CSI 201 - Introduction to Computer Science

Chapter 3
Procedural Abstraction and Functions That Return a Value

Brian R. King
Instructor

Introduction

- Several software development methodologies are used today. A couple common methods:
  - **Waterfall model**
    - Oldest and best-known model.
    - Steps follow the software life cycle
  - **Agile development**
    - Focuses on iterative development.
    - Uses smaller steps, and requires significant feedback between each step.
      - Enforces better and more frequent channels of communication in the development team.
    - **Extreme Programming** – current most popular agile process

The design step

- Most consider **design** to be the most important step in any methodology.
  - Recall, the **design** step leads to **algorithms**.
  - But, most real-world problems are too complex for just one algorithm to solve. So, a design must be broken up.
- Several methods exist in the development of a good design:
  - **Bottom-up design**
    - Individual parts of the system are specified in detail.
    - These parts are then combined to form a larger view of the system
  - **Top-down design**
    - A general solution is established, and then the solution is broken down into smaller, more manageable tasks.
    - Some programming languages call these subtasks subroutines, subprograms, procedures, or **functions**.
- We will focus on **top-down design** of solutions through breaking up our program into manageable subtasks which C++ calls **functions**.

Benefits of Top-Down Design

- Subtasks, or **functions** in C++, make programs
  - Easier to understand
  - Easier to implement well-defined subtasks
  - Easier to change
  - Easier to write the main program
    - Each sub-task is treated like a **black-box**:
      - You know the required inputs to the function
      - You know what to expect for the output of the function
      - You don't need to worry about **implementation** details of the subtask.
  - Easier to test
    - Well designed code provides mechanism to test at the function level.
  - Easier to debug
  - Easier for teams to develop
    - You can give programmers well-defined subtasks, allowing the lead programmer to place the completed sub-tasks together into one main program.
    - (In an idealistic world, this actually works!)

Predefined Functions

- C++ comes with numerous **libraries** of predefined functions at your disposal.
- Example: The `sqrt` function.
  - Purpose: Calculates the square root of a number.
  - If we were to use the function to calculate the square root of 9, our C++ code would look as follows:
    - ```
      double root;
      root = sqrt(9.0);
      ```
  - `sqrt` is the **function.**
  - `9.0` is the **actual parameter** used for the function (also called an **argument.)**
  - `root` is the **value returned** from the function.
  - `sqrt(9.0)` forms a new type of *expression,* called a **function call** or **function invocation**.
    - The `sqrt` function is called and the result returned.

$$\sqrt{9} = 3$$

Function Call Syntax

- *`Function_name`*`(`*`Argument_List`*`)`
  - *Argument_List* is a comma separated list of actual parameters passed into the function:

    `(`*`Argument_1, Argument_2, … , Argument_Last`*`)`
- The parameters in a function call are referred to as the **actual parameters** because they give the actual initial values that a function will use (to be contrasted with **formal parameters** shortly.)
- Example:
  - ```
    side = sqrt(area);
    ```
  - ```
    cout << "2.5 to the power 3.0 is "
         << pow(2.5, 3.0);
    ```


`#include` and header files

- Just like variables, functions also need to be **declared** before being used, intuitively called a **function declaration**.
- **Header files** contain information about the library of predefined functions you wish to use, including function declarations.
- **Include directives** include the header file for the library you want to use in your program.
  - You usually need the `using namespace std;` statement to include the proper standard library function declarations.
    - Otherwise, you would need to append the special scope resolution tag `std::` to the front of every name declared in the `std` namespace. (Covered in Chapter 9.)
  - Examples:
    - `#include <iostream>`
    - `#include <cmath>`

Some predefined functions

Some Predefined Functions

Name	Description	Type of Arguments	Type of Value Returned	Example	Value	Library Header
sqrt	square root	*double*	*double*	sqrt(4.0)	2.0	cmath
pow	powers	*double*	*double*	pow(2.0,3.0)	8.0	cmath
abs	absolute value for *int*	*int*	*int*	abs(-7) abs(7)	7 7	cstdlib
labs	absolute value for *long*	*long*	*long*	labs(-70000) labs(70000)	70000 70000	cstdlib
fabs	absolute value for *double*	*double*	*double*	fabs(-7.5) fabs(7.5)	7.5 7.5	cmath
ceil	ceiling (round up)	*double*	*double*	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	floor (round down)	*double*	*double*	floor(3.2) floor(3.9)	3.0 3.0	cmath
    ```

Exercises

- Convert the following mathematical expressions to C++:

$$\sqrt{x+y} \qquad \frac{x}{\sqrt{1-x^2}} \qquad \frac{-b+\sqrt{b^2-4ac}}{2a}$$

$$\left|a+b\right| \qquad \left\lfloor 0.2a \right\rfloor$$

ex1.cpp – Calculate the hypotenuse

- Remember the Pythagorean theorem:
  - If ABC is a right triangle, then the square of the length of the hypotenuse equals the sum of the squares of the two legs.
- Let A and B be the legs, and let C be the hypotenuse – the side opposite the right angle.
- What is the equation for the hypotenuse C?

