Assembly part 2

Areas for growth: I love feedback

- **Speed**, I will go slower.
- **Clarity.** I will take time to explain everything on the slides.
- **Feedback.** I give more Kahoot questions and explain each answer.
- **Pointers:** I use a pointer or pen to highlight the section of the slide that is currently being discussed.

- Feedback is good, give me more :) I will not share your feedback with class, but I will highlight areas for growth.

Last Time

- linking extras:
  - relocations and types dynamic linking (briefly)
- AT&T syntax
  - destination last
  - `O(B, I, S)` — $B + I \times S + O$
- condition codes — last arithmetic result
- Questions?

Goals Learning/Outcomes

- Review LEA
- Review Condition codes.
- Finish and review C code translation
- Intro to C
- && and II
- Pointer Arthematic

LEA tricks

`leaq (%rax,%rax,4), %rax`

rax ← rax × 5

rax ← `address-of(memory[rax + rax * 4])`

`leaq (%rbx,%rcx), %rdx`

rdx ← rbx + rcx

rdx ← `address-of(memory[rbx + rcx])`

exercise: what is this function?

```
mystery:
    leal 0(,%rdi,8), %eax
    subl %edi, %eax
    ret
```

```
int mystery(int arg) { return ...; }
```

A. arg * 9

B. -arg * 9

C. none of these

D. arg * 8

https://create.kahoot.it/kahoots/my-kahoots

Condition Codes (Implicit Setting)

- Single bit registers
  - **CF** Carry Flag (for unsigned) **SF** Sign Flag (for signed)
  - **ZF** Zero Flag **OF** Overflow Flag (for signed)

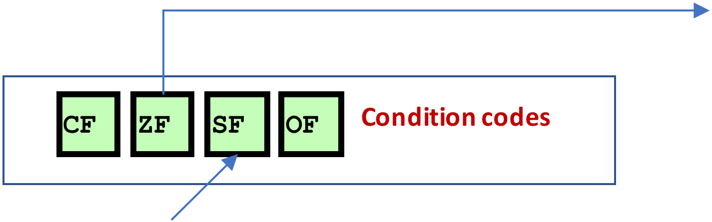
- Implicitly set (think of it as side effect) by arithmetic operations
  - Example: **`addq`** *Src,Dest* ↔ **`t = a+b`**
  - **CF set** if carry out from most significant bit (unsigned overflow)
  - **ZF set** if **`t == 0`**
  - **SF set** if **`t < 0`** (as signed)
  - **OF set** if two's-complement (signed) overflow
    `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

- Not set by **`leaq`** instruction

Condition codes and jumps

- `jg`, `jle`, etc. read condition codes
- named based on interpreting result of subtraction 0: equal; negative: less than; positive: greater than

Set 1 if result was zero.



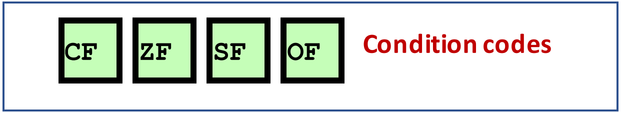
Set 1 if negative 0 if positive

JUMP instruction and their associated

[X86-guide]

Instruction	Description	Condition Code
jle	Jump if less or equal	(SF XOR OF) OR ZF
jg	Jump if greater (signed)	NOT (SF XOR 0F) & NOT ZF
je	Jump if equal	ZF

Why set the overflow flag



condition codes example (1)

```
movq $-10, %rax
movq $20, %rbx
subq %rax, %rbx // %rbx - %rax = 30
   // result > 0: %rbx was > %rax
jle foo // not taken; 30 > 0
```

jle	Jump if less or equal	(SF XOR OF) OR ZF

CF ZF SF OF **Condition codes**

https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf

condition codes example (2)

```
movq $10, %rax
movq $-20, %rbx
subq %rax, %rbx
jle foo
```

jle	Jump if less or equal	(SF XOR OF) OR ZF

CF ZF SF OF Condition codes

-20-10 = -30

Sign flag set

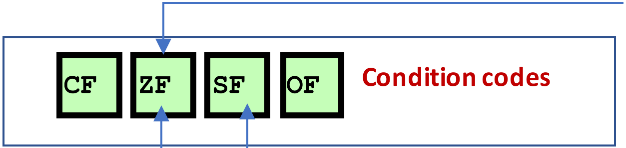
https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf

condition codes and cmpq

`cmp` does subtraction (but doesn't store result)

