ECE 220: Computer Systems & Programming

Lecture 8: Run-time Stack

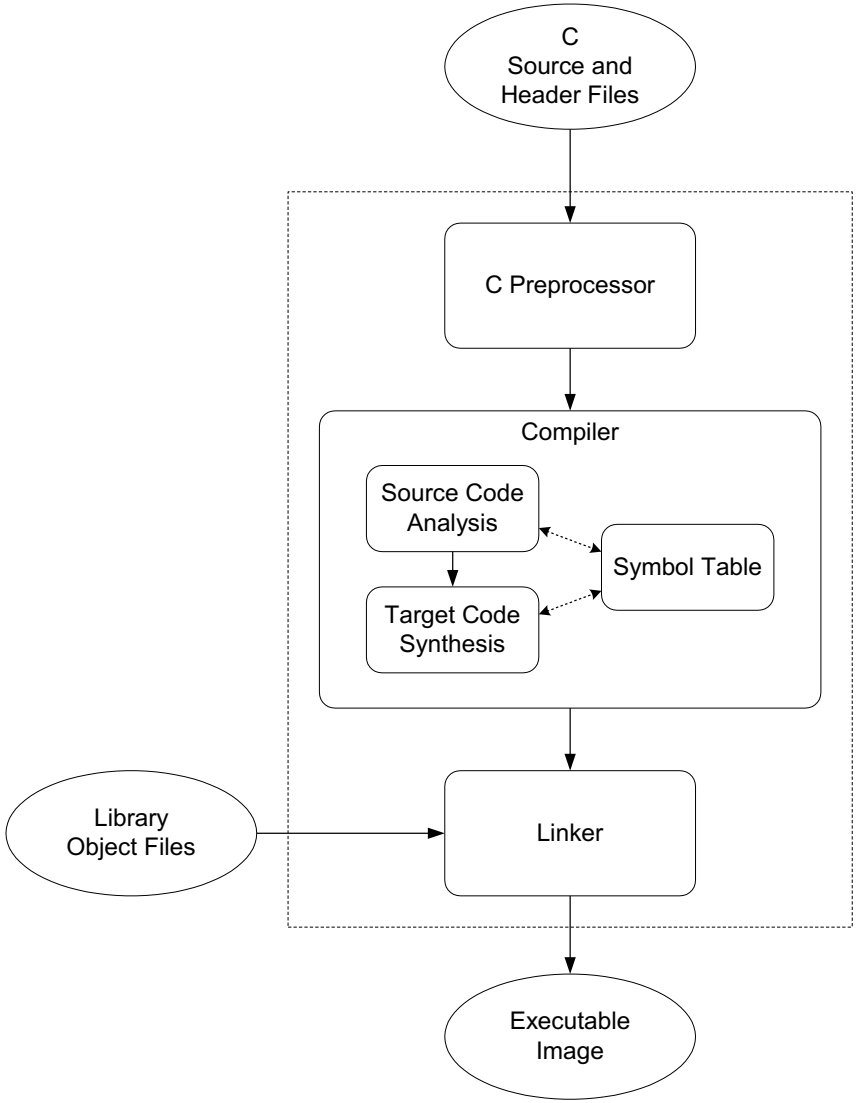
Thomas Moon

February 8, 2024

Memory Allocation for Variables

- When C-compiler compiles a program, it keeps track of variables in a program using a symbol table.

- Symbol table contains
  - name
  - type
  - location (as an offset)
  - scope



Symbol Table

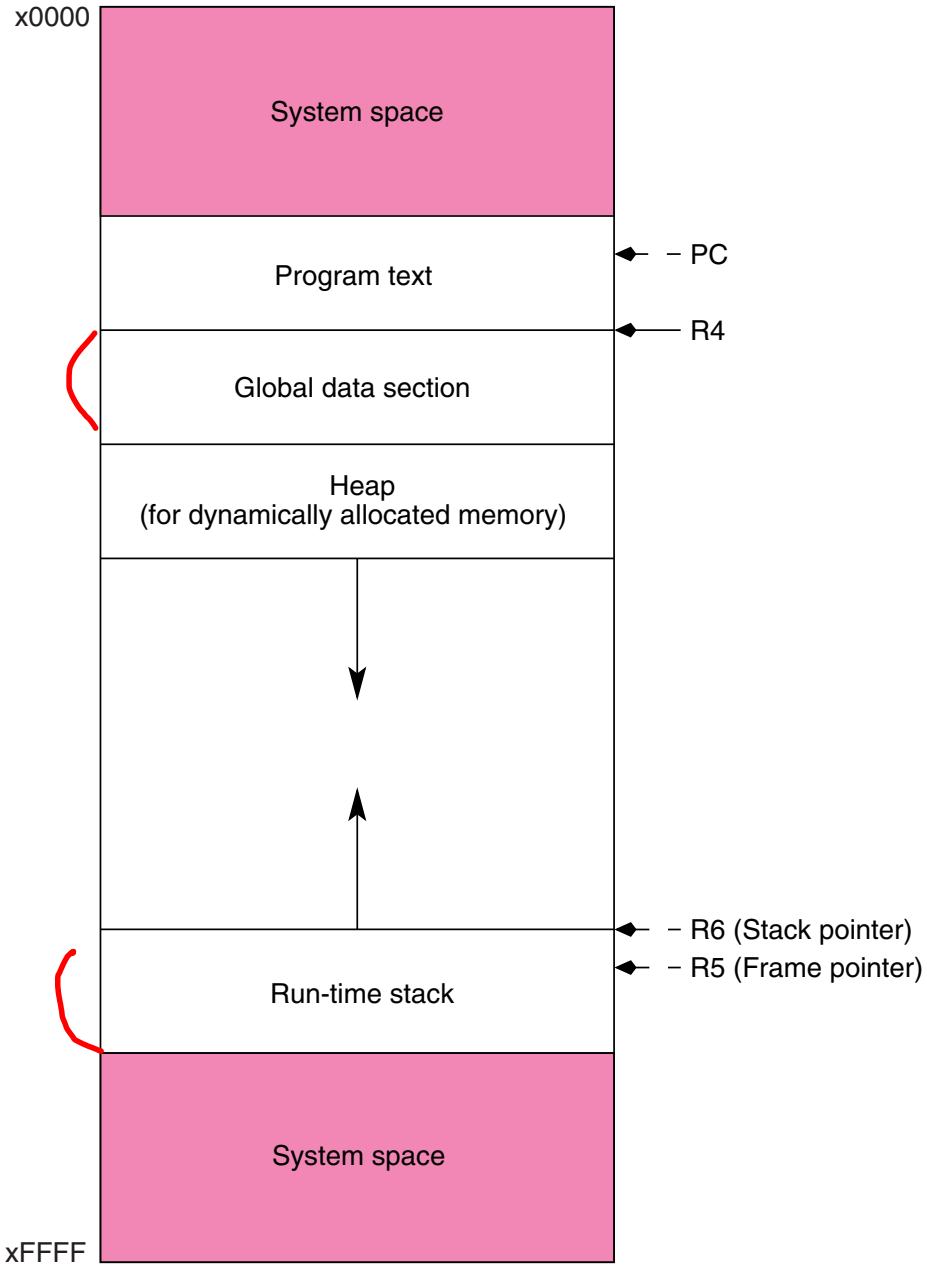
```
int inGlobal;
int outGlobal;

int dummy(int in1, int in2);

int main()
{
    int x,y,z;
    …

}
int dummy(int in1, int in2)
{
    int a,b,c;
    …
}
```

Name	Type	Location (as an offset)	Scope
inGlobal	int	0	global
outGlobal	int	1	global
x	int	0	main
y	int	-1	main
z	int	-2	main
a	int	0	dummy
b	int	-1	dummy
c	int	-2	dummy

Space for Variables

1. Global data section (global variables)
2. Run-time stack (local variables)

- R4 (global pointer) points the first global variable
- R5 (frame pointer) points the first local variable
- R6 (stack pointer) points the top of run-time stack



Global Variables – R4 (Global Pointer)

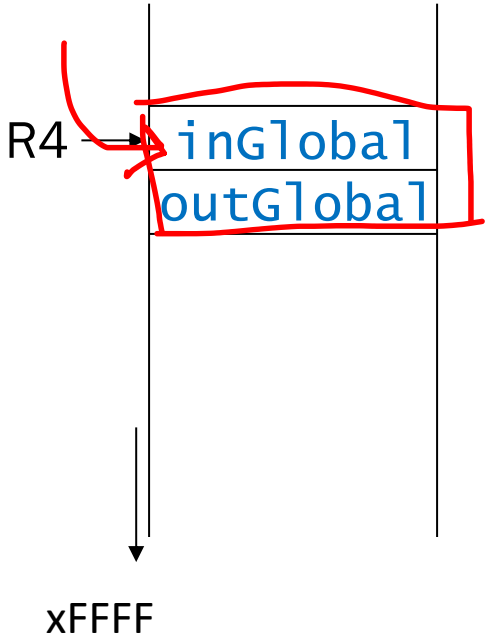
C Code

```
int inGlobal;
int outGlobal;

int main()
{
    inGlobal = 3;
    outGlobal = 0;

}
```

Symbol Table

Name	Type	Location (as an offset)	Scope
inGlobal	int	0	global
outGlobal	int	1	global

Global data sec



R4

LC-3 Code

```
AND     R0, R0, #0
ADD     R0, R0, #3
STR     R0, R4, #0 ; inGlobal = 3

AND     R0, R0, #0
STR     R0, R4, #1 ; outGlobal = 0
```