- Write a program that asks the user for the length of the two legs, and return the length of the hypotenuse.

ex1.cpp

```
// ex1.cpp
// Written by Brian King
//
// Purpose: Demonstrate using predefined functions.
//
// Computes the size of the hypotenuse given two sides.
// This program does no checking for valid length of sides.
//
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
  double side1, side2, hypotenuse;

  cout << "Enter the length of the legs of the right triangle\n";
  cin >> side1 >> side2;

  // FILL IN -- Calculate the length of the hypotenuse
  hypotenuse =

  cout << "The hypotenuse of a right triangle with sides "
       << side1 << " and " << side2
       << " is " << hypotenuse << endl;

  return 0;
}
```


Type casting

- Type casting: The process of changing the type of an expression into another type.
- There are two kinds of type casting
  - Explicit – programmer forced type cast
  - Implicit – compiler generated type cast
    - We've already seen implicit type casting with mixing types in assignment statements.
- How does this apply to functions? All formal parameters in a function are defined to take specific types.
  - `sqrt` has one parameter of type `double`.
  - Does the actual parameter used in this function call need to be an expression of type `double`?
- **If no information is lost when passing one type as a parameter that is defined to take another type, the compiler will generate the code to perform an implicit type cast without an error or warning message.**
  - Generally, the rules for mixing types in parameters follow the same rules as mixing types in assignment statements covered in the last chapter.

Automatic type-casting example

- If you have an actual parameter of type `int` you want to use in a function, but the formal parameter of the function is defined as type `double`, the compiler will automatically type cast it for you.
  - Example:

```
int i = 36;
double d = sqrt(i);
```

  - There is no warning generated. Why?

- The compiler **will generate a warning** when going the other way -- passing a `double` to a function that takes an `int`):

```
double d = -4.5;
int i = abs(d);
```

  - Why is a warning generated?

Explicit type casting

- C++ provides the means of explicitly type-casting an expression.
  - We can force the type of an expression into another type.
- Syntax:
  - `static_cast<`*`new_type`*`>(`*`Expression_of_old_type`*`)`
  - Example:

```
int i = 3,j = 2;
double d;
d = static_cast<double>(i)/j;
```

- If we did not use a type cast on variable `i`, then what would the value of variable d be?

```
d = i/j;
```

- You may occasionally see older, outdated forms of type casting:
  - `d = double(i)/j;`
  - `d = (double)i;`
- **Use the newer version.**

Programmer-defined functions

- What if you can't find a function to do exactly what you want? Write your own!
- There are two parts to writing your own function:
  - **Function declaration** (or function **prototype**)
    - Shows how the function is called
    - Must appear in the code before the function can be called
    - Syntax:
      `Type_returned  Function_Name(Parameter_list);`
  - **Function definition**
    - Describes how the function does its task
    - Can appear before or after the function is called
    - Syntax:

```
Type_returned  Function_Name(Parameter_list)
{
    // function body
}
```


Function Declaration

`Type_returned Function_Name(Type1 Par_Name1,…);`

- Tells the **name** of the function
- Tells the **return type** of the function
- Tells **how many parameters** are needed
- Tells the **formal parameter types**
- Tells the **formal parameter names**
  - Formal parameters are like placeholders for the actual parameters used when the function is called
  - Formal parameter names can be any valid identifier
  - NOTE: Formal parameter names are not required in the function declaration, but are helpful in remembering how the function is called.
- Example:

```
double calc_hypotenuse(double sideA, double sideB);
```

Function declaration comments

- You should have a comment under your function declaration describing the purpose of the function.
- It contains statements called **preconditions** and **postconditions**.
- Definition of a precondition:

- Definition of a postcondition:

- Example function declaration with comments:

```
double calc_hypotenuse(double sideA, double sideB);
// Precondition: sideA and sideB are lengths of the
//               legs of a valid right triangle (> 0)
// Postcondition: Return the length of the hyoptenuse.
```

- Common mistake with function declarations is to forget the semi-colon.

