CS107, Lecture 14
Calling: Callers, Callees, and Callbacks

Reading: B&O 3.1-3.4
int elem_arithmetic(int nums[], int y) {
    int z = nums[________] * ________;

    z -= ________;

    return ________;
}

----------
// nums in %rdi, y in %esi
elem_arithmetic:
    movl %esi, %eax
    imull 4(%rdi), %eax
    movslq %esi, %rsi
    subl (%rdi,%rsi,4), %eax
    lea 2(%rax, %rax), %eax
    ret
int elem_arithmetic(int nums[], int y) {
    int z = nums[1] * y;
    z -= __________;
    return __________;
}

---------
// nums in %rdi, y in %esi
elem_arithmetic:
    movl %esi, %eax          // copy y into %eax
    imull 4(%rdi), %eax      // multiply %eax by nums[1]
    movslq %esi, %rsi        // sign-extend %esi to %rsi
    subl (%rdi,%rsi,4), %eax
    lea 2(%rax, %rax), %eax
    ret

Work through the last two blanks in groups and input your answer for the first blank on PollEv: pollev.com/cs107 or text CS107 to 22333 once to join.
int elem_arithmetic(int nums[], int y) {
    int z = nums[1] * y;

    z -= nums[y];

    return 2 * z + 2;
}

----------
// nums in %rdi, y in %esi
elem_arithmetic:
    movl %esi, %eax       // copy y into %eax
    imull 4(%rdi), %eax   // multiply %eax by nums[1]
    movslq %esi, %rsi     // sign-extend %esi to %rsi
    subl (%rdi,%rsi,4), %eax // subtract nums[y] from %eax
    lea 2(%rax, %rax), %eax // multiply %rax by 2, and add 2
    ret
test practice: What’s the C code?

```c
0x400546 <test_func>  test  %edi,%edi
0x400548 <test_func+2>  jns  0x400550 <test_func+10>
0x40054a <test_func+4>  mov  $0xfeed,%eax
0x40054f <test_func+9>  retq
0x400550 <test_func+10>  mov  $0xaabbccdd,%eax
0x400555 <test_func+15>  retq
```
int test_func(int x) {
    if (x < 0) {
        return 0xfeed;
    }
    return 0xaabbccdd;  (or anything like this)
}
Practice: “Escape Room”

What must be passed to the escapeRoom function such that it returns true (1) and not false (0)?

You don’t have to reverse-engineer C code exactly!
What must be passed to the escapeRoom function such that it returns true (1) and not false (0)?

First param > 2 or == 1.
• Machine code instructions live in main memory in the text/data segment.

• `%rip` is a special register that stores a number (an address in the text/data segment) that corresponds to the next instruction to execute.

• It marks our place in the program’s instructions and advances us instruction by instruction through the program.

• Special hardware handles the variable op-code sizes and correctly updates `%rip`.

• `jmp` instructions work by adjusting `%rip` by a specified amount.
To call a function in assembly, we must do a few things:

- **Pass Control** – `%rip` must be adjusted to execute the callee’s instructions, and then resume the caller’s instructions afterwards.

- **Pass Data** – we must pass any parameters and receive any return value.

- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.
%rsp

• %rsp is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).
• `%rsp` is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).
%rsp is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).
• `%rsp` is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).
• `%rsp` is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).

Key idea: `%rsp` must point to the same place before a function is called and after that function returns, since stack frames go away when a function finishes.
push

- The `push` instruction pushes the data at the specified source onto the top of the stack, adjusting `%rsp` accordingly.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushq S</td>
<td>R[%rsp] ← R[%rsp] - 8; M[R[%rsp]] ← S</td>
</tr>
</tbody>
</table>

- This behavior is equivalent to the following, but pushq is a shorter instruction:

  ```
  subq $8, %rsp
  movq  S, (%rsp)
  ```

- Sometimes, you’ll see instructions just explicitly decrement the stack pointer to make room for future data.
The **pop** instruction pops the topmost data from the stack and stores it in the specified destination, adjusting `%rsp` accordingly.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
</table>
| `popq D`    | $D \leftarrow M[R[\%rsp]]$  
$R[\%rsp] \leftarrow R[\%rsp] + 8;$  |

This behavior is equivalent to the following, but `popq` is a shorter instruction:

- `movq (%rsp), D`  
- `addq $8, %rsp`

Sometimes, you’ll see instructions just explicitly increment the stack pointer to pop data.
Stack Example

Initially

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x123</td>
</tr>
<tr>
<td>%rdx</td>
<td>0</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

Stack "bottom"

Increasing addresses

Stack "top"

pushq %rax

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x123</td>
</tr>
<tr>
<td>%rdx</td>
<td>0</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x100</td>
</tr>
</tbody>
</table>

Stack "bottom"

Increasing addresses

popq %rdx

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x123</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x123</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

Stack "bottom"

Increasing addresses
To call a function in assembly, we must do a few things:

- **Pass Control** – %rip must be adjusted to execute the callee’s instructions, and then resume the caller’s instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.
**Problem:** %rip points to the next instruction to execute. To call a function, we must remember the next caller instruction to resume at after.

