



Discussion 0

C, x86

01/19/24

Staff

Announcements

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
		Homework 0 Release				
	Project 0 Release		Homework 0 Due	Homework 1 Release	Early Drop Deadline	Group Formation Form Due
		Project 0 Due	Project 1 Release			

Important Policies

Contact staff through Ed.

- Use cs162@eecs.berkeley.edu to reach head TAs and professor if Ed is insufficient.
- Only use individual emails for private matters.

3 midterms (check website for date and time).

- No conflicts allowed. There will be no alternate exams offered

Slip day policy:

- We provide 4 HW slip days and 5 Project slip days which are meant for emergencies
 - [Extension form](#) available for DSP and those with extenuating circumstances
 - Form is also accessible from the course website

Discussions 0 and 1 are **optional**.

- Feel free to attend different TAs' sections to find a teaching style that suits you best.
- Attendance for discussion 2 and onward are **mandatory**. Preference forms will be available soon.

Follow office hours policies of filling out a **detailed ticket** and being present in the OH room (Soda 347) when your ticket is taken.

- All OH will be **in person**.
- Take advantage of empty office hours by **starting assignments early**.

Post your questions on Ed in the appropriate threads.

- No private debugging posts allowed.

C

Resources

- [C Review Session](#)
- [CS 162 Ladder](#)
 - Overview of C and 61C topics
- [CS 61C Resources Page](#)
 - C staff notes, GDB reference card
- [Python Tutor C](#)
 - Just like CS 61A's Python Tutor, but for C

Types

C is **statically typed** (i.e. types are known at compile time).

C is **weakly typed** (i.e. can cast between any types).

Primitive types are char, short, int, long, float.

Arrays are contiguous pieces of memory of a homogenous type

- Denoted with [] (e.g. int [] for an array of integers).
- **String** is an array of chars with last element being null

Build compound types using structs

- Also contiguous in memory

Pointers are references that hold the address of an object in memory.

- Essentially just unsigned integers.
- Prefix a pointer with * to return the value at the memory address that the pointer is holding.
- Prefix a variable with & to return the memory address of the variable.

Little Endian	+3	+2	+1	+0	
0x7FFFFFFC	00	00	00	08	int a = 8
0x7FFFFFF8	FF	FF	FF	FF	int b = -1
0x7FFFFFF4	7F	FF	FF	FC	int *p = &a
0x7FFFFFF0	27	CE	00	'0'	27,CE are garbage
0x7FFFFFEC	'T'	'N'	'I'	'P'	char s[] = "PINT0"
0x7FFFFE8	00	00	00	02	struct point { int x; int y; }; struct point pt = {16, 2}
0x7FFFFE4	00	00	00	10	

Memory

Typical C program is divided into five segments.

- **Text** contains machine code of the compiled program.
- **(Un)initialized data** contains (un)initialized global/static memory.
- **Heap** contains dynamically allocated memory.
- **Stack** contains local variables and arguments.
- Initialized strings and global constants may be stored in read-only segments (.rodata).

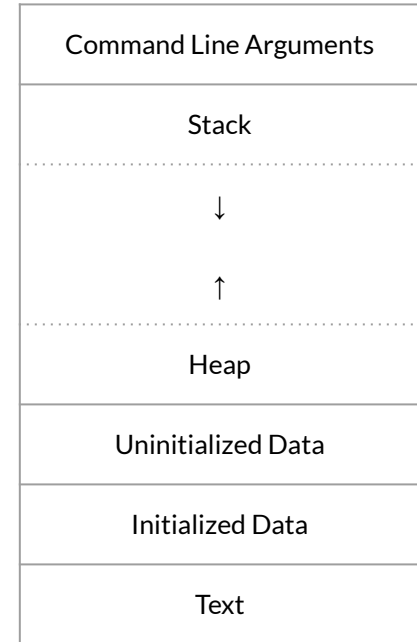
Think of memory as a giant array with elements of one byte.

- Memory addresses = indices of array.

Heap memory needs to be explicitly managed by the user.

- Allocate memory using `malloc`, `calloc`, `realloc` which return a pointer to a chunk of memory.
- Release memory using `free`.

High Address



Low Address

GNU Debugger (GDB)

Need to learn how to use it for 162 even if you skidded by 61C without it.

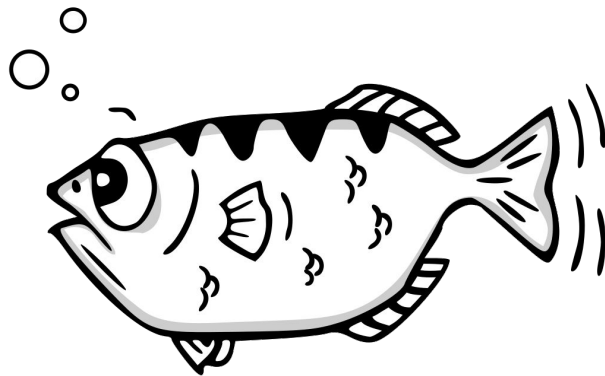
- Staff will not help during OH unless you are able to use GDB.

General workflow of using GDB.

1. Compile program using `-g` flag.
2. Start GDB using `gdb <executable name>`.
3. Set breakpoints using `break <line number>`. Can also break at functions using `break <function name>`.
4. Run program with `run`. If the program takes in arguments, pass in those after `run` (i.e. `run arg1 arg2 ...`).
5. Once breakpoint is hit, examine variables using `print`. Other commands like `display`, `watch`, and `set` are also useful.

Use GDB frequently to become familiar with the commands.

Check out [GDB manual](#) for more details.



Concept Check

1. Consider a valid double pointer `char** dbl_char` in a 32-bit system. What is returned by `sizeof(*dbl_char)`?
Bonus question: Does `sizeof(*dbl_char)` error if `dbl_char == NULL`?
2. Consider strings `char* a = "162 is the best"` and `char b[] = "162 is the best"`. Are `a` and `b` different?

3. Consider the following struct declaration:

```
struct point {  
    int x;  
    int y;  
};
```

Point out a few differences between

```
struct point p;  
printf("%d", p.x = 1);
```

and

```
struct point* p;  
printf("%d", p->x = 1);
```

Concept Check

1. Consider a valid double pointer `char** dbl_char` in a 32-bit system. What is returned by `sizeof(*dbl_char)`?

Dereferencing a double pointer gives a single pointer. 32-bit systems have 32-bit = 4 byte memory addresses.

Bonus question: Does `sizeof(*dbl_char)` error if `dbl_char == NULL`?

2. Consider strings `char* a = "162 is the best"` and `char b[] = "162 is the best"`. Are `a` and `b` different?

3. Consider the following struct declaration:

```
struct point {  
    int x;  
    int y;  
};
```

Point out a few differences between

```
struct point p;  
printf("%d", p.x = 1);
```

and

```
struct point* p;  
printf("%d", p->x = 1);
```

Concept Check

1. Consider a valid double pointer `char** dbl_char` in a 32-bit system. What is returned by `sizeof(*dbl_char)`?

Dereferencing a double pointer gives a single pointer. 32-bit systems have 32-bit = 4 byte memory addresses.

Bonus question: Does `sizeof(*dbl_char)` error if `dbl_char == NULL`?

No, since type sizes are known at compile time.

2. Consider strings `char* a = "162 is the best"` and `char b[] = "162 is the best"`. Are `a` and `b` different?

3. Consider the following struct declaration:

```
struct point {  
    int x;  
    int y;  
};
```

Point out a few differences between

```
struct point p;  
printf("%d", p.x = 1);
```

and

```
struct point* p;  
printf("%d", p->x = 1);
```

Concept Check

1. Consider a valid double pointer `char** dbl_char` in a 32-bit system. What is returned by `sizeof(*dbl_char)`?
Dereferencing a double pointer gives a single pointer. 32-bit systems have 32-bit = 4 byte memory addresses.
Bonus question: Does `sizeof(*dbl_char)` error if `dbl_char == NULL`? No, since type sizes are known at compile time.
2. Consider strings `char* a = "162 is the best"` and `char b[] = "162 is the best"`. Are `a` and `b` different?
Yes. `a` points to a string literal in the read-only segment (.rodata) while `b` resides on the stack.
3. Consider the following struct declaration:

```
struct point {  
    int x;  
    int y;  
};
```


Point out a few differences between

```
struct point p;  
printf("%d", p.x = 1);
```

and

```
struct point* p;  
printf("%d", p->x = 1);
```

Concept Check

1. Consider a valid double pointer `char** dbl_char` in a 32-bit system. What is returned by `sizeof(*dbl_char)`?
Dereferencing a double pointer gives a single pointer. 32-bit systems have 32-bit = 4 byte memory addresses.
Bonus question: Does `sizeof(*dbl_char)` error if `dbl_char == NULL`? No, since type sizes are known at compile time.

2. Consider strings `char* a = "162 is the best"` and `char b[] = "162 is the best"`. Are `a` and `b` different?
Yes. `a` points to a string literal in the read-only segment (.rodata) while `b` resides on the stack.

3. Consider the following struct declaration:

```
struct point {  
    int x;  
    int y;  
};
```

Point out a few differences between

```
struct point p;  
printf("%d", p.x = 1);
```

and

```
struct point* p;  
printf("%d", p->x = 1);
```

The first one allocates struct point that is the size of two ints. On the other hand, the second one puts a single pointer to a struct point on the stack, which is the size of a single int. Also, the second one will likely segfault, since the pointer is uninitialized.

Headers

```
#include <stdio.h>
#include "lib.h"

int main(int argc, char** argv) {
    helper_args_t helper_args;
    helper_args.string = argv[0];
    helper_args.target = '/';
    char* result = helper_func(&helper_args);
    printf("%s\n", result);
    return 0;
}
```

App.c

```
typedef struct helper_args {
#ifdef ABC
    char* aux;
#endif
    char* string;
    char target;
} helper_args_t;
char* helper_func(helper_args_t* args);
```

lib.h

```
#include "lib.h"

char* helper_func(helper_args_t* args) {
    int i;
    for (i = 0; args->string[i] != '\0'; i++)
        if (args->string[i] == args->target)
            return &args->string[i + 1];
    return args->string;
}
```

lib.c

You build the program on a 64-bit machine as follows.

> gcc -c app.c -o app.o

> gcc -c lib.c -o lib.o

> gcc app.o lib.o -o app

1. What is the size of a `helper_args_t` struct?
2. Suppose you add a `#define ABC` at the top of `lib.h`. What is the size of a `helper_args_t` struct?

Headers

```
#include <stdio.h>
#include "lib.h"

int main(int argc, char** argv) {
    helper_args_t helper_args;
    helper_args.string = argv[0];
    helper_args.target = '/';
    char* result = helper_func(&helper_args);
    printf("%s\n", result);
    return 0;
}
```

App.c

```
typedef struct helper_args {
    #ifdef ABC
        char* aux;
    #endif
    char* string;
    char target;
} helper_args_t;
char* helper_func(helper_args_t* args);
```

lib.h

```
#include "lib.h"

char* helper_func(helper_args_t* args) {
    int i;
    for (i = 0; args->string[i] != '\0'; i++)
        if (args->string[i] == args->target)
            return &args->string[i + 1];
    return args->string;
}
```

lib.c

1. What is the size of a `helper_args_t` struct?

16 bytes. Only `char* string` and `char target` meaning 9 bytes but GCC pads structs.

2. Suppose you add a `#define ABC` at the top of `lib.h`. What is the size of a `helper_args_t` struct?

Headers

```
#include <stdio.h>
#include "lib.h"

int main(int argc, char** argv) {
    helper_args_t helper_args;
    helper_args.string = argv[0];
    helper_args.target = '/';
    char* result = helper_func(&helper_args);
    printf("%s\n", result);
    return 0;
}
```

App.c

```
typedef struct helper_args {
    #ifdef ABC
        char* aux;
    #endif
        char* string;
        char target;
} helper_args_t;
char* helper_func(helper_args_t* args);
```

lib.h

```
#include "lib.h"

char* helper_func(helper_args_t* args) {
    int i;
    for (i = 0; args->string[i] != '\0'; i++)
        if (args->string[i] == args->target)
            return &args->string[i + 1];
    return args->string;
}
```

lib.c

1. What is the size of a `helper_args_t` struct?

16 bytes. Only `char* string` and `char target` meaning 9 bytes but GCC pads structs.

2. Suppose you add a `#define ABC` at the top of **lib.h**. What is the size of a `helper_args_t` struct?

24 bytes. Additional 8 bytes from `char* aux` since `ABC` is defined.

Headers

```
#include <stdio.h>
#include "lib.h"

int main(int argc, char** argv) {
    helper_args_t helper_args;
    helper_args.string = argv[0];
    helper_args.target = '/';
    char* result = helper_func(&helper_args);
    printf("%s\n", result);
    return 0;
}
```

App.c

```
typedef struct helper_args {
    #ifdef ABC
        char* aux;
    #endif
        char* string;
        char target;
    } helper_args_t;
char* helper_func(helper_args_t* args);
```

lib.h

```
#include "lib.h"

char* helper_func(helper_args_t* args) {
    int i;
    for (i = 0; args->string[i] != '\0'; i++)
        if (args->string[i] == args->target)
            return &args->string[i + 1];
    return args->string;
}
```

lib.c

3. Suppose you build the program in a different way with the original files (i.e. none of the changes from previous questions apply).

```
> gcc -DABC -c app.c -o app.o
> gcc -c lib.c -o lib.o
> gcc app.o lib.o -o app
```

The program will now exhibit undefined behavior. Why?

Headers

```
#include <stdio.h>
#include "lib.h"

int main(int argc, char** argv) {
    helper_args_t helper_args;
    helper_args.string = argv[0];
    helper_args.target = '/';
    char* result = helper_func(&helper_args);
    printf("%s\n", result);
    return 0;
}
```

App.c

```
typedef struct helper_args {
    #ifdef ABC
        char* aux;
    #endif
        char* string;
        char target;
    } helper_args_t;
char* helper_func(helper_args_t* args);
```

lib.h

```
#include "lib.h"

char* helper_func(helper_args_t* args) {
    int i;
    for (i = 0; args->string[i] != '\0'; i++)
        if (args->string[i] == args->target)
            return &args->string[i + 1];
    return args->string;
}
```

lib.c

3. Suppose you build the program in a different way with the original files (i.e. none of the changes from previous questions apply).

```
> gcc -DABC -c app.c -o app.o
> gcc -c lib.c -o lib.o
> gcc app.o lib.o -o app
```

The program will now exhibit undefined behavior. Why?

app.c is compiled with ABC defined but lib.c is not

- main stored argv[0] at address of helper_args + 8
- helper_func access address of args when accessing args->string
- First 8 bytes of args which helper_func is accessing is garbage

Debugging Segmentation Faults

```
void swap(char* a, int i, int j) {
    char t = a[i];
    a[i] = a[j];
    a[j] = t;
}

int partition(char* a, int l, int r){
    int pivot = a[l];
    int i = l, j = r+1;

    while (1) {
        do
            ++i;
        while (a[i] <= pivot && i <= r);

        do
            --j;
        while (a[j] > pivot);

        if (i >= j)
            break;

        swap(a, i, j);
    }

    swap(a, l, j);

    return j;
}
```

```
void sort(char* a, int l, int r){
    if (l < r){
        int j = partition(a, l, r);
        sort(a, l, j-1);
        sort(a, j+1, r);
    }
}

int main(int argc, char** argv){
    char* a = NULL;

    if (argc > 1)
        a = argv[1];
    else
        a = "IU is the best singer!";

    printf("Unsorted: \"%s\"\n", a);
    sort(a, 0, strlen(a) - 1);
    printf("Sorted   : \"%s\"\n", a);
}

> ./singer "Taeyeon is the best singer!"
Unsorted: "Taeyeon is the best singer!"
Sorted   : "      !Tabeeeeeghiinnorssstty"
> ./singer
Unsorted: "IU is the best singer!"
Segmentation fault (core dumped)
```

1. We want to debug the program using GDB. How should we compile the program?

Debugging Segmentation Faults

```
void swap(char* a, int i, int j) {
    char t = a[i];
    a[i] = a[j];
    a[j] = t;
}

int partition(char* a, int l, int r){
    int pivot = a[l];
    int i = l, j = r+1;

    while (1) {
        do
            ++i;
        while (a[i] <= pivot && i <= r);

        do
            --j;
        while (a[j] > pivot);

        if (i >= j)
            break;

        swap(a, i, j);
    }

    swap(a, l, j);

    return j;
}
```

```
void sort(char* a, int l, int r){
    if (l < r){
        int j = partition(a, l, r);
        sort(a, l, j-1);
        sort(a, j+1, r);
    }
}

int main(int argc, char** argv){
    char* a = NULL;

    if (argc > 1)
        a = argv[1];
    else
        a = "IU is the best singer!";

    printf("Unsorted: \"%s\"\n", a);
    sort(a, 0, strlen(a) - 1);
    printf("Sorted   : \"%s\"\n", a);
}
```

```
> ./singer "Taeyeon is the best singer!"
Unsorted: "Taeyeon is the best singer!"
Sorted   : "      !Tabeeeeeghiinnorsssty"
> ./singer
Unsorted: "IU is the best singer!"
Segmentation fault (core dumped)
```

1. We want to debug the program using GDB. How should we compile the program?

gcc -g singer.c -o singer

Need a -g flag for debugging.

Debugging Segmentation Faults

```
void swap(char* a, int i, int j) {
    char t = a[i];
    a[i] = a[j];
    a[j] = t;
}

int partition(char* a, int l, int r){
    int pivot = a[l];
    int i = l, j = r+1;

    while (1) {
        do
            ++i;
        while (a[i] <= pivot && i <= r);

        do
            --j;
        while (a[j] > pivot);

        if (i >= j)
            break;

        swap(a, i, j);
    }

    swap(a, l, j);

    return j;
}
```

```
void sort(char* a, int l, int r){
    if (l < r){
        int j = partition(a, l, r);
        sort(a, l, j-1);
        sort(a, j+1, r);
    }
}

int main(int argc, char** argv){
    char* a = NULL;

    if (argc > 1)
        a = argv[1];
    else
        a = "IU is the best singer!";

    printf("Unsorted: \"%s\"\n", a);
    sort(a, 0, strlen(a) - 1);
    printf("Sorted   : \"%s\"\n", a);
}
```

```
> ./singer "Taeyeon is the best singer!"
Unsorted: "Taeyeon is the best singer!"
Sorted   : "      !Tabeeeeeghiinnorssstty"
> ./singer
Unsorted: "IU is the best singer!"
Segmentation fault (core dumped)
```

2. When running the program without any arguments, what line does the segfault happen? Describe the memory operations happening in that line.

Debugging Segmentation Faults

```
void swap(char* a, int i, int j) {
    char t = a[i];
    a[i] = a[j];
    a[j] = t;
}

int partition(char* a, int l, int r){
    int pivot = a[l];
    int i = l, j = r+1;

    while (1) {
        do
            ++i;
        while (a[i] <= pivot && i <= r);

        do
            --j;
        while (a[j] > pivot);

        if (i >= j)
            break;

        swap(a, i, j);
    }

    swap(a, l, j);

    return j;
}
```

```
void sort(char* a, int l, int r){
    if (l < r){
        int j = partition(a, l, r);
        sort(a, l, j-1);
        sort(a, j+1, r);
    }
}

int main(int argc, char** argv){
    char* a = NULL;

    if (argc > 1)
        a = argv[1];
    else
        a = "IU is the best singer!";

    printf("Unsorted: \"%s\"\n", a);
    sort(a, 0, strlen(a) - 1);
    printf("Sorted   : \"%s\"\n", a);
}

> ./singer "Taeyeon is the best singer!"
Unsorted: "Taeyeon is the best singer!"
Sorted   : "      !Tabeeeeeghiinnorsssty"
> ./singer
Unsorted: "IU is the best singer!"
Segmentation fault (core dumped)
```

2. When running the program without any arguments, what line does the segfault happen? Describe the memory operations happening in that line.

Find segfaulting line by letting the program run until it encounters the fault.

```
> gcc -g singer.c -o singer
> gdb singer
(gdb) run
Starting program: /home/runner/intro/singer
Unsorted: "IU is the best char!"
```

```
Program received signal SIGSEGV, Segmentation
fault.
0x00005646308006c8 in swap (a=0x564630800904
"IU is the best singer!", i=1, j=21)
    at singer.c:6
6      a[i] = a[j];
```

Ignore “warning: Error disabling address space randomization: Operation not permitted” if using Replit.

Debugging Segmentation Faults

```
void swap(char* a, int i, int j) {
    char t = a[i];
    a[i] = a[j];
    a[j] = t;
}

int partition(char* a, int l, int r){
    int pivot = a[l];
    int i = l, j = r+1;

    while (1) {
        do
            ++i;
        while (a[i] <= pivot && i <= r);

        do
            --j;
        while (a[j] > pivot);

        if (i >= j)
            break;

        swap(a, i, j);
    }

    swap(a, l, j);

    return j;
}
```

```
void sort(char* a, int l, int r){
    if (l < r){
        int j = partition(a, l, r);
        sort(a, l, j-1);
        sort(a, j+1, r);
    }
}

int main(int argc, char** argv){
    char* a = NULL;

    if (argc > 1)
        a = argv[1];
    else
        a = "IU is the best singer!";

    printf("Unsorted: \"%s\"\n", a);
    sort(a, 0, strlen(a) - 1);
    printf("Sorted   : \"%s\"\n", a);
}

> ./singer "Taeyeon is the best singer!"
Unsorted: "Taeyeon is the best singer!"
Sorted   : "      !Tabeeeeeghiinnorssstty"
> ./singer
Unsorted: "IU is the best singer!"
Segmentation fault (core dumped)
```

2. When running the program without any arguments, what line does the segfault happen? Describe the memory operations happening in that line.

