ECE 220 Computer Systems & Programming

Lecture 9 – Functions in C & Run-Time Stack

February 12, 2019

ECE ILLINOIS

ILLINOIS

C Functions

Provides abstraction

- hide low-level details
- give high-level structure to program, easier to understand overall program flow
- enable separable, independent development
- reuse code

Structure of a function

- zero or multiple arguments passed in
- single result returned (optional)
- return value is always a particular type

Making a Function Call in C

```
#include <stdio.h>
/* our Factorial function prototype goes here */
int Fact(int n);

/* main function */
int main() {
    int number;
    int answer;

    printf("Enter a number: ");
    scanf("%d", &number);

    answer = Fact(number); /* function call */
    /* number - argument transferred from main to Factorial */
       answer - return value from Factorial to main */

    printf("factorial of %d is %d\n", number, answer);

    return 0;
}
```

Function “Fact”:

```
/* implementation of Factorial function goes here */
int Fact(int n) {
    int i, result=1; /* local variables in Factorial */

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

    return result; /* return value */
}
```

Function that does not return value:

```c
#include <stdio.h>

void PrintBanner();      /* Function declaration */

int main()
{
    PrintBanner();       /* Function call        */
    printf("A simple C program.\n");
    PrintBanner();
}

void PrintBanner()       /* Function definition  */
{
    printf("============================\n");
}
```

*Note: Functions do not necessarily have to be in the same file
(see the github example)

print.h ---> declares the function prototype
main.c ---> call the “print” function
print.c ---> print function

Example:

```c
#include <stdio.h>
void print_val(int x, int y);

void print_val(int x, int y)
{
    printf("first value:%d second value:%d\n", x, y);
}

int main()
{
    int z = 0;
    printf("value of z++:%d\n", z++);
    z = 0;
    printf("value of ++z:%d\n", ++z);

    int val = 10;
    print_val(val, val++);

    return 0;
}
```

How about the following “swap” function?

```c
#include <stdio.h>
void swap(int x, int y);
int main()
{
    int x = 1;
    int y = 2;

    printf("Before swap: x = %d, y = %d\n", x, y);
    swap(x, y);

    //Did the swap function work the way you expect?
    printf("After swap: x = %d, y = %d\n", x, y);

    return 0;
}

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

    temp = x;
    x = y;
    y = temp;
}
```

Possible Solution

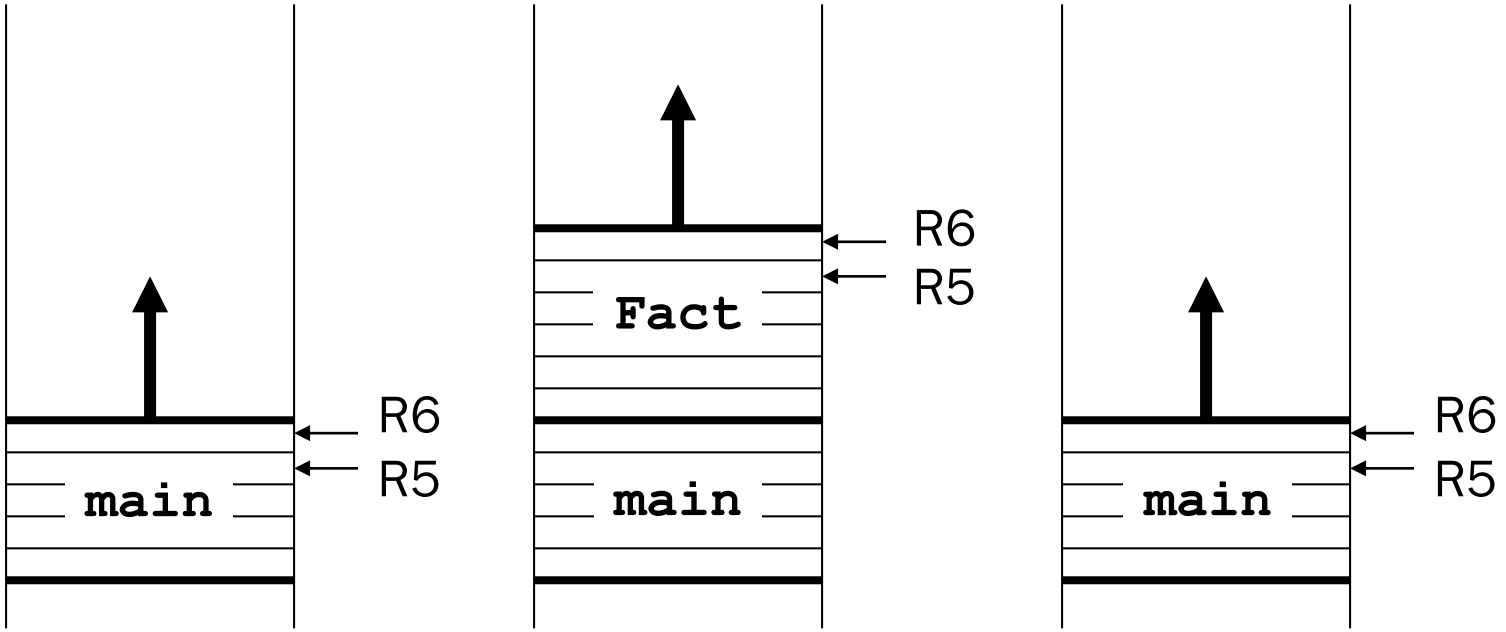
```c
#include <stdio.h>
void swap(int x, int y);
int z, k;

int main()
{
    int x = 1;
    int y = 2;

    printf("Before swap: x = %d, y = %d\n", x, y);
    swap(x, y);

    //Did the swap function work the way you expect?
    printf("After swap: x = %d, y = %d\n", z, k);

    return 0;
}

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

    temp = x;
    x = y;
    y = temp;
    z=x;
    k=y;
}
```

