Fundamental Data Types: Binary representation and Operators

Lecture overview

- Data types ←
- Integer representation
- Floating point representation
- Literals and variables
- Operators in C

Data type

- A data type (or simply type) is an attribute that classifies what type of data a variable can hold.
- Used to let the compiler know how the programmer intends to use the data, what operations can be applied, etc.

(Select) Fundamental data types in C

- Signed integers:
  - short
  - int
  - long (int)
  - long long (int)
- Unsigned integers:
  - char (discussed in 2 slides)
  - unsigned (int)
  - unsigned long (int)
  - unsigned long long (int)
- Floating point numbers:
  - float
  - double
  - long double

- Signed types are more commonly used than unsigned types.
- Each of the 3 categories has its binary representation.
- Different types within category vary in the space required to store it and in what it can hold.
- Other types (e.g., strings) are implemented using pointers (which we discuss in 2-3 weeks).

Data size

- Different data types take up a different amount of space
- The space required to store a type is **machine dependent**
- Use operator `sizeof(datatype)` to get # bytes.

- Default (integer) type in C is `int`
- Default floating point type in C is `double`

On our Linux server:

char:	1 byte
short:	2 bytes
int:	4 bytes
unsigned:	4 bytes
long:	8 bytes
long long:	8 bytes
float:	4 bytes
double:	8 bytes
long double:	16 bytes

code to produce this printout in `/share/classes/class05/type-sizes.c`

Data type "char"

- ASCII = American Standard Code for Information Interchange
- char is an unsigned integer that holds ASCII values 0-255
- requires 1 byte
- Just **like any other integer type**, so you can apply any integer operation (e.g., `'c'+1 ('d')`)

0	<NUL>	32	<SPC>	64	@	96	`	128	Ä	160	†	192	¿	224	‡
1	<SOH>	33	!	65	A	97	a	129	Å	161	°	193	¡	225	·
2	<STX>	34	"	66	B	98	b	130	Ç	162	¢	194	¬	226	‚
3	<ETX>	35	#	67	C	99	c	131	É	163	£	195	√	227	„
4	<EOT>	36	$	68	D	100	d	132	Ñ	164	§	196	ƒ	228	‰
5	<ENQ>	37	%	69	E	101	e	133	Ö	165	•	197	≈	229	Â
6	<ACK>	38	&	70	F	102	f	134	Ü	166	¶	198	Δ	230	Ê
7	<BEL>	39	'	71	G	103	g	135	á	167	ß	199	«	231	Á
8	<BS>	40	(	72	H	104	h	136	à	168	®	200	»	232	Ë
9	<TAB>	41	)	73	I	105	i	137	â	169	©	201	…	233	È
10	<LF>	42	*	74	J	106	j	138	ä	170	™	202		234	Í
11	<VT>	43	+	75	K	107	k	139	ã	171	´	203	À	235	Î
12	<FF>	44	,	76	L	108	l	140	å	172	¨	204	Ã	236	Ï
13	<CR>	45	-	77	M	109	m	141	ç	173	≠	205	Õ	237	Ì
14	<SO>	46	.	78	N	110	n	142	é	174	Æ	206	Œ	238	Ó
15	<SI>	47	/	79	O	111	o	143	è	175	Ø	207	œ	239	Ô
16	<DLE>	48	0	80	P	112	p	144	ê	176	∞	208	–	240	
17	<DC1>	49	1	81	Q	113	q	145	ë	177	±	209	—	241	Ò
18	<DC2>	50	2	82	R	114	r	146	í	178	≤	210	“	242	Ú
19	<DC3>	51	3	83	S	115	s	147	ì	179	≥	211	”	243	Û
20	<DC4>	52	4	84	T	116	t	148	î	180	¥	212	‘	244	Ù
21	<NAK>	53	5	85	U	117	u	149	ï	181	µ	213	’	245	ı
22	<SYN	54	6	86	V	118	v	150	ñ	182	∂	214	÷	246	ˆ
23	<ETB>	55	7	87	W	119	w	151	ó	183	∑	215	◊	247	˜
24	<CAN>	56	8	88	X	120	x	152	ò	184	∏	216	ÿ	248	¯
25	<EM>	57	9	89	Y	121	y	153	ô	185	π	217	Ÿ	249	˘
26	<SUB>	58	:	90	Z	122	z	154	ö	186	∫	218	/	250	˙
27	<ESC>	59	;	91	[	123	{	155	õ	187	ª	219	€	251	°
28	<FS>	60	<	92	\	124	|	156	ú	188	º	220	‹	252	¸
29	<GS>	61	=	93	]	125	}	157	ù	189	Ω	221	›	253	˝
30	<RS>	62	>	94	^	126	~	158	û	190	æ	222	fi	254	˛
31	<US>	63	?	95	_	127	<DEL>	159	ü	191	ø	223	fl	255	ˇ

