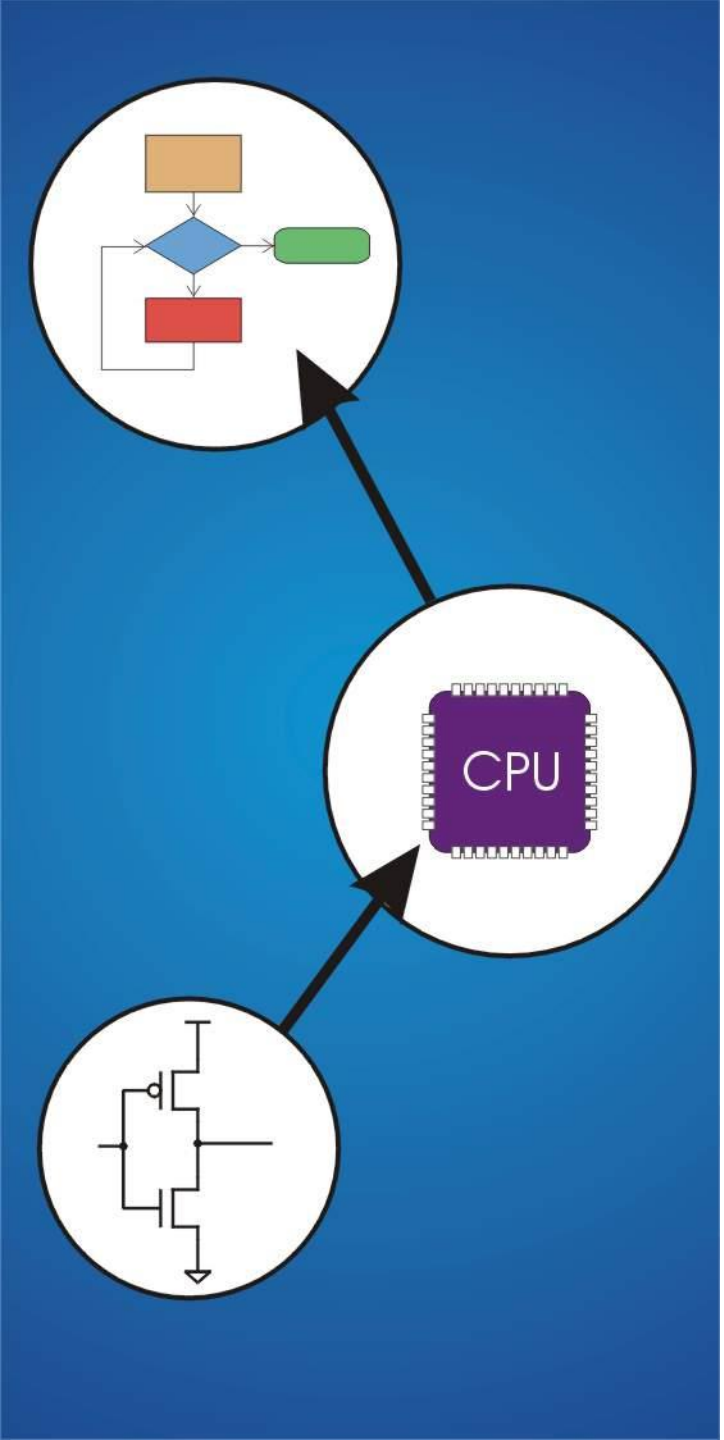


Chapter 14
Functions

Function

- **Smaller, simpler, subcomponent of program**
- **Provides abstraction**
  - **hides low-level details**
  - **gives high-level structure to program and makes it easier to understand overall program flow**
  - **enables separable, independent development**

C functions

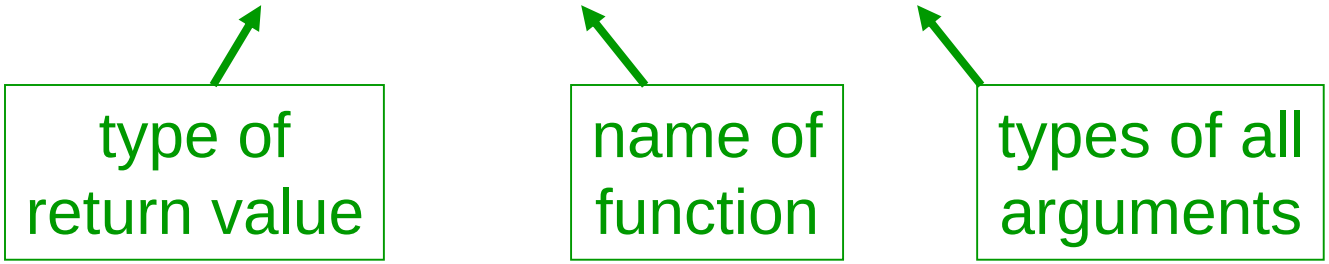
- **zero or multiple arguments passed in**
- **single result returned (optional)**
- **return value is always a particular type**

