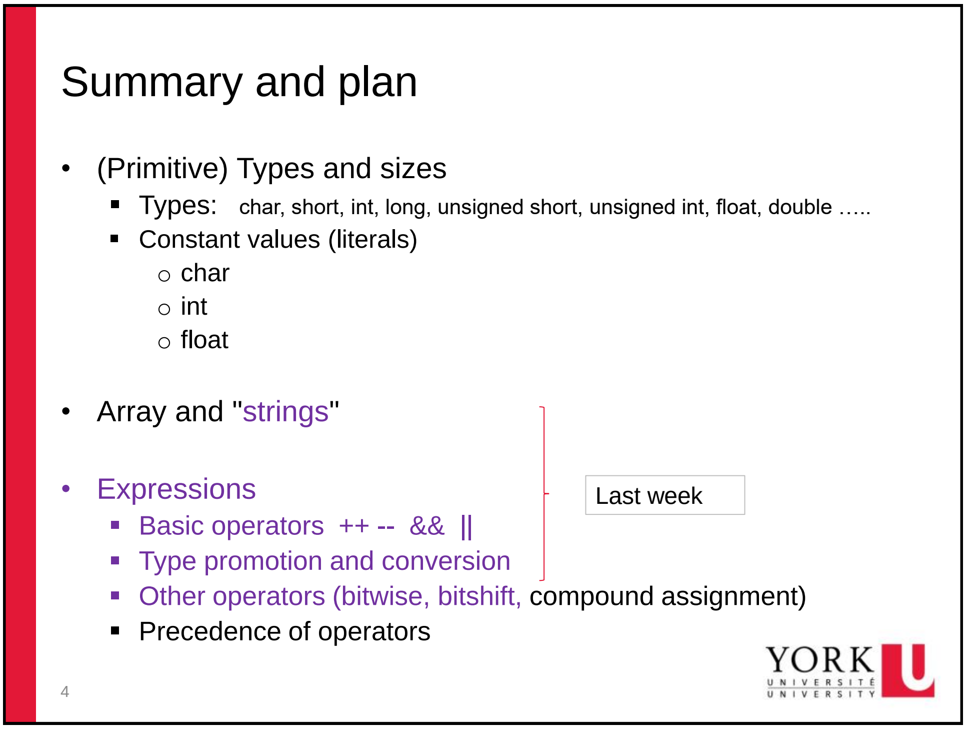
EECS2031 AC Software Tools

F 2021

Sep 29, 2021 Lecture 7

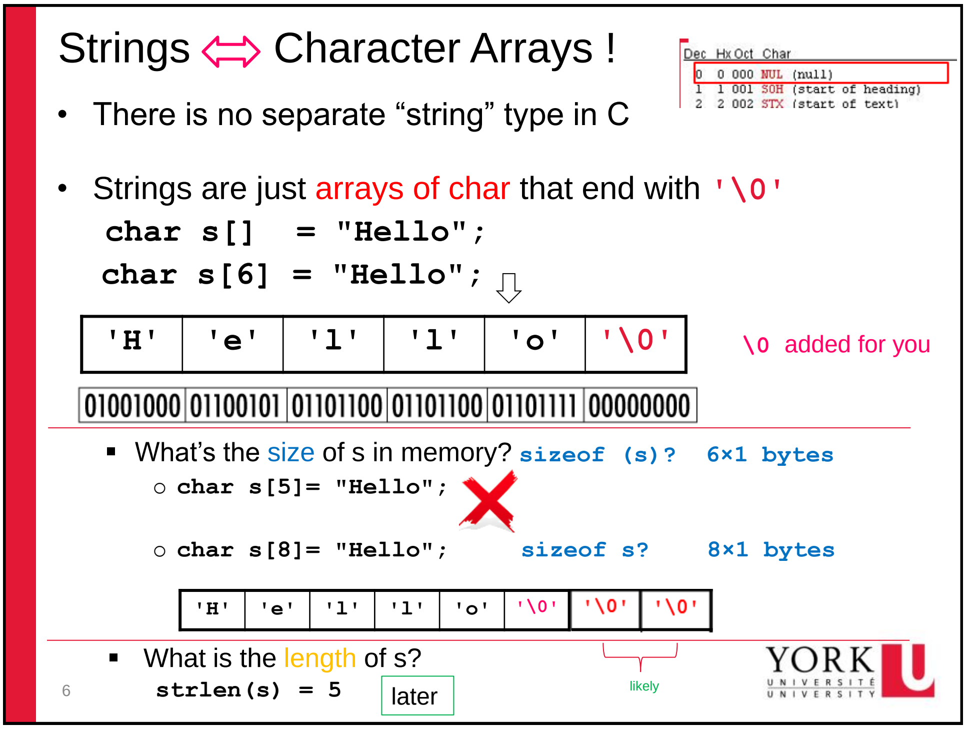
1



Summary and plan

- (Primitive) Types and sizes
  - Types: char, short, int, long, unsigned short, unsigned int, float, double …..
  - Constant values (literals)
    - char
    - int
    - float
- Array and "strings"
- Expressions
  - Basic operators ++ -- && ||
  - Type promotion and conversion
  - Other operators (bitwise, bitshift, compound assignment)
  - Precedence of operators

Last week



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"; ⇩
```

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

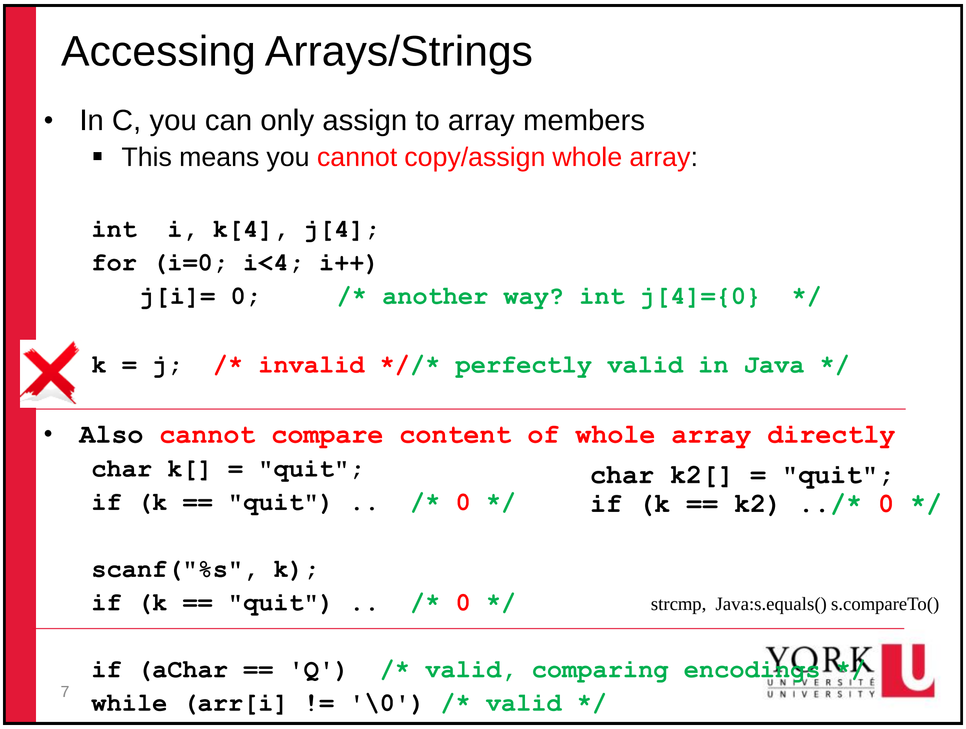
\0 added for you

- 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

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



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 */

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

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

strcmp, Java:s.equals() s.compareTo()

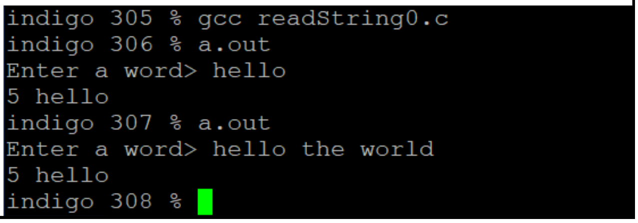
```
if (aChar == 'Q')  /* valid, comparing encodings */
while (arr[i] != '\0') /* valid */
```

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 input "Hello World"



```
indigo 305 % gcc readString0.c
indigo 306 % a.out
Enter a word> hello
5 hello
indigo 307 % a.out
Enter a word> hello the world
5 hello
indigo 308 % █
```

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 & needed!
Another big topic.
Investigate later

No need to give size

Outline

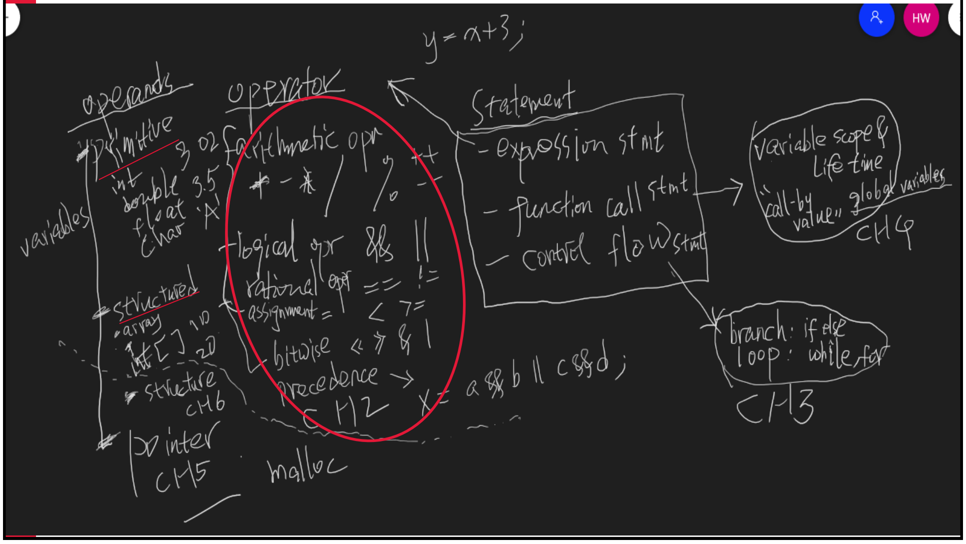
- Types and sizes
  - Types
  - Constant values (literals)
    - char
    - int
    - float

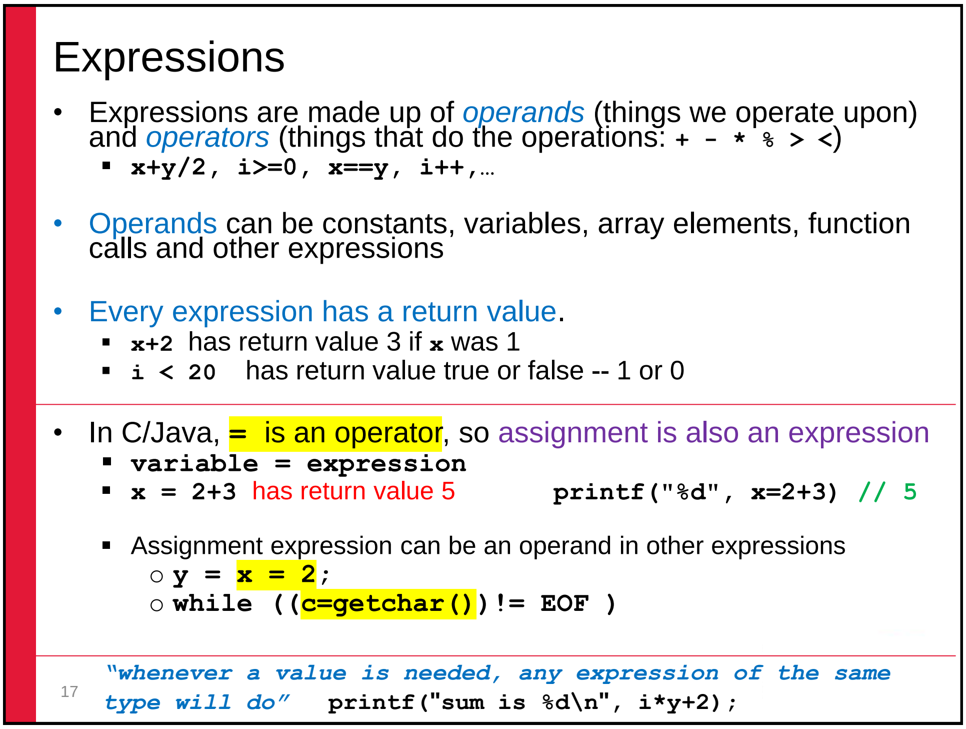
- Array and “strings” (Ch1.6,1.9)

- **Expressions**
  - **Basic operators (arithmetic, relational and logical)**
  - Type promotion and conversion
  - Other operators (bitwise, bit shifting , compound assignment)
  - Precedence of operators

Roadmap -- How the topics are related

RECALL





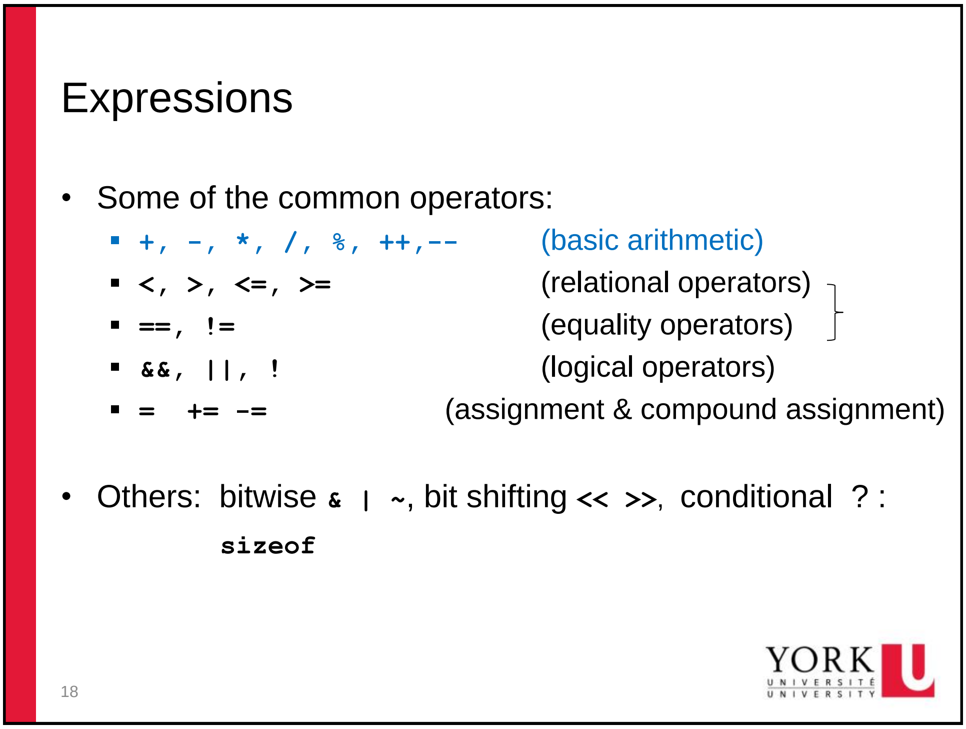
Expressions

- 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 )`

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

17



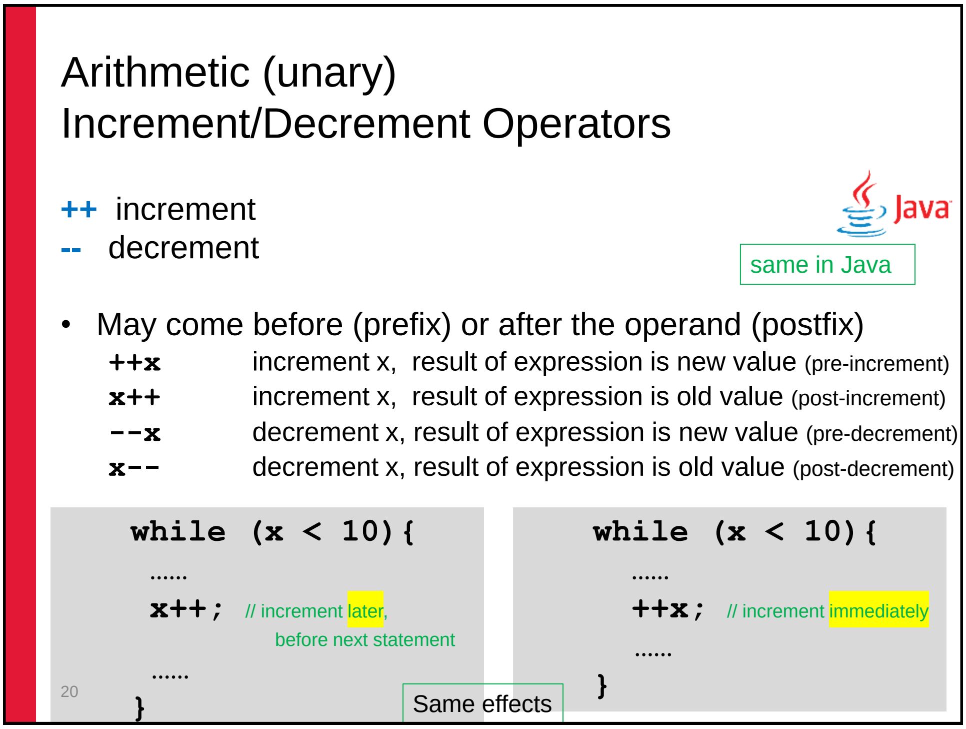
Expressions

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

18

YORK U UNIVERSITÉ UNIVERSITY

18



Arithmetic (unary) Increment/Decrement Operators

++ increment
-- decrement

same in Java

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

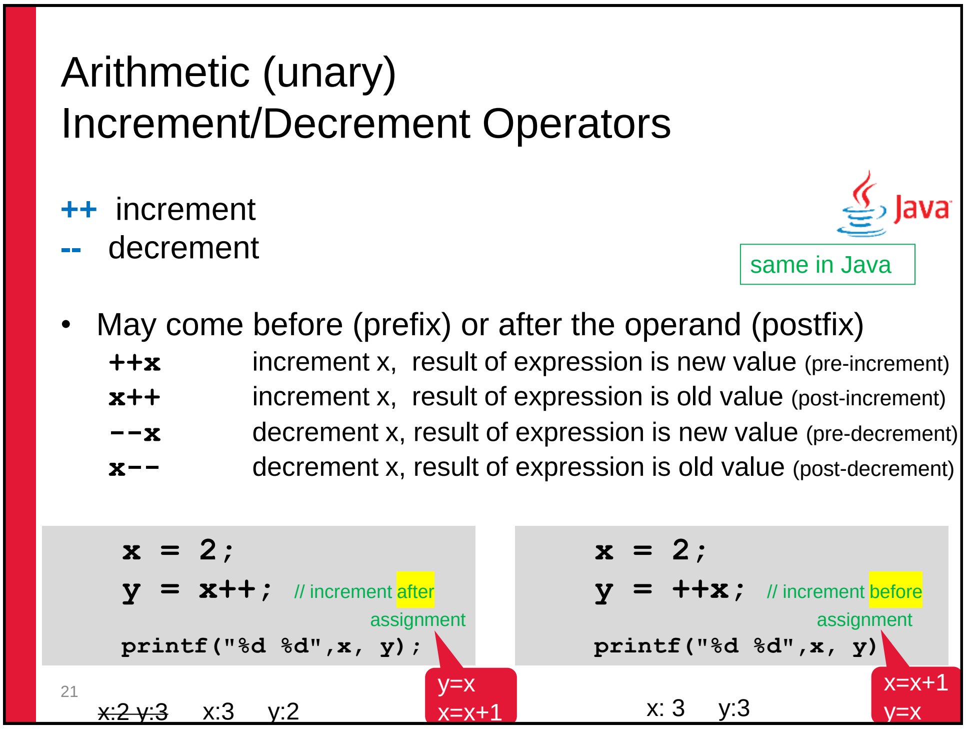
`++x`	increment x, result of expression is new value (pre-increment)
`x++`	increment x, result of expression is old value (post-increment)
`--x`	decrement x, result of expression is new value (pre-decrement)
`x--`	decrement x, result of expression is old value (post-decrement)

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

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

Same effects

20



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

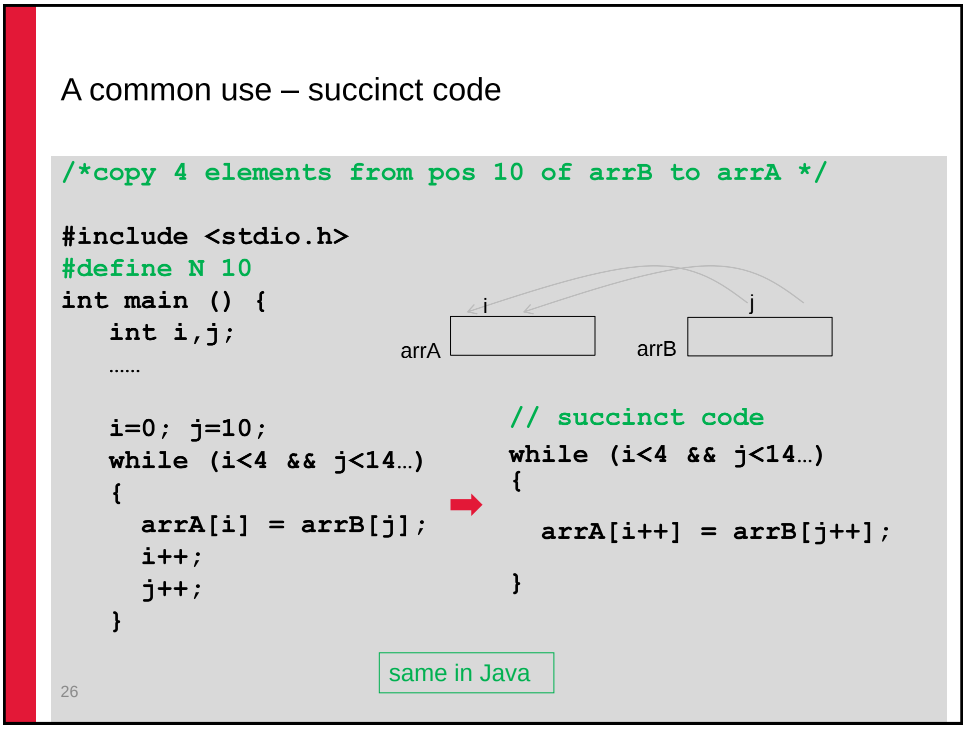
~~x:2 y:3~~ x:3 y:2

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

x=x+1
y=x

x: 3 y:3

21



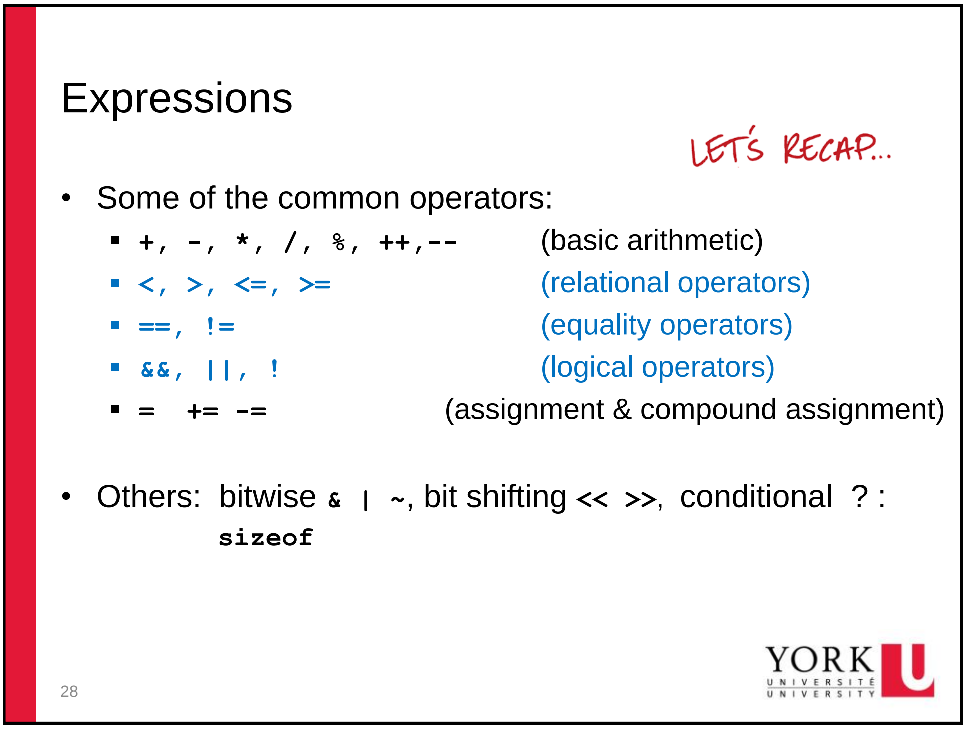
A common use – succinct code

