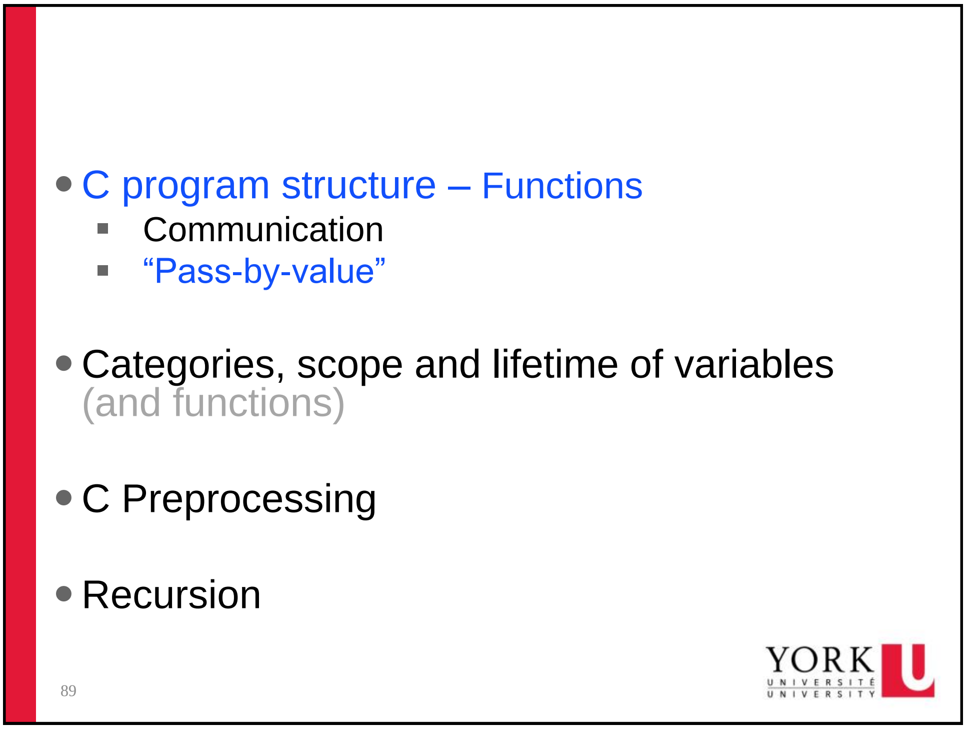
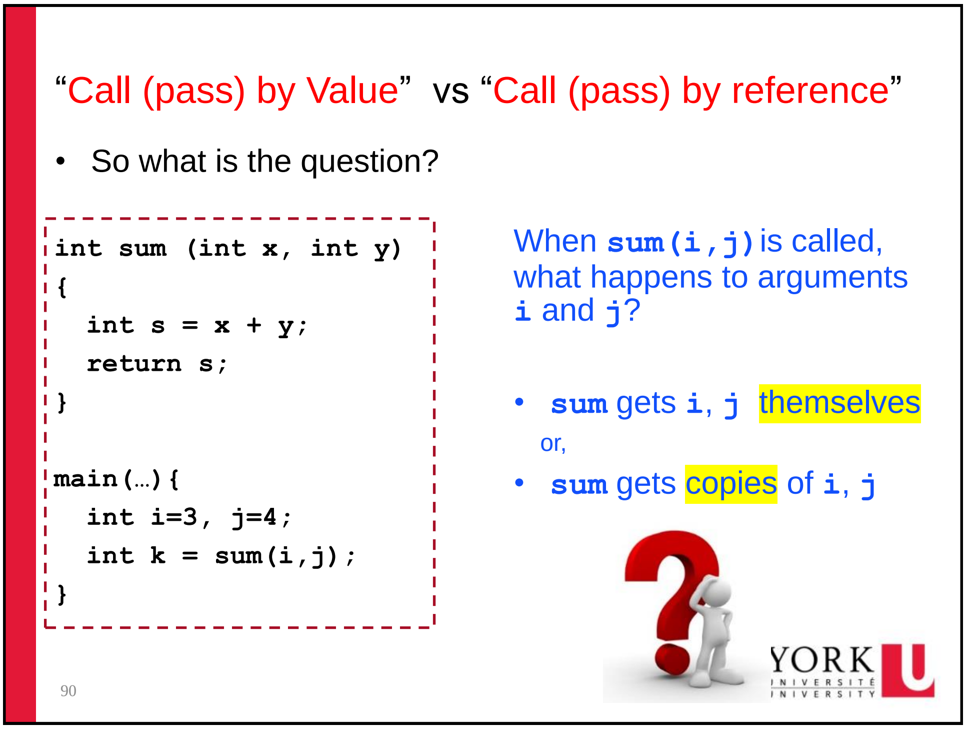


- C program structure – Functions
  - Communication
  - “Pass-by-value”
- Categories, scope and lifetime of variables (and functions)
- C Preprocessing
- Recursion



“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,

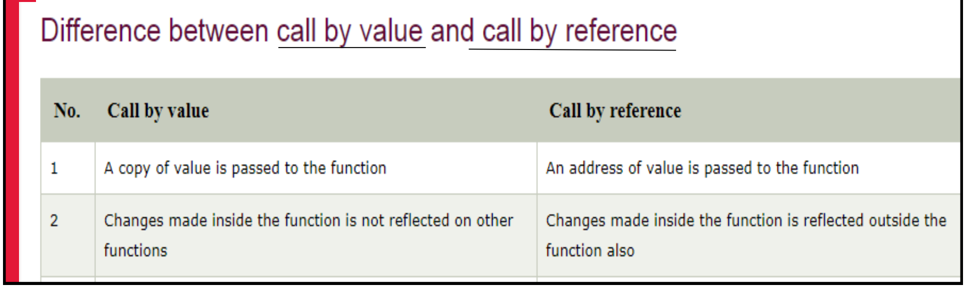
- `sum` gets copies of `i`, `j`

“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`



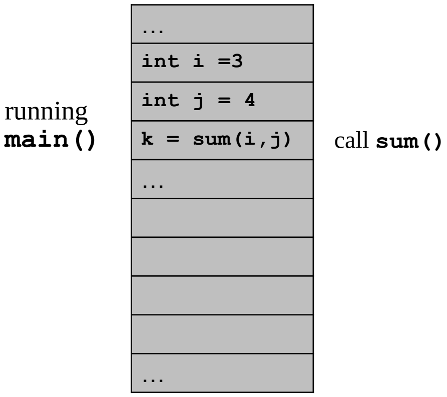
Difference between call by value and call by reference

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)

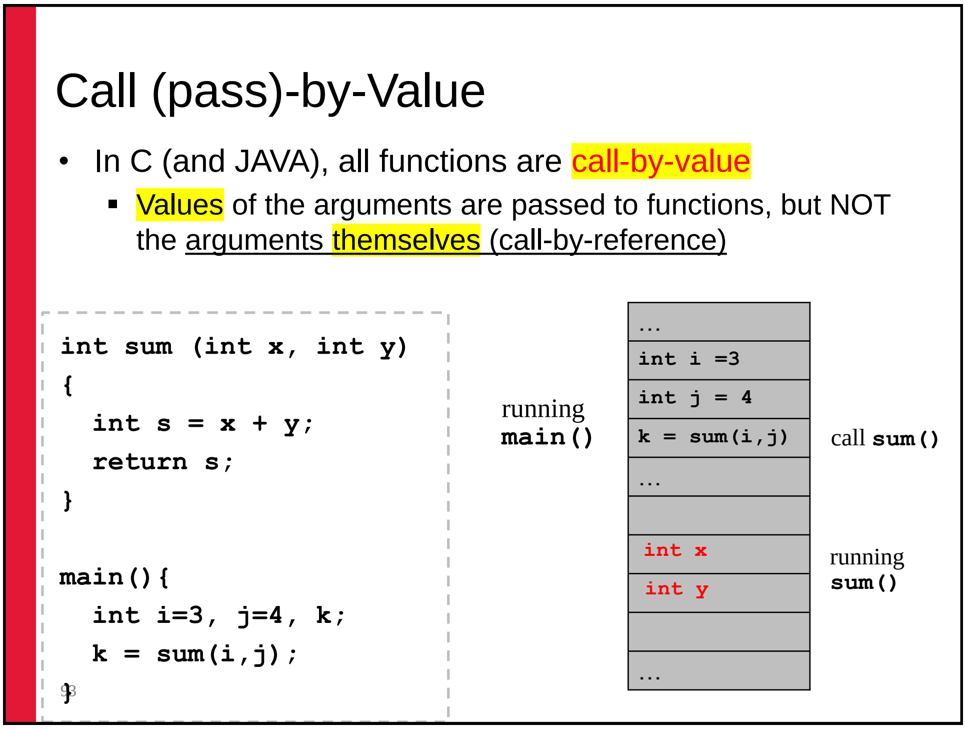
```
int sum (int x, int y)
{
   int s = x + y;
   return s;
}

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

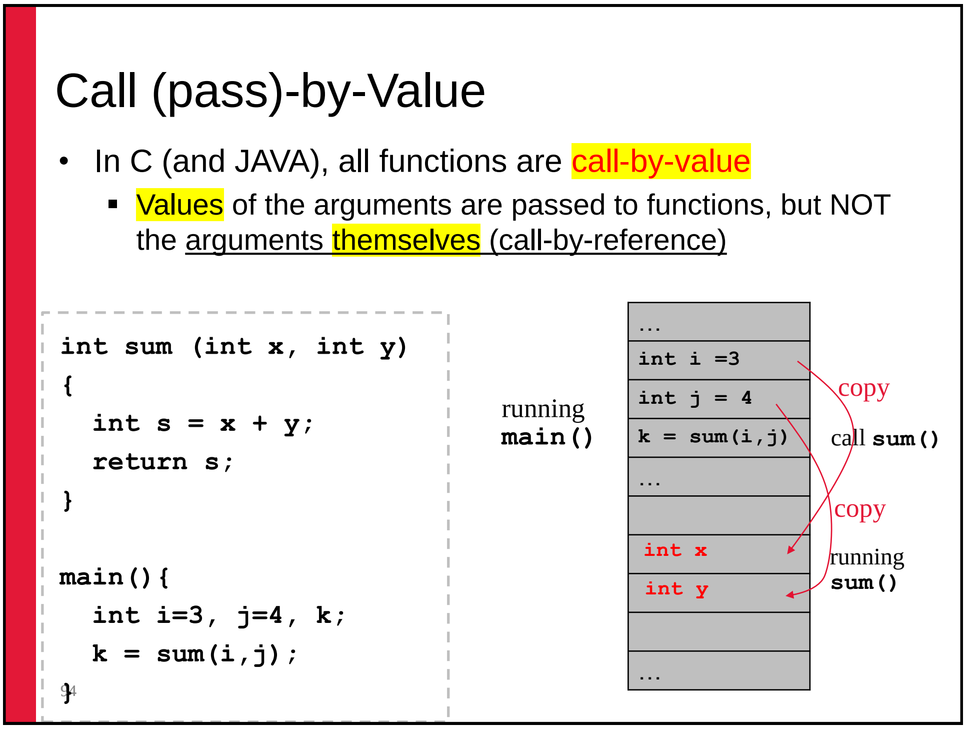
- In C (and JAVA), all functions are call-by-value
  - Values of the arguments are passed to functions, but NOT the arguments themselves (call-by-reference)

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

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

93



Call (pass)-by-Value

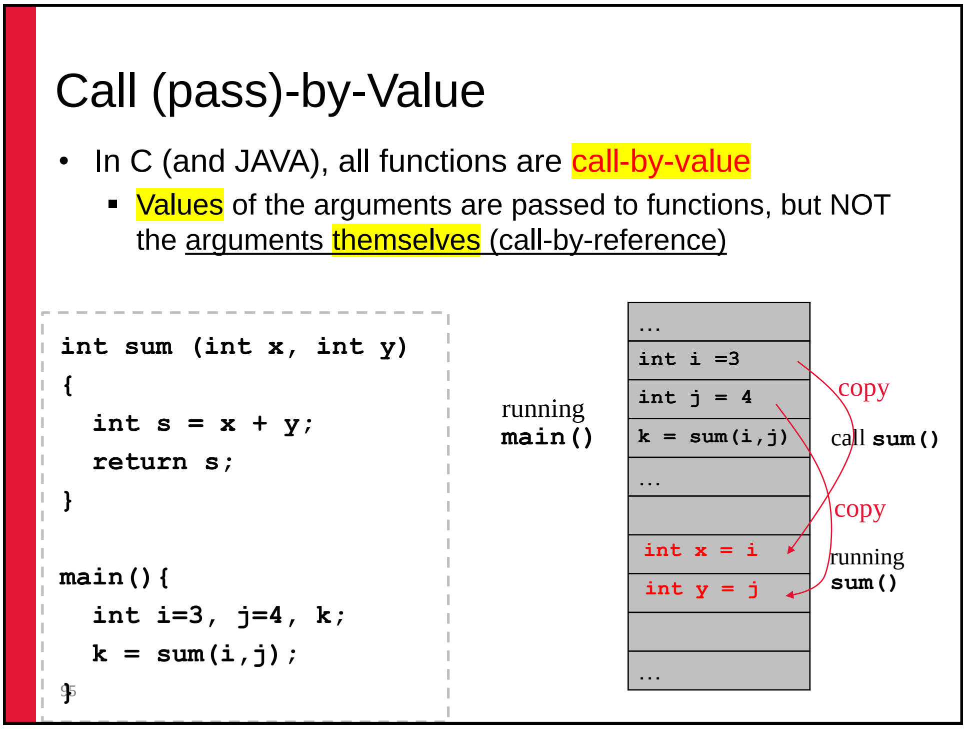
- In C (and JAVA), all functions are call-by-value
  - Values of the arguments are passed to functions, but NOT the arguments themselves (call-by-reference)

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

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

94

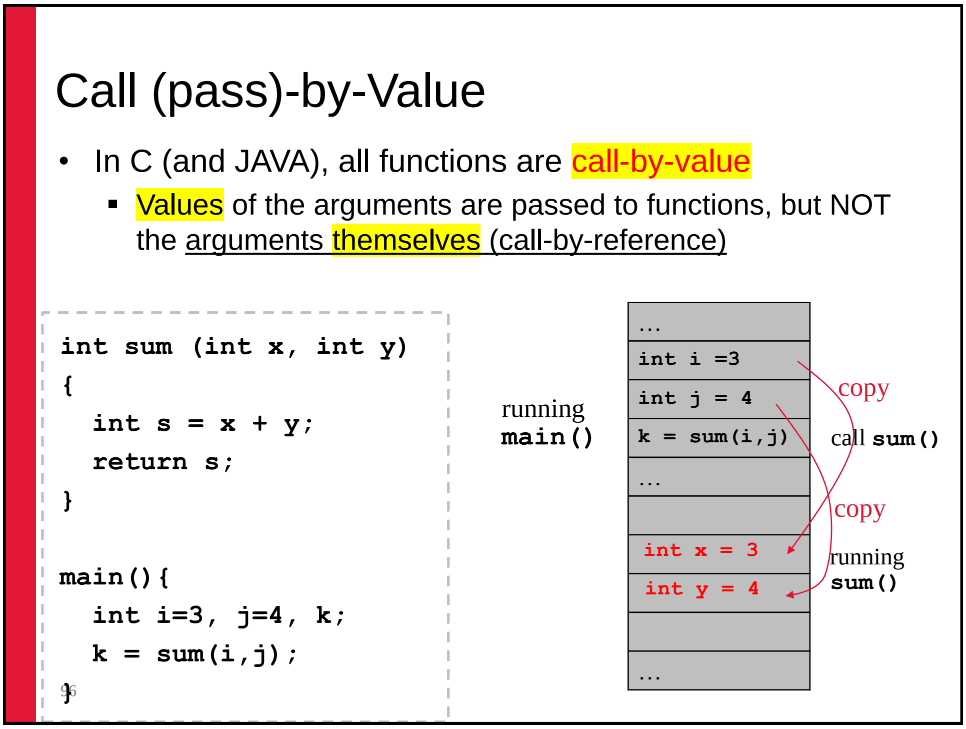


Call (pass)-by-Value

- 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);
}
```

