assembly 2 / compilation pipeline [if time]

Changelog

1 September 2020: condition code intro slide: correct JNE in question to JE to match assembly

1 September 2020: condition codes: closer look: correct ZF = 0 to ZF = 1 in a couple examples of what Jxx instructions check

1 September 2020: condition codes: closer look: order list of unsigned jumps to match order of list of signed jumps

1 September 2020: if-to-assembly (1): correct += 1 to += 10 in comment

assembly 2 / compilation pipeline [if time]

last time

syntax error in assembly labels exercise

- should have used `%bl` (one-byte register) instead of `%bx`

endianness

AT&T syntax

- destination last operand
- % before registers
- $ before constant operands to instructions
- D(B reg, I reg, S) = memory[D+B reg+I reg×S]

[9:30a sect] LEA

on LEA

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

effective address = computed address for memory access

syntax looks like a **mov** from memory, but...

skips the memory access — just uses the address

(sort of like & operator in C?)

`leaq 4(%rax), %rax` ≈ `addq $4, %rax`

on LEA

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

effective address = computed address for memory access

syntax looks like a **mov** from memory, but...

skips the memory access — just uses the address

(sort of like & operator in C?)

`leaq 4(%rax), %rax` ≈ `addq $4, %rax`

"address of memory[rax + 4]" = rax + 4

LEA tricks

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

rax $\leftarrow$ rax $\times$ 5

rax $\leftarrow$ `address-of(memory[rax + rax * 4])`

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

rdx $\leftarrow$ rbx $+$ rcx

rdx $\leftarrow$ `address-of(memory[rbx + rcx])`

exercise: what is this function?

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

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

A. `arg * 9`
B. `-arg * 9`
C. `arg * 8`
D. `-arg * 7`
E. none of these
F. it has a different prototype

exercise: what is this function?

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

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

A. `arg * 9`
B. `-arg * 9`
C. `arg * 8`
D. `-arg * 7`
E. none of these
F. it has a different prototype

Linux x86-64 calling convention

registers for first 6 arguments:

- `%rdi` (or `%edi` or `%di`, etc.), then
- `%rsi` (or `%esi` or `%si`, etc.), then
- `%rdx` (or `%edx` or `%dx`, etc.), then
- `%rcx` (or `%ecx` or `%cx`, etc.), then
- `%r8` (or `%r8d` or `%r8w`, etc.), then
- `%r9` (or `%r9d` or `%r9w`, etc.)

rest on stack

return value in `%rax`

don't memorize: Figure 3.28 in book

x86-64 calling convention example

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

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

call/ret

call:

push address of next instruction on the stack

ret:

pop address from stack; jump

callee-saved registers

functions must preserve these

`%rsp` (stack pointer), `%rbx`, `%rbp` (frame pointer, maybe)

`%r12`–`%r15`

caller/callee-saved

```
foo:
    pushq %r12 // r12 is callee-saved
    ... use r12 ...
    popq %r12
    ret

...
other_function:
    ...
    pushq %r11 // r11 is caller-saved
    callq foo
    popq %r11
```

things we won't cover (today)

special meaning of labels with %rip (`label(%rip)` ≈ `%rip`)

floating point; vector operations (multiple values at once)

- special registers: %xmm0 through %xmm15

segmentation (special registers: %ds, %fs, %gs, ...)

lots and lots of instructions

conditionals in x86 assembly

```
    if (rax != 0)
        foo();
```

```
    cmpq $0, %rax
    // ***
    je skip_call_foo
    call foo
skip_call_foo:
```

---

how does `je` know the result of the comparison?

what happens if we add extra instructions at the `***`?

condition codes

x86 has condition codes

special registers 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

alternate view: comparing result to 0

0: equal; negative: less than; positive: greater than

condition codes: closer look

ZF ("zero flag") — was result zero? (sub/cmp: equal)

- e.g. JE (jump if equal) checks for ZF = 1

SF ("sign flag") — was result negative? (sub/cmp: less)

- e.g. JL (jump if less than) checks for SF = 1 (plus extra case for overflow)
- e.g. JLE checks for SF = 1 or ZF = 1 (plus overflow)

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

condition codes: closer look

`ZF` ("zero flag") — was result zero? (sub/cmp: equal)

- e.g. JE (jump if equal) checks for ZF = 1

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

- e.g. JL (jump if less than) checks for SF = 1 (plus extra case for overflow)
- e.g. JLE checks for SF = 1 or ZF = 1 (plus overflow)

`OF` ("overflow flag") — did computation overflow (as signed)?