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

Functions in C

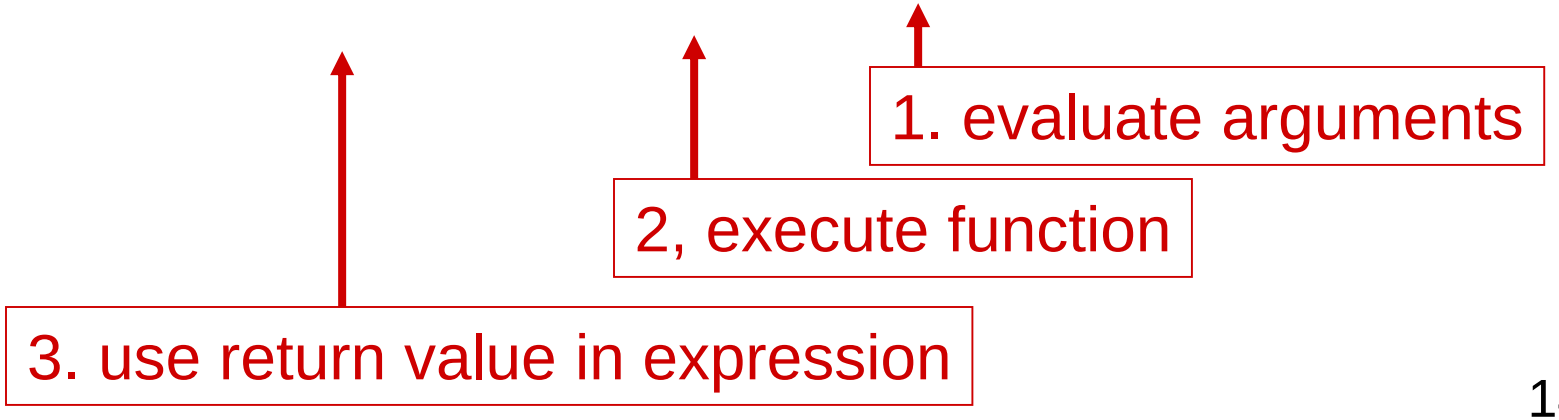
(1) Declaration (also called prototype)

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



(2) Function call -- used in expression

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



Functions in C

(3) Function Definition

State type, name, types of arguments

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

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

- **can use a function before its 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**

Another Example

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

main()
{
  ...
  dollars = ValueInDollars(francs,
                           DOLLARS_PER_FRANC);
  printf("%f francs equals %f dollars.\n",
         francs, dollars);
  ...
}

double ValueInDollars(double amount, double rate)
{
  return amount * rate;
}
```

declaration

function call (invocation)

definition

A Complete Example (1)

```
#include <stdio.h>

/* Function declarations */
double AreaOfCircle(double radius);

int main()
{
  double outer;                          /* Outer radius */
  double inner;                          /* Inner radius */
  double areaOfRing;                     /* Area of ring */

  printf("Enter outer radius: ");
  scanf("%lf", &outer);

  printf("Enter inner radius: ");
  scanf("%lf", &inner);

  areaOfRing = AreaOfCircle(outer) - AreaOfCircle(inner);
  printf("The area of the ring is %f\n", areaOfRing);
}

/* Calculate area of circle given a radius */
double AreaOfCircle(double radius)
{
  double pi = 3.14159265;

  return pi * radius * radius;
}
```

A Complete Example (2-1)

```
#include <stdio.h>

/* Function declaration */
char ToUpper(char inchar);

/* Function main:                                          */
/* Prompt for a line of text, Read one character,          */
/* convert to uppercase, print it out, then get another    */
int main()
{
  char echo = 'A';          /* Initialize input character */
  char upcase;                  /* Converted character    */

  while (echo != '\n') {
    scanf("%c", &echo);
    upcase = ToUpper(echo);
    printf("%c", upcase);
  }
}
```

A Complete Example (2-2)

```
/* Function ToUpper:                                    */
/* If the parameter is lower case return                */
/* its uppercase ASCII value                            */
char ToUpper(char inchar)
{
  char outchar;

  if ('a' <= inchar && inchar <= 'z')
    outchar = inchar - ('a' - 'A');
  else
    outchar = inchar;