`cmp %rax, %rdi -> rdi - rax`

Set zero flag if equal



0101 (decimal 5)
AND 0011 (decimal 3)
= 0001 (decimal 1)

similarly `test` does bitwise-and

`testq %rax, %rax` — result is %rax

Set zero flag if result of bitwise and is zero

Also sets the SF flag with most significant bit of the result

Omitting the cmp

```
    movq $99, %r12 // register for x
start_loop:
    call foo
    subq $1, %r12
    cmpq $0, %r12
    // compute r12 - 0 + sets cond. codes
    jge  start_loop  // r12 >= 0?
                     // or result >= 0?
```

```
    movq $99, %r12 // register for x
start_loop:
    call foo
    subq $1, %r12
    // new r12 = old r12 - 1 + sets cond. codes
    jge  start_loop  // old r12 >= 1?
                     // or result >= 0?
```

condition codes example (3)

```
movq $-10, %rax
movq $20, %rbx
subq %rax, %rbx
jle  foo // not taken, %rbx - %rax > 0 -> %rbx
```

Jump is take in result in rbx is <= 0

Instruction	Description	Condition Code
jle	Jump if less or equal	(SF XOR OF) OR ZF

condition codes example (3)

```
movq $20, %rbx
addq $−20, %rbx
je    foo // taken, result is 0
          // x - y = 0 -> x = y
```

Instruction	Description	Condition Code
je	Jump if equal	ZF

what sets condition codes

- *most* instructions that compute something set condition codes
- some instructions only set condition codes:
  - `cmp` ~ `sub`
  - `test` ~ `and` (bitwise and)
    - `Example: testq %rax, %rax` — result is %rax
- some instructions don't change condition codes:
  - `lea`, `mov`
  - control flow: `jmp`, `call`, `ret`, etc.

Computed Jumps

Computed jumps

Instruction	Description
jmpq *%rax	Intel syntax: jmp RAX goto address RAX
jmpq *1000(%rax,%rbx,8)	Intel syntax: jmp QWORD PTR[RAX+RBX*8+1000] read address from memory at RAX + RBX * 8 + 1 // go to that address

Table look up. (picture).

From C to Assembly

goto

```
    for (...) {
      for (...) {
        if (thingAt(i, j)) {
          goto found;
        }
      }
    }
    printf("not found!\n");
    return;
found:
    printf("found!\n");
```

goto

```
for (...) {
  for (...) {
    if (thingAt(i, j)) {
      goto found;
    }
  }
}
printf("not found!\n");
return;
found:
printf("found!\n");
```

assembly:
```
jmp found
```

assembly:
```
found:
```

if-to-assembly (1)

```
if (b >= 42) {
    a += 10;
} else {
    a *= b;
}
```

if-to-assembly (1)

```
if (b >= 42) {
    a += 10;
} else {
    a *= b;
}
```

```
            if (b < 42) goto after_then;
            a += 10;
            goto after_else;
after_then: a *= b;
after_else:
```

Break this slide down further

if-to-assembly (2)

```
           if (b < 42) goto after_then;
           a += 10;
           goto after_else;
after_then: a *= b;  after_else:
```

```
// a is in %rax, b is in %rbx
    cmpq $42, %rbx    // computes rbx - 42
    jl after_then     // jump if rbx - 42 < 0
                      // AKA rbx < 42
    addq $10, %rax    // a += 10
    jmp after_else
after_then:
    imulq %rbx, %rax // rax = rax * rbx
after_else:
```

Make each line appear one at a time.

