Functions

60-141 Introduction to Algorithms and Programming II

School of Computer Science

Term: Summer 2014

Instructor: Dr. Asish Mukhopadhyay

Motivation

An analogy

- A large C-program is like a string of pearls
  - Each pearl is a function
  - Setting up proper communication among the functions is the equivalent of stringing the pearls together

An example

- Approximate Ellipse Demo
  - Ellipse2DDouble.java
  - GeometryUtils.java
  - Main.java
  - MinCircleApprox.java
  - MinEllipse.java
  - MinEllipseApplet.java
  - MinEllipseApprox.java
  - MinEllipseFrame.java
  - Vector2DDouble.java

Program Complexity

- How do we manage it ?
  - split the computation into manageable pieces : a paradigm common to programming languages
    - These pieces (or modules) are called functions in C
    - C-functions can be user-defined or defined in the C Standard Library
  - Establish a protocol for communication among the various functions

Function Basics

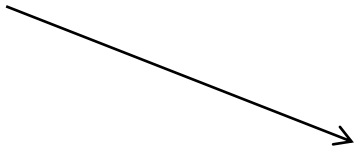
- **Problem**: Compute the Greatest Common Divisor (GCD) of two positive integers *m* and *n*
- Task decomposition
  - Read input
  - Compute GCD
  - Print GCD

m = 4 and n = 12
gcd(4, 12) = 4

m = 5 and n = 12
gcd(5, 12) = 1

Function Basics

```
1. #include <stdio.h>
2. int gcd(int, int);// function prototype

3. int main (void)
{
4. int m, n; //variables to store input
                // numbers
5.    scanf("%d %d", &m, &n); //read input
6. printf("The gcd of %d and %d is:  %d",
   m, n, gcd(m, n); // print input
7.   return 0;
} // end main
```



Function call

GCD function

```
/* definition of the gcd function */
int gcd(int m, int n)
{
    int rem;

    rem = m%n;
    /*loop until remainder is 0 */
    while(rem != 0)
    {
        m = n; // reset m and n
        n = rem;
        rem = m % n;
    } // end while
    return n;
}
```

Learning Points

?

Function definition format

```
return-type function-name (argument
declarations)
{
     declarations and statements
}
```

Function definition format

- Various parts may be absent
- If return-type is omitted, int is assumed
- A minimal function (does and returns nothing) is :

```
dummy() {}
```

Argument declarations

- A function is essentially an operator, though in a more general sense than the binary operators like +, -, *, etc.
- Operands are provided through the argument list

Return statement (1)

- The `return` statement is a mechanism for returning a value (result of an operation, we might say) from the called function to the caller
- Syntax:

`return` *`expression`*

Return statement (2)

- Any *expression* can follow return
- If necessary, its type is converted to that of the return type of the function
- If *expression* is absent then nothing is returned and control is turned to the caller when the function execution reaches the closing bracket

Function declarations

- Also known as *function prototypes*
- Placed at the beginning of the source file
- Subsequent definitions and function calls must match the declarations
- Facilitates *error-checking* by compiler and *type-coercion*

Function types

- User-defined
  - `gcd(m,n)`, for example
- Library functions
  - `scanf( )`
  - `printf( )`

Reinforce….

- Class-work:
  - **Problem**: Write a *modular* C-program to test if a positive integer is a palindromic string

Palindrome program

```
int palindrome(int number)
{
    int new_number, temp;

    new_number = 0;
    temp = number;
    while(temp >0) { /* loop until number
becomes 0 */
       new_number = new_number*10 + temp % 10;
       temp = temp /10;
    }
    return (number == new_number);
}  // palindrome
```

Palindrome program

```
#include <stdio.h>

int palindrome(int); // function prototype

int main(void)
{
    int number;

    scanf("%d", &number); /*read in number */
    if (palindrome(number)) printf("Number is a
palindrome\n");
    else
         printf("Number is not a palindrome\n");
} // end of main
```

A more complex problem (1)

