Assembly part 2 / C part 1

last time

linking extras:

- different kinds of relocations — addresses versus offset to addresses
- dynamic linking (briefly)

AT&T syntax

- destination last
- `O(B, I, S)` — $B + I \times S + O$
- `jmp *`

`lea` — `mov`, but don't do memory access

if/do-while to assembly

condition codes — last arithmetic result

reminder: quiz

post-quiz — after this lecture

pre-quiz — before next lecture

demo

condition codes

x86 has condition codes

set by (almost) all arithmetic instructions

- addq, subq, imulq, etc.

store info about last arithmetic result

- was it zero? was it negative? etc.

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

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
```

condition codes and cmpq

"last arithmetic result"???

then what is cmp, etc.?

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

similar `test` does bitwise-and

`testq %rax, %rax` — result is %rax

condition codes example (2)

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

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

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

what sets condition codes

most instructions that compute something set condition codes

some instructions only set condition codes:

- `cmp` ∼ `sub`
- `test` ∼ `and` (bitwise and — later)
- `testq %rax, %rax` — result is %rax

some instructions don't change condition codes:

- `lea`, `mov`
- control flow: `jmp`, `call`, `ret`, `jle`, etc.

condition codes examples (4)

```
movq $20, %rbx
addq $−20, %rbx // result is 0
movq $1, %rax // irrelevant
je   foo // taken, result is 0
```

while-to-assembly (1)

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

while-to-assembly (1)

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

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

while-to-assembly (2)

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

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

while exercise

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

Assume b is in callee-saved register %rbx. Which are correct assembly translations?

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

while to assembly (1)

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

while to assembly (1)

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

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

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:
    ...
    ...
    ...
```

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:
```

condition codes: closer look

x86 condition codes:

- `ZF` ("zero flag") — was result zero? (sub/cmp: equal)
- `SF` ("sign flag") — was result negative? (sub/cmp: less)

(and some more, e.g. to handle overflow)

GDB: part of "eflags" register

set by cmp, test, arithmetic

condition codes example (2)

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

%rbx - %rax = 30 — SF = 0 (not negative), ZF = 0 (not zero)

condition codes examples (4)

```
movq $20, %rbx
addq $−20, %rbx // result is 0
movq $1, %rax // irrelevant
je   foo // taken, result is 0
```

20 + -20 = 0 — SF = 0 (not negative), ZF = 1 (zero)

condition codes: closer look

x86 condition codes:

- `ZF` (“zero flag”) — was result zero? (sub/cmp: equal)
- `SF` (“sign flag”) — was result negative? (sub/cmp: less)
- `CF` (“carry flag”) — did computation overflow (as unsigned)?
- `OF` (“overflow flag”) — did computation overflow (as signed)?
- (and one more)

GDB: part of “eflags” register

set by cmp, test, arithmetic

closer look: condition codes (1)

```
  movq $−10, %rax
  movq $20, %rbx
  cmpq %rax, %rbx
// result = %rbx - %rax = 30
```

as signed: $20 - (-10) = 30$

as unsigned: $20 - (2^{64} - 10) = \cancel{-2^{64} - 30}\ 30$ (overflow!)

$\mathrm{ZF} = 0$ (false)   not zero   rax and rbx not equal

closer look: condition codes (1)

```
    movq $-10, %rax
    movq $20, %rbx
    cmpq %rax, %rbx
// result = %rbx - %rax = 30
```

as signed: $20 - (-10) = 30$

as unsigned: $20 - (2^{64} - 10) = \cancel{-2^{64} - 30}\ 30$ (overflow!)

$\mathtt{ZF} = 0$ (false) not zero rax and rbx not equal

closer look: condition codes (1)

```
  movq $−10, %rax
  movq $20, %rbx
  cmpq %rax, %rbx
// result = %rbx - %rax = 30
```

as signed: $20 - (-10) = 30$

as unsigned: $20 - (2^{64} - 10) = \xcancel{-2^{64} - 30}\ 30$ (overflow!)

$\mathrm{ZF} = 0$ (false)	not zero	rax and rbx not equal
$\mathrm{SF} = 0$ (false)	not negative	rax <= rbx

closer look: condition codes (1)

