Getting Started

To obtain a copy of today's activity, log into a shark machine and do the following:

```
$ wget http://www.cs.cmu.edu/~213/activities/machine-data.tar
$ tar xf machine-data.tar
$ cd machine-data
```

All of the questions below involve working with the program `data-layout` in the debugger, so get ready to do that now:

```
$ gdb ./data-layout
(gdb) r
```

You may find it helpful to pull up some older handouts: at the end of the handout from the first class on assembly language (https://www.cs.cmu.edu/~213/activities/gdb-and-assembly.pdf) is a list of useful GDB commands. At the end of the handout from the class on procedure calls (https://www.cs.cmu.edu/~213/activities/machine-procedures.pdf) is a table listing all the x86's integer registers and describing how each might be used in a procedure call (e.g. `%rdi` holds the first argument to the called function).

1 Integers and Local Variables

First, let's review storage of local variables.

Problem 1. This function has a 32-bit integer local variable:

```
int returnOne(void) {
   int local = -1;
   return abs(local);
}
```

Disassemble `returnOne` and notice where `local` is stored. Now, what if we needed to take the address of `local`? What problem might we run into, and what do you expect the compiler to do about it?

The variable `local` is stored in register %edi. The problem is, registers do not have addresses. If the code took the address of `local`, the compiler would have to store the variable on the stack, instead of keeping it in a register.

Problem 2. Test your expectation from Problem 1 by disassembling the function `returnOneTwo`. Its code is something like this:

```
int returnOne(void) {
   int local = -1;
   return absp(/* ??? */);
}
```

Based on what you see, what do you think the declaration (prototype) of `absp` is?

```
extern int absp(int *p);
```

2 Arrays

Now let's review arrays of integers. The `data-layout` program contains one, defined as:

```
int courses[4] = {0x15213, 0x15513, 0x18213, 0x18600};
```

Problem 3. Examine this array's memory layout:

```
(gdb) x/4wx courses
```

Also examine the disassembly of the function `getNth`, which accesses an array of integers:

```
int getNth(int *arr, size_t index) {
    return arr[index];
}
```

What is the *stride* of an array of `int` (the number of bytes occupied by each entry)?
__4__ bytes

The stride does not appear in the C code for `getNth`. Does it appear in the disassembly? If so, describe how it is used, in terms of the C-level variables.

Yes, it does. The compiler multiplies `index` by the stride, before adding it to the pointer `arr`, to get the address of the Nth element of the array being accessed. This is necessary because memory is byte addressable; that is, every memory address corresponds to an individual byte, even if it's only holding part of a wider value.

A while ago we told you that strings in C are "just" arrays of characters. 'c'ontinue execution of the program now. It will stop inside a function that receives a string as an argument. Print the string with this command:

```
(gdb) x/s $rdi
```

Print it out again, as an array of bytes, using this command:

```
(gdb) x/12bx $rdi
```

Problem 4. There's no information in any of the registers that says *how long* the string pointed to by `%rdi` is. How does the `x/s` command know how long it is?

There is a NUL character (ASCII character code 0, often written `'\0'`) at the end of every C string. To find out how long the string is, you have to scan the string, counting bytes, until you reach the NUL.

3 Structs

C allows you to define custom structured types comprising multiple named fields. Take for example the struct defined as follows:

```
struct course {
  int cs_ugrad;
  int cs_grad;
  int ece_ugrad;
  int ece_grad;
};
```

Continue execution of the program now (use the `c` command again). It will stop inside another function; this one takes a `struct course *` as its first argument.

Problem 5. Dump out the contents of the `struct course *` that was passed to the current function using this command:

```
(gdb) x/4wx $rdi
```

Did you notice anything familiar about the layout?

The layout to this 4 integer struct looks the same as the layout of the 4 integer array from Problem 3.

