Program Structure

EECS 2031

Song Wang
wangsong@eecs.yorku.ca
eecs.yorku.ca/~wangsong/

Acknowledgement

- Some of the covered materials are based on previous EECS2031 offerings:
  - Uyen Trang (UT) Nguyen, Pooja Vashisth, Hui Wang, Manos Papagelis

Declaring Functions

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

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

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

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

- We can omit parameter names

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

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

Program structure --Functions

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

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

"parameters",
"formal parameters"

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

"arguments",
"actual parameters"

Program structure --Functions

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

- Communication between functions
  - by arguments and return values
  - by external variable (ch1.10, ch4.3)

- Functions can occur
  - in a single source file
  - in multiple source files

Program structure --Functions

communication by arguments and return values

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

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

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

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

Communication by arguments and return values

Function communication by external variables Not recommended!

```
#include <stdio.h>

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

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

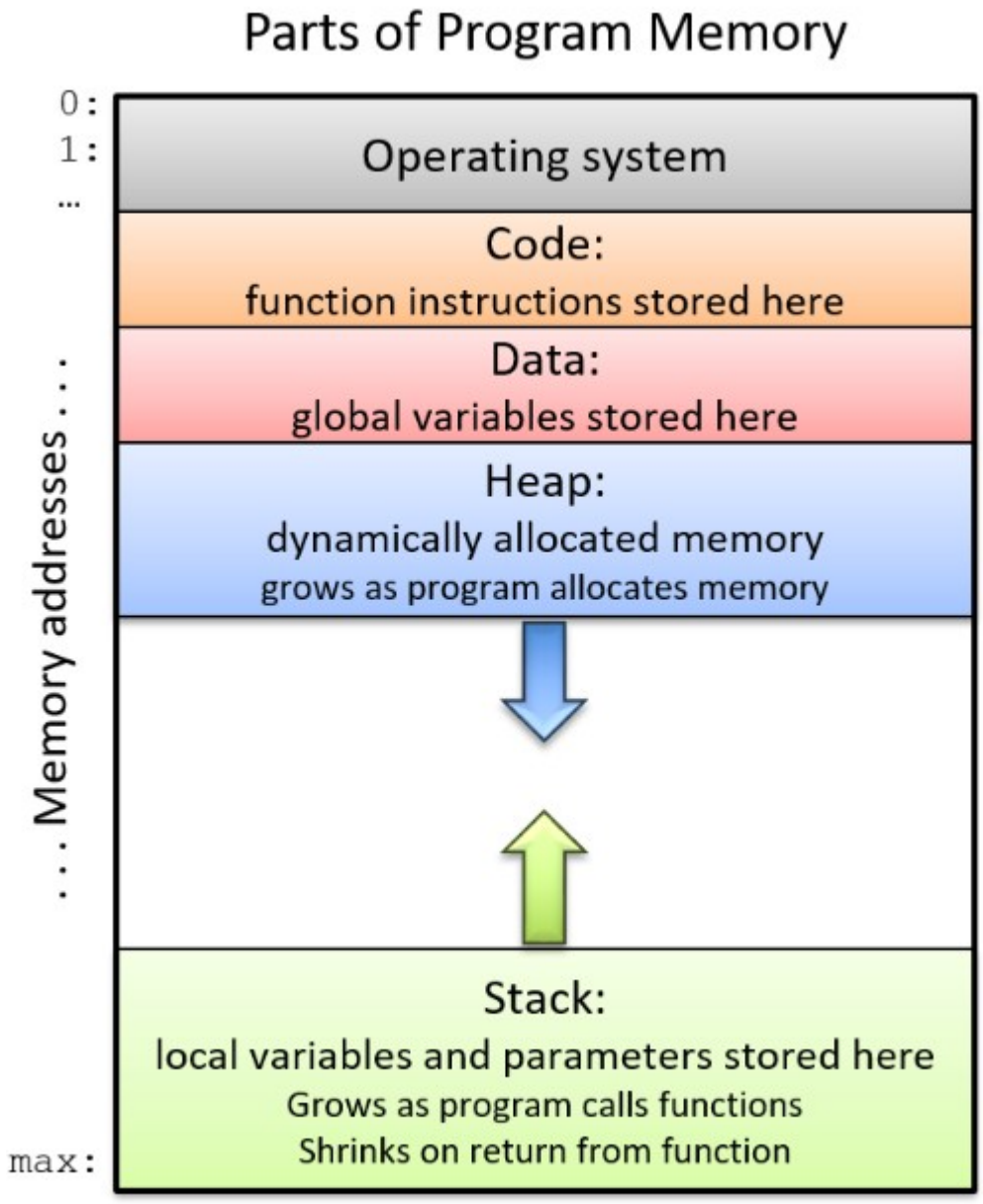


Figure 1. The parts of a program's address space.

Multiple source files

Can call a function defined in another file. How 

functions.c

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


// no main,
// no #include
```