Use backtrace for more comprehensive breakdown.

```
(gdb) backtrace
#0  0x00005646308006c8 in swap
(a=0x564630800904 "IU is the best singer!",
i=1, j=21)
    at singer.c:6
#1  0x0000564630800773 in partition
(a=0x564630800904 "IU is the best singer!",
l=0, r=21)
    at singer.c:26
#2  0x00005646308007bd in sort
(a=0x564630800904 "IU is the best singer!",
l=0, r=21)
    at singer.c:36
#3  0x0000564630800861 in main (argc=1,
argv=0x7ffd04ac7098) at singer.c:51
```

Debugging Segmentation Faults

```
void swap(char* a, int i, int j) {
    char t = a[i];
    a[i] = a[j];
    a[j] = t;
}

int partition(char* a, int l, int r){
    int pivot = a[l];
    int i = l, j = r+1;

    while (1) {
        do
            ++i;
        while (a[i] <= pivot && i <= r);

        do
            --j;
        while (a[j] > pivot);

        if (i >= j)
            break;

        swap(a, i, j);
    }

    swap(a, l, j);

    return j;
}
```

```
void sort(char* a, int l, int r){
    if (l < r){
        int j = partition(a, l, r);
        sort(a, l, j-1);
        sort(a, j+1, r);
    }
}

int main(int argc, char** argv){
    char* a = NULL;

    if (argc > 1)
        a = argv[1];
    else
        a = "IU is the best singer!";

    printf("Unsorted: \"%s\"\n", a);
    sort(a, 0, strlen(a) - 1);
    printf("Sorted : \"%s\"\n", a);
}
```

```
> ./singer "Taeyeon is the best singer!"
Unsorted: "Taeyeon is the best singer!"
Sorted : "      !Tabeeeeeghiinnorssstty"
> ./singer
Unsorted: "IU is the best singer!"
Segmentation fault (core dumped)
```

2. When running the program without any arguments, what line does the segfault happen? Describe the memory operations happening in that line.

Two memory operations

1. Read from `a[j]`.
2. Write to `a[i]`.

Debugging Segmentation Faults

```
void swap(char* a, int i, int j) {
    char t = a[i];
    a[i] = a[j];
    a[j] = t;
}

int partition(char* a, int l, int r){
    int pivot = a[l];
    int i = l, j = r+1;

    while (1) {
        do
            ++i;
        while (a[i] <= pivot && i <= r);

        do
            --j;
        while (a[j] > pivot);

        if (i >= j)
            break;

        swap(a, i, j);
    }

    swap(a, l, j);

    return j;
}
```

```
void sort(char* a, int l, int r){
    if (l < r){
        int j = partition(a, l, r);
        sort(a, l, j-1);
        sort(a, j+1, r);
    }
}

int main(int argc, char** argv){
    char* a = NULL;

    if (argc > 1)
        a = argv[1];
    else
        a = "IU is the best singer!";

    printf("Unsorted: \"%s\"\n", a);
    sort(a, 0, strlen(a) - 1);
    printf("Sorted   : \"%s\"\n", a);
}
```

```
> ./singer "Taeyeon is the best singer!"
Unsorted: "Taeyeon is the best singer!"
Sorted   : "      !Tabeeeeeghiinnorssstty"
> ./singer
Unsorted: "IU is the best singer!"
Segmentation fault (core dumped)
```

3. Run the program with and without an argument and observe the memory addresses of a in the segfaulting line. Why are the memory addresses so different?

Debugging Segmentation Faults

```
void swap(char* a, int i, int j) {
    char t = a[i];
    a[i] = a[j];
    a[j] = t;
}

int partition(char* a, int l, int r){
    int pivot = a[l];
    int i = l, j = r+1;

    while (1) {
        do
            ++i;
        while (a[i] <= pivot && i <= r);

        do
            --j;
        while (a[j] > pivot);

        if (i >= j)
            break;

        swap(a, i, j);
    }

    swap(a, l, j);

    return j;
}
```

```
void sort(char* a, int l, int r){
    if (l < r){
        int j = partition(a, l, r);
        sort(a, l, j-1);
        sort(a, j+1, r);
    }
}

int main(int argc, char** argv){
    char* a = NULL;

    if (argc > 1)
        a = argv[1];
    else
        a = "IU is the best singer!";

    printf("Unsorted: \"%s\\n\", a);
    sort(a, 0, strlen(a) - 1);
    printf("Sorted : \"%s\\n\", a);
}

> ./singer "Taeyeon is the best singer!"
Unsorted: "Taeyeon is the best singer!"
Sorted : "      !Tabeeeeeghiinnorssstty"
> ./singer
Unsorted: "IU is the best singer!"
Segmentation fault (core dumped)
```

3. Run the program with and without an argument and observe the memory addresses of a in the segfaulting line. Why are the memory addresses so different?

Break at line 6 using GDB.

```
> gdb singer
(gdb) break 6
Breakpoint 1 at 0x6ab: file singer.c, line 6.
(gdb) run
Starting program: /home/runner/intro/singer
Unsorted: "IU is the best singer!"

Breakpoint 1, swap (
    a=0x5624e4600904 "IU is the best singer!",
    i=1, j=21)
    at singer.c:6
6      a[i] = a[j];
(gdb) print a
$1 = 0x5624e4600904 "IU is the best singer!"
```

Debugging Segmentation Faults

```
void swap(char* a, int i, int j) {
    char t = a[i];
    a[i] = a[j];
    a[j] = t;
}

int partition(char* a, int l, int r){
    int pivot = a[l];
    int i = l, j = r+1;

    while (1) {
        do
            ++i;
        while (a[i] <= pivot && i <= r);

        do
            --j;
        while (a[j] > pivot);

        if (i >= j)
            break;

        swap(a, i, j);
    }

    swap(a, l, j);

    return j;
}
```

```
void sort(char* a, int l, int r){
    if (l < r){
        int j = partition(a, l, r);
        sort(a, l, j-1);
        sort(a, j+1, r);
    }
}

int main(int argc, char** argv){
    char* a = NULL;

    if (argc > 1)
        a = argv[1];
    else
        a = "IU is the best singer!";

    printf("Unsorted: \"%s\"\n", a);
    sort(a, 0, strlen(a) - 1);
    printf("Sorted : \"%s\"\n", a);
}

> ./singer "Taeyeon is the best singer!"
Unsorted: "Taeyeon is the best singer!"
Sorted : "      !Tabeeeeeghiinnorssstty"
> ./singer
Unsorted: "IU is the best singer!"
Segmentation fault (core dumped)
```

3. Run the program with and without an argument and observe the memory addresses of a in the segfaulting line. Why are the memory addresses so different?

Break at line 6 using GDB.

(gdb) run "Taeyeon is the best singer!"
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/runner/intro/asuna
"Taeyeon is the best singer!"
Unsorted: "Taeyeon is the best singer!"

Breakpoint 1, swap (
a=0x7ffcce01bfe5 "Taeyeon is the best singer!", i=1, j=26)
at asuna.c:6
(gdb) print a
\$2 = 0x7ffcce01bfe5 "Taeyeon is the best singer!"

Debugging Segmentation Faults

```
void swap(char* a, int i, int j) {
    char t = a[i];
    a[i] = a[j];
    a[j] = t;
}

int partition(char* a, int l, int r){
    int pivot = a[l];
    int i = l, j = r+1;

    while (1) {
        do
            ++i;
        while (a[i] <= pivot && i <= r);

        do
            --j;
        while (a[j] > pivot);

        if (i >= j)
            break;

        swap(a, i, j);
    }

    swap(a, l, j);

    return j;
}
```

```
void sort(char* a, int l, int r){
    if (l < r){
        int j = partition(a, l, r);
        sort(a, l, j-1);
        sort(a, j+1, r);
    }
}

int main(int argc, char** argv){
    char* a = NULL;

    if (argc > 1)
        a = argv[1];
    else
        a = "IU is the best singer!";

    printf("Unsorted: \"%s\"\n", a);
    sort(a, 0, strlen(a) - 1);
    printf("Sorted   : \"%s\"\n", a);
}
```

```
> ./singer "Taeyeon is the best singer!"
Unsorted: "Taeyeon is the best singer!"
Sorted   : "      !Tabeeeeeghiinnorsssty"
> ./singer
Unsorted: "IU is the best singer!"
Segmentation fault (core dumped)
```

3. Run the program with and without an argument and observe the memory addresses of a in the segfaulting line. Why are the memory addresses so different?

No argument: 0x5624e4600904

- Statically defined strings stored in read-only segment (.rodata).

With argument: 0x7ffcce01bfe5

- Arguments are passed in through the stack.

Debugging Segmentation Faults

```
void swap(char* a, int i, int j) {
    char t = a[i];
    a[i] = a[j];
    a[j] = t;
}

int partition(char* a, int l, int r){
    int pivot = a[l];
    int i = l, j = r+1;

    while (1) {
        do
            ++i;
        while (a[i] <= pivot && i <= r);

        do
            --j;
        while (a[j] > pivot);

        if (i >= j)
            break;

        swap(a, i, j);
    }

    swap(a, l, j);

    return j;
}
```

```
void sort(char* a, int l, int r){
    if (l < r){
        int j = partition(a, l, r);
        sort(a, l, j-1);
        sort(a, j+1, r);
    }
}

int main(int argc, char** argv){
    char* a = NULL;

    if (argc > 1)
        a = argv[1];
    else
        a = "IU is the best singer!";

    printf("Unsorted: \"%s\"\n", a);
    sort(a, 0, strlen(a) - 1);
    printf("Sorted : \"%s\"\n", a);
}
```

```
> ./singer "Taeyeon is the best singer!"
Unsorted: "Taeyeon is the best singer!"
Sorted : "      !Tabeeeeeghiinnorssstty"
> ./singer
Unsorted: "IU is the best singer!"
Segmentation fault (core dumped)
```

4. How should the code be changed to fix the segfault?

Debugging Segmentation Faults

```
void swap(char* a, int i, int j) {
    char t = a[i];
    a[i] = a[j];
    a[j] = t;
}

int partition(char* a, int l, int r){
    int pivot = a[l];
    int i = l, j = r+1;

    while (1) {
        do
            ++i;
        while (a[i] <= pivot && i <= r);

        do
            --j;
        while (a[j] > pivot);

        if (i >= j)
            break;

        swap(a, i, j);
    }

    swap(a, l, j);

    return j;
}
```

```
void sort(char* a, int l, int r){
    if (l < r){
        int j = partition(a, l, r);
        sort(a, l, j-1);
        sort(a, j+1, r);
    }
}

int main(int argc, char** argv){
    char* a = NULL;

    if (argc > 1)
        a = argv[1];
    else
        a = "IU is the best singer!";

    printf("Unsorted: \"%s\"\n", a);
    sort(a, 0, strlen(a) - 1);
    printf("Sorted   : \"%s\"\n", a);
}
```

```
> ./singer "Taeyeon is the best singer!"
Unsorted: "Taeyeon is the best singer!"
Sorted   : "      !Tabeeeeeghiinnorssstty"
> ./singer
Unsorted: "IU is the best singer!"
Segmentation fault (core dumped)
```

4. How should the code be changed to fix the segfault?

Write to `a[i]` is the problem since it's in read-only data → need to put `a` in writable memory.

Allocate memory on the heap and put the default string on there.

- Accomplish with `malloc` followed by `strcpy`.
- Equivalently, use [`strdup`](#).

Replace line 48.

Debugging Segmentation Faults

```
void swap(char* a, int i, int j) {
    char t = a[i];
    a[i] = a[j];
    a[j] = t;
}

int partition(char* a, int l, int r){
    int pivot = a[l];
    int i = l, j = r+1;

    while (1) {
        do
            ++i;
        while (a[i] <= pivot && i <= r);

        do
            --j;
        while (a[j] > pivot);

        if (i >= j)
            break;

        swap(a, i, j);
    }

    swap(a, l, j);

    return j;
}
```

```
void sort(char* a, int l, int r){
    if (l < r){
        int j = partition(a, l, r);
        sort(a, l, j-1);
        sort(a, j+1, r);
    }
}

int main(int argc, char** argv){
    char* a = NULL;

    if (argc > 1)
        a = argv[1];
    else
        a = strdup("IU is the best singer!");

    printf("Unsorted: \"%s\"\n", a);
    sort(a, 0, strlen(a) - 1);
    printf("Sorted   : \"%s\"\n", a);
}
```

```
> ./singer "Taeyeon is the best singer!"
Unsorted: "Taeyeon is the best singer!"
Sorted   : "      !Tabeeeeeghiinnorssstty"
> ./singer
Unsorted: "IU is the best singer!"
Segmentation fault (core dumped)
```

4. How should the code be changed to fix the segfault?

Write to `a[i]` is the problem since it's in read-only data → need to put `a` in writable memory.

Allocate memory on the heap and put the default string on there.

- Accomplish with `malloc` followed by `strcpy`.
- Equivalently, use [`strdup`](#).

Replace line 48.

x86

Registers

Registers are small storage spaces directly on the processor.

- Allows for fast memory access.

General purpose registers (GPR) store both data and addresses.

- x86 has 8, RISC-V has 32 (x0-x31).
- Started as 16-bits, extend to 32-bit using e prefix (e.g. EAX for AX).
- Access 8-bit LSB by replacing last letter with l (e.g. AL for AX).
- Access 8-bit MSB by replacing last letter with h (e.g. AH for AX) only for AX, BX, CX, DX.

Instruction pointer register holds address of next instruction to execute.

- Called ip which can be extended with e prefix like a GPR.
- Can't be read/modified like a GPR using regular memory instructions.

	Name	Purpose
AX	Accumulator	I/O port access, arithmetic, interrupt calls
BX	Base	Base pointer for memory access
CX	Counter	Loop counting, bit shifts
DX	Data	I/O port access, arithmetic, interrupt calls
SP	Stack Pointer	Top address of stack
BP	Base Pointer	Base address of stack
SI	Source Index	Source for stream operations (e.g. string)
DI	Destination Index	Destination for stream operations (e.g. string)

Syntax

Use **AT&T Syntax** not Intel which is used by GCC.

Prefix registers with % (e.g. %eax), constants with \$ (e.g. \$4).

General structure is **inst src, dest**.

Address memory with **offset(base, index, scale)**.

- base, index = registers, offset = any integer, scale = 1, 2, 4, or 8.
- Accesses data at address $\text{base} + \text{index} * \text{scale} + \text{offset}$.
- All parameters optional but will see base and offset usually.
- Using lea instruction will operate on the address itself not contents.

Suffix instructions to signify operand size.

- Not always necessary but should use them regardless.

<code>mov 8(%ebx), %eax</code>	Move contents from address EBX + 8 into EAX
<code>mov %ecx, -4(%esi, %ebx, 8)</code>	Move contents in ECX into address ESI + 8 * EBX - 4
<code>lea 8(%ebx), %eax</code>	Puts EBX + 8 into EAX

<code>movb \$0, (%esp)</code>	Zero out a single byte from ESP
<code>movw \$0, (%esp)</code>	Zero out two bytes from ESP
<code>movl \$0, (%esp)</code>	Zero out four bytes from ESP

Calling Convention

Calling convention is a procedure for how to call and return from functions.

- Specifies stack management, argument passing, register saving, etc.
- One set of rule *each* for the **caller** and **callee**

Many different calling conventions, will use the one defined in **i386 System V ABI** in this class.

Calling Convention

EBP

Existing Caller Stack Frame

ESP

Calling Convention

Caller

Before calling the function (i.e. **prologue**),

1. Save caller-saved GPRs (EAX, ECX, EDX) onto the stack *if needed after the function call.*

EBP

Existing Caller Stack Frame

ESP

Caller-Saved GPRs

Calling Convention

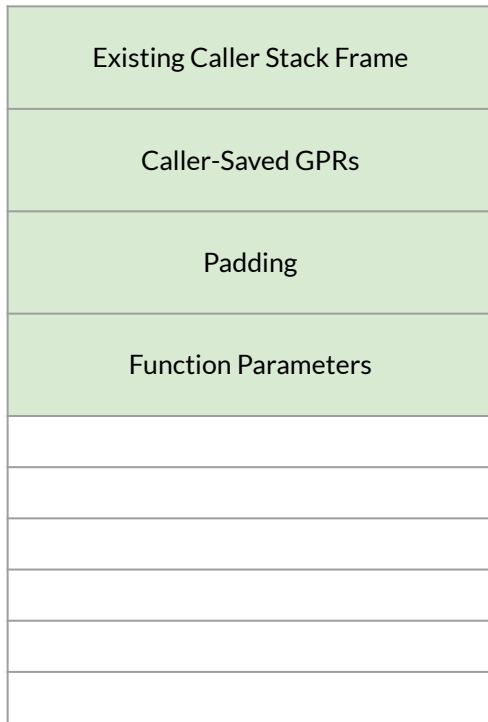
Caller

Before calling the function (i.e. **prologue**),

1. Save caller-saved GPRs (EAX, ECX, EDX) onto the stack *if needed after the function call*.
2. **Push parameters onto the stack in reverse order. Add necessary padding before the parameters to ensure a 16-byte alignment.**

EBP

ESP
(16 Byte aligned)



Calling Convention

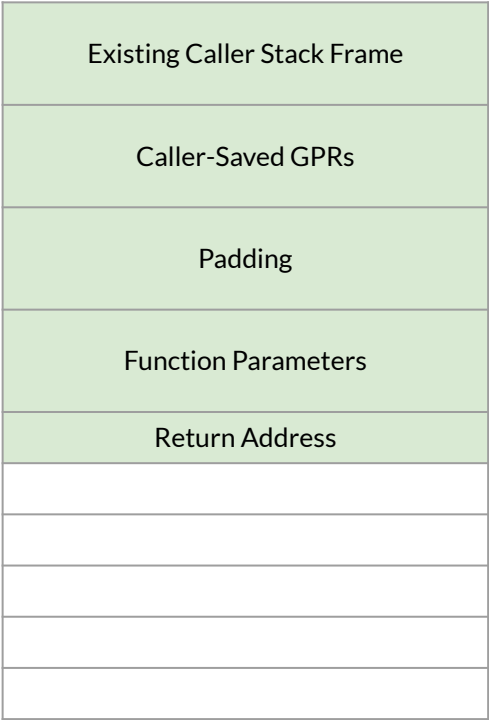
Caller

Before calling the function (i.e. **prologue**),

- 1. Save caller-saved GPRs (EAX, ECX, EDX) onto the stack *if needed after the function call*.
- 2. Push parameters onto the stack in reverse order. Add necessary padding *before the parameters* to ensure a 16-byte alignment.

Call function by pushing the return address onto the stack and jumping to the function.

EBP



ESP

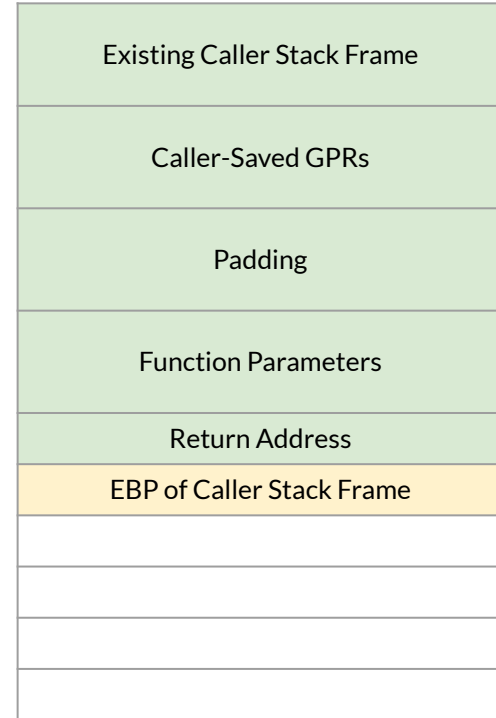
Calling Convention

Callee

Before executing any function logic (i.e. prologue),

1. **Push EBP onto the stack and set EBP to be the new ESP.**

EBP, ESP

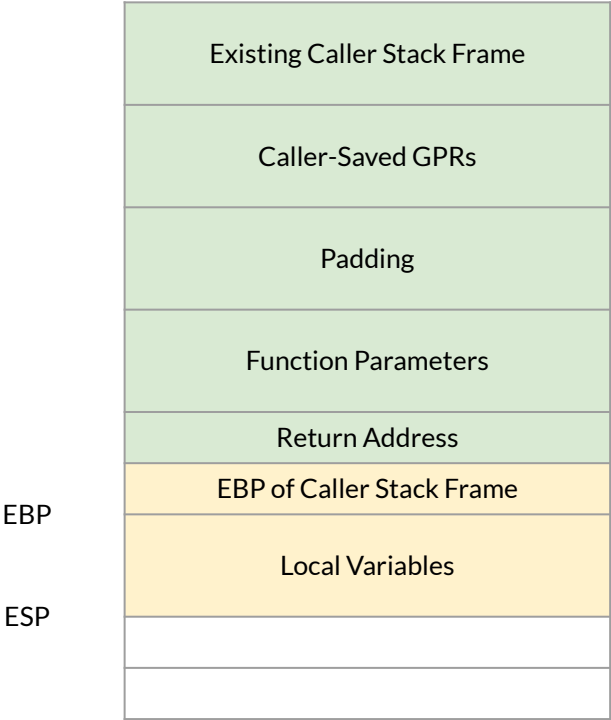


Calling Convention

Callee

Before executing any function logic (i.e. prologue),

- 1. Push EBP onto the stack and set EBP to be the new ESP.
- 2. **Allocate stack space for local variables.**

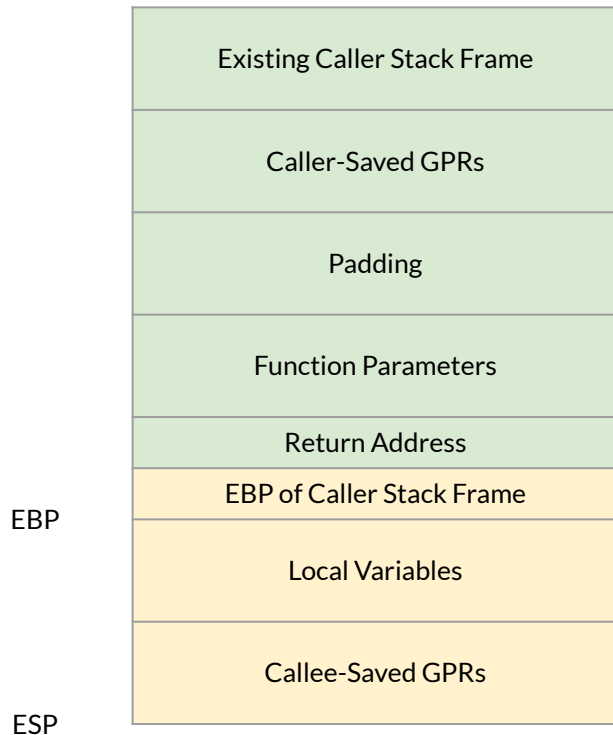


Calling Convention

Callee

Before executing any function logic (i.e. prologue),

1. Push EBP onto the stack and set EBP to be the new ESP.
2. Allocate stack space for local variables.
3. **Save callee-saved GPRs (EBX, EDI, ESI) onto the stack if used during the function logic.**



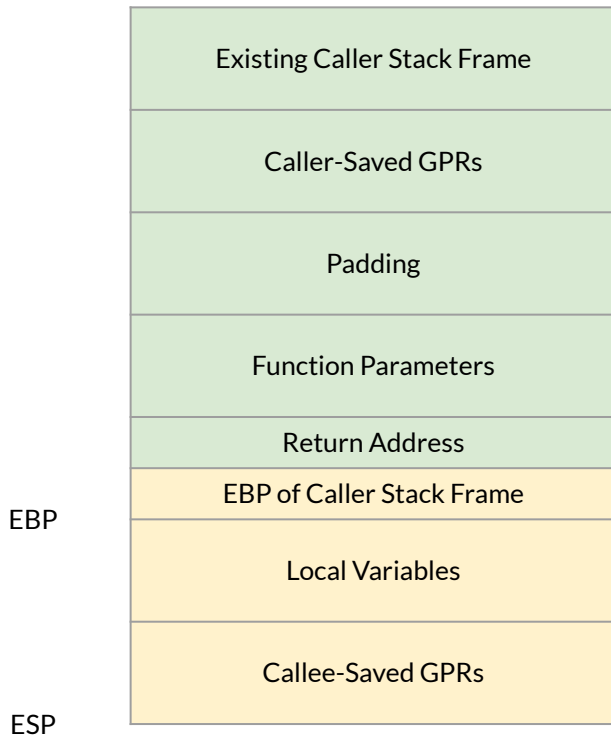
Calling Convention

Callee

Before executing any function logic (i.e. prologue),

1. Push EBP onto the stack and set EBP to be the new ESP.
2. Allocate stack space for local variables.
3. Save callee-saved GPRs (EBX, EDI, ESI) onto the stack *if used during the function logic*.