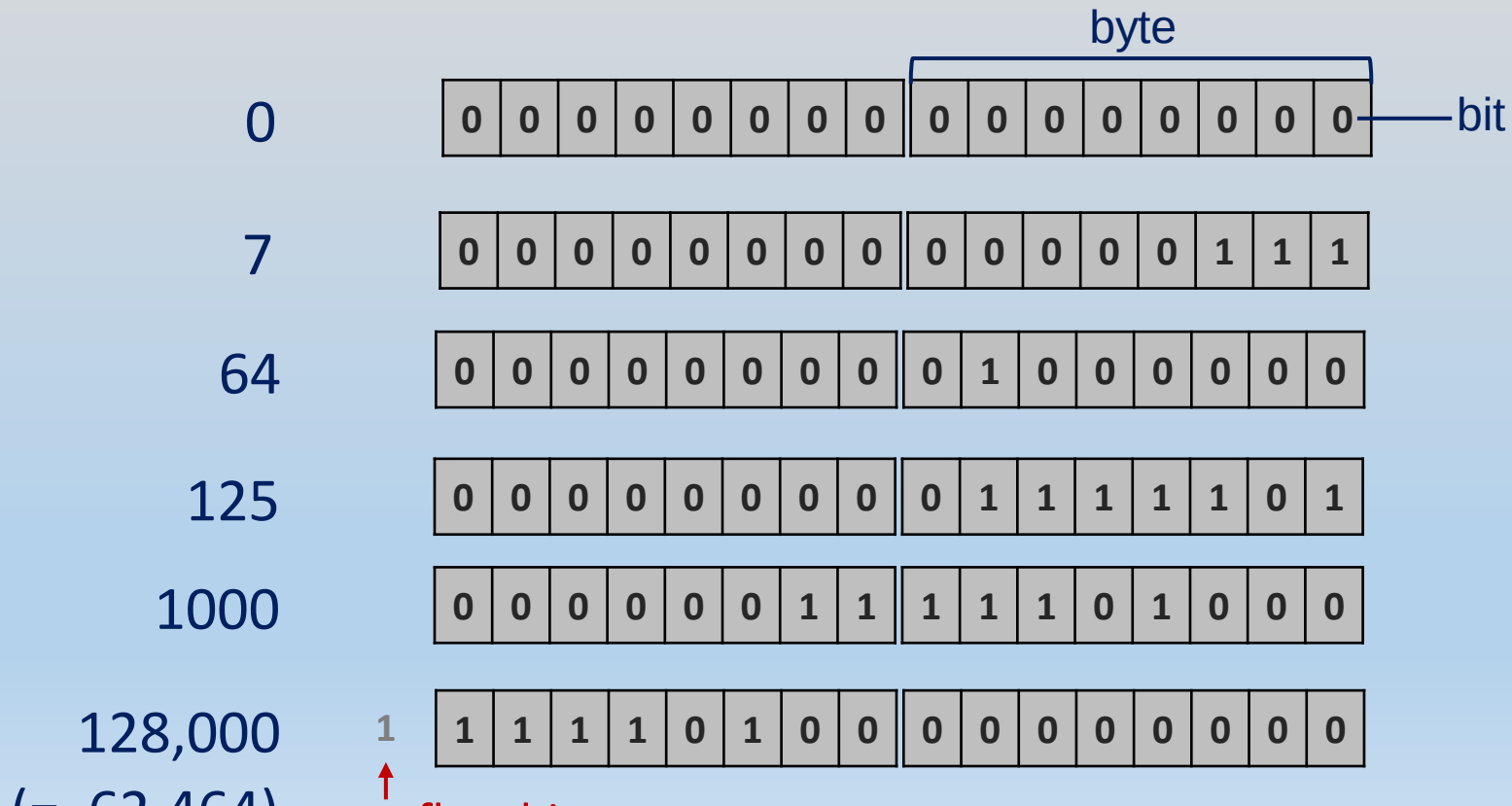
Lecture overview

- Data types
- Integer representation ←
- Floating point representation
- Literals and variables
- Operators in C

Binary representation

Unsigned integer types are used to hold non-negative numbers

- unsigned short int: (2 bytes)



Integer representation intervals

- An unsigned integer type that takes k bytes represents integer numbers in 0 … 2^{8k}-1 (only non-negative values)

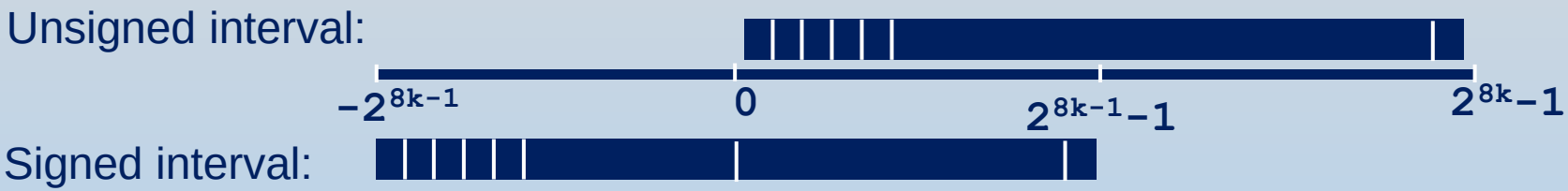
Unsigned interval:

- Examples: (assuming type size that's on our server)

char:	numbers in 0 … 2^{8}-1	(max=255)
unsigned short:	numbers in 0 … 2^{16}-1	(max=65,535)
unsigned int:	numbers in 0 … 2^{32}-1	(max≈4e+9)

Integer representation intervals

- An unsigned integer type that takes k bytes represents integer numbers in 0 … $2^{8k}-1$ (only non-negative values)
- Negative integers can be represented in signed types by "shifting" the range of representation:

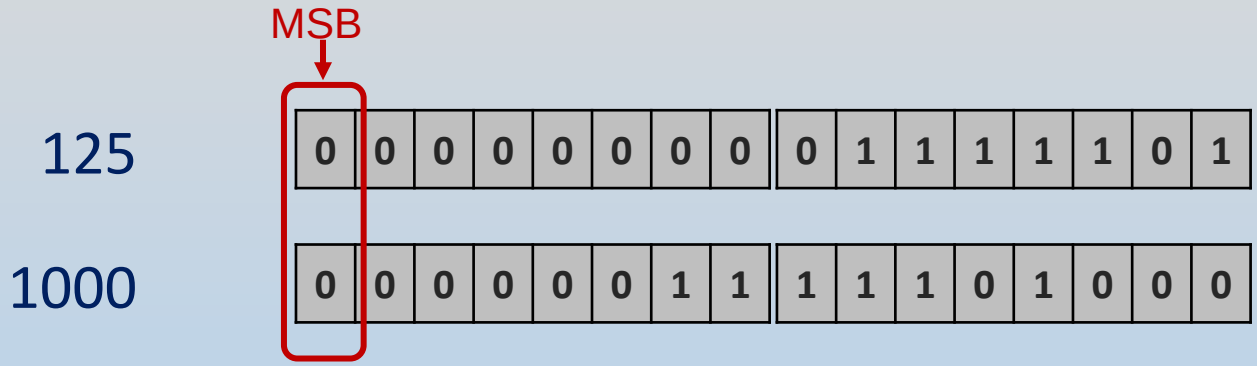


- Examples: (assuming type size that's on our server)
  signed short: numbers in -2^{15} … $2^{15}-1$ (max=32,767)
  signed int: numbers in -2^{31} … $2^{31}-1$ (max≈2e+9)

