program layout / C stuff

changelog

5 September 2022: adjust “what’s in those files” to indicate that we’re putting in the *address* of the data segment

last time

lea (load effective address)

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

condition codes

- special 1-bit registers w/ results of "last arithmetic"
- ZF = was zero?; SF = was negative [sign bit]?
- also OF, CF for overflow
- cmp = sub but only set condition codes
- jXX — named after comparing to 0 (or subtracting)

converting control flow to assembly

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 — 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:
```

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
```

compiling switches (2)

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

```
    // binary search
    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
cmpq $1, %rax
jl code_for_default
jmp *table - 8(,%rax,8)
```

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

compiling switches (3b)

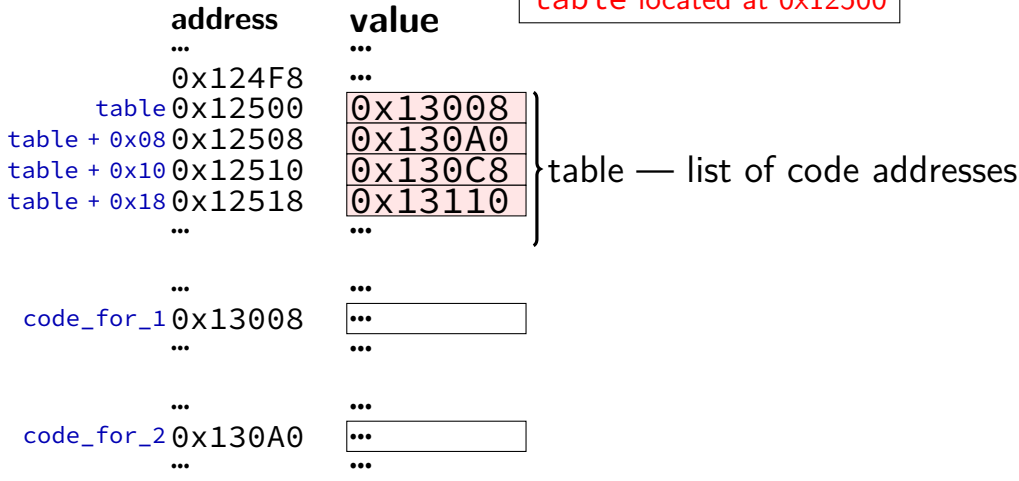
```
jmp *table-8(,%rax,8)
```

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

compiling switches (3b)

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

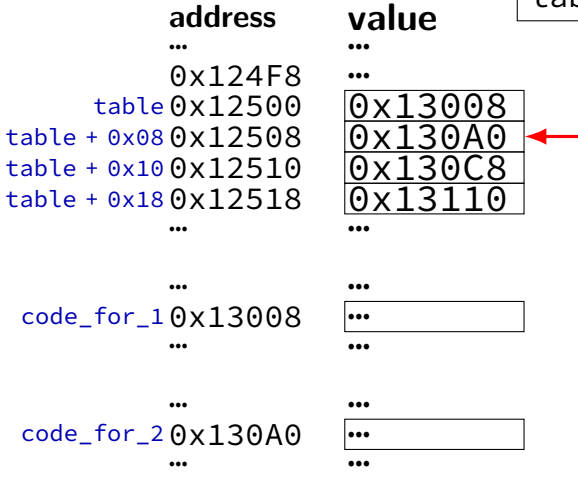
suppose RAX = 2,
`table` located at 0x12500



compiling switches (3b)