main.c

```
#include <stdio.h>



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

Multiple source files

Can call a function defined in another file. How 

functions.c

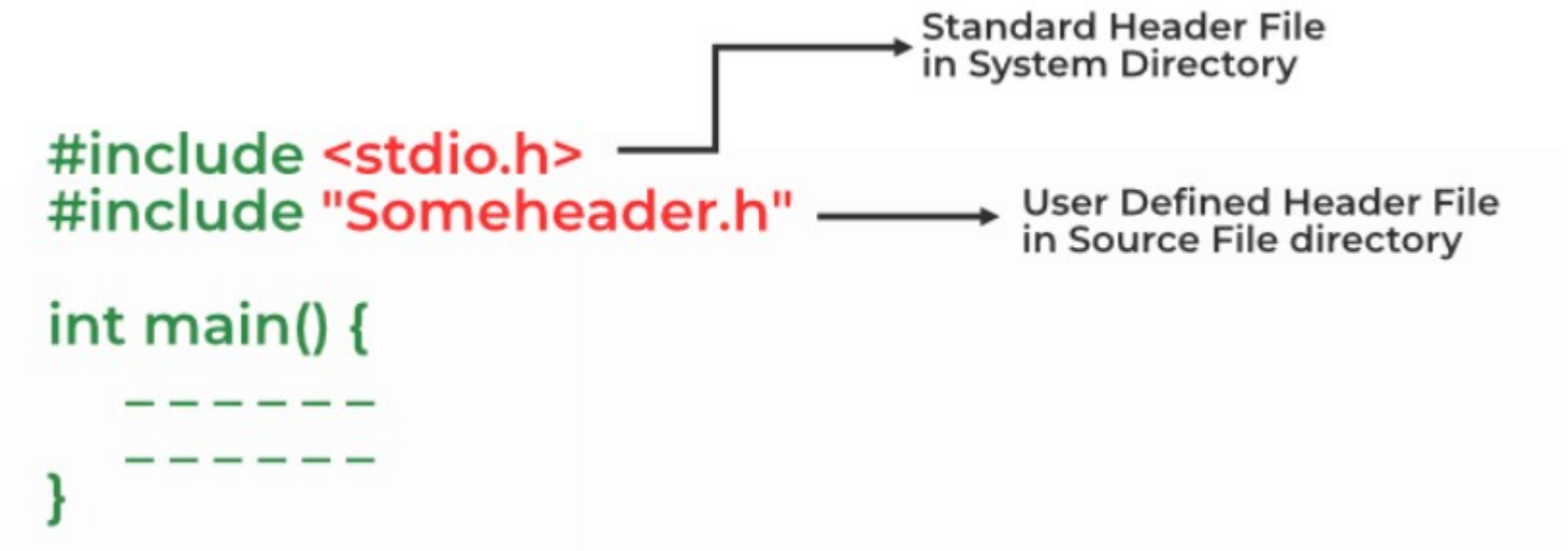
```
int sum (int x, int y)
{
    return x + y;
}
```

main.c

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

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

Works, but not a good practice


Standard Header File
in System Directory
#include <stdio.h>
#include "Someheader.h"
User Defined Header File
in Source File directory
int main() {
}

Multiple source files

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

functions.c

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

main.c

```
#include <stdio.h>

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

'extern' can be omitted (for function)

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

`gcc functions.c main.c`

Multiple source files

Can use a global variable defined in another file.

How  Declare it!

'extern' can be omitted (for function)

functions.c

```
//define global variable
int resu;

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

main.c

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

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

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

`gcc functions.c main.c`

Declaring external variables

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

- Declaring a global variable before using it, if it is defined in
  - library
  - later in the same source file
  - another source file of the program

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

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

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

"Call (pass) by Value" vs "Call (pass) by reference"

```
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 (pass) by reference"