**Solution:** push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.

E.g. main() calls foo():

```
%rip 0x3021
%rsp 0xff20
```
**Problem:** %rip points to the next instruction to execute. To call a function, we must remember the next caller instruction to resume at after.

**Solution:** push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.

*E.g. main() calls foo(*:*

```
Stack

... 0x3026

main()

... 0xff18

%rip 0x3021

%rsp 0xff18
```
**Problem:** %rip points to the next instruction to execute. To call a function, we must remember the next caller instruction to resume at after.

**Solution:** push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.

E.g. `main()` calls `foo()`:
**Problem:** %rip points to the next instruction to execute. To call a function, we must **remember** the next caller instruction to resume at after.

**Solution:** push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.

*E.g. main() calls foo():*
**Problem:** %rip points to the next instruction to execute. To call a function, we must remember the next caller instruction to resume at after.

**Solution:** push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.

_E.g. main() calls foo():_
Call And Return

The **call** instruction pushes the address of the instruction immediately following the **call** instruction onto the stack and sets %rip to point to the beginning of the specified function’s instructions.

```
call Label

call *Operand
```

The **ret** instruction pops this instruction address from the stack and stores it in %rip.

```
ret
```

The stored %rip value for a function is called its **return address**. It is the address of the instruction at which to resume the function’s execution. (not to be confused with **return value**, which is the value returned from a function).
To call a function in assembly, we must do a few things:

• **Pass Control** – `%rip` must be adjusted to execute the function being called and then resume the caller function afterwards.

• **Pass Data** – we must pass any parameters and receive any return value.

• **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.
Register Restrictions

There is only one copy of registers for all programs and functions.

• **Problem:** what if *funcA* is building up a value in register `%r10`, and calls *funcB* in the middle, which also has instructions that modify `%r10?* *funcA*’s value will be overwritten!

• **Solution:** make some “rules of the road” that callers and callees must follow when using registers so they do not interfere with one another.

• These rules define two types of registers: **caller-owned** and **callee-owned**
Caller/callee is terminology that refers to a pair of functions. A single function may be both a caller and callee simultaneously (e.g. function1 at right).

main is the caller, and function1 is the callee.

function1 is the caller, and function2 is the callee.
Register Restrictions

**Caller-Owned**
- Callee must *save* the existing value and *restore* it when done.
- Caller can store values and assume they will be preserved across function calls.

**Callee-Owned**
- Callee does not need to save the existing value.
- Caller’s values could be overwritten by a callee! The caller may consider saving values elsewhere before calling functions.
Caller-Owned Registers

main can use caller-owned registers and know that function1 will not permanently modify their values.

If function1 wants to use any caller-owned registers, it must save the existing values and restore them before returning.
Caller-Owned Registers

main

function1

calls

calls

function1:
  push %rbp
  push %rbx
  ...
  pop %rbx
  pop %rbp
  pop %rbp
retq
Callee-Owned Registers

main can use callee-owned registers but calling function1 may permanently modify their values.

If function1 wants to use any callee-owned registers, it can do so without saving the existing values.
Callee-Owned Registers

main:
...
push %r10
push %r11
callq function1
pop %r11
pop %r10
...

main function1
A Day In the Life of function1

**Caller-owned registers:**
- `function1` must save/restore existing values of any it wants to use.
- `function1` can assume that calling `function2` will not permanently change their values.

**Callee-owned registers:**
- `function1` does not need to save/restore existing values of any it wants to use.
- calling `function2` may permanently change their values.
• There are special registers that store parameters and the return value.
• To call a function, we must put any parameters we are passing into the correct registers. (%rdi, %rsi, %rdx, %rcx, %r8, %r9, in that order)
• Parameters beyond the first 6 are put on the stack.
• If the caller expects a return value, it looks in %rax after the callee completes.
To call a function in assembly, we must do a few things:

- **Pass Control** – `%rip` must be adjusted to execute the function being called and then resume the caller function afterwards.

- **Pass Data** – we must pass any parameters and receive any return value.

- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                     i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    ...
}
```c
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
```

```
0x40054f <+0>:  sub    $0x18,%rsp
0x400553 <+4>:  movl   $0x1,0xc(%rsp)
0x40055b <+12>: movl   $0x2,0x8(%rsp)
0x400563 <+20>: movl   $0x3,0x4(%rsp)
0x40056b <+28>: movl   $0x4,(%rsp)
```
```c
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, 
                      i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, 
          int v1, int v2, int v3, int v4) {
    ...
}
```

Assembly code:
```
0x40054f <+0>:    sub    $0x18,%rsp
0x400553 <+4>:    movl    $0x1,0xc(%rsp)
0x40055b <+12>:   movl    $0x2,0x8(%rsp)
0x400563 <+20>:   movl    $0x3,0x4(%rsp)
0x40056b <+28>:   movl    $0x4,(%rsp)
```