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

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

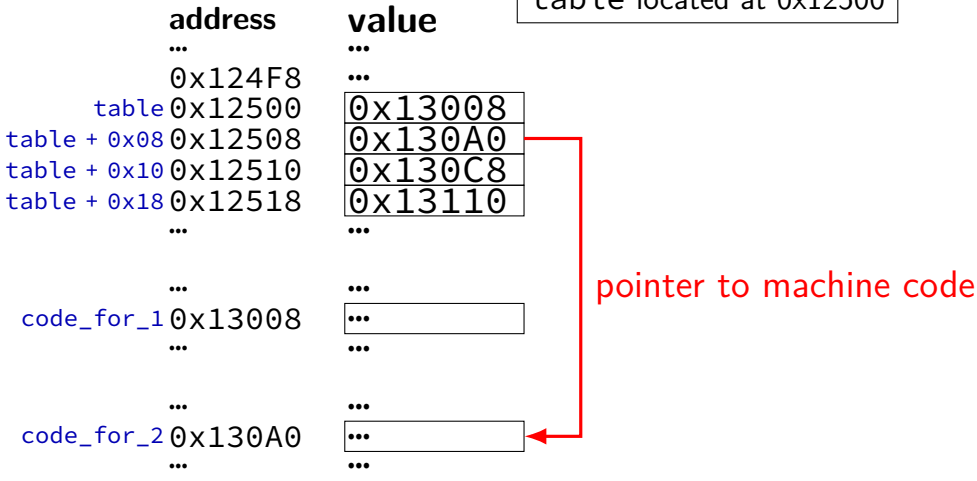


$(\text{table} - 8) + \text{rax} \times 8 =$
$\texttt{0x124F8} + \texttt{0x10} = \texttt{0x12508}$

compiling switches (3b)

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

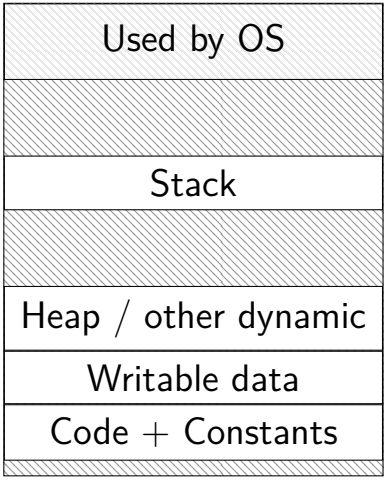
suppose RAX = 2,
`table` located at 0x12500



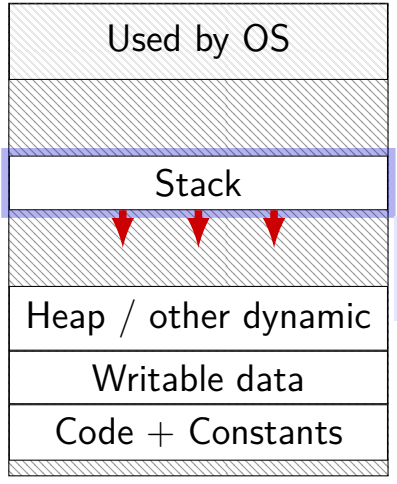
computed jumps

```
  cmpq $100, %rax
  jg code_for_default
  cmpq $1, %rax
  jl code_for_default
  // 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
  ...
```

program memory (x86-64 Linux)



program memory (x86-64 Linux)



0xFFFF FFFF FFFF FFFF

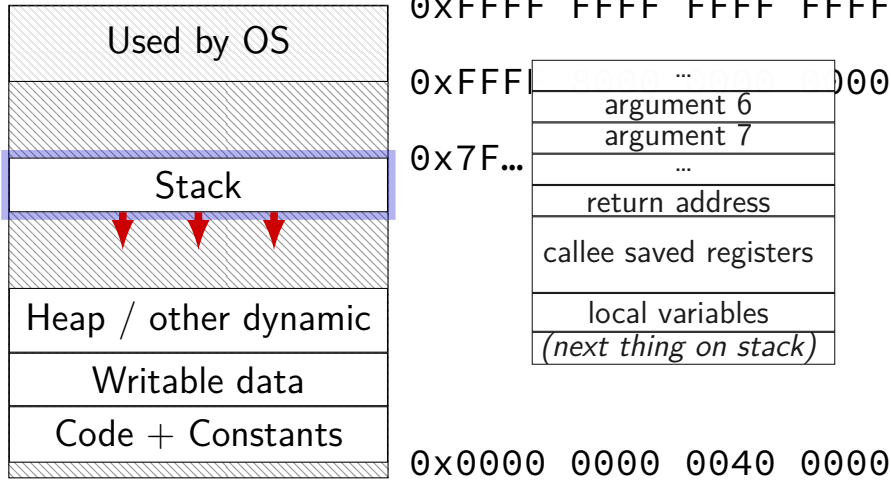
0xFFFF 8000 0000 0000

0x7F…

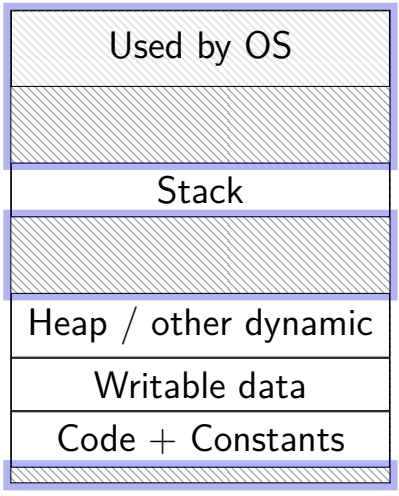
stack grows down
"top" has smallest address

