Assembly
Compilation Pipeline
Selected C Topics

January 1, 2023

last lecture topics

LEA = **L**oad **E**ffective **A**ddress

- effective address = address computed in middle of running instruction
- computes address, places in destination register
- processor doesn't check/care if "address" is valid in memory
- address compute logic often used for arithmetics

condition codes = 1-bit register flags, describe "last arithmetic"

- Zero Flag ZF = was zero?; Sign Flag SF = was negative?
- Overflow Flag OF, Carry Flag CF
- cmp = same as sub but only sets condition codes (result not stored)
- jXX - named after comparing to 0

converting control flow to assembly

- if
- continue today: while, switch

quiz demo

while-to-assembly (1)

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

---

while-to-assembly (1)

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

Re-write C code with *goto*'s:

```
start_loop:
    if (x < 0) goto end_loop; // (x >= 0) not true
    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
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:
    ...
    ...
    ...
    ...
```

```
// merge jge and jmp
  cmpq $10, %rbx
  jge end_loop // >=
start_loop:
  call foo
  addq $1, %rbx
  cmpq $10, %rbx
  jne start_loop//!=
end_loop:
    ...
    ...
// jge end_loop
// now outside loop
// "prefix" cost
```

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

```
// merge jge and jmp
  cmpq $10, %rbx
  jge end_loop // >=
start_loop:
  call foo
  addq $1, %rbx
  cmpq $10, %rbx
  jne start_loop//!=
end_loop:
    ...
    ...
// jge end_loop
// now outside loop
// "prefix" cost
```

```
  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:
    ...
// count down to 0
```

compiling switches (1)

```
switch (a) {
    case 1: ...; break;
    case 2: ...; break;
    ...
    default: ...
}
```

```
    // same as if statement?
    cmpq $1, %rax
    je code_for_1
    cmpq $2, %rax
    je code_for_2
    cmpq $3, %rax
    je code_for_3
    ...
    jmp code_for_default
    // Note: lots of cmpq's!
```

compiling switches (2)

```
switch (a) {
    case 1: ...; break;
    case 2: ...; break;
    ...
    case 100: ...; break;
    default: ...
}
```

```
    // binary search, less cmpq's
    cmpq $50, %rax
    jl code_for_less_than_50
    cmpq $75, %rax
    jl code_for_50_to_75
    ...
code_for_less_than_50:
    cmpq $25, %rax
    jl less_than_25_cases
    ...
```

compiling switches (3a)

```
switch (a) {
    case 1: ...; break;
    case 2: ...; break;
    ...
    case 100: ...; break;
    default: ...
}
```

```
// jump table
cmpq $100, %rax
jg code_for_default // >100
cmpq $1, %rax
jl code_for_default // <1
jmp *table-8(,%rax,8)
// displacement = table-8
```

```
table:
  // not instructions
  // .quad = 64-bit (4 x 16) consta
  .quad code_for_1
  .quad code_for_2
  .quad code_for_3
  .quad code_for_4
    ...
```

compiling switches (3b)

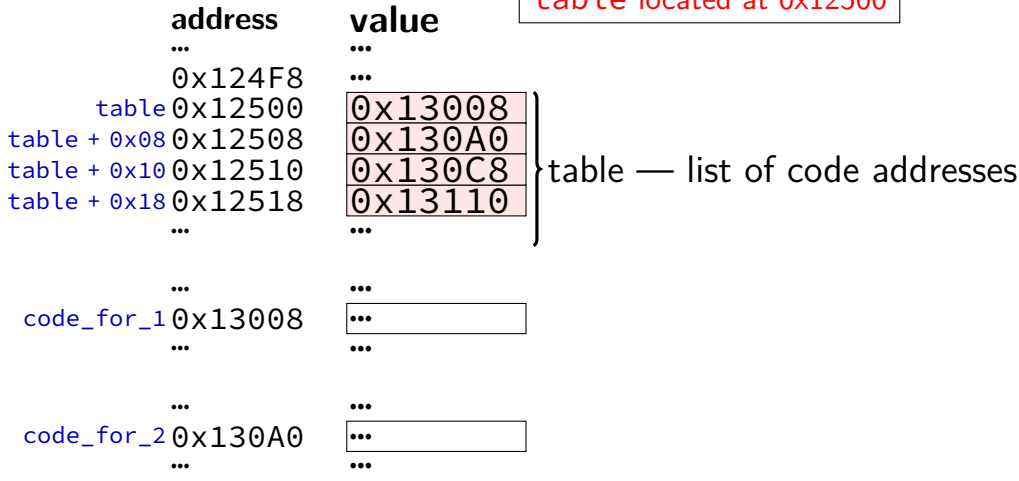
```
jmp *table-8(,%rax,8)
```

suppose RAX = 2,
`table` located at 0x12500

compiling switches (3b)

```
jmp *table-8(,%rax,8)
```

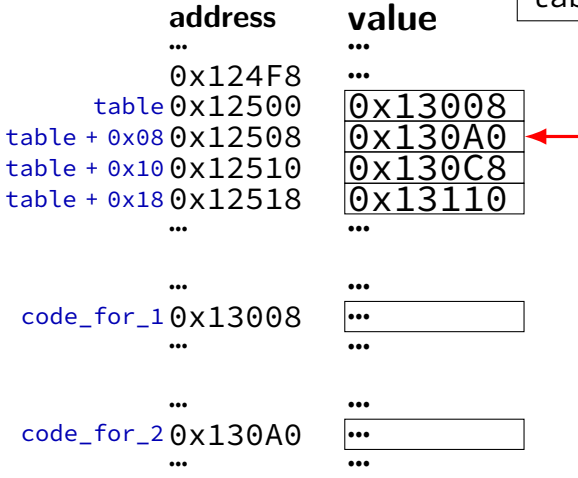
suppose RAX = 2,
`table` located at 0x12500