```
/*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

LET'S RECAP...

- 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*

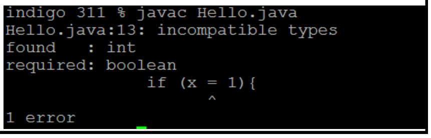
```
int x = 3;
x > 4     0        printf("%d", x<4);  1
x == 3    1
x != 4    1
if (x == 5)    not true

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

Relational and logical Operators

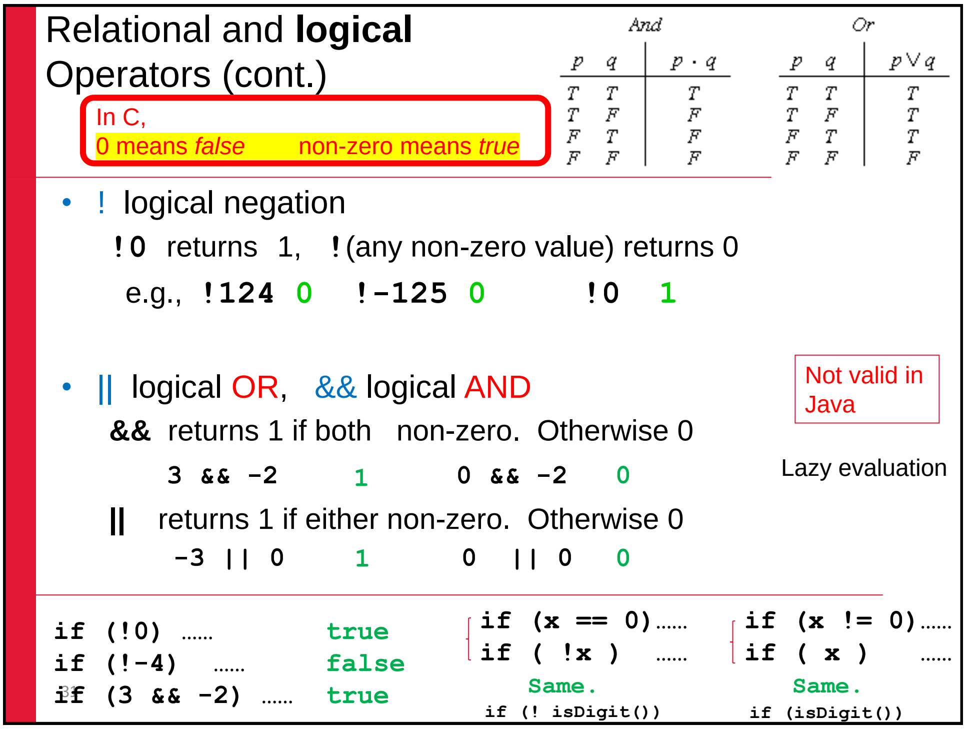
- Not as safe as Java -- probably why C99 and Java introduced bool, boolean

```
int num = 2;

if (num = -20)
  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);
```



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



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., `!124` 0 `!-125` 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

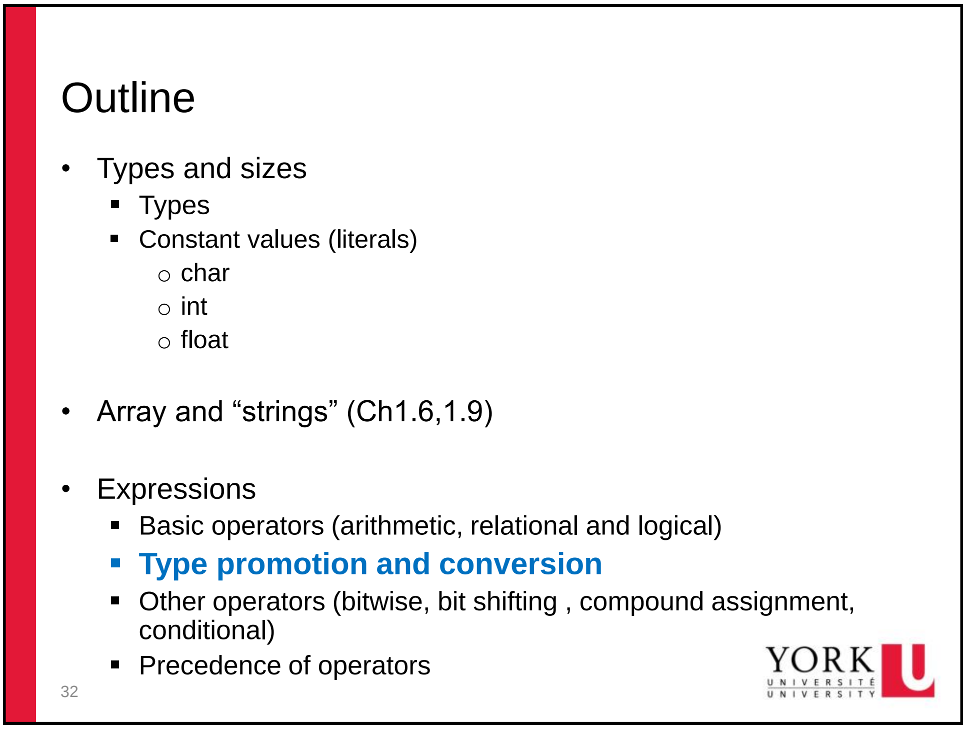
Not valid in Java

Lazy evaluation

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

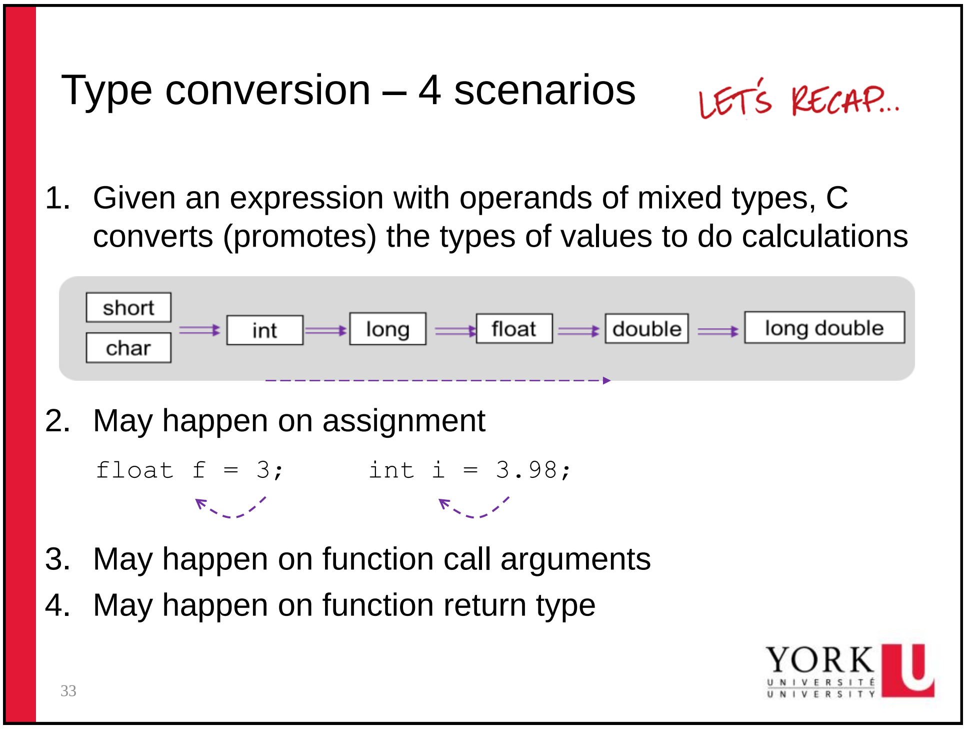
```
if (x == 0)......
if ( !x )   ......
Same.
if (! isDigit())
```

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



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



Type conversion – 4 scenarios

LET'S RECAP...

1. Given an expression with operands of mixed types, C converts (promotes) the types of values to do calculations

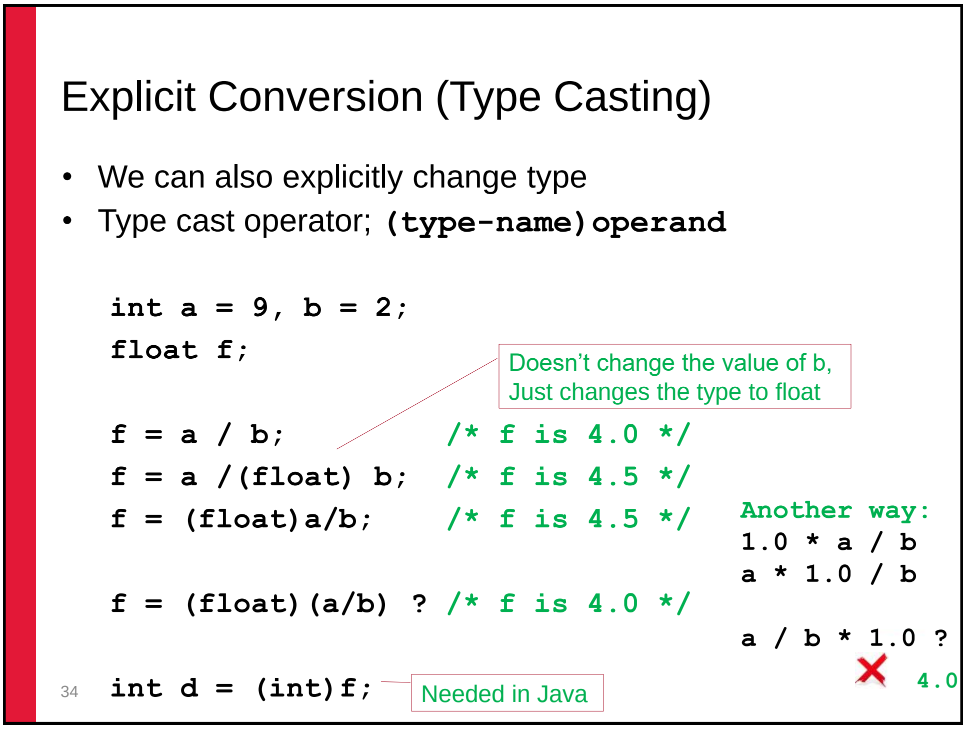
short
char
int
long
float
double
long double

2. May happen on assignment

```
float f = 3;      int i = 3.98;
```

3. May happen on function call arguments
4. May happen on function return type

YORK U UNIVERSITÉ UNIVERSITY



Explicit Conversion (Type Casting)

- We can also explicitly change type
- Type cast operator; **(type-name)operand**

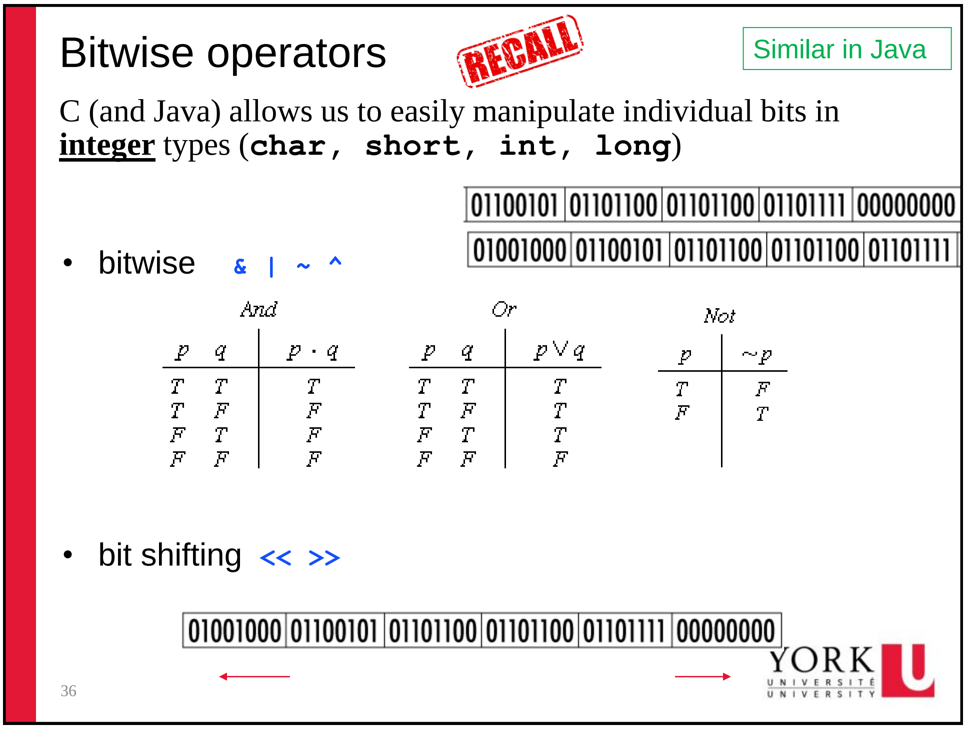
```
int a = 9, b = 2;
float f;

f = a / b;          /* f is 4.0 */
f = a /(float) b;   /* f is 4.5 */
f = (float)a/b;     /* f is 4.5 */

f = (float)(a/b) ? /* f is 4.0 */

int d = (int)f;
```

Doesn't change the value of b,
Just changes the type to float

Needed in Java

```
Another way:
1.0 * a / b
a * 1.0 / b

a / b * 1.0 ?
```

✗ 4.0



Bitwise operators

RECALL

Similar in Java

C (and Java) allows us to easily manipulate individual bits in **integer** types (**`char, short, int, long`**)

01100101 01101100 01101100 01101111 00000000

01001000 01100101 01101100 01101100 01101111

- bitwise `& | ~ ^`

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

Not

p	$\sim p$
T	F
F	T

- bit shifting `<< >>`

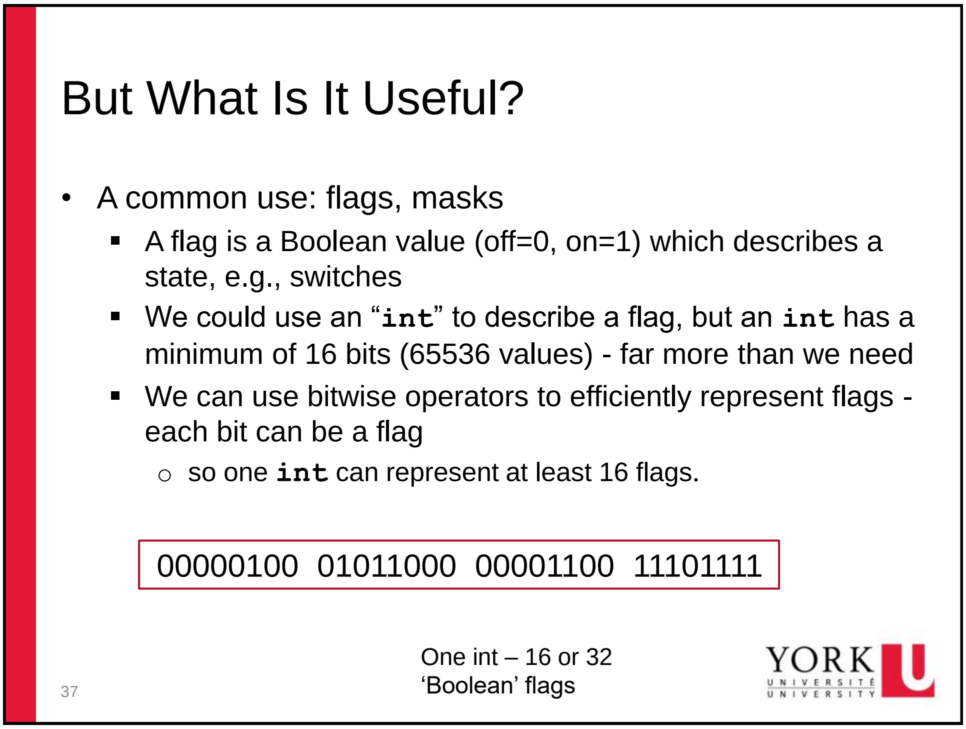
01001000 01100101 01101100 01101100 01101111 00000000

← →

36

YORK U UNIVERSITÉ UNIVERSITY

36



But What Is It Useful?

- A common use: flags, masks
  - A flag is a Boolean value (off=0, on=1) which describes a state, e.g., switches
  - We could use an "**`int`**" to describe a flag, but an **`int`** has a minimum of 16 bits (65536 values) - far more than we need
  - We can use bitwise operators to efficiently represent flags - each bit can be a flag
    - so one **`int`** can represent at least 16 flags.

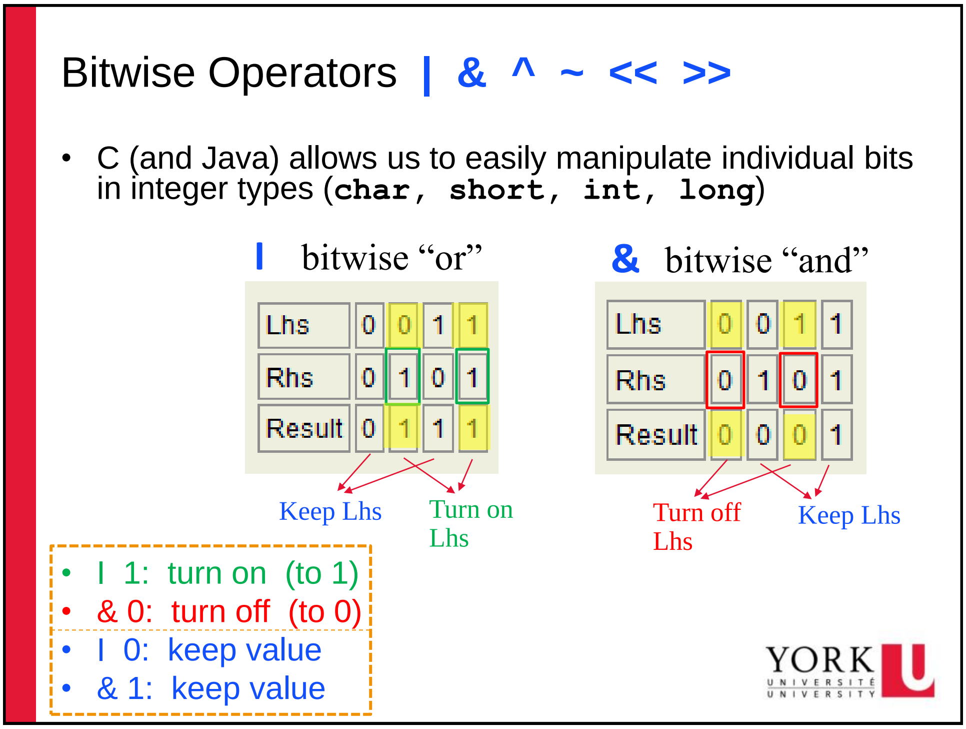
00000100 01011000 00001100 11101111

One int – 16 or 32
'Boolean' flags

37

YORK U UNIVERSITÉ UNIVERSITY

37



Bitwise Operators | & ^ ~ << >>

- C (and Java) allows us to easily manipulate individual bits in integer types (**`char, short, int, long`**)

bitwise "or"				
Lhs	0	0	1	1
Rhs	0	1	0	1
Result	0	1	1	1

Keep Lhs

Turn on Lhs

& bitwise "and"

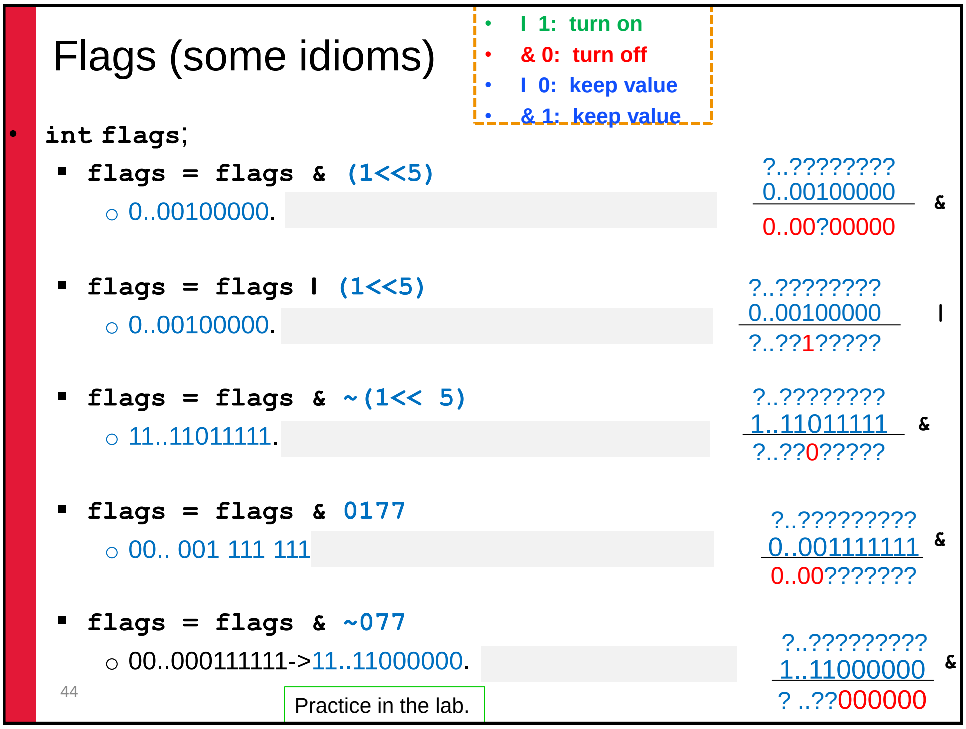
Lhs	0	0	1	1
Rhs	0	1	0	1
Result	0	0	0	1

Turn off Lhs

Keep Lhs

- | 1: turn on (to 1)
- & 0: turn off (to 0)
- | 0: keep value
- & 1: keep value

YORK UNIVERSITÉ UNIVERSITY



Flags (some idioms)

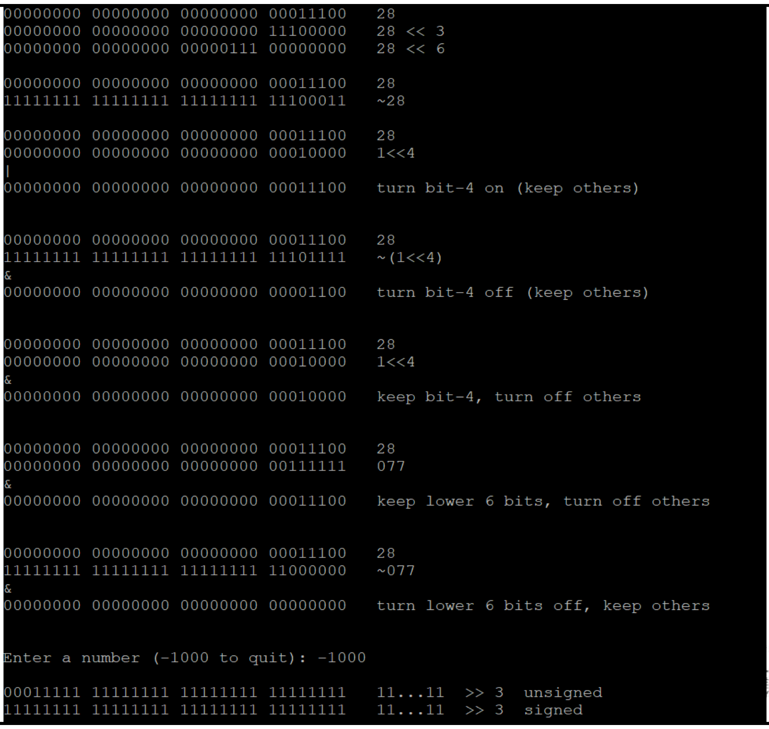
- | 1: turn on
- & 0: turn off
- | 0: keep value
- & 1: keep value

- **`int flags`**;
  - **`flags = flags & (1<<5)`**
    - 0..00100000.
  - **`flags = flags | (1<<5)`**
    - 0..00100000.
  - **`flags = flags & ~(1<< 5)`**
    - 11..11011111.
  - **`flags = flags & 0177`**
    - 00.. 001 111 111
  - **`flags = flags & ~077`**
    - 00..000111111->11..11000000.

```
?..????????
0..00100000   &
0..00?00000

?..????????
0..00100000   |
?..??1?????

?..????????
1..11011111   &
?..??0?????

?..?????????
0..001111111  &
0..00???????

?..?????????
1..11000000   &
? ..??000000
```

44

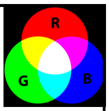
Practice in the lab.



```
00000000 00000000 00000000 00011100   28
00000000 00000000 00000000 11100000   28 << 3
00000000 00000000 00000111 00000000   28 << 6

00000000 00000000 00000000 00011100   28
11111111 11111111 11111111 11100011   ~28

00000000 00000000 00000000 00011100   28
00000000 00000000 00000000 00010000   1<<4
|
00000000 00000000 00000000 00011100   turn bit-4 on (keep others)


00000000 00000000 00000000 00011100   28
11111111 11111111 11111111 11101111   ~(1<<4)
&
00000000 00000000 00000000 00001100   turn bit-4 off (keep others)


00000000 00000000 00000000 00011100   28
00000000 00000000 00000000 00010000   1<<4
&
00000000 00000000 00000000 00010000   keep bit-4, turn off others


00000000 00000000 00000000 00011100   28
00000000 00000000 00000000 00111111   077
&
00000000 00000000 00000000 00011100   keep lower 6 bits, turn off others


00000000 00000000 00000000 00011100   28
11111111 11111111 11111111 11000000   ~077
&
00000000 00000000 00000000 00000000   turn lower 6 bits off, keep others


Enter a number (-1000 to quit): -1000

00011111 11111111 11111111 11111111   11...11  >> 3  unsigned
11111111 11111111 11111111 11111111   11...11  >> 3  signed
```

Some examples

- | 1: turn on
- & 0: turn off
- | 0: keep value
- & 1: keep value



- In Java, getRGB() packs 3 +1 values into a 32 bit (4 bytes) int
- How to get *red* value?

00001010 11111101 01001000 10101011

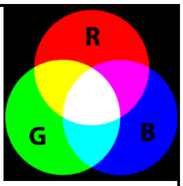
^ Alpha ^Red (253) ^Green (72) ^Blue(139)

Need to move “red bits” 11111101 to eight ends, turn off others.
How?

00000000 00000000 00000000 11111101   **253 (decimal)**

rgb << 8 >> 24 ? Okay if rgb is unsigned

Some examples



- 10101100 11111101 01001000 11111111
  ^ Alpha ^Red (253) ^Green (72) ^Blue (139)

How to get *red* value?

- First shift “red bits” to the right end (how?)

00000000 00000000 10101100 11111101

`rgb >> 16`

or

11111111 11111111 10101100 11111101

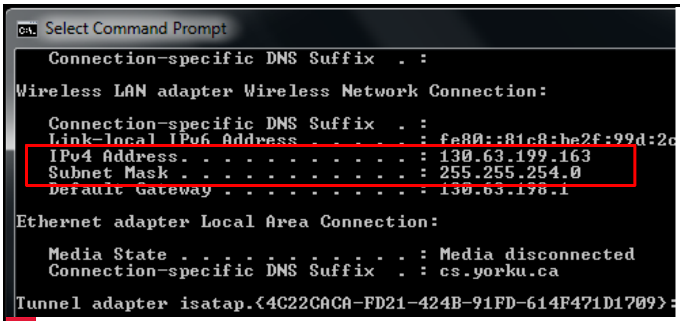
if signed (maybe)

What’s next

00000000 00000000 00000000 11111101

Stopped here

48

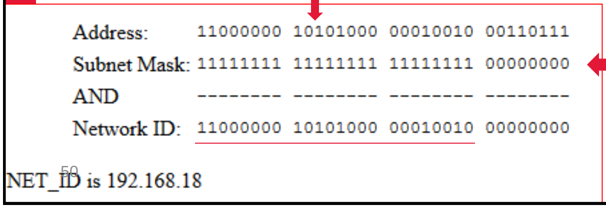


```
   Connection-specific DNS Suffix  . :

Wireless LAN adapter Wireless Network Connection:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::81c8:be2f:99d:2c
   IPv4 Address. . . . . . . . . . . : 130.63.199.163
   Subnet Mask . . . . . . . . . . . : 255.255.254.0
   Default Gateway . . . . . . . . . : 130.63.198.1

Ethernet adapter Local Area Connection:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . : cs.yorku.ca

Tunnel adapter isatap.{4C22CACA-FD21-424B-91FD-614F471D1709}:
```

Example: ip address 192.168.18.55, subnet mask: 255.255.255.0

11000000 10101000 00010010 00110111

11111111 11111111 11111111 00000000



Address:	11000000	10101000	00010010	00110111
Subnet Mask:	11111111	11111111	11111111	00000000
AND	--------	--------	--------	--------
Network ID:	11000000	10101000	00010010	00000000

50

NET_ID is 192.168.18

For your information

50

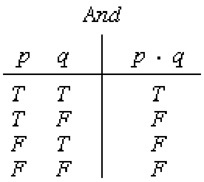
Somethings to Think About

- | looks similar to || Both do “OR”
- & looks similar to && Both do “AND”

- Can you substitute | for ||?
- Can you substitute & for &&?

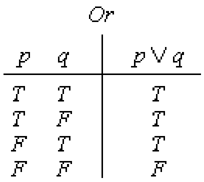
- | and & applies to bits, || and && apply to whole values

```
int x=1, y=2;
x && y    ?    1
x &  y    ?    0

x || y    ?    1
X |  y    ?    3
```



And

p	*q*	*p* · *q*
T	*T*	*T*
T	*F*	*F*
F	*T*	*F*
F	*F*	*F*



Or

p	*q*	*p* ∨ *q*
T	*T*	*T*
T	*F*	*T*
F	*T*	*T*
F	*F*	*F*

- ~ vs ! Both do “Negation” `~145` `!145`

Expressions

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

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

Expressions

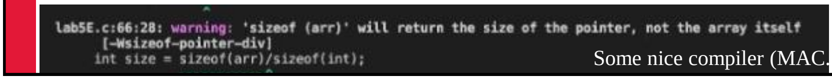
- sizeof

```
sizeof (int)

int a;
sizeof a;
```
or
```
sizeof (a);
```

- Not a function
- Don’t use **sizeof** on function array parameter

```
main(){
 char s[] = "Hello";
 printf("%d", sizeof s); // ?
 int a = indexOf(s, 'a');
}

int indexOf (char arr[], char c ) {
    for(i=0; i < sizeof arr; i++) …
```

`sizeof arr`
always 4 or 8
Explain later

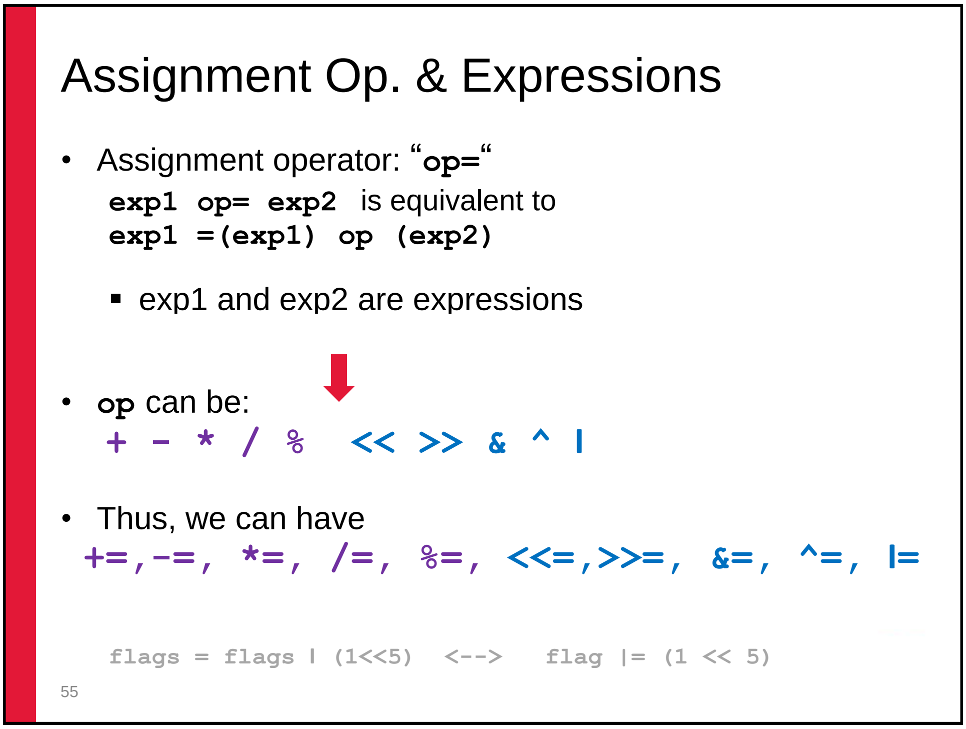


```
lab5E.c:66:28: warning: 'sizeof (arr)' will return the size of the pointer, not the array itself
      [-Wsizeof-pointer-div]
    int size = sizeof(arr)/sizeof(int);
```

Some nice compiler (MAC. not lab ☹)

(Compound) Assignment Operators

- C (and Java) provides other “short-hand” assignment operators (we’ve seen **++** and **--)**
- e.g.
  - `x += 5;      <-->      x = x + 5`
  - `x *= 5;      <-->      x = x * 5`



Assignment Op. & Expressions

- Assignment operator: “**op=**“

  exp1 op= exp2 is equivalent to
  exp1 =(exp1) op (exp2)

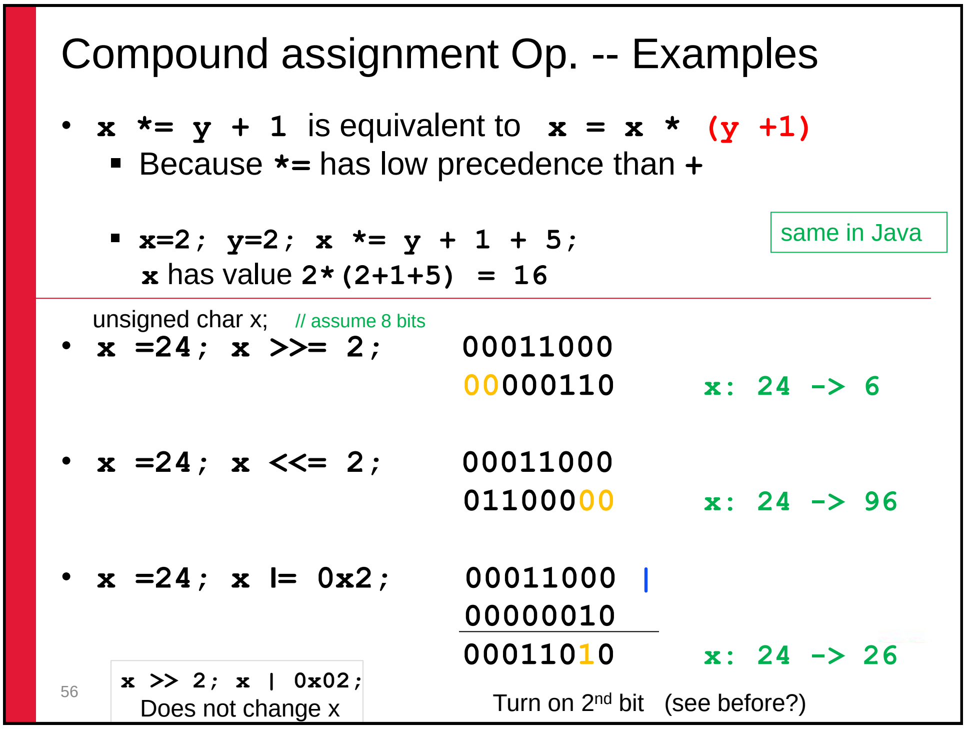
  - exp1 and exp2 are expressions

- **op** can be:
  `+ - * / %  << >> & ^ |`

- Thus, we can have
  `+=,-=, *=, /=, %=, <<=,>>=, &=, ^=, |=`

```
flags = flags | (1<<5)  <-->  flag |= (1 << 5)
```

55



Compound assignment Op. -- Examples

- **x *= y + 1** is equivalent to **x = x * (y +1)**
  - Because ***=** has low precedence than **+**
  - **x=2; y=2; x *= y + 1 + 5;**
    x has value **2*(2+1+5) = 16**

same in Java

unsigned char x; // assume 8 bits

- **x =24; x >>= 2;**

```
00011000
00000110      x: 24 -> 6
```

- **x =24; x <<= 2;**

```
00011000
01100000      x: 24 -> 96
```

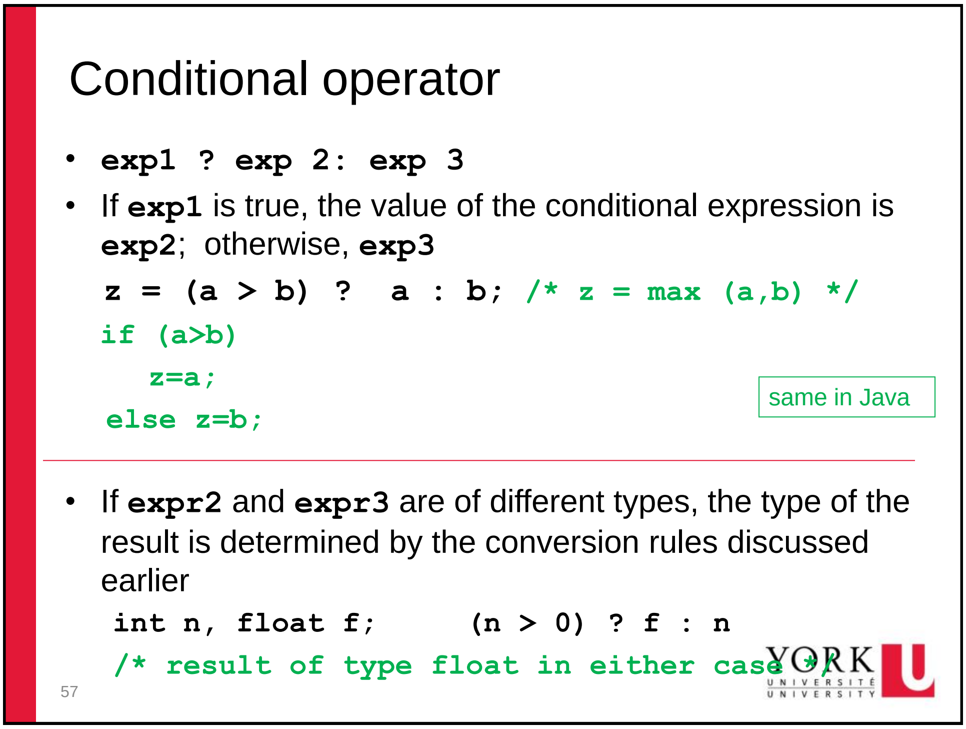
- **x =24; x |= 0x2;**

```
00011000 |
00000010
--------
00011010      x: 24 -> 26
```

Turn on 2nd bit (see before?)

x >> 2; x | 0x02;
Does not change x

56



Conditional operator

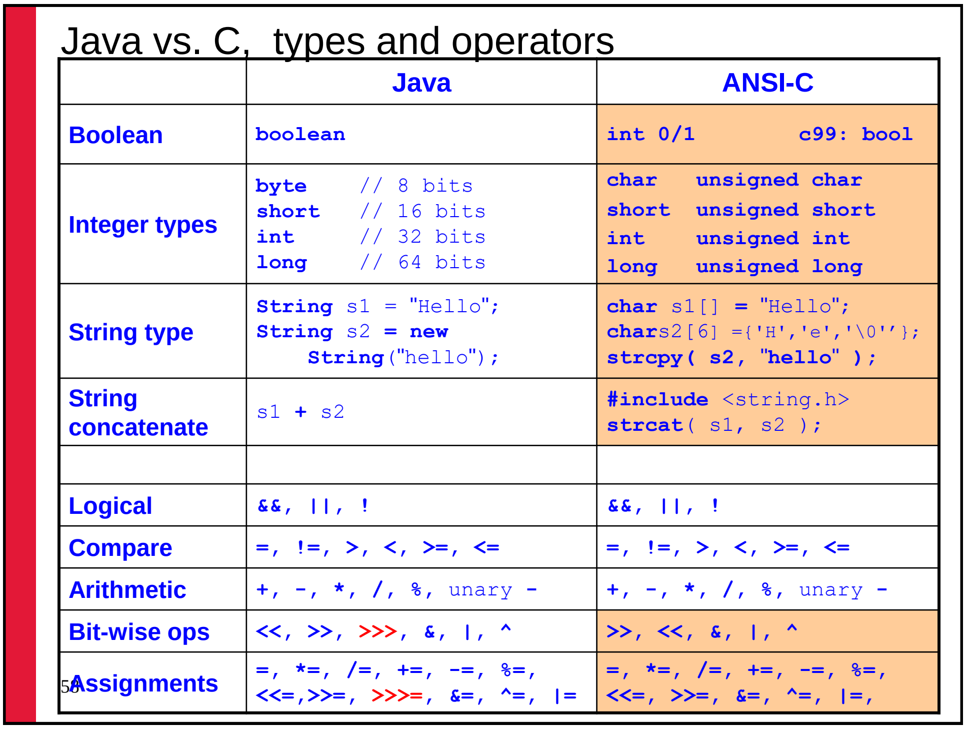
- **exp1 ? exp 2: exp 3**
- If **exp1** is true, the value of the conditional expression is **exp2**; otherwise, **exp3**

```
z = (a > b) ?  a : b; /* z = max (a,b) */
if (a>b)
   z=a;
else z=b;
```

same in Java

- If **expr2** and **expr3** are of different types, the type of the result is determined by the conversion rules discussed earlier

```
 int n, float f;     (n > 0) ? f : n
 /* result of type float in either case */
```



Java vs. C, types and operators

<table>
<tr><th></th><th>Java</th><th>ANSI-C</th></tr>
<tr><td>Boolean</td><td>boolean</td><td>int 0/1   c99: bool</td></tr>
<tr><td>Integer types</td><td>byte   // 8 bits
short   // 16 bits
int   // 32 bits
long   // 64 bits</td><td>char   unsigned char
short unsigned short
int   unsigned int
long   unsigned long</td></tr>
<tr><td>String type</td><td>String s1 = "Hello";
String s2 = new
    String("hello");</td><td>char s1[] = "Hello";
chars2[6] ={'H','e','\0''};
strcpy(s2, "hello");</td></tr>
<tr><td>String concatenate</td><td>s1 + s2</td><td>#include <string.h>
strcat(s1, s2);</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>Logical</td><td>&amp;&amp;, ||, !</td><td>&amp;&amp;, ||, !</td></tr>
<tr><td>Compare</td><td>=, !=, &gt;, <, &gt;=, &lt;=</td><td>=, !=, &gt;, <, &gt;=, &lt;=</td></tr>
<tr><td>Arithmetic</td><td>+, -, *, /, %, unary -</td><td>+, -, *, /, %, unary -</td></tr>
<tr><td>Bit-wise ops</td><td>&lt;&lt;, &gt;&gt;, &gt;&gt;&gt;, &amp;, |, ^</td><td>&gt;&gt;, &lt;&lt;, &amp;, |, ^</td></tr>
<tr><td>Assignments</td><td>=, *=, /=, +=, -=, %=,
<<=,>>=, >>&gt;=, &amp;=, ^=, |=</td><td>=, *=, /=, +=, -=, %=,
<<=, >&gt;=, &amp;=, ^=, |=,</td></tr>
</table>

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**

Precedence

- How do we interpret:
  - `a && b || c && d`
  - `i << 2 +1`   `flag | 1 << 4`
  - `i *= y+1`
  - `(int) f1/f2`

- Rules of precedence tell us what gets evaluated first:
  - `a && b || c && d`
  - `i << 2 + 1`   `flag | 1 << 4`
  - `i *=  y + 1`
  - `(int) f1 / f2`

Similar in Java

- Precedence should be familiar from basic math:
  - Given “**`x+y*5`**”, you evaluate “**`y*5`**” first:
    - `x + (y*5)`

Precedence

```
#include <stdio.h>

main(){
 int c;
 c = getchar();
 while(c != EOF)
 {
   putchar(c);
   c = getchar();/*read next*/
 }
}
```

Succinct code

```
#include <stdio.h>

main(){
 int c;

 while( c = getchar() != EOF )
 {
   putchar(c);

 }
}
```

YORK UNIVERSITÉ UNIVERSITY

Precedence p53 of K&R

- Observe that:
  - Parentheses, [ ] first
  - Negation(!,~) (cast) next
  - Arithmetic before Relational
    - Arithmetic: /, *, % before +-
  - Relational before Logical
    - Logical: && before ||
  - Bit shift << >> before & ^ |
  - Assignment += very low

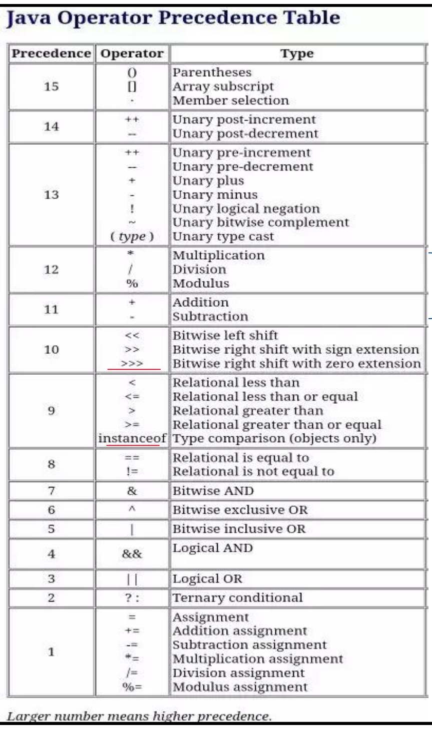
```
if ( a && b || c && d )
i << 2 + 1     // i << 3
flag | 1 << 4 // flag | 16
flag | ~(1 << 5)
x *= y + 1  // x=x*(y+1)
(int)f1/f2  // cast f1
while((c=getchar()) == EOF)
(*p).data
```

- When in doubt – use parentheses
  - also for clarity

```
flag | (1 << 4)
```

Similar in Java ➡

Operator Type	Operator
Primary Expression Operators	() [] . -> expr++ expr--
Unary Operators	* & + - ! ~ ++expr --expr (typecast) sizeof
Binary Operators	* / % arithmetic
	+ - arithmetic
	>> << bit shift
	< > <= >= relational
	== != relational
	& bitwise
	^ bitwise
	| bitwise
	&& logical
	|| logical
Ternary Operator	?:
Assignment Operators	= += -= *= /= %= >>= <<= &= ^= |=
Comma	,



Java Operator Precedence Table

Precedence	Operator	Type
15	() [] ·	Parentheses Array subscript Member selection
14	++ --	Unary post-increment Unary post-decrement
13	++ -- + - ! ~ (*type*)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast
12	* / %	Multiplication Division Modulus
11	+ -	Addition Subtraction
10	<< >> >>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension
9	< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)
8	== !=	Relational is equal to Relational is not equal to
7	&	Bitwise AND
6	^	Bitwise exclusive OR
5	|	Bitwise inclusive OR
4	&&	Logical AND
3	||	Logical OR
2	? :	Ternary conditional
1	= += -= *= /= %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment

Larger number means higher precedence.

arithmetic

bit shifting

