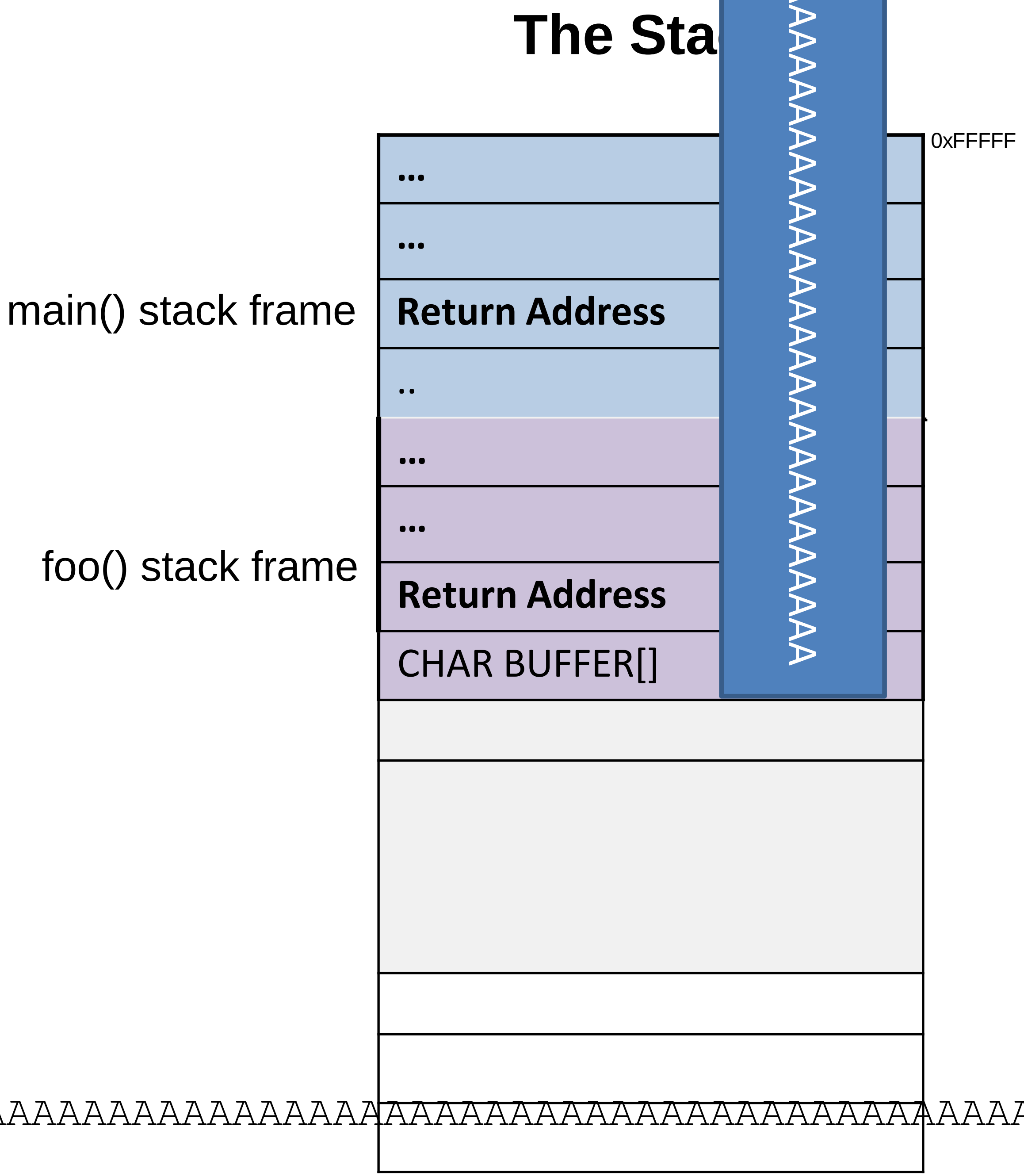
Instead of `./myprogram reese`

What if we did…..

The Stack



```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
        char buffer[10];
        strcpy(buffer, str);

}


int main(int argc, char *argv[])
{
        foo(argv[1]);
        printf("Returned Properly\n");
        return 0;

}
```



C doesn't care.

Instead of `./myprogram reese`

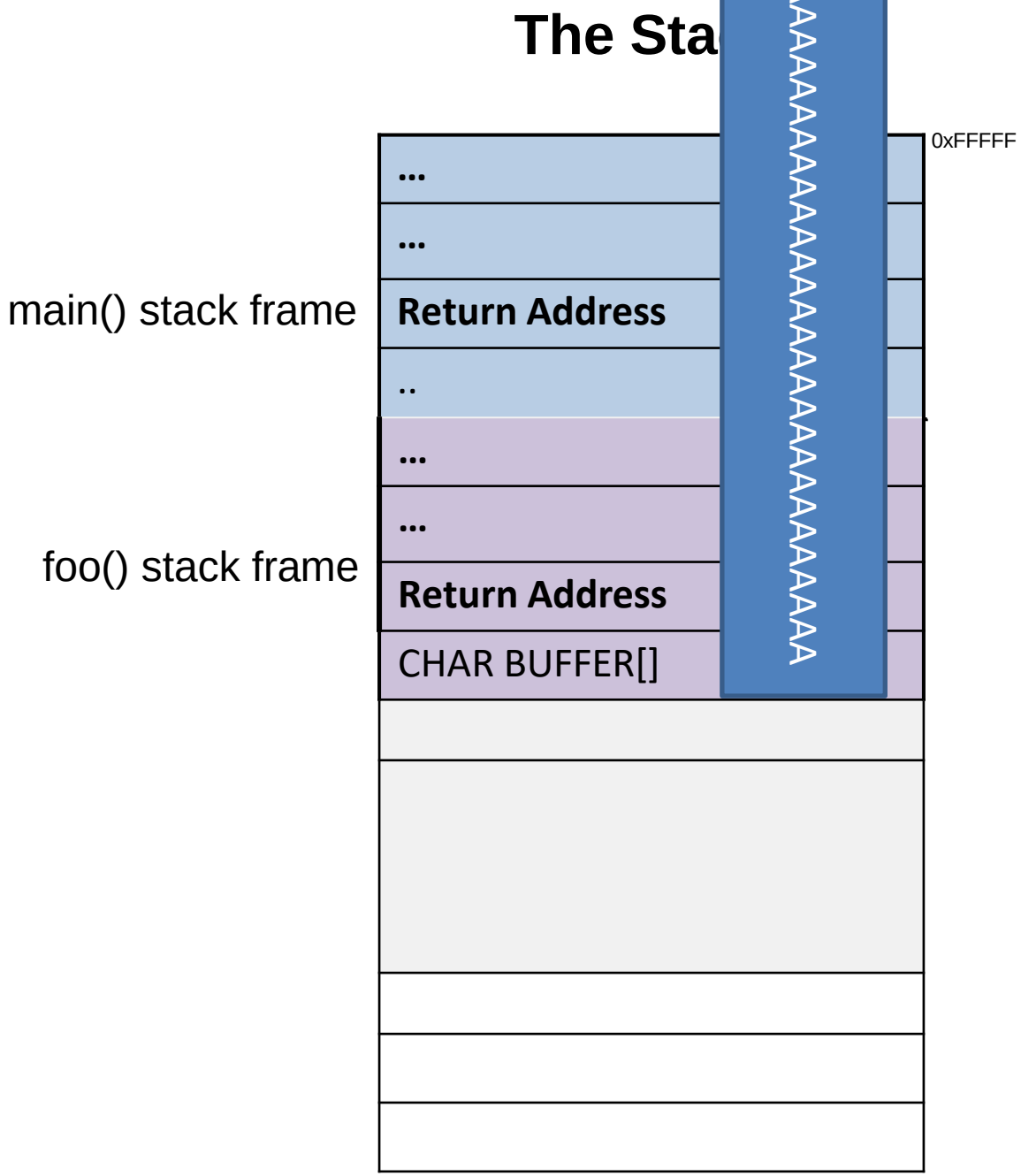
What if we did…..

`./myprogram AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`

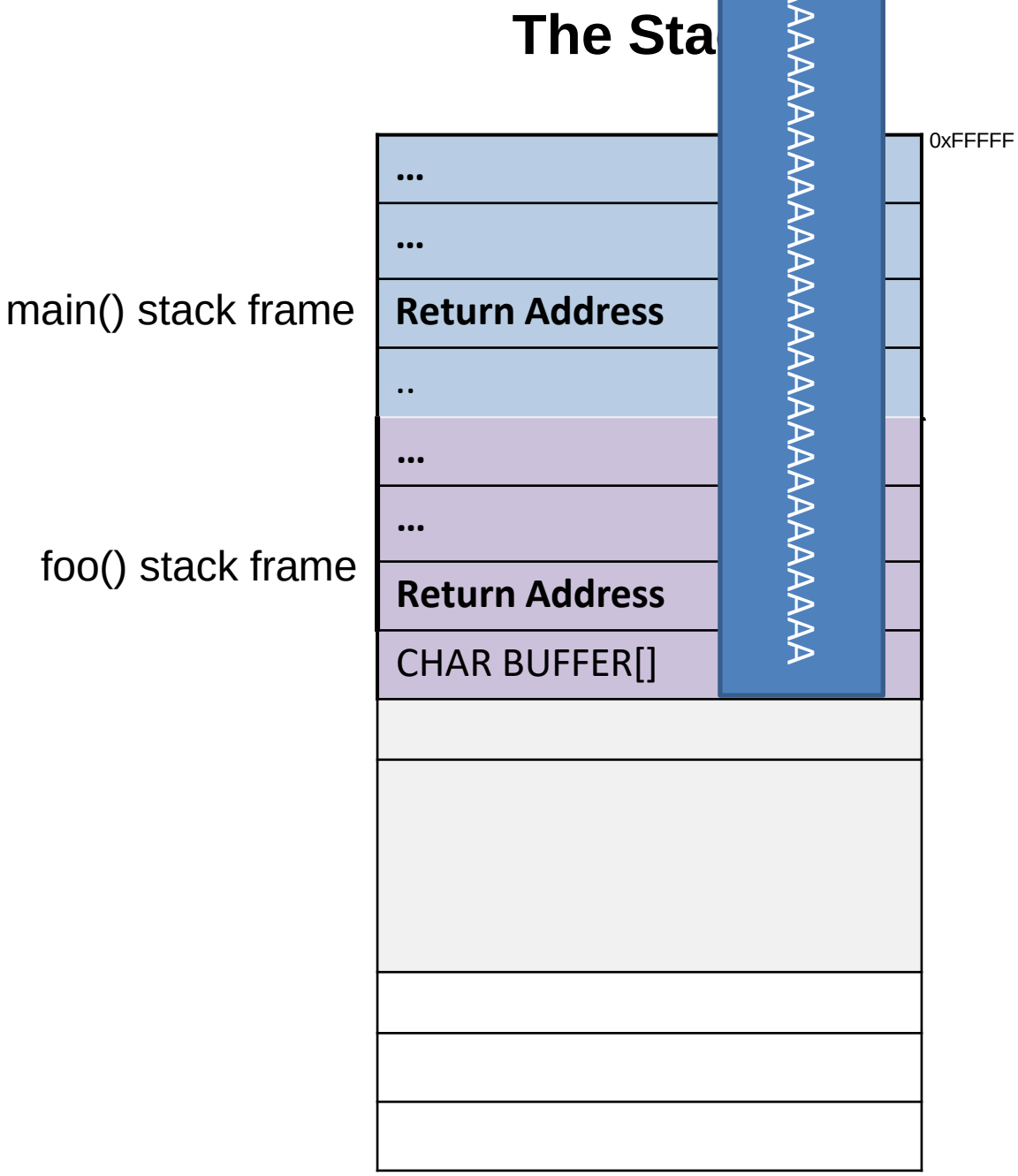
```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
        char buffer[10];
        strcpy(buffer, str);

}


int main(int argc, char *argv[])
{
        foo(argv[1]);
        printf("Returned Properly\n");
        return 0;

}
```

We can **overflow** this buffer!