Parameters and Return
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}

0x40054f <+0>: sub $0x18,%rsp
0x400553 <+4>: movl $0x1,0xc(%rsp)
0x40055b <+12>: movl $0x2,0x8(%rsp)
0x400563 <+20>: movl $0x3,0x4(%rsp)
0x40056b <+28>: movl $0x4,(%rsp)
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}

0x40054f <+0>:  sub    $0x18,%rsp
0x400553 <+4>:  movl   $0x1,0xc(%rsp)
0x40055b <+12>: movl   $0x2,0x8(%rsp)
0x400563 <+20>: movl   $0x3,0x4(%rsp)
0x400566 <+23>: add    $0x1,0x4(%rsp)
0x400569 <+26>: movl   $0x4,0xc(%rsp)
```c
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
```

The code snippet represents a C program with the following logic:

- The `main` function initializes four integer variables `i1`, `i2`, `i3`, and `i4` with values 1, 2, 3, and 4 respectively.
- It then calls the `func` function, passing addresses of `i1`, `i2`, `i3`, and `i4`, along with their values.
- The `func` function is not shown in detail, but it processes the arguments.
- The `main` function returns the result of the `func` call.

The assembly code snippet shows the execution of the `main` function at the address `0x400553` and the `func` function at `0x40055b`.

The assembly code includes instructions for loading addresses from the stack (`movl $0x1,0xc(%rsp)`, `movl $0x2,0x8(%rsp)`, `movl $0x3,0x4(%rsp)`), and pushing the result onto the stack (`pushq $0x4`).
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                        i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
          int v1, int v2, int v3, int v4) {
    ...
}

0x40055b <+12>:  movl  $0x2,0x8(%rsp)
0x400563 <+20>:  movl  $0x3,0x4(%rsp)
0x40056b <+28>:  movl  $0x4,(%rsp)
0x400572 <+35>:  pushq  $0x4
0x400574 <+37>:  pushq  $0x3
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
        i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
        int v1, int v2, int v3, int v4) {
    ...
}
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
Parameters and Return

```c
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
          int v1, int v2, int v3, int v4) {
    ...
}
```

```
0x400572 <+35>:  pushq  $0x4
0x400574 <+37>:  pushq  $0x3
0x400576 <+39>:  mov    $0x2,%r9d
0x40057c <+45>:  mov    $0x1,%r8d
0x400582 <+51>:  lea    0x10(%rsp),%rcx
```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}

0x40057c <+45>:  mov  $0x1,%r8d
0x400582 <+51>:  lea  0x10(%rsp),%rcx
0x400587 <+56>:  lea  0x14(%rsp),%rdx
0x40058c <+61>:  lea  0x18(%rsp),%rsi
0x400591 <+66>:  lea  0x1c(%rsp),%rdi
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}

0x400582 <+51>: lea 0x10(%rsp),%rcx
0x400587 <+56>: lea 0x14(%rsp),%rdx
0x40058c <+61>: lea 0x18(%rsp),%rsi
0x400591 <+66>: lea 0x1c(%rsp),%rdi
0x400596 <+71>: callq 0x400546 <func>
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4, i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4, int v1, int v2, int v3, int v4) {
    ...
}
```c
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    ...
}
```

Address locations:
- `%rsi` at 0xffe9e0
- `%rdi` at 0xffe9e8
- `%rax` at 0xffe9f0
- `%rcx` at 0xffe9f4
- `%r8d` at 0xffe9f8
- `%r9d` at 0xffe9fc
- `%rdx` at 0xffe9f0
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
        i1, i2, i3, i4);
    ...
}

int func(int *p1, int *p2, int *p3, int *p4,
        int v1, int v2, int v3, int v4) {
    ...
}
Local Storage

• So far, we’ve often seen local variables stored directly in registers, rather than on the stack.

• There are **three** common reasons that local data must be in memory:
  • We’ve run out of registers
  • The ‘&’ operator is used on it, so we must generate an address for it
  • They are arrays or structs (need to use address arithmetic)
Data Alignment

- Computer systems often put restrictions on the allowable addresses for primitive data types, requiring that the address for some objects must be a multiple of some value $K$ (normally 2, 4, or 8).
- These *alignment restrictions* simplify the design of the hardware.
- For example, suppose that a processor always fetches 8 bytes from the memory system, and an address must be a multiple of 8. If we can guarantee that any `double` will be aligned to have its address as a multiple of 8, then we can read or write the values with a single memory access.
- For x86-64, Intel recommends the following alignments for best performance:

<table>
<thead>
<tr>
<th>$K$</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>char</td>
</tr>
<tr>
<td>2</td>
<td>short</td>
</tr>
<tr>
<td>4</td>
<td>int, float</td>
</tr>
<tr>
<td>8</td>
<td>long, double, char *</td>
</tr>
</tbody>
</table>
The compiler enforces alignment by making sure that every data type is organized in such a way that every field within the struct satisfies the alignment restrictions.