  return outchar;
}
```

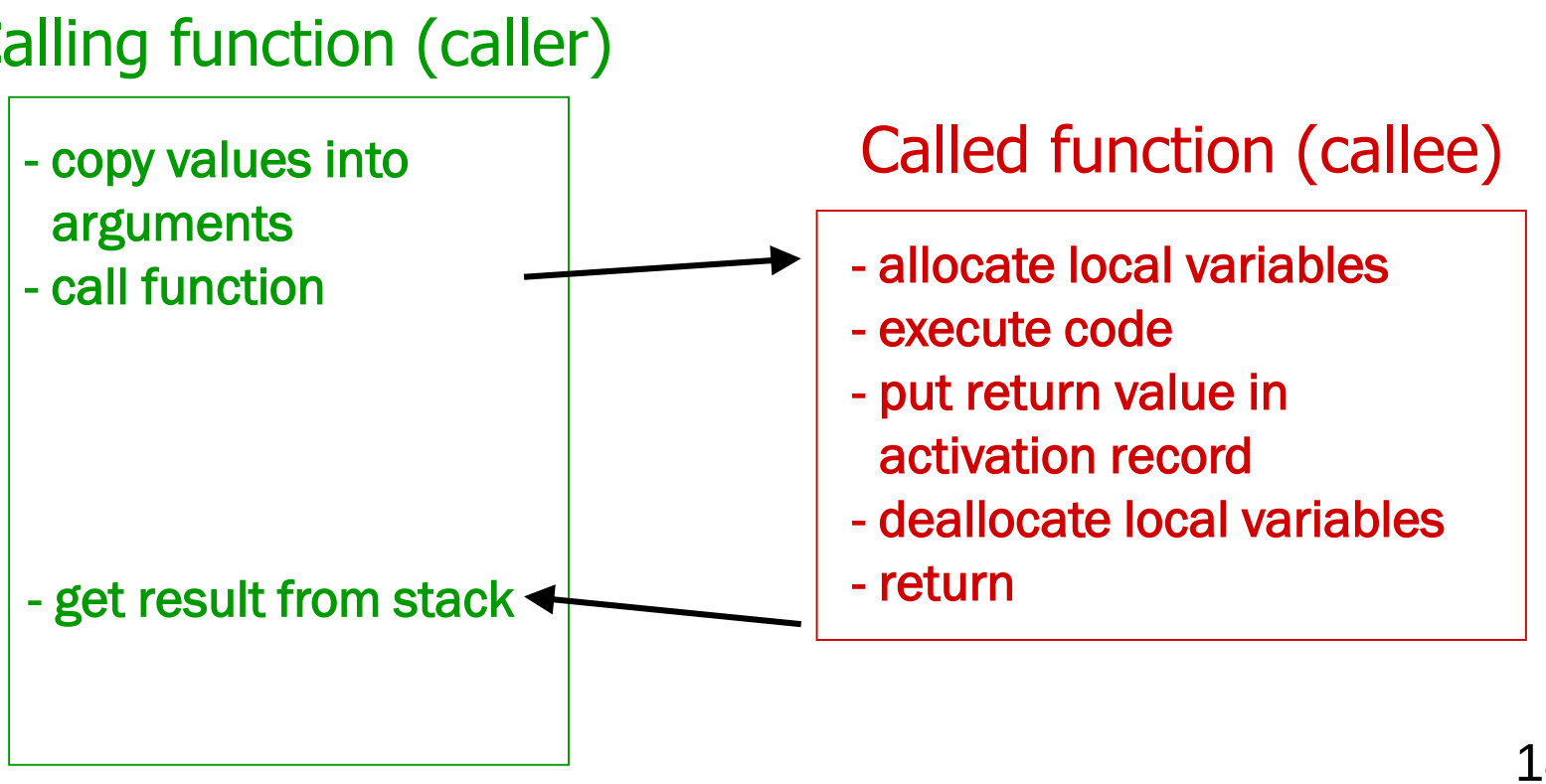
A Complete Example (3)

```
#include <stdio.h>

int Squared(int x);

int main()
{
  int sideA;
  int sideB;
  int sideC;
  int maxC;

  printf("Enter the maximum length of hypotenuse: ");
  scanf("%d", &maxC);

  for (sideC = 1; sideC <= maxC; sideC++) {
    for (sideB = 1; sideB <= maxC; sideB++) {
      for (sideA = 1; sideA <= maxC; sideA++) {
        if (Squared(sideC) == Squared(sideA) + Squared(sideB))
          printf("%d %d %d\n", sideA, sideB, sideC);
      }
    }
  }
}