By default, integer types are signed; unsigned types are typically avoided

Binary representation

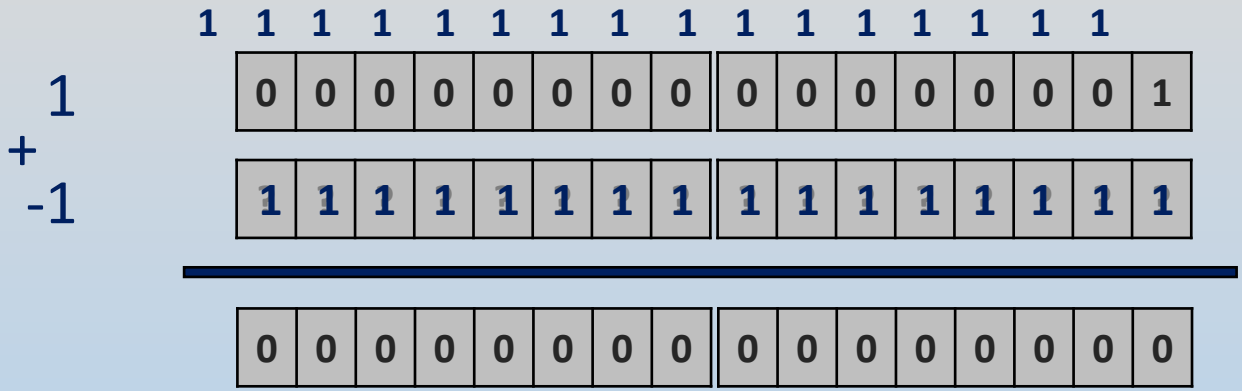
- **Positive numbers** are represented as in unsigned types with the most significant bit (MSB) = 0



- **Negative numbers** are represented with MSB = 1. But what values are put into the other bits? Answer: negative numbers are represented in **2's complement**.

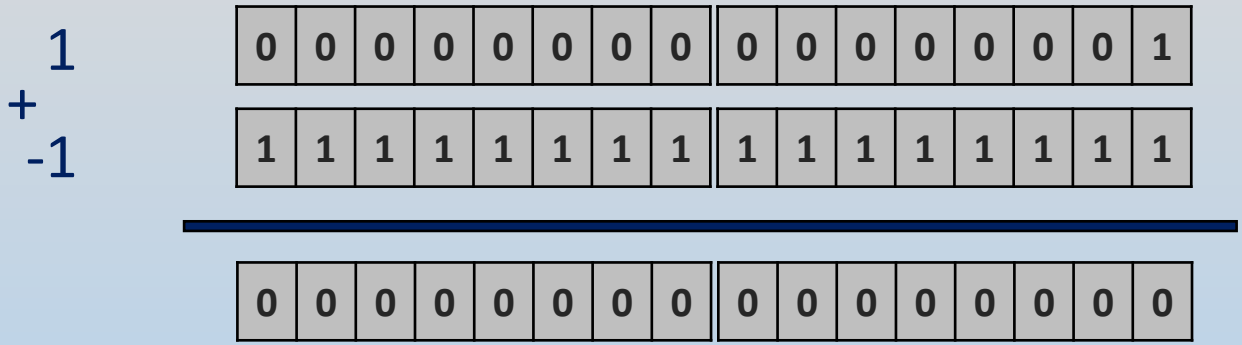
Two's complement for negative numbers

We require that x + (-x) will result in 0 in regular binary vector addition:



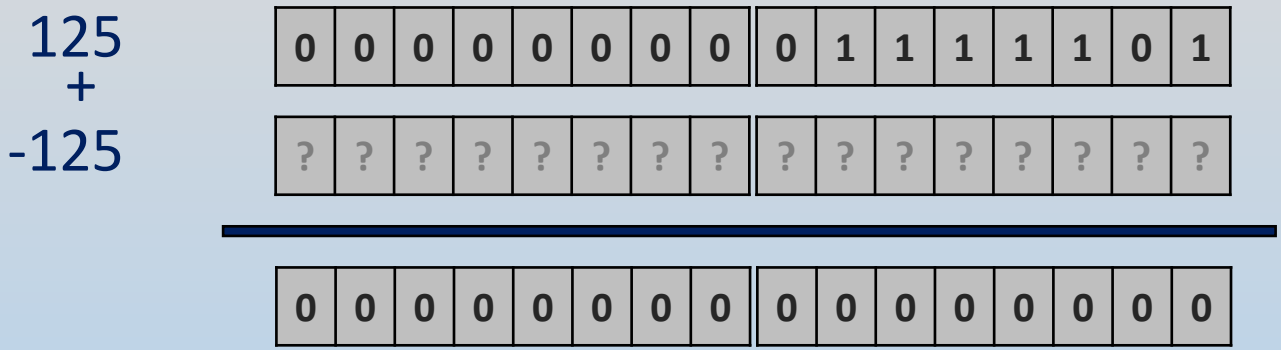
Two's complement for negative numbers

We require that x + (-x) will result in 0 in regular binary vector addition:



Two's complement for negative numbers

We require that x + (-x) will result in 0 in regular binary vector addition:



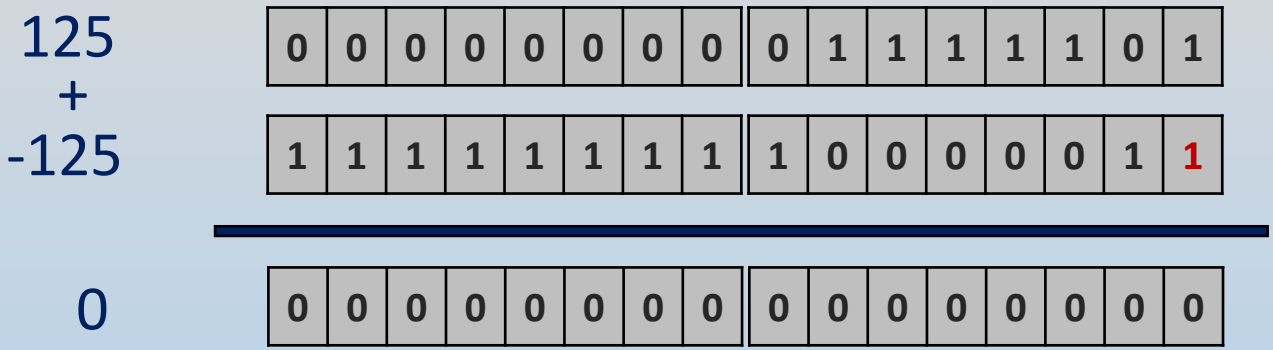
Two's complement for negative numbers

We require that x + (-x) will result in 0 in regular binary vector addition:

- A vector's complement is obtained by flipping all bits (0→1, 1→0)
- A vector + its complement equals -1

Two's complement for negative numbers

We require that x + (-x) will result in 0 in regular binary vector addition:

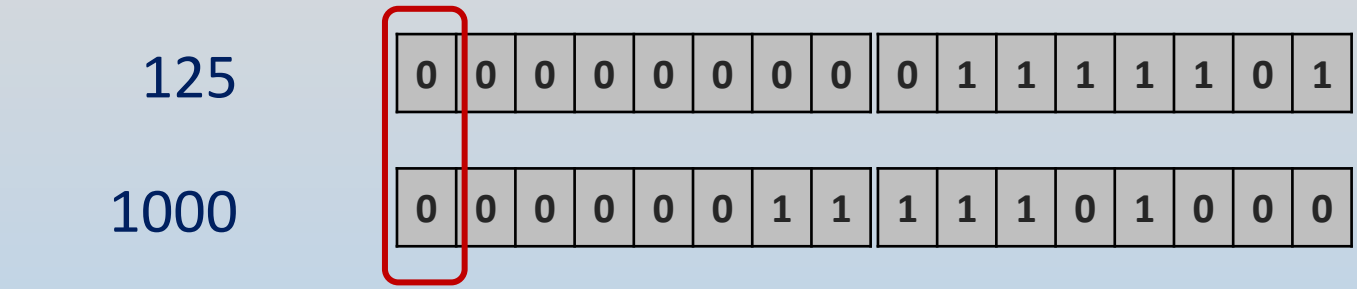


- A vector's complement is obtained by flipping all bits (0→1, 1→0)
- A vector + its complement equals -1

➔ -x = (complement of x) +1

Binary representation

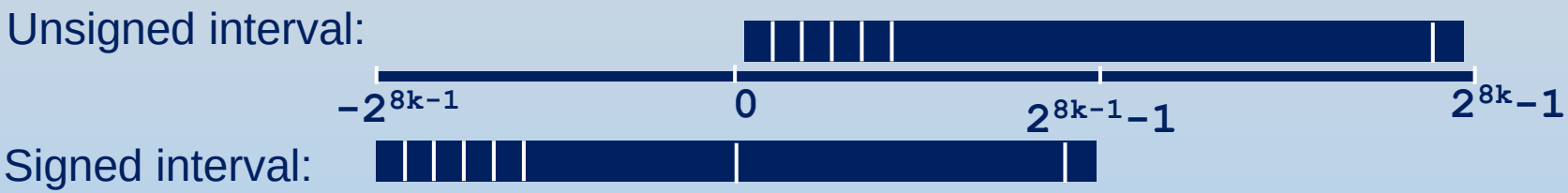
- **Positive numbers** are represented as in unsigned types with the most significant bit (MSB) = 0



- **Negative numbers** are represented in 2s complement

Integer representation intervals

- An unsigned integer type that takes k bytes represents integer numbers in 0 … $2^{8k}-1$ (only non-negative values)
- Negative integers can be represented in signed types by "shifting" the range of representation:



- Overflow in both unsigned and signed types results in "wrapping" around the representation interval

Alternative bases

It is sometimes useful to express the bit content of an integer type directly using power-2 bases

1000

01750

1 7 5 0

leading zero indicates octal representation

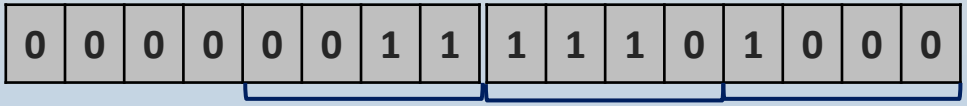
Base 8 (octal):

- Each octal digit (**0-7**) captures 3 bits

Alternative bases

It is sometimes useful to express the bit content of an integer type directly using power-2 bases

1000



0x3e8

3 e 8

leading 0x indicates hex. representation

Base 16 (hexadecimal):

- Each octal digit (**0-9,a-f**) captures 4 bits
- Hex representation is useful because each byte is captured by 2 hex digits

Online poll

What is the **octal** representation of decimal 40?

- `40`
- `101000`
- `50`
- `28`

What is the **hexadecimal** representation of decimal 40?

- `40`
- `101000`
- `50`
- `28`

What is the **hexadecimal** representation of decimal 30?

- `30`
- `36`
- `1E`
- `1D`

What is the character value of 'G'+3?

- `'j'`
- `'J'`
- `Syntax error`
- `59`
- `3`

What is the character value of 'G'+250?

- `'t'`
- `'G'`
- `'A'`
- `Syntax error`
- `250`

Online poll – solutions

What is the octal representation of decimal 40?

- `40`
- `101000`
- `50` ← ($40 = \underline{5}x8^1 + \underline{0}x8^0$)
- `28`

Online poll – solutions

What is the hexadecimal representation of decimal 40?

- `40`
- `101000`
- `50`
- `28` ← ($40 = \underline{2}x16^1 + \underline{8}x16^0$)

Online poll – solutions

What is the hexadecimal representation of decimal 30?

- `30`
- `36`
- `1E` ← ($30 = \underline{1} \times 16^1 + \underline{14} \times 16^0$ and $\underline{14}$ is represented by hex digit E)
- `1D`

Online poll – solutions

What is the character value of `'G'+3`?

- `'j'`
- `'J'` ← (ASCII values of letters are consecutive)
- `Syntax error`
- `59`
- `3`

Online poll – solutions

What is the character value of `'G'+250`?

- `'t'`
- `'G'`
- `'A'`
- `Syntax error`
- `250`

← (250 is equivalent to -6 in char, since representation has 8 bits and 2^8=256)

Lecture overview

- Data types
- Integer representation
- Floating point representation ←
- Literals and variables
- Operators in C

Floating point representation

Recall scientific notation in **decimal base**:

1.940×10^{-7}

7.345×10^{3}

(when using e=1 digits for exponent and m=4 digits for mantissa)

$$\textit{mantissa} \times 10^{\textit{exponent}}$$

exponent – integer in range [-N, N] where $N=10^{e}-1$ (e.g., N=99 if e=2)

- Determines the range of represented numbers:
  - Largest number is $9.999 \times 10^{N} \approx 10^{N+1}$
  - Smallest number (>0) is $1.000 \times 10^{-N} = 10^{-N}$

mantissa – number in the range [1,10)

- Determines relative precision, which is constant (10^{-m}) regardless of the scale of number
- Can be represented by integer in range $[10^{m-1}, 10^{m})$ (divided by 10^{m-1})

Floating point representation

Scientific notation in **binary base**:

$$mantissa \times 2^{exponent}$$ (when using e **bits** for exponent and m **bits** for mantissa)

exponent – integer in range [-N, N] where $\mathbf{N=2^{e-1}-1}$ (e.g., N=127 if e=8)

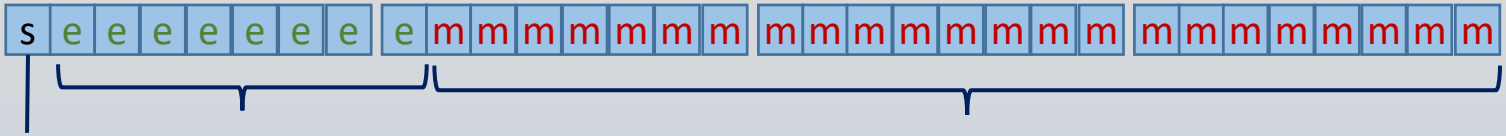
- Determines the range of represented numbers:
  - Largest number is $\approx \mathbf{2^{N+1}}$
  - Smallest number (>0) is $= \mathbf{2^{-N}}$

mantissa – number in the range [1,2)

- Determines relative precision, which is constant ($\mathbf{2^{-m}}$) regardless of the scale of number
- can be represented by integer in range $\mathbf{[0 , 2^m)}$ (divided by 2^m **plus 1**)

Floating point numbers (IEEE 754)

Type `float`:



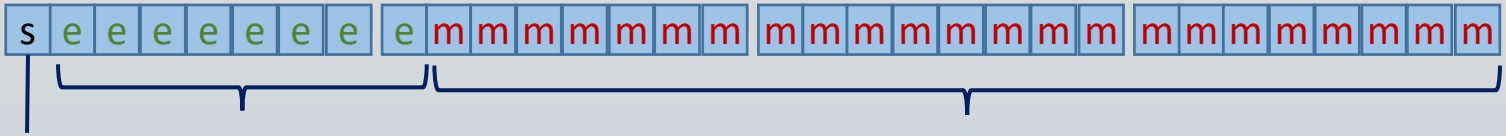
0 – pos

1 – neg

		`float`
mantissa bits	(m)	23
exponent bits	(e)	8
largest exponent	$(N=2^{e-1}-1)$	127
relative precision	(2^{-m})	1.19 e-07
largest number	(2^{N+1})	3.40 e+38
smallest number in range	(2^{-N+1})	1.18 e-38
smallest number > 0	(2^{-N+1-m})	1.40 e-45

Floating point numbers (IEEE 754)

Type `float`:



0 – pos

1 – neg

		`float`	`double`
mantissa bits	(m)	23	52
exponent bits	(e)	8	11
largest exponent	$(N=2^{e-1}-1)$	127	1023
relative precision	(2^{-m})	1.19 e-07	2.22 e-016
largest number	(2^{N+1})	3.40 e+38	1.80 e+308
smallest number in range	(2^{-N+1})	1.18 e-38	2.23 e-308
smallest number > 0	(2^{-N+1-m})	1.40 e-45	5.00 e-324

Lecture overview

- Data types
- Integer representation
- Floating point representation
- Literals and variables ←
- Operators in C

Literals

- A literal represents an explicit value of a certain type

Literal	Data Type	
`'c'`	char	- single character in `' '`
`178`	int	- integer number
`0262` (=178)	int	- octal representation (prefix zero)
`0xb2` (=178)	int	- hex. representation (prefix zero+x)
`178L`	long	- use suffix (L / U / UL)
`178U`	unsigned	
`178UL`	unsigned long	
`178.2`	double	- number with decimal point
`178.0`	double	
`1.78e2`	double	- scientific notation
`178D`	double	- number followed with D
`178.2F`	float	- number followed with F
`178F`	float	

Variables

- In C, when variable is defined then it is associated with specific type
- Define variables in the beginning of code block (e.g., function, loop)
- Can have one statement to define multiple vars of same type and initialize their values

```
int main() {
   int i, numIter=100;


   for(i=0;i<numIter;i++) {
      float x;
      printf("Enter a real number: ");
      while(1 > scanf("%f",&x))
        ;
      printf("%f^2 = %f\n",x,x*x);
   }
   return 0;
}
```

! Unlike Java !

- Note: should not define iterator variable inside **`for`** statement

Lecture overview

- Data types
- Integer representation
- Floating point representation
- Literals and variables
- Operators in C ←

Operators

C supports the same operators as Java:

- Bitwise `&, |, ~, ^, << , >>`
- Relational and logical `>, >=, <, <=, ==, !=, &&, ||`
- Arithmetic `+, -, *, /, %`
- Assignment `=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=`
- Increment / decrement `++, --`

Bitwise operators

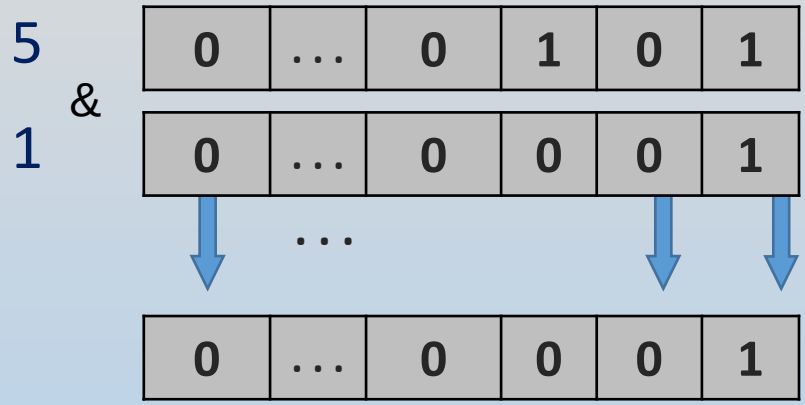
`&, |, ~, ^, << , >>`

- Logical operators on bits (0 – FALSE 1 – TRUE)
  - and (`&`) `x&y == 1` iff `x==y==1` binary
  - or (`|`) `x|y == 0` iff `x==y==0` binary
  - not (`~`) `~x == -x-1` unary
  - xor (`^`) `x^y == 0` iff `x==y` binary

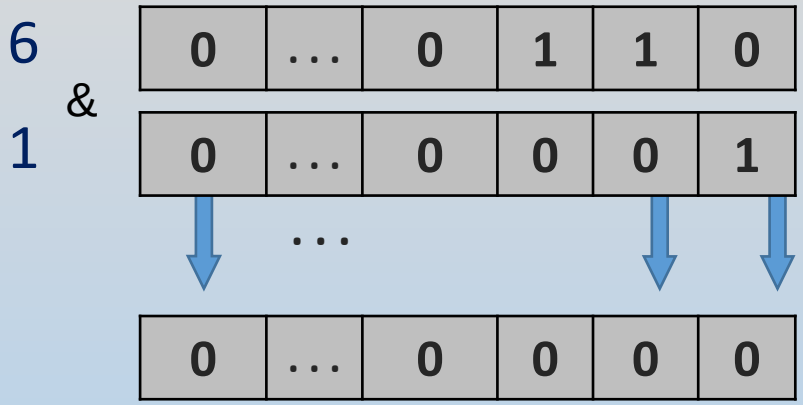
- Bitwise logical operators (`&`, `|`, `~`, `^`) perform bit operations on corresponding bits of two integers (bit vectors)

Bitwise operators – examples &, |, ~, ^, << , >>

5 & 1

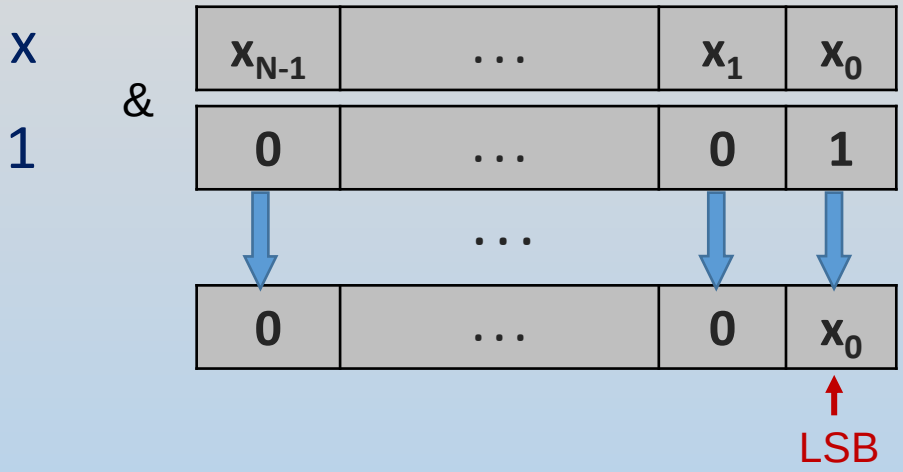


6 & 1



Bitwise operators – examples &, |, ~, ^, << , >>

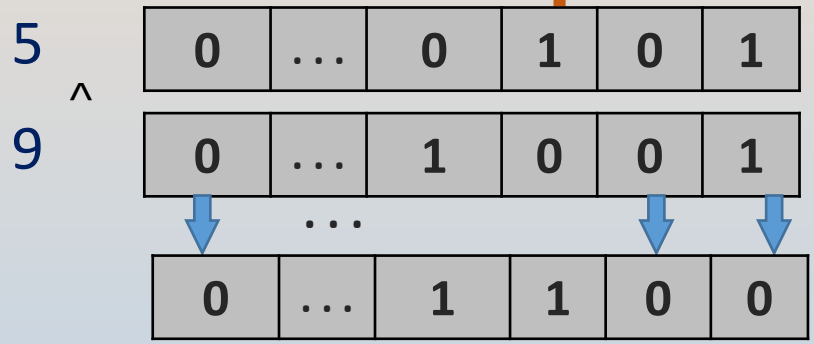
x & 1



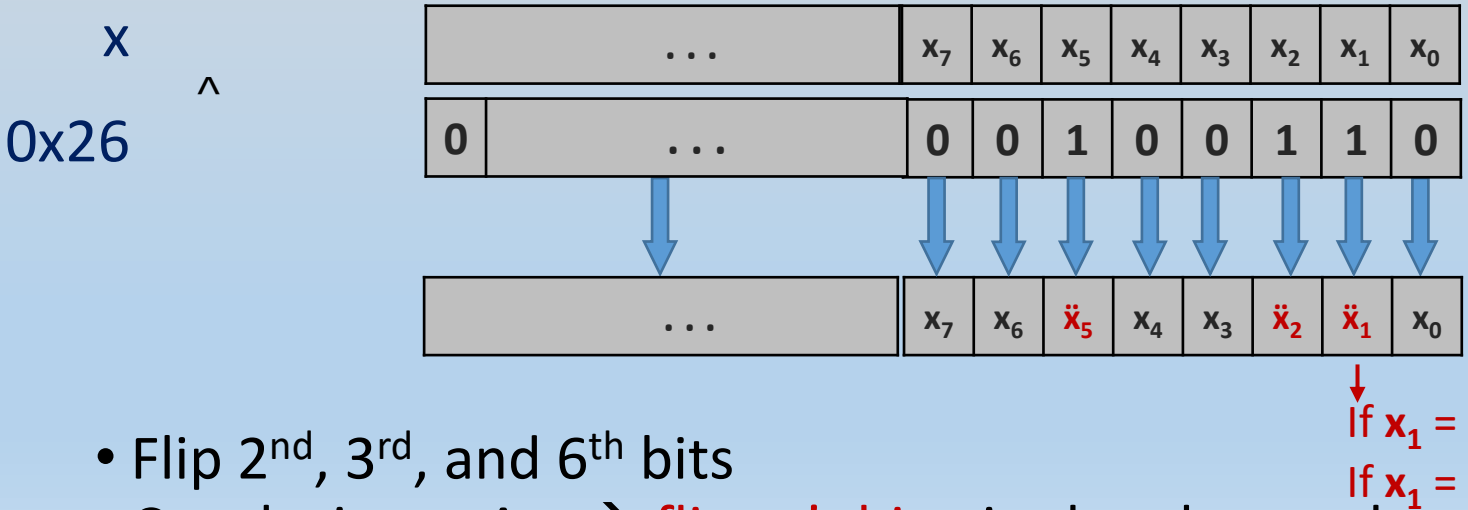
- Conclusion: x & 1 → gets least significant bit (LSB; or parity bit)

