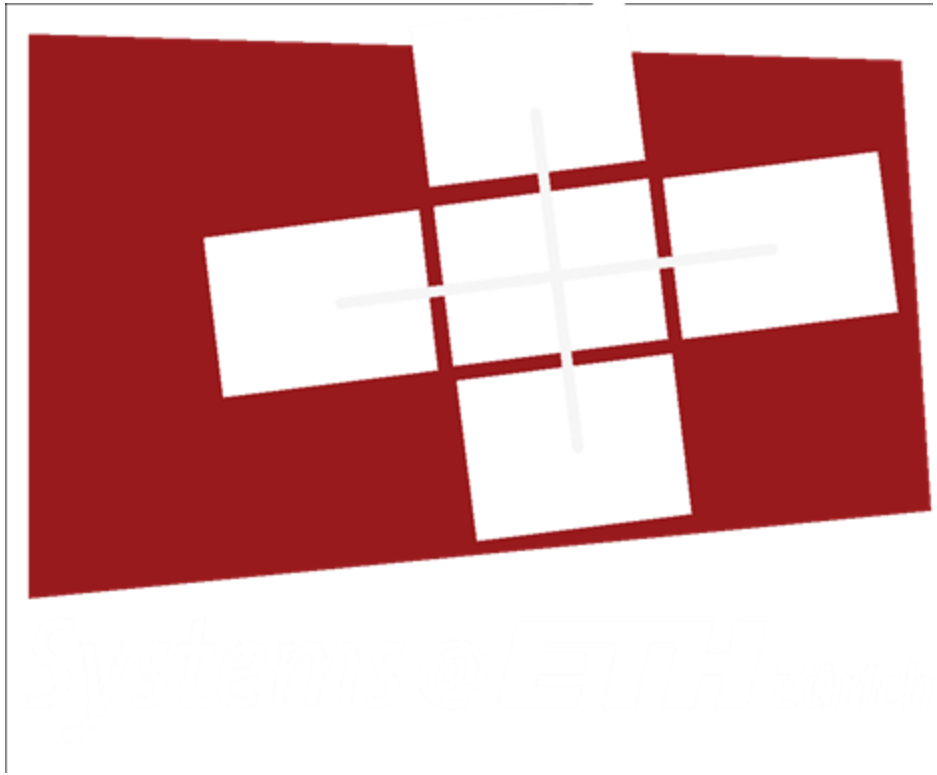


Exercise Session 6

2024 Autumn

Disclaimer

- **Website**: n.ethz.ch/~falkbe/
- **(Extra) Demos on GitHub**: github.com/falkbe
- My exercise slides have **additional slides** (which are not official part of the course) **having a blue heading**: they are there to complement and go into more depth where I found appropriate
- For the exam **only** the official exercise slides are relevant, if in doubt always check the ones on the official moodle page

Remark Todays Exercise Session

- Q: Everyone okay with 5-10min break and end at 15:50-15:55 for today?
- Q: From now on, Kahoots or rather still more theory?

Remark Exercises

- Pen and Paper **Exercise Sheet 4** will be published: highly recommend doing this, helps in understanding memory layout, assembly etc.
- **Code Expert**: highly recommend doing this too (if you understood assembly it can be done quite fast): **and very relevant for exam** (likely that there will be assembly programming tasks like this)

Agenda

- Assignment 4
- Lecture Recap
- Stack Calling Conventions
- Consolidation Q&A
- Exam Quiz
  - Pointers / Arrays
  - Assembly
  - C Declarations: Clockwise/Spiral Rule
- Kahoot
- More Q&A

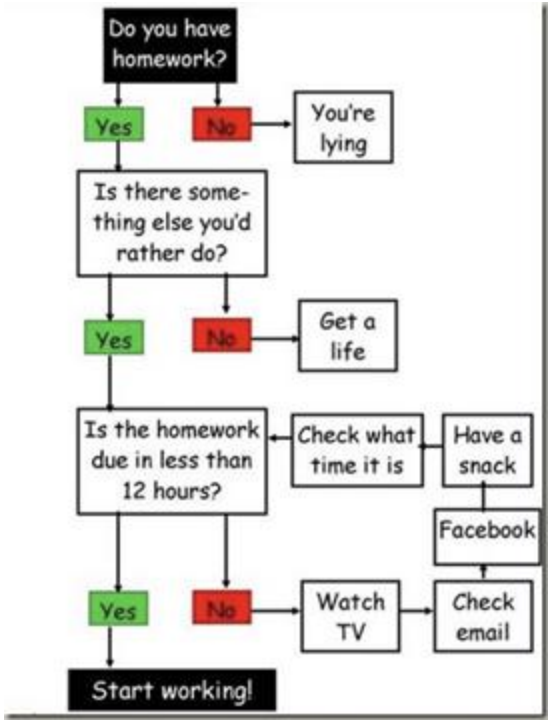
Assignment 4

Submit pen and paper solutions

via email to your teaching assistant

Rest of the assignment on Code Expert

Details on Handout



Lecture Recap

Compiling C Control Flow: Loops, Switch

Lecture Recap: CCF: if statement

Compiling `if (…) {…} else {…}`

```c
#include <stdio.h>

int putmax(int x, int y)
{
    int result;
    if (x <= y) {
        goto Else;
    }
    result = printf("%d\n", x);
    goto Done;
Else:
    result = printf("%d\n", y);
Done:
    return result;
}
```

```
putmax:
    subq    $8, %rsp
    cmpl    %esi, %edi
    jle     .L2
    movl    %edi, %edx
    movl    $.LC0, %esi
    movl    $1, %edi
    movl    $0, %eax
    call    __printf_chk
    jmp     .L3
.L2:
    movl    %esi, %edx
    movl    $.LC0, %esi
    movl    $1, %edi
    movl    $0, %eax
    call    __printf_chk
.L3:
    addq    $8, %rsp
    ret
```

Setup
Test
Body 1
Body 2
Finish

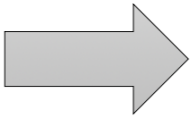
Lecture Recap: CCF: for loop

Systems@ETH Zürich

Compiling `for(…) {…}`

C code

```
for (<init>; <test>; <update>) {
    <body>
}
```



Assembly

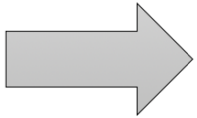
```
    <init>;
    goto middle;
loop:
    <body>;
    <update>;
middle:
    if (<test>)
        goto loop;
done:
```

Lecture Recap: CCF: while loop

Compiling `while(…) {…}`

C code

```
while (<test>) {
    <body>
}
```



Goto version

```
goto middle;
loop:
  <body>
middle:
  if (<test>)
    goto loop;
```

Lecture Recap: CCF: Switch table

Switch statement example

Lots going on here…

- Multiple case labels
  - Here: 5, 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

Lecture Recap: CCF: Switch table

Jump table structure

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table

jtab:

Targ0
Targ1
Targ2
• • •
Targn-1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

• • •

Targn-1: Code Block n–1

Approximate Translation

```
target = JTab[x];
goto *target;
```

Lecture Recap: CCF: Switch table

Switch statement

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Jump table:

```
   .section .rodata
   .align 8
   .align 4
.L4:
   .quad  .L8  # x=0
   .quad  .L3  # x=1
   .quad  .L5  # x=2
   .quad  .L9  # x=3
   .quad  .L8  # x=4
   .quad  .L7  # x=5
   .quad  .L7  # x=6
```

Setup:

```
switch_eg:
   movq  %rdx, %rcx
   cmpq  $6, %rdi       # x : 6?
   ja    .L8            # if > goto default
   jmp   *.L4(,%rdi,8) # goto Jtab[x]
```

Indirect jump

Lecture Recap: CCF: Switch table

Assembly setup explanation

- Table Structure
  - Each target requires 8 bytes
  - Base address at `.L4`

- Jumping

  Direct: `jmp .L8`
  - Jump target is denoted by label `.L8`

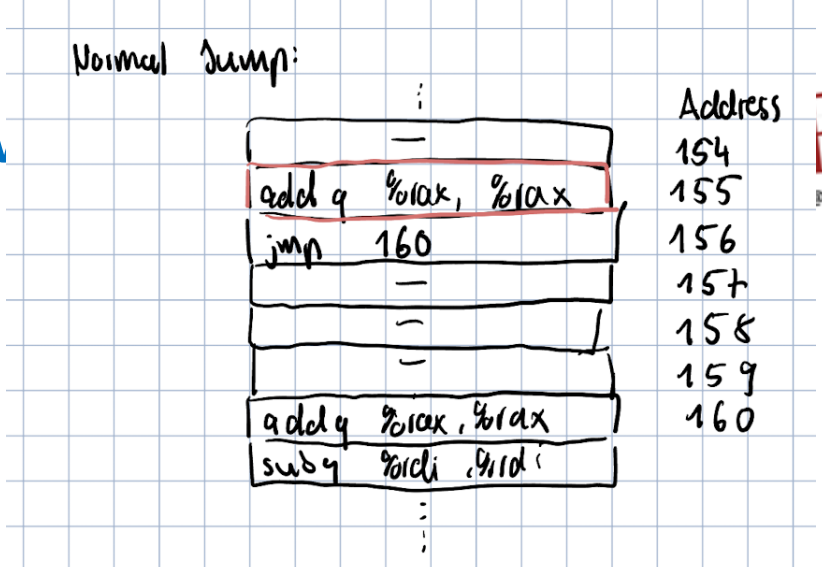
  Indirect: `jmp *.L4(,%rdi,8)`
  - Must scale by factor of 8 (labels are 64-bit = 8 Bytes on x86_64)
  - Fetch target from effective Address `.L4 + rdi*8`
    - Only for $0 \leq x \leq 6$

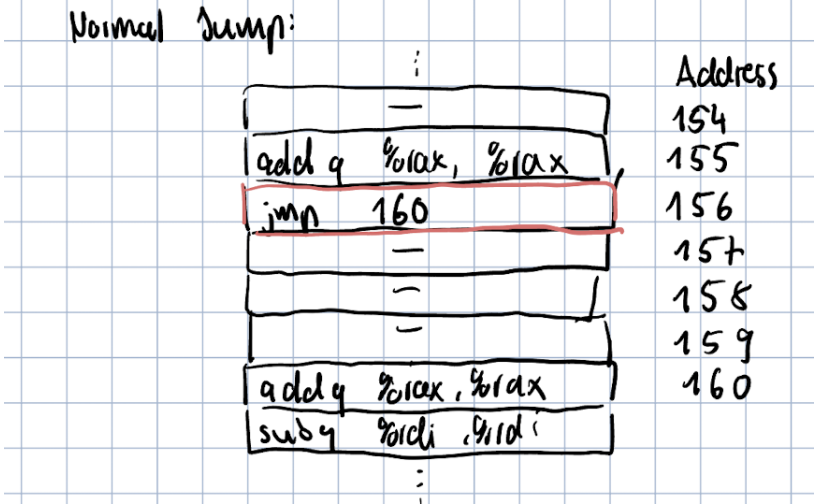
Jump table

```
  .section .rodata
  .align 8
  .align 4
.L4:
  .quad  .L8  # x=0
  .quad  .L3  # x=1
  .quad  .L5  # x=2
  .quad  .L9  # x=3
  .quad  .L8  # x=4
  .quad  .L7  # x=5
  .quad  .L7  # x=6
```

Lecture Recap: CCF: Sw

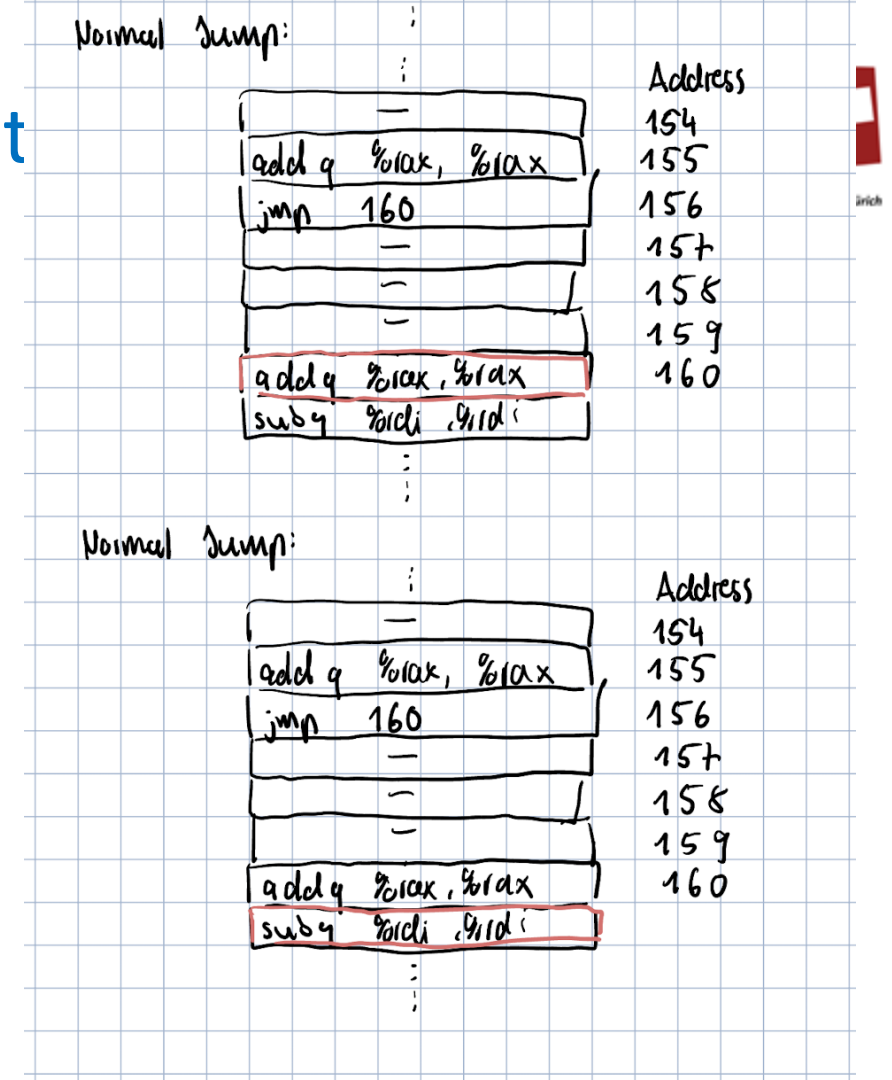
- Direct Jump: we **directly jump at the specified address**





