CS 3330: More C

30 August 2016

Layers of Abstraction

`x += y`	"Higher-level" language: C
`add %rbx, %rax`	Assembly: X86-64
60 03	Machine code: Y86
	(we'll talk later)
	Logic and Registers

Last time

compile / assemble / link

C data types, lack of bool

arrays and pointers

command-line tips:

man, chmod, `tar`, tab completion, history

`printf` and `<stdio.h>`

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)	%

goto

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

assembly:
jmp found

assembly:
found:

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);
```


typedef struct (1)

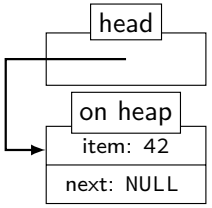
```
struct other_name_for_rational {
    int numerator;
    int denominator;
};
typedef struct other_name_for_rational rational;
// ...
rational two_and_a_half;
two_and_a_half.numerator = 5;
two_and_a_half.denominator = 2;
rational *pointer = &two_and_a_half;
printf("%d/%d\n",
       pointer->numerator,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

typedef struct (2)

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

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

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

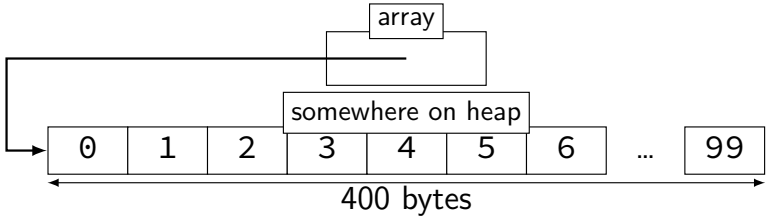

linked lists / dynamic allocation

```
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
```



Miss vector? (1)

```
typedef struct range_t {
    int size;
    int *data;
} range;

range vec;
vec.size = 100;
vec.data = malloc(sizeof(int) * 100);
// like: vector<int> vec(100);
```

Miss vector? (2)

```
typedef struct range_t {
    int size;
    int *data;
} range;

range vec2;
vec2.size = vec.size;
vec2.data = malloc(sizeof(int) * vec.size);
for (int i = 0; i < vec.size; ++i) {
    vec2.data[i] = vec.data[i];
}
// like: vector<int> vec2 = vec;
```

Why not `range vec2 = vec`?

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

```
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
```



```
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`

C evolution and standards

1978: Kernighan and Ritchie publish *The C Programming Language* — "K&R C"
- very different from moderm 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: Second ISO update — C11

Middle of blocks?

Examples of things not allowed in 1989 ANSI C:

```
printf("Before␣calling␣malloc()\n");
int *pointer = malloc(sizeof(int) * 100);
```

```
for (int x = 0; x < 10; ++x)
```


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
    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 (later)

Bit-twiddling

some truth tables

AND	**0**	**1**
0	0	0
1	0	1

&&, &

OR	**0**	**1**
0	0	1
1	1	1

||, |

XOR	**0**	**1**
0	0	1
1	1	0

(nothing), ^

Logical Operators

Treat value as true (1) or false (0)

Recall: false = 0 (only)

```
 1 && 1  == 1        1 || 1  == 1
 2 && 4  == 1        2 || 4  == 1
 1 && 0  == 0        1 || 0  == 1
 0 && 0  == 0        0 || 0  == 0
-1 && -2 == 1       -1 || -2 == 1
"" && "" == 1       "" || "" == 1
```

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	**0**	**1**
0	0	0
1	0	1

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	**0**	**1**
0	0	1
1	1	1

Bitwise AND — &

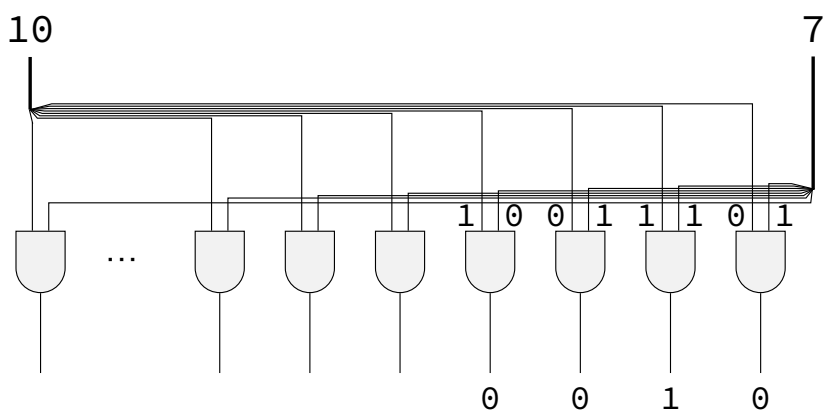
Treat value as array of bits