Quiz question

Which of the following represents the translations for the following c code:

```
// a is in %rax, b is in %rbx

if (b == 42) {
     a += 13;
} else {
   b -= 10;
}
```

https://create.kahoot.it/kahoots/my-kahoots

```
cmpq $42, %rbx
jne after_then
addq $13, %rax
jmp after_else
after_then:
  subq $10,  %rbx
after_else:
```

```
cmpq $42, %rbx
je after_then
addq $13, %rax
jmp after_else
after_then:
  subq $10,  %rbx
after_else:
```

```
cmpq $42, %rbx
jne after_then
  subq  %rbx, $10
jmp after_else
after_then:
  addq $13, %rax
after_else:
```

```
cmpq $42, %rbx
jmp after_else
addq $13, %rax
jne after_then
after_then:
  subq $10,  %rbx
after_else:
```

While-to-assembly: Step 1 Write C code with Goto's

```
while (x >= 0) {
    foo()
    x--;
}
```

C code

```
start_loop:
    if (x < 0) goto end_loop
    foo()
    x--;
    goto start_loop:
end_loop:
```

C code with gotos

Notice the sign change

Step (2) Translate each line to an assemble instruction

```
start_loop:
    if (x < 0) goto end_loop;
    foo()
    x--;
    goto start_loop:
end_loop:
```

C code with gotos

```
start_loop:
    cmpq $0, %r12
    jl end_loop // jump if r12 - 0 < 0
    call foo
    subq $1, %r12
    jmp start_loop
    end_loop:
```

Translate each line to it's corresponding assembly

while exercise

```
while (b < 10) { foo(); b += 1; }
```

Assume b is in callee-saved register %rbx.

```
// version A
start_loop:
  call foo
  addq $1, %rbx
  cmpq $10, %rbx
  jl start_loop
```

```
// version B
start_loop:
  cmpq $10, %rbx
  jge end_loop
  call foo
  addq $1, %rbx
  jmp start_loop
end_loop:
```

```
// version C
start_loop:
  movq $10, %rax
  subq %rbx, %rax
  jle end_loop
  call foo
  addq $1, %rbx
  jmp start_loop
end_loop:
```

Which are correct assembly translations?

While to assembly (Solution)

```
while (b < 10) {
    foo();
    b += 1;
}
```

```
start_loop: if (b < 10) goto end_loop;
            foo();
            b += 1;
            goto start_loop;
end_loop:
```

While to assembly solution

```
start_loop: if (b < 10) goto end_loop;
            foo();
            b += 1;
            goto start_loop;
end_loop:
```

```
start_loop:
  cmpq $10, %rbx
  jge end_loop
  call foo
  addq $1, %rbx
  jmp start_loop
end_loop:
```

while — levels of optimization

```
while (b < 10) { foo(); b += 1; }
```

```
start_loop:
   cmpq $10, %rbx
   jge end_loop
   call foo
   addq $1, %rbx
   jmp start_loop
end_loop:
    ...
    ...
    ...
    ...
```

```
   cmpq $10, %rbx
   jge end_loop
start_loop:
   call foo
   addq $1, %rbx
   cmpq $10, %rbx
   jne start_loop
end_loop:
    ...
    ...
    ...
```

```
   cmpq $10, %rbx
   jge end_loop
   movq $10, %rax
   subq %rbx, %rax
   movq %rax, %rbx
start_loop:
   call foo
   decq %rbx
   jne start_loop
   movq $10, %rbx
end_loop:
```

Think about this optimization

Some Arithmetic Operations

- Two Operand Instructions:

Format		***Computation***
`addq`	*Src,Dest*	Dest = Dest + Src
`subq`	*Src,Dest*	Dest = Dest – Src
`imulq`	*Src,Dest*	Dest = Dest * Src

- Watch out for argument order!