```
  movq $-10, %rax
  movq $20, %rbx
  cmpq %rax, %rbx
// result = %rbx - %rax = 30
```

as signed: $20 - (-10) = 30$

as unsigned: $20 - (2^{64} - 10) = \cancel{-2^{64} - 30}\ 30$ (overflow!)

$\mathtt{ZF} = 0$ (false)	not zero	rax and rbx not equal
$\mathtt{SF} = 0$ (false)	not negative	rax <= rbx
$\mathtt{OF} = 0$ (false)	no overflow as signed	correct for signed

closer look: condition codes (1)

```
  movq $−10, %rax
  movq $20, %rbx
  cmpq %rax, %rbx
// result = %rbx - %rax = 30
```

as signed: $20 - (-10) = 30$

as unsigned: $20 - (2^{64} - 10) = \xcancel{-2^{64} - 30}\ 30$ (overflow!)

$\mathtt{ZF} = 0$ (false)	not zero	rax and rbx not equal
$\mathtt{SF} = 0$ (false)	not negative	rax $<=$ rbx
$\mathtt{OF} = 0$ (false)	no overflow as signed	correct for signed
$\mathtt{CF} = 1$ (true)	overflow as unsigned	incorrect for unsigned

exercise: condition codes (2)

```
  // 2^63 - 1
  movq $0x7FFFFFFFFFFFFFFF, %rax
  // 2^63 (unsigned); -2**63 (signed)
  movq $0x8000000000000000, %rbx
  cmpq %rax, %rbx
// result = %rbx - %rax
```

ZF = ?

SF = ?

OF = ?

CF = ?

closer look: condition codes (2)

```
  // 2**63 - 1
  movq $0x7FFFFFFFFFFFFFFF, %rax
  // 2**63 (unsigned); -2**63 (signed)
  movq $0x8000000000000000, %rbx
  cmpq %rax, %rbx
// result = %rbx - %rax
```

as signed: $-2^{63} - \left(2^{63} - 1\right) = \cancel{-2^{64} + 1}\ 1$ (overflow)

as unsigned: $2^{63} - \left(2^{63} - 1\right) = 1$

$\mathrm{ZF} = 0$ (false) not zero rax and rbx not equal

closer look: condition codes (2)

```
  // 2**63 - 1
  movq $0x7FFFFFFFFFFFFFFF, %rax
  // 2**63 (unsigned); -2**63 (signed)
  movq $0x8000000000000000, %rbx
  cmpq %rax, %rbx
// result = %rbx - %rax
```

as signed: $-2^{63} - \left(2^{63} - 1\right) = \xcancel{-2^{64} + 1}\ 1$ (overflow)

as unsigned: $2^{63} - \left(2^{63} - 1\right) = 1$

$\mathtt{ZF} = 0$ (false)   not zero   rax and rbx not equal

closer look: condition codes (2)

```
  // 2**63 - 1
  movq $0x7FFFFFFFFFFFFFFF, %rax
  // 2**63 (unsigned); -2**63 (signed)
  movq $0x8000000000000000, %rbx
  cmpq %rax, %rbx
// result = %rbx - %rax
```

as signed: $-2^{63} - \left(2^{63} - 1\right) = \xcancel{-2^{64} + 1}\ 1$ (overflow)

as unsigned: $2^{63} - \left(2^{63} - 1\right) = 1$

$\mathrm{ZF} = 0$ (false)	not zero	rax and rbx not equal
$\mathrm{SF} = 0$ (false)	not negative	rax <= rbx (if correct)

closer look: condition codes (2)

```
  // 2**63 - 1
  movq $0x7FFFFFFFFFFFFFFF, %rax
  // 2**63 (unsigned); -2**63 (signed)
  movq $0x8000000000000000, %rbx
  cmpq %rax, %rbx
// result = %rbx - %rax
```

as signed: $-2^{63} - \left(2^{63} - 1\right) = \xcancel{-2^{64} + 1}\ 1$ (overflow)

as unsigned: $2^{63} - \left(2^{63} - 1\right) = 1$

$\mathtt{ZF} = 0$ (false)	not zero	rax and rbx not equal
$\mathtt{SF} = 0$ (false)	not negative	rax $<=$ rbx (if correct)
$\mathtt{OF} = 1$ (true)	overflow as signed	incorrect for signed

closer look: condition codes (2)