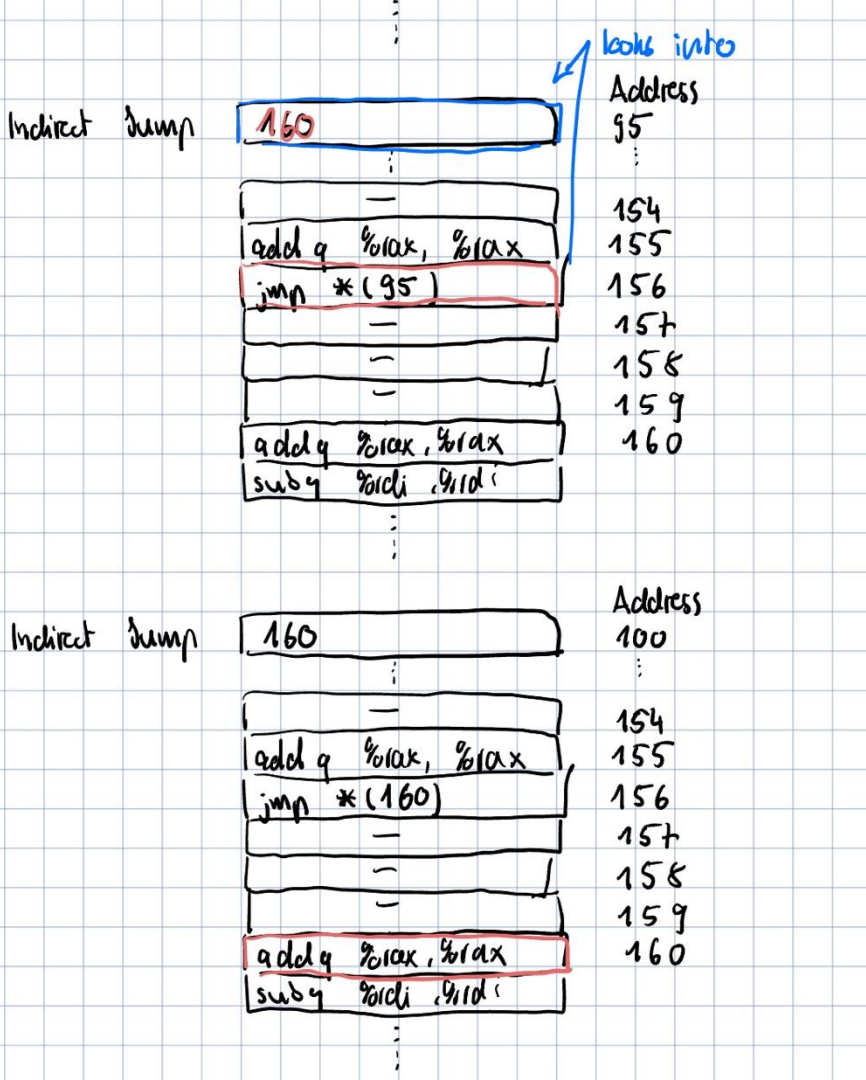
Lecture Recap: CCF: Swit

- Direct Jump: we **directly jump at the specified address**



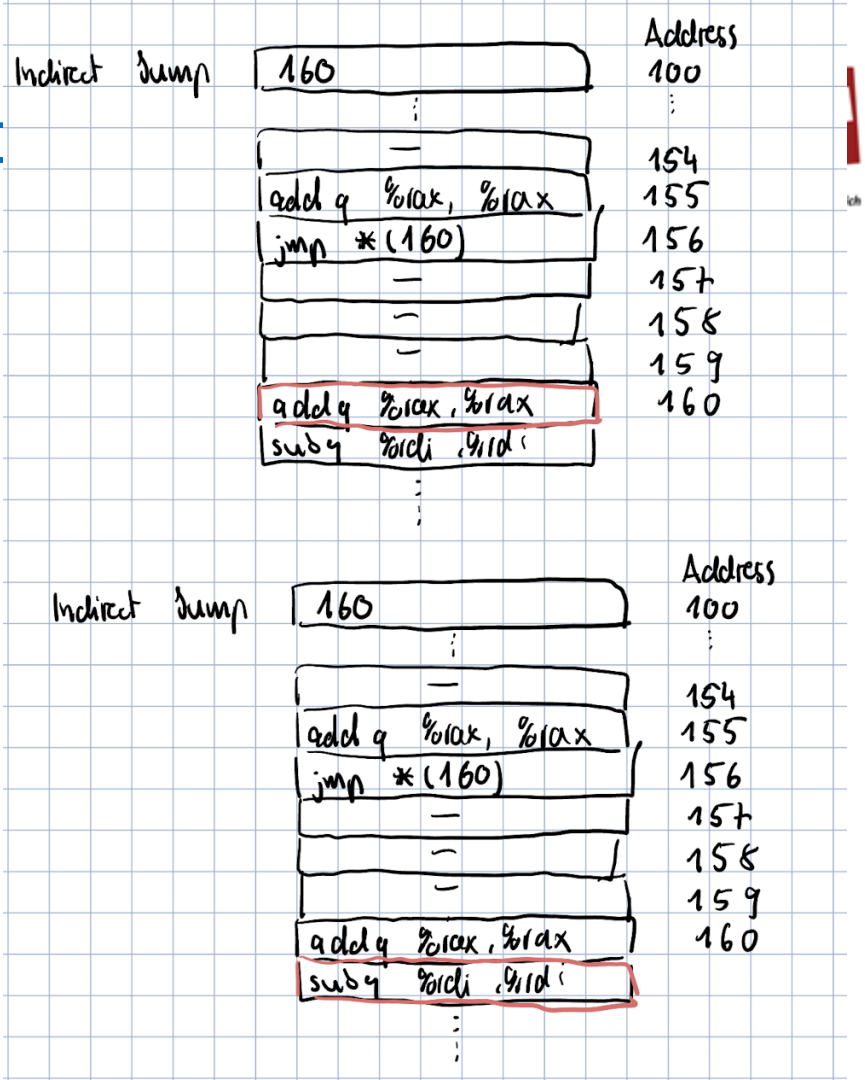
Lecture Recap: CCF: Switch

- **In**Direct Jump: we **look** at the value **stored** at the specified address, and jump to **what was stored there** (i.e. interpret that as an address)



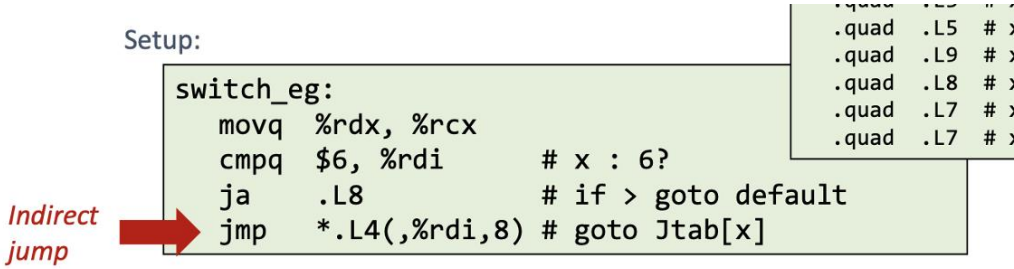
Lecture Recap: CCF: Swit

- **In**Direct Jump: we **look** at the value **stored** at the specified address, and jump to **what was stored there** (i.e. interpret that as an address)



Lecture Recap: CCF: Switch table

- **Note**: a label like ".L4" or ".main" are just addresses (so think of .L4 as something like 0x4000243)



Lecture Recap: CC

- Then here the same thing happens: look into the jump table at the specified value of x (%rdi*8) as x is a long, the JUMP to what the **jump table says**

Setup:

```
switch_eg:
   movq   %rdx, %rcx
   cmpq   $6, %rdi      # x : 6?
   ja     .L8           # if > goto default
   jmp    *.L4(,%rdi,8) # goto Jtab[x]
```

```
.quad  .L5  # x
.quad  .L9  # x
.quad  .L8  # x
.quad  .L7  # x
.quad  .L7  # x
```

Indirect jump

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table

jtab:

Targ0
Targ1
Targ2
• • •
Targn-1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

• • •

Targn-1: Code Block n–1

Approximate Translation

```
target = JTab[x];
goto *target;
```

Lecture Recap: CCF: Switch table: we don't care how big the code blocks in each switch statement are as we don't execute the code in there but jump to the targets

Jump table structure

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table

jtab:

Targ0
Targ1
Targ2
• • •
Targn-1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

• • •

Targn-1: Code Block n–1

Approximate Translation

```
target = JTab[x];
goto *target;
```

Lecture Recap: CCF: Switch table: multiple can jump to the same location (x=0, x=4)

Switch statement

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Jump table:

```
   .section .rodata
   .align 8
   .align 4
.L4:
   .quad  .L8  # x=0
   .quad  .L3  # x=1
   .quad  .L5  # x=2
   .quad  .L9  # x=3
   .quad  .L8  # x=4
   .quad  .L7  # x=5
   .quad  .L7  # x=6
```

Setup:

```
switch_eg:
   movq  %rdx, %rcx
   cmpq  $6, %rdi       # x : 6?
   ja    .L8            # if > goto default
   jmp   *.L4(,%rdi,8) # goto Jtab[x]
```

direct jump

Lecture Recap

Procedure call and return, Calling Conventions

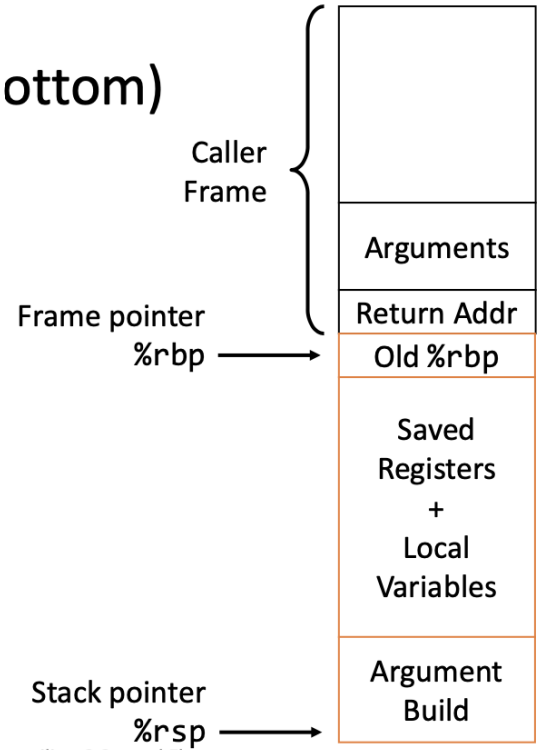
Lecture Recap: Procedure Call

- Let us look at the procedure call and return in three steps
- 1. The call (setup)
- 2. Within the call (using the stack frame)
- 3. The return (cleanup)

What you saw in the lecture:

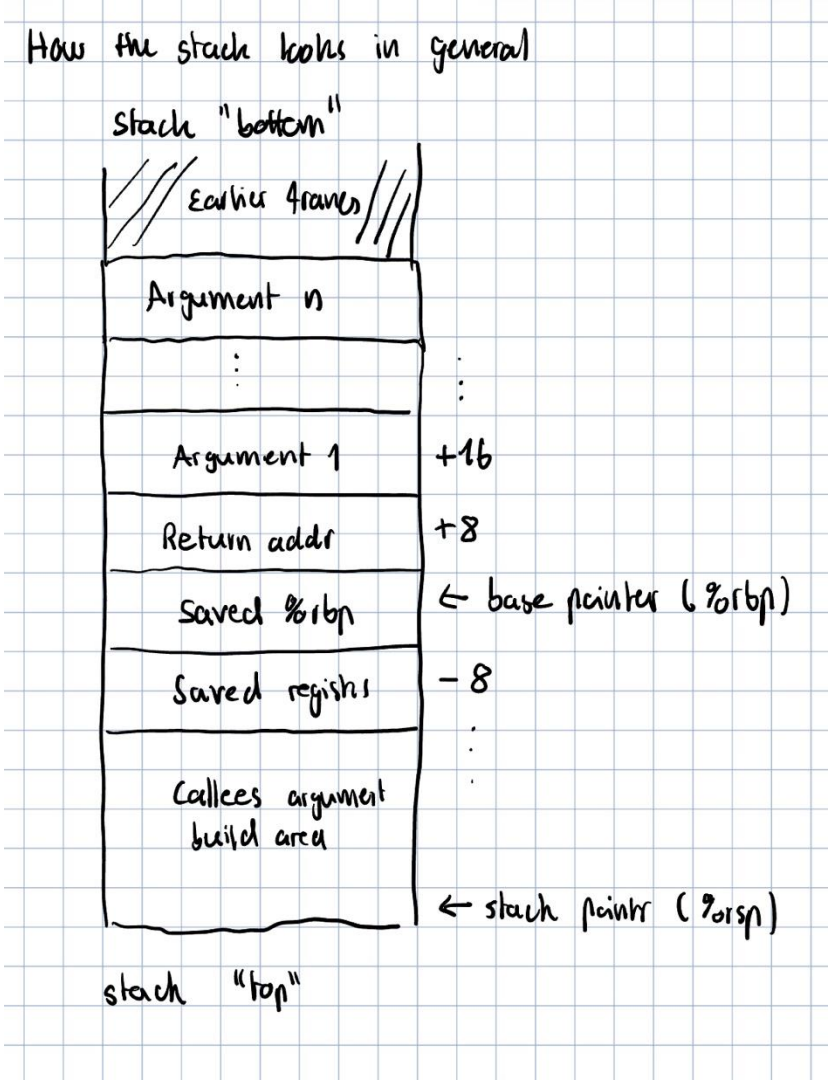
full x86_64/Linux stack frame

- Current stack frame ("top" to bottom)
  - "Argument build:"
    Parameters for function about to call
  - Local variables
    If can't keep in registers
  - Saved register context
  - Old frame pointer
- Caller stack frame
  - Return address
  - Pushed by `call` instruction
  - Arguments for this call



Stack here:

- This is what the full stack frame looks like: keep this in mind while we step through it
- In the end you should understand what the purpose of all the things on it are!



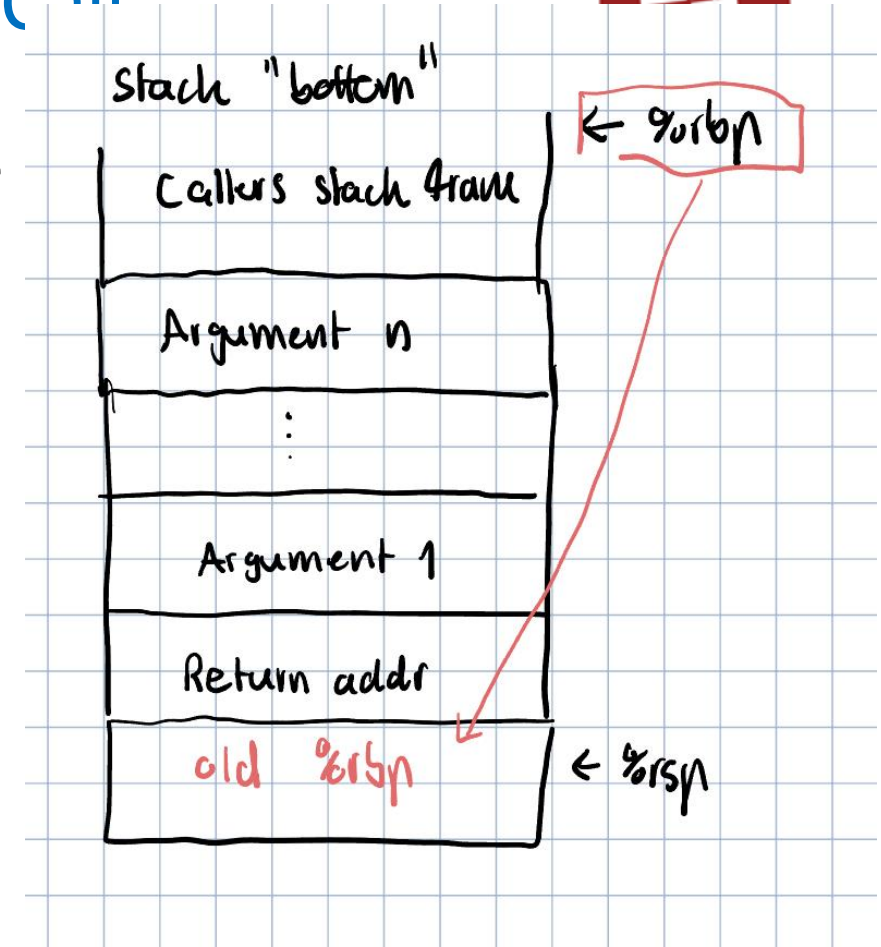
Lecture Recap: Procedure Call

Systems@ETH Zürich

1. The call (setup)

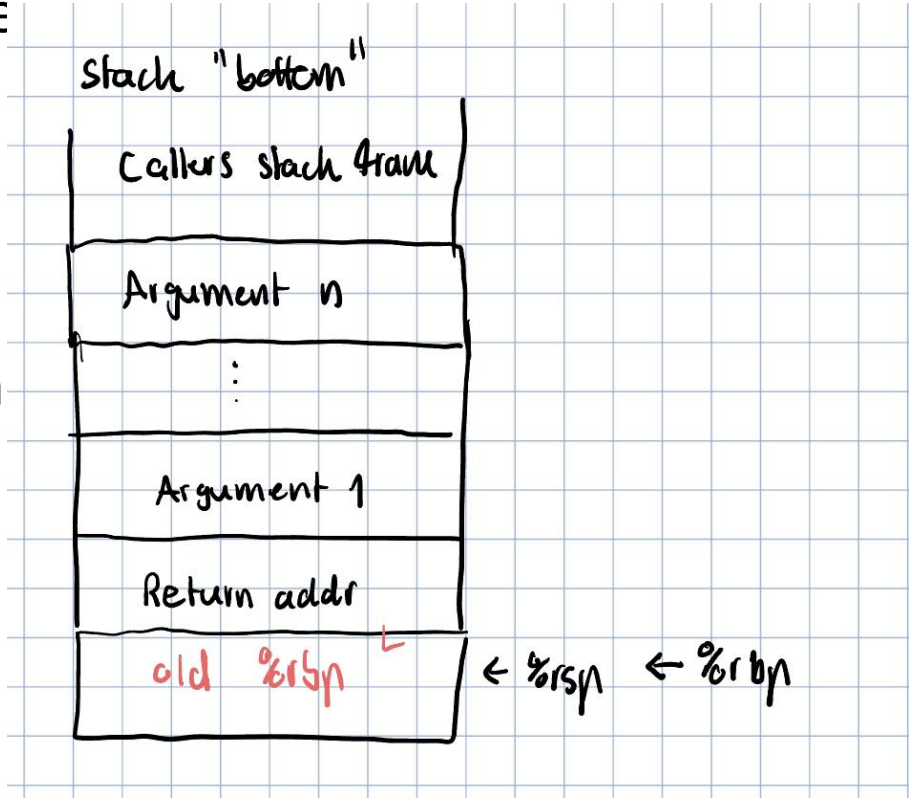
Lecture Recap: Procedure Call

- Callee now saves the callers base pointer (%rbp) by pushing it onto the stack
- “pushq %rbp”



Lecture Recap: Procedure Call

- Callee now saves the callers base pointer (%rbp) by pushing it onto the stack
- “pushq %rbp”
- Then sets up its own “rbp” to point at the beginning to its own stack frame which is **right here** where rsp currently points to
- “movq %rsp, %rbp”



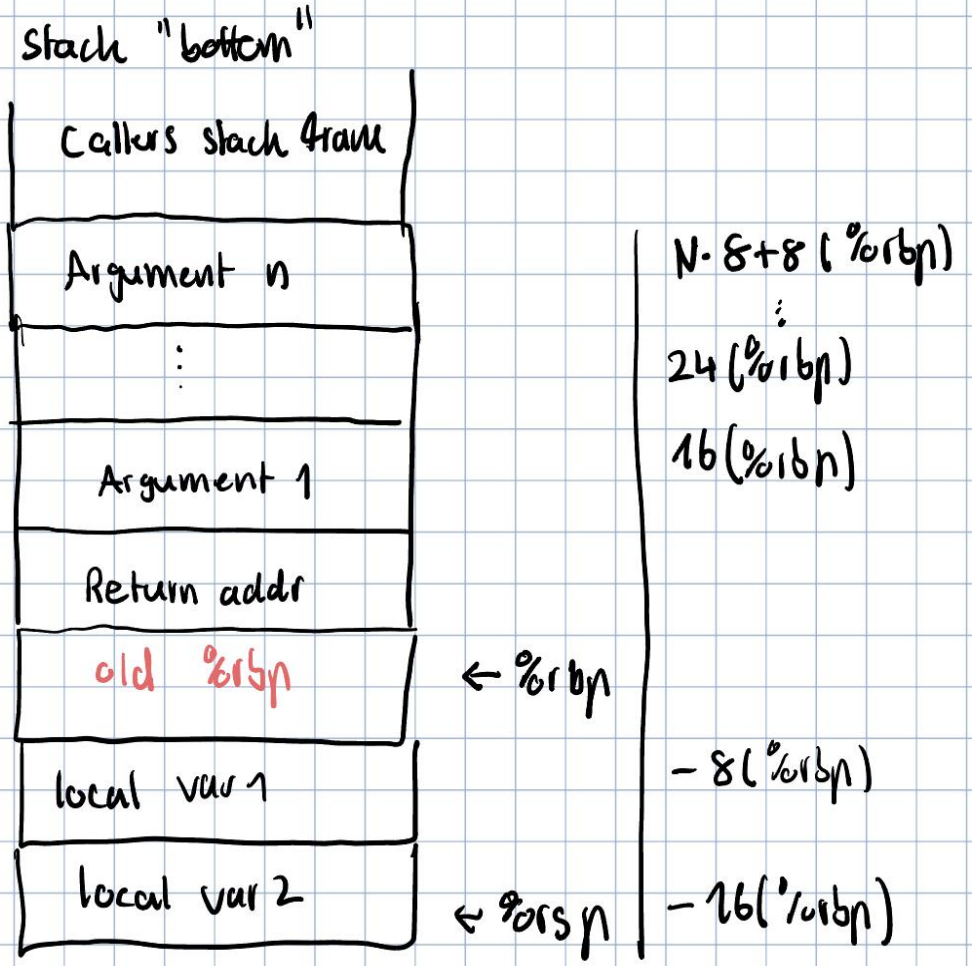
Lecture Recap: Procedure Call

Systems@ETH Zürich

2. Within the call (using the stack frame)

Lecture Recap: Proc call

- Now the callee is all set up and can start doing stuff it wants to do: for instance start pushing local variables on the stack
- Here it pushed two local variables on the stack



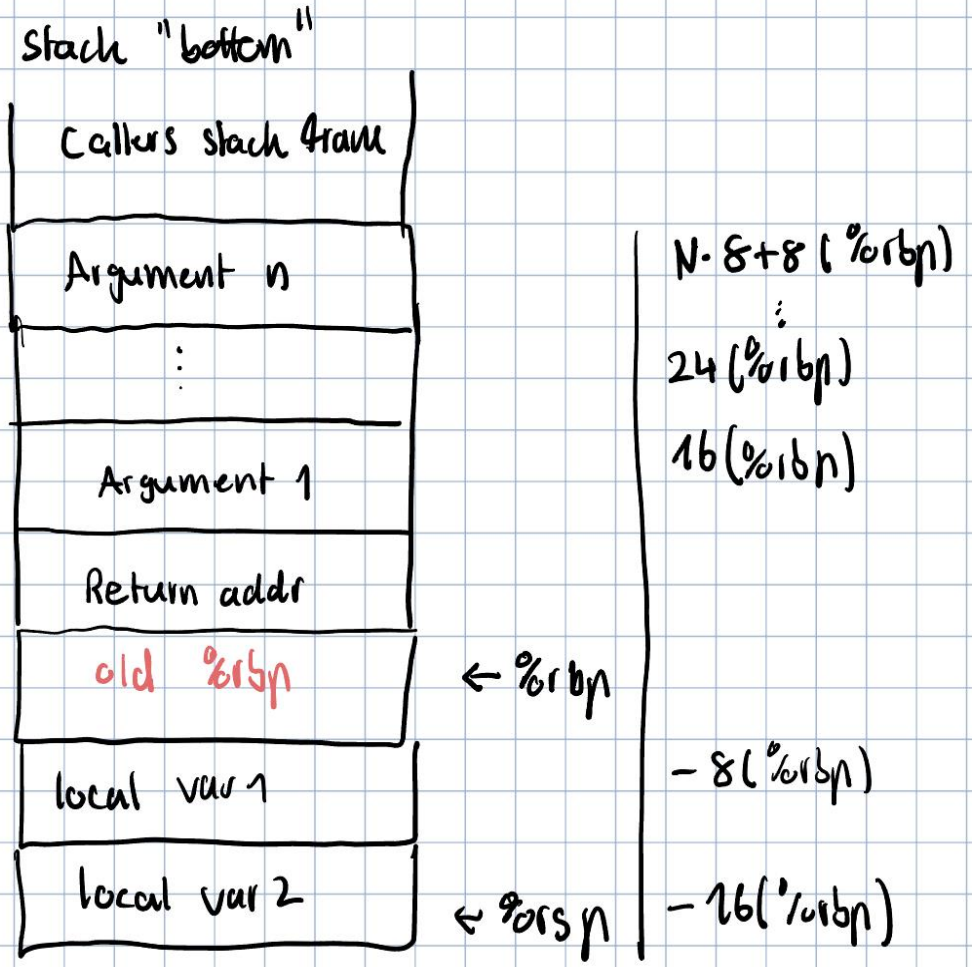
Lecture Recap: Procedure Call

Systems@ETH Zürich

Now whats the point of having %rbp and %rsp??