95

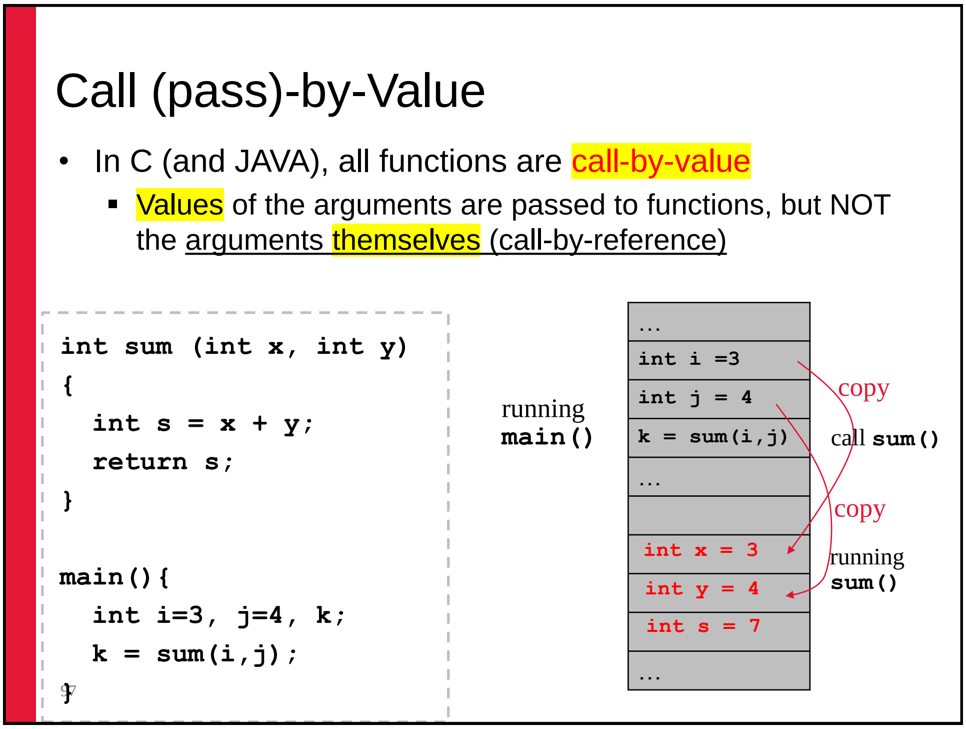


Call (pass)-by-Value

- 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);
}
```

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);
}
```

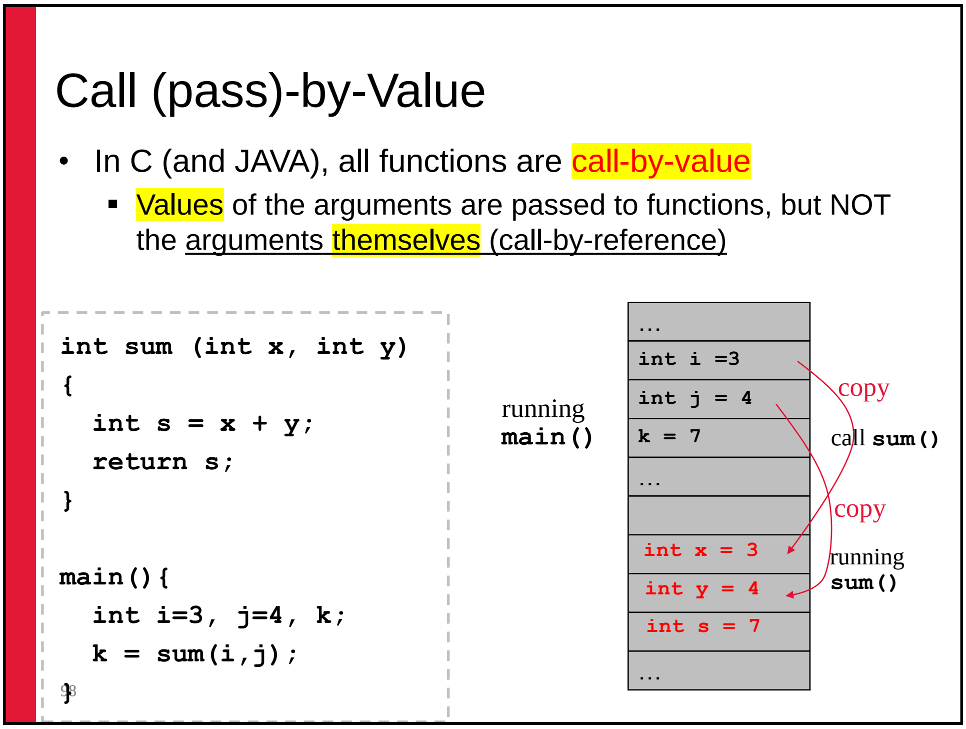
running
main()

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

int x = 3
int y = 4
int s = 7
…
```

copy
call **sum()**
copy
running
sum()

97

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);
}
```

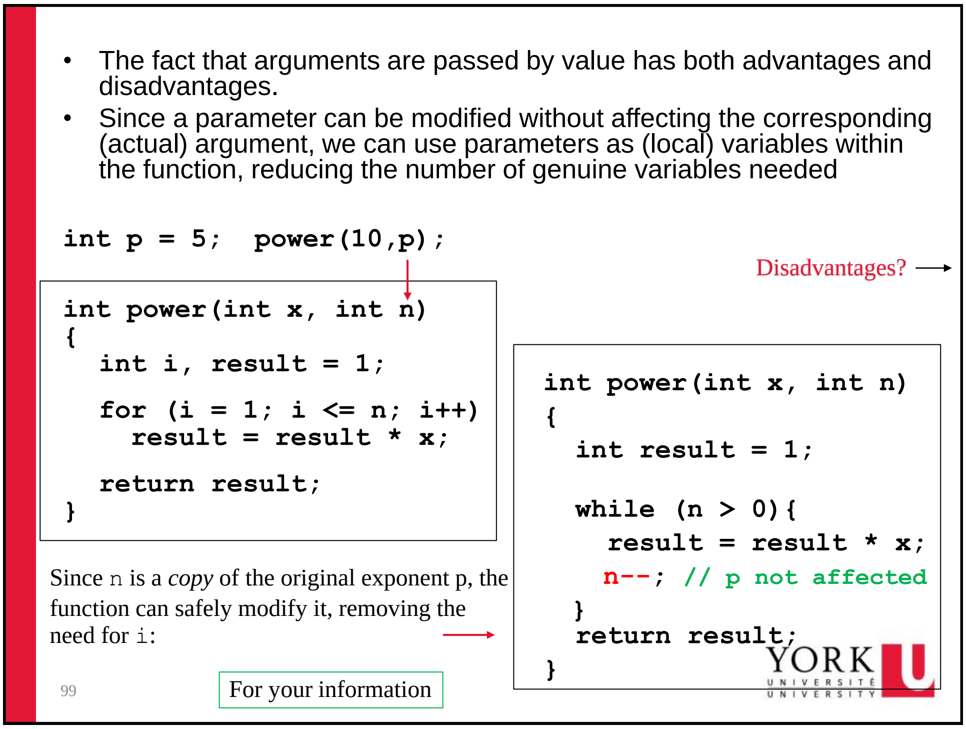
running
main()

```
…
int i =3
int j = 4
k = 7
…

int x = 3
int y = 4
int s = 7
…
```

copy
call **sum()**
copy
running
sum()

98

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);
```

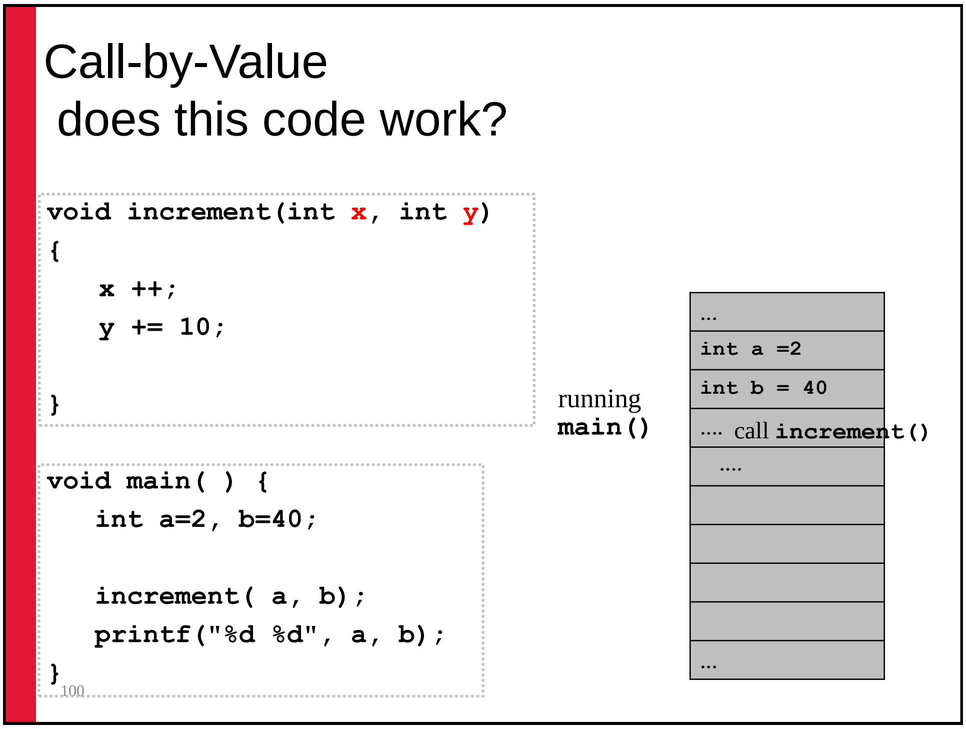
```
int power(int x, int n)
{
  int i, result = 1;

  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:

Disadvantages?

```
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?

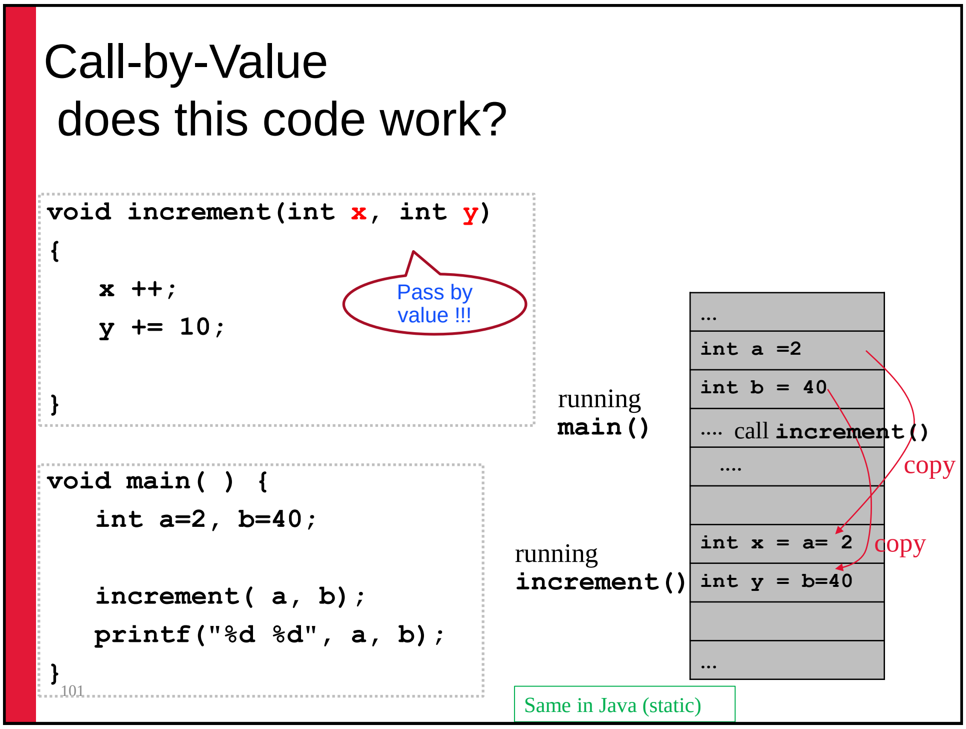
```
void increment(int x, int y)
{
   x ++;
   y += 10;

}
```

```
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()

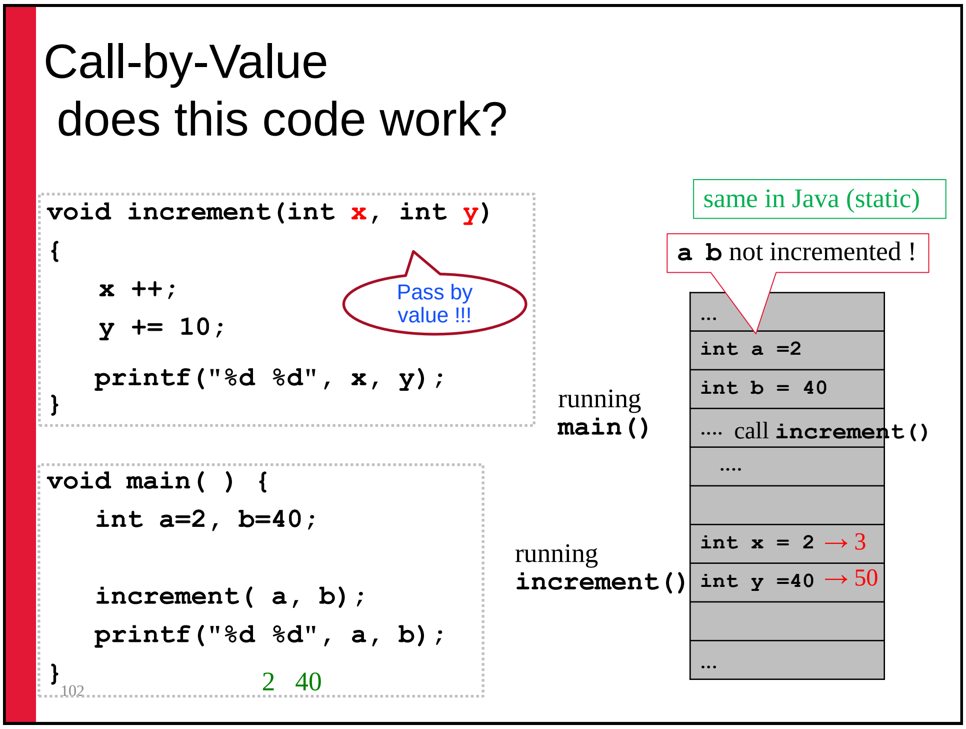
...
int a =2
int b = 40
.... call increment()
....