```
1 & 1 == 1
1 & 0 == 0
0 & 0 == 0
2 & 4 == 0
10 & 7 == 2
```

	…	0	0	1	0
&	…	0	1	0	0
	…	0	0	0	0

	…	1	0	1	0
&	…	0	1	1	1
	…	0	0	1	0

Bitwise hardware (10 & 7 == 2)



Bitwise OR — |

```
1 | 1 == 1
1 | 0 == 1
0 | 0 == 0
2 | 4 == 6
10 | 7 == 15
```

	…	0	0	1	0
|	…	0	1	0	0
	…	0	1	1	0

	…	1	0	1	0
|	…	0	1	1	1
	…	1	1	1	1

Bitwise xor — ^

```
1 ^ 1 == 0
1 ^ 0 == 1
0 ^ 0 == 0
2 ^ 4 == 6
10 ^ 7 == 13
```

	…	0	0	1	0
^	…	0	1	0	0
	…	0	1	1	0

	…	1	0	1	0
^	…	0	1	1	1
	…	1	1	0	1

Negation / Not — ~

~ ('complement') is bitwise version of !:

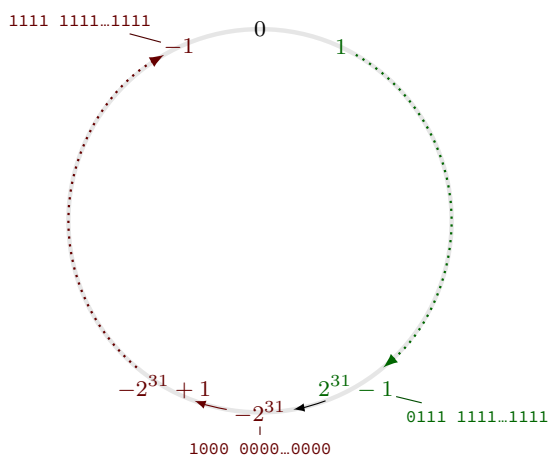
```
!0 == 1
!notZero == 0
```

	32 bits						
~	0	0	…	0	0	0	0
	1	1	…	1	1	1	1

```
~0 == (int) 0xFFFFFFFF~0 == −1
~2 == −3
~−7 == 6

~((unsigned) 2) == 0xFFFFFFFD
```


Two's complement refresher

	-2^{31}	$+2^{30}$	$+2^{29}$		$+2^{2}$	$+2^{1}$	$+2^{0}$
$-1 =$	1	1	1	…	1	1	1



Two's Complement and ~

`-x == ~x + 1` (flip the bits and add one)

`-x - 1 == ~x`

Negation / Not — ~

~ ('complement') is bitwise version of `!`:

```
!0 == 1

!notZero == 0

~0 == (int) 0xFFFFFFFF
~2 == -3
~-7 == 6

~((unsigned) 2) == 0xFFFFFFFD
```

32 bits

~	0	0	…	0	0	0	0
	1	1	…	1	1	1	1

~0 == -1

Left shift

```
1 << 0 == 1          0000 0001
1 << 1 == 2          0000 0010
1 << 2 == 4          0000 0100

10 << 0 == 10        0000 1010
10 << 1 == 20        0001 0100
10 << 2 == 40        0010 1000
```

$$x \texttt{ << } y = x \times 2^y$$

Right shift

Undefined: ~~`x << -1`~~ Instead: `x >> 1`

```
1 >> 0 == 1          0000 0001
1 >> 1 == 0          0000 0000
1 >> 2 == 0          0000 0000

10 >> 0 == 10        0000 1010
10 >> 1 == 5         0000 0101
10 >> 2 == 2         0000 0010
```

$$x \texttt{ >> } y = \lfloor x \times 2^{-y} \rfloor$$

Shifts and negative numbers

`−10 >> 1 == ???` (-10 = `1111 … 1111 0110`)

binary `?111 … 1111 1011`

Option 1: binary `1111 … 1011` = $-5 = -10 \times 2^{-k}$

- copy sign bit
- arithmetic

Option 2: binary `0111 … 1011` = $2^{31} - 5$

- always use zero
- logical

Aside: implementation-defined behavior

C standard: negative `>> 1` is implementation-defined

compiler chooses

Arithmetic shift

```
−1 >> 0 == −1          1111 1111
−1 >> 1 == −1          1111 1111
−1 >> 2 == −1          1111 1111

