YORK U UNIVERSITÉ UNIVERSITY

EECS2031 A Software Tools

SU 2019

May 27, 2019 Lecture 4

1

Summary of last two lectures

- 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' avoid x>=48 && x<=57
      - ✓ int: 122, 0122, 0x12F convert between Decimal, Bin, Oct, Hex
    - Arrays (one dimension) and strings (Ch1.6,1.9)
      - ✓ "hello" has size 6 byte

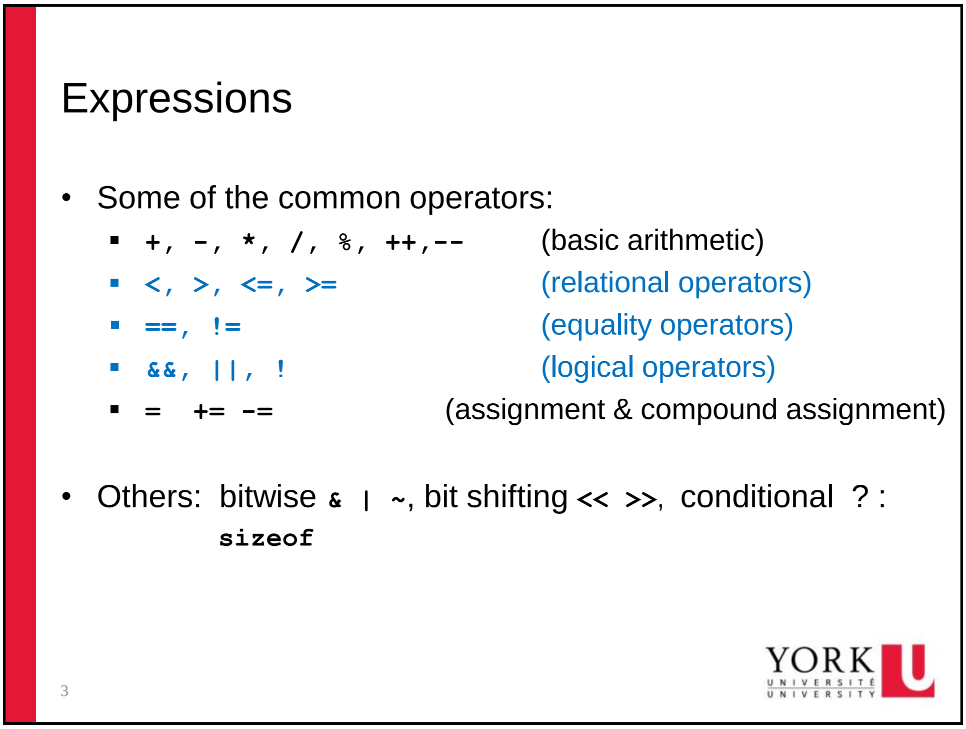
H	e	l	l	o	\0

  - Expressions
    - Basic operators (arithmetic, relational and logical)
      - ✓ y=x++; y=++x;
      - ✓ !0 !-3 if (x = 2)
    - Type conversion and promotion 9/2*2.0 2.0*9/2 int i= 3.4
    - Other operators (bitwise, bit shifting , compound assignment, conditional)
      - ✓ Bit: |, &, ~. ^, << >>
      - ✓ Compound: x += 10; x >>= 10; x += y + 3
    - Precedence of operators flag | 1 << 3
- Functions and Program Structure (Chapter 4)

Week 2

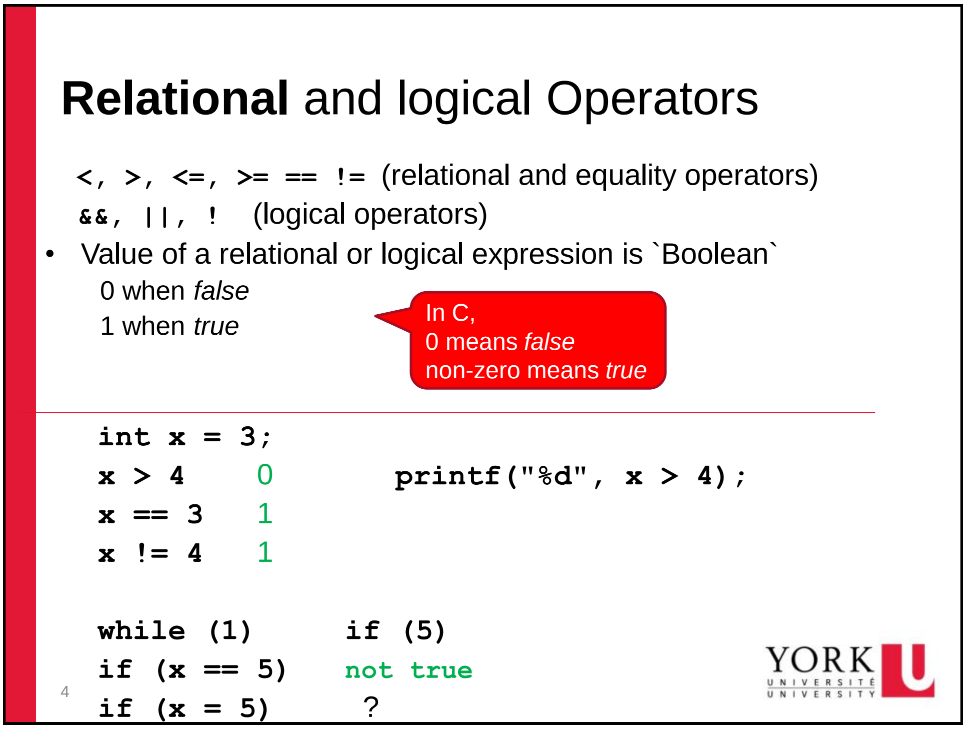
Previous lecture

2



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`
  0 when *false*
  1 when *true*

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

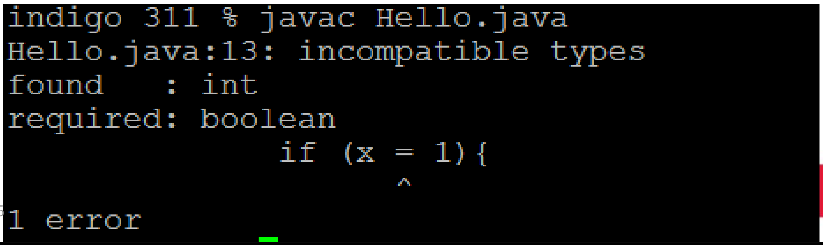
```
int x = 3;
x > 4      0        printf("%d", x > 4);
x == 3     1
x != 4     1

while (1)       if (5)
if (x == 5)     not true
if (x = 5)       ?
```

Relational and logical Operators

- Not as safe as Java -- probably why Java introduce Boolean

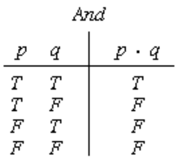
```
int x = 2;
if  (x = 1)
  ......
else if  (x=2)…
  ......
```

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



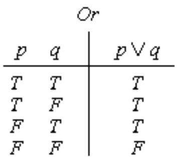
```
indigo 311 % javac Hello.java
Hello.java:13: incompatible types
found    : int
required: boolean
              if (x = 1){
                    ^
1 error
```

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



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

- ! Logical negation

  !0 returns 1, **!**(any non-zero value) returns 0 **!-4** **0**

Not valid in Java

- || logical OR, && logical AND
  - **&&** returns 1 if both non-zero. Otherwise 0 **3 && -2** **1**
  - || returns 1 if either non-zero. Otherwise 0 **3 || !5** **1**

```
if (!0) ......         true

if (!-4)   ......      false

if (3 && -2) ......    true
```

Not valid in Java

Outline

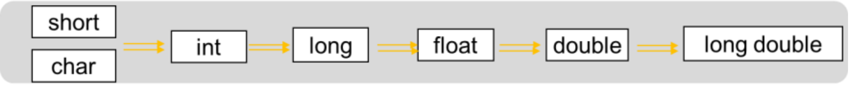
- Types and sizes
  - Types
  - Constant values (literals)
    - char
    - int
    - float

- 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

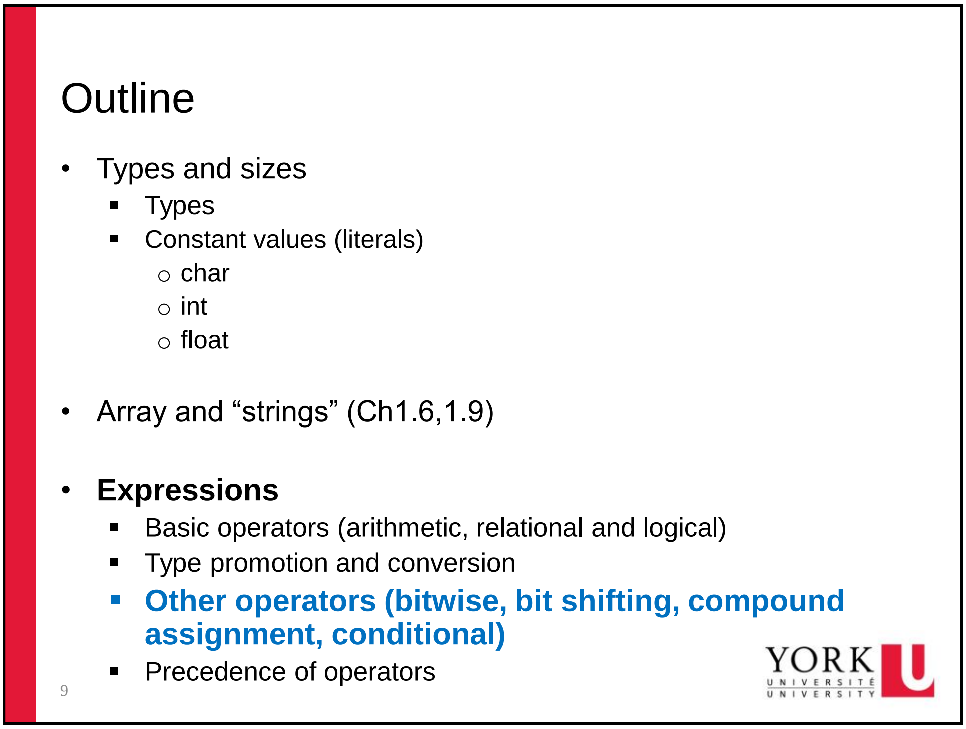
1. Given an expression with operands of mixed types, C converts (promotes) the types of values to do calculations



2. Conversion may happen on assignment

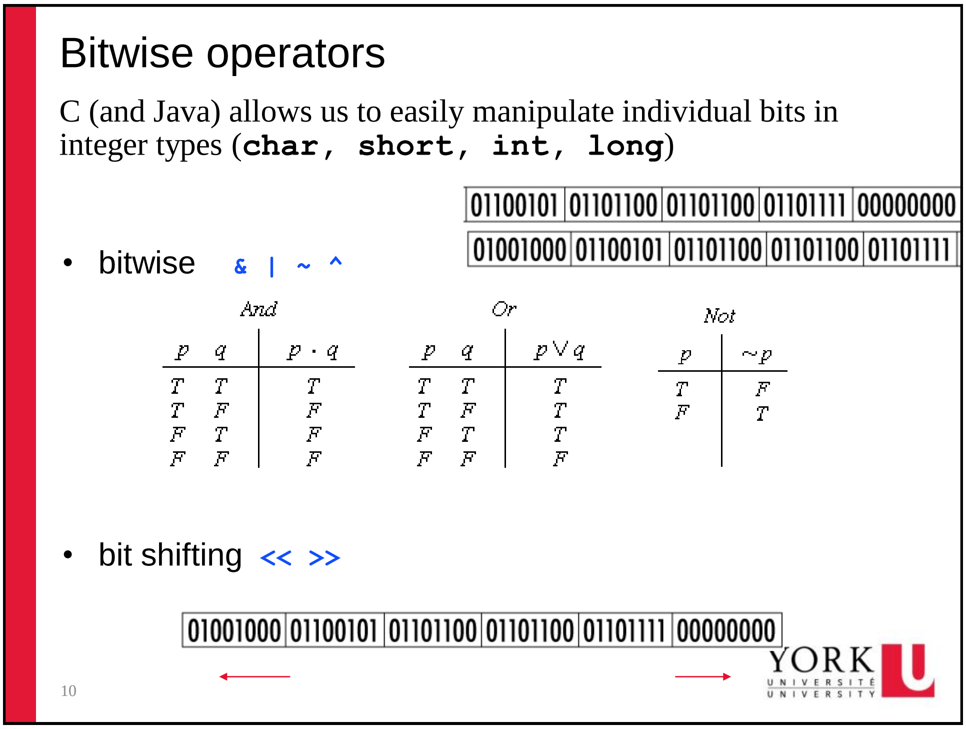
```
float f = 3;      int i = 3.8;
```

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



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



Bitwise operators

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
```

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"

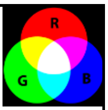
Lhs	0	0	1	1
Rhs	0	1	0	1
Result	0	0	0	1

Turn off Lhs Keep Lhs

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

Some examples

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



- In Java, getRGB() packS 3 +1 values into a 32 bit (4 bytes) int

00001010 11111101 01001000 11111010

^ Alpha ^Red ^Green ^Blue

&

00000000 00000000 00000000 11111111

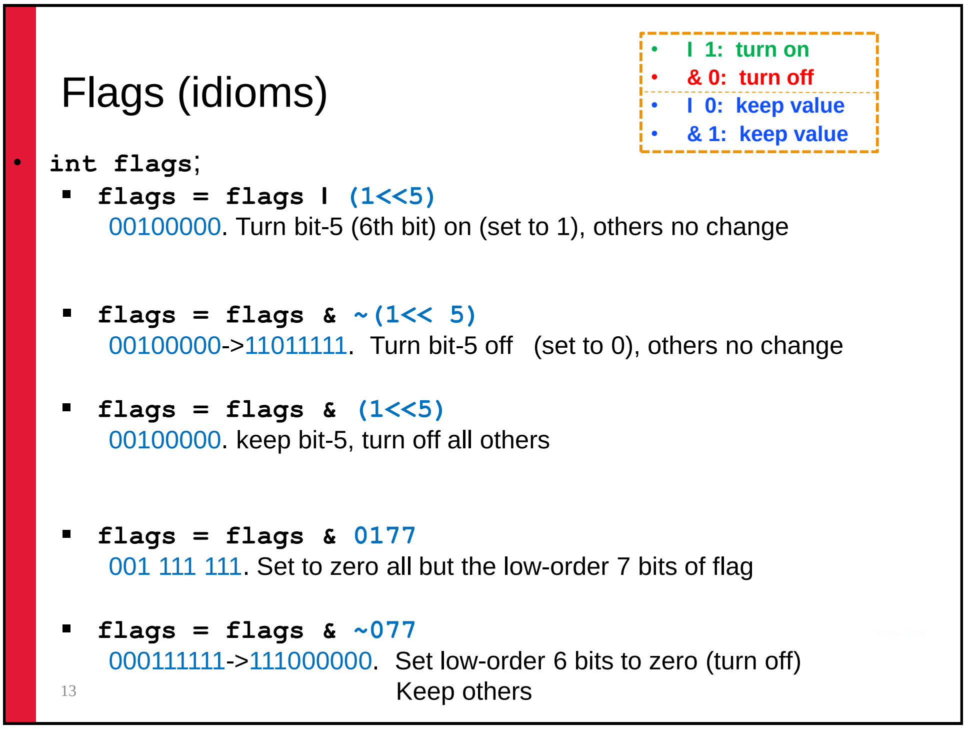
0xFF 255 0377

Turn off ⇩ keep value

00000000 00000000 00000000 11111010

- `int blue = (rgb_pack      ) & 0377;` // rgb_pack not changed
- `int green= (rgb_pack >>8 ) & 0xFF;` /* shift and mask */
- `int red =  (rgb_pack >>16) & 255;`

- Mask and then shift? Shift only? /* be careful. >> on unsigned! */

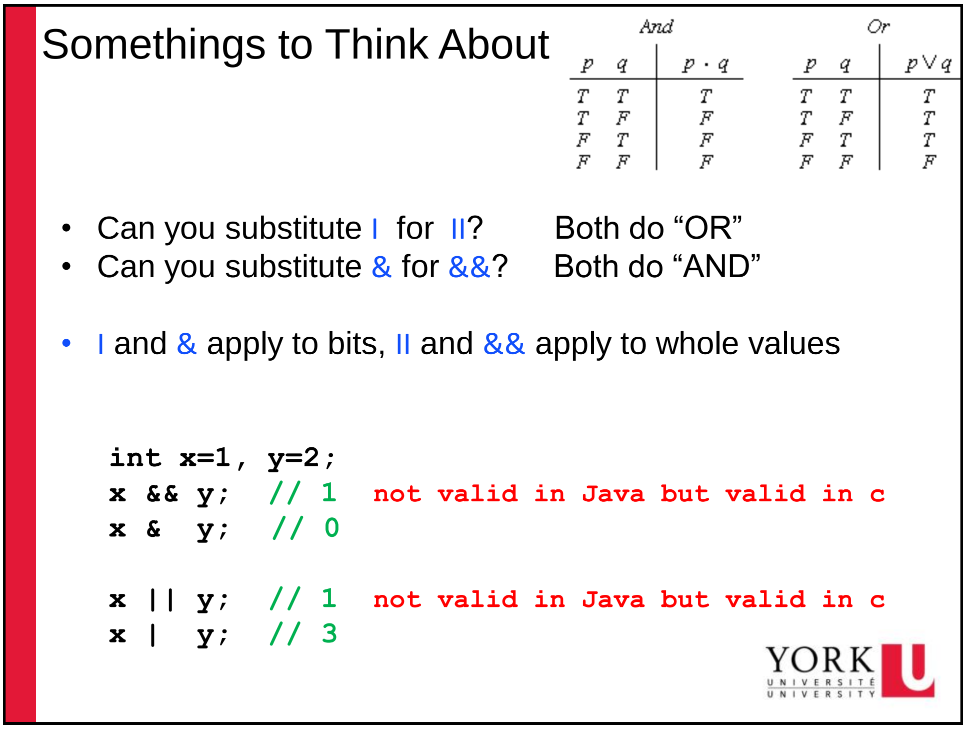


Flags (idioms)

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

- `int flags;`
  - `flags = flags | (1<<5)`
    00100000. Turn bit-5 (6th bit) on (set to 1), others no change
  - `flags = flags & ~(1<< 5)`
    00100000->11011111. Turn bit-5 off (set to 0), others no change
  - `flags = flags & (1<<5)`
    00100000. keep bit-5, turn off all others
  - `flags = flags & 0177`
    001 111 111. Set to zero all but the low-order 7 bits of flag
  - `flags = flags & ~077`
    000111111->111000000. Set low-order 6 bits to zero (turn off)
    Keep others

13



Somethings to Think About

And

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

Or

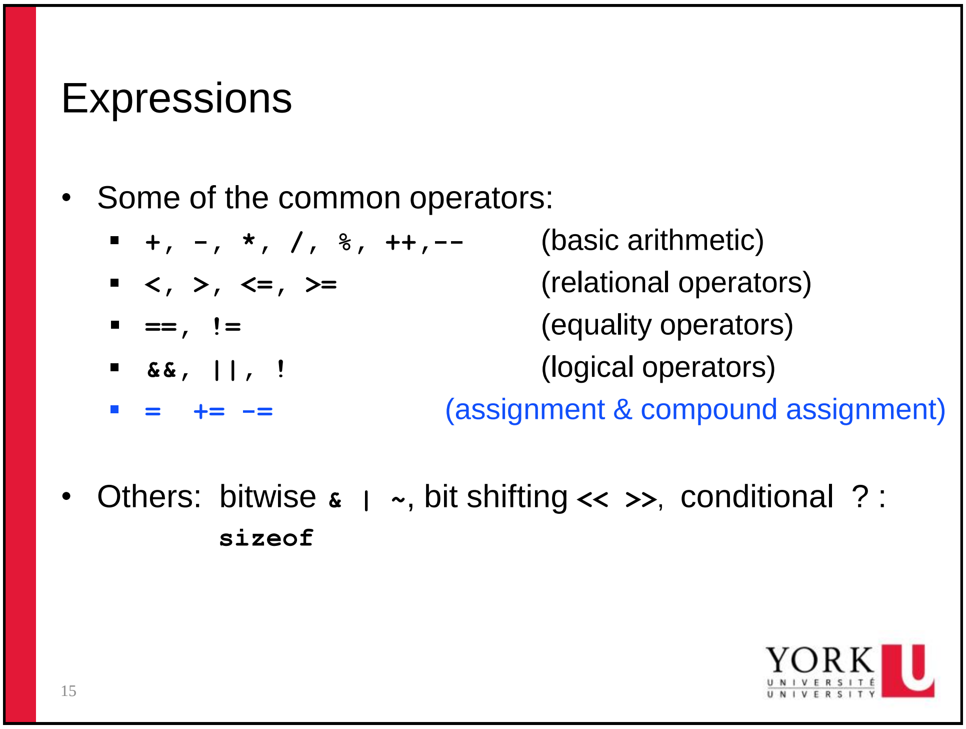
p	*q*	*p ∨ q*
T	*T*	*T*
T	*F*	*T*
F	*T*	*T*
F	*F*	*F*

- Can you substitute | for ||? Both do "OR"
- Can you substitute & for &&? Both do "AND"

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

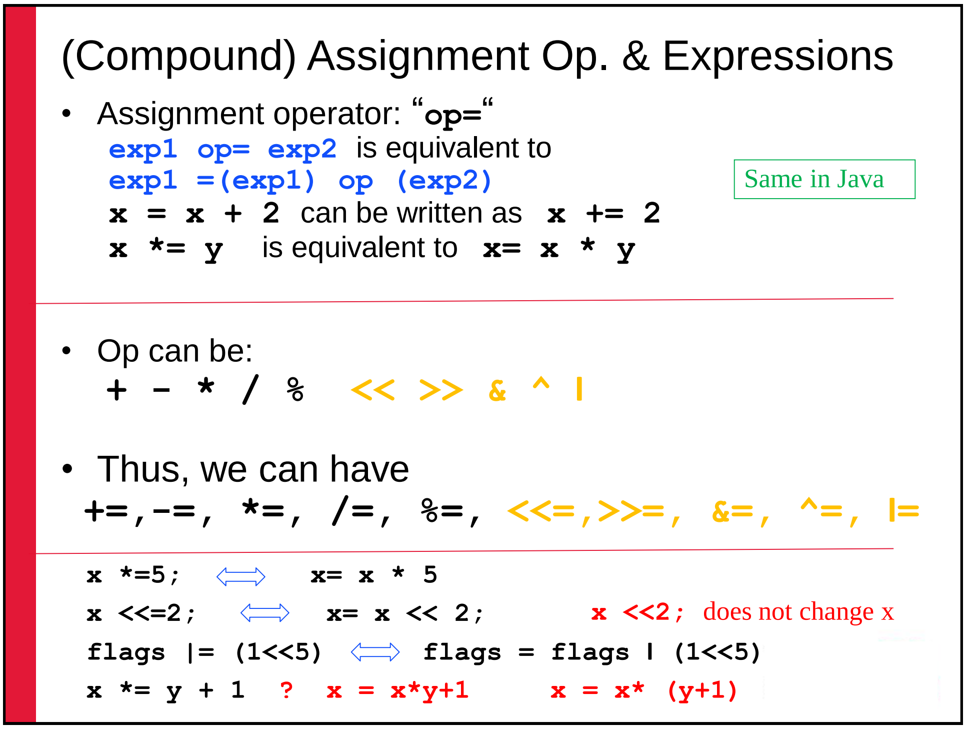
```
int x=1, y=2;
x && y;  // 1   not valid in Java but valid in c
x &  y;  // 0

x || y;  // 1   not valid in Java but valid in c
x |  y;  // 3
```

YORK U UNIVERSITÉ UNIVERSITY



Expressions

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



(Compound) Assignment Op. & Expressions

- Assignment operator: "**op=**"
  `exp1 op= exp2` is equivalent to
  `exp1 =(exp1) op (exp2)`
  `x = x + 2` can be written as `x += 2`
  `x *= y` is equivalent to `x= x * y`

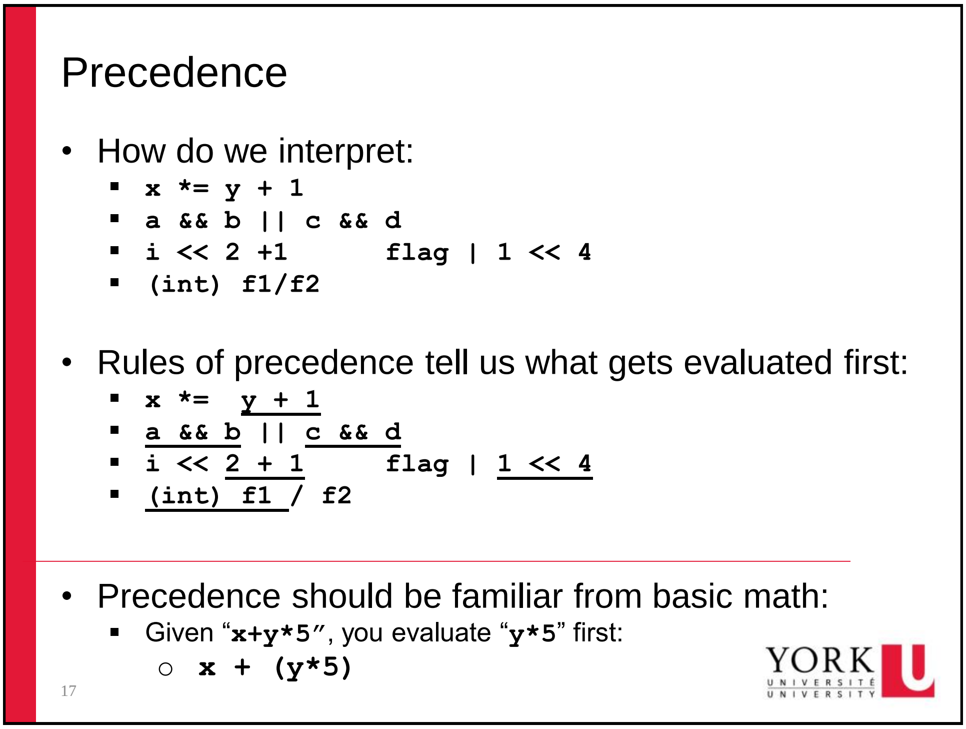
Same in Java

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

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

```
x *=5;          ⟺     x= x * 5
x <<=2;          ⟺     x= x << 2;       x <<2;
flags |= (1<<5)  ⟺  flags = flags | (1<<5)
x *= y + 1  ?  x = x*y+1     x = x* (y+1)
```

does not change x



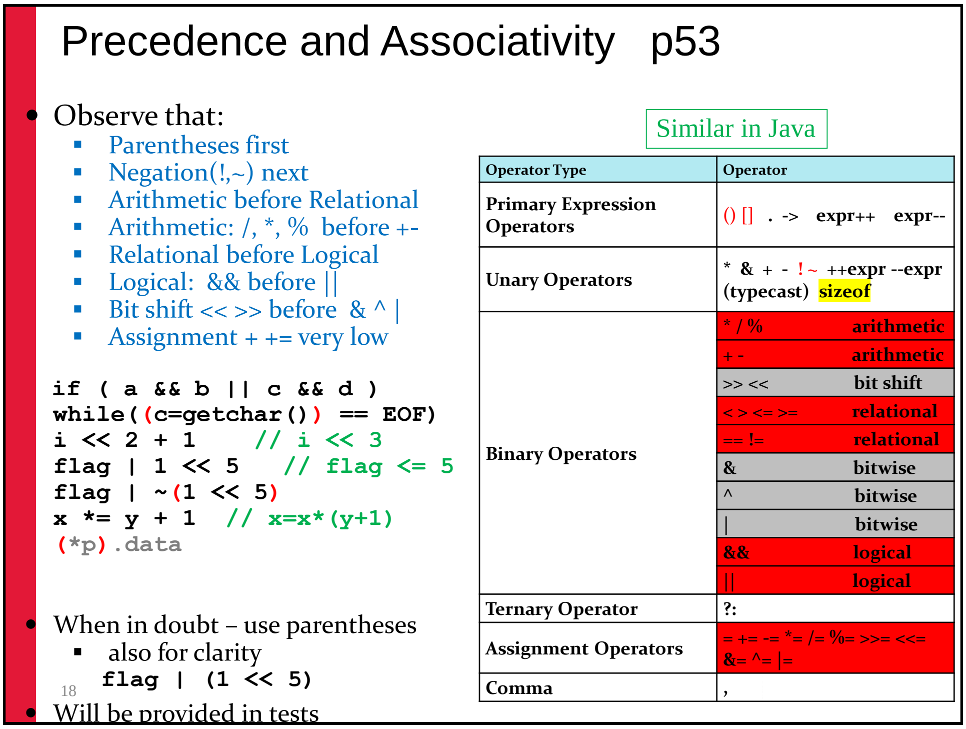
Precedence

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

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

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

YORK U UNIVERSITÉ UNIVERSITY



Precedence and Associativity   p53

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

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

- When in doubt – use parentheses
  - also for clarity