Bitwise operators – examples &, |, ~, ^, <<, >>

5 ^ 9 = 12 (0xC)



x ^ 0x26



- Flip 2nd, 3rd, and 6th bits
- Conclusion: x ^ y → flips x's bits, in the places where y has a 1 bit
  - x ^ 0 = x

Bitwise operators

`&, |, ~, ^, << , >>`

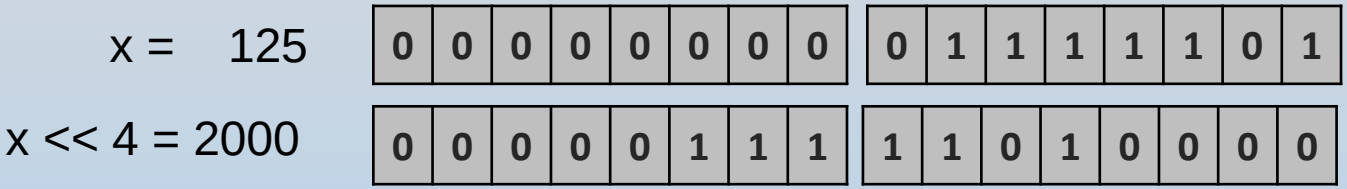
- Logical operators on bits (0 – FALSE 1 – TRUE)
  - and (`&`) `x&y == 1` iff `x==y==1` binary
  - or (`|`) `x|y == 0` iff `x==y==0` binary
  - not (`~`) `~x == -x-1` unary
  - xor (`^`) `x^y == 0` iff `x==y` binary

- Bitwise logical operators (`&`, `|`, `~`, `^`) perform bit operations on corresponding bits of two integers (bit vectors)
- Bit shift operators shift all bits in vector left (`<<`) or right (`>>`)

Bitwise operators – examples &, |, ~, ^, << , >>

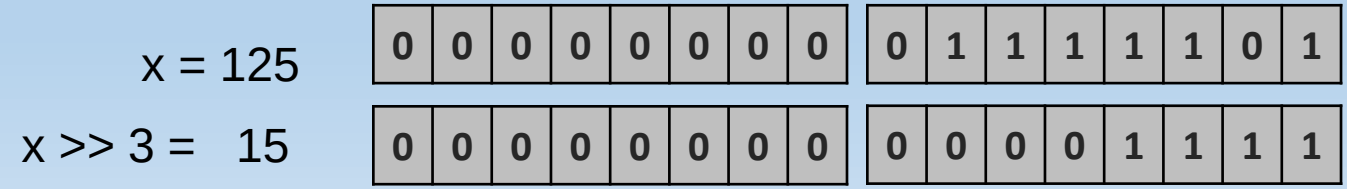
x << 4

- Shift all bits four positions to left
  ➔ multiply by 16=2^4 (could overflow)



x >> 3

- Shift all bits three position to right
  ➔ divide by 8 (without remainder)



Bit shift operators

<< , >>

Efficient implementation of multiplication and division by powers of 2

Left shift **<<** (mult by 2^n):

- Inserted LSB is 0
- Overflow of MSB occurs when product exceeds max value

Right shift **>>** (div by 2^n):

- Overflow of LSB occurs when division has remainder
- Inserted MSB is 0 when operator is applied to unsigned types
- Inserted MSB is **copied** from original vector when operator is applied to signed types.

  Example: -2 >> 1 yields -1 (sign is preserved)

Online poll

What is the value of 3&6 ? (bitwise AND)

- 1
- 2
- 3
- 5
- 6
- 7

What is the value of 3|6 ? (bitwise OR)

- 1
- 2
- 3
- 5
- 6
- 7

What is the value of 3^6 ? (bitwise XOR)

- 1
- 2
- 3
- 5
- 6
- 7

What is the value of (-3)&6 ? (bitwise AND)

- -2
- -1
- 1
- 2
- 4
- 6

What is the value of 3<<4 ? (shift left)

- 7
- 12
- 48
- 81

What is the value of 7>>2 ? (shift right)

- 1
- 3
- 5
- 7

What is the value of (-7)>>2 ? (shift right)

- -2
- -1
- 0
- 1
- 3

Online poll – solutions

What is the value of 3&6 ? (bitwise AND)

- 1
- 2 ←
- 3
- 5
- 6
- 7

```
  3 = 0 … 0 0 1 1
  6 = 0 … 0 1 1 0
-------------------
3&6 = 0 … 0 0 1 0 = 2
```

Online poll – solutions

What is the value of 3|6 ? (bitwise OR)

- 1
- 2
- 3
- 5
- 6
- 7 ←

```
  3 = 0 … 0 0 1 1
  6 = 0 … 0 1 1 0
-------------------
3|6 = 0 … 0 1 1 1 = 7
```

Online poll – solutions

What is the value of 3^6 ? (bitwise XOR)

- 1
- 2
- 3
- 5 ←
- 6
- 7

```
  3 = 0 … 0 0 1 1
  6 = 0 … 0 1 1 0
-------------------
3^6 = 0 … 0 1 0 1 = 5
```

(Note that a^b + a&b = a|b)

Online poll – solutions

What is the value of (-3)&6 ? (bitwise AND)

- -2
- -1
- 1
- 2
- 4 ←
- 6

```
 -3 = 1 … 1 1 0 1
  6 = 0 … 0 1 1 0
-3&6 = 0 … 0 1 0 0 = 4
```

Online poll – solutions

What is the value of 3<<4 ? (shift left)

- 7
- 12
- 48 ←
- 81

```
  3 = 0 … 0 0 0 0 1 1
3<<4 = 0 … 1 1 0 0 0 0 = 48
```

(= $3x2^4$)

Online poll – solutions

What is the value of 7>>2 ? (shift right)

- 1 ←
- 3
- 5
- 7

```
  7 = 0 … 0 0 0 1 1 1
7>>2 = 0 … 0 0 0 0 0 1 = 1
```

$(= 7/2^2)$

Online poll – solutions

What is the value of (-7)>>2 ? (shift right)

- -2 ←
- -1
- 0
- 1
- 3

```
   -7 = 1 … 1 1 1 0 0 1
-7>>2 = 1 … 1 1 1 1 1 0 = -2
```

