Discussion 0: C, x86

January 21, 2022

Contents

1   C

C is often the programming language of choice in operating systems. C grants programmers low-level access to memory which is very useful. You'll be reading and writing a lot of C code throughout this class.

Types

C is **statically typed** where types are known at compile time. However, C is also **weakly typed** meaning you can cast between any types. This gives the necessary flexibility for working with low-level memory, but it also opens up many avenues for errors if you're not careful.

The primitive types include `char`, `short`, `int`, `long`, `float`, and `double`. The size of data types may vary depending on the operating system, so it's best to check using the operator `sizeof`.

Arrays, denoted with `[]` (e.g. `int[]`), are contiguous regions of memory of fixed size. Each element is the size of the data type corresponding to that array. A **string** in C is just an array of characters with a last element as `null` to indicate the end of the string.

Users can define compound data types using `struct`s which are contiguous pieces of memory comprised of multiple other data types.

Pointers are references that hold the address of an object in memory. Fundamentally, pointers are just unsigned integers of size equal to the number of bits supported by the operating system. Prefixing a pointer with `*` will return the value at the memory address that the pointer is holding. On the other hand, prefixing a variable with `&` will return the memory address of the variable.

Memory

A typical C program's memory is divided into five segments.

Segment	Purpose
Text	Machine code of the compiled program
Initialized Data	Initialized global and static memory
Uninitialized Data	Uninitialized global and static memory
Heap	Dynamically allocated memory
Stack	Local variables and argument passing

In general, memory can be thought of as a giant array with elements of one byte where a memory address indexes into this array.

Unlike stack memory, heap memory needs to be explicitly managed by the user. Memory can be allocated in the heap using `malloc`, `calloc`, or `realloc`. These all return a pointer to a chunk of memory in heap that is the size of the amount requested. When the memory is no longer being used, it needs to be explicitly released using `free`.

GNU Debugger (GDB)

GNU Debugger (**GDB**) is a powerful tool to debug your programs. While you may have skidded by CS 61C without using it, the complicated codebase for this class will require you to be proficient with GDB. Help will not be given to those who are not able to use GDB.

The general workflow of using GDB is as follows.

1. Compile the program using the `-g` flag.

2. Start GDB using `gdb <executable>`.

3. Set breakpoints using `break <linenum>`. You can also break at functions by using `break <func>`.

4. Run program with `run`. If the program takes in arguments, pass in those (i.e. `run arg1 arg2`).

5. Once you hit your breakpoint, examine using `print`. Other commands like `display`, `watch`, `set`, and many more will come in handy. You can also step line by line using `next`.

While you don't have to memorize all the GDB commands, you'll grow familiar with them the more you use them. When looking for certain functionality, check out the GDB User Manual[1].

1.1 Concept Check

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

> 4. `dbl_char` is a double pointer, so dereferencing it once will still give a pointer. A 32-bit system means memory addresses will be 32 bits or equivalently 4 bytes.

2. Consider strings initialized as

```
char* a = "162 is the best";
char b[] = "162 is the best";
```

Are `a` and `b` different?

> Yes. Since it's defined as a literal, `a` points to a string literal in a read only section of memory. On the other hand, `b` resides on the stack. It is equivalent to declaring
> `char b[] = {'1', '6', '2', ' ', 'i', 's', ' ', 'b', 'e', 's', 't', 0}`.

3. Suppose you have an integer array `int nums[3] = {152, 161, 162}`. What are the differences between `nums`, `&nums`, `&nums[0]`?

> The three will all return memory address that are equivalent to each other. However, there is a subtle difference in that `nums` and `&nums[0]` point to the first element (i.e. an integer), while `&nums` points to the entire array (i.e. an array of integers). As a result, if you increment each of them by one, `nums` and `&nums[0]` will be incremented by `sizeof(int)`, while `&nums` will be incremented by `3 * sizeof(int)`.

1.2 Headers

```c
1  #include <stdio.h>
2  #include "lib.h"
3
4  int main(int argc, char** argv) {
5    helper_args_t helper_args;
6    helper_args.string = argv[0];
7    helper_args.target = '/';
8    char* result = helper_func(&helper_args);
9    printf("%s\n", result);
10   return 0;
11 }
```
<center>app.c</center>

```c
1  typedef struct helper_args {
2  #ifdef ABC
3    char* aux;
4  #endif
5    char* string;
6    char target;
7  } helper_args_t;
8  char* helper_func(helper_args_t* args);
```
<center>lib.h</center>

```c
1  #include "lib.h"
2
3  char* helper_func(helper_args_t* args) {
4    int i;
5    for (i = 0; args->string[i] != '\0'; i++)
6      if (args->string[i] == args->target)
7        return &args->string[i + 1];
8    return args->string;
```

---

[1]https://sourceware.org/gdb/current/onlinedocs/gdb/

```
9 }
```
<div align="center">lib.c</div>

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?

