CS 33, Summer 2020, with Prof. Ghaforyfard

Week 5 Worksheet - Solutions

**1.** What will the following print out?

typedef struct {

char shookie;

int tata;

char cookie;

double chimmy;

} bt;

void main(int argc, char\*\* argv) {

bt band[7];

printf( “%d\n”, (int)sizeof(band));

}

[1 + (3) + 4 + 1 + (7) + 8] \* 7 = 168

Due to alignment, we need to add the numbers in parentheses

**2.** What is the best\* ordering of the following variables if you want to have a struct that uses all of them? Assume a 64-bit architecture with 4-byte ints.

*\* the ordering that will result in the optimal usage of space.*

char tully;

long stark;

float\* lannister;

double targaryen;

int greyjoy;

float arryn;

We ought to order from largest size to smallest, as structs are x-aligned, where x is the size of the largest data type in the struct. Note that there are several equivalent solutions, and they will all result in a 40-byte struct.

struct Westeros{

float\* lannister; // ALL pointers are 8 bytes

double targaryen; // doubles are 8 bytes

long stark; // longs are 8 bytes

float arryn; // floats are 4 bytes

int greyjoy; // ints are 4 bytes

char tully; // chars are 1 byte

};

**3.** How many bytes of space would these declarations require?

char vermilion[9]; 9 bytes (1 \* 9)

char\* amber[9]; 72 bytes (8 \* 9)

char\*\* chartreuse[9]; 72 bytes (8 \* 9)

int teal[6]; 24 bytes (4 \* 6)

int violet[6][3]; 72 bytes (4 \* 6 \* 3)

int\* magenta[6][3]; 144 bytes (8 \* 6 \* 3)

**4.** Consider the following struct:

typedef struct {

char first;

int second;

short third;

} stuff;

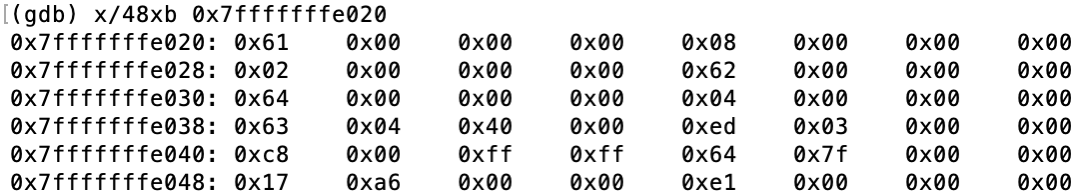
Say we are debugging an application in execution using gdb on a 64-bit, little-endian architecture. The application has a variable called array - defined as:

stuff array[2][2];

Using gdb we find the following information at a particular stage in the application:



And:



What is the value of **array[1][0].second** at this particular stage of the application?

i.e. what would be returned from the statement: printf("%d\n", array[1][0].second);

**1005**

Because of alignment, each object of type “stuff” is 12 bytes.

Due to how arrays are stored in memory,

The array is stored as:

array[0][0], array[0][1], array[1][0], array[1][1]

From the gdb output, we can tell that the array starts at 0x7fffffffe020

array[1][0] is 0x7fffffffe038 to 0x7fffffffe043

Note: this is in hex, so 0x7fffffffe038 + 8 = 0x7fffffffe040

Second is an integer, and is the 5th to 8th byte of an object of type “stuff”

These are bytes 0x7fffffffe03c to 0x7fffffffe03f

They have the values 0xed, 0x03, 0x00, 0x00

Since this system is little endian, the value is 0x000003ed

This is equivalent to 1005

**5.** Consider the following C code:

typedef struct {

char first;

int second;

short third;

int\* fourth;

} stuff;

stuff array[5];

int func0(int index, int pos, long dist) {

char\* ptr = (char\*) &(array[index].first);

ptr += pos;

\*ptr = index + dist;

return \*ptr;

}

int func1() {

int x = func0(1, 4, 12);

return x;

}

Clearly some code is missing - your job is to fill in the blanks! Note that the size of the blanks is not significant. The two functions will be compiled using the following assembly code:

0000000000400492 <func0>:

400492: 8d 04 17 lea (%rdi,%rdx,1),%eax

400495: 48 63 ff movslq %edi,%rdi

400498: 48 63 f6 movslq %esi,%rsi

40049b: 48 8d 14 7f lea (%rdi,%rdi,2),%rdx

40049f: 88 84 d6 60 10 60 00 mov %al,0x601060(%rsi,%rdx,8)

4004a6: 0f be c0 movsbl %al,%eax

4004a9: c3 retq

00000000004004aa <func1>:

4004aa: c6 05 cb 0b 20 00 0d movb $0xd,0x200bcb(%rip)

# 60107c <array+0x1c>

4004b1: b8 0d 00 00 00 mov $0xd,%eax

4004b6: c3 retq

The answer can be derived by tackling func0 first, then func1

**func0**

* From instruction 400492, we can see that the return value is set to %rdi + %rdx, where %rdi is index and %rdx is dist
  + %rdi is set to the first parameter, %rsi to the second parameter, %rdx to the third
  + %eax is unchanged, until instruction 4004a6 with %al
    - This makes sense, since we’re returning the value from dereferencing a pointer to a char, aka a single byte (%al is a single byte)
  + Thus we know **\*ptr = index + dist**
* From instruction 40049b:
  + %rdx is set to 3 \* %rdi
  + %rdx is thus 3 \* index
* From instruction 40049f:
  + 0x601060 is presumably the start of the array
    - This is confirmed in instruction 4004aa, where 60107c is shown to be <array+0x1c>
  + The destination of instruction 40049f is thus:
    - (Start of the array) +  
       8 \* (3 \* %rdi) +  
       pos
    - = (start of array) + (24 \* index) + pos
  + Each object of type stuff is 24 bytes (alignment)
  + ptr from func0 is thus pointing to **array[index].first**
    - The “+ pos” comes from the second line of func0

**func1**

* **(note) there is no call to func0, as this code was produced from gcc -O**
  + Optimization has not been covered yet, but the gist of the problem should be understandable
* From base + displacement addressing modes, students should understand that 0x200bcb(%rip) from instruction 4004aa is location <array + 0x1c>, but just in case, objdump also displays the comment # 0x60107c <array + 0x1c>
  + 0x1c = 28
  + Since each object of type stuff is 24 bytes, we know the second parameter (pos) was called with value 4
    - array[1].first would be at byte 24
    - ptr += 4 would bring us to 28
    - Thus we know **pos = 28 - 24 = 4**
* 0xd = 13
  + Thus we know that the **third parameter (dist) was called with value 12**