Perform function logic.



Calling Convention

Callee

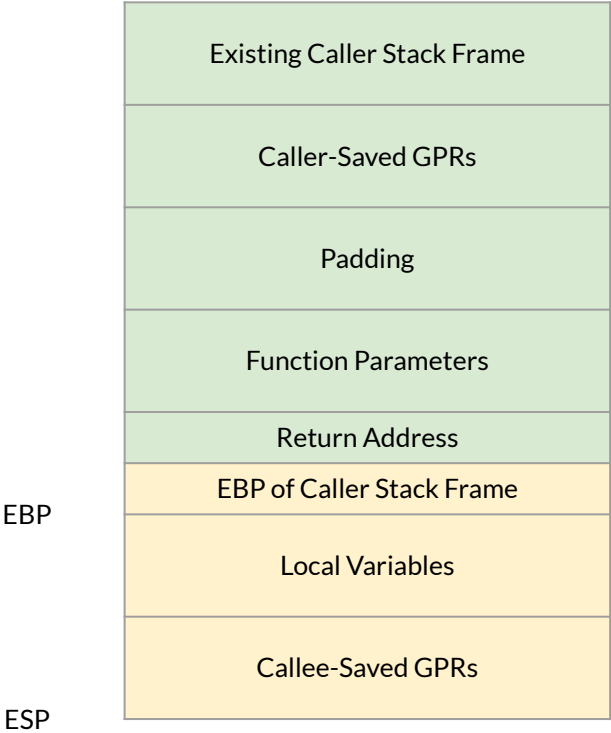
Before executing any function logic (i.e. prologue),

- 1. Push EBP onto the stack and set EBP to be the new ESP.
- 2. Allocate stack space for local variables.
- 3. Save callee-saved GPRs (EBX, EDI, ESI) onto the stack *if used during the function logic*.

Perform function logic.

Before returning (i.e. **epilogue**),

- 1. **Store return value in EAX.**



Calling Convention

Callee

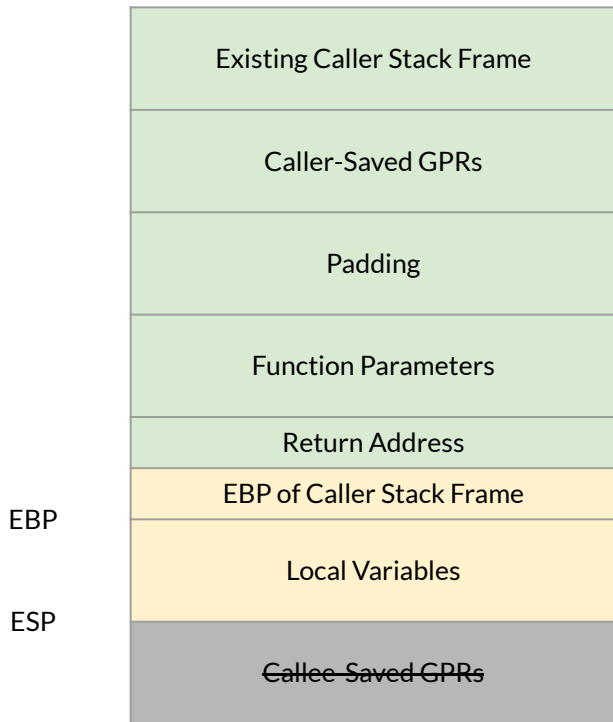
Before executing any function logic (i.e. prologue),

1. Push EBP onto the stack and set EBP to be the new ESP.
2. Allocate stack space for local variables.
3. Save callee-saved GPRs (EBX, EDI, ESI) onto the stack *if used during the function logic*.

Perform function logic.

Before returning (i.e. **epilogue**),

1. Store return value in EAX.
2. **Restore callee-saved GPRs if any from the prologue.**



Calling Convention

Callee

Before executing any function logic (i.e. prologue),

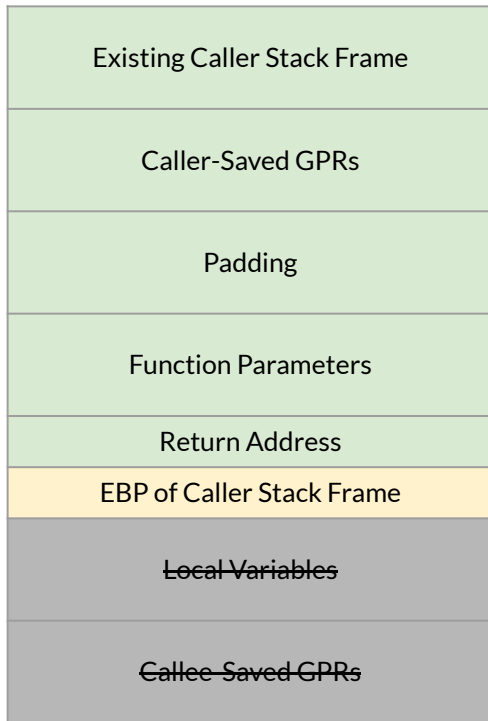
1. Push EBP onto the stack and set EBP to be the new ESP.
2. Allocate stack space for local variables.
3. Save callee-saved GPRs (EBX, EDI, ESI) onto the stack *if used during the function logic*.

Perform function logic.

Before returning (i.e. **epilogue**),

1. Store return value in EAX.
2. Restore callee-saved GPRs if any from the prologue.
3. **Deallocate local variables.**

EBP, ESP



Calling Convention

Callee

Before executing any function logic (i.e. prologue),

1. Push EBP onto the stack and set EBP to be the new ESP.
2. Allocate stack space for local variables.
3. Save callee-saved GPRs (EBX, EDI, ESI) onto the stack *if used during the function logic*.

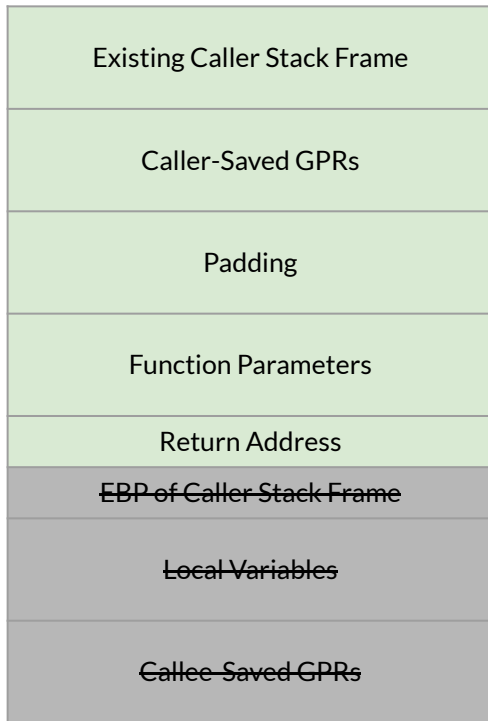
Perform function logic.

Before returning (i.e. **epilogue**),

1. Store return value in EAX.
2. Restore callee-saved GPRs if any from the prologue.
3. Deallocate local variables.
4. **Restore caller's EBP from stack.**

EBP

ESP



Calling Convention

Callee

Before executing any function logic (i.e. prologue),

1. Push EBP onto the stack and set EBP to be the new ESP.
2. Allocate stack space for local variables.
3. Save callee-saved GPRs (EBX, EDI, ESI) onto the stack *if used during the function logic*.

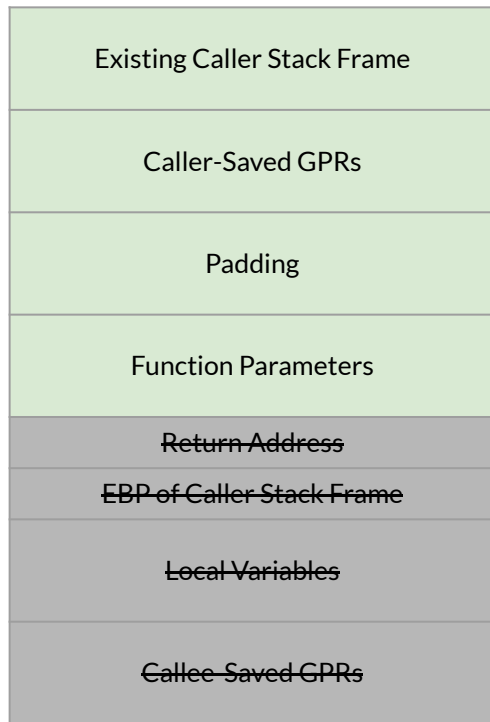
Perform function logic.

Before returning (i.e. **epilogue**),

1. Store return value in EAX.
2. Restore callee-saved GPRs if any from the prologue.
3. Deallocate local variables.
4. Restore caller's EBP from stack.
5. **Return from function call by popping the return address pushed by the caller in its prologue and jumping to it.**

EBP

ESP



Calling Convention

Caller

Before calling the function (i.e. **prologue**),

1. Save caller-saved GPRs (EAX, ECX, EDX) onto the stack *if needed after the function call*.
2. Push parameters onto the stack in reverse order. Add necessary padding *before the parameters* to ensure a 16-byte alignment.

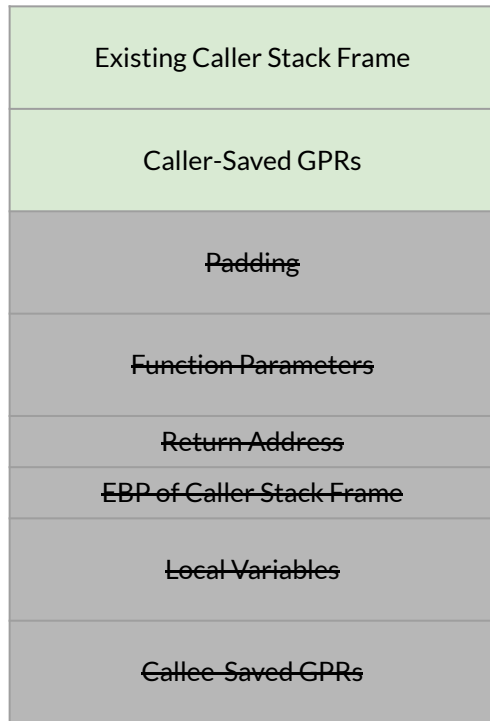
Call function by pushing the return address onto the stack and jumping to the function.

Once function call returns (i.e. **epilogue**),

1. Remove parameters from the stack.

EBP

ESP



Calling Convention

Caller

Before calling the function (i.e. **prologue**),

1. Save caller-saved GPRs (EAX, ECX, EDX) onto the stack *if needed after the function call*.
2. Push parameters onto the stack in reverse order. Add necessary padding *before the parameters* to ensure a 16-byte alignment.

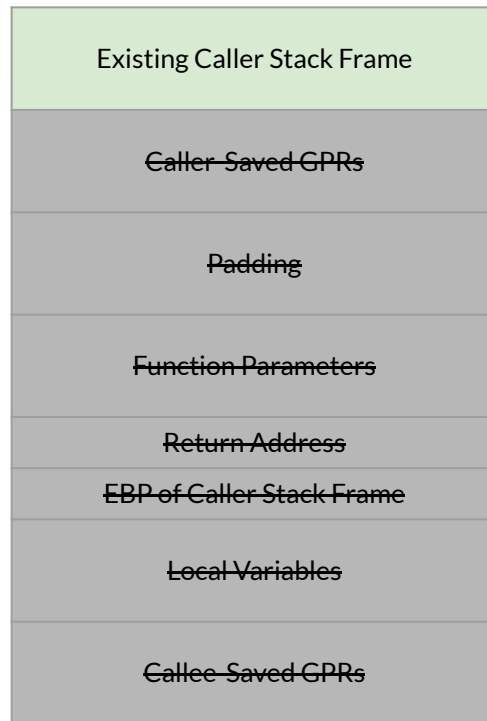
Call function by pushing the return address onto the stack and jumping to the function.

Once function call returns (i.e. **epilogue**),

1. Remove parameters from the stack.
2. **Restore caller-saved GPRs if any from the prologue.**

EBP

ESP



Calling Convention

Caller

Before calling the function (i.e. **prologue**),

1. Save caller-saved GPRs (EAX, ECX, EDX) onto the stack *if needed after the function call*.
2. Push parameters onto the stack in reverse order. Add necessary padding *before the parameters* to ensure a 16-byte alignment.

Call function by pushing the return address onto the stack and jumping to the function.

Once function call returns (i.e. **epilogue**),

1. Remove parameters from the stack.
2. Restore caller-saved GPRs if any from the prologue.

Callee

Before executing any function logic (i.e. prologue),

1. Push EBP onto the stack and set EBP to be the new ESP.
2. Allocate stack space for local variables.
3. Save callee-saved GPRs (EBX, EDI, ESI) onto the stack *if used during the function logic*.

Perform function logic.

Before returning (i.e. **epilogue**),

1. Store return value in EAX.
2. Restore callee-saved GPRs if any from the prologue.
3. Deallocate local variables.
4. Restore caller's EBP from stack.
5. Return from function call by popping the return address pushed by the caller in its prologue and jumping to it.

Calling Convention

Instruction	Purpose	Effective
<code>pushl src</code>	Push <code>src</code> onto stack	<code>subl \$4, %esp</code> <code>movl src, (%esp)</code>
<code>popl dest</code>	Pop from stack into <code>dest</code>	<code>movl (%esp), dest</code> <code>addl \$4, %esp</code>
<code>call addr</code>	Push return address onto stack and jump to <code>addr</code>	<code>pushl %eip</code> <code>jump addr</code>
<code>leave</code>	Restore EBP and ESP to previous stack frame	<code>movl %ebp, %esp</code> <code>popl %ebp</code>
<code>ret</code>	Pop return address from stack and jump to it	<code>popl %eip</code>

Concept Check

1. Between SP and BP, which has a higher memory address?
2. Write three different ways to clear the EAX register (i.e. store a 0).
3. True or False: Right before the caller jumps to the desired function, the stack must be 16-byte aligned.

Concept Check

1. Between SP and BP, which has a higher memory address?

BP. Stack grows downwards → top of the stack (SP) moves towards lower addresses

2. Write three different ways to clear the EAX register (i.e. store a 0).

3. True or False: Right before the caller jumps to the desired function, the stack must be 16-byte aligned.

Concept Check

1. Between SP and BP, which has a higher memory address?

BP. Stack grows downwards → top of the stack (SP) moves towards lower addresses

2. Write three different ways to clear the EAX register (i.e. store a 0).

`movl $0, %eax`

`subl %eax, %eax`

`xorl %eax, %eax`

3. True or False: Right before the caller jumps to the desired function, the stack must be 16-byte aligned.

Concept Check

1. Between SP and BP, which has a higher memory address?

BP. Stack grows downwards → top of the stack (SP) moves towards lower addresses

2. Write three different ways to clear the EAX register (i.e. store a 0).

`movl $0, %eax`

`subl %eax, %eax`

`xorl %eax, %eax`

3. **True or False: Right before the caller jumps to the desired function, the stack must be 16-byte aligned.**

False. Stack needs to be 16-byte aligned *after parameters have been pushed* onto the stack. Return address is pushed right before jumping.

Reverse Engineering

```
int main(void) {
    for (int a = 1; _____) {
        int a2 = _____;
        for (int b = _____) {
            int b2 = _____;
            int c = _____;
            int c2 = _____;
            if (_____){
                return _____;
            }
        }
    }
    return 0;
}
```

1. What is the memory address of a relative to the base pointer?

```

1 main:
2     pushl    %ebp
3     movl    %esp, %ebp
4     subl    $32, %esp
5     movl    $1, -4(%ebp)
6     jmp     .L2
7 .L7:
8     movl    -4(%ebp), %eax
9     imull   %eax, %eax
10    movl    %eax, -12(%ebp)
11    movl    -4(%ebp), %eax
12    movl    %eax, -8(%ebp)
13    jmp     .L3
14 .L6:
15    movl    -8(%ebp), %eax
16    imull   %eax, %eax
17    movl    %eax, -16(%ebp)
18    movl    $1000, %eax
19    subl    -4(%ebp), %eax
20    subl    -8(%ebp), %eax
21    movl    %eax, -20(%ebp)
22    movl    -20(%ebp), %eax
23    imull   %eax, %eax
24    movl    %eax, -24(%ebp)
25    movl    -12(%ebp), %edx
26    movl    -16(%ebp), %eax
27    addl    %edx, %eax

```

```

28      cmpl    %eax, -24(%ebp)
29      jne     .L4
30      movl    -4(%ebp), %eax
31      imull   -8(%ebp), %eax
32      imull   -20(%ebp), %eax
33      jmp     .L5
34 .L4:
35      addl    $1, -8(%ebp)
36 .L3:
37      cmpl    $666, -8(%ebp)
38      jle     .L6
39      addl    $1, -4(%ebp)
40 .L2:
41      cmpl    $333, -4(%ebp)
42      jle     .L7
43      movl    $0, %eax
44 .L5:
45      leave
46      ret

```

Reverse Engineering

```
int main(void) {  
    for (int a = 1; _____) {  
        int a2 = _____;  
        for (int b = _____) {  
            int b2 = _____;  
            int c = _____;  
            int c2 = _____;  
            if (_____) {  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

1. What is the memory address of **a**, relative to the base pointer?
4 bytes below the base pointer (i.e. EBP - 4).

1	main:						
2		pushl	%ebp	28	cmpl	%eax, -24(%ebp)	
3		movl	%esp, %ebp	29	jne	.L4	
4		subl	\$32, %esp	30	movl	-4(%ebp), %eax	
5		movl	\$1, -4(%ebp)	31	imull	-8(%ebp), %eax	
6		jmp	.L2	32	imull	-20(%ebp), %eax	
7	.L7:			33	jmp	.L5	
8		movl	-4(%ebp), %eax	34	.L4:		
9		imull	%eax, %eax	35		addl	\$1, -8(%ebp)
10		movl	%eax, -12(%ebp)	36	.L3:		
11		movl	-4(%ebp), %eax	37		cmpl	\$666, -8(%ebp)
12		movl	%eax, -8(%ebp)	38		jle	.L6
13		jmp	.L3	39		addl	\$1, -4(%ebp)
14	.L6:			40	.L2:		
15		movl	-8(%ebp), %eax	41		cmpl	\$333, -4(%ebp)
16		imull	%eax, %eax	42		jle	.L7
17		movl	%eax, -16(%ebp)	43		movl	\$0, %eax
18		movl	\$1000, %eax	44	.L5:		
19		subl	-4(%ebp), %eax	45		leave	
20		subl	-8(%ebp), %eax	46		ret	
21		movl	%eax, -20(%ebp)				
22		movl	-20(%ebp), %eax				
23		imull	%eax, %eax				
24		movl	%eax, -24(%ebp)				
25		movl	-12(%ebp), %edx				
26		movl	-16(%ebp), %eax				
27		addl	%edx, %eax				

Reverse Engineering

```
int main(void) {  
    for (int a = 1; _____) {  
        int a2 = _____;  
        for (int b = _____) {  
            int b2 = _____;  
            int c = _____;  
            int c2 = _____;  
            if (_____) {  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

2. What is the end condition for the outer loop using a?

1 main:	28 cmpl %eax, -24(%ebp)
2 pushl %ebp	29 jne .L4
3 movl %esp, %ebp	30 movl -4(%ebp), %eax
4 subl \$32, %esp	31 imull -8(%ebp), %eax
5 movl \$1, -4(%ebp)	32 imull -20(%ebp), %eax
6 jmp .L2	33 jmp .L5
7 .L7:	34 .L4:
8 movl -4(%ebp), %eax	35 addl \$1, -8(%ebp)
9 imull %eax, %eax	36 .L3:
10 movl %eax, -12(%ebp)	37 cmpl \$666, -8(%ebp)
11 movl -4(%ebp), %eax	38 jle .L6
12 movl %eax, -8(%ebp)	39 addl \$1, -4(%ebp)
13 jmp .L3	40 .L2:
14 .L6:	41 cmpl \$333, -4(%ebp)
15 movl -8(%ebp), %eax	42 jle .L7
16 imull %eax, %eax	43 movl \$0, %eax
17 movl %eax, -16(%ebp)	44 .L5:
18 movl \$1000, %eax	45 leave
19 subl -4(%ebp), %eax	46 ret
20 subl -8(%ebp), %eax	
21 movl %eax, -20(%ebp)	
22 movl -20(%ebp), %eax	
23 imull %eax, %eax	
24 movl %eax, -24(%ebp)	
25 movl -12(%ebp), %edx	
26 movl -16(%ebp), %eax	
27 addl %edx, %eax	

