C++ Lecture 5

* Using Functions, Defining Functions, Variables in Functions, Overloading
* CIS 251 • Shelby-Hoover Campus

Top-Down Design

* All of the tasks required in a program are usually crafted into a single algorithm
* Once completed, this algorithm can be broken into smaller units of logic that can be coded separately
  + Smallest subtasks are easy to code
  + Divide responsibility for units among programmers, teams
  + Well-written code units can be reused in other programs
  + The process is sometimes called **stepwise refinement** or **divide and conquer**
* C++ subtasks are known as **functions**

Calling Functions

* A program may use functions that are predefined in  
  C++ or written by another programmer
* This may require extra #include statements
* The statement used to begin the execution of another function is known as a **function call** or **invocation**
* Function calls may take various forms:  
    
   functionName();  
    
   functionName(argument1, argument2);  
    
   variable = functionName(argument);  
    
   cout << functionName(a1, a2, a3) << endl;

Arguments

* An **argument** is a value required by the function being called
  + Listed in the function call in the parentheses after the function name
  + Multiple arguments are separated by commas
* Arguments are said to be **passed** to the function
* An argument may be a value, a variable, or an expression as long as it matches the required type (or equivalent) for that argument
* Not every function requires an argument (the parentheses after the function name may be empty)

Return Values

* Most (but not all) functions perform a calculation that results in a new value
* This value is **returned** back to the calling function (the function that contains the call statement)
* The function call must be placed inside another statement to make use of the returned value:  
  + Output: cout << myFunction() << endl;
  + Assignment: myVariable = myFunction(arg);
  + Calculation: myVar = myFun(arg) / 3.0;
  + Another Function: myFun1(myFun2(argA, argB));

Math Functions

* The following functions are provided by the <cmath> library and require that #include statement  
  + theRoot = sqrt(9.0);  
    /\* returns a double containing the square root of the double argument \*/
  + thePower = pow(myBase, 3);   
    /\* returns a double containing the first double argument raised to the power of the second double argument \*/
  + cout << fabs(userValue) << endl;   
    /\* returns a double containing the absolute value (positive form) of the double argument \*/
  + roundUp = ceil(startVal);  
    /\* returns a double containing the **ceiling** (rounded-up value) of the double argument \*/
  + roundDown = floor(startVal);  
    /\* returns a double containing the **floor** (rounded-down value) of the double argument \*/

More Math Functions

* There are two other forms of absolute value from the library <cstdlib>:  
  + absoluteInt = abs(myInt);   
    /\* returns an int containing the absolute value of the int argument \*/
  + absoluteLong = labs(myLong);   
    /\* returns a long containing the absolute value of the long argument \*/
* The argument(s) to any of these functions can be variables, literal values, or expressions
* A program may have as many #include statements at the top as needed
* Don’t forget the standard namespace (it should be listed only once)

Math Function Example

* Calculating the hypotenuse of a right triangle from the lengths of its legs:  
    
  #include <iostream>  
  #include <cmath>  
  using namespace std;  
    
  int main()  
  {  
   double leg1, leg2, hypotenuse;  
   cout << "Enter the length of the first leg: ";  
   cin >> leg1;  
   cout << "Enter the length of the second leg: ";  
   cin >> leg2;  
   hypotenuse = sqrt(pow(leg1, 2.0) + pow(leg2, 2.0));  
   cout.setf(ios::fixed);  
   cout.setf(ios::showpoint);  
   cout.precision(2);  
   cout << "The length of the hypotenuse is " << hypotenuse << endl;  
   return 0;  
  }

Random Numbers

* A program often requires the generation of random numbers (rolling dice, selecting a card, choosing which player goes first, simulations, etc.)
* C++ generates **pseudorandom numbers** that follow a sequence starting with an initial number (the formula that determines the series varies by implementation)
* The function rand is provided in the library <cstdlib> and generates a pseudorandom number between 0 and a constant named RAND\_MAX also defined in <cstdlib>

Using rand

* To obtain a pseudorandom number in a specific range, use arithmetic operators such as addition and modulus  
  + number = rand() % 100; /\* generates a pseudorandom number between 0 and 99 \*/
  + number = rand() % 100 + 1; /\*generates a pseudorandom number between 1 and 100 \*/
* To ensure that the pseudorandom number series starts with a different value each time, provide a different seed (initial value) using srand (also in <cstdlib>) before calling rand for the first time
  + The library <ctime> contains a function named time that, when called with the argument 0, returns the number of seconds that have passed since January 1, 1970
  + This function call can be the argument to srand:  
      
    srand(time(0));

Type Casting

* Chapter 2 mentions that division involving two integers results in truncation (the loss of the decimal component)
* A program can treat a variable or value as if it were another type (a process known as **type casting**) using the static\_cast<newType> function:  
    
   ans = static\_cast<double>(dividend) / divisor;  
  + The new type should be listed in the angle brackets
  + Be careful to cast an operand, not the entire expression, in division; if the cast is performed after the division, the decimal component will still be lost
* The older notation for a cast places the new type in parentheses (without the static\_cast keyword) before the expression to be cast, but this may not be supported by future C++ compilers

Defining Functions

* There are several concerns to be addressed when including your own functions in a program
* The compiler must know the general form of the function before it can be called
  + Some programmers place the entire definitions of other functions before they are called (main is listed last)
  + Many programmers prefer to define their other functions after main, so they must use other means to note the form of each function above the definition of the main function

