BBM 101 – Introduction to Programming I
Fall 2014, Lecture 7

Aykut Erdem, Erkut Erdem, Fuat Akal
Today

- **Functions**
  - Definitions
  - Invocation
  - Parameter Lists
  - Return Values
  - Prototypes

- **Recursion**
  - Recursion
  - Inductive reasoning
  - Divide and conquer

- **Variable Scopes**
  - Block Structure
  - Global and Local Variables
  - Static Variables
Today

- Functions
  - Definitions
  - Invocation
  - Parameter Lists
  - Return Values
  - Prototypes

- Variable Scopes
  - Block Structure
  - Global and Local Variables
  - Static Variables

- Recursion
  - Recursion
  - Inductive reasoning
  - Divide and conquer
Introduction

*Structured Programming* is a problem-solving strategy and a programming methodology that includes the following two guidelines:

- The flow of control in a program should be as simple as possible.
- The construction of a program should embody *top-down* design.
Top-down Design

*Top-down design* (stepwise refinement, or *divide and conquer*) consists of repeatedly decomposing a problem into smaller problems.

- A program is constructed from smaller pieces (components, modules)
- Each piece is more manageable than the original program
Functions

- Programs combine *user-defined* functions with *library* functions
  - C standard library has a wide variety of functions, e.g. several math functions are included in `math.h`

- Invoking functions
  - Provide function name and arguments
  - Function performs operations or manipulations
  - Function returns results

- Function call analogy
  - Boss asks worker to complete task
  - Worker gets information, does task, returns result
  - Information hiding: boss does not know details
Math Library Functions

- Math library functions
  - perform common mathematical calculations
  - `#include <math.h>`

- Format for calling functions
  - `FunctionName( argument );`
    - If multiple arguments, use comma-separated list
  - `y = sqrt( 900.0 );`
    - Calls function `sqrt`, which returns the square root of its argument

- Arguments may be any r-value (constants, variables, or expressions)
# Math Library Functions

<table>
<thead>
<tr>
<th>Function Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int abs(int num)</td>
<td>Returns the absolute value of an integer element</td>
</tr>
<tr>
<td>double fabs(double num)</td>
<td>Returns the absolute value of a double precision element.</td>
</tr>
<tr>
<td>double pow(double x, double y)</td>
<td>Returns x raised to the power of y.</td>
</tr>
<tr>
<td>int rand(void)</td>
<td>Returns a random number</td>
</tr>
<tr>
<td>double sin(double angle)</td>
<td>Returns the sine of an angle; the angle should be in Radius.</td>
</tr>
<tr>
<td>double cos(double angle)</td>
<td>Returns the cosine of an angle; the angle should be in Radius.</td>
</tr>
<tr>
<td>double sqrt(double num)</td>
<td>Returns the square root of a double</td>
</tr>
</tbody>
</table>
Math Library Functions (Example)

■ How to code square root of \((x_1 - x_2)^2 + (y_1 - y_2)^2\) by using math.h library functions?

\[a = x_1 - x_2;\]
\[b = y_1 - y_2;\]
\[c = \text{pow}(a, 2) + \text{pow}(b, 2);\]
\[d = \text{sqrt}(c);\]
Functions

■ We have already written/called some functions before.

```
/* Welcome to BBM 101 */
#include <stdio.h>

int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

■ main is a function that must exist in every C program.
■ printf is a library function which we have already used in our program.

Let’s create and call our own functions now!
Function Definition

- Syntax

  ```
  type name (parameters)
  {
    variables;
    statements;
  }
  ```

- **name** is the name of the function
- **type** is the type of the returned value by the function
  - *void* means the function returns nothing
  - Functions return *int* value if nothing is specified
- **parameters** specify the types and names of the parameters separated by comma
Let’s define a function to compute the cube of a number:

```c
int cube ( int num ) {
    int result;
    result = num * num * num;
    return result;
}
```

This function can be called as:

```c
n = cube(5);
```
void Function (Example)

/* function definition */
void print_message(void) {
    printf("A message for you: ");
    printf("Have a nice day!\n");
}

int main(void) {
    /* function invocation */
    print_message();
    return 0;
}
Function Invocation

- A program is made up of one or more functions, one of them being `main()`. 

- When a program encounters a function, the function is called or invoked.

- After the function does its work, program control is passed back to the calling environment, where program execution continues.

