ECE 220 Computer Systems & Programming

Lecture 10 – Implementing Function in C and Run-Time Stack

September 26, 2019

ECE ILLINOIS

ILLINOIS

Memory Allocation for Variables

- When a C compiler compiles a program, it keeps track of variables in a program using a *symbol table*. Whenever it finds a new variable declaration, it creates a new entry in its symbol table corresponding to the variable being declared.
- Symbol table contains enough information for the compiler to allocate storage in memory for the variable and for the generation of the proper sequence of machine code to access the value of that variable when it is used in the program.
- Each entry in the symbol table has 4 items:
  - Name of the variable
  - Its type
  - Place in memory the variable has been allocated storage
  - Identifier for the block in which the variable is declared

Example: Next slide

Example Code with Global and Local Variables:

```c
/* Include the standard I/O header file */
#include <stdio.h>

int inGlobal;     /* inGlobal is a global variable because */
                  /* it is declared outside of all blocks  */

int main()
{
  int inLocal;    /* inLocal, outLocalA, outLocalB are all */
  int outLocalA; /* local to main                          */
  int outLocalB;

  /* Initialize */
  inLocal  = 5;
  inGlobal = 3;

  /* Perform calculations */
  outLocalA = inLocal & ~inGlobal;
  outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);

  /* Print out results */
  printf("outLocalA = %d, outLocalB = %d\n", outLocalA, outLocalB);
}
```

Memory Allocation, Symbol Table

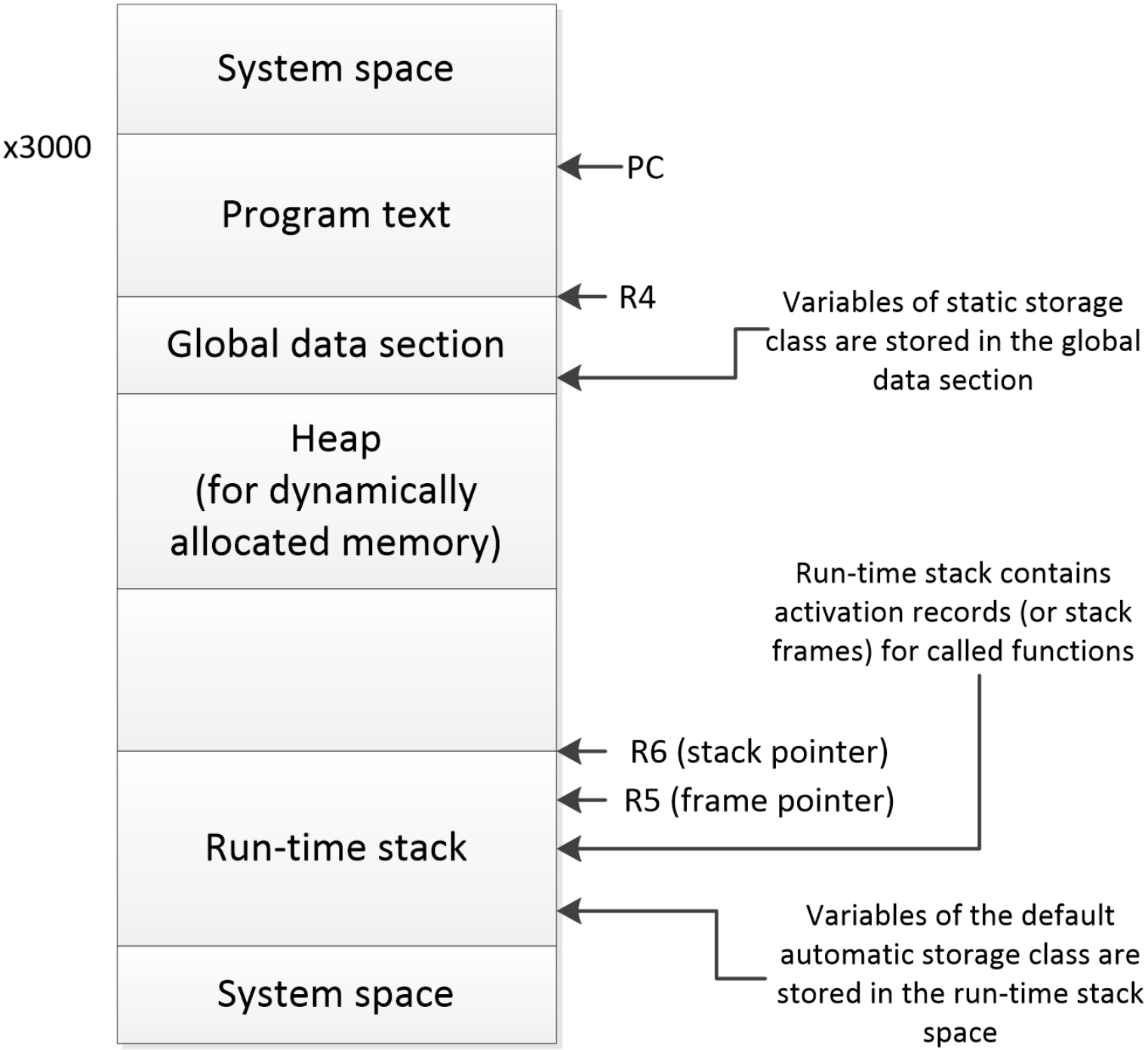
```c
/* Include the standard I/O header file */
#include <stdio.h>

int inGlobal;    /* inGlobal is a global variable because */
                 /* it is declared outside of all blocks  */

int main()
{
  int inLocal;   /* inLocal, outLocalA, outLocalB are all */
  int outLocalA; /* local to main                         */
  int outLocalB;

  /* Initialize */
  inLocal  = 5;
  inGlobal = 3;

  /* Perform calculations */
  outLocalA = inLocal & ~inGlobal;
  outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);

  /* Print out results */
  printf("outLocalA = %d, outLocalB = %d\n", outLocalA, outLocalB);
}
```