Function Definition

```
Type_returned Function_Name(Type1 Par_Name1, Type2 Par_Name2, …)  ← Function Header
{
    // Code to make the function work  ← Function Body
}
```

- A function definition is made up of two distinguishing parts:
  - **Function Header**
  - **Function Body**
- Function Header
  - Consists of a **return type**, a **function name**, and a **formal parameter list**. Each parameter requires a type and an identifier.
  - The return type, the function name, and the type of each parameter MUST match the function declaration.
    - The type AND identifier of each parameter are *required* in the function header. The function declaration only requires the type of each parameter.
    - The parameter identifier names can be different between the function declaration and function header, but this is not recommended, as it tends to lead to confusion. Be consistent!
  - Unlike the function declaration, the function header does NOT have a semi-colon.

Formal vs. Actual Parameters

- **Formal parameters** refer to the parameters declared in the header of the function definition.
- **Actual parameters** (or **arguments**) refer to the values given to a function during its invocation that initialize the formal parameters of the function.
- **REMEMBER: Formal parameters in the function definition are initialized with the actual parameters in the function call.**

Function Body

- The function body is always delimited by braces.
- In between the braces is a series of C++ statements that accomplish a subtask.
  - Remember the *block statement* definition? A function body is a block statement.
- Recall that a function call (or invocation) is an expression, and has a value. How does a function definition determine the value for the function call?
  - The last C++ statement that is executed in a function is the `return` statement. It returns a value for the function call.
  - Syntax:
    - `return (`*expr*`);`
  - It returns the value of the expression *expr* as the value of the function call.
- Let's convert the hypotenuse program to use a programmer-defined function

ex2.cpp

```
#include <iostream>
#include <cmath>
using namespace std;

double calc_hypotenuse(double sideA, double sideB);
// Precondition: sideA and sideB are valid lengths of the legs
//               on a right triangle
// Postcondition: returns the length of the hypotenuse

int main()
{
  double side1, side2, hypotenuse;

  cout << "Enter the length of the legs of the right triangle\n";
  cin >> side1 >> side2;

  hypotenuse = calc_hypotenuse(side1,side2);

  cout << "The hypotenuse of a right triangle with sides "
       << side1 << " and " << side2
       << " is " << hypotenuse << endl;

  return 0;
}
```

continued on next slide…

ex2.cpp continued

```
//////////////////////////////////////////////////////////
// calc_hypotenuse
//
// Calculate the hypotenuse of a triangle with legs of length
// sideA and sideB
//
double calc_hypotenuse(double sideA, double sideB)
{
  double hypot;

  hypot = sqrt( pow(sideA, 2) + pow(sideB, 2));

  return hypot;
}
```


Anatomy of a function call

- The *values* of the parameters in the *function call* are plugged into the formal parameters of the *function definition*.
  - This substitution process is known as **call-by-value**.
    - Remember the distinction between a **variable** and its **value**. Recall, a *variable* references a place in memory. The *value* of a variable is the value in that memory location, interpreted through its type.
    - When a function call has variables in the parameter list, the function definition uses the **values** of the variables in the function call. It does NOT use the actual variables themselves.

Analysis of ex2.cpp

1. The following statement begins executing:
   `hypotenuse = calc_hypotenuse(side1,side2);`
2. The value of `side1` is plugged in for parameter `sideA`, and the value of `side2` is plugged in for parameter `sideB`:

   ```
   double calc_hypotenuse(double sideA, double sideB)
   {
     double hypot;
     hypot = sqrt( pow(sideA, 2) + pow(sideB, 2));
     return hypot;
   }
   ```
3. The body of the function is executed.
   1. A variable is created called `hypot`.
   2. `hypot` is assigned the value of the result of `sqrt( pow(sideA, 2) + pow(sideB, 2));`
4. The `return` statement returns the value of `hypot` to the calling function. The variable `hypot` disappears when the function returns, and the variable `hypotenuse` in the main function is assigned this value.

Local Variables

- Variables declared within a function body are called **local variables.**
- Local variables are only visible within the function where they are declared.
  - Variables declared within the main function are local to `main`. *This means they are not visible to other functions in your program!*
- Variables can also be declared within a **block statement**
  - A **block statement** is a grouping of one or more C++ statements, delimited by braces:

```
if (x > 5)
{
    int total;
    …
}
```

  - `total` is a local variable within the block statement attached to the `if`. It is *not* visible to any part of your program outside this block.
- When a variable is local to one function, other functions can have variables with the same name within their own context (or **scope**, to be discussed in a few slides).
- Formal parameters for a function definition are treated just like local variables within the function body.