```
flag | (1 << 5)
```

- Will be provided in tests

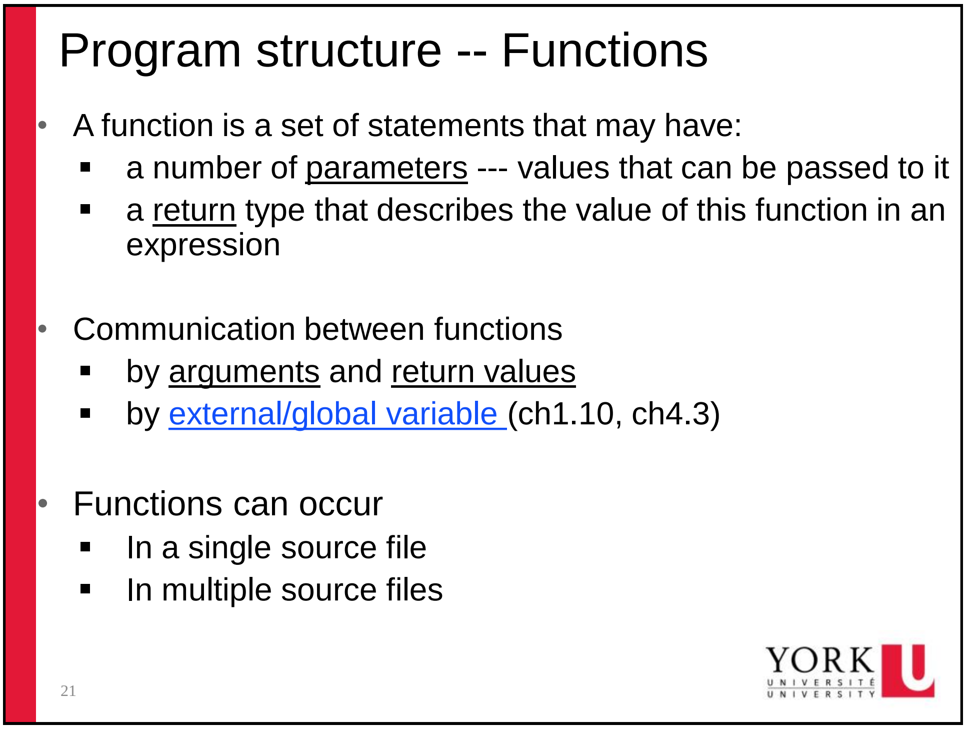
Similar in Java

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	,

COSC2031 - Software Tools

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

- C program structure
  - Functions
- Categories, scope and life time of variables (and functions)
- C Preprocessing

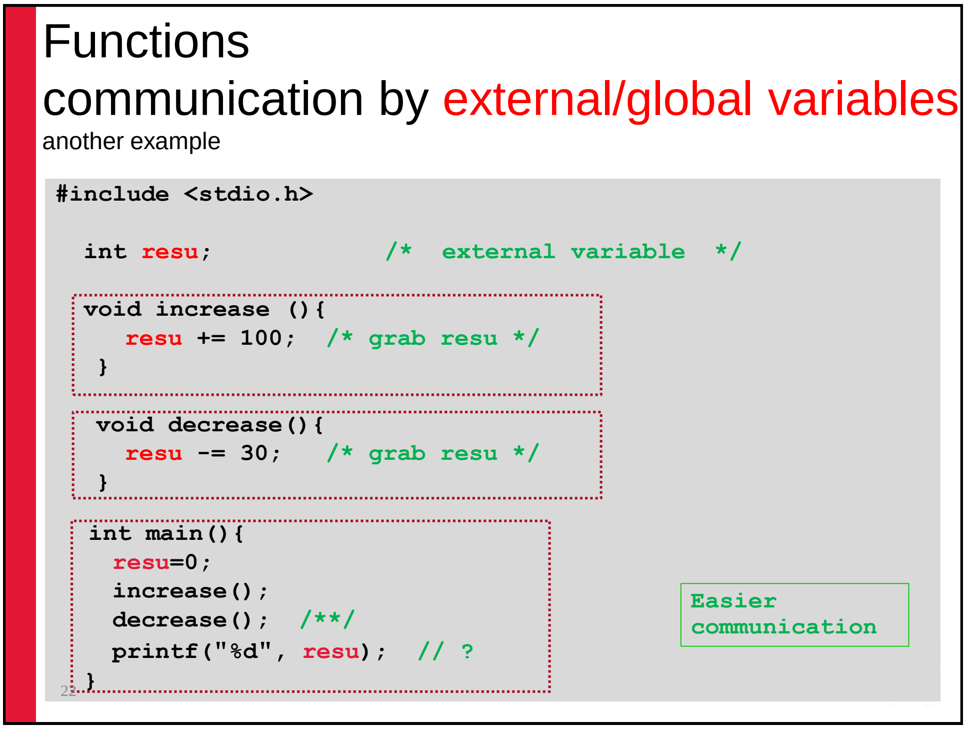


Program structure -- Functions

- 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/global variable (ch1.10, ch4.3)
- Functions can occur
  - In a single source file
  - In multiple source files

21

21



Functions communication by external/global variables

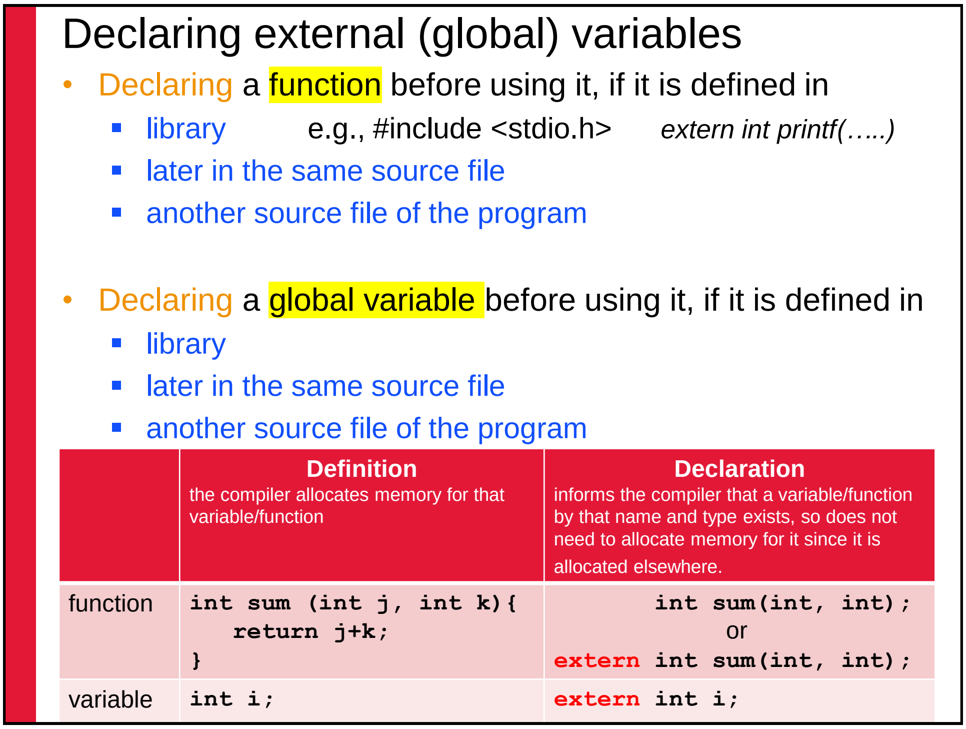
another example

```
#include <stdio.h>

  int resu;              /*  external variable  */

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

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

   int main(){
     resu=0;
     increase();
     decrease();  /**/
     printf("%d", resu);  // ?
   }
```

Easier communication

22

22



Declaring external (global) variables

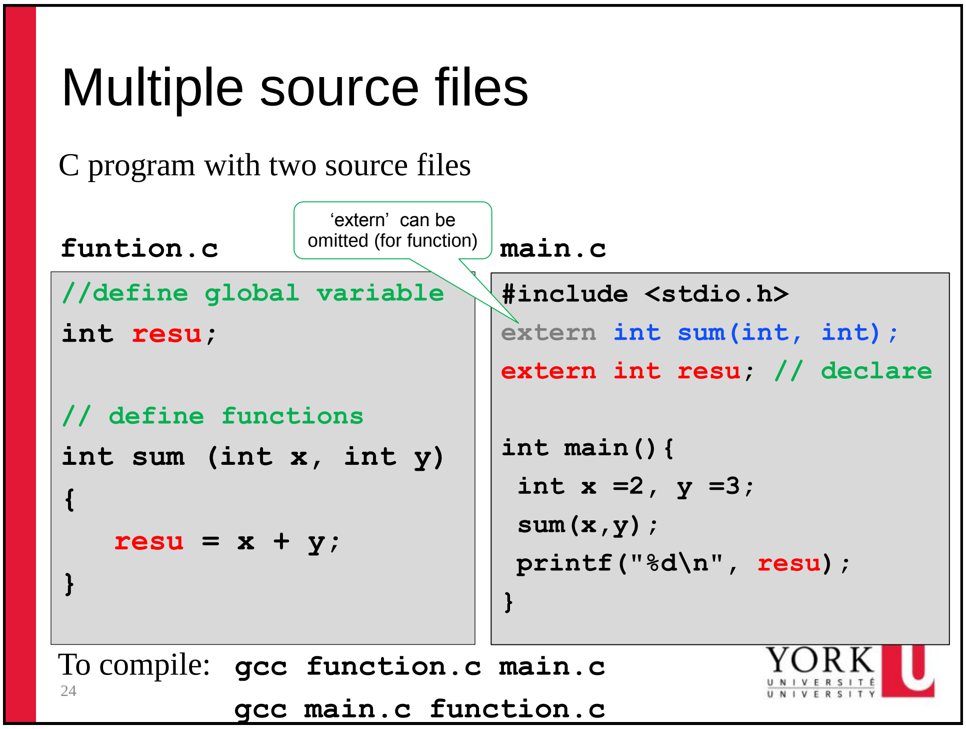
- Declaring a **function** before using it, if it is defined in
  - library e.g., #include <stdio.h> *extern int printf(…..)*
  - later in the same source file
  - another source file of the program

- 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 is 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;`



Multiple source files

C program with two source files

funtion.c

```
//define global variable
int resu;

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

‘extern’ can be omitted (for function)

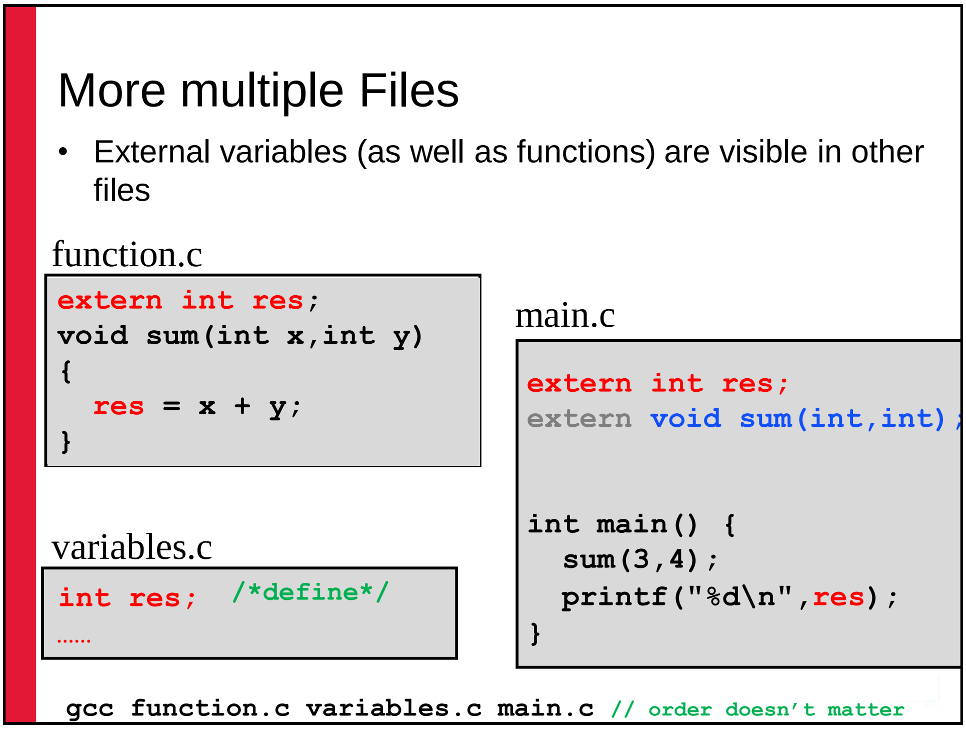
main.c

```
#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 function.c main.c`

`gcc main.c function.c`

24



More multiple Files

- External variables (as well as functions) are visible in other files

function.c

```
extern int res;
void sum(int x,int y)
{
  res = x + y;
}
```

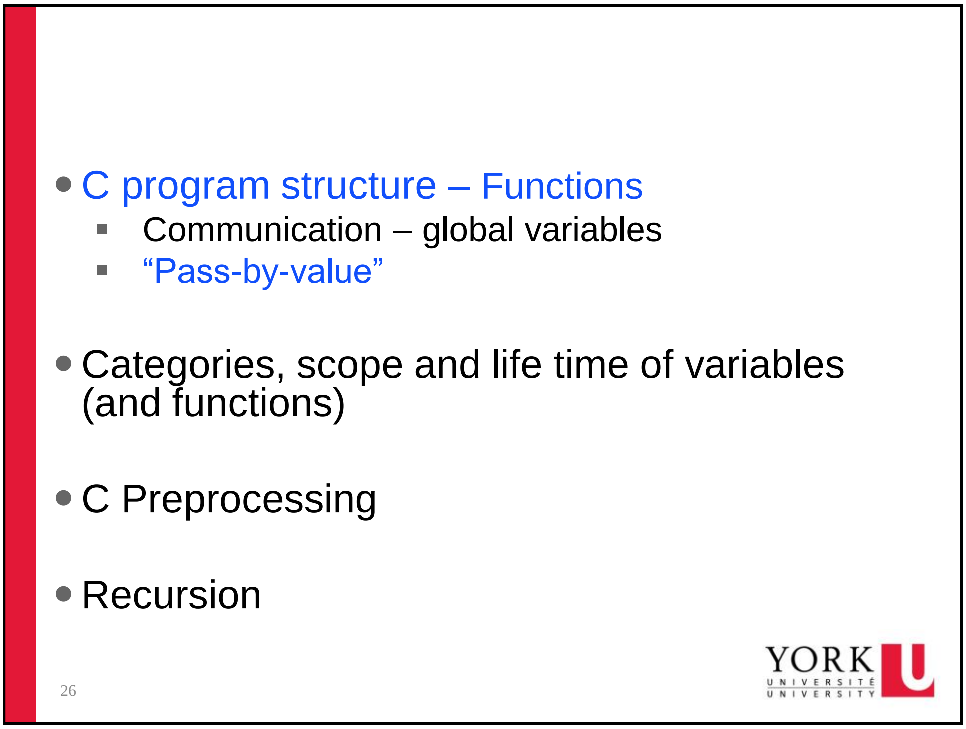
variables.c

```
int res;   /*define*/
……
```

main.c

```
extern int res;
extern void sum(int,int);


int main() {
  sum(3,4);
  printf("%d\n",res);
}
```

```
gcc function.c variables.c main.c // order doesn't matter
```



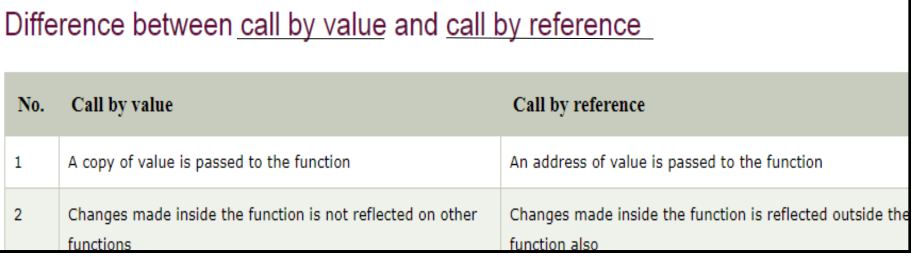
- C program structure – Functions
  - Communication – global variables
  - “Pass-by-value”
- Categories, scope and life time of variables (and functions)
- C Preprocessing
- Recursion

Call (pass)-by-Value vs 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 and 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



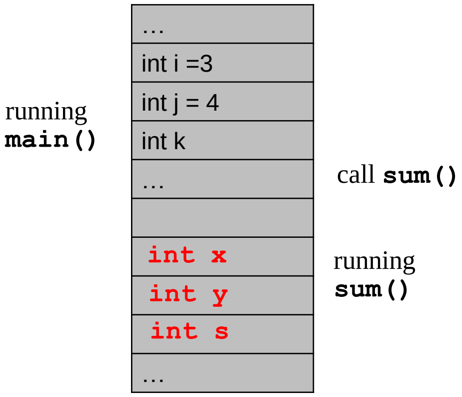
Difference between call by value and call by reference

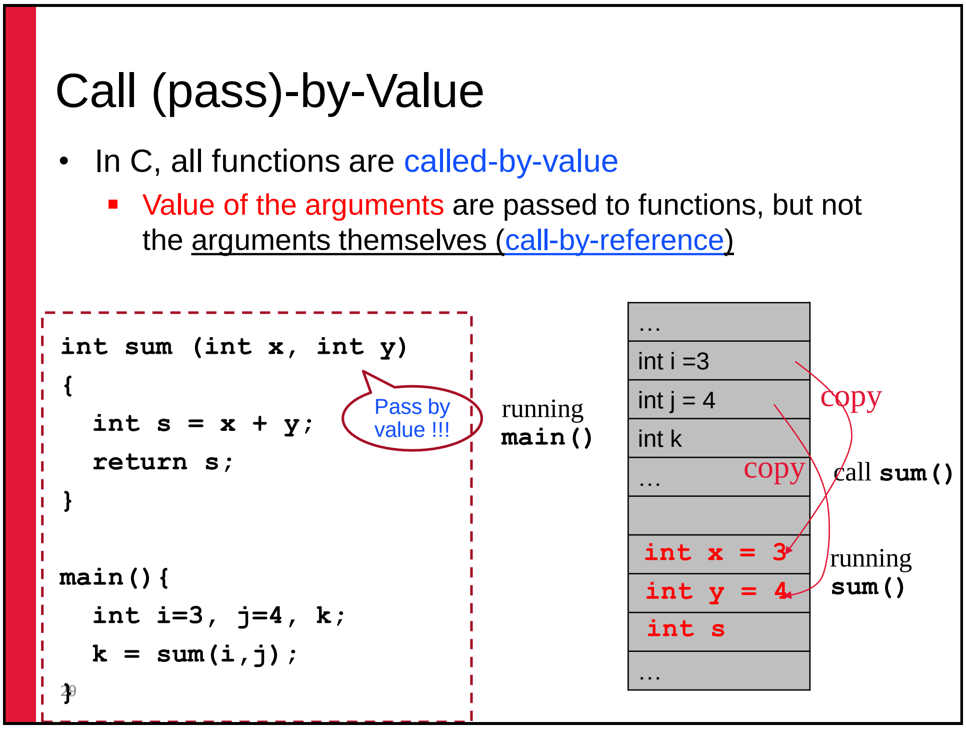
No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also

Call (pass)-by-Value

- In C, all functions are called-by-value
  - Value 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);
}
```





Call (pass)-by-Value

- In C, all functions are called-by-value
  - Value 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);
}
```

Pass by value !!!

running **main()**

…

int i =3

int j = 4

int k

…

int x = 3

int y = 4

int s

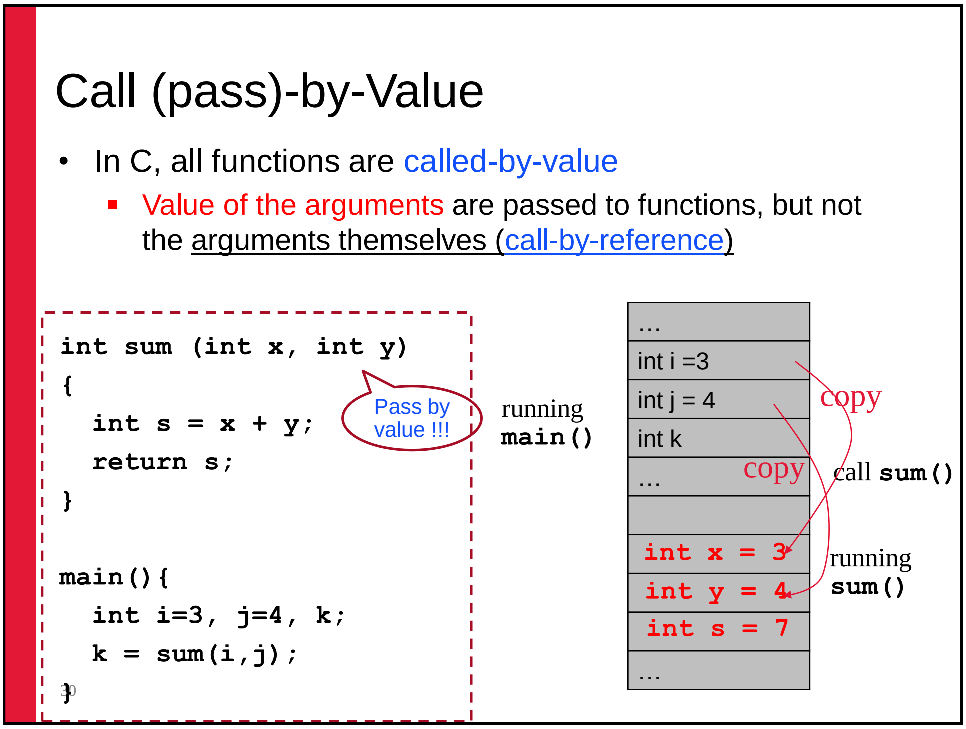
…

copy

copy

call **sum()**

running **sum()**



Call (pass)-by-Value

- In C, all functions are called-by-value
  - Value 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);
}
```

Pass by value !!!

running **main()**

…

int i =3

int j = 4

int k

…

int x = 3

int y = 4

int s = 7

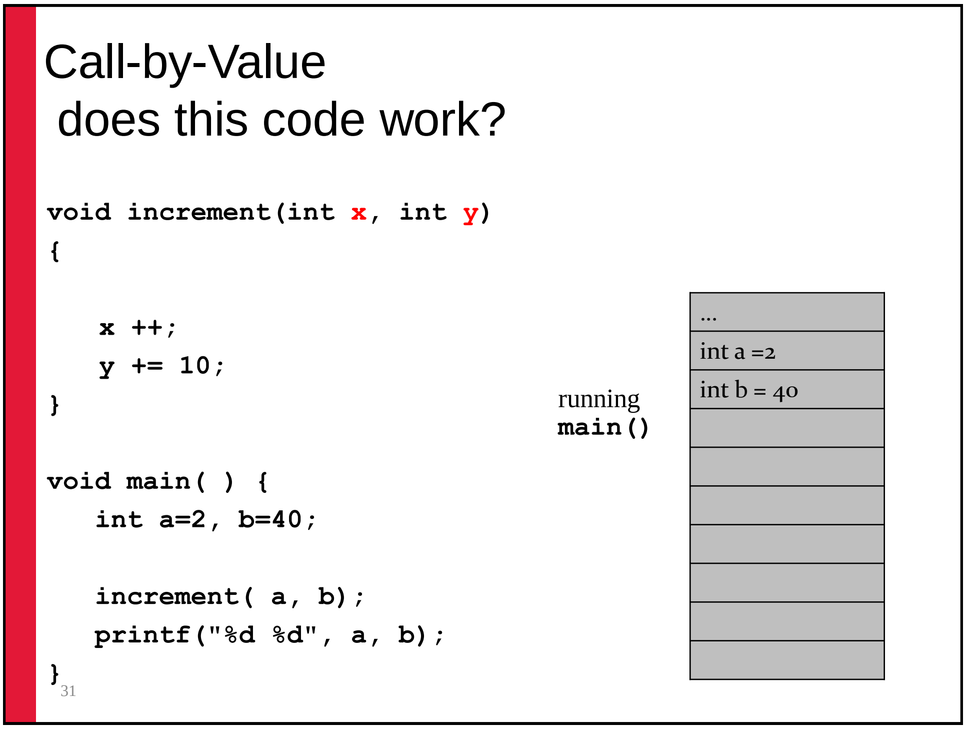
…

copy

copy

call **sum()**

running **sum()**



Call-by-Value
does this code work?

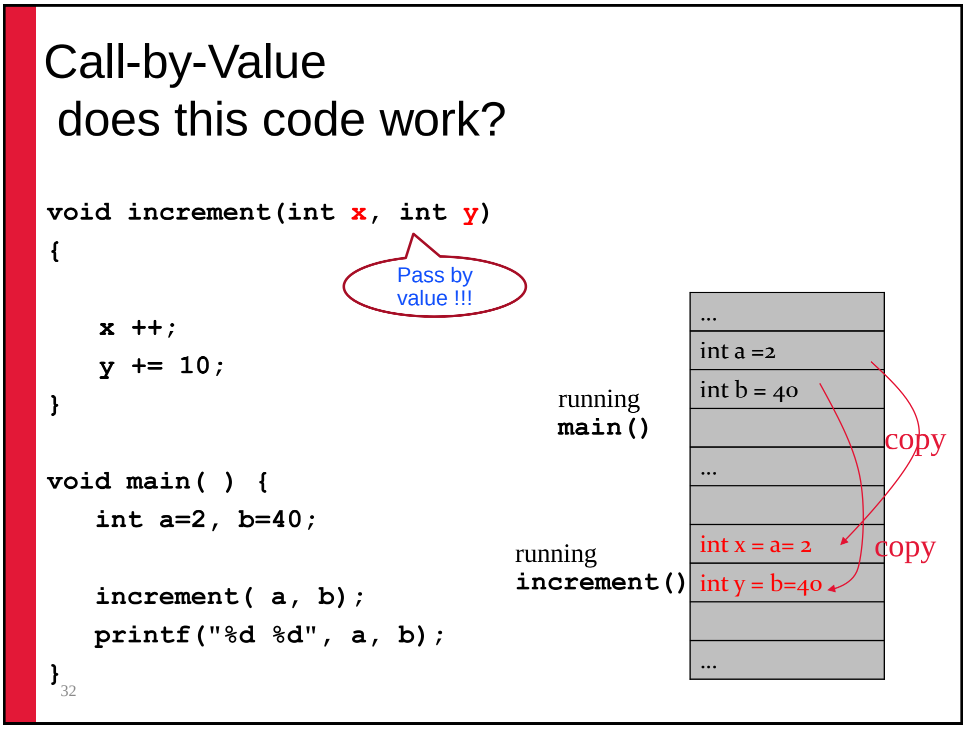
```
void increment(int x, int y)
{

    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

31

31



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);
}
```

Pass by value !!!

running
main()

running
increment()

...
int a =2
int b = 40

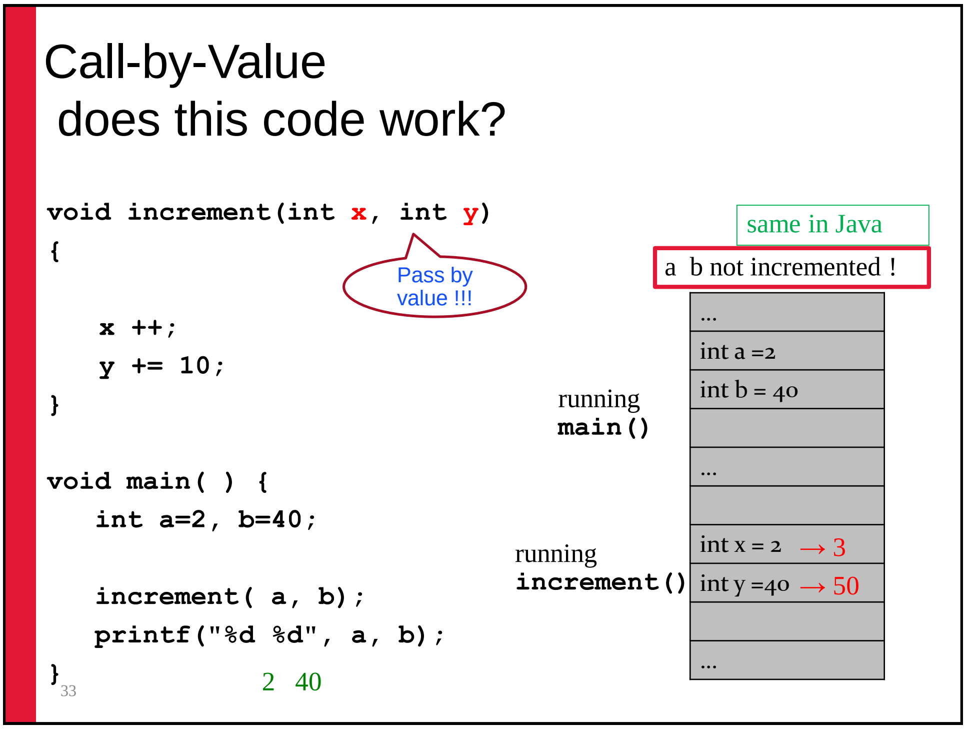
...

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

...

