

Chapter 14

Functions

Function

Smaller, simpler, subcomponent of program

Provides abstraction

- **hide low-level details**
- **give high-level structure to program, easier to understand overall program flow**
- **enables separable, independent development**

C functions

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

In other languages, called procedures, subroutines, ...

Example of High-Level Structure

```
main()  
{  
    SetupBoard();    /* place pieces on board */  
  
    DetermineSides();    /* choose black/white */  
  
    /* Play game */  
    do {  
        WhitesTurn();  
        BlacksTurn();  
    } while (NoOutcomeYet());  
}
```

Structure of program
is evident, even without
knowing implementation.

Functions in C

Declaration (also called prototype)

```
int Factorial(int n);
```

type of
return value

name of
function

types of all
arguments

Function call -- used in expression

```
a = x + Factorial(f + g);
```

1. evaluate arguments

2. execute function


3. use return value in expression

Function Definition

State type, name, types of arguments

- must match function declaration
- give name to each argument (doesn't have to match declaration)

```
int Factorial(int n)
{
    int i;
    int result = 1;
    for (i = 1; i <= n; i++)
        result *= i;
    return result;
}
```



gives control back to
calling function and
returns value

Why Declaration?

Since function definition also includes return and argument types, why is declaration needed?

- **Use might be seen before definition.**
Compiler needs to know return and arg types and number of arguments.
- **Definition might be in a different file, written by a different programmer.**
 - include a "header" file with function declarations only
 - compile separately, link together to make executable

Example

```
double ValueInDollars(double amount, double rate);
```

 **declaration**

```
main()
```

```
{
```

```
...
```

```
dollars = ValueInDollars(francs,  
                        DOLLARS_PER_FRANC);
```

```
printf("%f francs equals %f dollars.\n",  
       francs, dollars);
```

```
...
```

```
}
```

 **function call (invocation)**

 **definition**

```
double ValueInDollars(double amount, double rate)
```

```
{
```

```
    return amount * rate;
```

```
}
```

Implementing Functions: Overview

Activation record

- information about each function, including arguments and local variables
- stored on run-time stack

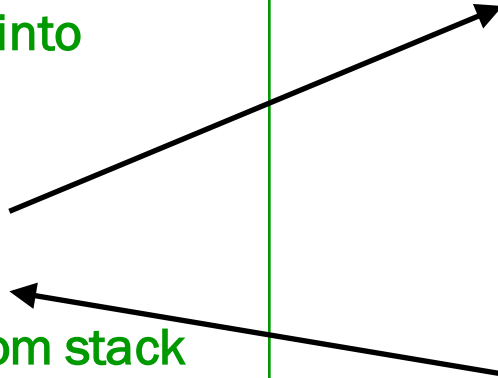
Calling function

push new activation
record
copy values into
arguments
call function

get result from stack

Called function

execute code
put result in
activation record
pop activation record
from stack
return