`struct course` is quite simple: all its members have the same type. Structs can be much more complicated, though. Their members don't even all have to have the same size! Consider this one:

```
struct increasing {
  char a;
  short b;
  int c;
  long d;
};
```

Problem 6. Suppose you had an instance of `struct increasing` whose fields were initialized to 0x0a, 0x0b0b, 0x0c0c0c0c, and 0x0d0d0d0d0d0d0d0d, respectively. The table below has boxes for 32 bytes, which should be more than enough to hold a `struct increasing`. Write, in each box, what value ('a', 'b', 'c', or 'd') you think will be in each byte. If you think a byte will be unused, leave it blank.

0x00								
0x08								
0x10								
0x18								

After you fill in the table above, use the `c` command to resume execution. GDB will stop inside a function that has received a `struct increasing *` as its first argument; its fields have been initialized to 0x0a, 0x0b0b, 0x0c0c0c0c, and 0x0d0d0d0d0d0d0d0d, just like we said before. Print out its contents, byte by byte, and use that information to fill in the table below with the values each byte actually has—'a', 'b', 'c', or 'd', just like before. If GDB says a byte is zero, leave it blank.

```
(gdb) x/32bx $rdi
```

0x00	a		b	b	c	c	c	c
0x08	d	d	d	d	d	d	d	d
0x10								
0x18								

Compare the two tables. If the data in memory isn't where you thought it would be, why do you think that might have happened?

The fields of a struct need to be *aligned*. Any "scalar" field (not an array or a nested struct) must be located at an offset (from the beginning of the struct) that is a multiple of its size. The C compiler cannot change the order of fields within a struct, so instead, to give all the fields the alignment they need, it inserts *padding* in between the fields. In the case of `struct increasing`, there needs to be one byte of padding before `b` but then the math works out neatly so that `c` and `d` get the alignment they need without any additional padding.

Problem 7. Now consider this struct, which has the same fields as `struct increasing`, but with the fields in a different order.

```
struct rearranged {
  char a;
  long b;
  short c;
  int d;
};
```

Will this type take up more or less space than `struct increasing`?

This type will take up more space because the order of its elements produces more padding.

Problem 8. The function GDB is currently stopped in received a pointer to a `struct rearranged` as its *second* argument. Use this to check your answer to Problem 7 and fill in the table below with the layout.

0x00	a							
0x08	b	b	b	b	b	b	b	b
0x10	c	c			d	d	d	d
0x18								

4 Arrays of Structs

Next, we'll look at a way to store many instances of a particular structured type: an array of structs. For instance, we might have:

```
struct pair {
  int large;
  char small;
};
struct pair pairs[2] = {
  {0xabababab, 0x1},
  {0xcdcdcdcd, 0x2}
};
```

Problem 9. What stride do you expect this array to have? 8 bytes

Problem 10. Check your guess:

```
(gdb) x/16bx &pairs
```

What stride did the array actually have? 8 bytes
Where did the compiler insert padding, if any? After `small`

Why did it need to do that?

Padding after `small` makes `struct pair`'s size be a multiple of 4. That's necessary so that the *second* element of the array will have an address that's a multiple of 4, satisfying the alignment requirement of `large`. In general, a struct will always be padded at the end to make its size a multiple of the alignment required by its fields.

Conversely, structs can contain arrays. In this case, the struct will be aligned to the width of the array's *element* type. Here's an example:

```
struct triple {
  short large[2];
  char small;
};
```

Problem 11. How will this struct's size compare to that of `pair`?

This struct is smaller, since it requires less padding. Both have elements whose total size is 5 bytes. `pair` is aligned to the width of `int` (4 bytes), so it gets padded to a length of 8. `triple` is only aligned to the width of `short`, (2 bytes), so it only gets padded to a length of 6.

5 2-D Arrays

Types can be nested arbitrarily. We'll continue by looking at arrays of arrays.

There are actually two different ways to create arrays of arrays ("multi-dimensional arrays") in C. Both ways allow arrays with arbitrarily many dimensions. Each is more convenient in some circumstances.