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`

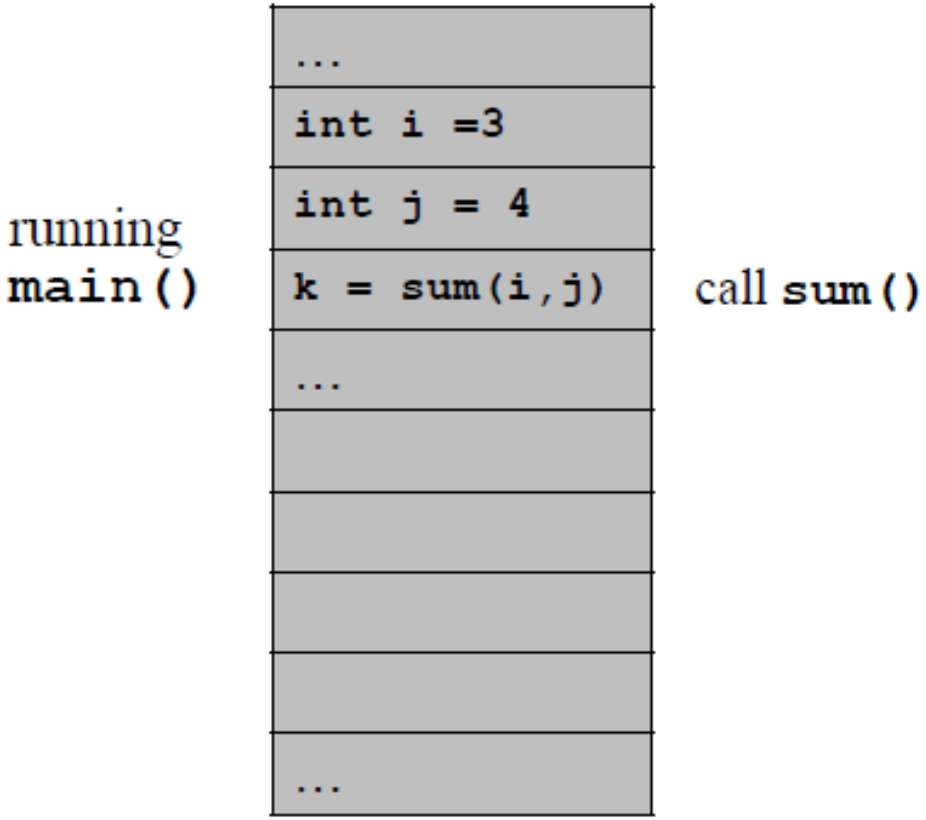
No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

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);
}
```

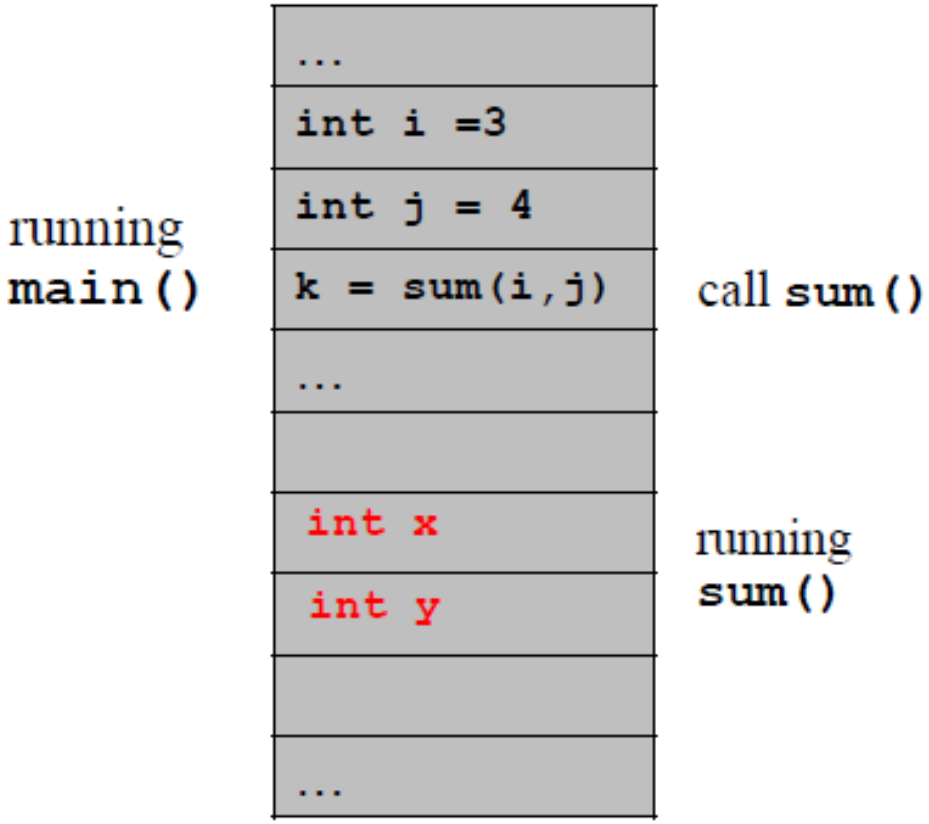


Call (pass)-by-Value

- 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);
}
```