For example, let's look at the following struct:

```c
struct S1 {
    int i;
    char c;
    int j;
};
```

If the compiler used a minimal allocation:

This would make it impossible to align fields `i` (offset 0) and `j` (offset 5). Instead, the compiler inserts a 3-byte gap between fields `c` and `j`:

So, don't be surprised if your structs have a `sizeof()` that is larger than you expect!
GCC Optimizations
Optimization

Most of what **you** need to do with optimization can be summarized by:

1) If doing something seldom and only on small inputs, do whatever is simplest to code, understand, and debug
2) If doing things a lot, or on big inputs, make the primary algorithm’s Big-O cost reasonable
3) **Let gcc do its magic from there**
4) Optimize explicitly as a last resort
Optimizations you’ll see

nop

• **nop/nopl** are “no-op” instructions – they do nothing!
• Intent: Make functions align on address boundaries that are nice multiples of 8.
• “Sometimes, doing nothing is how to be most productive” – Philosopher Nick

```asm
mov %ebx,%ebx
```

• Zeros out the top 32 register bits (because a mov on an e-register zeros out rest of 64 bits).
GCC For Loop Output

GCC Common For Loop Output

Initialization
Test
Jump past loop if success
Body
Update
Jump to test

Possible Alternative

Initialization
Jump to test
Body
Update
Test
Jump to body if success
GCC For Loop Output

GCC Common For Loop Output

- Initialization
- Test
- Jump past loop if success
- Body
- Update
- Jump to test

```
for (int i = 0; i < n; i++)  // n = 100
```
GCC Common For Loop Output

Initialization
Test
Jump past loop if success
Body
Update
Jump to test

 GCC For Loop Output

for (int i = 0; i < n; i++)  // n = 100

Initialization
Test
No jump
Body
Update
Jump to test
Test
No jump
Body
Update
Jump to test
...

// n = 100
GCC For Loop Output

GCC Common For Loop Output

Initialization
Test
Jump past loop if success
Body
Update
Jump to test

for (int i = 0; i < n; i++) // n = 100

Initialization
Test
No jump
Body
Update
Jump to test
Test
No jump
Body
Update
Jump to test
...

// n = 100

for (int i = 0; i < n; i++) // n = 100

Initialization
Jump to test
Test
Jump to body
Body
Update
Test
Jump to body
Body
Update
Test
Jump to body
...

Possible Alternative
Initialization
Jump to test
Body
Update
Test
Jump to body if success
for (int i = 0; i < n; i++) // n = 100

Initialization
Jump to test
Test
Jump to body
Body
Update
Test
Jump to body
Body
Update
Test
Jump to body...

Possible Alternative
Initialization
Jump to test
Body
Update
Test
Jump to body if success
GCC Common For Loop Output

Initialization
Test
Jump past loop if passes
Body
Update
Jump to test

Possible Alternative

Initialization
Jump to test
Body
Update
Test
Jump to body if success

Which instructions are better when n = 0? n = 1000?

for (int i = 0; i < n; i++)
Optimizing Instruction Counts

• Both versions have the same static instruction count (# of written instructions).
• But they have different dynamic instruction counts (# of executed instructions when program is run).
  • If n = 0, left (GCC common output) is best b/c fewer instructions
  • If n is large, right (alternative) is best b/c fewer instructions
• The compiler may emit a static instruction count that is several times longer than an alternative, but it may be more efficient if loop executes many times.
• Does the compiler know that a loop will execute many times? (in general, no)
• So what if our code had loops that always execute a small number of times? How do we know when gcc makes a bad decision?
• (take EE108, EE180, CS316 for more!)
• **Conditional Moves** can sometimes eliminate “branches” (jumps), which are particularly inefficient on modern computer hardware.

• Processors try to *predict* the future execution of instructions for maximum performance. This is difficult to do with jumps.
Today, we’ll be comparing two levels of optimization in the gcc compiler:
- gcc –O0  // mostly just literal translation of C
- gcc –O2  // enable nearly all reasonable optimizations
  (we also use –Og, like –O0 but more debugging friendly)

There are other custom and more aggressive levels of optimization, e.g.:
- -O3     //more aggressive than O2, trade size for speed
- -Os     //optimize for size
- -Ofast  //disregard standards compliance (!!)

Exhaustive list of gcc optimization-related flags:
Compiler optimizations

How many GCC optimization levels are there?

Gcc supports numbers up to 3. Anything above is interpreted as 3

Gcc supports numbers up to 3. Anything above is interpreted as 3

How many GCC optimization levels are there?

109

I tried gcc -O1, gcc -O2, gcc -O3, and gcc -O4

If I use a really large number, it won't work.

However, I have tried

35

gcc -0100

and it compiled.