Local Variable Example

```
int main()
{
  int x = 0, y = 0;   // x and y are local to main
  x = functionA(2);
  cout << x << " " << y << endl;
  y = functionB(x);
  cout << x << " " << y << endl;
  return 0;
}

int functionA(int a) // parameter a is local to functionA
{
  int x, y;          // x and y are local to functionA
  x = a;
  y = x * 10;
  return y;
}

int functionB(int a)  // parameter a is local to functionB
{
  int x = 10;         // x is local to functionB
  return (a+x);       // I can return any expr matching the return type
}
```

- What is the output of this program? Answer discussed in class!

Global Variables

- Global variables are variables declared outside of any function, including `main`.
- They are usually used for constants that are used throughout your program in multiple functions.
- If a constant is used in only one function, then declare it within the function.
- Only use global variables when it makes sense to do so.
  - (There is seldom reason to do so!)

ex3.cpp

```
#include <iostream>
using namespace std;

// INTEREST_RATE - Annual percentage rate
const double INTEREST_RATE = .059;

double calc_interest(double balance, int months);
// Precondition: balance contains the current balance of the account,
//               and months contains the number of months to
//               calculate interest.
// Postcondition: Return the updated balance, assuming no more money
//                has been added to the account.

////////////////////////////////////////////////
// Main
////////////////////////////////////////////////
int main()
{
  double balance;  // Balance of the account
  int months;      // Number of months to calculate interest

  cout << "Enter your current savings balance:\n";
  cin >> balance;

  cout << "Enter the number of months to calculate:\n";
  cin >> months;
```

ex3.cpp (continued)

```
  // Format the output
  cout.setf(ios::fixed);
  cout.setf(ios::showpoint);
  cout.precision(2);

  // Output results
  cout << "Current interest rate: "
       << INTEREST_RATE * 100.0 << "%\n";
  cout << "After " << months << " months, your account will have $"
       << calc_interest(balance,months) << ".\n";

  return 0;
}

////////////////////////////////////////////
// calc_interest
//
double calc_interest(double balance, int months)
{
  while (months > 0)
  {
    balance = balance + balance * (INTEREST_RATE / 12.0);
    months = months - 1;
  }

  return balance;
}
```


Scope

- We use the word "**scope**" when referring to the visibility of a variable within the context of a program.
  - If a variable has **local scope**, it is only visible within the context of the body of the function
    - A variable can also be local to a **block** of code delimited by braces { }. (See below)
  - A global variable has **global scope**, meaning, it is visible to all functions and blocks.
  - Scope is hierarchical. If a local statement block has a variable of the same name as a variable outside the scope of the local block, the compiler resolves the identifier to the block most recently entered.
  - Once the block is exited, whether it's a function body or a statement block, the variable disappears.
  - Example:

```
int main()
{
  int x = 10, total = 5;
  if (x > 5)
  {
    int total;
    total = 10;
  }
  cout << total;
}
```

What does this output?

Gotchas

- Parameter identifiers between the function declaration, function definition, or function invocation do NOT need to be the same. Only the **TYPES of the parameters must be identical.**
- Pay attention to the order of your arguments. The compiler can not catch out of order arguments in your function call.
- Function declarations must always be placed before the function definition and any function invocations.
  - Technically, function declarations are not required if your function definition is placed before any function call.
  - Function declarations are ALWAYS required in this class!

Function Design - The Black Box Analogy

- Based on the **Principle of Information Hiding**, the text describes the **BLACK BOX** analogy.
- When **using** a function, we know nothing about how the function does its job. We know only the specifications for the function.
  - Example: The sqrt function
- When **writing** a function, we know nothing *but* the specifications about how the function is to be used.
  - We know the required input into the function, and the desired return value. It's our job to produce the desired values based on the given input.
- When applied to a function definition, the principle of **procedural abstraction** means that your function should be written so that it can be used like a **black box**.
  - The user of a function should not need to look at the internals of the function to see how the function works. The function declaration and accompanying commentary should provide enough information for the programmer to use the function.
- This is why comments around your function declarations are so important! Programmers should not have to examine function definitions in order to understand the results of a function they want to use.

More on Function Comments

- To ensure that your functions are commented properly, you should adhere to these rules:
  - The declaration comment should tell the programmer any and all conditions that are required of the parameters of the function and should describe the value returned by the function. (Preconditions and Postconditions)
  - The function definition comments should explain any specific implementation details.
  - All variables used in the function body should be declared and commented inside the function body. (The formal parameters are already declared in the function header.)

