1. After running this command:

```
ctt -g -c product.cpp
```

a product object executable is generated. (product.o)

2. After running this command:

```
nm product.o | grep product
```

This is the output:

```
0000000000000zeb T _Z7productPKim
```

This symbol is called `_Z7productPKim`.

3. After running the following command:

```
c++filt _Z7productPKim
```

This ctt name was demangled into the signature

```
product(int const*, unsigned long)
```

4. After running the command:

```
nm product.o | grep main
```

These are the outputs:

```
000000000000038e t _GLOBAL__sub_I_main
0000 0000 0000 0000 T main
```

The symbol name for the main function is main. This differs from the product symbol name because since main() cannot be overloaded because it is treated as a C function instead of C++, the name does not need to be mangled. Therefore, the symbol name is just the function name, main.

5. After running the following command:

`make product`

these commands are executed:

`$(CXX) $(LDFLAGS) product.o -o product`

where `gtt` is passed to `CXX` and `-g` is passed to `LDFLAGS`.

6. After running this command:

`objdump -S product.o | ctt filt > product.o.dump`

and opening the product.o.dump file, we find the numeric offset: `0000000000000zeb`

I also noticed that there is an overload of product of type int that takes a const int that is a pointer, and a len of type size_t.

7. After running the following command:

objdump -S --disassemble=_Z7productPKim product.o | c++filt > product.dump

this says product.o . dump

A new dump file is created which only contains the disassembly of the product function.

8. After running this command:

objdump -S --disassemble=_Z7productPKim product | cttfilt product.dump

We see the difference is that the instructions are stored at different places in memory but each memory location is the same distance apart.

9. Command-line arguments:

Ø ./product

./product - computes a product of integers

Usage: ./product INTEGER...

Ø

2.2.1 We observe that the product isn't actually calculated, the output is simply each inputted argument on it's own new line.

2: The output is the same as step 9 in 2.1.

3: The output is the same as step 1 in 2.2.

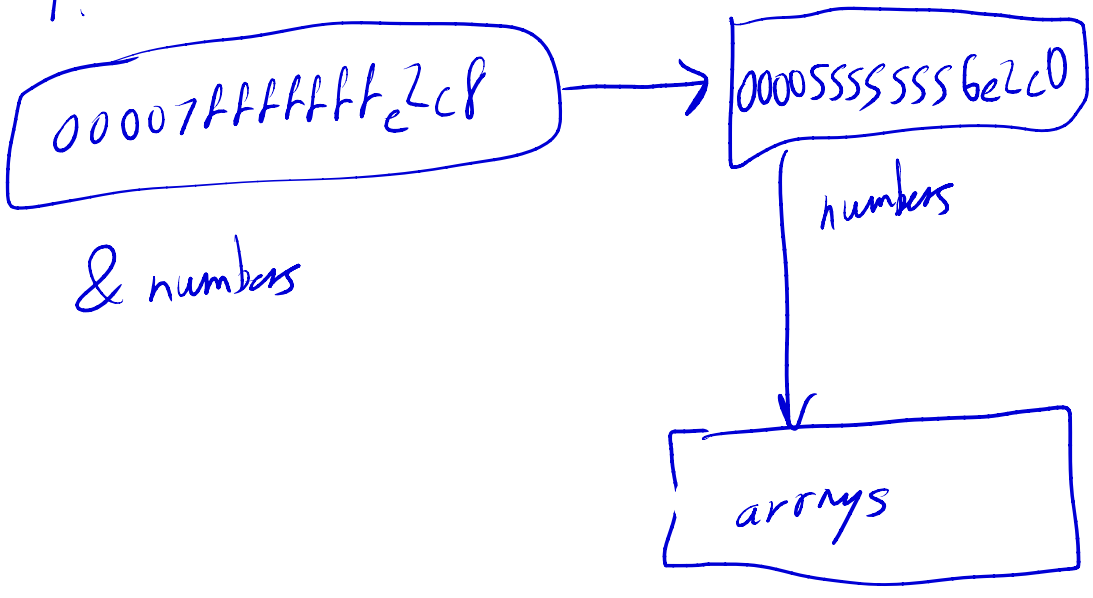
4: By running breakpoint set -l 45 in lldb.

5: The next instruction to be executed is rip.

6. The numbers pointer points to 0000555555556e2c0.

7. The address of the numbers pointer is 00007fffffffe2c8.

8-4:


0000 7fffffffe2c8
& numbers
0000 5555 5556 e2c0
numbers
arrays

10. Addresses in call stack | Instructions

288c: 4883 7de0 00	cmpq $0x0, -0x20(%rbp)
2891: 75 07	jne 289a
2893: 48 8b 45 e8	mov -0x18(%rbp), %rax
2897: 8b 18	mov (%rax), %ebx
2899: 48 8b 45 e0	mov -0x20(%rbp), %rax
289d: 48 8d 50 ff	lea -0x1(%rax), %rdx
28a1: 48 8b 45 e8	mov -0x18(%rbp), %rax
28a5: 48 83 c0 04	add $0x4, %rax
28a9: 48 89 d6	mov %rdx, %rsi
28ac: 48 89 c7	mov %rax, %rdi
28af: e8 bc ff ff ff	call 2870
28b4: 0f af c3	imul %ebx, %eax
28b7: 48 8b 5d f8	mov -0x8(%rbp), %rbx
28bb: c9	leave
28bc: c3	ret

11. 0x5555 5556 e2cc: 00 00 ... 21 0000

0x5555 5556 e2dc: 00 00 0000 6e 555555 05 00000bf046f75

these are the addresses pointed to by numbers.

0x00007 ffffffff e238

this is the address of the numbers pointer.

12. a)

@ 0x7fffffffe238: cce2 5655 5555 0000 c8e2 5655 5555 0000

b) Here is a p at 0x7fffffffe2d8.

Here is an r at 0x7fffffffe3a8.

p stands for parameter, r stands for return address.