Identifier	Type	Location (as an offset)	Scope	Other info...
inGlobal	int	0	global	...
inLocal	int	0	main	...
outLocalA	int	-1	main	...
outLocalB	int	-2	main	...

LC-3 Memory:

- There are two regions of memory in which C variables are allocated storage space:
  - Global data section
    - Where all global variables are stored
    - Or more generally where variables of the static storage class are allocated
  - Run-time stack
    - Where local variables (of the default automatic storage class) are allocated
    - We will talk about it in the next lecture

LC-3 Memory Organization:



Registers R4, R5 and R6 :

- Note the use of registers R4, R5, and R6
  - R4 points to the first address of memory allocated for global variables
  - R5 contains frame pointer – memory region inside the function's activation record where local variables are stored
  - R6 contains address of the top of the run-time stack

Symbol Table and LC-3 Code

```c
/* Include the standard I/O header file */
#include <stdio.h>

int inGlobal;    /* inGlobal is a global variable because */
                 /* it is declared outside of all blocks  */

int main()
{
  int inLocal;   /* inLocal, outLocalA, outLocalB are all */
  int outLocalA; /* local to main                         */
  int outLocalB;

  /* Initialize */
  inLocal  = 5;
  inGlobal = 3;

  /* Perform calculations */
  outLocalA = inLocal & ~inGlobal;
  outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);

  /* Print out results */
  printf("outLocalA = %d, outLocalB = %d\n", outLocalA, outLocalB);
}
```

