1. Lecture 3 – C++ Functions, Pointers, and References
   1. Functions Parts
      1. Function Declaration (Protoype)
         1. Information for compiler to properly interpret calls.
      2. Function Definition (Implementation)
         1. Actual Implementation (i.e., code) for function.
      3. Function Call
         1. How function is actually used by program.
         2. Transfers execution control to the function.
   2. **Function Declaration** (Prototype)
      1. Gives compiler information about the function
         1. ⮚ How to interpret calls to the function.  
            1. **<return type> <function\_name> (<parameter\_types>);**
            2. **double sprintSum (int, double, char []); double sprintSum (int n, double d, char s[]);**
         2. Must list the parameters’ data types (at least).
         3. Placed before any calls  
            In declaration space of **main()**.  
            Or above **main()** for global access.
   3. **Function Definition** (Implementation)
      1. Definition of the function:
         1. **double printSum (int n, double d, char s[]) double sum = d + n;  
            sprintf(s, "%.3f", sum);  
            return sum;**
      2. ⮚ Function definition must match prototype.
         1. Placed AFTER the function **main()** (NOT inside).
      3. ⮚ All definitions of equal order, no function *needs* to be contained inside another.
      4. ⮚ Function name, parameter(s) type, and return type all must match the prototype’s. ⮚ **return** statement sends data back to the caller.
   4. **Function Call** 
      1. Much like a standard C call:
         1. **char printSum[256];  
            double returnSum = sprintSum(10, 0.1, printSum);**
         2. ⮚ Returns a **double**.  
            (Assigned to variable **returnSum**)
         3. Arguments:
            1. ⮚ The literals **10, 0.01**.   
               (Can also pass variables – do they have to be **int** and **double**?)
            2. ⮚ A **char** array variable **printSum**(Has to match **char []** type – formally **char \***.)
   5. Function Parts
      1. Function Prototype
      2. Function Definition
      3. Function Call
   6. **return Statement(s)**
      1. Transfers control back to the calling function.
      2. Special case: “**void**” functions:
         1. No value back, Functions that only have side effects (e.g., print out information).
         2. Similar declaration to “regular” functions
         3. **void printResults(double cost, double tax);**
         4. Optional **return** statement (all other return types must have a return statement).
      3. Typically the last statement in the definition.
      4. Can also have multiple **return** statements.
         1. ⮚ Transfers control *early*, (anything past it in the function Block is not executed).
         2. ⮚ Can have multiple exit points in a function.
      5. Typical use: *guard statements* ( **if(somethingWrong) return;** ).
   7. **Function Parameters / Arguments** 
      1. (Function) Parameter:
         1. ⮚ Formal variable, as it appears in the function prototype. ⮚ Part of the *Function Signature* (more on that later).
         2. (Function) Argument:
            1. ⮚ Actual value or variable.
            2. ⮚ An expression used when making the function call.
         3. Multiple Parameters / Arguments:
            1. **double precisionSum(double a, double b);   
               cout << precisionSum(0.1 \* 1000000, 1e-3)**
   8. **Function Pre / Post - Conditions** 
      1. Include function headers in your code.
         1. ⮚ Contain name, pre / post – conditions:
            1. Conditions include assumptions about program state, not just the input and output.

**// Function name: showInterest  
// Pre-condition: balance is nonnegative account // balance; rate is interest rate as percentage // Post-condition: amount of interest on given  
// balance, at given rate   
void showInterest(double balance, double rate**

* 1. C++ Function Libraries
     1. Full of useful functions!  
        Must “**#include**” appropriate library.
        1. ⮚ Correspondence to “**C**” libraries: **<cmath> ~ <math.h> <cstdlib> ~ <stdlib.h> <cstring> ~ <string.h>**
        2. ⮚ Console-File I/O:  
           (e.g. **std::cout**, **std::cin**) **<iostream>**
        3. ⮚ Many more...
  2. **The main() Function** 
     1. “Special” function, serves as entry point to the program. Only one **main()** can exist in a program.
        1. Called by the Operating System, not by the programmer!
        2. Should **return** an integer (**0** is traditional, Clean-termination/No-error return code).
     2. **Function Functionalities** 
        1. ⮚ Build “blocks” of programs
        2. ⮚Increases readability and reusability ⮚Divide and conquer large problems
        3. ⮚Separate source files from main() for easy sharing.
     3. **Note:** Functions in **C++** can only **return** one thing!
  3. Functions and Parameters
     1. Methods of passing arguments to functions:
        1. ⮚ Pass-by-Value:
           1. A “*Copy*” of the value of the actual argument is used.
        2. ⮚ Pass-by-Reference:
           1. The “*Actual”* argument itself is used.
        3. ⮚ Pass-by-Address:
           1. A “*Copy*” of the value of the argument is used ...

(*but:)* the argument is a special type that allows to in-directly use another variable.

1. Pointers
   1. Pointer
      1. *Variable* whose Value holds the *Address-Of* something somewhere in memory.
   2. Pointer Utility
      1. Pointers are incredibly useful in programming.
         1. ⮚ Allow functions to:  
            Modify multiple arguments.  
            Use and modify arrays as arguments.
         2. ⮚ Increase program (compiled function) efficiency.  
            ⮚ Creation / handling / use of Dynamic Objects (more on that later).
   3. Pointer Declaration
      1. A pointer is just like any regular variable. It has: ⮚ Type
         1. ⮚ Name
         2. ⮚ Value (what kind?)
      2. Pointer declaration /creation requires the (**\***) symbol.
   4. Pointer Value
      1. As earlier stated, pointers are “Just Variables”.
         1. ⮚ Pointer’s Value: an *Address* in memory (instead of storing an **int**/**float**/**char**/etc.) Note: Pointer’s size in memory is not guaranteed (implementation-defined).
   5. Pointer Assignment
      1. Value (pointed-to Address) assignment:
         1. ⮚ To get the *Address-Of* a variable we use the ampersand (**&**) operator.
            1. **int x=5;  
               int \* xPtr = NULL;**
         2. ⮚ Pointer-to-pointer assignment (also valid):
            1. **int \* yPtr;  
               yPtr = xPtr;**
   6. **Indirection** (Dereference) **Operator** ( **\*** ) or “Value-Pointed-By”
      1. To refer to the *Value-Pointed-By* a pointer, we pre-pend the star (**\***) operator to its name.
         1. **... = \*ptr   
            \*ptr = ...;**
      2. At this point what follows depends on purpose of Dereferencing.
      3. A Dereference can be in three “places”:
         1. ⮚ On the *left hand* side of the assignment operator.
         2. ⮚ On the *right hand* side of the assignment operator.
         3. ⮚ In an expression with *no assignment* operator (e.g. a **cout** statement).
   7. Pointers as Function Parameters
      1. Common Paradigm:
         1. A function that modifies more than one values.
            1. Example: How to multiply Two int values by an order of magnitude.

**void increaseOrder( <two ints> ) {  
// multiply first int by 10  
// multiply second int by 10  
// have the values persist after control is return’ed -- how?**   
**}**

* + - 1. ⮚ Can’t use Pass-by-Value, then return & assign method. **return** will only give back One value.
      2. ⮚ Can use Pass-by-Reference (working directly on passed arguments).
      3. ⮚ But also ...
  1. **Reference-Types** 
     1. Reference-Type variable declaration with the ampersand (**&**) symbol.
        1. **int x = 10;   
           int & xRef = x;**
     2. Once created, they don’t need the ampersand (**&**) or asterisk (**\***) in their use.
        1. `⮚ They are actually “*Aliases*” to pre-existing variables. (They look like normal variables)
     3. Rules:
        1. ⮚ References be initialized at declaration (they have to *Alias* something).
           1. Once initialized, they are forever tied to the thing they reference.
        2. ⮚ References cannot be changed (any attempt to assign just references the aliased variable).
        3. ⮚ References are another “name” for a variable (dereferencing does not make sense).