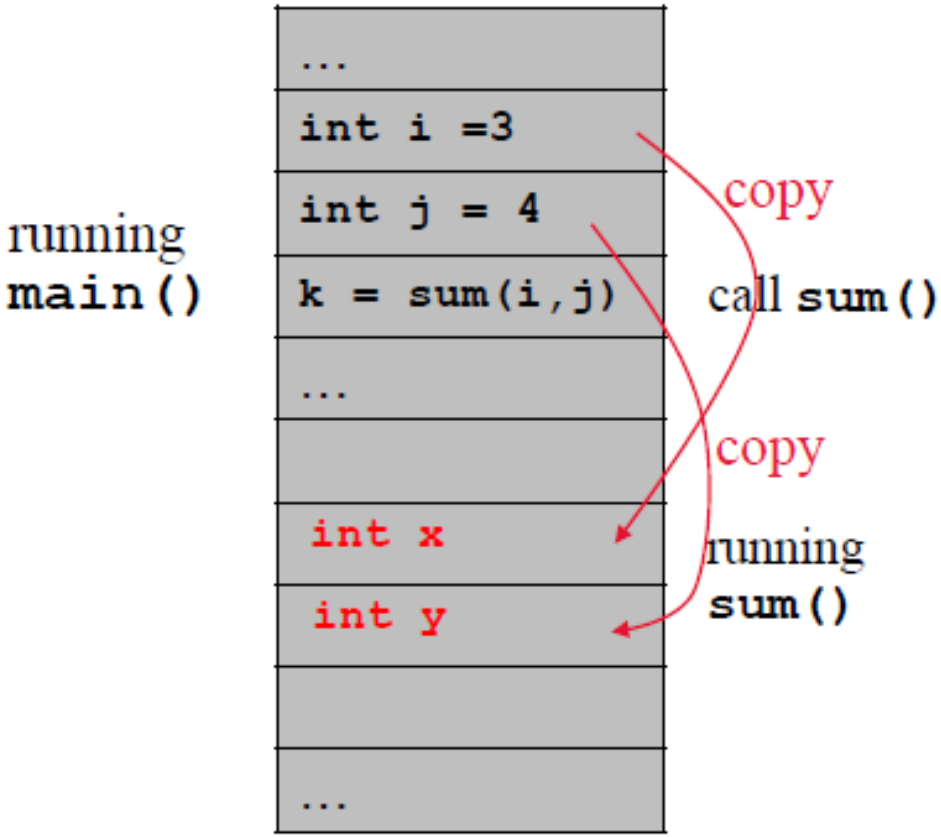


Call (pass)-by-Value

- 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);
}
```

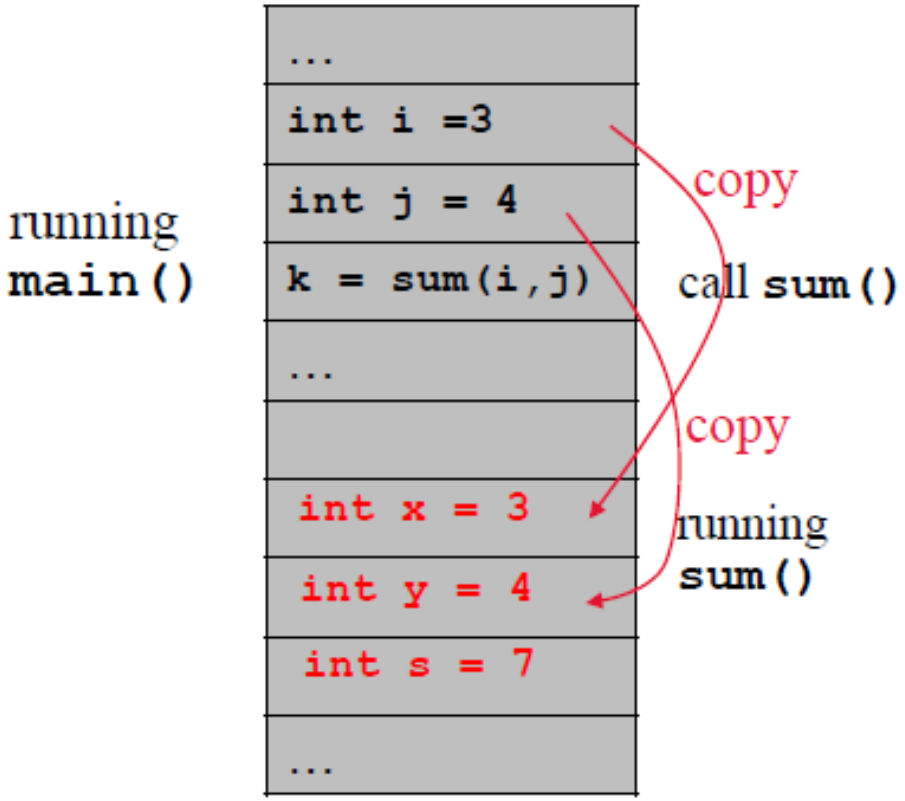


Call (pass)-by-Value

- 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);
}
```

