Faculty of Computer Science, Dalhousie University *15-Oct-2018*

CSCI 2132 — Software Development

Lecture 17: Functions and Recursion

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lecture

- Example: binary search
- Multidimensional arrays
- Variable-length arrays
- Example: Latin square

16 Functions and Recursion

A simplified definition of a *function* is C is that it is a sequence of statements grouped together and given a name. This concept is similar to the concept of a method in Java, with a difference that C functions are not members of any class, as C does not provide language features that directly support object-oriented programming. Knowing this, we can easily think of a few advantages of using functions in our programs: We can use functions to divide a program into small pieces that are easy to comprehend. Using functions allows us to avoid duplicating code that is used more than once. Functions can also be reused in different programs. In addition, recursive functions provide natural solutions to certain problems.

16.1 Function Definitions and Declarations

The following is an example of function definitions:

```
int max(int a, int b) {
   int c;
   c = (a > b) ? a : b;
   return c;
}
```

In the first line of the the above example, the `int` at the beginning of the line is the return type of the function, `max` is the function name, and variables defined within the round parentheses are function parameters. The remaining lines form the function body. In the function body, we defined a local variable `c`. This kind of variables is called local because they cannot be used outside the function body. This function takes from its caller two `int` values, and returns the one that is larger.

The above function is a very simple function, which could be rewritten in this shorter way:

```
int max(int a, int b) { return (a > b) ? a : b; }
```

A function does not have to return a value. If this is the case, then use `void` as its return type. It may still have return statements, but these statements cannot include return values. Functions do not have to have parameters, either. If a function does not have parameters, we can simply use empty parentheses, as in `( )` or put the keyword `void` inside the parentheses.

To call a function, we simply write down its name, followed by an argument list. For example, we can call the above function in the following main function:

```
int main() {
  int a = 5;
  int b = 4;

  printf("%d\n", max(a,b) );
  return 0;
}
```

For efficiency reasons, C functions cannot return arrays. This would require copying of arrays, which is slow when the arrays are large. (For the same reason, we have to write a loop to copy the values from one array to another.)

Now, let us revisit the functions `printf` and `scanf` and examine what values they return.

The `printf` function returns the number of characters printed. In most cases, we do not make use of this value. We can use the `void` keyword to explicitly specify that this is the case. For example, we can write the following statement:

```
(void) printf("hello, world\n");
```

This `(void)` is optional, and rarely used, but it can be used to add additional documentation to a program.

The return value of the `scanf` function is used more often, as it is the number of data items successfully read and assigned to the variables. If an input failure occurs before any data item can be read, a constant `EOF` (a negative macro constant) is returned. For example, if we use the `scanf` function to read two `int` values, and we would like to check whether it successfully reads two `int` values, we can write the following code:

```
if ( 2 != scanf("%d %d", &i, &j) ) {
  printf("invalid input\n");
  return 1;
}
```

Function Declarations or Function Prototypes

Before we learn more about functions, let us learn more about function declarations. The syntax of a function declaration, or function prototype, is:

```
return-type function-name(parameters);
```

This is essentially a function definition without the function body, but with an additional trailing semicolon. The function declaration is used to inform the compiler about the function, its return value, and its parameter, so that compiler can properly recognize it even before it is defined. It is particularly important in situations where a function is use in one source file and defined in another source file.

This is example, which is an extension of the previous example with the `max` function, in which we use a function declaration of the function `max`:

```
#include <stdio.h>
```

```
int max(int a, int b);

int main(void) {
  int a = 5, b = 4;
  printf("%d\n", max(a,b));
  return 0;
}

int max (int a, int b) {
  return (a < b) ? a : b;
}
```

The function declarations are also called *function prototypes.* The parameter names can be omitted in the function declarations, so the above function declaration can be written also as:

```
int max(int, int);
```

In C99, either a declaration or a definition of a function must be present prior to the statement that calls this function. This is optional in C89 or earlier. However, even in C89 and earlier versions, it was always strongly recommended to use function declarations, because the default rules about function return types and parameters that compiler uses when encountering a call to an undeclared and yet undefined function do not necessarily match what programmer intended and may likely lead to some bugs or compilation issues.

A good programming style is to provide prototypes of all functions (except the main) before the main function, and then place their definitions after main, as shown in the example above.

16.2 Arguments

Although the terms function *arguments* and *parameters* are frequently used interchangeably, we usually call *arguments* (or *actual parameters*) the expressions used in a function call, while *parameters* (or *formal parameters*) are used to refer to the parameters as used in a function definition.

Arguments Passed by Value

The arguments in C are passed *by value,* unlike passing by reference, which is used in some other languages. This means that the values of the actual parameters are passed to the function, and any changes to the formal parameters in the function will not affect the original variables used as arguments in the function call.

Arrays behave in a special way, since if we make some changes to array parameters, those changes will affect the original arrays. This means that they appear to be passed by reference. However, we still say that they are passed by value, but as pointers. This will be better explained later when we talk about the pointers.