Lecture Recap: Proc call

- **Notice** how rsp keeps moving, but **rbp ALWAYS** points to the “base”, i.e. the beginning of the stack frame
- So accessing the passed arguments from the caller are **constant from %rbp**, NOT from %rsp as it moves

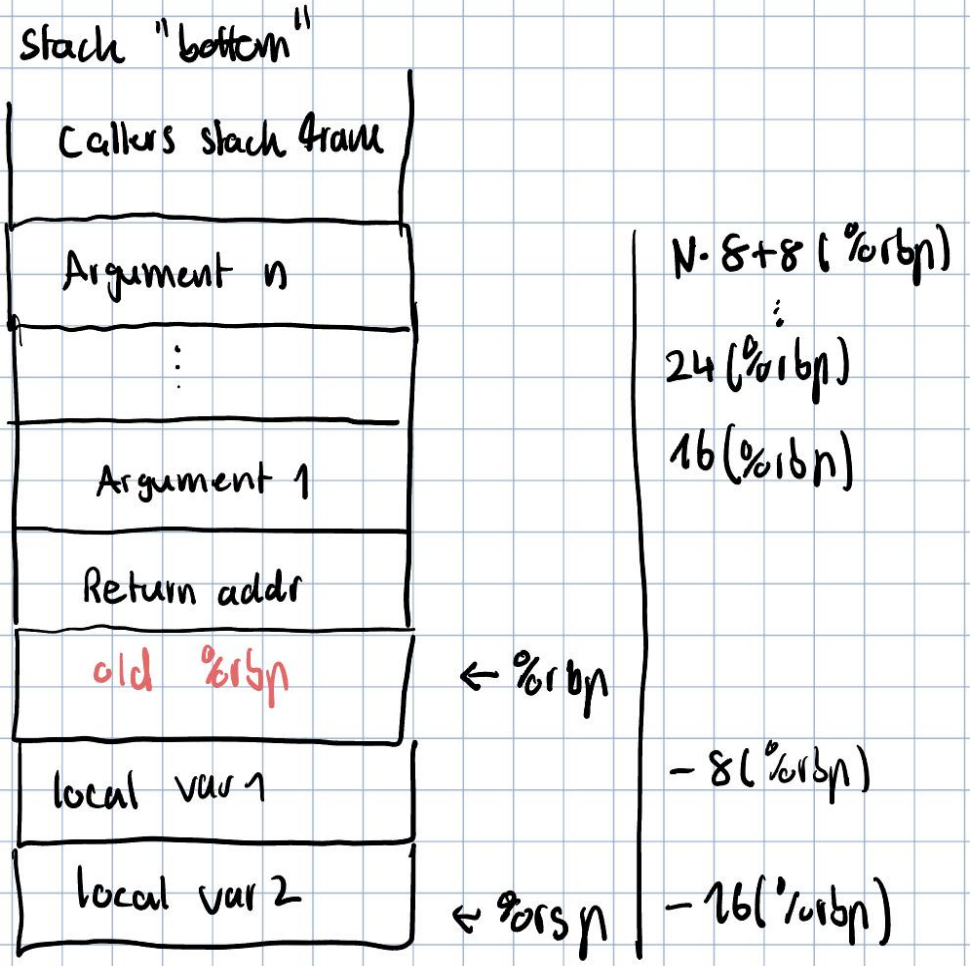


Lecture Recap: Procedure Call

3. The return (cleanup)

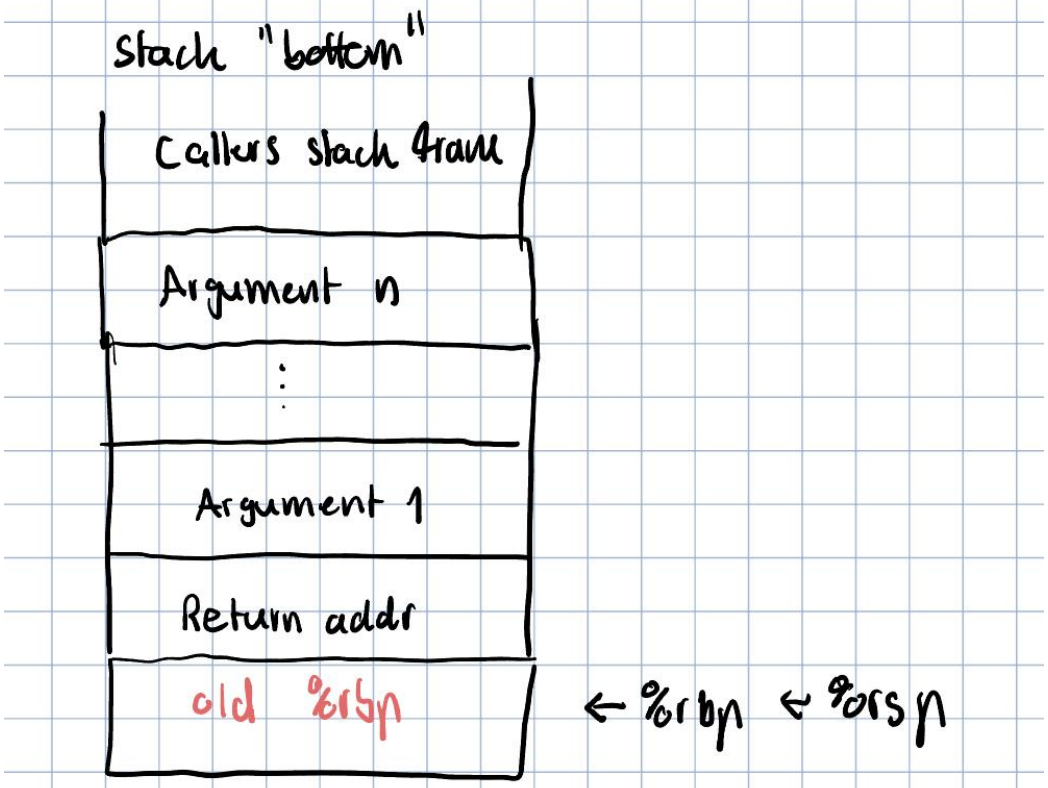
Lecture Recap: Proc call

- Stack pointer needs to point to begging of stack frame: luckily %rbp still points to it



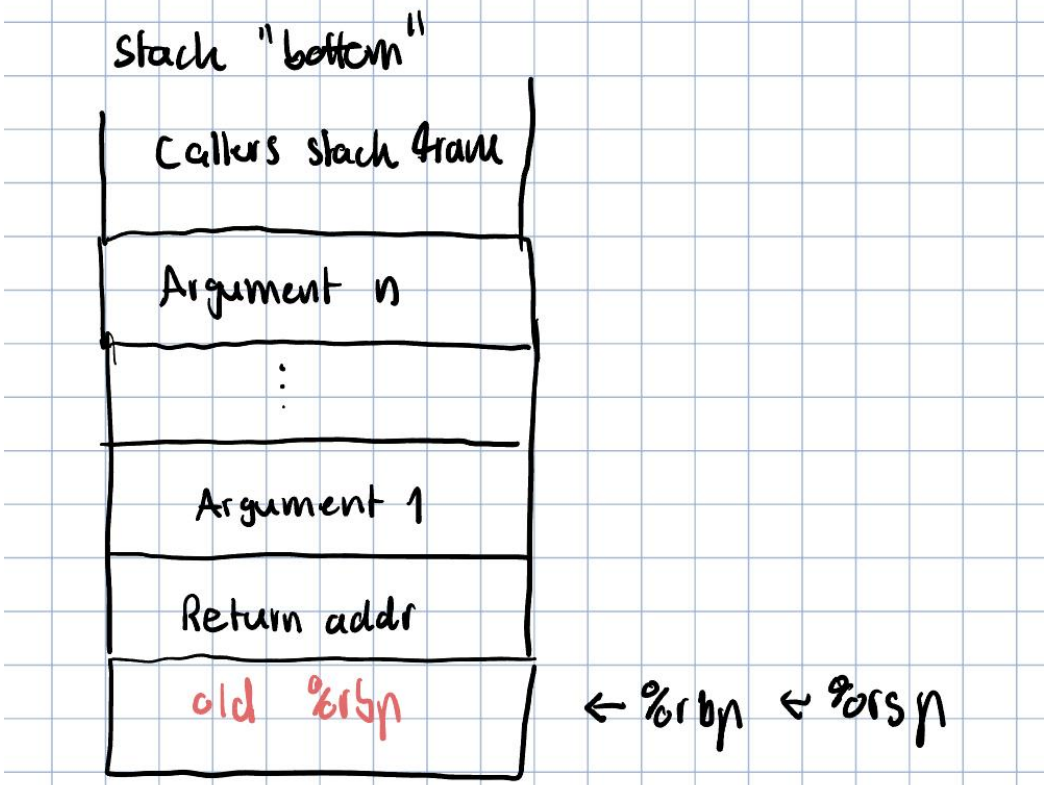
Lecture Recap: Procedure Call 2. Within the call

- Stack pointer needs to point to begging of stack frame: luckily %rbp still points to it
- “Movq %rbp, %rsp”
- (Theoretically, increment we could increment rsp but not needed here)



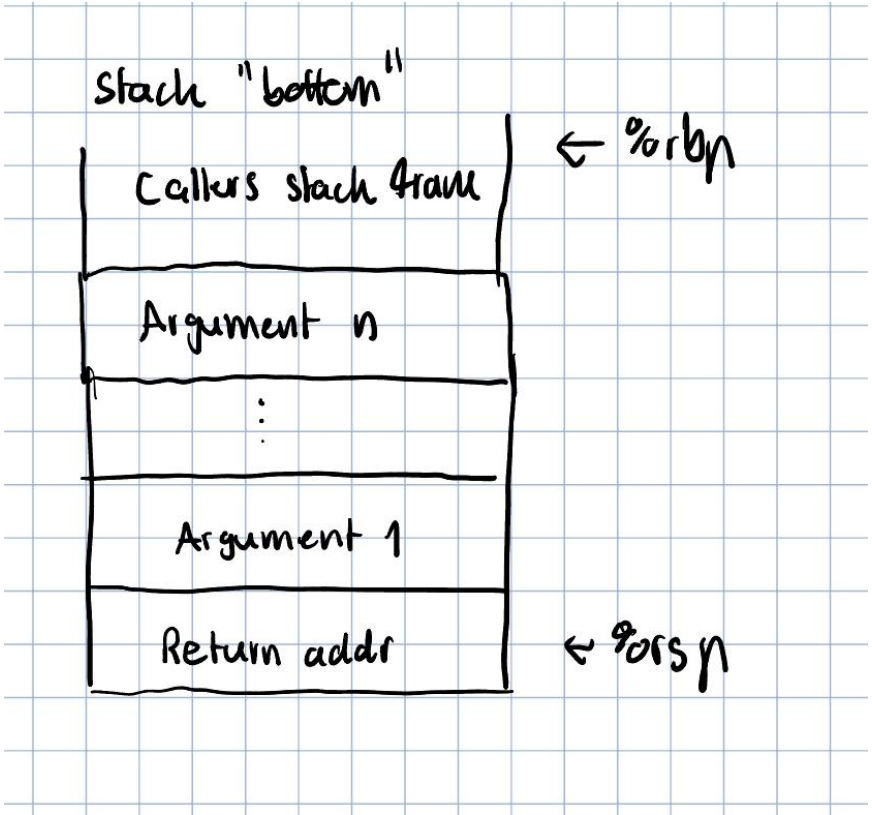
Lecture Recap: Procedure Call 2. Within the call