Function Prototype

* A **function prototype**, also known as a **function declaration**, is a single line that indicates the name, return type, and argument types required by a programmer-defined function
  + The first element is the return type: what type of value will be returned by this function?
  + The second is the function name, which must follow C++ rules for identifiers
  + The third is the parameter list in parentheses
    - A **parameter** is a local name for an argument being used in a function
    - In the prototype, each parameter has a type and (optionally) a name, with commas between them
    - If the function does not require arguments, use empty parentheses
  + Each function prototype ends with a semicolon
  + Function prototypes are usually listed above / before main

The main Function Defined

* The main function demonstrates the form other functions will take
* Return type: int
* Function name: main
* Parameter list: usually empty (there is another form that accepts arguments from the command prompt)
* Function body in curly brackets
* Return statement in the body: return 0; /\* matches the return type int \*/

Function Definitions

* A **function definition** indicates what statements will be executed when a function is called
* The **function header** is similar to the prototype
  + Begins with the function’s return type and name
  + The parameter list requires both types and names
  + **No semicolon** at the end
* The **function body** is the set of statements to be executed when the function is called
  + Enclosed within curly brackets after the header
  + The function must contain a **return statement** for every flow of control within the function; the value (literal, variable, or expression) in the return statement must match the return type

Function Example

* Calculating the cost of an order based on quantity, unit price, and sales tax
* Prototype (choose one of the following; either version will work):  
    
  double totalCost(int quantity, double unitPrice);  
  double totalCost(int, double);
* Definition:  
    
  double totalCost(int quantity, double unitPrice)  
  {  
   const double TAX\_RATE = 0.05;  
   double subtotal;  
    
   subtotal = quantity \* unitPrice;  
   return (subtotal + subtotal \* TAX\_RATE);  
  }

Function Call Example

* The function call should be placed in the body of another function at the point where its code should be executed:  
    
  #include <iostream>  
  using namespace std;  
    
  double totalCost(int quantity, double unitPrice); // function prototype  
    
  int main()  
  {  
   double priceEach, bill;  
   int units;  
    
   cout << "Enter the number of units purchased: ";  
   cin >> units;  
   cout << "Enter the price per unit: ";  
   cin >> priceEach;  
    
   bill = totalCost(units, priceEach); // function call  
    
   cout.setf(ios::fixed);  
   cout.setf(ios::showpoint);  
   cout.precision(2);  
    
   cout << "Final bill with tax is $" << bill << endl;  
    
   return 0;  
  }  
    
  double totalCost(int quantity, double unitPrice)  
  {  
   const double TAX\_RATE = 0.05;  
   double subtotal;  
    
   subtotal = quantity \* unitPrice;  
   return (subtotal + subtotal \* TAX\_RATE);  
  }

Function Call Details

* In a function call, the argument list must match the parameter list in terms of the number, type, and order, but the names do not have to match (some are literal values without names)
* Inside the function, use parameter names, not argument names
* Argument values are copied to the parameter (**call by value** or **pass by value**), so the original argument values are not changed even if the parameter changes
* A function can return a value of any type, including bool (a call to the function replaces a more complex Boolean expression)
* The compiler will not check to see if the arguments are in the correct logical order, so care must be taken to place them in the right order in the call

Functions Applied

* Code broken down into functions is easier to read
  + The **black box**: a call to a function doesn’t reveal the details of what’s going on behind the scenes
  + Also known as **information hiding** or **procedural abstraction**
  + Different implementations of a function may arrive at the same result
  + If sharing a function with other programmers, make sure that the prototype reveals enough details that the other programmers don’t need to see the definition
* A nested loop may be brought out into its own function (Display 4.9)
* The process of separating a program’s code into functions begins when planning the logic, not writing the code itself

Function Variables

* Each function has its own variable space
  + A variable declared inside a function is **local** to, or **in scope** within, that function
  + This applies to parameter variables as well; these are local variables that do not require separate declaration
* Variables and named constants may be declared **globally** (outside the functions, usually before the header of the first function) to make them accessible to multiple functions
  + Appropriate for the use of a constant in several functions
  + Using global variables may make programming more difficult, as functions become dependent on information external to the definition, and it may be difficult to isolate errors
* using directives may also be placed locally to limit the use of identifiers to a specific function, but this makes more sense if different functions require separate namespaces

Overloading Functions

* A function name may apply to several different sets of data in different ways; you may want to use a single identifier for several different function bodies
* You can **overload** a function by writing multiple definitions with the same function name but different parameter lists
  + The parameter lists must differ by type, number, or order; changing the names alone will not compile
  + The return types may be the same or different
* A function call will be matched to the closest definition (if there’s not an exact match between argument and parameter types)

Overloading Example

* Determining the unit price of a round pizza versus a square pizza
* Prototypes:  
    
  double unitPrice(int diameter, double price);  
  double unitPrice(int length, int width, double price);
* Definitions:  
    
  double unitPrice(int diameter, double price)  
  {  
   const double PI = 3.14159;  
   double radius, area;  
   radius = static\_cast<double>(diameter) / 2;  
   area = PI \* radius \* radius;  
   return (price / area);  
  }  
    
  double unitPrice(int length, int width, double price)  
  {  
   double area = length \* width;  
   return (price / area);  
  }

Overloading and Calls

* Because this function call has three arguments, it will cause the second definition to be called:  
    
  unitPriceRectangular = unitPrice(length, width, priceRectangular);
* Because this function call has two arguments, it will cause the first definition to be called:  
    
  unitPriceRound = unitPrice(diameter, priceRound);
* Don’t overload a function name if the behaviors are different (e.g., using mpg for miles per gallon and measure of perfect goals)