> 16 bytes. Since `ABC` is not defined, there is only `char* string` and `char target`, which is a total of $8 + 1 = 9$ bytes. However, GCC pads the structure for alignment, so it fills in the remaining 7 bytes after `char target`.

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

> 24 bytes. Now that `ABC` is defined, there's an additional 8 bytes from `char* aux`.

3. Suppose we build the program in a different way with the original files (i.e. none of the changes from previous question 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. What is the issue?

   > `app.c` is compiled with `ABC` defined (as seen from the `-DABC` flag), but `lib.c` is compiled without it. As a result, they have differing definitions of `helper_args_t`. In `main`, `argv[0]` is put in to address of `helper_args` + 8 bytes since `app.c` was compiled with `char* aux` member, making `char* string` member be 8 bytes offset from `args`. However, in `helper_func`, the program will access the address of `args` when it accesses `args->string` since it was compiled without `char* aux` member, making `char* string` the first member. This leads to undefined behavior since `char* aux` was never initialized in `main`, so it contains garbage value.
   >
   > An important observation is that this won't always lead to a segfault. The value in `char* aux` member is garbage, meaning it coul d potentially be a non-illegal memory access by complete chance.

1.3 Debugging Segmentation Faults

Observe the following program from `singer.c` which aims to sort a string using quicksort. The program will use a string provided as the argument or defaults to `"IU is the best singer!"` if none is provided.

When the program is compiled and run, we get the output on the right. Use GDB to fix the issue.

```
4  void swap(char* a, int i, int j) {
5    char t = a[i];
6    a[i] = a[j];
7    a[j] = t;
8  }
9
10 int partition(char* a, int l, int r){
```

```
> ./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)
```

```
11    int pivot = a[l];
12    int i = l, j = r+1;
13
14    while (1) {
15      do
16        ++i;
17      while (a[i] <= pivot && i <= r);
18
19      do
20        --j;
21      while (a[j] > pivot);
22
23      if (i >= j)
24        break;
25
26      swap(a, i, j);
27    }
28
29    swap(a, l, j);
30
31    return j;
32  }
33
34  void sort(char* a, int l, int r){
35    if (l < r){
36      int j = partition(a, l, r);
37      sort(a, l, j-1);
38      sort(a, j+1, r);
39    }
40  }
41
42  int main(int argc, char** argv){
43    char* a = NULL;
44
45    if (argc > 1)
46      a = argv[1];
47    else
48      a = "IU is the best singer!";
49
50    printf("Unsorted: \"%s\"\n", a);
51    sort(a, 0, strlen(a) - 1);
52    printf("Sorted  : \"%s\"\n", a);
53  }
```

<div align="center">singer.c</div>

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

> `gcc -g singer.c -o singer`. We need a `-g` flag for debugging information. The `-o` simply controls the name of the executable that is created which defaults to `a.out` if not specified.

2. When running the program without any arguments (i.e. using the default argument), what line does the segfault happen? Describe the memory operations happening in that line.

> First, we compile the program using the command from the previous problem.
> ```
> > gcc -g singer.c -o singer
> ```

The first step in debugging a segfault is seeing which line it occured in. You can get to the line by letting the program run until it encounters the fault. To get a more holistic view, you can also get the backtrace of the error using `backtrace`.

```
> 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];
(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
```

This tells us the segfault occured on line 6 in `partition` function, which is `a[i] = a[j]`. This line performs two memory operations: a read from `a[j]` and a write to `a[i]`.

When running on Replit, you may get a warning "warning: Error disabling address space randomization: Operation not permitted". Feel free to ignore this as this is a limitation from Replit.

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

Using GDB, break at line 6 where the segfault occurs.

```
> 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!"
(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!"
```

When no argument is passed in, the program defaults to a statically defined string, meaning it resides in the read-only segment. On the other hand, when a specific string is passed in, it is passed in on the stack as an argument. As a result, the two addresses differ significantly

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

The segfaulting line occurs in the `swap` function which is called by `partition` in lines 26 and 29. However, notice that the read from `a[j]` is not a problem as the same read is performed in line 21. As a result, this indicates that a write to `a[i]` is the issue.