- we won't test on this/use it in later assignments
- signed conditional jumps: JL, JLE, JG, JGE, ...

`CF` ("carry flag") — did computation overflow (as unsigned)?

- we won't test on this/use it in later assignments
- unsigned conditional jumps: JB, JBE, JA, JAE, ...

(and one more)

condition codes: closer look

`ZF` ("zero flag") — was result zero? (sub/cmp: equal)

- e.g. JE (jump if equal) checks for ZF = 1

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

- e.g. JL (jump if less than) checks for SF = 1 (plus extra case for overflow)
- e.g. JLE checks for SF = 1 or ZF = 1 (plus overflow)

`OF` ("overflow flag") — did computation overflow (as signed)?

- we won't test on this/use it in later assignments
- signed conditional jumps: JL, JLE, JG, JGE, ...

`CF` ("carry flag") — did computation overflow (as unsigned)?

- we won't test on this/use it in later assignments
- unsigned conditional jumps: JB, JBE, JA, JAE, ...

(and one more)

condition codes (and other flags) in GDB

```
(gdb) info registers
rax            0x0                 0
rbx            0x555555555150      93824992235856
rcx            0x555555555150      93824992235856
...
rip            0x55555555513a      0x55555555513a <m
eflags         0x246               [ PF ZF IF ]
cs             0x33                51
ss             0x2b                43
...
```

ZF = 1 (listed); SF, OF, CF clear

some other flags that you can lookup (PF, IF) also shown

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 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
```

30: SF = 0 (not negative), ZF = 0 (not zero)

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

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.

how do you know? — check processor's manual

condition codes example (2)

```
movq $-10, %rax  // rax <- (-10)
movq $20, %rbx   // rbx <- 20
cmpq %rax, %rbx  // set cond codes w/ rbx - rax
jle foo // not taken; %rbx - %rax > 0
```

condition codes example (2)

```
movq $-10, %rax  // rax <- (-10)
movq $20, %rbx   // rbx <- 20
cmpq %rax, %rbx  // set cond codes w/ rbx - rax
jle foo // not taken; %rbx - %rax > 0
```

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

omitting the cmp

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

```
    movq $99, %r12          // x (r12) ← 99
start_loop:
    call foo                // foo()
    subq $1, %r12           // x (r12) ← x - 1
    jge  start_loop         // new r12 >= 0?
```

condition code exercise

```
movq %rcx, %rdx
subq $1, %rdx
addq %rdx, %rcx
```

Assuming no overflow, possible values of SF, ZF?

A. SF $= 0$, ZF $= 0$
B. SF $= 1$, ZF $= 0$
C. SF $= 0$, ZF $= 1$
D. SF $= 1$, ZF $= 1$

condition code exercise

```
movq %rcx, %rdx
subq $1, %rdx
addq %rdx, %rcx
```

~~Assuming no overflow,~~ possible values of SF, ZF?

A. SF $= 0$, ZF $= 0$
B. SF $= 1$, ZF $= 0$
C. SF $= 0$, ZF $= 1$
D. SF $= 1$, ZF $= 1$

if-to-assembly (1)

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

if-to-assembly (1)

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

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

if-to-assembly (2)

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

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

exercise

```
    subq %rax, %rbx
    addq %rbx, %rcx
    je after
    addq %rax, %rcx
after:
```

This is equivalent to what C code? (rax = var. assigned to register %rax, etc.)

A

```
rbx -= rax;
rcx += rbx;
if (rcx == 0) {
    rcx += rax;
}
```

B

```
rbx -= rax;
rcx += rbx;
if (rbx == rcx) {
    rcx += rax;
}
```

C

```
rbx -= rax;
rcx += rbx;
if (rbx + rcx == 0) {
    rcx += rax;
}
```

D

```
rcx += (rbx - rax);
if (rcx == (rbx - rax)) {
    rcx += rax;
}
```

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 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:
```

