CSCI 2132
Software Development

Lecture 17:

Functions and Recursion

Instructor: Vlado Keselj

Faculty of Computer Science

Dalhousie University

Previous Lecture

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

Functions

- The main code abstraction method:
  - sequence of statements that can be called by a name; i.e., executed
- Functions have arguments and usually return a value, and that is where name comes from
- Function definition example

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

Function Return Value

- `int` by default
- Not necessary; can use: `void`
- An array cannot be return value
- Example of calling a function:
  `printf("%d\n", max(a,b) );`
- Some known examples of functions:
  - Return value of `printf`: number of printed characters
  - Return value of `scanf`: number of converted values (or, `EOF` if error or mismatch occurs before the first value is converted)

Example with `scanf`

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

Function Declarations or Function Prototypes

- Used to declare function argument types and return type before defining the body of the function
- Needed if we call function in source code before being defined, so that compiler does not need to guess
- Syntax:

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

- Example:

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

- or:

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

An Example

```
#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;
}
```

Arguments

- Arguments vs. parameters
  - Similar terms and frequently used interchangeably
  - Strictly speaking, they are different
- Arguments, or actual parameters:
  - Expressions used in function call; e.g.:

```
max( a , 3+(b-1)/2 );
```

- Parameters, or formal parameters:
  - Variables used in function definition; e.g.:

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

Arguments Passed by Value

- Arguments in programming languages can be passed:
  - by value
  - by reference
  - and few other variations
- Arguments in C are passed by value
- Arrays behave in a special way
  - they appear to be passed by reference
  - however, they are still passed by value, but as pointers
  - to be better explained later

Code Example

```
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);
```

Passing Array Parameters

- Functions cannot tell the length of an array parameter
- Usually need to pass array length as another parameter
- For example:

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

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

Call Stack: Process Memory Partition

- What happens when a function is called?
- Remember memory partition: code, data, stack, heap
- *Code:* stores program code
- *Data:* stores static data; e.g.,