R4 points the first global variable

Local Variables

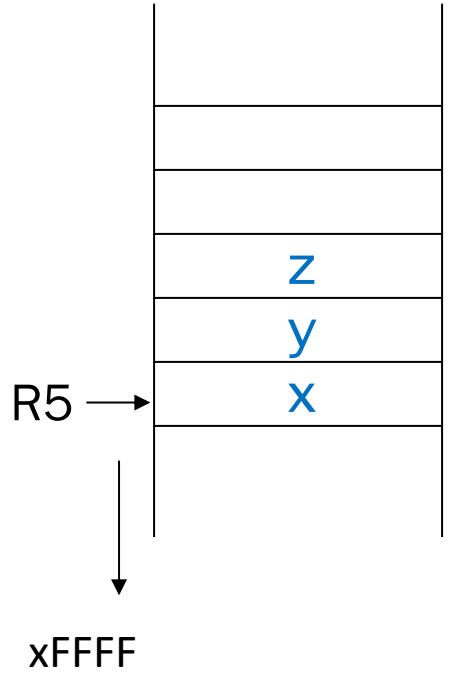
C Code

```
int main()
{
    int x,y,z;

    x = 3;
    y = 0;

}
```

Symbol Table

Name	Type	Location (as an offset)	Scope
x	int	0	main
y	int	-1	main
z	int	-2	main

Run-time stack (partial)

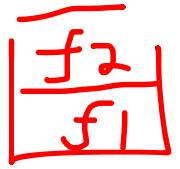


LC-3 Code

```
AND     R0, R0, #0
ADD     R0, R0, #3
STR     R0, R5, #0 ; x = 3

AND     R0, R0, #0
STR     R0, R5, #-1 ; y = 0
```

R5 points the first local variable

Local Variables in Activation Record



- Every *function call* creates an *activation record (or stack frame)* and *pushes* it onto the run-time stack.
- Local variables are one part of the activation record.
- Whenever a function *completes (return)*, the activation record is *popped* off the run-time stack.
- Whenever a function calls another one (*nested*), the run time stack *grows* (push another activation record onto the run-time stack).

one function call = one activation record

one function ≠ one activation record

Life of functions

```
int dummy(int in1, int in2);

int main()
{
    int x,y,z;
    x = dummy(10, 20);
    …

    return 0;
}
int dummy(int in1, int in2)
{
    int a,b,c;
    …

    return a;
}
```

main in dummy in dummy out main out

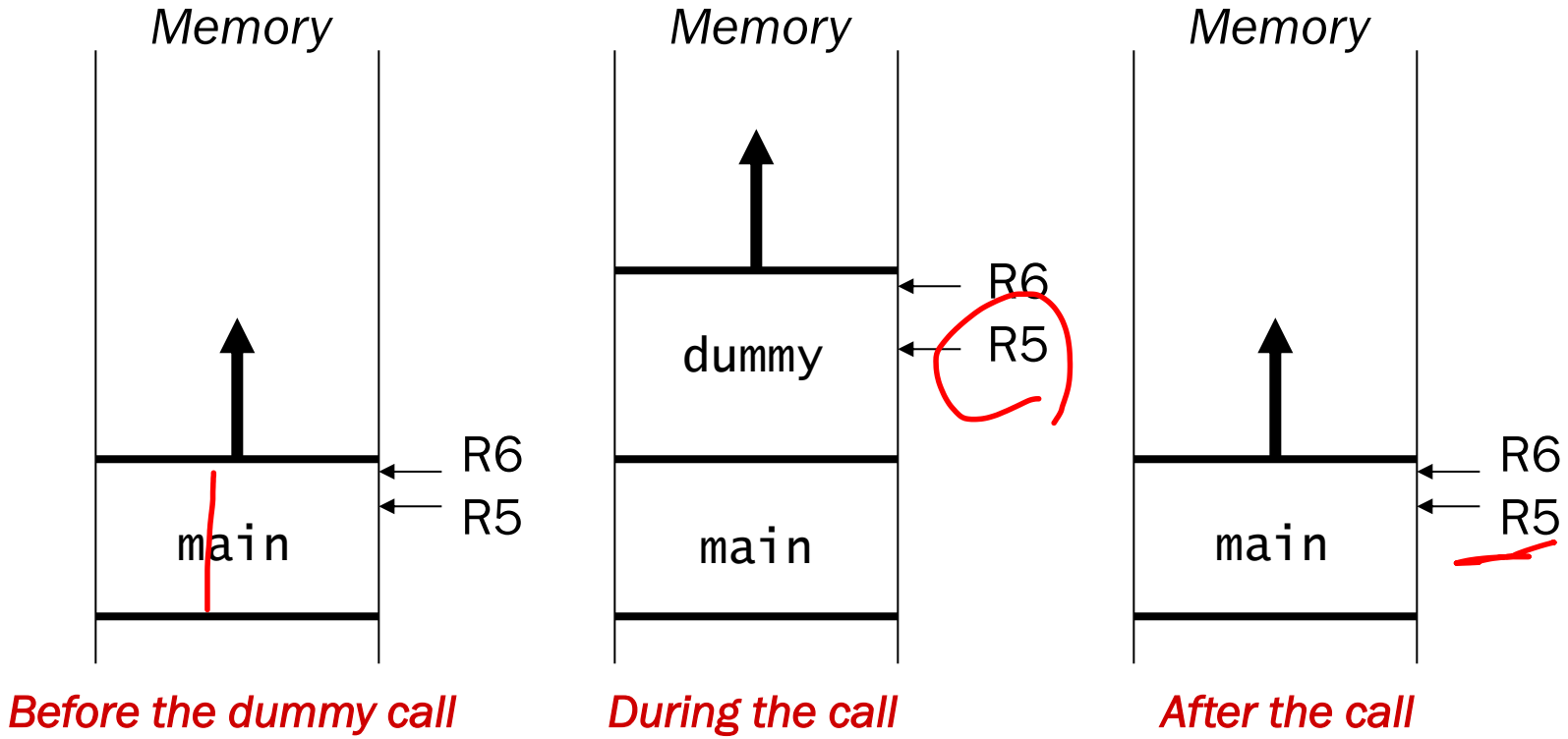
For saving “Activation Record” in the memory, which data structure will be the best?