- See book for more instructions

x86-64 calling convention example

```
int foo(int x, int y, int z) { return 42; }
...
    foo(1, 2, 3);
...
```

```
...
    // foo(1, 2, 3)
    movl $1, %edi
    movl $2, %esi
    movl $3, %edx
    call foo  // call pushes address of next instruction
              // then jumps to foo
...
foo:
    movl $42, %eax
    ret
```

Key Registers Review

%rax	Return value	%r8	Argument #5
%rbx	Callee saved	%r9	Argument #6
%rcx	Argument #4	%r10	Caller saved
%rdx	Argument #3	%r11	Caller Saved
%rsi	Argument #2	%r12	Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved	%r15	Callee saved

x86-64 calling convention example

```
int foo(int x, int y, int z) { return 42; }
...
    foo(1, 2, 3);
...
```

```
...
    // foo(1, 2, 3)
    movl $1, %edi
    movl $2, %esi
    movl $3, %edx
    call foo  // call pushes address of next instruction
              // then jumps to foo
...
foo:
    movl $42, %eax
    ret
```

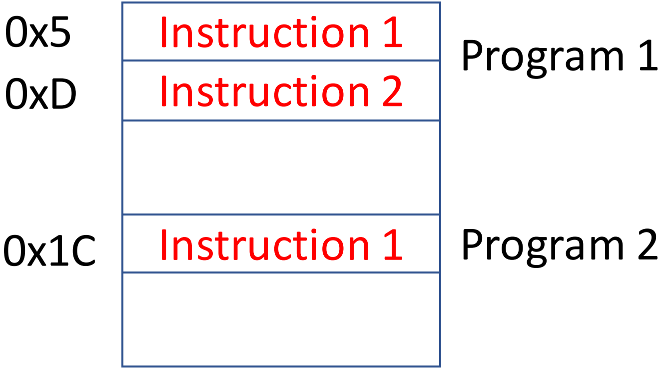
call/ret

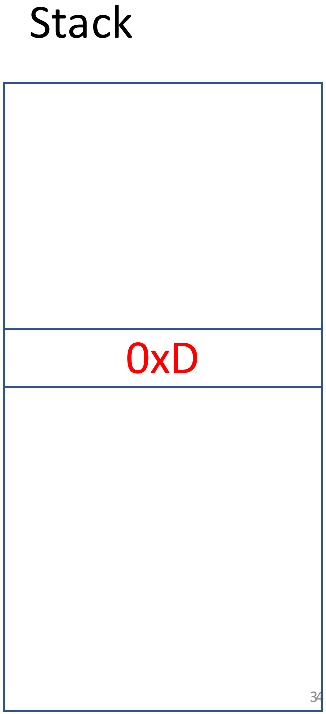
call:

push address of next instruction on the stack

ret:

pop address from stack; jump





callee-saved registers

functions must preserve these

%rax	Return value	%r8	Argument #5
%rbx	Callee saved	%r9	Argument #6
%rcx	Argument #4	%r10	Caller saved
%rdx	Argument #3	%r11	Caller Saved
%rsi	Argument #2	%r12	Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved	%r15	Callee saved

`%rsp` (stack pointer),
`%rbx`, (ordinary register) `%rbp` (frame pointer – the compiler does use frame pointers)

`%r12-%r15 (ordinary callee registers)`

Question

```
pushq $0x1
pushq $0x2
addq $0x3, 8(%rsp)
popq %rax
popq %rbx
```

What is value of `%rax` and `%rbx` after this?

a. `%rax = 0x2, %rbx = 0x4`

b. `%rax = 0x5, %rbx = 0x1`

c. `%rax = 0x2, %rbx = 0x1`

d. the snippet has invalid syntax or will crash

Question

00	00	00	00	00	00	00	01
00	00	00	00	00	00	00	02

```
pushq $0x1
pushq $0x2
addq $0x3, 8(%rsp)
popq %rax
popq %rbx
```

What is value of `%rax` and `%rbx` after this?