This will **overwrite** other values on the Stack



```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
        char buffer[10];
        strcpy(buffer, str);

}


int main(int argc, char *argv[])
{
        foo(argv[1]);
        printf("Returned Properly\n");
        return 0;

}
```

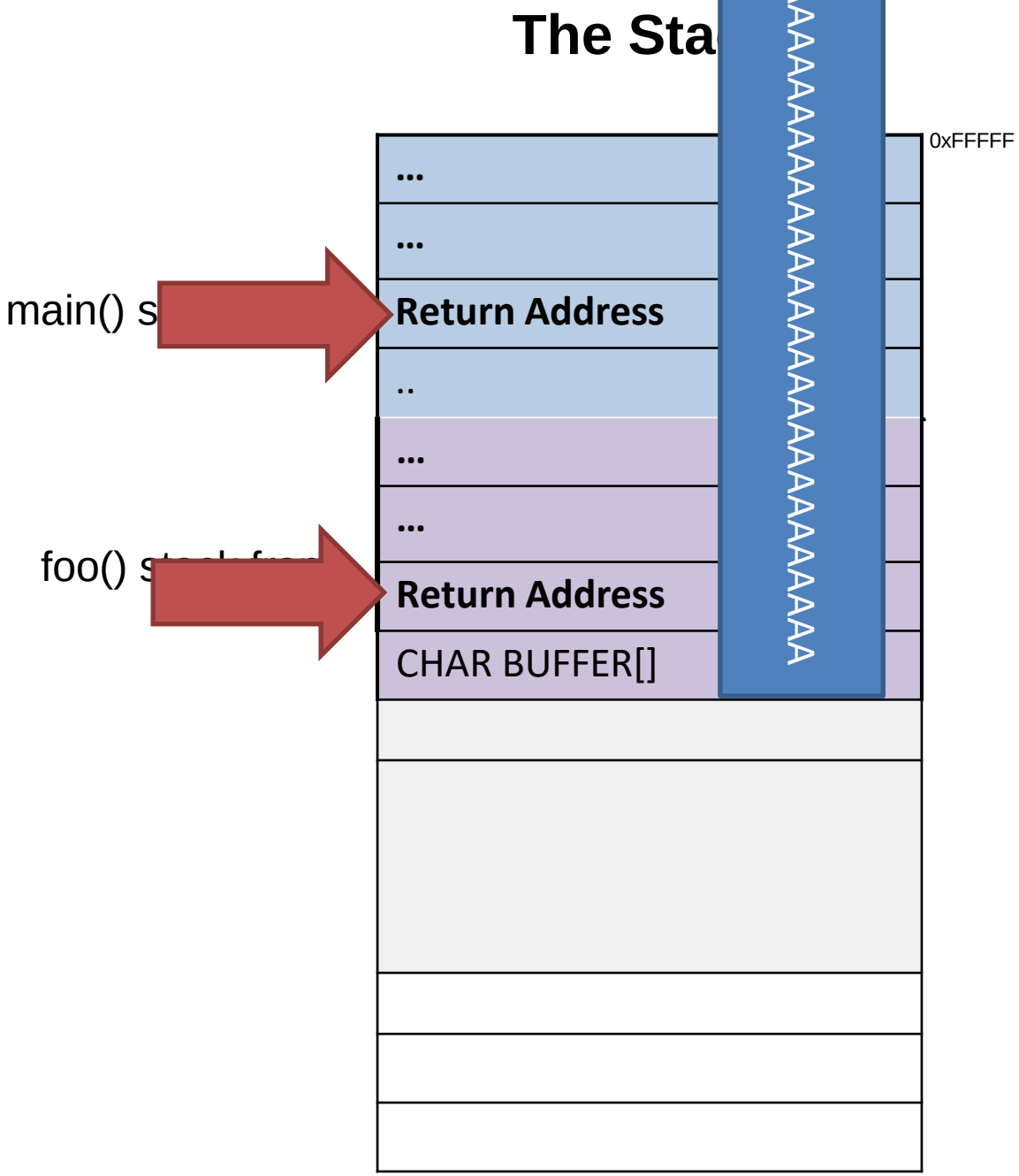


We can **overflow** this buffer!

This will **overwrite** other values on the Stack

What can our input control ?

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
        char buffer[10];
        strcpy(buffer, str);

}


int main(int argc, char *argv[])
{
        foo(argv[1]);
        printf("Returned Properly\n");
        return 0;

}
```



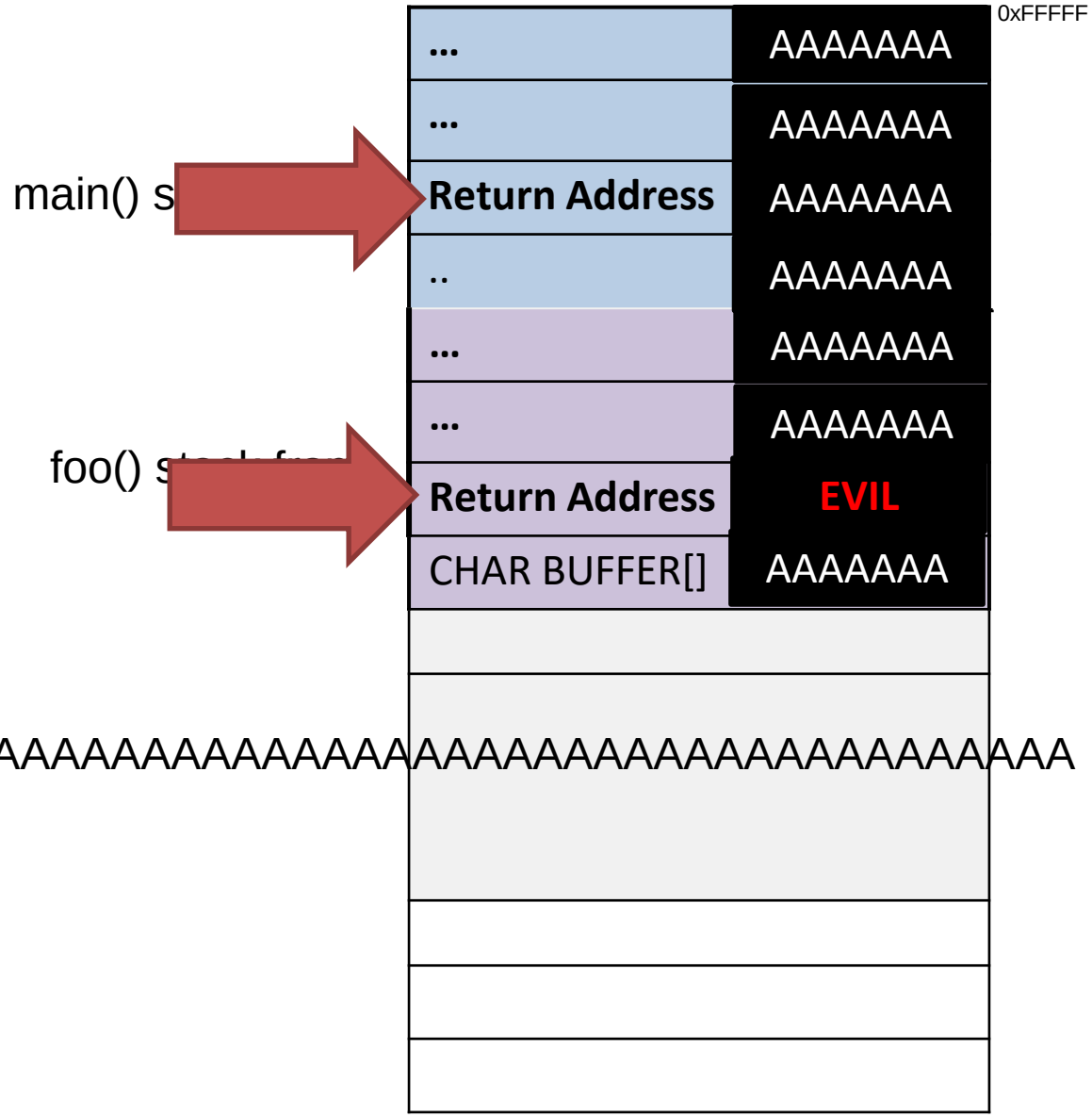
We can **overflow** this buffer!

This will **overwrite** other values on the Stack

Our input can overwrite values on the stack, specifically, the **return address**

The Stack

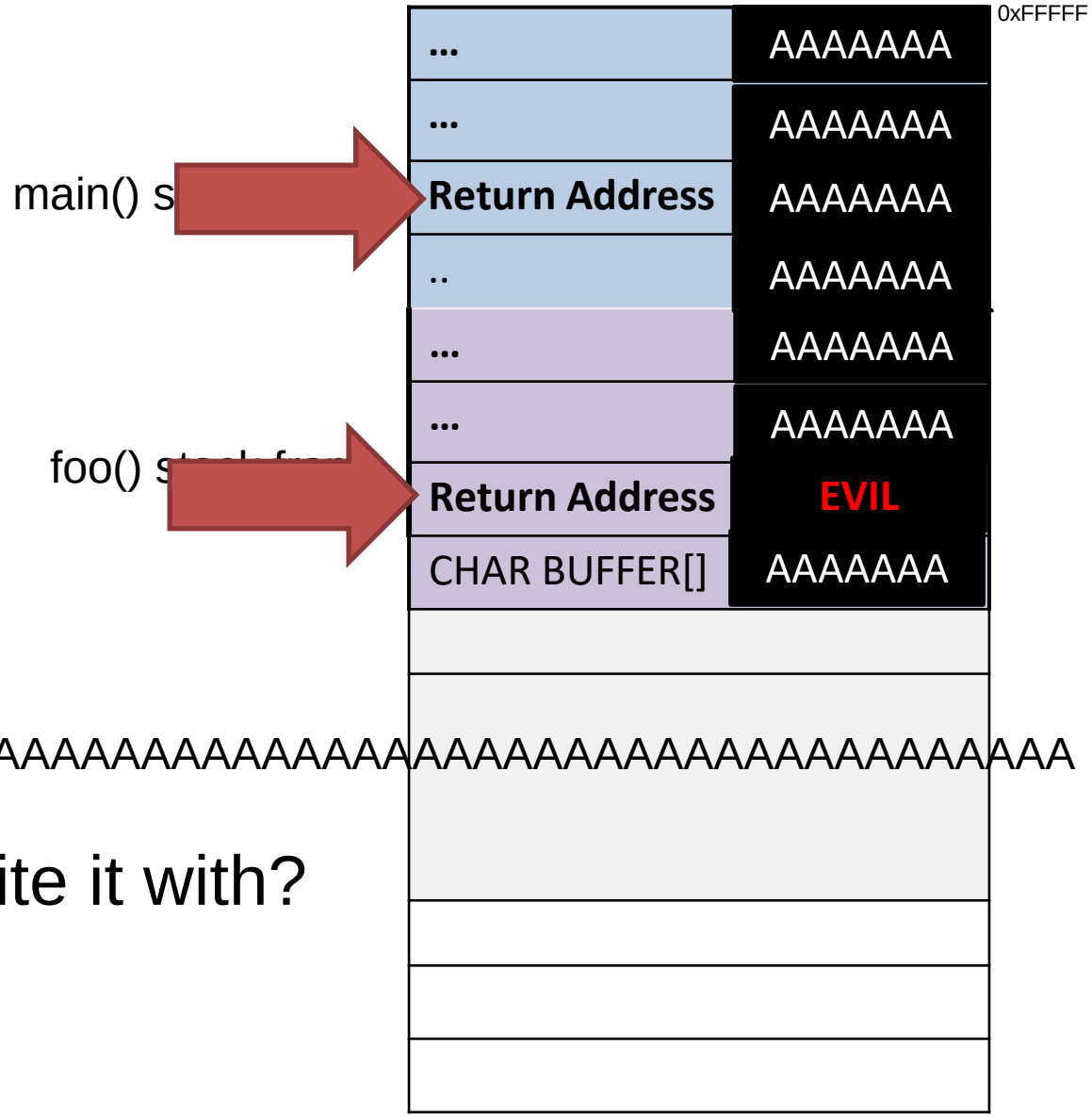
```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
        char buffer[10];
        strcpy(buffer, str);

}


int main(int argc, char *argv[])
{
        foo(argv[1]);
        printf("Returned Properly\n");
        return 0;

}
```



./myprogram AAAAAAAA**EVIL**AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

The Stack

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

void foo(char *str)
{
        char buffer[10];
        strcpy(buffer, str);

}


int main(int argc, char *argv[])
{
        foo(argv[1]);
        printf("Returned Properly\n");
        return 0;

}
```

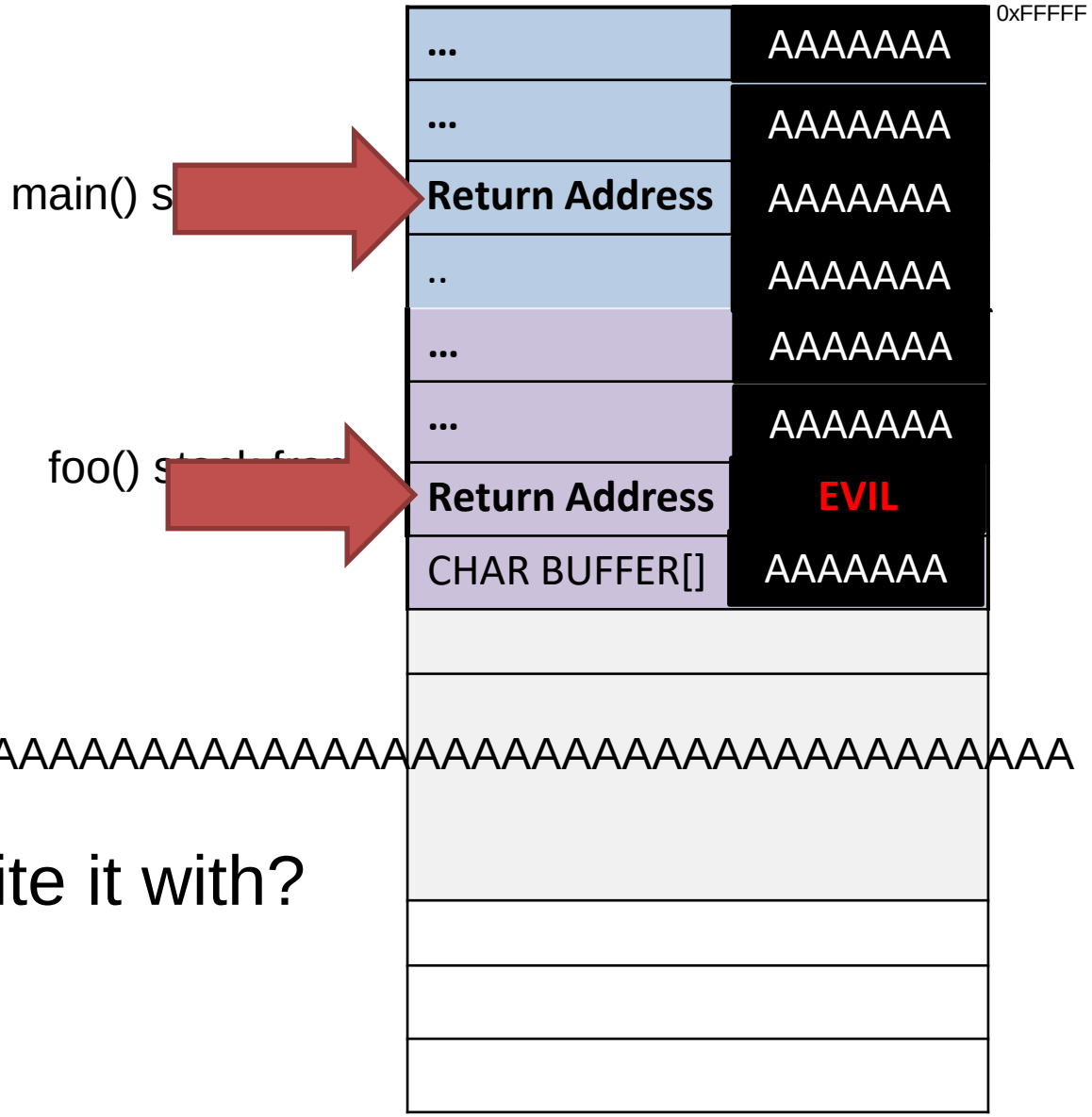



```
./myprogram AAAAAAAAEVILAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Instead of **EVIL**, what could we overwrite it with?

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
        char buffer[10];
        strcpy(buffer, str);

}


int main(int argc, char *argv[])
{
        foo(argv[1]);
        printf("Returned Properly\n");
        return 0;

}
```

