Week 6 Thursday
Generics Review
Assembly I

Fill in the check-in form on cs107a.stanford.edu!
Starter Code

```bash
git clone /afs/ir/class/archive/cs/cs107a/cs107a.1226/WWW/exercises/generics3
```
Announcements

- Get extra credit for filling out an evaluation: https://edstem.org/us/courses/20849/discussion/1480721
- Midterm grading ~this weekend
- 2nd round of 1:1s next week to coincide with assign5 release
- assign4 due next Wednesday
Unix Tip Spotlight

- Procrastinating via `emacs`
  - M-X doctor
    - Talk to a virtual psychotherapist
  - F10 → Tools → Games
    - Exactly what it sounds like
- “`emacs` is a great operating system, it just lacks a good text editor”
Agenda

- Generics Review: move_to_back.c
- Generics Review: struct_sort.c
- Intro x86
- Assembly Exercises
Generics Review

move_to_back.c
Generics Review

struct_sort.c
Intro x86
CPUs don’t speak C, they speak assembly

- There are many assembly languages, one for each family of CPUs (for each “instruction set architecture” (ISA))
  - x86 (old desktops/laptops/servers)
  - x86-64 (most desktops/laptops/servers)
  - ARM (phones, modern Macbooks, some video game consoles)
  - RISC-V (up-and-coming)
  - PowerPC (really old Macs)
  - MIPS (some universities use this to teach assembly)
C Compilation - what happens when you run make

```c
#define MAGIC 7

int x = 5;
x += MAGIC;
```

```
int x = 5;
x += 7;
```

```
movl $5, %rax
addl $7, %rax
```

```
010101010100101
101000101001010
010110101001101
```

```
+ 010101010100101
101000101001010
```

```
010101010100101
010110101001101
```

```
./myuniq
```

- C Source Code
- Preprocessed C Code
- Assembly (e.g. x86)
- Binary Object File
- RunnableExecutable

Steps:
- Preprocessing
- Compilation
- Assembling
- Linking
Why do we care about assembly?

- C and other languages are convenient to program in, and abstract away a lot of the details and ugliness of assembly. But sometimes we really do care about the details. Some examples:
  - Security - understanding what happens when a bug occurs
  - Performance - fine-tuning the code that actually runs
  - Compilers - when you make your own programming language
  - Operating Systems - some things are too low-level to use anything other than assembly
  - Sometimes you don’t have the source code
How to inspect the assembly of a program

- `objdump -d <program name>`
  - Pass the program, not the C file
- `Use gdb, go into layout asm or layout split`
  - I generally recommend this, as when you explore assembly, it often helps to step through the program and verify your inferences about how it works
Part 1: ATM
○ You will be given buggy C code simulating an ATM machine.
○ Your job is to find the bugs in the C code, discover the corresponding buggy behaviors in assembly, and craft inputs that exploits the flaws.

Part 2: Secure Vault (formerly Binary Bomb)
○ You will be given an executable (no C code) simulating a password-protected bank vault
○ Your job is to discover the correct passwords needed to break into the vault by reading the verification code in the assembly and inferring what password is needed to pass it.
○ You will need to use gdb carefully to avoid triggering any alarms.
We won’t be spending too much time reviewing lecture material
  ○ Assembly language is a lot of memorization and we don’t have time to review all of it
  ○ We’ll focus on exercises, where you’ll put into practice the content from lecture
  ○ By next Tuesday, we’ll have finished all the assembly lectures. You’ll need to be caught up to get the most of out CS 107A exercises.
x86 Instructions

- Use the CS 107 x86 Reference:
- [https://web.stanford.edu/class/archive/cs/cs107/cs107.1226/guide/x86-64.html](https://web.stanford.edu/class/archive/cs/cs107/cs107.1226/guide/x86-64.html)
General-Purpose Registers

<table>
<thead>
<tr>
<th>64-bit</th>
<th>RAX</th>
<th>RBX</th>
<th>RCX</th>
<th>RDX</th>
<th>RSI</th>
<th>RDI</th>
<th>RBP</th>
<th>RSP</th>
<th>R8</th>
<th>R9</th>
<th>R10</th>
<th>R11</th>
<th>R12</th>
<th>R13</th>
<th>R14</th>
<th>R15</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit</td>
<td>EAX</td>
<td>EBX</td>
<td>ECX</td>
<td>EDX</td>
<td>ESI</td>
<td>EDI</td>
<td>EBP</td>
<td>ESP</td>
<td>R8D</td>
<td>R9D</td>
<td>R10D</td>
<td>R11</td>
<td>R12D</td>
<td>R13D</td>
<td>R14D</td>
<td>R15D</td>
</tr>
<tr>
<td>16-bit</td>
<td>AX</td>
<td>BX</td>
<td>CX</td>
<td>DX</td>
<td>SI</td>
<td>DI</td>
<td>BP</td>
<td>SP</td>
<td>R8W</td>
<td>R9W</td>
<td>R10W</td>
<td>R11W</td>
<td>R12W</td>
<td>R13W</td>
<td>R14W</td>
<td>R15W</td>
</tr>
<tr>
<td>8-bit</td>
<td>AL</td>
<td>ZBL</td>
<td>CL</td>
<td>DL</td>
<td>SIL</td>
<td>DIL</td>
<td>BPL</td>
<td>SPL</td>
<td>R8B</td>
<td>R9B</td>
<td>R10B</td>
<td>R11B</td>
<td>R12B</td>
<td>R13B</td>
<td>R14B</td>
<td>R15B</td>
</tr>
</tbody>
</table>
Some Registers with Special Meanings (memorize these)

<table>
<thead>
<tr>
<th>Register</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rax</code></td>
<td>Return value of a function</td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td>First argument to a function</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>Second argument to a function</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>Third argument to a function</td>
</tr>
<tr>
<td><code>%rip</code></td>
<td>Address of next instruction to execute</td>
</tr>
<tr>
<td><code>%rsp</code></td>
<td>Address of the top of the current stack</td>
</tr>
</tbody>
</table>
mov

- `mov $3, $eax`
  - Put 3 in %eax
- `mov $3, 4($rax)`
  - Put 3 at the location %rax+4
- `mov $eax, $ebx`
  - Put %eax’s value into %ebx
- `mov $eax, 4($rax, $rbp, 2)`
  - Put %eax’s value at location %rax + 2*%rbp + 4
- `mov 4($rax, $rbp, 2), $eax`
  - Put the value at location %rax + 2*%rbp + 4 into %eax
Assembly Exercises