- Need set the %rbp to the old value (luckily we pushed in on the stack before): s.t. the caller still has this value as it was before



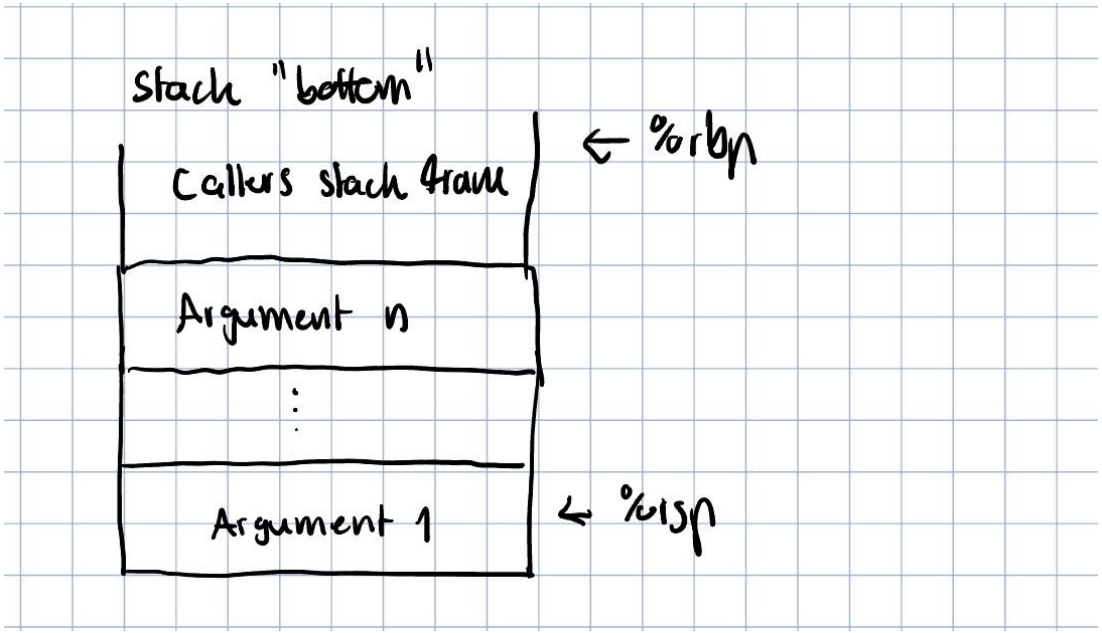
Lecture Recap: Procedure Call 2. Within the call

- Need set the %rbp to the old value (luckily we pushed in on the stack before): s.t. the caller still has this value as it was before
- “popq %rbp” (i.e. pop where %rsp is currently pointing to into %rbp)



Lecture Recap: Procedure Call 2. Within the call

- Lastly, return the %rip (instruction pointer) to the old value
- The **ret** statement **pops** the value currently pointed by %rsp into **%rip**
- Next execution executed will be in the caller again



Remark

- This explains the
- “pushq %rbp”
- “movq %rsp, %rbp”
- …
- “popq %rbp”
- “retq”

```
## %bb.0:
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register %rbp
        subq    $16, %rsp
        movl    $0, -4(%rbp)
        movl    %edi, -8(%rbp)
        movq    %rsi, -16(%rbp)
        leaq    L_.str(%rip), %rdi
        movb    $0, %al
        callq   _printf
        xorl    %eax, %eax
        addq    $16, %rsp
        popq    %rbp
        retq
        .cfi_endproc
```

Remark

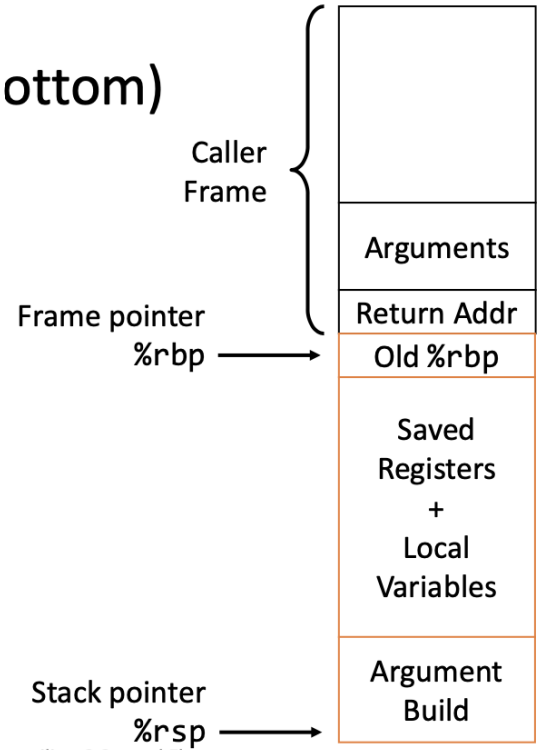
- Notice how he didn't do "movq %rbp, %rsp" here: the compiler manually added and decremented the stack pointer s.t. it points in the end to %rbp again

```
## %bb.0:
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register %rbp
        subq    $16, %rsp
        movl    $0, -4(%rbp)
        movl    %edi, -8(%rbp)
        movq    %rsi, -16(%rbp)
        leaq    L_.str(%rip), %rdi
        movb    $0, %al
        callq   _printf
        xorl    %eax, %eax
        addq    $16, %rsp
        popq    %rbp
        retq
        .cfi_endproc
```

This should be very clear now!

full x86_64/Linux stack frame

- Current stack frame ("top" to bottom)
  - "Argument build:"
    Parameters for function about to call
  - Local variables
    If can't keep in registers
  - Saved register context
  - Old frame pointer
- Caller stack frame
  - Return address
  - Pushed by `call` instruction
  - Arguments for this call



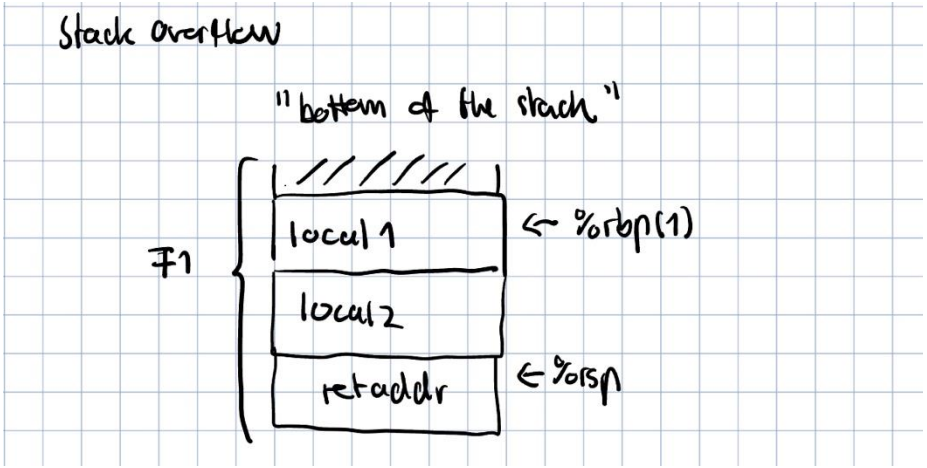
Remark: Stack Overflow

Systems@ETH Zürich

- Remark: now **knowing this** can someone tell me, what is a **stack overflow**? (which you might have already seen in eprog, pprog etc.)
- For instance, issue with a recursive function?

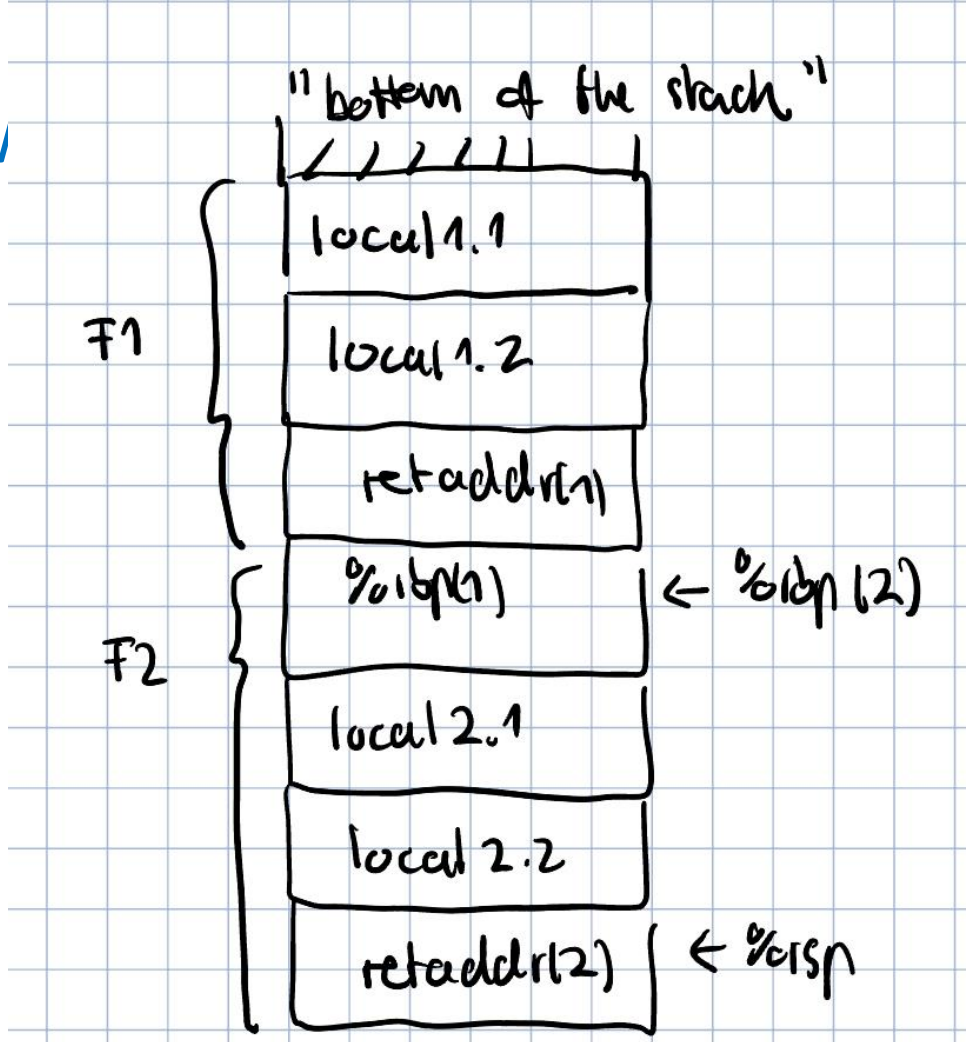
Remark: Stack Overflow

- Recursive function without appropriate base case (or too deep function calls) fill up the stack
- Say each function stores 2 arguments and then calls itself recursively:



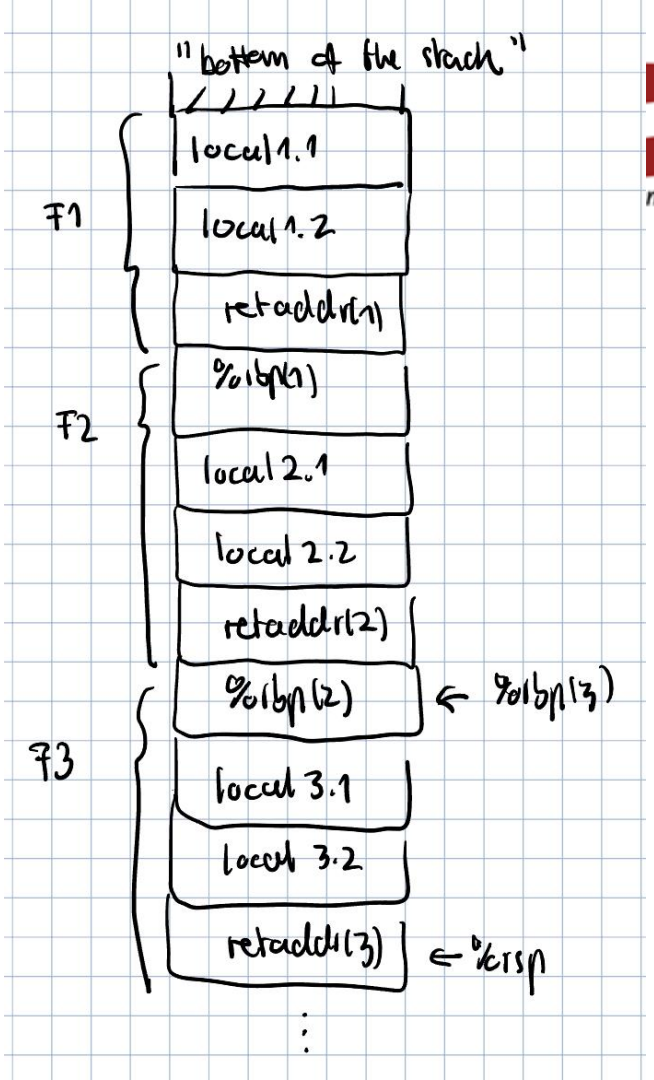
Remark: Stack Overflow

- Recursive function without appropriate base case (or too deep function calls) fill up the stack
- Say each function stores 2 arguments and then calls itself recursively:



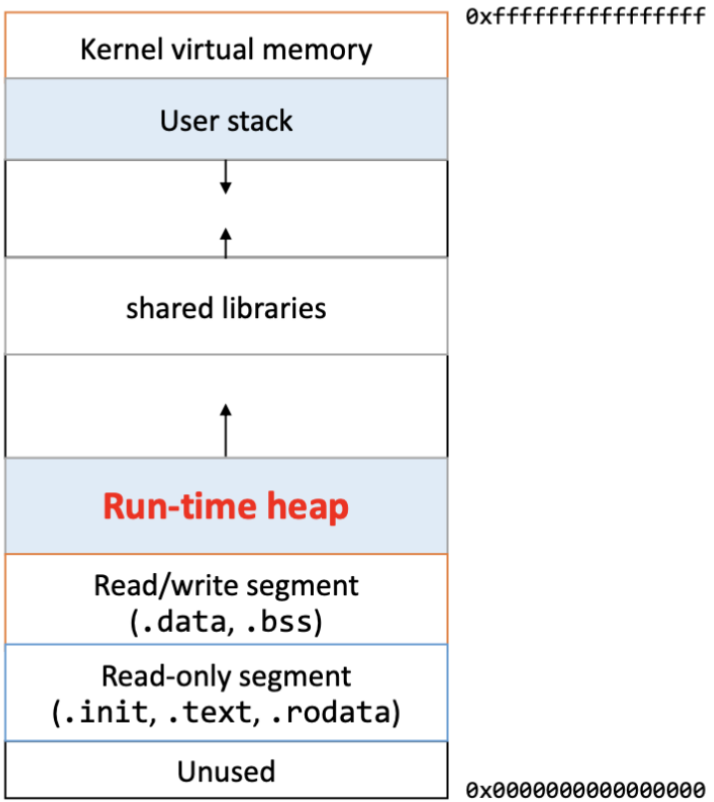
Remark: Stack Overflow

- Recursive function without appropriate base case (or too deep function calls) fill up the stack
- Say each function stores 2 arguments and then calls itself recursively:



Remark: Stack Overflow

- But the stack has limited space: once its starts growing into the middle of the address space where the shared libraries are we **get a stack overflow**



Lecture Recap

Calling Conventions

What does this mean?

Register saving conventions

- When procedure yoo calls who:
  - yoo is the ***caller***
  - who is the ***callee***

- Can register be used for temporary storage?
- Conventions
  - ***"Caller Save"***
    - Caller saves temporary in its frame before calling
  - ***"Callee Save"***
    - Callee saves temporary in its frame before using

Calling Conventions

Systems@ETH Zürich

- We saw, that callee (the called function) always **stores the base pointer** of its parent function (caller), why does he care?
- S.t. there are registers where caller can be assured they are the same as when he called the callee
- “Callee saved”: if the caller wants to change them **he has to save them**
- “Caller saved”: if the caller wants to keep them, **he has to save them**

Calling Conventions

- We saw, that callee (the called function) always **stores the base pointer** of its parent function (caller), why does he care?
- S.t. there are registers where caller can be assured they are the same as when he called the callee

%rax	Return value, # varargs
%rbx	Callee saved; base ptr
%rcx	Argument #4
%rdx	Argument #3 (& 2nd return)
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved; frame ptr

%r8	Argument #5
%r9	Argument #6
%r10	Static chain ptr
%r11	Used for linking
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

Calling Conventions

Systems@ETH Zürich

- I find “callee saved” and ”caller saved” confusing: I remember
- “**callee owned**” (caller saved): callee owns them, so he can do whatever he wants with them
- “**caller owned**” (callee saved): caller owns them, so if the callee wants to do something with it he has to save them

Systems@ETH Zürich

Lecture Recap

Compiling C Data Structures: (Struct) Alignment

Lecture Recap: Alignment

Systems@ETH Zürich

Satisfying alignment with structures

- Within structure:
  - Must satisfy element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement K
    - K = Largest alignment of any element
  - Initial address & structure length must be multiples of K

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

Lecture Recap: Alignment

Specific cases of alignment (x86-64)

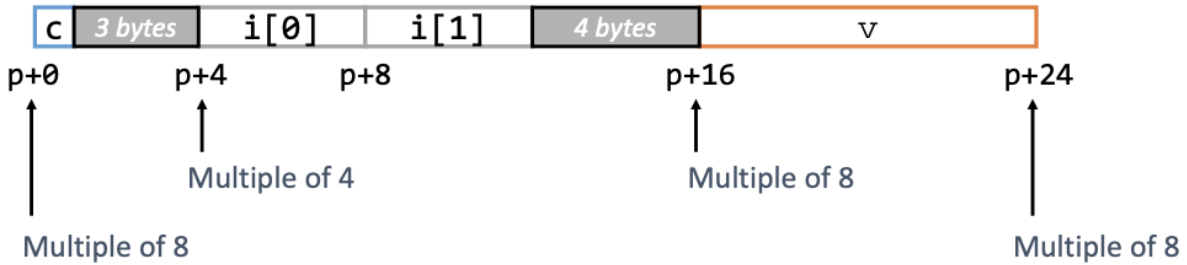
- 1 byte: `char`, …
  - no restrictions on address
- 2 bytes: `short`, …
  - lowest 1 bit of address must be 0_2
- 4 bytes: `int`, `float`, …
  - lowest 2 bits of address must be 00_2
- 8 bytes: `double`, `char *`, …
  - Windows & Linux:
    - lowest 3 bits of address must be 000_2
- 16 bytes: `long double`
  - Windows & Linux:
    - lowest 4 bits of address must be 0000_2

Lecture Recap: Alignment

Satisfying alignment with structures

- Example
  (under Windows or x86-64):
  - K = 8, due to double element

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

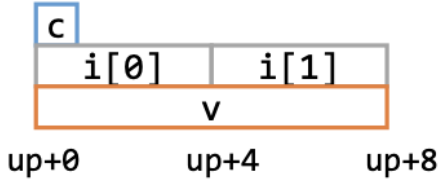


Lecture Recap: Union vs structs

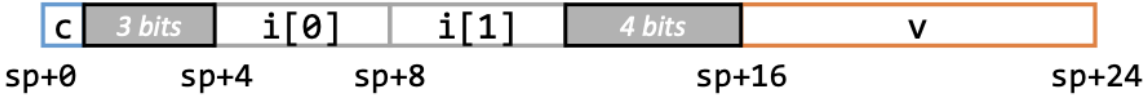
Union allocation

- Allocate according to largest element
- Can only use one field at a time

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```




```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



Lecture Overview

Where are we in the course

Lecture Overview

- Now you looked at
- **1. C programming**: what kind of constructs exists (source code), if statements, loops etc.
- **2. x86 Assembly**: what happens with your highlevel sourcecode: gets translated to x86, now you looked at how the loops, if statements etc. get translated
- **Upcomingin 2.**: unorthodox control flow (almost threads), Linking, Floating Point, optimizing Compilers
- **3. Computer Architecture**: Architecture, Caches, Exception, Virtual Memory etc.

General Consolidation Q&A

- C Basics
- C Integers
- Pointers
- Dynamic Memory in C
- C Pre-Processor
- C Compilation Pipeline
- Dynamic Memory Allocators
- Assembly Basics (Registers, Instructions, Memory Addressing)
- Compiling C Control Flow
- Compiling C Data Structures
- x86 Calling Conventions

With your neighbor(s), take a moment to think about the topics

Q&A

Exam Quiz

Pointer Dereference & Array Access

Pointers 1:

Replacing `// XXXXX` with A-G, which print "5 5"?

```
int val = 3;
int new_val = 5;
int* val_ptr = &val;
// XXXXX
printf("%d %d\n",
       val, *val_ptr);
```

A. 5 5

B. 5 5

C. 3 5

D. 5 5

E. 5 5

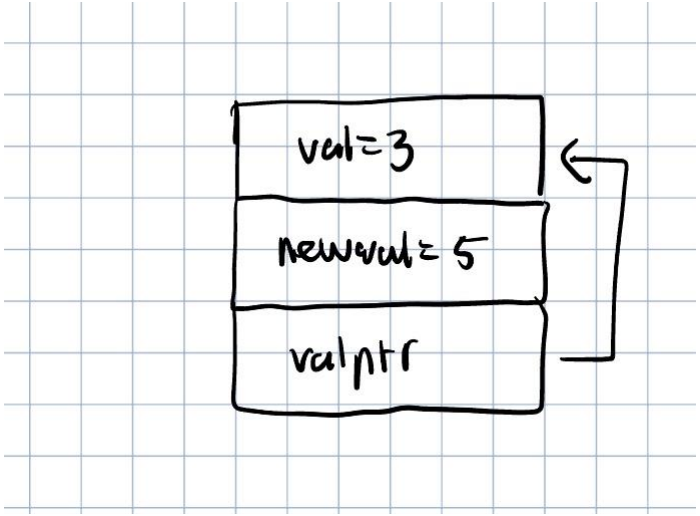
F. 5 5

G. 5 5

```
A. val = 5;
B. *val_ptr = new_val;
C. val_ptr = &new_val;
D. val = new_val;
E. *val_ptr = *val_ptr + 2;
F. val = val + 2;
G. val = *val_ptr + 2;
```

Pointers 1: Solution

- **Care**: this is NOT allocating stuff on the heap, as there is no malloc: this is simply a pointer to a value on the stack



Pointers 2: HS21-5b)

Systems@ETH Zürich

On a 64-bit machine, which of the following C expressions is equivalent to: (2 points)

```
(x[2] + 4)[3]
```

A. `*((*(x + 16)) + 28)`

B. `*((*(x + 2)) + 7)`

C. `**(x + 28)`

D. `*(((* x) + 2) + 7)`

E. `(**(x + 2) + 7)`

Pointers 2: Solution Recall

Systems@ETH Zürich

In fact…

- `A[i]` is *always* rewritten `*(A+i)` in the compiler

```
int a[10];
assert(a == &(a[0]));
assert(a[5] == 5[a]);
```

```
int get_2(int i)
{
    int *p = a;
    return i[p];
}
void set_2(int i, int v)
{
    int *p = a;
    i[p] = v;
}
```

Pointers 2: Solution

- `(x[2] + 4)[3]`
- =(*(x+2)+4)[3]
- =*(*(x+2)+4+3)
- =*(*(x+2)+7)

Exam Quiz

Assembly: What is ARG?

For the following task, recall addressing modes

Systems@ETH Zürich

- Most General Form:

D(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]+ D]

  - D: Constant "displacement" 1, 2, or 4 bytes (not 8!)
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for %rsp (Unlikely you'd use %rbp, either)
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

- Special Cases:

(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]]
D(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]+D]
(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]]

Disp (reg1, reg2, factor) ≡ reg1 + reg2 · factor + Disp

Ex: 2 (%rax, %rdi, 4) = %rax + 4 · %rdi + 2

For the following task, recall addressing modes

- Recall: only **lea** does not dereference, i.e.
- Leaq (%rdi), %rax ⇔ %rax<-%rdi, literally “copy value of %rdi into %rax

- Recall: for **any other instruction** supporting the full addressing mode (movq, addq, subq etc.) it dereferences
- Movq (%rdi), %rax ⇔ %rax<- *(%rdi), look into memory address at location given by %rdi and take this value

Remark

- These are **actually** exam tasks: pay attention and check that you really understand how this works, o/w ask!

Systems@ETH Zürich

Question 2 [18 points]

Consider the following C function, where ARG is a decimal constant defined elsewhere:

```
#include <stdint.h>
int64_t multiply( int64_t a )
{
    return a * ARG;
}
```

The following disassemblies are of versions of the `multiply` function, compiled with different values for `ARG` and different optimization settings in the compiler.

Recall that the first (in this case, only) argument to a function is passed in the `rdi` register, and the result returned in the `rax` register.

For each one, say what the value of `ARG` was, and explain why.

```
0000000000000000 <multiply>:
   0: 55                    push   %rbp
   1: 48 89 e5              mov    %rsp,%rbp
   4: 48 89 7d f8           mov    %rdi,-0x8(%rbp)
   8: 48 8b 45 f8           mov    -0x8(%rbp),%rax
   c: 48 c1 e0 02           shl    $0x2,%rax
  10: 5d                    pop    %rbp
  11: c3                    retq
```

(3 points)

Answer:

Assembly: HS17-2-(0)

Systems@ETH Zürich

You have the following C Program:

```
int64_t main(int64_t a) {
 return a * ARG;
}
```

In the following disassemblies, what is ARG?

Assembly: HS17-2-(1)

```
int64_t main(int64_t a) {
 return a * ARG;
}
```

What is ARG?