Possible Solution (advanced topics coming soon!)

```c
#include <stdio.h>

void swap(int *, int *);

int main()
{
    int x = 1;
    int y = 2;

    printf("Before swap: x = %d, y = %d\n", x, y);
    swap(&x, &y);

    //Did the swap function work the way you expect?
    printf("After swap: x = %d, y = %d\n", x, y);

    return 0;
}

void swap(int *x, int *y)
{
    int temp;

    temp = *x;
    *x = *y;
    *y = temp;
}
```

Run-Time Stack

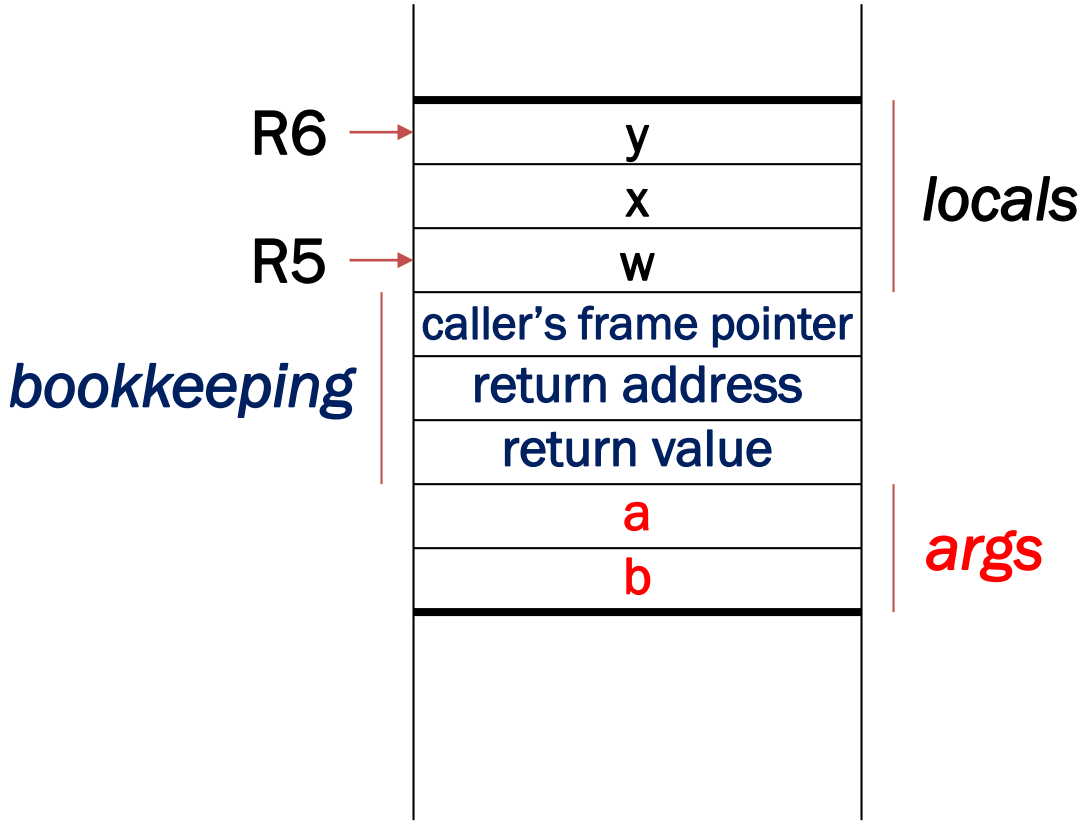
- R5 – **Frame Pointer**. It points to the beginning of a region of activation record that stores local variables for the current function.
- R6 – **Stack Pointer**. It points to the top most occupied location on the stack.
- Arguments are pushed to the stack _____.
- Local variables are pushed to the stack _____.