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)

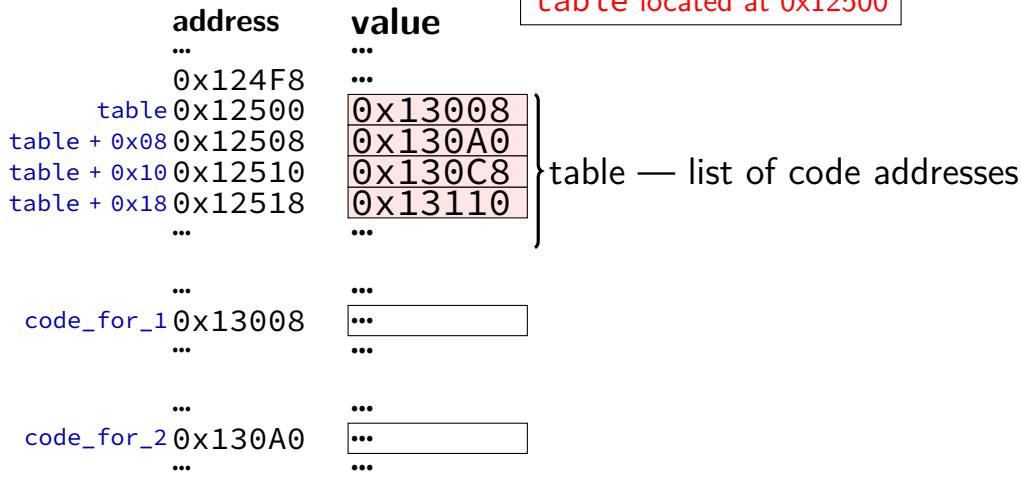
```
jmp *table-8(,%rax,8)
```

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

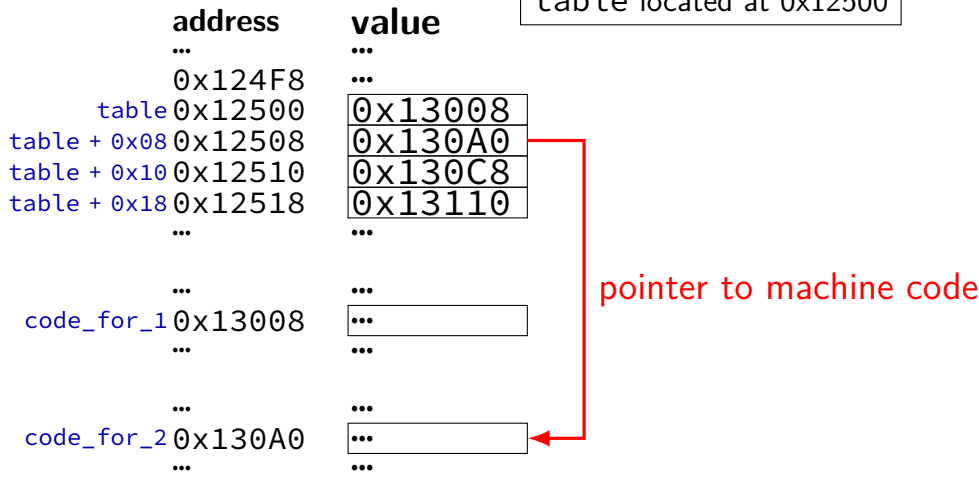
	address	value
	…	…
	0x124F8	…
table	0x12500	0x13008
table + 0x08	0x12508	0x130A0
table + 0x10	0x12510	0x130C8
table + 0x18	0x12518	0x13110
	…	…
	…	…
code_for_1	0x13008	…
	…	…
	…	…
code_for_2	0x130A0	…
	…	…

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

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
  ...
```

backup slides

condition codes example: no cmp (3)

```
movq $-10, %rax  // rax ← (-10)
movq $20, %rbx   // rbx ← 20
subq %rax, %rbx  // rbx ← rbx - rax = 30
jle  foo // not taken, %rbx - %rax > 0