/* Calculate the square of a number */
int Squared(int x)
{
  return x * x;
}
```

Implementing Functions: Overview

Activation record

- **information about each function, including arguments, return value, local variables**
- **stored on run-time stack**



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

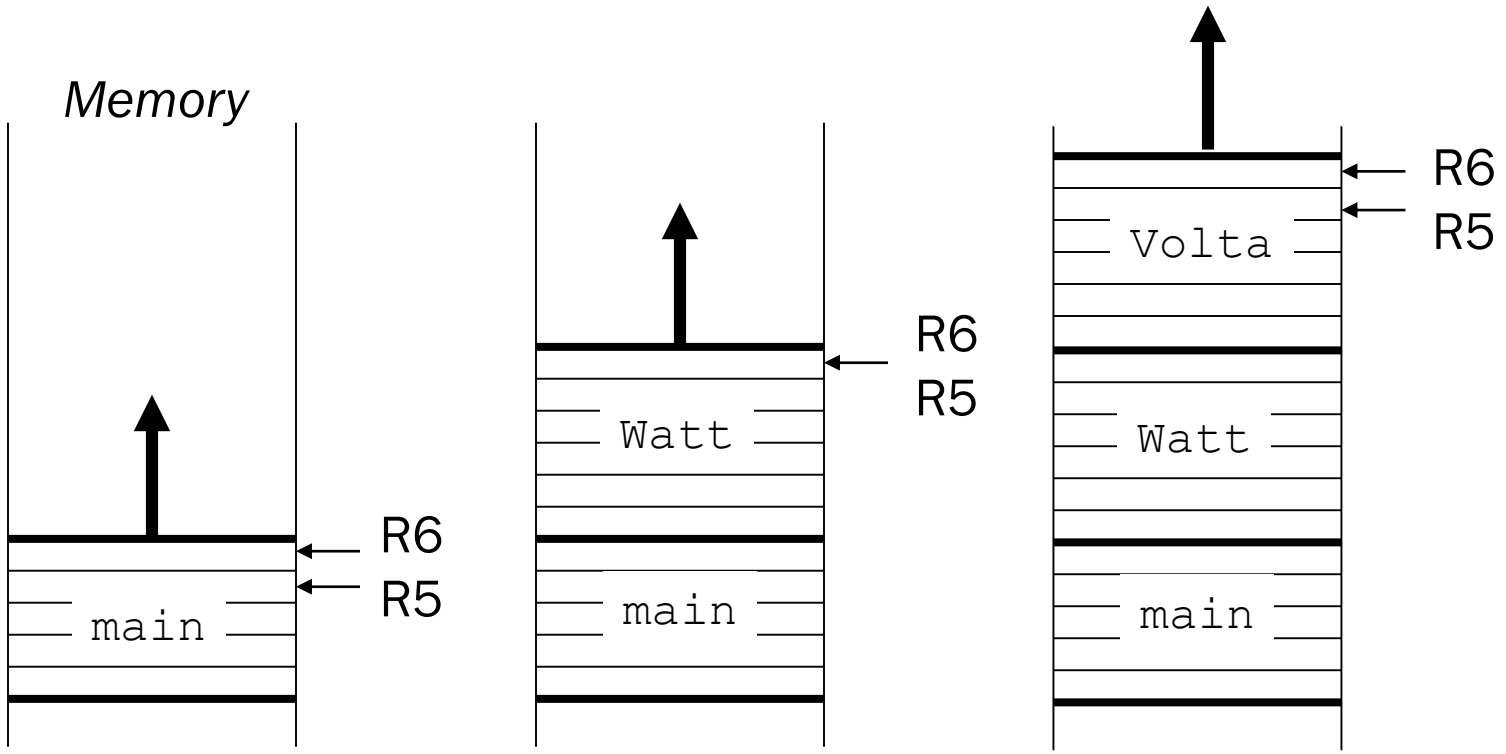
When a new function is called, its activation record is pushed on the stack;

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

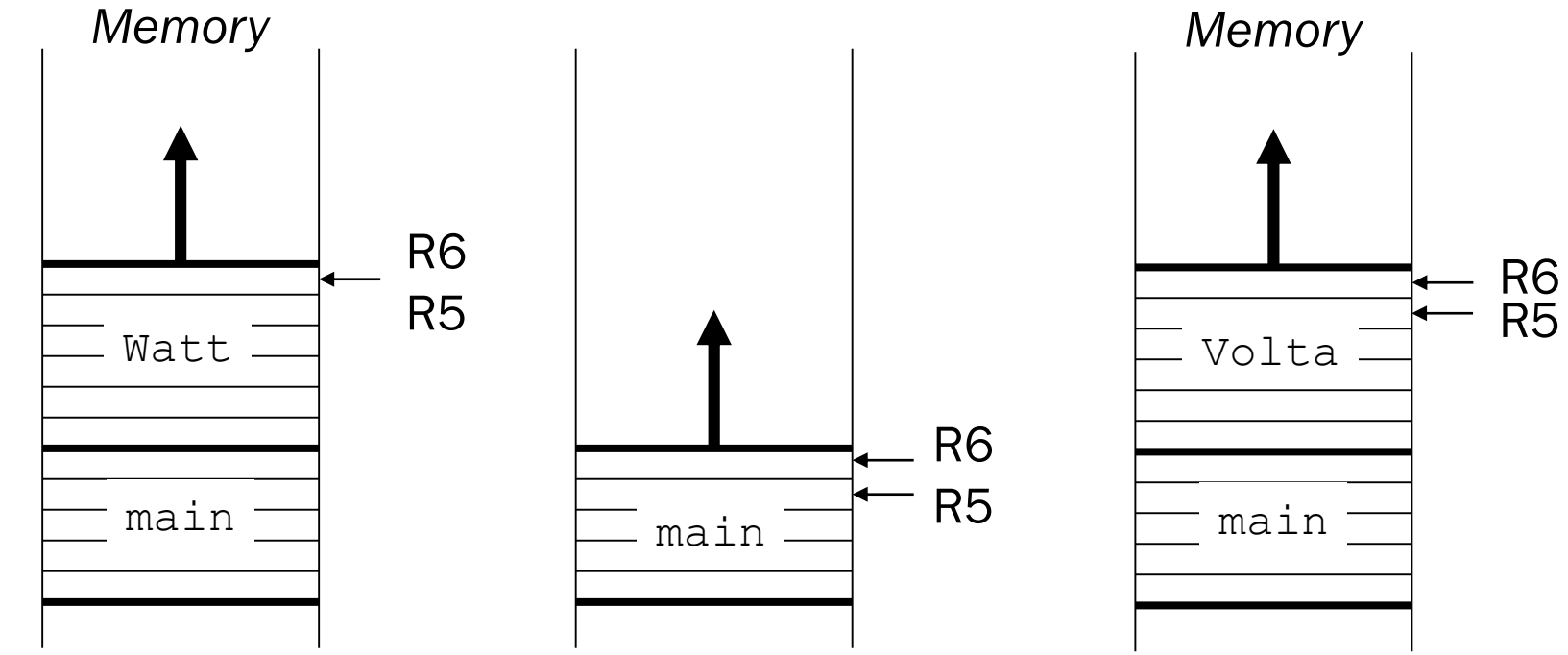
Example