How many optimization levels are there?

https://stackoverflow.com/questions/1778538/how-many-gcc-optimization-levels-are-there

72
GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling
**Constant Folding** pre-calculates constants at compile-time where possible.

```c
int seconds = 60 * 60 * 24 * n_days;
```
int fold(int param) {
    char arr[5];
    int a = 0x107;
    int b = a * sizeof(arr);
    int c = sqrt(2.0);
    return a * param + (a + 0x15 / c + strlen("Hello") * b - 0x37) / 4;
}
Constant Folding: Before (-O0)

```
11b9:  55                      push   %rbp
11ba:  48 89 e5                mov    %rsp,%rbp
11bd:  41 54                  push   %r12
11bf:  53                      push   %rbx
11c0:  48 83 ec 30              sub    $0x30,%rsp
11c4:  89 7d cc               mov    %edi,-0x34(%rbp)
11c7:  c7 45 ec 07 01 00 00     movl   $0x107,-0x14(%rbp)
11ce:  8b 45 ec                mov    -0x14(%rbp),%eax
11d1:  48 98                   cltq
11d3:  89 c2                   mov    %eax,%edx
11d5:  89 d0                   mov    %edx,%eax
11d7:  c1 e0 02                shl    $0x2,%eax
11da:  89 45 e8                mov    %eax,-0x18(%rbp)
11df:  48 8b 05 2a 0e 00 00    mov    0xe2a(%rip),%rax        # 2010 <_IO_stdin_used+0x10>
11e6:  66 48 0f 6e c0           movq   %rax,%%xmm0
11eb:  e8 b0 fe ff ff          callq  10a0 <sqrt@plt>
11f0:  f2 0f 2c c0             cvttsd2si %xmm0,%eax
11f4:  8b 45 ec                mov    -0x14(%rbp),%eax
11f7:  0f af 45 cc             imul   -0x34(%rbp),%eax
11fe:  41 89 c4                mov    %eax,%r12d
1201:  b8 15 00 00 00          mov    $0x15,%eax
1206:  99                      cltd
1207:  f7 7d e4             idivl  -0x1c(%rbp)
120a:  8b 45 ec                mov    -0x14(%rbp),%eax
120c:  01 d0                   add    %edx,%eax
120f:  48 63 d8                movslq %eax,%rbx
1211:  48 8d 3d ed 0d 00 00 00 lea    0xed(%rip),%rdi        # 2008 <_IO_stdin_used+0x8>
1214:  e8 20 fe ff ff          callq  1840 <strlen@plt>
121b:  8b 55 e8                mov    -0x18(%rbp),%edx
1223:  48 63 d2                movslq %edx,%rdx
1226:  48 0f af c2             imul   %rdx,%rax
1229:  8b 01 d8                add    %rbx,%rax
122d:  48 83 e8 37              sub    $0x37,%rax
1231:  48 c1 e8 02              shr    $0x2,%rax
1235:  44 01 e0                add    %r12d,%eax
1238:  48 83 c4 30              add    $0x30,%rsp
123c:  5b                      pop    %rbx
123d:  41 5c                 pop    %r12
123f:  5d                     pop    %rbp
1240:  c3                      retq
```
What is the consequence of this for you as a programmer? What should you do differently or the same knowing that compilers can do this for you?
GCC Optimizations

• Constant Folding
• **Common Sub-expression Elimination**
• Dead Code
• Strength Reduction
• Code Motion
• Tail Recursion
• Loop Unrolling
Common Sub-Expression Elimination prevents the recalculation of the same thing many times by doing it once and saving the result.

```c
int a = (param2 + 0x107);
int b = param1 * (param2 + 0x107) + a;
return a * (param2 + 0x107) + b * (param2 + 0x107);
```
Common Sub-Expression Elimination prevents the recalculation of the same thing many times by doing it once and saving the result.

```c
int a = (param2 + 0x107);
int b = param1 * (param2 + 0x107) + a;
return a * (param2 + 0x107) + b * (param2 + 0x107);
// = 2 * a * a + param1 * a * a
```

000000000000011b0 <subexp>:  // param1 in %edi, param2 in %esi
```
11b0: lea  0x107(%rsi),%eax       // %eax stores a
11b6: imul  %eax,%edi             // param1 * a
11b9: lea  (%rdi,%rax,2),%esi     // 2 * a + param1 * a
11bc: imul  %esi,%eax             // a * (2 * a + param1 * a)
11bf: retq
```
Why should we bother saving repeated calculations in variables if the compiler has common subexpression elimination?