Reverse Engineering

```
int main(void) {  
    for (int a = 1; _____) {  
        int a2 = _____;  
        for (int b = _____) {  
            int b2 = _____;  
            int c = _____;  
            int c2 = _____;  
            if (_____) {  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

2. What is the end condition for the outer loop using a?
a must be greater than 333.

1 main:	28 cmpl %eax, -24(%ebp)
2 pushl %ebp	29 jne .L4
3 movl %esp, %ebp	30 movl -4(%ebp), %eax
4 subl \$32, %esp	31 imull -8(%ebp), %eax
5 movl \$1, -4(%ebp)	32 imull -20(%ebp), %eax
6 jmp .L2	33 jmp .L5
7 .L7:	34 .L4:
8 movl -4(%ebp), %eax	35 addl \$1, -8(%ebp)
9 imull %eax, %eax	36 .L3:
10 movl %eax, -12(%ebp)	37 cmpl \$666, -8(%ebp)
11 movl -4(%ebp), %eax	38 jle .L6
12 movl %eax, -8(%ebp)	39 addl \$1, -4(%ebp)
13 jmp .L3	40 .L2:
14 .L6:	41 cmpl \$333, -4(%ebp)
15 movl -8(%ebp), %eax	42 jle .L7
16 imull %eax, %eax	43 movl \$0, %eax
17 movl %eax, -16(%ebp)	44 .L5:
18 movl \$1000, %eax	45 leave
19 subl -4(%ebp), %eax	46 ret
20 subl -8(%ebp), %eax	
21 movl %eax, -20(%ebp)	
22 movl -20(%ebp), %eax	
23 imull %eax, %eax	
24 movl %eax, -24(%ebp)	
25 movl -12(%ebp), %edx	
26 movl -16(%ebp), %eax	
27 addl %edx, %eax	

Reverse Engineering

```
int main(void) {  
    for (int a = 1; _____) {  
        int a2 = _____;  
        for (int b = _____) {  
            int b2 = _____;  
            int c = _____;  
            int c2 = _____;  
            if (_____) {  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

1	main:		28	cmpl	%eax, -24(%ebp)
2		pushl	29	jne	.L4
3		movl	30	movl	-4(%ebp), %eax
4		subl	31	imull	-8(%ebp), %eax
5		movl	32	imull	-20(%ebp), %eax
6		jmp	33	jmp	.L5
7	.L7:		34	.L4:	
8		movl	35	addl	\$1, -8(%ebp)
9		imull	36	.L3:	
10		movl	37	cmpl	\$666, -8(%ebp)
11		movl	38	jle	.L6
12		movl	39	addl	\$1, -4(%ebp)
13		jmp	40	.L2:	
14	.L6:		41	cmpl	\$333, -4(%ebp)
15		movl	42	jle	.L7
16		imull	43	movl	\$0, %eax
17		movl	44	.L5:	
18		movl	45	leave	
19		subl	46	ret	
20		subl			
21		movl			
22		movl			
23		imull			
24		movl			
25		movl			
26		movl			
27		addl			

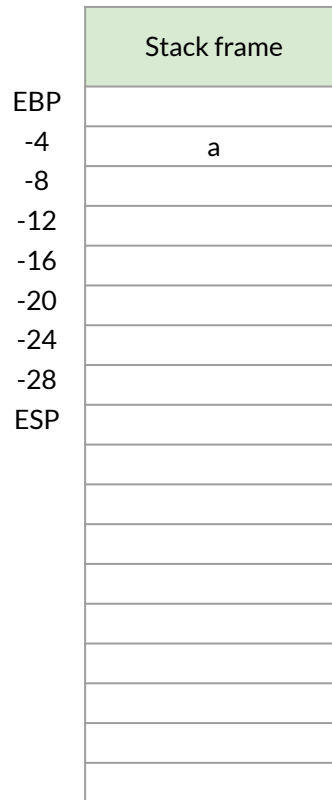
Reverse Engineering

```
int main(void) {
    for (int a = 1; _____) {
        int a2 = _____;
        for (int b = _____) {
            int b2 = _____;
            int c = _____;
            int c2 = _____;
            if (_____) {
                return _____;
            }
        }
    }
    return 0;
}
```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

```
.L2:
41      cmpl    $333, -4(%ebp)
42      jle     .L7
43      movl    $0, %eax
```

```
1 main:
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $32, %esp
5      movl    $1, -4(%ebp)
6      jmp     .L2
7 .L7:
8      movl    -4(%ebp), %eax
9      imull   %eax, %eax
10     movl    %eax, -12(%ebp)
11     movl    -4(%ebp), %eax
12     movl    %eax, -8(%ebp)
13     jmp     .L3
14 .L6:
15     movl    -8(%ebp), %eax
16     imull   %eax, %eax
17     movl    %eax, -16(%ebp)
18     movl    $1000, %eax
19     subl    -4(%ebp), %eax
20     subl    -8(%ebp), %eax
21     movl    %eax, -20(%ebp)
22     movl    -20(%ebp), %eax
23     imull   %eax, %eax
24     movl    %eax, -24(%ebp)
25     movl    -12(%ebp), %edx
26     movl    -16(%ebp), %eax
27     addl    %edx, %eax
```



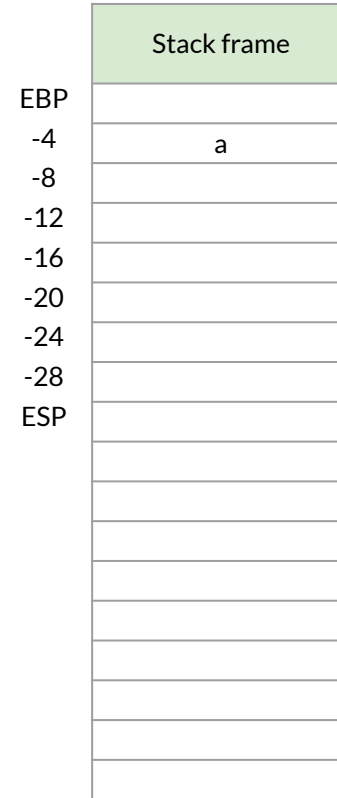
EAX	
EDX	
a	1

Reverse Engineering

```
int main(void) {  
    for (int a = 1; _____) {  
        int a2 = _____;  
        for (int b = _____) {  
            int b2 = _____;  
            int c = _____;  
            int c2 = _____;  
            if (_____){  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

```
1 main:  
2     pushl   %ebp  
3     movl    %esp, %ebp  
4     subl    $32, %esp  
5     movl    $1, -4(%ebp)  
6     jmp     .L2  
7 .L7:  
8     movl    -4(%ebp), %eax  
9     imull   %eax, %eax  
10    movl    %eax, -12(%ebp)  
11    movl    -4(%ebp), %eax  
12    movl    %eax, -8(%ebp)  
13    jmp     .L3  
14 .L6:  
15    movl    -8(%ebp), %eax  
16    imull   %eax, %eax  
17    movl    %eax, -16(%ebp)  
18    movl    $1000, %eax  
19    subl    -4(%ebp), %eax  
20    subl    -8(%ebp), %eax  
21    movl    %eax, -20(%ebp)  
22    movl    -20(%ebp), %eax  
23    imull   %eax, %eax  
24    movl    %eax, -24(%ebp)  
25    movl    -12(%ebp), %edx  
26    movl    -16(%ebp), %eax  
27    addl    %edx, %eax
```



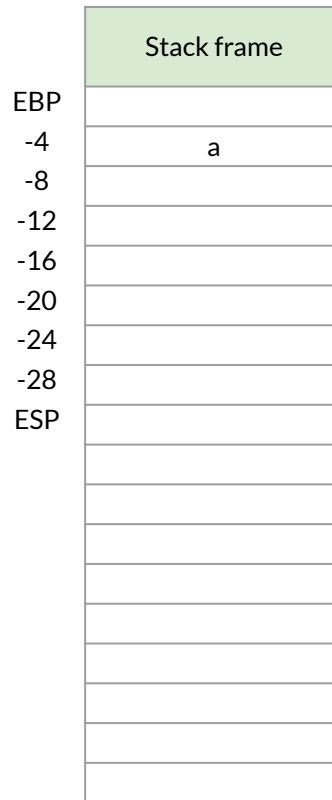
EAX	a
EDX	
a	1

Reverse Engineering

```
int main(void) {  
    for (int a = 1; _____) {  
        int a2 = _____;  
        for (int b = _____) {  
            int b2 = _____;  
            int c = _____;  
            int c2 = _____;  
            if (_____){  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

```
1 main:  
2     pushl   %ebp  
3     movl    %esp, %ebp  
4     subl    $32, %esp  
5     movl    $1, -4(%ebp)  
6     jmp     .L2  
7 .L7:  
8     movl    -4(%ebp), %eax  
9     imull   %eax, %eax  
10    movl    %eax, -12(%ebp)  
11    movl    -4(%ebp), %eax  
12    movl    %eax, -8(%ebp)  
13    jmp     .L3  
14 .L6:  
15    movl    -8(%ebp), %eax  
16    imull   %eax, %eax  
17    movl    %eax, -16(%ebp)  
18    movl    $1000, %eax  
19    subl    -4(%ebp), %eax  
20    subl    -8(%ebp), %eax  
21    movl    %eax, -20(%ebp)  
22    movl    -20(%ebp), %eax  
23    imull   %eax, %eax  
24    movl    %eax, -24(%ebp)  
25    movl    -12(%ebp), %edx  
26    movl    -16(%ebp), %eax  
27    addl    %edx, %eax
```



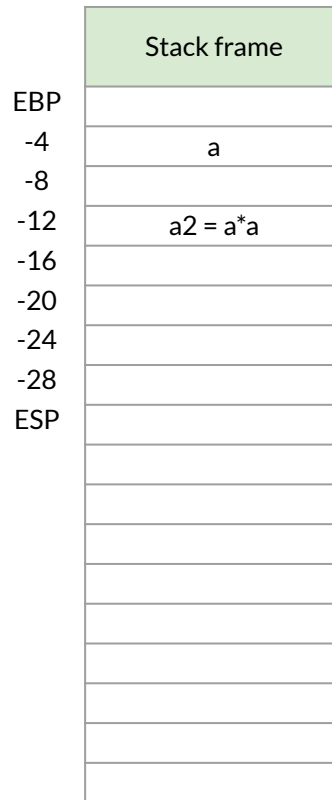
EAX	a*a
EDX	
a	1

Reverse Engineering

```
int main(void) {  
    for (int a = 1; _____) {  
        int a2 = _____;  
        for (int b = _____) {  
            int b2 = _____;  
            int c = _____;  
            int c2 = _____;  
            if (_____){  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

```
1 main:  
2     pushl   %ebp  
3     movl    %esp, %ebp  
4     subl    $32, %esp  
5     movl    $1, -4(%ebp)  
6     jmp     .L2  
7 .L7:  
8     movl    -4(%ebp), %eax  
9     imull   %eax, %eax  
10    movl    %eax, -12(%ebp)  
11    movl    -4(%ebp), %eax  
12    movl    %eax, -8(%ebp)  
13    jmp     .L3  
14 .L6:  
15    movl    -8(%ebp), %eax  
16    imull   %eax, %eax  
17    movl    %eax, -16(%ebp)  
18    movl    $1000, %eax  
19    subl    -4(%ebp), %eax  
20    subl    -8(%ebp), %eax  
21    movl    %eax, -20(%ebp)  
22    movl    -20(%ebp), %eax  
23    imull   %eax, %eax  
24    movl    %eax, -24(%ebp)  
25    movl    -12(%ebp), %edx  
26    movl    -16(%ebp), %eax  
27    addl    %edx, %eax
```



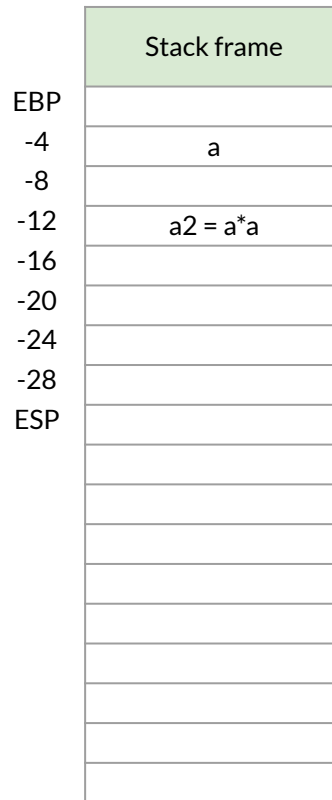
EAX	a*a
EDX	
a	1

Reverse Engineering

```
int main(void) {  
    for (int a = 1; _____) {  
        int a2 = _____;  
        for (int b = _____) {  
            int b2 = _____;  
            int c = _____;  
            int c2 = _____;  
            if (_____){  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

```
1 main:  
2     pushl   %ebp  
3     movl    %esp, %ebp  
4     subl    $32, %esp  
5     movl    $1, -4(%ebp)  
6     jmp     .L2  
7 .L7:  
8     movl    -4(%ebp), %eax  
9     imull   %eax, %eax  
10    movl    %eax, -12(%ebp)  
11    movl    -4(%ebp), %eax  
12    movl    %eax, -8(%ebp)  
13    jmp     .L3  
14 .L6:  
15    movl    -8(%ebp), %eax  
16    imull   %eax, %eax  
17    movl    %eax, -16(%ebp)  
18    movl    $1000, %eax  
19    subl    -4(%ebp), %eax  
20    subl    -8(%ebp), %eax  
21    movl    %eax, -20(%ebp)  
22    movl    -20(%ebp), %eax  
23    imull   %eax, %eax  
24    movl    %eax, -24(%ebp)  
25    movl    -12(%ebp), %edx  
26    movl    -16(%ebp), %eax  
27    addl    %edx, %eax
```



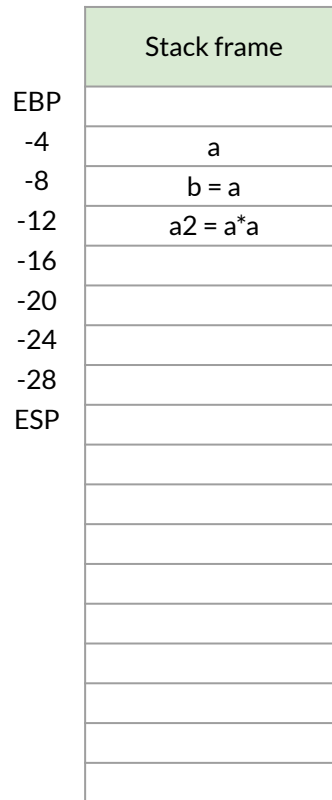
EAX	a
EDX	
a	1

Reverse Engineering

```
int main(void) {  
    for (int a = 1; _____) {  
        int a2 = _____;  
        for (int b = _____) {  
            int b2 = _____;  
            int c = _____;  
            int c2 = _____;  
            if (_____){  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

```
1 main:  
2     pushl   %ebp  
3     movl    %esp, %ebp  
4     subl    $32, %esp  
5     movl    $1, -4(%ebp)  
6     jmp     .L2  
7 .L7:  
8     movl    -4(%ebp), %eax  
9     imull   %eax, %eax  
10    movl    %eax, -12(%ebp)  
11    movl    -4(%ebp), %eax  
12    movl    %eax, -8(%ebp)  
13    jmp     .L3  
14 .L6:  
15    movl    -8(%ebp), %eax  
16    imull   %eax, %eax  
17    movl    %eax, -16(%ebp)  
18    movl    $1000, %eax  
19    subl    -4(%ebp), %eax  
20    subl    -8(%ebp), %eax  
21    movl    %eax, -20(%ebp)  
22    movl    -20(%ebp), %eax  
23    imull   %eax, %eax  
24    movl    %eax, -24(%ebp)  
25    movl    -12(%ebp), %edx  
26    movl    -16(%ebp), %eax  
27    addl    %edx, %eax
```



EAX	a
EDX	
a	1
b	a

Reverse Engineering

```
int main(void) {
    for (int a = 1; _____) {
        int a2 = _____;
        for (int b = _____) {
            int b2 = _____;
            int c = _____;
            int c2 = _____;
            if (_____){
                return _____;
            }
        }
    }
    return 0;
}
```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

```

36 .L3:
37     cmpl    $666, -8(%ebp)
38     jle     .L6
39     addl    $1, -4(%ebp)

```

```

1 main:
2     pushl    %ebp
3     movl    %esp, %ebp
4     subl    $32, %esp
5     movl    $1, -4(%ebp)
6     jmp     .L2
7 .L7:
8     movl    -4(%ebp), %eax
9     imull   %eax, %eax
10    movl    %eax, -12(%ebp)
11    movl    -4(%ebp), %eax
12    movl    %eax, -8(%ebp)
13    jmp     .L3
14 .L6:
15    movl    -8(%ebp), %eax
16    imull   %eax, %eax
17    movl    %eax, -16(%ebp)
18    movl    $1000, %eax
19    subl    -4(%ebp), %eax
20    subl    -8(%ebp), %eax
21    movl    %eax, -20(%ebp)
22    movl    -20(%ebp), %eax
23    imull   %eax, %eax
24    movl    %eax, -24(%ebp)
25    movl    -12(%ebp), %edx
26    movl    -16(%ebp), %eax
27    addl    %edx, %eax

```

[illegible]

EAX	b
EDX	
a	1
b	a

Stack visual credit: TA Diana!

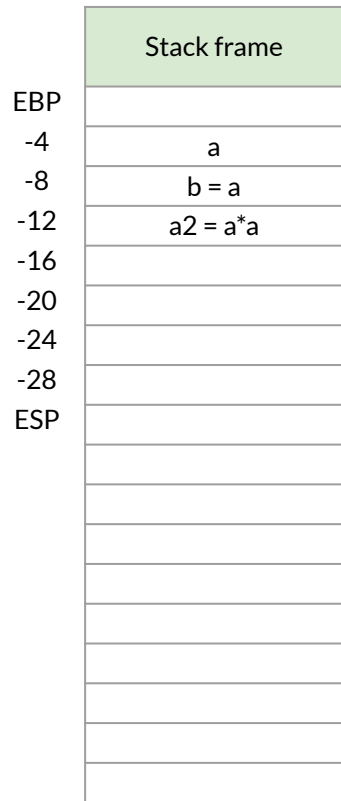
Reverse Engineering

```
int main(void) {  
    for (int a = 1; _____) {  
        int a2 = _____;  
        for (int b = _____) {  
            int b2 = _____;  
            int c = _____;  
            int c2 = _____;  
            if (_____){  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

```
36 .L3:  
37     cmpl    $666, -8(%ebp)  
38     jle     .L6  
39     addl    $1, -4(%ebp)
```

```
1 main:  
2     pushl   %ebp  
3     movl    %esp, %ebp  
4     subl    $32, %esp  
5     movl    $1, -4(%ebp)  
6     jmp     .L2  
7 .L7:  
8     movl    -4(%ebp), %eax  
9     imull   %eax, %eax  
10    movl    %eax, -12(%ebp)  
11    movl    -4(%ebp), %eax  
12    movl    %eax, -8(%ebp)  
13    jmp     .L3  
14 .L6:  
15    movl    -8(%ebp), %eax  
16    imull   %eax, %eax  
17    movl    %eax, -16(%ebp)  
18    movl    $1000, %eax  
19    subl    -4(%ebp), %eax  
20    subl    -8(%ebp), %eax  
21    movl    %eax, -20(%ebp)  
22    movl    -20(%ebp), %eax  
23    imull   %eax, %eax  
24    movl    %eax, -24(%ebp)  
25    movl    -12(%ebp), %edx  
26    movl    -16(%ebp), %eax  
27    addl    %edx, %eax
```



EAX	b*b
EDX	
a	1
b	a

Reverse Engineering

```
int main(void) {
    for (int a = 1; _____) {
        int a2 = _____;
        for (int b = _____) {
            int b2 = _____;
            int c = _____;
            int c2 = _____;
            if (_____){
                return _____;
            }
        }
    }
    return 0;
}
```

```

1 main:
2     pushl    %ebp
3     movl    %esp, %ebp
4     subl    $32, %esp
5     movl    $1, -4(%ebp)
6     jmp     .L2
7 .L7:
8     movl    -4(%ebp), %eax
9     imull   %eax, %eax
10    movl    %eax, -12(%ebp)
11    movl    -4(%ebp), %eax
12    movl    %eax, -8(%ebp)
13    jmp     .L3
14 .L6:
15    movl    -8(%ebp), %eax
16    imull   %eax, %eax
17    movl    %eax, -16(%ebp)
18    movl    $1000, %eax
19    subl    -4(%ebp), %eax
20    subl    -8(%ebp), %eax
21    movl    %eax, -20(%ebp)
22    movl    -20(%ebp), %eax
23    imull   %eax, %eax
24    movl    %eax, -24(%ebp)
25    movl    -12(%ebp), %edx
26    movl    -16(%ebp), %eax
27    addl    %edx, %eax

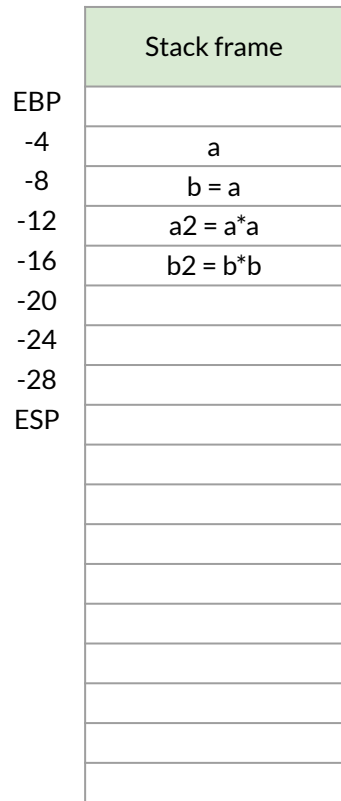
```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

```

36 .L3:
37     cmpl    $666, -8(%ebp)
38     jle     .L6
39     addl    $1, -4(%ebp)

```



EAX	b*b
EDX	
a	1
b	a

Stack visual credit: TA Diana!