```
int main()
{
  int a;
  int b;
  ...
  b = Watt(a);
  b = Volta(a, b);
}

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

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

Run-Time Stack



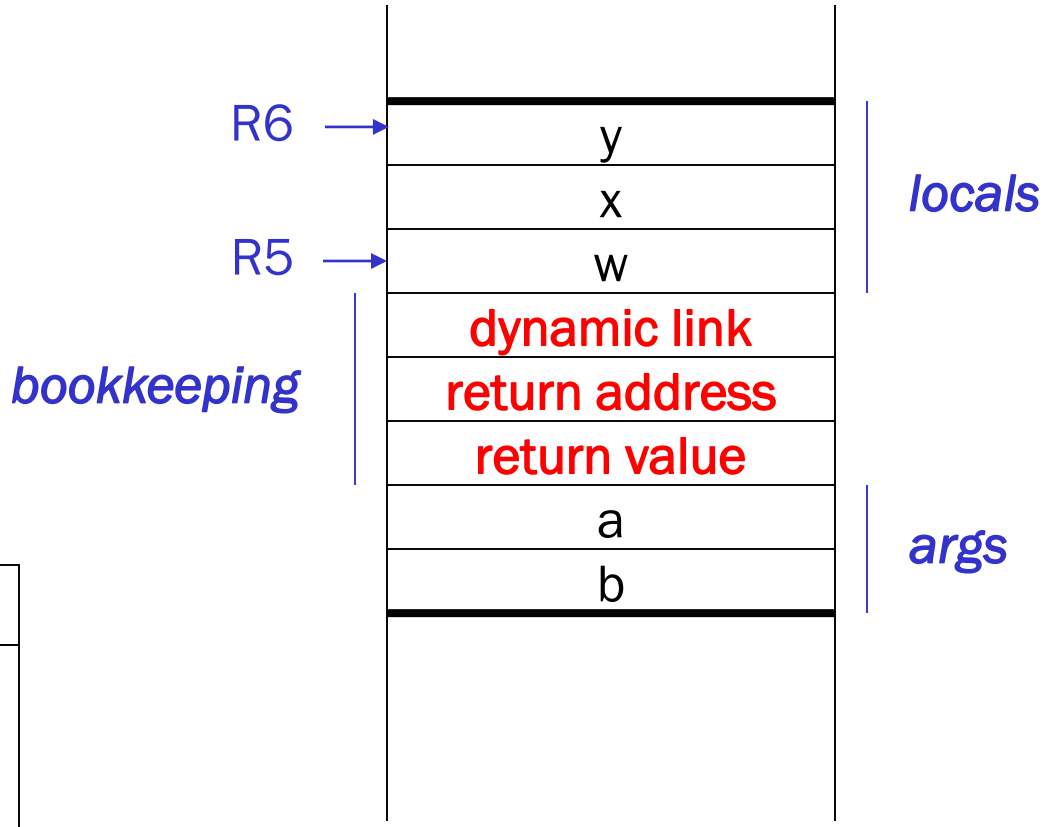
Run-Time Stack



When Volta completes *When Watt completes* *When Volta executes*

Activation Record