0x0000 0000 0040 0000

program memory (x86-64 Linux)



program memory (x86-64 Linux)



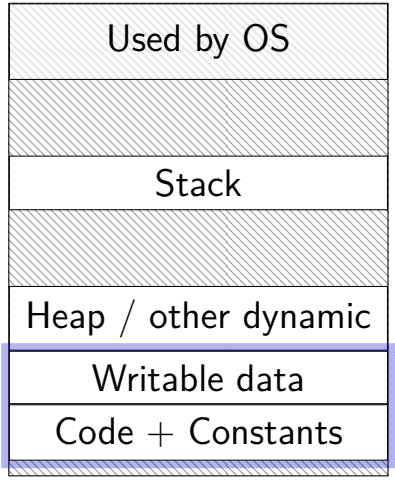
0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F…

0x0000 0000 0040 0000

program memory (x86-64 Linux)



0xFFFF FFFF FFFF FFFF

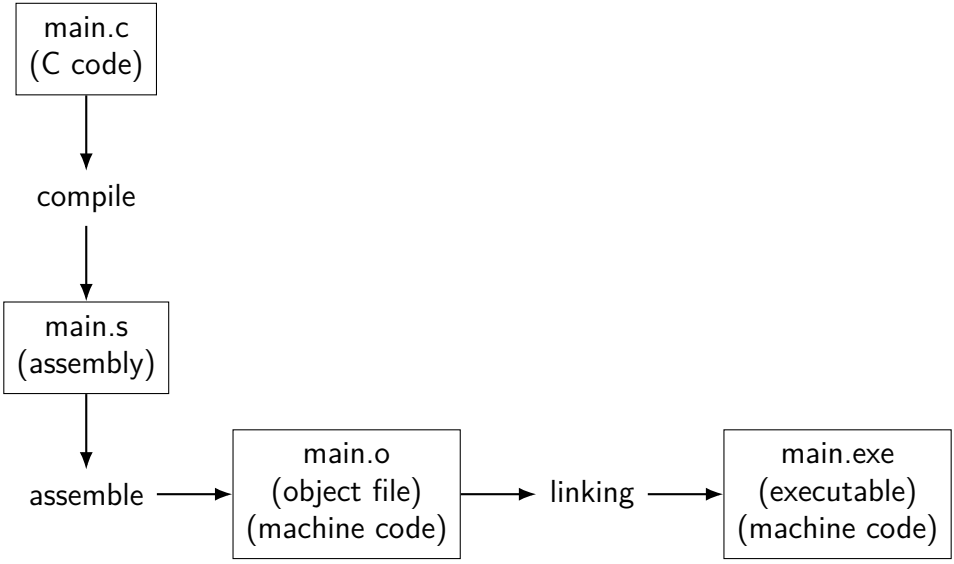
0xFFFF 8000 0000 0000

0x7F…

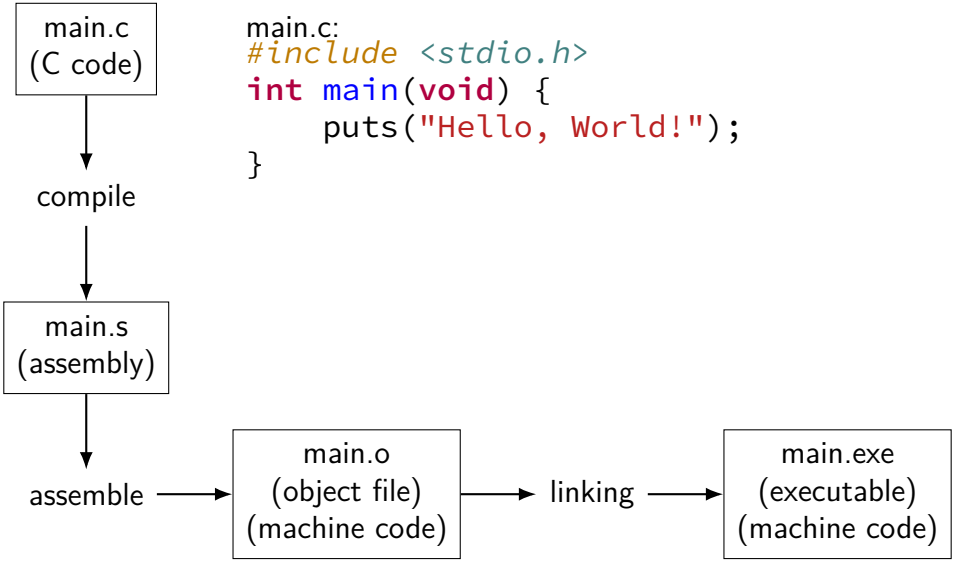
loaded from executable file

0x0000 0000 0040 0000

compilation pipeline



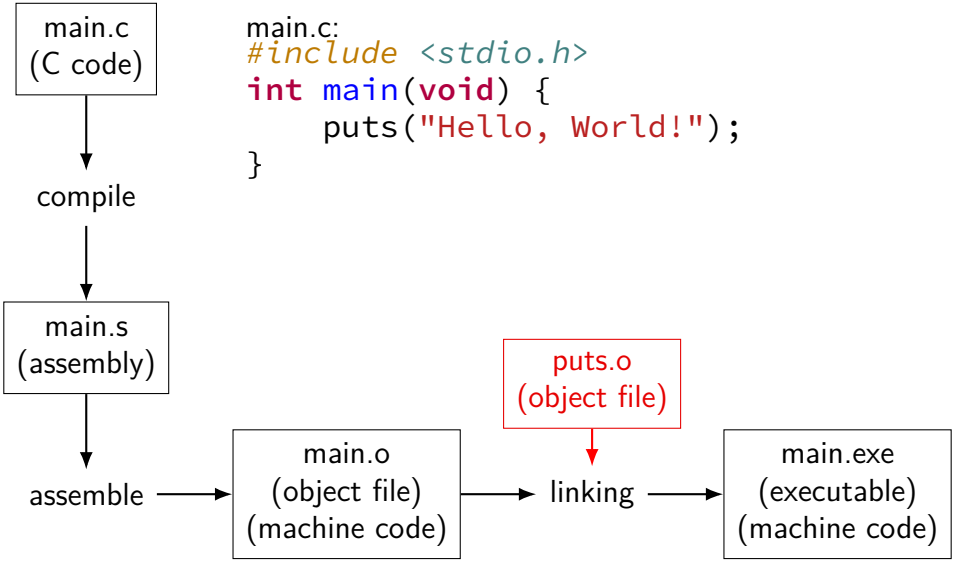
compilation pipeline



main.c:

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

compilation pipeline



main.c:

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

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`

...

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
  ...
```

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
  ...
```

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?

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
  ...
```

normal part of making machine code:
choosing + filling in addresses

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"
```

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
    xor  %eax, %eax