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

...

copy
copy

Same in Java (static)



Call-by-Value
does this code work?

same in Java (static)

```
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

a b not incremented !

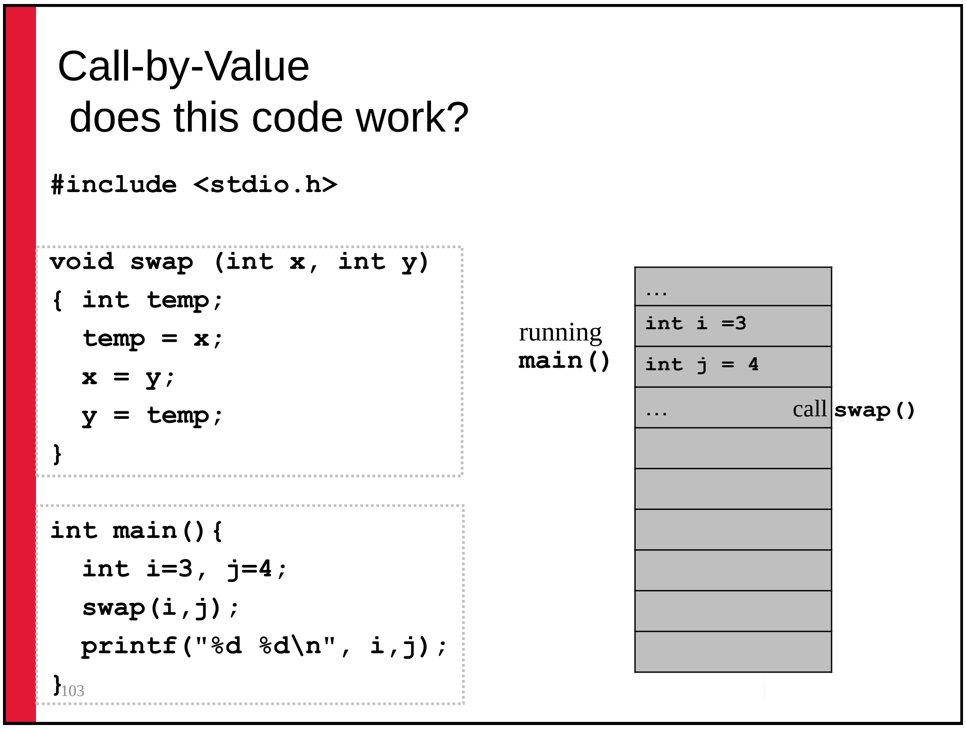
running
main()

running
increment()

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

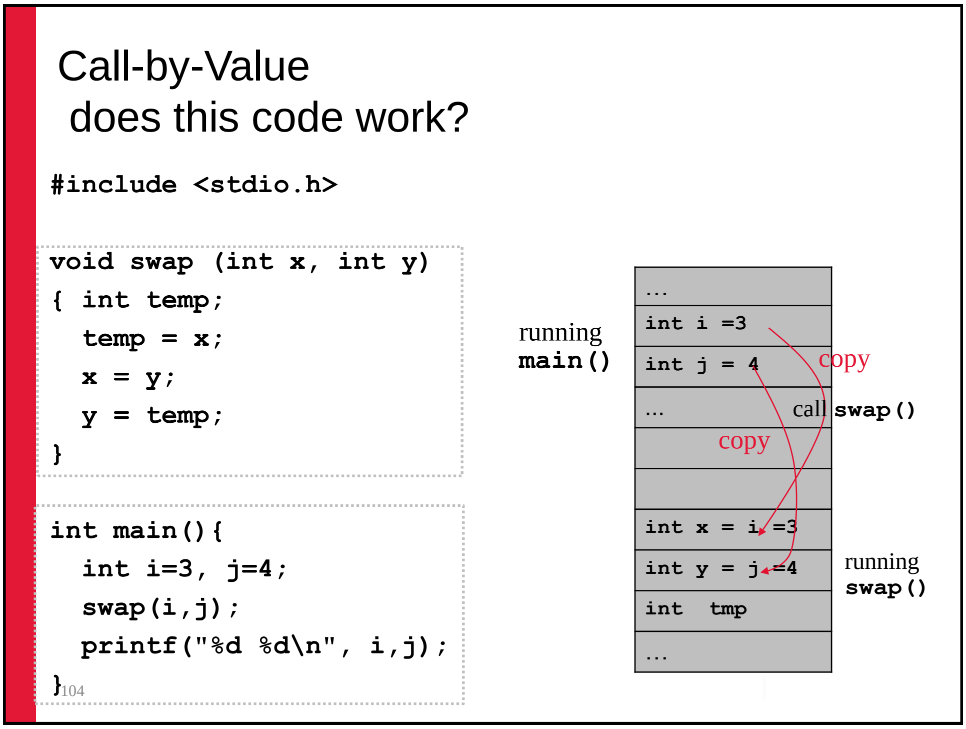
int x = 2 → 3
int y =40 → 50

...



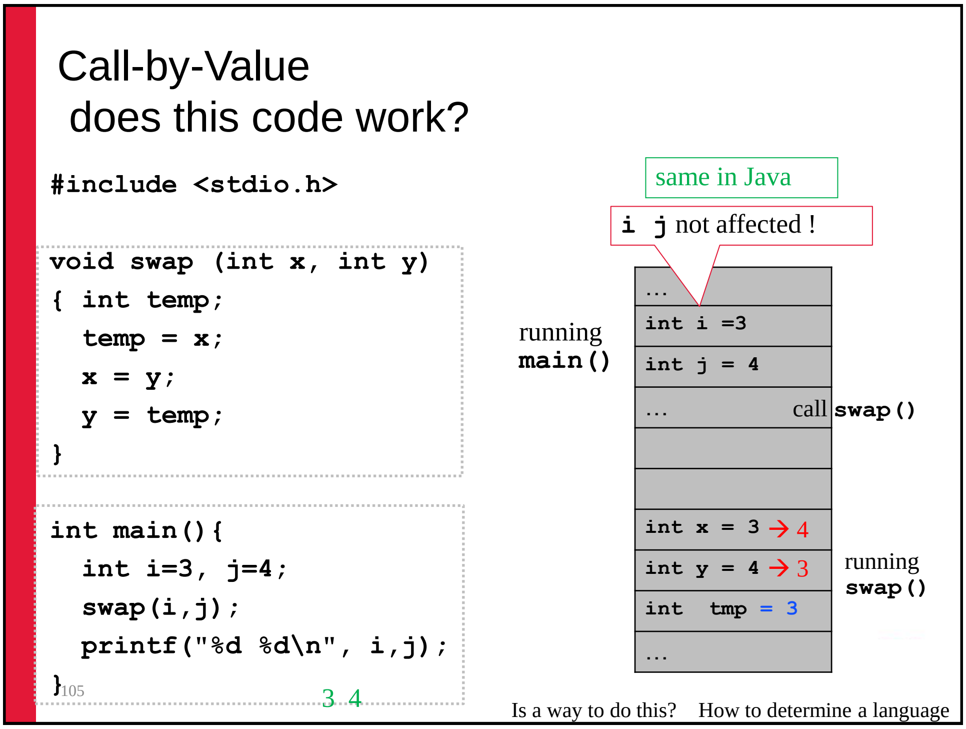
Call-by-Value
does this code work?

```
#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);
}
```



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);
}
```



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);
}
```

105

3 4

same in Java

i j not affected !

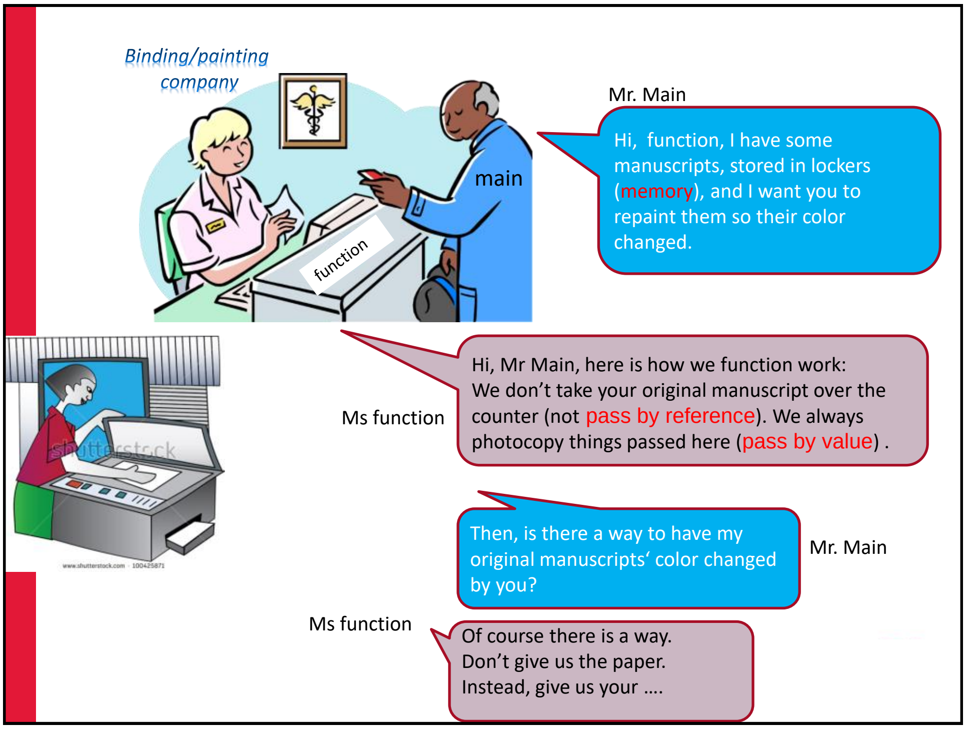
running **main()**

…
`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

105



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) .

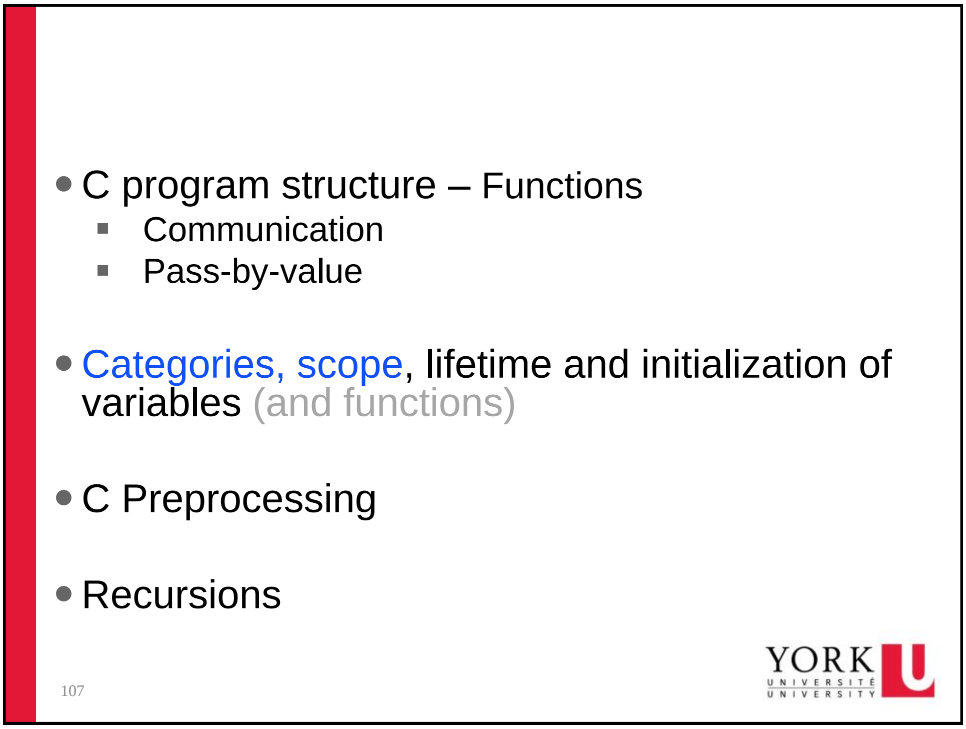
Mr. Main

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

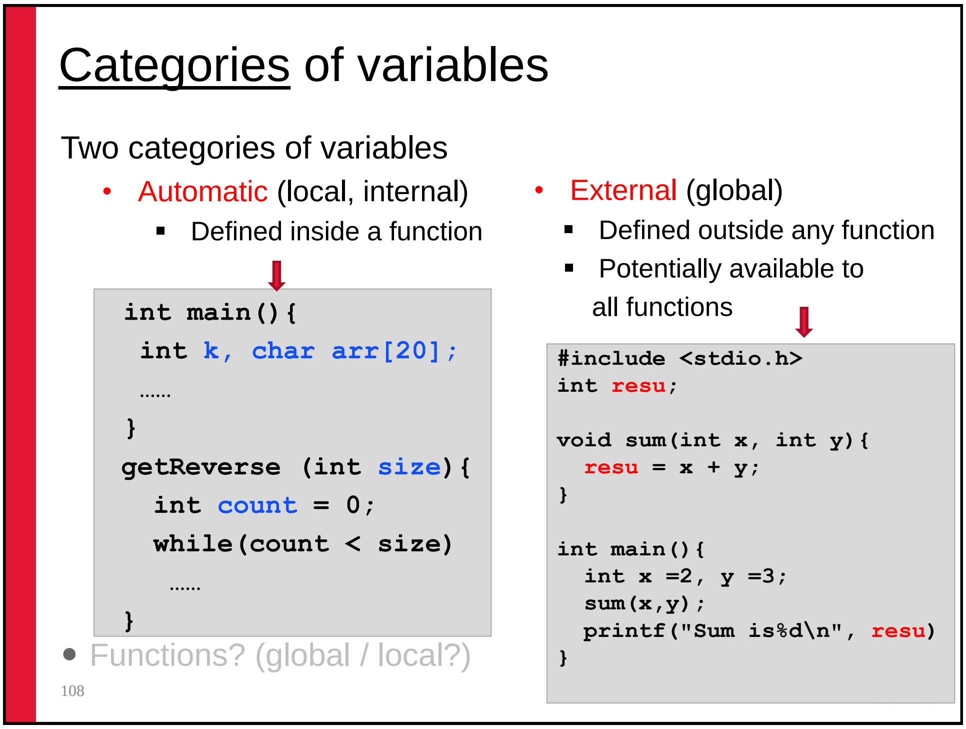
Ms function

Of course there is a way. Don't give us the paper. Instead, give us your ….

106



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



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)
   ......
}
```

- Functions? (global / local?)