- The compiler may not always be able to optimize every instance. Plus, it can help reduce redundancy!
- Makes code more readable!
GCC Optimizations

• Constant Folding
• Common Sub-expression Elimination
• **Dead Code**
• Strength Reduction
• Code Motion
• Tail Recursion
• Loop Unrolling
Dead code elimination removes code that doesn’t serve a purpose:

```c
if (param1 < param2 && param1 > param2) {
    printf("This test can never be true!\n");
}

// Empty for loop
for (int i = 0; i < 1000; i++);

// If/else that does the same operation in both cases
if (param1 == param2) {
    param1++;
} else {
    param1++;
}

// If/else that more trickily does the same operation in both cases
if (param1 == 0) {
    return 0;
} else {
    return param1;
}
```
Dead Code: Before (-O0)

```
00000000000011a9 <dead_code>:
  11a9:  55                   push   %rbp
  11aa:  48 89 e5             mov    %rsp,%rbp
  11ad:  48 83 ec 20          sub    $0x20,%rsp
  11b1:  89 7d ec             mov    %edi,-0x14(%rbp)
  11b4:  89 75 e8             mov    %esi,-0x18(%rbp)
  11b7:  8b 45 e8             mov    -0x14(%rbp),%eax
  11ba:  3b 45 e8             cmp    -0x18(%rbp),%eax
  11bd:  7d 19                jge    11d8 <dead_code+0x2f>
  11bf:  8b 45 e8             mov    -0x14(%rbp),%eax
  11c2:  3b 45 e8             cmp    -0x18(%rbp),%eax
  11c5:  7e 11                jle    11d8 <dead_code+0x2f>
  11c7:  48 8d 3d 36 0e 00 00 lea    0xe36(%rip),%rdi        # 2004 <_IO_stdin_used+0x4>
  11ce:  b8 00 00 00 00       mov    $0x0,%eax
  11df:  b8 00 00 00 00       mov    $0x0,%eax
  11e1:  83 45 fc 01          addl   $0x1,-0x4(%rbp)
  11e5:  83 45 fc 01          addl   $0x1,-0x4(%rbp)
  11ec:  7e f3                jmp    1200 <dead_code+0x57>
  11ee:  8b 45 ec             mov    -0x18(%rbp),%eax
  11f1:  3b 45 e8             cmp    -0x18(%rbp),%eax
  11f4:  75 06                jne    120d <dead_code+0x64>
  11f6:  83 45 ec 01          addl   $0x1,-0x14(%rbp)
  11fa:  83 45 ec 01          addl   $0x1,-0x14(%rbp)
  11fc:  83 7d ec 00          cmpl   $0x0,-0x14(%rbp)
  1200:  75 07                jne    1210 <dead_code+0x67>
  1204:  b8 00 00 00 00       mov    $0x0,%eax
  1206:  b8 00 00 00 00       mov    $0x0,%eax
  120b:  eb 03                jmp    1210 <dead_code+0x67>
  120d:  c9                   leaveq
  1210:  c3                   retq
```
Dead Code: After (-O2)

00000000000011b0 <dead_code>:
  11b0:   8d 47 01          lea   0x1(%rdi),%eax
  11b3:   c3              retq
GCC Optimizations

• Constant Folding
• Common Sub-expression Elimination
• Dead Code
• **Strength Reduction**
• Code Motion
• Tail Recursion
• Loop Unrolling
Strength Reduction

**Strength reduction** changes divide to multiply, multiply to add/shift, and mod to AND to avoid using instructions that cost many cycles (multiply and divide).

```c
int a = param2 * 32;
int b = a * 7;
int c = b / 2;
int d = param2 % 2;

for (int i = 0; i <= param2; i++) {
    c += param1[i] + 0x107 * i;
}

return c + d;
```
• int a = param2 * 32;
  Becomes:
  • int a = param2 * 32;

• int b = a * 7;
  Becomes:
  • int b = a + (a << 2) + (a << 1);

• int c = b / 2;
  Becomes
  • int c = b >> 1
GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- **Code Motion**
- Tail Recursion
- Loop Unrolling
Code motion moves code outside of a loop if possible.

```java
for (int i = 0; i < n; i++) {
    sum += arr[i] + foo * (bar + 3);
}
```

Common subexpression elimination deals with expressions that appear multiple times in the code. Here, the expression appears once, but is calculated each loop iteration, even though none of its values change during the loop.
Code motion moves code outside of a loop if possible.

```c
int temp = foo * (bar + 3);
for (int i = 0; i < n; i++) {
    sum += arr[i] + temp;
}
```

Moving it out of the loop allows the computation to happen only once.
Practice: GCC Optimization

```c
int char_sum(char *s) {
    int sum = 0;
    for (size_t i = 0; i < strlen(s); i++) {
        sum += s[i];
    }
    return sum;
}
```

What is the bottleneck? What (if anything) can GCC do?
int char_sum(char *s) {
    int sum = 0;
    for (size_t i = 0; i < strlen(s); i++) {
        sum += s[i];
    }
    return sum;
}

What is the bottleneck? What (if anything) can GCC do?