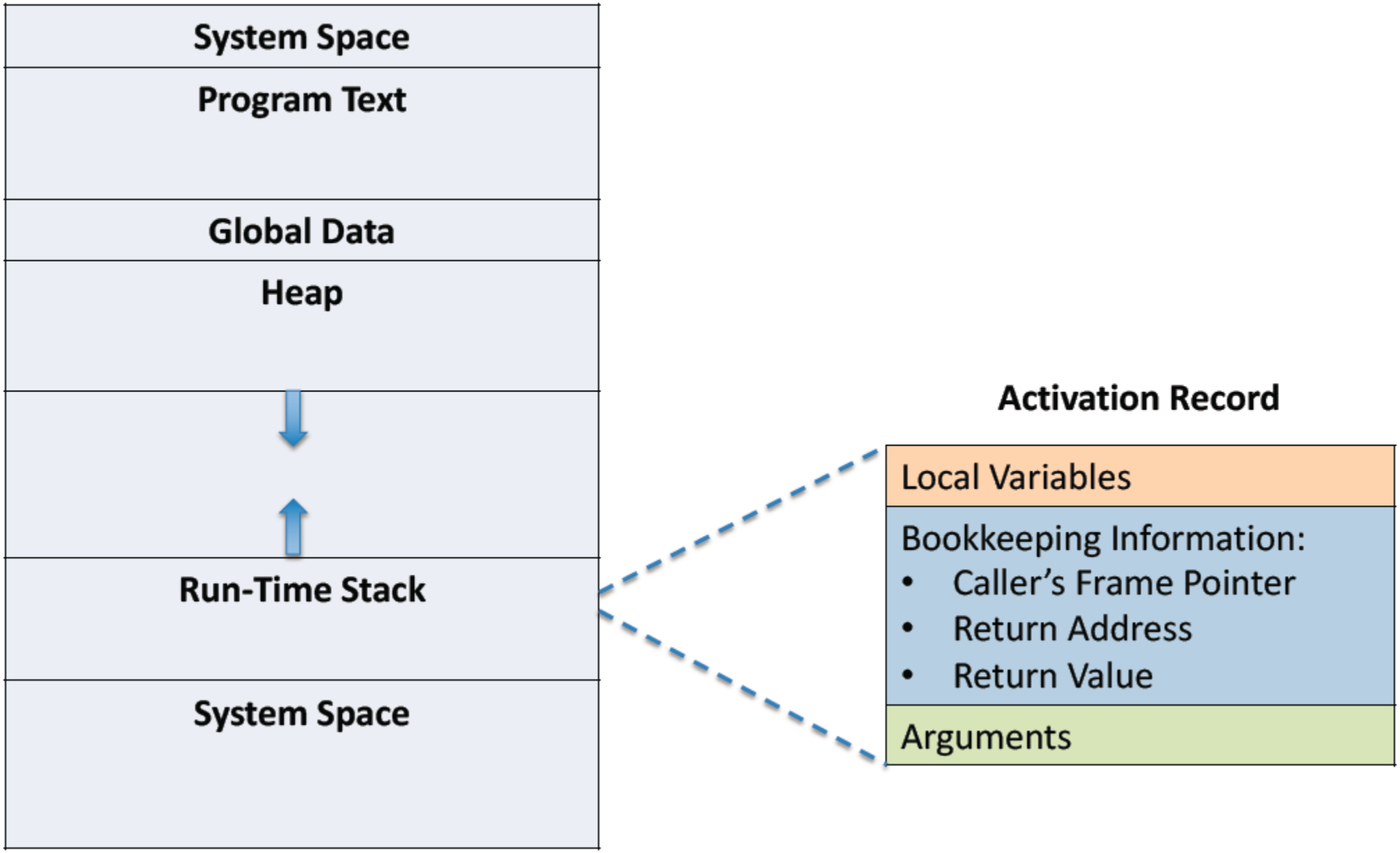
Before call *During call* *After call*

Activation Record

```
int func(int a, int b)
{
   int w, x, y;
   .
   .
   .
   return y;
}
```



LC-3 Memory Map



Stack Built-up and Tear-down

Caller function

1. *caller setup*: push callee's arguments onto stack
2. pass control to callee (invoke function)

Callee function

3. *callee setup*: (push bookkeeping info and local variables onto stack)
4. execute function
5. *callee teardown*: (pop local variables, caller's frame pointer, and return address from stack)
6. return to caller

Caller function

7. *caller teardown*: (pop callee's return value and arguments from stack)