    add  $8, %rsp
    ret

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

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
    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

```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
    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

```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
    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
    xor  %eax, %eax
    add  $8, %rsp
    ret

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

mark used by other files

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
    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

```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
    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

```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
    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
    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

```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
    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
    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

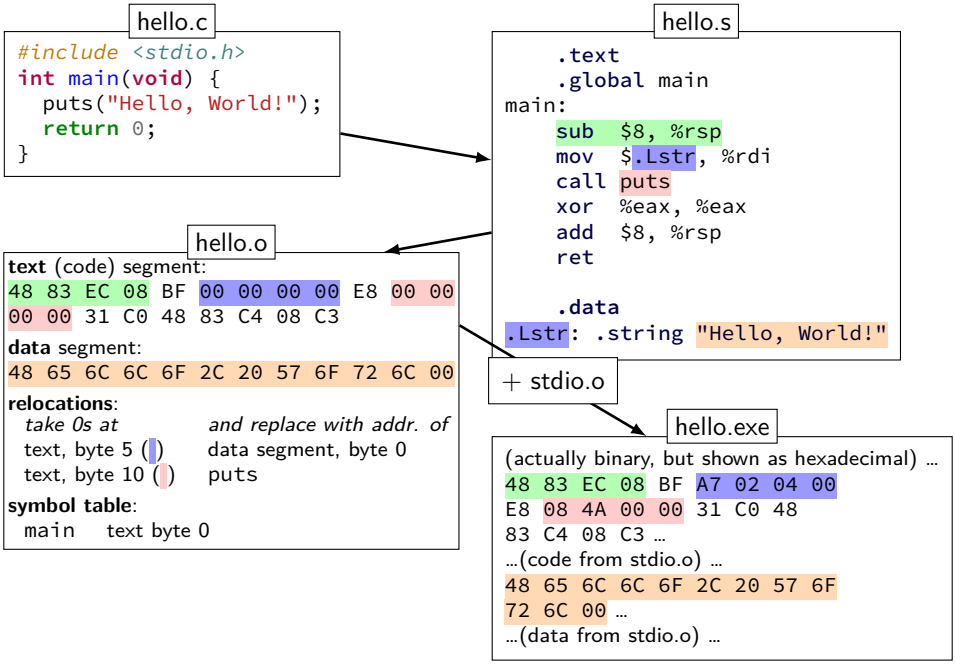
symbol table:

```
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?



hello.s

```
        .section        .rodata.str1.1,"aMS",@progb
.LC0:
        .string "Hello, World!"
        .text
        .globl  main
main:
        subq    $8, %rsp
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        addq    $8, %rsp
        ret
```

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:

```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

dynamic linking (very briefly)

dynamic linking — done when application is loaded

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

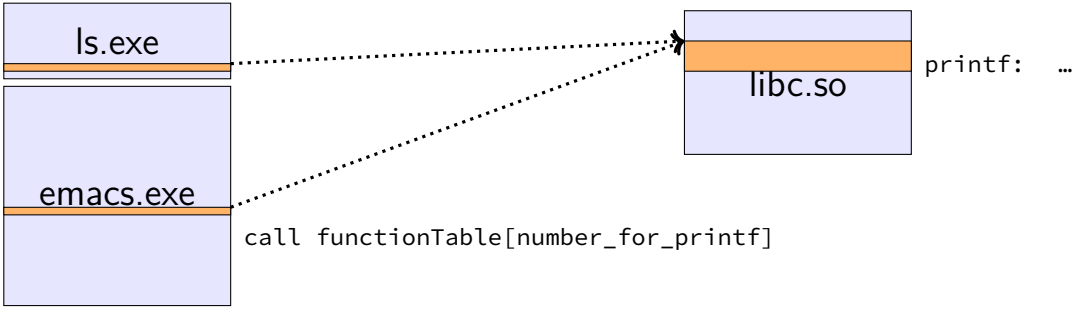
other type of linking: *static* (`gcc -static`)

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)
```

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`**

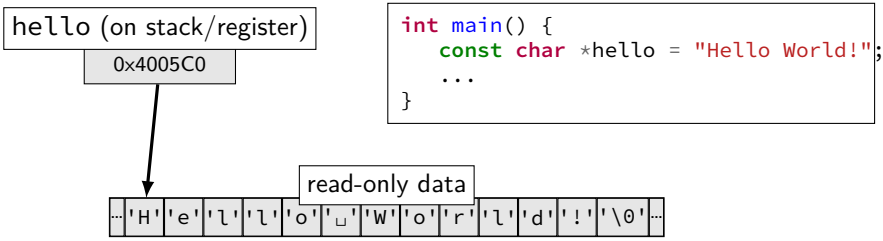
`1` if true; `0` if false

false values in C

`0`

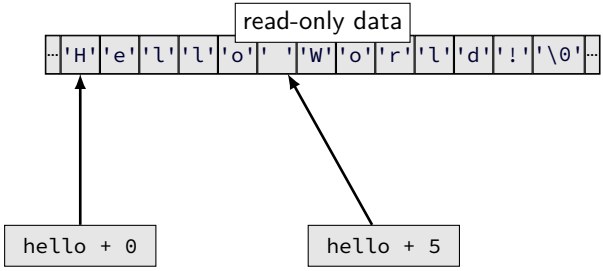
including null pointers — `0` cast to a pointer

strings in C

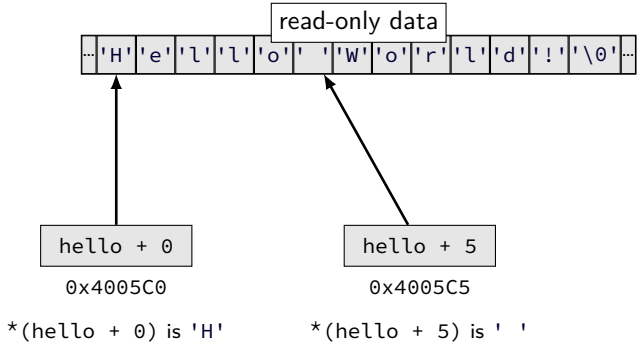



```c
int main() {
    const char *hello = "Hello World!";
    ...
}
```

pointer arithmetic



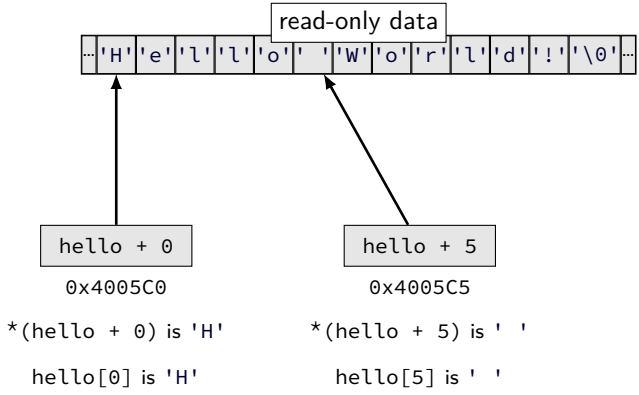
pointer arithmetic



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

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

pointer arithmetic



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

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

Final value of `foo`?

A. `"fao"`

B. `"zao"`

C. `"baz"`

D. `"bao"`

E. something else/crash

exercise

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

Final value of `foo`?

A. `"fao"`

B. `"zao"`

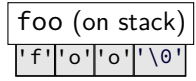
C. `"baz"`

D. `"bao"`

E. something else/crash

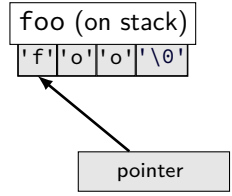
exercise explanation

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



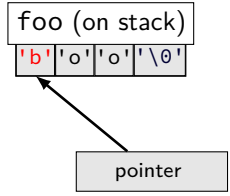
exercise explanation

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



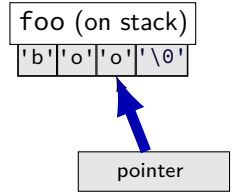
exercise explanation

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



exercise explanation

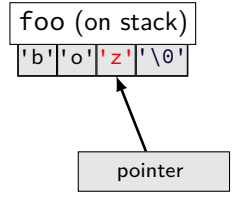
```
1  char foo[4] = "foo";
2      // {'f', 'o', 'o', '\0'}
3  char *pointer;
4  pointer = foo;
5  *pointer = 'b';
6  pointer = pointer + 2;
7  pointer[0] = 'z';
8  *(foo + 1) = 'a';
```



exercise explanation

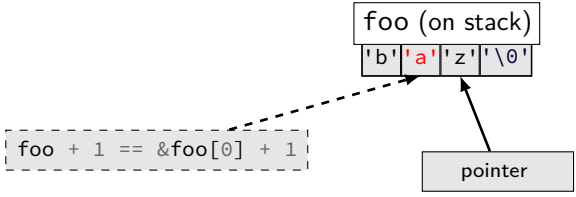
```
1  char foo[4] = "foo";
2      // {'f', 'o', 'o', '\0'}
3  char *pointer;
4  pointer = foo;
5  *pointer = 'b';
6  pointer = pointer + 2;
7  pointer[0] = 'z';
8  *(foo + 1) = 'a';
```

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



exercise explanation

```
1  char foo[4] = "foo";
2      // {'f', 'o', 'o', '\0'}
3  char *pointer;
4  pointer = foo;
5  *pointer = 'b';
6  pointer = pointer + 2;
7  pointer[0] = 'z';     better style: *pointer = 'z';
8  *(foo + 1) = 'a';     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;
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

```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);
```

typedef

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;
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 (1)

```c
struct other_name_for_rational {
    int numerator;
    int denominator;
};
typedef struct other_name_for_rational 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)

```
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)

```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)

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

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_variable;
```

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_variable;
```

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;
```

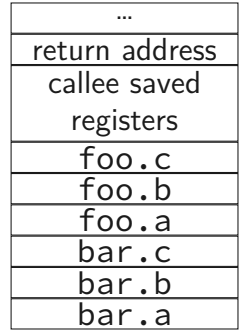
INVALID ways:

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

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;
// 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)

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

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

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));
}
```

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));
}
```

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

backup slides

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 exercise: translating?

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

while exercise: translating?

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

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

objdump -sx test.o (Linux) (1)

```
test.o:     file format elf64-x86-64
test.o
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000000  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  00000040  2**0
                  ALLOC
  3 .rodata.str1.1 0000000e  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .text.startup 00000014  0000000000000000  0000000000000000  0000004e  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  5 .comment      0000002b  0000000000000000  0000000000000000  00000062  2**0
                  CONTENTS, READONLY
  6 .note.GNU-stack 00000000  0000000000000000  0000000000000000  0000008d  2**0
                  CONTENTS, READONLY
  7 .eh_frame     00000030  0000000000000000  0000000000000000  00000090  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

objdump -sx test.o (Linux) (2)

```
SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 test.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .rodata.str1.1 0000000000000000 .rodata.str1.1
0000000000000000 l    d  .text.startup  0000000000000000 .text.startup
0000000000000000 l    d  .note.GNU-stack        0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l       .rodata.str1.1 0000000000000000 .LC0
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000000 g     F .text.startup  0000000000000014 main
0000000000000000         *UND*  0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000000000         *UND*  0000000000000000 puts
```

columns:

- memory address (not yet assigned, so 0)
- flags: l=local, g=global, F=function, ...
- section (.text, .data, .bss, ...)
- offset in section
- name of symbol

objdump -sx test.o (Linux) (3)

```
RELOCATION RECORDS FOR [.text.startup]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_PC32     .LC0-0x0000000000000004
000000000000000c R_X86_64_PLT32    puts-0x0000000000000004


RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text.startup

Contents of section .rodata.str1.1:
 0000 48656c6c 6f2c2057 6f726c64 2100      Hello, World!.
Contents of section .text.startup:
 0000 488d3d00 00000048 83ec08e8 00000000  H.=....H........
 0010 31c05ac3                             1.Z.
Contents of section .comment:
 0000 00474343 3a202855 62756e74 7520372e  .GCC: (Ubuntu 7.
 0010 332e302d 32377562 756e7475 317e3138  3.0-27ubuntu1~18
 0020 2e303429 20372e33 2e3000             .04) 7.3.0.
Contents of section .eh_frame:
 0000 14000000 00000000 017a5200 01781001  .........zR..x..
 0010 1b0c0708 90010000 14000000 1c000000  ................
 0020 00000000 14000000 004b0e10 480e0800  .........K..H...
```

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;
```

linked lists / dynamic allocation

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

linked lists / dynamic allocation

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

linked lists / dynamic allocation

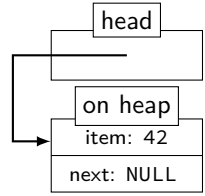
```
typedef struct list_t {
    int item;
    struct list_t *next;
} list;
// ...


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

linked lists / dynamic allocation

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

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

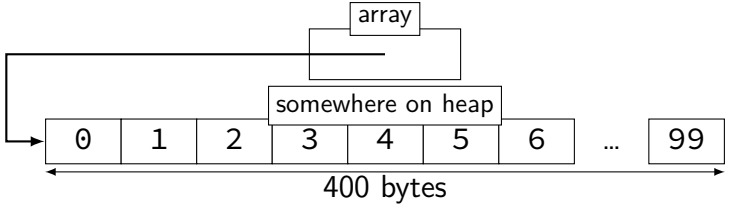


dynamic arrays

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

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
```



a note on precedence

`&foo[42]` is the same as `&(foo[42])` (*not* `(&foo)[42]`)

`*foo[42]` is the same as `*(foo[42])` (*not* `(*foo)[42]`)

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

unsigned and signed types

type	min	max
`signed int` = `signed` = `int`	-2^{31}	$2^{31}-1$
`unsigned int` = `unsigned`	0	$2^{32}-1$
`signed long` = `long`	-2^{63}	$2^{63}-1$
`unsigned long`	0	$2^{64}-1$

⋮

unsigned/signed comparison trap (1)

```
int x = -1;
unsigned int y = 0;
printf("%d\n", x < y);
```

unsigned/signed comparison trap (1)

```
int x = -1;
unsigned int y = 0;
printf("%d\n", x < y);
```

result is 0

unsigned/signed comparison trap (1)

```
int x = -1;
unsigned int y = 0;
printf("%d\n", x < y);
```

result is 0

short solution: don't compare signed to unsigned:

```
(long) x < (long) y
```

unsigned/sign comparison trap (2)