```
0000000000000000 <main>:
    0:  55              push %rbp
    1:  48 89 e5        mov  %rsp, %rbp
    4:  48 89 7d f8     mov  %rdi, -0x8(%rbp)
    8:  48 8b 45 f8     mov  -0x8(%rbp), %rax
    c:  48 c1 e0 02     shl  $0x2, %rax
   10:  5d              pop  %rbp
   11:  c3              retq
```

Assembly: HS17-2-(1)

```
int64_t main(int64_t a) {
 return a * ARG;
}
```

What is ARG?

ARG == 4

```
0000000000000000 <main>:
    0:  push %rbp
    1:  mov  %rsp, %rbp
    4:  mov  %rdi, -0x8(%rbp)  # store a
    8:  mov  -0x8(%rbp), %rax  # load a => rax = a
    c:  shl  $0x2, %rax        # rax = 4 * a
    10: pop  %rbp
    11: retq
```

Assembly: HS17-2-(2)


```
int64_t main(int64_t a) {
 return a * ARG;
}
```

What is ARG?

```
0000000000000000 <main>:
  0: 48 8d 04 bd 00 00 00 00  lea  0x0(, %rdi, 4), %rax
  8: c3                       retq
```

Assembly: HS17-2-(2)

```
int64_t main(int64_t a) {
 return a * ARG;
}
```

What is ARG?

ARG == 4

```
0000000000000000 <main>:
  0: lea  0x0(, %rdi, 4), %rax  # rax = 0 + 4*a + 0
  8: retq
```

Assembly: HS17-2-(3)


```
int64_t main(int64_t a) {
 return a * ARG;
}
```

What is ARG?

```
0000000000000000 <main>:
   0: 48 8d 04 7f  lea  0x0(%rdi, %rdi, 2), %rax
   4: 48 8d 04 87  lea  0x0(%rdi, %rax, 4), %rax
   8: c3           retq
```

Assembly: HS17-2-(3)


```
int64_t main(int64_t a) {
 return a * ARG;
}
```

What is ARG?

ARG == 13

```
0000000000000000 <main>:
   0: lea  0x0(%rdi, %rdi, 2), %rax  # rax = a + 2a
   4: lea  0x0(%rdi, %rax, 4), %rax  # rax = a + 4*(3a)
   8: retq
```

Assembly: HS17-2-(4)

```
int64_t main(int64_t a) {
 return a * ARG;
}
```

```
0000000000000000 <main>:
  0: 31 c0 xor  %eax %eax
  2: c3    retq
```

What is ARG?

Assembly: HS17-2-(4)


```
int64_t main(int64_t a) {
 return a * ARG;
}
```

```
0000000000000000 <main>:
  0: xor  %eax %eax # eax = 0
  2: retq
```

What is ARG?

ARG == 0

Assembly: HS17-2-(5)


```
int64_t main(int64_t a) {
 return a * ARG;
}
```

What is ARG?

```
0000000000000000 <main>:
   0: 48 8d 04 fd 00 00 00 00 lea  0x0(, %rdi, 8), %rax
   8: 48 29 f8                sub  %rdi, %rax
   b: 48 8d 04 c7             lea  (%rdi, %rax, 8), %rax
   f: c3                      retq
```

Assembly: HS17-2-(5)

```
int64_t main(int64_t a) {
 return a * ARG;
}
```

What is ARG?

ARG == 57

```
0000000000000000 <main>:
  0: lea  0x0(, %rdi, 8), %rax   # rax = 8 * a
  8: sub  %rdi, %rax             # rax = 7a
  b: lea  (%rdi, %rax, 8), %rax  # rax = a + 8 * (7a)
  f: retq
```

Assembly: HS17-2-(6)


```
int64_t main(int64_t a) {
 return a * ARG;
}
```

What is ARG?

```
0000000000000000 <main>:
  0:   48 89 f8    mov  %rdi, %rax
  3:   48 f7 d8     neg  %rax
  6:   48 c1 e0 02 shl  $0x2, %rax
  a:   48 29 f8    sub  %rdi, %rax
  d:   48 c1 e0 02 shl  $0x2, %rax
  11: c3            retq
```

Assembly: HS17-2-(6)


```
int64_t main(int64_t a) {
 return a * ARG;
}
```

What is ARG?

ARG == -20

```
0000000000000000 <main>:
  0:  mov  %rdi, %rax  # rax = a
  3:  neg  %rax        # rax = -a
  6:  shl  $0x2, %rax  # rax = -4a
  a:  sub  %rdi, %rax  # rax = -4a – a = -5a
  d:  shl  $0x2, %rax  # rax = 4 * (-5a) = -20a
  11: retq
```

Side Note: `objdump -d`

showed:

```
0000000000000000 <main>:
   0: 48 8d 04 bd 00 00 00 00 lea  0x0(, %rdi, 4), %rax
   8: c3                      retq
```

```
0000000000000000 <main>:
   0: 48 8d 04 bd 00 00 00    lea  0x0(, %rdi, 4), %rax

   8: c3                      retq
```

Side Note: `objdump -d`

use

no wrap: `objdump -d --wide`

custom wrap: `objdump -d --insn-width=8`

What even is objdump

- If you did malloclab you know; if you do the attack or bomb lab you will get to use objdump too

- It simply translates **machine code** back into **assembly**

What even is ob

simpleadd.c

Users > benediktfalk > ctesting >

```
1   int add(){
2     int a=5;
3     int b=10;
4     int c=a+b;
5
6     return c;
7
8   }
9
10  int main(int argc, cha
11    add();
12    return 0;
13  }
```

```
0000000000401106 <add>:
  401106:       55                      push   %rbp
  401107:       48 89 e5                mov    %rsp,%rbp
  40110a:       c7 45 fc 05 00 00 00    movl   $0x5,-0x4(%rbp)
  401111:       c7 45 f8 0a 00 00 00    movl   $0xa,-0x8(%rbp)
  401118:       8b 55 fc                mov    -0x4(%rbp),%edx
  40111b:       8b 45 f8                mov    -0x8(%rbp),%eax
  40111e:       01 d0                   add    %edx,%eax
  401120:       89 45 f4                mov    %eax,-0xc(%rbp)
  401123:       8b 45 f4                mov    -0xc(%rbp),%eax
  401126:       5d                      pop    %rbp
  401127:       c3                      retq

0000000000401128 <main>:
  401128:       55                      push   %rbp
  401129:       48 89 e5                mov    %rsp,%rbp
  40112c:       48 83 ec 10             sub    $0x10,%rsp
  401130:       89 7d fc                mov    %edi,-0x4(%rbp)
  401133:       48 89 75 f0             mov    %rsi,-0x10(%rbp)
  401137:       b8 00 00 00 00          mov    $0x0,%eax
  40113c:       e8 c5 ff ff ff          callq  401106 <add>
  401141:       b8 00 00 00 00          mov    $0x0,%eax
  401146:       c9                      leaveq
  401147:       c3                      retq
```

What even is objdump

Systems@ETH Zürich

- Objdump: flags
- -d: show disassembly in human readable format
- --wide: comprehensive analysis without line breaks (different view)
- --insn-width=8: show instructions in a manageable 8-byte width

- Google for more flags

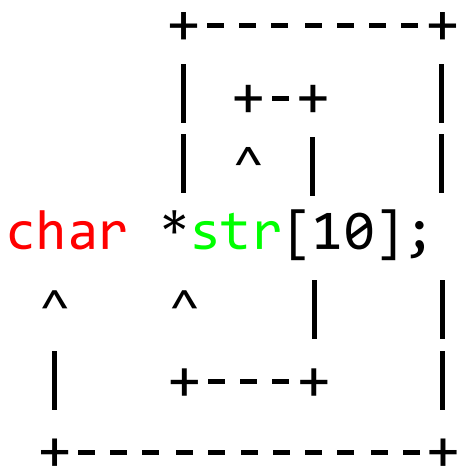
C Declarations:
Clockwise/Spiral Rule

Clockwise/Spiral Rule

c-faq.com/decl/spiral.anderson.html

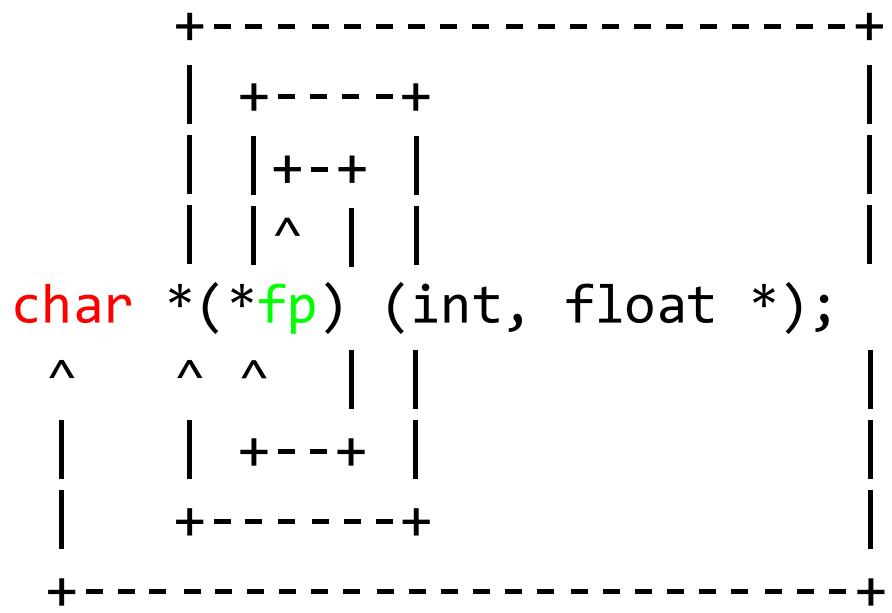
- Start at the variable name
- Always end with the basic type (int, char, long, etc.)
- Then fill in the “middle part”
- “go right when you can, go left when you must”

Clockwise/Spiral Rule



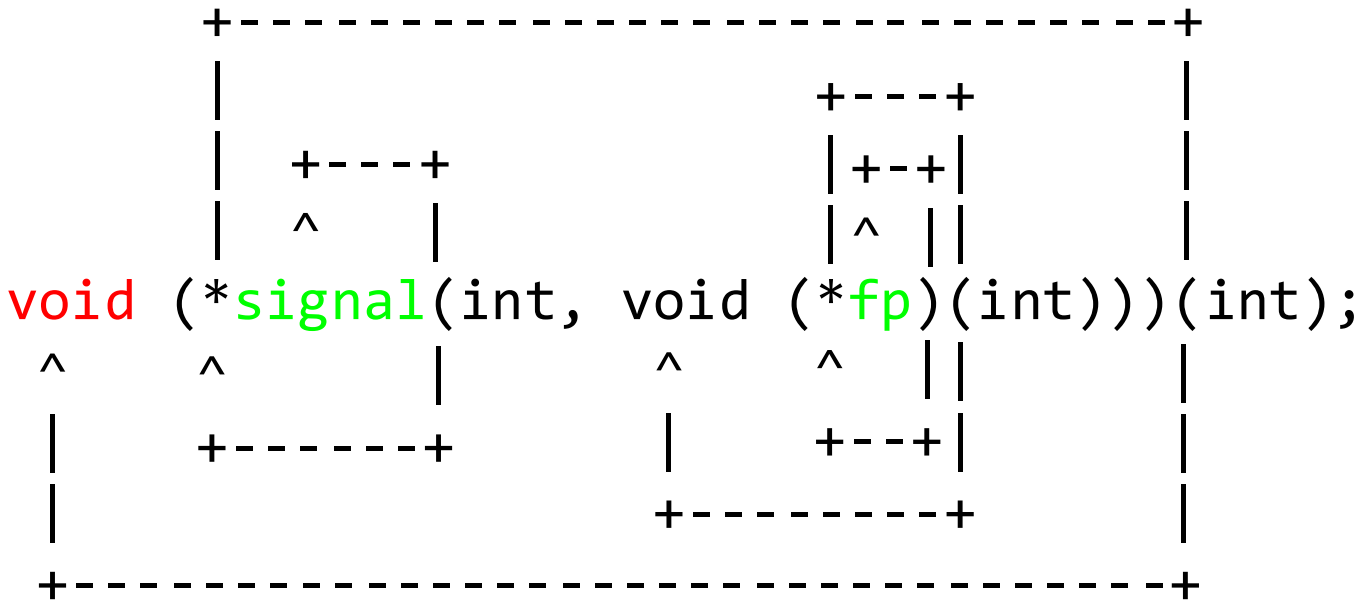
str is an array 10 of pointers to char

Clockwise/Spiral Rule



fp is a pointer to a function passing an int and a pointer to float returning a pointer to a char

Clockwise/Spiral Rule

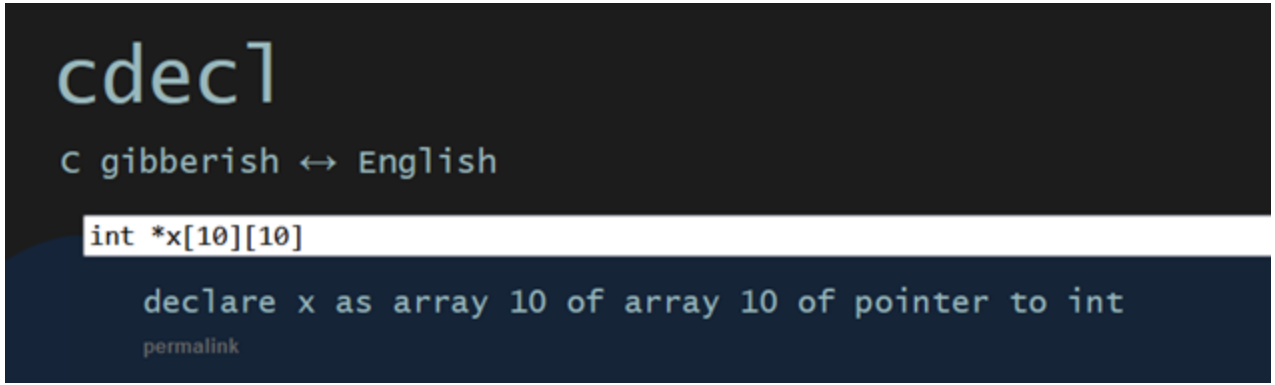


signal is a function passing an int and a
pointer to a function passing an int returning void returning
a pointer to a function passing an int returning void

Tool

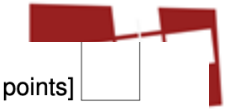
cdecl.org

good for training



Remark

- These are **actually** exam tasks: pay attention and check that you really understand how this works, o/w ask!
- Do the following and you will be very good at it (its not hard, after a doing a few **by yourself** you got it)
- HS18 Ex. 7
- HS19 Ex. 3
- HS20 Ex. 6



Question 7 [10 points]

Consider the following list of C syntax fragments:

```
1. int *x;
2. int x[10];
3. (int *)x;
4. int *x[10];
5. int **x[10];
6. int (*x [10])[10];
7. int *x[10][10];
8. int x[10][5];
9. int x[5][10];
10. int (*x)[](int *);
11. int *x(int []);
12. int (*x[])(int *);
13. int *x(int *);
14. int *x[](int *);
15. (int (*)(int *, int))x;
16. int *x(int *, int);
17. (int (*)(int[], int))x;
18. int *x(int *, int)[];
19. (int *(*)(int *, int))x;
20. int (*x)(int *, int);
21. int *x[](int *, int);
```

For each of the descriptions on the following page, give the number of the C syntax fragment that corresponds precisely to it.

Remark

- Two kind of those questions: either **i) casting** or **ii) type decleration**

Int *x; ⇔ “declare x as pointer to int”

(int*) x; ⇔ “cast x into pointer to int”

Remark

- **Remark**: multidimensional array binds higher
- int *x[10][10]; ⇔ “declare x as array 10 of array 10 of pointer to int” NOT “declare x as a array 10 of pointer to array 10 of int)

Exam Quiz

C Declarations

Remark

- Lets go through some of the HS18 together and then you can try the following

C Declarations: HS18-7-(1)

```
1. int *x
2. int x[10]
3. (int *)x
4. int *x[10]
5. int **x[10]
6. int(*x[10])[10]
7. int *x[10][10]
```

A. cast x into pointer to int

B. declare x as array 10 of pointer to int

C. declare x as array 10 of pointer to array 10 of int

Match statements (right) to code (left).

C Declarations: HS18-7-(2)

```
7. int *x[10][10]
8. int x[10][5]
9. int x[5][10]
10.int (*x)[](int *)
11.int *x(int[])
12.int (*x[])(int *)
13.int *x(int *)
```

D. Declare x as array 10 of array 5 of int

E. Declare x as pointer to array of function (pointer to int) returning int

F. Declare x as array of pointer to function (pointer to int) returning int

Match statements (right) to code (left).

C Declarations: HS18-7-(2)

```
13.int *x(int *)
14.int *x[](int *)
15.(int (*)(int *, int))x
16.int *x(int *, int)
17.(int (*)(int[], int))x
18.int *x(int *, int)[]
19.(int *(*)(int *, int))x
20.int (*x)(int *, int)
21.int *x[](int *, int)
```

G. Declare x as array of function (pointer to int) returning pointer to int

H. Cast x into pointer to function (array of int, int) returning int

I. Cast x into pointer to function (pointer to int, int) returning pointer to int

J. Declare x as pointer to function (pointer to int, int) returning int

Match statements (right) to code (left).

Kahoot

https://create.kahoot.it/share/sysprog-q5/baee3cb6-319f-469b-abd4-ae6ac0afa13b

(16’)

Q&A

(again)

Exam Quiz

Read Assembly and Arrays

WARNING

this is hard

```
.build_version macos, 12, 0 sdk_version 12, 0
_copy:
  pushq   %rbp
  movq    %rsp, %rbp
  movslq  %edx, %rax
  movslq  %edi, %rcx
  movslq  %esi, %rdx
  movq    %rcx, %rsi
  shlq    $6, %rsi
  leaq    (%rsi, %rcx, 4), %rsi
  imulq   $340, %rax, %rdi
  addq    _array2@GOTPCREL(%rip), %rdi
  addq    %rsi, %rdi
  movl    (%rdi, %rdx, 4), %esi
  movq    %rdx, %rdi
  shlq    $7, %rdi
  leaq    (%rdi, %rdx, 4), %rdx
  imulq   $2244, %rcx, %rcx
  addq    _array1@GOTPCREL(%rip), %rcx
  addq    %rdx, %rcx
  movl    %esi, (%rcx, %rax, 4)
  popq    %rbp
  retq
.subsections_via_symbols
```

```
int array1[X][Y][Z];
int array2[Z][X][Y];
void copy(int i, int j, int k) {
 array1[i][j][k] = array2[k][i][j];
}
```

`movslq`: 32->64 bit sign extension

`foo@GOTPCREL(%rip)` is the GOT entry for symbol `foo`, accessed with RIP-relative addressing mode.

i.e. this loads `foo`.

```
# starts with: %edi = i, %esi = j, %edx = k
pushq   %rbp
movq    %rsp, %rbp
movslq  %edx, %rax
movslq  %edi, %rcx
movslq  %esi, %rdx
movq    %rcx, %rsi

shlq    $6, %rsi
leaq    (%rsi, %rcx, 4), %rsi
imulq   $340, %rax, %rdi
addq    _array2@GOTPCREL(%rip), %rdi
addq    %rsi, %rdi
movl    (%rdi, %rdx, 4), %esi
movq    %rdx, %rdi
shlq    $7, %rdi
leaq    (%rdi, %rdx, 4), %rdx
imulq   $2244, %rcx, %rcx
addq    _array1@GOTPCREL(%rip), %rcx
addq    %rdx, %rcx
movl    %esi, (%rcx, %rax, 4)

popq    %rbp
retq
```

- new stack frame: %rsp and %rbp
- handle arguments

```
int array1[X][Y][Z];
int array2[Z][X][Y];
void copy(int i, int j, int k) {
 array1[i][j][k] = array2[k][i][j];
}
```

%rax	Return value	%r8	5th argument
%rbx	Callee saved	%r9	6th argument
%rcx	4th argument	%r10	Scratch register
%rdx	3rd argument	%r11	Scratch register
%rsi	2nd argument	%r12	Callee saved
%rdi	1st argument	%r13	Callee saved
%rbp	Callee saved	%r14	Callee saved
%rsp	Stack pointer	%r15	Callee saved

```
# starts with: %edi = i, %esi = j, %edx = k
pushq   %rbp
movq    %rsp, %rbp
movslq  %edx, %rax                      # rax = k
movslq  %edi, %rcx                      # rcx = i
movslq  %esi, %rdx                      # rdx = j
movq    %rcx, %rsi                      # rsi = i

shlq    $6, %rsi                        # rsi = 64 * i
leaq    (%rsi, %rcx, 4), %rsi           # rsi = 4i + 64i = 68i
imulq   $340, %rax, %rdi                # rdi = 340 * k
addq    _array2@GOTPCREL(%rip), %rdi    # rdi = &array2[0][0][0] + 340k
addq    %rsi, %rdi                      # rdi = 68i + (array2 + 340k)
movl    (%rdi, %rdx, 4), %esi           # esi = *(array2 + 340k + 68i + 4j)
movq    %rdx, %rdi                      # rdi = j
shlq    $7, %rdi                        # rdi = 128 * j
leaq    (%rdi, %rdx, 4), %rdx           # rdx = 4j + 128j = 132j
imulq   $2244, %rcx, %rcx               # rcx = 2244i
addq    _array1@GOTPCREL(%rip), %rcx    # rcx = array1 + 2244i
addq    %rdx, %rcx                      # rcx = array1 + 2244i + 132j
movl    %esi, (%rcx, %rax, 4)           # *(array1 + 2244i + 132j + 4k) = esi

popq    %rbp
retq
```

```
int array1[X][Y][Z];
int array2[Z][X][Y];
void copy(int i, int j, int k) {
 array1[i][j][k] = array2[k][i][j];
}
```

```
int array1[X][Y][Z];
int array2[Z][X][Y];
void copy(int i, int j, int k) {
 array1[i][j][k] = array2[k][i][j];
}
```


```
# combining the analysis of the assembly code shows:
*(array1 + 2244i + 132j + 4k) = *(array2 + 340k + 68i + 4j)

# integer size is 4 B => in C:
*(array1 + 561i + 33j + k) = *(array2 + 85k + 17i + j)

# rewrite:
*(array1 + (17 * 33) * i + 33 * j + k) = *(array2 + (5 * 17) * k + 17 * i + j)

*(array1 + 33 * (17i + j) + k) = *(array2 + 17 * (5k + i) + j)

# => array1: outermost dimension must be 33, middle 17
# => array2: outermost dimension must be 17, middle 5

# looking at the declarations of array1 and array2:

X = 5
Y = 17
Z = 33
```

```
int array1[X][Y][Z];
int array2[Z][X][Y];
void copy(int i, int j, int k) {
 array1[i][j][k] = array2[k][i][j];
}
```

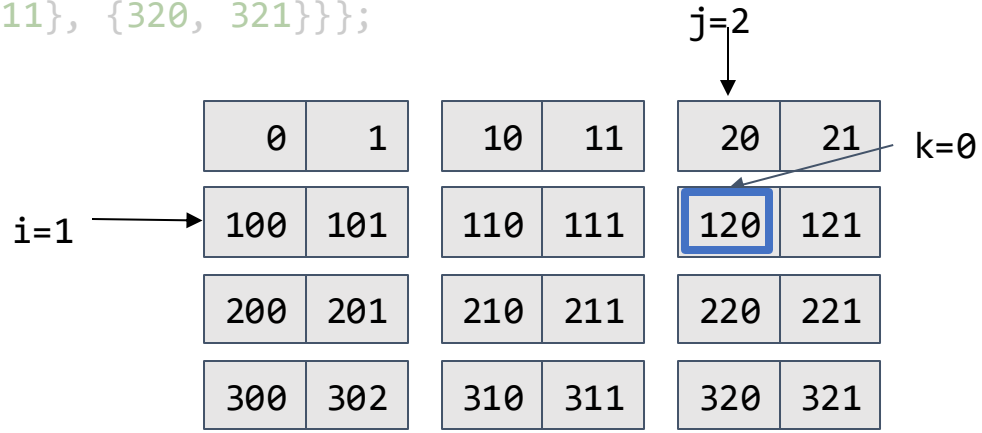
smaller example:
array1:

```
X = 4, Y = 3, Z = 2;
int array1[4][3][2]; // 4 x 3 x 2

int array1[4][3][2] = {{{  0,   1}, { 10,  11}, { 20,  21}},
                       {{100, 101}, {110, 111}, {120, 121}},
                       {{200, 201}, {210, 211}, {220, 221}},
                       {{300, 302}, {310, 311}, {320, 321}}};

int i = 1, j = 2, k = 0;
array1[1][2][0];
```

address:

```
array1 + 4 * ( 1 * (3 * 2) + 2 * 2 + 0 );
array1 + 4 * ( i * (Y * Z) + j * Z + k );
```

int is 4 bytes



Alternative, faster approach:

guess dimensions from constants

```
int array1[X][Y][Z];
int array2[Z][X][Y];
void copy(int i, int j, int k) {
 array1[i][j][k] = array2[k][i][j];
}
```


```
                                # 2 suspiciously important constants
shlq    $6, %rsi
leaq    (%rsi, %rcx, 4), %rsi
imulq   $340, %rax, %rdi        # 340/4 = 75  =>  75 = 5*17
addq    _array2@GOTPCREL(%rip), %rdi
addq    %rsi, %rdi
movl    (%rdi, %rdx, 4), %esi
movq    %rdx, %rdi
shlq    $7, %rdi
leaq    (%rdi, %rdx, 4), %rdx   # knowing 5 & 17:
imulq   $2244, %rcx, %rcx       # 2244/4 = 561  =>  561/17 = 33
addq    _array1@GOTPCREL(%rip), %rcx#                  (divide by 5 does not work)
addq    %rdx, %rcx
movl    %esi, (%rcx, %rax, 4)
                                # => 5, 17, 33. Order??
```

in the exam:

5 points / 173

in 180 min

-> not worth it