```c
int main()
{
    Function1();
    ...
    return;
}

Function1()
{
    ...
    return;
}

Function2()
{
    ...
    return;
}
```
The return Statement

- When a return statement is executed, the execution of the function is terminated and the program control is immediately passed back to the calling environment.

- If an expression follows the keyword return, the value of the expression is returned to the calling environment as well.

- A return statement can be one of the following two forms:

  ```
  return;
  return expression;
  ```

  **Examples:**
  ```
  return;
  return 1.5;
  return result;
  return a+b*c;
  return x < y ? x : y;
  ```
The return Statement (Example)

- Define a function to check if asked year is a leap year

```c
int IsLeapYear(int year){
    return ( ((year % 4 == 0) && (year % 100 != 0))
            || (year % 400 == 0) );
}
```

- This function may be called as:

```c
if (IsLeapYear(2005))
    printf("29 days in February.\n");
else
    printf("28 days in February.\n");
```
Example: Find minimum of two integers

```c
#include <stdio.h>

int min(int a, int b){
    if (a < b)
        return a;
    else
        return b;
}

int main (void){
    int j, k, m;

    printf("Input two integers: ");
    scanf("%d %d", &j, &k);

    m = min(j, k);

    printf("\nThe minimum is %d.\n", m);
    return 0;
}
```

Input two integers: 11 3
The minimum is 3.
Function Parameters

- A function can have zero or more parameters.

- The *formal parameter list* in declaration header

  \[
  \text{int } f \ (\text{int } x, \ \text{double } y, \ \text{char } c); \]

  Parameter variables and their types are declared here.

- The *actual parameter list* in function calling:

  \[
  \text{value } = f(\text{age, } 100*\text{score}, \ \text{initial}); \]

  Cannot be told what their types are from here.
Rules for Parameter Lists

- The number of parameters in the actual and formal parameter lists must be *consistent*.

- Parameter association is *position*al: the first *actual* parameter matches the first *formal* parameter, the second matches the second, and so on.

- *Actual* parameters and *formal* parameters must be of compatible *data types*.

- *Actual* parameters may be a variable, constant, any expression matching the type of the corresponding formal parameter.
Call-by-Value Invocation

- Each argument is evaluated, and its value is used locally in place of the corresponding formal parameter.

- If a variable is passed to a function, the stored value of that variable in the calling environment will not be changed.

- In C, all calls are call-by-value.
Function Call

- The type of a function-call expression is the same as the type function being called, and its value is the value returned by the function.

- Function calls can be embedded in other function calls.

```plaintext
t = cubesum(i);
j = cubesum(t);
```

is equivalent to

```plaintext
j = cubesum(cubesum(i));
```
Function Call (Example)

```c
#include <stdio.h>
int compute_sum (int n)
{
    int sum = 0;
    for ( ; n > 0; --n)
        sum += n;
    printf("%d ", n);
    return sum;
}

int main (void)
{
    int n = 3, sum;
    printf("%d ", n);
    sum = compute_sum(n);
    printf("%d ", n);
    printf("%d", sum);
    return 0;
}
```

3 0 3 6
Example: Find maximum of three integers

#include <stdio.h>

int maximum(int a, int b, int c) {
  int max = a;

  if (b > max)
    max = b;
  if (c > max)
    max = c;

  return max;
}

int main (void) {
  int j, k, l, m;

  printf("Input three integers: ");
  scanf("%d %d %d", &j, &k, &l);

  printf("\nThe maximum is %d.\n", maximum(j, k, l));
  return 0;
}
Function Prototypes

- General form for a function prototype declaration:

  \[
  \text{return\_type \ function\_name (parameter\_type\_list)}
  \]

- Used to validate functions
  - Prototype only needed if function definition comes after use in program

- The function with the prototype

  \[
  \text{int maximum(int, int, int);} \\
  \]

  (Takes in 3 \text{ints}, returns an \text{int})
Using Function Prototypes