a. `%rax = 0x2, %rbx = 0x4`

b. `%rax = 0x5, %rbx = 0x1`

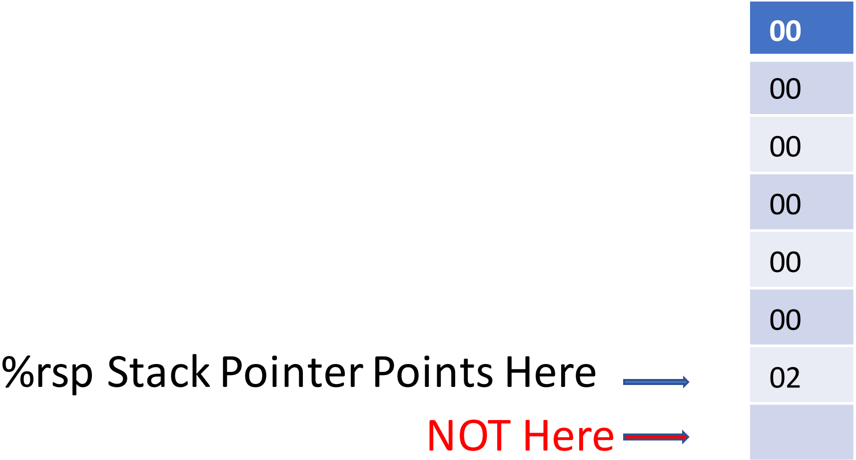
c. `%rax = 0x2, %rbx = 0x1`

d. the snippet has invalid syntax or will crash

00
00
00
00
01
00
00
00
00
00
00
00
02

Pop reads from where the stack pointer is now

- %rsp points to the most recently pushed value, not to the next unused stack address.



C

C Data Types

For machines that you this course:

type	size (bytes)
`char`	1
`short`	2
`int`	4
`long`	8
`float`	4
`double`	8
`void *`	8
`anything` `*`	8

O

Truth

~~`Bool`~~ There is no Boolean type

`x == 4` is an `int`

`1` if true; `0` if false

The only values that are false in c is 0 and null pointer

Everything else is true

`0` including null pointers — `0` cast to a pointer

short-circuit (| |)

```c
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
     printf("> %d\n", zero() || one());
     printf("> %d\n", one() || zero());
     return 0;
}
```

```
zero()
one()
> 1
one()
> 1
```

Lazy evaluation

short-circuit (&&)

```
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
     printf("> %d\n", zero() && one());
     printf("> %d\n", one() && zero());
     return 0;
}
```

```
zero()
> 0
one()
zero()
> 0
```

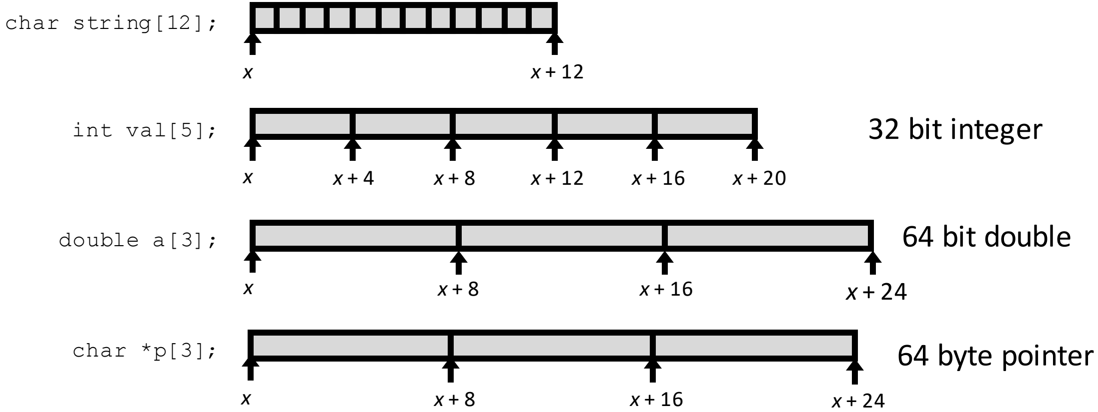
Lazy evaluation

Pointer Arithmetic & Arrays

Array Allocation

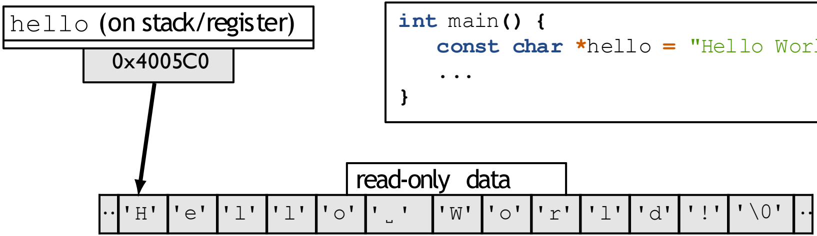
- Basic Principle

  T **A[*L*];**
  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **sizeof**(*T*) bytes in memory