The Stack

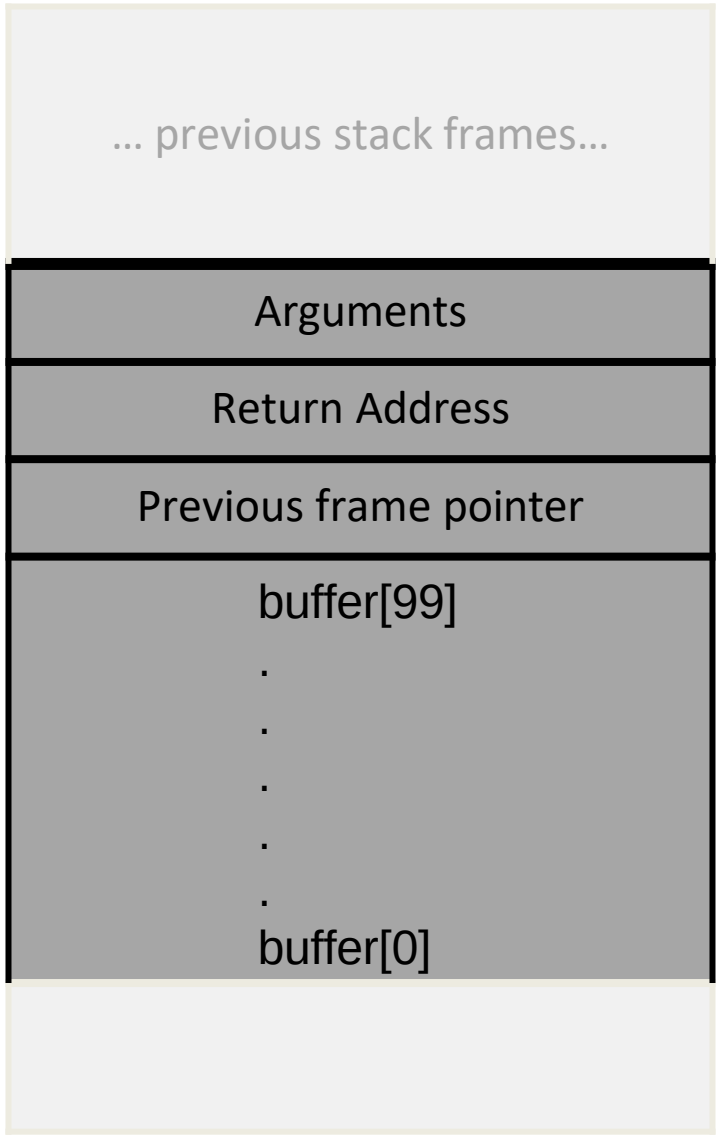


./myprogram AAAAAAAA**EVIL**AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Instead of **EVIL**, what could we overwrite it with?

Our own malicious code

THE STACK

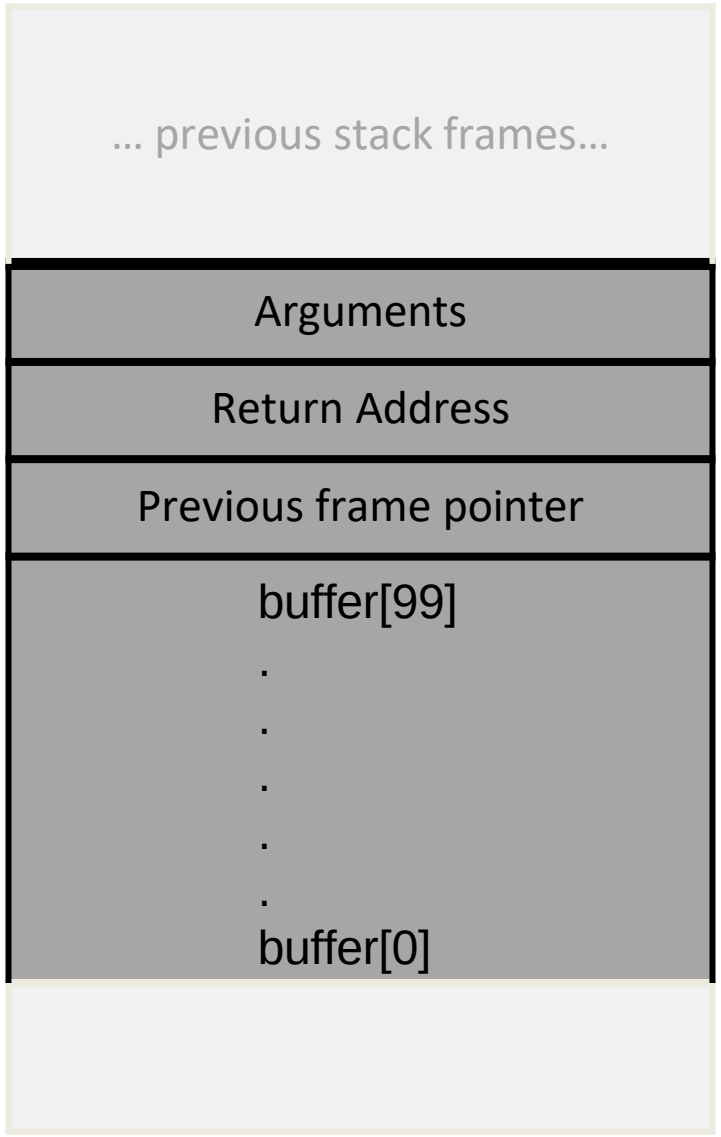


The CPU needs to keep track of two things:

1. The location of the top of stack

2. The location of the current stack frame we are executing

THE STACK

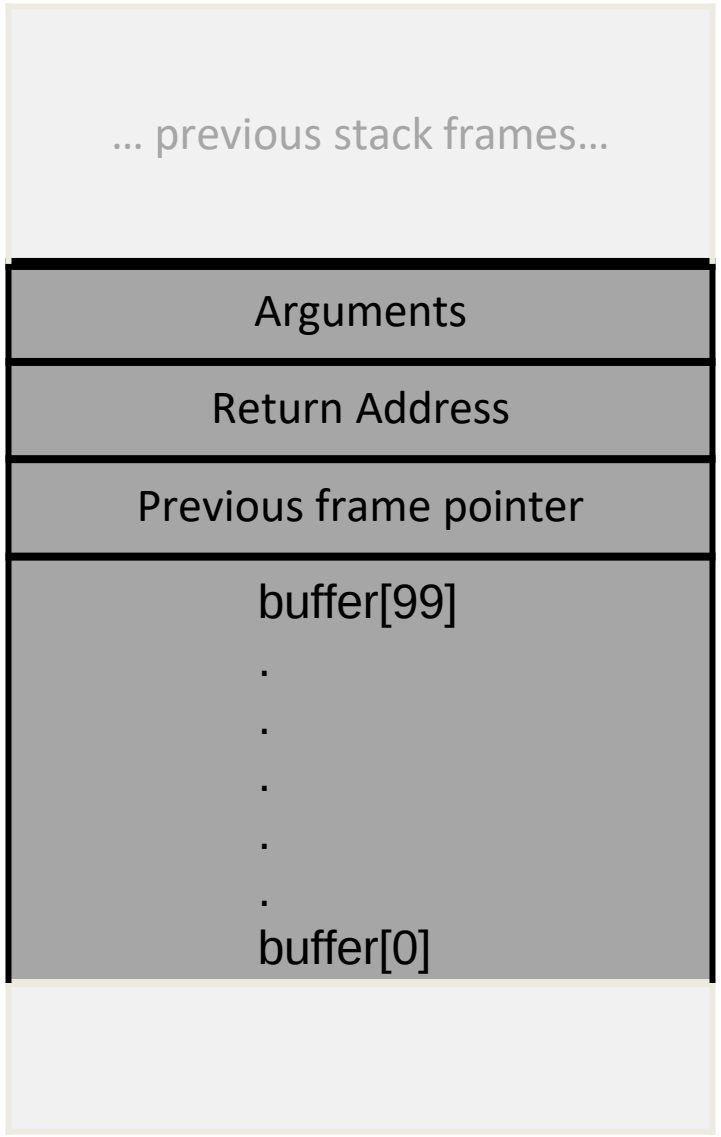


The CPU needs to keep track of two things:

1. The location of the top of stack

2. The location of the current stack frame we are executing

THE STACK

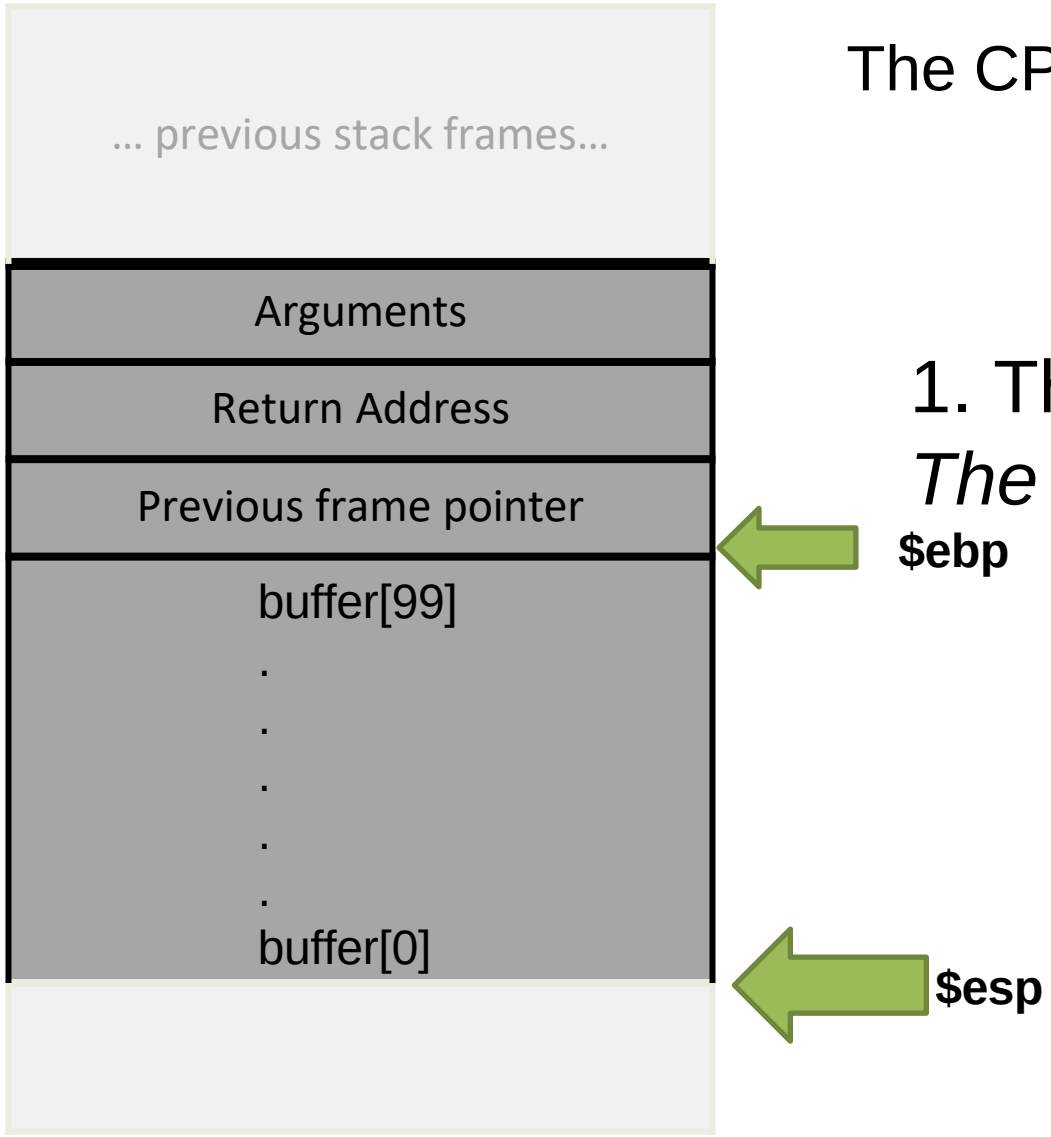


The CPU needs to keep track of two things:

1. The location of the top of stack
*The register **$esp** points to the top of the **s**tack*

2. The location of the current stack frame we are executing

THE STACK



The CPU needs to keep track of two things:

1. The location of the top of stack

*The register **$esp** points to the top of the **s**tack*

2. The location of the current stack frame we are executing

*The register **$ebp** points to the **b**ase of the current stack frame*

THE STACK

Every time a function is called, the **function prologue** occurs

$ ebp

… previous stack frames…

$ esp

```
void main()
{
    foo(2,3);
    return 0;
}
```

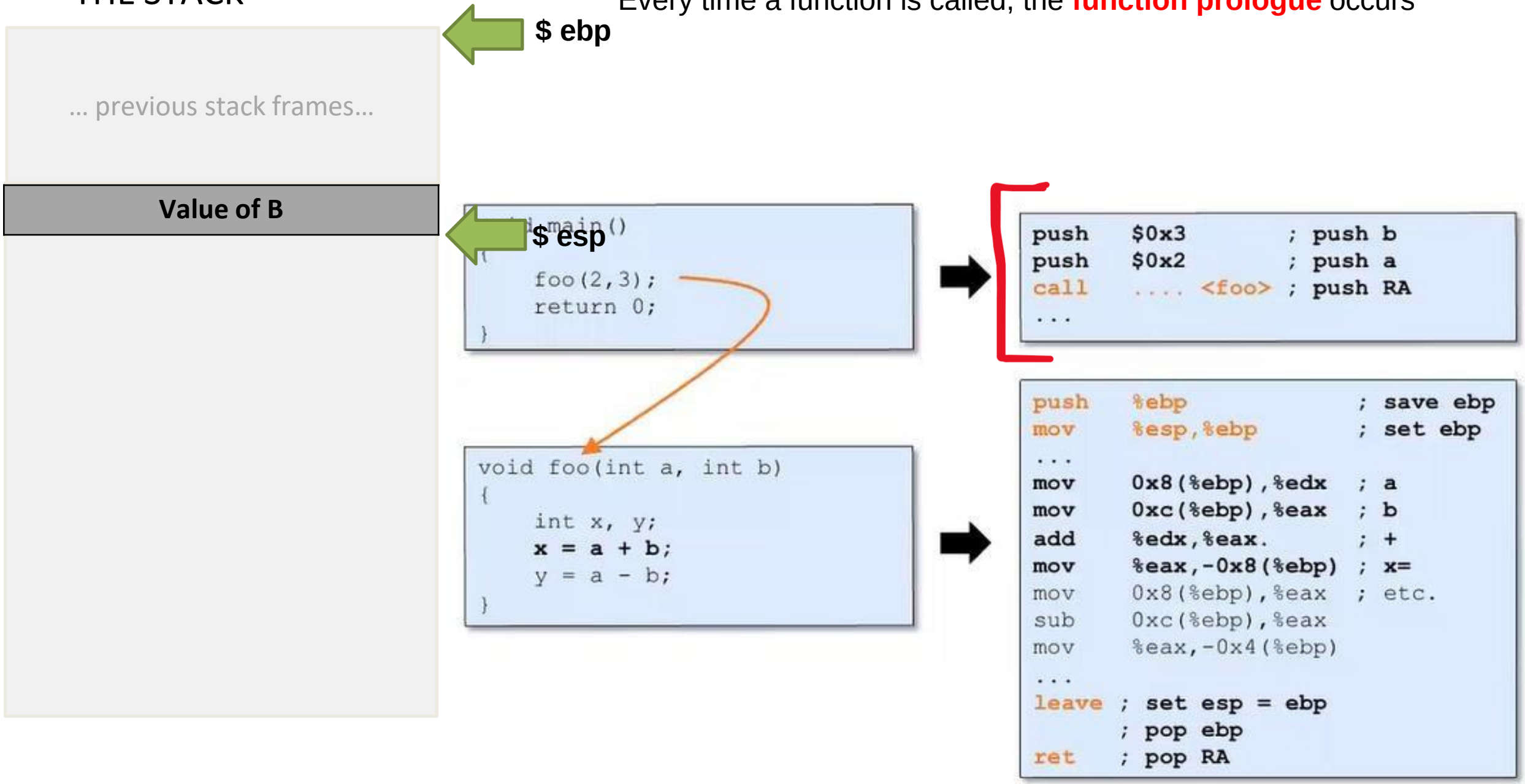
```
push    $0x3          ; push b
push    $0x2          ; push a
call    .... <foo>    ; push RA
...
```

```
void foo(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```

```
push    %ebp                ; save ebp
mov     %esp,%ebp           ; set ebp
...
mov     0x8(%ebp),%edx      ; a
mov     0xc(%ebp),%eax      ; b
add     %edx,%eax.          ; +
mov     %eax,-0x8(%ebp)     ; x=
mov     0x8(%ebp),%eax      ; etc.
sub     0xc(%ebp),%eax
mov     %eax,-0x4(%ebp)
...
leave ; set esp = ebp
      ; pop ebp
ret   ; pop RA
```

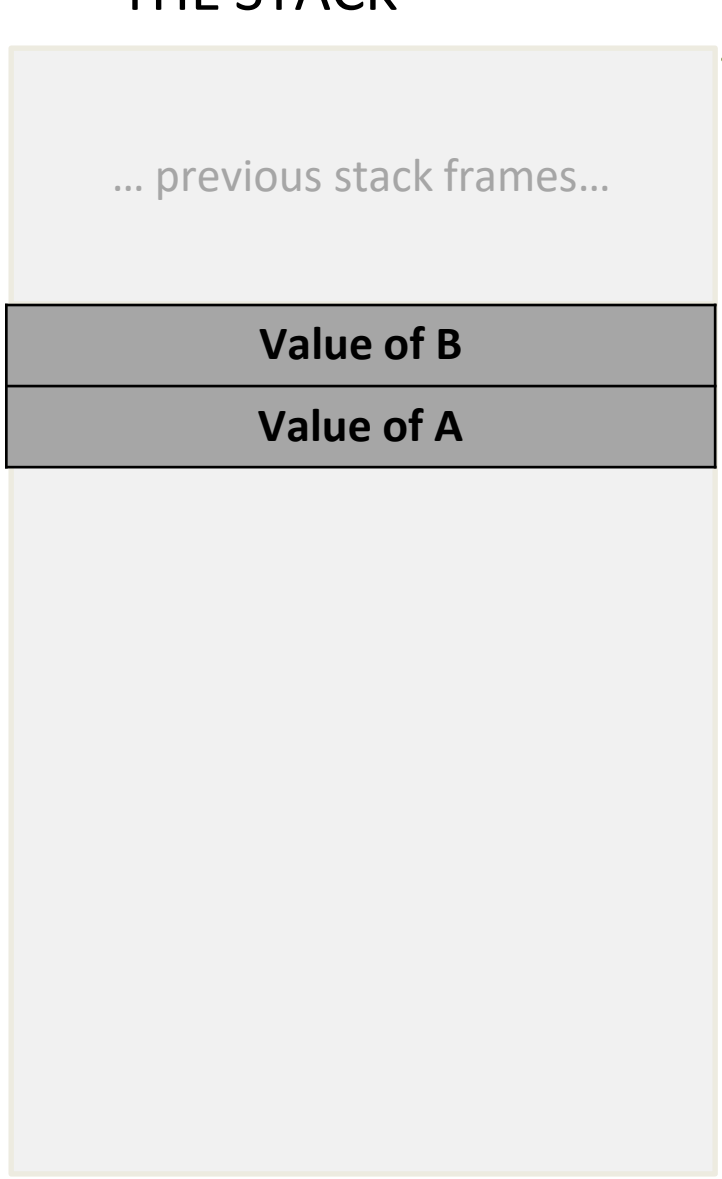
THE STACK

Every time a function is called, the **function prologue** occurs



THE STACK

Every time a function is called, the **function prologue** occurs




```
void main()
{
    foo(2,3);
    return 0;
}
```

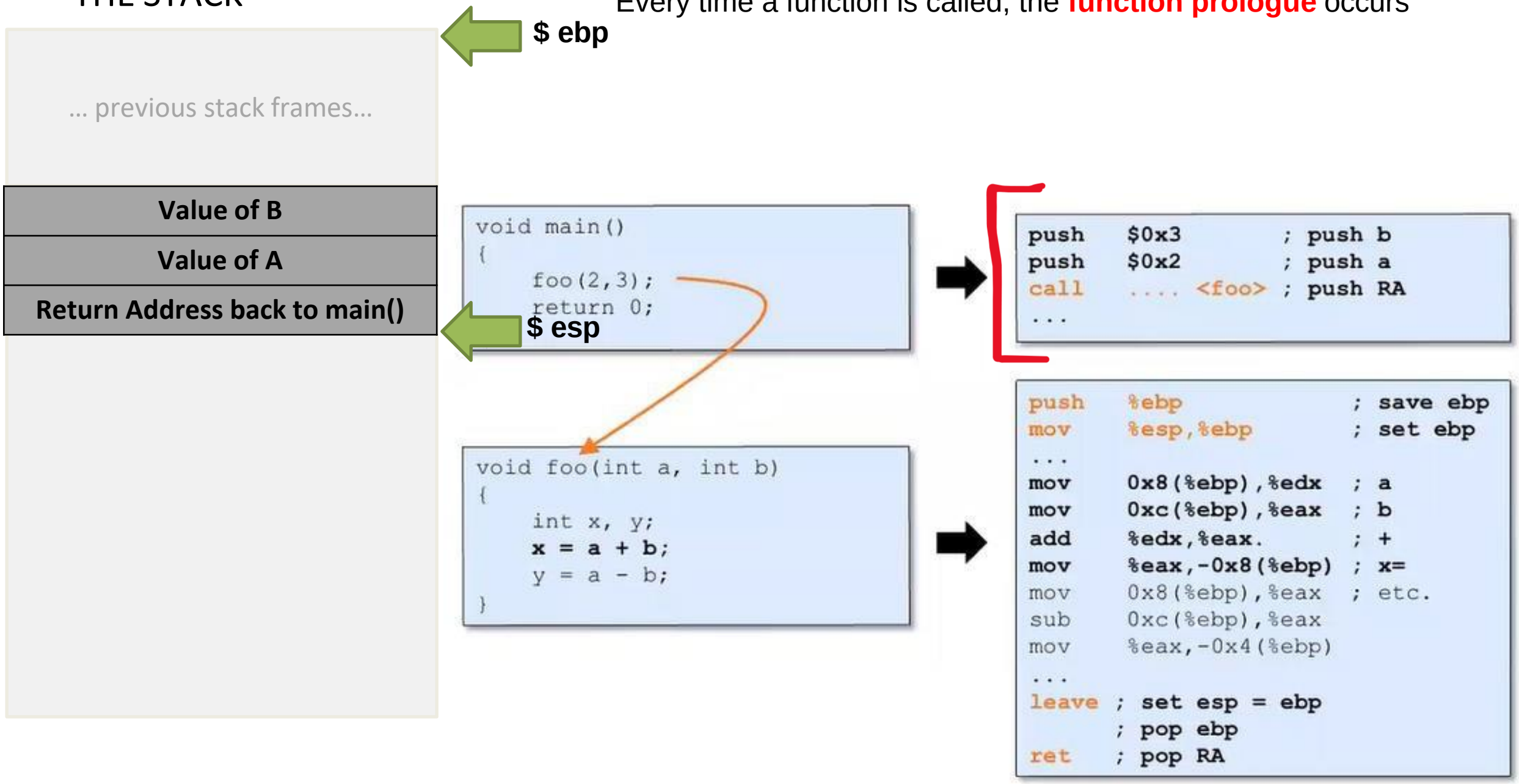
```
void foo(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```

```
push    $0x3            ; push b
push    $0x2            ; push a
call    .... <foo>      ; push RA
...
```

```
push    %ebp                    ; save ebp
mov     %esp,%ebp               ; set ebp
...
mov     0x8(%ebp),%edx          ; a
mov     0xc(%ebp),%eax          ; b
add     %edx,%eax.              ; +
mov     %eax,-0x8(%ebp)         ; x=
mov     0x8(%ebp),%eax          ; etc.
sub     0xc(%ebp),%eax
mov     %eax,-0x4(%ebp)
...
leave   ; set esp = ebp
        ; pop ebp
ret     ; pop RA
```

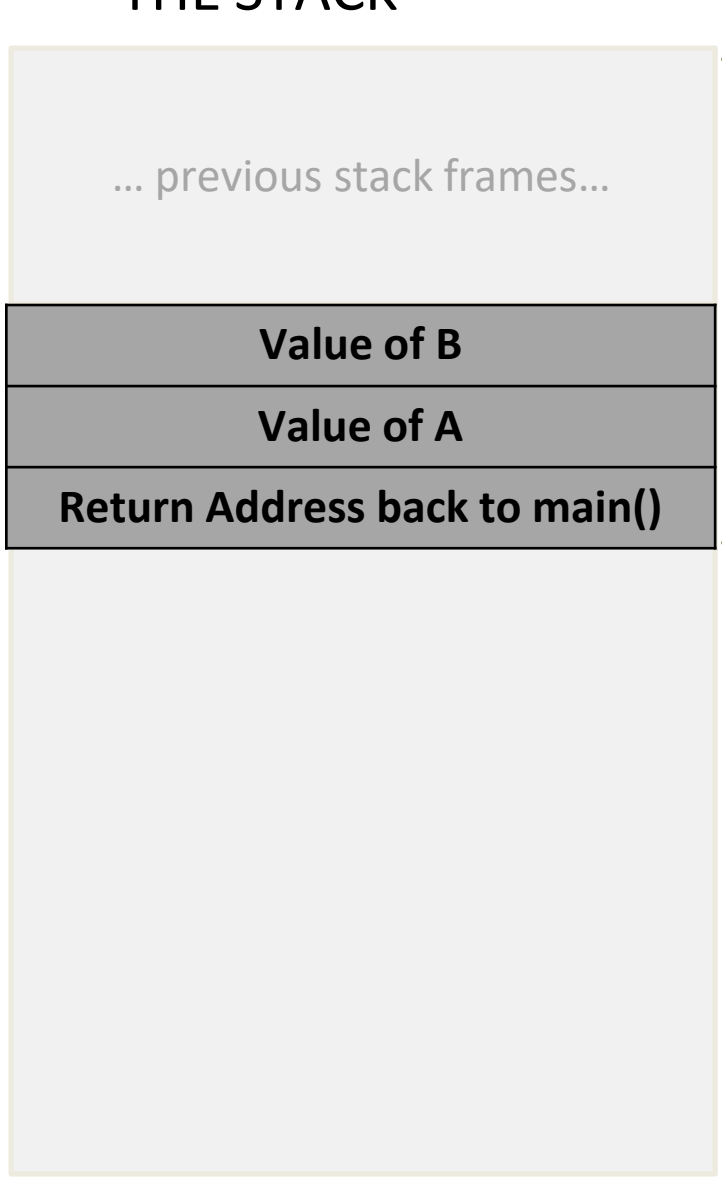
THE STACK

Every time a function is called, the **function prologue** occurs



THE STACK

Every time a function is called, the **function prologue** occurs




```
void main()
{
    foo(2,3);
    return 0;
}
```

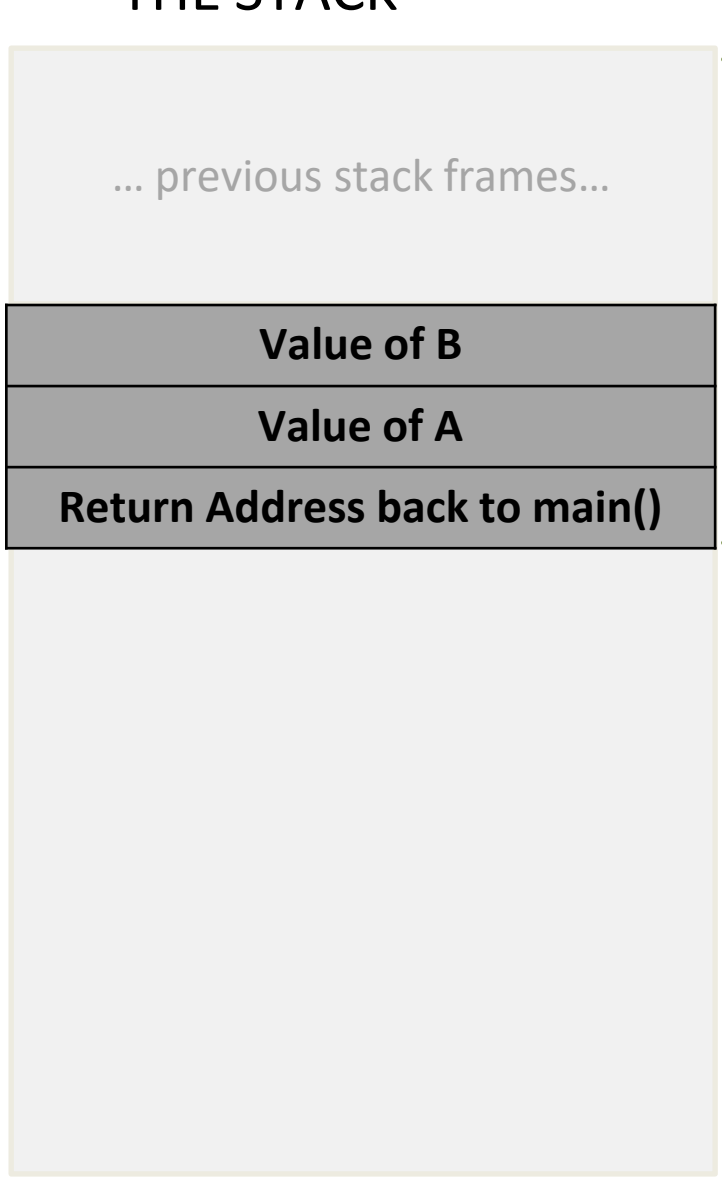
```
void foo(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```

```
push    $0x3          ; push b
push    $0x2          ; push a
call    .... <foo>    ; push RA
...
```

```
push    %ebp                ; save ebp
mov     %esp,%ebp           ; set ebp
...
mov     0x8(%ebp),%edx      ; a
mov     0xc(%ebp),%eax      ; b
add     %edx,%eax.          ; +
mov     %eax,-0x8(%ebp)     ; x=
mov     0x8(%ebp),%eax      ; etc.
sub     0xc(%ebp),%eax
mov     %eax,-0x4(%ebp)
...
leave ; set esp = ebp
      ; pop ebp
ret   ; pop RA
```

THE STACK

Every time a function is called, the **function prologue** occurs




```
void main()
{
    foo(2,3);
    return 0;
}
```

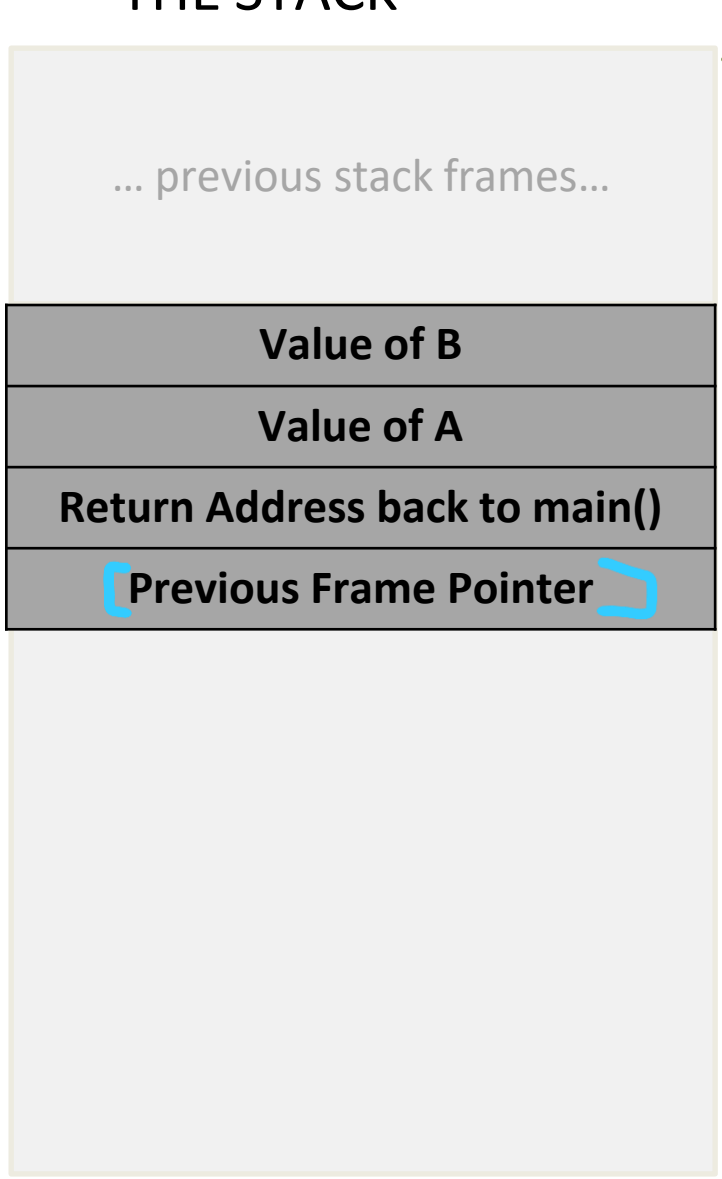
```
push   $0x3          ; push b
push   $0x2          ; push a
call   .... <foo>    ; push RA
...
```

```
void foo(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```

```
push   %ebp                ; save ebp
mov    %esp,%ebp           ; set ebp
...
mov    0x8(%ebp),%edx      ; a
mov    0xc(%ebp),%eax      ; b
add    %edx,%eax.          ; +
mov    %eax,-0x8(%ebp)     ; x=
mov    0x8(%ebp),%eax      ; etc.
sub    0xc(%ebp),%eax
mov    %eax,-0x4(%ebp)
...
leave  ; set esp = ebp
       ; pop ebp
ret    ; pop RA
```

THE STACK

Every time a function is called, the **function prologue** occurs




```
void main()
{
    foo(2,3);
    return 0;
}
```

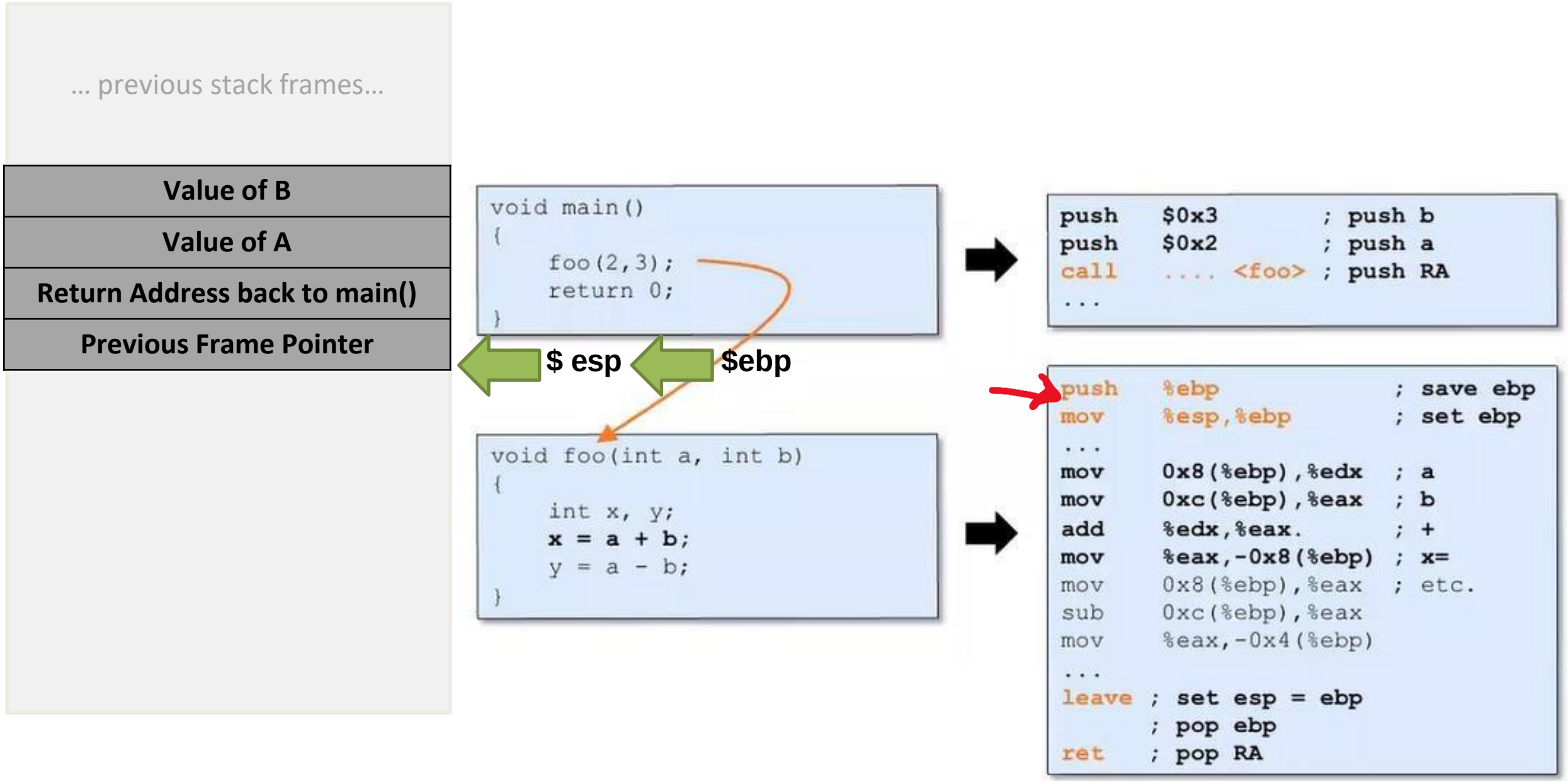
```
push    $0x3          ; push b
push    $0x2          ; push a
call    .... <foo>    ; push RA
...
```

```
void foo(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```

```
push    %ebp                ; save ebp
mov     %esp,%ebp           ; set ebp
...
mov     0x8(%ebp),%edx      ; a
mov     0xc(%ebp),%eax      ; b
add     %edx,%eax.          ; +
mov     %eax,-0x8(%ebp)     ; x=
mov     0x8(%ebp),%eax      ; etc.
sub     0xc(%ebp),%eax
mov     %eax,-0x4(%ebp)
...
leave ; set esp = ebp
      ; pop ebp
ret   ; pop RA
```

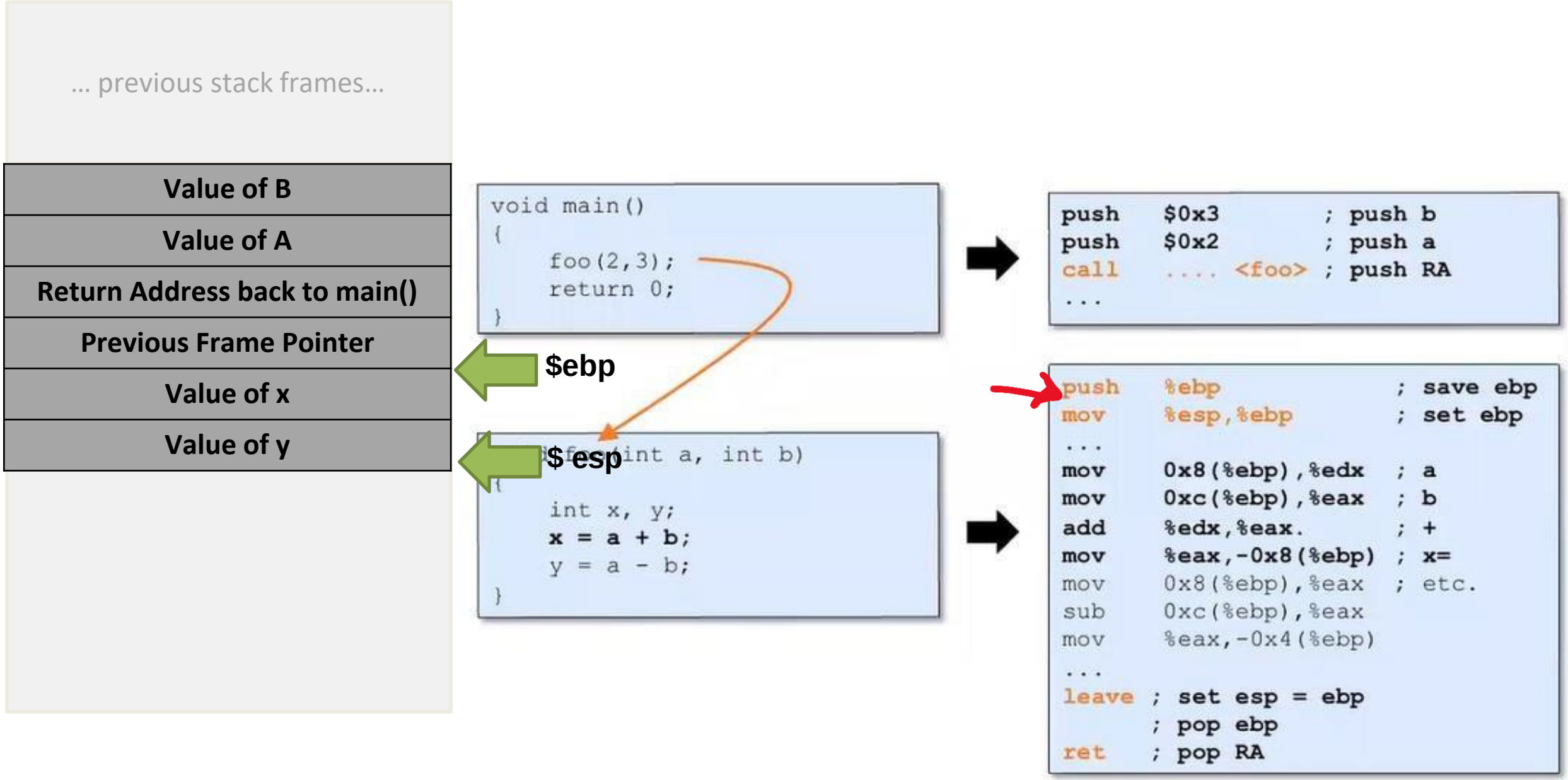
THE STACK

Every time a function is called, the **function prologue** occurs



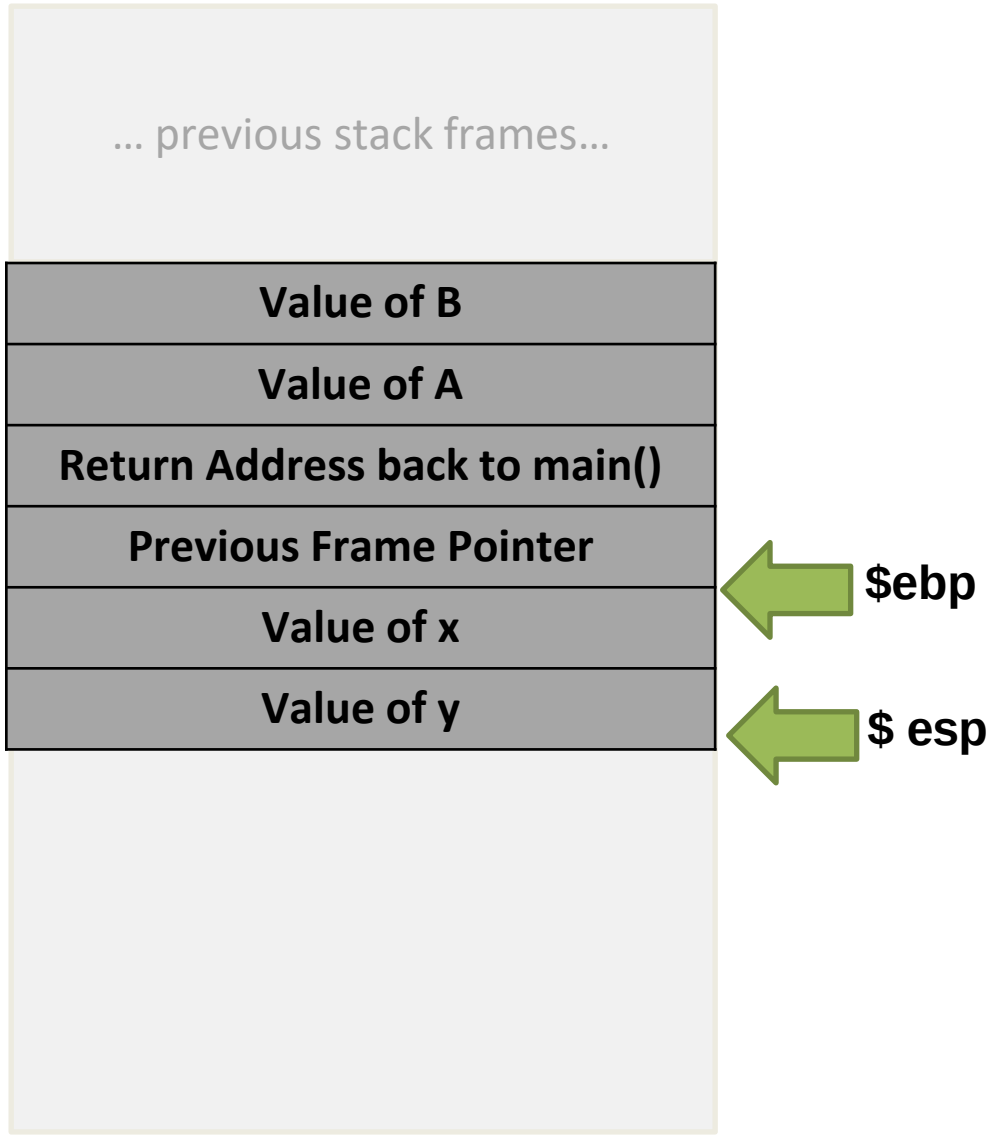
THE STACK

Every time a function is called, the **function prologue** occurs



THE STACK

Every time a function is called, the **function prologue** occurs

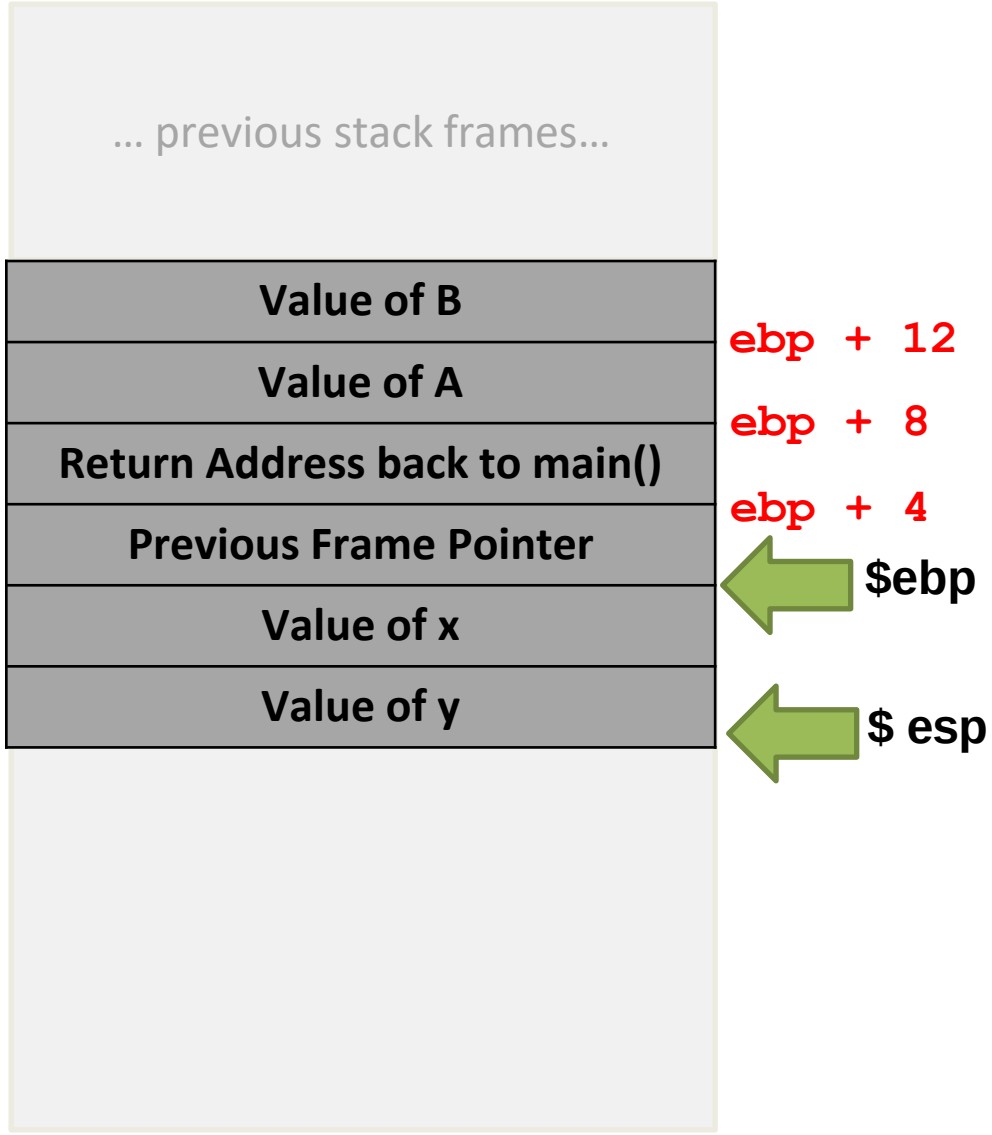


Why is this helpful knowledge?

This tells us how the return address in put onto the stack, and how these important pointers are managed

THE STACK

Every time a function is called, the **function prologue** occurs

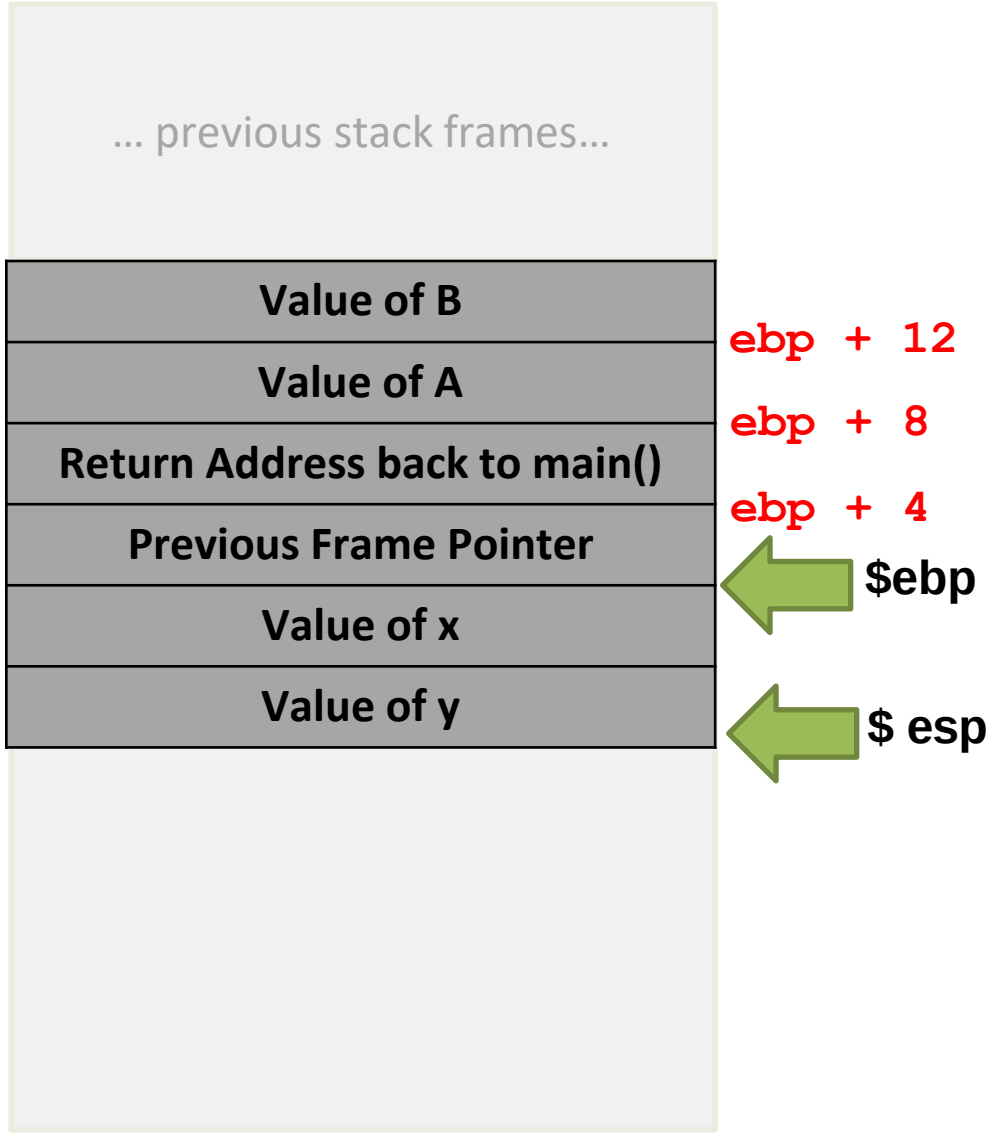


Why is this helpful knowledge?

This tells us how the return address in put onto the stack, and how these important pointers are managed

THE STACK

Every time a function is called, the **function prologue** occurs



Why is this helpful knowledge?

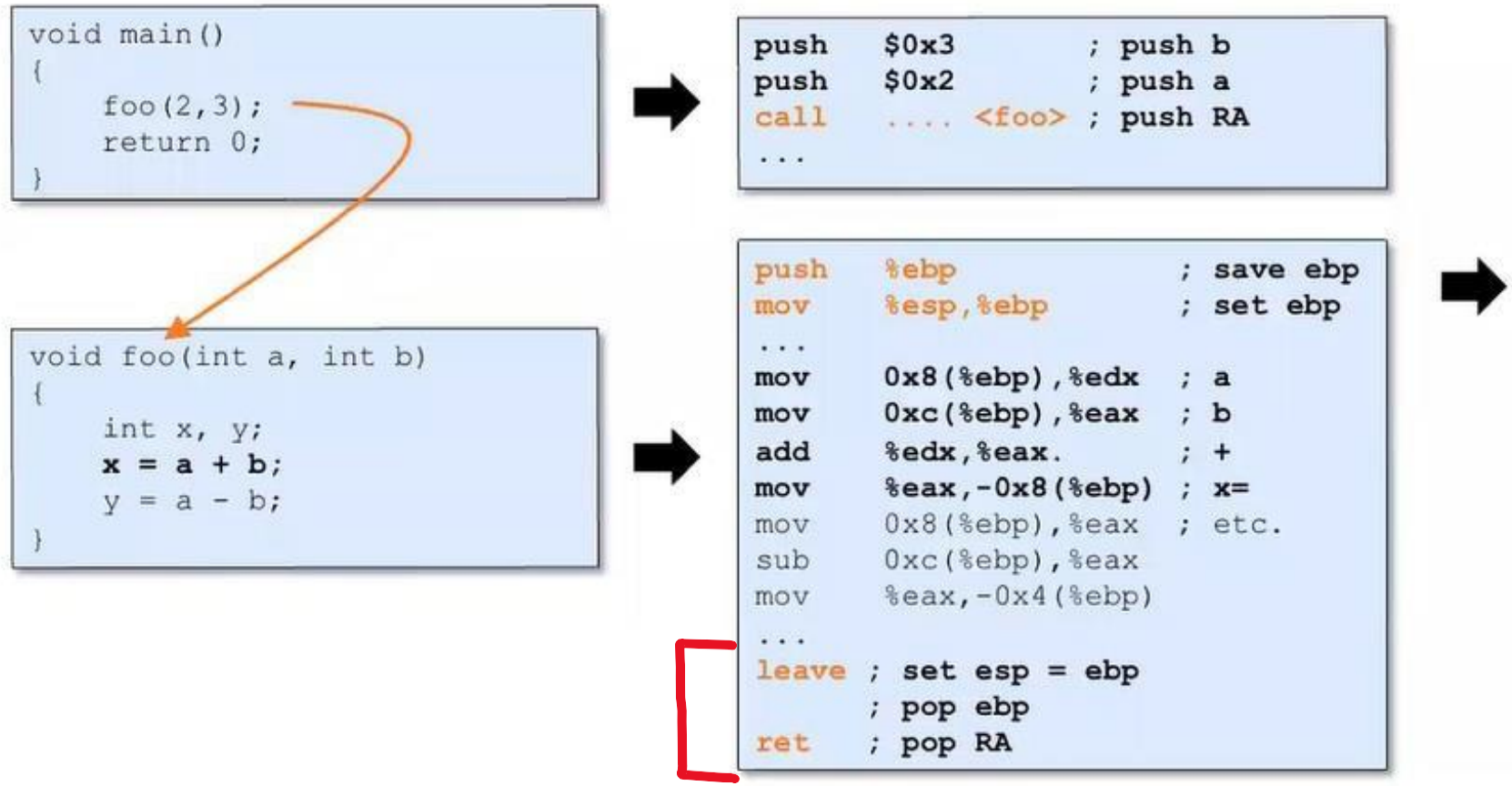
This tells us how the return address in put onto the stack, and how these important pointers are managed

THE STACK

… previous stack frames…

Every time a function is called, the **function prologue** occurs

When a function finishes, a **function epilogue** occurs and cleans up the stack