```c
#include <stdio.h>
int max (int a, int b) {
    ...
}

int min (int a, int b) {
    ...
}

int main(void) {
    ...
    min(x, y);
    max(u, v);
    ...
}

#include <stdio.h>
int max(int int);
int min(int, int);

int main(void) {
    min(x, y);
    max(u, v);
    ...
}

int max (int a, int b) {
    ...
}

int min (int a, int b) {
    ...
}
```
Today

- Functions
  - Definitions
  - Invocation
  - Parameter Lists
  - Return Values
  - Prototypes

- Recursion
  - Recursion
  - Inductive reasoning
  - Divide and conquer

- Variable Scopes
  - Block Structure
  - Global and Local Variables
  - Static Variables
Block Structure and Variable Scope

#include <stdio.h>

int total, count;

int main(int argc, const char * argv[]){
    total = count = 0;
    {
        int count = 0;
        while (1) {
            if (count > 10)
                break;
            total += count;
            count++;
        }
        printf("%d\n", count);
    }
    printf("%d\n", count);
    count++;
    printf("%d\n", count);
    return 0;
}

Global count variable is valid in the whole program.

Local count variable is only valid in the red block here.
External Variables

- Local variables can only be accessed in the function in which they are defined.

- If a variable is defined outside any function at the same level as function definitions, it is available to all the functions defined below in the same source file
  → external variable

- **Global variables** are external variables defined before any function definition
  - Their scope will be the whole program
# Local Variables

#include <stdio.h>

void func1 (void){
    int i = 5;
    printf("%d\n", i);
    i++;
    printf("%d\n", i);
}

int main (void){
    int i = 5;
    printf("%d \n", i);
    func1();
    printf("%d \n",i);
    return 0;
}
Static Variables

- A variable is said to be **static** if it is allocated storage at the beginning of the program execution and the storage remains allocated until the program execution terminates.

- External variables are always static.

- Within a block, a variable can be specified to be static by using the keyword static before its type declaration:
  
  
  ```
  static type variable-name;
  ```

- Variable declared static can be initialized only with constant expressions (**if not, its default value is zero**).
Static Variables (Example)

```c
#include <stdio.h>
void incr(void);

int main(void){
    int i;
    void incr(void);

    for (i=0; i<3; i++)
        incr();
    return 0;
}

void incr(void){
    static int static_i = 0;
    printf("static_i = %d\n", static_i++);
}
```

A static variable inside a function keeps its value between invocations.

Output:
```
static_i = 0
static_i = 1
static_i = 2
```
#include <stdio.h>

void put_stars(int n){
    static int static_n;
    int i;

    for (i=0; i<static_n; i++)
        printf(" ");
    for (i=0; i<n; i++)
        printf("*");

    printf("\n");
    static_n+= n;
}

int main(void){
    put_stars(3);
    put_stars(2);
    put_stars(3);
    return 0;
}
Today

- **Functions**
  - Definitions
  - Invocation
  - Parameter Lists
  - Return Values
  - Prototypes

- **Variable Scopes**
  - Block Structure
  - Global and Local Variables
  - Static Variables

- **Recursion**
  - Recursion
  - Inductive reasoning
  - Divide and conquer
Recursion

- **Recursion** is the process whereby a construct operates on itself.

- In C, a function may directly or indirectly call itself in the course of execution.
  
  - **direct**: The call to a function occurs inside the function itself
  - **indirect**: A function calls another function, which in turn makes a call to the first one

- Recursion is a programming technique that naturally implements the divide-and-conquer problem solving methodology.
Iterative Algorithms

- Looping constructs (e.g. while or for loops) lead naturally to **iterative** algorithms
- Can conceptualize as capturing computation in a set of "state variables" which update on each iteration through the loop
Iterative multiplication by successive additions

- Imagine we want to perform multiplication by successive additions:
  - To multiply a by b, add a to itself b times

- State variables:
  - i – iteration number; starts at b
  - result – current value of computation; starts at 0

- Update rules
  - i ← i - 1; stop when 0
  - result ← result + a