Reverse Engineering

```
int main(void) {
    for (int a = 1; _____) {
        int a2 = _____;
        for (int b = _____) {
            int b2 = _____;
            int c = _____;
            int c2 = _____;
            if (_____){
                return _____;
            }
        }
    }
    return 0;
}
```

```

1 main:
2     pushl    %ebp
3     movl    %esp, %ebp
4     subl    $32, %esp
5     movl    $1, -4(%ebp)
6     jmp     .L2
7 .L7:
8     movl    -4(%ebp), %eax
9     imull   %eax, %eax
10    movl    %eax, -12(%ebp)
11    movl    -4(%ebp), %eax
12    movl    %eax, -8(%ebp)
13    jmp     .L3
14 .L6:
15    movl    -8(%ebp), %eax
16    imull   %eax, %eax
17    movl    %eax, -16(%ebp)
18    movl    $1000, %eax
19    subl    -4(%ebp), %eax
20    subl    -8(%ebp), %eax
21    movl    %eax, -20(%ebp)
22    movl    -20(%ebp), %eax
23    imull   %eax, %eax
24    movl    %eax, -24(%ebp)
25    movl    -12(%ebp), %edx
26    movl    -16(%ebp), %eax
27    addl    %edx, %eax

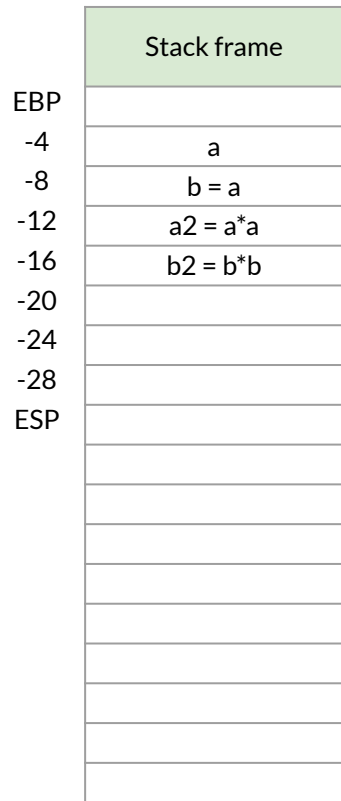
```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

```

36 .L3:
37     cmpl    $666, -8(%ebp)
38     jle     .L6
39     addl    $1, -4(%ebp)

```



EAX	1000
EDX	
a	1
b	a

Stack visual credit: TA Diana!

Reverse Engineering

```
int main(void) {
    for (int a = 1; _____) {
        int a2 = _____;
        for (int b = _____) {
            int b2 = _____;
            int c = _____;
            int c2 = _____;
            if (_____){
                return _____;
            }
        }
    }
    return 0;
}
```

```

1 main:
2     pushl    %ebp
3     movl    %esp, %ebp
4     subl    $32, %esp
5     movl    $1, -4(%ebp)
6     jmp     .L2
7 .L7:
8     movl    -4(%ebp), %eax
9     imull   %eax, %eax
10    movl    %eax, -12(%ebp)
11    movl    -4(%ebp), %eax
12    movl    %eax, -8(%ebp)
13    jmp     .L3
14 .L6:
15    movl    -8(%ebp), %eax
16    imull   %eax, %eax
17    movl    %eax, -16(%ebp)
18    movl    $1000, %eax
19    subl    -4(%ebp), %eax
20    subl    -8(%ebp), %eax
21    movl    %eax, -20(%ebp)
22    movl    -20(%ebp), %eax
23    imull   %eax, %eax
24    movl    %eax, -24(%ebp)
25    movl    -12(%ebp), %edx
26    movl    -16(%ebp), %eax
27    addl    %edx, %eax

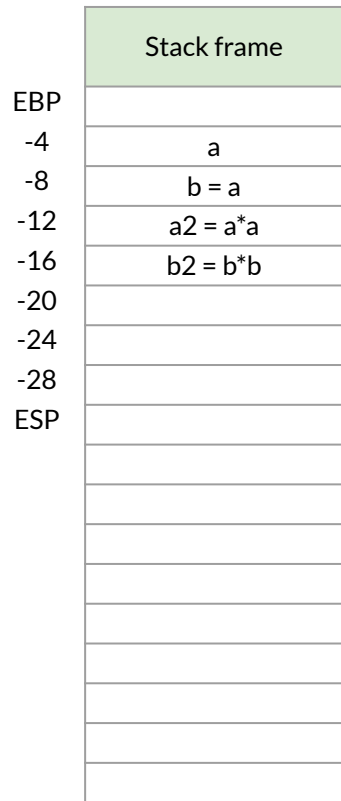
```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

```

36 .L3:
37     cmpl    $666, -8(%ebp)
38     jle     .L6
39     addl    $1, -4(%ebp)

```



EAX	1000-a
EDX	
a	1
b	a

Stack visual credit: TA Diana!

Reverse Engineering

```
int main(void) {
    for (int a = 1; _____) {
        int a2 = _____;
        for (int b = _____) {
            int b2 = _____;
            int c = _____;
            int c2 = _____;
            if (_____){
                return _____;
            }
        }
    }
    return 0;
}
```

```

1 main:
2     pushl    %ebp
3     movl    %esp, %ebp
4     subl    $32, %esp
5     movl    $1, -4(%ebp)
6     jmp     .L2
7 .L7:
8     movl    -4(%ebp), %eax
9     imull   %eax, %eax
10    movl    %eax, -12(%ebp)
11    movl    -4(%ebp), %eax
12    movl    %eax, -8(%ebp)
13    jmp     .L3
14 .L6:
15    movl    -8(%ebp), %eax
16    imull   %eax, %eax
17    movl    %eax, -16(%ebp)
18    movl    $1000, %eax
19    subl    -4(%ebp), %eax
20    subl    -8(%ebp), %eax
21    movl    %eax, -20(%ebp)
22    movl    -20(%ebp), %eax
23    imull   %eax, %eax
24    movl    %eax, -24(%ebp)
25    movl    -12(%ebp), %edx
26    movl    -16(%ebp), %eax
27    addl    %edx, %eax

```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

[illegible]

EAX	1000-a-b
EDX	
a	1
b	a

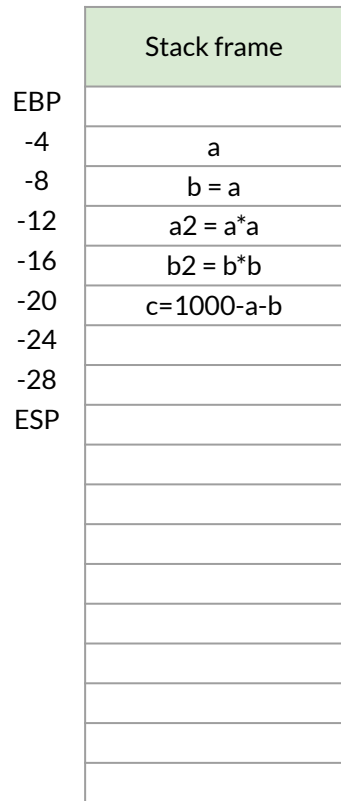
Stack visual credit: TA Diana!

Reverse Engineering

```
int main(void) {  
    for (int a = 1; _____) {  
        int a2 = _____;  
        for (int b = _____) {  
            int b2 = _____;  
            int c = _____;  
            int c2 = _____;  
            if (_____){  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

```
1 main:  
2     pushl   %ebp  
3     movl    %esp, %ebp  
4     subl    $32, %esp  
5     movl    $1, -4(%ebp)  
6     jmp     .L2  
7 .L7:  
8     movl    -4(%ebp), %eax  
9     imull   %eax, %eax  
10    movl    %eax, -12(%ebp)  
11    movl    -4(%ebp), %eax  
12    movl    %eax, -8(%ebp)  
13    jmp     .L3  
14 .L6:  
15    movl    -8(%ebp), %eax  
16    imull   %eax, %eax  
17    movl    %eax, -16(%ebp)  
18    movl    $1000, %eax  
19    subl    -4(%ebp), %eax  
20    subl    -8(%ebp), %eax  
21    movl    %eax, -20(%ebp)  
22    movl    -20(%ebp), %eax  
23    imull   %eax, %eax  
24    movl    %eax, -24(%ebp)  
25    movl    -12(%ebp), %edx  
26    movl    -16(%ebp), %eax  
27    addl    %edx, %eax
```



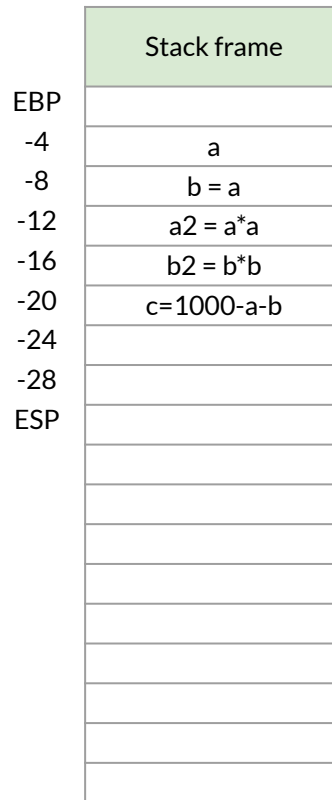
EAX	1000-a-b
EDX	
a	1
b	a

Reverse Engineering

```
int main(void) {  
    for (int a = 1; _____) {  
        int a2 = _____;  
        for (int b = _____) {  
            int b2 = _____;  
            int c = _____;  
            int c2 = _____;  
            if (_____){  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

```
1 main:  
2     pushl   %ebp  
3     movl    %esp, %ebp  
4     subl    $32, %esp  
5     movl    $1, -4(%ebp)  
6     jmp     .L2  
7 .L7:  
8     movl    -4(%ebp), %eax  
9     imull   %eax, %eax  
10    movl    %eax, -12(%ebp)  
11    movl    -4(%ebp), %eax  
12    movl    %eax, -8(%ebp)  
13    jmp     .L3  
14 .L6:  
15    movl    -8(%ebp), %eax  
16    imull   %eax, %eax  
17    movl    %eax, -16(%ebp)  
18    movl    $1000, %eax  
19    subl    -4(%ebp), %eax  
20    subl    -8(%ebp), %eax  
21    movl    %eax, -20(%ebp)  
22    movl    -20(%ebp), %eax  
23    imull   %eax, %eax  
24    movl    %eax, -24(%ebp)  
25    movl    -12(%ebp), %edx  
26    movl    -16(%ebp), %eax  
27    addl    %edx, %eax
```



EAX	1000-a-b
EDX	
a	1
b	a

Reverse Engineering

```
int main(void) {
    for (int a = 1; _____) {
        int a2 = _____;
        for (int b = _____) {
            int b2 = _____;
            int c = _____;
            int c2 = _____;
            if (_____){
                return _____;
            }
        }
    }
    return 0;
}
```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

```

1 main:
2     pushl    %ebp
3     movl    %esp, %ebp
4     subl    $32, %esp
5     movl    $1, -4(%ebp)
6     jmp     .L2
7 .L7:
8     movl    -4(%ebp), %eax
9     imull   %eax, %eax
10    movl    %eax, -12(%ebp)
11    movl    -4(%ebp), %eax
12    movl    %eax, -8(%ebp)
13    jmp     .L3
14 .L6:
15    movl    -8(%ebp), %eax
16    imull   %eax, %eax
17    movl    %eax, -16(%ebp)
18    movl    $1000, %eax
19    subl    -4(%ebp), %eax
20    subl    -8(%ebp), %eax
21    movl    %eax, -20(%ebp)
22    movl    -20(%ebp), %eax
23    imull   %eax, %eax
24    movl    %eax, -24(%ebp)
25    movl    -12(%ebp), %edx
26    movl    -16(%ebp), %eax
27    addl    %edx, %eax

```

[illegible]

EAX	c
EDX	
a	1
b	a

Stack visual credit: TA Diana!

Reverse Engineering

```
int main(void) {
    for (int a = 1; _____) {
        int a2 = _____;
        for (int b = _____) {
            int b2 = _____;
            int c = _____;
            int c2 = _____;
            if (_____){
                return _____;
            }
        }
    }
    return 0;
}
```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

```

1 main:
2     pushl    %ebp
3     movl    %esp, %ebp
4     subl    $32, %esp
5     movl    $1, -4(%ebp)
6     jmp     .L2
7 .L7:
8     movl    -4(%ebp), %eax
9     imull   %eax, %eax
10    movl    %eax, -12(%ebp)
11    movl    -4(%ebp), %eax
12    movl    %eax, -8(%ebp)
13    jmp     .L3
14 .L6:
15    movl    -8(%ebp), %eax
16    imull   %eax, %eax
17    movl    %eax, -16(%ebp)
18    movl    $1000, %eax
19    subl    -4(%ebp), %eax
20    subl    -8(%ebp), %eax
21    movl    %eax, -20(%ebp)
22    movl    -20(%ebp), %eax
23    imull   %eax, %eax
24    movl    %eax, -24(%ebp)
25    movl    -12(%ebp), %edx
26    movl    -16(%ebp), %eax
27    addl    %edx, %eax

```

[illegible]

EAX	c*c
EDX	
a	1
b	a

Stack visual credit: TA Diana!

Reverse Engineering

```
int main(void) {  
    for (int a = 1; _____) {  
        int a2 = _____;  
        for (int b = _____) {  
            int b2 = _____;  
            int c = _____;  
            int c2 = _____;  
            if (_____){  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

```
1 main:  
2     pushl   %ebp  
3     movl    %esp, %ebp  
4     subl    $32, %esp  
5     movl    $1, -4(%ebp)  
6     jmp     .L2  
7 .L7:  
8     movl    -4(%ebp), %eax  
9     imull   %eax, %eax  
10    movl    %eax, -12(%ebp)  
11    movl    -4(%ebp), %eax  
12    movl    %eax, -8(%ebp)  
13    jmp     .L3  
14 .L6:  
15    movl    -8(%ebp), %eax  
16    imull   %eax, %eax  
17    movl    %eax, -16(%ebp)  
18    movl    $1000, %eax  
19    subl    -4(%ebp), %eax  
20    subl    -8(%ebp), %eax  
21    movl    %eax, -20(%ebp)  
22    movl    -20(%ebp), %eax  
23    imull   %eax, %eax  
24    movl    %eax, -24(%ebp)  
25    movl    -12(%ebp), %edx  
26    movl    -16(%ebp), %eax  
27    addl    %edx, %eax
```

	Stack frame
EBP	
-4	a
-8	b = a
-12	a2 = a*a
-16	b2 = b*b
-20	c=1000-a-b
-24	c2=c*c
-28	
ESP	

EAX	c*c
EDX	
a	1
b	a

Reverse Engineering

```
int main(void) {  
    for (int a = 1; _____) {  
        int a2 = _____;  
        for (int b = _____) {  
            int b2 = _____;  
            int c = _____;  
            int c2 = _____;  
            if (_____){  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

3. What are the memory addresses the local variables (a2, b, b2, c, c2) relative to the base pointer?

a2 = EBP - 12

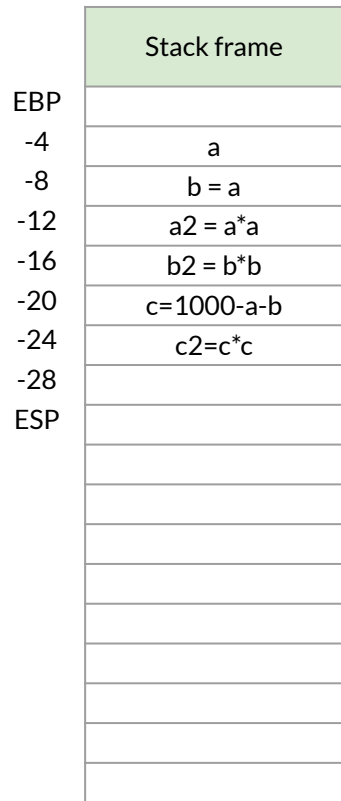
b = EBP - 8

b2 = EBP - 16

c = EBP - 20

c2 = EBP - 24

```
1 main:  
2     pushl    %ebp  
3     movl     %esp, %ebp  
4     subl     $32, %esp  
5     movl     $1, -4(%ebp)  
6     jmp      .L2  
7 .L7:  
8     movl     -4(%ebp), %eax  
9     imull    %eax, %eax  
10    movl     %eax, -12(%ebp)  
11    movl     -4(%ebp), %eax  
12    movl     %eax, -8(%ebp)  
13    jmp      .L3  
14 .L6:  
15    movl     -8(%ebp), %eax  
16    imull    %eax, %eax  
17    movl     %eax, -16(%ebp)  
18    movl     $1000, %eax  
19    subl     -4(%ebp), %eax  
20    subl     -8(%ebp), %eax  
21    movl     %eax, -20(%ebp)  
22    movl     -20(%ebp), %eax  
23    imull    %eax, %eax  
24    movl     %eax, -24(%ebp)  
25    movl     -12(%ebp), %edx  
26    movl     -16(%ebp), %eax  
27    addl     %edx, %eax
```



EAX	c*c
EDX	
a	1
b	a

Reverse Engineering

```
int main(void) {  
    for (int a = 1; _____) {  
        int a2 = _____;  
        for (int b = _____) {  
            int b2 = _____;  
            int c = _____;  
            int c2 = _____;  
            if (_____){  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

4. Fill in the missing code.

1	main:			28	cmpl	%eax, -24(%ebp)
2		pushl	%ebp	29	jne	.L4
3		movl	%esp, %ebp	30	movl	-4(%ebp), %eax
4		subl	\$32, %esp	31	imull	-8(%ebp), %eax
5		movl	\$1, -4(%ebp)	32	imull	-20(%ebp), %eax
6		jmp	.L2	33	jmp	.L5
7	.L7:			34	.L4:	
8		movl	-4(%ebp), %eax	35	addl	\$1, -8(%ebp)
9		imull	%eax, %eax	36	.L3:	
10		movl	%eax, -12(%ebp)	37	cmpl	\$666, -8(%ebp)
11		movl	-4(%ebp), %eax	38	jle	.L6
12		movl	%eax, -8(%ebp)	39	addl	\$1, -4(%ebp)
13		jmp	.L3	40	.L2:	
14	.L6:			41	cmpl	\$333, -4(%ebp)
15		movl	-8(%ebp), %eax	42	jle	.L7
16		imull	%eax, %eax	43	movl	\$0, %eax
17		movl	%eax, -16(%ebp)	44	.L5:	
18		movl	\$1000, %eax	45	leave	
19		subl	-4(%ebp), %eax	46	ret	
20		subl	-8(%ebp), %eax			
21		movl	%eax, -20(%ebp)			
22		movl	-20(%ebp), %eax			
23		imull	%eax, %eax			
24		movl	%eax, -24(%ebp)			
25		movl	-12(%ebp), %edx			
26		movl	-16(%ebp), %eax			
27		addl	%edx, %eax			

Reverse Engineering

```
int main(void) {  
    for (int a = 1; a <= 333; a++) {  
        int a2 = ____;  
        for (int b = _____) {  
            int b2 = ____;  
            int c = _____;  
            int c2 = ____;  
            if (_____) {  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

4. Fill in the missing code.

1	main:			28	cmpl	%eax, -24(%ebp)
2		pushl	%ebp	29	jne	.L4
3		movl	%esp, %ebp	30	movl	-4(%ebp), %eax
4		subl	\$32, %esp	31	imull	-8(%ebp), %eax
5		movl	\$1, -4(%ebp)	32	imull	-20(%ebp), %eax
6		jmp	.L2	33	jmp	.L5
7	.L7:			34	.L4:	
8		movl	-4(%ebp), %eax	35	addl	\$1, -8(%ebp)
9		imull	%eax, %eax	36	.L3:	
10		movl	%eax, -12(%ebp)	37	cmpl	\$666, -8(%ebp)
11		movl	-4(%ebp), %eax	38	jle	.L6
12		movl	%eax, -8(%ebp)	39	addl	\$1, -4(%ebp)
13		jmp	.L3	40	.L2:	
14	.L6:			41	cmpl	\$333, -4(%ebp)
15		movl	-8(%ebp), %eax	42	jle	.L7
16		imull	%eax, %eax	43	movl	\$0, %eax
17		movl	%eax, -16(%ebp)	44	.L5:	
18		movl	\$1000, %eax	45	leave	
19		subl	-4(%ebp), %eax	46	ret	
20		subl	-8(%ebp), %eax			
21		movl	%eax, -20(%ebp)			
22		movl	-20(%ebp), %eax			
23		imull	%eax, %eax			
24		movl	%eax, -24(%ebp)			
25		movl	-12(%ebp), %edx			
26		movl	-16(%ebp), %eax			
27		addl	%edx, %eax			

Reverse Engineering

```
int main(void) {  
    for (int a = 1; a <= 333; a++) {  
        int a2 = a * a;  
        for (int b = _____) {  
            int b2 = _____;  
            int c = _____;  
            int c2 = _____;  
            if (_____){  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

4. Fill in the missing code.

1	main:			28	cmpl	%eax, -24(%ebp)
2		pushl	%ebp	29	jne	.L4
3		movl	%esp, %ebp	30	movl	-4(%ebp), %eax
4		subl	\$32, %esp	31	imull	-8(%ebp), %eax
5		movl	\$1, -4(%ebp)	32	imull	-20(%ebp), %eax
6		jmp	.L2	33	jmp	.L5
7	.L7:			34	.L4:	
8		movl	-4(%ebp), %eax	35	addl	\$1, -8(%ebp)
9		imull	%eax, %eax	36	.L3:	
10		movl	%eax, -12(%ebp)	37	cmpl	\$666, -8(%ebp)
11		movl	-4(%ebp), %eax	38	jle	.L6
12		movl	%eax, -8(%ebp)	39	addl	\$1, -4(%ebp)
13		jmp	.L3	40	.L2:	
14	.L6:			41	cmpl	\$333, -4(%ebp)
15		movl	-8(%ebp), %eax	42	jle	.L7
16		imull	%eax, %eax	43	movl	\$0, %eax
17		movl	%eax, -16(%ebp)	44	.L5:	
18		movl	\$1000, %eax	45	leave	
19		subl	-4(%ebp), %eax	46	ret	
20		subl	-8(%ebp), %eax			
21		movl	%eax, -20(%ebp)			
22		movl	-20(%ebp), %eax			
23		imull	%eax, %eax			
24		movl	%eax, -24(%ebp)			
25		movl	-12(%ebp), %edx			
26		movl	-16(%ebp), %eax			
27		addl	%edx, %eax			

Reverse Engineering

```
int main(void) {  
    for (int a = 1; a <= 333; a++) {  
        int a2 = a * a;  
        for (int b = a; b <= 666; b++) {  
            int b2 = _____;  
            int c = _____;  
            int c2 = _____;  
            if (_____) {  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

4. Fill in the missing code.

1	main:			28	cmpl	%eax, -24(%ebp)
2		pushl	%ebp	29	jne	.L4
3		movl	%esp, %ebp	30	movl	-4(%ebp), %eax
4		subl	\$32, %esp	31	imull	-8(%ebp), %eax
5		movl	\$1, -4(%ebp)	32	imull	-20(%ebp), %eax
6		jmp	.L2	33	jmp	.L5
7	.L7:			34	.L4:	
8		movl	-4(%ebp), %eax	35	addl	\$1, -8(%ebp)
9		imull	%eax, %eax	36	.L3:	
10		movl	%eax, -12(%ebp)	37	cmpl	\$666, -8(%ebp)
11		movl	-4(%ebp), %eax	38	jle	.L6
12		movl	%eax, -8(%ebp)	39	addl	\$1, -4(%ebp)
13		jmp	.L3	40	.L2:	
14	.L6:			41	cmpl	\$333, -4(%ebp)
15		movl	-8(%ebp), %eax	42	jle	.L7
16		imull	%eax, %eax	43	movl	\$0, %eax
17		movl	%eax, -16(%ebp)	44	.L5:	
18		movl	\$1000, %eax	45	leave	
19		subl	-4(%ebp), %eax	46	ret	
20		subl	-8(%ebp), %eax			
21		movl	%eax, -20(%ebp)			
22		movl	-20(%ebp), %eax			
23		imull	%eax, %eax			
24		movl	%eax, -24(%ebp)			
25		movl	-12(%ebp), %edx			
26		movl	-16(%ebp), %eax			
27		addl	%edx, %eax			

Reverse Engineering

```
int main(void) {  
    for (int a = 1; a <= 333; a++) {  
        int a2 = a * a;  
        for (int b = a; b <= 666; b++) {  
            int b2 = b * b;  
            int c = _____;  
            int c2 = _____;  
            if (_____) {  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

4. Fill in the missing code.