32

32



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);
}
```

Pass by value !!!

2 40

same in Java

a b not incremented !

running **main()**

running **increment()**

...

int a =2

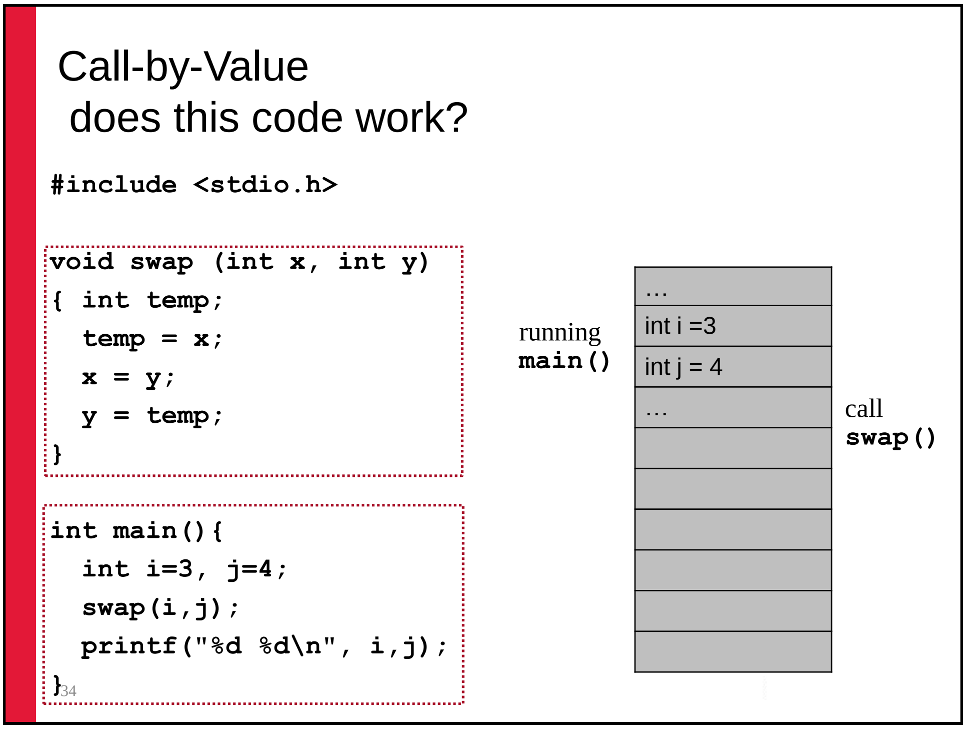
int b = 40

...

int x = 2 → 3

int y =40 → 50

...



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);
}
```

running **main()**

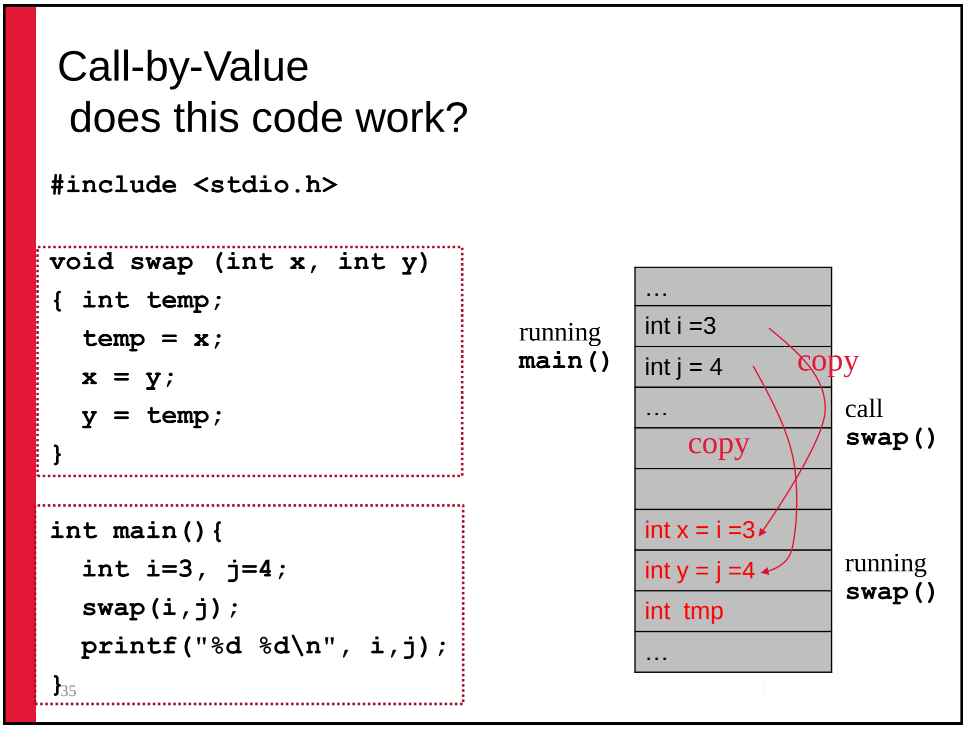
…

int i =3

int j = 4

…

call **swap()**



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);
}
```

running
main()

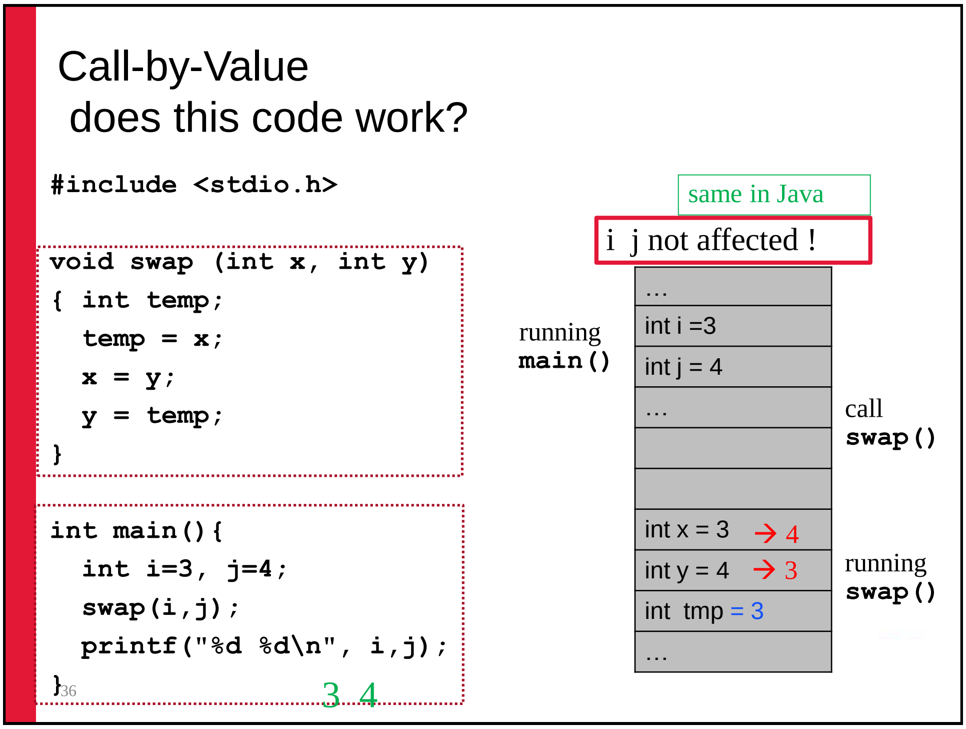
…
int i =3
int j = 4
…
copy

int x = i =3
int y = j =4
int tmp
…

copy

call
swap()

running
swap()



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 !

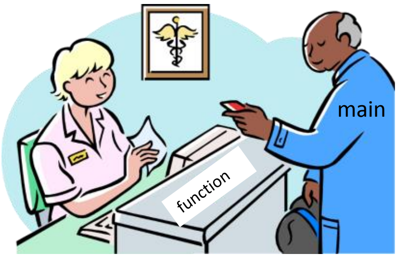
running
main()

…
int i =3
int j = 4
…

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

call
swap()

running
swap()



Mr. Main

Hi, function, I have some manuscripts, stored in lockers (memory), and I want you to repaint them so their color changed.

Ms function

Hi, Mr Main, here is how we function work: We don't take your original manuscript over the counter (not pass by reference). We always photocopy things passed here (pass by value) , and work on copies.

Mr. Main: Then, is there a way to have my original manuscripts' color changed by you?

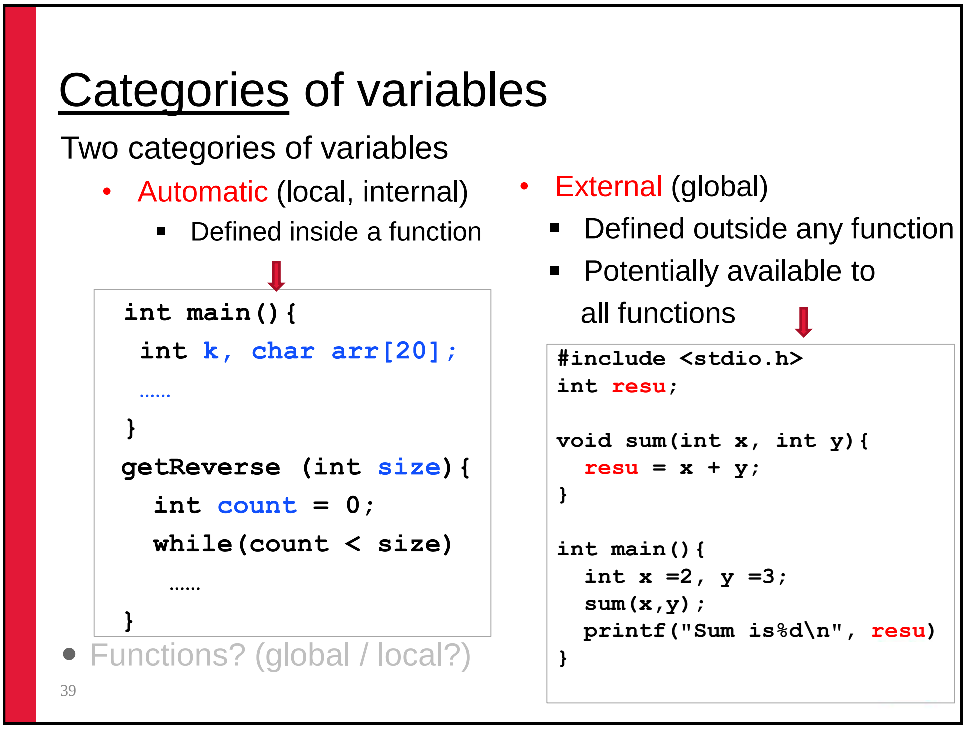
Mr. Main

Ms function: Don't worry. Of course there is a way. Give us your locker address (pointer) instead ....

Stopped here last time

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

today



Categories of variables

Two categories of variables

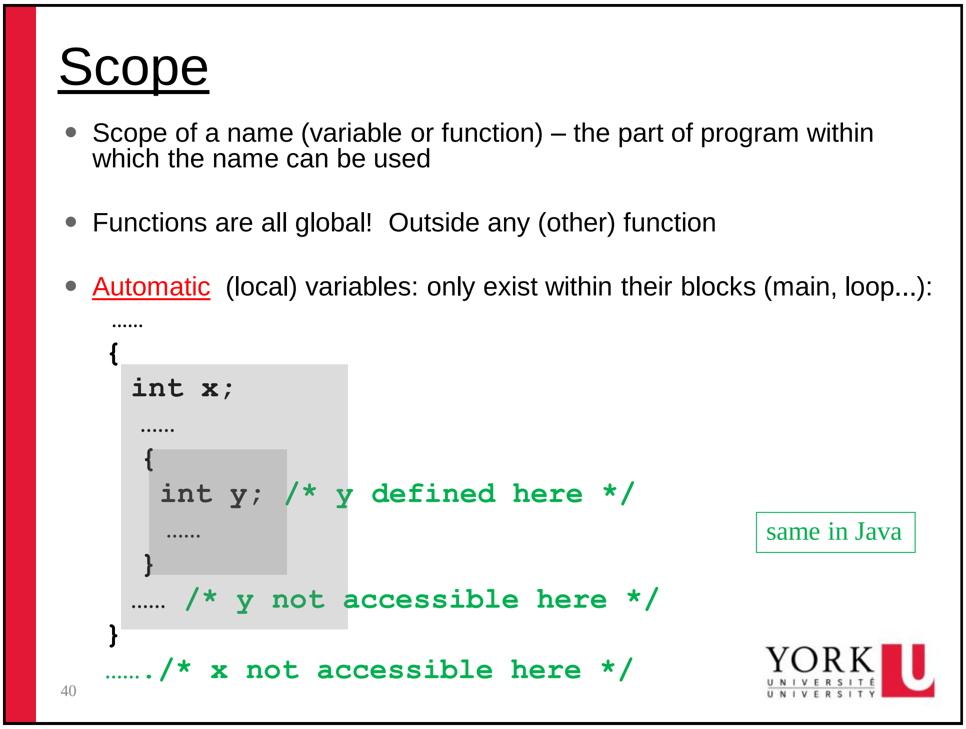
- Automatic (local, internal)
  - Defined inside a function

```
int main(){
 int k, char arr[20];
 ......
}
getReverse (int size){
  int count = 0;
  while(count < size)
   ......
}
```

- External (global)
  - Defined outside any function
  - Potentially available to all functions

```
#include <stdio.h>
int resu;

void sum(int x, int y){
  resu = x + y;
}

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

- Functions? (global / local?)

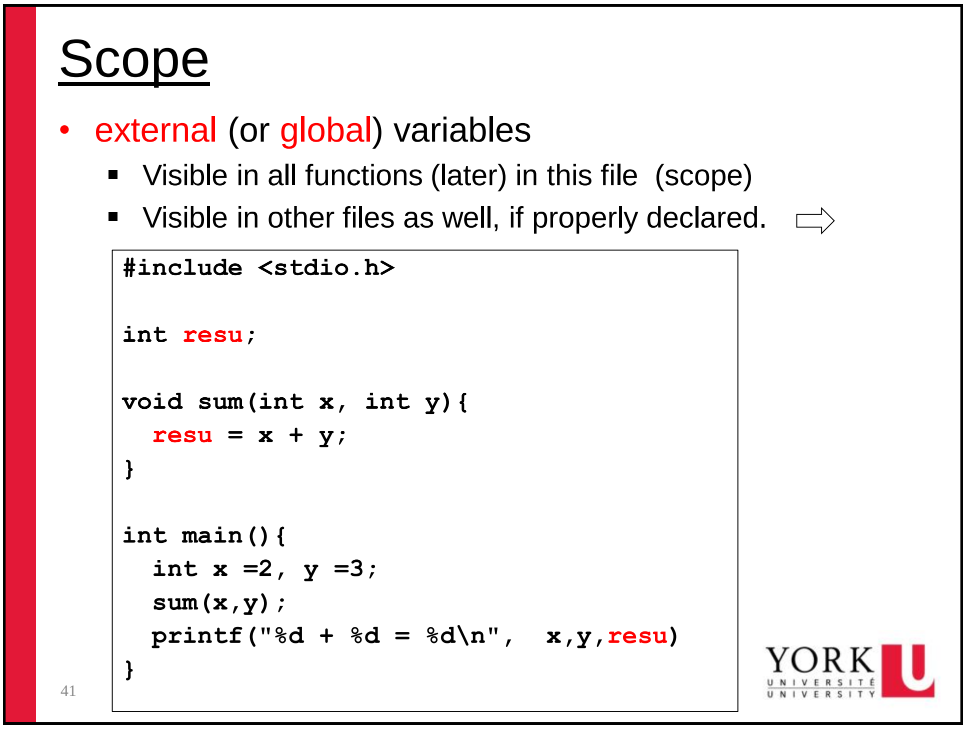


Scope

- Scope of a name (variable or function) – the part of program within which the name can be used
- Functions are all global! Outside any (other) function
- Automatic (local) variables: only exist within their blocks (main, loop...):

```
 ......
{
  int x;
  ......
  {
   int y; /* y defined here */
   ......
  }
  ......  /* y not accessible here */
}
......./* x not accessible here */
```

same in Java



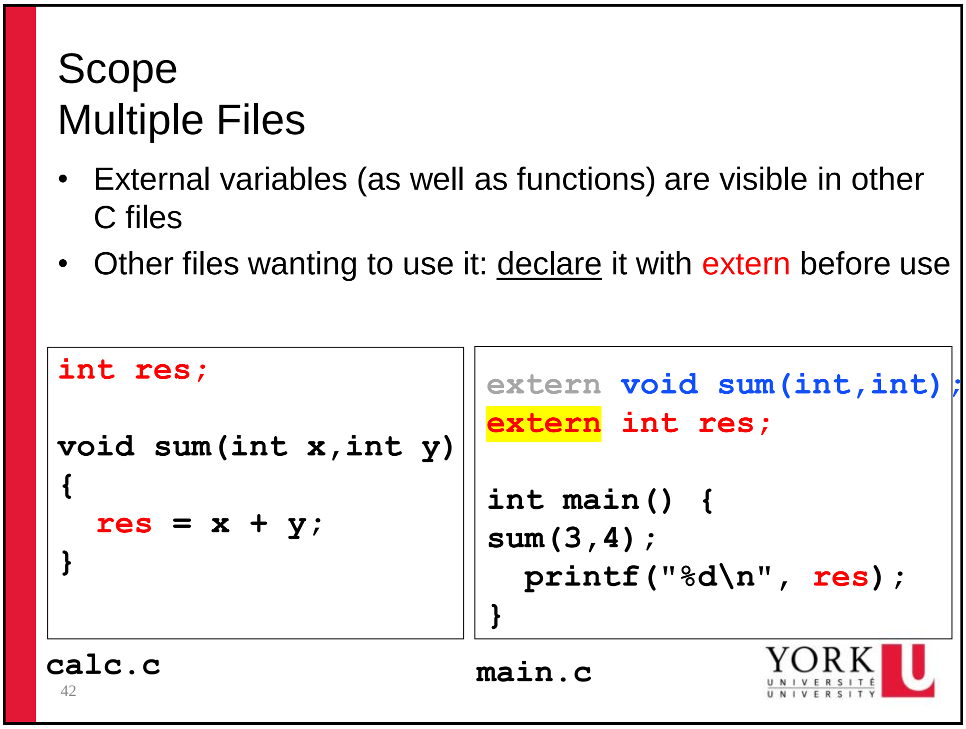
Scope

- external (or global) variables
  - Visible in all functions (later) in this file (scope)
  - Visible in other files as well, if properly declared. ⇨

```
#include <stdio.h>

int resu;

void sum(int x, int y){
  resu = x + y;
}

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

41

41



Scope
Multiple Files

- External variables (as well as functions) are visible in other C files
- Other files wanting to use it: declare it with extern before use

```
int res;

void sum(int x,int y)
{
  res = x + y;
}
```

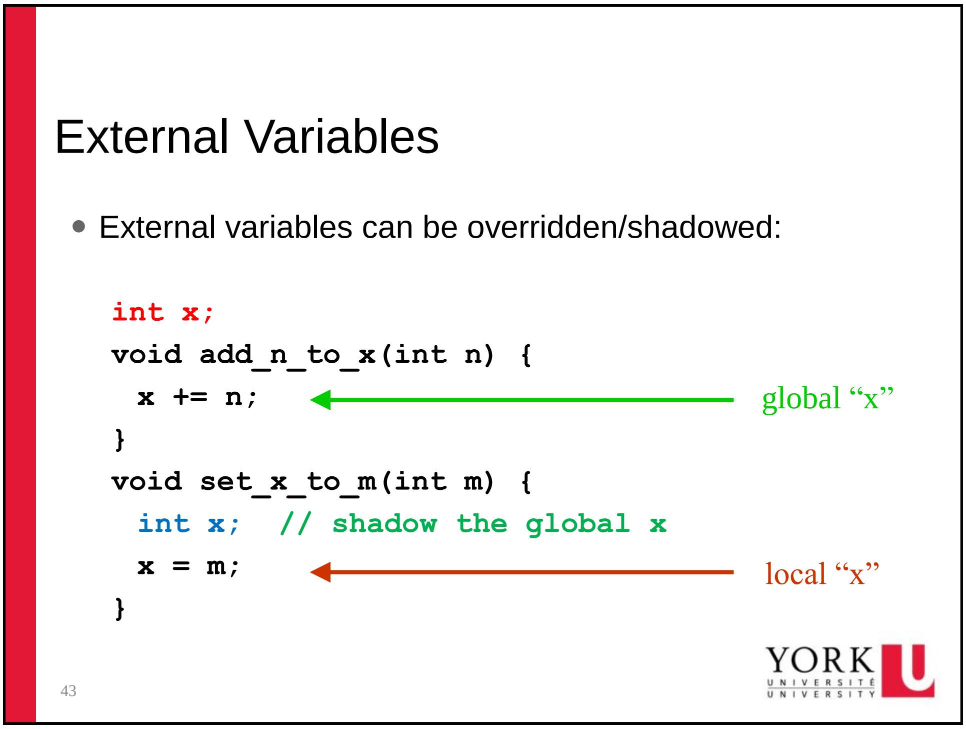
calc.c

```
extern void sum(int,int);
extern int res;

int main() {
sum(3,4);
  printf("%d\n", res);
}
```

main.c

42

42



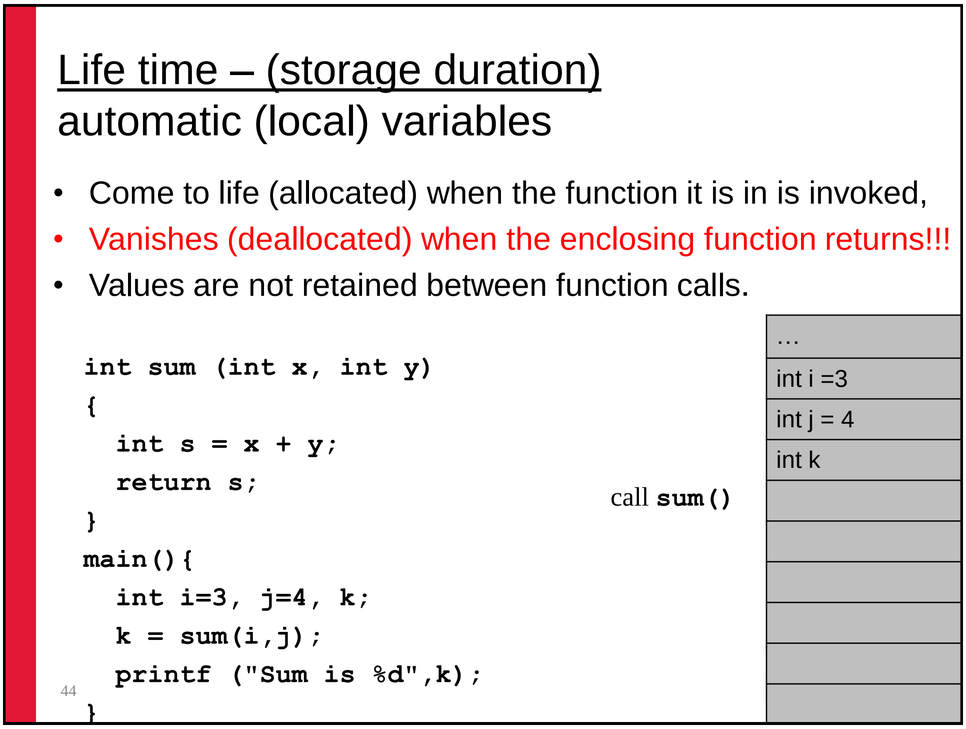
External Variables

- External variables can be overridden/shadowed:

```
int x;
void add_n_to_x(int n) {
  x += n;
}
void set_x_to_m(int m) {
  int x;  // shadow the global x
  x = m;
}
```

global “x”

local “x”



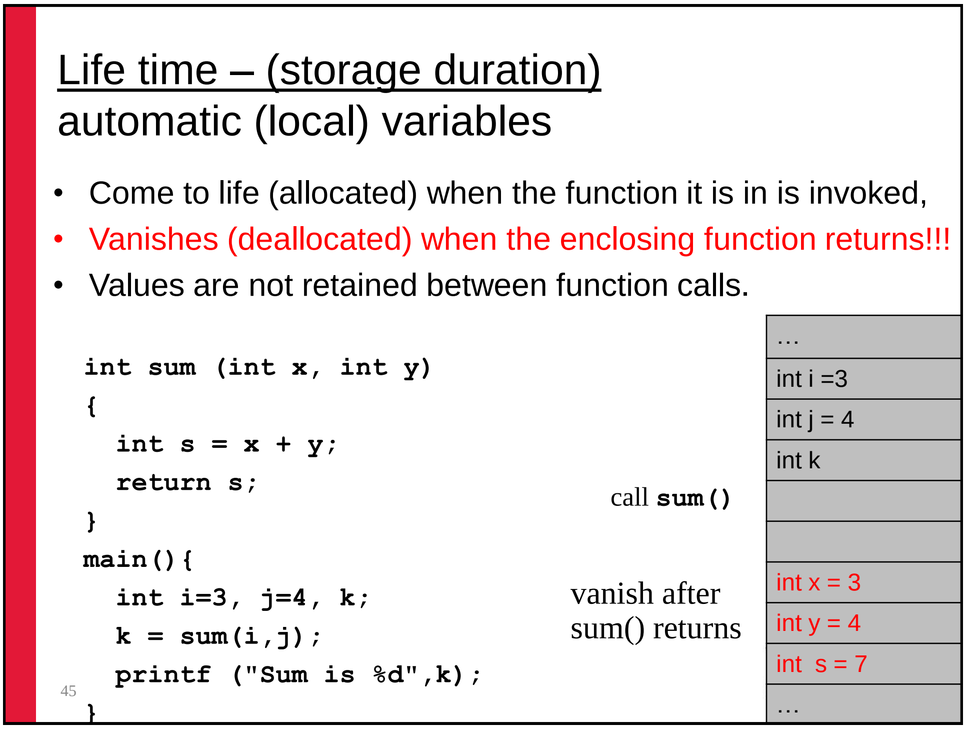
Life time – (storage duration)
automatic (local) variables

- Come to life (allocated) when the function it is in is invoked,
- Vanishes (deallocated) when the enclosing function returns!!!
- Values are not retained between function calls.

```
int sum (int x, int y)
{
  int s = x + y;
  return s;
}
main(){
  int i=3, j=4, k;
  k = sum(i,j);
  printf ("Sum is %d",k);
}
```

call **sum()**

…
int i =3
int j = 4
int k



Life time – (storage duration)
automatic (local) variables

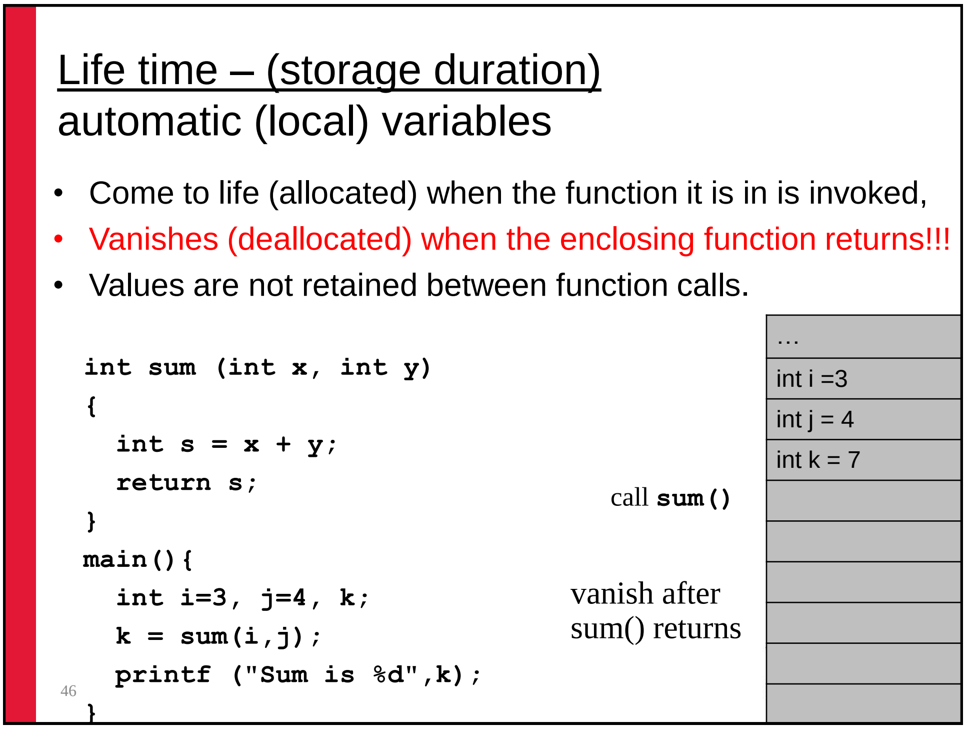
- Come to life (allocated) when the function it is in is invoked,
- Vanishes (deallocated) when the enclosing function returns!!!
- Values are not retained between function calls.

```
int sum (int x, int y)
{
  int s = x + y;
  return s;
}
main(){
  int i=3, j=4, k;
  k = sum(i,j);
  printf ("Sum is %d",k);
}
```

call **sum()**

vanish after
sum() returns

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



Life time – (storage duration)
automatic (local) variables

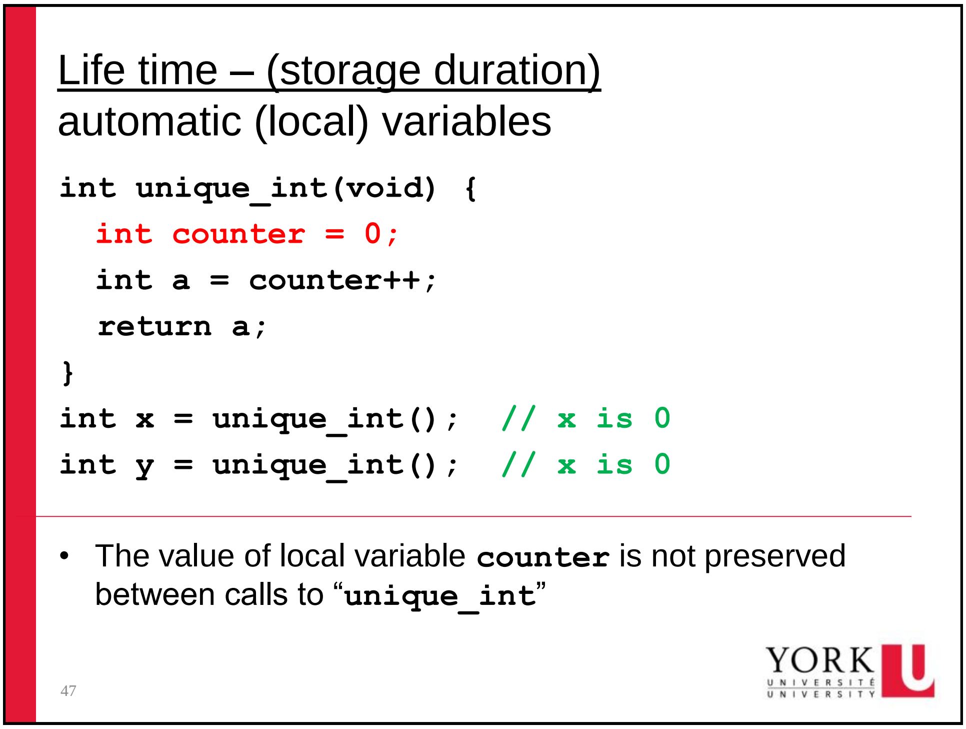
- Come to life (allocated) when the function it is in is invoked,
- Vanishes (deallocated) when the enclosing function returns!!!
- Values are not retained between function calls.