- *Problem*: Print each line of a text that contains a given pattern

A more complex problem (2)

- Thus if pattern = “area” and the text is

  My research *area* is theoretical computer science

  Many interesting problems to work on in this *area*

  I love working in this field

- The output is:

  My research *area* is theoretical computer science

  Many interesting problems to work on in this *area*

Example (2)

- Task decomposition

while (there's another line)
    if (the line contains the pattern)
     print it

Example

```
#include <stdio.h>
#define MAXLINE 1000 /* maximum input line length */

// function prototypes

int getline(char line[], int max);
int strindex(char source[], char searchfor[]);

// pattern to search for
char pattern[] = “area";
```

Example (3)

```
/*find all lines matching pattern*/
int main (void)
{
    char line[MAXLINE]
    int found = 0;

    while (getline(line, MAXLINE) >0)
        if (strindex(line, pattern) >= 0) {
            printf("%s", line);
            found++;
        }
    return found;
}
```

Example (4)

- getline(line, MAXLINE)

```
/*getline: get line into s, return length*/
int getline(char s[], int lim)
{
  int c, i;
  i= 0;
  while (--lim > 0 && (c = getchar()) !=EOF && c != '\n')
      s[i++] = c;
  if (c == '\n')
      s[i++] = c;
  s[i] = '\0';
  return i;
}
```

Example (4)

- strindex(line, pattern)

```
/* strindex: return index of t in s, -1 if none */
int strindex(char s[], char t[])
{
       int i, j, k;
       for (i=0; s[i] != '\0'; i++) {
           for (j=i, k=0; t[k] != '\0' && s[j] == t[k]; j++, k++)
       if (k > 0 && t[k] == '\0')
          return i;
       }
       return -1;
}
```

Another Example

```
/* atof: convert string s to double */

double atof(char s[])
{
        double val, power;
        int i, sign;

        for (i=0; isspace(s[i]); i++);  /* skip
white spaces */
        sign = (s[i] == '-') ? -1: 1;
        if (s[i] == '+' || s[i] == '-') i++;
        for (val = 0.0; isdigit(s[i]); i++)
             val = 10.0 * val + (s[i] - '0');
        if (s[i] = '.') i++;
        for (power = 1.0; isdigit(s[i]); i++){
             val = 10.0 * val + (s[i] - '0');
             power *= 10.0;
        }
        return sign * val/power;
}
```

Learning Points

?

Function Basics

- C-Functions
  - Cannot be nested
  - Can call each other (itself ?)
  - Communicate through
    - Arguments
    - Return values
    - External (global) variables
  - Can occur
    - In any order in a source file
    - Can be distributed across multiple source files

Function Basics

- Any C- program consists of a set *external* objects
    - What are these ?

External Objects

- *All functions*, as C does not permit nesting of functions
- Variables declared outside all functions
  - These allow communication between different functions
  - This comes with a warning ！
    - Undisciplined use can lead to serious programming errors

Point to Note

- In contrast, variables inside a function definition are called *automatic (or local)* variables
- Function parameters are also in effect local variables

Scope Rules (1)

- The scope of a name (identifier) is the part of the program within which the name can be used
- Thus the scope of an
  - *automatic variable (local variable)* is the function in which it is declared*
  - External *variable* or *function* is from the point where it is *declared* to the end of the file being compiled

Scope Rules (2)

- If an *external variable* is to be referred to
  - before it is defined or
  - is defined in a *different* source file
- then, an **extern** *declaration* is mandatory
  - Let's see an example …….