1. First-In First-Out (FIFO)
2. Last-In First-Out (LIFO)

Activation Record

- stored in run-time stack
- *function call = push activation record*
- *function return = pop activation record*

```
int dummy(int in1, int in2);

int main()
{
    int x,y,z;
    x = dummy(10, 20);

}
```

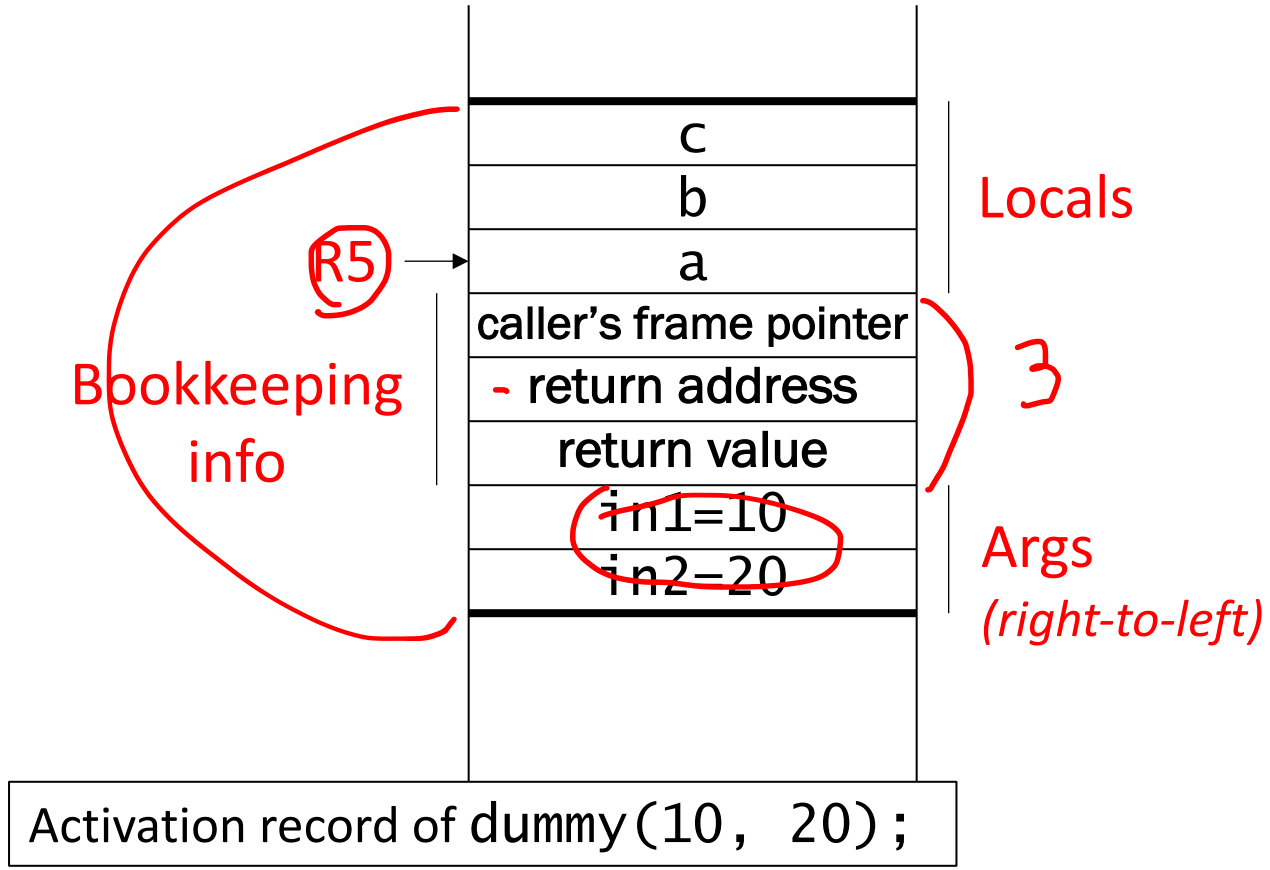


Building Activation Record

- Information about each function call, including

1. Arguments *2. Bookkeeping info* *3. Local variables*

```
int main()
{
    int x,y,z;
    x = dummy(10, 20);

}
int dummy(int in1, int in2)
{
    int a,b,c;
    .
    .
    .
    return a;
}
```



Bookkeeping (comsumes 3 memory spaces)

- Return value
  - space for value returned by function
  - memory allocated even if function does not return a value (void function)

- Return address
  - save pointer to next instruction in calling function
  - store R7 here

- Caller's frame pointer (Dynamic link)
  - previous R5

Stack Build-up and Tear-down

Caller function	1. Caller setup (push callee's arguments onto stack) 2. Pass control to callee (JSR/JSRR)
Callee function	3. Callee setup (push bookkeeping info and local variables onto stack) 4. Execute function ← 5. Callee tear-down (update return value, pop local variables, caller's frame pointer and return address from stack) 6. Return to caller (RET)
Caller function	7. Caller tear-down (pop callee's return value and arguments from stack)

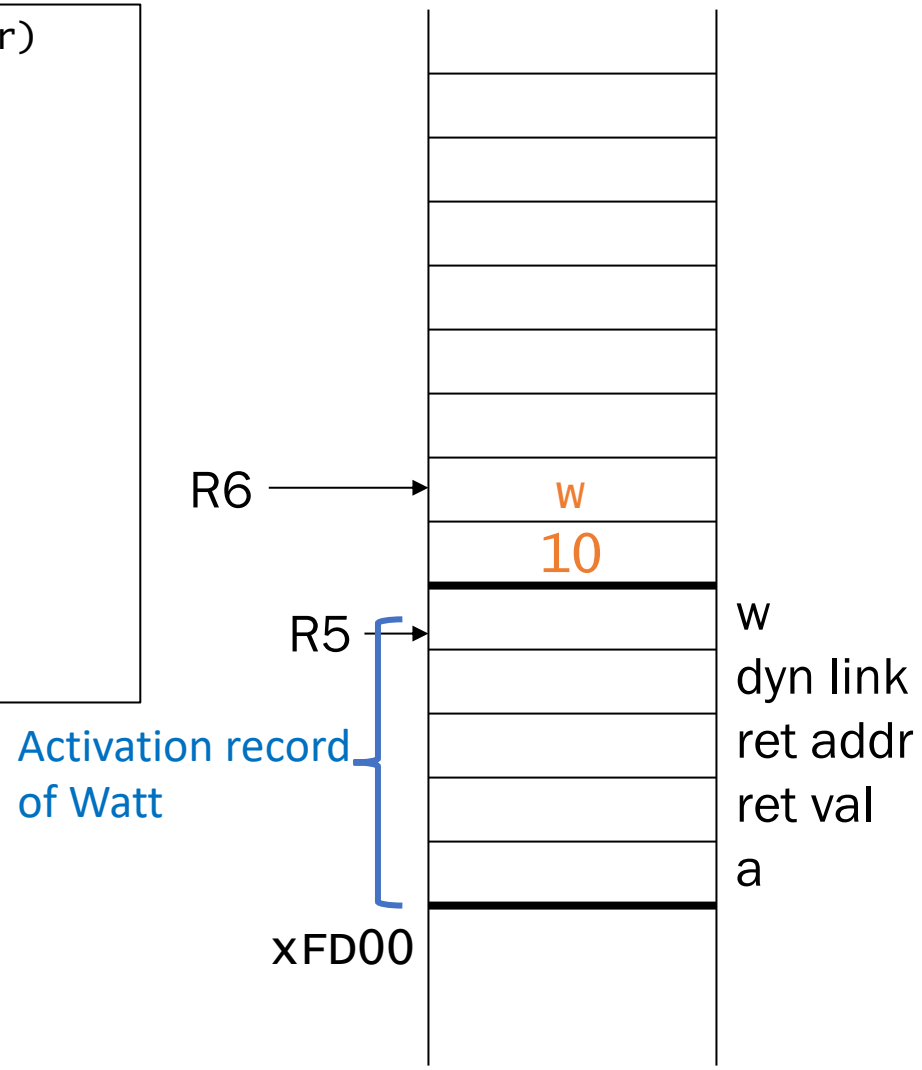
Example Function Call

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


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

1. Caller setup (push callee's arguments onto stack)