To illustrate passing by value, we can look at the following code snippet:

```
void swap (int a, int b) {
   int temp = a;
   a = b;
   b = temp;
}

...
```

```
int a = 4;
int b = 5;
swap(a, b);
printf("a=%d, b=%d\n", a, b);
```

The above printf statement would print "`a=4, b=5`". Thus the swap function is not implemented correctly. Later on, we will learn how to use pointers to implement the swap function correctly.

Passing Array Parameters

When an array is the parameter of a function, we usually want to pass its length as another parameter. The following is the prototype and definition of a function that is supposed to return the maximum value of an array (the actual code in its body is omitted):

```
int max_array(int, int []);

int max_array(int len, int a[]) {
...
}
```

If a function changes the elements of an array parameter, the change is reflected in the corresponding argument. This appears to contradict the statement that arguments are always passed by value in C, but in fact there is no contradiction, and we will know why not when we learn pointers in C. For now, it is sufficient to know this fact.

16.3 Call Stack and Recursion

At the low level, what exactly happens when a function is called? To answer this we need remind ourselves of the process memory partition into: code, data, stack, and heap. So far, we should understand that the compiled code of a program is stored in the code part of the memory. The data part of the memory is used for static data, such as literal strings and static variables; i.e., the permanent variables unlike the temporary variables used in functions. The static variables in C are declared outside any function body, as in the following example:

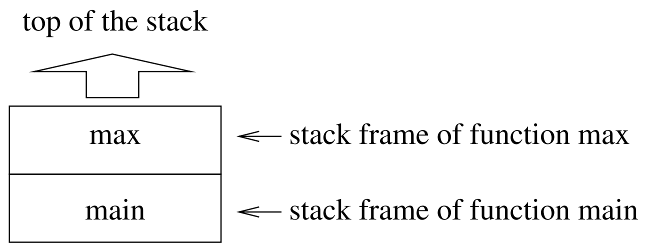
```
#include <stdio.h>

int A[10][10], Asize; /* Static variables */

int main() {
    scanf("%d", &Asize);
    /* etc... */
    return 0;
}
```

So far we have not seen how exactly are *stack* and *heap* used. In particular, the *stack* becomes important when talking about local variables in functions and function calls. By this time, you should be familiar with the stack as a data structure. (If not, you can check your notes for the course Computer Science II.) A *call stack* is a stack data structure in memory that stores information about active functions of a program.

When a function is called, the compiler generates code for creating a stack frame containing information needed for execution of a function, such as memory storage for its parameters, return value, local variables, and return address when function is finished. This stack frame is also called an *activation record* and it is pushed onto the call stack. Thus, the top of the call stack always store the stack frame of the most recently called function. For example, when the `main` function calls the `max` function, the content of the call stack is:



In this way, information in the stack frame at the top is readily available. In the above example, max is currently executing, so its stack frame is at the top and we can access its arguments and local variables. This is also the place where the function will leave its return value. The stack frame contains the return instruction pointer, so that CPU knows where to continue execution once we leave the function, and also the previous base of the stack frame; i.e., the beginning of the stack frame of the calling function.

When the function whose stack frame is at the top of the call stack returns, its frame is popped out of the call stack, and its return value is given to the function that called it. The stack frame of the latter is now at the top of the call stack, so that it can resume executing.

Recursion

Recursion is very useful for tasks that can be defined in terms of similar subtasks. You learned recursion in Computer Science II, so let us see how to write recursive functions in C.

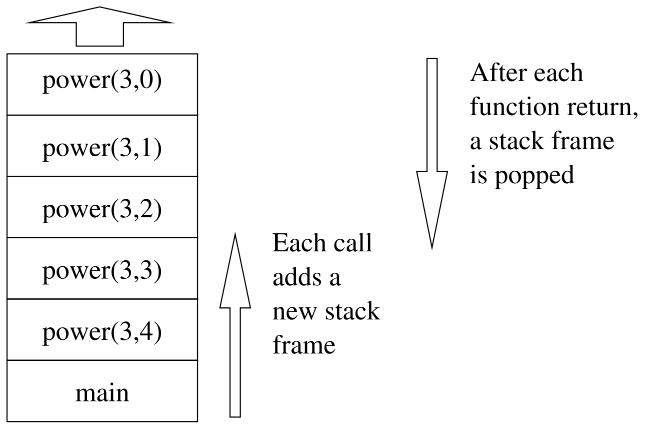
First, let us look at a simple example of a function `power` that can be used to compute x^n for given two integers x and n, where $n \geq 0$:

```
int power(int x, int n) {
    if (n == 0)          /* Base case */
        return 1;
    else                 /* Recursive case */
        return x * power(x, n-1);
}
```