\textit{strlen} is called every loop iteration – \texttt{code motion} can pull it out of the loop
Tail recursion is an example of where GCC can identify recursive patterns that can be more efficiently implemented iteratively.

```c
long factorial(int n) {
    if (n <= 1) {
        return 1;
    }
    else return n * factorial(n - 1);
}
```
Recall the factorial problem from assembly lectures:

```c
unsigned int factorial(unsigned int n) {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

What happens with `factorial(-1)`?

• Infinite recursion → Literal stack overflow!
• Compiled with `-Og`!
Factorial: -0g vs –02

-02:
• What happened?
• Did the compiler “fix” the infinite recursion?

-02:

```
401146 <+0>: cmp $0x1,%edi
401149 <+3>: jbe 0x40115b <factorial+21>
40114b <+5>: push %rbx
40114c <+6>: mov %edi,%ebx
40114e <+8>: lea -0x1(%rdi),%edi
401151 <+11>: callq 0x401146 <factorial>
401156 <+16>: imul %ebx,%eax
401159 <+19>: pop %rbx
40115a <+20>: retq
40115b <+21>: mov $0x1,%eax
401160 <+26>: retq
```

```
4011e0 <+0>: mov $0x1,%eax
4011e5 <+5>: cmp $0x1,%edi
4011e8 <+8>: jbe 0x4011fd <factorial+29>
4011ea <+10>: nopw 0x0(%rax,%rax,1)
4011f0 <+16>: mov %edi,%edx
4011f2 <+18>: sub $0x1,%edi
4011f5 <+21>: imul %edx,%eax
4011f8 <+24>: cmp $0x1,%edi
4011fb <+27>: jne 0x4011f0 <factorial+16>
4011fd <+29>: retq
```
4011e0 <+0>:  mov    $0x1,%eax        # Initialize %eax with 1.
4011e5 <+5>:  cmp    $0x1,%edi        # Compare input value (%edi) with 1.
4011e8 <+8>:  jbe    0x4011fd <factorial+29> # If input <= 1 (unsigned check), jump to return.
4011ea <+10>: nopw   0x0(%rax,%rax,1) # No operation (probably for alignment).
4011f0 <+16>: mov    %edi,%edx        # Copy current value of %edi to %edx.
4011f2 <+18>: sub    $0x1,%edi        # Decrement %edi.
4011f5 <+21>: imul   %edx,%eax        # Multiply %eax by %edx and store result in %eax.
4011f8 <+24>: cmp    $0x1,%edi        # Compare decremented value of %edi with 1.
4011fb <+27>: jne    0x4011f0 <factorial+16> # If %edi is not 1, repeat the multiplication.
4011fd <+29>: retq                    # Return with the result in %eax.

-02:
• Recursive -> Iterative
• No Stack Overflow, Saves Memory and Operations
 GCC Optimizations

• Constant Folding
• Common Sub-expression Elimination
• Dead Code
• Strength Reduction
• Code Motion
• Tail Recursion
• **Loop Unrolling**
Loop Unrolling: Do \( n \) loop iterations’ worth of work per actual loop iteration, so we save ourselves from doing the loop overhead (test and jump) every time, and instead incur overhead only every \( n \)-th time.

```c
for (int i = 0; i <= n - 4; i += 4) {
    sum += arr[i];
    sum += arr[i + 1];
    sum += arr[i + 2];
    sum += arr[i + 3];
} // after the loop handle any leftovers
```
Some Extra Reading
Key GDB Tips For Assembly

• Examine 4 giant words (8 bytes) on the stack:
  (gdb) x/4g $rsp
  0x7fffffff870: 0x0000000000000005 0x0000000000400559
  0x7fffffff880: 0x0000000000000000 0x0000000000400575

• display/undisplay (prints out things every time you step/next)
  (gdb) display/4w $rsp
  1: x/4xw $rsp
  0x7fffffff8a8:
  0xf7a2d830 0x00007fff 0x00000000 0x00000000
Key GDB Tips For Assembly

- **stepi/finish**: step into current function call/return to caller:
  (gdb) finish

- Set register values during the run
  (gdb) p $rdi = $rdi + 1

(Might be useful to write down the original value of $rdi somewhere)

- Tui things
  - refresh
  - focus cmd – use up/down arrows on gdb command line (vs focus asm, focus regs)
  - layout regs, layout asm
gdb tips

layout split (ctrl-x a: exit, ctrl-l: resize)

info reg

p $eax

p $eflags

b *0x400546

b *0x400550 if $eax > 98

ni

si

View C, assembly, and gdb (lab5)
Print all registers
Print register value
Print all condition codes currently set
Set breakpoint at assembly instruction
Set **conditional breakpoint**
Next assembly instruction
Step into assembly instruction (will step into function calls)
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p/x $rdi</td>
<td>Print register value in hex</td>
</tr>
<tr>
<td>p/t $rsi</td>
<td>Print register value in binary</td>
</tr>
<tr>
<td>x $rdi</td>
<td>Examine the byte stored at this address</td>
</tr>
<tr>
<td>x/4bx $rdi</td>
<td>Examine 4 bytes starting at this address</td>
</tr>
<tr>
<td>x/4wx $rdi</td>
<td>Examine 4 ints starting at this address</td>
</tr>
</tbody>
</table>
• Arrays in C map in a fairly straightforward way to X86 assembly code, thanks to the addressing modes available in instructions.
• When we perform pointer arithmetic, the assembly code that is produced will have address computations built into them.
• Optimizing compilers are very good at simplifying the address computations (in lab you will see another optimizing compiler benefit in the form of division — if the compiler can avoid dividing, it will!). Because of the transformations, compiler-generated assembly for arrays often doesn't look like what you are expecting.
• Consider the following form of a data type $T$ and integer constant $N$:

\[
T \ A[N]
\]
• The starting location is designated as $x_A$
• The declaration allocates $N \times \text{sizeof}(T)$ bytes, and gives us an identifier that we can use as a pointer (but it isn't a pointer!), with a value of $x_A$. 
Array Allocation and Access

- **Example:**

<table>
<thead>
<tr>
<th>Array</th>
<th>Element Size</th>
<th>Total Size</th>
<th>Start address</th>
<th>Element i</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>A[12]</td>
<td>1</td>
<td>12</td>
<td>xA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xA + i</td>
</tr>
<tr>
<td>char</td>
<td>*B[8]</td>
<td>8</td>
<td>64</td>
<td>xB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xB + 8i</td>
</tr>
<tr>
<td>int</td>
<td>C[6]</td>
<td>4</td>
<td>24</td>
<td>xC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xC + 4i</td>
</tr>
<tr>
<td>double</td>
<td>*D[5]</td>
<td>8</td>
<td>40</td>
<td>xD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xD + 8i</td>
</tr>
</tbody>
</table>

- The memory referencing operations in x86-64 are designed to simplify array access. Suppose we wanted to access C[3] above. If the address of C is in register `%rdx`, and 3 is in register `%rcx`

- The following copies C[3] into `%eax`,