2. Pass control to callee

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

; call subroutine
JSR  Volta
```

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

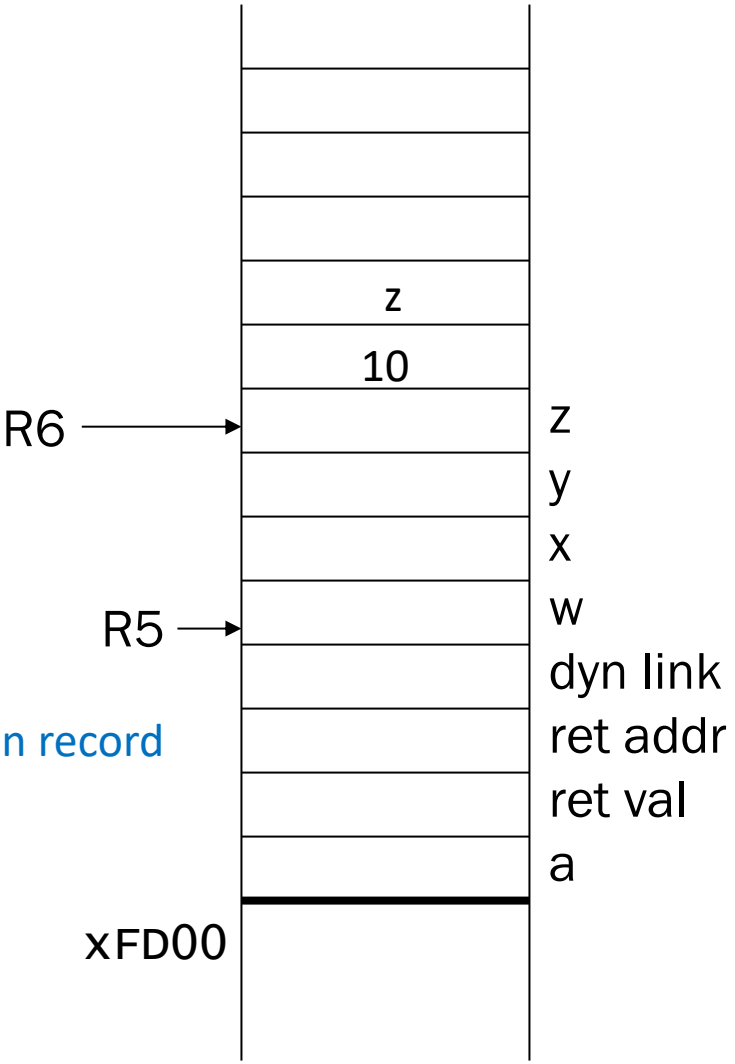


Q1. Draw the activation record of Watt.

Q2. Find the offset value to push “z”.

```
Watt
...
; push second arg
AND  R0, R0, #0
ADD  R0, R0, #10
ADD  R6, R6, #-1
STR  R0, R6, #0
; push z
LDR  R0, R5, #-3
ADD  R6, R6, #-1
STR  R0, R6, #0

; call subroutine
JSR  Volta
```

```
int Volta(int q, int r)
{
  int k;
  int m;
  ...
  return k;
}
int Watt(int a)
{
  int w, x, y, z;
  ...
  w = Volta(z,10);
  ...
  return w;
}
```

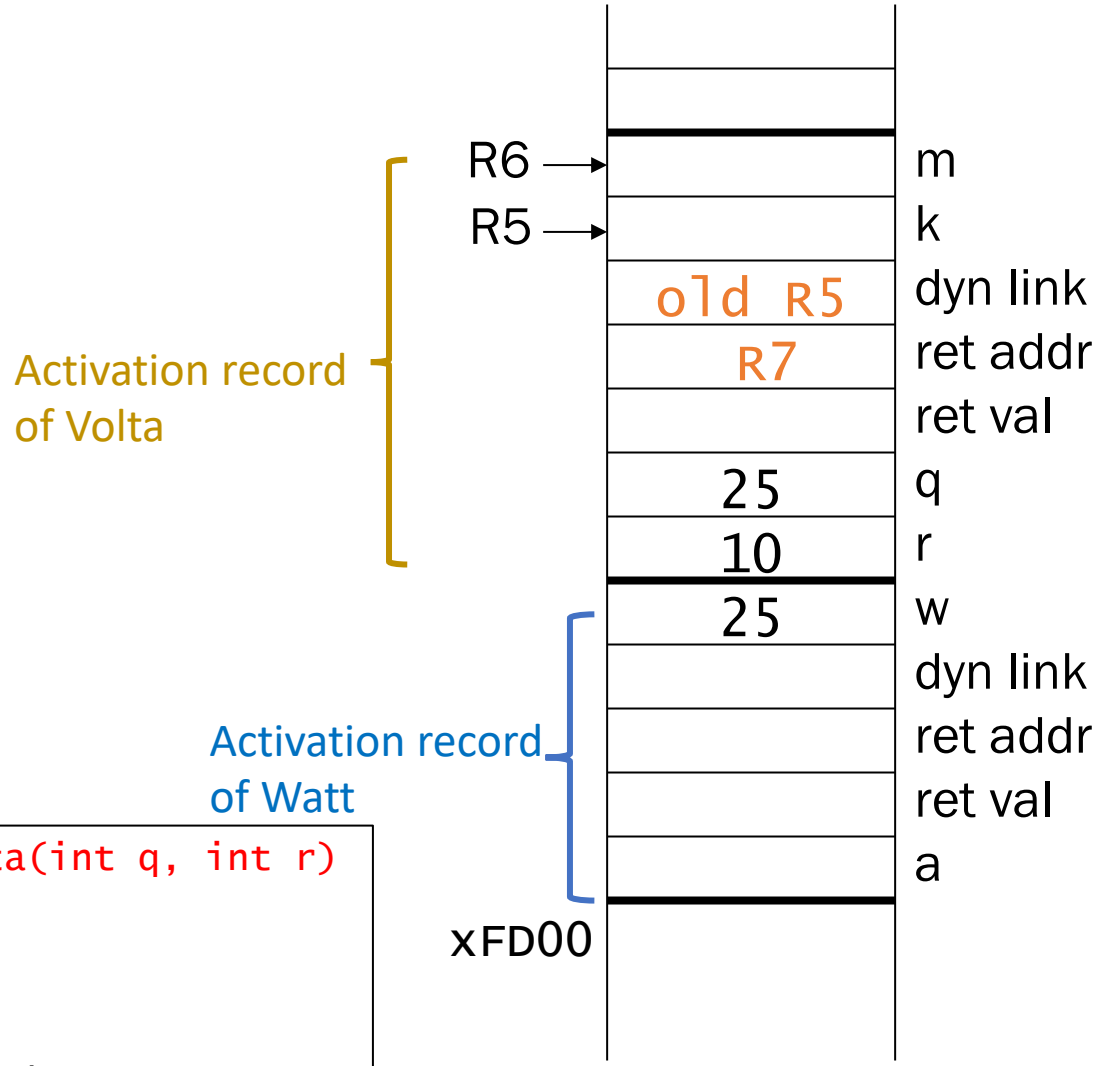


3. Callee setup (push bookkeeping info and local variables onto stack)
4. Execute function

