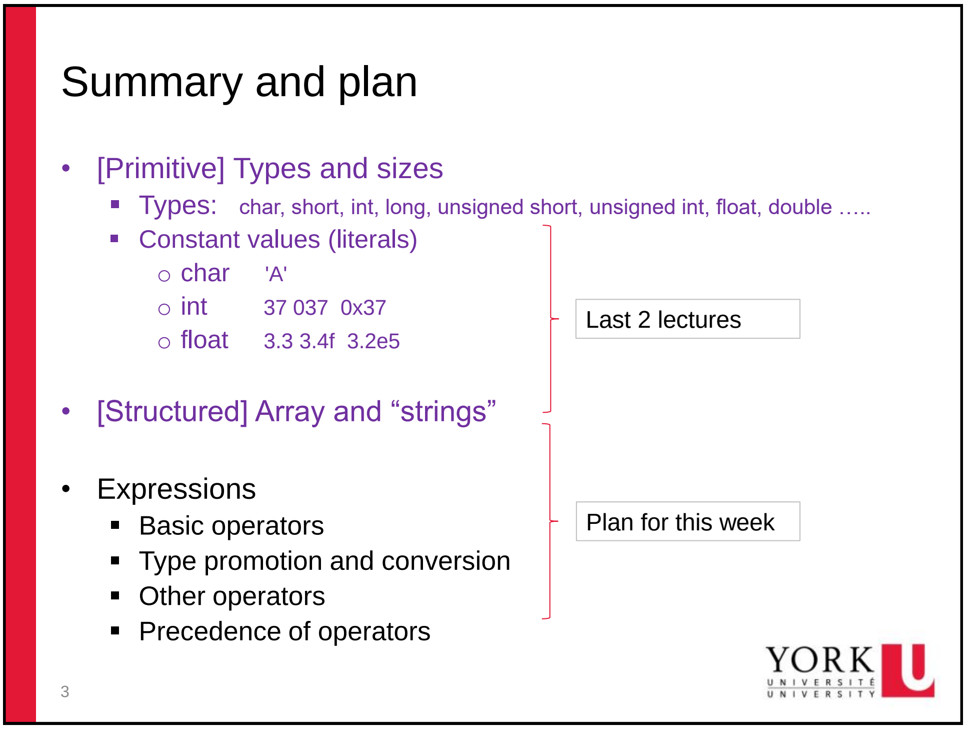
EECS2031 A Software Tools

Su 2021

Week3 Sep 22/23, 2021 Lecture 5

1



Summary and plan

- [Primitive] Types and sizes
  - Types: char, short, int, long, unsigned short, unsigned int, float, double …..
  - Constant values (literals)
    - char 'A'
    - int 37 037 0x37
    - float 3.3 3.4f 3.2e5
- [Structured] Array and “strings”

Last 2 lectures

- Expressions
  - Basic operators
  - Type promotion and conversion
  - Other operators
  - Precedence of operators

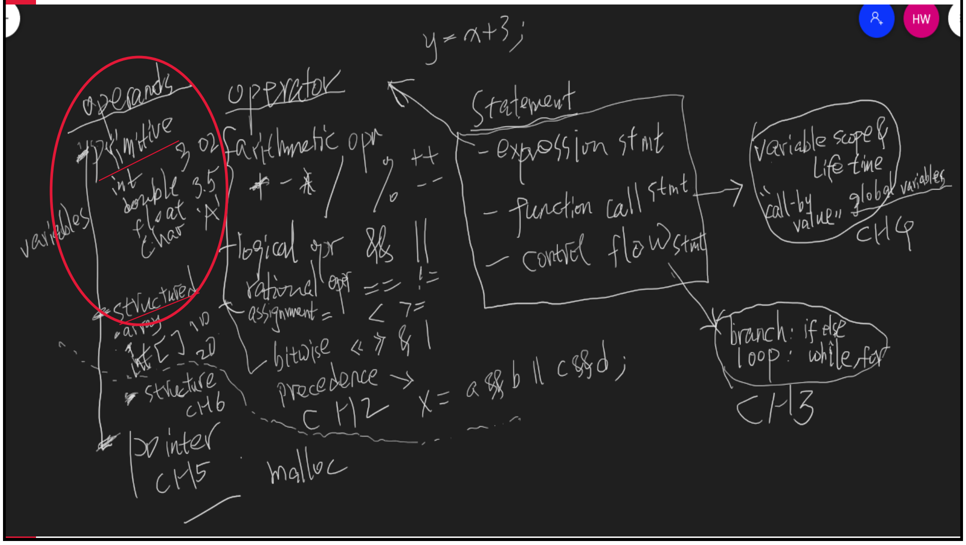
Plan for this week

3

3

Roadmap -- How the topics are related

RECALL



4

C (Primitive)Types & sizes

Text book:
4 basic types: char, int, float, double

3 qualifiers: short, long, unsigned

- Variables and values have types

- There are two basic types in ANSI-C: integer, and floating point

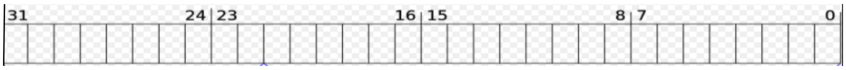
  - **Integer type**
    - ○ **char** - character, 1 byte (8 bits)
    - ○ **short (int)** - short integer, usually 2 bytes (16 bits)
    - ○ **int** - integer, usually 2 or 4 bytes (16 or 32 bits)
    - ○ **long (int)** - long integer, usually 4 or 8 bytes (32 or 64 bits)

  - **Floating point**
    - ○ **float** - single-precision, usually 4 bytes (32 bits)
    - ○ **double** - double-precision, usually 8 bytes (64 bits)
    - ○ **long double** - extended-precision

YORK UNIVERSITÉ UNIVERSITY

6

Qualifiers (modifiers) for integer type



Assume all 32 bits are magnitudes.

- Max value: 1111111….11111
- Min value: 0000000….00000
- How many values: 2^{32} values 2^n
- Range: 0~2^{32}-1 0~2^n-1

Negative number?

7

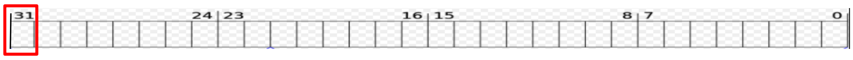
7

Qualifiers (modifiers) for integer type

- signed, unsigned qualifiers can be applied to integer types
  - Signed: default. Positive/negative. Left most bit signifies sign
    0: positive 1: negative
  - Unsigned: positive only. Left most bit contributes to magnitude too

```
▪ (signed) char
▪ (signed) int
▪ (signed) short int
▪ (signed) long  int

▪ unsigned char
▪ unsigned int
▪ unsigned short int
▪ unsigned long  int
```

Java: no direct support for unsigned int -- always signed



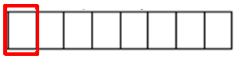
`unsigned int`	0~2^{32}-1	2^{32} values	Max: 1111111….11111
`(signed) int`	-2^{31}~2^{31}-1	2^{32} values	Max: 0111111….11111
	-2^{n-1} ~ 2^{n-1}-1	2^n values	Min?

8

8

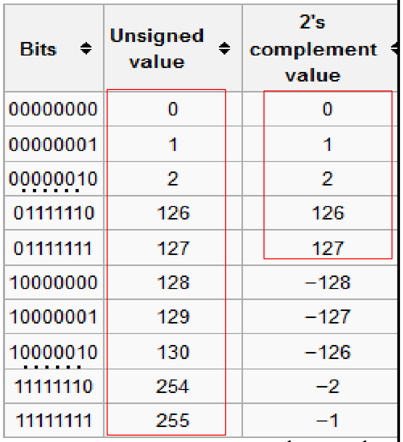
Qualifiers (modifiers) for integer type

- **signed/unsigned** can be applied to char
  - **signed** char $-2^7 \sim 2^7-1$ /* -128 ~~ 127 */
  - **unsigned** char 0 $\sim 2^8-1$ /* 0 ~~ 255 */



- 2's complement: "flip + 1"
  - -2's binary representation?
    - 2's binary representation flip + 1
    - 11111101 + 1 = 11111110
  - 11111110's decimal?
    - - (flip +1)
    - - (00000001+1) = -(00000010) = -2

signed value



Bits	Unsigned value	2's complement value
00000000	0	0
00000001	1	1
00000010	2	2
01111110	126	126
01111111	127	127
10000000	128	−128
10000001	129	−127
10000010	130	−126
11111110	254	−2
11111111	255	−1

$0 \sim\sim 2^n-1$ $\quad -2^{n-1} \sim\sim 2^{n-1}-1$

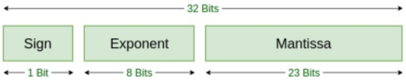
2^n=256 values $\quad 2^n$=256 values

Qualifiers for floating points

- "`long`" can be used with double:
  - `long double`
- Thus, there are three types of floating points:
  - `float` `/* single-precision floating point */`
  - `double` `/* double-precision floating point */`
  - `long double` `/* extended-precision floating point */`
- More bits, more precise.
  - 3.1415926535….

- scanf ("%f") for float, ("%lf") for double, ("%Lf") for long double
- printf ("%f") for float, double or %lf double ("%Lf") for long double

- Storage of floating point is complicated.
  - `float x=4.8, float y = 6.4/2+1.6; x== y` may not always true.
- No unsigned. All signed



Summary

- Integer types:

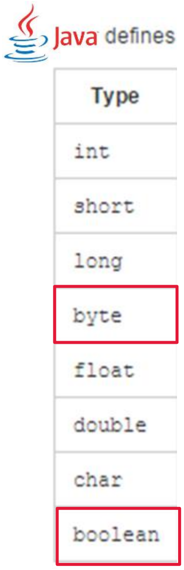
```
▪ char
  signed char           unsigned char
▪ (signed) short       unsigned short
▪ (signed) int         unsigned int
▪ (signed) long        unsigned long
```

- There are three types of floating points:

```
▪ float        /* single-precision */
▪ double       /* double precision */
▪ long double  /* extended-precision */
```

- C99 added:

```
▪ (signed) long long int
▪ unsigned long long int bool
```

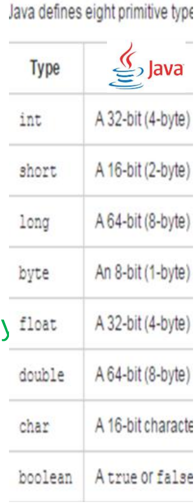

Java defines

Type
int
short
long
byte
float
double
char
boolean

Size of Types

- Exact sizes of types depend on machine

char	= 8 bits	[for sure]	1 byte
short	≥ 16 bits	[usually 16 bits]	2 bytes
int	≥ 16 bits	[usually 32 bits]	4 bytes
long	≥ 32 bits	[usually 32 or 64 bits]	4 or 8 by
float	≥ 32 bits	[usually 32 bits]	4 bytes
double	≥ 64 bits	[usually 64 bits]	8 bytes


Java defines eight primitive type

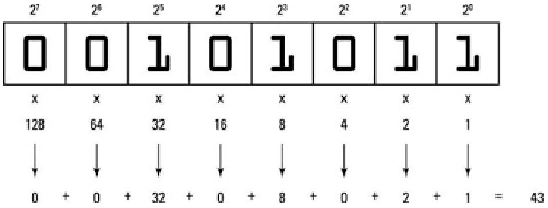
Type	Java
int	A 32-bit (4-byte)
short	A 16-bit (2-byte)
long	A 64-bit (8-byte)
byte	An 8-bit (1-byte)
float	A 32-bit (4-byte)
double	A 64-bit (8-byte)
char	A 16-bit characte
boolean	A true or false

- Relations of sizes:
  - **short ≤ int ≤ long**
  - **float ≤ double ≤ long double**

- To get exact size of a type in a machine, use **sizeof** operator
  - **sizeof (int)** or **int a; sizeof a**; or **sizeof (a)**

In Java, no direct equivalent

Internal representation of characters

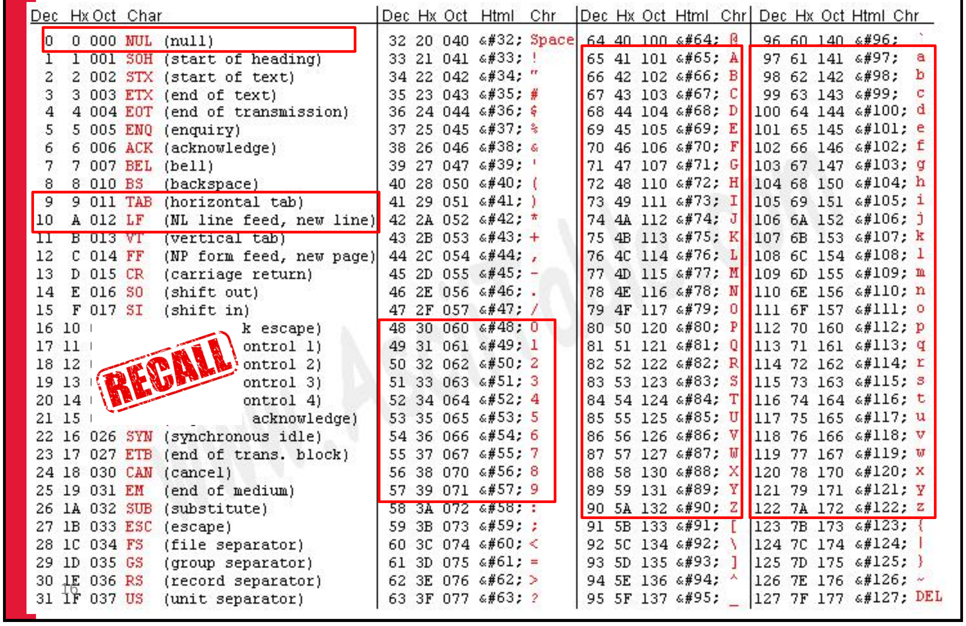


```
int i =  43;

char a = 'A';
```

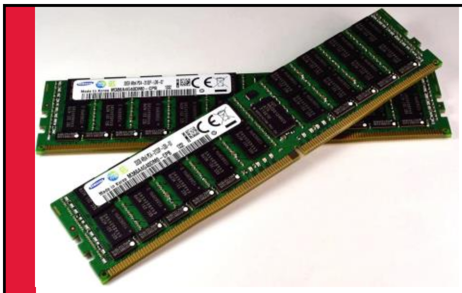
How to represent 'A' using 0s and 1s

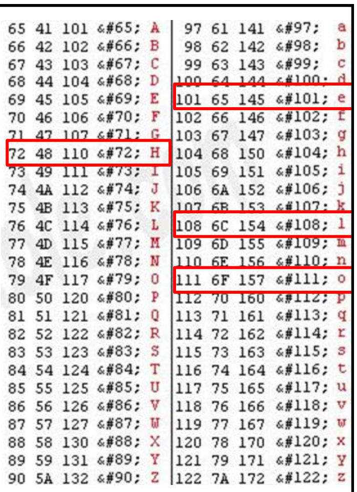
01100101 01101100 01101100 01101111 00000000

Internal Representation of Characters



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10		k escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11		ontrol 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12		ontrol 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13		ontrol 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14		ontrol 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15		acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL





65	41	101	&#65;	A	97	61	141	&#97;	a
66	42	102	&#66;	B	98	62	142	&#98;	b
67	43	103	&#67;	C	99	63	143	&#99;	c
68	44	104	&#68;	D	100	64	144	&#100;	d
69	45	105	&#69;	E	101	65	145	&#101;	e
70	46	106	&#70;	F	102	66	146	&#102;	f
71	47	107	&#71;	G	103	67	147	&#103;	g
72	48	110	&#72;	H	104	68	150	&#104;	h
73	49	111	&#73;	I	105	69	151	&#105;	i
74	4A	112	&#74;	J	106	6A	152	&#106;	j
75	4B	113	&#75;	K	107	6B	153	&#107;	k
76	4C	114	&#76;	L	108	6C	154	&#108;	l
77	4D	115	&#77;	M	109	6D	155	&#109;	m
78	4E	116	&#78;	N	110	6E	156	&#110;	n
79	4F	117	&#79;	O	111	6F	157	&#111;	o
80	50	120	&#80;	P	112	70	160	&#112;	p
81	51	121	&#81;	Q	113	71	161	&#113;	q
82	52	122	&#82;	R	114	72	162	&#114;	r
83	53	123	&#83;	S	115	73	163	&#115;	s
84	54	124	&#84;	T	116	74	164	&#116;	t
85	55	125	&#85;	U	117	75	165	&#117;	u
86	56	126	&#86;	V	118	76	166	&#118;	v
87	57	127	&#87;	W	119	77	167	&#119;	w
88	58	130	&#88;	X	120	78	170	&#120;	x
89	59	131	&#89;	Y	121	79	171	&#121;	y
90	5A	132	&#90;	Z	122	7A	172	&#122;	z

'H'	'e'
72	101
01001000	01100101

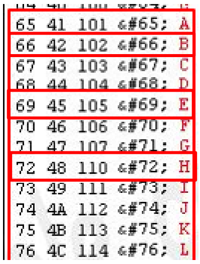
'H'	'e'	'l'	'l'	'o'
72	101	108	108	111
01001000	01100101	01101100	01101100	01101111

YORK U UNIVERSITÉ UNIVERSITY

Characters

RECALL

- **chars** are treated in C as <u>small integers</u>, **char** variables and constants are identical to **int** in arithmetic expressions:
  - **char c** is converted to its encoding (index in the character set table)

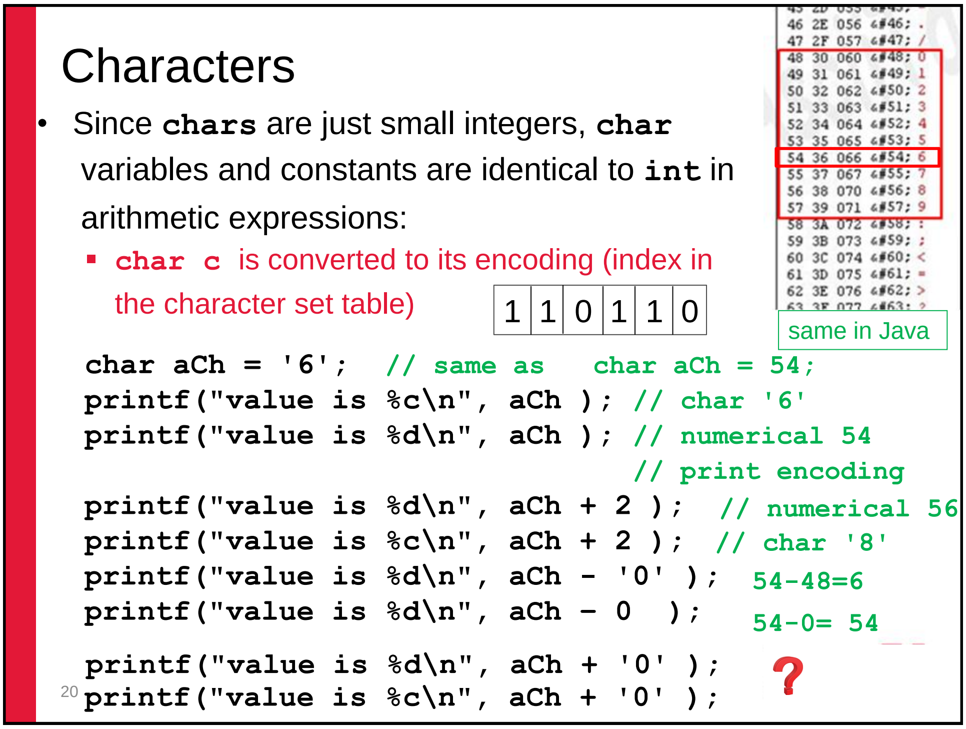


65	41	101	&#65;	A
66	42	102	&#66;	B
67	43	103	&#67;	C
68	44	104	&#68;	D
69	45	105	&#69;	E
70	46	106	&#70;	F
71	47	107	&#71;	G
72	48	110	&#72;	H
73	49	111	&#73;	I
74	4A	112	&#74;	J
75	4B	113	&#75;	K
76	4C	114	&#76;	L

```
int aChar = getChar();   // read 'E' encoding 69
aChar + 8        // expression with value 69+8 = 77
aChar + 'B'        // expression with value 69+66 = 135
aChar - 'B'        // expression with value 69-66 = 3
```

- Same for other expressions. In relational expression, characters can be compared directly, comparing indexes/encodings

```
aChar == EOF      // index == -1?  ➔ expr with value 0 (false)

aChar == 'H'      // index == 72?  ➔ expr with value 0 (false)

aChar == '\n'     // index == 10?  ➔ expr with value 0 (false)

aChar < 'H' // 69 < 72? Earlier in table? ➔ expr with 1 (true)
aChar < 72
```



Characters

- Since **chars** are just small integers, **char** variables and constants are identical to **int** in arithmetic expressions:
  - **char c** is converted to its encoding (index in the character set table)

1	1	0	1	1	0

46	2E	056	&#46;	.
47	2F	057	&#47;	/
48	30	060	&#48;	0
49	31	061	&#49;	1
50	32	062	&#50;	2
51	33	063	&#51;	3
52	34	064	&#52;	4
53	35	065	&#53;	5
54	36	066	&#54;	6
55	37	067	&#55;	7
56	38	070	8	8
57	39	071	&#57;	9
58	3A	072	&#58;	:
59	3B	073	&#59;	;
60	3C	074	&#60;	<
61	3D	075	&#61;	=
62	3E	076	&#62;	>

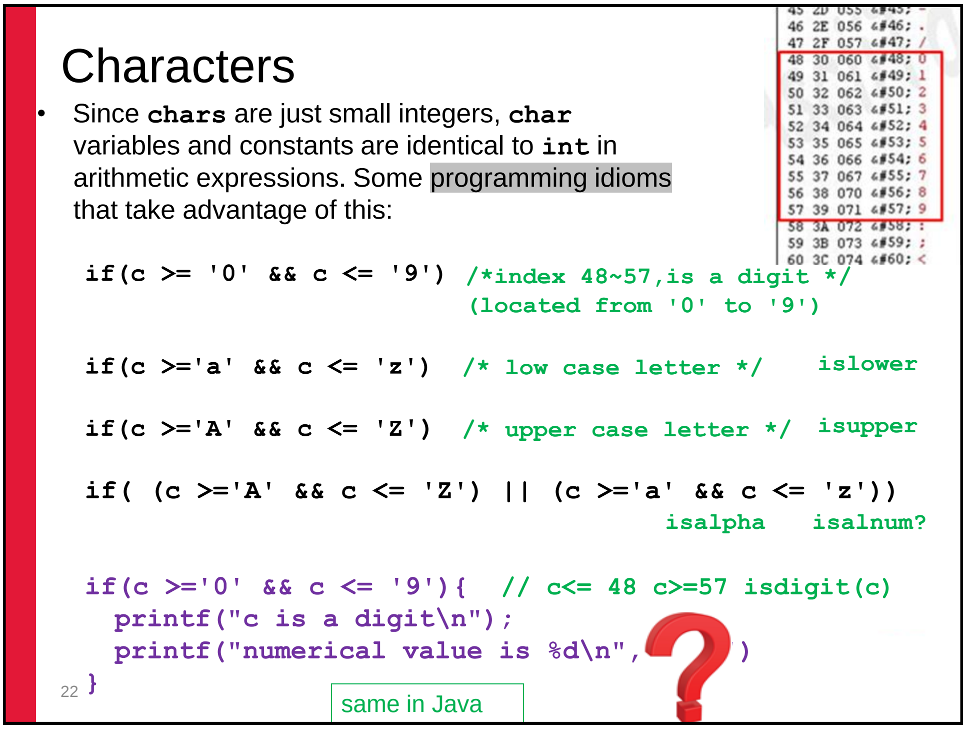
same in Java

```
char aCh = '6';   // same as   char aCh = 54;
printf("value is %c\n", aCh ); // char '6'
printf("value is %d\n", aCh ); // numerical 54
                               // print encoding
printf("value is %d\n", aCh + 2 );  // numerical 56
printf("value is %c\n", aCh + 2 );  // char '8'
printf("value is %d\n", aCh - '0' );  54-48=6
printf("value is %d\n", aCh - 0  );   54-0= 54

printf("value is %d\n", aCh + '0' );
printf("value is %c\n", aCh + '0' );
```

?