First, let's see how this declaration is laid out in memory:

```
int8_t nested[2][3] = {{0x00, 0x01, 0x02}, {0x10, 0x11, 0x12}};
```

You'll probably want a command such as:

```
(gdb) x/6bx &nested
```

Problem 12. What stride do the "inner" arrays have? 1 bytes
How about the "outer" ones? 3 bytes

Problem 13. Disassemble the function `access`. Take note of how array strides are embedded in its assembly code. Here is its source code:

```
int8_t access(int8_t (*arr)[3], size_t row, size_t column) {
   return arr[row][column];
}
```

This function is designed to be used with an array like `nested`. Could it also be used with an array declared like this: `int8_t flipped[3][2]`?

No, the dimensions (specifically, the outer stride) do not match. So the compiler will not be able to access the fields of the array correctly.

Problem 14. Now, experiment with GDB commands to examine the layout of *this* multi-dimensional array, which is structured differently:

```
int8_t first[3] = {0x00, 0x01, 0x02};
int8_t second[3] = {0x10, 0x11, 0x12};
int8_t *multilevel[2] = {first, second};
```

(If you need a hint, 'c'ontinue the program and read what it prints.) What stride does the outer array have this time? 8 bytes

Problem 15. An accessor for this type of 2-D array appears below; note the subtle difference in its signature. Disassemble it to see what a difference this makes!

```
int8_t accessMultilevel(int8_t **arr, size_t row, size_t column) {
  return arr[row][column];
}
```

Do you think this function would still be useful if `first` and `second` each had 4 elements? How about if they had two different lengths?

This function would still work because its assembly does not use the lengths of `first` and `second`.

Problem 16. Imagine if we had instead defined multilevel as:

```
int8_t *multilevel[2] = {first, first};
```

What effect would we observe if we modified an element of `first`?

The modification would be observable via both `multilevel[0]` and `multilevel[1]`.

6 Endianness (Optional)

If you have extra time, let's take a more detailed look at the byte-level representation of multi-byte integers.

When multi-byte data is stored in byte-addressable memory, it becomes possible to observe it two different ways: as a single "word" (multi-byte unit), or as a sequence of bytes. Given `int x`, for instance, the hardware must consistently treat `((char *)&x)[0]` as some specific 8-bit subset of the 32-bit `int`. This has given rise to a sometimes heated debate over *endianness*, the rule for which part of a number should "come first"

in memory. Should it be the 8 bits with the *highest* place value ("big-endian") or the 8 bits with the *lowest* place value ("little-endian")?[1]

Problem 17. To see a demonstration of endianness in action, let's look back at the `courses` global variable. (Recall that it is an array of 32-bit integers.) Run this GDB command:

```
(gdb) x/4wx &courses
```

That command interprets every consecutive 4 bytes of the array as a single integer.

But what happens if we ask GDB to print each byte individually? Run this command:

```
(gdb) x/16bx &courses
```

Stare carefully at that mess until you have convinced yourself that it really is the same data you saw before! The reason it looks different is that x86-64 is a *little-endian* architecture: it stores the lowest-order bits of a wide type in the byte with the lowest memory address.

What disadvantage of little-endian did you just observe?

Little endianness is harder to read in a byte-by-byte memory dump, because it is the opposite of the way we write numbers on paper—we could say that our conventional notation for large numbers, e.g. "123 456 789", is big-endian. When debugging, this means you must transpose the bytes of a memory dump in your head, or know debugger commands that do it for you.

Problem 18. Now let's look at an *advantage* of little-endian byte order. Disassemble the function `narrowingCast`. Its C source code looks like this:

```
int narrowingCast(long *num) {
  return (int) *num;
}
```

How would the assembly of this function differ if x86-64 were a big-endian architecture?

We would have `mov 4(%rdi), %eax` instead of `mov (%rdi), %eax`.

Little-endian byte order means the address of an integer's lowest bits is the same as the address of the complete integer. This means we can truncate integers simply by reading fewer bytes from the same memory address.

[1] These names are a reference to the 19th century satirical novel *Gulliver's Travels*, in which, at one point, two countries fight a war over the best way to crack open hard-boiled eggs.