This lines up with our observations from the previous question. Since the data is in a read-only segment, it makes sense that the program cannot write to it. As a result, we need to move the initial string to somewhere else on the address space (e.g. heap). To do this, we need to allocate the appropriate amount of space and then copy over the string. This could be accomplished with a `malloc` followed by a `strcpy`, or `strdup`[a] which does the same thing.

Replacing line 48 with `a = strdup("IU is the best char!")` will fix the segfault. You might say that we should free `a` since it'll be allocated on the heap. However, this isn't necessary as the last line `a` is used is line 52, which is also the last line of the program.

---

[a]https://man7.org/linux/man-pages/man3/strdup.3.html

2   x86

x86 is a family of instruction set architectures (ISA) developed by Intel. Unlike RISC-V from CS 61C, x86 is based on based on the complex instruction set computer (CISC) architecture. x86 being a family of ISAs means there are a variety of different dialects of this language. In this class, we will focus on the 32-bit ISA called **IA-32** or **i386** which is the common denominator for all 32-bit x86 processors and hence used in Pintos. While heavily related, this should not be confused with the 32-bit microprocessor i386, also known as Intel 80386). However, we will still occasionally mention the 64-bit ISA **x86-64** as you may come across during some non-Pintos assignments.

Registers

Registers are small storage spaces directly on the processor, allowing for fast memory access.

Recall from CS 61C that RISC-V had 31 **general purpose registers** (**GPR**) `x0` - `x31` with appropriate ABI names (e.g. `x2` = `sp` for stack pointer). Due to architectural differences, x86 only has 8 GPRs.

Register	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)

Due to x86's 16-bit history, the GPRs started as 16-bits and were extended to 32-bits with the `e` prefix (e.g. `eax` for `ax`) and 64-bits with the `r` prefix (e.g. `rax` for `ax`). Each 16-bit GPR can be addressed by the 8-bit LSB (i.e. lower 8 bits) by replacing the last letter with `l` (e.g. `al` for `ax`). `ax`, `cx`, `dx`, and `bx` can also be addressed by the 8-bit MSB (i.e. higher 8 bits) by replacing the last letter with `h` (e.g. `ah` for `ax`).

Akin to the program counter register `pc` from RISC-V, x86 has an **instruction pointer register** `ip`. Like the GPRs, it is extended to 32-bits with the `e` prefix and 64-bits with `r` prefix. `ip` is a special register since it cannot be read and modified like a GPR (i.e. cannot use vanilla memory instructions).

There are other registers such as segment, EFLAGS, control, debug, test, floating point and many more. However, the GPRs and the instruction pointer register are the main ones you will work with in this class.

Syntax

Although IA-32 specifies the registers and instructions, there are two different syntaxes: Intel and AT&T. In this class, we will use the **AT&T syntax** because it is used by the GNU Assembler, the assembler for GCC and thus the standard for Pintos and most Unix-like operating systems like Linux. The two syntaxes have significant differences, so make sure to check which syntax is being used when referencing documentation.

Registers are preceded by a percent sign (e.g. `%eax` for `eax`). Immediates such as constants are preceded by a dollar sign (e.g. `$162` for the constant 162).

The general structure of a line of code is `inst src, dest`. For example, `$movl %ebx, %eax` will move the contents of `%ebx` into `%eax`.

Addressing memory uses the synax of `offset(base, index, scale)` where `base` and `index` are registers, and `offset` and `scale` are integers (`scale` can only take on values of 1, 2, 4, or 8). This accesses the data at memory address `base + index * scale + offset`. All parameters are optional, though most cases you will see will have `base` and `offset`. `scale` The following are some use cases of addressing memory with different instructions.

```
mov 8(%ebx), %eax              Move contents from the address ebx + 8 into eax
mov %ecx, -4(%esi, %ebx, 8)    Move contents in ecx into address esi + 8 * ebx - 4
```

One exception to the above syntax is when using the `lea` instruction which stands for "load effective address". `lea` operates directly on the memory addresses themselves and not the contents contained in the memory addresses. For instance, `lea 8(%ebx), %eax` would put the address ebx + 8 into eax, not the contents at that address.