```
movl (%rdx,%rcx,4), %eax
```
• C allows arithmetic on pointers, where the computed value is calculated according to the size of the data type referenced by the pointer.
• The array reference A[i] is identical to *(A+i)
• Example: if the address of array E is in %rdx, and the integer index, i, is in %rcx, the following are some expressions involving E:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>int *</td>
<td>x_E</td>
<td>movq %rdx, %rax</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E[0]</td>
<td>int</td>
<td>M[x_E]</td>
<td>movl (%rdx), %eax</td>
</tr>
<tr>
<td>E[i]</td>
<td>int</td>
<td>M[x_E+4i]</td>
<td>movl (%rdx,%rcx,4) %eax</td>
</tr>
<tr>
<td>&amp;E[2]</td>
<td>int *</td>
<td>x_E+8</td>
<td>leaq 8(%rdx), %rax</td>
</tr>
<tr>
<td>E+i-1</td>
<td>int *</td>
<td>x_E+4i-4</td>
<td>leaq -4(%rdx,%rcx,4), %rax</td>
</tr>
<tr>
<td>*(E+i-3)</td>
<td>int</td>
<td>M[x_E+4i-12]</td>
<td>movl -12(%rdx,%rcx,4) %eax</td>
</tr>
<tr>
<td>&amp;E[i]-E</td>
<td>long</td>
<td>i</td>
<td>movq %rcx,%rax</td>
</tr>
</tbody>
</table>
• Practice: \( x_\text{S} \) is the address of a short integer array, \( S \), stored in \%rdx, and a long integer index, \( i \), is stored in register \%rcx.

• For each of the following expressions, give its type, a formula for its value, and an assembly-code implementation. The result should be stored in \%rax if it is a pointer, and the result should be in register \%ax if it has a data type short.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S + 1 )</td>
<td>short *</td>
<td>( x_\text{S} + 2 )</td>
<td>leaq 2(%rdx),%rax</td>
</tr>
<tr>
<td>( S[3] )</td>
<td>short</td>
<td>( M[x_\text{S} + 6] )</td>
<td>movw 6(%rdx),%ax</td>
</tr>
<tr>
<td>&amp;( S[i] )</td>
<td>short *</td>
<td>( x_\text{S} + 2i )</td>
<td>leaq (%rdx,%rcx,2),%rax</td>
</tr>
<tr>
<td>( S[4i+1] )</td>
<td>short</td>
<td>( M[x_\text{S} + 8i + 2] )</td>
<td>movw 2(%rdx,%rcx,8),%ax</td>
</tr>
<tr>
<td>( S+i-5 )</td>
<td>short *</td>
<td>( x_\text{S} + 2i - 10 )</td>
<td>leaq -10(%rdx,%rcx,2),%rax</td>
</tr>
</tbody>
</table>
References and Advanced

• References:
  - Stanford guide to x86-64: https://web.stanford.edu/class/cs107/guide/x86-64.html
  - CS107 one-page of x86-64: https://web.stanford.edu/class/cs107/resources/onepage_x86-64.pdf
  - gdbtui: https://beej.us/guide/bggdb/
  - More gdbtui: https://sourceware.org/gdb/onlinedocs/gdb/TUI.html
  - Compiler explorer: https://gcc.godbolt.org

• Advanced Reading:
  - Stack frame layout on x86-64: https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64
  - history of x86 instructions: https://en.wikipedia.org/wiki/X86_instruction_listings
  - x86-64 Wikipedia: https://en.wikipedia.org/wiki/X86-64