- 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)
}
```

Scope

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

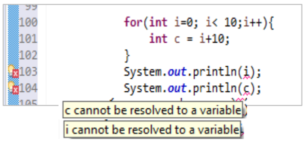
```
 ......
{
  int x;
   ......
  {
    int y; /* y defined here */
     ......
  }
  ......  /* y not accessible here */
}
...... /* x not accessible here */
```

Scope

- Scope of a name (variable or function) – the part of program within which the name can be used – spatial feature
- Global variable (and 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++; /* x not accessible here */
```

same in Java

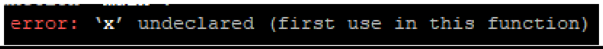


```
for(int i=0; i< 10;i++){
    int c = i+10;
}
System.out.println(i);
System.out.println(c);
```

c cannot be resolved to a variable

i cannot be resolved to a variable

i c defined in for loop.
not accessible after loop



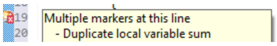
```
error: 'x' undeclared (first use in this function)
```

Scope

- Automatic (local) variables: only exist within their blocks (main, loop...):
- Inner variable can shadow/mask/hide outer variable.

```
//count the sum of numerical values ……
int sum=0;
int arr[4]={3,4,5,6};
int i=0;
for (i=0; i<4; i++)
{
  int sum = sum + arr[i];
  ……
  ……
}
 printf("%d", sum); // 0
```

Not compile in Java

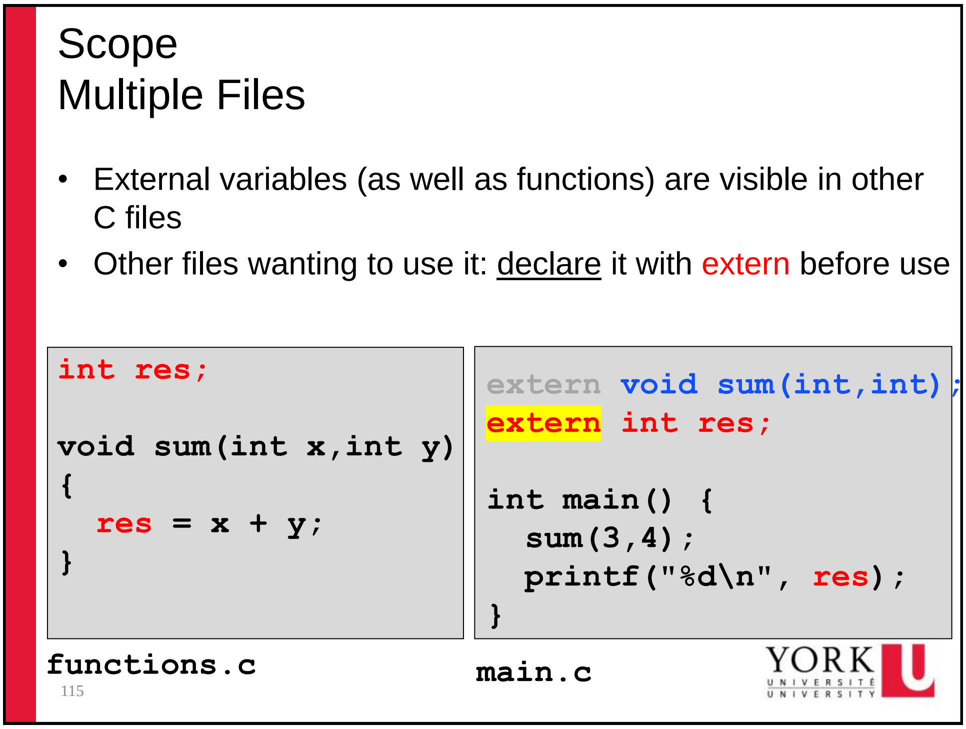


```
Multiple markers at this line
  - Duplicate local variable sum
```

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)
}
```



Scope
Multiple Files

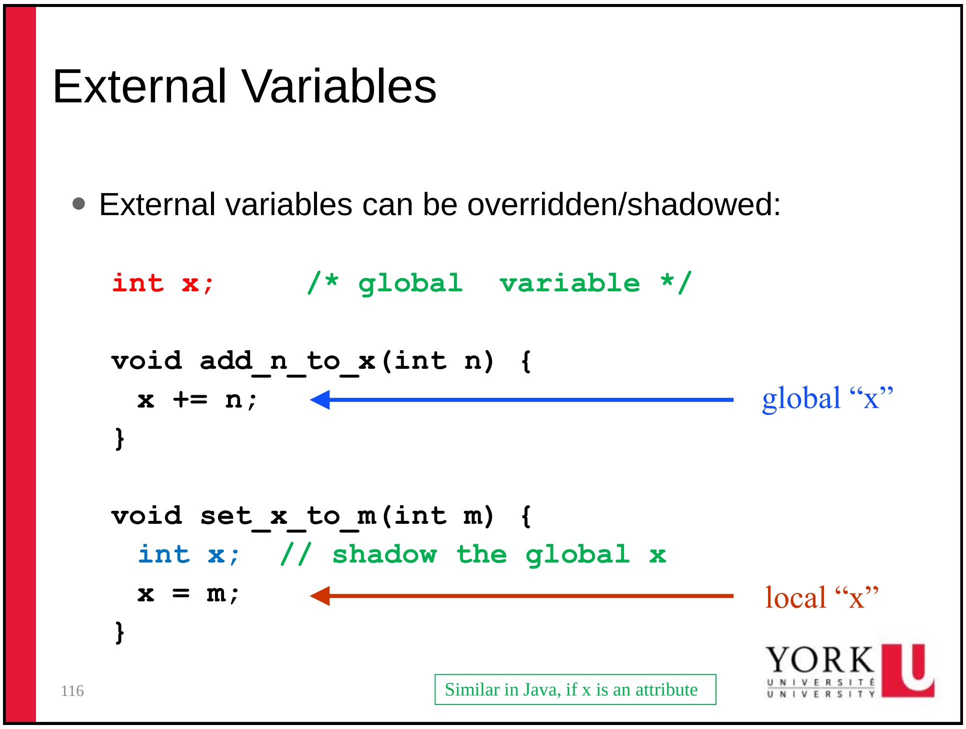
- External variables (as well as functions) are visible in other C files
- Other files wanting to use it: declare it with extern before use

```
int res;

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

functions.c

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

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

main.c



External Variables

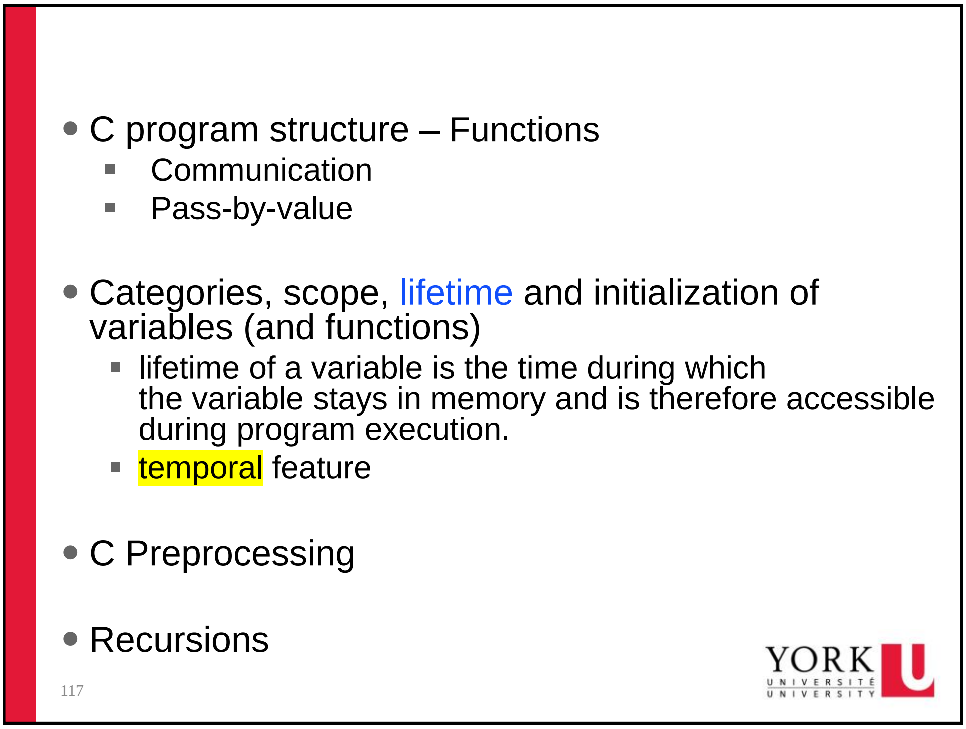
- External variables can be overridden/shadowed:

```
int x;      /* global  variable */

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"

Similar in Java, if x is an attribute

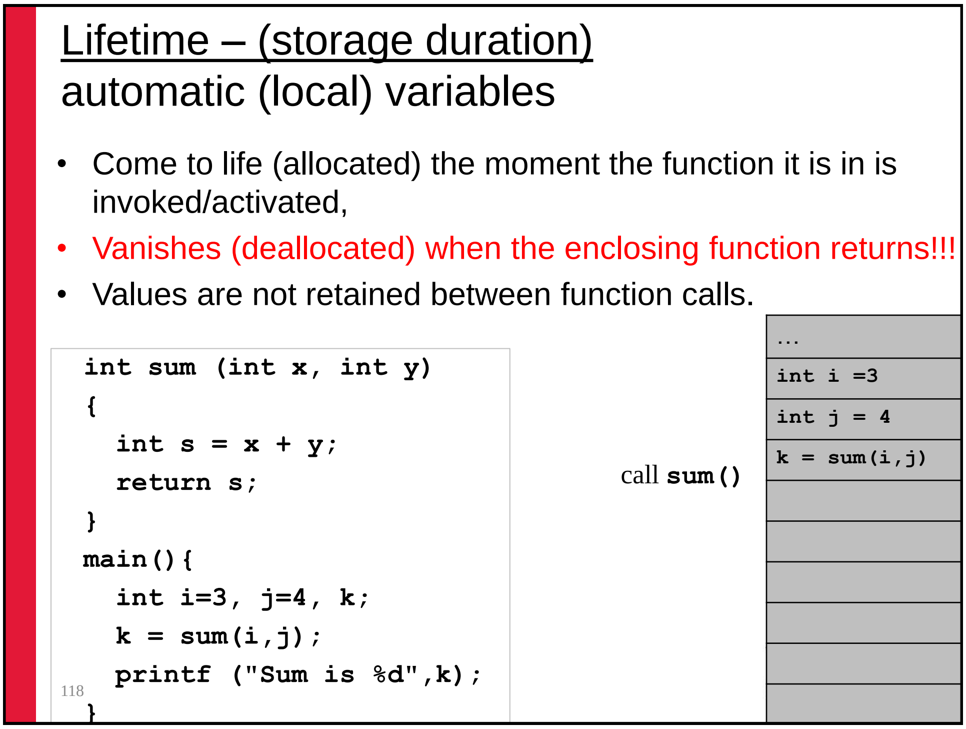


- C program structure – Functions
  - Communication
  - Pass-by-value
- Categories, scope, lifetime and initialization of variables (and functions)
  - lifetime of a variable is the time during which the variable stays in memory and is therefore accessible during program execution.
  - temporal feature
- C Preprocessing
- Recursions

117

YORK U
UNIVERSITÉ
UNIVERSITY

117



Lifetime – (storage duration)
automatic (local) variables

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

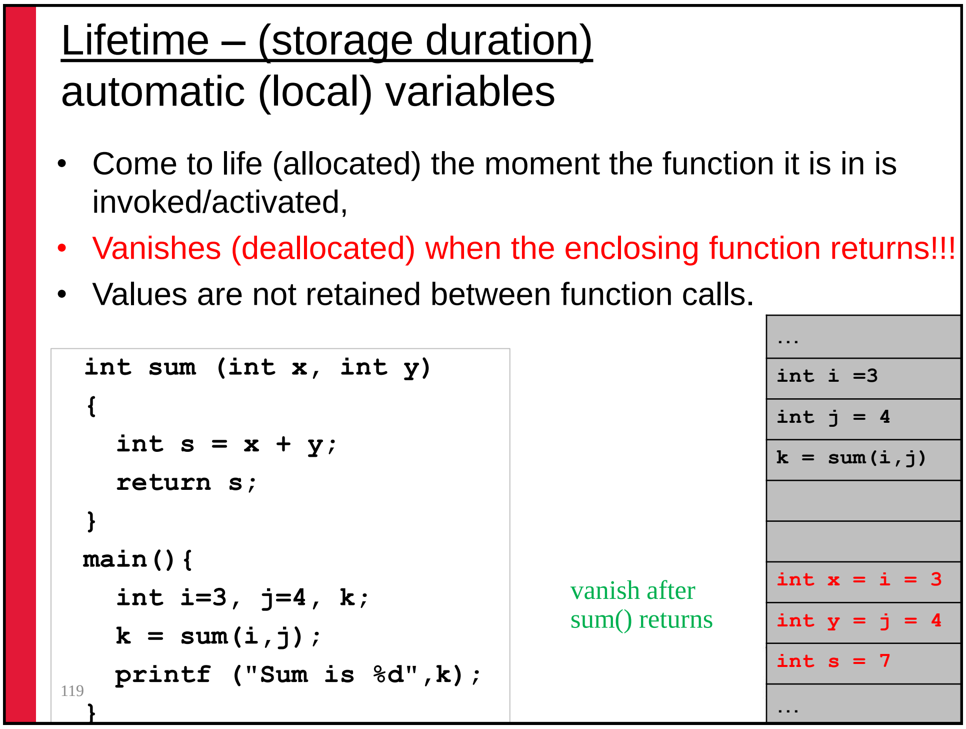
```
int sum (int x, int y)
{
  int s = x + y;
  return s;
}
main(){
  int i=3, j=4, k;
  k = sum(i,j);
  printf ("Sum is %d",k);
}
```

call **sum()**

…
int i =3
int j = 4
k = sum(i,j)

118

118



Lifetime – (storage duration)
automatic (local) variables

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

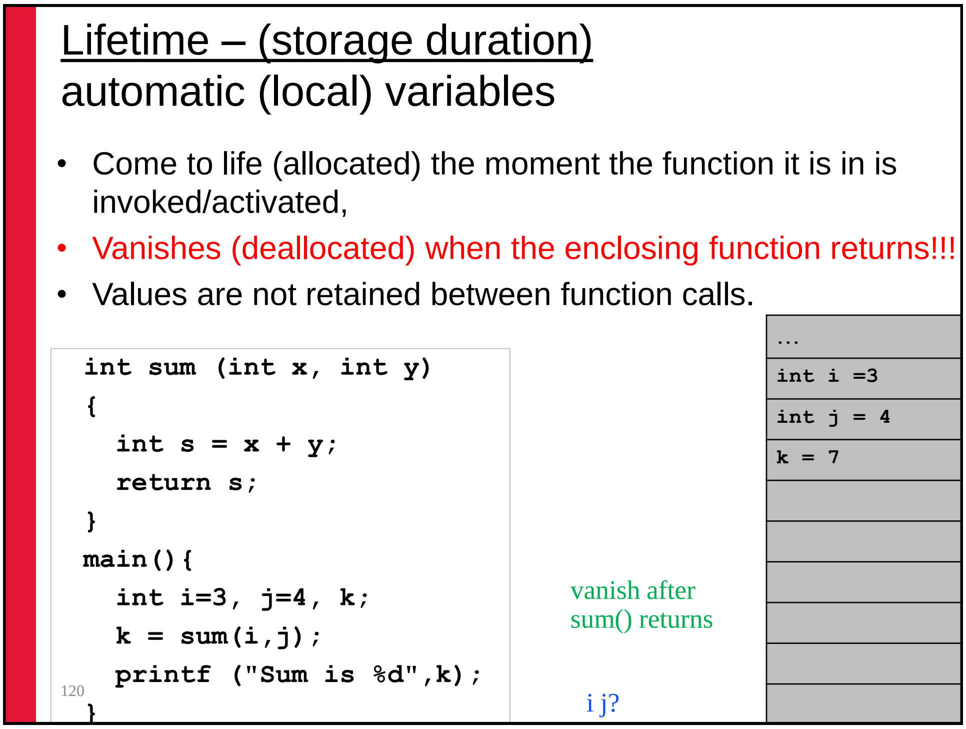
```
int sum (int x, int y)
{
  int s = x + y;
  return s;
}
main(){
  int i=3, j=4, k;
  k = sum(i,j);
  printf ("Sum is %d",k);
}
```

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

vanish after
sum() returns

119

119



Lifetime – (storage duration)
automatic (local) variables

- Come to life (allocated) the moment the function it is in is invoked/activated,
- 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);
}
```