20

20



Characters

- Since **chars** are just small integers, **char** variables and constants are identical to **int** in arithmetic expressions. Some programming idioms that take advantage of this:

46	2E	056	&#46;	.
47	2F	057	&#47;	/
48	30	060	&#48;	0
49	31	061	&#49;	1
50	32	062	&#50;	2
51	33	063	&#51;	3
52	34	064	&#52;	4
53	35	065	&#53;	5
54	36	066	&#54;	6
55	37	067	&#55;	7
56	38	070	8	8
57	39	071	&#57;	9
58	3A	072	&#58;	:
59	3B	073	&#59;	;
60	3C	074	&#60;	<

```
if(c >= '0'  && c <= '9') /*index 48~57,is a digit */
                          (located from '0' to '9')

if(c >='a'  && c <= 'z')  /* low case letter */     islower

if(c >='A'  && c <= 'Z')  /* upper case letter */  isupper

if( (c >='A' && c <= 'Z') || (c >='a' && c <= 'z'))
                                isalpha   isalnum?

if(c >='0'  && c <= '9'){  // c<= 48 c>=57 isdigit(c)
  printf("c is a digit\n");
  printf("numerical value is %d\n",  ?  )
}
```

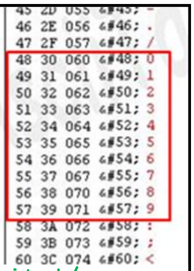
22

same in Java

22

Characters

- Since **chars** are just small integers, **char** variables and constants are identical to **int** in arithmetic expressions. Some programming idioms that take advantage of this:



45	2D	055	&#45;	-
46	2E	056	&#46;	.
47	2F	057	&#47;	/
48	30	060	&#48;	0
49	31	061	&#49;	1
50	32	062	&#50;	2
51	33	063	&#51;	3
52	34	064	&#52;	4
53	35	065	&#53;	5
54	36	066	&#54;	6
55	37	067	&#55;	7
56	38	070	&#56;	8
57	39	071	9	9
58	3A	072	&#58;	:
59	3B	073	&#59;	;
60	3C	074	&#60;	<

```
if(c >= '0' && c <= '9') /*index 48~57,is a digit */
                          (located from '0' to '9')

if(c >='a' && c <= 'z')  /* low case letter */    islower

if(c >='A' && c <= 'Z')  /* upper case letter */  isupper

if( (c >='A' && c <= 'Z') || (c >='a' && c <= 'z'))
                                   isalpha    isalnum?

if(c >='0' && c <= '9'){  // c<= 48 c>=57 isdigit(c)
  printf("c is a digit\n");
  printf("numerical value is %d\n", c-'0')
}
```

c-48 works but avoid

same in Java

Outline

- Types and sizes
  - Types
  - **Constant values (literals)**
    - **char** treated as small int
    - **int** different bases
    - float
- Array and "strings"
- Expressions
  - Basic operators
  - Type promotion and conversion
  - Other operators
  - Precedence of operators

RECALL

Integer Constants

0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1

Stored always binary 2^2 2^1 2^0

- Integer constants can be expressed in three different ways:
  1. **Decimal** [base 10]
     - `int x = 31`

     same in Java
  2. **Octal** [base 8]
     - Start with zero **0**
     - `int x = 037` (31 in decimal)

     same in Java
  3. **Hexadecimal** [base 16]
     - Start with **0x** or **0X**
     - `int x = 0x1F` (31 in decimal)

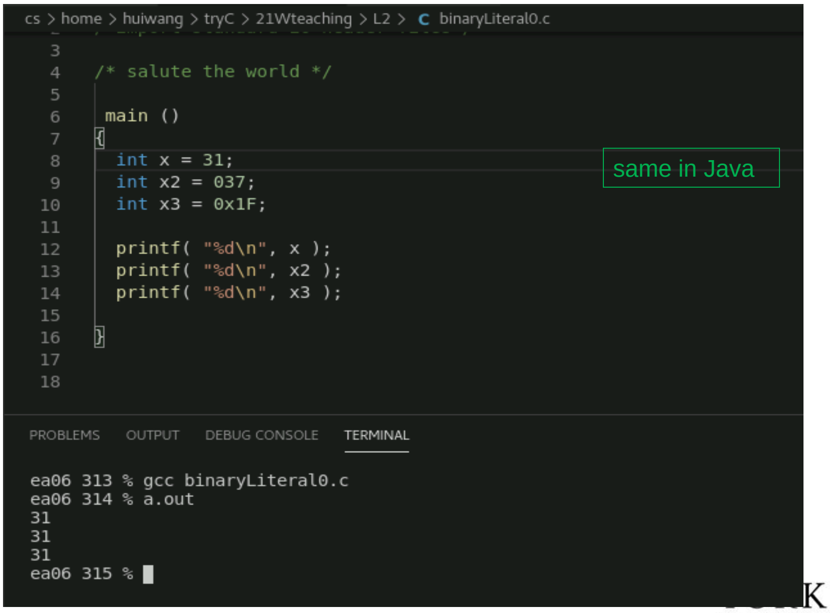
     same in Java

Ways for people to write numbers. Nothing to do with how the numbers are stored –- always binary.

Java also has the 4th way: binary
`int x = 0b00011111`

27

27


cs > home > huiwang > tryC > 21Wteaching > L2 > C binaryLiteral0.c
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL

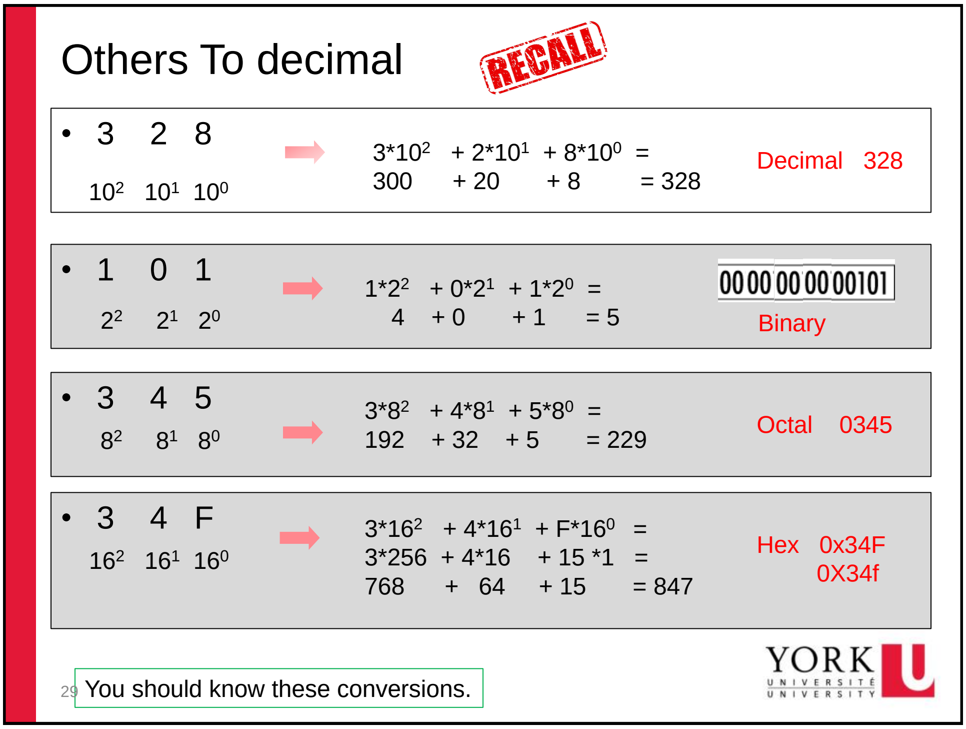
```
/* salute the world */

 main ()
{
  int x = 31;
  int x2 = 037;
  int x3 = 0x1F;

  printf( "%d\n", x );
  printf( "%d\n", x2 );
  printf( "%d\n", x3 );

}
```

```
ea06 313 % gcc binaryLiteral0.c
ea06 314 % a.out
31
31
31
ea06 315 %
```

same in Java

28

28



Others To decimal

RECALL

- 3 2 8

  10^2 10^1 10^0 → $3*10^2 + 2*10^1 + 8*10^0 =$

  300 + 20 + 8 = 328

  Decimal 328

- 1 0 1

  2^2 2^1 2^0 → $1*2^2 + 0*2^1 + 1*2^0 =$

  4 + 0 + 1 = 5

  0000000000101

  Binary

- 3 4 5

  8^2 8^1 8^0 → $3*8^2 + 4*8^1 + 5*8^0 =$

  192 + 32 + 5 = 229

  Octal 0345

- 3 4 F

  16^2 16^1 16^0 → $3*16^2 + 4*16^1 + F*16^0 =$

  3*256 + 4*16 + 15 *1 =

  768 + 64 + 15 = 847

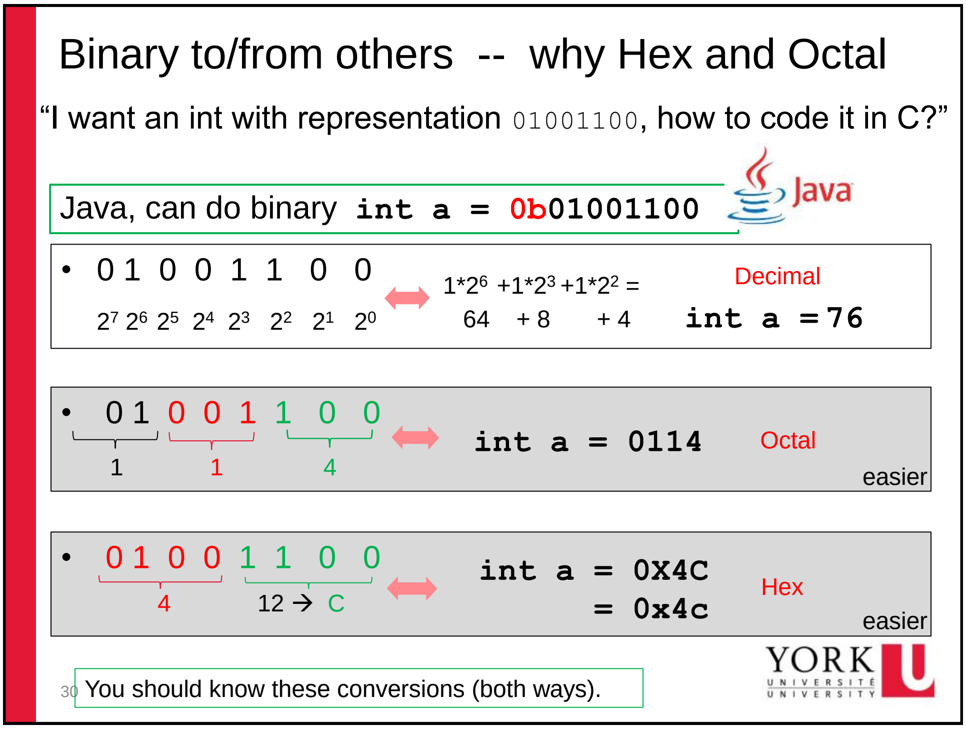
  Hex 0x34F

  0X34f

29 You should know these conversions.

YORK UNIVERSITÉ UNIVERSITY

29



Binary to/from others -- why Hex and Octal

“I want an int with representation `01001100`, how to code it in C?”

