Implementing Functions in C

Lecture Topics

- Memory allocation for variables
- Run-time stack
- Activation records

Lecture materials

- Textbook § 12.5, 14.3

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:

```
int main()
{
      double x = 0.0;
      double sin_x = 0.0;
      double r = 0.0;

…

double mysin(double x, double r)  /* function implementation */
{
      double sin_x_num;
      double sin_x_den;
      double error;
      int n;
      …
```

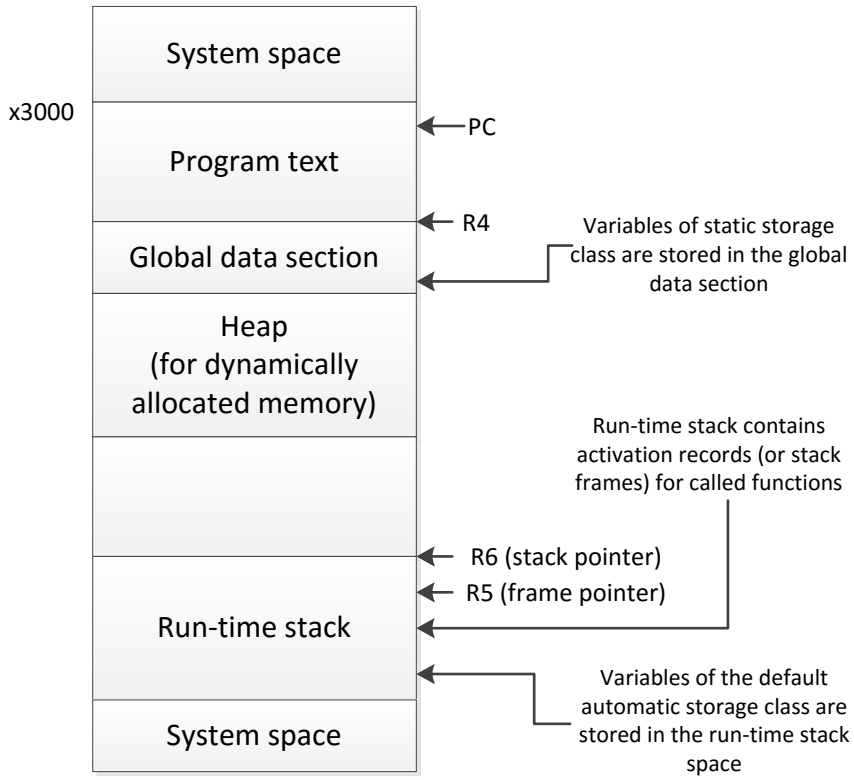
  - Corresponding symbol table:

Variable identifier (name)	type	Location (as an offset)	Scope
x	double	0	main
sin_x	double	-1	main
r	double	-2	main
sin_x_num	double	0	mysin
sin_x_den	double	-1	mysin
error	double	-2	mysin
n	int	-3	mysin

    - 7 variables total
    - Variable's location in memory is recorded as an offset which indicates relative position of the variable within the region of memory it is allocated.
      - This indicates how many locations from the base of the section a variable is allocated storage.
- 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

- Memory map for a C program



- Note 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
- 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
  - Its exact structure depends on the compiler implementation; we will study just one particular example implementation