Run-Time Stack

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

— R6

— R5

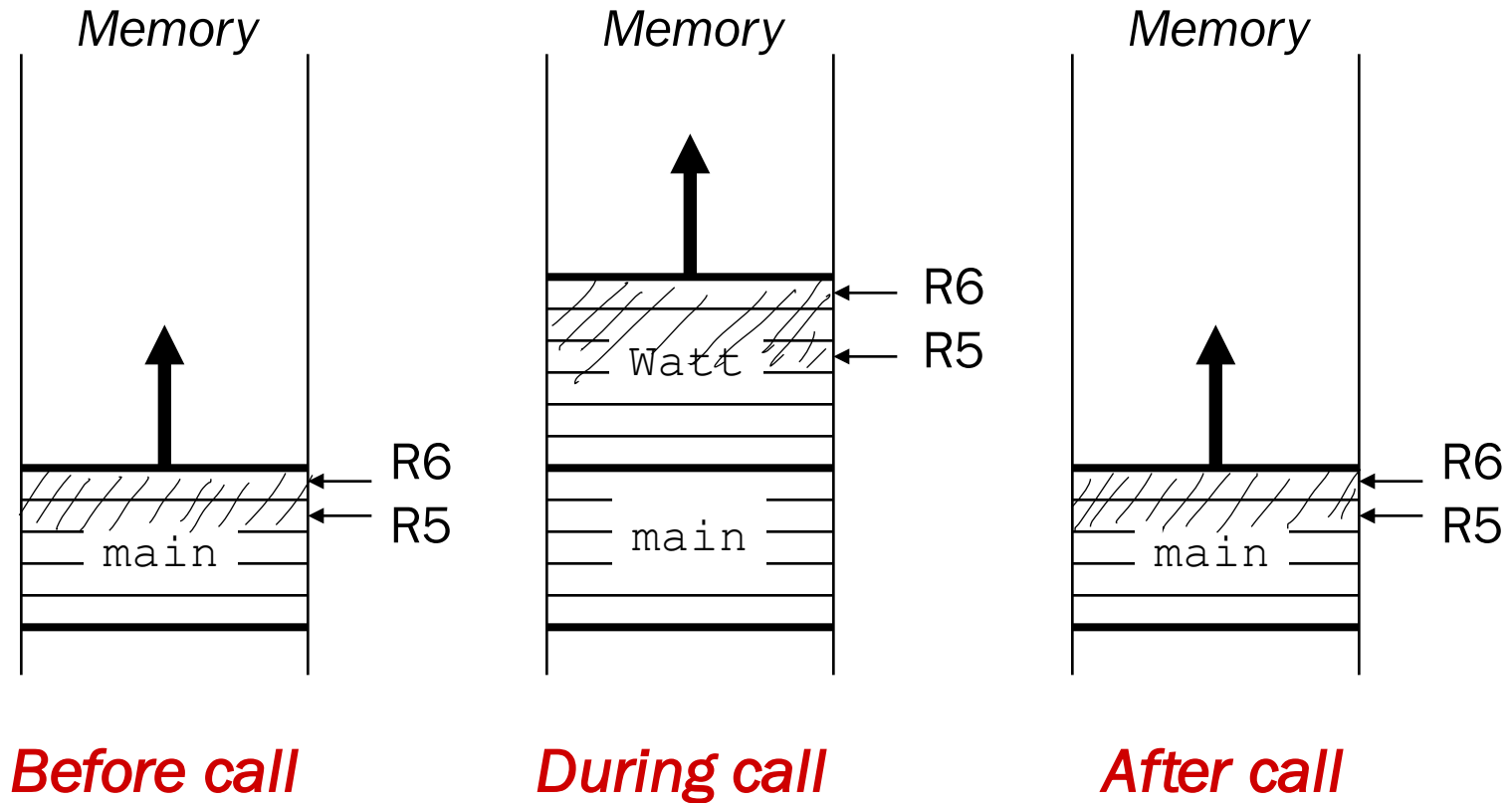
R7

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.

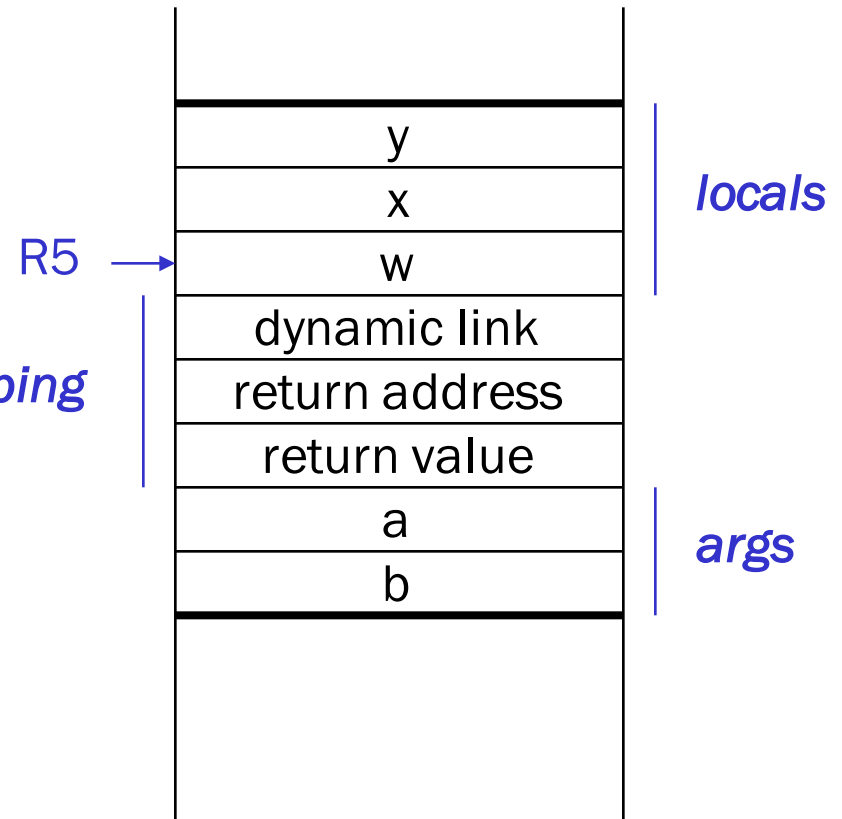
Run-Time Stack



Activation Record

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

Name	Type	Offset	Scope
a	int	4	NoName
b	int	5	NoName
w	int	0	NoName
x	int	-1	NoName
y	int	-2	NoName



Activation Record Bookkeeping

Return value

- space for value returned by function
- allocated even if function does not return a value

Return address

- save pointer to next instruction in calling function
- convenient location to store R7 in case another function (JSR) is called

Dynamic link

- caller's frame pointer
- used to pop this activation record from stack

Handwritten diagram showing a stack frame structure with labels: Caller's frame pointer, R7, and Dynamic link.

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

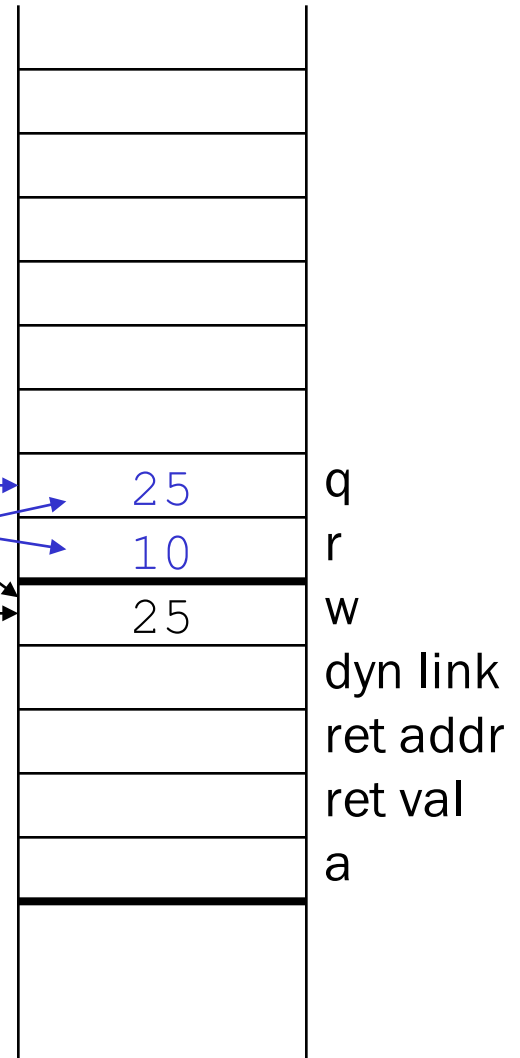
JSR Volta

new R6 →

R6 →

R5 →

xFD00



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

new R6 →