Java, can do binary `int a = 0b01001100`

Java

- 0 1 0 0 1 1 0 0

  2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0 ⇔ $1*2^6 + 1*2^3 + 1*2^2 =$

  64 + 8 + 4

  Decimal

  `int a = 76`

- $\underbrace{0\ 1}_{1}\ \underbrace{0\ 0\ 1}_{1}\ \underbrace{1\ 0\ 0}_{4}$ ⇔ `int a = 0114` Octal

  easier

- $\underbrace{0\ 1\ 0\ 0}_{4}\ \underbrace{1\ 1\ 0\ 0}_{12 \rightarrow C}$ ⇔ `int a = 0X4C`

  `= 0x4c`

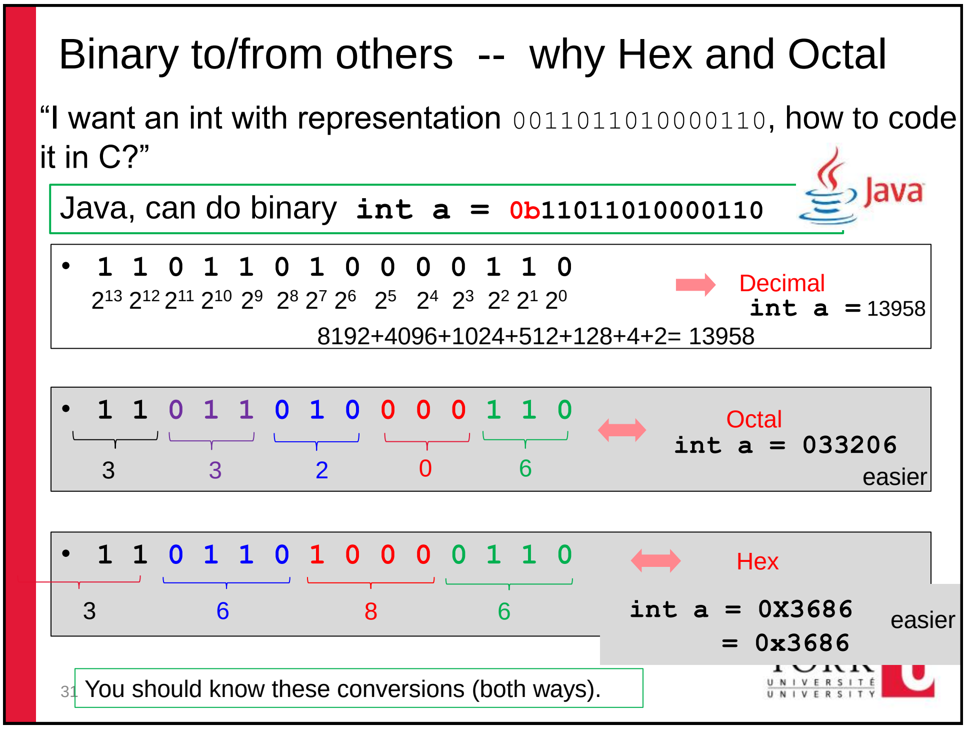
  Hex

  easier

30 You should know these conversions (both ways).

YORK UNIVERSITÉ UNIVERSITY

30



Binary to/from others -- why Hex and Octal

"I want an int with representation `0011011010000110`, how to code it in C?"

Java, can do binary `int a = 0b11011010000110`

- `1 1 0 1 1 0 1 0 0 0 0 1 1 0`
  2^{13} 2^{12} 2^{11} 2^{10} 2^{9} 2^{8} 2^{7} 2^{6} 2^{5} 2^{4} 2^{3} 2^{2} 2^{1} 2^{0}
  8192+4096+1024+512+128+4+2= 13958

Decimal
`int a =` 13958

- `1 1 0 1 1 0 1 0 0 0 0 1 1 0`
  $\underbrace{11}_{3}\ \underbrace{011}_{3}\ \underbrace{010}_{2}\ \underbrace{000}_{0}\ \underbrace{110}_{6}$

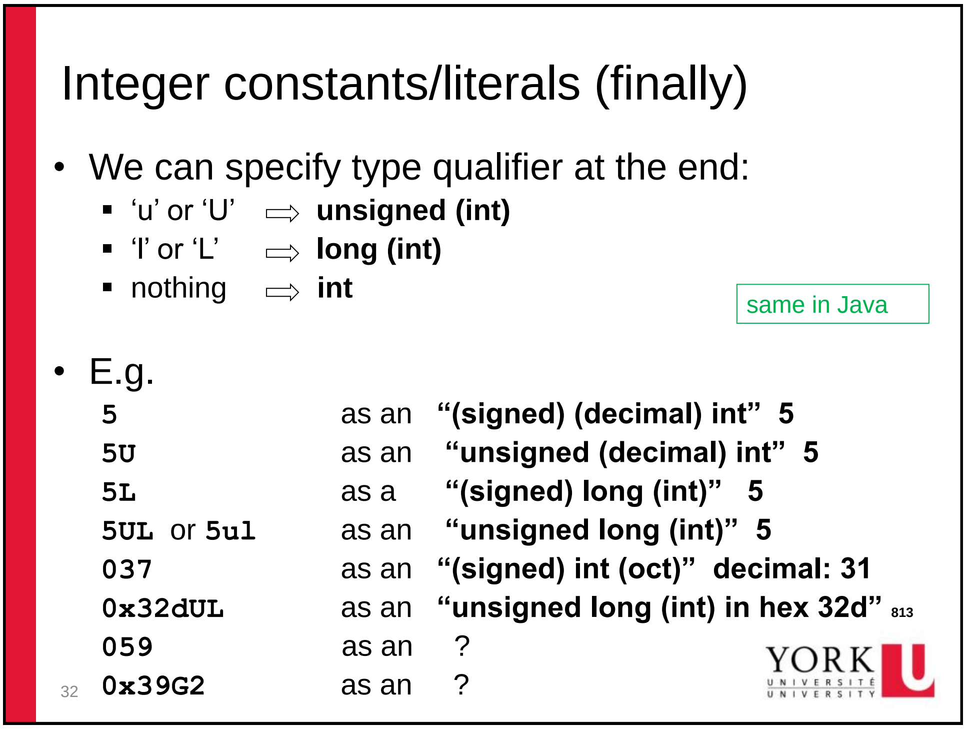
Octal
`int a = 033206`
easier

- `1 1 0 1 1 0 1 0 0 0 0 1 1 0`
  $\underbrace{11}_{3}\ \underbrace{0110}_{6}\ \underbrace{1000}_{8}\ \underbrace{0110}_{6}$

Hex
`int a = 0X3686`
`= 0x3686`
easier

31 You should know these conversions (both ways).

31



Integer constants/literals (finally)

- We can specify type qualifier at the end:
  - 'u' or 'U' ⇨ **unsigned (int)**
  - 'l' or 'L' ⇨ **long (int)**
  - nothing ⇨ **int**

same in Java

- E.g.

`5`	as an	**"(signed) (decimal) int" 5**
`5U`	as an	**"unsigned (decimal) int" 5**
`5L`	as a	**"(signed) long (int)" 5**
`5UL` or `5ul`	as an	**"unsigned long (int)" 5**
`037`	as an	**"(signed) int (oct)" decimal: 31**
`0x32dUL`	as an	**"unsigned long (int) in hex 32d"** 813
`059`	as an	?
`0x39G2`	as an	?

32

32

Floating Point Constants

- All floating point constants contain a decimal point('.') and/or an exponent ('e' of "E")
  - E.g. `1.532   3e5   4.112e-10`
  - `5.3e12` == 5.3 × 10^{12}
  - `printf("%E %e", 0.00137, 123.025);`

```
1.370000E-03 1.230250e+02
```

0.00137	1.37 x 10^{-3}
15237	1.5237 x 10^{4}
59000005	5.9000005 x 10^{7}
123.025	1.23025 x 10^{2}
0.00005025	5.025 x 10^{-5}

- Floating point constants are of type 'double'
  - Nothing – means **"double"** e.g., `double x = 1.532`

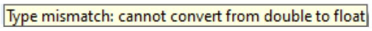
    same in Java

  - 'f' or 'F' - means **"float"** e.g. `float x = 1.532f`

    `float x = 1.532` OK

    same in Java

    Not OK in Java



Type mismatch: cannot convert from double to float

  - 'l' or 'L' - means **"long double"** e.g. `long double x=1.5L`

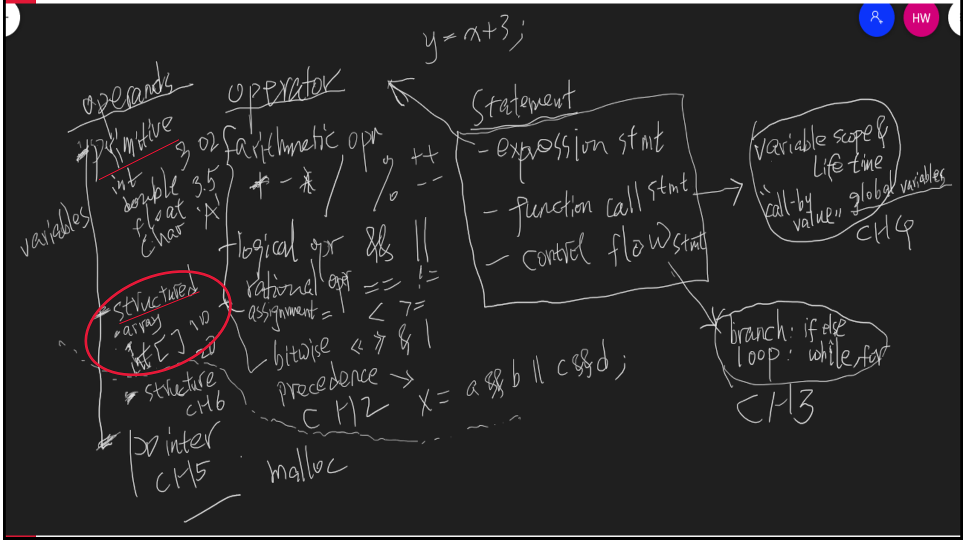
    same in Java

Outline

- Types and sizes
  - Types
  - Constant values (literals)
    - char
    - int
    - float
- **Array and "strings" (Ch1.6,1.9)**
- Expressions
  - Basic operators
  - Type promotion and conversion
  - Other operators
  - Precedence of operators

Roadmap -- How the topics are related

RECALL



36

Declaring Arrays

LET'S RECAP...

int[] k = new int[3];

int[] k = {1, 2, 3};

- Declare and initialize (how to do in Java?)

Java

```
int k[5];  /* each element get some garble value*/
           -5  122 45623 85 58

int k[5] = {1,5,3,2,25};    1 5 3 2 25

int k[5] = {1,5};           1 5 0 0 0

int k[] =  {1,5,3,2,25};    1 5 3 2 25

int k[3] = {1,5,3,2,25}  ✗

int k[];  ✗

sizeof k?         // assuming 4 bytes int

sizeof(k)/sizeof(k[0]) = 20/4 = 5
```

YORK UNIVERSITÉ UNIVERSITY

40

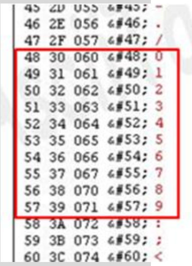
40

An example involving array and chars

What does this program do?

```
/*counting digits*/
#include <stdio.h>
#define N 10

int main () {
   int c, i;
   int digit[N];

   for (i=0; i< N; i++)
     digit[i]=0;

   while ((c = getchar()) != EOF){
     if ( c == '0') digit[0]++;
     elseif ( c == '1') digit[1]++;
     elseif ( c == '2') digit[2]++;
     …
     elseif ( c == '9') digit[9]++;
   }

   for (i=0; i< N; i++) // has to use loop
      printf ("%d ", digit[i]);

   return 0;
}
```

needed



45	2D	055	&#45;	-
46	2E	056	&#46;	.
47	2F	057	&#47;	/
48	30	060	&#48;	0
49	31	061	&#49;	1
50	32	062	&#50;	2
51	33	063	&#51;	3
52	34	064	&#52;	4
53	35	065	&#53;	5
54	36	066	&#54;	6
55	37	067	&#55;	7
56	38	070	&#56;	8
57	39	071	&#57;	9
58	3A	072	&#58;	:
59	3B	073	&#59;	;
60	3C	074	&#60;	<

Simpler code? Lab2

Stopped here

43

Accessing Arrays

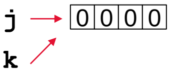
- In C, you can only assign to array members
  - This means you cannot copy/assign to a whole array:

```
int  i, k[4], j[4];
for (i=0; i<4; i++)
   j[i]= 0;     /* another way? int j[4]={0}  */

k = j; /* invalid *//* perfectly valid in Java */
```

```
for (i=0; i<4; i++)
   k[i] = j[i];
```

or

```
i=0;
while(i<4)
{
   k[i] = j[i];
   i++;
}
```



```
for (i=0; i<10; i++)
   k[i] = j[i];
```

Compiles, may or may not crash
no boundary checking

`k=j`   `k==j`   explain later

44

Summary and plan

- [Primitive] Types and sizes
  - Types: char, short, int, long, unsigned short, unsigned int, float, double …..
  - Constant values (literals)
    - ○ char 'A'
    - ○ int 37 037 0x37
    - ○ float 3.3 3.4f 3.2e5
- [Structured] Array and "**strings**"

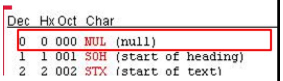
Last 2 lectures

- Expressions
  - Basic operators
  - Type promotion and conversion
  - Other operators
  - Precedence of operators

Plan for this week

45

Strings ⇔ Character Arrays !



```
Dec Hx Oct Char
0   0  000 NUL (null)
1   1  001 SOH (start of heading)
2   2  002 STX (start of text)
```

- There is no separate "string" type in C
- Strings are just arrays of char that end with `'\0'`
  - `char s[]= "Hello";`

⇩

'H'	'e'	'l'	'l'	'o'	'\0'

'\0' added for you

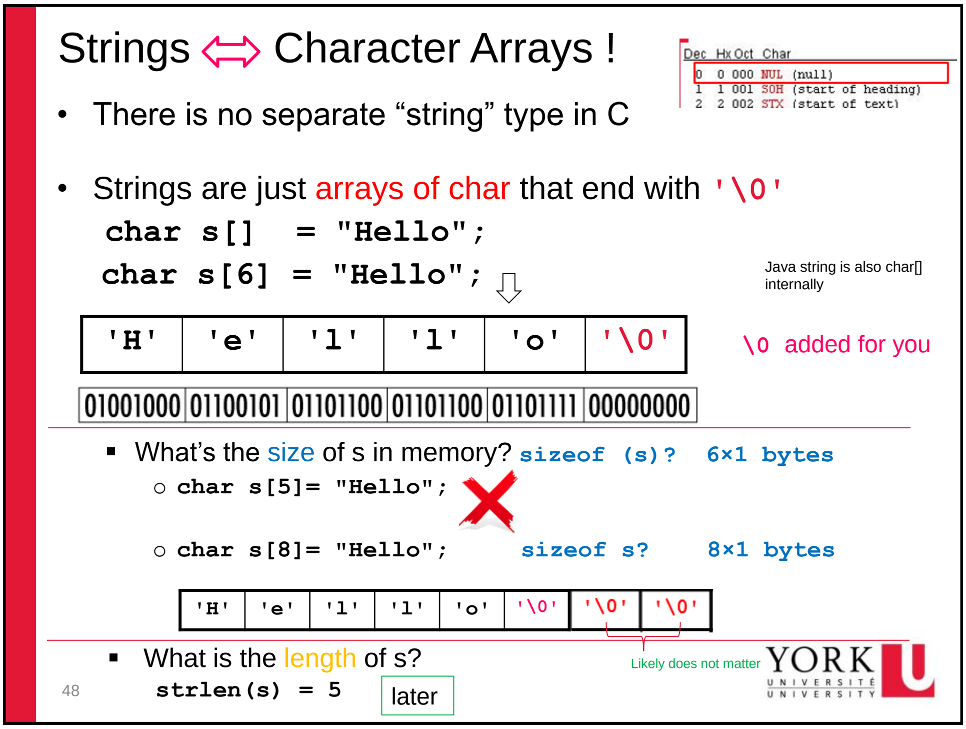
‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’
72	101	108	108	111	0
01001000	01100101	01101100	01101100	01101111	00000000

is equivalent to

```
char s[]= {'H', 'e', 'l', 'l', 'o', '\0'}
```

46

No '\0' valid?



Strings ⇔ Character Arrays !

Dec	Hx	Oct	Char
0	0	000	NUL (null)
1	1	001	SOH (start of heading)
2	2	002	STX (start of text)

- There is no separate “string” type in C
- Strings are just arrays of char that end with `'\0'`

```
char s[]  = "Hello";
char s[6] = "Hello";
```

Java string is also char[] internally

'H'	'e'	'l'	'l'	'o'	'\0'
01001000	01100101	01101100	01101100	01101111	00000000

\0 added for you

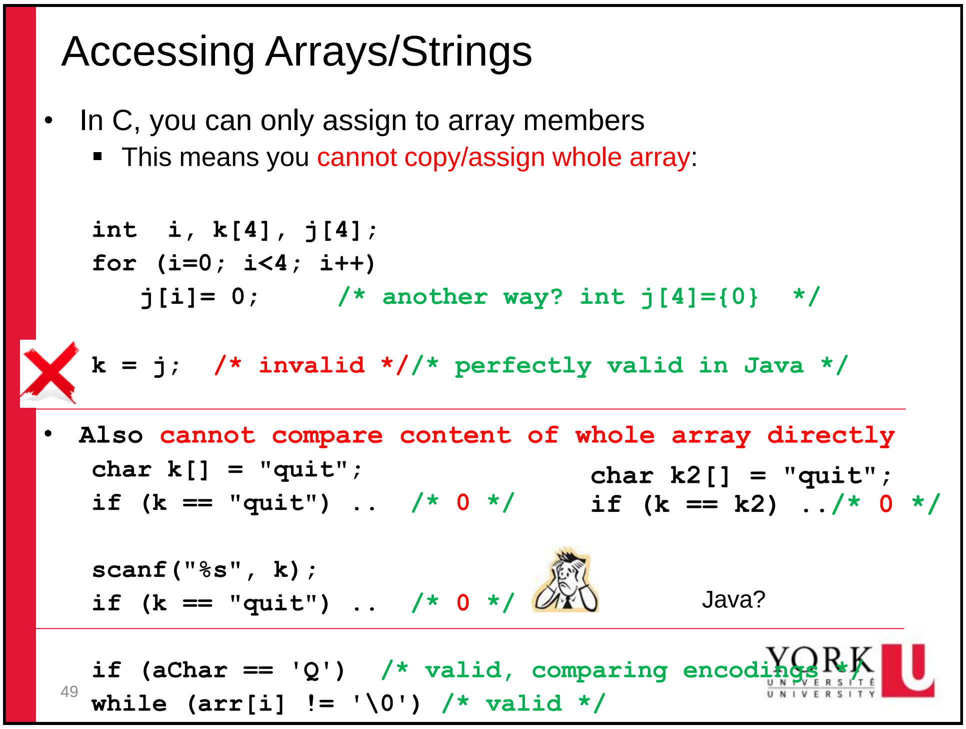
- What’s the size of s in memory? `sizeof (s)?` `6×1 bytes`
  - `char s[5]= "Hello";`
  - `char s[8]= "Hello";` `sizeof s?` `8×1 bytes`

'H'	'e'	'l'	'l'	'o'	'\0'	'\0'	'\0'

Likely does not matter

- What is the length of s?
  `strlen(s) = 5` later

48



Accessing Arrays/Strings

- In C, you can only assign to array members
  - This means you cannot copy/assign whole array:

```
int  i, k[4], j[4];
for (i=0; i<4; i++)
   j[i]= 0;     /* another way? int j[4]={0}  */

k = j;  /* invalid *//* perfectly valid in Java */
```

- Also cannot compare content of whole array directly

```
char k[] = "quit";
if (k == "quit") ..  /* 0 */
```

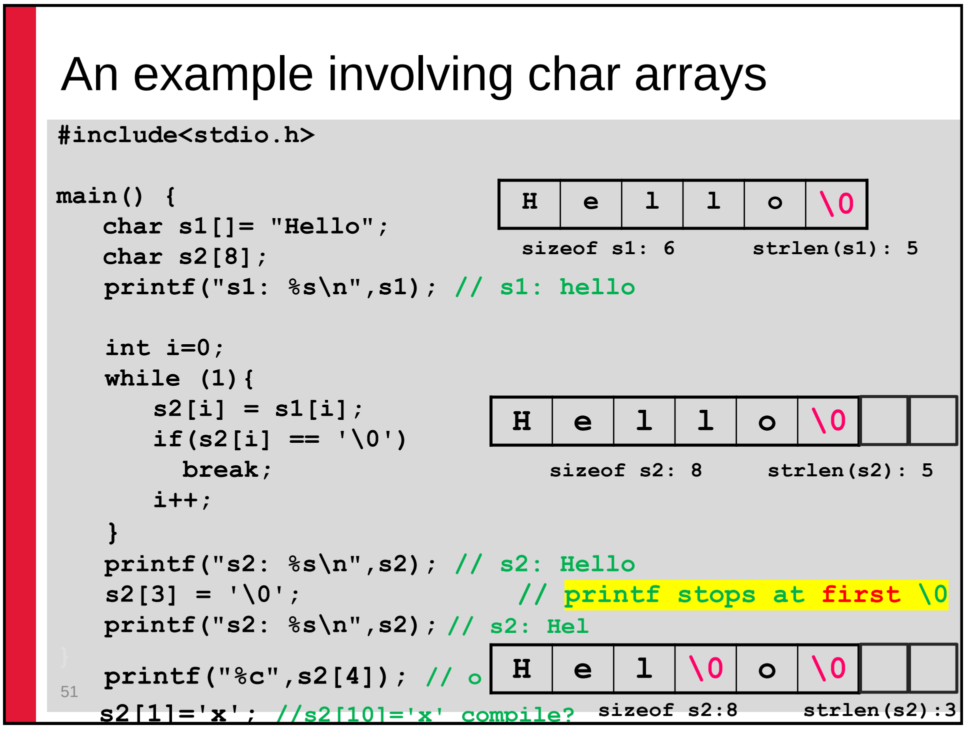
```
char k2[] = "quit";
if (k == k2) ../* 0 */
```

```
scanf("%s", k);
if (k == "quit") ..  /* 0 */
```

Java?

```
if (aChar == 'Q')  /* valid, comparing encodings */
while (arr[i] != '\0') /* valid */
```

49



An example involving char arrays

```
#include<stdio.h>

main() {
   char s1[]= "Hello";
   char s2[8];
   printf("s1: %s\n",s1); // s1: hello

   int i=0;
   while (1){
      s2[i] = s1[i];
      if(s2[i] == '\0')
        break;
      i++;
   }
   printf("s2: %s\n",s2); // s2: Hello
   s2[3] = '\0';              // printf stops at first \0
   printf("s2: %s\n",s2); // s2: Hel
}
   printf("%c",s2[4]); // o
   s2[1]='x'; //s2[10]='x' compile?
```

H	e	l	l	o	\0

sizeof s1: 6 strlen(s1): 5

H	e	l	l	o	\0		

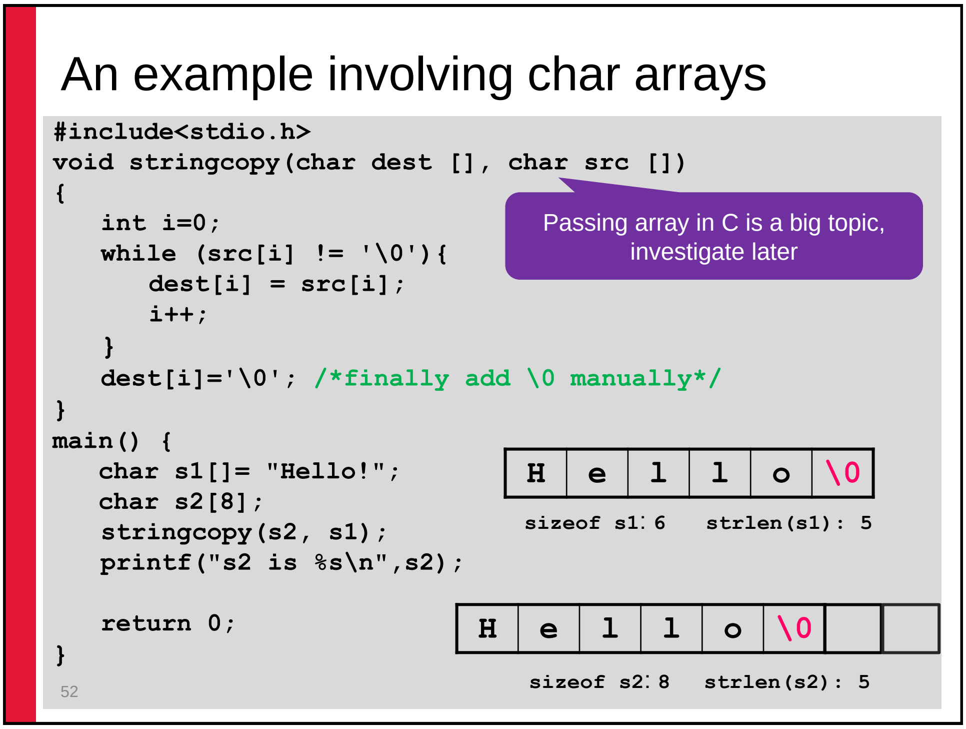
sizeof s2: 8 strlen(s2): 5

H	e	l	\0	o	\0		

sizeof s2:8 strlen(s2):3

51

51



An example involving char arrays

```
#include<stdio.h>
void stringcopy(char dest [], char src [])
{
   int i=0;
   while (src[i] != '\0'){
      dest[i] = src[i];
      i++;
   }
   dest[i]='\0'; /*finally add \0 manually*/
}
main() {
   char s1[]= "Hello!";
   char s2[8];
   stringcopy(s2, s1);
   printf("s2 is %s\n",s2);

   return 0;
}
```

Passing array in C is a big topic, investigate later

H	e	l	l	o	\0

sizeof s1: 6 strlen(s1): 5

H	e	l	l	o	\0		

sizeof s2: 8 strlen(s2): 5

52

52

An example involving char arrays

```
#include<stdio.h>
void stringcopy2(char dest [], char src [])
{
   int i=0;
   while (1){                          /* Another version  */
      dest[i] = src[i];
      if (src[i] == '\0')          // if (dest[i] == '\0')
         break;

      i++;
   }
}

main() {
   char s1[]= "Hello!";
   char s2[8];
   stringcopy2(s2, s1);
   printf("s2 is %s\n",s2);

   return 0;
}
```

H	e	l	l	o	\0

sizeof s1: 6   strlen(s1): 5

H	e	l	l	o	\0		

sizeof s2: 8   strlen(s2): 5

53

53

Read string using scanf

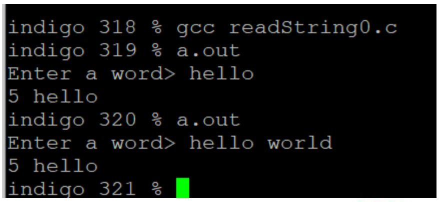
Which is correct?

```
char my_strg[100];
scanf ("%s", &my_strg);
scanf ("%s", my_strg);

printf("%s", my_strg);
```

Output with space in input?

"EECS2031 AC fall"

"Hello World"



```
indigo 318 % gcc readString0.c
indigo 319 % a.out
Enter a word> hello
5 hello
indigo 320 % a.out
Enter a word> hello world
5 hello
indigo 321 %
```

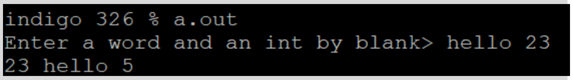
55

55

An example involving reading char arrays

```
#include<stdio.h>
int length (char []);

main() {
   char my_strg[100];
   int a;

   printf("Enter a word and an int separated by blank>");
   scanf("%s %d", my_strg, &a);
   printf("%d %s %d", a, my_strg, length(my_strg));
}

int length(char arr[]){
    int i = 0;
    while (arr[i] != '\0')
      i++;
    return i;
}
```

No & needed!
Another big topic.
Investigate later



```
indigo 326 % a.out
Enter a word and an int by blank> hello 23
23 hello 5
```

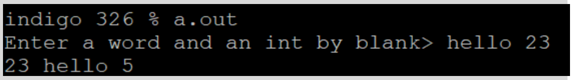
56

An example involving reading char arrays

```
#include<stdio.h>
int length (char []);

main() {
   char my_strg[100];
   int a;

   printf("Enter a word and an int separated by blank>");
   scanf("%s %d", my_strg, &a);
   printf("%d %s %d", a, my_strg, length(my_strg));
}

int length(char arr[]){
    int i = 0;
    while (arr[i] != '\0')
      i++;
    return i;
}
```

No need to give size

No need to give size



```
indigo 326 % a.out
Enter a word and an int by blank> hello 23
23 hello 5
```

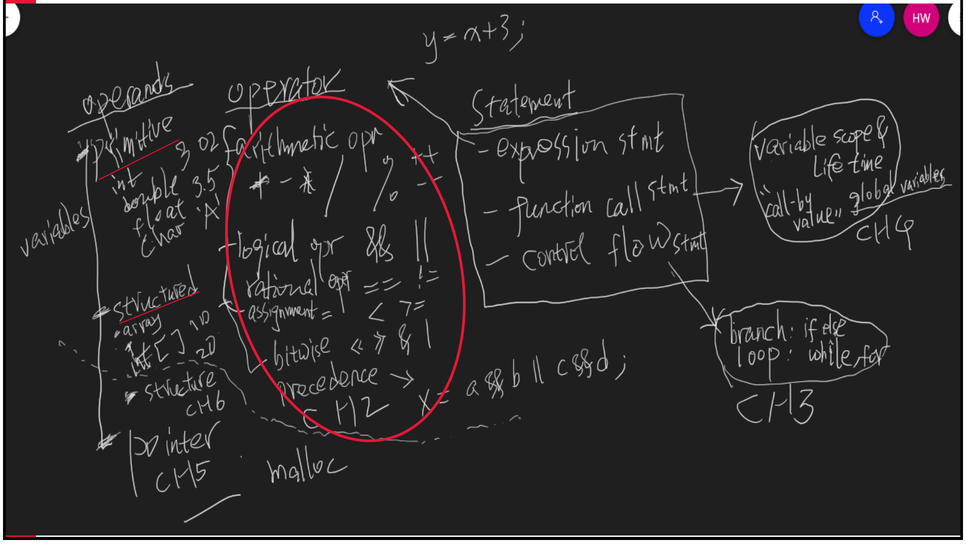
57

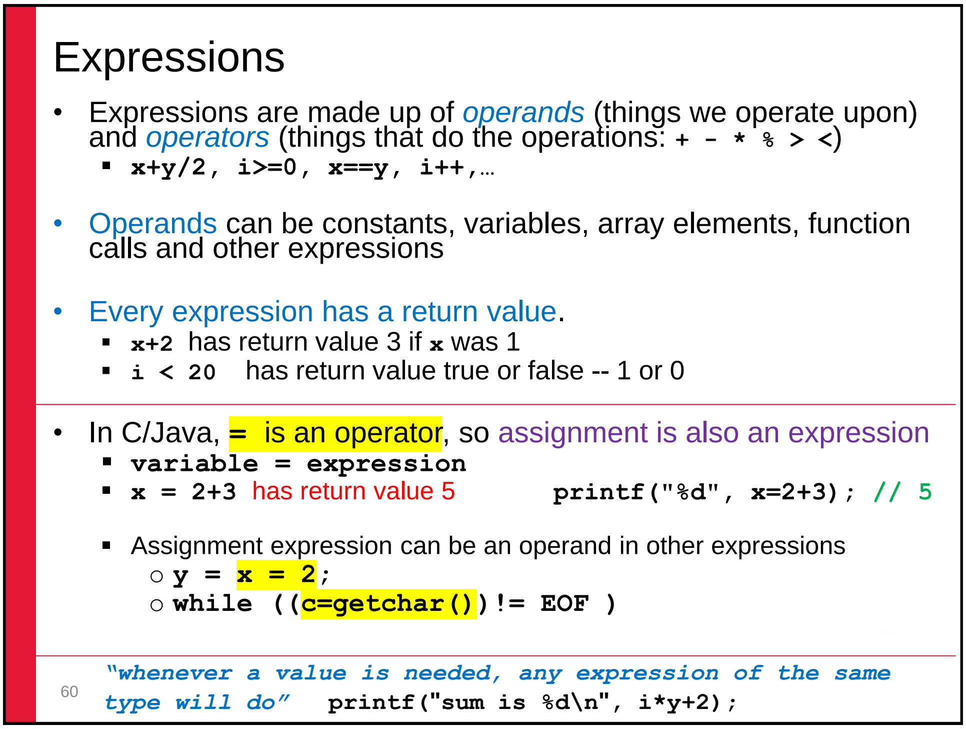
Outline

- Types and sizes
  - Types
  - Constant values (literals)
    - char
    - int
    - float
- Array and “strings” (Ch1.6,1.9)
- **Expressions**
  - **Basic operators (arithmetic, relational and logical)**
  - Type promotion and conversion
  - Other operators (bitwise, bit shifting , compound assignment, conditional)
  - Precedence of operators

Roadmap -- How the topics are related

RECALL





Expressions

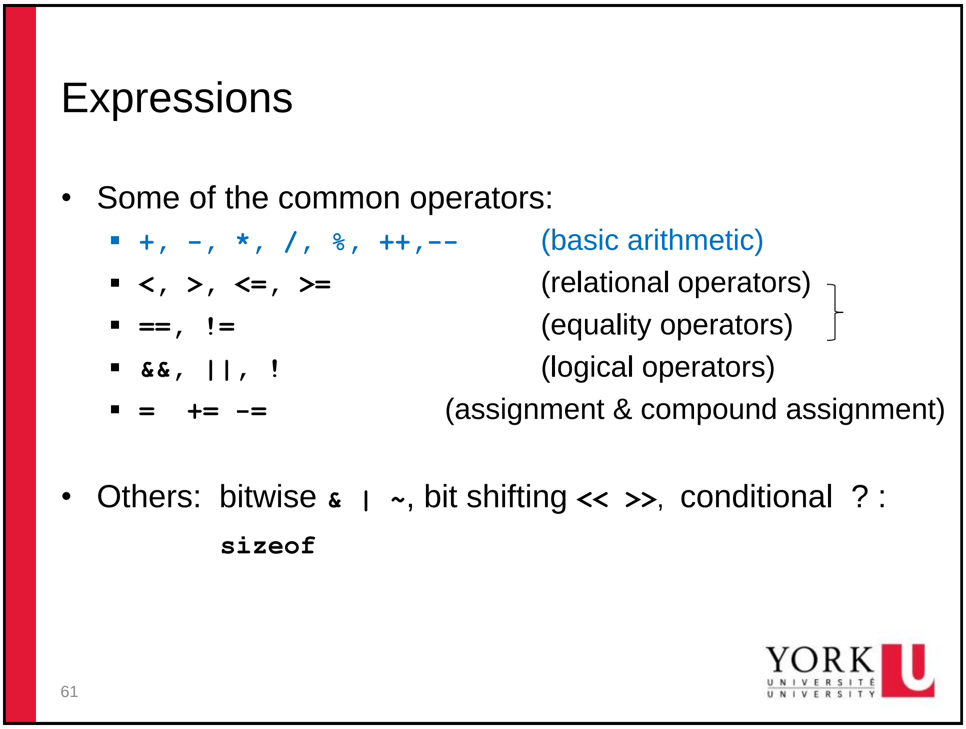
- Expressions are made up of *operands* (things we operate upon) and *operators* (things that do the operations: `+ - * % > <`)
  - `x+y/2, i>=0, x==y, i++,…`
- Operands can be constants, variables, array elements, function calls and other expressions
- Every expression has a return value.
  - `x+2` has return value 3 if `x` was 1
  - `i < 20` has return value true or false -- 1 or 0

- In C/Java, `=` is an operator, so assignment is also an expression
  - `variable = expression`
  - `x = 2+3` has return value 5 `printf("%d", x=2+3); // 5`
  - Assignment expression can be an operand in other expressions
    - ○ `y = x = 2;`
    - ○ `while ((c=getchar())!= EOF )`

"whenever a value is needed, any expression of the same type will do" `printf("sum is %d\n", i*y+2);`

60

60



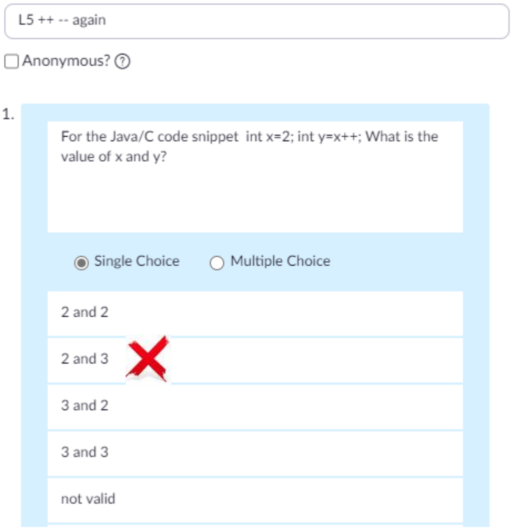
Expressions

- Some of the common operators:
  - `+, -, *, /, %, ++,--` (basic arithmetic)
  - `<, >, <=, >=` (relational operators)
  - `==, !=` (equality operators)
  - `&&, ||, !` (logical operators)
  - `= += -=` (assignment & compound assignment)