An example

```
#include <stdio.h>

void f(void); // function prototype
extern int i;  // declaring i before definition; essentially changes
               // the scope of i

int main(void)
{
   printf("The value of i is: %d\n", i);
   f();
   return 0;
}

int  i = 1;  // definition of i
void f(void)
{
    printf("The value of i is: %d\n", i);
}
```

Example (1)

- This program prints 1, following scope rules

```
#include <stdio.h>

int i = 0;

int main(void)
{
  int i = 1
  printf("%d", i);
  return 0;
}
```

Example (2)

- This program prints 0, following rules for external variables

```
#include <stdio.h>

int i = 0;

int main(void)
{
  extern int i;
  printf("%d", i);
  return 0;
}
```

Example (3)

- This program also prints 0, as the reference is to the external variable i

```
#include <stdio.h>

int i = 0;

int main(void)
{
  printf("%d", i);
  return 0;
}
```

Static variables

- Static *declaration* applied to
  - an *external* variable limits the scope of the variable to the rest of the source file being compiled
  - a *function* limits the visibility of the function to the file in which it is declared
  - an *internal* variable makes the variable remain in existence rather than be transient with function activation

Register variables

- Declaration Syntax:

```
register int x;
register char x;
```

- Indicates to the compiler that the variables will be heavily used ; so, placing them in machine registers will result in faster programs
- Only automatic variables and function arguments can be thus declared
- There is an upper limit on the number of such variables

Block Structure

- Despite no function-nesting, variables can be declared inside a function in a block-structure fashion thus:

```
if (n > 0) {
    int i; /* declare a new i */
    for (i=0; i < n; i++)
      ….
}
```

- *i* remains in existence until } , and any identically named variable outside the block is hidden from the scope of the block

Recursion

- A C-function can call itself to implement a recursive computation
- Consider this problem:
  - Print an integer number as a character string
- Discounting the sign of the number, the difficulty here is that while the printing must be done from left to right, the digits can be extracted from right to left

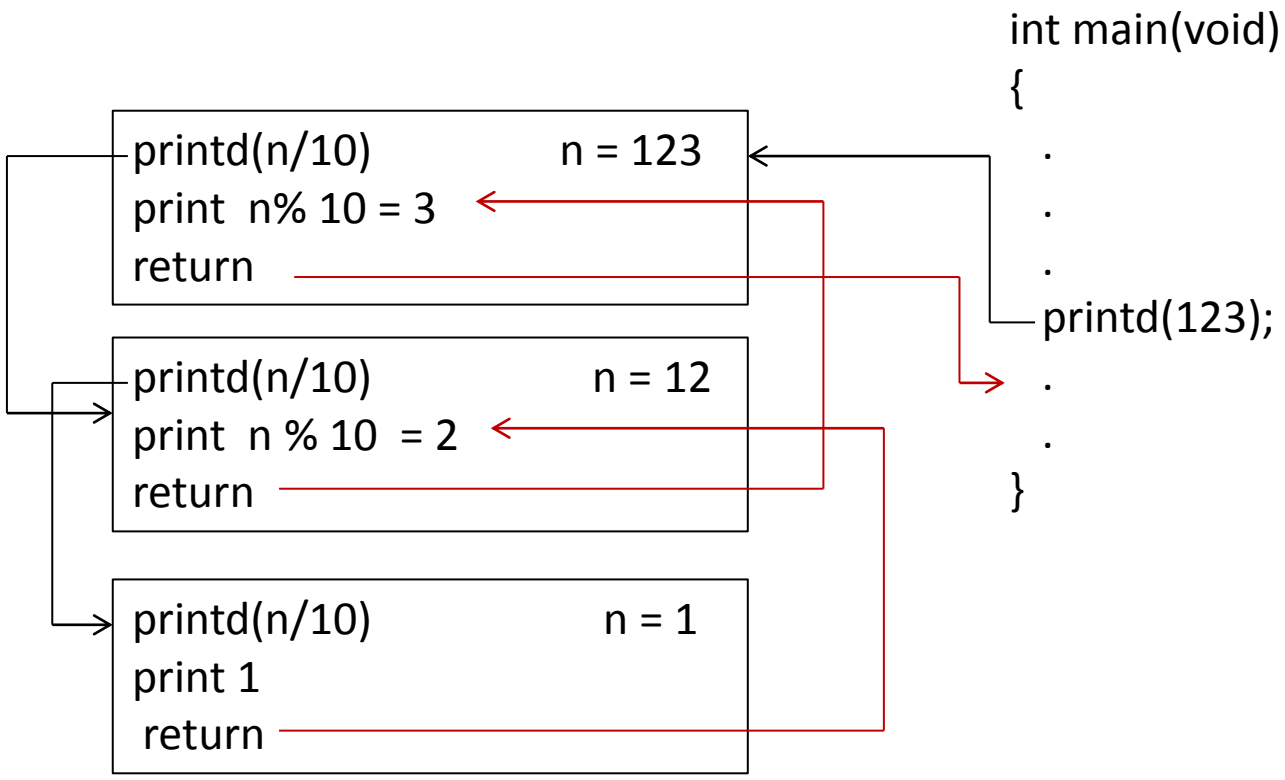
Printing Solution

```
#include <stdio.h>

/*printd: print n in decimal */
void printd(int n)
{
    if(n < 0) {
        putchar('-');
        n = -n;
     } // end of if
     If (n/10)
         printd(n/10);  // call to printd with a smaller argument
      putchar(n%10 + '0');
} // end of printd
```