```
1 main:  
2     pushl    %ebp  
3     movl    %esp, %ebp  
4     subl    $32, %esp  
5     movl    $1, -4(%ebp)  
6     jmp     .L2  
7 .L7:  
8     movl    -4(%ebp), %eax  
9     imull    %eax, %eax  
10    movl    %eax, -12(%ebp)  
11    movl    -4(%ebp), %eax  
12    movl    %eax, -8(%ebp)  
13    jmp     .L3  
14 .L6:  
15    movl    -8(%ebp), %eax  
16    imull    %eax, %eax  
17    movl    %eax, -16(%ebp)  
18    movl    $1000, %eax  
19    subl    -4(%ebp), %eax  
20    subl    -8(%ebp), %eax  
21    movl    %eax, -20(%ebp)  
22    movl    -20(%ebp), %eax  
23    imull    %eax, %eax  
24    movl    %eax, -24(%ebp)  
25    movl    -12(%ebp), %edx  
26    movl    -16(%ebp), %eax  
27    addl    %edx, %eax  
  
28    cmpl    %eax, -24(%ebp)  
29    jne     .L4  
30    movl    -4(%ebp), %eax  
31    imull    -8(%ebp), %eax  
32    imull    -20(%ebp), %eax  
33    jmp     .L5  
34 .L4:  
35    addl    $1, -8(%ebp)  
36 .L3:  
37    cmpl    $666, -8(%ebp)  
38    jle     .L6  
39    addl    $1, -4(%ebp)  
40 .L2:  
41    cmpl    $333, -4(%ebp)  
42    jle     .L7  
43    movl    $0, %eax  
44 .L5:  
45    leave  
46    ret
```

Reverse Engineering

```
int main(void) {  
    for (int a = 1; a <= 333; a++) {  
        int a2 = a * a;  
        for (int b = a; b <= 666; b++) {  
            int b2 = b * b;  
            int c = 1000 - a - b;  
            int c2 = _____;  
            if (_____) {  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

4. Fill in the missing code.

1	main:		28	cmpl	%eax, -24(%ebp)
2	pushl	%ebp	29	jne	.L4
3	movl	%esp, %ebp	30	movl	-4(%ebp), %eax
4	subl	\$32, %esp	31	imull	-8(%ebp), %eax
5	movl	\$1, -4(%ebp)	32	imull	-20(%ebp), %eax
6	jmp	.L2	33	jmp	.L5
7	.L7:		34	.L4:	
8	movl	-4(%ebp), %eax	35	addl	\$1, -8(%ebp)
9	imull	%eax, %eax	36	.L3:	
10	movl	%eax, -12(%ebp)	37	cmpl	\$666, -8(%ebp)
11	movl	-4(%ebp), %eax	38	jle	.L6
12	movl	%eax, -8(%ebp)	39	addl	\$1, -4(%ebp)
13	jmp	.L3	40	.L2:	
14	.L6:		41	cmpl	\$333, -4(%ebp)
15	movl	-8(%ebp), %eax	42	jle	.L7
16	imull	%eax, %eax	43	movl	\$0, %eax
17	movl	%eax, -16(%ebp)	44	.L5:	
18	movl	\$1000, %eax	45	leave	
19	subl	-4(%ebp), %eax	46	ret	
20	subl	-8(%ebp), %eax			
21	movl	%eax, -20(%ebp)			
22	movl	-20(%ebp), %eax			
23	imull	%eax, %eax			
24	movl	%eax, -24(%ebp)			
25	movl	-12(%ebp), %edx			
26	movl	-16(%ebp), %eax			
27	addl	%edx, %eax			

Reverse Engineering

```
int main(void) {  
    for (int a = 1; a <= 333; a++) {  
        int a2 = a * a;  
        for (int b = a; b <= 666; b++) {  
            int b2 = b * b;  
            int c = 1000 - a - b;  
            int c2 = c * c;  
            if (_____) {  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

4. Fill in the missing code.

```
1 main:  
2     pushl    %ebp  
3     movl    %esp, %ebp  
4     subl    $32, %esp  
5     movl    $1, -4(%ebp)  
6     jmp     .L2  
7 .L7:  
8     movl    -4(%ebp), %eax  
9     imull    %eax, %eax  
10    movl    %eax, -12(%ebp)  
11    movl    -4(%ebp), %eax  
12    movl    %eax, -8(%ebp)  
13    jmp     .L3  
14 .L6:  
15    movl    -8(%ebp), %eax  
16    imull    %eax, %eax  
17    movl    %eax, -16(%ebp)  
18    movl    $1000, %eax  
19    subl    -4(%ebp), %eax  
20    subl    -8(%ebp), %eax  
21    movl    %eax, -20(%ebp)  
22    movl    -20(%ebp), %eax  
23    imull    %eax, %eax  
24    movl    %eax, -24(%ebp)  
25    movl    -12(%ebp), %edx  
26    movl    -16(%ebp), %eax  
27    addl    %edx, %eax  
28    cmpl    %eax, -24(%ebp)  
29    jne     .L4  
30    movl    -4(%ebp), %eax  
31    imull    -8(%ebp), %eax  
32    imull    -20(%ebp), %eax  
33    jmp     .L5  
34 .L4:  
35    addl    $1, -8(%ebp)  
36 .L3:  
37    cmpl    $666, -8(%ebp)  
38    jle     .L6  
39    addl    $1, -4(%ebp)  
40 .L2:  
41    cmpl    $333, -4(%ebp)  
42    jle     .L7  
43    movl    $0, %eax  
44 .L5:  
45    leave  
46    ret
```

Reverse Engineering

```
int main(void) {  
    for (int a = 1; a <= 333; a++) {  
        int a2 = a * a;  
        for (int b = a; b <= 666; b++) {  
            int b2 = b * b;  
            int c = 1000 - a - b;  
            int c2 = c * c;  
            if (a2 + b2 == c2) {  
                return _____;  
            }  
        }  
    }  
    return 0;  
}
```

4. Fill in the missing code.

1	main:		28	cmpl	%eax, -24(%ebp)
2		pushl	29	jne	.L4
3		movl	30	movl	-4(%ebp), %eax
4		subl	31	imull	-8(%ebp), %eax
5		movl	32	imull	-20(%ebp), %eax
6		jmp	33	jmp	.L5
7	.L7:		34	.L4:	
8		movl	35	addl	\$1, -8(%ebp)
9		imull	36	.L3:	
10		movl	37	cmpl	\$666, -8(%ebp)
11		movl	38	jle	.L6
12		movl	39	addl	\$1, -4(%ebp)
13		jmp	40	.L2:	
14	.L6:		41	cmpl	\$333, -4(%ebp)
15		movl	42	jle	.L7
16		imull	43	movl	\$0, %eax
17		movl	44	.L5:	
18		movl	45	leave	
19		subl	46	ret	
20		subl			
21		movl			
22		movl			
23		imull			
24		movl			
25		movl			
26		movl			
27		addl			

Reverse Engineering

```
int main(void) {  
    for (int a = 1; a <= 333; a++) {  
        int a2 = a * a;  
        for (int b = a; b <= 666; b++) {  
            int b2 = b * b;  
            int c = 1000 - a - b;  
            int c2 = c * c;  
            if (a2 + b2 == c2) {  
                return a * b * c;  
            }  
        }  
    }  
    return 0;  
}
```

4. Fill in the missing code.

1	main:		28	cmpl	%eax, -24(%ebp)
2	pushl	%ebp	29	jne	.L4
3	movl	%esp, %ebp	30	movl	-4(%ebp), %eax
4	subl	\$32, %esp	31	imull	-8(%ebp), %eax
5	movl	\$1, -4(%ebp)	32	imull	-20(%ebp), %eax
6	jmp	.L2	33	jmp	.L5
7	.L7:		34	.L4:	
8	movl	-4(%ebp), %eax	35	addl	\$1, -8(%ebp)
9	imull	%eax, %eax	36	.L3:	
10	movl	%eax, -12(%ebp)	37	cmpl	\$666, -8(%ebp)
11	movl	-4(%ebp), %eax	38	jle	.L6
12	movl	%eax, -8(%ebp)	39	addl	\$1, -4(%ebp)
13	jmp	.L3	40	.L2:	
14	.L6:		41	cmpl	\$333, -4(%ebp)
15	movl	-8(%ebp), %eax	42	jle	.L7
16	imull	%eax, %eax	43	movl	\$0, %eax
17	movl	%eax, -16(%ebp)	44	.L5:	
18	movl	\$1000, %eax	45	leave	
19	subl	-4(%ebp), %eax	46	ret	
20	subl	-8(%ebp), %eax			
21	movl	%eax, -20(%ebp)			
22	movl	-20(%ebp), %eax			
23	imull	%eax, %eax			
24	movl	%eax, -24(%ebp)			
25	movl	-12(%ebp), %edx			
26	movl	-16(%ebp), %eax			
27	addl	%edx, %eax			

Stack Frame

1	p:		call.s	27	pushl	\$3
2		.zero	4	28	call	bar
3	bar:			29	addl	\$16, %esp
4		pushl	%ebp	30	addl	%ebx, %eax
5		movl	%esp, %ebp	31	movl	%eax, p
6		subl	\$16, %esp	32	nop	
7		movl	8(%ebp), %edx	33	movl	-4(%ebp), %ebx
8		movl	12(%ebp), %eax	34	leave	
9		addl	%edx, %eax	35	ret	
10		subl	16(%ebp), %eax			
11		movl	%eax, -4(%ebp)			
12		movl	-4(%ebp), %eax			
13		addl	\$1, %eax			
14		leave				
15		ret				
16	foo:					
17		pushl	%ebp			
18		movl	%esp, %ebp			
19		pushl	%ebx			
20		subl	\$4, %esp			
21		movl	8(%ebp), %edx			
22		movl	12(%ebp), %eax			
23		leal	(%edx,%eax), %ebx			
24		subl	\$4, %esp			
25		pushl	\$5			
26		pushl	\$4			

1. Which lines of code correspond to a caller/callee prologue?

Stack Frame

```
call.s
1 p:
2 .zero 4
3 bar:
4 pushl %ebp
5 movl %esp, %ebp
6 subl $16, %esp
7 movl 8(%ebp), %edx
8 movl 12(%ebp), %eax
9 addl %edx, %eax
10 subl 16(%ebp), %eax
11 movl %eax, -4(%ebp)
12 movl -4(%ebp), %eax
13 addl $1, %eax
14 leave
15 ret
16 foo:
17 pushl %ebp
18 movl %esp, %ebp
19 pushl %ebx
20 subl $4, %esp
21 movl 8(%ebp), %edx
22 movl 12(%ebp), %eax
23 leal (%edx,%eax), %ebx
24 subl $4, %esp
25 pushl $5
26 pushl $4
```

```
27 pushl $3
28 call bar
29 addl $16, %esp
30 addl %ebx, %eax
31 movl %eax, p
32 nop
33 movl -4(%ebp), %ebx
34 leave
35 ret
```

1. Which lines of code correspond to a caller/callee prologue?

foo/bar as caller/callee

24-27 = caller prologue

29 = caller epilogue

4-6 = callee prologue

13-15 = callee epilogue

foo as callee

17-20 = callee prologue

33-35 = callee epilogue

Stack Frame

```

1  p:
2      .zero 4
3  bar:
4      pushl %ebp
5      movl  %esp, %ebp
6      subl  $16, %esp
7      movl  8(%ebp), %edx
8      movl  12(%ebp), %eax
9      addl  %edx, %eax
10     subl  16(%ebp), %eax
11     movl  %eax, -4(%ebp)
12     movl  -4(%ebp), %eax
13     addl  $1, %eax
14     leave
15     ret
16 foo:
17     pushl %ebp
18     movl  %esp, %ebp
19     pushl %ebx
20     subl  $4, %esp
21     movl  8(%ebp), %edx
22     movl  12(%ebp), %eax
23     leal  (%edx,%eax), %ebx
24     subl  $4, %esp
25     pushl $5
26     pushl $4
27     pushl $3
28     call  bar
29     addl  $16, %esp
30     addl  %ebx, %eax
31     movl  %eax, p
32     nop
33     movl  -4(%ebp), %ebx
34     leave
35     ret
```

2. What does line 19 do in `call.s`? Why is it necessary?

Stack Frame

```

                                call.s
1  p:
2      .zero    4
3  bar:
4      pushl    %ebp
5      movl     %esp, %ebp
6      subl     $16, %esp
7      movl     8(%ebp), %edx
8      movl     12(%ebp), %eax
9      addl     %edx, %eax
10     subl     16(%ebp), %eax
11     movl     %eax, -4(%ebp)
12     movl     -4(%ebp), %eax
13     addl     $1, %eax
14     leave
15     ret
16 foo:
17     pushl     %ebp
18     movl     %esp, %ebp
19     pushl     %ebx
20     subl     $4, %esp
21     movl     8(%ebp), %edx
22     movl     12(%ebp), %eax
23     leal     (%edx,%eax), %ebx
24     subl     $4, %esp
25     pushl     $5
26     pushl     $4
                                27     pushl     $3
                                28     call     bar
                                29     addl     $16, %esp
                                30     addl     %ebx, %eax
                                31     movl     %eax, p
                                32     nop
                                33     movl     -4(%ebp), %ebx
                                34     leave
                                35     ret
```

2. What does line 19 do in `call.s`? Why is it necessary?

Saves EBX register since it's callee-saved and foo uses it. Doesn't matter that bar never uses EBX.

Stack Frame

1	p:		call.s	27	pushl	\$3
2		.zero	4	28	call	bar
3	bar:			29	addl	\$16, %esp
4		pushl	%ebp	30	addl	%ebx, %eax
5		movl	%esp, %ebp	31	movl	%eax, p
6		subl	\$16, %esp	32	nop	
7		movl	8(%ebp), %edx	33	movl	-4(%ebp), %ebx
8		movl	12(%ebp), %eax	34	leave	
9		addl	%edx, %eax	35	ret	
10		subl	16(%ebp), %eax			
11		movl	%eax, -4(%ebp)			
12		movl	-4(%ebp), %eax			
13		addl	\$1, %eax			
14		leave				
15		ret				
16	foo:					
17		pushl	%ebp			
18		movl	%esp, %ebp			
19		pushl	%ebx			
20		subl	\$4, %esp			
21		movl	8(%ebp), %edx			
22		movl	12(%ebp), %eax			
23		leal	(%edx,%eax), %ebx			
24		subl	\$4, %esp			
25		pushl	\$5			
26		pushl	\$4			

3. Why is EDX not saved by foo before calling bar despite the register being overwritten in bar?

Stack Frame

1	p:		call.s	27	pushl	\$3
2		.zero	4	28	call	bar
3	bar:			29	addl	\$16, %esp
4		pushl	%ebp	30	addl	%ebx, %eax
5		movl	%esp, %ebp	31	movl	%eax, p
6		subl	\$16, %esp	32	nop	
7		movl	8(%ebp), %edx	33	movl	-4(%ebp), %ebx
8		movl	12(%ebp), %eax	34	leave	
9		addl	%edx, %eax	35	ret	
10		subl	16(%ebp), %eax			
11		movl	%eax, -4(%ebp)			
12		movl	-4(%ebp), %eax			
13		addl	\$1, %eax			
14		leave				
15		ret				
16	foo:					
17		pushl	%ebp			
18		movl	%esp, %ebp			
19		pushl	%ebx			
20		subl	\$4, %esp			
21		movl	8(%ebp), %edx			
22		movl	12(%ebp), %eax			
23		leal	(%edx,%eax), %ebx			
24		subl	\$4, %esp			
25		pushl	\$5			
26		pushl	\$4			

3. Why is EDX not saved by foo before calling bar despite the register being overwritten in bar?

EDX does not need to persist after the function call.

Stack Frame

```

1  p:                                     call.s
2                                     .zero 4
3  bar:
4      pushl    %ebp
5      movl     %esp, %ebp
6      subl     $16, %esp
7      movl     8(%ebp), %edx
8      movl     12(%ebp), %eax
9      addl     %edx, %eax
10     subl     16(%ebp), %eax
11     movl     %eax, -4(%ebp)
12     movl     -4(%ebp), %eax
13     addl     $1, %eax
14     leave
15     ret
16  foo:
17     pushl    %ebp
18     movl     %esp, %ebp
19     pushl    %ebx
20     subl     $4, %esp
21     movl     8(%ebp), %edx
22     movl     12(%ebp), %eax
23     leal     (%edx,%eax), %ebx
24     subl     $4, %esp
25     pushl    $5
26     pushl    $4

```

```

27         pushl    $3
28         call     bar
29         addl     $16, %esp
30         addl     %ebx, %eax
31         movl     %eax, p
32         nop
33         movl     -4(%ebp), %ebx
34         leave
35         ret

```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	
EBX	
EDX	

EBP

ESP

Stack Frame of foo's Caller

Stack Frame

```

1  p:                                     call.s
2                                     .zero 4
3  bar:
4      pushl    %ebp
5      movl     %esp, %ebp
6      subl     $16, %esp
7      movl     8(%ebp), %edx
8      movl     12(%ebp), %eax
9      addl     %edx, %eax
10     subl     16(%ebp), %eax
11     movl     %eax, -4(%ebp)
12     movl     -4(%ebp), %eax
13     addl     $1, %eax
14     leave
15     ret
16  foo:
17     pushl    %ebp
18     movl     %esp, %ebp
19     pushl    %ebx
20     subl     $4, %esp
21     movl     8(%ebp), %edx
22     movl     12(%ebp), %eax
23     leal     (%edx,%eax), %ebx
24     subl     $4, %esp
25     pushl    $5
26     pushl    $4

```

```

27         pushl    $3
28         call     bar
29         addl     $16, %esp
30         addl     %ebx, %eax
31         movl     %eax, p
32         nop
33         movl     -4(%ebp), %ebx
34         leave
35         ret

```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	
EBX	
EDX	

EBP

Stack Frame of foo's Caller

Padding

b

a

Return Addr of foo's Caller

ESP

Stack Frame

```

call.s
1  p:
2      .zero    4
3  bar:
4      pushl    %ebp
5      movl     %esp, %ebp
6      subl     $16, %esp
7      movl     8(%ebp), %edx
8      movl     12(%ebp), %eax
9      addl     %edx, %eax
10     subl     16(%ebp), %eax
11     movl     %eax, -4(%ebp)
12     movl     -4(%ebp), %eax
13     addl     $1, %eax
14     leave
15     ret
16 foo:
17     pushl    %ebp
18     movl     %esp, %ebp
19     pushl    %ebx
20     subl     $4, %esp
21     movl     8(%ebp), %edx
22     movl     12(%ebp), %eax
23     leal     (%edx,%eax), %ebx
24     subl     $4, %esp
25     pushl    $5
26     pushl    $4

```

```

27         pushl    $3
28         call     bar
29         addl     $16, %esp
30         addl     %ebx, %eax
31         movl     %eax, p
32         nop
33         movl     -4(%ebp), %ebx
34         leave
35         ret

```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	
EBX	
EDX	

EBP

[illegible]

ESP

Stack Frame

```

call.s
1  p:
2      .zero    4
3  bar:
4      pushl    %ebp
5      movl     %esp, %ebp
6      subl     $16, %esp
7      movl     8(%ebp), %edx
8      movl     12(%ebp), %eax
9      addl     %edx, %eax
10     subl     16(%ebp), %eax
11     movl     %eax, -4(%ebp)
12     movl     -4(%ebp), %eax
13     addl     $1, %eax
14     leave
15     ret
16 foo:
17     pushl    %ebp
18     movl     %esp, %ebp
19     pushl    %ebx
20     subl     $4, %esp
21     movl     8(%ebp), %edx
22     movl     12(%ebp), %eax
23     leal     (%edx,%eax), %ebx
24     subl     $4, %esp
25     pushl    $5
26     pushl    $4

```