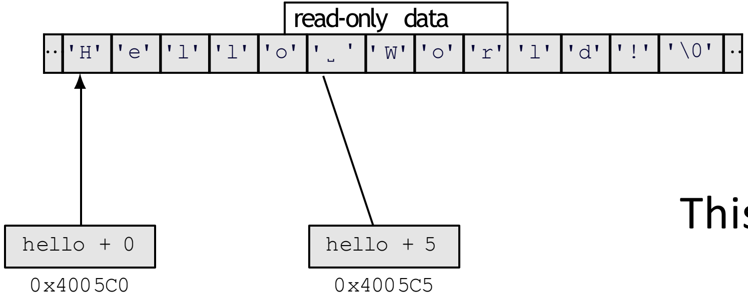


strings in C

```c
int main() {
    const char *hello = "Hello World!";
    ...
}
```



pointer arithmetic



`*(hello + 0)` is `'H'`

`hello[0]` is `'H'`

`*(hello + 5)` is `'␣'`

`hello[5]` is `'␣'`

This is a valid C

arrays of non-bytes

`array[2]` and `*(array + 2)` still the same

```
int numbers[4] = {10, 11, 12, 13};
int *pointer;
pointer = numbers;
*pointer = 20;   // numbers[0] = 20;
pointer = pointer + 2;
/* adds 8 (2 ints) to address */
*pointer = 30;   // numbers[2] = 30;
// numbers is 20, 11, 30, 13
```

Arrays: not quite pointers

```
int array[100];
int *pointer;
```

Legal: `pointer = array;`

Same As: `pointer = &(array[0]);`

~~Illegal: array = pointer;~~

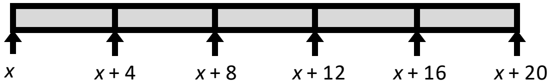
arrays: not quite pointers (2)

```
int array[100];
int *pointer = array;
```

```
sizeof(array) == 400
```

Size of all elements in the array

int val[5];



32 bit integer

```
sizeof(pointer) == 8
```

size of address

exercise

```
char foo[4] = "foo";
      // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```

Final value of `foo`?

A. "fao"
B. "zao"
C. "baz"
D. "bao"

exercise

```
char foo[4] = "foo";
      // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```

Final value of `foo`?

A. "fao"
B. "zao"
C. "baz"
D. "bao"

exercise explanation

```
char foo[4] = "foo";
     // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```

better style: `*pointer = 'z';`

better style: `foo[1] = 'a';`

foo (on stack)			
'f'	'o'	'o'	'\0'

`foo + 1 == &foo[0] + 1`

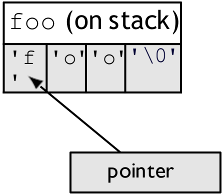
pointer

exercise explanation

```
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```

better style: `*pointer = 'z';`

better style: `foo[1] = 'a';`



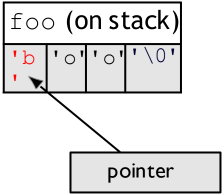

```
foo + 1 == &foo[0] + 1
```

exercise explanation

```
char foo[4] = "foo";
     // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```

better style: `*pointer = 'z';`

better style: `foo[1] = 'a';`



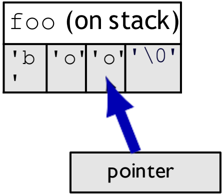
`foo + 1 == &foo[0] + 1`

exercise explanation

```
char foo[4] = "foo";
      // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```

better style: `*pointer = 'z';`

better style: `foo[1] = 'a';`



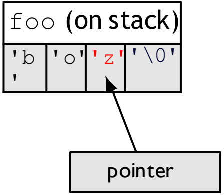
`foo + 1 == &foo[0] + 1`

exercise explanation

```c
char foo[4] = "foo";
      // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```

better style: `*pointer = 'z';`

better style: `foo[1] = 'a';`



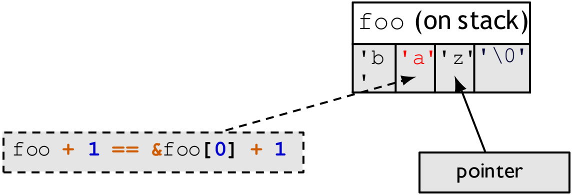