new R5 →

xFCFB

x3100

R6 →

25

10

R5 →

25

xFD00

m

k

dyn link

ret addr

ret val

q

r

w

dyn link

ret addr

ret val

a

Ending the Callee Function

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

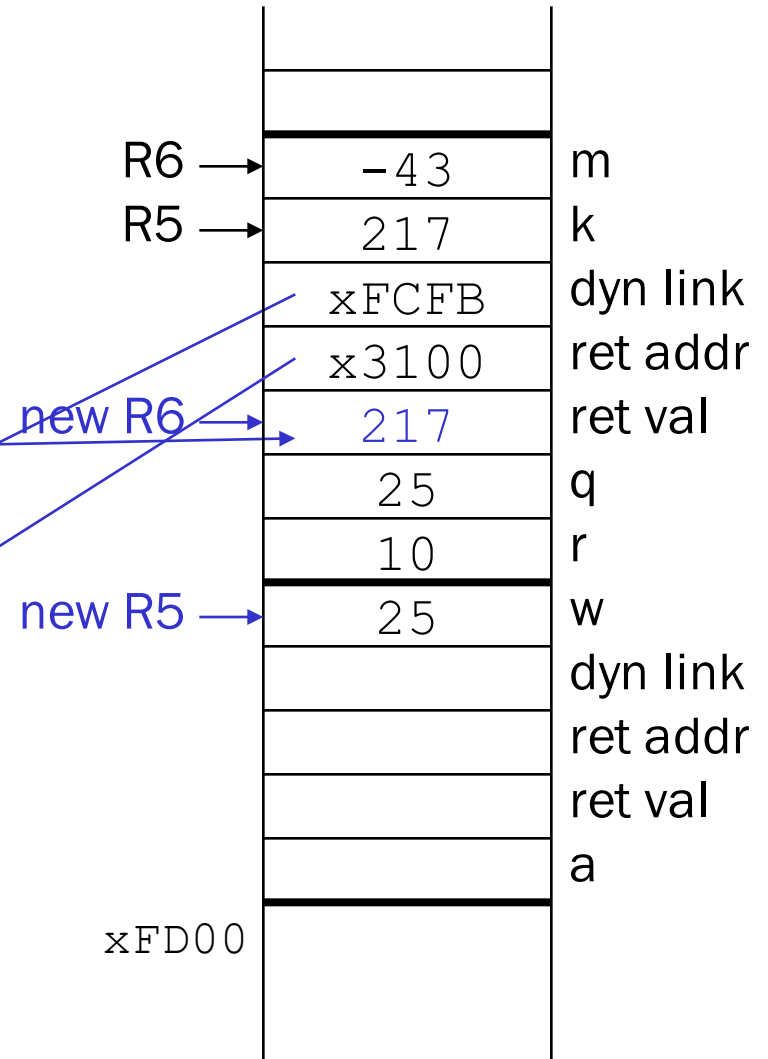
; pop return addr (into R7)

LDR R7, R6, #0

ADD R6, R6, #1

; return control to caller

RET



Resuming the Caller Function

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

JSR Volta

```
; load return value (top of stack)
```

```
LDR    R0, R6, #0
```

```
; perform assignment
```

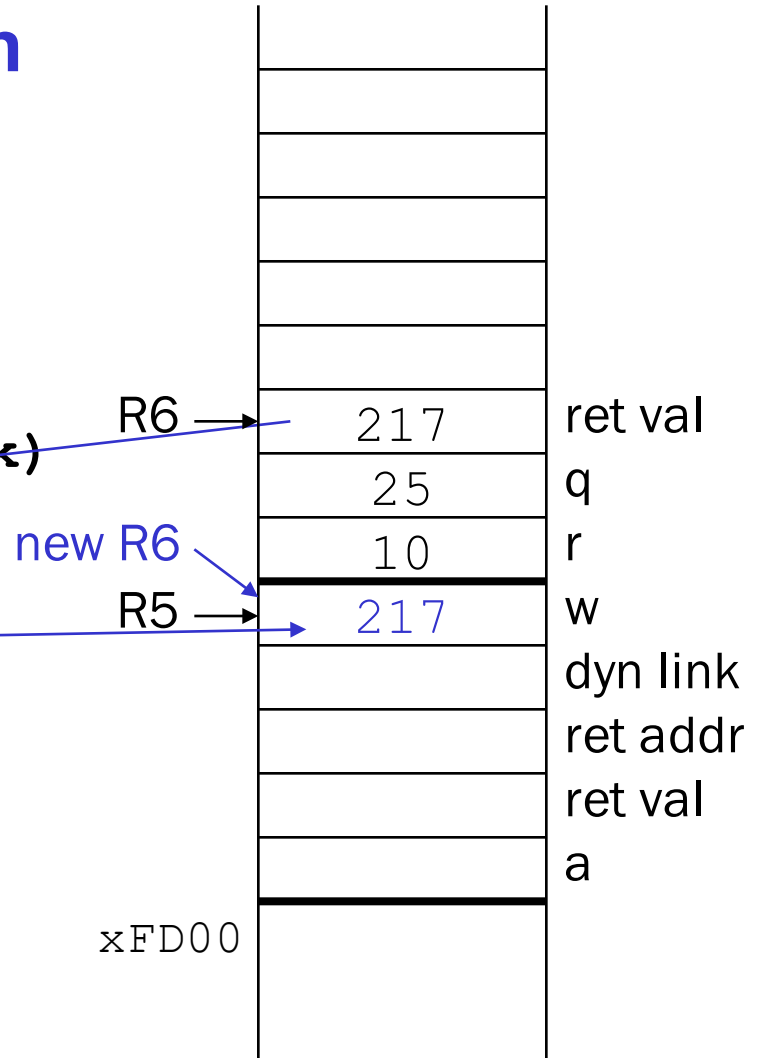
STR R0, R5, #0

```
; pop return value
```

```
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...
- 