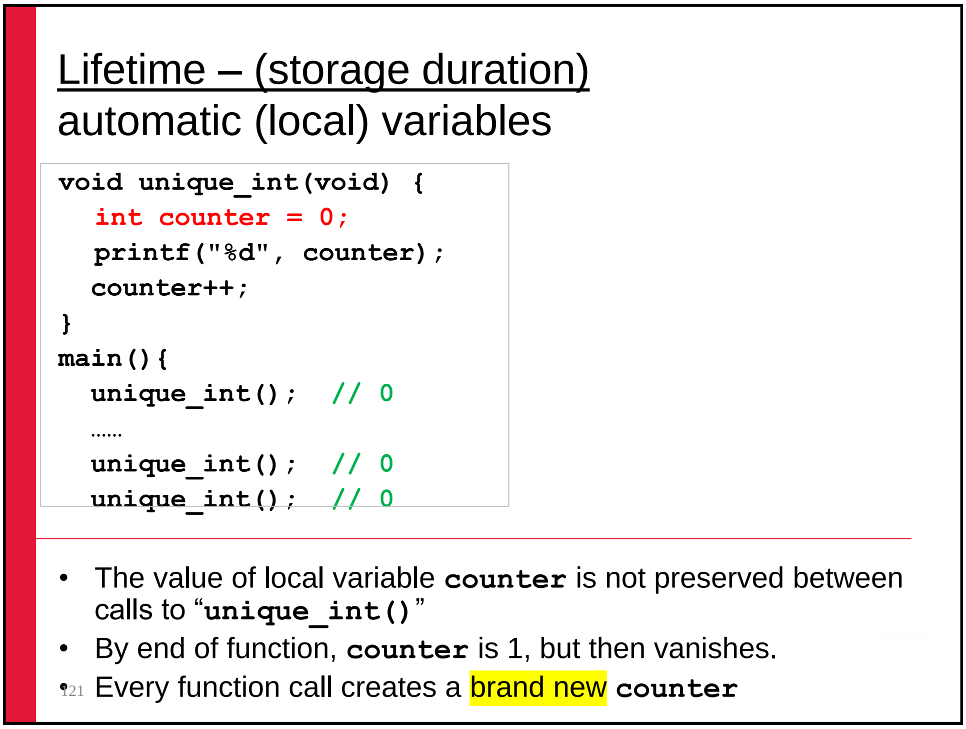
…
int i =3
int j = 4
k = 7

vanish after
sum() returns

i j?

120

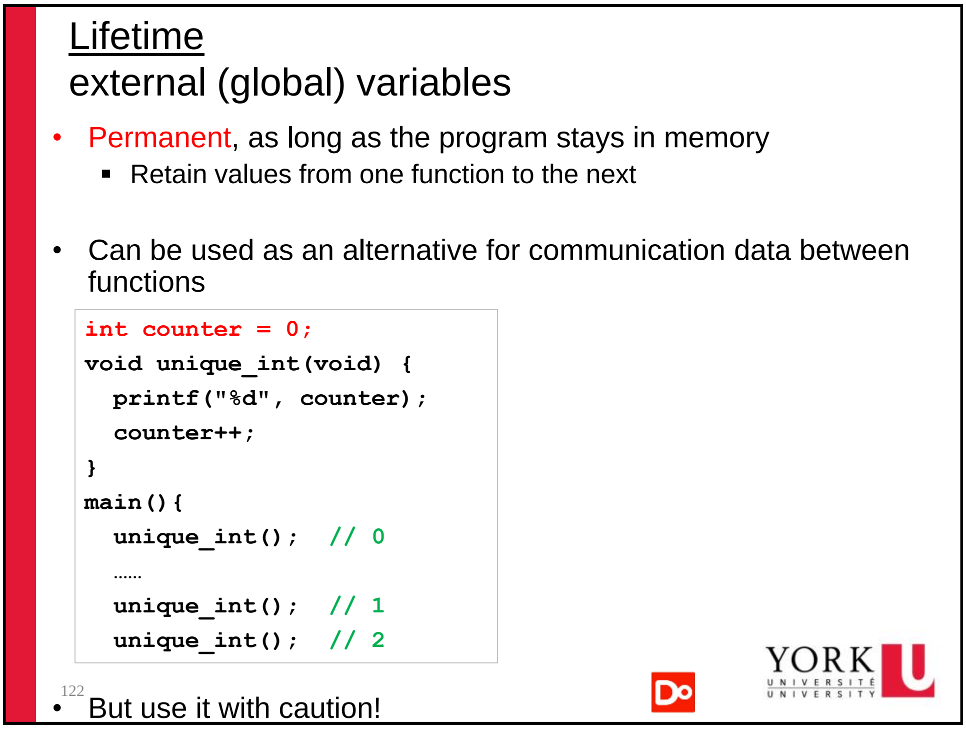
120



Lifetime – (storage duration)
automatic (local) variables

```
void unique_int(void) {
   int counter = 0;
   printf("%d", counter);
   counter++;
}
main(){
   unique_int();  // 0
   ……
   unique_int();  // 0
   unique_int();  // 0
```

- The value of local variable **counter** is not preserved between calls to "**unique_int()**"
- By end of function, **counter** is 1, but then vanishes.
- Every function call creates a brand new **counter**

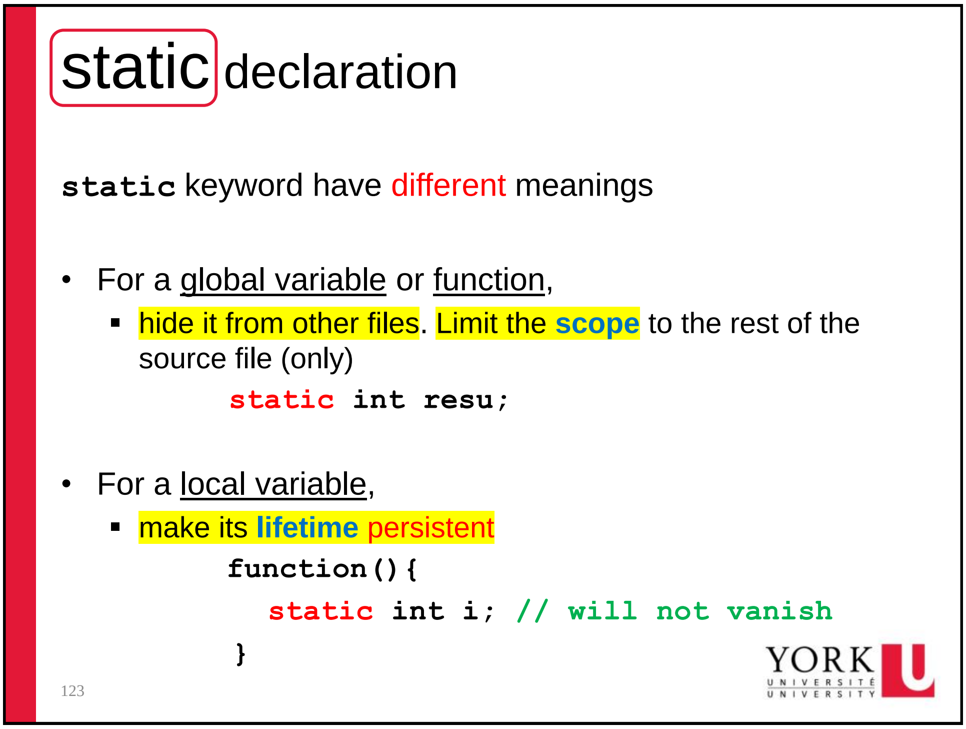


Lifetime
external (global) variables

- 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

```
int counter = 0;
void unique_int(void) {
   printf("%d", counter);
   counter++;
}
main(){
   unique_int();  // 0
   ……
   unique_int();  // 1
   unique_int();  // 2
```

- But use it with caution!



static declaration

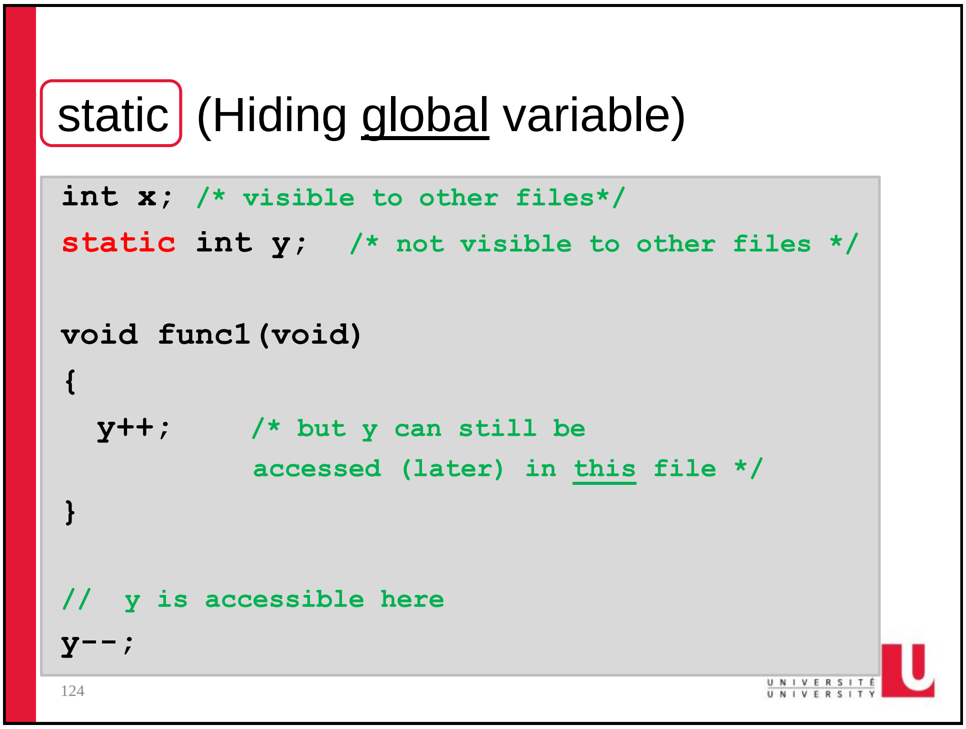
`static` keyword have different meanings

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

```
static int resu;
```

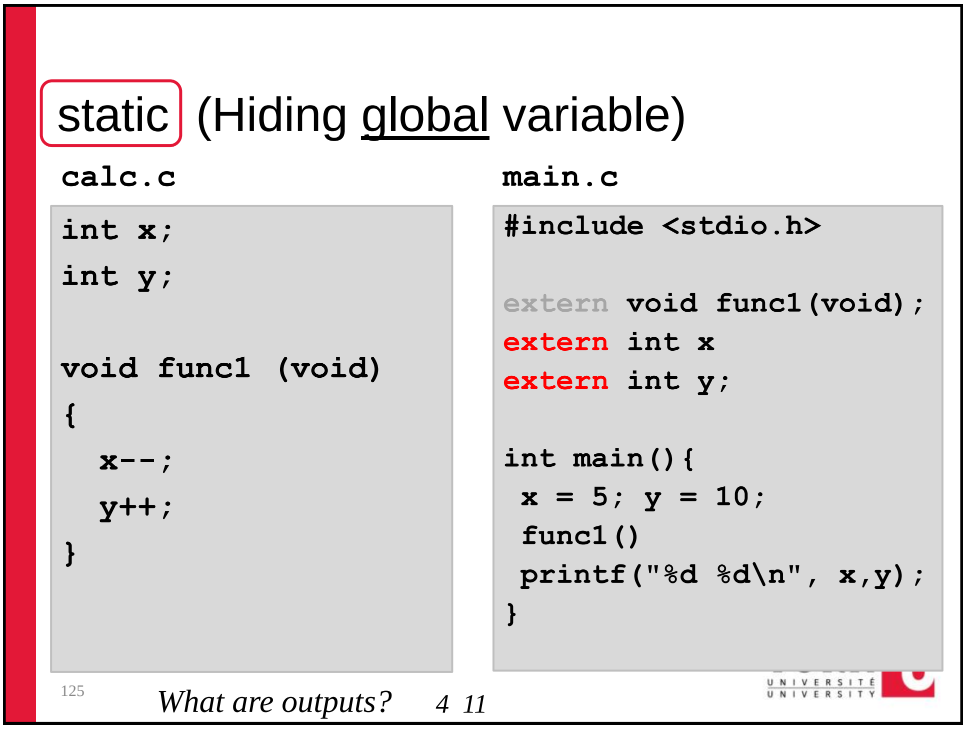
- For a local variable,
  - make its **lifetime** persistent

```
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 */
}

//  y is accessible here
y--;
```