```c
foo + 1 == &foo[0] + 1
```

exercise explanation

```
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```

better style: `*pointer = 'z';`

better style: `foo[1] = 'a';`

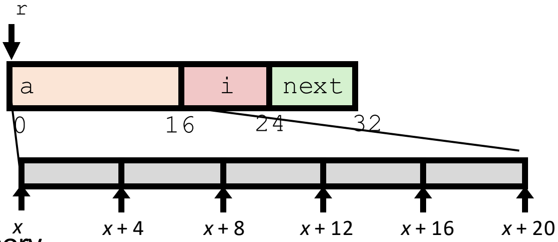


What is a struct

You can think of a struct as a class without methods.

Structure Representation

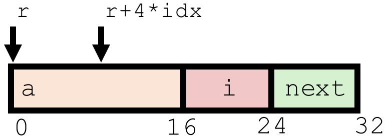
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Structure represented as block of memory
  - **Big enough to hold all of the fields**
- Fields ordered according to declaration
  - **Even if another ordering could yield a more compact representation**
- Compiler determines overall size + positions of fields
  - **Machine-level program has no understanding of the structures in the source code**

Generating Pointer to Structure Member

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Compute as **`r + 4*idx`**

```
int *get_ap(struct rec *r, size_t idx)
{
  return &r->a[idx];
}
```

```
  # r in %rdi, idx in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```

struct

```c
struct rational {
    int numerator;
    int denominator;
};
// ...
struct rational two_and_a_half;
two_and_a_half.numerator = 5;
two_and_a_half.denominator = 2;
struct rational *pointer = &two_and_a_half;
printf("%d/%d\n",
       pointer->numerator,
       pointer->denominator);
```

struct

Struct are class without methods

```
struct rational {
    int numerator;
    int denominator;
};
// ...
struct rational two_and_a_half;
two_and_a_half.numerator = 5;
two_and_a_half.denominator = 2;
struct rational *pointer = &two_and_a_half;
printf("%d/%d\n",
       pointer->numerator,
       pointer->denominator);
```

The key word struct is mandatory

typedef struct (1)

Define a new name for a type

```
typedef struct rationals {
    int numerator;
    int denominator;
}rational;
```

```
// ...
rational two_and_a_half;
two_and_a_half.numerator = 5;
two_and_a_half.denominator = 2;
rational *pointer = &two_and_a_half;
printf("%d/%d\n",
           pointer->numerator,
           pointer->denominator);
```

typedef struct (2)

```c
struct other_name_for_rational {
    int numerator;
    int denominator;
};
typedef struct other_name_for_rational rational;
// same as:
typedef struct other_name_for_rational{
    int numerator;
    int denominator;
} rational;
```

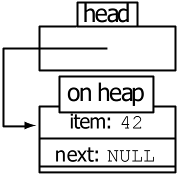
typedef struct (2)

```
struct other_name_for_rational {
    int numerator;
    int denominator;
};
typedef struct other_name_for_rational rational;
// same as:
typedef struct other_name_for_rational{
    int numerator;
    int denominator;
} rational;
// almost the same as:
typedef struct {
    int numerator;
    int denominator;
} rational;
```

linked lists / dynamic allocation

```c
typedef struct list_t {
    int item;
    struct list_t *next;
} list;
// ...
```

```c
list* head = malloc(sizeof(list));
    /* C++: new list; */
head->item = 42;
head->next = NULL;
// ...
free(head);
    /* C++: delete list */
```



dynamic arrays

```
int *array = malloc(sizeof(int)*100);
  // C++: new int[100]
for (i = 0; i < 100; ++i) {
    array[i] = i;
}
// ...
free(array); // C++: delete[] array
```