compiling switches (3b)

```
jmp *table-8(,%rax,8)
```

suppose RAX = 2,
`table` located at 0x12500

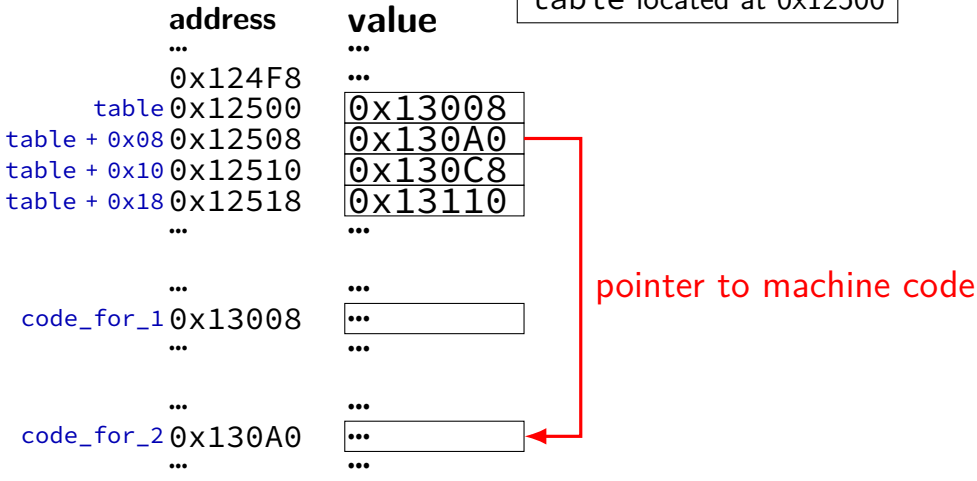


$(\text{table} - 8) + \text{rax} \times 8 =$
$\texttt{0x124F8} + \texttt{0x10} = \texttt{0x12508}$

compiling switches (3b)

```
jmp *table-8(,%rax,8)
```

suppose RAX = 2,
`table` located at 0x12500

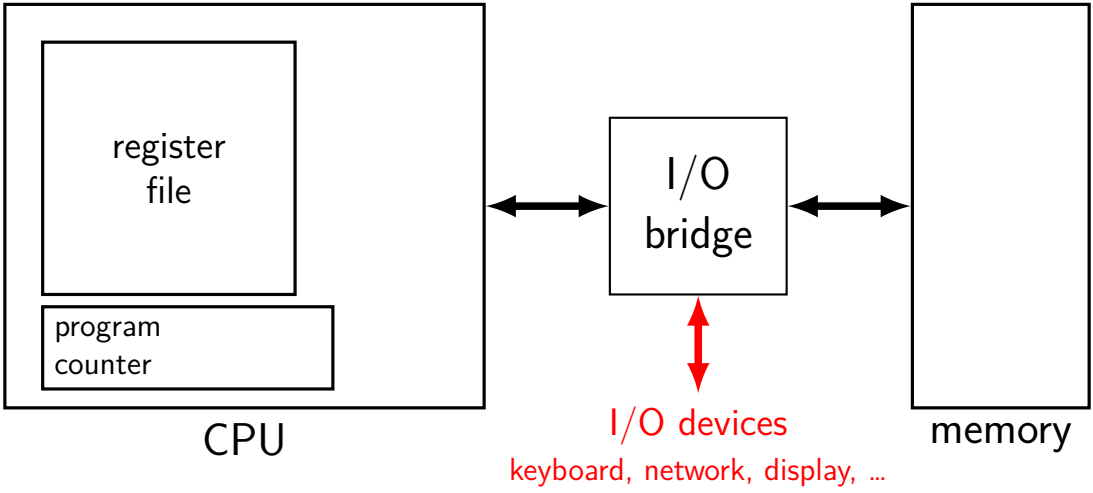


computed jumps

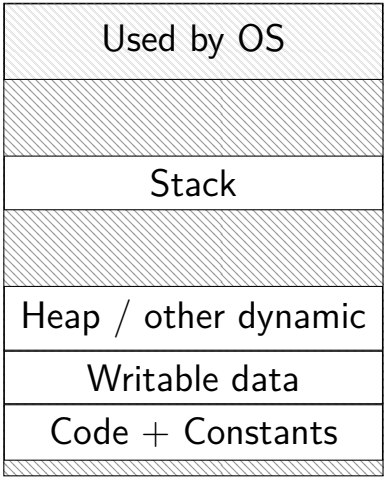
Idea: use pointers instead of value from memory

```
  cmpq $100, %rax
  jg code_for_default // >100
  cmpq $1, %rax
  jl code_for_default // <1
  // jump to memory[table + rax * 8]
    // table of pointers to instructions
  jmp *table(,%rax,8)
  // intel: jmp QWORD PTR[rax*8 + table]
  ...
table:
  .quad code_for_1
  .quad code_for_2
  .quad code_for_3
```

