• Implementation of C functions at the lower level. Part of the complexity associated with implementing functions is that in C, a function can be called from any other function in the source file (and even from functions in other object files). To assist in dealing with this, we adopt a general calling convention for calling one function from another. To assist with the fact that some functions might even call themselves, we base this calling convention on the run-time stack. The calling convention involves the caller passing the value of its arguments by pushing them onto the stack, then calling the callee. The arguments written by the caller become the parameters of the callee’s activation record. The callee does its task and then pops its activation record off the stack, leaving behind its return value for the caller.

• Using functions when programming. It is conceivable to write all your programs without ever using functions, the result would be that your code would be hard to read, maintain, and extend and would probably be hugger than if your code used functions. Functions enable abstraction: we can write a function to perform a particular task, debug it, test it, and then use it within the program wherever it is needed.

14.1 What is the significance of the function main? Why must all programs contain this function?

14.2 Refer to the structure of an activation record for these questions.
   a. What is the purpose of the dynamic link?
   b. What is the purpose of the return address?
   c. What is the purpose of the return value?

14.3 Refer to the C syntax of functions for these questions.
   a. What is a function declaration? What is its purpose?
   b. What is a function prototype?
   c. What is a function definition?
   d. What are arguments?
   e. What are parameters?

14.4 For each of the following items, identify whether the caller function or the callee function performs the action.
   a. Writing the arguments into the activation record.
   b. Writing the return value.
   c. Writing the dynamic link.
   d. Modifying the value in R5 to point within the called function’s activation record.
14.5 What is the output of the following program? Explain.

```c
void MyFunc(int z);

int main()
{
    int z = 2;

    MyFunc(z);
    MyFunc(z);
}

void MyFunc(int z)
{
    printf("%d ", z);
    z++;
}
```

14.6 What is the output of the following program?

```c
#include <stdio.h>

int Multiply(int d, int b);

int d = 3;

int main()
{
    int a, b, c;
    int e = 4;

    a = 1;
    b = 2;

    c = Multiply(a, b);
    printf("%d %d %d %d\n", a, b, c, d, e);
}

int Multiply(int d, int b)
{
    int a;
    a = 2;
    b = 3;

    return (a * b);
}
14.7 Following is the code for a C function named Bump.

```c
int Bump(int x)
{
    int a;
    a = x + 1;
    return a;
}
```

a. Draw the activation record for Bump.
b. Write one of the following in each entry of the activation record to indicate what is stored there.
   (1) Local variable
   (2) Argument
   (3) Address of an instruction
   (4) Address of data
   (5) Other

c. Some of the entries in the activation record for Bump are written by the function that calls Bump; some are written by Bump itself. Identify the entries written by Bump.

14.8 What is the output of the following code? Explain why the function Swap behaves the way it does.

```c
int main()
{
    int x = 1;
    int y = 2;

    Swap(x, y);
    printf("x = %d  y = %d\n", x, y);
}

void Swap(int y, int x)
{
    int temp
    temp = x;
    x = y;
    y = temp;
}
```

14.9 Are the arguments to a function placed on the stack before or after the JSR to that function? Why?
14.10 A C program containing the function `food` has been compiled into LC-3 assembly language. The partial translation of the function into LC-3 is:

```c
food:
  ADD R6, R6, #-2
  STR R7, R6, #0
  ADD R6, R6, #-1
  STR R5, R6, #0
  ADD R5, R6, #-1
  ADD R6, R6, #-4
...
```

a. How many local variables does this function have?
b. Say this function takes two integer parameters `x` and `y`. Generate the code to evaluate the expression `x + y`.

14.11 Following is the code for a C function named `Unit`.

```c
int main()
{
  int a = 1;
  int b = 2;

  a = Init(a);
  b = Unit(b);

  printf("a = %d b = %d\n", a, b);
}

int Init(int x)
{
  int y = 2;
  return y + x;
}

int Unit(int x)
{
  int z;

  return z + x;
}
```

a. What is the output of this program?
b. What determines the value of local variable `z` when function `Unit` starts execution?

14.12 Modify the example in Figure 14.10 to also convert each character to lowercase. The new program should print out both the lower- and uppercase versions of each input character.

14.13 Write a function to print out an integer value in base 4 (using only the digits 0, 1, 2, 3). Use this function to write a program that reads two integers from the keyboard and displays both numbers and their sum in base 4 on the screen.
14.14 Write a function that returns 1 if the first integer input parameter is evenly divisible by the second. Using this function, write a program to find the smallest number that is evenly divisible by all integers less than 10.

14.15 The following C program is compiled into LC-3 machine language and loaded into address x3000 before execution. Not counting the JSRs to library routines for I/O, the object code contains three JSRs (one to function f, one to g, and one to h). Suppose the addresses of the three JSR instructions are x3102, x3301, and x3304. And suppose the user provides 4, 5, 6 as input values. Draw a picture of the run-time stack, providing the contents of locations, if possible, when the program is about to return from function f. Assume the base of the run-time stack is location xEFFFA.

```c
#include <stdio.h>

int f(int x, int y, int z);
int g(int arg);
int h(int arg1, int arg2);

int main()
{
    int a, b, c;

    printf("Type three numbers: ");
    scanf("%d %d %d", &a, &b, &c);
    printf("%d", f(a, b, c));
}

int f(int x, int y, int z)
{
    int x1;
    x1 = g(x);
    return h(y, z) * x1;
}

int g(int arg)
{
    return arg * arg;
}

int h(int arg1, int arg2)
{
    return arg1 / arg2;
}
```
14.16 Referring once again to the machine-busy example from previous chapters, remember that we represent the busyness of a set of 16 machines with a bit pattern. Recall that a 0 in a particular bit position indicates the corresponding machine is busy and a 1 in that position indicates that machine is idle.

a. Write a function to count the number of busy machines for a given busyness pattern. The input to this function will be a bit pattern (which can be represented by an integer variable), and the output will be an integer corresponding to the number of busy machines.

b. Write a function to take two busyness patterns and determine which machines have changed state, that is, gone from busy to idle, or idle to busy. The output of this function is simply another bit pattern with a 1 in each position corresponding to a machine that has changed its state.

c. Write a program that reads a sequence of 10 busyness patterns from the keyboard and determines the average number of busy machines and the average number of machines that change state from one pattern to the next. The user signals the end of busyness patterns by entering a pattern of all 1s (all machines idle). Use the functions you developed for parts 1 and 2 to write your program.

14.17

a. Write a C function that mimics the behavior of a 4-to-1 multiplexor. See Figure 3.13 for a description of a 4-to-1 MUX.

b. Write a C function that mimics the behavior of the LC-3 ALU.

14.18 Notice that on a telephone keypad, the keys labeled 2, 3, 4, . . . , 9 also have letters associated with them. For example, the key labeled 2 corresponds to the letters A, B, and C. Write a program that will map a seven-digit telephone number into all possible character sequences that the phone number can represent. For this program, use a function that performs the mapping between digits and characters. The digits 1 and 0 map to nothing.