```c
int iterMul(int a, int b) {
    int result = 0;
    while (b > 0) {
        result += a;
        b--;
    }
    return result;
}
```
Recursive version

- An alternative is to think of this computation as:

\[ a \times b = a + a + \ldots + a \]

\[ \underbrace{a + a + \ldots + a}_{b \text{ copies}} = a + a \times (b - 1) \]
Recursion

- This is an instance of a **recursive** algorithm
  - Reduce a problem to a simpler (or smaller) version of the same problem, plus some simple computations

- **Recursive step**
  - Keep reducing until reach a simple case that can be solved directly

- **Base case**
  - a*b=a; if b=1 *(Basecase)*
  - a * b = a + a * (b-1); otherwise *(Recursive case)*

```java
int recurMul(int a, int b) {
  if (b==1)
    return a;
  else
    return a + recurMul(a, b-1);
}
```
Inductive reasoning

- How do we know that our recursive code will work?
- `iterMul` terminates because b is initially positive, and decrease by 1 each time around loop; thus must eventually become less than 1
- `recurMul` called with b = 1 has no recursive call and stops
- `recurMul` called with b > 1 makes a recursive call with a smaller version of b; must eventually reach call with b = 1
Mathematical Induction

To prove a statement indexed on integers is true for all values of $n$:

- Prove it is true when $n$ is smallest value (e.g. $n = 0$ or $n = 1$)
- Then prove that if it is true for an arbitrary value of $n$, one can show that it must be true for $n+1$
Example

0+1+2+3+...+n=(n(n+1))/2

Proof

- If n = 0, then LHS is 0 and RHS is 0*1/2 = 0, so true
- Assume true for some k, then need to show that
  - 0 + 1 + 2 + ... + k + (k+1) = ((k+1)(k+2))/2
  - LHS is k(k+1)/2 + (k+1) by assumption that property holds for problem of size k
  - This becomes, by algebra, ((k+1)(k+2))/2
- Hence expression holds for all n >= 0
What does this have to do with code?

- Same logic applies

```java
int recurMul(int a, int b)
{
    if (b==1)
        return a;
    else
        return a + recurMul(a,b-1);
}
```

- Base case, we can show that `recurMul` must return correct answer

- For recursive case, we can assume that `recurMul` correctly returns an answer for problems of size smaller than `b`, then by the addition step, it must also return a correct answer for problem of size `b`

- Thus by induction, code correctly returns answer

Slide credit: E. Grimson, J. Guttag and C. Terman
Factorial Function – *Iterative Definition*

\[ n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 \quad \text{for any integer } n > 0 \]

\[ 0! = 1 \]

**Iterative Definition in C:**