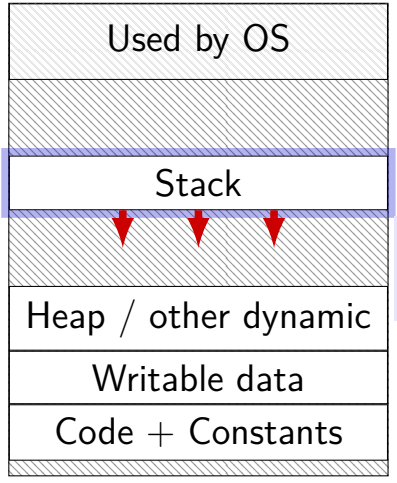
Reminder: processors and memory and I/O



program memory (x86-64 Linux)



program memory (x86-64 Linux)



0xFFFF FFFF FFFF FFFF

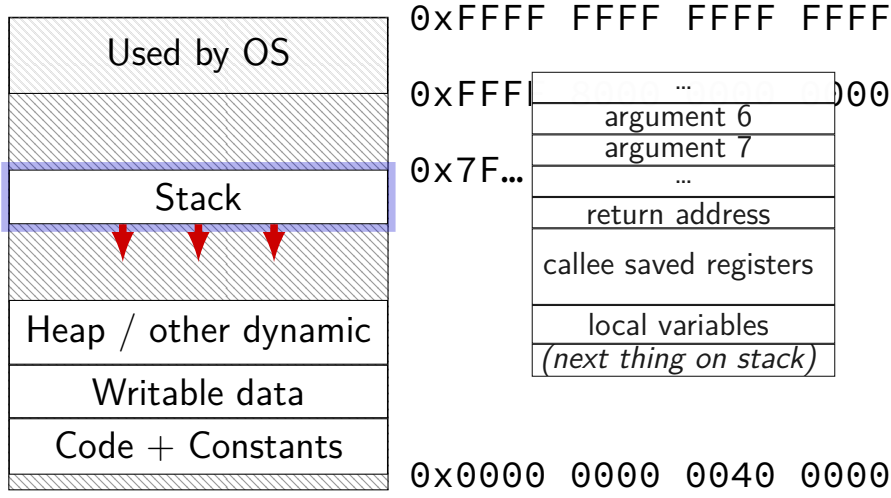
0xFFFF 8000 0000 0000

0x7F…

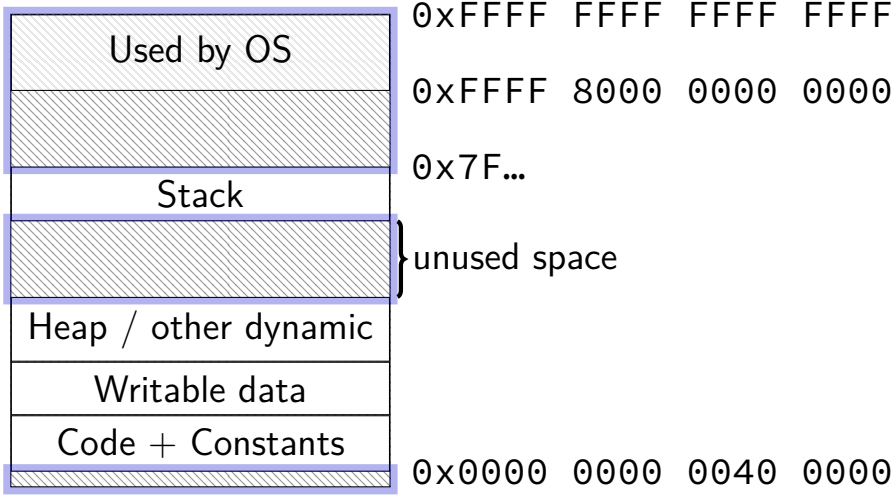
stack grows down
"top" has smallest address

0x0000 0000 0040 0000

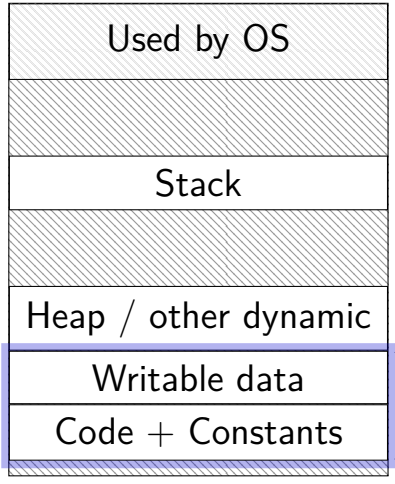
program memory (x86-64 Linux)



program memory (x86-64 Linux)



program memory (x86-64 Linux)