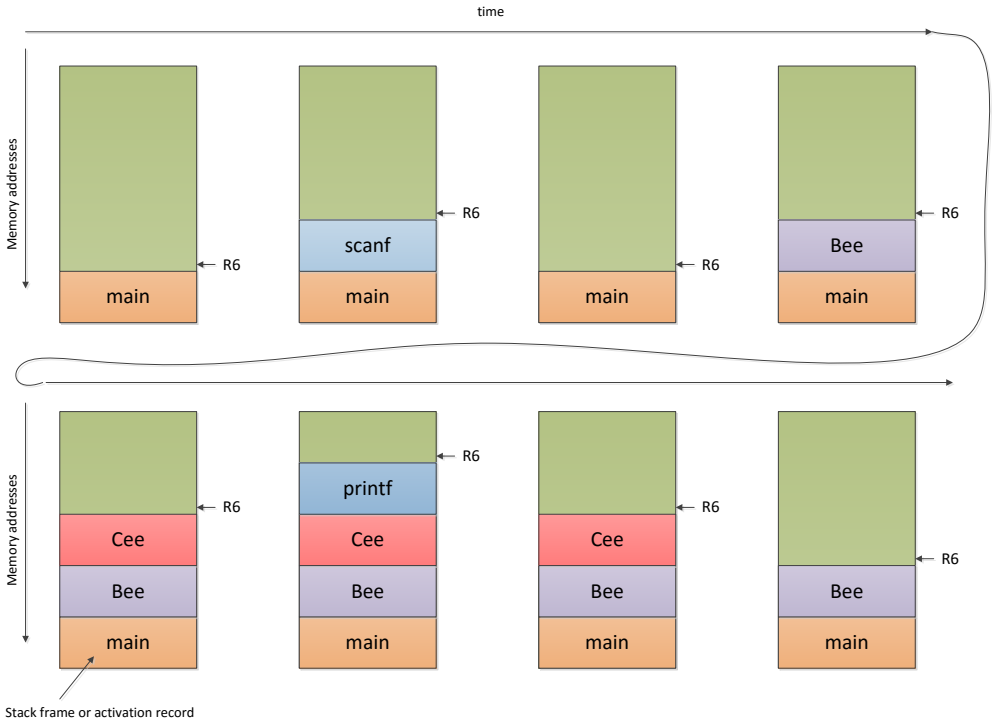
Run-time stack

- Consider the following code sample

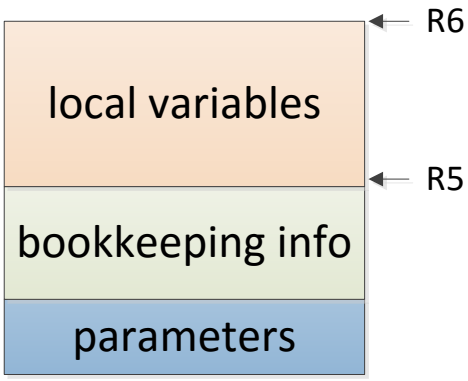
		tracing the execution of the program
	`#include <stdio.h>`	<function name>.<instruction number>
	`int Bee(int a, int b);`	
	`int Cee(int q, char K);`	
	`int Dee(int x);`	
	`int main()`	
	`{`	
	`int beeVal = 0;`	
#1	`scanf("%d", &beeVal);`	
#2	`beeVal = Bee(10, beeVal);`	
#3	`printf("%d\n", beeVal);`	
#4	`return 0;`	
	`}`	
	`int Bee(int a, int b)`	
	`{`	
	`int tempCee = 0;`	
	`int tempDee = 0;`	
#1	`tempCee = a * Cee(b, 'k');`	
#2	`tempDee = tempCee + Dee(b);`	
#3	`return tempDee;`	
	`}`	
	`int Cee(int q, char k)`	
	`{`	
#1	`printf("%c", k);`	
#2	`return q+2;`	
	`}`	
	`int Dee(int x)`	
	`{`	
#1	`return x * x;`	
	`}`	

- main.1
  - scanf.1
    …
  - scanf.n
- main.1
- main.2
  - Bee.1
    - Cee.1
      - printf.1
        …
      - printf.m
    - Cee.1
    - Cee.2
  - Bee.1
  - Bee.2
    - Dee.1
  - Bee.2
  - Bee.3
- main.2
- main.3
  - printf.1
    …
  - printf.m
- main.3
- main.4

- How do we keep track of which function is executed and how do we know how to return back to the proper location?
- This is accomplished by placing some useful info (assembled into the ***activation record*** or ***stack frame***) about each function invocation onto the **run-time stack**
  - As a function is called, its *activation record* is pushed onto the run-time stack
  - When the function exist, its *activation record* is popped off the stack
    - R6 keeps track of the top of the stack address