- Others: bitwise `& | ~`, bit shifting `<< >>`, conditional `? :`
  `sizeof`

61

61



L5 ++ -- again

☐ Anonymous? ⓘ

1. For the Java/C code snippet int x=2; int y=x++; What is the value of x and y?

● Single Choice ○ Multiple Choice

- 2 and 2
- 2 and 3 ✗
- 3 and 2
- 3 and 3
- not valid

62

62

Arithmetic (unary) Increment/Decrement Operators

++ increment

-- decrement

same in Java

- May come before (prefix) or after the operand (postfix)

`++x`	increment x, result of expression is new value (pre-increment)
`x++`	increment x, result of expression is old value (post-increment)
`--x`	decrement x, result of expression is new value (pre-decrement)
`x--`	decrement x, result of expression is old value (post-decrement)

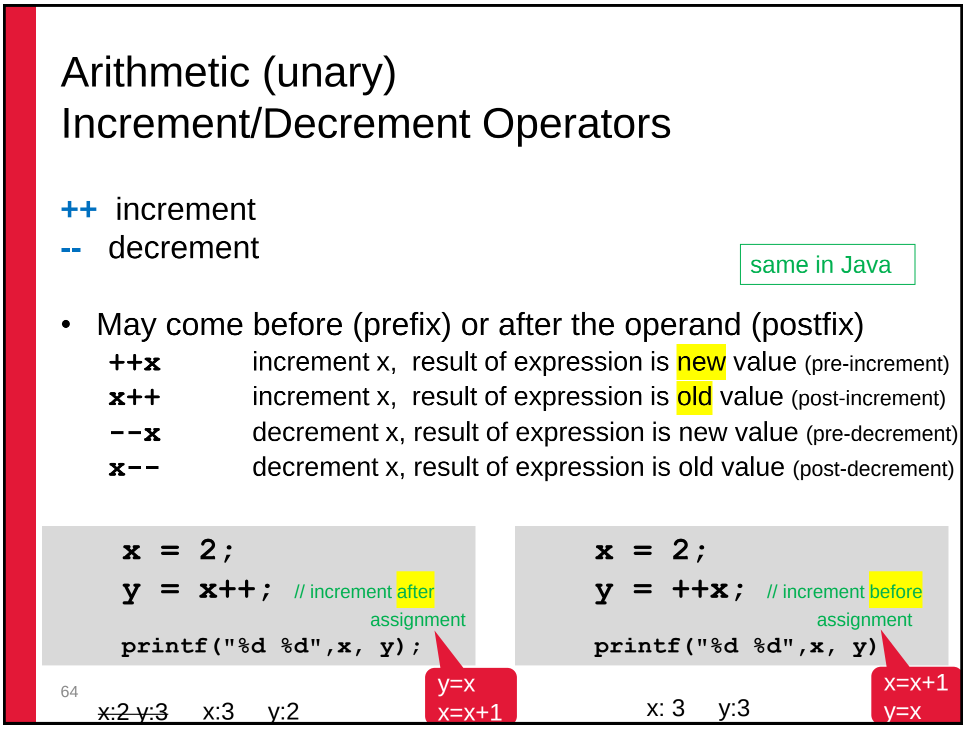
```
while (x < 10){
  ......
  x++;  // increment later,
           before next statement
  ......
}
```

```
while (x < 10){
  ......
  ++x;  // increment immediately
  ......
}
```

Same effects

63

63



Arithmetic (unary) Increment/Decrement Operators

++ increment
-- decrement

same in Java

- May come before (prefix) or after the operand (postfix)

`++x`	increment x, result of expression is new value (pre-increment)
`x++`	increment x, result of expression is old value (post-increment)
`--x`	decrement x, result of expression is new value (pre-decrement)
`x--`	decrement x, result of expression is old value (post-decrement)

```
x = 2;
y = x++;  // increment after
                       assignment
printf("%d %d",x, y);
```

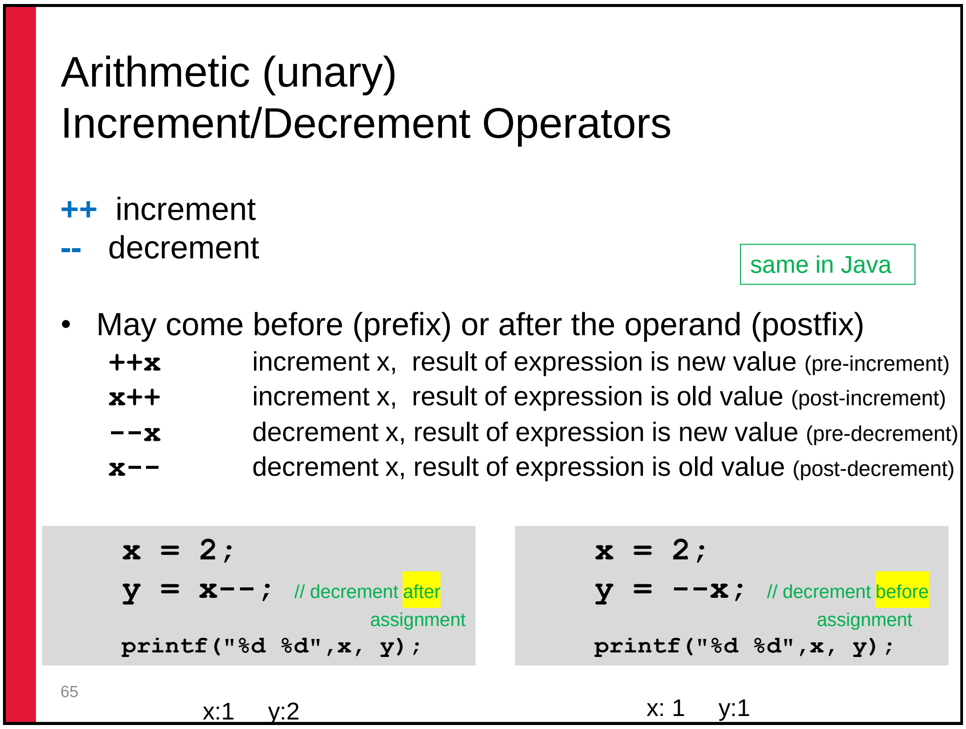
y=x
x=x+1

~~x:2 y:3~~ x:3 y:2

```
x = 2;
y = ++x;  // increment before
                       assignment
printf("%d %d",x, y)
```

x=x+1
y=x

x: 3 y:3



Arithmetic (unary) Increment/Decrement Operators

++ increment
-- decrement

same in Java

- May come before (prefix) or after the operand (postfix)

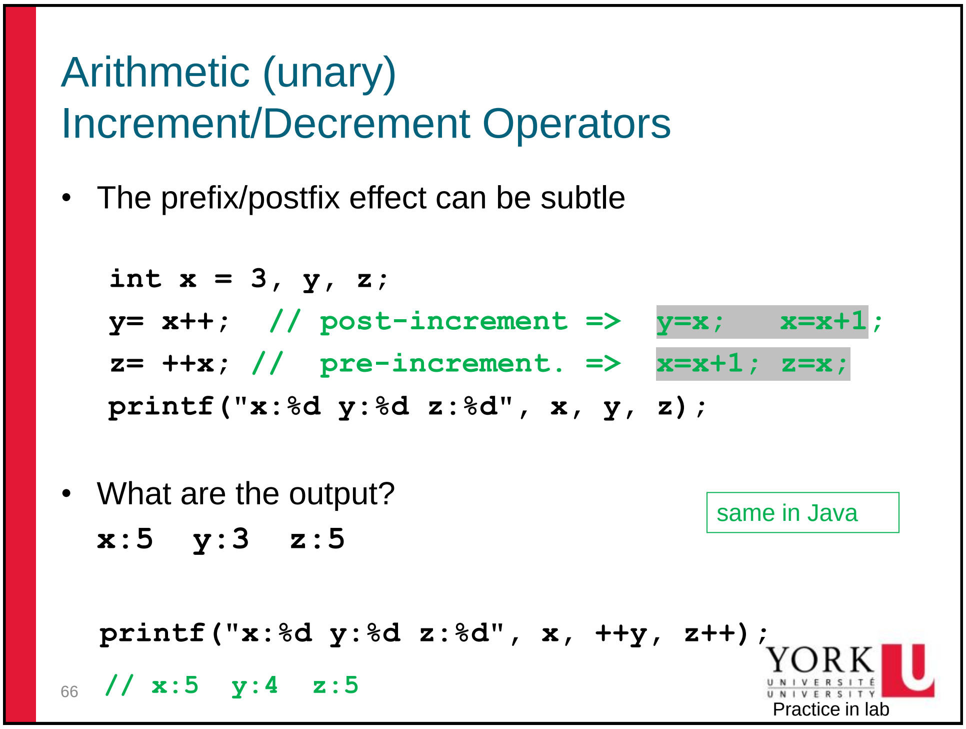
`++x`	increment x, result of expression is new value (pre-increment)
`x++`	increment x, result of expression is old value (post-increment)
`--x`	decrement x, result of expression is new value (pre-decrement)
`x--`	decrement x, result of expression is old value (post-decrement)

```
x = 2;
y = x--;  // decrement after
                       assignment
printf("%d %d",x, y);
```

x:1 y:2

```
x = 2;
y = --x;  // decrement before
                       assignment
printf("%d %d",x, y);
```

x: 1 y:1



Arithmetic (unary) Increment/Decrement Operators

- The prefix/postfix effect can be subtle

```
int x = 3, y, z;
y= x++;  // post-increment =>  y=x;   x=x+1;
z= ++x; //  pre-increment. =>  x=x+1; z=x;
printf("x:%d y:%d z:%d", x, y, z);
```

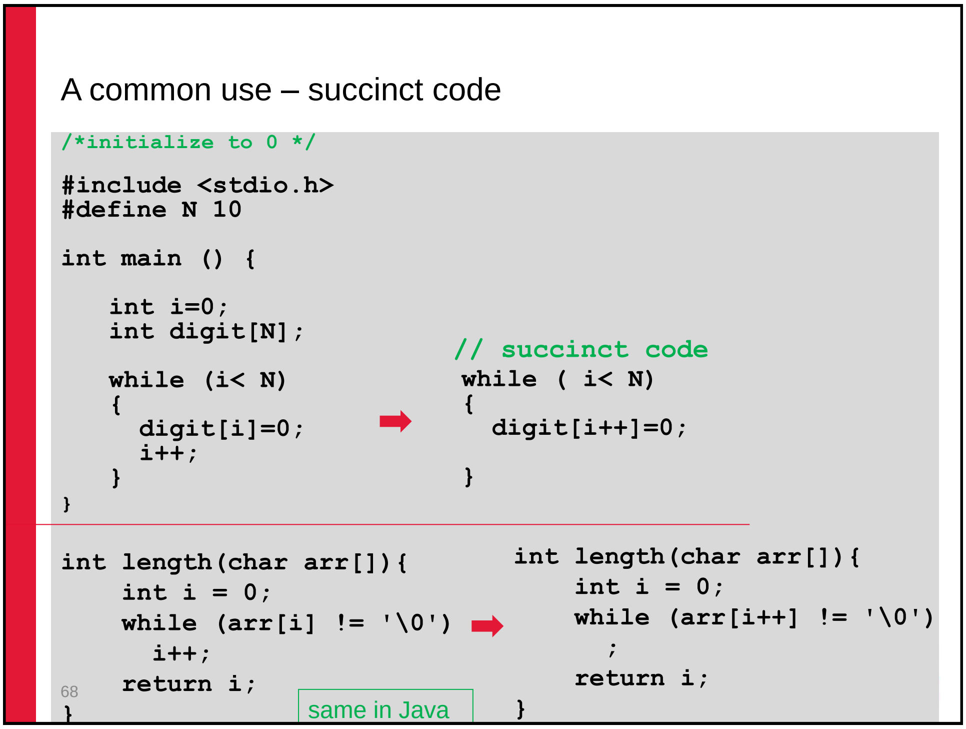
- What are the output?

```
x:5  y:3  z:5
```