0xFFFF FFFF FFFF FFFF

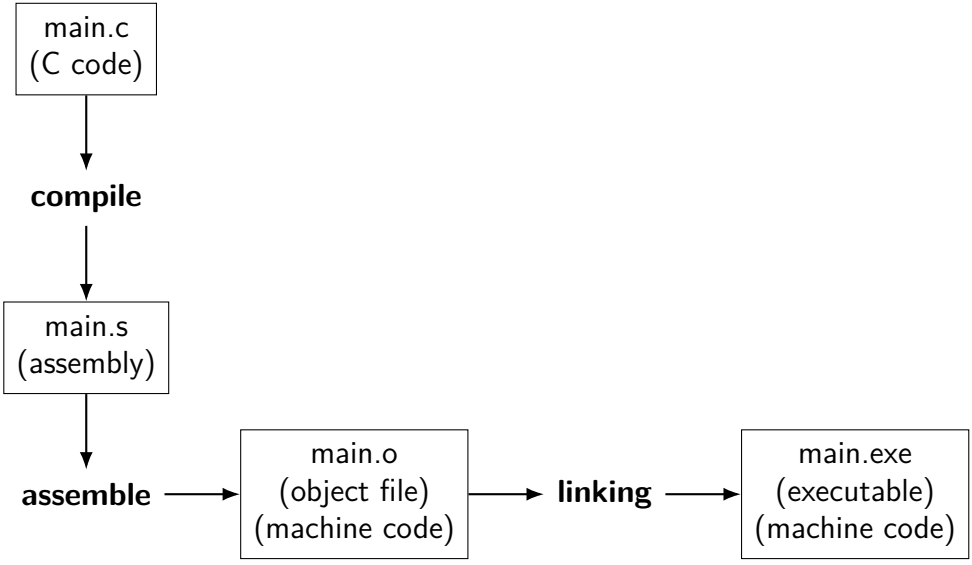
0xFFFF 8000 0000 0000

0x7F…

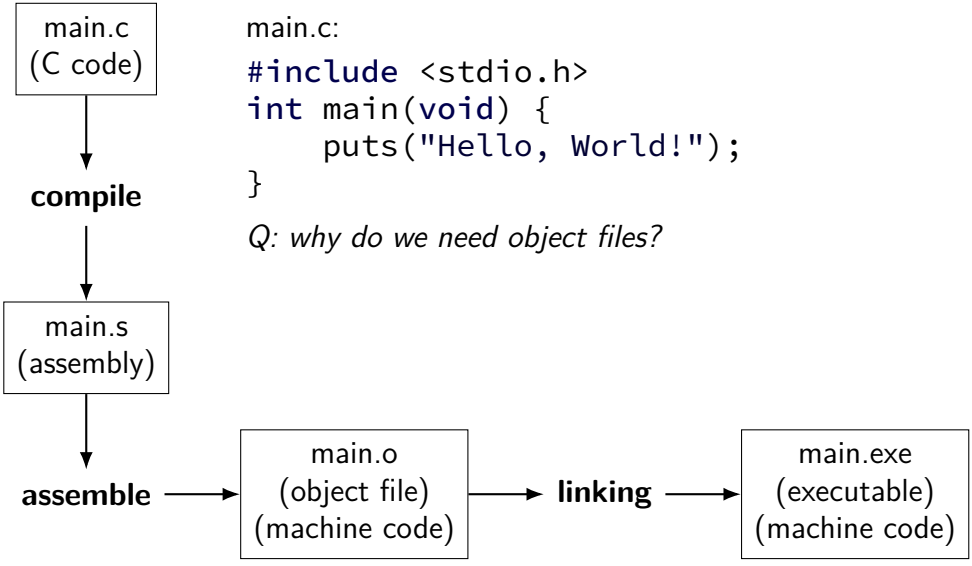
loaded from executable file

0x0000 0000 0040 0000

compilation pipeline



compilation pipeline

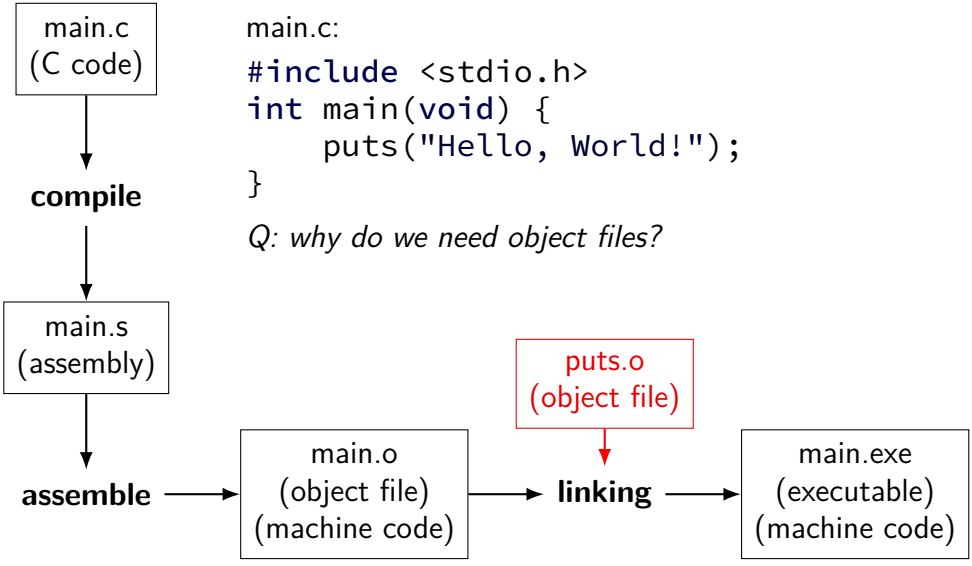


main.c:

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
}
```

Q: why do we need object files?

compilation pipeline



main.c:

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
}
```

Q: why do we need object files?

compilation commands

compile:	`gcc -S file.c`	⇒	`file.s` (assembly)
assemble:	`gcc -c file.s`	⇒	`file.o` (object file)
link:	`gcc -o file file.o`	⇒	`file` (executable)
c+a:	`gcc -c file.c`	⇒	`file.o`
c+a+l:	`gcc -o file file.c`	⇒	`file`

...

