Reference

x86_64 Registers We've Used

`rax`	Return values/expression results
`rsp`	Stack Pointer, refers to return address at start of function, used to look up variables
`rdi`	Holds 1st argument in "standard" x86-64 calling convention
`rsi`	Holds 2nd argument in "standard" x86-64 calling convention
`rdx`	Holds 3rd argument in "standard" x86-64 calling convention
`rbx/rcx`	Used by us as temporary storage/for tag checking
`r15`	In our class convention, stores the address of the next free space to allocate

x86-64 Instructions

`mov <reg>, <val>`	Move value to register
`mov <mem>, <val>`	Move value to memory (val can be register or immediate)
`push <val>`	Subtract 8 from `rsp` and store `<val>` at `[rsp]`
`pop <reg>`	Load value from `[rsp]` into `<reg>` and add 8 to `rsp`
`add/sub/imul <reg>, <val>`	Arithmetic
`and/or/xor <reg>, <val>`	Bitwise operators
`shr <reg>, <val>`	Shift `<reg>` right by `<val>` bits, filling with 0s
`sar <reg>, <val>`	Shift `<reg>` right by `<val>` bits, maintaining sign bits
`shl <reg>, <val>`	Shift `<reg>` left by `<val>` bits, filling with 0s
`test <reg>, <val>`	Bitwise and `<val>` and `<reg>` for condition codes, reg unchanged
`cmp <reg>, <val>`	Subtract `<val>` from `<reg>` and set condition codes, `<reg>` unchanged
`cmove/cmovl/cmovne/... <reg1>, <reg2>`	Move the value from `reg2` to `reg1` if the condition codes match
`<label>:`	Create a label (not really an instruction)
`jmp <label>`	Unconditional jump
`je/jne/jg/jge/jl/jle/jo <label>`	Conditional jumps based on condition codes
`call <label>`	Push (as with `push`) the address of next instruction and jump to `<label>`
`ret`	Pop the stack (as with `pop`) and jump to it
`qword`	Not an instruction, but a *size modifier*. Some instructions, like `push [r15]`, don't know if it's intended to move 1, 4, or 8 bytes. We've often used `qword` to disambiguate which means 8 bytes.

Rust Reference

`e >> n`	Shift `e` to the right by `n` bits. Do signed/unsigned shift based on type (e.g. `i64` shifts signed, `u64` shifts unsigned)
`e1 & e2, e1 | e2`	Bitwise operators
`e as t`	Interpret the bits of the value `e` as type `t`. For example `let num_unsigned = num as u32;` when `num` is `i64` will reinterpret the lower 32 bits of the signed integer as an unsigned one.
`char`	A type in Rust, a single Unicode “scalar value”, 32 bits/4 bytes long.
`v[..]`	Create a *slice* of a vector or string value `v`. Useful for pattern matching vectors and for getting a `&str` from a `String`.
`*v`	Access the memory at a raw pointer `v`, which must have a type like `*mut T` or `*const T`. Must appear in an `unsafe` block
`*v = e`	Assign the result of `e` into memory at the address given by raw pointer `v`, which must be `*mut T` with `e` having type `T`
`v.offset(n)`	For a raw pointer `v`, return a new raw pointer offset by `n * size` bytes, where `size` is the number of bytes in the type of `v`
`*mut T`	A raw pointer of type `T` that allows reading and mutation at the given address
`*const T`	A raw pointer of type `T` that allows reading but not mutation at the given address
`unsafe  e`	Allows raw pointer manipulation inside the block (and other unsafe operations)
`isize`	A type representing a size of some data. In this exam/in our programs, it’s OK to freely convert (with `as`) between integer types like `i64` and `isize`. Expected as the argument for e.g. `offset`