```
int x = -1;
unsigned int y = 0;
printf("%d\n", x < y);
```

compiler converts both to same type first

- `int` if all possible values fit
- otherwise: first operand (`x`, `y`) type from this list:
  - `unsigned long`
  - `long`
  - `unsigned int`
  - `int`

stdio.h

C does not have `<iostream>`

instead `<stdio.h>`

stdio

```
cr4bd@power1
: /if22/cr4bd ; man stdio

…
STDIO(3)                Linux Programmer's Manual                STDIO(3)

NAME
       stdio - standard input/output library functions

SYNOPSIS
       #include <stdio.h>

       FILE *stdin;
       FILE *stdout;
       FILE *stderr;

DESCRIPTION
       The  standard  I/O  library  provides  a  simple  and  efficient
       buffered stream I/O interface.  Input and output is mapped  into
       logical  data  streams  and the physical I/O characteristics are
       concealed.  The functions and  macros  are  listed  below;  more
       information is available from the individual man pages.
```

stdio

```
STDIO(3)                Linux Programmer's Manual                STDIO(3)

NAME
       stdio - standard input/output library functions

…

   List of functions
       Function      Description
       ------------------------------------------------------------------
       clearerr      check and reset stream status
       fclose        close a stream

…

       printf        formatted output conversion

…
```

printf

```
int custNo = 1000;
const char *name = "Jane Smith"
    printf("Customer #%d: %s\n " ,
       custNo, name);
// "Customer #1000: Jane Smith"
// same as:
cout << "Customer #" << custNo
     << ": " << name << endl;
```

printf

```
1 int custNo = 1000;
2 const char *name = "Jane Smith"
3     printf("Customer #%d: %s\n " ,
4         custNo, name);
5 // "Customer #1000: Jane Smith"
6 // same as:
7 cout << "Customer #" << custNo
8      << ": " << name << endl;
```

printf

```
int custNo = 1000;
const char *name = "Jane Smith"
    printf("Customer #%d: %s\n " ,
       custNo, name);
// "Customer #1000: Jane Smith"
// same as:
cout << "Customer #" << custNo
     << ": " << name << endl;
```

format string must match types of argument

printf formats quick reference

Specifier	Argument Type	Example(s)
%s	char *	Hello, World!
%p	any pointer	0x4005d4
%d	int/short/char	42
%u	unsigned int/short/char	42
%x	unsigned int/short/char	2a
%ld	long	42
%f	double/float	42.000000
		0.000000
%e	double/float	4.200000e+01
		4.200000e-19
%g	double/float	42, 4.2e-19
%%	(no argument)	%

printf formats quick reference

Specifier	Argument Type	Example(s)
%s	char *	Hello, World!
%p	any pointer	0x4005d4
%d	int/short/char	42
%u	unsigned int/short/char	42
%x	unsigned int/short/char	2a
%ld		
%f	double/float	42.000000
		0.000000
%e	double/float	4.200000e+01
		4.200000e-19
%g	double/float	42, 4.2e-19
%%	(no argument)	%

detailed docs: `man 3 printf`

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 (||)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf("> %d\n", zero() || one());
6     printf("> %d\n", one() || zero());
7     return 0;
8 }
```

```
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 (||)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf("> %d\n", zero() || one());
6     printf("> %d\n", one() || zero());
7     return 0;
8 }
```

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

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

short-circuit (||)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf("> %d\n", zero() || one());
6     printf("> %d\n", one() || zero());
7     return 0;
8 }
```

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

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

short-circuit (&&)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf("> %d\n", zero() && one());
6     printf("> %d\n", one() && zero());
7     return 0;
8 }
```

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

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

short-circuit (&&)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf("> %d\n", zero() && one());
6     printf("> %d\n", one() && zero());
7     return 0;
8 }
```

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

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

short-circuit (&&)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf("> %d\n", zero() && one());
6     printf("> %d\n", one() && zero());
7     return 0;
8 }
```

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

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

short-circuit (&&)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf("> %d\n", zero() && one());
6     printf("> %d\n", one() && zero());
7     return 0;
8 }
```

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

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

short-circuit (&&)

```
1 #include <stdio.h>
2 int zero() { printf("zero()\n"); return 0; }
3 int one() { printf("one()\n"); return 1; }
4 int main() {
5     printf("> %d\n", zero() && one());
6     printf("> %d\n", one() && zero());
7     return 0;
8 }
```

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

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