Note: combined compilation still does all the steps

object files, combining assembly files

main.s

```
.text
.global main
main:
  mov $str, %rdi
  call puts
  ret
.data
str:
  .string "Hello!"
```

puts.s

```
.text
.global puts
puts:
  ...
  call putchar
  ...
```

object files, combining assembly files

main.s

```
.text
.global main
main:
  mov $str, %rdi
  call puts
  ret
.data
str:
  .string "Hello!"
```

puts.s

```
.text
.global puts
puts:
  ...
  call putchar
  ...
```

combined?

```
.text
main:
  mov $str, %rdi
  call puts
  ret
.data
str:
    .string "Hello"

.text
puts:
  ...
  call putchar
  ...
```

object files, combining assembly files

main.s

```
.text
.global main
main:
  mov $str, %rdi
  call puts
  ret
.data
str:
  .string "Hello!"
```

puts.s

```
.text
.global puts
puts:
  ...
  call putchar
  ...
```

combined?

```
.text
main:
  mov $str, %rdi
  call puts
  ret
.data
str:
    .string "Hello"

.text
puts:
  ...
  call putchar
  ...
```

problem:
how many times do we generate library machine code?
repeated assemble

object files, combining assembly files

main.s

```
.text
.global main
main:
  mov $str, %rdi
  call puts
  ret
.data
str:
  .string "Hello!"
```

puts.s

```
.text
.global puts
puts:
  ...
  call putchar
  ...
```

combined?

```
.text
main:
  mov $str, %rdi
  call puts
  ret
.data
str:
    .string "Hello"

.text
puts:
  ...
  call putchar
  ...
```

challenge with making of machine code:
choosing + filling in addresses

object files, combining assembly files

main.s

```
.text
.global main
main:
  mov $str, %rdi
  call puts
  ret
.data
str:
  .string "Hello!"
```

puts.s

```
.text
.global puts
puts:
  ...
  call putchar
  ...
```

combined?

```
.text
main:
  mov $str, %rdi
  call puts
  ret
.data
str:
    .string "Hello"

.text
puts:
  ...
  call putchar
  ...
```

(with addrs)

```
0x10000:
  mov $0x20000, %rdi
  call 0x10040
  ret


0x10040:
  ...
  call 0x10800
  ...
  ...

0x20000:
  .string "Hello"
```

object files, combining assembly files

main.s

```
.text
.global main
main:
  mov $str, %rdi
  call puts
  ret
.data
str:
  .string "Hello!"
```

puts.s

```
.text
.global puts
puts:
  ...
  call putchar
  ...
```

main.s as machine code

```
mov $???str, %rdi
call ???puts
ret
.string "Hello"
```

puts.s as machine code

```
...
call ???putchar
...
```

idea:
translate each .s to machine code and combine later

problem:
can't put labels in machine code

what's in those files?

hello.c

```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

what's in those files?

hello.c

```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts // put string
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts // put stri
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

hello.s (Intel syntax)

```
.text
main:
  sub RSP, 8
  mov RDI, .Lstr
  call puts
  xor EAX, EAX
  add RSP, 8
  ret

.data
.Lstr: .string "Hello, Wor
```

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts // put string
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

Linux x86-64 calling convention: stack addr. must be multiple of 16

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts // put string
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

sets eax to 0 (shorter machine code than mov)

what's in those files?

hello.c

```c
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts // put string
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

mark used by other files

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts // put string
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts // put string
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts // put string
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts // put string
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

relocations:

take 0s at	*and replace with addr. of*
text, byte 5 ( )	data segment, byte 0
text, byte 10 ( )	puts

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts // put string
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

relocations:

take 0s at	*and replace with addr. of*
text, byte 5 ( )	data segment, byte 0
text, byte 10 ( )	puts

symbol table:

main	text byte 0

what's in those files?

hello.c

```
#include <stdio.h>
int main(void) {
  puts("Hello, World!");
  return 0;
}
```

hello.s

```
    .text
    .global main
main:
    sub  $8, %rsp
    mov  $.Lstr, %rdi
    call puts // put string
    xor  %eax, %eax
    add  $8, %rsp
    ret

    .data
.Lstr: .string "Hello, World!"
```

hello.o

text (code) segment:

```
48 83 EC 08 BF 00 00 00 00 E8 00 00
00 00 31 C0 48 83 C4 08 C3
```

data segment:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 00
```

relocations:

take 0s at	*and replace with addr. of*
text, byte 5 ( )	data segment, byte 0
text, byte 10 ( )	puts