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



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

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

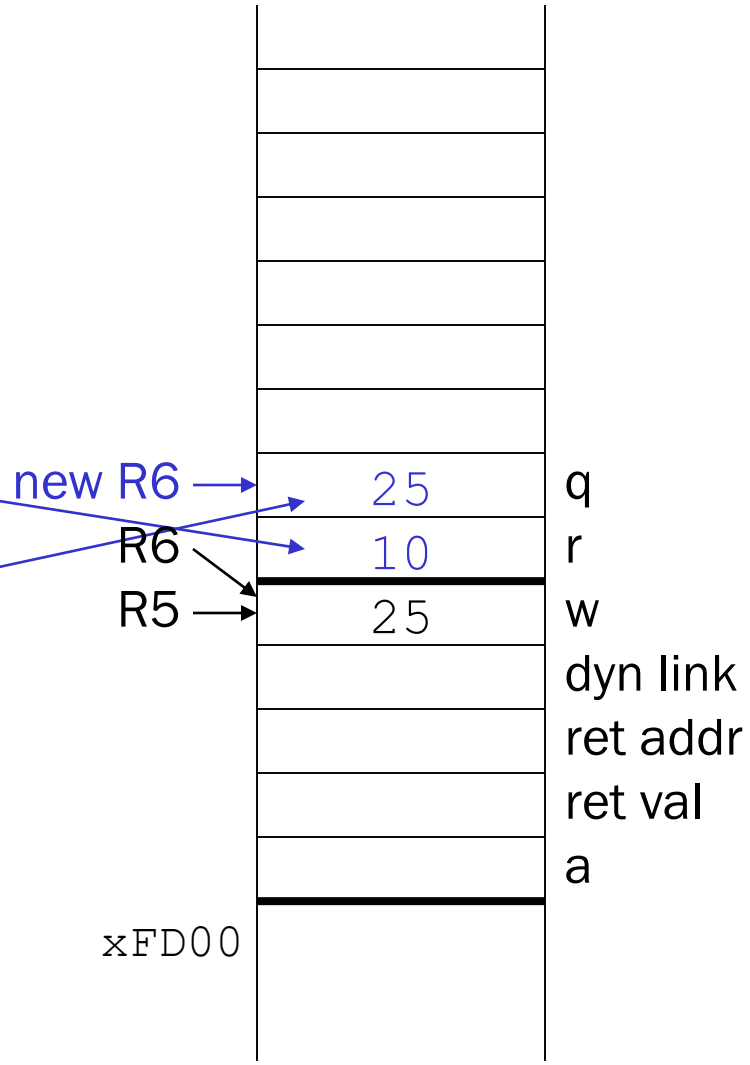
Example Function Call

```
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

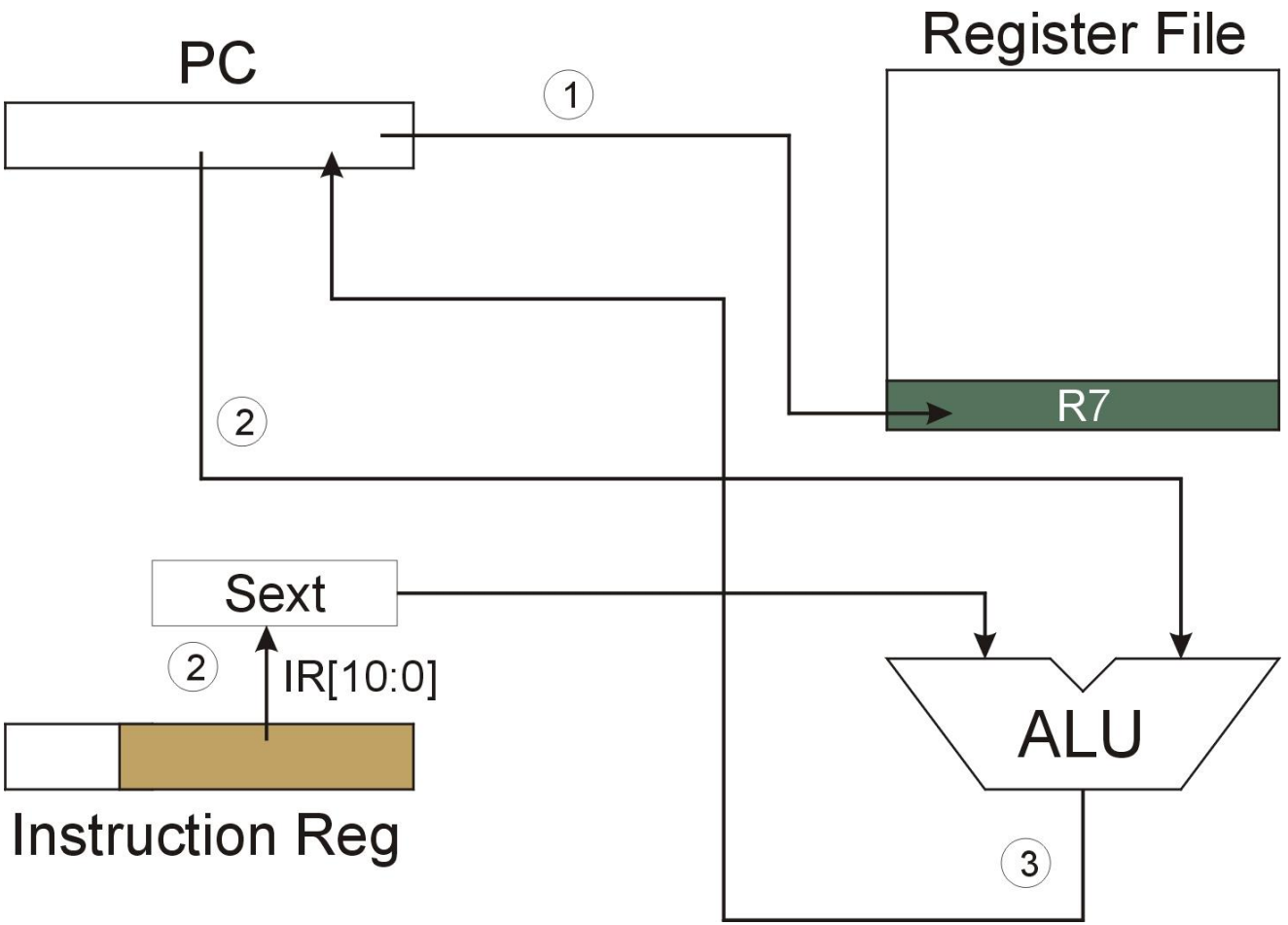
```
w = Volta(w, 10);
```