Points to Note

- The function printd (n)
  - calls itself with a *smaller value*, viz., n/10 if it is greater than 0 (inductive step)
  - else prints the one-digit number (base step)
- This is characteristic of a recursive function
- Each invocation gets a fresh set of automatic variables, independent of the previous set
- Recursive code is compact but not necessarily *faster* or *space-saving*

Unwinding the recursion



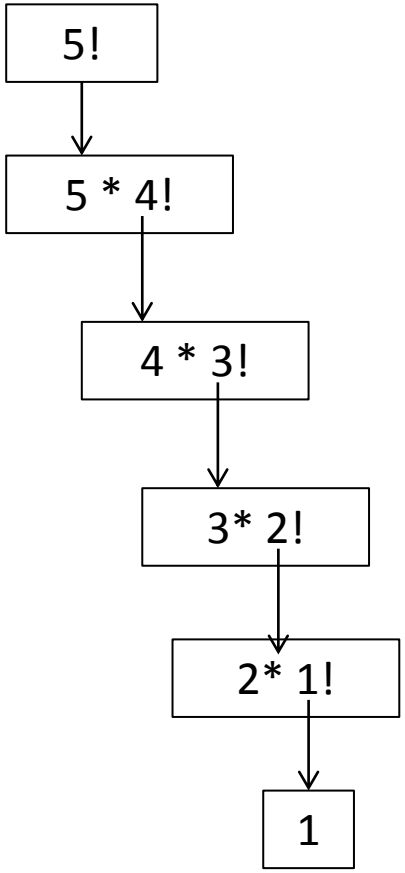
Recursive Computation: Another example

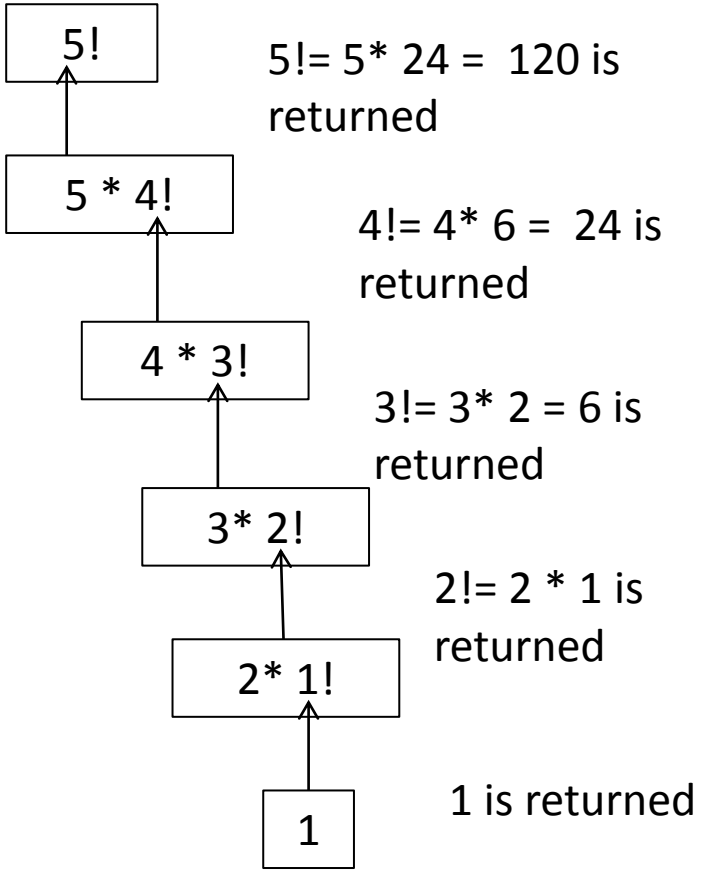
- Consider this computation:
  - factorial(n) = n* factorial(n-1), n >= 1
  - factorial(0) = 1
- Characteristic features:
  - Can't know what is factorial(n) until we know factorial(n-1), …..
  - That factorial(0) = 1 helps us compute factorial(n) by going backwards