(= $(-7)/2^2$)

Relational operators

>, >=, <, <=, ==, !=

- Operations compare values of same type
- Return value is of type int
  - 0 - FALSE
  - 1 - TRUE

There is no Boolean type in C !! *! Unlike Java !*

```
char c='C', d='D';
int  a = (c>d);
```

← variable a equals 0

Logical operators

`&&, ||, !`

- Return value is of type int
  - 0 - FALSE
  - 1 - TRUE
- Receive int operands
  - 0 - FALSE
  - any non-zero value (including negative values) - TRUE

```
char c='C', d='D';
int  a = (c>d) || 3;
```

← variable a equals 1

Note: Conditional statements (e.g., **`if`** , **`while`**) receive an int value and evaluate to TRUE iff value != 0

- **`if (i!=0)`** same as **`if(i)`**

Logical operators – short circuits

As in Java, short circuiting is applied when evaluating logical operators `&&` and `||`

- 2nd, 3rd . . . terms of composite logical expression may not be computed (and side effects may not take place)
- Often used to implement series of validity tests

Logical operators – short circuits

Example: write an if statement's condition that tests if `i/j` is odd and `j!=0` (otherwise `i/j` is undefined)

- Option 1: (triggers runtime error when `j==0`)

```
if(((i/j)%2==1) && (j!=0)) { ... }
```

- Option 2: (short circuits when `j==0`)

```
if((j!=0) && ((i/j)%2==1)) { ... }
```

```
if((j)    && ((i/j)%2))    { ... }
```

(these 2 are equivalent)

Arithmetic operators

+, −, *, /, %

- Each type has its own version of each arithmetic operator
- In particular, operations on integer types are not the same as operations on floating point numbers
- Division '/' is actually a different operation in integers and floating-point numbers

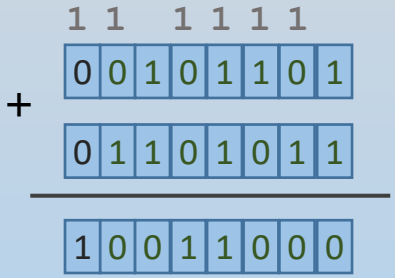
➔ 3/4 returns 0

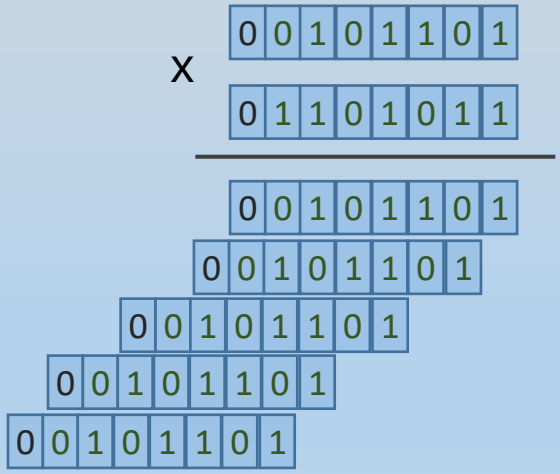
➔ 3.0/4.0 returns 0.75

Complexity of arithmetic operators

Integer arithmetic implementation:

- sum / difference is **linear** in size of representation (number of bits)
- product / division / modulo are **quadratic** in size or representation
- Keep in mind possible **overflow** and loss of information





Floating point arithmetic is more complicated (we don't see here)

Operator complexity – summary

- Relational operators (e.g., `>`) involve simple comparisons that can be done in linear time or less
- Logical operators (e.g., `&&`) involve comparing a series of integers to (linear or less)
- Bitwise operators (e.g., `|`) typically involve linear scans of the bit vector
- Integer arithmetic operators are linear (e.g., +,-) or quadratic (*,/)
- Floating point arithmetic operations are more complex

Assignment operators

=, +=, -=, *=, /=, %=,
&=, |=, ^=, <<=, >>=

- Assignment operators are binary
- Left operand must be a variable (target of assignment)
- Operator has side effect – variable value changed

```
int a=3;

a *= a;
```

← side effect: a's value is changed to 3

← side effect: a's value is changed to 9 (`a = a * a;`)

```
int a=3;

a <<= 1;
```

← side effect: a's value is changed to 3

← side effect: a's value is changed to 6 (`a = a << 1;`)

```
int a=3;

a |= 3;
```

← side effect: a's value is changed to 3

← side effect: a's value is changed to 3 (`a = a | 3;`)

Increment/decrement operators ++,--

- Inc/dec operators are unary
- Single operand must be a variable
- Operator has side effect – value inc/dec by 1
- What is the difference between **++x** and **x++**?

- Example 1:

```
double x = 1.2;
double y = ++x;
double z = x++;
```

← y = 2.2
← z = 2.2

Increment/decrement operators ++,--

- Inc/dec operators are unary
- Single operand must be a variable
- Operator has side effect – value inc/dec by 1
- What is the difference between **++x** and **x++**?

- Example 2:

```
int a = -1;
if (a) ;
if (a++) ;
if (++a) ;
```

vs.

```
int a = -1;
if (a) ;
if (++a) ;
if (a++) ;
```

Increment/decrement operators ++,--

- Inc/dec operators are unary
- Single operand must be a variable
- Operator has side effect – value inc/dec by 1
- What is the difference between **++x** and **x++**?

- Example 3 (involving short-circuit evaluation):

```
int i;
i = 7; if (i==3 && ++i); //i = 7
i = 7; if (i==7 && ++i); //i = 8
i = 7; if (i==3 || ++i); //i = 8
i = 7; if (i==7 || ++i); //i = 7
```

Multiple operators

- General question: how do we parse expressions that involve multiple operators?

```
int a = 3 / 4.0;
```

```
y = ++x;
```

```
a = c<d << 3;
```

```
float f = 3/4;
```

```
printf ("%d\n", a<a<<2 && a-7);
```

```
if(a<c || --b); vs. if(a < c||--b);
```

```
int a = b -= 2;
```

At home…

- Lecture #6 will be given by Zoom:
  - Ilan's groups (both of them) – Tuesday, May 9
  - Sara's group – Tuesday, May 16
- 3rd homework due Sunday, May 12, @ 9 p.m.
- 4th homework will be published on May 16.
- Review guide on control flow statements in C by next week.