

CS107 Lecture 16: Two-Minute Etudes

Problem: Reverse Compilation

Each function below is shown with one or two lines of C replaced by blanks, alongside the complete x86-64 assembly **gcc** generated for the entire function, including the instructions for the missing lines. Work out what each missing line of C must be.

The majority of the work is designed to exercise your understanding of addressing modes and the various **mov** instructions. If I need to employ something I've not taught you yet, I'll explain it right in the problem statement.

- a. Recall that **%rdi** stores a function's first parameter, which here is the address `arr`. The **add** instruction here updates **%rax** to be 5 greater than what it was before.

C Code:

```
void rotate(long arr[]) {  
    long first = arr[0];  
    _____;  
    _____;  
}
```

x86:

```
rotate:  
    movq    (%rdi), %rax  
    movq    16(%rdi), %rdx  
    movq    %rdx, (%rdi)  
    addq    $5, %rax  
    movq    %rax, 16(%rdi)  
    ret
```

- b. The first three parameters are passed using **%rdi**, **%rsi**, and **%rdx**, and **%rax** (and the 32-bit **%eax** view within it) ultimately stores the return value. That `sub` instruction subtracts **%edx** from **%eax** and updates **%eax** itself with the result.

C Code:

```
int grab(int arr[],  
         long i, long j) {  
    int x = arr[i];  
    _____;  
    _____;  
    return x - y;  
}
```

x86:

```
grab:  
    salq    $2, %rsi  
    movl    (%rdi,%rsi), %eax  
    movl    12(%rdi,%rdx,4), %edx  
    movl    %edx, -4(%rdi,%rsi)  
    subl    %edx, %eax  
    ret
```

- c. You'll need to intuit how `add` works more generally based on the previous problem, but otherwise there's nothing here you've not seen in lecture thus far.

C Code:

```
int mark(char *s,
          unsigned char *t,
          long k) {
    int a = s[k];
    _____;
    return a + b;
}
```

x86:

```
mark:
    movsbl    (%rdi,%rdx), %eax
    movzbl    2(%rdx,%rsi), %edx
    addl      %edx, %eax
    ret
```

- d. The missing line below includes a typecast. Otherwise, everything should feel familiar.

C Code:

```
long blend(short *arr,
            long i) {
    long a = arr[i];
    _____;
    return a + b;
}
```

x86:

```
blend:
    movswq    (%rdi,%rsi,2), %rax
    movslq    (%rdi,%rsi,4), %rdx
    addq      %rdx, %rax
    ret
```

- e. This next example relies on the following struct definition:

```
typedef struct {
    short x, y; int id; char *label;
} point;
```

Compilers generally pack the fields of a record together in memory, and that's the case here. With the above definition, the **point** would be 16 bytes: the first two bytes would house the **x** field, the next two would house **y**, the next four bytes would be used for **id**, and the remaining eight would be used for **label**.

C Code:

```
void decor(point *p,
            point *q) {
    _____;
    _____;
}
```

x86:

```
decor:
    movl      4(%rsi), %eax
    addl      $1, %eax
    movl      %eax, 4(%rdi)
    movswq    2(%rdi), %rax
    addq      8(%rsi), %rax
    movq      %rax, 8(%rdi)
    ret
```

Answers:

```
a.arr[0] = arr[2]; arr[2] = first + 5;
b.int y = arr[j] + 3; arr[j] - 1 = y;
c.int b = q[t[k] + 2];
d.long b = (int *)arr[t[j]];
e.p->id = q->id + 1; p->label = q->label + p->y;
```