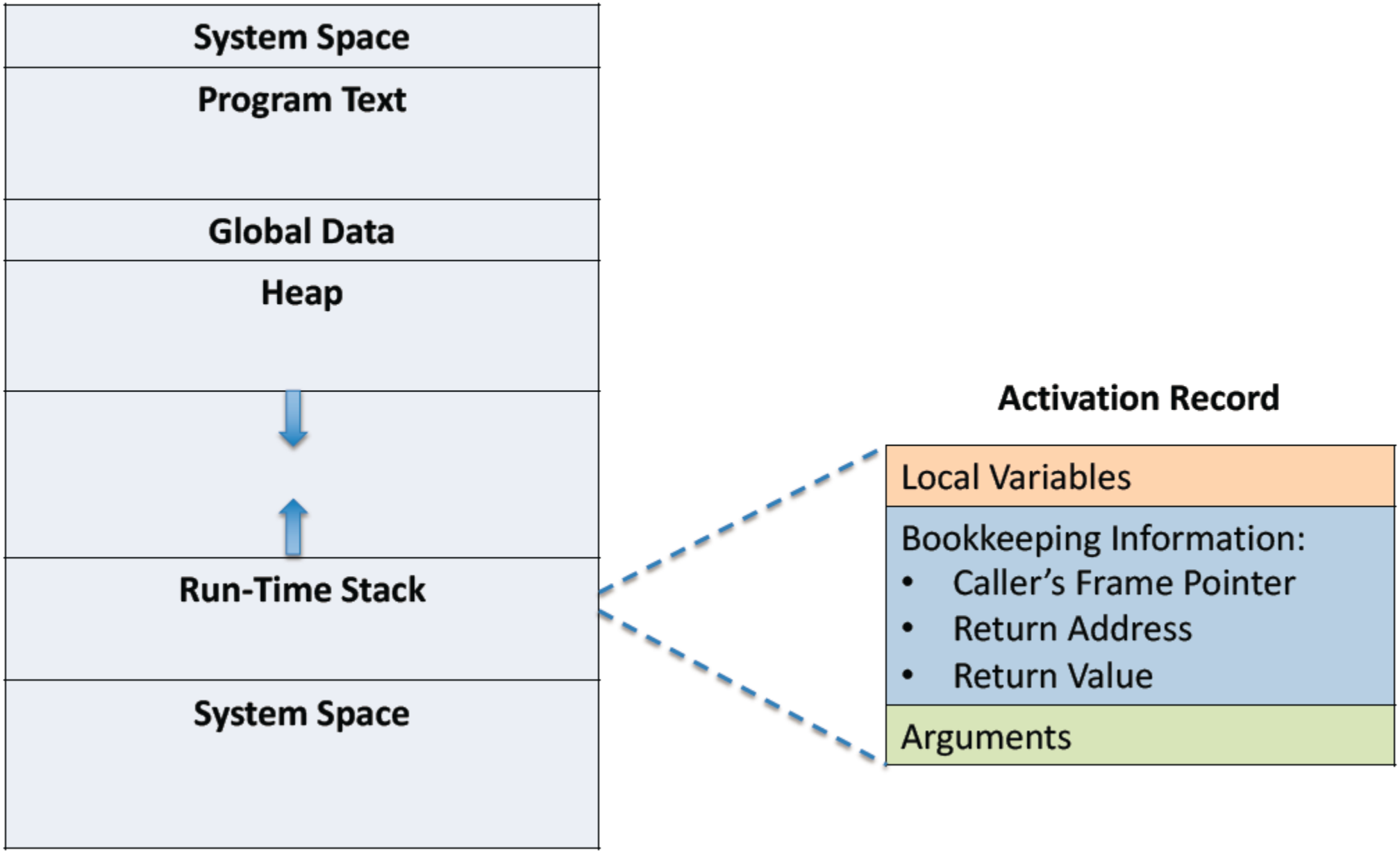
Identifier	Type	Location (as an offset)	Scope	Other info...
inGlobal	int	0	global	...
inLocal	int	0	main	...
outLocalA	int	–1	main	...
outLocalB	int	–2	main	...

```
1 main:
2 :
3 :
4 <startup code>
5 :
6 :
7 AND R0, R0, #0
8 ADD R0, R0, #5      ; inLocal is at offset 0
9 STR R0, R5, #0      ; inLocal = 5;
10
11 AND R0, R0, #0
12 ADD R0, R0, #3     ; inGlobal is at offset 0, in globals
13 STR R0, R4, #0     ; inGlobal = 3;
14
15 LDR R0, R5, #0     ; get value of inLocal
16 LDR R1, R4, #0     ; get value of inGlobal
17 NOT R1, R1         ; ~inGlobal
18 AND R2, R0, R1     ; calculate inLocal & ~inGlobal
19 STR R2, R5, #-1    ; outLocalA = inLocal & ~inGlobal;
20                    ; outLocalA is at offset -1
21
22 LDR R0, R5, #0     ; get value of inLocal
23 LDR R1, R4, #0     ; get value of inGlobal
24 ADD R0, R0, R1     ; calculate inLocal + inGlobal
25
26 LDR R2, R5, #0     ; get value of inLocal
27 LDR R3, R4, #0     ; get value of inGlobal
28 NOT R3
29 ADD R3, R3, #1     ; calculate -inGlobal
30
31 ADD R2, R2, R3     ; calculate inLocal - inGlobal
32 NOT R2
33 ADD R2, R2, #1     ; calculate -(inLocal - inGlobal)
34
35 ADD R0, R0, R2     ; (inLocal + inGlobal) - (inLocal - inGlobal)
36 STR R0, R5, #-2    ; outLocalB = ...
37                    ; outLocalB is at offset -2
38 :
39 :
40 <code for calling the function printf>
41 :
42 :
```

Activation Records:

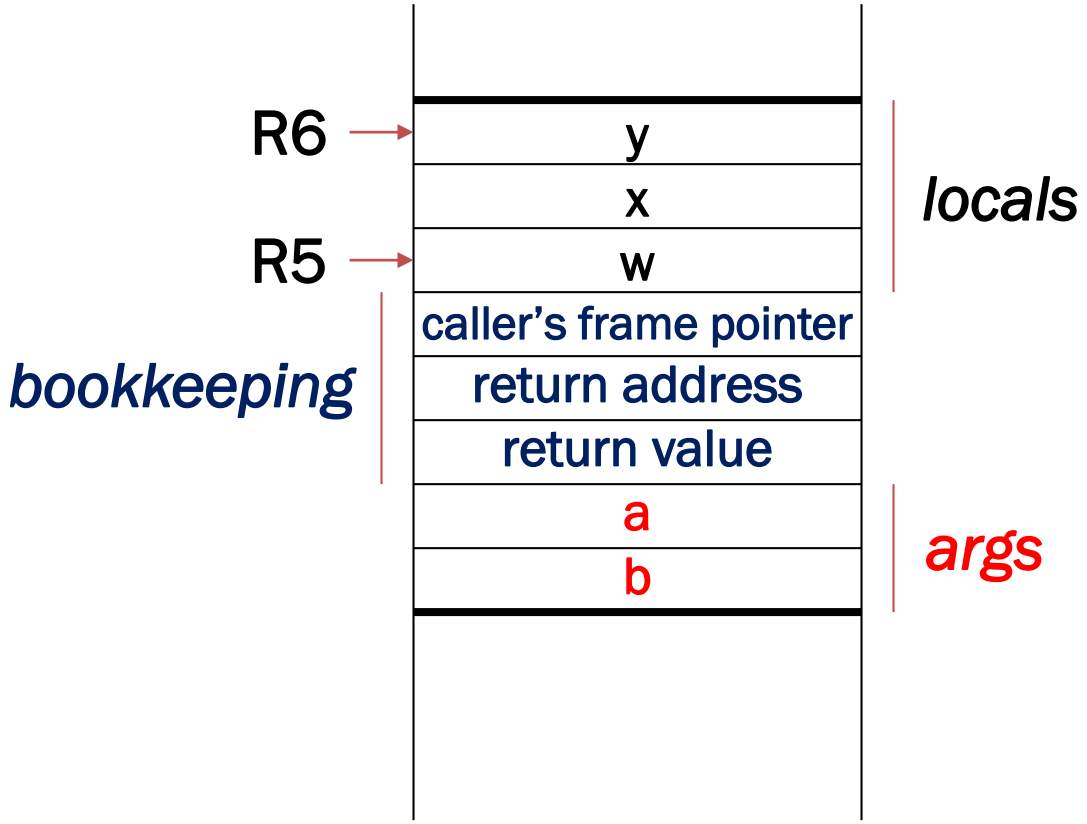
- When we call a function in C, its *activation record* is pushed onto the run-time stack
- Whenever a function completes, its *activation record* is popped off the run-time stack
- Function's activation record contains all the data local to the function involved in the function invocation, execution, and transfer of the results back to the calling function