static (Hiding global variable)

calc.c

```
int x;
int y;

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

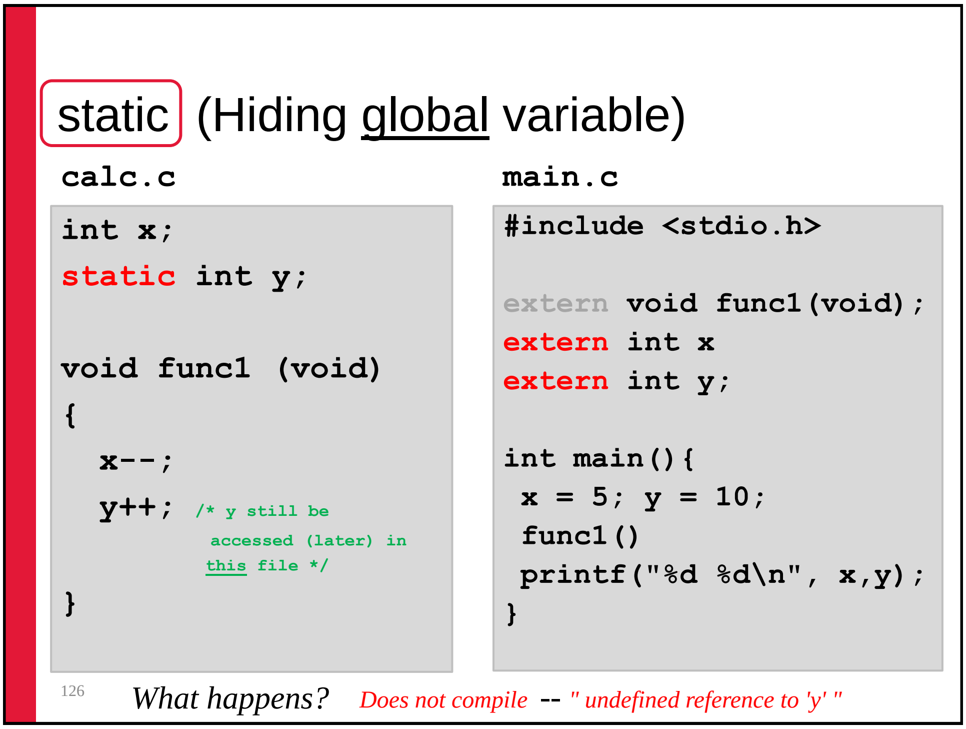
main.c

```
#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);
}
```

125

What are outputs? *4 11*

125



static (Hiding global variable)

calc.c

```
int x;
static int y;

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

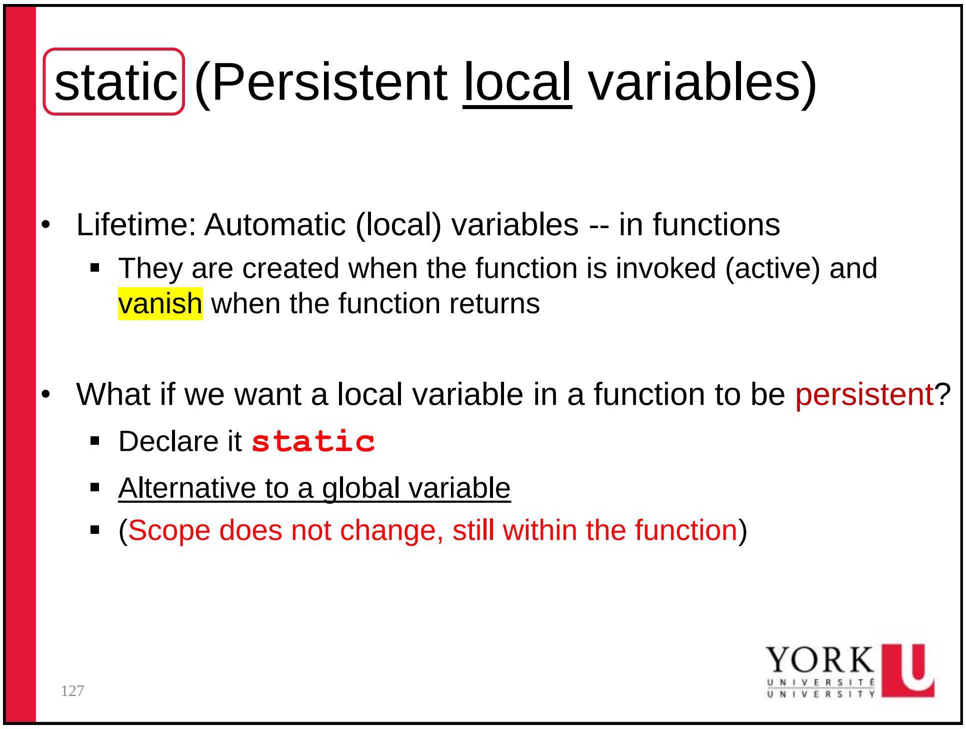
main.c

```
#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);
}
```

126

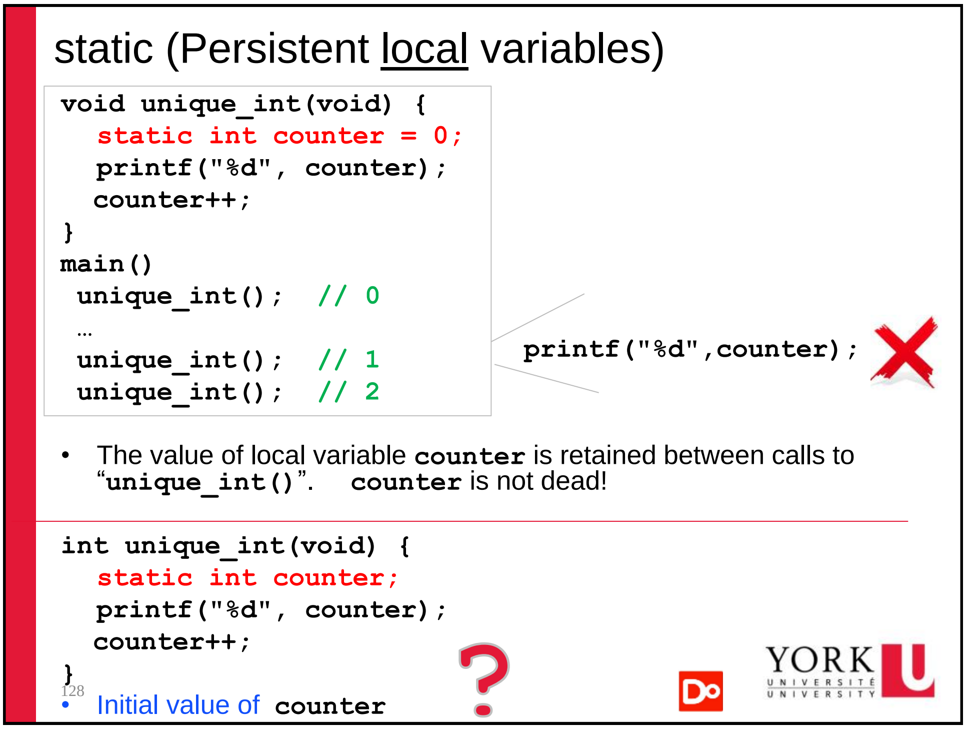
What happens? *Does not compile* -- *" undefined reference to 'y' "*

126



static (Persistent local variables)

- Lifetime: Automatic (local) variables -- in functions
  - They are created when the function is invoked (active) 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)

```
void unique_int(void) {
   static int counter = 0;
   printf("%d", counter);
   counter++;
}
main()
 unique_int();   // 0
 …
 unique_int();   // 1
 unique_int();   // 2
```

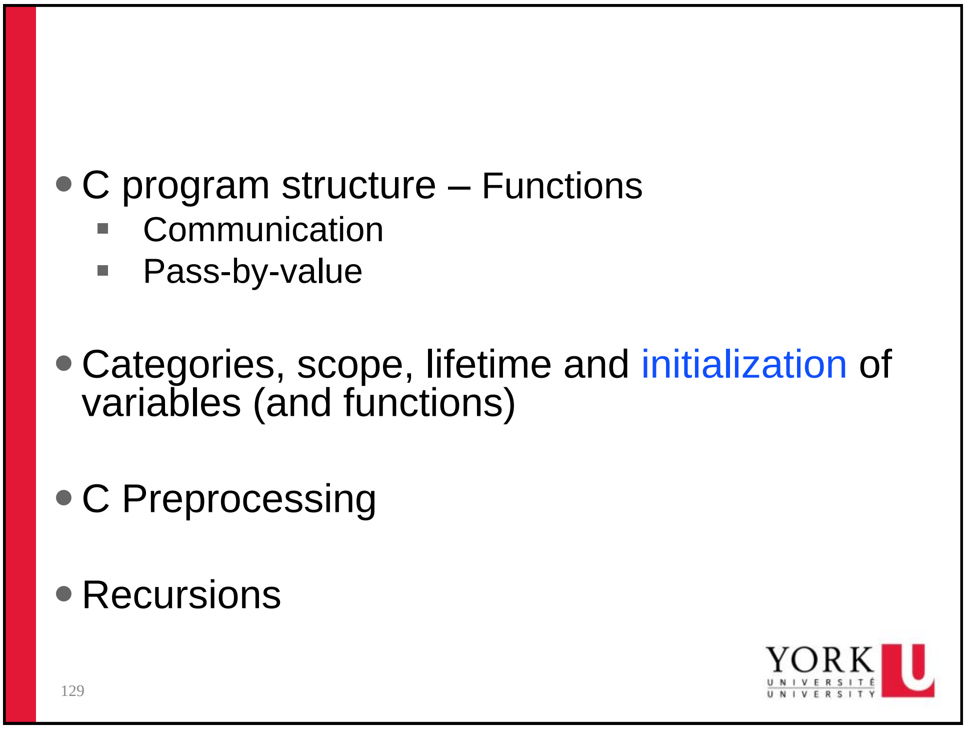
```
printf("%d",counter);
```

✗

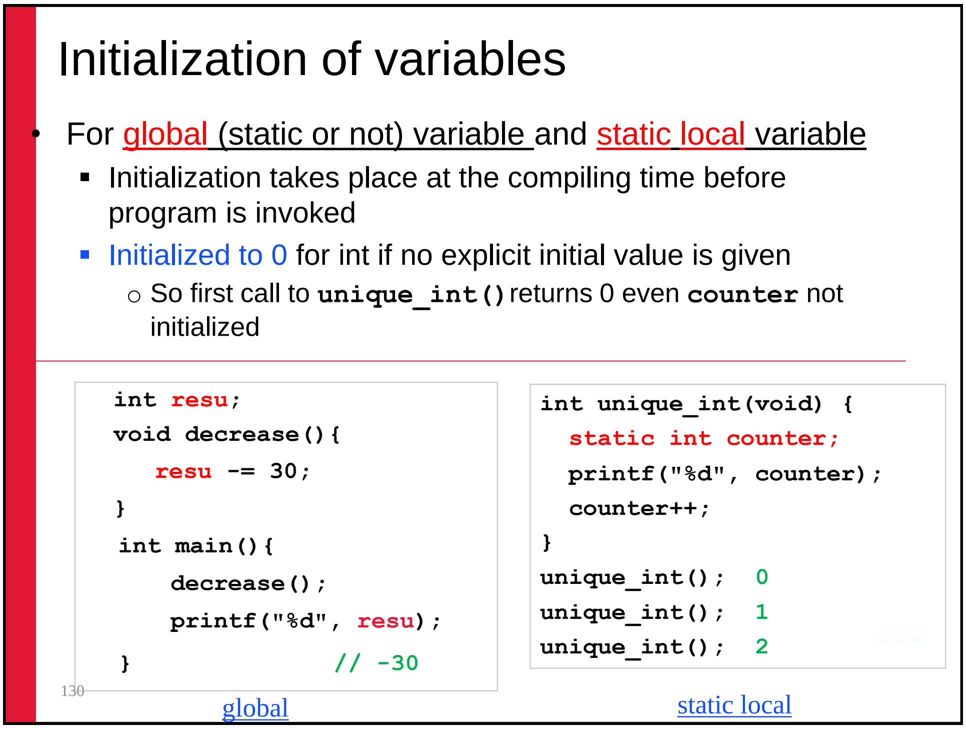
- The value of local variable **`counter`** is retained between calls to “**`unique_int()`**”.   **`counter`** is not dead!

```
int unique_int(void) {
   static int counter;
   printf("%d", counter);
   counter++;
}
```

- Initial value of **`counter`** ?



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



Initialization of variables

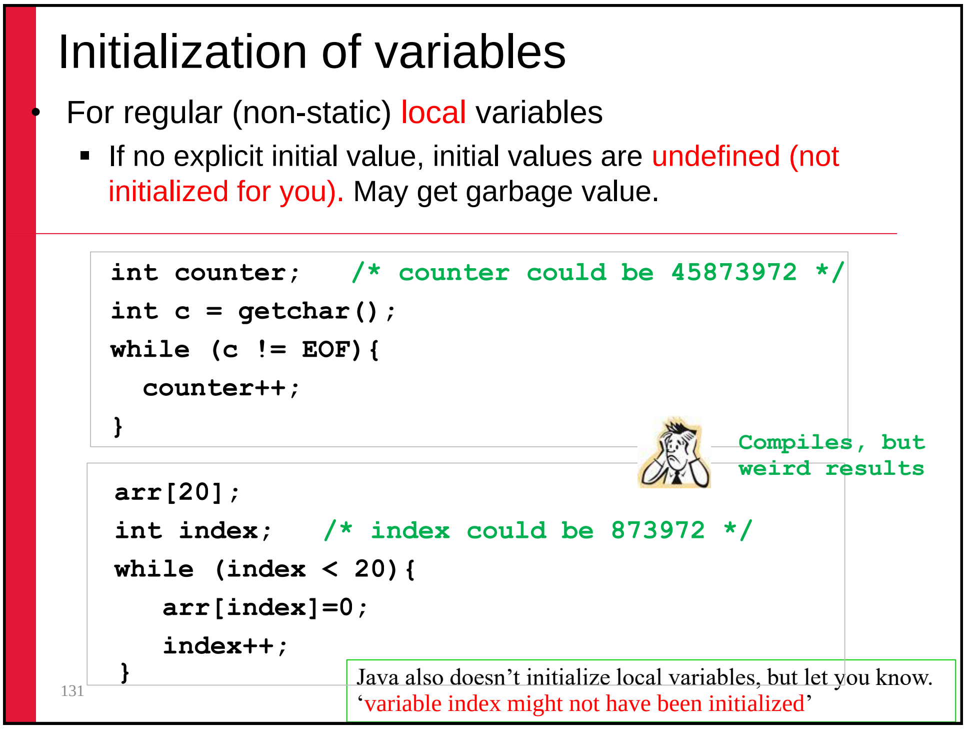
- For global (static or not) variable and static local variable
  - Initialization takes place at the compiling time before program is invoked
  - Initialized to 0 for int if no explicit initial value is given
    - So first call to **unique_int()** returns 0 even **counter** not initialized

```
int resu;
void decrease(){
   resu -= 30;
}
int main(){
    decrease();
    printf("%d", resu);
}                // -30
```

global

```
int unique_int(void) {
  static int counter;
  printf("%d", counter);
  counter++;
}
unique_int();  0
unique_int();  1
unique_int();  2
```

static local



Initialization of variables

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

```
int counter;   /* counter could be 45873972 */
int c = getchar();
while (c != EOF){
  counter++;
}
```