```
; 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
```



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

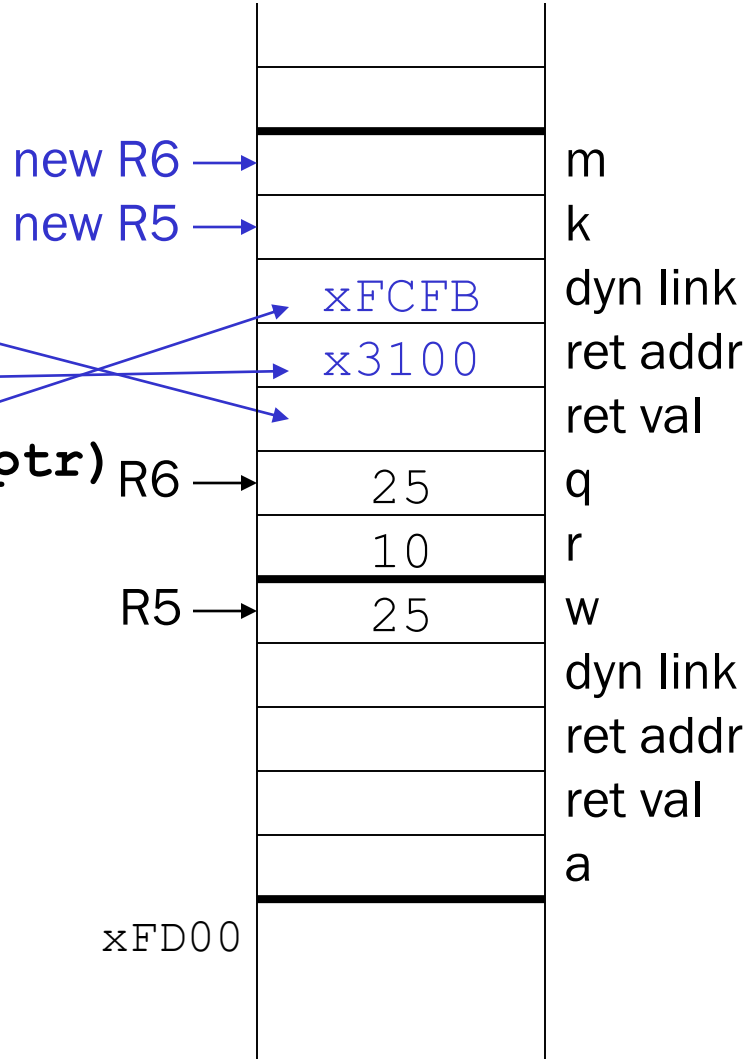
JSR



NOTE: PC has already been incremented during instruction fetch stage.

Starting the Callee Function

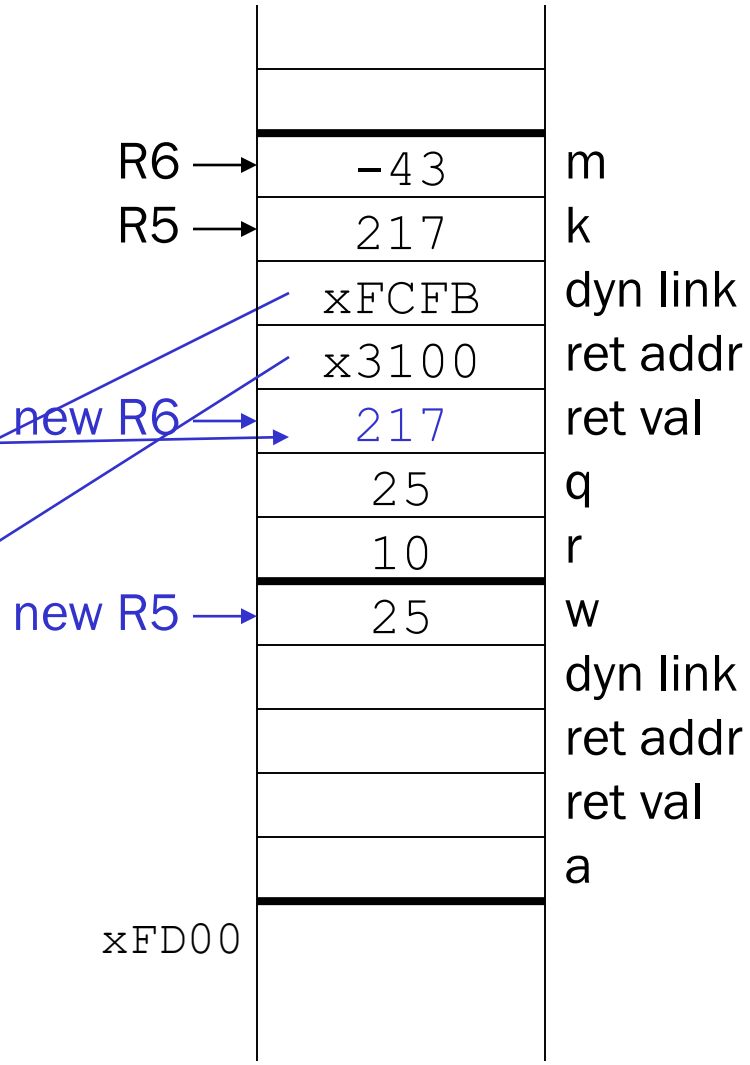
```
; leave space for return value
ADD  R6, R6, #-1
; push return address
ADD  R6, R6, #-1
STR  R7, R6, #0
; push dyn link (caller's frame ptr)
ADD  R6, R6, #-1
STR  R5, R6, #0
; set new frame pointer
ADD  R5, R6, #-1
; allocate space for locals
ADD  R6, R6, #-2
```



Ending the Callee Function