```
Volta
; 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
```

depends on # local var

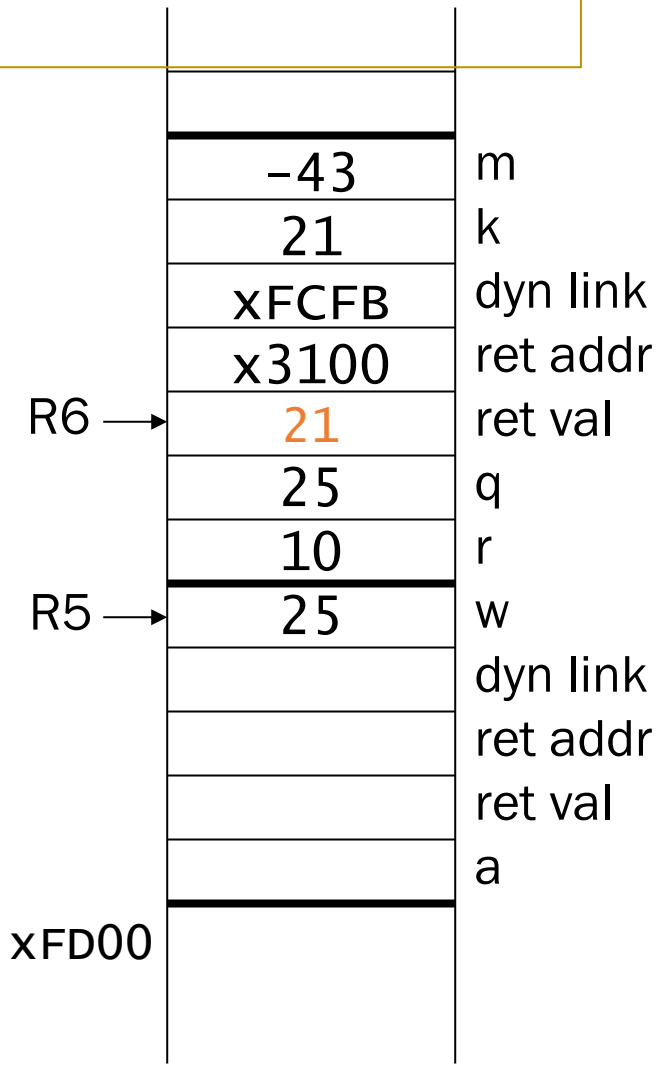
```
int Volta(int q, int r)
{
  int k;
  int m;
  ...
  return k;
}
```



5. Callee tear-down (update return value, pop local variables, caller's frame pointer and return address from stack)
6. Return to caller

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

depends on which variable




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

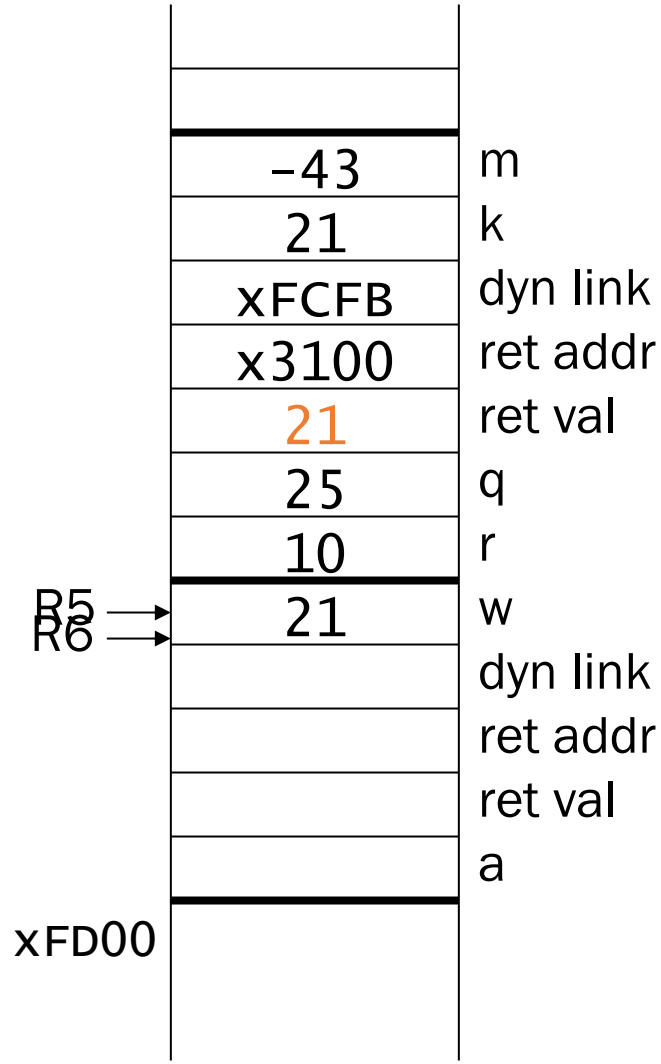
7. Caller tear-down (pop callee's return value and arguments from stack)