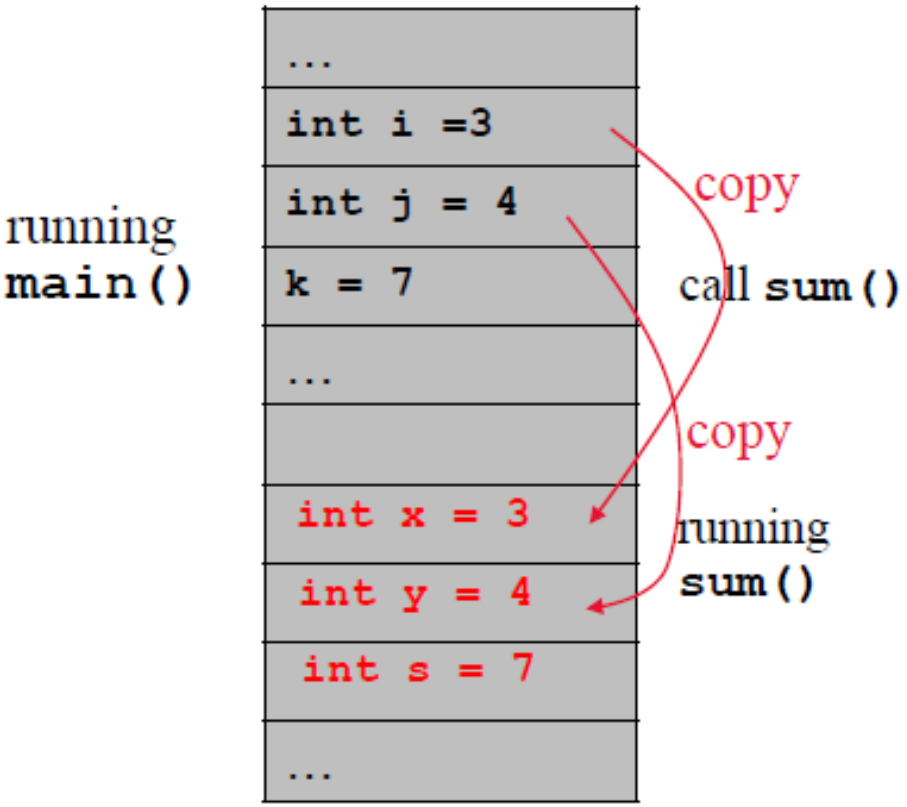


Call (pass)-by-Value

- 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);
}
```



- 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;
}
```

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

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

Call-by-Value does this code work?

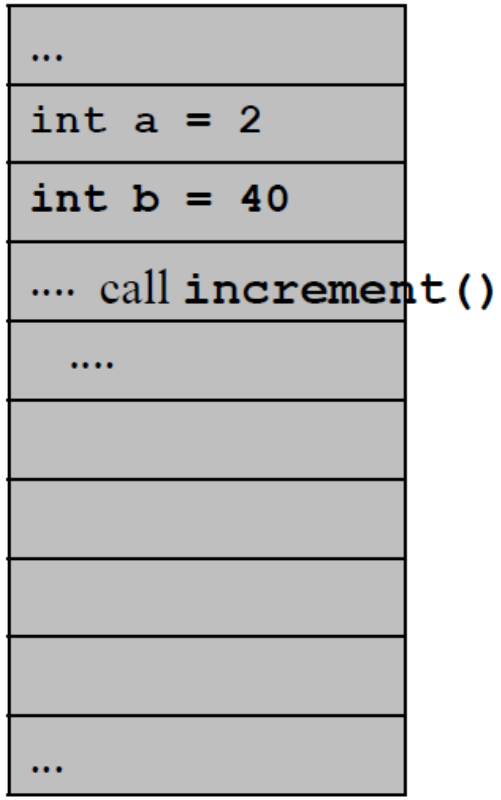
```
void increment(int x, int y)
{
    x ++;
    y += 10;

}
```

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

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

running
main()



Call-by-Value does this code work?