C-program for n!

```
long factorial(int n)
{
    if (n == 0) return 1;
    else return (n*factorial(n-1));
} // end of factorial
```

Activation stacks





Tower of Hanoi

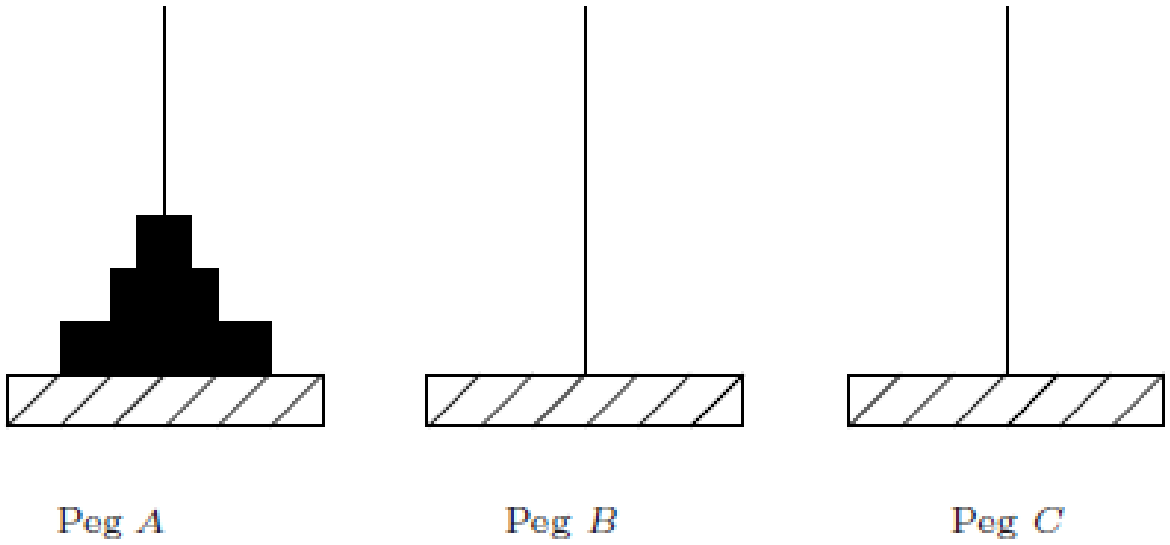


Figure 4.1: Towers of Hanoi: Initial configuration

Tower of Hanoi

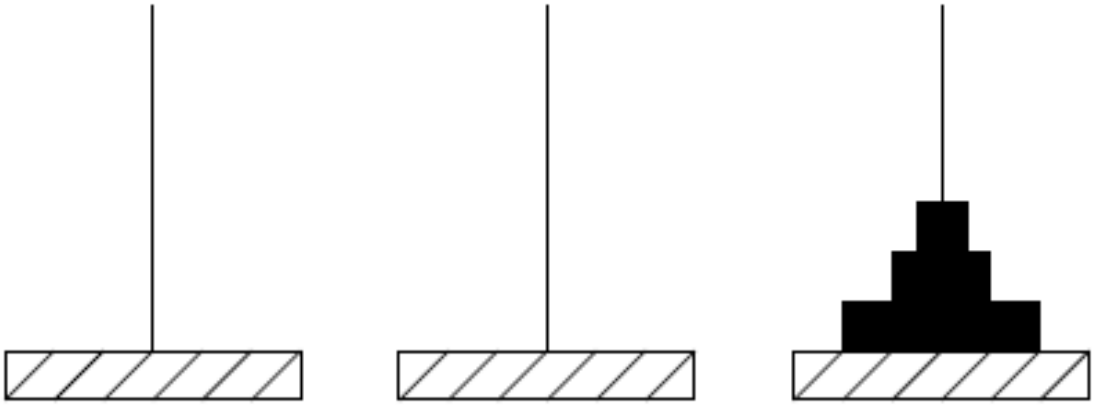


Figure 4.2: Towers of Hanoi - Final configuration

Tower of Hanoi

- Rule 1
  - We can move only one disk at a time
- Rule 2
  - We may not place a larger disk atop a smaller one in any move

Recursive solution

$$\text{Move}(n, A, C, B) =$$

$$\text{Move}(n-1, A, B, C) +$$

$$\text{Move}(1, A, C, B) +$$

$$\text{Move}(n-1, B, C, A)$$

Move(n, sourceDisk, targetDisk, workDisk)

Number of moves ?

- $2^n - 1$

- Proof ?!!!!! Again!!!