```
int sum (int x, int y)
{
  int s = x + y;
  return s;
}
main(){
  int i=3, j=4, k;
  k = sum(i,j);
  printf ("Sum is %d",k);
}
```

call **sum()**

vanish after
sum() returns

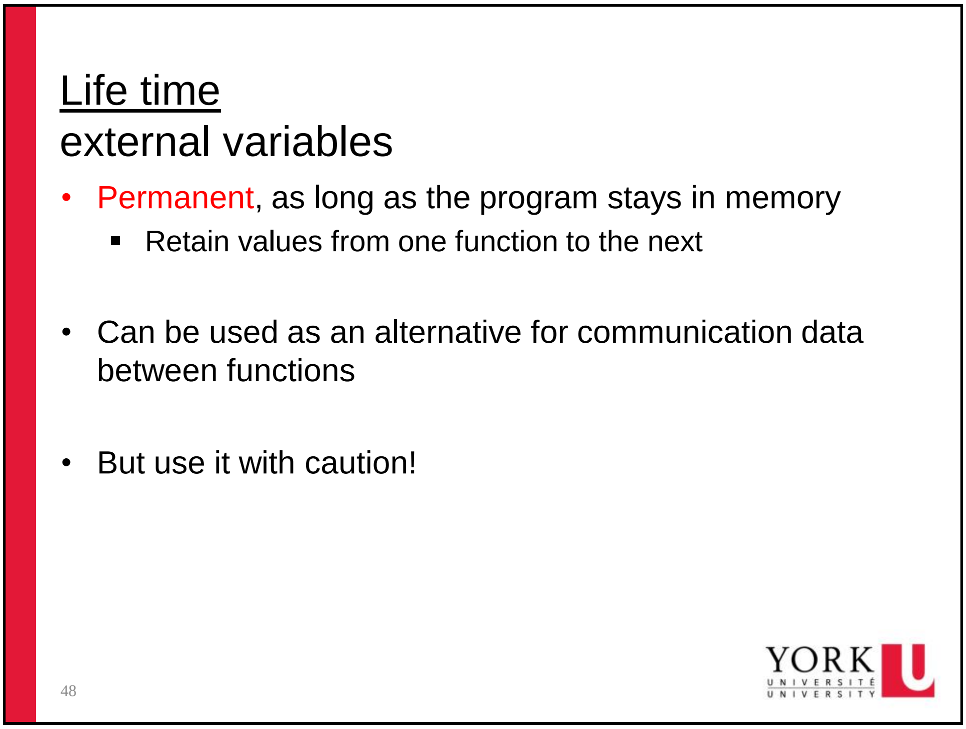
…
int i =3
int j = 4
int k = 7



Life time – (storage duration)
automatic (local) variables

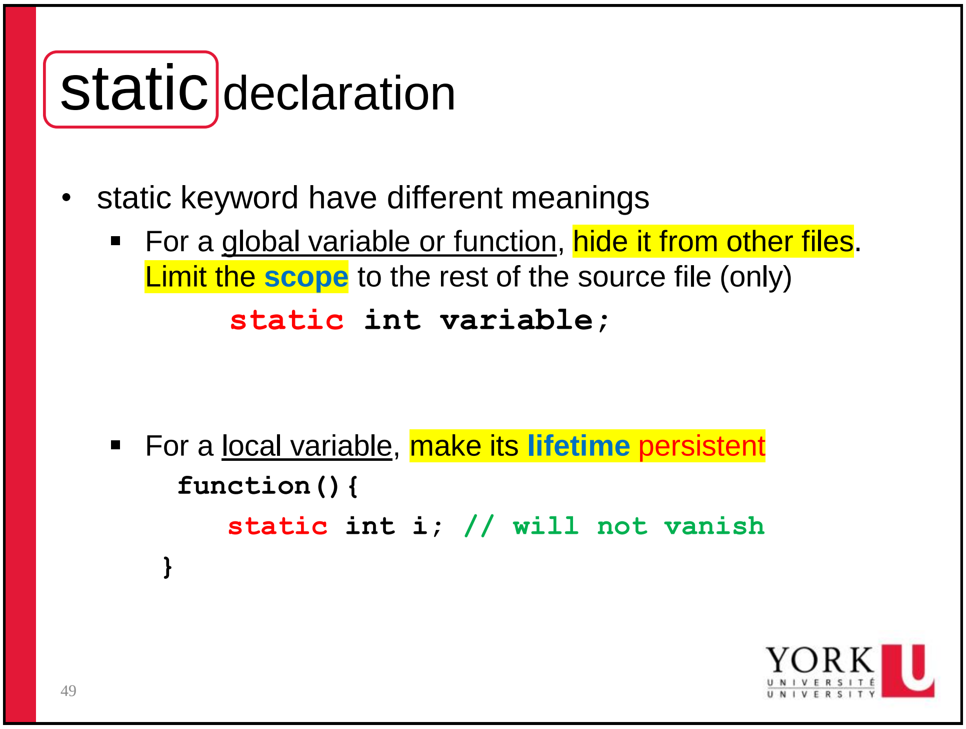
```
int unique_int(void) {
  int counter = 0;
  int a = counter++;
  return a;
}
int x = unique_int();  // x is 0
int y = unique_int();  // x is 0
```

- The value of local variable **`counter`** is not preserved between calls to “**`unique_int`**”



Life time
external variables

- Permanent, as long as the program stays in memory
  - Retain values from one function to the next
- Can be used as an alternative for communication data between functions
- But use it with caution!



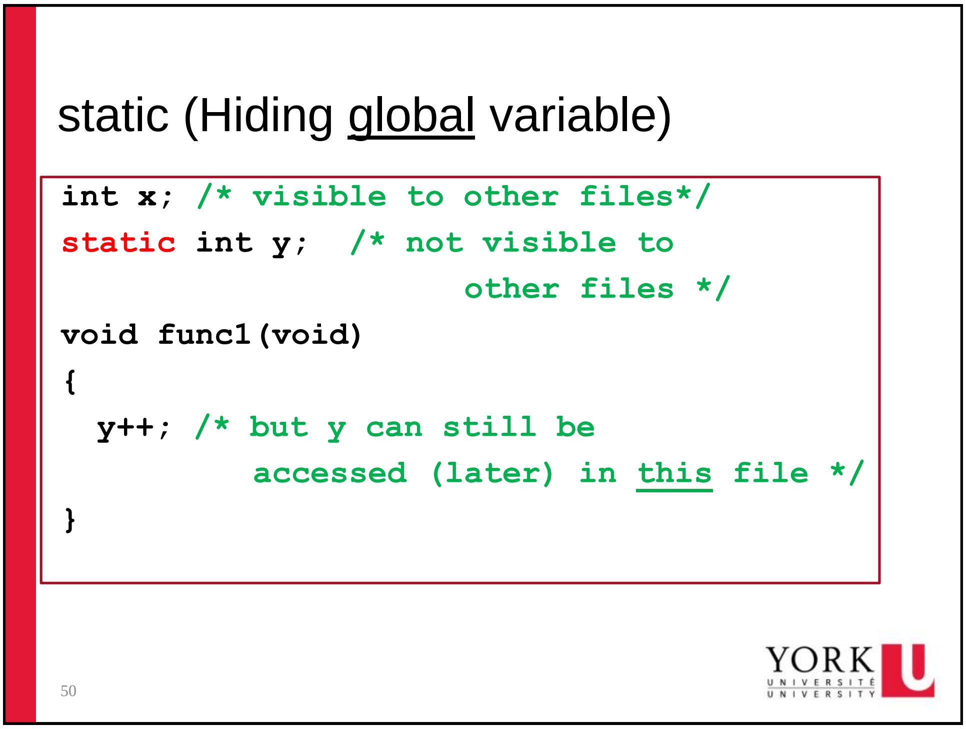
static declaration

- static keyword have different meanings
  - For a global variable or function, hide it from other files. Limit the **scope** to the rest of the source file (only)

```
static int variable;
```

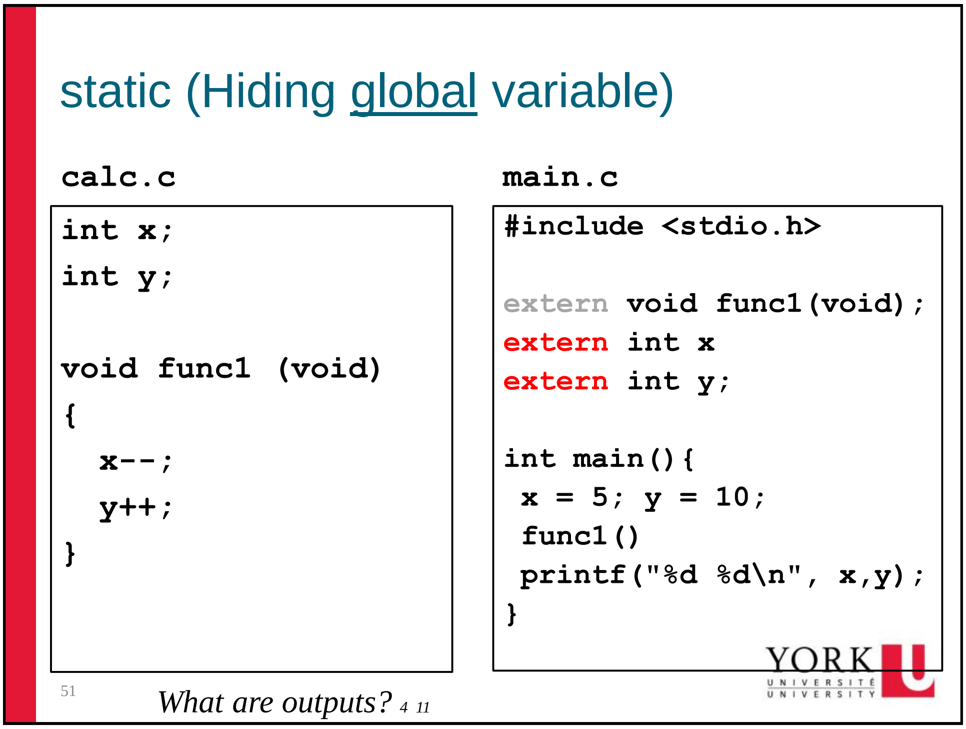
  - For a local variable, make its **lifetime** persistent

```
function(){
   static int i; // will not vanish
 }
```



static (Hiding global variable)

```
int x; /* visible to other files*/
static int y;  /* not visible to
                   other files */
void func1(void)
{
  y++; /* but y can still be
          accessed (later) in this file */
}
```



static (Hiding global variable)

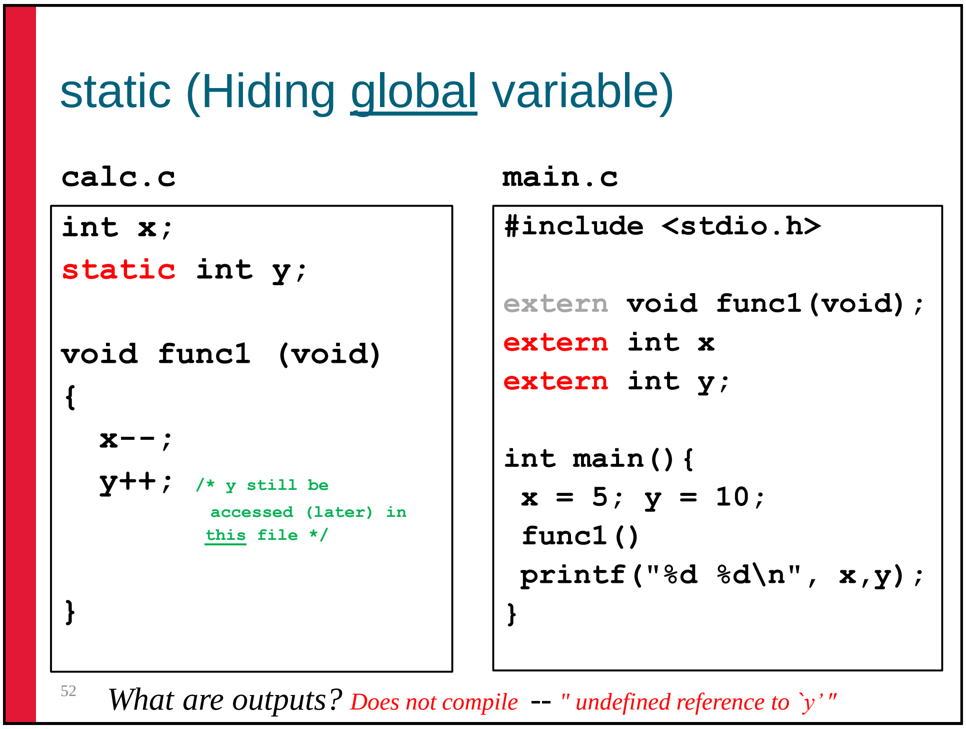
calc.c

```
int x;
int y;

void func1 (void)
{
   x--;
   y++;
}
```

main.c

```
#include <stdio.h>

extern void func1(void);
extern int x
extern int y;

int main(){
 x = 5; y = 10;
 func1()
 printf("%d %d\n", x,y);
}
```

What are outputs? 4 11



static (Hiding global variable)

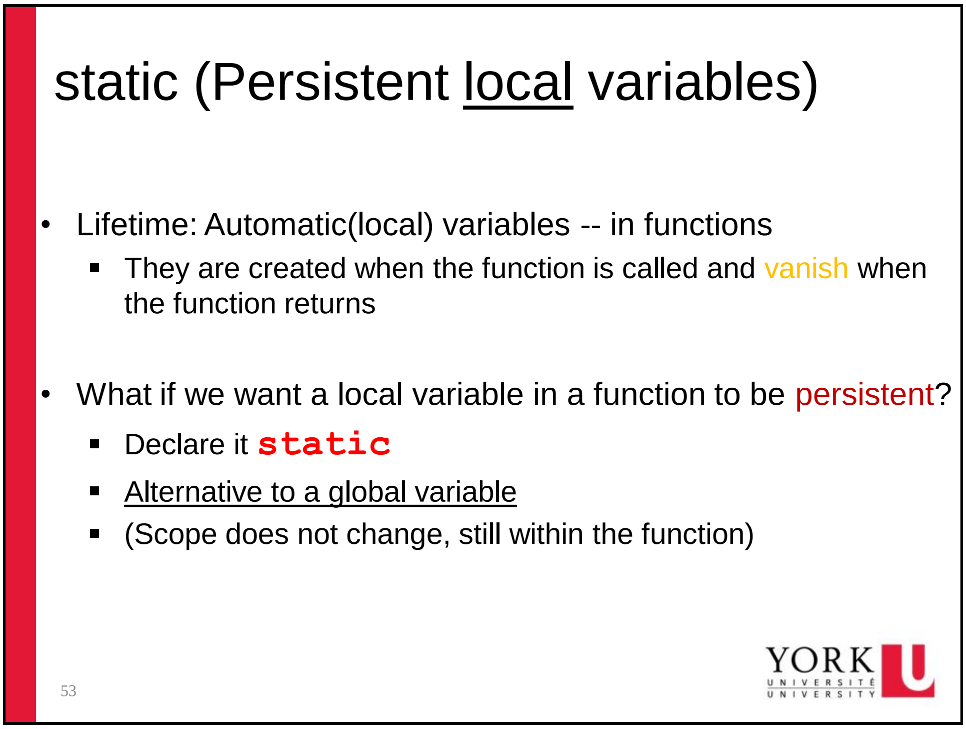
calc.c

```
int x;
static int y;

void func1 (void)
{
   x--;
   y++;  /* y still be
          accessed (later) in
          this file */


}
```

main.c

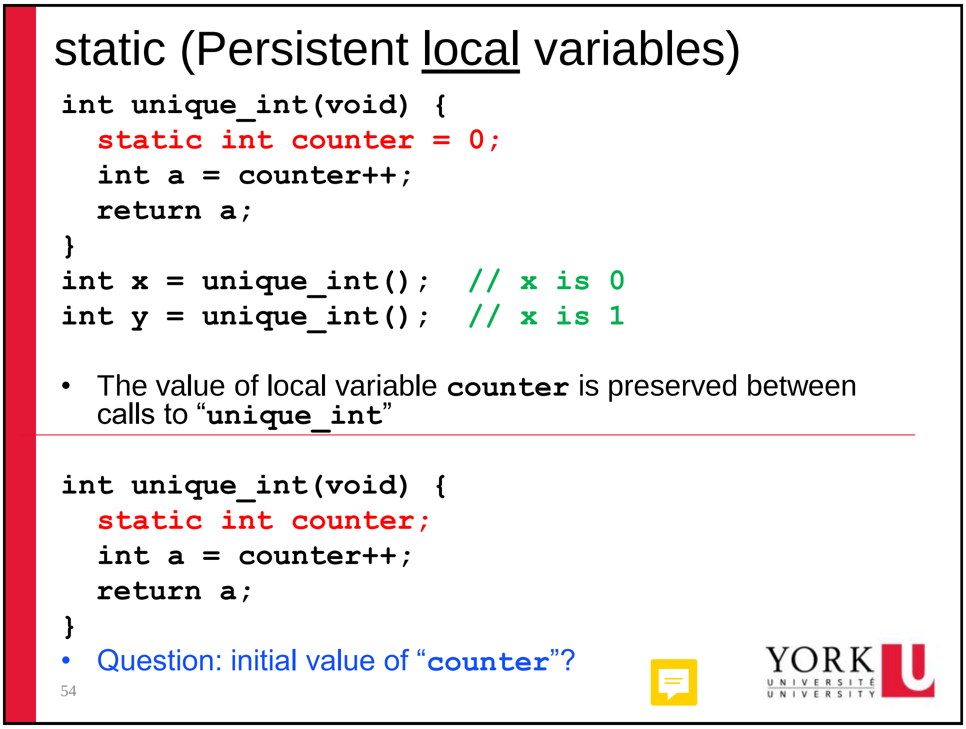
```
#include <stdio.h>

extern void func1(void);
extern int x
extern int y;

int main(){
 x = 5; y = 10;
 func1()
 printf("%d %d\n", x,y);
}
```

What are outputs? Does not compile -- " undefined reference to `y' "



static (Persistent local variables)

- Lifetime: Automatic(local) variables -- in functions
  - They are created when the function is called and vanish when the function returns
- What if we want a local variable in a function to be persistent?
  - Declare it **`static`**
  - Alternative to a global variable
  - (Scope does not change, still within the function)



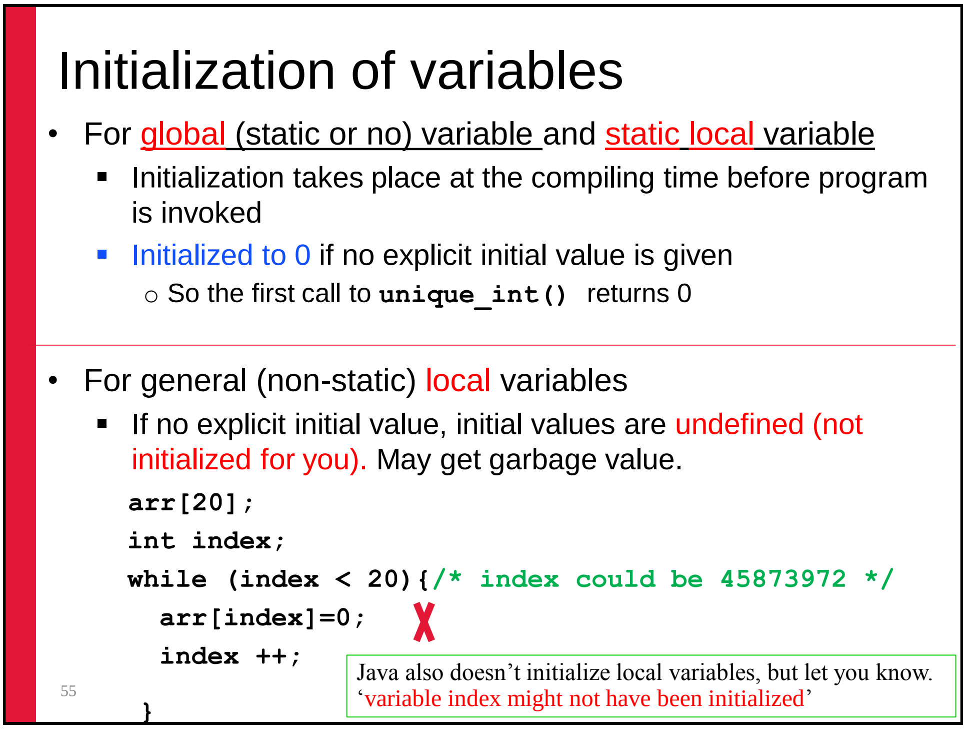
static (Persistent local variables)

```
int unique_int(void) {
  static int counter = 0;
  int a = counter++;
  return a;
}
int x = unique_int();  // x is 0
int y = unique_int();  // x is 1
```

- The value of local variable **`counter`** is preserved between calls to “**`unique_int`**”

```
int unique_int(void) {
  static int counter;
  int a = counter++;
  return a;
}
```

- Question: initial value of “**`counter`**”?



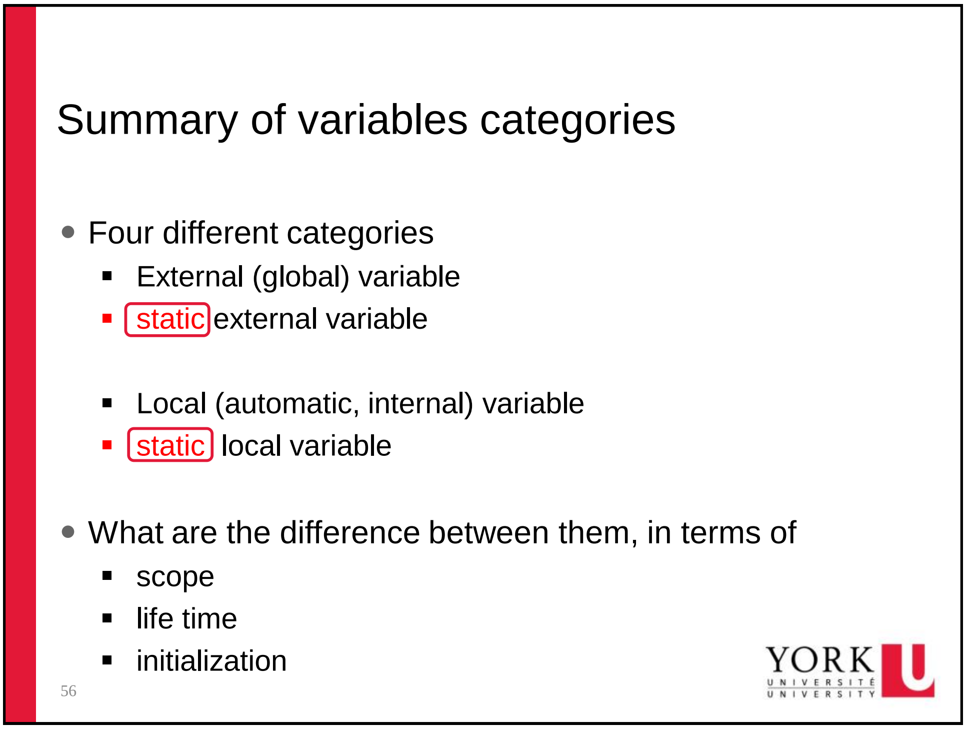
Initialization of variables

- For global (static or no) variable and static local variable
  - Initialization takes place at the compiling time before program is invoked
  - Initialized to 0 if no explicit initial value is given
    - So the first call to **`unique_int()`** returns 0

- For general (non-static) local variables
  - If no explicit initial value, initial values are undefined (not initialized for you). May get garbage value.

```
arr[20];
int index;
while (index < 20){/* index could be 45873972 */
  arr[index]=0;    ✗
  index ++;
 }
```

Java also doesn't initialize local variables, but let you know.
'variable index might not have been initialized'

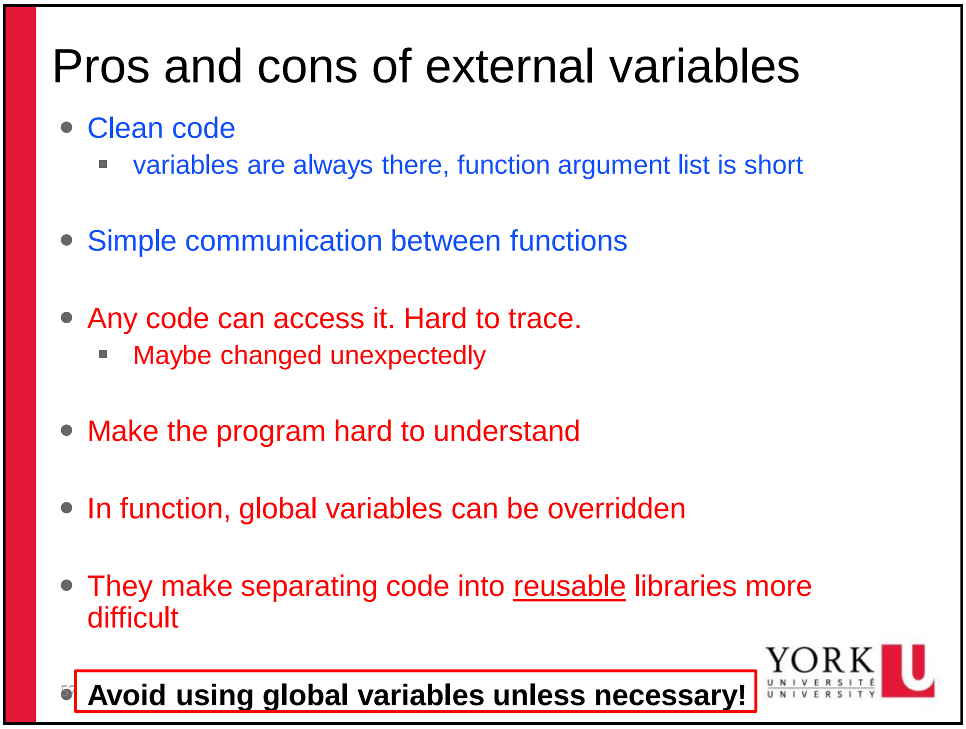


Summary of variables categories

- Four different categories
  - External (global) variable
  - static external variable

  - Local (automatic, internal) variable
  - static local variable

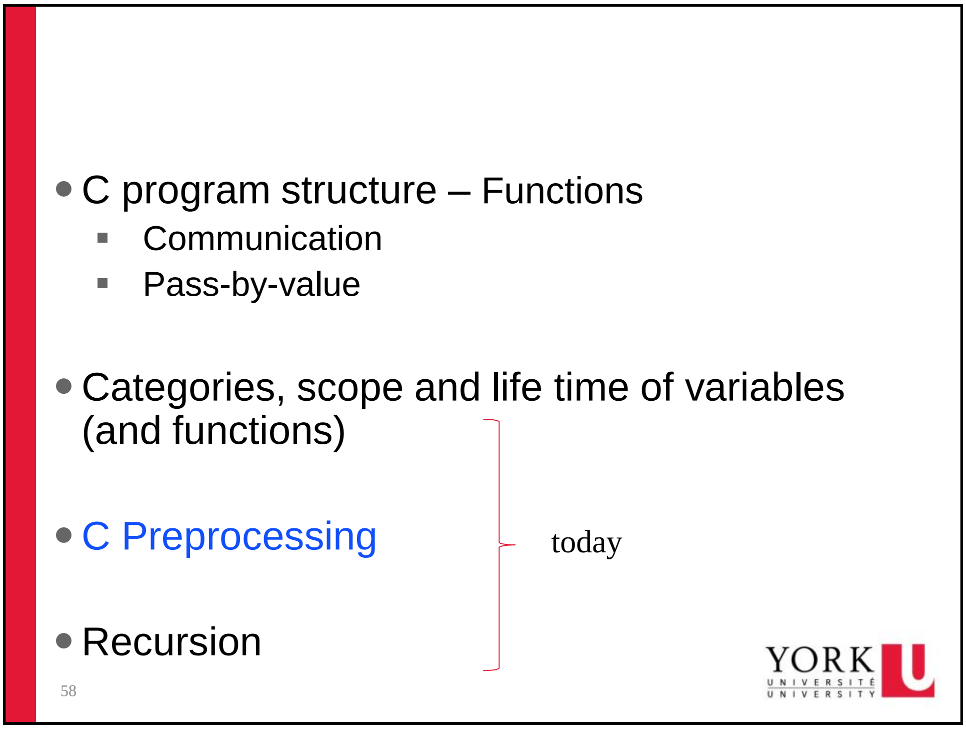
- What are the difference between them, in terms of
  - scope
  - life time
  - initialization



Pros and cons of external variables

- Clean code
  - variables are always there, function argument list is short
- Simple communication between functions
- Any code can access it. Hard to trace.
  - Maybe changed unexpectedly
- Make the program hard to understand
- In function, global variables can be overridden
- They make separating code into reusable libraries more difficult
- **Avoid using global variables unless necessary!**

57



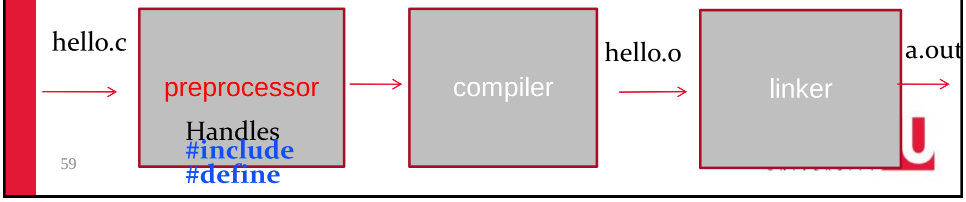
- C program structure – Functions
  - Communication
  - Pass-by-value
- Categories, scope and life time of variables (and functions)
- C Preprocessing
- Recursion

today

58

How C Programs are Compiled

- C programs go through three stages to be compiled:
  - Preprocessor - handles #include and #define etc
  - Compiler - converts C code into binary processor instructions (“object code”)
  - Linker - puts multiple files together, load necessary library functions (e.g., printf), and creates an executable program



hello.c

preprocessor

Handles
#include
#define

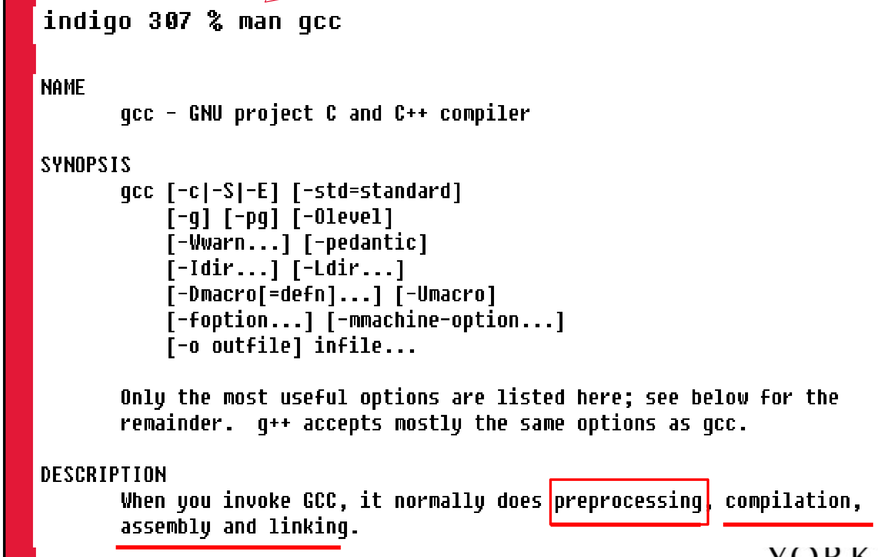
compiler

hello.o

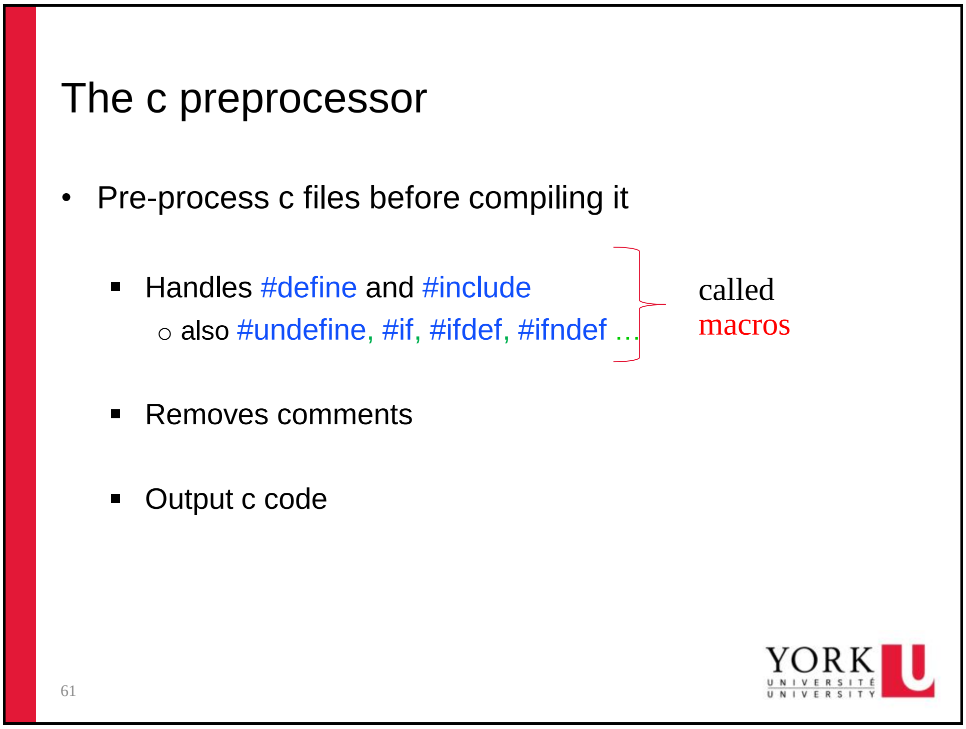
linker

a.out

“manual”, get used to it for help



```
indigo 307 % man gcc

NAME
       gcc - GNU project C and C++ compiler

SYNOPSIS
       gcc [-c|-S|-E] [-std=standard]
           [-g] [-pg] [-Olevel]
           [-Wwarn...] [-pedantic]
           [-Idir...] [-Ldir...]
           [-Dmacro[=defn]...] [-Umacro]
           [-foption...] [-mmachine-option...]
           [-o outfile] infile...

       Only the most useful options are listed here; see below for the
       remainder.  g++ accepts mostly the same options as gcc.

DESCRIPTION
       When you invoke GCC, it normally does preprocessing, compilation,
       assembly and linking.
```



The c preprocessor

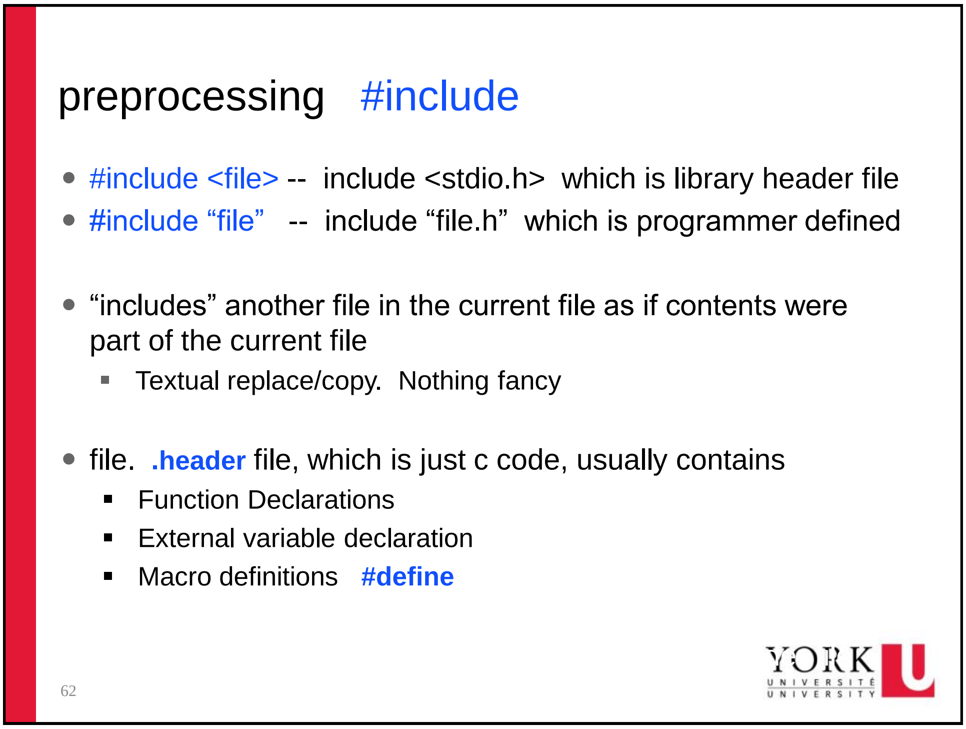
- Pre-process c files before compiling it
  - Handles #define and #include
    - also #undefine, #if, #ifdef, #ifndef …

called
macros

  - Removes comments
  - Output c code

61

61



preprocessing #include

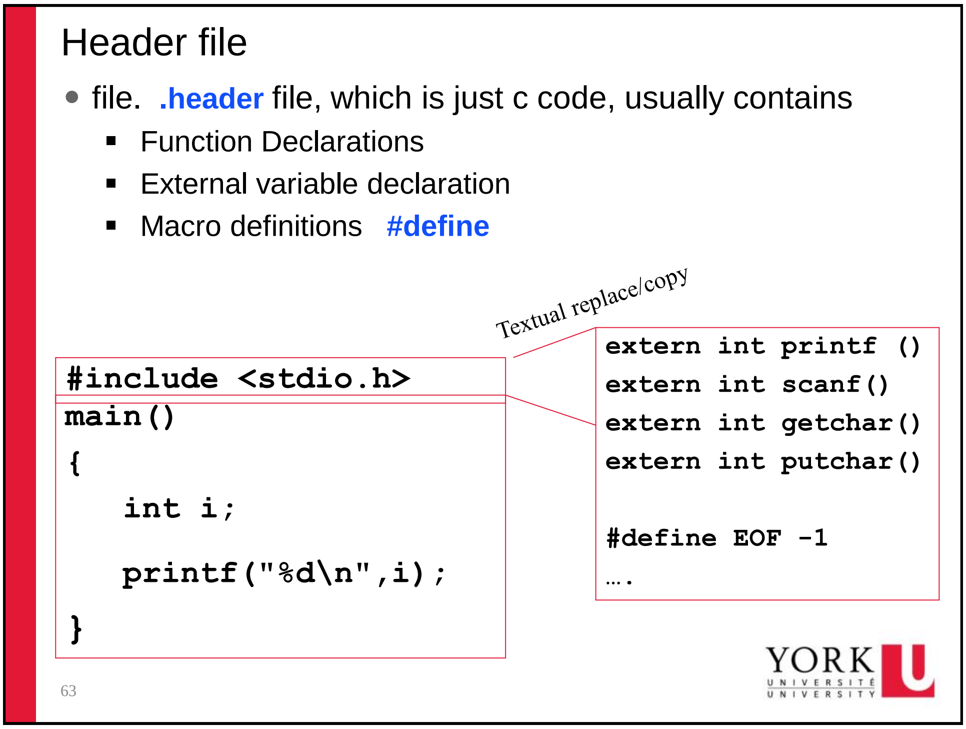
- #include <file> -- include <stdio.h> which is library header file
- #include “file” -- include “file.h” which is programmer defined

- “includes” another file in the current file as if contents were part of the current file
  - Textual replace/copy. Nothing fancy

- file. **.header** file, which is just c code, usually contains
  - Function Declarations
  - External variable declaration
  - Macro definitions **#define**

62

62



Header file

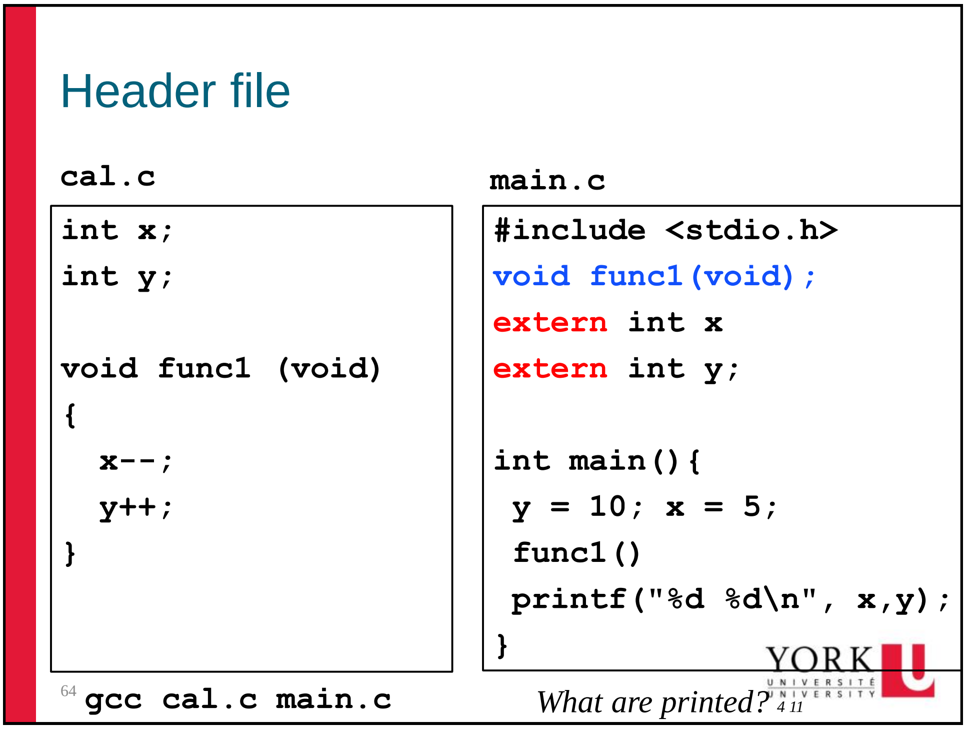
- file. **.header** file, which is just c code, usually contains
  - Function Declarations
  - External variable declaration
  - Macro definitions **#define**

```
#include <stdio.h>
main()
{
   int i;

   printf("%d\n",i);
}
```

Textual replace/copy

```
extern int printf ()
extern int scanf()
extern int getchar()
extern int putchar()

#define EOF -1
….
```

63



Header file

cal.c

```
int x;
int y;

void func1 (void)
{
  x--;
  y++;
}
```

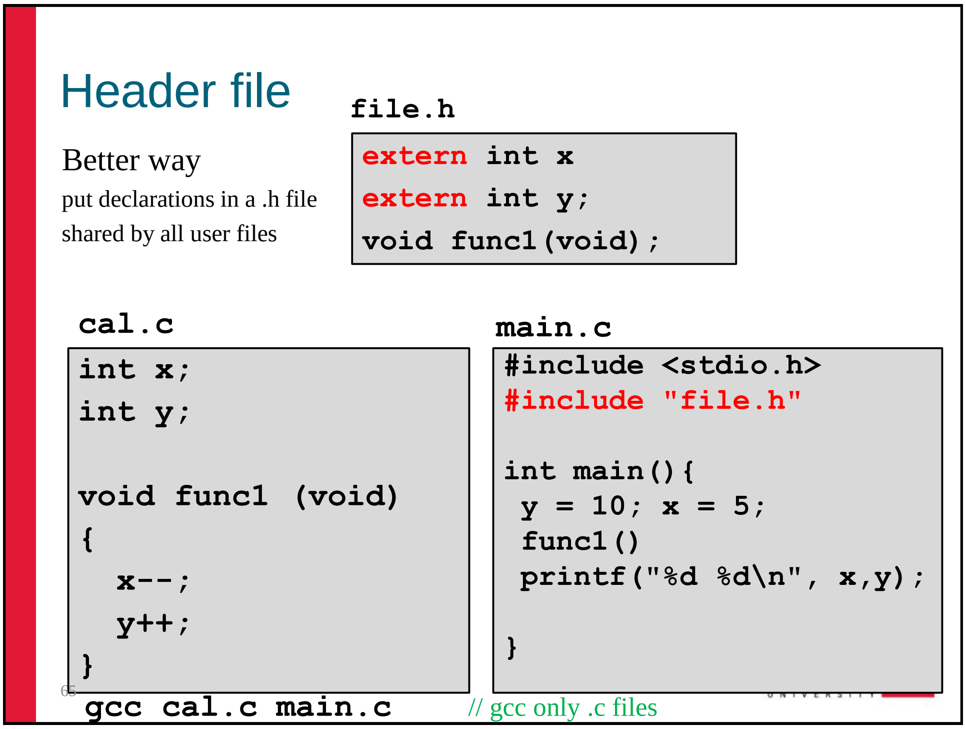
main.c

```
#include <stdio.h>
void func1(void);
extern int x
extern int y;

int main(){
 y = 10; x = 5;
 func1()
 printf("%d %d\n", x,y);
}
```

```
gcc cal.c main.c
```

What are printed? 4 11

64



Header file

Better way

put declarations in a .h file

shared by all user files

file.h

```
extern int x
extern int y;
void func1(void);
```

cal.c

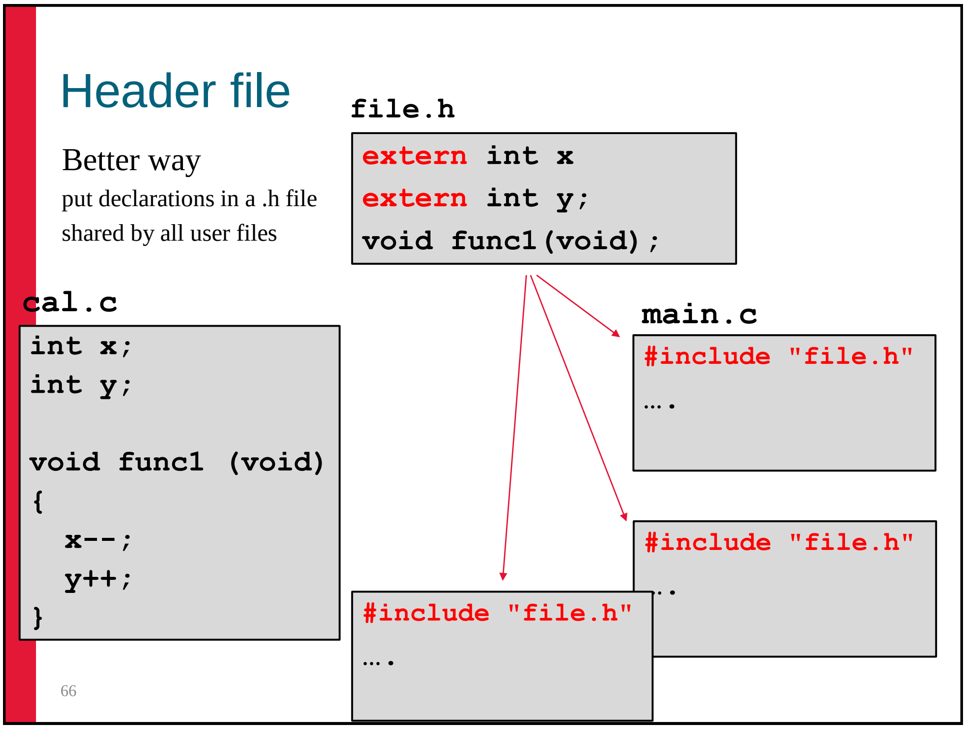
```
int x;
int y;

void func1 (void)
{
  x--;
  y++;
}
```

main.c

```
#include <stdio.h>
#include "file.h"

int main(){
 y = 10; x = 5;
 func1()
 printf("%d %d\n", x,y);

}
```

```
gcc cal.c main.c
```

// gcc only .c files



Header file

Better way

put declarations in a .h file

shared by all user files

file.h

```
extern int x
extern int y;
void func1(void);
```

cal.c

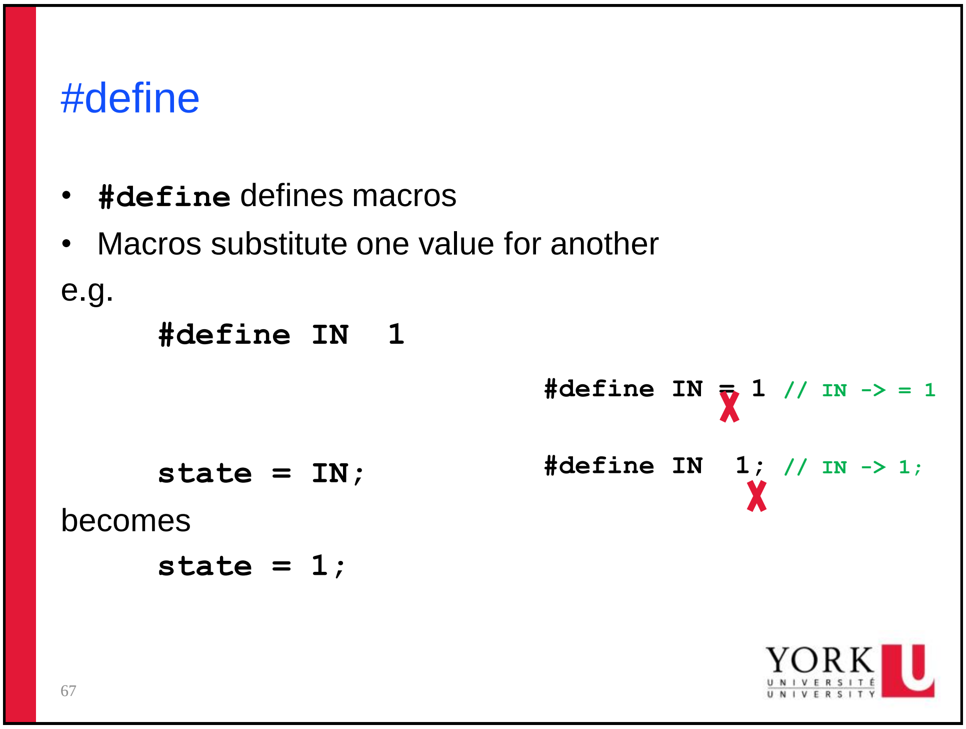
```
int x;
int y;

void func1 (void)
{
  x--;
  y++;
}
```

main.c

```
#include "file.h"
….
```

```
#include "file.h"
..
```

```
#include "file.h"
….
```



#define

- **`#define`** defines macros
- Macros substitute one value for another

e.g.

```
    #define IN  1
```

```
#define IN = 1 // IN -> = 1
```

```
#define IN  1; // IN -> 1;
```

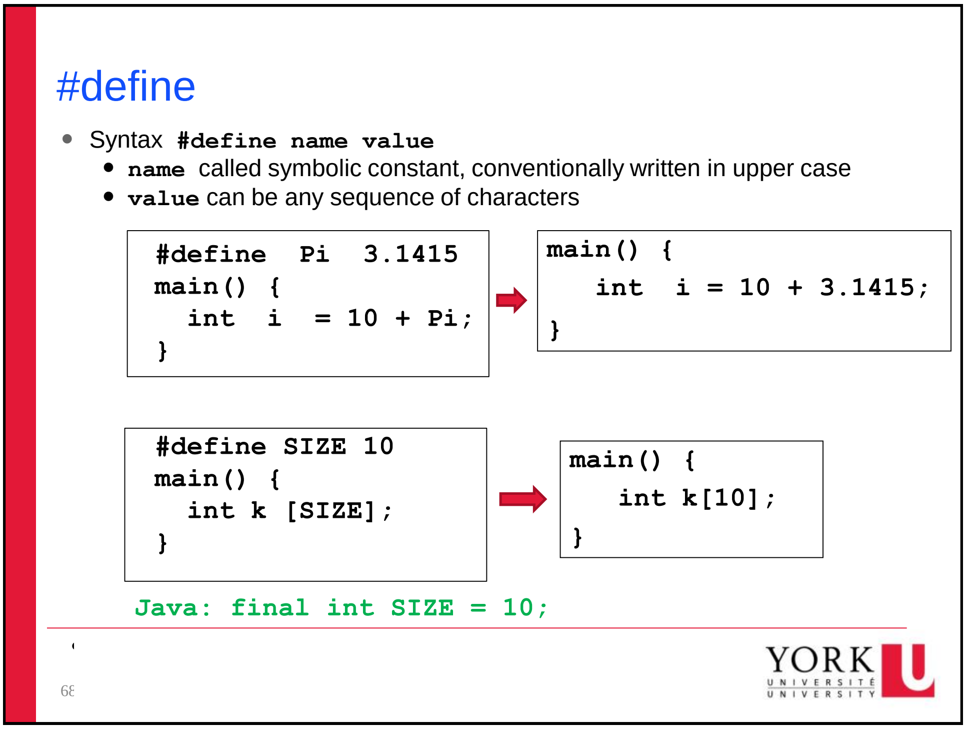
```
    state = IN;
```

becomes

```
    state = 1;
```

67

67



#define

- Syntax **`#define name value`**
  - **`name`** called symbolic constant, conventionally written in upper case
  - **`value`** can be any sequence of characters

```
#define  Pi  3.1415
main() {
  int  i  = 10 + Pi;
}
```

```
main() {
   int  i = 10 + 3.1415;
}
```

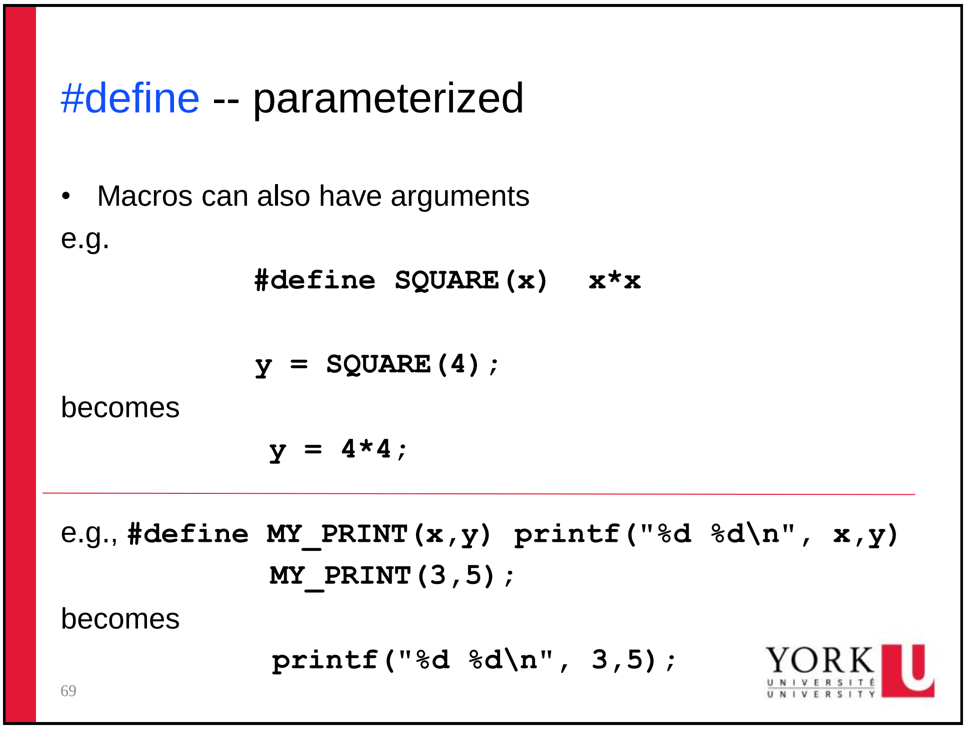
```
#define SIZE 10
main() {
  int k [SIZE];
}
```

```
main() {
   int k[10];
}
```

```
Java: final int SIZE = 10;
```

68

68



#define -- parameterized

- Macros can also have arguments

e.g.

```
#define SQUARE(x)  x*x

y = SQUARE(4);
```

becomes

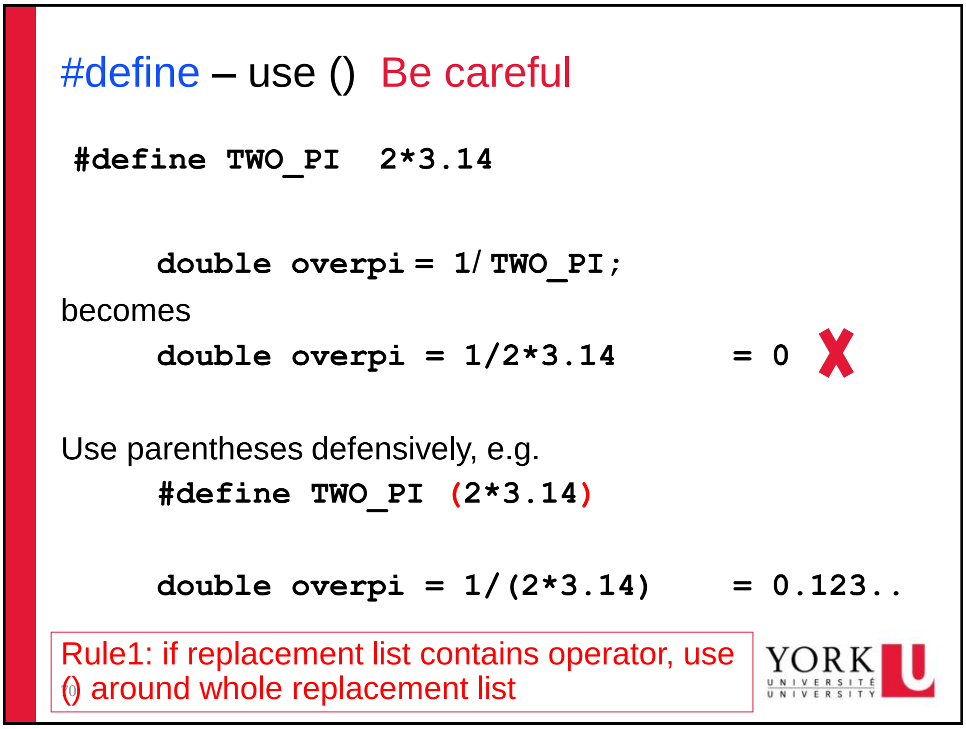
```
 y = 4*4;
```

e.g., `#define MY_PRINT(x,y) printf("%d %d\n", x,y)`

```
MY_PRINT(3,5);
```

becomes

```
printf("%d %d\n", 3,5);
```



#define – use () Be careful

```
#define TWO_PI  2*3.14


    double overpi = 1/ TWO_PI;
```

becomes

```
    double overpi = 1/2*3.14      = 0 ✗
```

Use parentheses defensively, e.g.

```
    #define TWO_PI (2*3.14)

    double overpi = 1/(2*3.14)    = 0.123..
```

Rule1: if replacement list contains operator, use () around whole replacement list

#define – parameterized. Be careful with arguments

```
#define TRIPLE(x)  x * 3
```

Be careful with arguments

```
     y= TRIPLE(5+2)
```

becomes

```
     y=5+2 * 3      = 11   ✗
```

Use parentheses defensively, e.g.

```
     #define TRIPLE(x)   ((x) * 3 )
     y= ((5+2) * 3)      = 21
```

Rule2: for parameterized, put () around each parameter occurrence in the replacement list

#define – parameterized. Be careful with arguments

```
#define SQUARE(x)  x*x
```

Be careful with arguments

```
     SQUARE(5+2)
```

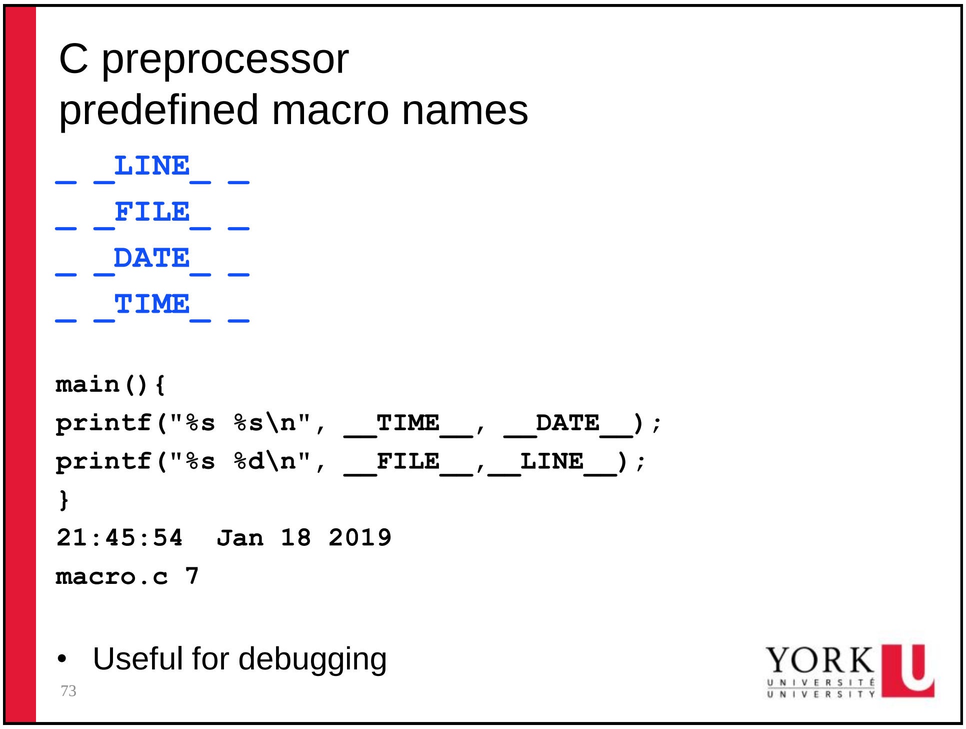
becomes

```
     5+2*5+2        = 17   ✗
```

Use parentheses defensively, e.g.

```
     #define SQUARE(x) ( (x)*(x) )
     ((5+2)*(5+2))       = 49
```

Rule2: for parameterized, put () around each parameter occurrence in the replacement list

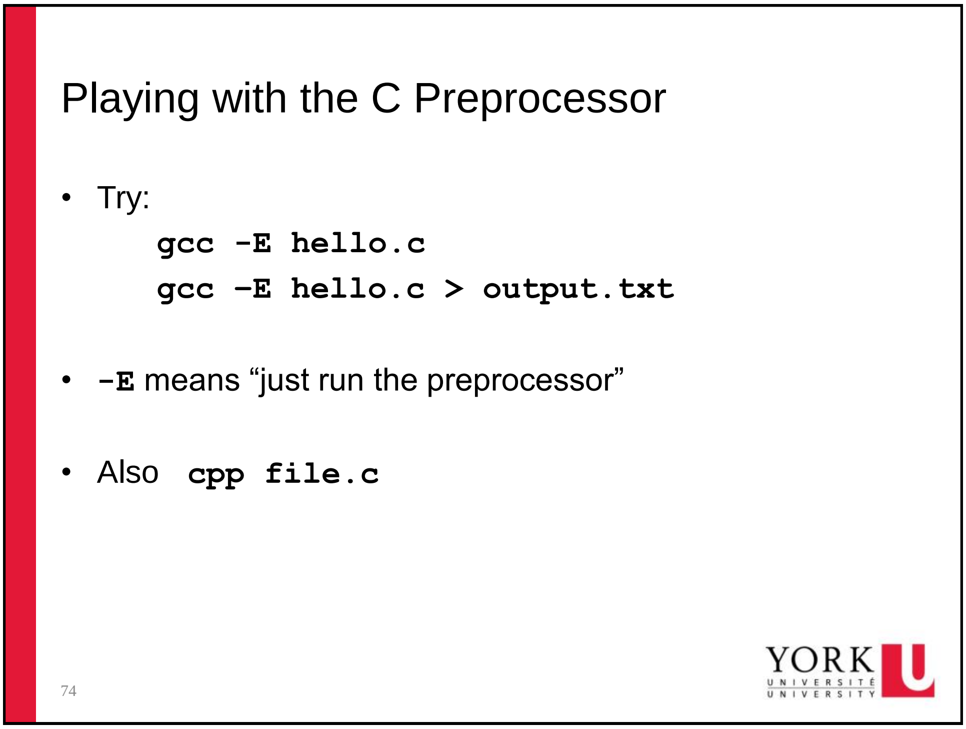


C preprocessor
predefined macro names

```
_ _LINE_ _
_ _FILE_ _
_ _DATE_ _
_ _TIME_ _
```

```
main(){
printf("%s %s\n", __TIME__, __DATE__);
printf("%s %d\n", __FILE__,__LINE__);
}
21:45:54  Jan 18 2019
macro.c 7
```

- Useful for debugging



Playing with the C Preprocessor

- Try:

```
gcc -E hello.c
gcc –E hello.c > output.txt
```

- **-E** means “just run the preprocessor”
- Also **cpp file.c**

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

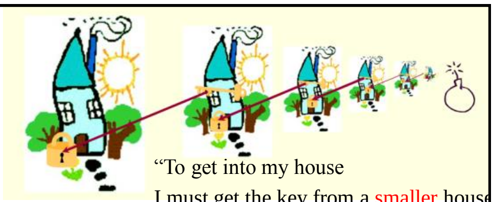
Recursion

“To get into my house
I must get the key from a smaller house

```
int length (string s)
   if (s contains no letter)
     return 0;
   return 1 + length(substring on the right);
}
```

length(“ABCD”)
= 1 + length(“BCD”)
= 1 + (1 + length(“CD”))
= 1 + (1 +   (1 + length(“D”)))
= 1 + (1 +   (1 + (1+ (1+length(“”))))
= 1 + (1 +   (1 + (1+ (1+0)))) = 4

Recursion



“To get into my house
I must get the key from a smaller house

```
int length (String s) // Java
   if (s.equals("") // contains no letter
     return 0;
   return 1 + length(s.substring(1));
}
```

length(“ABCD”)
= 1 + length(“BCD”)
= 1 + (1 + length(“CD”))
= 1 + (1 + (1 + length(“D”)))
= 1 + (1 + (1 + (1+ (1+length(“”))))
= 1 + (1 + (1 + (1+ (1+0)))) = 4

C version?

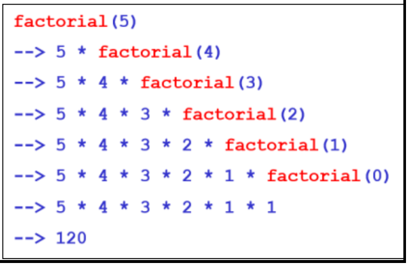
Recursion

- C supports recursion
- Think/define recursively

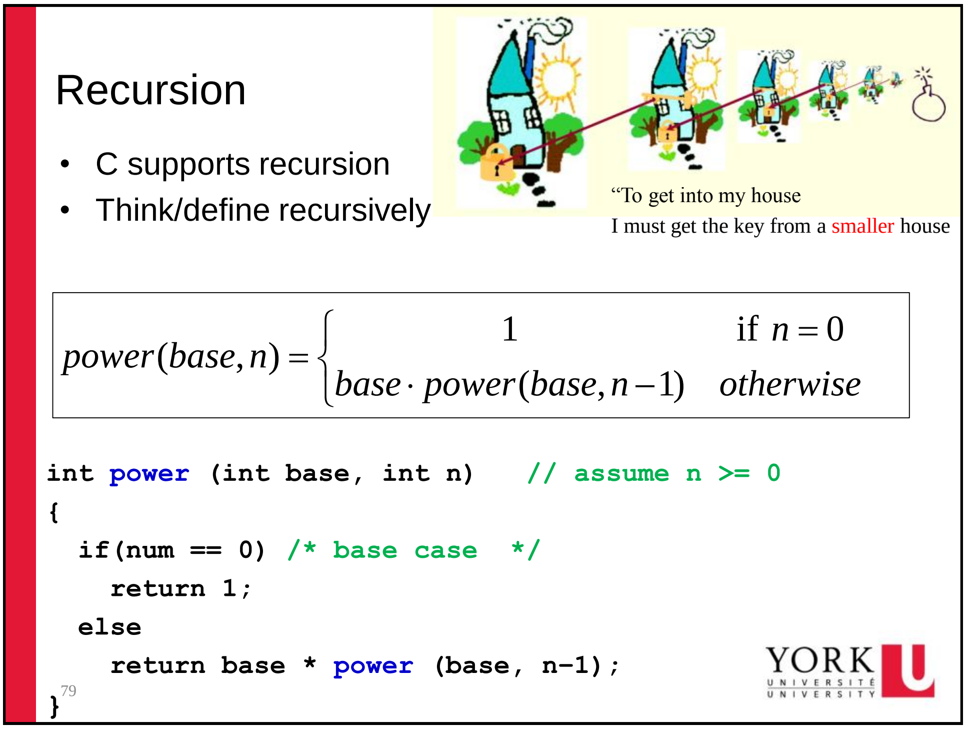
“To get into my house
I must get the key from a smaller house

$$factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot factorial(n-1) & otherwise \end{cases}$$

```
int factorial (int n)
{
  if(n == 0) /* base case  */
   return 1;
  else
   return n * factorial (n - 1);
}
```



```
factorial(5)
--> 5 * factorial(4)
--> 5 * 4 * factorial(3)
--> 5 * 4 * 3 * factorial(2)
--> 5 * 4 * 3 * 2 * factorial(1)
--> 5 * 4 * 3 * 2 * 1 * factorial(0)
--> 5 * 4 * 3 * 2 * 1 * 1
--> 120
```



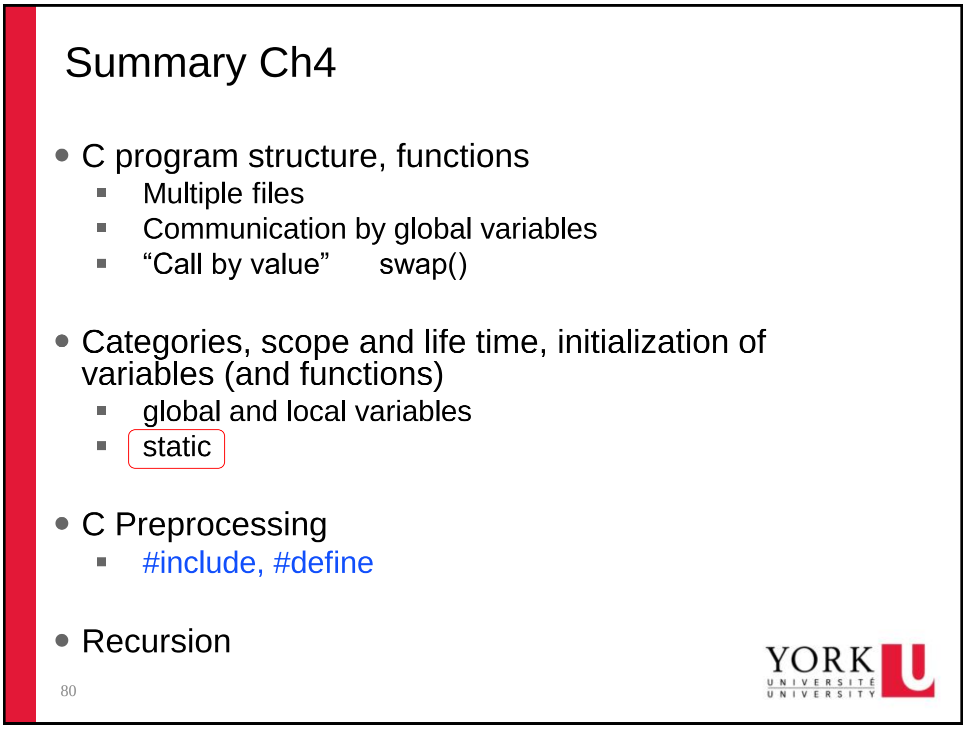
Recursion

- C supports recursion
- Think/define recursively

“To get into my house
I must get the key from a smaller house

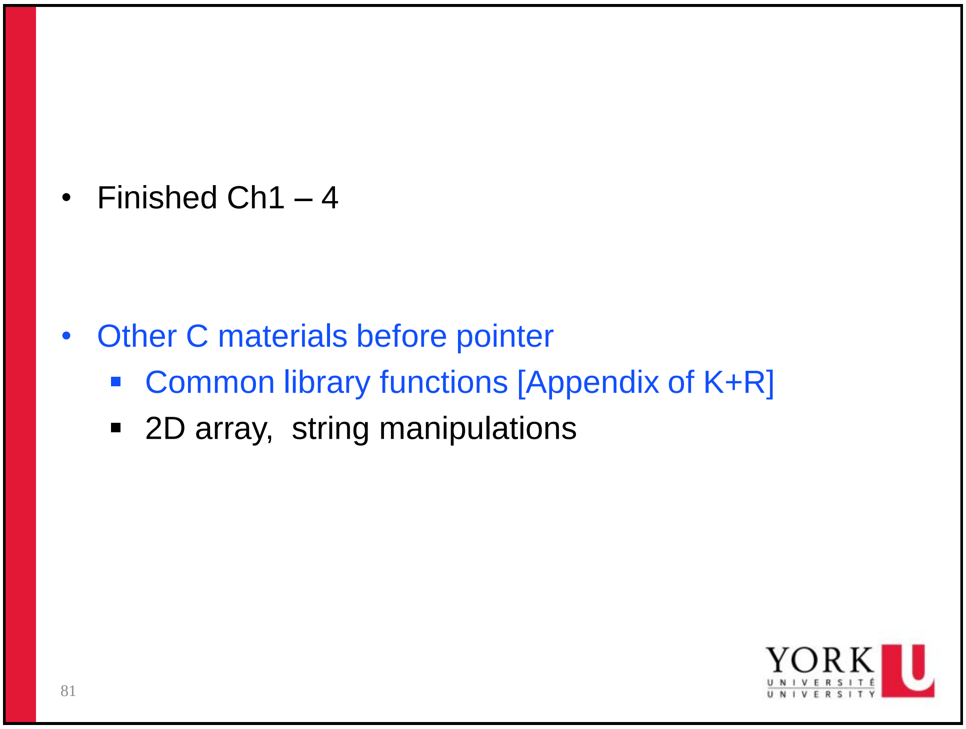
$$power(base, n) = \begin{cases} 1 & \text{if } n = 0 \\ base \cdot power(base, n-1) & otherwise \end{cases}$$

```
int power (int base, int n)    // assume n >= 0
{
  if(num == 0) /* base case  */
    return 1;
  else
    return base * power (base, n-1);
}
```



Summary Ch4

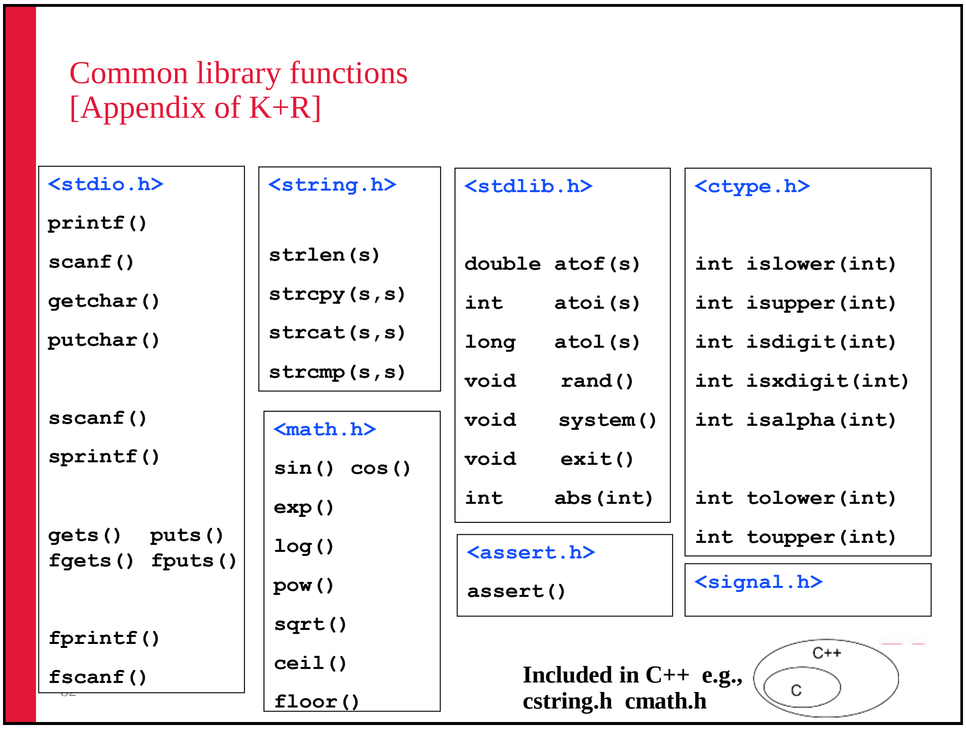
- C program structure, functions
  - Multiple files
  - Communication by global variables
  - “Call by value”   swap()
- Categories, scope and life time, initialization of variables (and functions)
  - global and local variables
  - static
- C Preprocessing
  - #include, #define
- Recursion



- Finished Ch1 – 4

- Other C materials before pointer
  - Common library functions [Appendix of K+R]
  - 2D array, string manipulations

YORK
UNIVERSITÉ
UNIVERSITY
U
81



Common library functions [Appendix of K+R]

```
<stdio.h>
printf()
scanf()
getchar()
putchar()

sscanf()
sprintf()

gets()   puts()
fgets()  fputs()

fprintf()
fscanf()
```

```
<string.h>

strlen(s)
strcpy(s,s)
strcat(s,s)
strcmp(s,s)
```

```
<math.h>
sin()  cos()
exp()
log()
pow()
sqrt()
ceil()
floor()
```

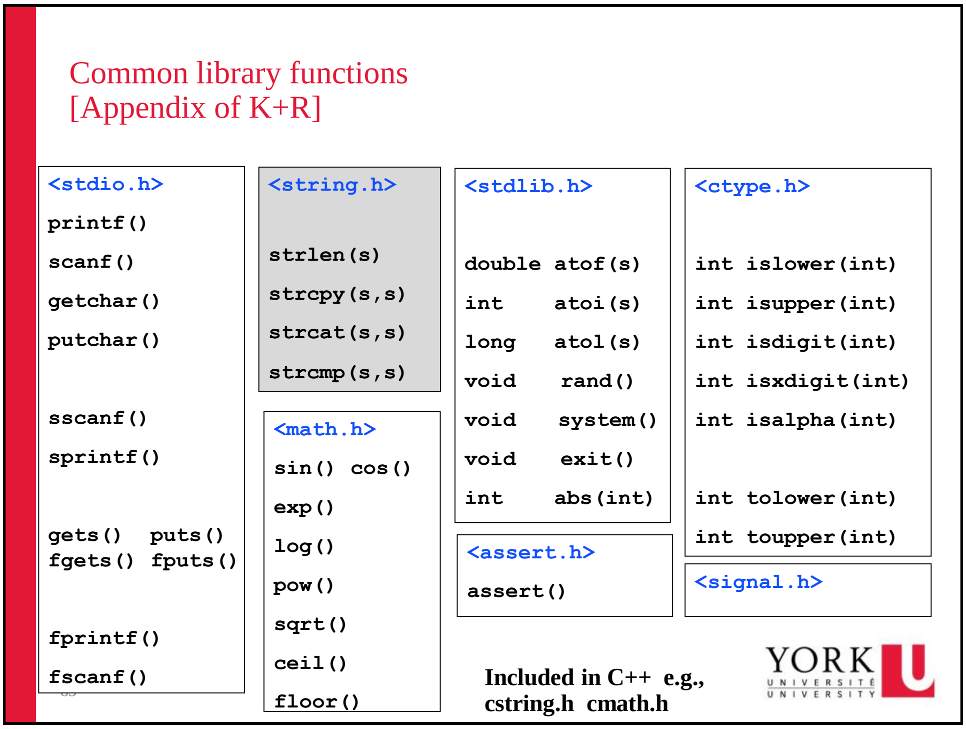
```
<stdlib.h>

double  atof(s)
int     atoi(s)
long    atol(s)
void     rand()
void    system()
void     exit()
int     abs(int)
```

```
<assert.h>
assert()
```

```
<ctype.h>

int islower(int)
int isupper(int)
int isdigit(int)
int isxdigit(int)
int isalpha(int)

int tolower(int)
int toupper(int)
```

```
<signal.h>
```

Included in C++ e.g., cstring.h cmath.h



Common library functions [Appendix of K+R]

```
<stdio.h>
printf()
scanf()
getchar()
putchar()

sscanf()
sprintf()

gets()   puts()
fgets()  fputs()

fprintf()
fscanf()
```

```
<string.h>

strlen(s)
strcpy(s,s)
strcat(s,s)
strcmp(s,s)
```

```
<math.h>
sin() cos()
exp()
log()
pow()
sqrt()
ceil()
floor()
```

```
<stdlib.h>

double atof(s)
int    atoi(s)
long   atol(s)
void    rand()
void    system()
void    exit()
int    abs(int)
```

```
<assert.h>
assert()
```

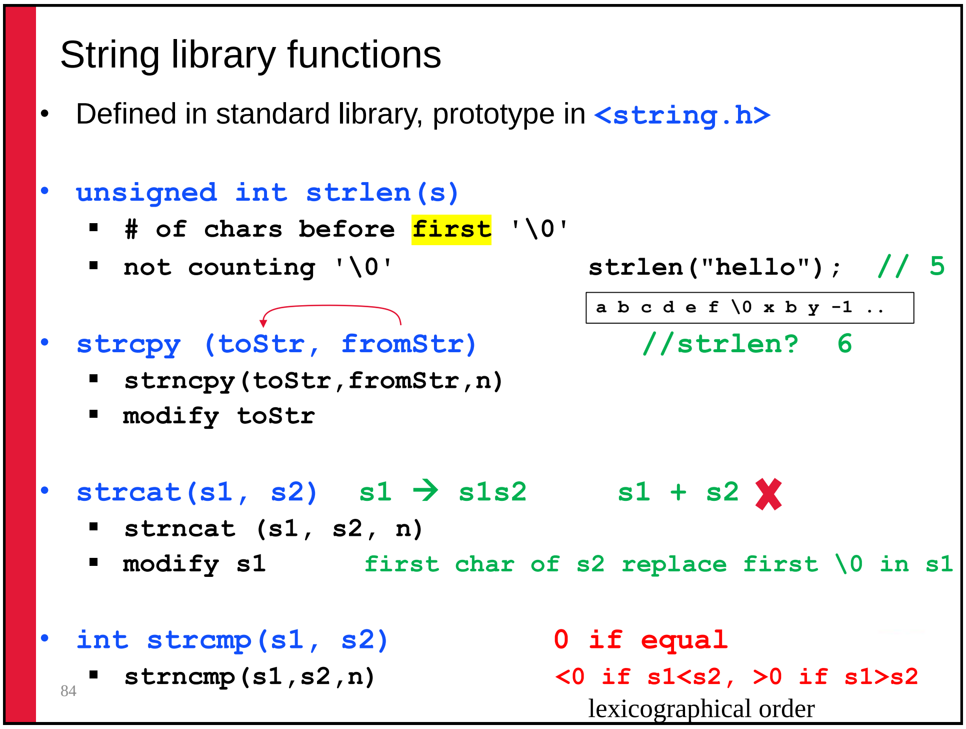
```
<ctype.h>

int islower(int)
int isupper(int)
int isdigit(int)
int isxdigit(int)
int isalpha(int)

int tolower(int)
int toupper(int)
```

```
<signal.h>
```

83

Included in C++ e.g., cstring.h cmath.h

83



String library functions

- Defined in standard library, prototype in `<string.h>`

- `unsigned int strlen(s)`
  - `# of chars before first '\0'`
  - `not counting '\0'`

`strlen("hello");   // 5`

a	b	c	d	e	f	\0	x	b	y	-1	..

`//strlen?   6`

- `strcpy (toStr, fromStr)`
  - `strncpy(toStr,fromStr,n)`
  - `modify toStr`

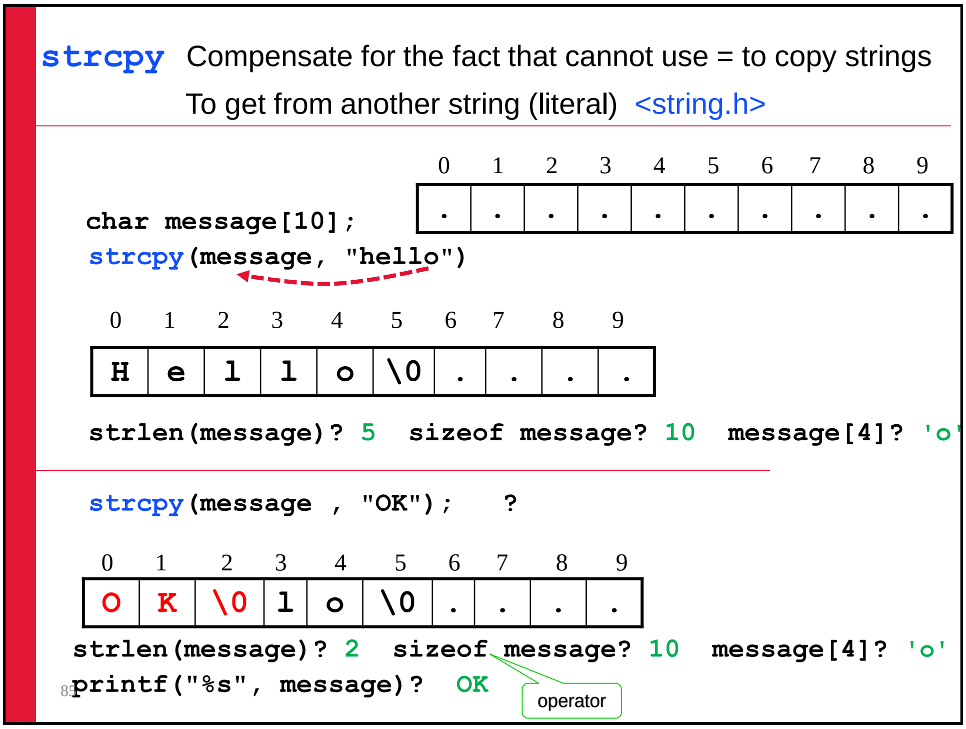
- `strcat(s1, s2)   s1 → s1s2      s1 + s2 ✗`
  - `strncat (s1, s2, n)`
  - `modify s1       first char of s2 replace first \0 in s1`

- `int strcmp(s1, s2)          0 if equal`
  - `strncmp(s1,s2,n)          <0 if s1<s2, >0 if s1>s2`

lexicographical order

84

84



strcpy Compensate for the fact that cannot use = to copy strings

To get from another string (literal) <string.h>

0	1	2	3	4	5	6	7	8	9
.	.	.	.	.	.	.	.	.	.

```
char message[10];
strcpy(message, "hello")
```

0	1	2	3	4	5	6	7	8	9
H	e	l	l	o	\0	.	.	.	.

```
strlen(message)? 5   sizeof message? 10   message[4]? 'o'
```

```
strcpy(message , "OK");    ?
```

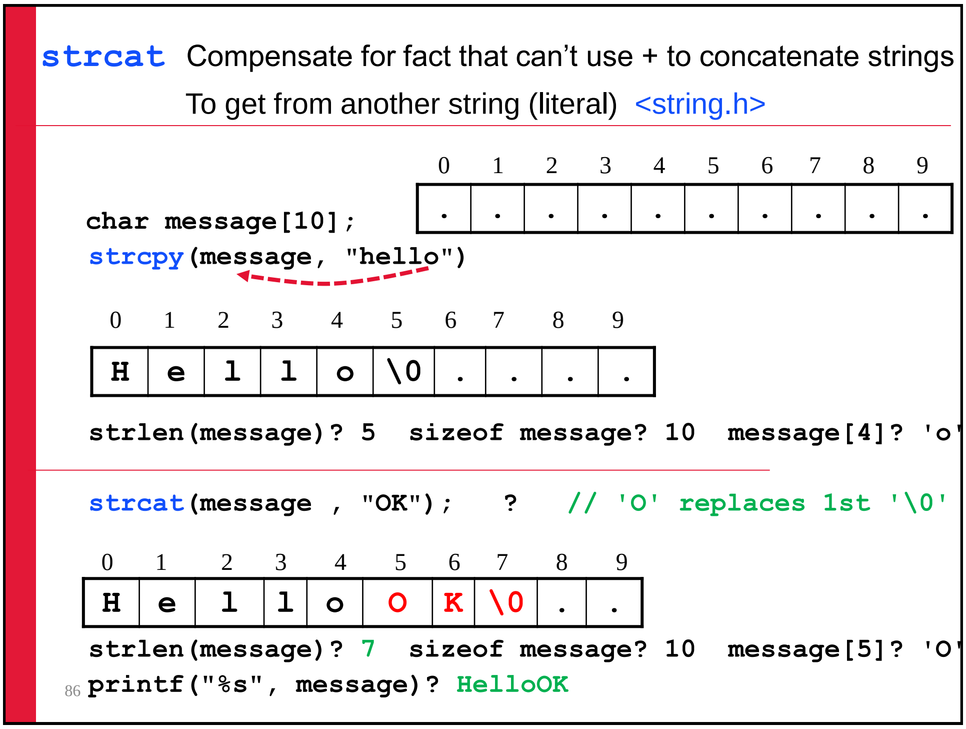
0	1	2	3	4	5	6	7	8	9
O	K	\0	l	o	\0	.	.	.	.