```
System.out.printf("%d\n", -1 >> 3); // -1
System.out.printf("%d\n", -1 >>> 3); // 536870911
```

relational

bitwise

logical

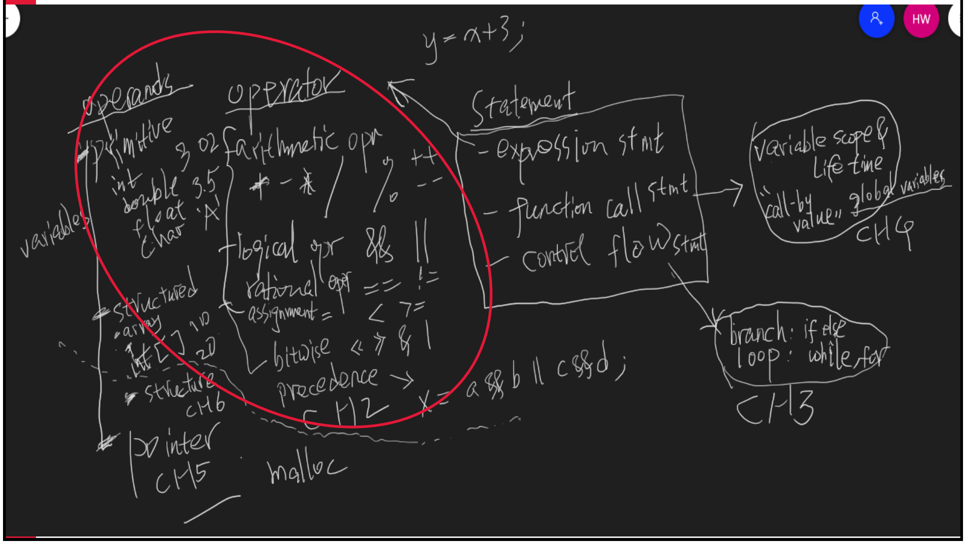
(compound) assignment

For your information

Summary of ch2

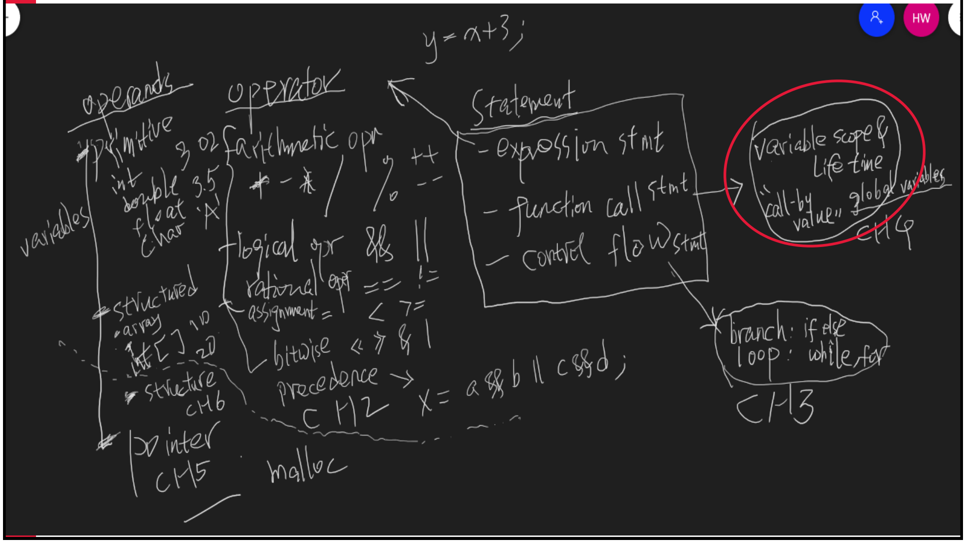
- Type, operators and expressions (Chapter 2）：
  - Types and sizes
    - Basic types, their size and constant values (literals)
      - ✓ char: x > 'a' && x < 'z';   x > '0' && x < '9'
      - ✓ int: 122, 0122, 0x12F   convert between Decimal, Bin, Oct, Hex
    - Arrays (one dimension) and strings (Ch1.6,1.9)
      - ✓ "hello" has size 6 byte | H | e | l | l | o | \0 |
  - Expressions
    - Basic operators (arithmetic, relational and logical)
      - ✓ y=x++; y=++x;
      - ✓ int as Boolean   !0   !-3   if (x = 2)
    - Type conversion and promotion
    - Other operators (bitwise, bit shifting , compound assignment, conditional), sizeof
      - ✓ Bit: |, &, ~. ^, << >>
      - ✓ Compound: x += 10;   x >>= 10;   x += y + 3
    - Precedence of operators
- Next: Functions and Program Structure (Chapter 4)

Roadmap -- How the topics are related



66

Roadmap -- How the topics are related



72

COSC2031 - Software Tools

Functions and Program Structure
(K+R Ch.1.5-10, Ch.4)

Program Structure

- C programs consist of a set of variables and functions.
  - we have discussed variables, expressions (ch2) and control flow (ch3).
  - now let’s combine these into a program (ch4)
- C programs consist of statements
  - expression statements (ch2)
  - control flow statements (ch3)
  - block, function call statements (ch4)

- C program structure – Functions
  - Communication
  - “Pass-by-value”
- Categories, scope and lifetime of variables (and functions)
- C Preprocessing
- Recursions

Declaring Functions (review)

- Either a declaration or a definition must be present prior to any call of the function.

- Declaring a function before using it, if it is defined in
  - library e.g., include <stdio.h>
  - later in the same source file
  - **another source file of the program**

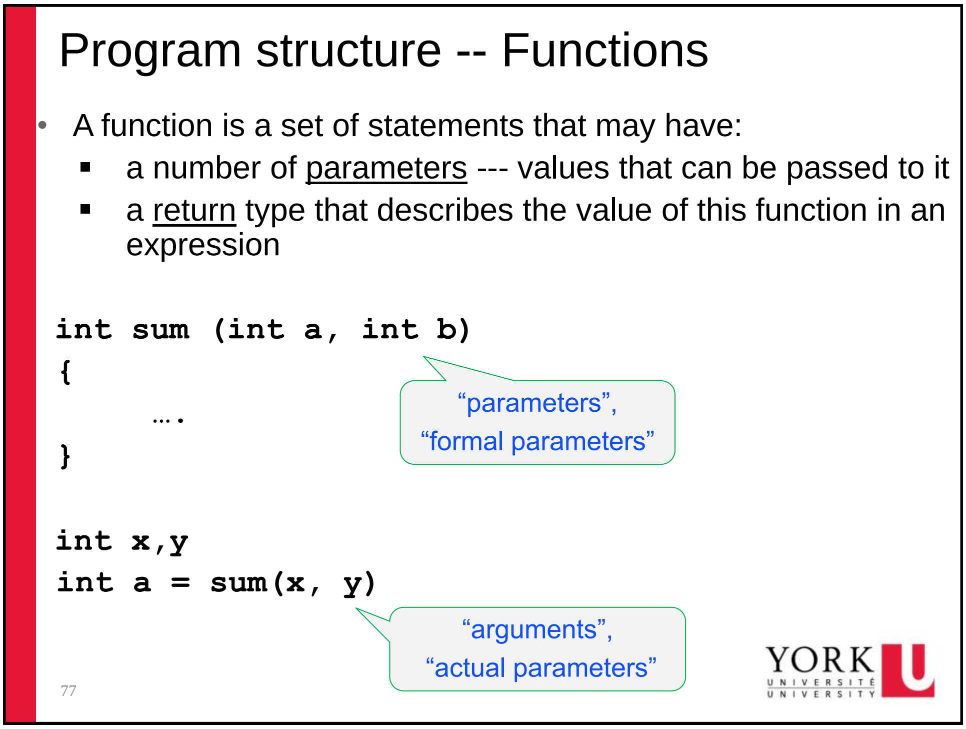
- Declaring a function tells its **return type** and **parameters** but not its code.

```
int power (int base, int pow);
```

- We can omit parameter names

```
int power  (int, int);
```

  - The type of parameters (and return type) is what matters for compiler



Program structure -- Functions

- A function is a set of statements that may have:
  - a number of parameters --- values that can be passed to it
  - a return type that describes the value of this function in an expression

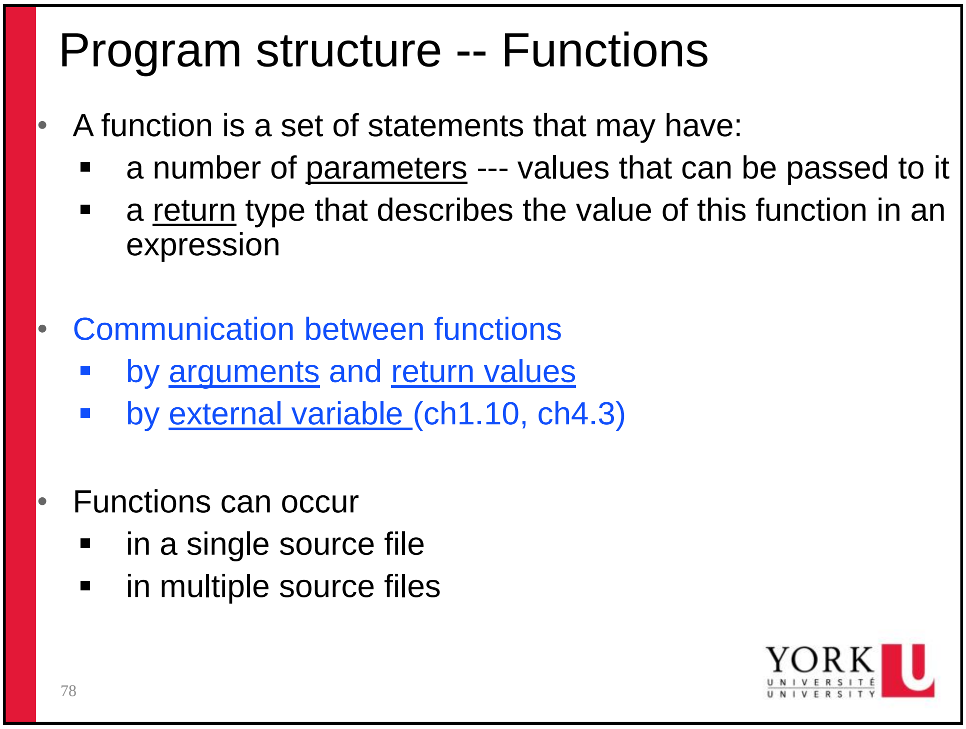
```
int sum (int a, int b)
{
      ….
}

int x,y
int a = sum(x, y)
```

“parameters”,
“formal parameters”

“arguments”,
“actual parameters”

77

77

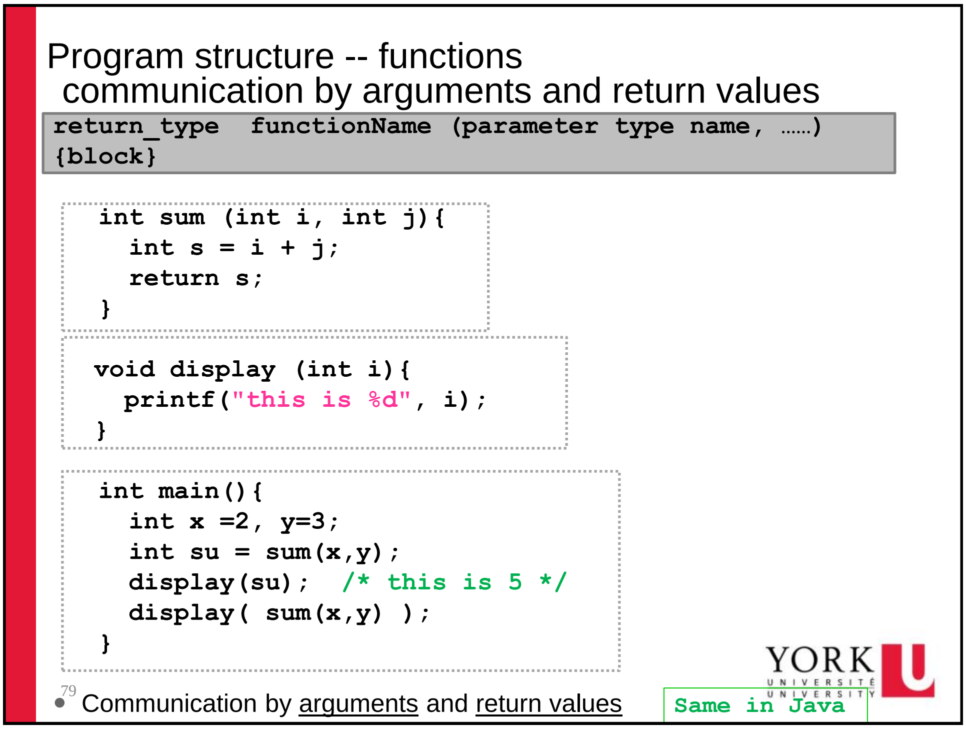


Program structure -- Functions

- A function is a set of statements that may have:
  - a number of parameters --- values that can be passed to it
  - a return type that describes the value of this function in an expression
- Communication between functions
  - by arguments and return values
  - by external variable (ch1.10, ch4.3)
- Functions can occur
  - in a single source file
  - in multiple source files

78

78



Program structure -- functions communication by arguments and return values

```
return_type  functionName (parameter type name, ……)
{block}
```

```
  int sum (int i, int j){
    int s = i + j;
    return s;
  }
```

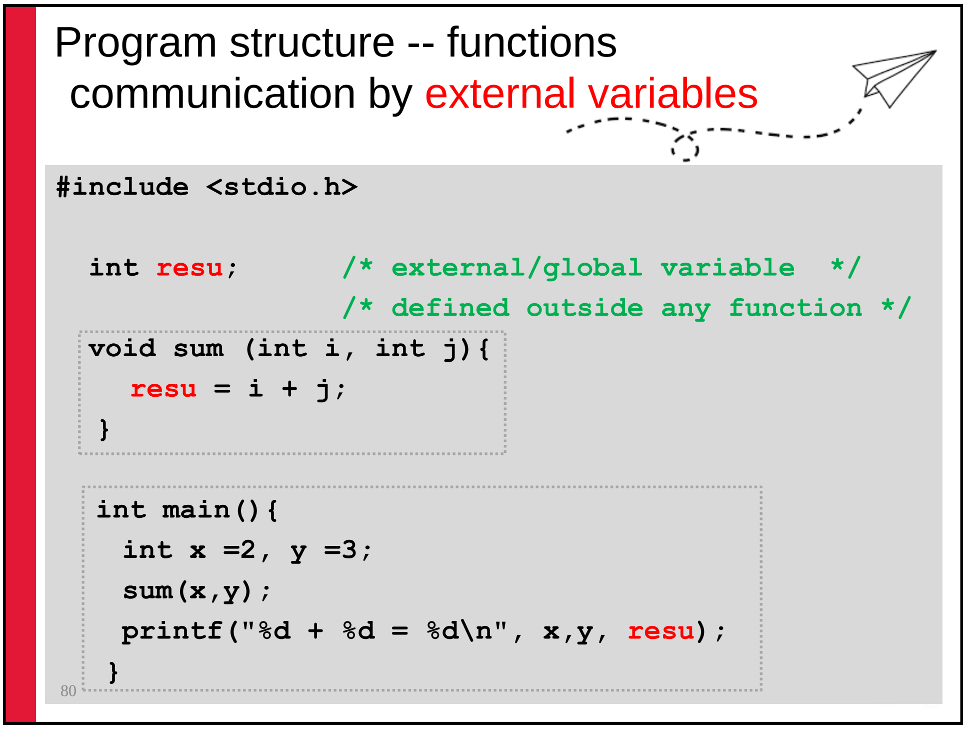
```
  void display (int i){
    printf("this is %d", i);
  }
```

```
  int main(){
    int x =2, y=3;
    int su = sum(x,y);
    display(su);  /* this is 5 */
    display( sum(x,y) );
  }
```

79

- Communication by arguments and return values

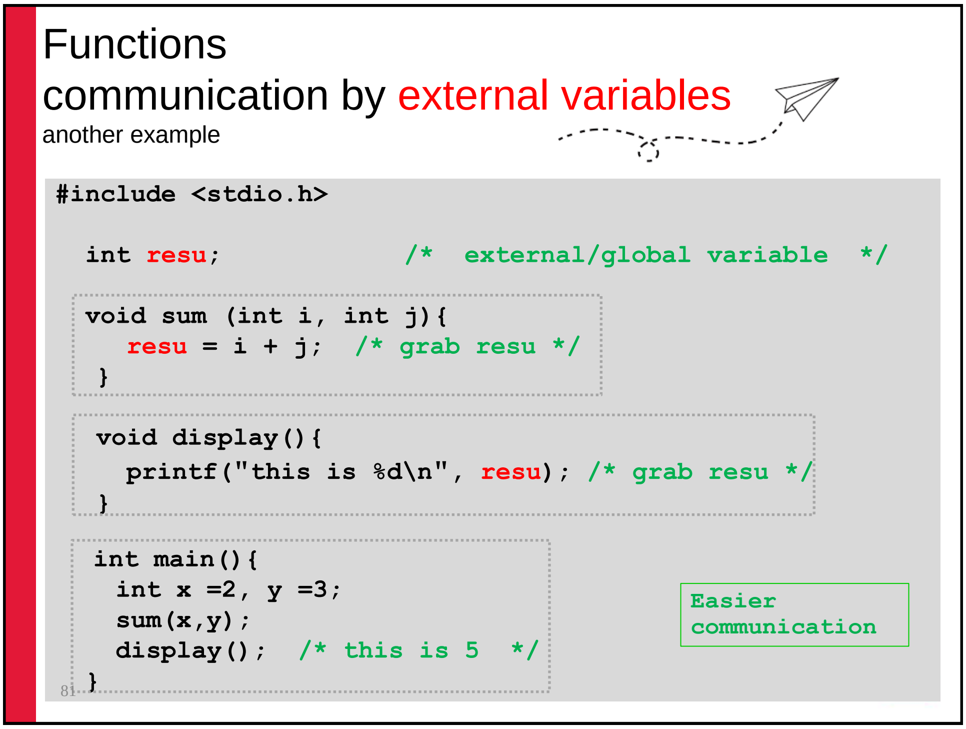
Same in Java



Program structure -- functions communication by external variables

```
#include <stdio.h>

  int resu;        /* external/global variable  */
                   /* defined outside any function */
  void sum (int i, int j){
     resu = i + j;
   }

   int main(){
     int x =2, y =3;
     sum(x,y);
     printf("%d + %d = %d\n", x,y, resu);
    }
```

80



Functions

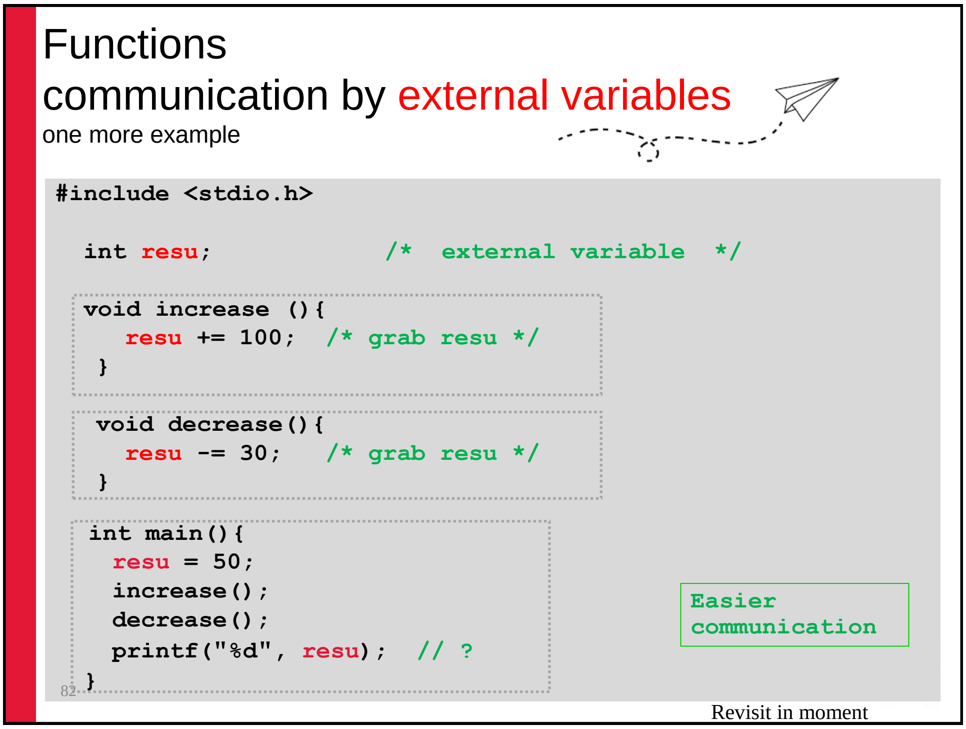
communication by external variables

another example