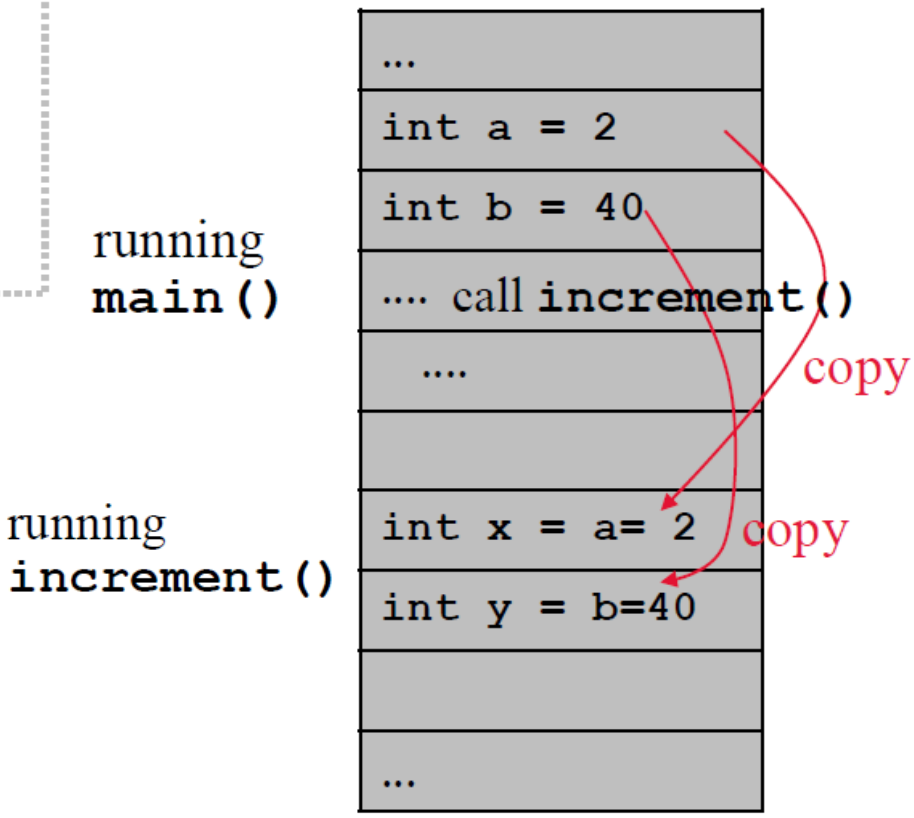
```
void increment(int x, int y)
{
    x ++;
    y += 10;

}
```

Pass by value !!!

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

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



Call-by-Value does this code work?

```
void increment(int x, int y)
{
    x ++;
    y += 10;
    printf("%d %d", x, y);
}
```

Pass by value !!!

3 50

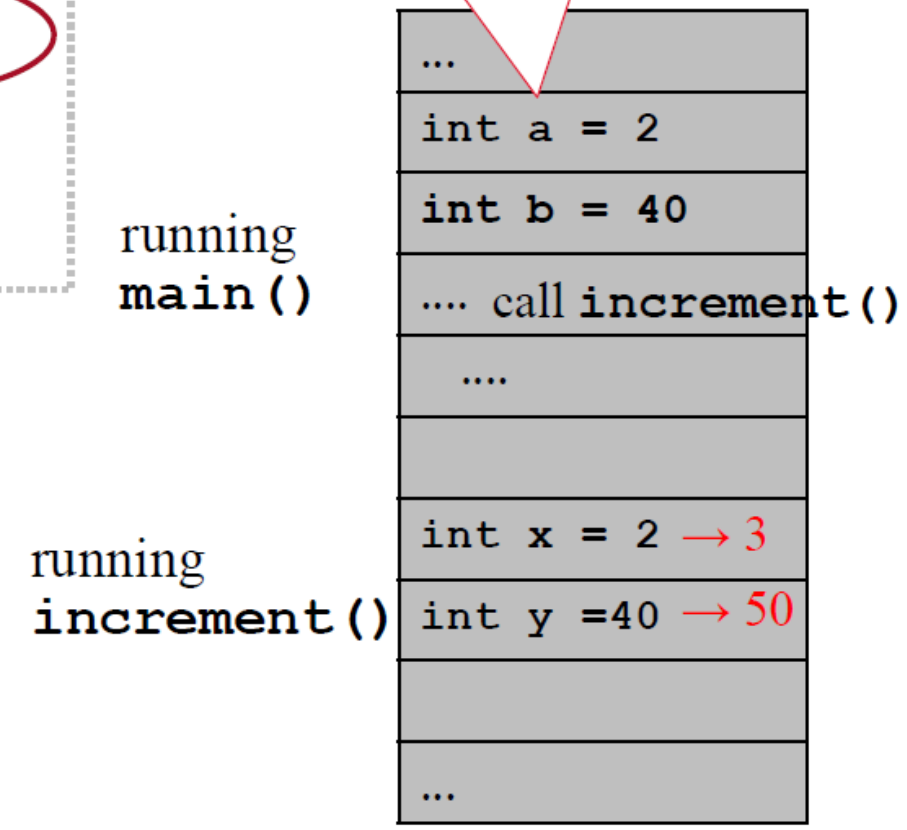
```
void main( ) {
    int a=2, b=40;

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

2 40

same in Java (static)

a b not incremented !

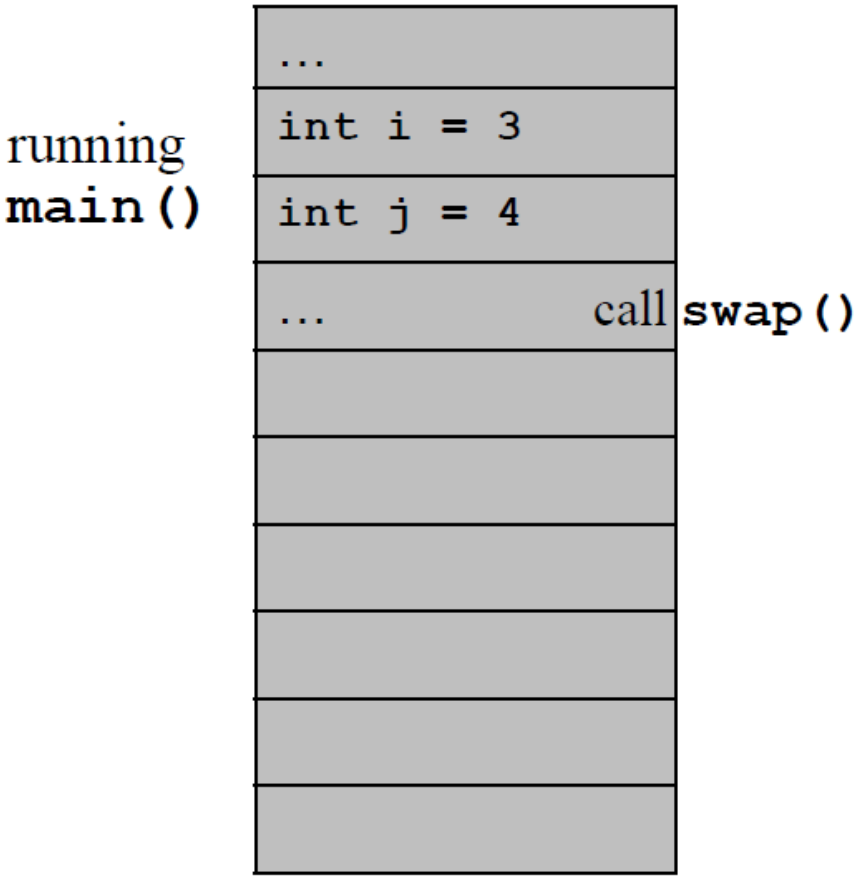


Call-by-Value does this code work?