```c
fval = 1;
for (i = n; i >= 1; i--) {
    fval = fval * i;
}
```
Factorial Function – *Recursive Definition*

- To define $n!$ recursively, $n!$ must be defined in terms of the factorial of a smaller number.

- Observation (problem size is reduced):
  \[ n! = n \times (n-1)! \]

- Base case $\rightarrow 0! = 1$

- We can reach the base case, by subtracting 1 from $n$ if $n$ is a positive integer.

**Recursive Definition:**

\[
\begin{align*}
n! &= 1 & \text{if } n = 0 \\
n! &= n \times (n-1)! & \text{if } n > 0
\end{align*}
\]
Recursive Factorial Function Definition in C

/* Computes the factorial of a nonnegative integer.  
   Precondition: n must be greater than or equal to 0. 
   Postcondition: Returns the factorial of n; n is unchanged. */

int fact(int n)
{
    if (n == 0)
        return (1);
    else
        return (n * fact(n-1));
}

This fact function satisfies the four criteria of a recursive solution.
How does it Compute?

printf("%d", fact (3));
Tracing a Recursive Function

- A stack is used to keep track of function calls.

- Whenever a new function is called
  - For each function call, an **activation record (AR)** is created on the stack.
  - AR consists of the function’s parameters and local variables are pushed onto the stack along with the memory address of the calling statement (return point).

- To trace a recursive function, the **box method** can be used.
  - The box method is a systematic way to trace the actions of a recursive function.
  - The box method illustrates how compilers implement recursion.
  - Each box in the box method roughly corresponds to an activation record.
The Box Method

Label each recursive call in the body of the recursive function.

- These labels help us to keep track of the correct place to which we must return after a function call completes.
- After each recursive call, we return to the labeled location, and substitute that recursive call with returned value.

```java
if (n == 0)
    return 1;
else
    return n * fact(n-1);
```

A
The Box Method (cont’d.)

■ Every time a function is called, a new box is created to represent its local environment.

■ Each box contains:
  ▪ The values of the arguments
  ▪ The function’s local variables
  ▪ A placeholder for the value returned from each recursive call from the current box (label in the previous step).
  ▪ The value of the function itself.

n = 3
A: fact(n-1) = ?
return ?
The Box Method (cont’d.)

- Draw an arrow from the statement that initiates the recursive process to the first box.
  - Then draw an arrow to a new box created after a recursive call, put a label on that arrow.

```
printf("%d", fact (3));
```

```
\begin{center}
\begin{tikzpicture}
    \node[draw] (n3) at (0,0) {
        \begin{tabular}{l}
        n = 3 \\
        A: fact(n-1) = ? \\
        return ?
        \end{tabular}
    };
    \node[draw] (n2) at (2,0) {
        \begin{tabular}{l}
        n = 2 \\
        A: fact(n-1) = ? \\
        return ?
        \end{tabular}
    };
    \draw[->, thick] (n3) -- (n2) node [midway, above] {A};
\end{tikzpicture}
\end{center}
```
The Box Method (cont’d.)

- After a new box is created, we start to execute the body of the function.

- On exiting a function, cross off the current box and follow its arrow back to the box that called the function.
  - This box becomes the current box.
  - Substitute the value returned by the just-terminated function call into the appropriate item in the current box.
  - Continue the execution from the returned point.
Box Trace of $\text{fact(3)}$

- The initial call is made, and the function $\text{fact}$ begins execution.

- At point A, a recursive call is made, and the new invocation of $\text{fact}$ begins execution.

- At point A, a recursive call is made, and the new invocation of $\text{fact}$ begins execution.
Box Trace of $\text{fact}(3)$ (cont’d.)

- At point A, a recursive call is made, and the new invocation of fact begins execution.

- This is the base case, so this invocation of fact completes.

- The function value is returned to the calling box, which continues the execution.

- The current invocation of fact completes.
Box Trace of $\text{fact}(3)$ (cont’d.)

- The function value is returned to the calling box, which continues the execution.

- The current invocation of fact completes.

- The function value is returned to the calling box, which continues the execution.

- The current invocation of fact completes.

- The value 6 is returned to the initial call.
Example: Find the reverse of an input string

```c
/* reads n characters and prints them in reverse order. */
void reverse(int n) {
    char next;
    if (n == 1) {
        /* stopping case */
        scanf("%c", &next);
        printf("%c", next);
    } else {
        scanf("%c", &next);
        reverse(n-1);
        printf("%c", next);
    }
    return;
}

int main() {
    printf("Enter a string: ");
    reverse(3);
    printf("\n");
}
```
Trace of `reverse(3)`