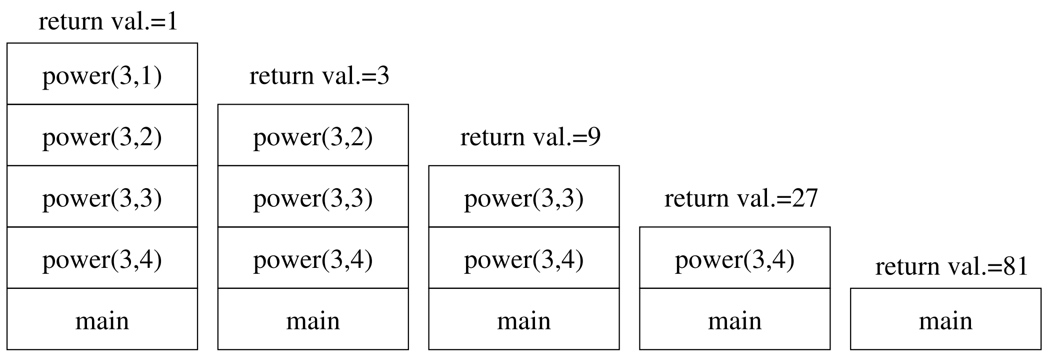
Admittedly, this is not a great example, since using a loop would use less memory, but it will do for educational purposes.

In this example, when n is equal to 0, we need not call the function `power` recursively to compute the result. This is called the *base case,* which handles a subtasks the function can perform without calling itself. The other case, handled in the else part of the if statement, is the *recursive case,* in which the function calls itself.

Now, let us see what happens in the call stack when we call `power(3, 4)` in the `main` function. In the beginning, the call stack has the stack frame of the `main` function only. When it calls `power(3, 4)`, the stack frame of `power(3, 4)` is pushed into the call stack. Then, `power(3, 4)` calls `power(3, 3)`, so another stack frame for `power(3, 3)` is pushed. The following is the content of the call stack when `power(3, 0)` is called:


power(3,0)
power(3,1)
power(3,2)
power(3,3)
power(3,4)
main
Each call adds a new stack frame
After each function return, a stack frame is popped

The call `power(3, 0)` is the base case of this recursive function. When it finishes executing, it returns 1 to `power(3, 1)` and its stack frame is popped out of the call stack:



This keeps on going until the value 81 is returned from `power(3, 4)` and `main` is the only function whose stack frame is in the call stack.

To compute x to the power of n, we can simply write an iterative solution using a for loop by multiplying x with itself $n - 1$ times. Which solution is better here, the iterative one or the recursive one? Now that we learned how call stack works, we can understand that the iterative solution is more efficient, as pushing and popping stack frames requires more CPU time and memory storage than a simple iterative solution.

16.4 Merge Sort Example

Merge Sort Example

Let us see some examples for which the recursive solutions are more natural. One example is the *merge sort* algorithm that you probably learned in a previous course. The underlaying algorithm design paradigm is called divide-and-conquer, and it consists of three steps:

Divide: Divide the n-element array to be sorted into two subarrays of $n/2$ elements each.

Conquer: Sort the two subarrays recursively using the same algorithm: merge sort

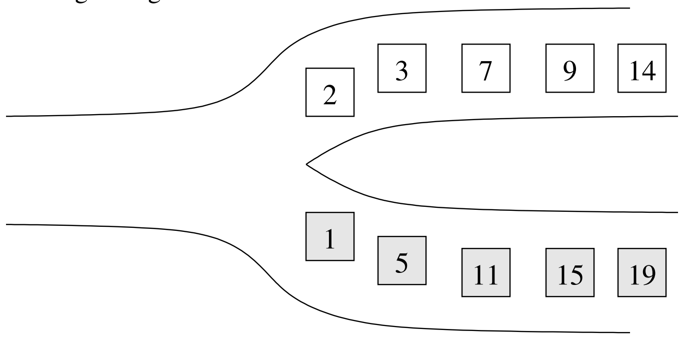
Combine: Merge the two sorted subarrays to produce the sorted answer.

Merge Sort: Example

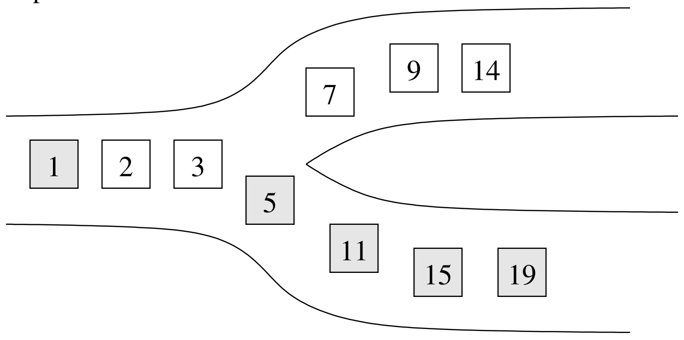
- Let us consider the following array of 10 elements:
  (9, 7, 14, 2, 3, 19, 11, 5, 1, 15)
- We divide the array into two approximately equally long-subarrays: (9, 7, 14, 2, 3) and (19, 11, 5, 1, 15)
- We assume that these subarrays are sorted, which is exactly the same task with smaller arrays, so we get:
  (2, 3, 7, 9, 14) and (1, 5, 11, 15, 19)
- Now, we can get the final sorted array using a merge operation

Example of a Merge Procedure

- Starting configuration:



- In process:


7
9
14
1
2
3
5
11
15
19