```
#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);
}
```

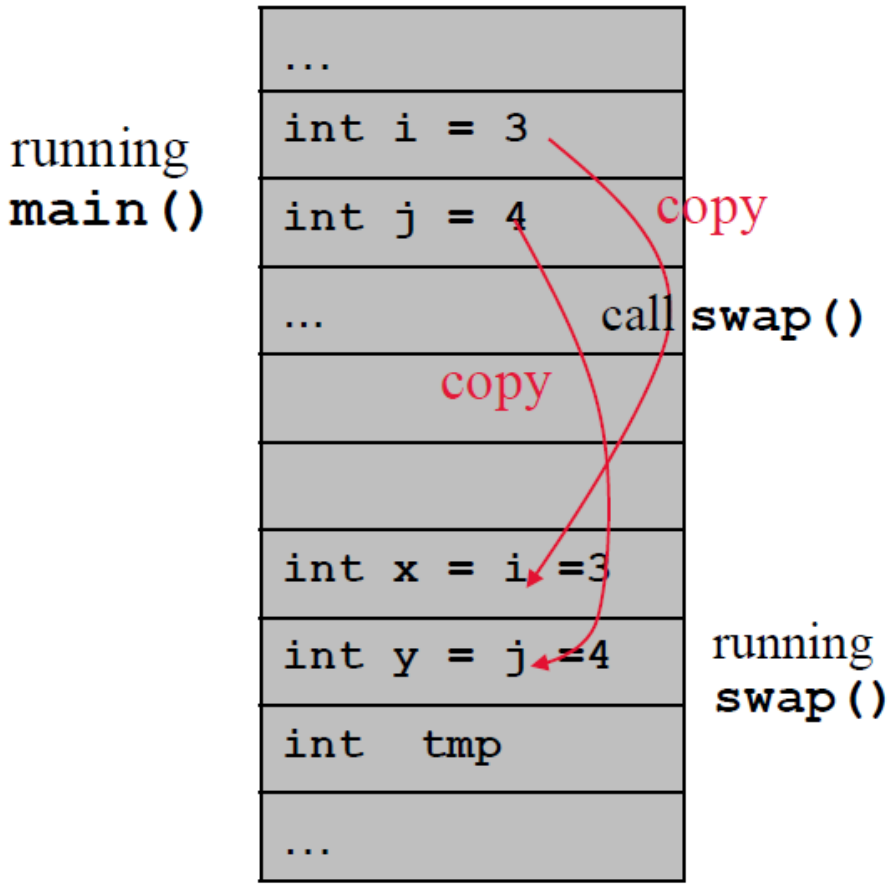


Call-by-Value does this code work?