```
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
```

depends on which variable

depends on # arguments

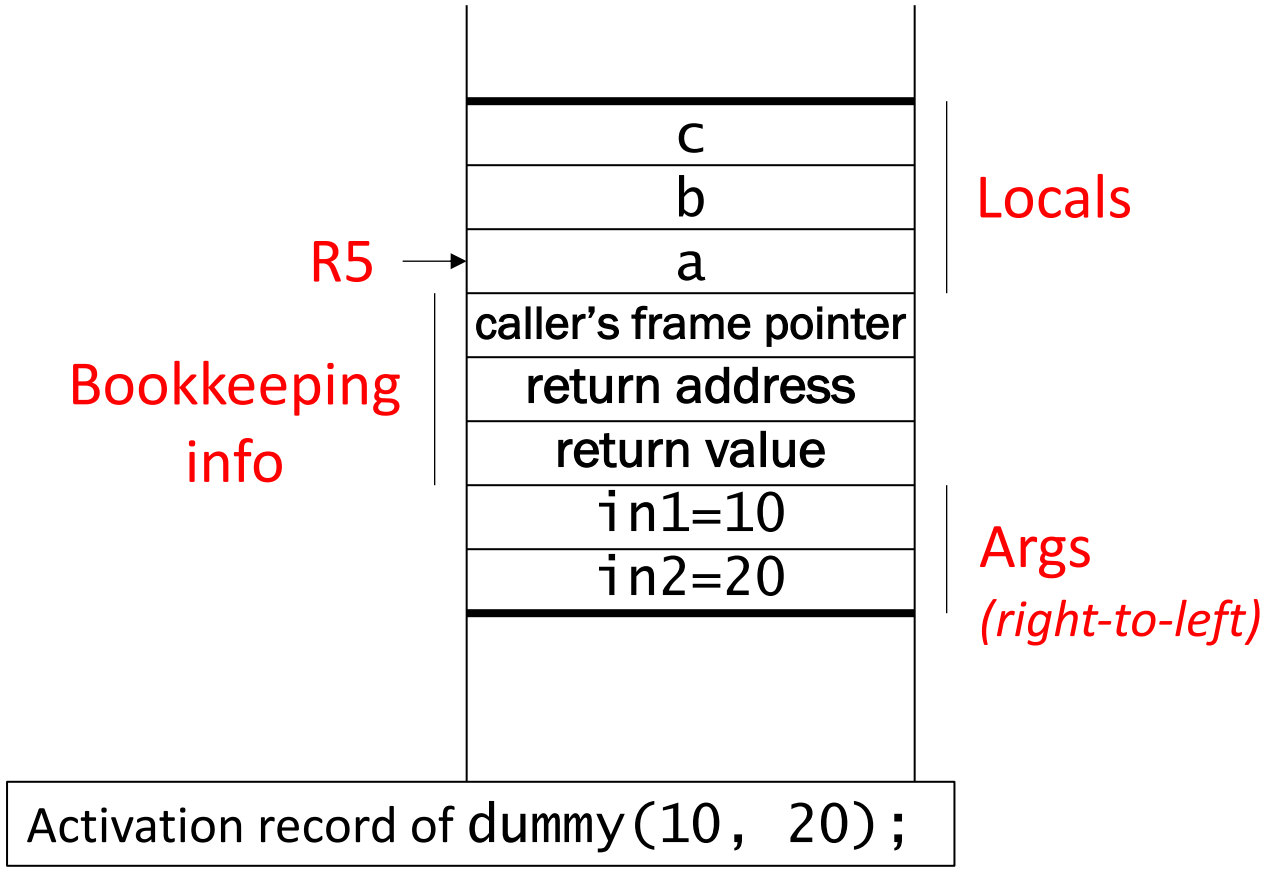
```
int Watt(int a)
{
  int w;
  ...
  w = Volta(w,10);
  ...
  return w;
}
```



Q. Who push them, Caller or Callee?

1. Arguments *2. Bookkeeping info* *3. Local variables*

```
int main()
{
    int x,y,z;
    x = dummy(10, 20);

}
int dummy(int in1, int in2)
{
    int a,b,c;
    .
    .
    .
    return a;
}
```



Why Run-time Stack?

- Option1: Assign each activation record at fixed memory location

  Problem: What happen function A calls itself?

- Option2: Use Run-time Stack

  Each invocation of a function gets its own space in memory
  -> Permits functions to be ***recursive***!