```
#include <stdio.h>

  int resu;              /*  external/global variable  */

  void sum (int i, int j){
     resu = i + j;  /* grab resu */
   }

   void display(){
     printf("this is %d\n", resu); /* grab resu */
   }

   int main(){
     int x =2, y =3;
     sum(x,y);
     display();  /* this is 5  */
  }
```

Easier communication

81



Functions

communication by external variables

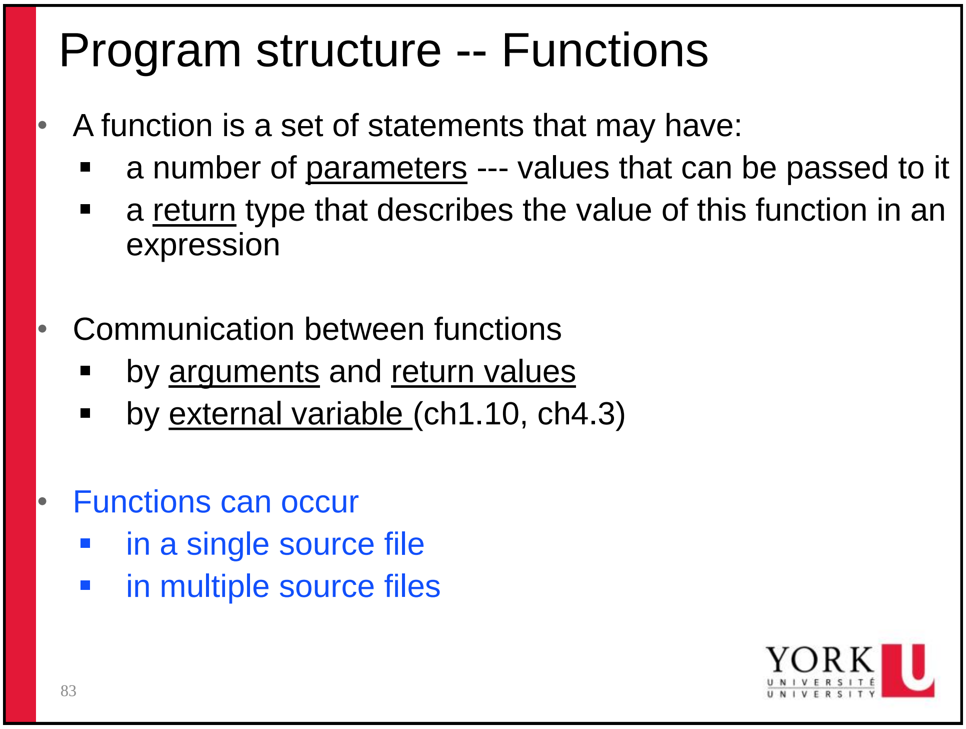
one more example

```
#include <stdio.h>

  int resu;              /*  external variable  */

  void increase (){
     resu += 100;  /* grab resu */
   }

   void decrease(){
     resu -= 30;   /* grab resu */
   }

  int main(){
    resu = 50;
    increase();
    decrease();
    printf("%d", resu);   // ?
  }
```

Easier communication

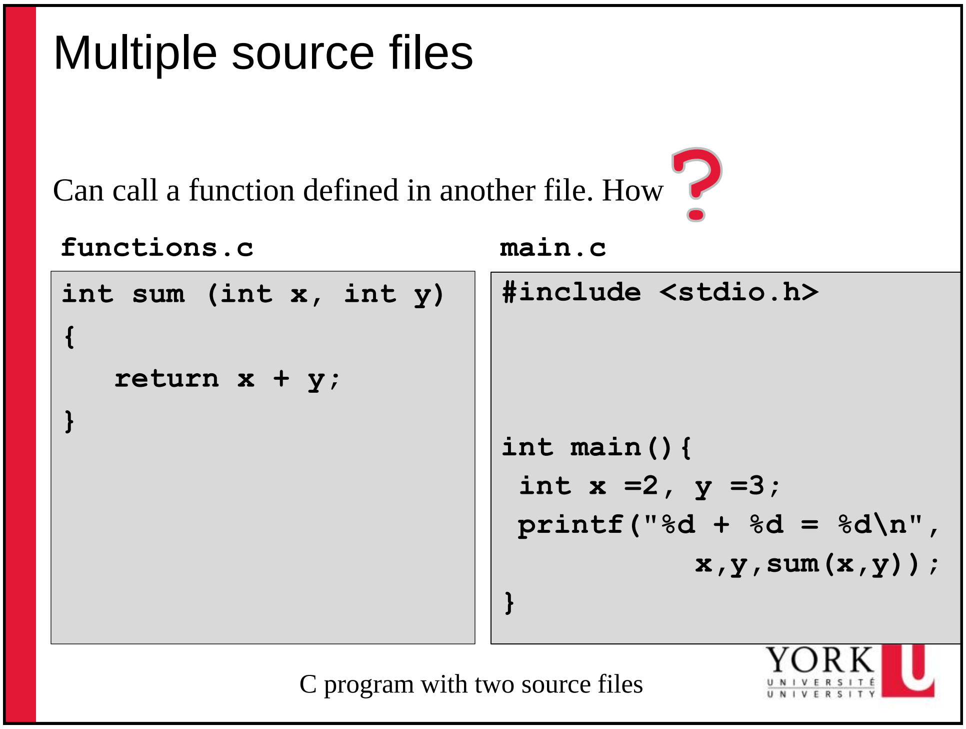
Revisit in moment

82



Program structure -- Functions

- A function is a set of statements that may have:
  - a number of parameters --- values that can be passed to it
  - a return type that describes the value of this function in an expression
- Communication between functions
  - by arguments and return values
  - by external variable (ch1.10, ch4.3)
- Functions can occur
  - in a single source file
  - in multiple source files



Multiple source files

Can call a function defined in another file. How ?

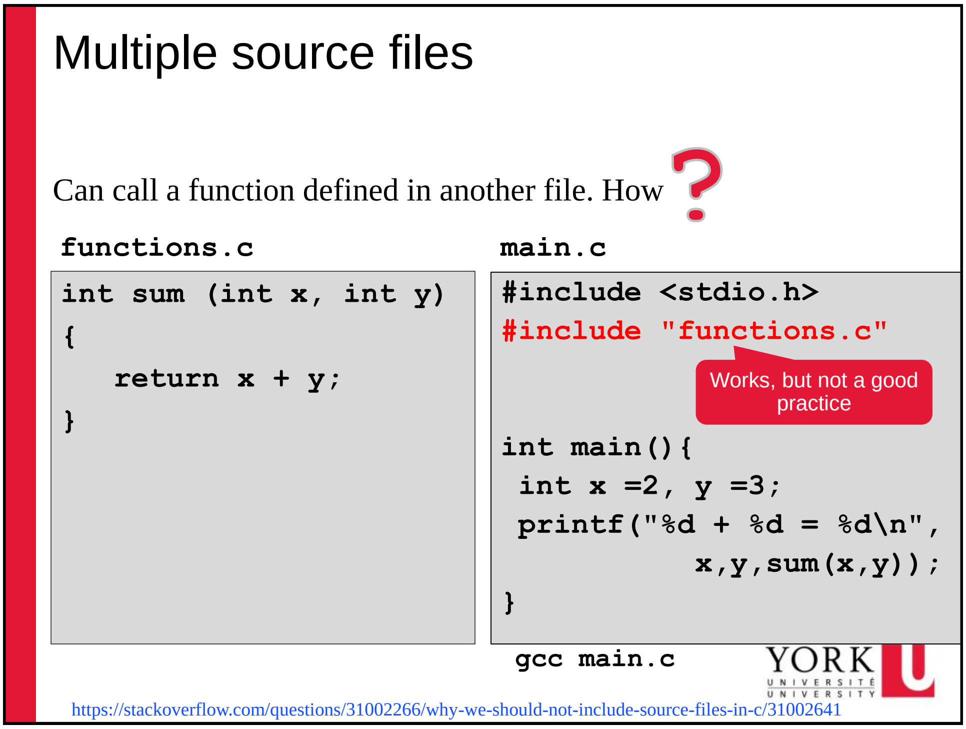
functions.c

```
int sum (int x, int y)
{
   return x + y;
}
```

main.c

```
#include <stdio.h>



int main(){
 int x =2, y =3;
 printf("%d + %d = %d\n",
           x,y,sum(x,y));
}
```

C program with two source files



Multiple source files

Can call a function defined in another file. How ?

`functions.c`

```
int sum (int x, int y)
{
    return x + y;
}
```

`main.c`

```
#include <stdio.h>
#include "functions.c"


int main(){
 int x =2, y =3;
 printf("%d + %d = %d\n",
          x,y,sum(x,y));
}
```

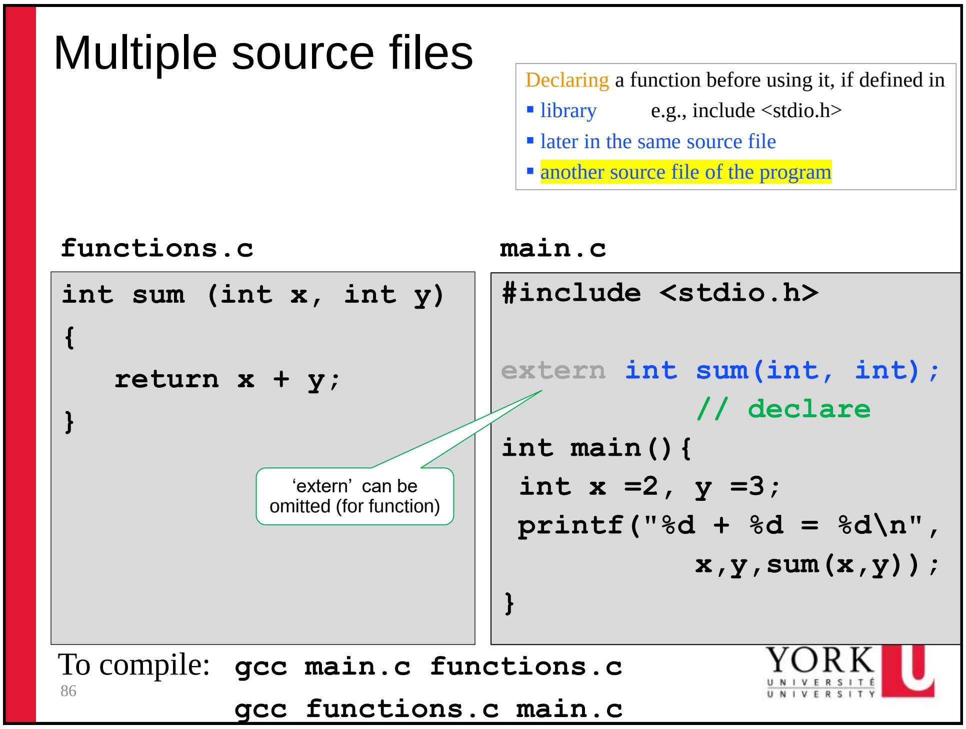
Works, but not a good practice

```
gcc main.c
```

YORK UNIVERSITÉ UNIVERSITY

https://stackoverflow.com/questions/31002266/why-we-should-not-include-source-files-in-c/31002641

85



Multiple source files

Declaring a function before using it, if defined in

- library   e.g., include <stdio.h>
- later in the same source file
- another source file of the program

`functions.c`

```
int sum (int x, int y)
{
    return x + y;
}
```

'extern' can be omitted (for function)

`main.c`

```
#include <stdio.h>

extern int sum(int, int);
              // declare
int main(){
 int x =2, y =3;
 printf("%d + %d = %d\n",
          x,y,sum(x,y));
}
```

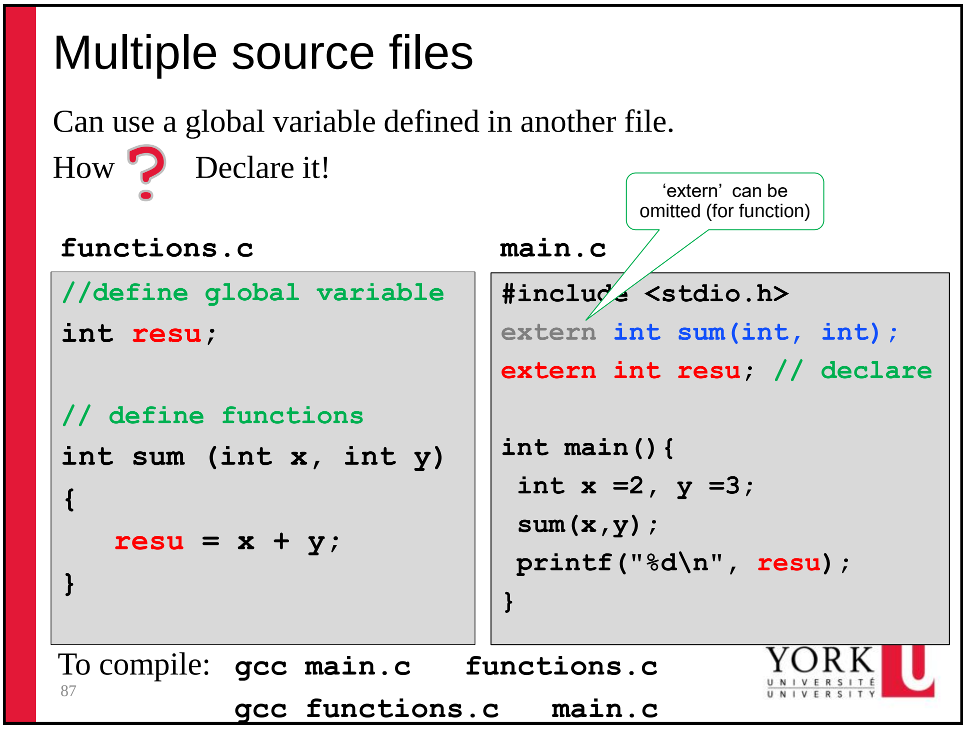
To compile:

```
gcc main.c functions.c
gcc functions.c main.c
```

YORK UNIVERSITÉ UNIVERSITY

86

86



Multiple source files

Can use a global variable defined in another file.

How ? Declare it!

'extern' can be omitted (for function)

functions.c

```
//define global variable
int resu;

// define functions
int sum (int x, int y)
{
   resu = x + y;
}
```

main.c

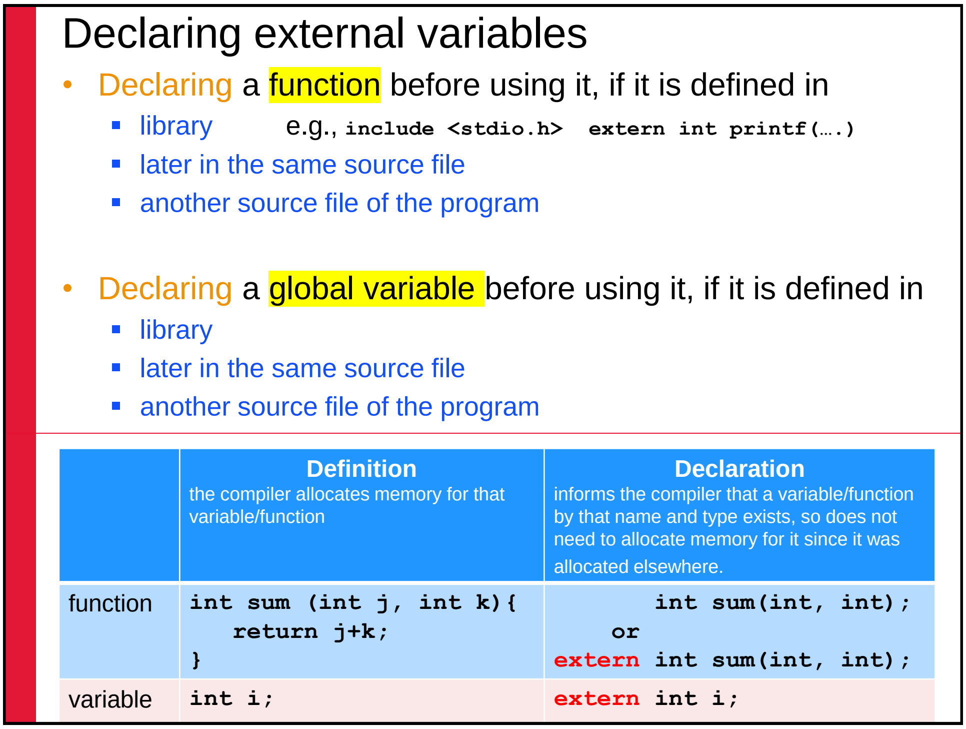
```
#include <stdio.h>
extern int sum(int, int);
extern int resu; // declare

int main(){
 int x =2, y =3;
 sum(x,y);
 printf("%d\n", resu);
}
```

To compile: `gcc main.c functions.c`

`gcc functions.c main.c`

87

87



Declaring external variables

- Declaring a function before using it, if it is defined in
  - library e.g., `include <stdio.h>` `extern int printf(….)`
  - later in the same source file
  - another source file of the program
- Declaring a global variable before using it, if it is defined in
  - library
  - later in the same source file
  - another source file of the program

	Definition the compiler allocates memory for that variable/function	**Declaration** informs the compiler that a variable/function by that name and type exists, so does not need to allocate memory for it since it was allocated elsewhere.

function	`int sum (int j, int k){`
	`    return j+k;`
	`}`
		`int sum(int, int);`
		or
		`extern int sum(int, int);`

variable	`int i;`	`extern int i;`

88

- C program structure – Functions
  - Communication
  - “Pass-by-value”
- Categories, scope and lifetime of variables (and functions)
- C Preprocessing
- Recursion

YORK UNIVERSITÉ UNIVERSITY

“Call (pass) by Value” vs “Call (pass) by reference”

- So what is the question?