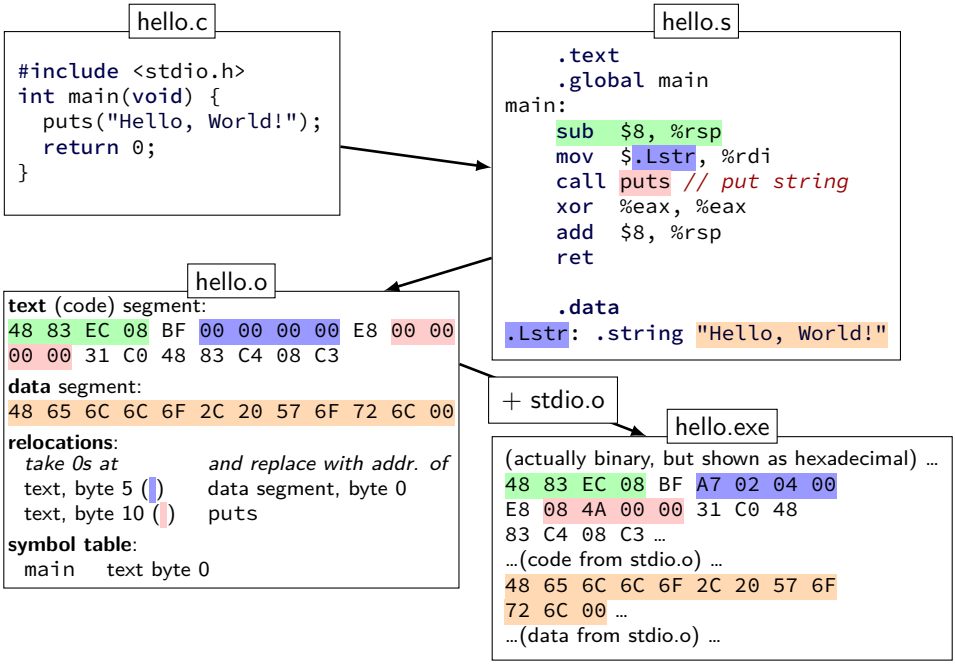
symbol table:

main text byte 0

`.Lstr` location specified w/o name
and not usable by other files
so no symbol table entry needed

(convention: .L…labels always local)

what's in those files?



exercise (1)

main.c:

```c
#include <stdio.h>
void sayHello(void) {
    puts("Hello, World!");
}
int main(void) {
    sayHello();
}
```

Which files likely contain the **memory address** of `sayHello`?

A. main.s (assembly)
B. main.o (object)
C. main.exe (executable)
D. B and C
E. A, B and C
F. something else

exercise (2)

main.c:

```
#include <stdio.h>
void sayHello(void) {
    puts("Hello, World!");
}
int main(void) {
    sayHello();
}
```

Which files likely contain **literal ASCII string** of `Hello, World!`?

A. main.s (assembly)
B. main.o (object)
C. main.exe (executable)
D. B and C
E. A, B and C
F. something else

main.s contains it?

```
.text
.global sayHello
sayHello:
    mov $.Lstr, %rdi
    call puts
    ...
.data
.Lstr:
    .string "Hello, World!"
```

```
.text
.global sayHello
sayHello:
    mov $.Lstr, %rdi
    call puts
    ...
.data
.Lstr:
    .byte 72,101,108,108,111,44,32,87,111,114,108,100,33,0
```

main.o contains it?

complaint: in hexadecimal, like we've shown?

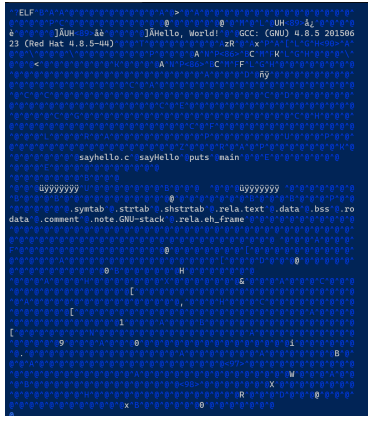
most object file formats aim for **efficiency**

simpler for linker to copy raw bytes

similar argument for main.exe and program loading

results with gcc

```
        .file   "sayhello.c"
        .section        .rodata
.LC0:
        .string "Hello, World!"
        .text
        .globl  sayHello
        .type   sayHello, @function
sayHello:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   sayHello, .-sayHello
        .globl  main
        .type   main, @function
main:
.LFB1:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        call    sayHello
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1:
        .size   main, .-main
        .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-4)"
        .section        .note.GNU-stack,"",@progbits
```



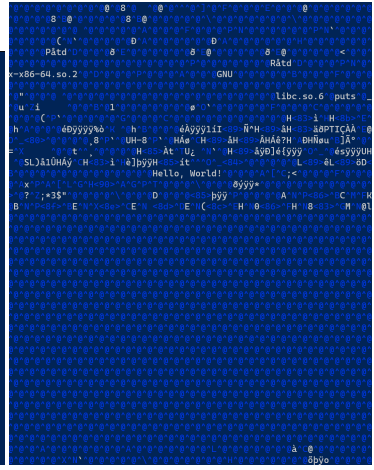


Figure: left to right: assembly, object, executable

dynamic linking (very briefly)

dynamic linking — done when application is loaded

idea: don't have N copies of `printf` on disk

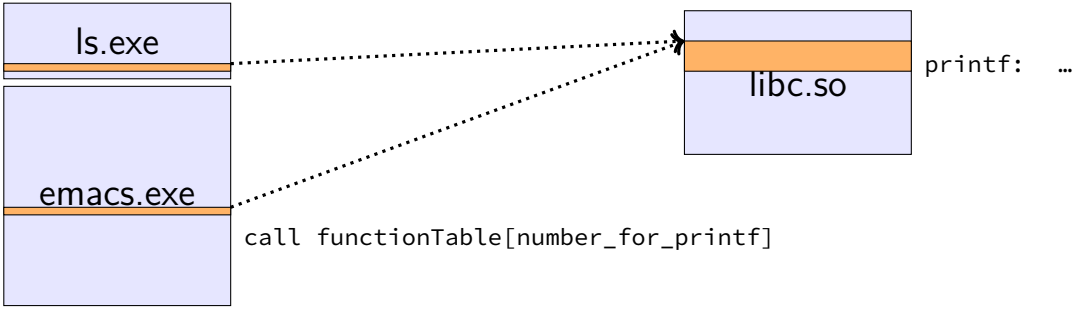
other type of linking: *static* (`gcc -static`)

load executable file + its libraries into memory when app starts

often extra indirection:

`call functionTable[number_for_printf]`

linker fills in `functionTable` instead of changing `call`s



ldd /bin/ls

```
$ ldd /bin/ls
    linux-vdso.so.1 =>  (0x00007ffcca9d8000)
    libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1
            (0x00007f851756f000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
            (0x00007f85171a5000)
    libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3
            (0x00007f8516f35000)
    libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2
            (0x00007f8516d31000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f8517791000)
    libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0
            (0x00007f8516b14000)
$ ldd cs3330WS/sayhello
        linux-vdso.so.1 =>  (0x00007ffd601ba000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f07a7f22000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f07a82f0000)
```

relocation types

machine code doesn't always use addresses as is

"call function 4303 bytes later"

linker needs to compute "4303"

- extra field on relocation list

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

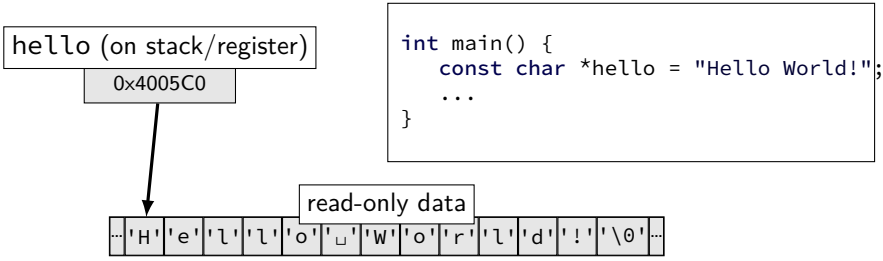
false values in C

`0`

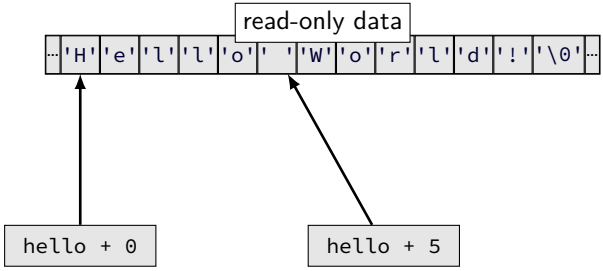
including null pointers — `0` cast to a pointer

Everything else is *true* (-1, 1, 4, other non-zero values)

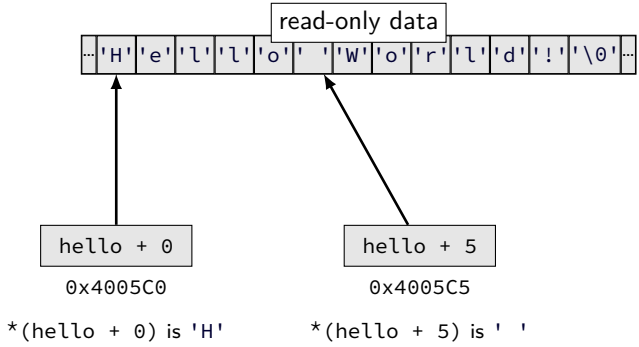
strings in C


hello (on stack/register)
0x4005C0
int main() {
const char *hello = "Hello World!";
...
}
read-only data
'H' 'e' 'l' 'l' 'o' '␣' 'W' 'o' 'r' 'l' 'd' '!' '\0'

pointer arithmetic



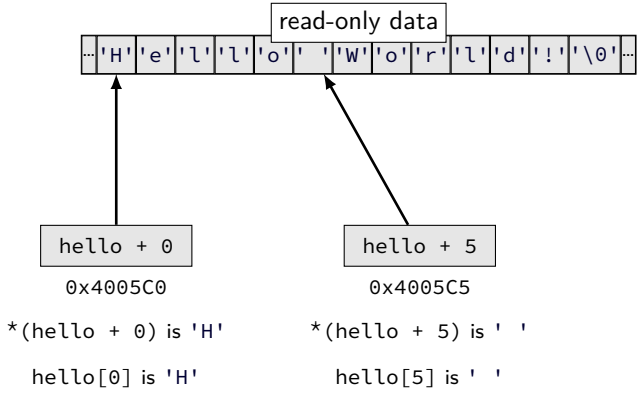
pointer arithmetic



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

`*(hello + 5)` is `' '`

pointer arithmetic



`*(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

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

arrays of non-bytes

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

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

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

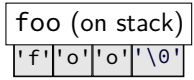
```
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```

Final value of `foo`?

A. `"fao"`
B. `"zao"`
C. `"baz"`
D. `"bao"`
E. something else/crash

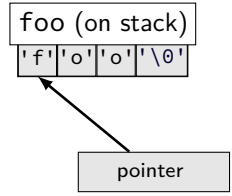
exercise explanation

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



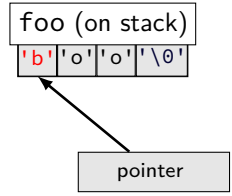
exercise explanation

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



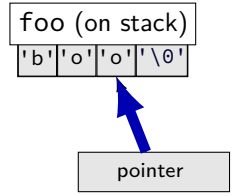
exercise explanation

```
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

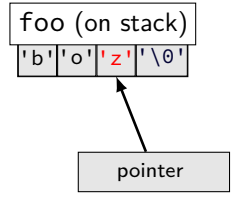
```
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```



exercise explanation