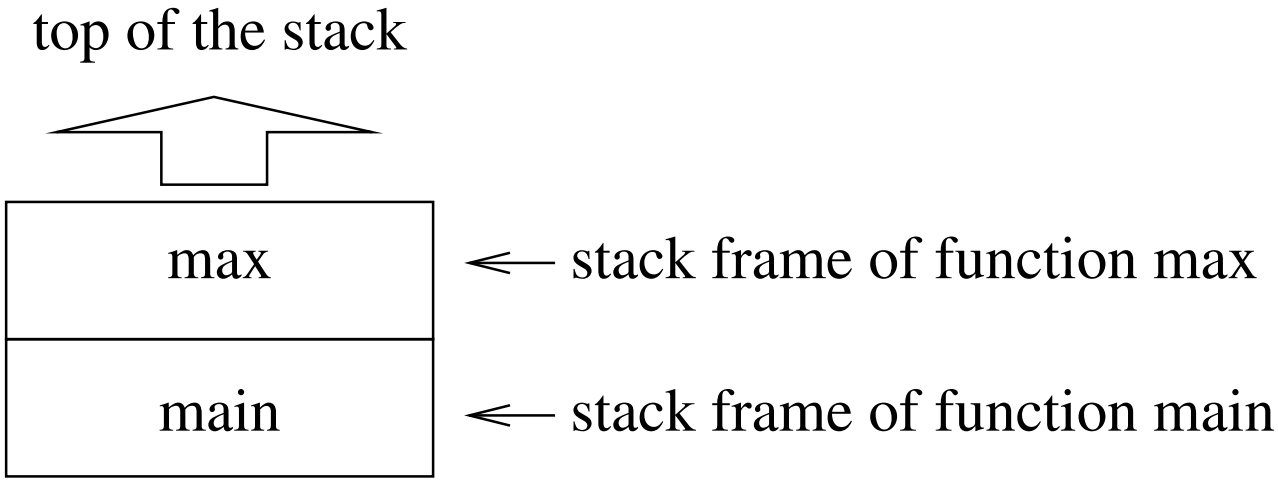
```
#include <stdio.h>

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

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

Use of Stack: Call Stack

- Have not seen yet how *stack* and *heap* are used
- Call stack is a stack in data structures sense
- Important in the context of function calling and local variables
- Call stack keeps information about active functions; i.e., functions being executed
- Whenever function starts execution: new *stack frame* is pushed on stack; also called *activation record*
- Contains information such as: arguments, local variables, return value, return instruction pointer, previous stack base address

Example

- Assume that function `main` calls function `max`
- The call stack would look as follows:



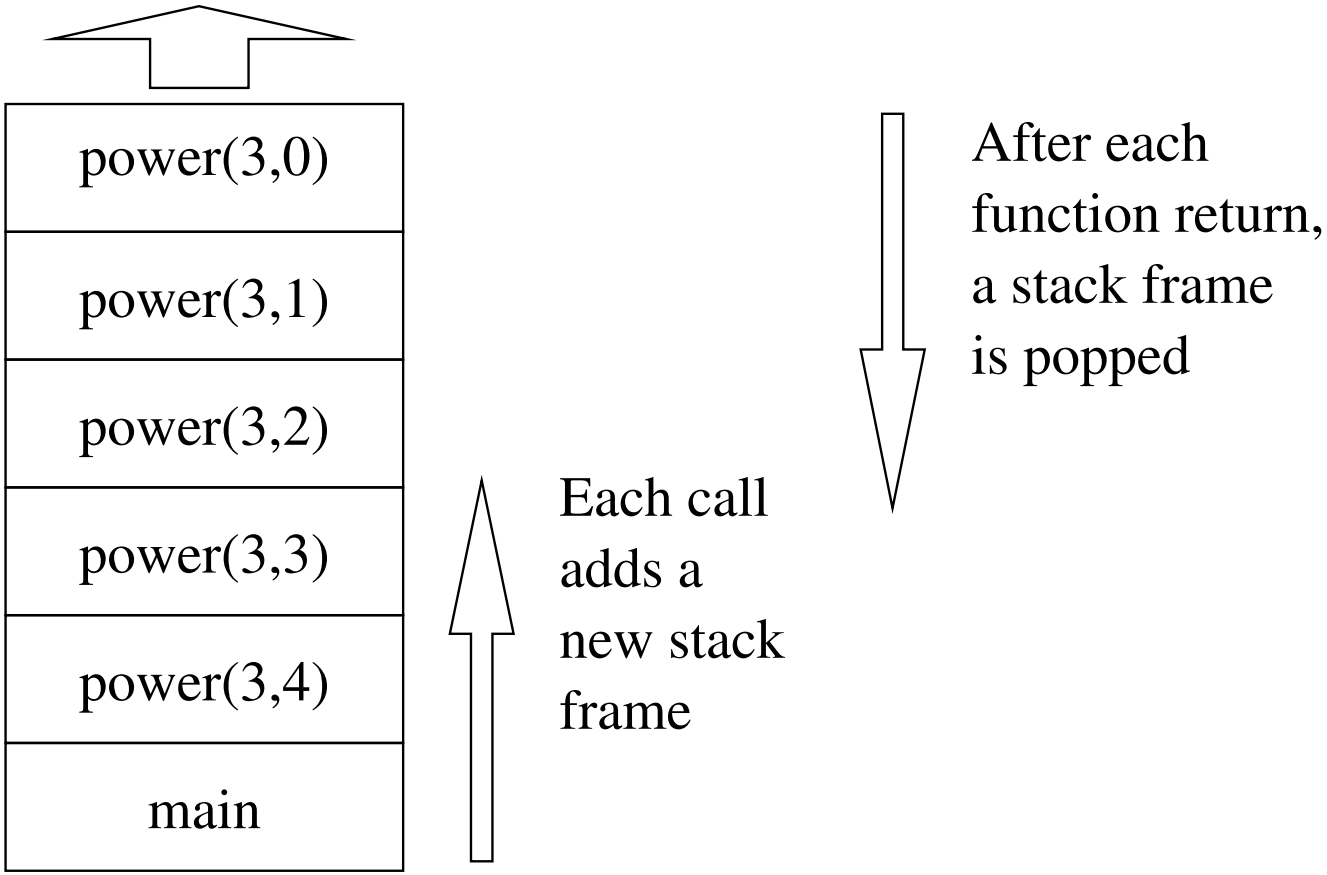
Recursion

- Let us see what role the call stack plays in recursion
- Consider the following example:

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

- Let us assume that `power(3,4)` is called from `main`

Call Stack: `power(3,4)`



Merge Sort Example

- Divide-and-conquer paradigm:

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

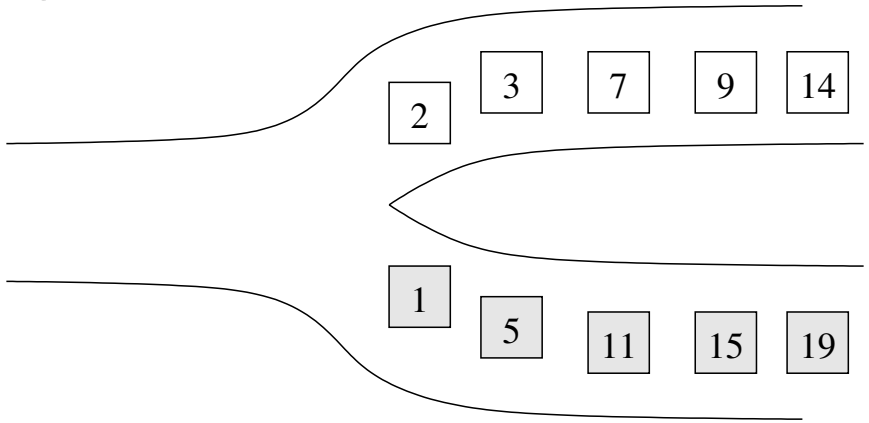
Combine: Merge the two sorted subarrays to produce the sorted answer.

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: