# Functions & Modularity

*Lecture Outline & Notes*

## Lecture Outline

1. Introduction
    - Developers do not work in a vacuum: never develop programs from scratch
    - We don't reinvent the wheel: many libraries, frameworks, etc. have been developed; we reuse them (code reuse)
    - First step to problem solving: determine if the problem is already solved!
    - Top-down design: break a problem down into smaller pieces until either its trivially solvable or a solution already exists; encapsulate (enclose) these pieces into their own functions
    - Libraries and standard code is well-tested, well-designed, optimized, thousands of man-hours and millions of CPU hours have run on them
    - Procedural abstraction: doesn't matter how the function executes/gets its answer, just that we can use it (relieves us of worrying about details)
    - Already familiar with: main as a function, standard library functions (math, I/O)
2. Functions
    a. Declaration
        - Just like variables, functions must be declared before they can be used (so that types: input/output can be determined)
        - A function *signature* consists of its interface:
            - Return type
            - Identifier (name)
            - Parameter list
                - Type
                - Name (local to the function)
                - Order matters
    b. Using
        - Variable scope
            - Variables can be declared inside functions
            - Scope of variables is *local* to the function and invalid outside the function
            - Parameters are also local
        - Calling a function
            - Providing *arguments*
            - Program stack & control flow (functions can call other functions)
                - When a function is called ("invoked"), control flow is given to the function

        - So that control flow can be returned, a new frame is created on the call stack (parameters copied to this frame)
        - Stack: a data structure where things are placed on the top (pushed) and only accessible from the top (popping)
        - When the function returns, control is restored to the calling function (and the frame on top of the stack is forgotten)
    - Using the results: functions can only ever return (at most) *one* value
    - Void functions
c. Call by reference/call by value
    - Call by value: *copies* of the value of the variables at the time of the function calls are created and given to the function
        - Any changes to the variable within the function are not changes to the original variable, only the local one
        - Variables/values are created on the call stack
        - Demonstration:

```
double computeEuclideanDistance(double x_1, double y_1,
double x_2, double y_2) {
 double temp = (x_1-x_2)*(x_1-x_2) + (y_1-y_2)*(y_1-y_2);
 double dist = sqrt(temp);
 return dist;
}
```

    - Call by reference: a *reference* (memory address)
        - The memory address of the variable is passed to the function
        - The function can now affect the original variable's value
        - Allows us to "return" multiple values
        - Allows us to pass around "large" items (rather than copying an entity, we just communicate the smaller memory address)
        - Demonstration: later
d. Modularity & Organization

3. Best Practices
    - Functions should be well-designed, well-organized, reusable
    - Rigorously test functions by writing *drivers* that call them on various inputs to test for correct output
4. Exercises
    - Modify a prior exercise to compute the roots of a quadratic equation by creating functions to handle the work; trace the function calls used.
    - Create a general purpose round function and use it in other "convenience" rounding functions
    - Write and use functions that use call-by-reference to swap two variable values, order two values, another to scale a value by a certain amount, etc.

# C Functions

1. Functions in C
    a. Prototyping
        - The compiler/code needs to know a function's signature in order to use it
        - Declaring a *prototype* allows us to use the function without specifying its internals (which can be done later)
        - Syntax: prototype needs a semicolon, must be done prior to using
        - Definition can be provided later (same signature, plus function body)
    b. Declaration syntax
        - Use of `void` keyword
        - Use of `const` keyword
    c. Modularity: best practice
        - Separating related code into different source files leads to better organization
        - Preprocessor directives, constants, and prototypes should be declared in a *header* file (.h file)
        - Corresponding function definitions should be placed in a *source* file (.c file of the same name)
        - Compiling can now be done piecemeal (compile, but do not link into an object file)
        - Items can now be included using the #include directive
        - Convention: use double quotes rather than <>
        - GCC:
            - `gcc -c foo.c` (compiles, but does not link foo.c into an object file, foo.o)
            - `gcc –lm foo.o mainFoo.c` (compiles and links with your object file)
    d. Pointers
        - Passing by reference is achieved through pointers
        - A pointer is a reference to a memory address containing a specific *type* (int, double)
        - Syntax: star
```
int *a;
double *b;
```
        - Keyword: `NULL` points to nothing, but practice: always assign null at declaration:
```
int *a = NULL;
double *b = NULL;
//checking for nullity:
if(a != NULL) { ...
```
        - Referencing operator: & gives the memory address of a regular variable
        - Dereferencing: star operator gives the *value* of a pointer variable (value stored in that memory location)
        - Usage: assigning values
```
int x = 10;
double y;
a = &x;
```

```
y = *b;
printf("value = %d", *a);
```

- Pass by reference functions:
  `void func(double *a, int b, int *c)`
- Returning a pointer:
  `int * funcB(int n);`
- Printing memory addresses: %x, %X (hex), %p:
  `printf("The value of a is %d and is located at memory address %p\n", a, &a);`
- Demonstration

e. Pitfalls:

- `scanf` revisited: why we use ampersand
- You can treat integers as arbitrary memory addresses, but it leads to: undefined behavior, bus errors, seg faults (illegal memory access):

```
int *a;
a = 10; //make a point to memory address "10"
```

-

## Java Methods

1. Syntax & usage
2. Key Differences
   - Usually referred to as "methods"
   - No "global" methods: all methods belong to some class
   - Use of the `static` keyword
     - Methods belong to the class (not to instances of the class)
     - Common practice: place related static methods in a "utilities" class
   - Non static methods are member methods (need to instantiate instances in order to use them—more later)
   - Use of the `final` keyword
     - No `const` in Java
     - In a method parameter: constant (not able to be changed)
     - With a variable: the variable is a constant
   - No pointers
     - Everything is a reference (except primitive types)
     - However: most built-in types are *immutable*: once created, cannot be changed
     - Consequence: no "swap" –like methods, no "multiple return values"
3. Best Practices

a. Java: methods should be included in the classes that own the data they act on; use of static methods should be minimized (only used when it does not rely on any of the class's state)
b. Collect similar methods into "utility" classes (java.lang.Math, java.util.Arrays)

## Exercises

1. In C: redesign the quadratic roots program to:
    a. Perform the calculations in two functions (with proper prototypes and definitions)
    b. Separate the functions and prototypes into their own source and header files and compile them
    c. Consolidate the two functions into one function that uses call by reference to "return" two values
    d. Change the return type of the function to return an *error code* indicating various invalid inputs
2. Revisit older programs that used some rounding functionality and create several functions to provide it instead; separate their functionality into source/header files