```
int sum (int x, int y)
{
   int s = x + y;
   return s;
}

main(…){
   int i=3, j=4;
   int k = sum(i,j);
}
```

When `sum(i,j)` is called, what happens to arguments `i` and `j`?

- `sum` gets `i`, `j` themselves

or,

- `sum` gets copies of `i`, `j`

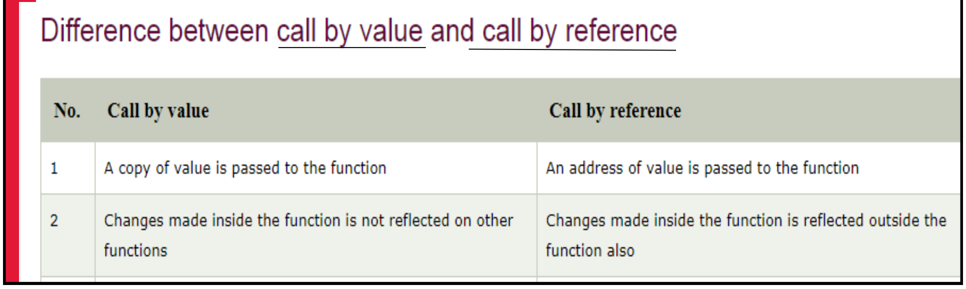
YORK UNIVERSITÉ UNIVERSITY

“call (pass) by value” vs “call by reference”

- So what is the question?

When `sum(int x, int y)` is called with `sum(i,j)`, what happens to arguments `i j`?

- `i j` themselves passed to `sum()` -- **“pass by reference”**
  - `x y` are alias of `i j` `x++` changes `i`
- copies of `i j` are passed to `sum()` -- **“pass by value”**
  - `x y` are copies of `i j` `x++` does not change `i`



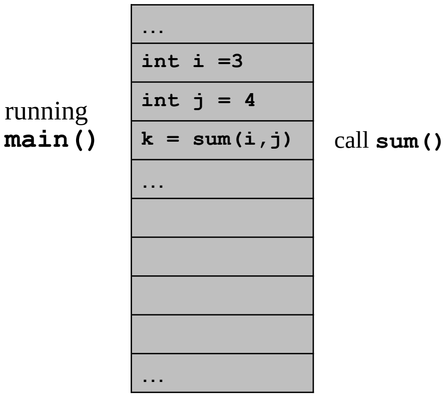
Difference between call by value and call by reference

No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also

Call (pass)-by-Value

- In C (and JAVA), all functions are call-by-value
  - Values of the arguments are passed to functions,
  - But NOT the arguments themselves (call-by-reference)

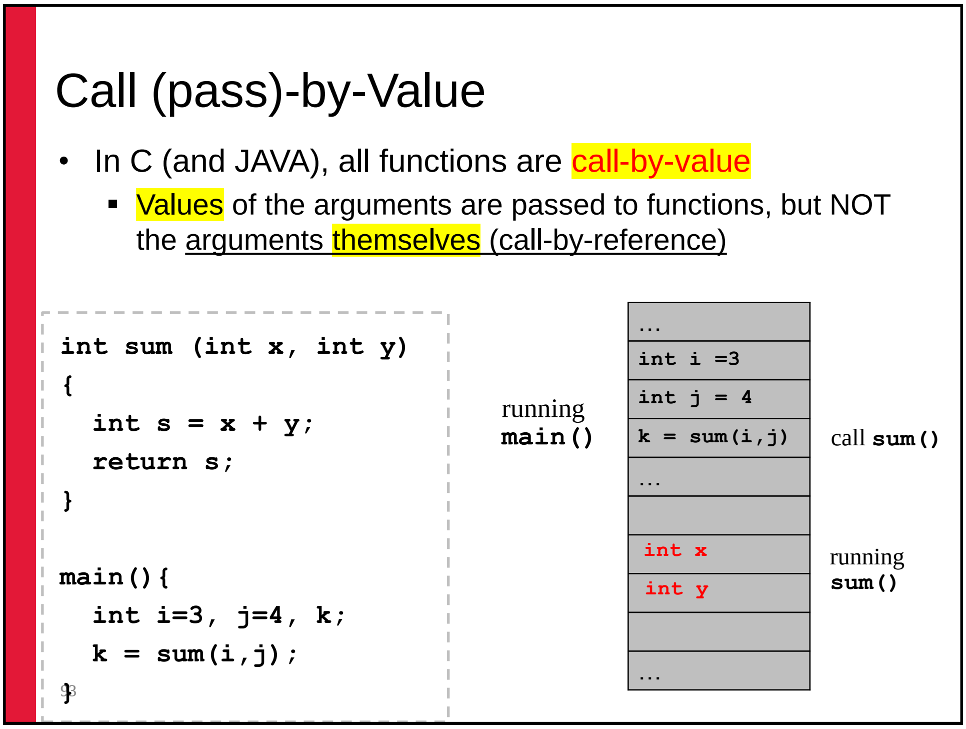
```
int sum (int x, int y)
{
   int s = x + y;
   return s;
}

main(){
   int i=3, j=4, k;
   k = sum(i,j);
}
```



running `main()`

…
`int i =3`
`int j = 4`
`k = sum(i,j)`
…
…

call `sum()`



Call (pass)-by-Value

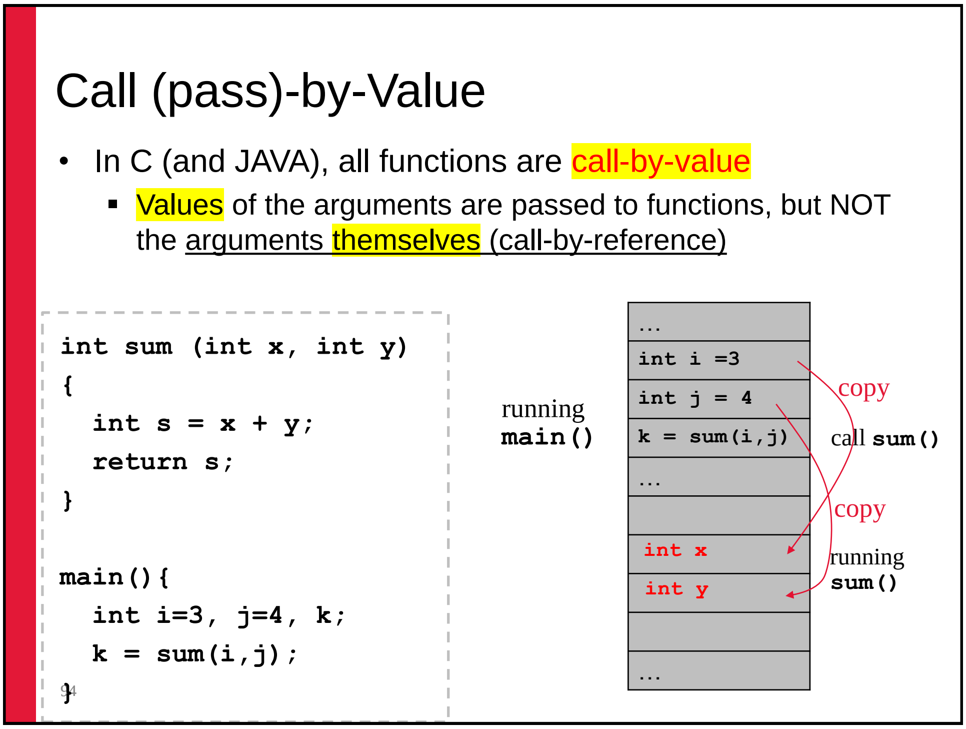
- In C (and JAVA), all functions are call-by-value
  - Values of the arguments are passed to functions, but NOT the arguments themselves (call-by-reference)

```
int sum (int x, int y)
{
   int s = x + y;
   return s;
}

main(){
   int i=3, j=4, k;
   k = sum(i,j);
}
```

	…	
	int i =3	
running **main()**	int j = 4	
	k = sum(i,j)	call **sum()**
	…	
	int x	running **sum()**
	int y	
	…	

93

93



Call (pass)-by-Value

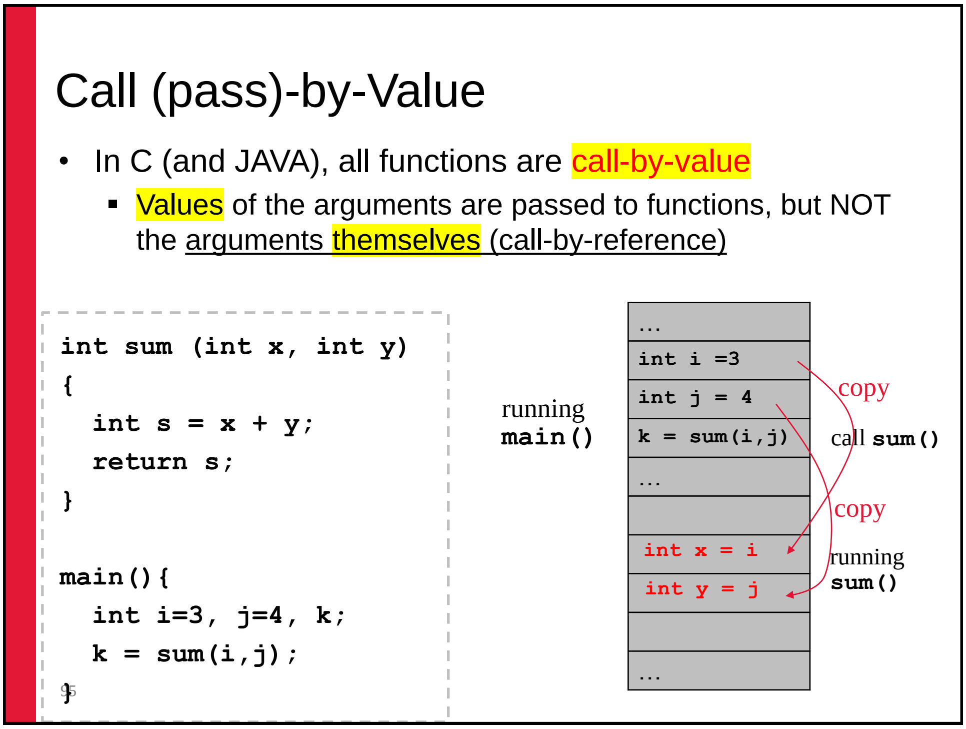
- In C (and JAVA), all functions are call-by-value
  - Values of the arguments are passed to functions, but NOT the arguments themselves (call-by-reference)

```
int sum (int x, int y)
{
   int s = x + y;
   return s;
}

main(){
   int i=3, j=4, k;
   k = sum(i,j);
}
```

	…	
	int i =3	copy
running **main()**	int j = 4	
	k = sum(i,j)	call **sum()**
	…	
		copy
	int x	running **sum()**
	int y	
	…	

94

94



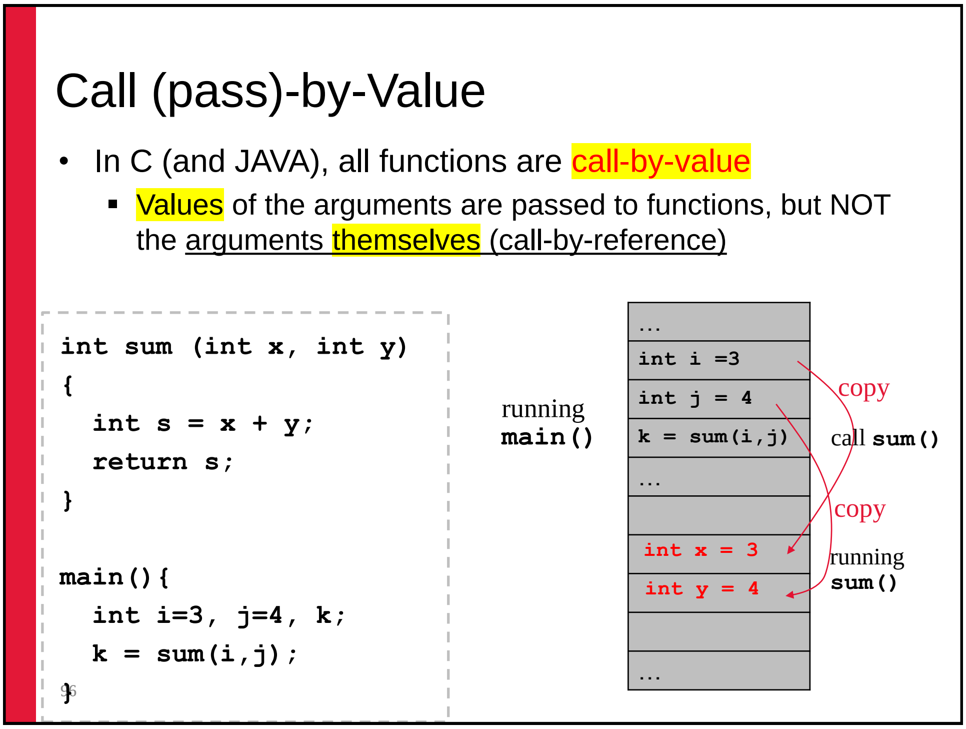
Call (pass)-by-Value

- In C (and JAVA), all functions are call-by-value
  - Values of the arguments are passed to functions, but NOT the arguments themselves (call-by-reference)

```
int sum (int x, int y)
{
  int s = x + y;
  return s;
}

main(){
  int i=3, j=4, k;
  k = sum(i,j);
}
```

```
…
int i =3
int j = 4
k = sum(i,j)
…

int x = i
int y = j

…
```

running main()
copy
call sum()
copy
running sum()
95



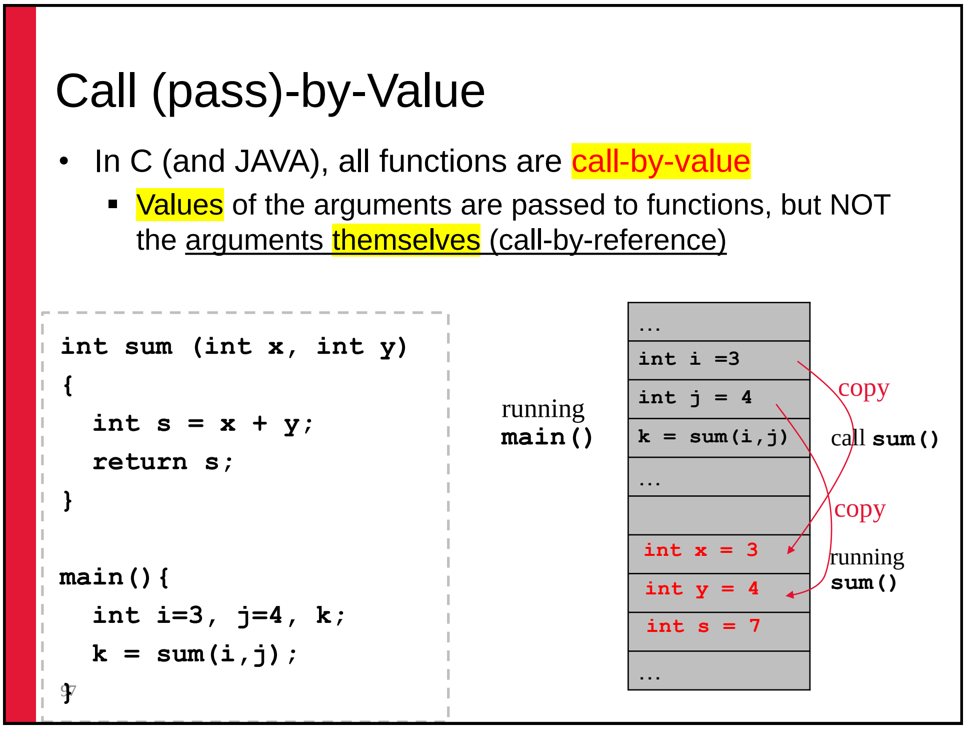
Call (pass)-by-Value

- In C (and JAVA), all functions are call-by-value
  - Values of the arguments are passed to functions, but NOT the arguments themselves (call-by-reference)

```
int sum (int x, int y)
{
  int s = x + y;
  return s;
}

main(){
  int i=3, j=4, k;
  k = sum(i,j);
}
```

```
…
int i =3
int j = 4
k = sum(i,j)
…

int x = 3
int y = 4

…
```

running main()
copy
call sum()
copy
running sum()
96



Call (pass)-by-Value

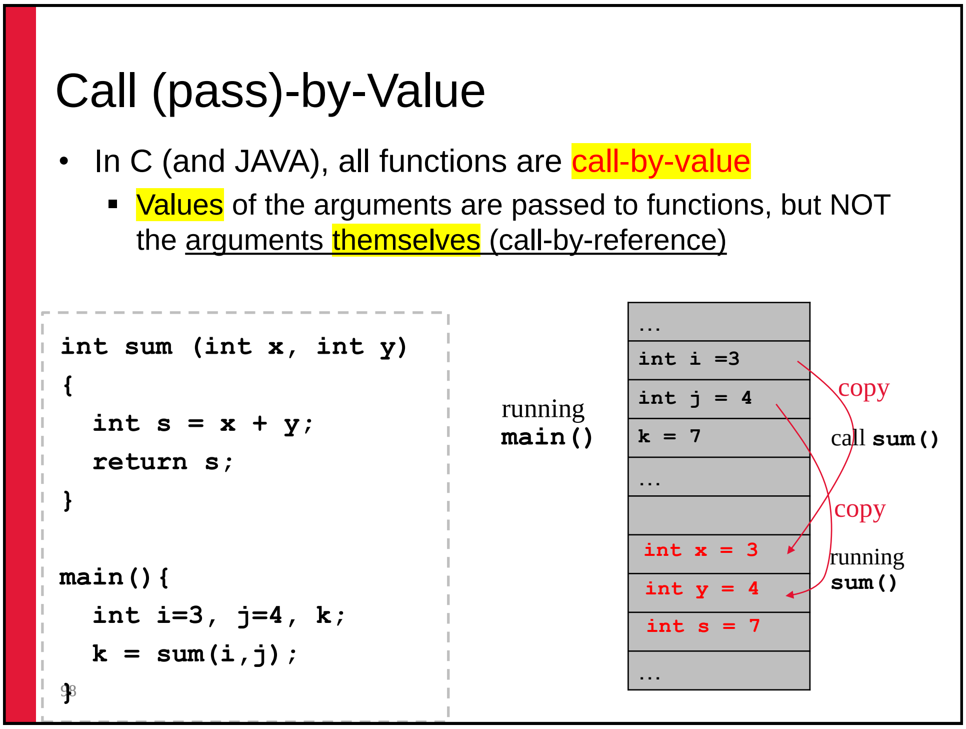
- In C (and JAVA), all functions are call-by-value
  - Values of the arguments are passed to functions, but NOT the arguments themselves (call-by-reference)