LC-3 Memory Map



Activation Record

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



Run-Time Stack

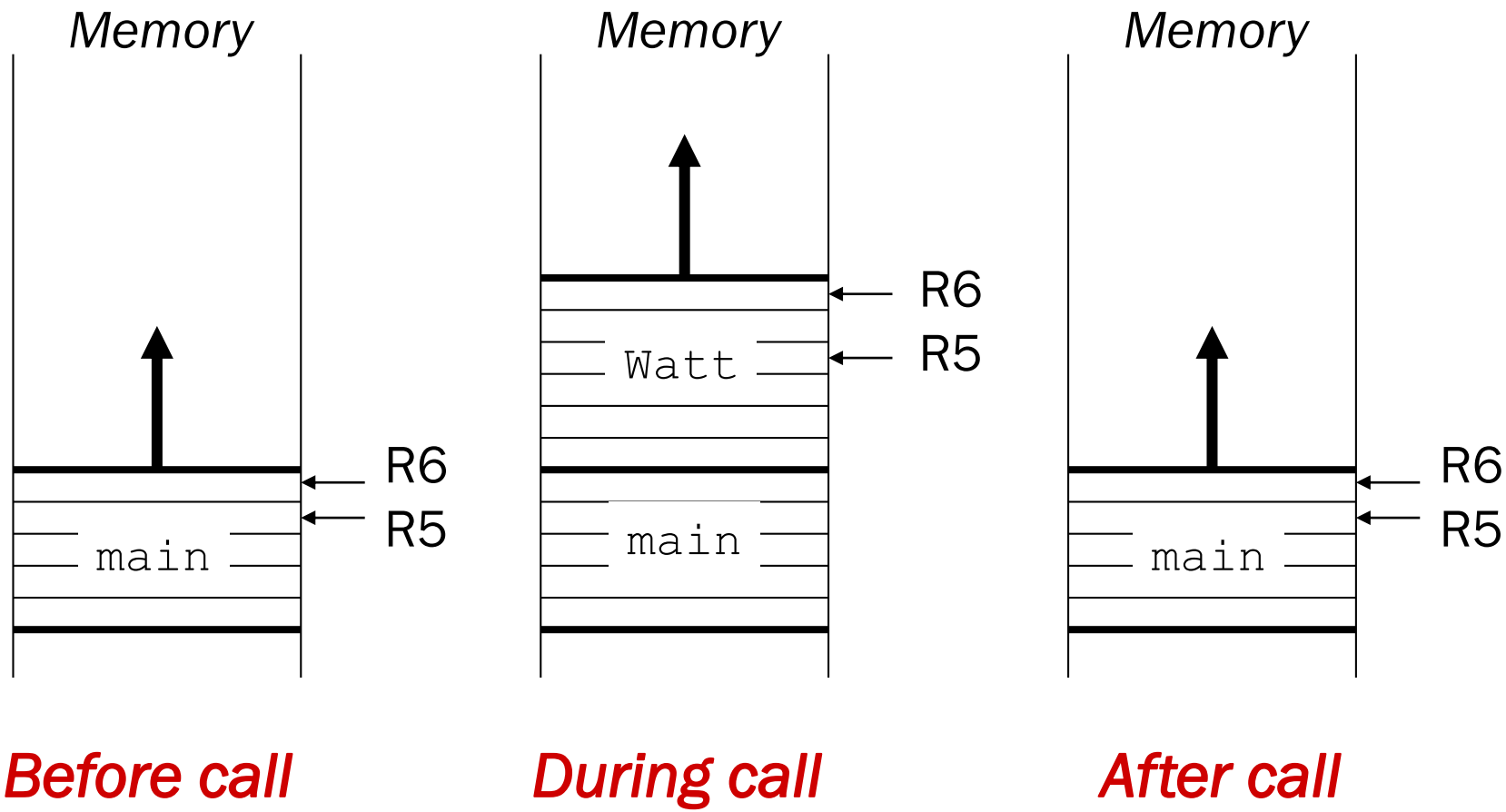
Recall that local variables are stored on the run-time stack in an *activation record*

Frame pointer (R5) points to the beginning of a region of activation record that stores local variables for the current function

When a new function is called, its activation record is pushed on the stack;

when it returns, its activation record is popped off of the stack.