Overloaded Functions

- C++ allows more than one function to have the same name. This is called **overloading** a function.
- Overloaded functions must have
  - Different numbers of formal parameters AND / OR
  - Must have at least one different type of parameter.
- All overloaded functions must return a value of the same type
- The compiler compares the parameters between the function call and the overloaded functions definitions.
  - It will choose an exact match if one is available.
  - If an exact match is not found, an integral type will be promoted to a larger integral type first, or a floating point type if necessary in order to get a match.
  - If no match is found, the compiler generates an ERROR.

Overloading Example

- ```
  double ave(double n1, double n2)
  {
            return ((n1 + n2) / 2);
  }
  ```
- ```
  double ave(double n1, double n2, double n3)
  {
       return (( n1 + n2 + n3) / 3);
  }
  ```
- The compiler checks the number and types of arguments in the function call to decide which function to use:
  - `cout << ave(10, 20, 30);`
    uses the second definition

Overloaded Function Exercises

- Suppose you have three function definitions with the following declarations:

```
1. double the_answer(double data1, double data2);
2. double the_answer(double time, int count);
3. double the_answer(double d1, double d2, double d3);

   double x, y;
```

- Which function would be invoked with the following functions calls?
  - `x = the_answer(2.0,3);`
  - `x = the_answer(y,6.0,2);`
  - `x = the_answer(5,6,4);`
  - `x = the_answer(5,6.4);`
  - `x = the_answer(y,6.0);`

ex4.cpp

```
// ex4.cpp
//Illustrates overloading the function name ave.
#include <iostream>
using namespace std;

double ave(double n1, double n2);
//Returns the average of the two numbers n1 and n2.
double ave(double n1, double n2, double n3);
//Returns the average of the three numbers n1, n2, and n3.

int main( )
{
    cout << "The average of 2.0, 2.5, and 3.0 is "
         << ave(2.0, 2.5, 3.0) << endl;
    cout << "The average of 4.5 and 5.5 is "
         << ave(4.5, 5.5) << endl;
    return 0;
}

double ave(double n1, double n2)
{
    return ((n1 + n2)/2.0);
}

double ave(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3)/3.0);
}
```


Case Study… best pizza value

- Problem definition:
  Write a program that compares two sizes of pizza and determines which pizza is the better buy
- Input:
  - Diameter of pizza in inches
  - The price of the pizza
- Output:
  - Cost per square inch for each pizza
  - Tell the user which one is the better buy, meaning, which pizza has the lowest cost per square inch.

Design of a solution

1. Get the input data for both pizzas.
2. Compute the price per square inch for the first pizza.
3. Compute the price per square inch for the second pizza.
4. Determine which pizza is the better buy.
5. Output the results.

- Notice that step 2 and step 3 is precisely the same task, however, with different data. This is a good use of a function!
- Whenever a subtask takes some values and returns a single value, this is a natural use of a function.

Get the input…

```
int main()
{
    int diameter1, diameter2;
    double price1, price2;
    double unitprice1, unitprice2;

    // Ask the user for the information
    cout << "Enter the diameter of a pizza in inches: ";
    cin >> diameter1;
    cout << "Enter the price of that pizza: ";
    cin >> price1;
    cout << "Enter the diameter of another pizza in inches: ";
    cin >> diameter2;
    cout << "Enter the price of that pizza: ";
    cin >> price2;
```

Calculate the unit prices…

```
// Calculate the unit price of each pizza
unitprice1 = calc_unitprice(diameter1, price1);
unitprice2 = calc_unitprice(diameter2, price2);
```

- And output the results…

```
cout << "Pizza 1" << endl;
cout << "  Diameter: " << diameter1 << endl;
cout << "  Price: $" << price1 << endl;
cout << "  Price per square inch: $" << unitprice1
     << endl << endl;

cout << "Pizza 2" << endl;
cout << "  Diameter: " << diameter2 << endl;
cout << "  Price: $" << price2 << endl;
cout << "  Price per square inch: $" << unitprice2
     << endl << endl;
```


Tell user which is the better buy…

```
// Tell user which pizza is better
   if (unitprice1 < unitprice2)
       cout << "Pizza 1 is the better buy." << endl;
   else if (unitprice1 > unitprice2)
       cout << "Pizza 2 is the better buy." << endl;
   else
       cout << "Both pizzas are the same value. Get 'em
 both!" << endl;
```


Still need the calc_unitprice function…

```
double calc_unitprice(int diameter, double price)
{
    double radius, area;

    radius = diameter / 2.0;
    area = PI * pow(radius,2.0);
    return (price/area);
}
```

- If we didn't know how to calculate the unitprice right away, we could have used a **function stub**… (more on this in next chapter.)

```
double calc_unitprice(int diameter, double price)
{
    // TODO!!! Need to write this function…
    return 0.0;
}
```

- Finished program is ex5.cpp…

The End