C-program for Tower of Hanoi

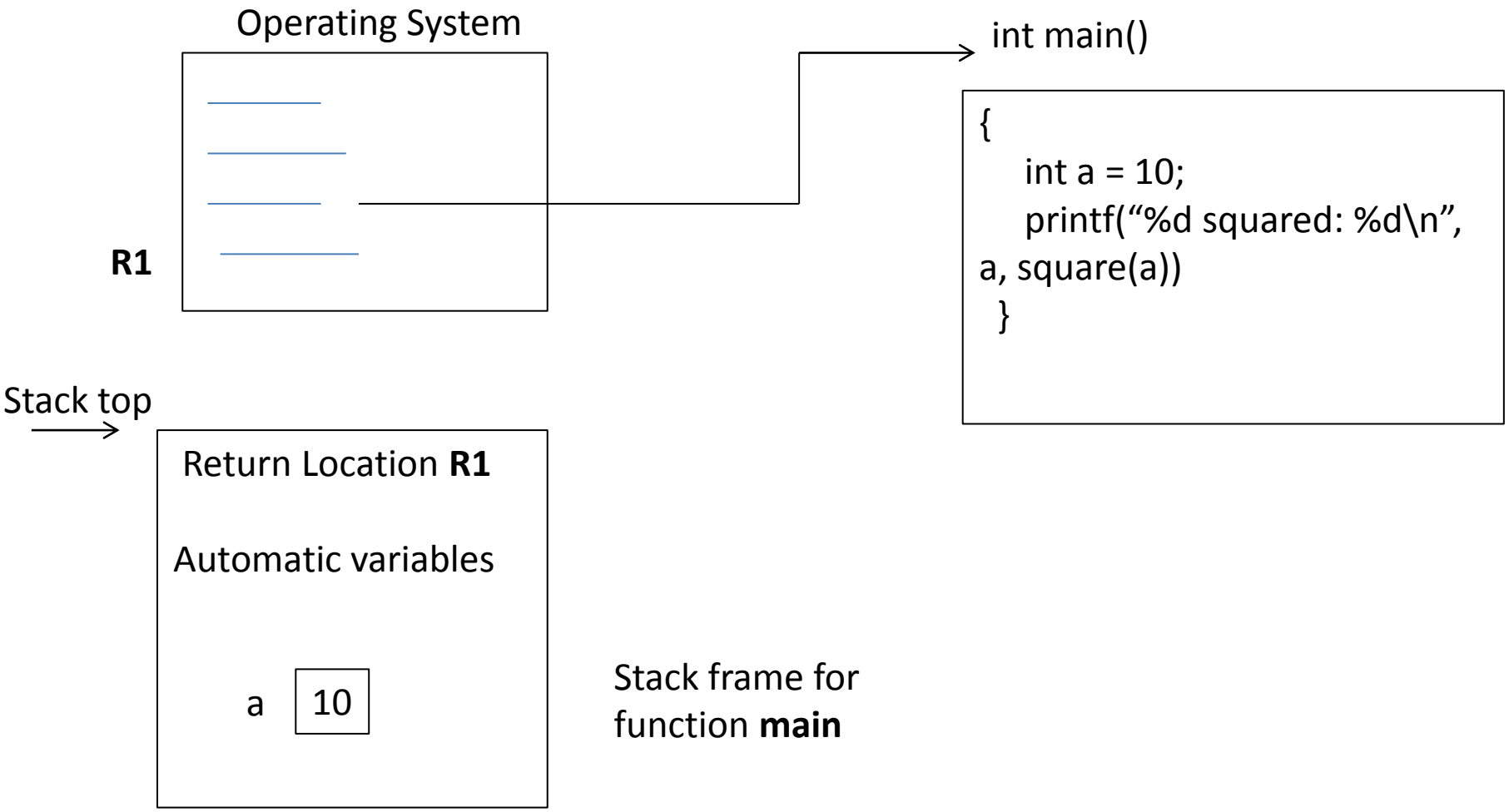
```
#include <stdio.h>
void generateMoves(int, int, int, int);


int main(void)
{

    int numberOfPegs;
    int pegA = 1;
    int pegB = 2;
    int pegC = 3;

    printf("Input the number of pegs:");
    scanf("%d", &numberOfPegs);
    printf("\n");
    generateMoves(numberOfPegs, pegA, pegB,
pegC);
}
```

```
void generateMoves(int numberOfPegs, int
pegA, int pegB, int pegC)
{
     //if (numberOfPegs == 0) printf("This
is the end of all moves \n");
     if (numberOfPegs == 0) return;
     else{
           generateMoves(numberOfPegs - 1,
pegA, pegC, pegB);
           printf("Move disk from %d to
%d\n\n", pegA, pegC);
           generateMoves(numberOfPegs - 1,
pegB, pegA, pegC);
           }
}
```