same in Java

```
printf("x:%d y:%d z:%d", x, ++y, z++);
// x:5  y:4  z:5
```

Practice in lab

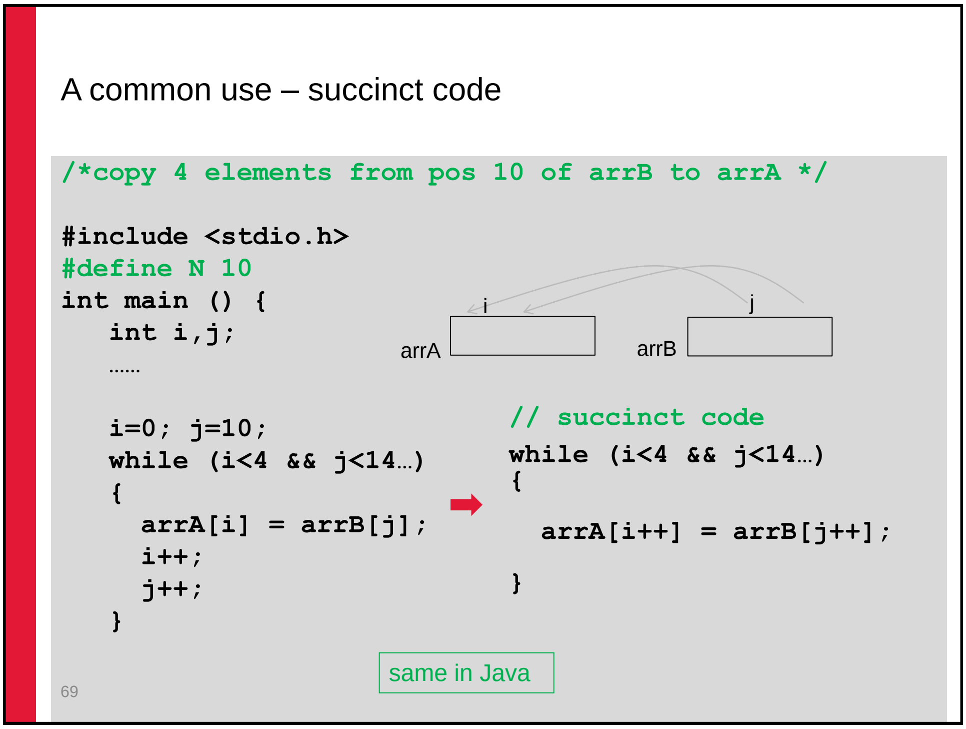


A common use – succinct code

```
/*initialize to 0 */

#include <stdio.h>
#define N 10

int main () {

   int i=0;
   int digit[N];
                                // succinct code
   while (i< N)                 while ( i< N)
   {                            {
     digit[i]=0;       ➡          digit[i++]=0;
     i++;
   }                            }
}
```

```
int length(char arr[]){          int length(char arr[]){
     int i = 0;                       int i = 0;
     while (arr[i] != '\0')  ➡        while (arr[i++] != '\0')
       i++;                              ;
     return i;                        return i;
}                                }
```

same in Java



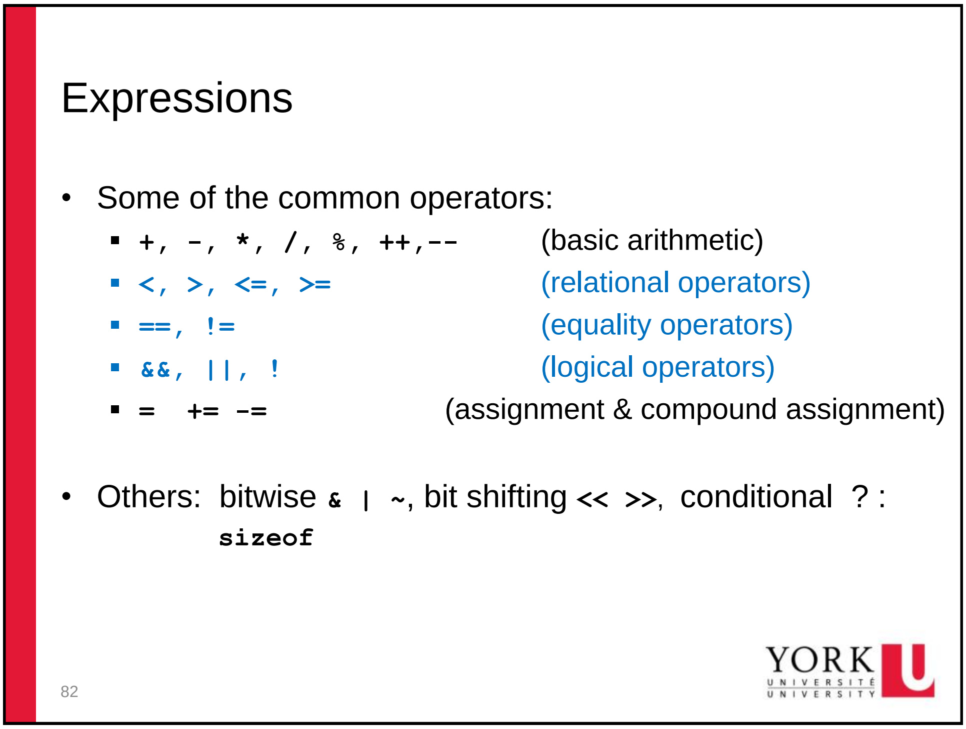
A common use – succinct code

```
/*copy 4 elements from pos 10 of arrB to arrA */

#include <stdio.h>
#define N 10
int main () {
   int i,j;
   ……

   i=0; j=10;
   while (i<4  && j<14…)
   {
     arrA[i] = arrB[j];
     i++;
     j++;
   }
```

i
j
arrA
arrB

```
// succinct code
while (i<4  && j<14…)
{

  arrA[i++] = arrB[j++];

}
```

same in Java



Expressions

- Some of the common operators:
  - `+, -, *, /, %, ++,--` (basic arithmetic)
  - `<, >, <=, >=` (relational operators)
  - `==, !=` (equality operators)
  - `&&, ||, !` (logical operators)
  - `=  += -=` (assignment & compound assignment)
- Others: bitwise `& | ~`, bit shifting `<< >>`, conditional ? :
  `sizeof`

Relational and logical Operators

`< > <= >= == !=` (relational and equality operators)

`&& || !` (logical operators)

- Value of a relational or logical expression is `Boolean`

return 0 when evaluated *false*
return 1 when evaluated *true*

0 is treated as *false*
non-zero is treated as *true*

Relational and logical Operators

`< > <= >= == !=` (relational and equality operators)

`&& || !` (logical operators)

- Value of a relational or logical expression is `Boolean`

return 0 when evaluated *false*
return 1 when evaluated *true*

0 is treated as *false*
non-zero is treated as *true*

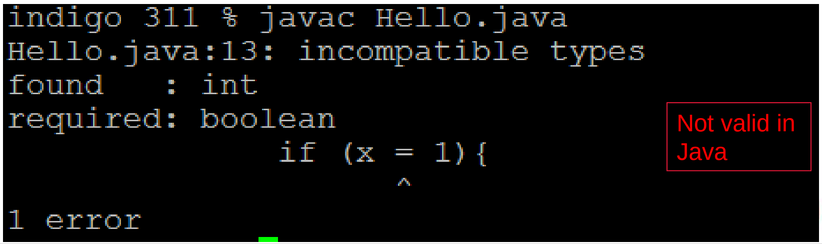
```
int x = 3;
x > 4                printf("%d", x<4);
x == 3
x != 4
if (x == 5)     not true

while (1) true loop       while (-10) true loop
if (5) true
if (x = 5)        ? java?
```

Relational and logical Operators

- Not as safe as Java -- probably why C99 and Java introduced bool, Boolean

```
int x = 2;
……
if (x = 1)
  print 1
else
  print 2
```

```
int x = 2;
……
while(x = 3)
    ……
    ……
```



```
indigo 311 % javac Hello.java
Hello.java:13: incompatible types
found   : int
required: boolean
              if (x = 1){
                    ^
1 error
```

Not valid in Java

85

85

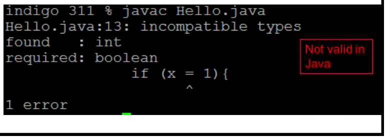
Relational and logical Operators

- Not as safe as Java -- probably why C99 and Java introduced bool, Boolean

```
int num = 2;

if (num = 10)
  num = num + 1;
else
  num = num + 2;
printf("%d\n",num);
```

```
int num = 2;

if (num = 0)
 num = num + 1;
else
 num = num + 2;
printf("%d\n", num);
```

11   2

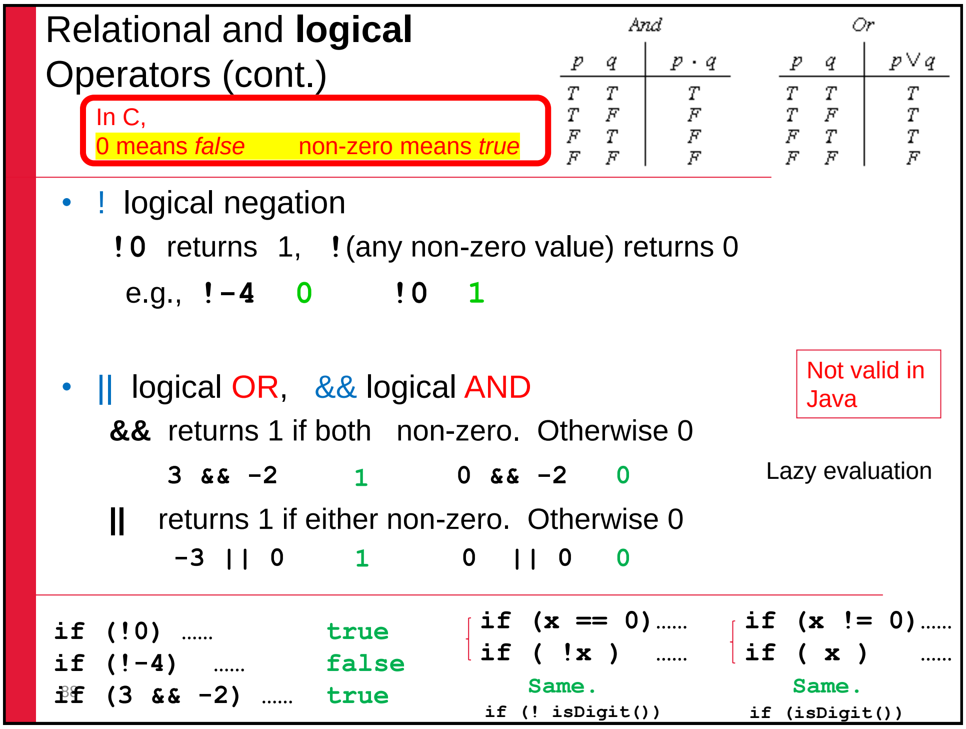


```
indigo 311 % javac Hello.java
Hello.java:13: incompatible types
found   : int
required: boolean
              if (x = 1){
                    ^
1 error
```

Not valid in Java

86

86



Relational and **logical** Operators (cont.)

In C,
0 means *false*   non-zero means *true*

And

p	q	$p \cdot q$
T	T	T
T	F	F
F	T	F
F	F	F

Or

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

- ! logical negation

  !0 returns 1, **!**(any non-zero value) returns 0

  e.g., **!-4** 0 **!0** 1

- || logical OR, && logical AND

  && returns 1 if both non-zero. Otherwise 0

  `3 && -2` 1 `0 && -2` 0

  || returns 1 if either non-zero. Otherwise 0

  `-3 || 0` 1 `0 || 0` 0

Not valid in Java

Lazy evaluation

```
if (!0) ......           true
if (!-4) ......          false
if (3 && -2) ......      true
```

```
if (x == 0)......
if ( !x )   ......
   Same.
if (! isDigit())
```

```
if (x != 0)......
if ( x )   ......
   Same.
if (isDigit())
```

88