```
int sum (int x, int y)
{
   int s = x + y;
   return s;
}

main(){
   int i=3, j=4, k;
   k = sum(i,j);
}
```

…
int i =3
int j = 4
k = sum(i,j)
…
int x = 3
int y = 4
int s = 7
…

running
main()
copy
call sum()
copy
running
sum()

97



Call (pass)-by-Value

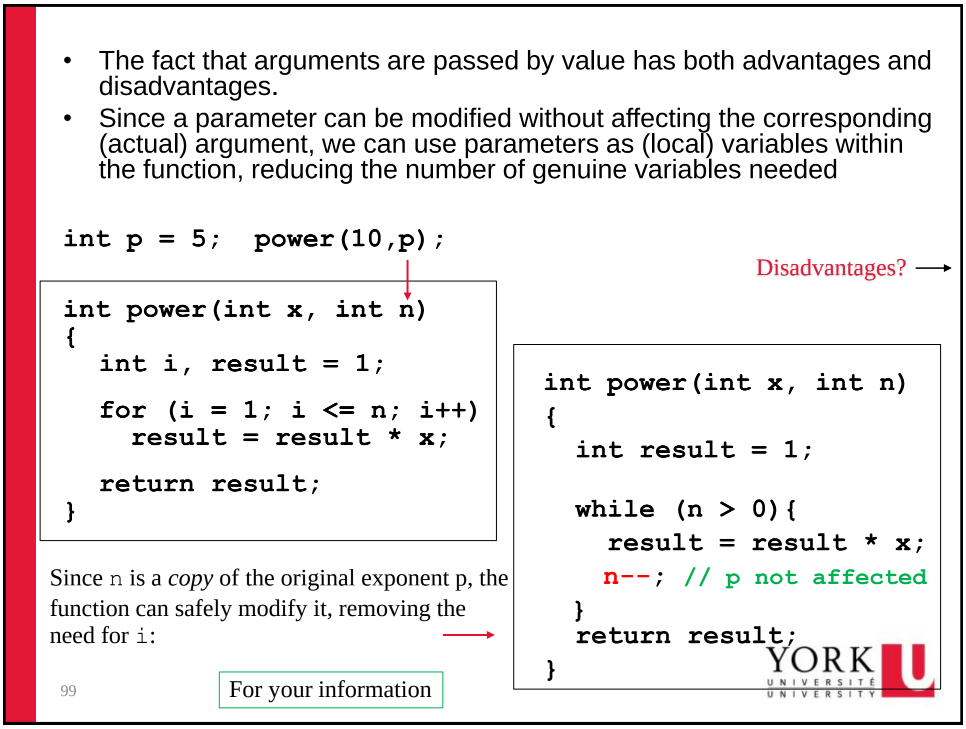
- In C (and JAVA), all functions are call-by-value
  - Values of the arguments are passed to functions, but NOT the arguments themselves (call-by-reference)

```
int sum (int x, int y)
{
   int s = x + y;
   return s;
}

main(){
   int i=3, j=4, k;
   k = sum(i,j);
}
```

…
int i =3
int j = 4
k = 7
…
int x = 3
int y = 4
int s = 7
…

running
main()
copy
call sum()
copy
running
sum()

98



- The fact that arguments are passed by value has both advantages and disadvantages.
- Since a parameter can be modified without affecting the corresponding (actual) argument, we can use parameters as (local) variables within the function, reducing the number of genuine variables needed

```
int p = 5;  power(10,p);
```

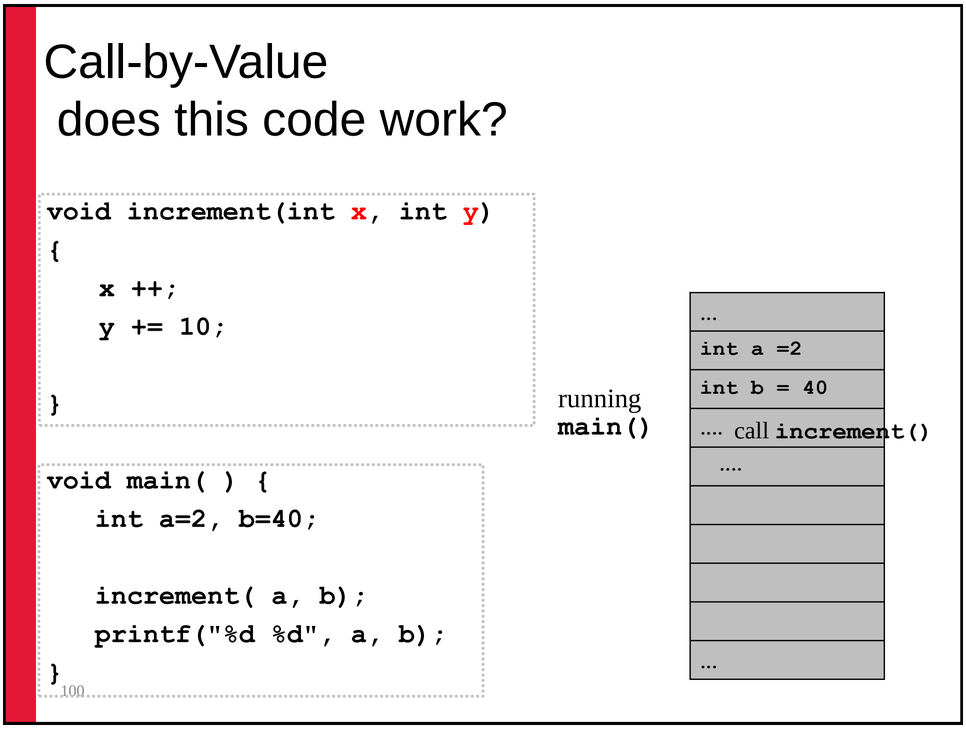
Disadvantages? →

```
int power(int x, int n)
{
  int i, result = 1;

  for (i = 1; i <= n; i++)
    result = result * x;

  return result;
}
```

Since `n` is a *copy* of the original exponent p, the function can safely modify it, removing the need for `i`:

```
int power(int x, int n)
{
  int result = 1;

  while (n > 0){
    result = result * x;
   n--; // p not affected
  }
  return result;
}
```

For your information



Call-by-Value
does this code work?

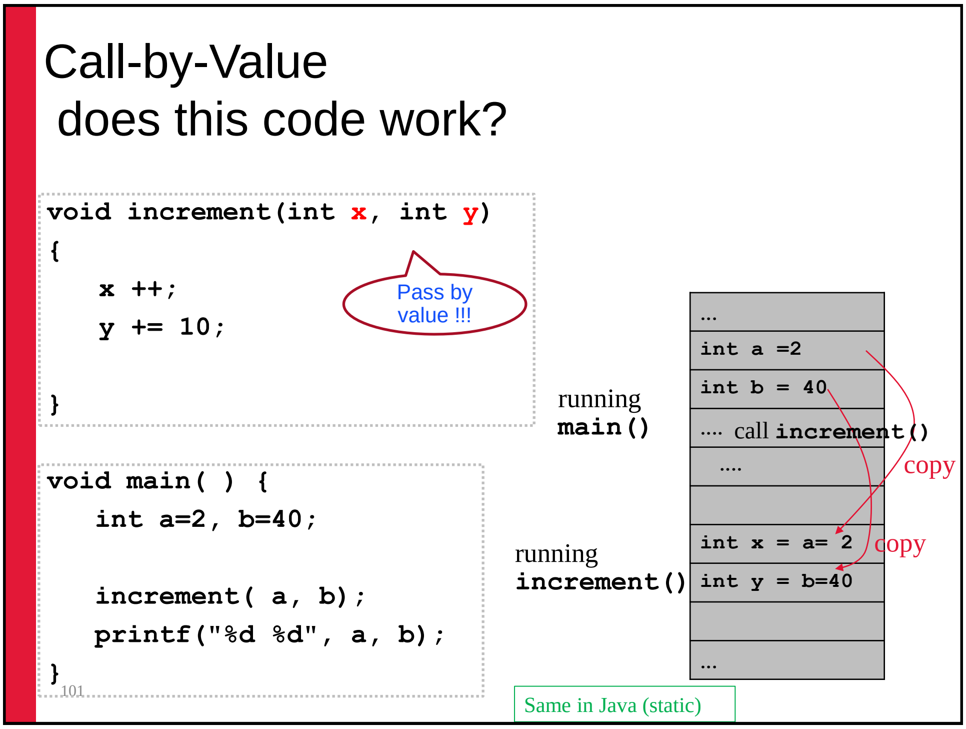
```
void increment(int x, int y)
{
   x ++;
   y += 10;


}
```

```
void main( ) {
   int a=2, b=40;

   increment( a, b);
   printf("%d %d", a, b);
}
```

running `main()`

…
`int a =2`
`int b = 40`
…. call `increment()`
….

…



Call-by-Value
 does this code work?

```
void increment(int x, int y)
{
    x ++;
    y += 10;

}
```

Pass by value !!!

```
void main( ) {
    int a=2, b=40;

    increment( a, b);
    printf("%d %d", a, b);
}
```

101

running
main()

running
increment()

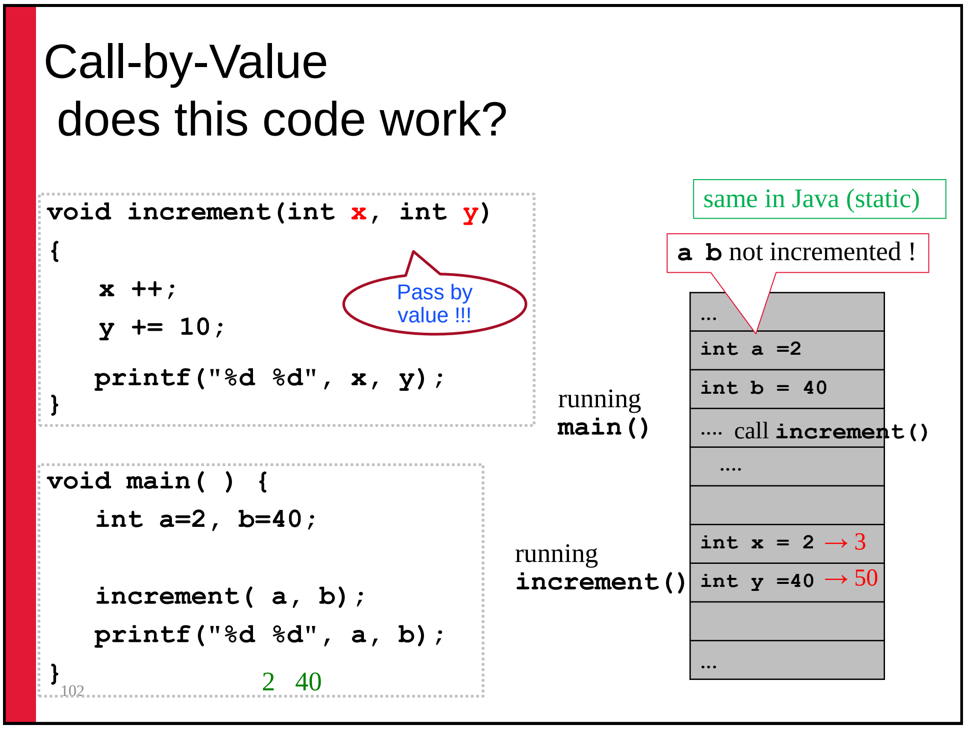
```
...
int a =2
int b = 40
.... call increment()
    ....

int x = a= 2
int y = b=40

...
```

copy

copy

Same in Java (static)

101



Call-by-Value
 does this code work?

```
void increment(int x, int y)
{
    x ++;
    y += 10;

    printf("%d %d", x, y);
}
```

Pass by value !!!

```
void main( ) {
    int a=2, b=40;

    increment( a, b);
    printf("%d %d", a, b);
}
```

102

2   40

same in Java (static)

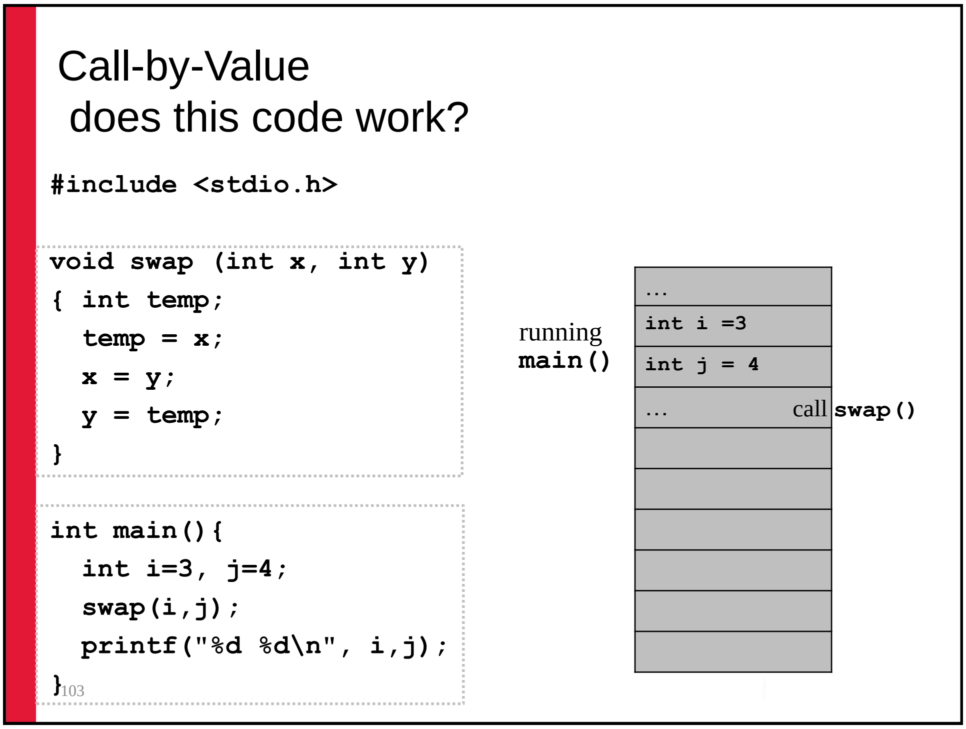
a b not incremented !

running
main()

running
increment()

```
...
int a =2
int b = 40
.... call increment()
    ....

int x = 2 → 3
int y =40 → 50

...
```

102



Call-by-Value
does this code work?

```
#include <stdio.h>

void swap (int x, int y)
{ int temp;
  temp = x;
  x = y;
  y = temp;
}

int main(){
  int i=3, j=4;
  swap(i,j);
  printf("%d %d\n", i,j);
}
```

103

running
main()

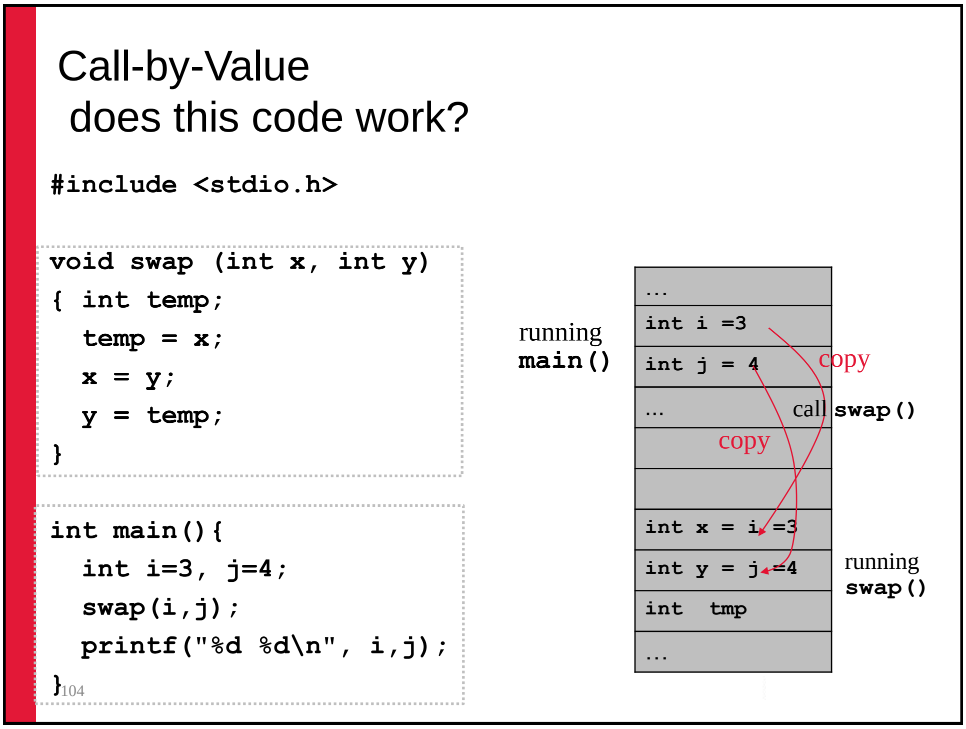
…

int i =3

int j = 4

… call swap()

103



Call-by-Value
does this code work?

```
#include <stdio.h>

void swap (int x, int y)
{ int temp;
  temp = x;
  x = y;
  y = temp;
}

int main(){
  int i=3, j=4;
  swap(i,j);
  printf("%d %d\n", i,j);
}
```

104

running
main()

…

int i =3

int j = 4

… call swap()

copy

copy

int x = i =3

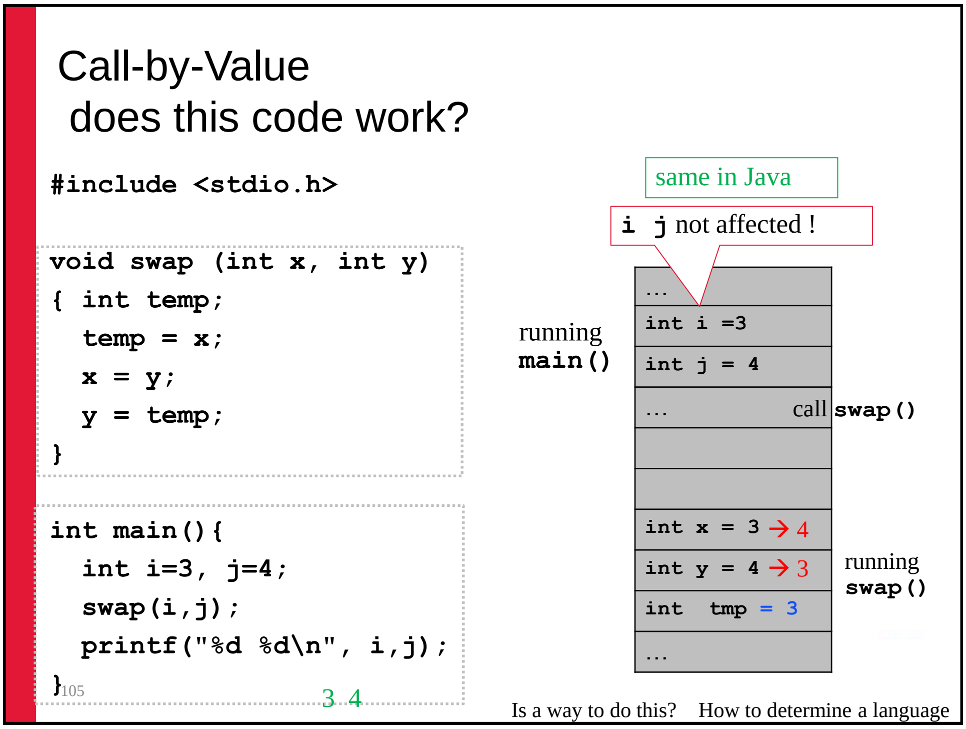
int y = j =4

int tmp

…

running
swap()

104



Call-by-Value
does this code work?

```
#include <stdio.h>

void swap (int x, int y)
{ int temp;
  temp = x;
  x = y;
  y = temp;
}

int main(){
  int i=3, j=4;
  swap(i,j);
  printf("%d %d\n", i,j);
}
```

3 4

same in Java

i j not affected !

running
main()

```
…
int i =3
int j = 4
…
```

call swap()

```
int x = 3 → 4
int y = 4 → 3
int  tmp = 3
…
```

running
swap()

Is a way to do this?   How to determine a language