Run-Time Stack



Stack Built-up and Tear-down during Function Call and Return

Caller function

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

Callee function

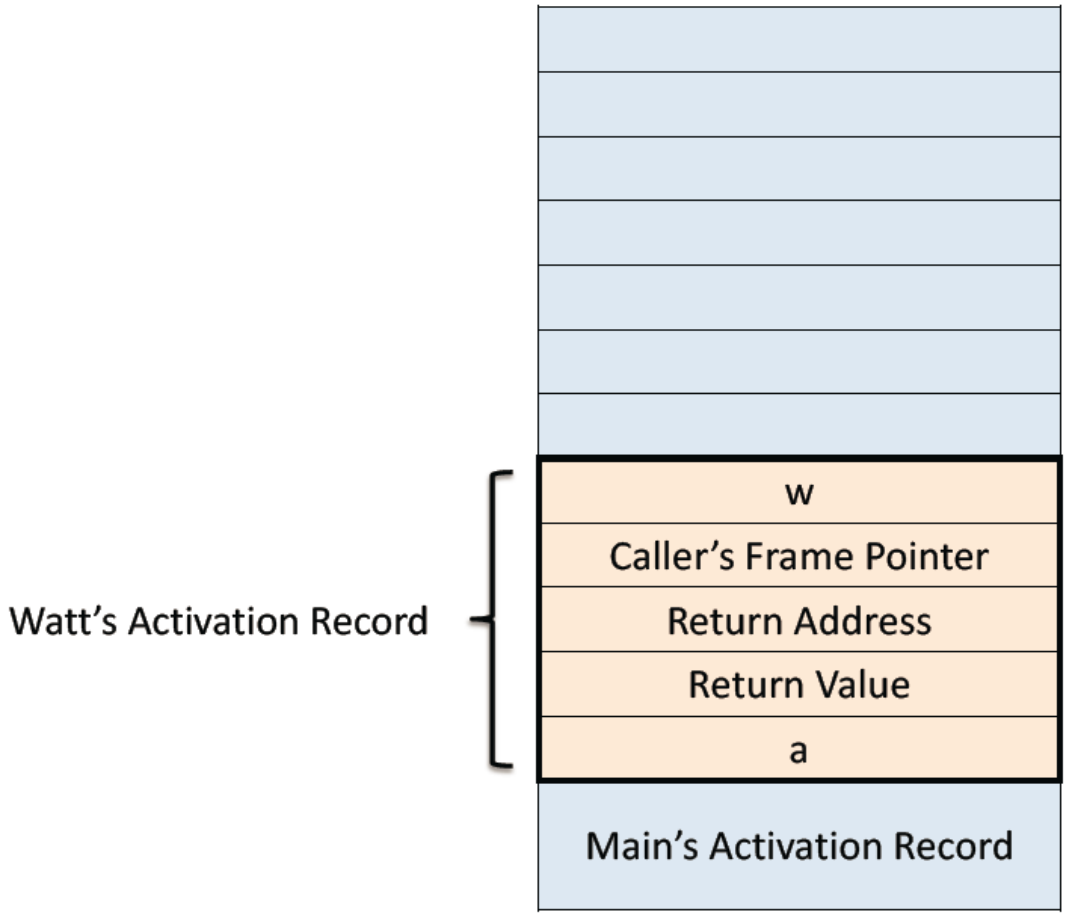
3. *callee setup*: (push bookkeeping info and local variables onto stack)
4. execute function
5. *callee teardown*: (pop local variables, caller's frame pointer, and return address from stack)
6. return to caller

Caller function

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

Example Function Call

```
int Volta(int q, int r)
{
   int k;
   int m;
   ...
   return k;
}

int Watt(int a)
{
   int w;
   ...
   w = Volta(w,10);
   ...
   return w;
}
```