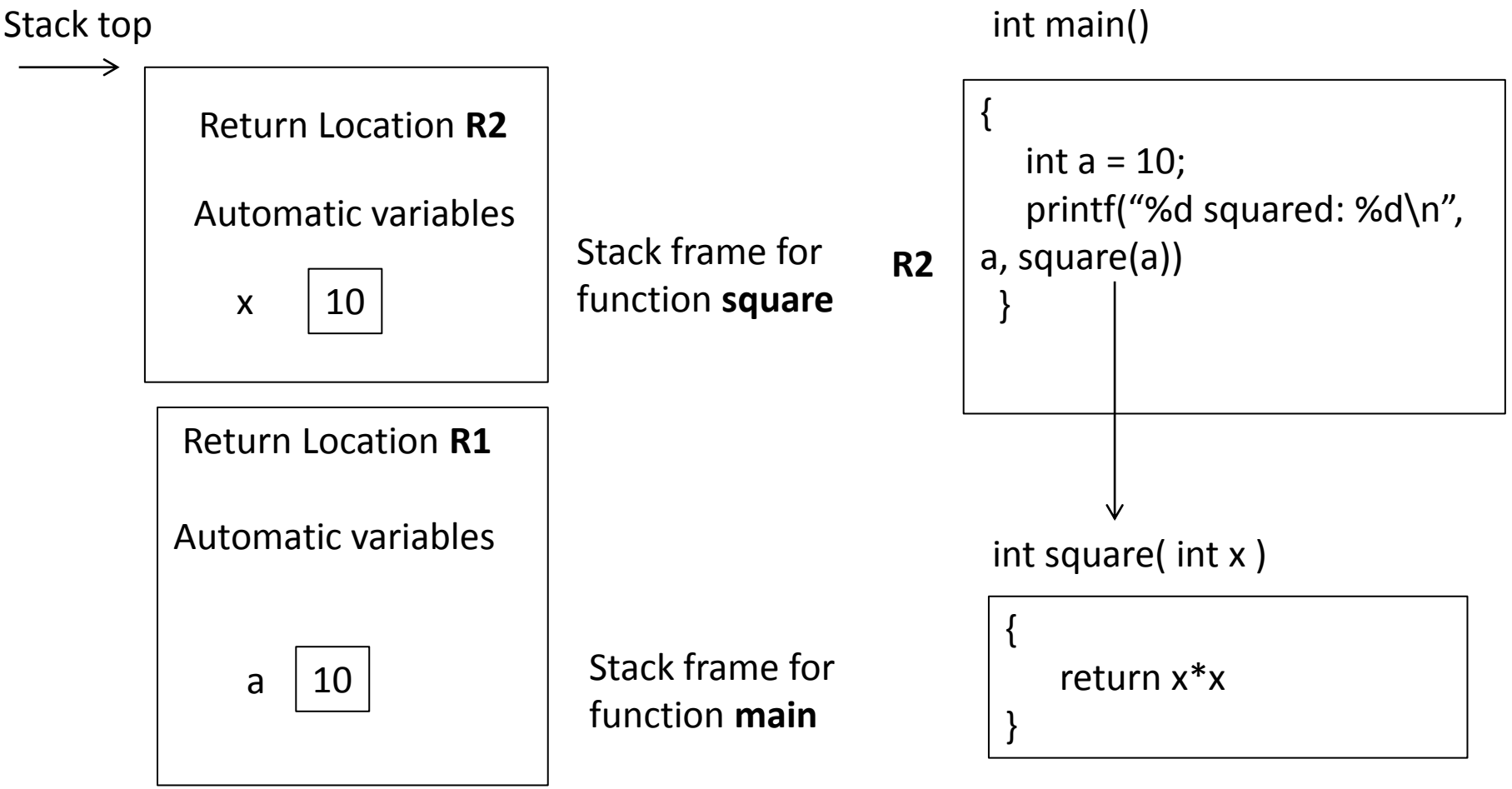
Function call Activation Frames

- Just as a recursive C program is implemented using a stack of activation calls, functions calling each other is also implemented using a stack frame

Example


Operating System
R1
int main()
{
int a = 10;
printf("%d squared: %d\n",
a, square(a))
}
Stack top
Return Location R1
Automatic variables
a
10
Stack frame for
function main

Example


Stack top
Return Location **R2**
Automatic variables
x
10
Stack frame for function **square**
Return Location **R1**
Automatic variables
a
10
Stack frame for function **main**
int main()
{
int a = 10;
printf("%d squared: %d\n",
a, square(a))
}
R2
int square(int x)
{
return x*x
}

Yet another example

- Write a C-program to reverse a string in place recursively

C-program

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

void  reverseR(char *, int , int);  // function prototype

int main(void)
{

   char s[] = "abcde";

   reverseR(s, 0, strlen(s));
   printf("%s\n", s);

}
```

C-Program (2)

```
void reverseR(char *s, int i, int len)
{
    int c, j;

    j = len - (i+1);
    //printf("j is %d: \n", j);

    if(i < j) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
        reverseR(s, ++i, len);
    }
    else return;
}
```

To do

- Read Chapter 5 of your Deitel and Deitel text, focussing on the worked out examples, review exercises.
- Solve as many self-review exercises as you can
- Next lecture on arrays and pointers (maybe)