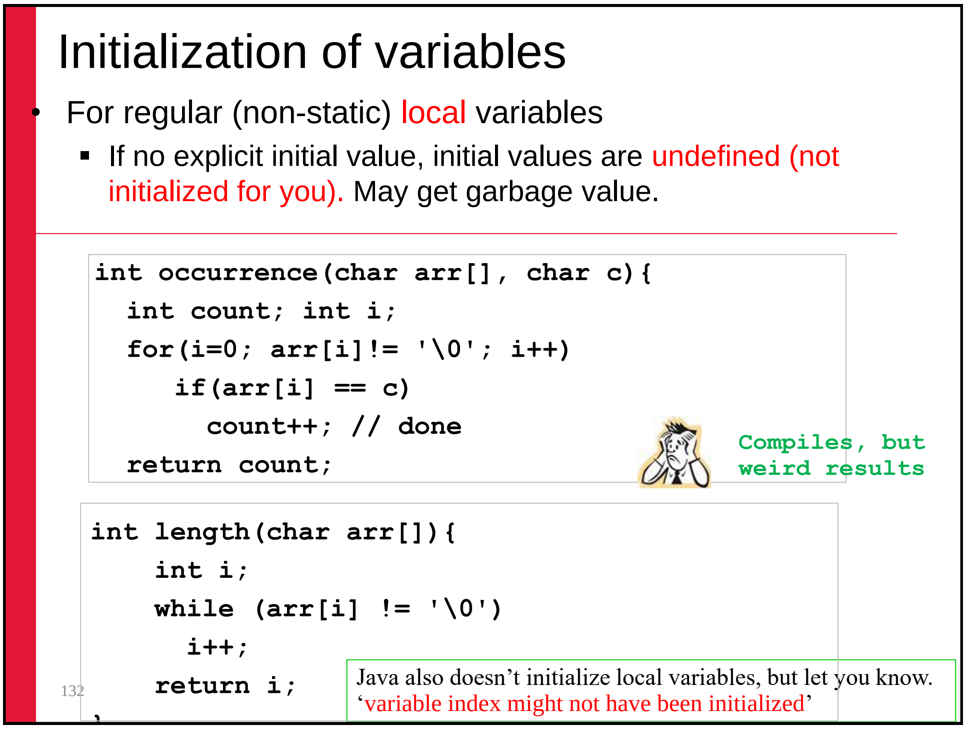
Compiles, but weird results

```
arr[20];
int index;   /* index could be 873972 */
while (index < 20){
   arr[index]=0;
   index++;
}
```

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

131

131



Initialization of variables

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

```
int occurrence(char arr[], char c){
  int count; int i;
  for(i=0; arr[i]!= '\0'; i++)
     if(arr[i] == c)
       count++; // done
  return count;
```

Compiles, but weird results

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

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

132

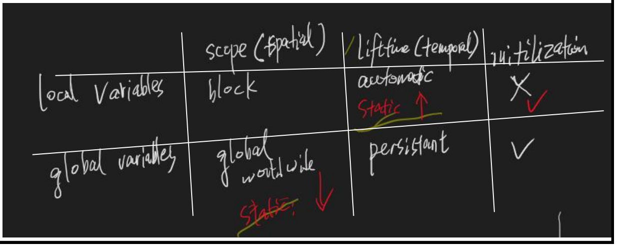
132

Summary of Categories, scope, lifetime and initialization of variables

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

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

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



	scope (spatial)	liftime (temporal)	initilization
local Variables	block	automatic static ↑	X ✓
global variables	global worldwide ~~static~~ ↓	persistant	✓

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!**

Summary Ch4

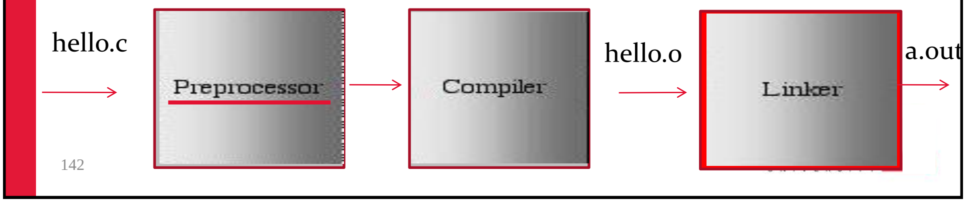
- C program structure, functions
  - Multiple files
  - Communication by global variables
  - "Call-by-value"   increment(), swap()
- Categories, scope, lifetime and initialization of variables (and functions)
  - global and local variables
  - static
- C Preprocessing
  - #include, #define
- Recursion

today

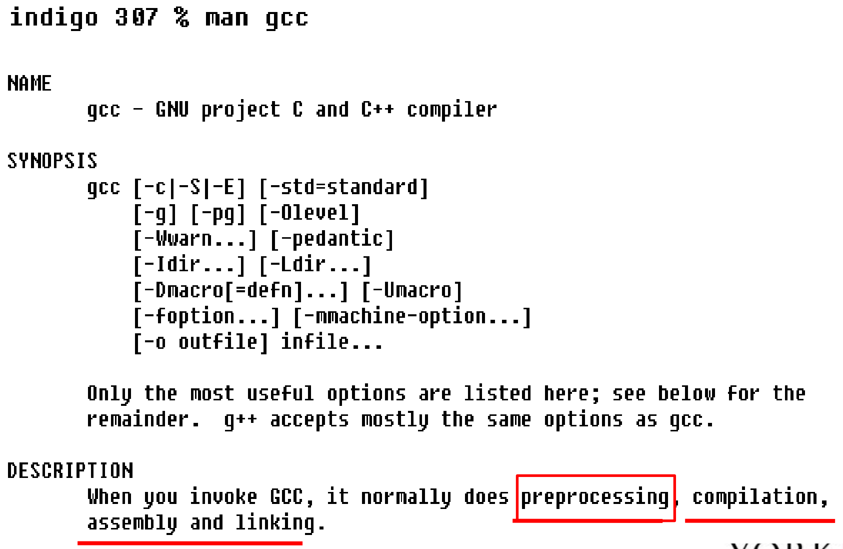
YORK U UNIVERSITÉ UNIVERSITY

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, strlen) and creates an executable program



“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 (to compiler)

Pre-processing #include

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

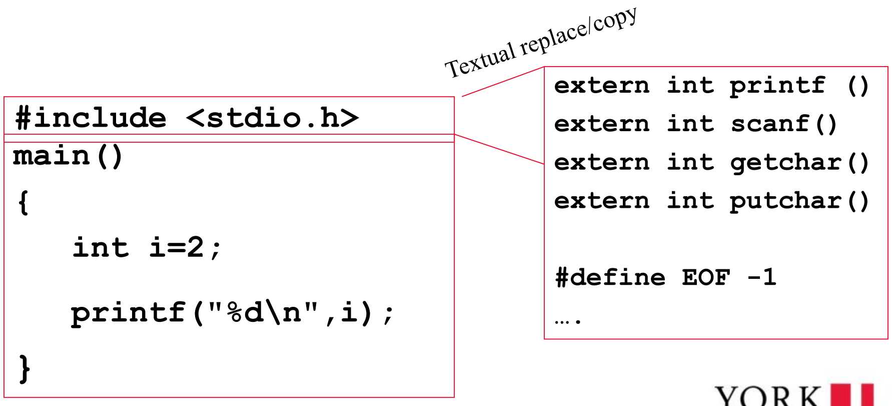
- includes another file in the current file as if contents were part of the current file
  - Textual replace/copy. Nothing fancy

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

Header file

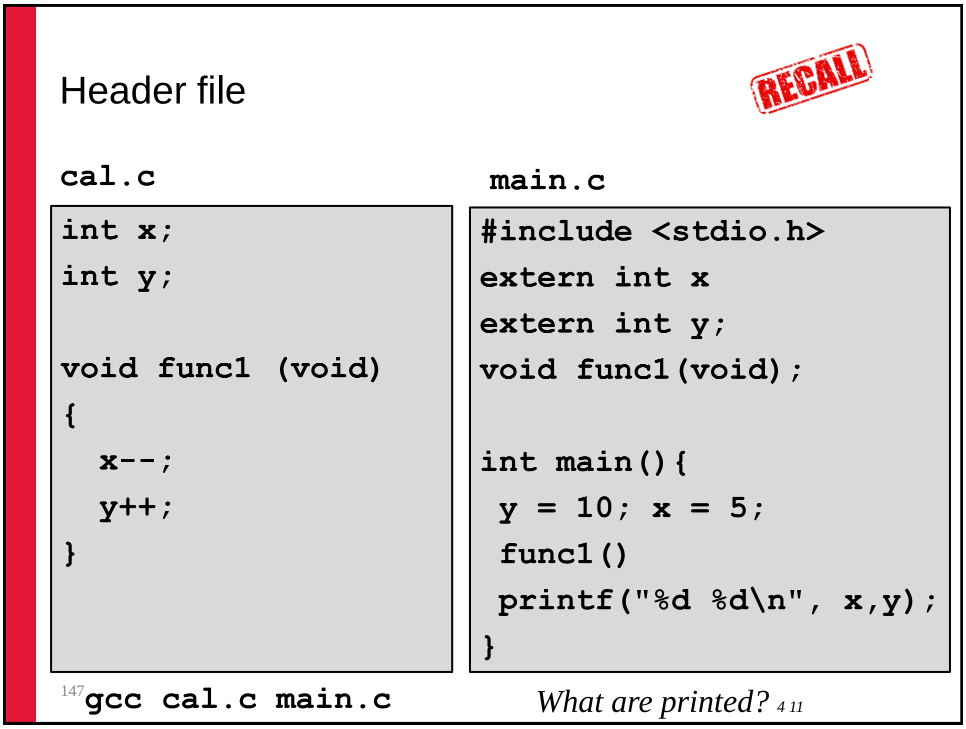
- file **.header** file, which is just c code, usually contains
  - Function Declarations
  - External variable declaration
  - Macro definitions **#define**

RECALL



```
#include <stdio.h>
main()
{
    int i=2;

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

}
```

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

#define EOF -1
….
```



Header file

RECALL

cal.c

```
int x;
int y;

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

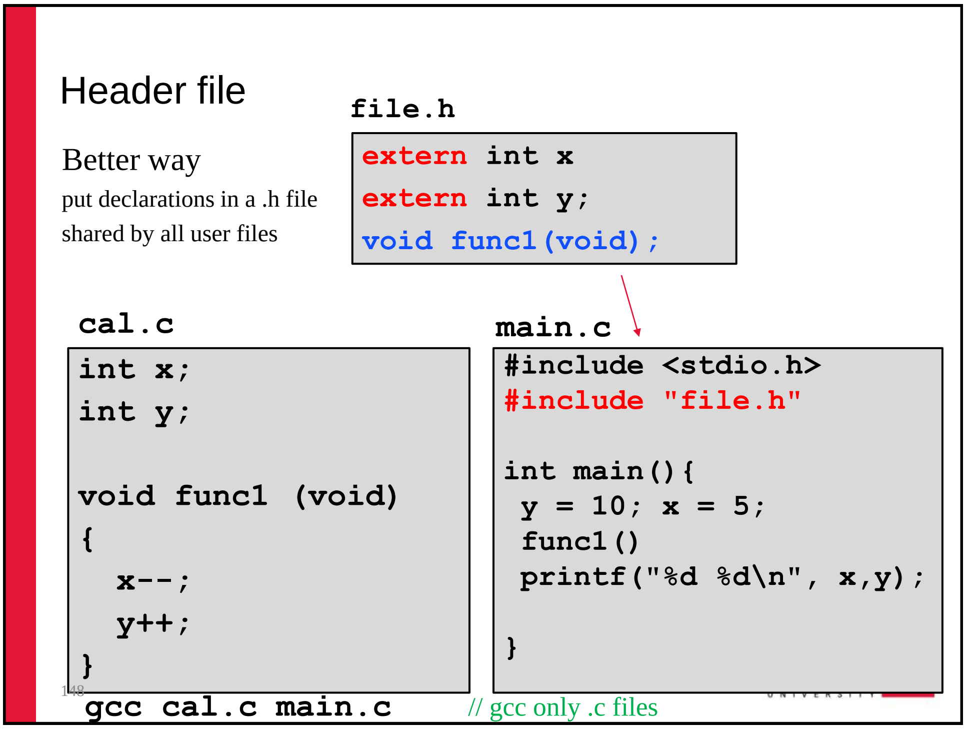
main.c

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

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

147

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

What are printed? 4 11



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++;
}
```

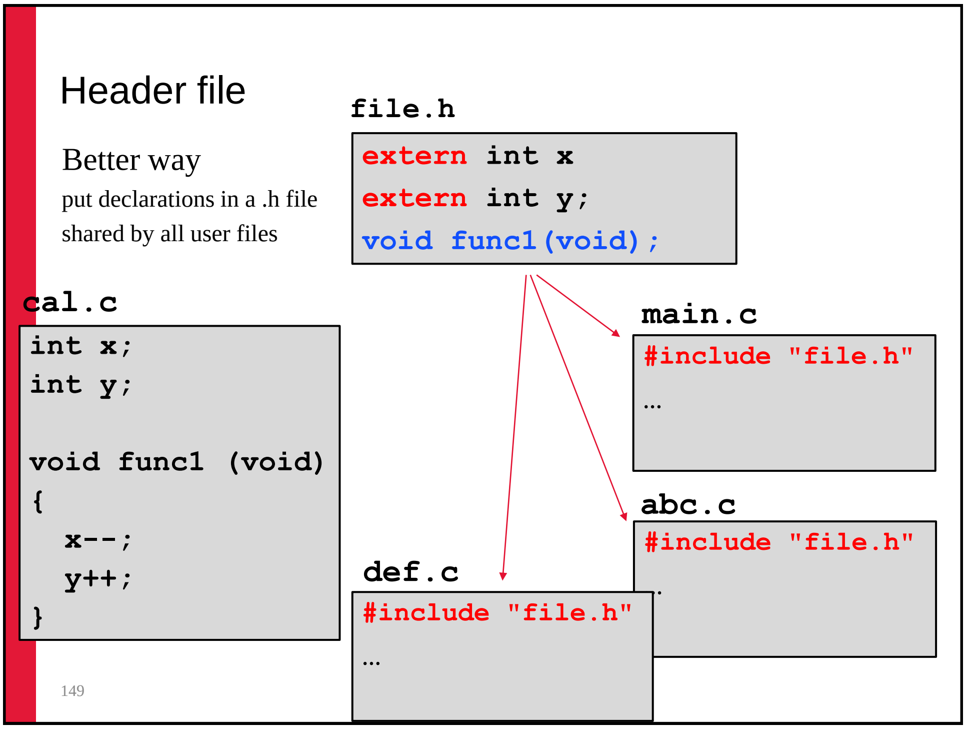
main.c

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

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

}
```