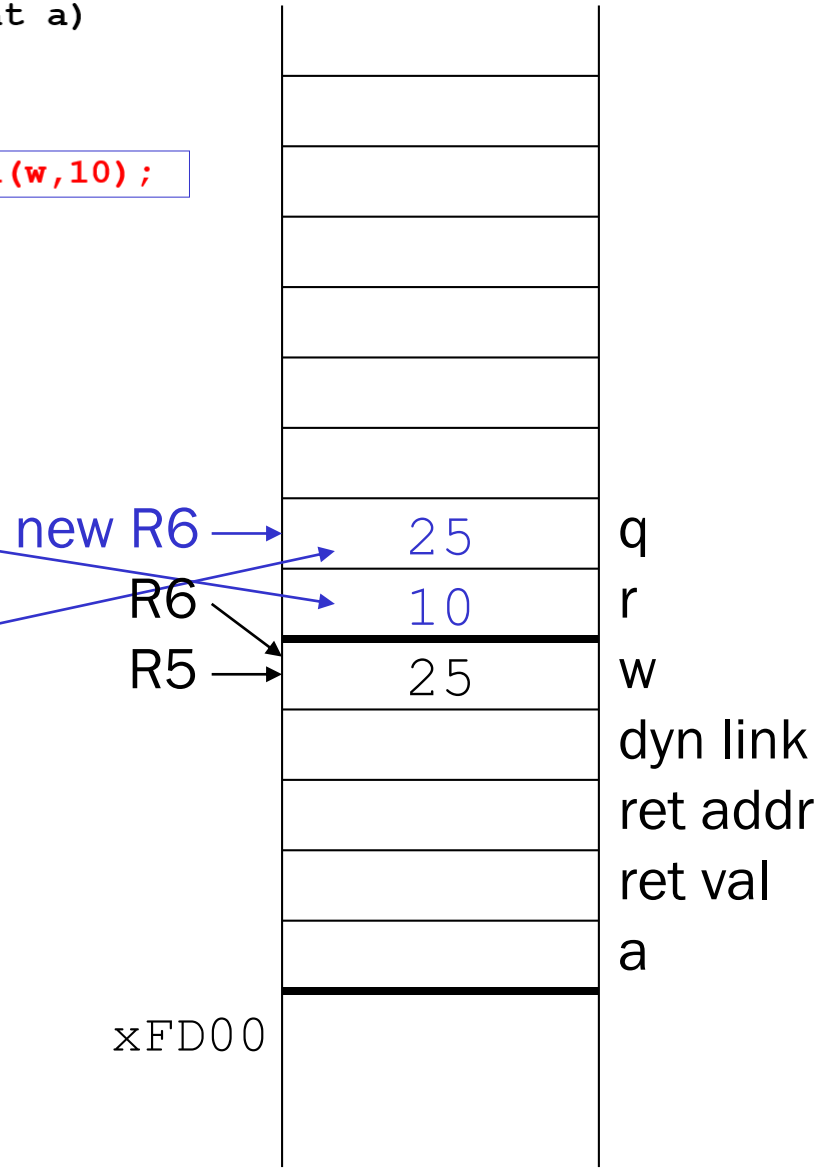


Calling the Function

```
w = Volta(w, 10);
```

```
; push second arg
AND  R0, R0, #0
ADD  R0, R0, #10
ADD  R6, R6, #-1
STR  R0, R6, #0
; push first argument
LDR  R0, R5, #0
ADD  R6, R6, #-1
STR  R0, R6, #0

; call subroutine
JSR  Volta
```

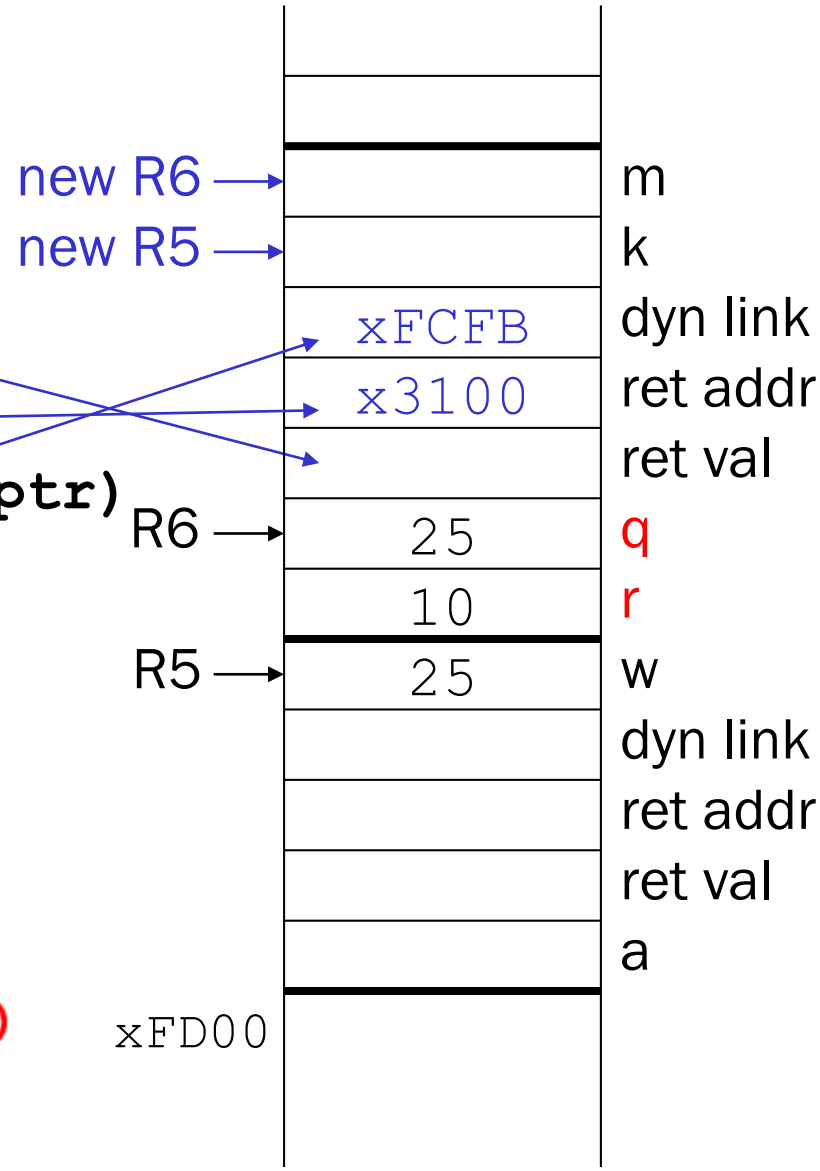
```
int Watt(int a)
{
  int w;
  ...
  w = Volta(w,10);
  ...
  return w;
}
```