```
#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);
}
```



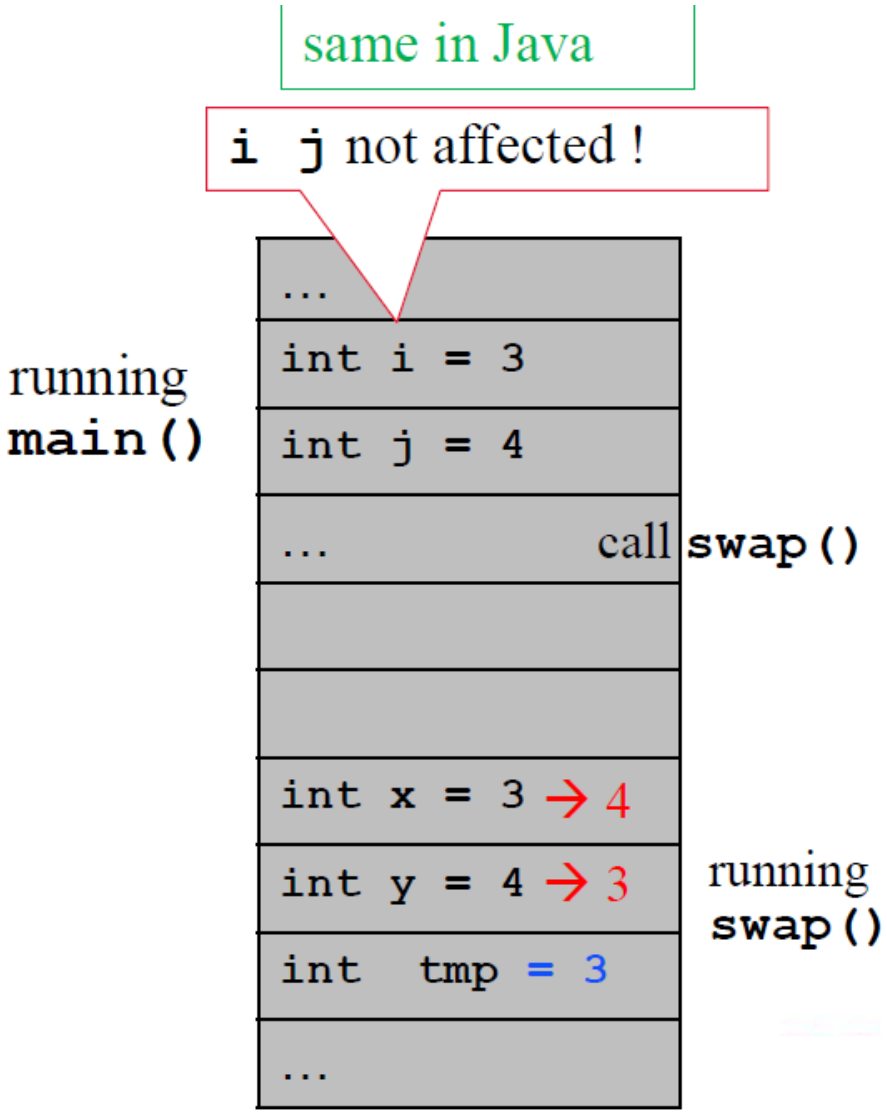
Call-by-Value does this code work?

```
#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);
}
```

3 4



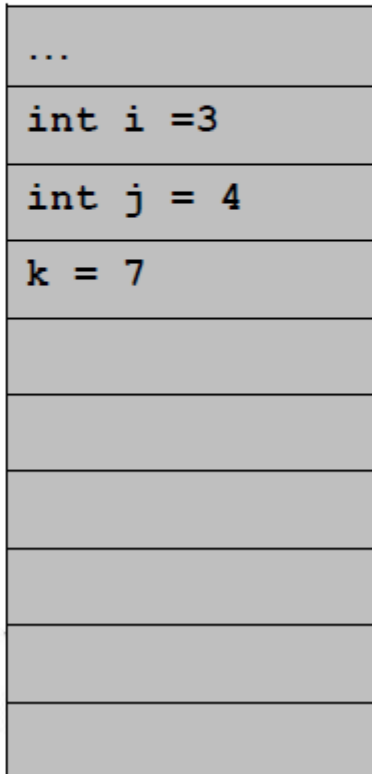
Lifetime –(storage duration) variables

- 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);
}
```

vanish after sum() returns

i j?



Lifetime –(storage duration) 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 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 betweer 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 external 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 external 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?

Static (hiding external 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?

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
```

- 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++;
}
```

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++;
}
```

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!**