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

do-while-to-assembly (1)

```
int x = 99;
do {
    foo()
    x--;
} while (x >= 0);
```

do-while-to-assembly (1)

```
int x = 99;
do {
    foo()
    x--;
} while (x >= 0);
```

```
    int x = 99;
start_loop:
    foo()
    x--;
    if (x >= 0) goto start_loop;
```

do-while-to-assembly (2)

```
int x = 99;
do {
    foo()
    x--;
} while (x >= 0);
```

```
    movq $99, %r12 // register for x
start_loop:
    call foo
    subq $1, %r12
    cmpq $0, %r12
    // computes r12 - 0 = r12
    jge start_loop // jump if r12 - 0 >= 0
```

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 to cond. codes
je   foo        // taken, result is 0
```

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

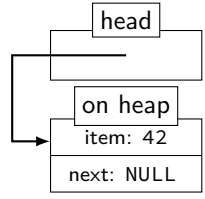
```
typedef struct list_t {
    int item;
    struct list_t *next;
} list;
// ...


  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 */
```

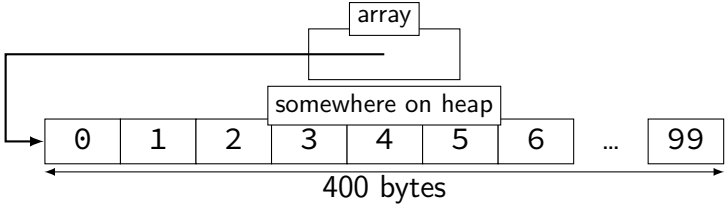


dynamic arrays

```
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
```



man chmod

```
File  Edit  View  Search  Terminal  Help
CHMOD(1)                          User Commands                          CHMOD(1)

NAME
       chmod - change file mode bits

SYNOPSIS
       chmod [OPTION]... MODE[,MODE]... FILE...
       chmod [OPTION]... OCTAL-MODE FILE...
       chmod [OPTION]... --reference=RFILE FILE...

DESCRIPTION
       This  manual page documents the GNU version of chmod.  chmod changes the file mode bits
       of each given file according to mode, which can be either a symbolic representation  of
       changes to make, or an octal number representing the bit pattern for the new mode bits.

       The  format  of a symbolic mode is [ugoa...][[-+=][perms...]...], where perms is either
       zero or more letters from the set rwxXst, or a single letter from the set ugo.   Multi-
       ple symbolic modes can be given, separated by commas.

       A  combination  of  the  letters  ugoa controls which users' access to the file will be
       changed: the user who owns it (u), other users in the file's group (g), other users not
       in  the  file's group (o), or all users (a).  If none of these are given, the effect is
       as if (a) were given, but bits that are set in the umask are not affected.

       The operator + causes the selected file mode bits to be added to the existing file mode
       bits  of  each  file;  -  causes  them to be removed; and = causes them to be added and
       causes unmentioned bits to be removed except that a directory's  unmentioned  set  user
       and group ID bits are not affected.

       The  letters  rwxXst select file mode bits for the affected users: read (r), write (w),
Manual page chmod(1) line 1/125 27% (press h for help or q to quit)
```

chmod

```
chmod --recursive og-r /home/USER
```

chmod

```
chmod --recursive og-r /home/USER
```

others and group (student)
− remove
read

chmod

```
chmod --recursive og-r /home/USER
```

user (yourself) / group / others
− remove / + add
read / write / execute or search

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

```
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

```
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

```
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;
```

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	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)	%

detailed docs: `man 3 printf`

struct

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

struct

```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)

```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 (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;
```

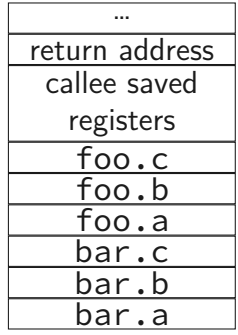
INVALID ways:

```
/* 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
```



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...
```

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

on %rip

%rip (**I**nstruction **P**ointer) = address of next instruction

```
movq 500(%rip), %rax
```

rax ← memory[next instruction address + 500]

on %rip

%rip (**I**nstruction **P**ointer) = address of next instruction

`movq 500(%rip), %rax`

rax ← memory[next instruction address + 500]

`label(%rip)` ≈ `label`

different ways of writing address of label in machine code
(with %rip — relative to next instruction)