−10 >> 0 == −10        1111 0110
−10 >> 1 == −5         1111 1011
−10 >> 2 == −3         1111 1101

10 >> 2 == 2           0000 0010
```

$$x \texttt{ >> } y = \lfloor x \times 2^{-y} \rfloor$$

signed versus unsigned types

```
/*signed*/ int x = −10;
/* arithmetic: */
x >> 1 == −5
x >> 4 == −1

unsigned int y = 0xFFFFFFF6;
/* logical */
y >> 1 == 0x7FFFFFFB
y >> 4 == 0x0FFFFFFF
```

Sign-extension vs. zero-extension

```
signed char x = -10;     //          1111 0110
int y = x;               // 1111.. 1111 0110
printf("%d\n", y);       // outputs "-10"

unsigned char x = 0xF6; //          1111 0110
int y = x;               // 0000.. 1111 0110
printf("%d\n", y);       // outputs "246"
```


Aside: integer promotions

```
unsigned short number = 1;
unsigned short offset = 20;
printf("0x%x\n", number << offset);
```

Outputs (on lab machines)?

A. `0x100000` (2^{20}) **B.** `0x0`

C. Undefined behavior — varies

integers types smaller than **`int`** converted to **`int`**

Shifts: undefined behavior

`0 >> 32` is undefined behavior

`0 << 32` is undefined behavior

`(long) 0 << 32` is okay

`(long) 0 << 64` is undefined behavior

Summary

`struct` — functionless classes

`typedef struct` or write **`struct`** `typeName`

`malloc`, `free` instead of new/delete.

undefined behavior — who knows what'll happen

logical operators — &&, ||, !: only care if 0/not 0

bitwise operators — &, |, ^, ~: all bits in parallel

bitshift — <<, >>: same as multiplying by $2^{\pm x}$

arithmetic right shift — borrow sign bit

Backup Slides

What's in those files?

hello.c

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

hello.s

```
    .text
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*
text, byte 6 ( )	data segment, byte 0
text, byte 10 ( )	address of puts

symbol table:

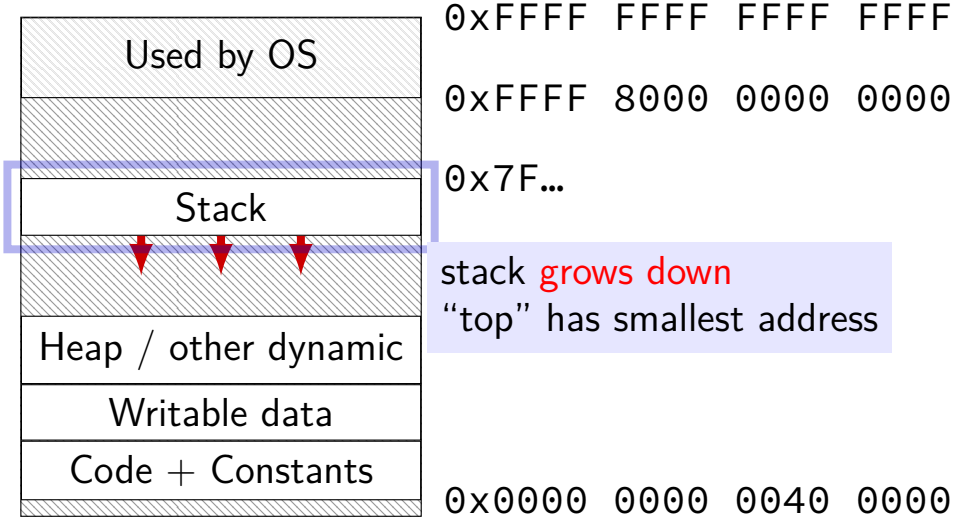
main	text byte 0

+ stdio.o

hello.exe

```
48 83 EC 08 BF A7 02 04 00
E8 08 4A 04 00 31 C0 48
83 C4 08 C3 …
…(code from stdio.o) …
48 65 6C 6C 6F 2C 20 57
6F 72 6C 00 …
…(data from stdio.o) …
```


Program Memory (x86-64 Linux)



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

`sizeof(pointer) == 8`

size of address

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

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

chmod

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

user (yourself) / group / others

– remove / + add

read / write / execute or search

tar

the standard Linux/Unix file archive utility

Table of contents: `tar tf filename.tar`

eXtract: `tar xvf filename.tar`

Create: `tar cvf filename.tar directory`

(`v`: verbose; `f`: file — default is tape)