```
strlen(message)? 2   sizeof message? 10   message[4]? 'o'
printf("%s", message)?  OK
```

operator

85

85



strcat Compensate for fact that can't use + to concatenate strings

To get from another string (literal) <string.h>

0	1	2	3	4	5	6	7	8	9
.	.	.	.	.	.	.	.	.	.

```
char message[10];
strcpy(message, "hello")
```

0	1	2	3	4	5	6	7	8	9
H	e	l	l	o	\0	.	.	.	.

```
strlen(message)? 5   sizeof message? 10   message[4]? 'o'
```

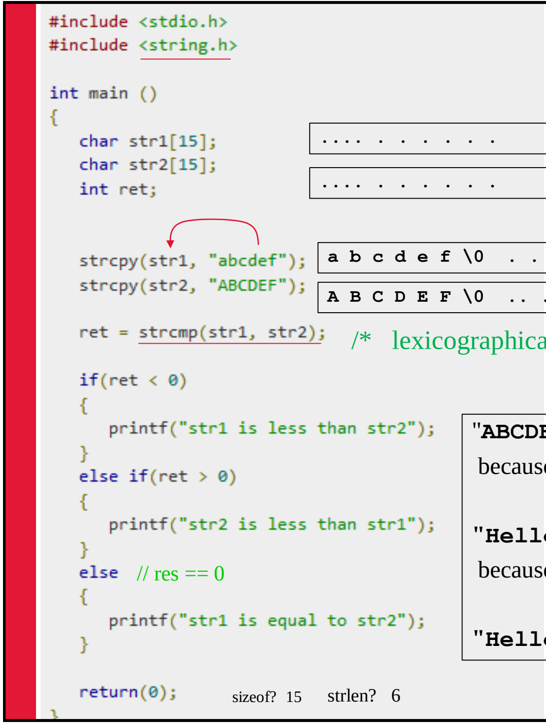
```
strcat(message , "OK");    ?   // 'O' replaces 1st '\0'
```

0	1	2	3	4	5	6	7	8	9
H	e	l	l	o	O	K	\0	.	.

```
strlen(message)? 7   sizeof message? 10   message[5]? 'O'
printf("%s", message)? HelloOK
```

86

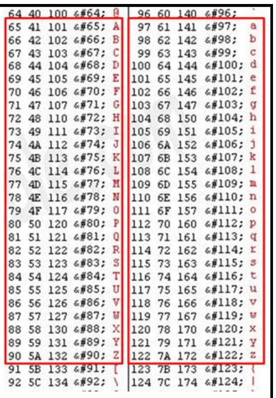
86



```c
#include <stdio.h>
#include <string.h>

int main ()
{
   char str1[15];
   char str2[15];
   int ret;


   strcpy(str1, "abcdef");
   strcpy(str2, "ABCDEF");

   ret = strcmp(str1, str2);   /*  lexicographical order  */

   if(ret < 0)
   {
      printf("str1 is less than str2");
   }
   else if(ret > 0)
   {
      printf("str2 is less than str1");
   }
   else   // res == 0
   {
      printf("str1 is equal to str2");
   }

   return(0);
}
```

```
. . . .  .  .  .  .  .  .
. . . .  .  .  .  .  .  .
a b c d e f \0  . . .. .
A B C D E F \0  .. . . .
```

sizeof? 15   strlen? 6



```
64 40 100 &#64; @   96 60 140 &#96;  `
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
91 5B 133 &#91; [  123 7B 173 &#123; {
92 5C 134 &#92; \  124 7C 174 &#124; |
```

"**ABCDEF**" is less than (<, precedes) "**abcdef**" because 'A' precedes 'a' in ASCII

"**Hellothere**" > "**HelloWorld**" because 't' located after 'W' in ASCII

"**Hello!**" equals "**Hello!**"

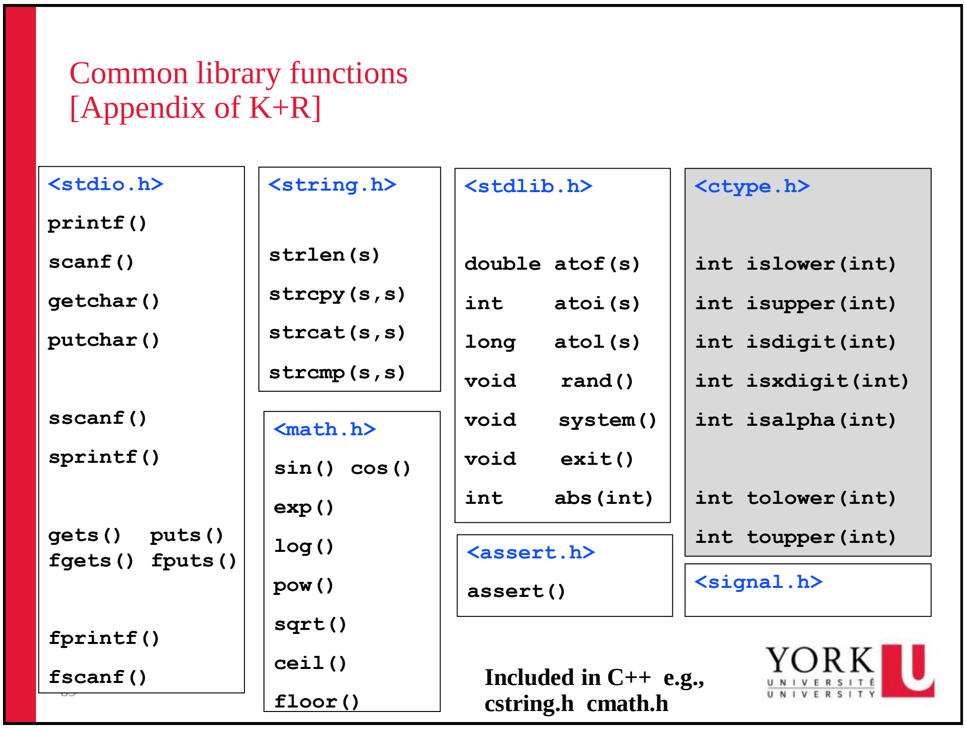
Same as Java s.compareTo(s2)

```
int strcmp(s1, s2);
0 if equal      <0 if s1<s2,      >0 if s1>s2
```

```c
int isQuit (char arr[]){

 int i;
 if (arr[0]=='q' && arr[1]=='u' && arr[2]=='i' &&
arr[3]=='t' && arr[4]=='\0' )
    return 1;
 else return 0; }
```

```c
isQuit(char arr[]){

  if ( strcmp(arr, "quit") == 0 )
    return 1;            // equal
  else return 0
}
```

```c
while ( strcmp (arr, "quit") !=0 )
  …..
```



Common library functions [Appendix of K+R]

```
<stdio.h>
printf()
scanf()
getchar()
putchar()

sscanf()
sprintf()

gets()   puts()
fgets()  fputs()

fprintf()
fscanf()
```

```
<string.h>

strlen(s)
strcpy(s,s)
strcat(s,s)
strcmp(s,s)
```

```
<math.h>
sin()  cos()
exp()
log()
pow()
sqrt()
ceil()
floor()
```

```
<stdlib.h>

double  atof(s)
int     atoi(s)
long    atol(s)
void    rand()
void    system()
void    exit()
int     abs(int)
```

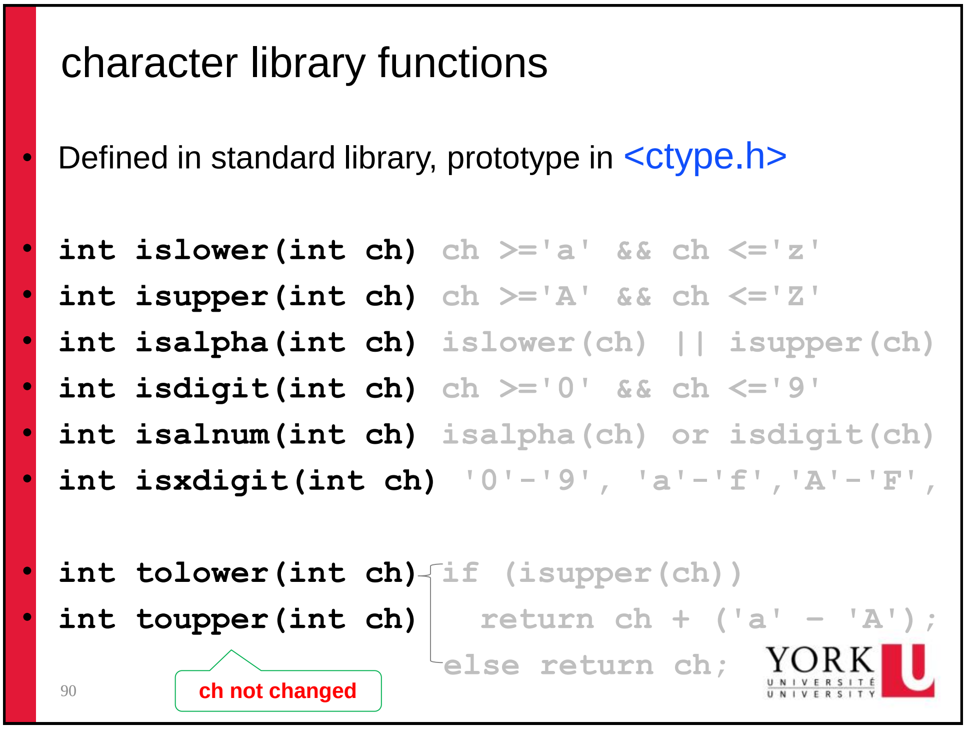
```
<assert.h>
assert()
```

```
<ctype.h>

int islower(int)
int isupper(int)
int isdigit(int)
int isxdigit(int)
int isalpha(int)

int tolower(int)
int toupper(int)
```

```
<signal.h>
```

Included in C++ e.g., cstring.h cmath.h

89



character library functions

- Defined in standard library, prototype in <ctype.h>

```
int islower(int ch)   ch >='a'  && ch <='z'
int isupper(int ch)   ch >='A'  && ch <='Z'
int isalpha(int ch)   islower(ch)  || isupper(ch)
int isdigit(int ch)   ch >='0'  && ch <='9'
int isalnum(int ch)   isalpha(ch) or isdigit(ch)
int isxdigit(int ch)   '0'-'9', 'a'-'f','A'-'F',

int tolower(int ch)   if  (isupper(ch))
int toupper(int ch)      return ch + ('a' - 'A');
                      else return ch;
```

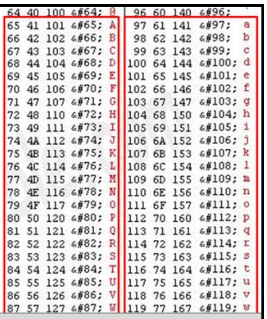
ch not changed

90

90

Example

```
#include<stdio.h>

/*copying input to output with
converting upper-case to lower-case letters */
main(){
    int c;
    c= getchar();
    while (c != EOF)
    {
        if (c >= 'A' && c <= 'Z’)
            c +=  'a'- 'A’;

        putchar(c);

        c = getchar();
    }
    return 0;
}
```



64	40	100	&#64;	@	96	60	140	&#96;	`
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

```
    c= getchar();
    while (c != EOF)
    {
        if (isupper(c))
            c = tolower(c);

        putchar(c);

        c = getchar();
    }
    return 0;
}
```

Common library functions
[Appendix of K+R]

```
<stdio.h>
printf()
scanf()

getchar()
putchar()
getc()
putc()

sscanf()
sprintf()

gets()   puts()
fgets()  fputs()
```

```
<string.h>

strlen(s)
strcpy(s,s)
strcat(s,s)
strcmp(s,s)
```

```
<math.h>
sin()  cos()
exp()
log()
pow()
sqrt()
ceil()
floor()
```

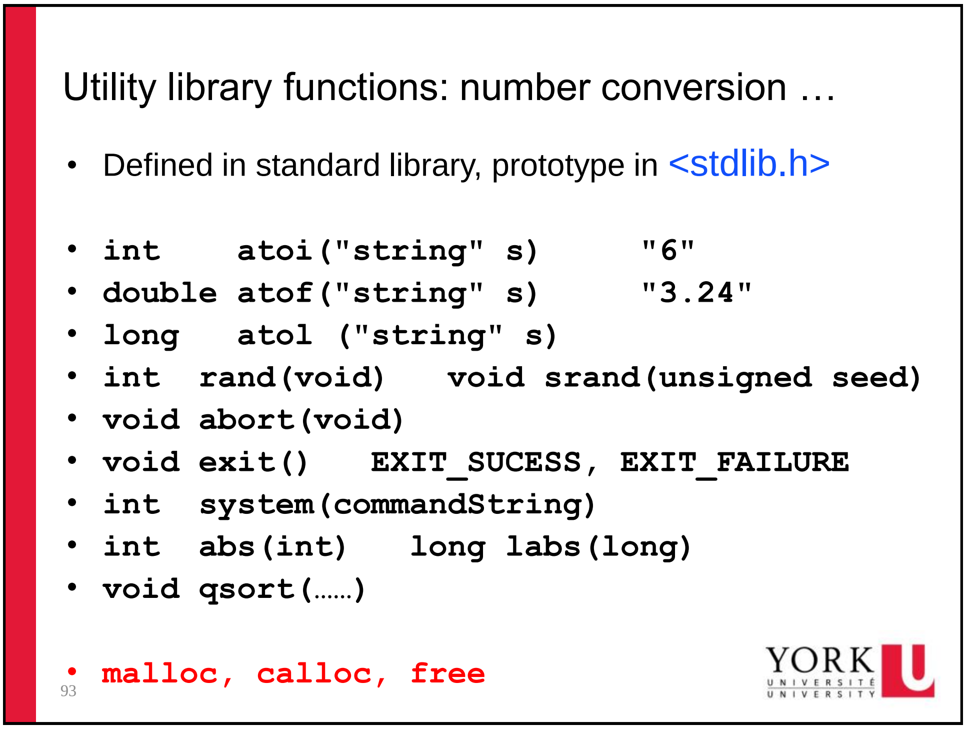
```
<stdlib.h>

int     atoi(s)
double  atof(s)
long    atol(s)
void    rand()
void    system()
void    exit()
int     abs(int)
```

```
<assert.h>
assert()
```

```
<ctype.h>

int islower(int)
int isupper(int)
int isdigit(int)
int isxdigit(int)
int isalpha(int)

int tolower(int)
int toupper(int)
```

```
<signal.h>
```

**Included in C++ e.g.,
cstring.h cmath.h**



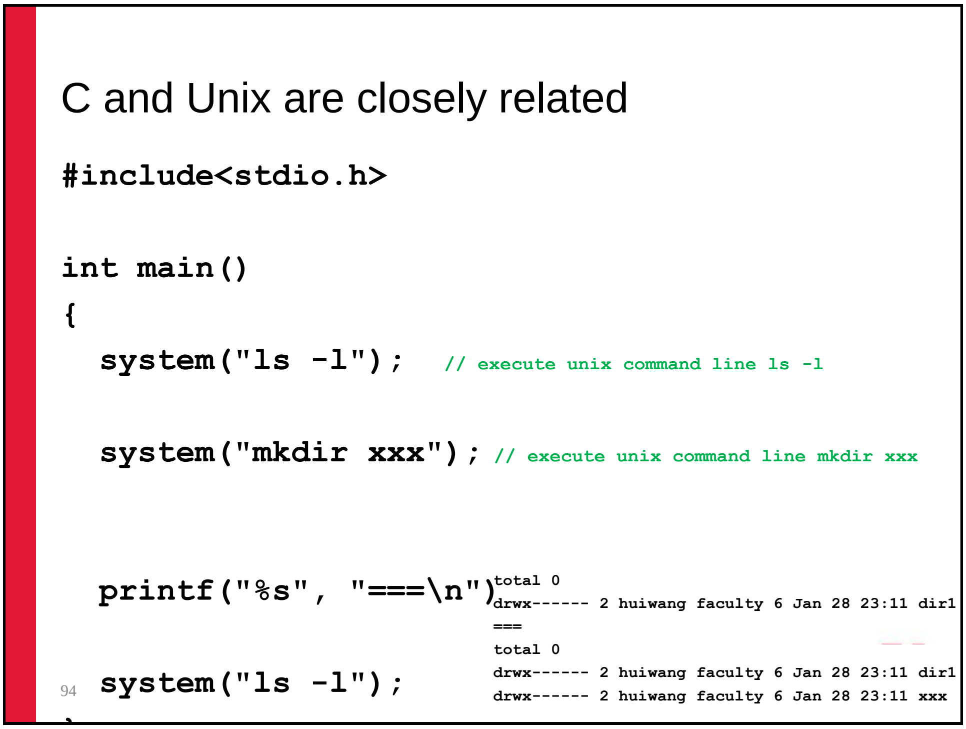
Utility library functions: number conversion …

- Defined in standard library, prototype in <stdlib.h>

```
• int    atoi("string" s)      "6"
• double atof("string" s)      "3.24"
• long   atol ("string" s)
• int  rand(void)   void srand(unsigned seed)
• void abort(void)
• void exit()   EXIT_SUCESS, EXIT_FAILURE
• int  system(commandString)
• int  abs(int)   long labs(long)
• void qsort(……)

• malloc, calloc, free
```

93

93



C and Unix are closely related

```
#include<stdio.h>

int main()
{
  system("ls -l");   // execute unix command line ls -l

  system("mkdir xxx"); // execute unix command line mkdir xxx


  printf("%s", "===\n");

  system("ls -l");
```

```
total 0
drwx------ 2 huiwang faculty 6 Jan 28 23:11 dir1
===
total 0
drwx------ 2 huiwang faculty 6 Jan 28 23:11 dir1
drwx------ 2 huiwang faculty 6 Jan 28 23:11 xxx
```

94

94

```
void initializeHardware(void)
{  /* initialize hardware */
   if(connect_robot("/dev/ttyUSB0", 38400) == FALSE){
      fprintf(stderr,"unable to connect to robot\n");

     sprintf(buf, "aplay ./sounds/%s", sth_wrong.wav);
      system(buf); //system("aplay ./sounds/sth_wrong.wav")
                   // execute unix command aplay ./…wav
     exit(EXIT_FAILURE);
   }
   // else connected
  enableSonars();
  system("aplay ./sounds/wakingUp.wav");
}
```

aplay - command-line sound recorder and player for ALSA soundcard driver

Common library functions [Appendix of K+R]

<stdio.h>

- printf()
- scanf()
- getchar()
- putchar()
- sscanf()
- sprintf()
- gets() puts()
- fgets() fputs()
- fprintf()
- fscanf()

<string.h>

- strlen(s)
- strcpy(s,s)
- strcat(s,s)
- strcmp(s,s)

<math.h>

- sin() cos()
- exp()
- log()
- pow()
- sqrt()
- ceil()
- floor()

<stdlib.h>

- double atof(s)
- int atoi(s)
- long atol(s)
- void rand()
- void system()
- void exit()
- int abs(int)

<assert.h>

- assert()

<ctype.h>

- int islower(int)
- int isupper(int)
- int isdigit(int)
- int isxdigit(int)
- int isalpha(int)
- int tolower(int)
- int toupper(int)

<signal.h>

Included in C++ e.g., cstring.h cmath.h

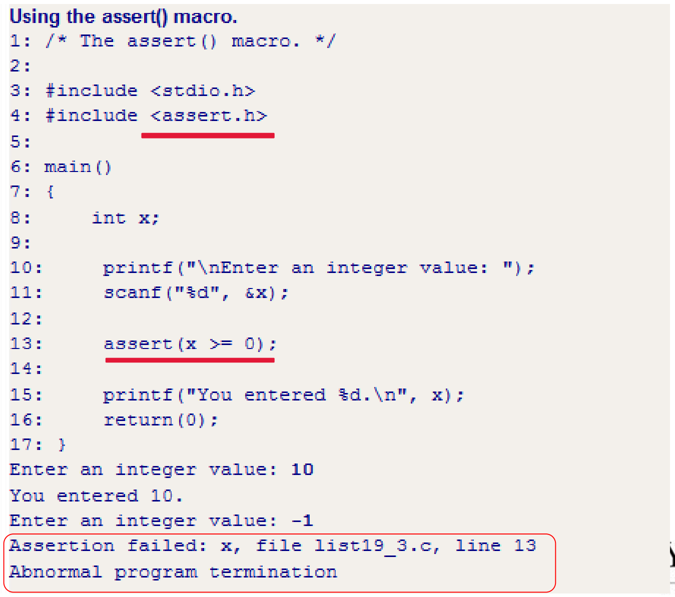
Diagnostics library functions

- Defined in standard library, prototype in <assert.h>
- **void assert(int *expression*)**

```
int x = -1;
assert(x > 0)
```

print Assertion failed: *expression*, file *file*, line *lnum*

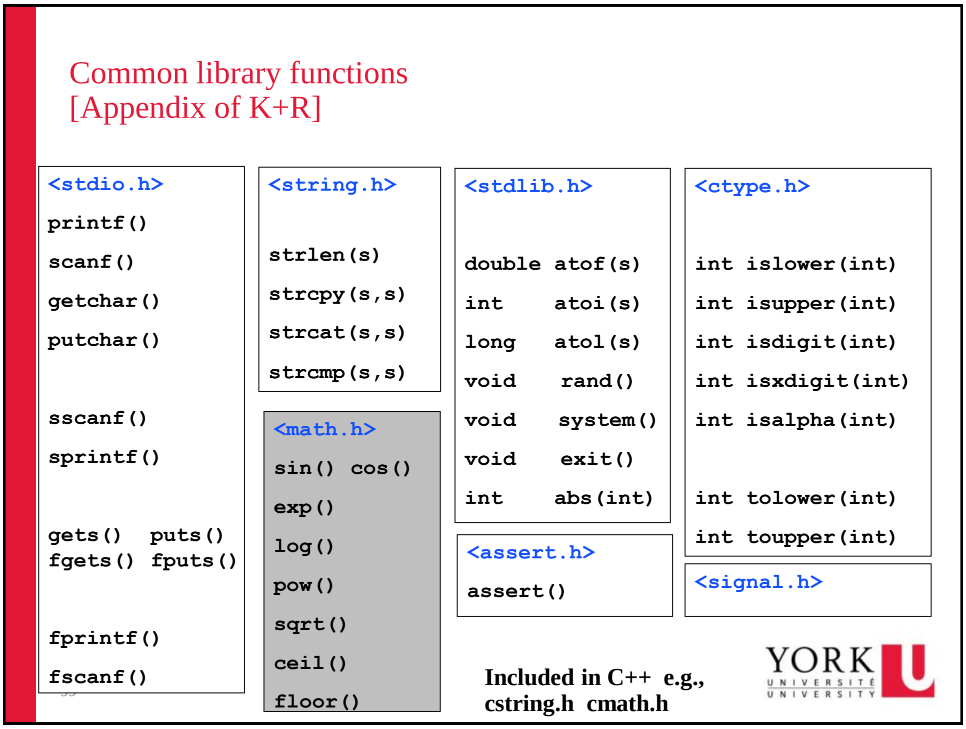
Then **abort()**



Using the assert() macro.

```
1: /* The assert() macro. */
2:
3: #include <stdio.h>
4: #include <assert.h>
5:
6: main()
7: {
8:     int x;
9:
10:     printf("\nEnter an integer value: ");
11:     scanf("%d", &x);
12:
13:     assert(x >= 0);
14:
15:     printf("You entered %d.\n", x);
16:     return(0);
17: }
```

```
Enter an integer value: 10
You entered 10.
Enter an integer value: -1
Assertion failed: x, file list19_3.c, line 13
Abnormal program termination
```



Common library functions [Appendix of K+R]

<stdio.h>

- printf()
- scanf()
- getchar()
- putchar()
- sscanf()
- sprintf()
- gets() puts()
- fgets() fputs()
- fprintf()
- fscanf()

<string.h>

- strlen(s)
- strcpy(s,s)
- strcat(s,s)
- strcmp(s,s)

<math.h>

- sin() cos()
- exp()
- log()
- pow()
- sqrt()
- ceil()
- floor()

<stdlib.h>

```
double  atof(s)
int     atoi(s)
long    atol(s)
void    rand()
void    system()
void    exit()
int     abs(int)
```

<assert.h>

- assert()

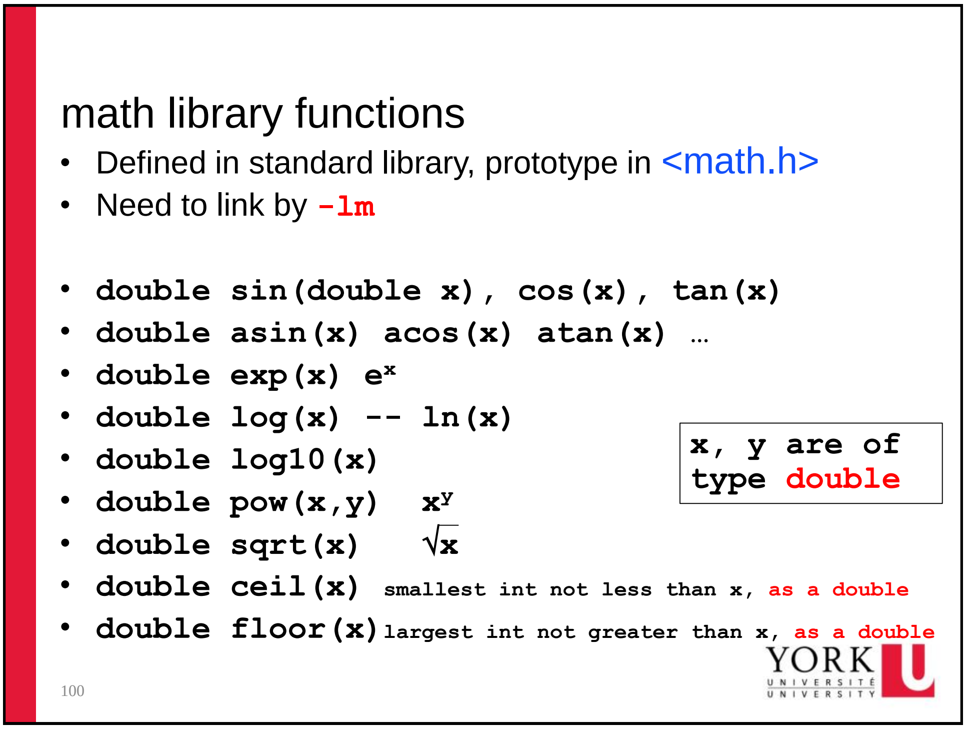
<ctype.h>

```
int islower(int)
int isupper(int)
int isdigit(int)
int isxdigit(int)
int isalpha(int)

int tolower(int)
int toupper(int)
```

<signal.h>

Included in C++ e.g., cstring.h cmath.h

99



math library functions

- Defined in standard library, prototype in <math.h>
- Need to link by **-lm**

- **double sin(double x), cos(x), tan(x)**
- **double asin(x) acos(x) atan(x)** …
- **double exp(x)** e^x
- **double log(x) -- ln(x)**
- **double log10(x)**
- **double pow(x,y)** x^y
- **double sqrt(x)** $\sqrt{x}$
- **double ceil(x)** smallest int not less than x, as a double
- **double floor(x)** largest int not greater than x, as a double

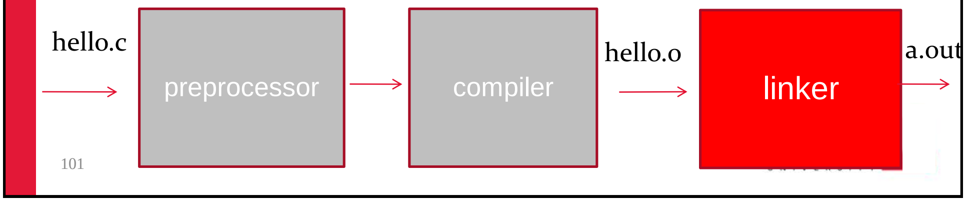
x, y are of type double

100

100

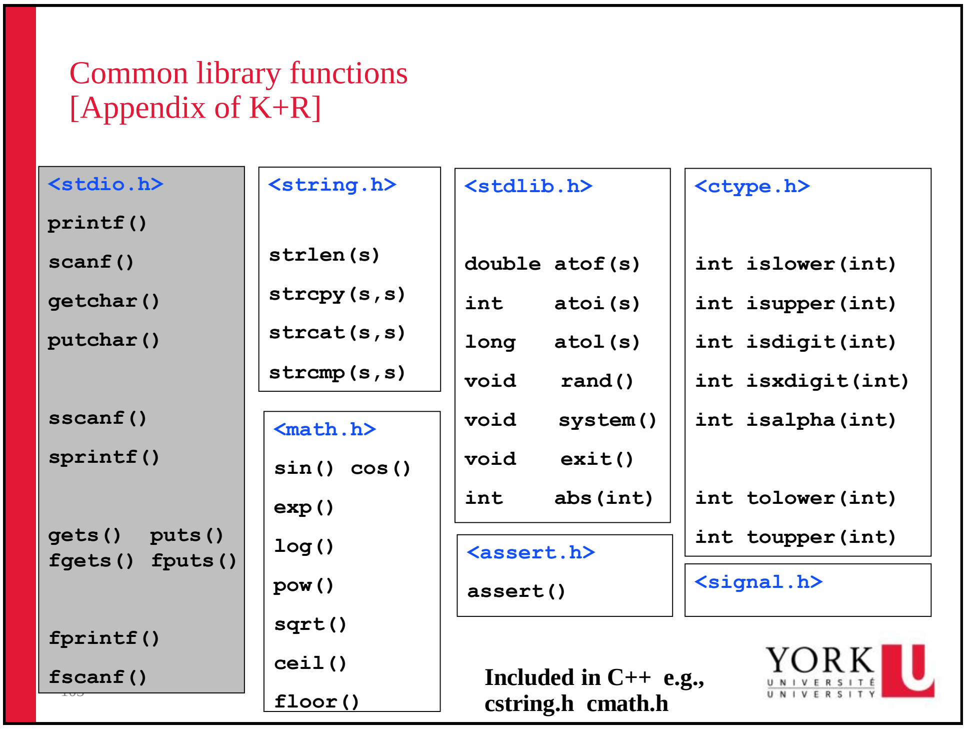
How C Programs are Compiled

- C programs go through three stages to be compiled:
  - Preprocessor - handles #include and #define etc
  - Compiler - converts C code into binary processor instructions (“object code”)
  - Linker - puts multiple files together, load library function (e.g. printf) and creates an executable program



Some complier options

- Option: -l*library*
  - Link with object library `gcc main.c -lm`
    - ○ Links math object library (if use `pow() ceil() sin()` etc)
    - ○ Don’t forget to use `#include <math.h>` at the beginning
  - `gcc main.c -lp`
    - ○ Links pthread …..



Common library functions [Appendix of K+R]

```
<stdio.h>
printf()
scanf()
getchar()
putchar()

sscanf()
sprintf()

gets()   puts()
fgets()  fputs()

fprintf()
fscanf()
```

```
<string.h>

strlen(s)
strcpy(s,s)
strcat(s,s)
strcmp(s,s)
```

```
<math.h>
sin()  cos()
exp()
log()
pow()
sqrt()
ceil()
floor()
```

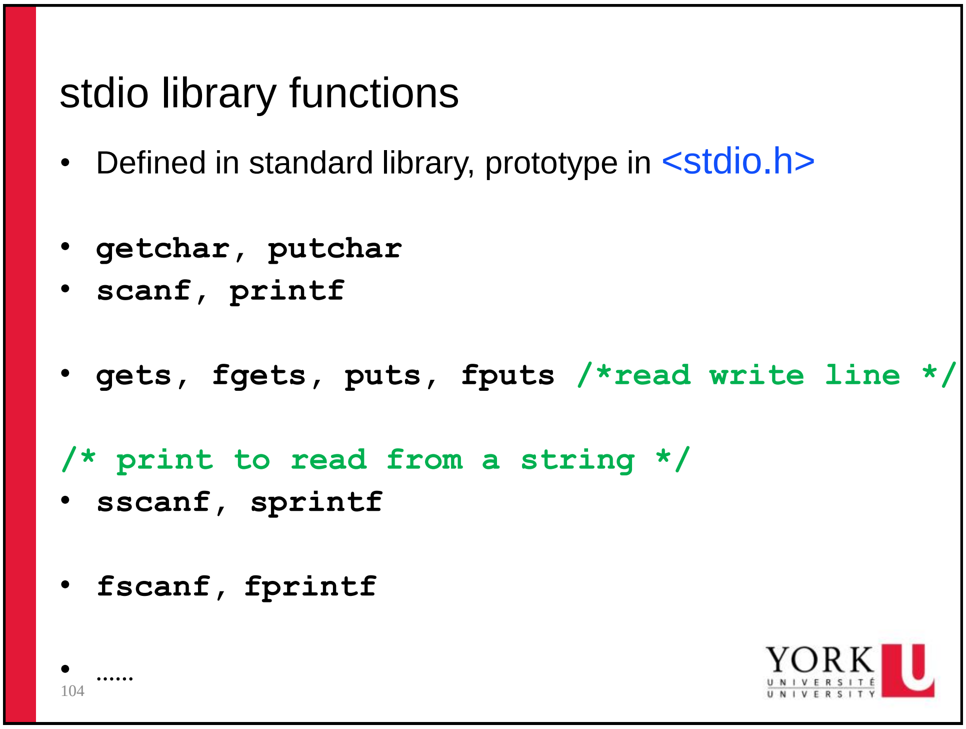
```
<stdlib.h>

double  atof(s)
int     atoi(s)
long    atol(s)
void     rand()
void     system()
void     exit()
int     abs(int)
```

```
<assert.h>
assert()
```

```
<ctype.h>

int islower(int)
int isupper(int)
int isdigit(int)
int isxdigit(int)
int isalpha(int)

int tolower(int)
int toupper(int)
```

```
<signal.h>
```

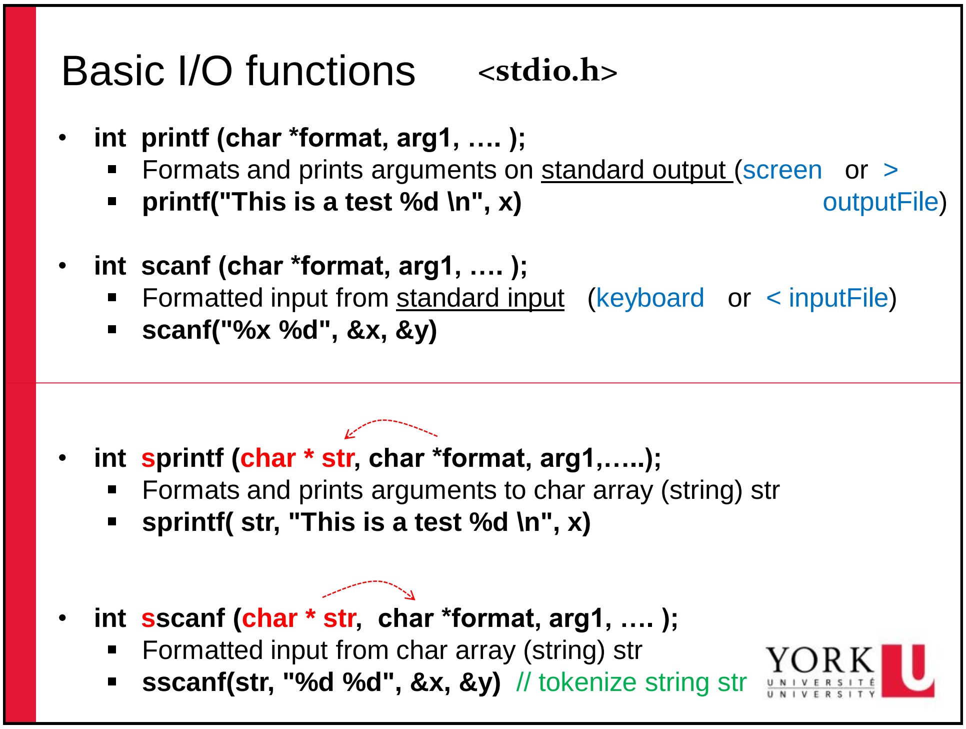
Included in C++ e.g.,
cstring.h cmath.h



stdio library functions

- Defined in standard library, prototype in <stdio.h>

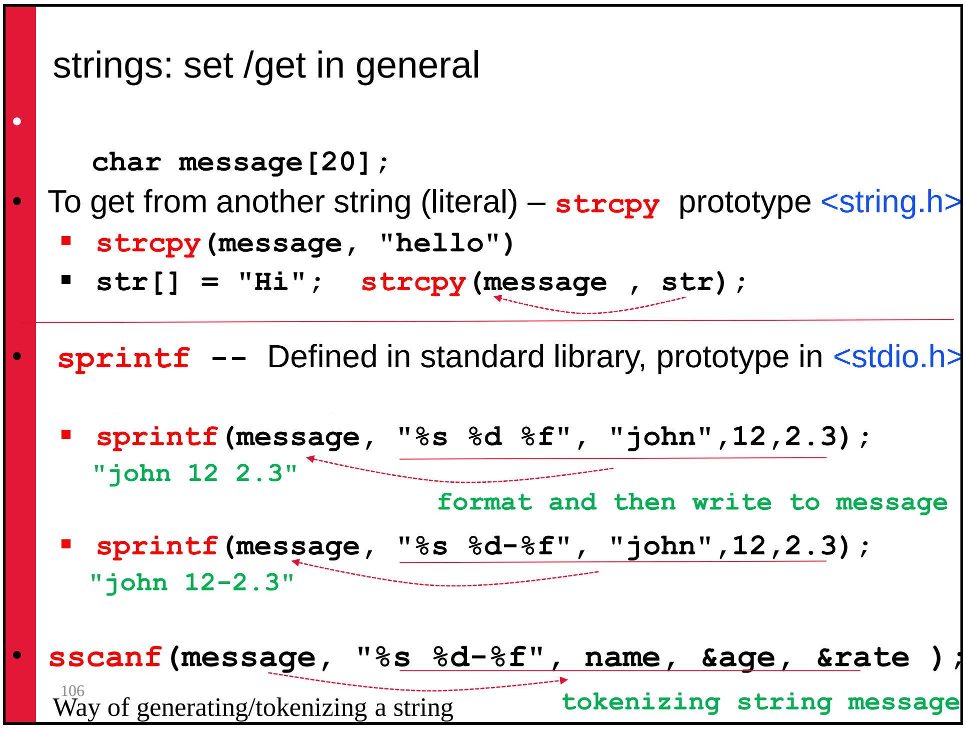
```
• getchar, putchar
• scanf, printf

• gets, fgets, puts, fputs /*read write line */

/* print to read from a string */
• sscanf, sprintf

• fscanf, fprintf

• ......
```



Basic I/O functions <stdio.h>

- int printf (char *format, arg1, ….);
  - Formats and prints arguments on standard output (screen or > outputFile)
  - printf("This is a test %d \n", x)
- int scanf (char *format, arg1, ….);
  - Formatted input from standard input (keyboard or < inputFile)
  - scanf("%x %d", &x, &y)

- int sprintf (char * str, char *format, arg1,…..);
  - Formats and prints arguments to char array (string) str
  - sprintf(str, "This is a test %d \n", x)
- int sscanf (char * str, char *format, arg1, ….);
  - Formatted input from char array (string) str
  - sscanf(str, "%d %d", &x, &y) // tokenize string str



strings: set /get in general

-

char message[20];

- To get from another string (literal) – strcpy prototype <string.h>
  - strcpy(message, "hello")
  - str[] = "Hi"; strcpy(message , str);

- sprintf -- Defined in standard library, prototype in <stdio.h>
  - sprintf(message, "%s %d %f", "john",12,2.3);

    "john 12 2.3"

    format and then write to message
  - sprintf(message, "%s %d-%f", "john",12,2.3);

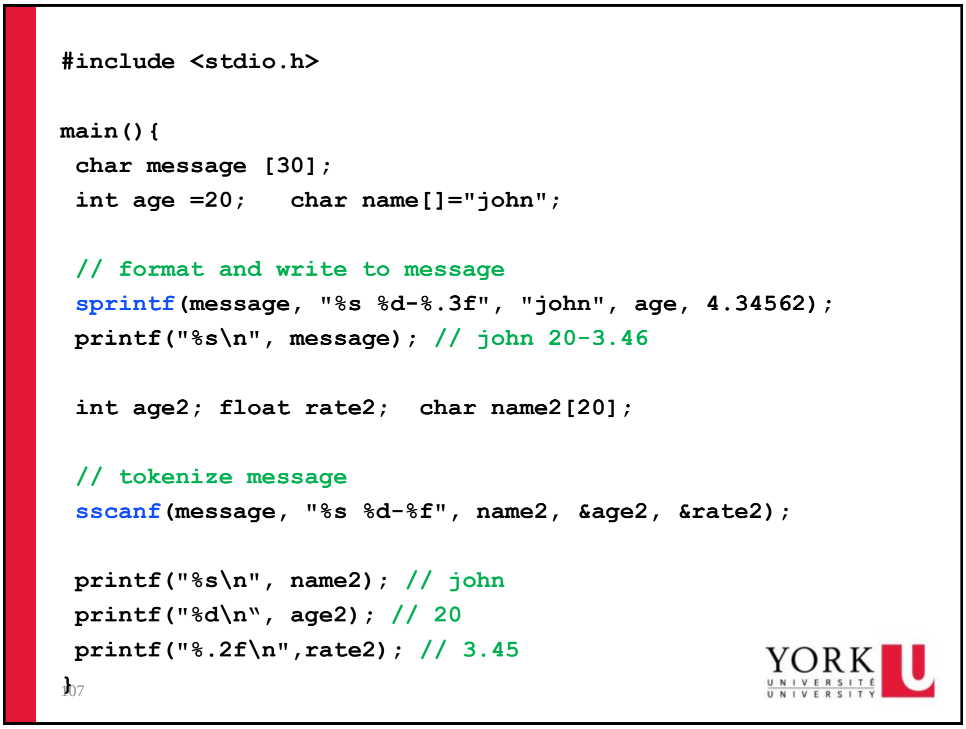
    "john 12-2.3"

- sscanf(message, "%s %d-%f", name, &age, &rate);

106

Way of generating/tokenizing a string

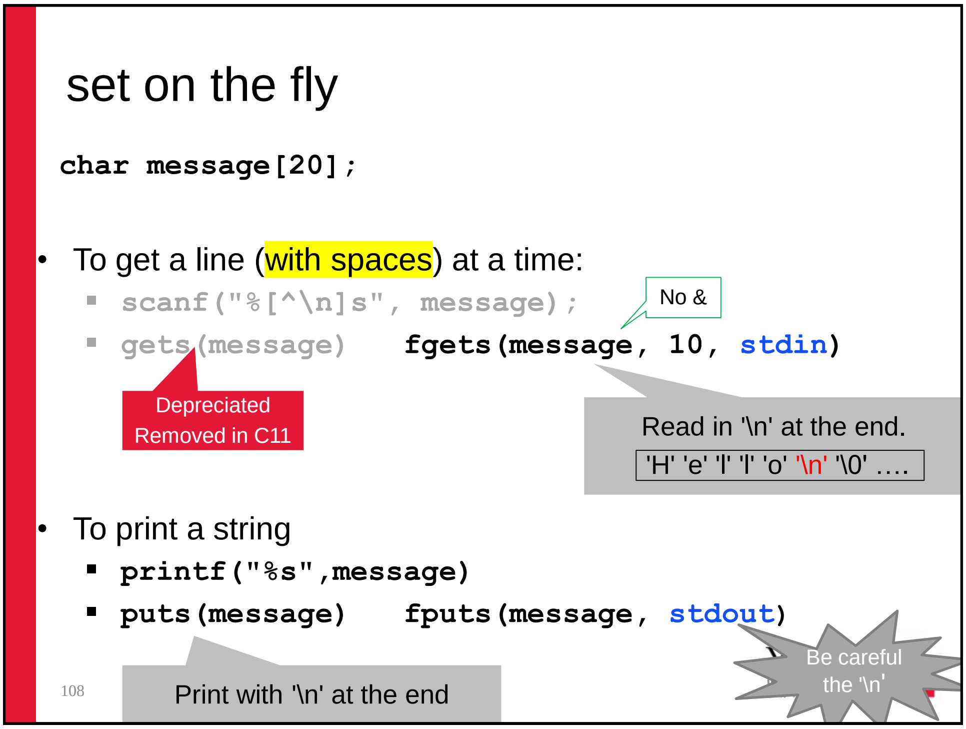
tokenizing string message



```
#include <stdio.h>

main(){
 char message [30];
 int age =20;   char name[]="john";

 // format and write to message
 sprintf(message, "%s %d-%.3f", "john", age, 4.34562);
 printf("%s\n", message); // john 20-3.46

 int age2; float rate2;  char name2[20];

 // tokenize message
 sscanf(message, "%s %d-%f", name2, &age2, &rate2);

 printf("%s\n", name2); // john
 printf("%d\n", age2); // 20
 printf("%.2f\n",rate2); // 3.45
}
```

107

107



set on the fly

`char message[20];`

- To get a line (with spaces) at a time:
  - `scanf("%[^\n]s", message);`
  - `gets(message)`   `fgets(message, 10, stdin)`

No &

Depreciated
Removed in C11

Read in '\n' at the end.
'H' 'e' 'l' 'l' 'o' '\n' '\0' ….

- To print a string
  - `printf("%s",message)`
  - `puts(message)`   `fputs(message, stdout)`

Print with '\n' at the end

Be careful
the '\n'

108

108

```
int main()
{
   char str[40];
   fgets (str, 40, stdin);

/* write content to stdout */
   fputs(str, stdout);
   //printf("s", str);
 }
```

No \n needed

```
red 199 % a.out
hello the world!
hello the world!
red 200 %
```

```
int main()
{
   char str[40];
   while (1)
   {
      fgets (str, 40, stdin);
      if (! strcmp(str, "quit\n"))
         break;              // ==0
      fputs(str, stdout);
   }
}
```

No &

Be careful the '\n'

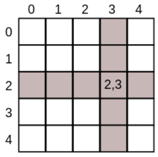
```
red 199 % a.out
hello the world!
hello the world!
This is good
This is good
quit
red 200 %
```

```
void initializeHardware(void)
{  /* initialize hardware */
   if(connect_robot("/dev/ttyUSB0", 38400) == FALSE){
      fprintf(stderr,"unable to connect to robot\n");

      sprintf(buf, "aplay ./sounds/%s", sth_wrong.wav);
      system(buf); //buf: "aplay ./sounds/sth_wrong.wav"

      exit(EXIT_FAILURE);
   }
   // else connected
   enableSonars();
   system("aplay ./sounds/wakingUp.wav");
}
```

aplay - command-line sound recorder and player for ALSA soundcard driver

- Finished Ch1 – 4

- Other C materials before pointer
  - Common library functions [Appendix of K+R]
  - **2D array, string manipulations**



Multi-dimension array, array of strings

- int arr2D [3][2]; // 3 row, 2 column

	0	1
0	**a[0][0]**	**a[0][1]**
1	**a[1][0]**	**a[1][1]**
2	**a[2][0]**	**a[2][1]**

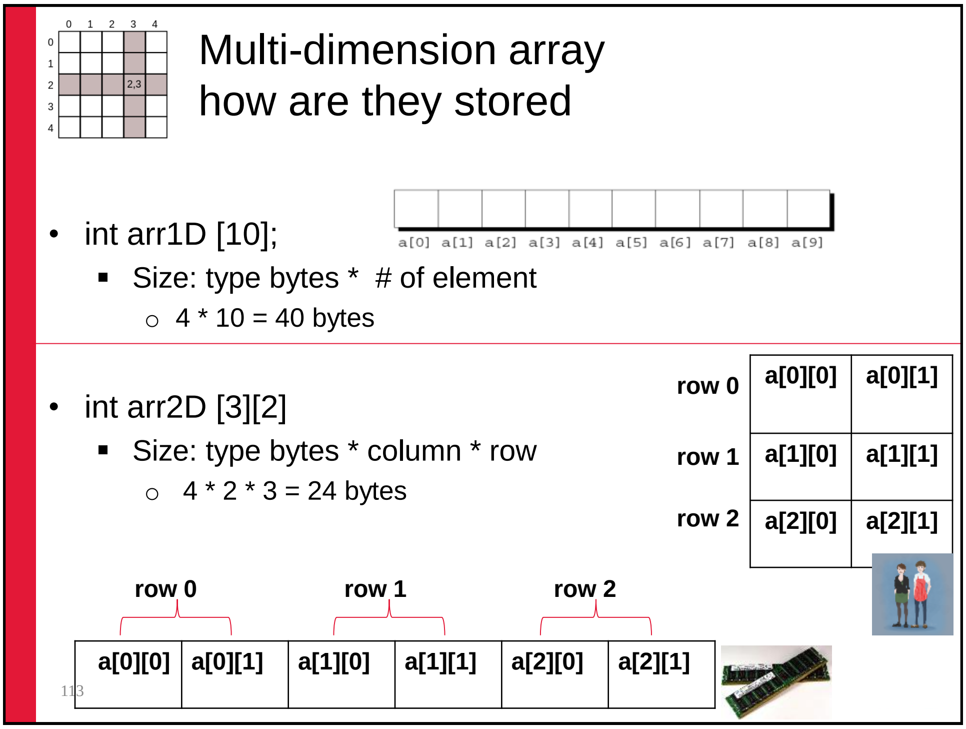
- Initialization:
  - int arr2D [3][2] = {1,1,2,4,3,9};
  - int arr2D [3][2] = {{1,1},{2,4},{3,9}}

	0	1
0	**1**	**1**
1	**2**	**4**
2	**3**	**9**

- Access: arry[2][1] = ?   9

- size? How stored?

Same in Java



0 1 2 3 4
0
1
2
3
4
2,3

Multi-dimension array
how are they stored

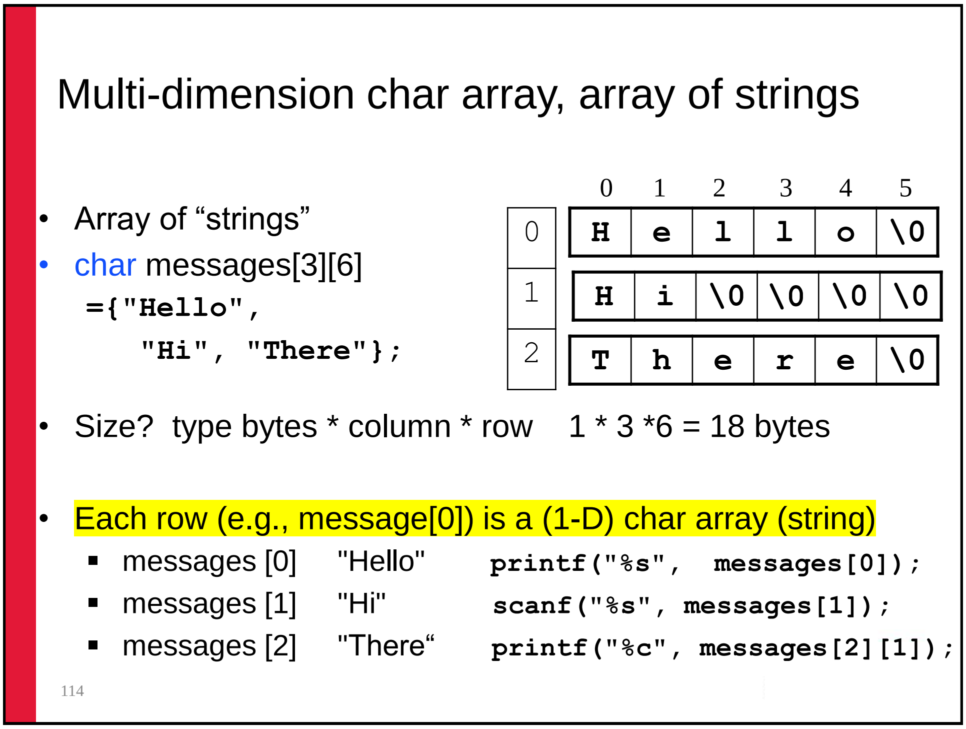
- int arr1D [10];
  - Size: type bytes * # of element
    - 4 * 10 = 40 bytes

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

- int arr2D [3][2]
  - Size: type bytes * column * row
    - 4 * 2 * 3 = 24 bytes

row 0	a[0][0]	a[0][1]
row 1	a[1][0]	a[1][1]
row 2	a[2][0]	a[2][1]

row 0		row 1		row 2	
a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]



Multi-dimension char array, array of strings

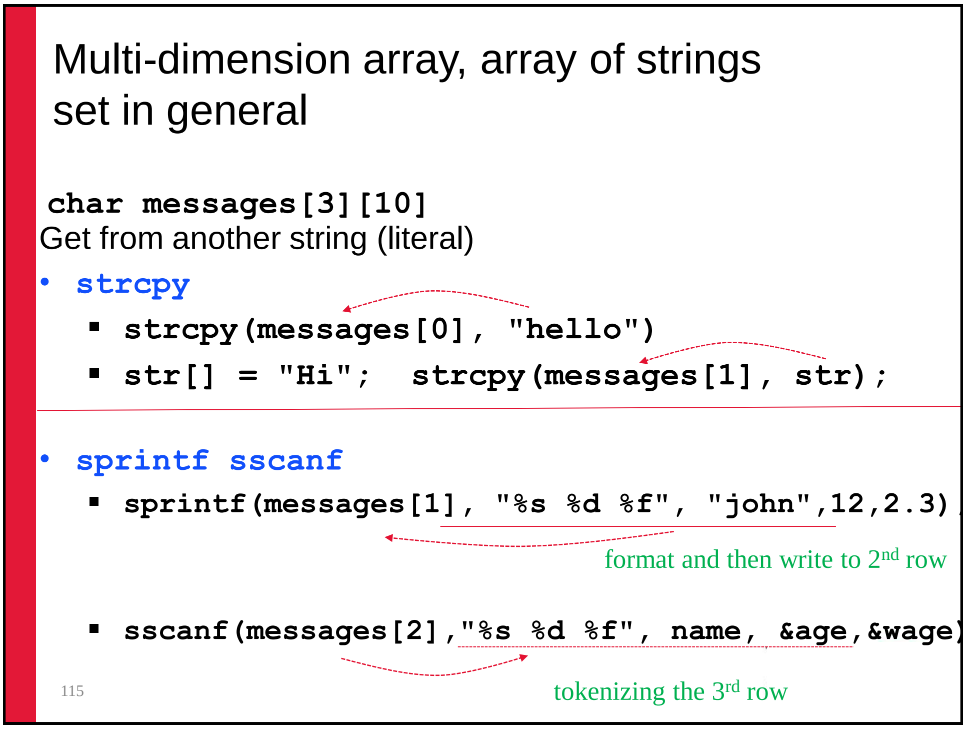
- Array of “strings”
- char messages[3][6]

```
 ={"Hello",
     "Hi",  "There"};
```

	0	1	2	3	4	5
0	H	e	l	l	o	\0
1	H	i	\0	\0	\0	\0
2	T	h	e	r	e	\0

- Size? type bytes * column * row 1 * 3 *6 = 18 bytes

- Each row (e.g., message[0]) is a (1-D) char array (string)
  - messages [0] "Hello" `printf("%s", messages[0]);`
  - messages [1] "Hi" `scanf("%s", messages[1]);`
  - messages [2] "There“ `printf("%c", messages[2][1]);`



Multi-dimension array, array of strings set in general

`char messages[3][10]`

Get from another string (literal)

- `strcpy`
  - **`strcpy(messages[0], "hello")`**
  - **`str[] = "Hi";  strcpy(messages[1], str);`**

- `sprintf sscanf`
  - **`sprintf(messages[1], "%s %d %f", "john",12,2.3)`**

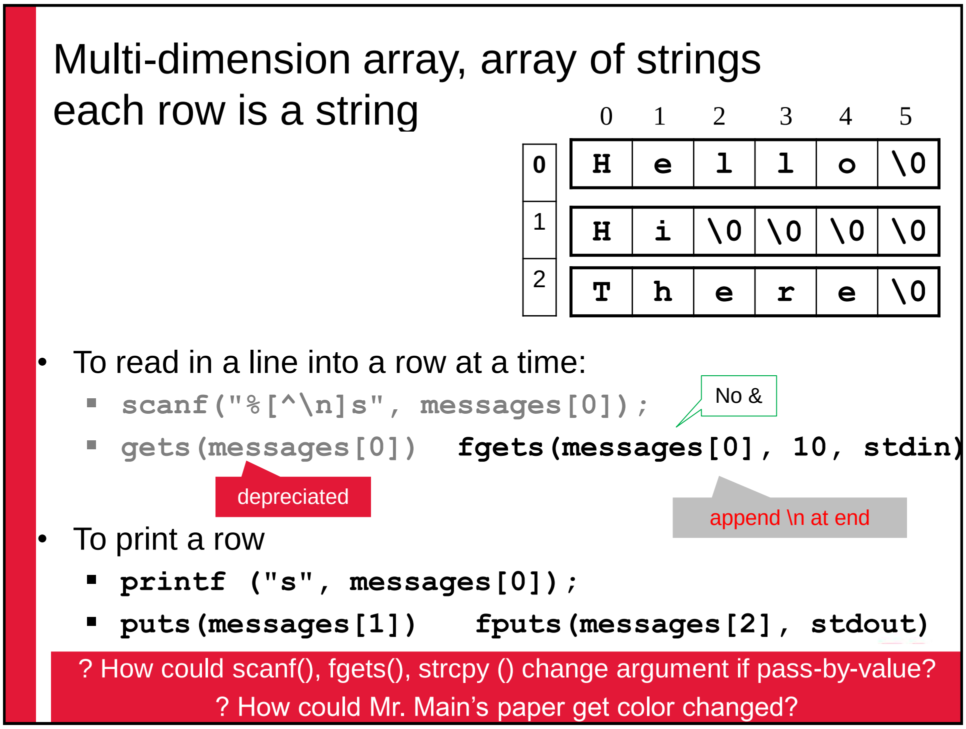
    format and then write to 2nd row

  - **`sscanf(messages[2],"%s %d %f", name, &age,&wage)`**

    tokenizing the 3rd row

115

115



Multi-dimension array, array of strings each row is a string

	0	1	2	3	4	5
0	H	e	l	l	o	\0
1	H	i	\0	\0	\0	\0
2	T	h	e	r	e	\0

- To read in a line into a row at a time:
  - `scanf("%[^\n]s", messages[0]);` No &
  - `gets(messages[0])` **`fgets(messages[0], 10, stdin)`**

    depreciated

    append \n at end

- To print a row
  - **`printf ("s", messages[0]);`**
  - **`puts(messages[1])` `fputs(messages[2], stdout)`**

? How could scanf(), fgets(), strcpy () change argument if pass-by-value?

? How could Mr. Main's paper get color changed?

116

- Finished Ch1 – 4

- Other C materials before pointer
  - Common library functions [Appendix of K+R]
  - 2D array, string manipulations

- Now it is time to start POINTERS!!!