```
  // 2**63 - 1
  movq $0x7FFFFFFFFFFFFFFF, %rax
  // 2**63 (unsigned); -2**63 (signed)
  movq $0x8000000000000000, %rbx
  cmpq %rax, %rbx
// result = %rbx - %rax
```

as signed: $-2^{63} - \left(2^{63} - 1\right) = \xcancel{-2^{64} + 1}\ 1$ (overflow)

as unsigned: $2^{63} - \left(2^{63} - 1\right) = 1$

$\mathtt{ZF} = 0$ (false)	not zero	rax and rbx not equal
$\mathtt{SF} = 0$ (false)	not negative	rax $<=$ rbx (if correct)
$\mathtt{OF} = 1$ (true)	overflow as signed	incorrect for signed
$\mathtt{CF} = 0$ (false)	no overflow as unsigned	correct for unsigned

closer look: condition codes (3)

```
  movq  $−1, %rax
  addq  $−2, %rax
// result = -3
```

as signed: $-1 + (-2) = -3$

as unsigned: $(2^{64} - 1) + (2^{64} - 2) = \cancel{2^{65} - 3}\ 2^{64} - 3$ (overflow)

$\mathtt{ZF} = 0$ (false) not zero result not zero

closer look: condition codes (3)

```
    movq  $−1, %rax
    addq  $−2, %rax
// result = -3
```

as signed: $-1 + (-2) = -3$

as unsigned: $(2^{64} - 1) + (2^{64} - 2) = \xcancel{2^{65} - 3}\ 2^{64} - 3$ (overflow)

$\mathtt{ZF} = 0$ (false)	not zero	result not zero
$\mathtt{SF} = 1$ (true)	negative	result is negative
$\mathtt{OF} = 0$ (false)	no overflow as signed	correct for signed
$\mathtt{CF} = 1$ (true)	overflow as unsigned	incorrect for unsigned

C Data Types

Varies between machines(!). For this course:

type	size (bytes)
`char`	1
`short`	2
`int`	4
`long`	8

C Data Types

Varies between machines(!). For this course:

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

C Data Types

Varies between machines(!). For this course:

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

truth

~~`bool`~~

truth

~~`bool`~~

`x == 4` is an `int`

`1` if true; `0` if false

false values in C

`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
```

OR	**false**	**true**
false	false	true
true	true	true

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
```

OR	**false**	**true**
false	false	true
true	true	true

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
```

OR	**false**	**true**
false	false	true
true	true	true

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
```

OR	**false**	**true**
false	false	true
true	true	true

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
```

OR	**false**	**true**
false	false	true
true	true	true

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
```

AND	**false**	**true**
false	false	false
true	false	true

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
```