```c
reverse(3);  /* Assume input is abc */

n=3
3 <= 1 ? false
read next : a
reverse(2)
write a
return

n=2
2 <= 1 ? false
read next : b
reverse(1)
write b
return

n=1
1 <= 1 ? true
read next : c
write c
return
```
Recursion with multiple base cases

- Fibonacci numbers
  - Leonardo of Pisa (aka Fibonacci) modeled the following challenge
    - Newborn pair of rabbits (one female, one male) are put in a pen
    - Rabbits mate at age of one month
    - Rabbits have a one month gestation period
    - Assume rabbits never die, that female always produces one new pair (one male, one female) every month from its second month on.
    - How many female rabbits are there at the end of one year?
Fibonacci Sequence

- After one month (call it 0) – 1 female
- After second month – still 1 female (now pregnant)
- After third month – two females, one pregnant, one not
- In general, females(n) = females(n-1) + females(n-2)
  - Every female alive at month n-2 will produce one female in month n;
  - These can be added those alive in month n-1 to get total alive in month n

<table>
<thead>
<tr>
<th>Month</th>
<th>Females</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
</tr>
</tbody>
</table>

Slide credit: E. Grimson, J. Guttag and C. Terman
Fibonacci Sequence

- **Base cases:**
  - Females(0) = 1
  - Females(1) = 1

- **Recursive case**
  - Females(n) = Females(n-1) + Females(n-2)

```c
/* assumes x an int >= 0 and returns Fibonacci of x */
int fib(int n){
    if (n < 2)
        return n;
    else
        return (fib(n-2) + fib(n-1));
}
```
Fibonacci Sequence

```c
/* assumes x an int >= 0 and returns Fibonacci of x */
int fib(int n){
    if (n < 2)
        return n;
    else
        return (fib(n-2) + fib(n-1));
}
```

- This is an example of non-linear recursion. Because total number of recursive calls grows exponentially.
- `fib(n-1)` expression must be evaluated completely before its value can be added to the expression `fib(n-2)` which must also be evaluated completely.
- Recursion tree is useful in tracing the values of variables during non-linear recursion.
Recursion Tree for Fibonacci Sequence

\[
fib(6) \rightarrow 7\text{th Fibonacci number}
\]

\[
fib(5) + fib(4)
\]

\[
fib(4) + fib(3)
\]

\[
fib(3) + fib(2)
\]

\[
fib(2) + fib(1)
\]

\[
fib(1) + fib(0)
\]

......
Example: Fibonacci Sequence (Iterative Version)

```c
int Fib(int n) {
    int Prev1, Prev2, Temp, j;

    if (n==0 || n== 1)
        return n;
    else {
        Prev1=0;
        Prev2 = 1;
        for (j=1; j <= n; j++){
            Temp = Prev1 + Prev2;
            Prev2 = Prev1;
            Prev1 = Temp;
        }
        return Prev1;
    }
}
```
Recursion vs. Iteration

- In general, an iterative version of a program will execute more efficiently in terms of time and space than a recursive version.
  - This is because the overhead involved in entering and exiting a function is avoided in iterative version.

- However, a recursive solution can be sometimes the most natural and logical way of solving a problem.
  - Conflict: machine efficiency versus programmer efficiency.

- It is always true that recursion can be replaced with iteration and a stack (and vice versa).
Summary

Functions
- Definitions
- Invocation
- Parameter Lists
- Return Values
- Prototypes

Variable Scopes
- Block Structure
- Global and Local Variables
- Static Variables

Recursion
- Recursion
- Inductive reasoning
- Divide and conquer
Next week

■ Debugging
  ▪ Testing and debugging
  ▪ Black box testing
  ▪ Glass box testing
  ▪ Integration testing and unit testing
  ▪ Debugging approaches

■ Arrays
  ▪ Declaring Arrays
  ▪ Examples
  ▪ Passing Arrays to Functions
  ▪ Sorting Arrays
  ▪ Multi-Dimensional Arrays
  ▪ Command Line Input