```c
int bof(char *str)
{
    char buffer[BUF_SIZE];

    // potential buffer overflow!
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

```c
int bof(char *str)
{
    char buffer[BUF_SIZE];

    // potential buffer overflow!
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

Reads (up to) 517 bytes of data from **`badfile`**

```c
int bof(char *str)
{
    char buffer[BUF_SIZE];

    // potential buffer overflow!
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

Reads (up to) 517 bytes of data from **`badfile`**

Storing the file contents into a str variable of size 517 bytes

```c
int bof(char *str)
{
    char buffer[BUF_SIZE];

    // potential buffer overflow!
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

Reads (up to) 517 bytes of data from **`badfile`**

Storing the file contents into a str variable of size 517 bytes

Calls the `dummy_function()` which calls `bof()`

```c
int bof(char *str)
{
    char buffer[BUF_SIZE];

    // potential buffer overflow!
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

`bof()` function uses `strcopy` to copy function argument into buffer

`BUF_SIZE = 100`

Reads (up to) 517 bytes of data from **`badfile`**

Storing the file contents into a str variable of size 517 bytes

Calls the `dummy_function()` which calls `bof()`

```c
int bof(char *str)
{
    char buffer[BUF_SIZE];

    // potential buffer overflow!
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

`bof()` function uses `strcopy` to copy function argument into buffer

`BUF_SIZE = 100`

There is no check if `str` is bigger than the `buffer`, so buffer overflow can occur!

Reads (up to) 517 bytes of data from **`badfile`**

Storing the file contents into a str variable of size 517 bytes

Calls the `dummy_function()` which calls `bof()`

```c
int bof(char *str)
{
    char buffer[BUF_SIZE];

    // potential buffer overflow!
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

`bof()` function uses `strcopy` to copy function argument into buffer

`BUF_SIZE = 100`

There is no check if `str` is bigger than the `buffer`, so buffer overflow can occur!

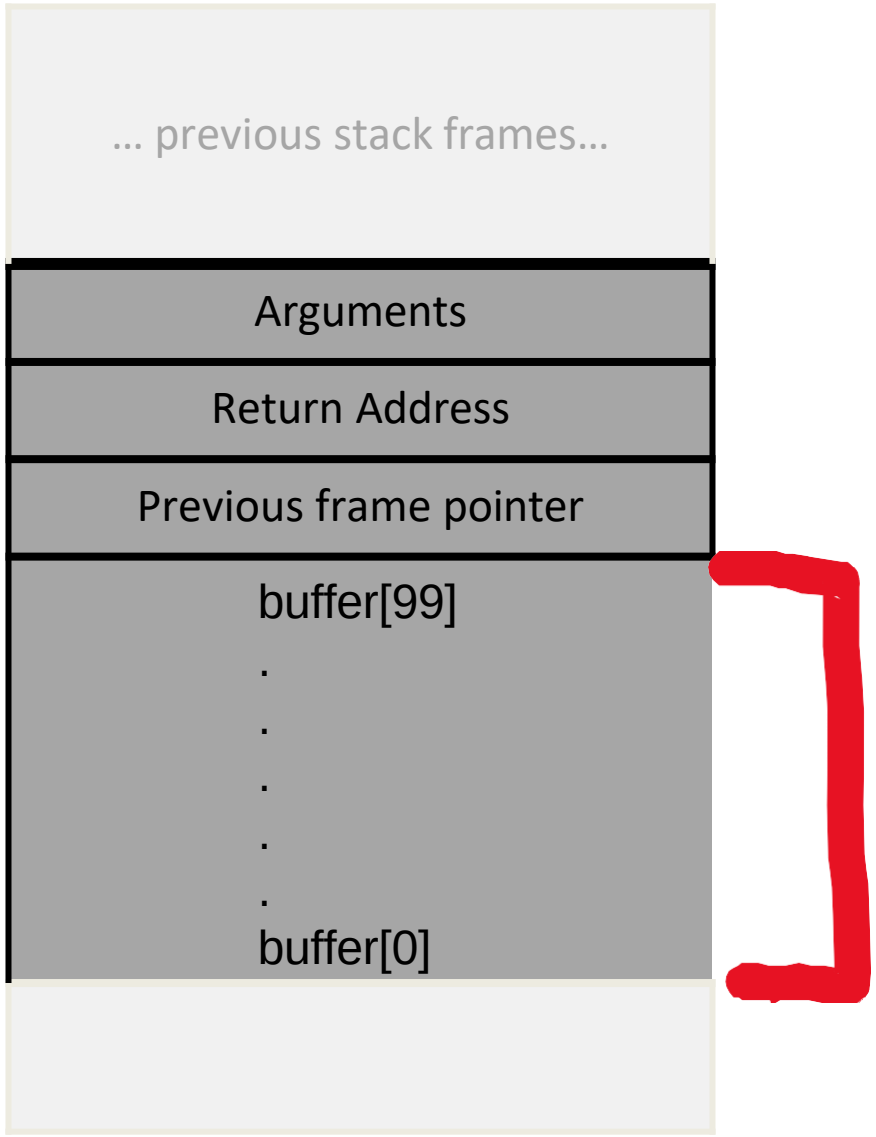
Reads (up to) 517 bytes of data from **`badfile`**

Storing the file contents into a str variable of size 517 bytes

Calls the `dummy_function()` which calls `bof()`

`buffer` is a stack variable, so we can overwrite other values on the stack with a buffer overflow!