Note: Caller needs to know number and type of arguments, doesn't know about local variables.

Starting the Callee Function

```
; leave space for return value
ADD  R6, R6, #-1
; push return address
ADD  R6, R6, #-1
STR  R7, R6, #0
; push dyn link (caller's frame ptr)
ADD  R6, R6, #-1
STR  R5, R6, #0
; set new frame pointer
ADD  R5, R6, #-1
; allocate space for locals
ADD  R6, R6, #-2
```



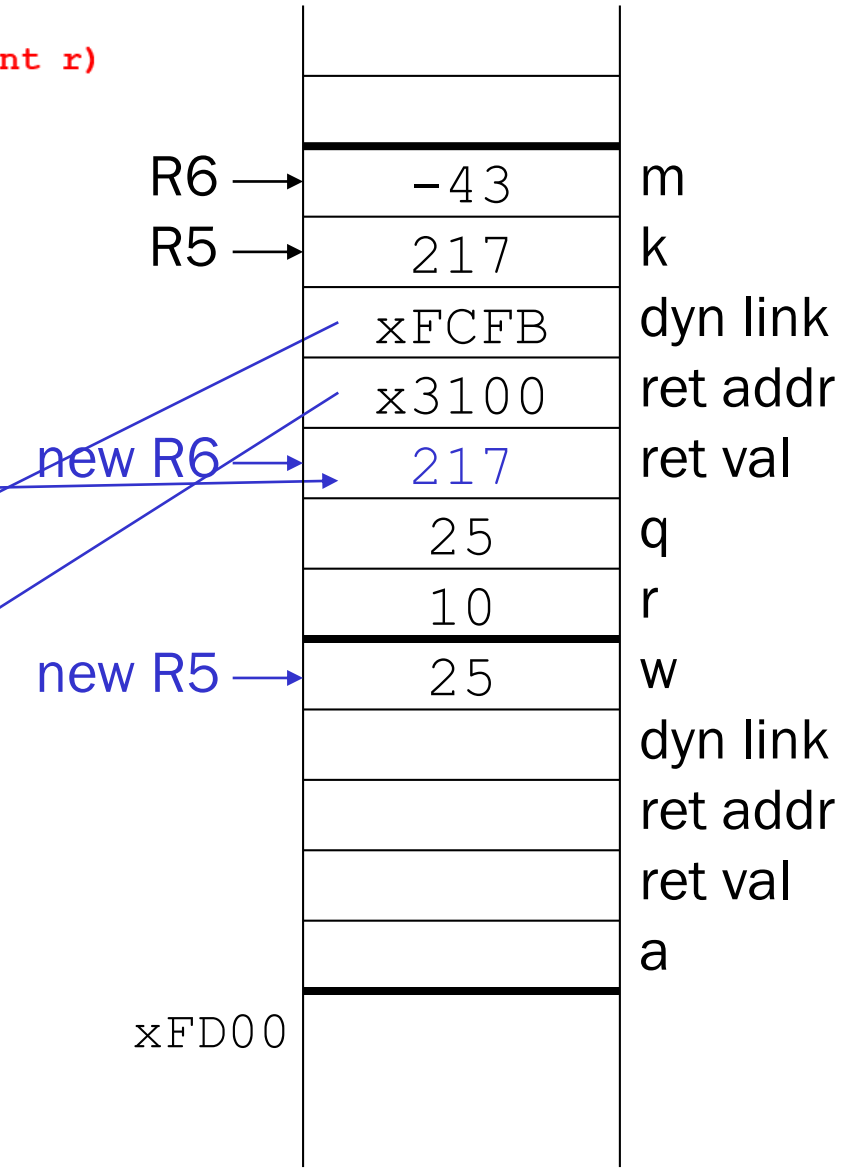

```
int Volta(int q, int r)
{
   int k;
   int m;
   ...
   return k;
}
```

Ending the Callee Function

```
return k;
```

```
int Volta(int q, int r)
{
  int k;
  int m;
  ...
  return k;
}
```

```
; copy k into return value
LDR  R0, R5, #0
STR  R0, R5, #3
; pop local variables
ADD  R6, R5, #1
; pop dynamic link (into R5)
LDR  R5, R6, #0
ADD  R6, R6, #1
; pop return addr (into R7)
LDR  R7, R6, #0
ADD  R6, R6, #1
; return control to caller
RET
```