148

```
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"
…
```

abc.c

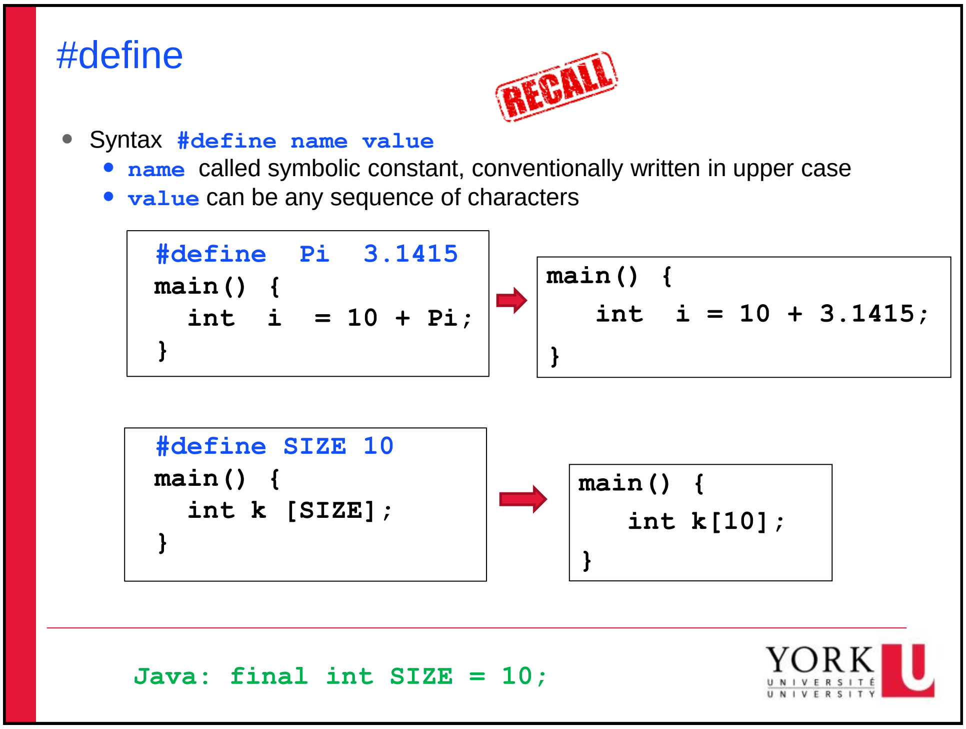
```
#include "file.h"
.
```

def.c

```
#include "file.h"
…
```

149

149



#define

RECALL

- 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;
}
```

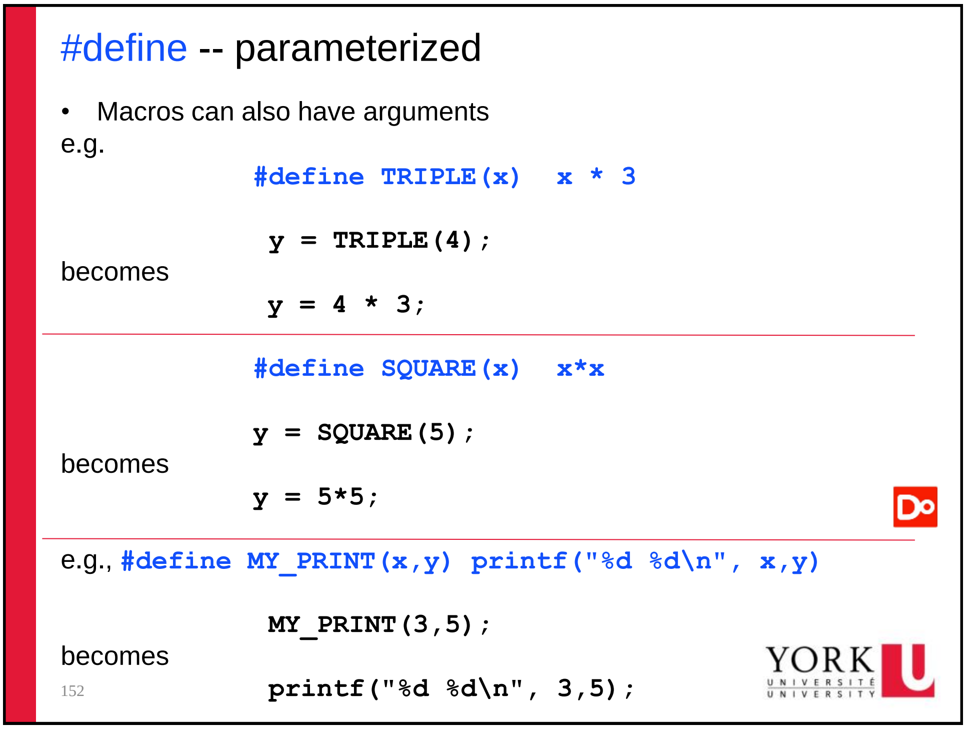
```
#define SIZE 10
main() {
  int k [SIZE];
}
```

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

`Java: final int SIZE = 10;`

YORK U UNIVERSITÉ UNIVERSITY

151



#define -- parameterized

- Macros can also have arguments

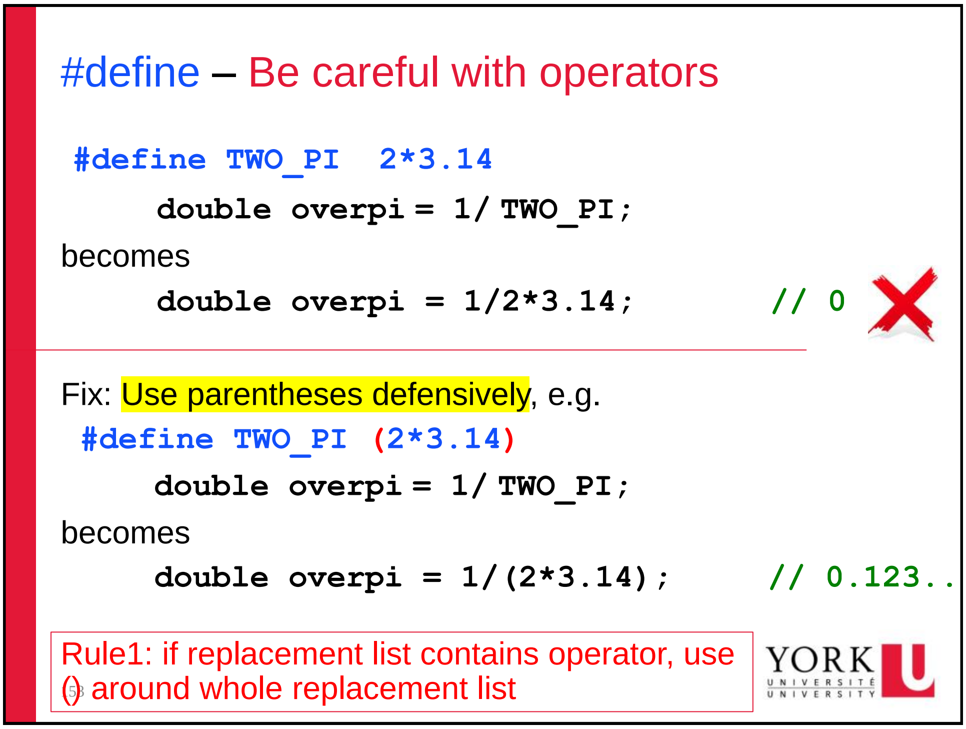
e.g.

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

             y = TRIPLE(4);
becomes
             y = 4 * 3;
```

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

            y = SQUARE(5);
becomes
            y = 5*5;
```

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

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



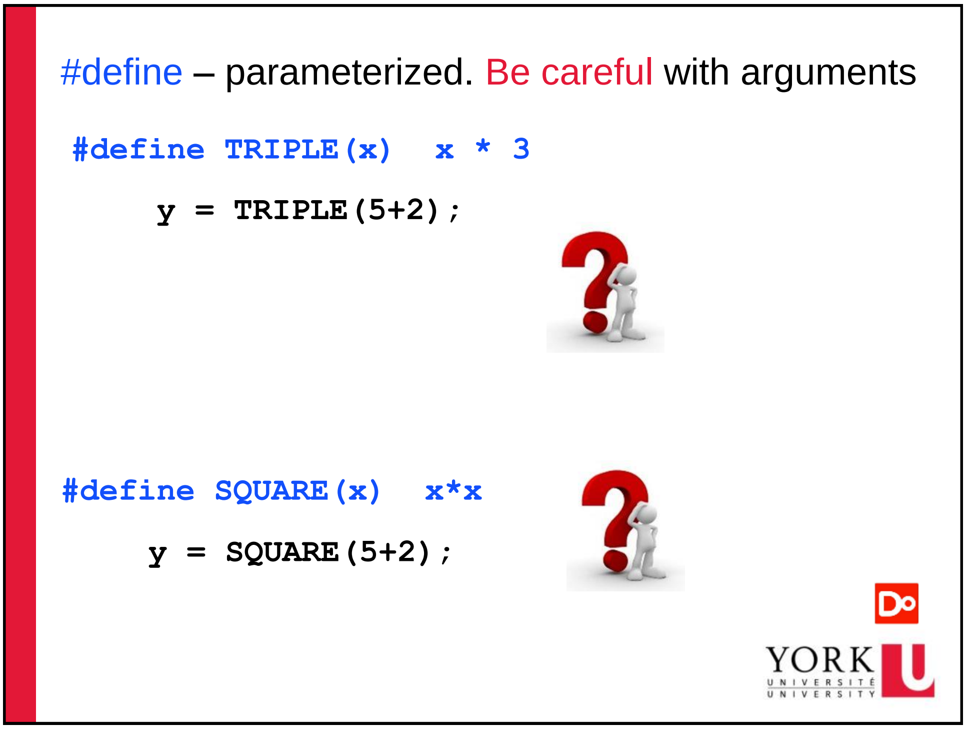
#define – Be careful with operators

```
 #define TWO_PI  2*3.14
     double overpi = 1/ TWO_PI;
becomes
     double overpi = 1/2*3.14;        // 0  ✗
```

Fix: Use parentheses defensively, e.g.

```
 #define TWO_PI (2*3.14)
     double overpi = 1/ TWO_PI;
becomes
     double overpi = 1/(2*3.14);     // 0.123..
```

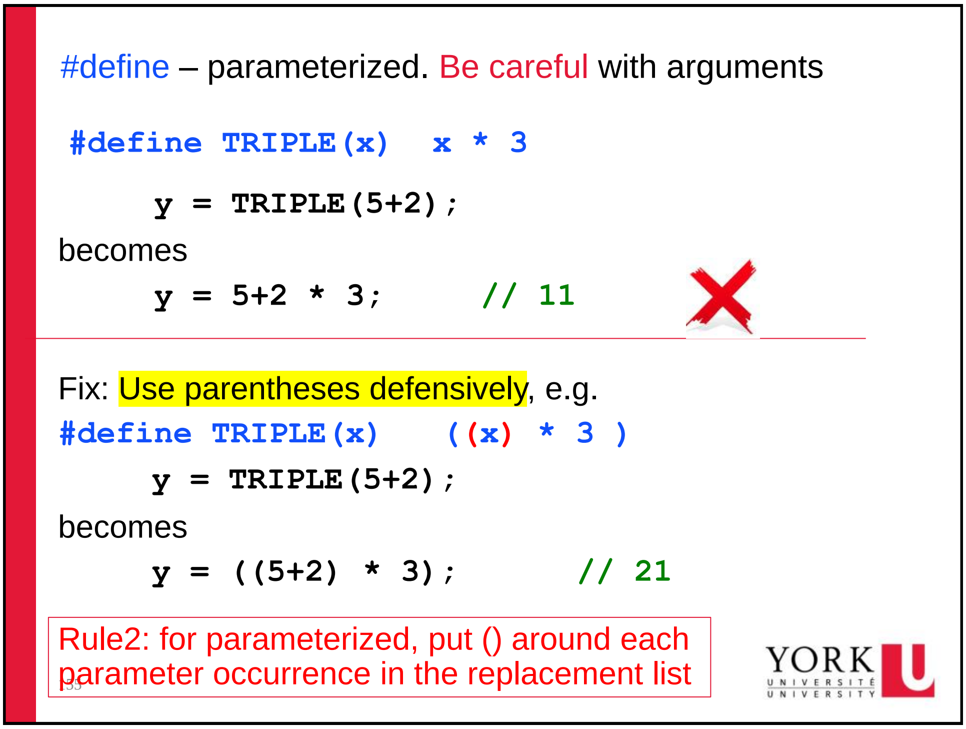
Rule1: if replacement list contains operator, use () around whole replacement list



#define – parameterized. Be careful with arguments

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

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



#define – parameterized. Be careful with arguments

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

becomes

```
     y = 5+2 * 3;      // 11
```

Fix: Use parentheses defensively, e.g.

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

becomes

```
     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

     y = SQUARE(5+2);
```

becomes

```
     y = 5+2*5+2;          // 17
```

Fix: Use parentheses defensively, e.g.

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

becomes

```
      y = ((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__
```

```
#include <stdio.h>
main(){
  printf("%s %s\n", __TIME__, __DATE__);
  printf("File: %s Line: %d\n", __FILE__,__LINE__);
}
```

```
21:45:54  Feb 6 2021
File: macro.c  Line:4
```

- Useful for debugging

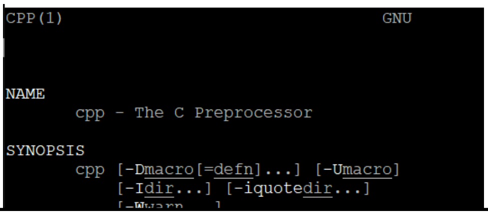
Playing with the C Preprocessor

- Try:

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

- **-E** means “just run the preprocessor”

- Also **`cpp file.c`**



```
CPP(1)                                   GNU



NAME
       cpp - The C Preprocessor

SYNOPSIS
       cpp [-Dmacro[=defn]...] [-Umacro]
           [-Idir...] [-iquotedir...]
           [-Wwarn...]
```

Summary Ch4

- C program structure, functions
  - Multiple files
  - Communication by global variables
  - “Call-by-value”   increment(), swap()

- Categories, scope, lifetime and initialization of variables (and functions)
  - global and local variables
  - static

- C Preprocessing
  - #include, #define

today

- Recursion