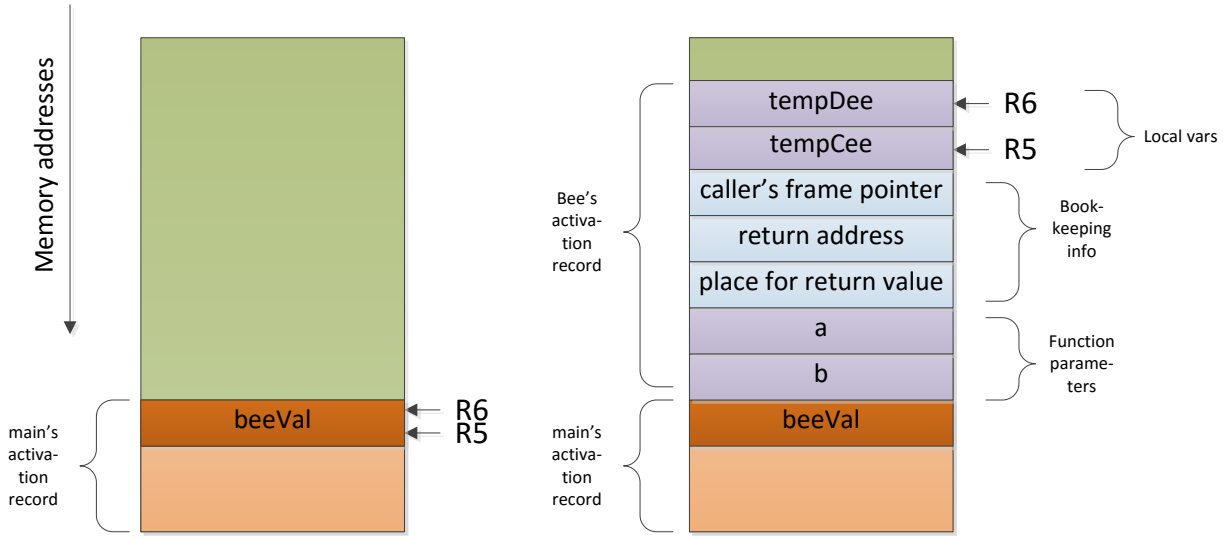
Activation record (stack frame)



- Bookkeeping info
  - Caller's frame pointer
  - Return address
  - Return value

- Step-by-step walkthrough the function invocation
  - Code in the caller function copies its arguments into a region of memory accessible by the callee
  - The code at the beginning of the callee function pushes its activation record onto the stack and saves some bookkeeping info
  - The callee executes
  - When finished, callee's activation record is popped off the stack and control is returned to the caller
  - Colle's return value is retrieved

- Example: calling **Bee**(int a, int b) from **main**:
  - `beeVal = Bee(10, beeVal);`



  - R5 – frame pointer; points to first local variable
  - R6 – stack pointer; points to the top of the stack
  - The function arguments are pushed onto the stack from right to left
  - Local variables are allocated memory in the order they appear in the code
- **The call:** **`beeVal = Bee(10, beeVal);`**
  - 1. Arguments `(10, beeVal)` are pushed onto the run-time stack

```
LDR R0, R5, #0  ; load beeVal in R0
ADD R6, R6, #-1 ; move stack pointer
STR R0, R6, #0  ; push beeVal onto the stack
AND R0, R0, #0  ; clear R0
ADD R0, R0, #10 ; R0 <- 10
ADD R6, R6, #-1 ; move stack pointer
STR R0, R6, #0  ; push 10 onto the stack
```

  - 2. Control is transferred to Bee function
    - `JSR BEE`
- **Starting the callee function:** **`int Bee(int a, int b) {`**
  - 1. Memory is allocated for the return value (callee pushed a memory location onto the stack)

```
BEE   ADD R6, R6, #1 ; allocate space for the return
value of type int
```

  - 2. Return address which is currently stored in R7 is saved

```
ADD R6, R6, #-1
STR R7, R6, #0 ; push value from R7 onto the stack
```

- 3. Caller's frame pointer (current value of R5) is saved onto the stack
  - ```
    ADD R6, R6, #-1
    STR R5, R6, #0 ; push value from R5 onto the stack
    ```
- 4. The callee allocates enough space on the stack for its local variables and adjusts frame pointer (R5) to point to its first local variable
  - ```
    ADD R5, R6, #-1 ;  R5 <- R6-1
    ADD R6, R6, #-1 ; space for tempCee
    ADD R6, R6, #-1 ; space for tempDee
    ```