```

27         pushl    $3
28         call     bar
29         addl     $16, %esp
30         addl     %ebx, %eax
31         movl     %eax, p
32         nop
33         movl     -4(%ebp), %ebx
34         leave
35         ret

```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	
EBX	
EDX	

EBP, ESP

[illegible]

Stack Frame

```

1  p:                                     call.s
2                                     .zero 4
3  bar:
4      pushl    %ebp
5      movl     %esp, %ebp
6      subl     $16, %esp
7      movl     8(%ebp), %edx
8      movl     12(%ebp), %eax
9      addl     %edx, %eax
10     subl     16(%ebp), %eax
11     movl     %eax, -4(%ebp)
12     movl     -4(%ebp), %eax
13     addl     $1, %eax
14     leave
15     ret
16  foo:
17     pushl     %ebp
18     movl     %esp, %ebp
19     pushl    %ebx
20     subl     $4, %esp
21     movl     8(%ebp), %edx
22     movl     12(%ebp), %eax
23     leal     (%edx,%eax), %ebx
24     subl     $4, %esp
25     pushl    $5
26     pushl    $4

```

```

27         pushl    $3
28         call     bar
29         addl     $16, %esp
30         addl     %ebx, %eax
31         movl     %eax, p
32         nop
33         movl     -4(%ebp), %ebx
34         leave
35         ret

```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	
EBX	
EDX	

EBP
ESP

[illegible]

Stack Frame

```

1  p:                                     call.s
2                                     .zero 4
3  bar:
4      pushl    %ebp
5      movl     %esp, %ebp
6      subl     $16, %esp
7      movl     8(%ebp), %edx
8      movl     12(%ebp), %eax
9      addl     %edx, %eax
10     subl     16(%ebp), %eax
11     movl     %eax, -4(%ebp)
12     movl     -4(%ebp), %eax
13     addl     $1, %eax
14     leave
15     ret
16  foo:
17     pushl    %ebp
18     movl     %esp, %ebp
19     pushl    %ebx
20     subl     $4, %esp
21     movl     8(%ebp), %edx
22     movl     12(%ebp), %eax
23     leal     (%edx,%eax), %ebx
24     subl     $4, %esp
25     pushl    $5
26     pushl    $4

```

```

27         pushl    $3
28         call     bar
29         addl     $16, %esp
30         addl     %ebx, %eax
31         movl     %eax, p
32         nop
33         movl     -4(%ebp), %ebx
34         leave
35         ret

```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	
EBX	
EDX	

[illegible]

Stack Frame

```

1  p:                                     call.s
2                                     .zero 4
3  bar:
4      pushl    %ebp
5      movl     %esp, %ebp
6      subl     $16, %esp
7      movl     8(%ebp), %edx
8      movl     12(%ebp), %eax
9      addl     %edx, %eax
10     subl     16(%ebp), %eax
11     movl     %eax, -4(%ebp)
12     movl     -4(%ebp), %eax
13     addl     $1, %eax
14     leave
15     ret
16  foo:
17     pushl    %ebp
18     movl     %esp, %ebp
19     pushl    %ebx
20     subl     $4, %esp
21     movl     8(%ebp), %edx
22     movl     12(%ebp), %eax
23     leal     (%edx,%eax), %ebx
24     subl     $4, %esp
25     pushl    $5
26     pushl    $4

```

```

27         pushl    $3
28         call     bar
29         addl     $16, %esp
30         addl     %ebx, %eax
31         movl     %eax, p
32         nop
33         movl     -4(%ebp), %ebx
34         leave
35         ret

```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	
EBX	
EDX	a

[illegible]

Stack Frame

```

1  p:
2      .zero    4
3  bar:
4      pushl    %ebp
5      movl     %esp, %ebp
6      subl     $16, %esp
7      movl     8(%ebp), %edx
8      movl     12(%ebp), %eax
9      addl     %edx, %eax
10     subl     16(%ebp), %eax
11     movl     %eax, -4(%ebp)
12     movl     -4(%ebp), %eax
13     addl     $1, %eax
14     leave
15     ret
16 foo:
17     pushl    %ebp
18     movl     %esp, %ebp
19     pushl    %ebx
20     subl     $4, %esp
21     movl     8(%ebp), %edx
22     movl     12(%ebp), %eax
23     leal     (%edx,%eax), %ebx
24     subl     $4, %esp
25     pushl    $5
26     pushl    $4

```

```

27         pushl    $3
28         call     bar
29         addl     $16, %esp
30         addl     %ebx, %eax
31         movl     %eax, p
32         nop
33         movl     -4(%ebp), %ebx
34         leave
35         ret

```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	b
EBX	
EDX	a

[illegible]

Stack Frame

```

1  p:
2      .zero    4
3  bar:
4      pushl    %ebp
5      movl     %esp, %ebp
6      subl     $16, %esp
7      movl     8(%ebp), %edx
8      movl     12(%ebp), %eax
9      addl     %edx, %eax
10     subl     16(%ebp), %eax
11     movl     %eax, -4(%ebp)
12     movl     -4(%ebp), %eax
13     addl     $1, %eax
14     leave
15     ret
16 foo:
17     pushl     %ebp
18     movl     %esp, %ebp
19     pushl     %ebx
20     subl     $4, %esp
21     movl     8(%ebp), %edx
22     movl     12(%ebp), %eax
23     leal     (%edx,%eax), %ebx
24     subl     $4, %esp
25     pushl     $5
26     pushl     $4

```

```

27         pushl    $3
28         call     bar
29         addl     $16, %esp
30         addl     %ebx, %eax
31         movl     %eax, p
32         nop
33         movl     -4(%ebp), %ebx
34         leave
35         ret

```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	b
EBX	a + b
EDX	a

[illegible]

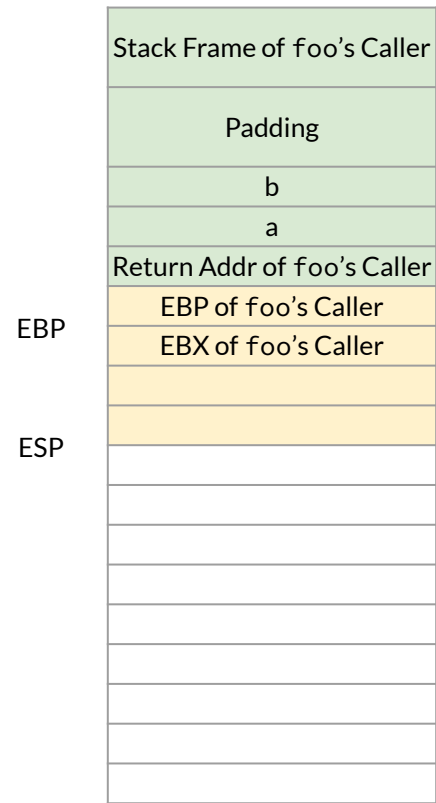
Stack Frame

```
1 p:                                call.s
2 .zero 4
3 bar:
4     pushl %ebp
5     movl  %esp, %ebp
6     subl  $16, %esp
7     movl  8(%ebp), %edx
8     movl  12(%ebp), %eax
9     addl  %edx, %eax
10    subl  16(%ebp), %eax
11    movl  %eax, -4(%ebp)
12    movl  -4(%ebp), %eax
13    addl  $1, %eax
14    leave
15    ret
16 foo:
17     pushl %ebp
18     movl  %esp, %ebp
19     pushl %ebx
20     subl  $4, %esp
21     movl  8(%ebp), %edx
22     movl  12(%ebp), %eax
23     leal  (%edx,%eax), %ebx
24     subl  $4, %esp
25     pushl $5
26     pushl $4
```

```
27     pushl $3
28     call  bar
29     addl  $16, %esp
30     addl  %ebx, %eax
31     movl  %eax, p
32     nop
33     movl  -4(%ebp), %ebx
34     leave
35     ret
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	b
EBX	a + b
EDX	a



Stack Frame

```
1 p:                                call.s      27
2                                     .zero      4
3 bar:                               28
4     pushl   %ebp                    29
5     movl    %esp, %ebp              30
6     subl    $16, %esp               31
7     movl    8(%ebp), %edx            32
8     movl    12(%ebp), %eax           33
9     addl    %edx, %eax               34
10    subl    16(%ebp), %eax           35
11    movl    %eax, -4(%ebp)
12    movl    -4(%ebp), %eax
13    addl    $1, %eax
14    leave
15    ret
16 foo:
17     pushl   %ebp
18     movl    %esp, %ebp
19     pushl   %ebx
20     subl    $4, %esp
21     movl    8(%ebp), %edx
22     movl    12(%ebp), %eax
23     leal    (%edx,%eax), %ebx
24     subl    $4, %esp
25     pushl   $5
26     pushl   $4
```

```
27     pushl   $3
28     call    bar
29     addl    $16, %esp
30     addl    %ebx, %eax
31     movl    %eax, p
32     nop
33     movl    -4(%ebp), %ebx
34     leave
35     ret
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	b
EBX	a + b
EDX	a

EBP

ESP

Stack Frame of foo's Caller
Padding
b
a
Return Addr of foo's Caller
EBP of foo's Caller
EBX of foo's Caller
5

Stack Frame

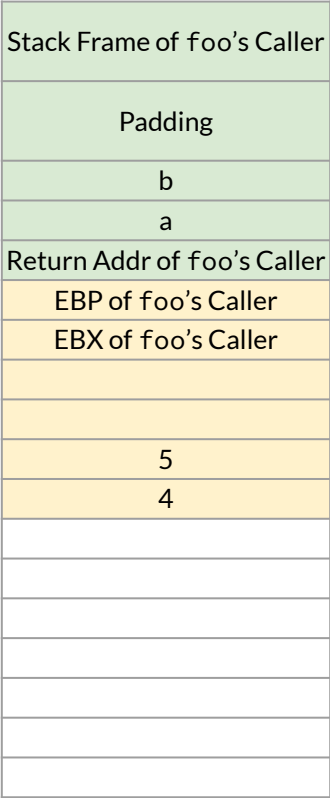
```
1  p:                                call.s      27
2                                     .zero      4
3  bar:                               28
4      pushl   %ebp                    29
5      movl    %esp, %ebp              30
6      subl    $16, %esp               31
7      movl    8(%ebp), %edx           32
8      movl    12(%ebp), %eax          33
9      addl    %edx, %eax              34
10     subl    16(%ebp), %eax           35
11     movl    %eax, -4(%ebp)
12     movl    -4(%ebp), %eax
13     addl    $1, %eax
14     leave
15     ret
16 foo:                               27
17     pushl    %ebp                    28
18     movl     %esp, %ebp              29
19     pushl    %ebx                    30
20     subl     $4, %esp                31
21     movl     8(%ebp), %edx           32
22     movl     12(%ebp), %eax          33
23     leal     (%edx,%eax), %ebx        34
24     subl     $4, %esp                35
25     pushl    $5
26     pushl    $4
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	b
EBX	a + b
EDX	a

EBP

ESP



Stack Frame

```
1 p:
2   .zero 4
3 bar:
4   pushl %ebp
5   movl  %esp, %ebp
6   subl  $16, %esp
7   movl  8(%ebp), %edx
8   movl  12(%ebp), %eax
9   addl  %edx, %eax
10  subl  16(%ebp), %eax
11  movl  %eax, -4(%ebp)
12  movl  -4(%ebp), %eax
13  addl  $1, %eax
14  leave
15  ret
16 foo:
17  pushl %ebp
18  movl  %esp, %ebp
19  pushl %ebx
20  subl  $4, %esp
21  movl  8(%ebp), %edx
22  movl  12(%ebp), %eax
23  leal  (%edx,%eax), %ebx
24  subl  $4, %esp
25  pushl $5
26  pushl $4
```

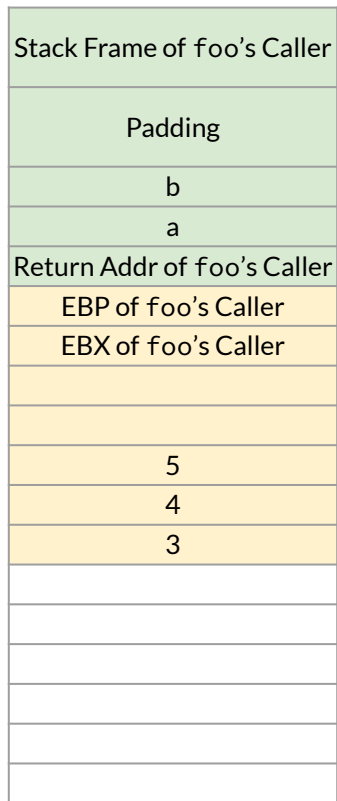
```
27  pushl $3
28  call  bar
29  addl  $16, %esp
30  addl  %ebx, %eax
31  movl  %eax, p
32  nop
33  movl  -4(%ebp), %ebx
34  leave
35  ret
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	b
EBX	a + b
EDX	a

EBP

ESP



Stack Frame

```
1  p:                                call.s
2                                     .zero 4
3  bar:                               pushl %ebp
4                                     movl %esp, %ebp
5                                     subl $16, %esp
6                                     movl 8(%ebp), %edx
7                                     movl 12(%ebp), %eax
8                                     addl %edx, %eax
9                                     subl 16(%ebp), %eax
10                                    movl %eax, -4(%ebp)
11                                    movl -4(%ebp), %eax
12                                    addl $1, %eax
13                                    leave
14                                    ret
15
16 foo:                               pushl %ebp
17                                    movl %esp, %ebp
18                                    pushl %ebx
19                                    subl $4, %esp
20                                    movl 8(%ebp), %edx
21                                    movl 12(%ebp), %eax
22                                    leal (%edx,%eax), %ebx
23                                    subl $4, %esp
24                                    pushl $5
25                                    pushl $4
```

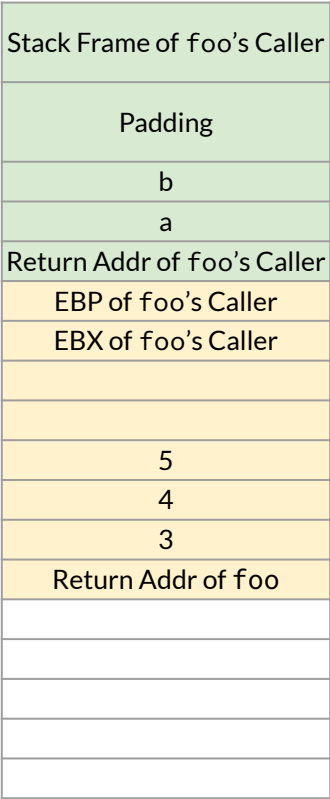
```
27                                pushl $3
28                                call bar
29                                addl $16, %esp
30                                addl %ebx, %eax
31                                movl %eax, p
32                                nop
33                                movl -4(%ebp), %ebx
34                                leave
35                                ret
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	b
EBX	a + b
EDX	a

EBP

ESP



Stack Frame

```
1  p:                                call.s
2                                     .zero 4
3  bar:
4      pushl %ebp
5      movl %esp, %ebp
6      subl $16, %esp
7      movl 8(%ebp), %edx
8      movl 12(%ebp), %eax
9      addl %edx, %eax
10     subl 16(%ebp), %eax
11     movl %eax, -4(%ebp)
12     movl -4(%ebp), %eax
13     addl $1, %eax
14     leave
15     ret
16 foo:
17     pushl %ebp
18     movl %esp, %ebp
19     pushl %ebx
20     subl $4, %esp
21     movl 8(%ebp), %edx
22     movl 12(%ebp), %eax
23     leal (%edx,%eax), %ebx
24     subl $4, %esp
25     pushl $5
26     pushl $4
```

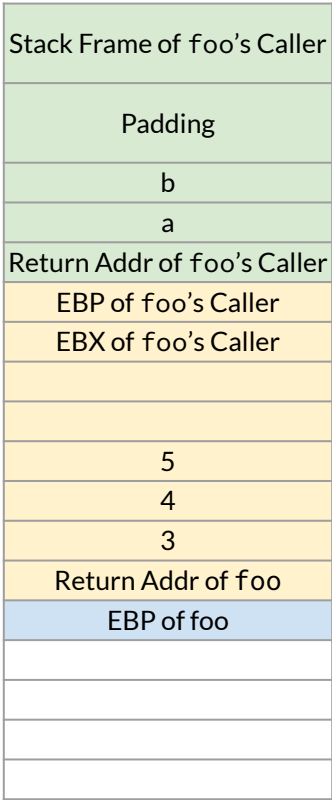
```
27     pushl $3
28     call bar
29     addl $16, %esp
30     addl %ebx, %eax
31     movl %eax, p
32     nop
33     movl -4(%ebp), %ebx
34     leave
35     ret
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	b
EBX	a + b
EDX	a

EBP

ESP



Stack Frame

```
1 p:                                call.s
2                                .zero 4
3 bar:
4                                pushl %ebp
5                                movl %esp, %ebp
6                                subl $16, %esp
7                                movl 8(%ebp), %edx
8                                movl 12(%ebp), %eax
9                                addl %edx, %eax
10                               subl 16(%ebp), %eax
11                               movl %eax, -4(%ebp)
12                               movl -4(%ebp), %eax
13                               addl $1, %eax
14                               leave
15                               ret
16 foo:
17                               pushl %ebp
18                               movl %esp, %ebp
19                               pushl %ebx
20                               subl $4, %esp
21                               movl 8(%ebp), %edx
22                               movl 12(%ebp), %eax
23                               leal (%edx,%eax), %ebx
24                               subl $4, %esp
25                               pushl $5
26                               pushl $4
```

```
27                               pushl $3
28                               call bar
29                               addl $16, %esp
30                               addl %ebx, %eax
31                               movl %eax, p
32                               nop
33                               movl -4(%ebp), %ebx
34                               leave
35                               ret
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	b
EBX	a + b
EDX	a

EBP, ESP

Stack Frame of foo's Caller
Padding
b
a
Return Addr of foo's Caller
EBP of foo's Caller
EBX of foo's Caller
5
4
3
Return Addr of foo
EBP of foo

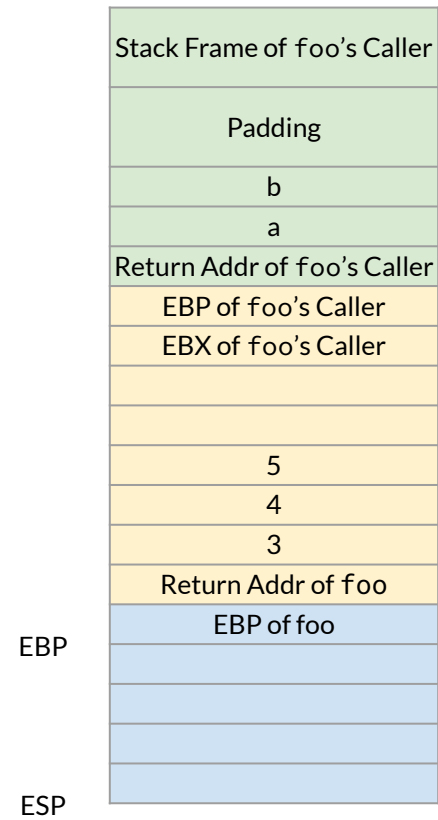
Stack Frame

```
1  p:                                call.s      27
2                                     .zero      4
3  bar:                               pushl      %ebp
4                                     movl      %esp, %ebp
5                                     subl      $16, %esp
6                                     movl      8(%ebp), %edx
7                                     movl      12(%ebp), %eax
8                                     addl      %edx, %eax
9                                     subl      16(%ebp), %eax
10                                    movl      %eax, -4(%ebp)
11                                    movl      -4(%ebp), %eax
12                                    addl      $1, %eax
13                                    leave
14                                    ret
15
16 foo:                               pushl      %ebp
17                                    movl      %esp, %ebp
18                                    pushl      %ebx
19                                    subl      $4, %esp
20                                    movl      8(%ebp), %edx
21                                    movl      12(%ebp), %eax
22                                    leal      (%edx,%eax), %ebx
23                                    subl      $4, %esp
24                                    pushl      $5
25                                    pushl      $4
```

```
27  pushl      $3
28  call      bar
29  addl      $16, %esp
30  addl      %ebx, %eax
31  movl      %eax, p
32  nop
33  movl      -4(%ebp), %ebx
34  leave
35  ret
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	b
EBX	a + b
EDX	a



Stack Frame

```
1 p:                                call.s
2                                     .zero 4
3 bar:
4     pushl %ebp
5     movl  %esp, %ebp
6     subl  $16, %esp
7     movl  8(%ebp), %edx
8     movl  12(%ebp), %eax
9     addl  %edx, %eax
10    subl  16(%ebp), %eax
11    movl  %eax, -4(%ebp)
12    movl  -4(%ebp), %eax
13    addl  $1, %eax
14    leave
15    ret
16 foo:
17    pushl %ebp
18    movl  %esp, %ebp
19    pushl %ebx
20    subl  $4, %esp
21    movl  8(%ebp), %edx
22    movl  12(%ebp), %eax
23    leal  (%edx,%eax), %ebx
24    subl  $4, %esp
25    pushl $5
26    pushl $4
```

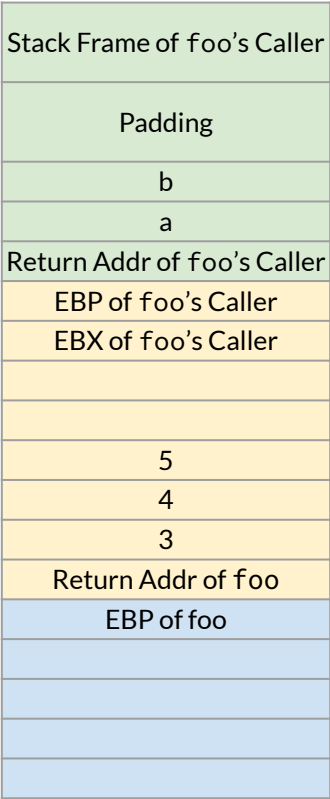
```
27    pushl $3
28    call  bar
29    addl  $16, %esp
30    addl  %ebx, %eax
31    movl  %eax, p
32    nop
33    movl  -4(%ebp), %ebx
34    leave
35    ret
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	b
EBX	a + b
EDX	3