Each instruction also has a suffix that signifies the operand size. `b` means byte (8 bits), `w` means word (16 bits), and `l` means long (32 bits). These are used when the the intended data size is ambiguous (e.g. `mov $0, (%esp)`).

```
movb $0, (%esp)     Zero out a single byte from the stack pointer
movw $0, (%esp)     Zero out two bytes from the stack pointer
movl $0, (%esp)     Zero out four bytes from the stack pointer
```

The suffixes aren't always necessary when the intended data size can be inferred in some cases (e.g. using a 32-bit register as an operand means a 32-bit operation), but it is good practice to use them regardless.

A key difference of x86 from RISC-V is how much one instruction accomplishes due to its complicated instruction set. For instance, let's examine

Calling Convention

Calling convention is a procedure for how to call and return from functions. They specify stack management, passing in parameters, any registers that need to be saved, stack management, returning values, and more. There are two sets of rules: one for the caller of the function and one for the callee of the function.

Calling conventions are heavily tied into the language that's being compiled. In this class, we will use the calling convention defined by the i386 System V ABI as the default calling convention.

Caller

Before calling the function (i.e. prologue), the caller needs to

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 (i.e. store first parameter at the lowest address). Add necessary padding *before the parameters* to ensure a 16-byte alignment.

Then the caller calls the function by pushing the return address onto the stack and jumping to the function. Once the function call returns (i.e. epilogue), the caller needs to

1. Remove the parameters from the stack.

2. Restore caller-saved GPRs (if any) from the prologue.

Callee

Before executing any function logic (i.e. prologue), the callee needs to

1. Push EBP onto the stack and set ebp to be the new esp (i.e. stack pointer after pushing the ebp). This marks the start of a new stack frame.

2. Allocate stack space for any local variables (i.e. decrement esp).

3. Save callee-saved GPRs (ebx, edi, esi) onto the stack if used during the function call.

Then the callee performs the function logic. Before returning (i.e. epilogue), the callee needs too

1. Store the return value in eax.

2. Restore callee-saved GPRs (if any) from the prologue.

3. Deallocate local variables. While subtracting the correct amount will technically work, a less error prone way is to set esp to be the current ebp, effectively clearing the stack frame.

4. Restore caller's ebp from the stack.

5. Return from the function by popping the return address pushed by the caller in its prologue and jumping to it.

Instructions

There are a few commonly used instructions that will show up in nearly every assembly code due to the calling convention.

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

Keep in mind the effective column shows what the instruction is doing, but it may not exactly be what the processor does. In fact, `call` and `ret` access EIP directly using `mov` which is not allowed.

Optimizations

When compiling with some optimizations using GCC (e.g. `O3` flag), you may notice some violations of these calling conventions, most notably the saving of the base pointer. This is because the base pointer is not a necessity: the stack pointer is sufficient to address anything we need. Even when using RISC-V, the equivalent frame pointer was not saved during calling convention.

Another notable omission is the 16-byte stack alignment which GCC omits by default even without any optimizations. As a result, you will need to specify the `-fno-ipa-stack-alignment` flag when compiling to get the necessary 16-byte alignment.

2.1   Concept Check

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

    > `bp` has a higher memory address. The stack grows downwards, meaning the top of the stack moves towards lower addresses.

2. Based on the differences of RISC and CISC, why might x86 have less GPRs compared to RISC-V?

    > The reduced instruction set of RISC-V means the processor requires less hardware space for transistors, leaving more room for GPRs.

3. Write three different ways to clear the `eax` register (i.e. store a 0).

```
movl $0, %eax

subl %eax, %eax

xorl %eax, %eax
```

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

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

2.2 Reverse Engineering

A Pythagorean triplet $a < b < c$ satisfies the property $a^2 + b^2 = c^2$. `triplet.s` on the right returns the product of a Pythagorean triplet for which $a + b + c = 1000$. `triplet.s` has been assembled without optimizations based on a complete version of `triplet.c` given on the left. Unused labels and directives have been omitted in `triplet.s` for simplicity.

```c
1  int main(void) {
2    for (int a = 1; _____) {
3      int a2 = _____;
4      for (int b = _____) {
5        int b2 = _____;
6        int c = _____;
7        int c2 = _____;
8        if (_____) {
9          return _____;
10       }
11     }
12   }
13   return 0;
14 }
```
<center>triplet.c</center>

```
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
```

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

> `triplet.c` shows that `a` is defined to be a value of 1. In line 5 of `triplet.s`, we see a local variable being defined on the stack with the constant 1. Since `a` is the first variable that needs to be defined in the code, line 5 must correspond to `a`, meaning `a` is defined at EBP - 4 (i.e. four bytes below the base pointer).