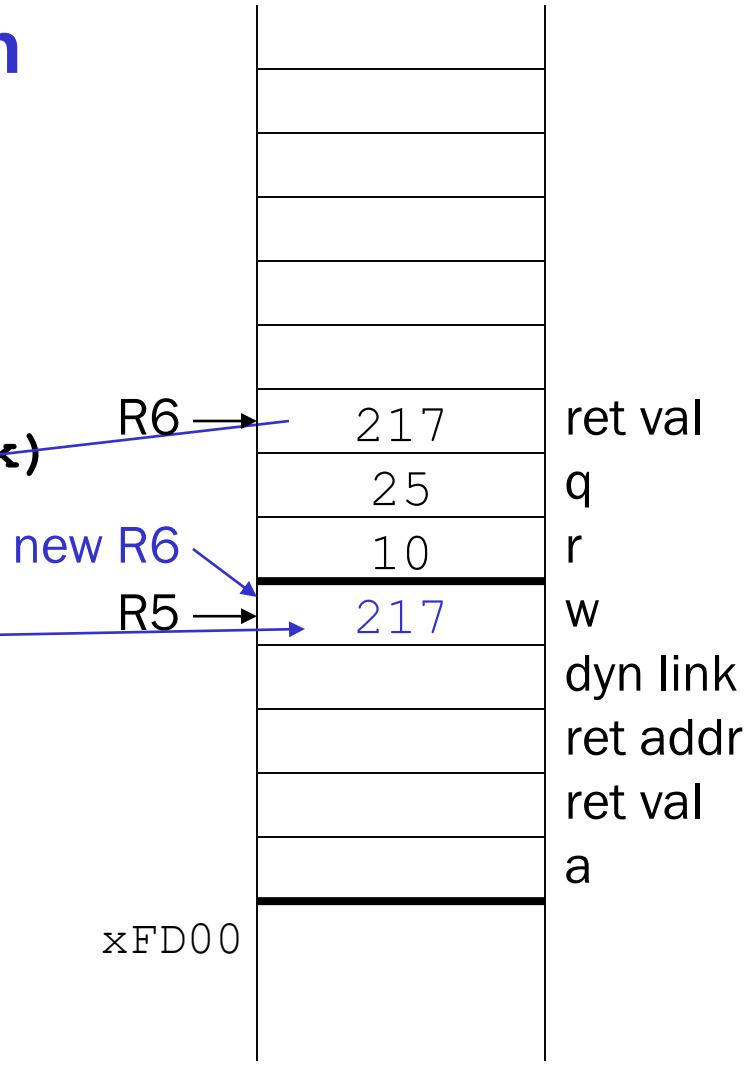
```
return k;
```

```
; 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…**

꼭 기억해야 할 것

- Function
  - Declaration
  - Invocation
  - Definition
- Activation Record on the stack contains
  - Argument values (allocated & set up & deallocated by caller)
  - return value (allocated & set up by callee and used & deallocated by caller)
  - return address (allocated & set up & deallocated by callee)
  - dynamic link – caller's R5 (allocated & set up & deallocated by callee)
  - local variables - more precisely automatic variables (allocated & set up and deallocated by callee)