EBP

ESP



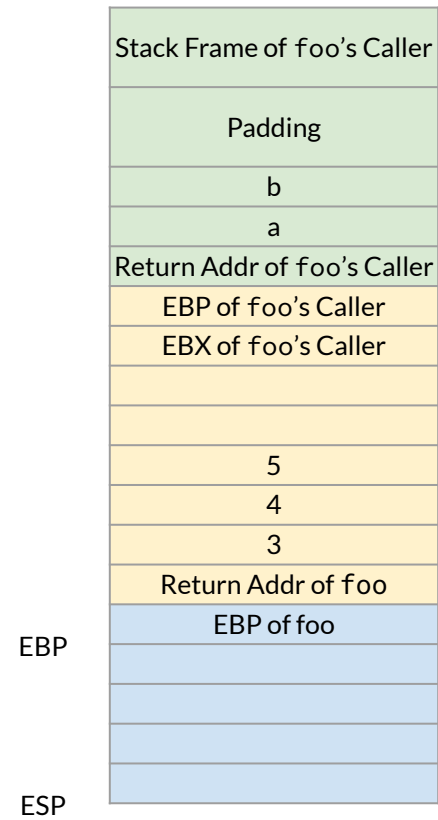
Stack Frame

```
1 p:
2   .zero 4
3 bar:
4   pushl %ebp
5   movl  %esp, %ebp
6   subl  $16, %esp
7   movl  8(%ebp), %edx
8   movl  12(%ebp), %eax
9   addl  %edx, %eax
10  subl  16(%ebp), %eax
11  movl  %eax, -4(%ebp)
12  movl  -4(%ebp), %eax
13  addl  $1, %eax
14  leave
15  ret
16 foo:
17  pushl %ebp
18  movl  %esp, %ebp
19  pushl %ebx
20  subl  $4, %esp
21  movl  8(%ebp), %edx
22  movl  12(%ebp), %eax
23  leal  (%edx,%eax), %ebx
24  subl  $4, %esp
25  pushl $5
26  pushl $4
```

```
27  pushl $3
28  call  bar
29  addl  $16, %esp
30  addl  %ebx, %eax
31  movl  %eax, p
32  nop
33  movl  -4(%ebp), %ebx
34  leave
35  ret
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	4
EBX	a + b
EDX	3



Stack Frame

```
1 p:                                call.s
2                                .zero 4
3 bar:
4    pushl %ebp
5    movl  %esp, %ebp
6    subl  $16, %esp
7    movl  8(%ebp), %edx
8    movl  12(%ebp), %eax
9    addl  %edx, %eax
10   subl  16(%ebp), %eax
11   movl  %eax, -4(%ebp)
12   movl  -4(%ebp), %eax
13   addl  $1, %eax
14   leave
15   ret
16 foo:
17   pushl %ebp
18   movl  %esp, %ebp
19   pushl %ebx
20   subl  $4, %esp
21   movl  8(%ebp), %edx
22   movl  12(%ebp), %eax
23   leal  (%edx,%eax), %ebx
24   subl  $4, %esp
25   pushl $5
26   pushl $4
```

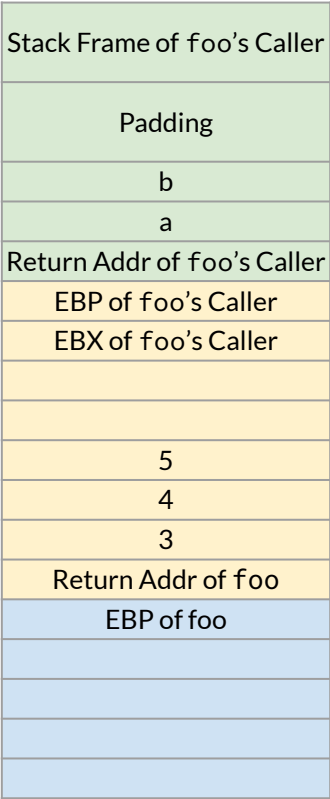
```
27   pushl $3
28   call  bar
29   addl  $16, %esp
30   addl  %ebx, %eax
31   movl  %eax, p
32   nop
33   movl  -4(%ebp), %ebx
34   leave
35   ret
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	7
EBX	a + b
EDX	3

EBP

ESP



Stack Frame

```
1 p:                                call.s
2                                .zero 4
3 bar:
4    pushl %ebp
5    movl  %esp, %ebp
6    subl  $16, %esp
7    movl  8(%ebp), %edx
8    movl  12(%ebp), %eax
9    addl  %edx, %eax
10   subl  16(%ebp), %eax
11    movl  %eax, -4(%ebp)
12    movl  -4(%ebp), %eax
13    addl  $1, %eax
14    leave
15    ret
16 foo:
17    pushl %ebp
18    movl  %esp, %ebp
19    pushl %ebx
20    subl  $4, %esp
21    movl  8(%ebp), %edx
22    movl  12(%ebp), %eax
23    leal  (%edx,%eax), %ebx
24    subl  $4, %esp
25    pushl $5
26    pushl $4
```

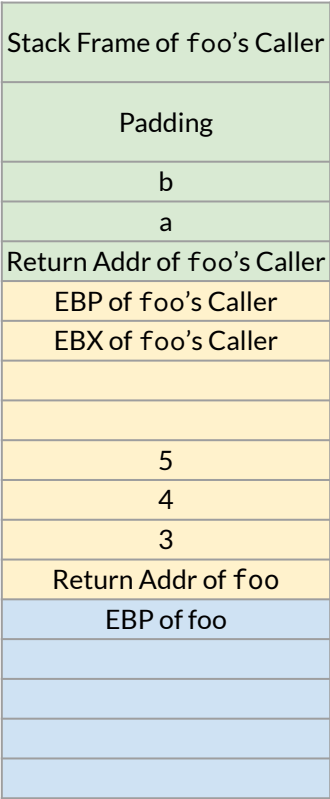
```
27    pushl $3
28    call  bar
29    addl  $16, %esp
30    addl  %ebx, %eax
31    movl  %eax, p
32    nop
33    movl  -4(%ebp), %ebx
34    leave
35    ret
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	2
EBX	a + b
EDX	3

EBP

ESP



Stack Frame

```
1 p:
2   .zero 4
3 bar:
4   pushl %ebp
5   movl  %esp, %ebp
6   subl  $16, %esp
7   movl  8(%ebp), %edx
8   movl  12(%ebp), %eax
9   addl  %edx, %eax
10  subl  16(%ebp), %eax
11  movl  %eax, -4(%ebp)
12  movl  -4(%ebp), %eax
13  addl  $1, %eax
14  leave
15  ret
16 foo:
17  pushl %ebp
18  movl  %esp, %ebp
19  pushl %ebx
20  subl  $4, %esp
21  movl  8(%ebp), %edx
22  movl  12(%ebp), %eax
23  leal  (%edx,%eax), %ebx
24  subl  $4, %esp
25  pushl $5
26  pushl $4
```

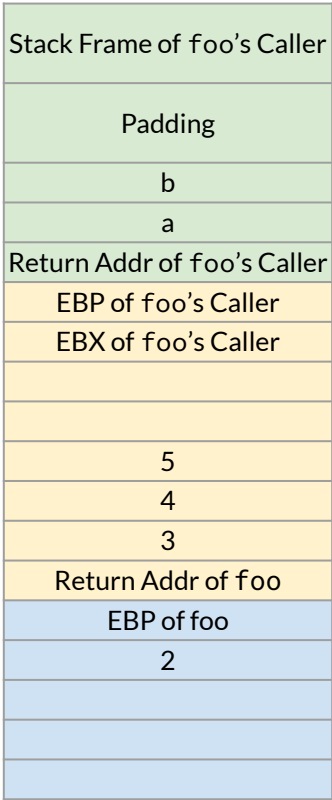
```
27  pushl $3
28  call  bar
29  addl  $16, %esp
30  addl  %ebx, %eax
31  movl  %eax, p
32  nop
33  movl  -4(%ebp), %ebx
34  leave
35  ret
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	2
EBX	a + b
EDX	3

EBP

ESP



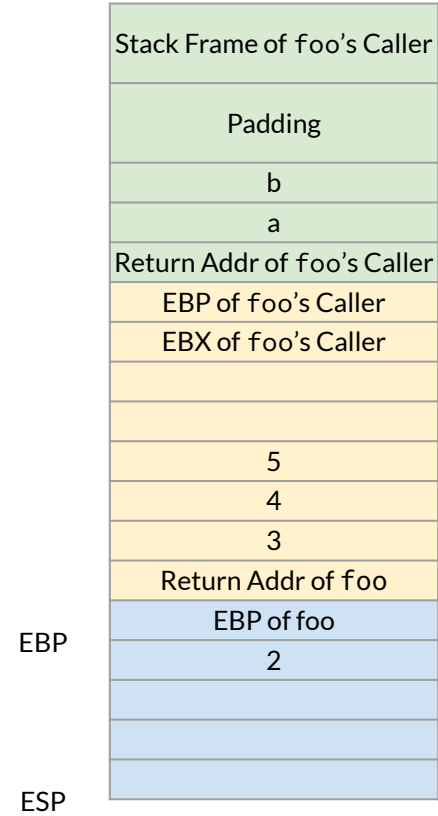
Stack Frame

```
1 p: call.s
2 .zero 4
3 bar:
4 pushl %ebp
5 movl %esp, %ebp
6 subl $16, %esp
7 movl 8(%ebp), %edx
8 movl 12(%ebp), %eax
9 addl %edx, %eax
10 subl 16(%ebp), %eax
11 movl %eax, -4(%ebp)
12 movl -4(%ebp), %eax
13 addl $1, %eax
14 leave
15 ret
16 foo:
17 pushl %ebp
18 movl %esp, %ebp
19 pushl %ebx
20 subl $4, %esp
21 movl 8(%ebp), %edx
22 movl 12(%ebp), %eax
23 leal (%edx,%eax), %ebx
24 subl $4, %esp
25 pushl $5
26 pushl $4
```

```
27 pushl $3
28 call bar
29 addl $16, %esp
30 addl %ebx, %eax
31 movl %eax, p
32 nop
33 movl -4(%ebp), %ebx
34 leave
35 ret
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	2
EBX	a + b
EDX	3



Stack Frame

```
1  p:                                call.s      27
2                                     .zero      4
3  bar:                               pushl      %ebp
4                                     movl      %esp, %ebp
5                                     subl      $16, %esp
6                                     movl      8(%ebp), %edx
7                                     movl      12(%ebp), %eax
8                                     addl      %edx, %eax
9                                     subl      16(%ebp), %eax
10                                    movl      %eax, -4(%ebp)
11                                    movl      -4(%ebp), %eax
12                                    addl      $1, %eax
13                                    leave
14                                    ret
15
16 foo:                               pushl      %ebp
17                                    movl      %esp, %ebp
18                                    pushl      %ebx
19                                    subl      $4, %esp
20                                    movl      8(%ebp), %edx
21                                    movl      12(%ebp), %eax
22                                    leal      (%edx,%eax), %ebx
23                                    subl      $4, %esp
24                                    pushl      $5
25                                    pushl      $4
```

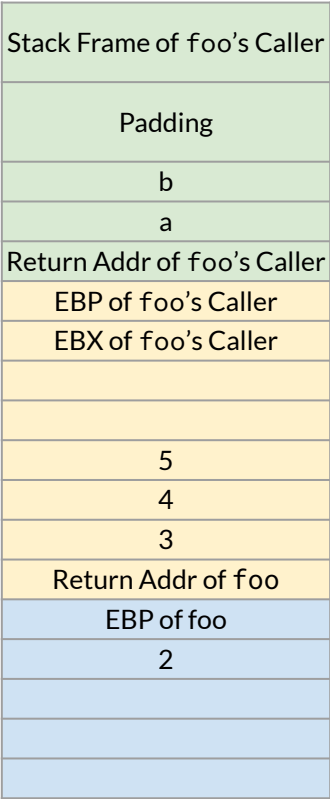
```
27  pushl      $3
28  call      bar
29  addl      $16, %esp
30  addl      %ebx, %eax
31  movl      %eax, p
32  nop
33  movl      -4(%ebp), %ebx
34  leave
35  ret
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	3
EBX	a + b
EDX	3

EBP

ESP



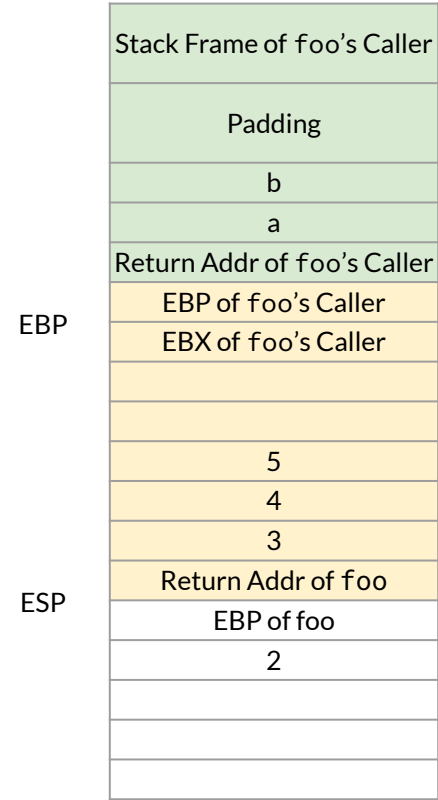
Stack Frame

```
1 p:      call.s      27
2          .zero      4      28
3 bar:    pushl      %ebp      29
4          movl      %esp, %ebp      30
5          subl      $16, %esp      31
6          movl      8(%ebp), %edx      32
7          movl      12(%ebp), %eax      33
8          addl      %edx, %eax      34
9          subl      16(%ebp), %eax      35
10         movl      %eax, -4(%ebp)
11         movl      -4(%ebp), %eax
12         addl      $1, %eax
13         leave
14         ret
15
16 foo:    pushl      %ebp
17         movl      %esp, %ebp
18         pushl      %ebx
19         subl      $4, %esp
20         movl      8(%ebp), %edx
21         movl      12(%ebp), %eax
22         leal      (%edx,%eax), %ebx
23         subl      $4, %esp
24         pushl      $5
25         pushl      $4
```

```
27      pushl      $3
28      call      bar
29      addl      $16, %esp
30      addl      %ebx, %eax
31      movl      %eax, p
32      nop
33      movl      -4(%ebp), %ebx
34      leave
35      ret
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	3
EBX	a + b
EDX	3



Stack Frame

```
1 p:
2     .zero 4
3 bar:
4     pushl %ebp
5     movl  %esp, %ebp
6     subl  $16, %esp
7     movl  8(%ebp), %edx
8     movl  12(%ebp), %eax
9     addl  %edx, %eax
10    subl  16(%ebp), %eax
11    movl  %eax, -4(%ebp)
12    movl  -4(%ebp), %eax
13    addl  $1, %eax
14    leave
15    ret
16 foo:
17    pushl %ebp
18    movl  %esp, %ebp
19    pushl %ebx
20    subl  $4, %esp
21    movl  8(%ebp), %edx
22    movl  12(%ebp), %eax
23    leal  (%edx,%eax), %ebx
24    subl  $4, %esp
25    pushl $5
26    pushl $4
```

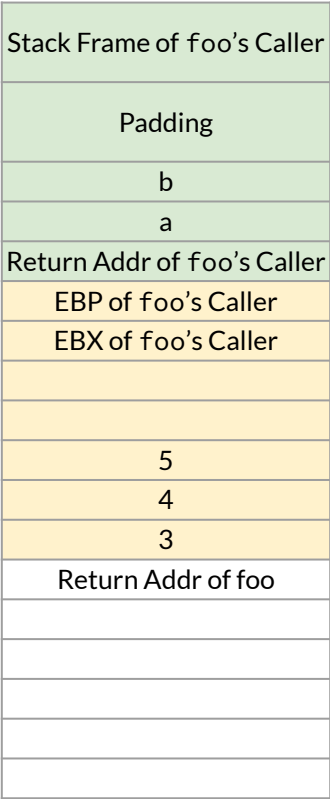
```
27    pushl $3
28    call  bar
29    addl  $16, %esp
30    addl  %ebx, %eax
31    movl  %eax, p
32    nop
33    movl  -4(%ebp), %ebx
34    leave
35    ret
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	3
EBX	a + b
EDX	3

EBP

ESP



Stack Frame

```
1 p:
2   .zero 4
3 bar:
4   pushl %ebp
5   movl  %esp, %ebp
6   subl  $16, %esp
7   movl  8(%ebp), %edx
8   movl  12(%ebp), %eax
9   addl  %edx, %eax
10  subl  16(%ebp), %eax
11  movl  %eax, -4(%ebp)
12  movl  -4(%ebp), %eax
13  addl  $1, %eax
14  leave
15  ret
16 foo:
17  pushl %ebp
18  movl  %esp, %ebp
19  pushl %ebx
20  subl  $4, %esp
21  movl  8(%ebp), %edx
22  movl  12(%ebp), %eax
23  leal  (%edx,%eax), %ebx
24  subl  $4, %esp
25  pushl $5
26  pushl $4
```

```
27   pushl $3
28   call  bar
29   addl  $16, %esp
30   addl  %ebx, %eax
31   movl  %eax, p
32   nop
33   movl  -4(%ebp), %ebx
34   leave
35   ret
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	3
EBX	a + b
EDX	3

EBP

ESP

Stack Frame of foo's Caller
Padding
b
a
Return Addr of foo's Caller
EBP of foo's Caller
EBX of foo's Caller
5
4
3

Stack Frame

```

1  p:                                     call.s
2                                     .zero 4
3  bar:
4      pushl    %ebp
5      movl     %esp, %ebp
6      subl     $16, %esp
7      movl     8(%ebp), %edx
8      movl     12(%ebp), %eax
9      addl     %edx, %eax
10     subl     16(%ebp), %eax
11     movl     %eax, -4(%ebp)
12     movl     -4(%ebp), %eax
13     addl     $1, %eax
14     leave
15     ret
16  foo:
17     pushl    %ebp
18     movl     %esp, %ebp
19     pushl    %ebx
20     subl     $4, %esp
21     movl     8(%ebp), %edx
22     movl     12(%ebp), %eax
23     leal     (%edx,%eax), %ebx
24     subl     $4, %esp
25     pushl    $5
26     pushl    $4

```

```

27         pushl    $3
28         call     bar
29         addl     $16, %esp
30         addl     %ebx, %eax
31         movl     %eax, p
32         nop
33         movl     -4(%ebp), %ebx
34         leave
35         ret

```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	$3 + a + b$
EBX	$a + b$
EDX	3

[illegible]

Stack Frame

```

1  p:                                     call.s
2                                     .zero 4
3  bar:
4      pushl    %ebp
5      movl     %esp, %ebp
6      subl     $16, %esp
7      movl     8(%ebp), %edx
8      movl     12(%ebp), %eax
9      addl     %edx, %eax
10     subl     16(%ebp), %eax
11     movl     %eax, -4(%ebp)
12     movl     -4(%ebp), %eax
13     addl     $1, %eax
14     leave
15     ret
16  foo:
17     pushl    %ebp
18     movl     %esp, %ebp
19     pushl    %ebx
20     subl     $4, %esp
21     movl     8(%ebp), %edx
22     movl     12(%ebp), %eax
23     leal     (%edx,%eax), %ebx
24     subl     $4, %esp
25     pushl    $5
26     pushl    $4

```

```

27      pushl    $3
28      call    bar
29      addl    $16, %esp
30      addl    %ebx, %eax
31      movl    %eax, p
32      nop
33      movl    -4(%ebp), %ebx
34      leave
35      ret

```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	3 + a + b
EBX	EBX of foo's Caller
EDX	3

[illegible]

Stack Frame

```

1  p:                                     call.s
2                                     .zero 4
3  bar:
4      pushl    %ebp
5      movl     %esp, %ebp
6      subl     $16, %esp
7      movl     8(%ebp), %edx
8      movl     12(%ebp), %eax
9      addl     %edx, %eax
10     subl     16(%ebp), %eax
11     movl     %eax, -4(%ebp)
12     movl     -4(%ebp), %eax
13     addl     $1, %eax
14     leave
15     ret
16  foo:
17     pushl    %ebp
18     movl     %esp, %ebp
19     pushl    %ebx
20     subl     $4, %esp
21     movl     8(%ebp), %edx
22     movl     12(%ebp), %eax
23     leal     (%edx,%eax), %ebx
24     subl     $4, %esp
25     pushl    $5
26     pushl    $4

```

```

27      pushl    $3
28      call    bar
29      addl    $16, %esp
30      addl    %ebx, %eax
31      movl    %eax, p
32      nop
33      movl    -4(%ebp), %ebx
34      leave
35      ret

```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	$3 + a + b$
EBX	EBX of foo's Caller
EDX	3

[illegible]

Stack Frame

```
1 p:
2   .zero 4
3 bar:
4   pushl %ebp
5   movl  %esp, %ebp
6   subl  $16, %esp
7   movl  8(%ebp), %edx
8   movl  12(%ebp), %eax
9   addl  %edx, %eax
10  subl  16(%ebp), %eax
11  movl  %eax, -4(%ebp)
12  movl  -4(%ebp), %eax
13  addl  $1, %eax
14  leave
15  ret
16 foo:
17  pushl %ebp
18  movl  %esp, %ebp
19  pushl %ebx
20  subl  $4, %esp
21  movl  8(%ebp), %edx
22  movl  12(%ebp), %eax
23  leal  (%edx,%eax), %ebx
24  subl  $4, %esp
25  pushl $5
26  pushl $4
```

```
27  pushl $3
28  call  bar
29  addl  $16, %esp
30  addl  %ebx, %eax
31  movl  %eax, p
32  nop
33  movl  -4(%ebp), %ebx
34  leave
35  ret
```

4. Draw the stack frame with ESP and EBP and contents of EAX, EBX, EDX registers.

EAX	3 + a + b
EBX	EBX of foo's Caller
EDX	3