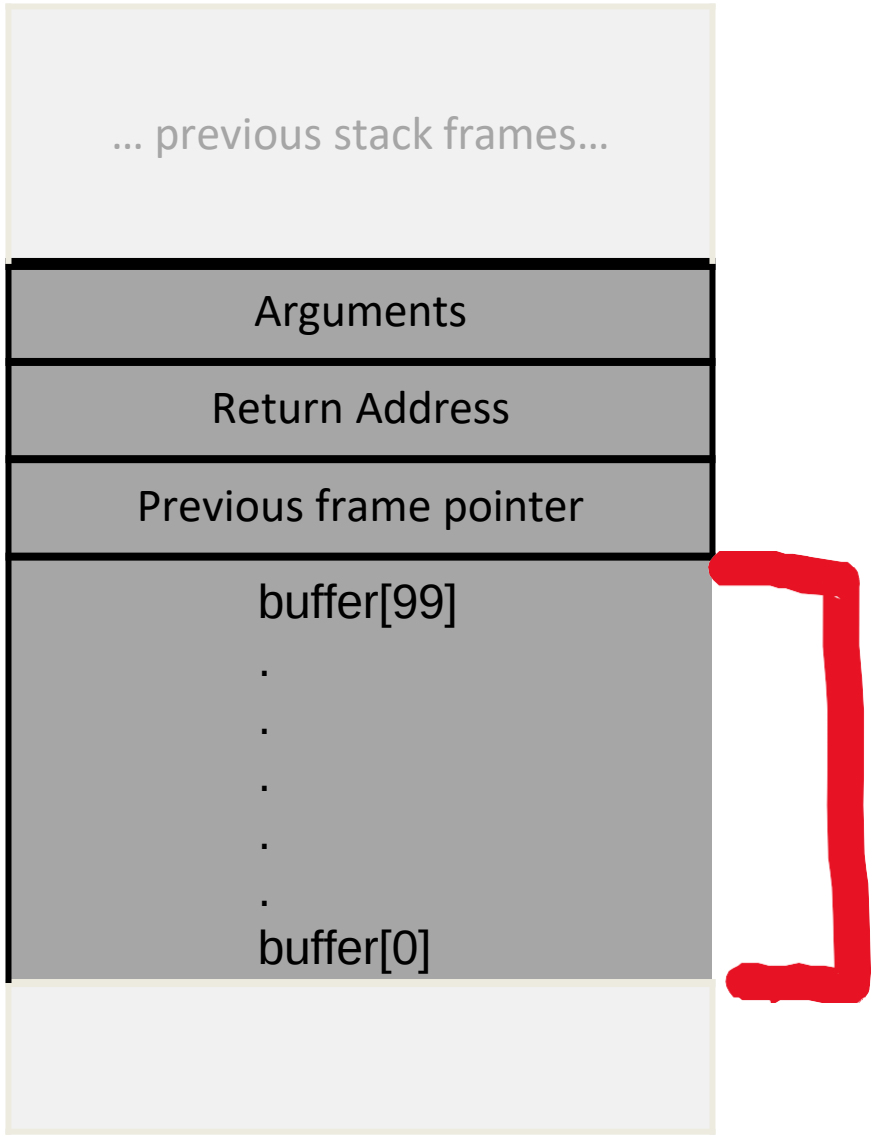
THE STACK



Here is the current stack frame in bof()

We can control the contents of buffer[] with our `badfile`

THE STACK



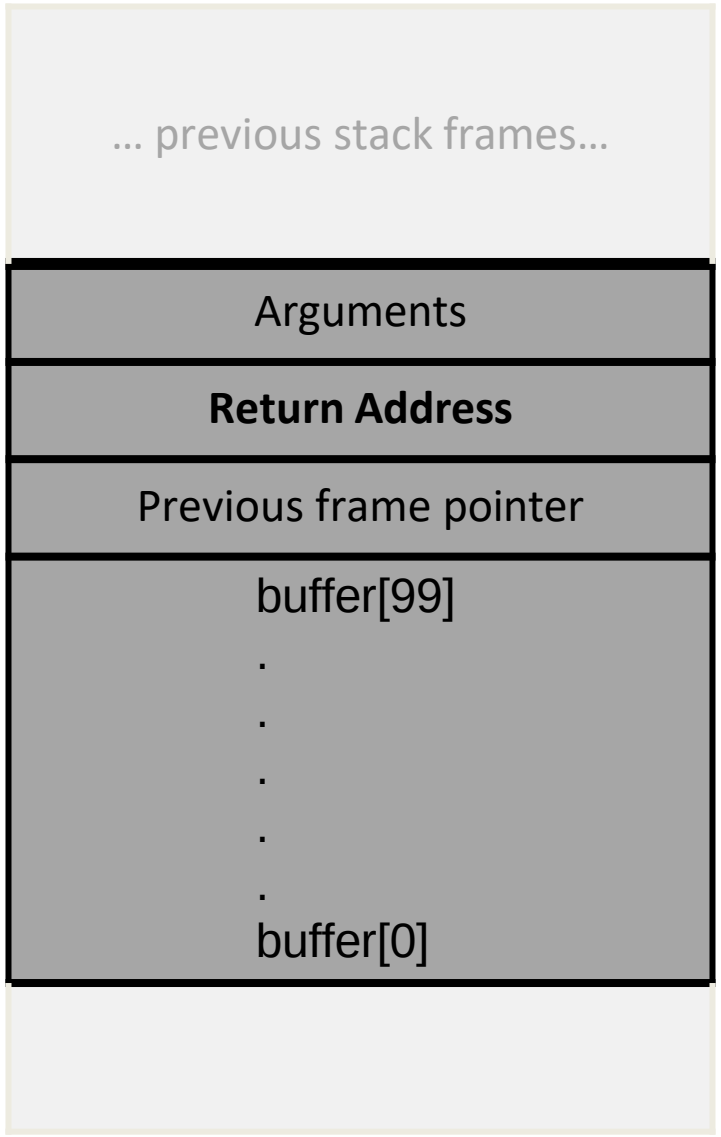
Here is the current stack frame in bof()

We can control the contents of buffer[] with our `badfile`

Badfile =
AAAAAAAAAAAAAAA
AAAAAAAAAAAAAAA
AAAAAAAAAAAAAAA
AAAAAAAAAAAAAAA
AAAAAAAAAAAAAAA
AAAAAAAAAAAAA

We can overflow this buffer and overwrite the contents above it

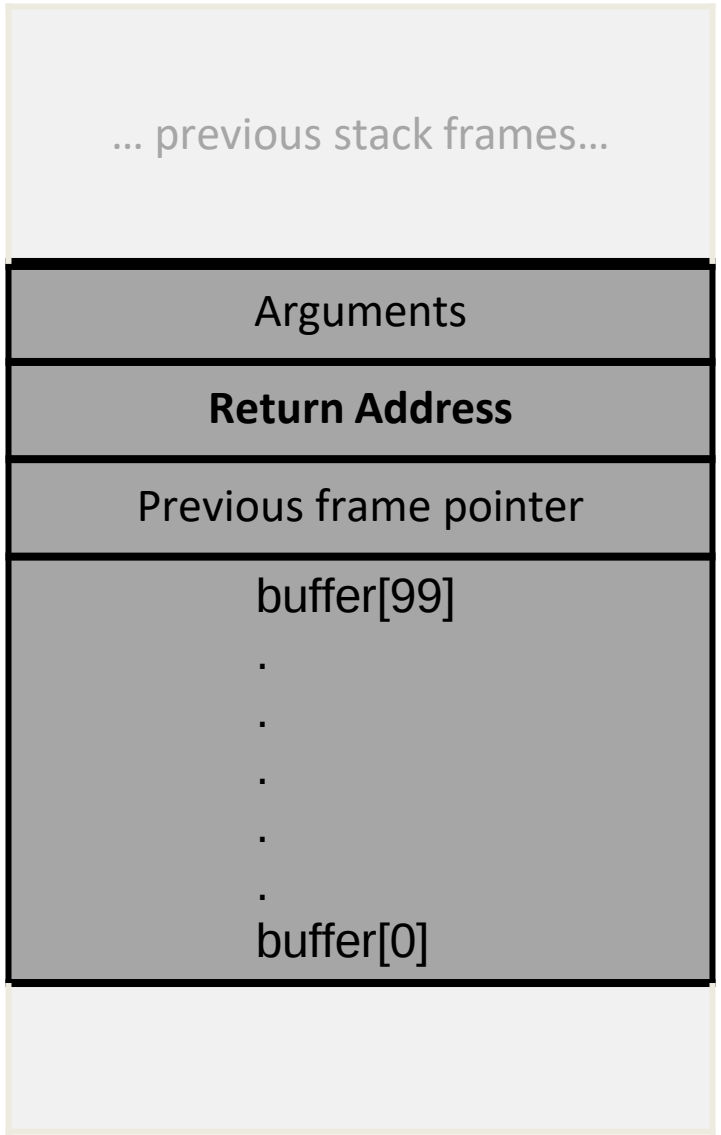
THE STACK



The juicy piece of information here in the **return address**

The program will jump to that address and continue to execute code

THE STACK



The juicy piece of information here in the **return address**

The program will jump to that address and continue to execute code

Overwriting the return address with something else can lead to:

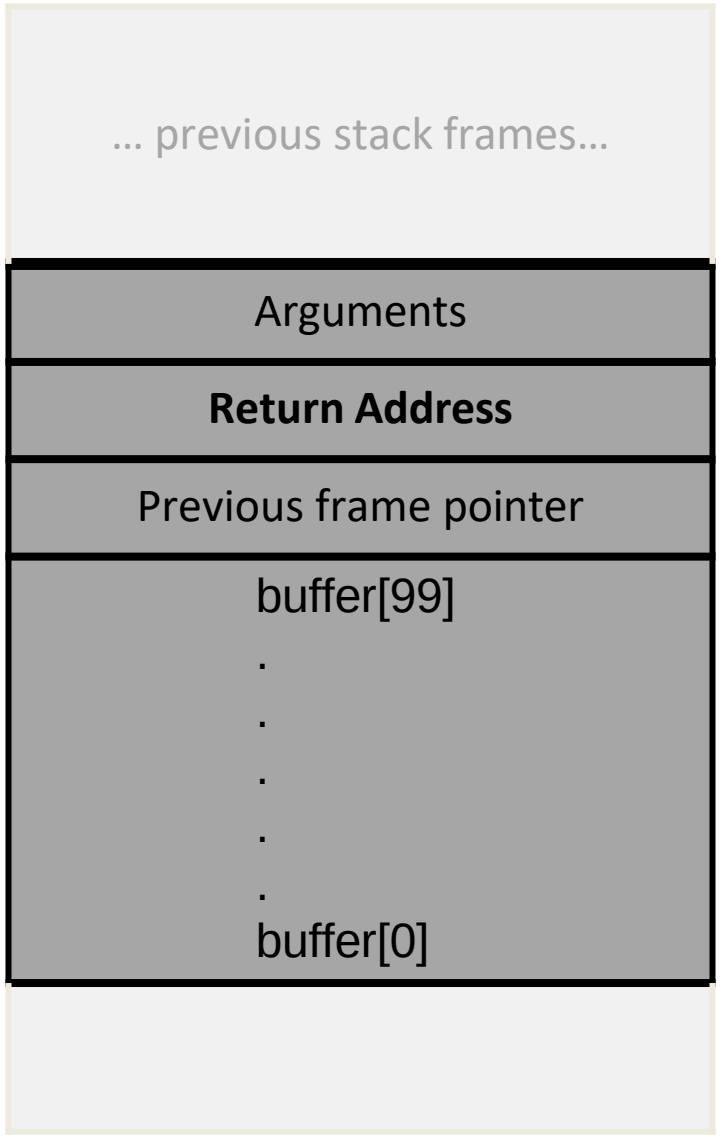
Non-existent address
➔ CRASH

Access Violation
➔ CRASH

Invalid Instruction
➔ CRASH

Execution of attacker's code! ➔ Oh no!!

THE STACK

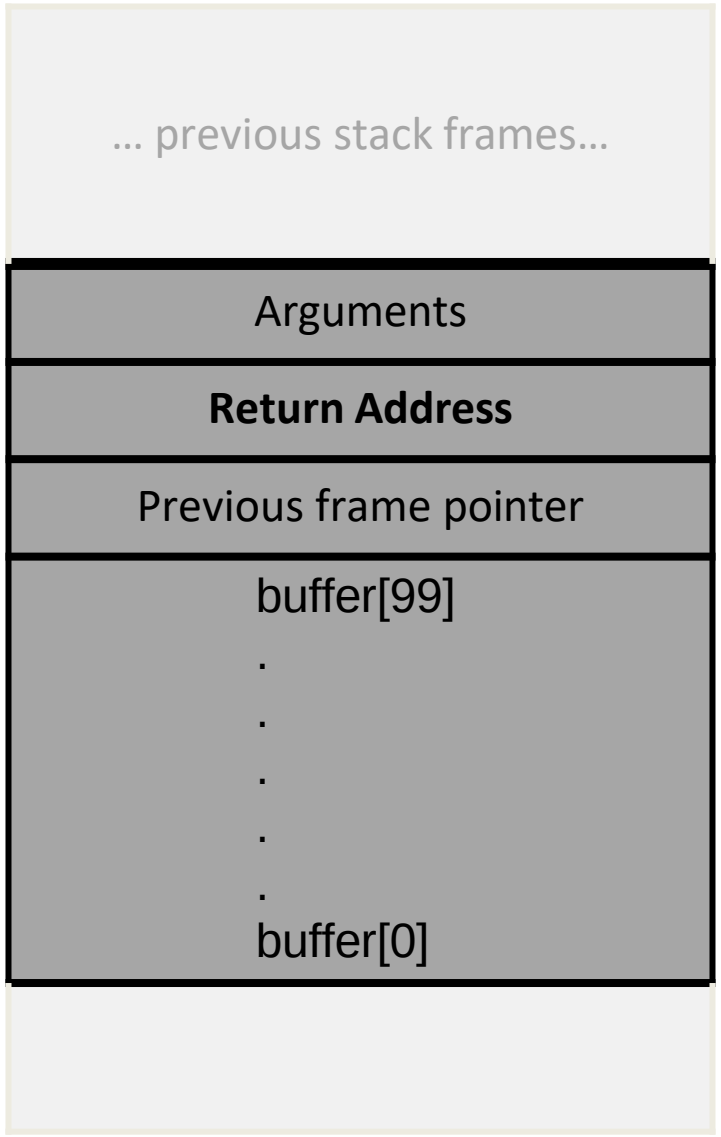


The juicy piece of information here in the **return address**

The program will jump to that address and continue to execute code

We can overwrite it, so if it points to the location of our own code we also inject, it will execute that code!

THE STACK



The juicy piece of information here in the **return address**

The program will jump to that address and continue to execute code

We can overwrite it, so if it points to the location of our own code we also inject, it will execute that code!

And our code will get a root shell

(there are many things our code can do, but we will be focused on getting a root shell)