```
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';`

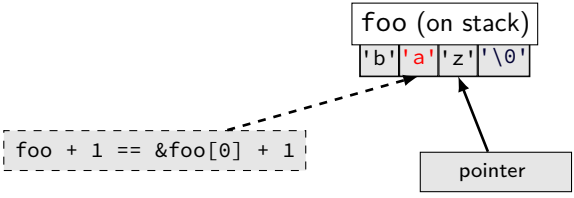


exercise explanation

```
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';`



arrays: not quite pointers (1)

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

Legal: `pointer = array;`

same as `pointer = &(array[0]);`

arrays: not quite pointers (1)

```
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

arrays: not quite pointers (2)

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

sizeof(array) == 400
```

size of all elements

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

size of address

arrays: not quite pointers (2)

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

`sizeof(array) == 400`

size of all elements

`sizeof(pointer) == 8`

size of address

`sizeof(&array[0]) == ???`

(`&array[0]` same as `&(array[0])`)

struct

```
struct rational {
    int numerator;
    int denominator;
};
// ...
struct rational two_and_a_half; // must use 'struct'
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);
```

typedef (to the rescue)

instead of writing:

```
...
unsigned int a;
unsigned int b;
unsigned int c;
```

can write:

```
typedef unsigned int uint;
...
uint a;
uint b;
uint c;
```

typedef struct (1)

```
struct other_name_for_rational {
    int numerator;
    int denominator;
};
typedef struct other_name_for_rational rational;
// ...
rational two_and_a_half; // now we don't have to use 'struct'
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)

```
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;
}; // define struct inside typedef
```

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;
}; // define struct inside typedef
```

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;
}; // define struct inside typedef

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

typedef struct (3)

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

valid ways to declare an instance:

```
struct other_name_for_rational some_variable;
rational some_variable;
```

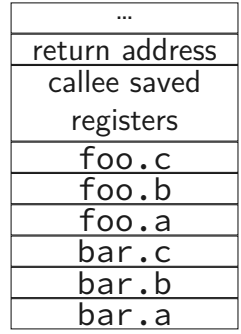
INVALID ways:

```
/* INVALID: */ struct rational some_variable;
/* INVALID: */ other_name_for_rational some_variabl
```

structs aren't references

```
typedef struct {
    long a; long b; long c;
} triple;
...

triple foo;
foo.a = foo.b = foo.c = 3;
triple bar = foo;
bar.a = 4; // makes a copy!
// foo is {3, 3, 3}
// bar is {4, 3, 3}
```



C evolution and standards

1978: Kernighan and Ritchie publish *The C Programming Language* — "K&R C"

very different from modern C

C evolution and standards

1978: Kernighan and Ritchie publish *The C Programming Language* — "K&R C"

- very different from modern C

1989: ANSI standardizes C — C89/C90/`-ansi`

- compiler option: `-ansi`, `-std=c90`
- looks mostly like modern C

C evolution and standards

1978: Kernighan and Ritchie publish *The C Programming Language* — "K&R C"

- very different from modern C

1989: ANSI standardizes C — C89/C90/`-ansi`

- compiler option: `-ansi`, `-std=c90`
- looks mostly like modern C

1999: ISO (and ANSI) update C standard — C99

- compiler option: `-std=c99`
- adds: declare variables in middle of block
- adds: `//` comments

C evolution and standards

1978: Kernighan and Ritchie publish *The C Programming Language* — "K&R C"

- very different from modern C

1989: ANSI standardizes C — C89/C90/`-ansi`

- compiler option: `-ansi`, `-std=c90`
- looks mostly like modern C

1999: ISO (and ANSI) update C standard — C99

- compiler option: `-std=c99`
- adds: declare variables in middle of block
- adds: `//` comments

2011, 2017: Second/Third ISO update — C11, C17

undefined behavior example (1)

```
#include <stdio.h>
#include <limits.h>
int test(int number) {
    return (number + 1) > number;
}

int main(void) {
    printf("%d\n", test(INT_MAX)); // INT_MAX+1
}
```

undefined behavior example (1)

```
#include <stdio.h>
#include <limits.h>
int test(int number) {
    return (number + 1) > number;
}

int main(void) {
    printf("%d\n", test(INT_MAX)); // INT_MAX+1
}
```

without optimizations: `0`

undefined behavior example (1)

```
#include <stdio.h>
#include <limits.h>
int test(int number) {
    return (number + 1) > number;
}

int main(void) {
    printf("%d\n", test(INT_MAX)); // INT_MAX+1
}
```

without optimizations: `0`

with optimizations: `1`

undefined behavior example (2)

```
int test(int number) {
    return (number + 1) > number;
}
```

Optimized:

```
test:
    movl    $1, %eax        # eax <- 1
    ret
```

Less optimized:

```
test:
    leal    1(%rdi), %eax # eax <- rdi + 1
    cmpl    %eax, %edi
    setl    %al           # al <- eax < edi
    movzbl  %al, %eax     # eax <- al (pad with zeros)
    ret
```

undefined behavior

compilers can do whatever they want

- what you expect
- crash your program
- ...

common types:

- *signed* integer overflow/underflow
- out-of-bounds pointers
- integer divide-by-zero
- writing read-only data
- out-of-bounds shift

undefined behavior

why undefined behavior?

different architectures work differently

- allow compilers to expose whatever processor does "naturally"
- don't encode any particular machine in the standard

flexibility for optimizations