- **Ending the collee function: `return tempDee; }`**
  - 1. Return value is written into the return value entry of the activation region
    - ```
      LDR R0, R5, #-1; R0 <- tempDee
      STR R0, R5, #3 ; return value stored
      ```
  - 2. The local variables are popped off the stack
    - ```
      ADD R6, R6, #2 ; two local variables are gone
      ```
  - 3. Caller's frame pointer and return address are restored
    - ```
      LDR R5, R6, #0 ; pop the frame pointer
      ADD R6, R6, #1
      LDR R7, R6, #0 ; pop the return address
      ADD R6, R6; #1
      ```
  - 4. Control is returned to the caller's function
    - ```
      RET
      ```
- **Returning to the caller function: `beeVal = Bee(10, beeVal);`**
  - 1. Return value is popped off the stack and copied to the caller's local variable
    - ```
      LDR R0, R6, #0 ; R0 <- return value
      STR R0, R5, #0 ; beeVal <- R0
      ADD R6, R6, #1
      ```
  - 2. The arguments are popped off the stack
    - ```
      ADD R5, R6, #2 ; two arguments are gone
      ```
  - At this point, the execution continues in the main function as if nothing happened!

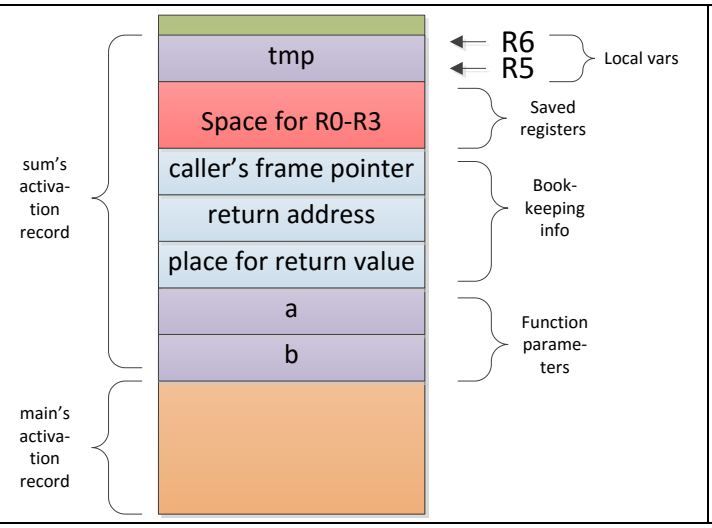
Saving registers in the activation record

- On saving values of registers R0-R3
  - Registers R4-R7 are reserved (used for specific purposes), therefore we should not use them in our code
  - Registers R0-R3 are available for use
  - In general, we would like to make sure the callee function does not modify R0-R3
    - We can save their values in the function's activation record and then restore them
    - Saving/restoring can be done in the caller function (caller-save) or in the callee function (callee-save).

```
main()
{
     int a, b, c;
     …
     a = 10;
     b = -10;
     c = sum(a, b);
     …
}

int sum(int a, int b)
{
     int tmp;
     tmp = a + b;
    return tmp;
}
```

1. The call: `c = sum(a, b);`
   - *SAVE R0-R3 (IF CALLER-SAVE)*
   - Push a, b (arguments) onto the run-time stack
   - JSR SUM
2. starting the callee function `int sum(int a, int b) {`
   - Allocate space for return value, R5, and R7
   - Store R5 and R7
   - *Allocate space for R0-R3 registers*
   - *SAVE R0-R3 (IF CALLEE-SAVE)*
   - Set new frame pointer in R5
   - Allocate memory for local variables
3. execute the function: `tmp = a + b;`
4. ending the collee function: `return tmp;`
   - Store return value
   - Pop off the stack local variables
   - *RESTORE R0-R3 (IF CALLEE-SAVE)*
   - Restore R7 and R5
   - `RET`
5. returning control to the caller function: `c = sum(a, b);`
   - Get returned value
   - *RESTORE R0-R3 (IF CALLER-SAVE)*
   - Pop off the stack return value and parameters



1. Caller-save
   - main function saves values of R0-R3 in its bookkeeping info region before calling sum subroutine
   - main function restores R0-R3 once the control is returned back to it
2. Callee-save
   - sum function saves values of R0-R3 into its bookkeeping region at the start
   - sum function restores R0-R3 just before returning