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

> After `a` is defined on line 7, we jump to `.L2` in line 8. This brings us to line 43 where we compare `a` (i.e. `-4(%ebp)`) with 333 using the `cmpl` instruction, which subtracts `a` from 333. In line 44, we jump if 333 - `a` is less than 0 to `.L7`. Otherwise, we proceed with the rest of the code which loads 0 into `eax` and performs a return procedure by executing `.L5`. Therefore, `a` must be less than or equal to 333.

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

> Since the code is unoptimized, we trace the code with the appropriate jumps to see where each local variable is defined.
>
> After `a` is defined, we jump to `.L7` where we see in line 12 that `a2` is the next variable defined at EBP - 12. Afterwards, `b` is defined at EBP - 8 in line 14. This is confirmed by the similar loop structure as the code jumps to `.L3` where `cmpl` and `jle` exists as seen for `a`.
>
> The code then jumps to `.L6` where `b2`, `c`, and `c2` are defined at 16, 20, and 24 bytes below the base pointer in lines 19, 23, and 26, respectively.
>
> The stack viewed from the current base pointer looks like

ebp
a
b
a2
b2
c
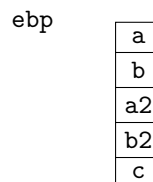
> An important observation is how local variables are not always placed on the stack in the order they are defined in, so it's important to not make such an assumption and instead trace the code. This is due to GCC's stack slot assignment optimizations[a] (not in scope).

4. Fill in the missing code for `triplet.c`.

```
1  int main(void) {
2    for (int a = 1; a <= 333; a++) {
3      int a2 = a * a;
4      for (int b = a; b <= 666; b++) {
5        int b2 = b * b;
6        int c = 1000 - a - b;
7        int c2 = c * c;
8        if (a2 + b2 == c2) {
9          return a * b * c;
10       }
11     }
12   }
13   return 0;
14 }
```

triplet.c

A breakdown of how this code was compiled can be found on this Compiler Explorer snippet[a] which highlights corresponding lines between C and x86 and shows the exact compiler flags used. Feel free to play around with different optimizations and other compiler flags to see how the corresponding x86 code changes.

2.3   Stack Frame

```
1  int p = 0;
2
3  int bar(int x, int y, int z) {
4    int w = x + y - z;
5    return w + 1;
6  }
7
8  void foo(int a, int b) {
9    p = a + b + bar(3, 4, 5);
10 }
```

foobar.c

```
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 the code correspond to a caller/calle prologue/epilogue?

> The most apparent caller/callee pair is `foo`/`bar`. Lines 24-27 are the prologue of the caller that setup the arguments. Line 29 is the epilogue of the caller that cleans up the stack by simply moving up the stack pointer. Lines 4-6 are the prologue of the callee that push the base pointer and make room for local variables. While there's only one local variable `w` in `bar`, the compiler still makes room for 16 bytes due to the default stack boundary of 16 bytes per GCC's `-mpreferred-stack-boundary` flag[a]. Lines 13-15 are the epilogue of the callee that restores the stack pointer and returns back to the caller.
>
> It's important to recognize `foo` is a function as well, meaning it will be in the callee position when it's called by another function. Similar to when `bar` was the caller, lines 17-20 and lines 33-35 serve as the prologue and epilogue respectively when `foo` is the callee.

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

> Line 19 saves the EBX register by pushing it onto the stack. EBX is a callee-saved register, so `foo` is responsible for saving it before using it.

3. Why is EDX not saved by `foo` before calling `bar` despite being used in `bar`?

> GPRS are saved only if the register content needs to persist after the function call. EDX was used as a temporary register for computation in lines 21-23. As a result, its value does not need to persist past the call of `bar`.

4. Draw the stack frame and contents of relevant registers after executing line 13 of `call.s`.

---

[a]https://gcc.gnu.org/onlinedocs/gcc-4.2.4/gcc/i386-and-x86_002d64-Options.html

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`
2

ESP points to the bottom of the stack. EBP points to the "EBP of `foo`". EAX holds 3, EBX holds `a + b`, and EDX holds 3.