AND	**false**	**true**
false	false	false
true	false	true

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

AND	**false**	**true**
false	false	false
true	false	true

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
```

AND	**false**	**true**
false	false	false
true	false	true

short-circuit (&&)

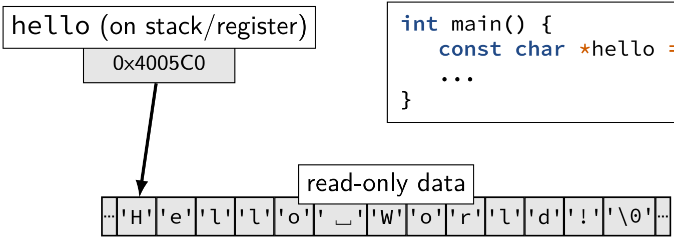
```c
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() && one());
    printf(">␣%d\n", one() && zero());
    return 0;
}
```

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

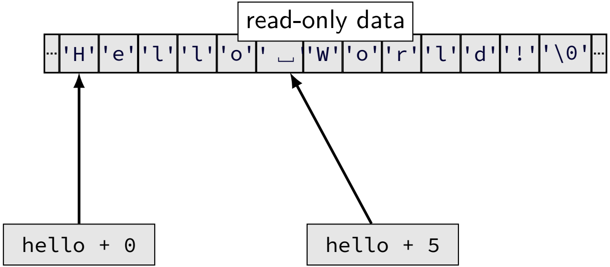
AND	**false**	**true**
false	false	false
true	false	true

strings in C

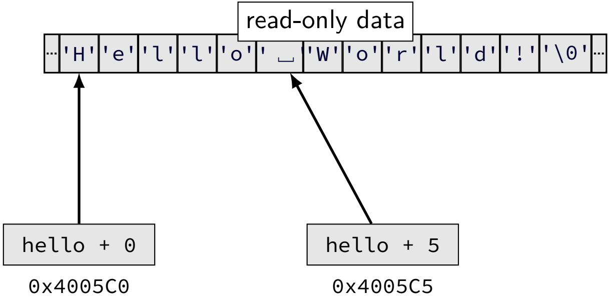
```
int main() {
   const char *hello = "Hello World!";
   ...
}
```



pointer arithmetic

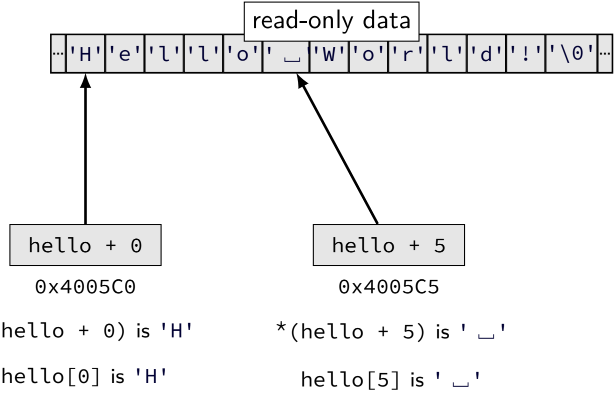


pointer arithmetic



`*(hello + 0)` is `'H'`   `*(hello + 5)` is `' ␣'`

pointer arithmetic


read-only data
... 'H' 'e' 'l' 'l' 'o' '␣' 'W' 'o' 'r' 'l' 'd' '!' '\0' ...
hello + 0
0x4005C0
hello + 5
0x4005C5

*(hello + 0) is 'H'

hello[0] is 'H'

*(hello + 5) is '␣'

hello[5] is '␣'

arrays and pointers

`*(foo + bar)` exactly the same as `foo[bar]`

arrays ‘decay’ into pointers

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 of non-bytes

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

```c
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
```

a note on precedence

`&foo[1]` is the same as `&(foo[1])` (*not* `(&foo)[1]`)

`*foo[0]` is the same as `*(foo[0])` (*not* `(*foo)[0]`)

`*foo++` is the same as `*(foo++)` (*not* `(*foo)++`)

exercise

```c
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"

E. something else/crash

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"

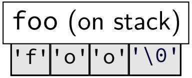
C. "baz"

D. "bao"

E. something else/crash

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';
```

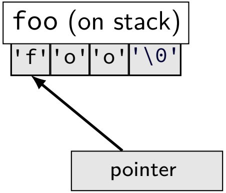


foo (on stack)

'f'	'o'	'o'	'\0'

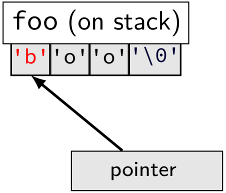
exercise explanation

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



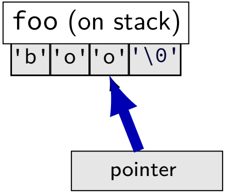
exercise explanation

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



exercise explanation

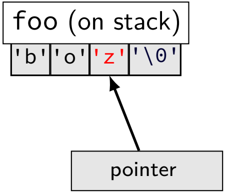
```c
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```



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';`

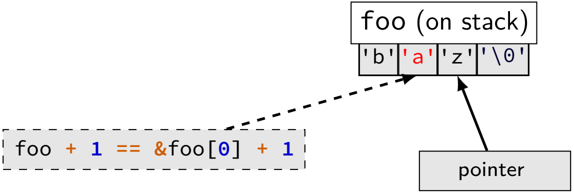


exercise explanation

```
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)

'b' 'a' 'z' '\0'

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

pointer

backup slides

example: C that is not C++

valid C and invalid C++:

```
char *str = malloc(100);
```

valid C and valid C++:

```
char *str = (char *) malloc(100);
```

valid C and invalid C++:

```
int class = 1;
```