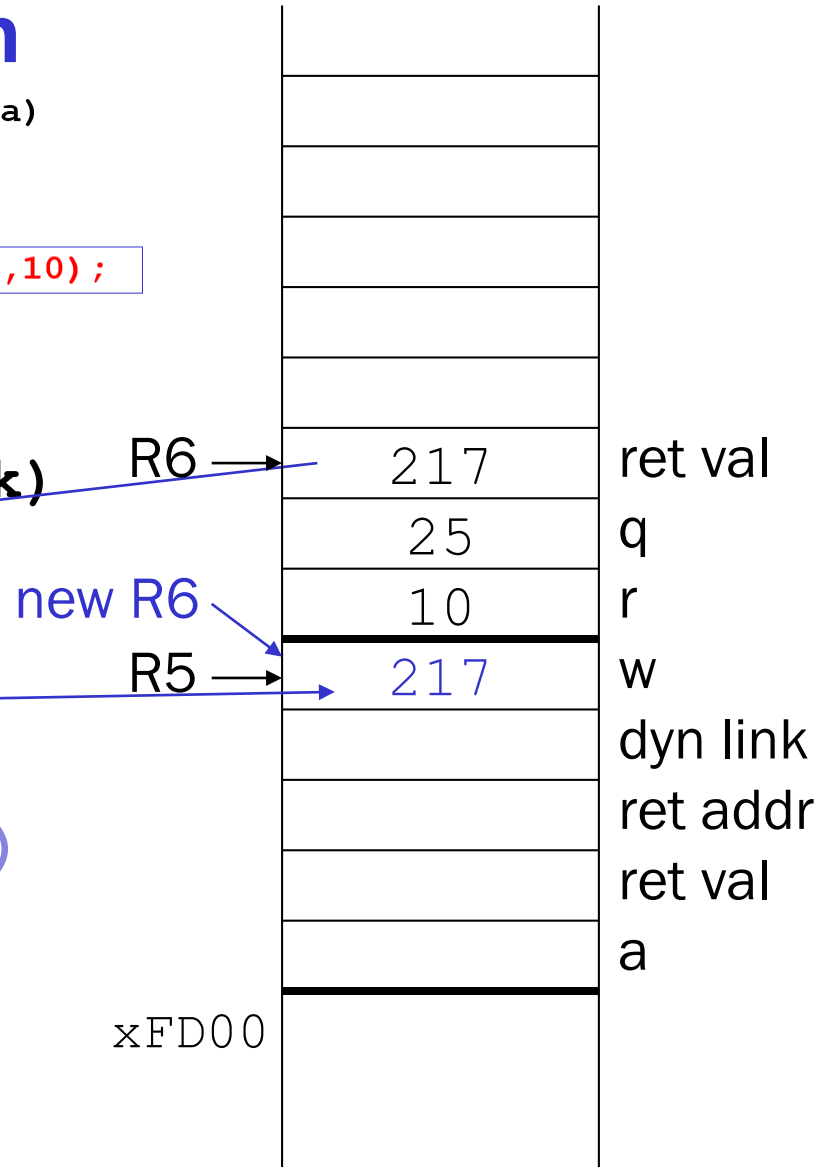


Resuming the Caller Function

```
w = Volta(w,10);
```

```
int Watt(int a)
{
  int w;
  ...
  w = Volta(w,10);
  ...
  return w;
}
```

```
JSR  Volta

; load return value (top of stack)
LDR  R0, R6, #0
; perform assignment
STR  R0, R5, #0

; pop return value; W=Volta(W,10)
ADD  R6, R6, #1

; pop arguments
ADD  R6, R6, #2
```



Summary of LC-3 Function Call Implementation

1. **Caller pushes arguments (last to first).**
2. **Caller invokes subroutine (JSR).**
3. **Callee allocates return value, pushes R7 and R5.**
4. **Callee allocates space for local variables.**
5. **Callee executes function code.**
6. **Callee stores result into return value slot.**
7. **Callee pops local vars, pops R5, pops R7.**
8. **Callee returns (JMP R7).**
9. **Caller loads return value and pops arguments.**
10. **Caller resumes computation…**

Run-Time Stack Exercise

Adopted from Prof. Yuting's lecture notes

```
#include <stdio.h>
int Fact(int n);

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

    printf("Enter a number: ");
    scanf("%d", &number);
    answer = Fact(number);
    printf("factorial of %d is %d\n", number, answer);
    return 0;
}
```

```
/* Function definition of Factorial function */
int Fact(int n) {
    int i, result=1;

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

    return result;
}
```

x3FF0	
x3FF1	
x3FF2	
x3FF3	
x3FF4	
x3FF5	
x3FF6	
x3FF7	
x3FF8	
x3FF9	
x3FFA	
x3FFB	
x3FFC	
x3FFD	
x3FFE	
x3FFF	answer
x4000	number