CSI 1430
Week 8 Resource

Colin Burdine

3/8/21

Major Topics:

1. Advanced Functions
2. Code Organization

Keywords: *functions, header file, source file, vectors, STL vector*

Reminder: Group tutoring sessions for this course will be held online every Monday from 7:00pm to 8:00pm CDT. For more information on how to sign up for these sessions, go to: https://www.baylor.edu/tutoring.

1 Notes on Major Topics:

1.1 Advanced Functions

Last week we introduced *functions* and how they can be used to group together operations in your code. You may have also used functions defined by the C++ standard libraries (such as the ones included in `cmath`). In this sense, functions are a powerful tool because they an *abstraction* around a particular operations. For example, the square root function and power function in the `cmath` library have the following declarations:

(*Note*: you can see these for yourself in the C++ documentation: [sqrt|`pow`])

```
1
2 // square root function:
3 double sqrt (double x);
4
```

```
// power function:
double pow (double base, double exponent);
```

You may not be aware of the underlying algorithm used to calculate the square root of a double value, however, the online documentation (and your textbook) give you a clear understanding of what the expected input arguments are and what the expected output is. This is what we mean by *abstraction*- a programmer does not need to be aware of the implementation of a function to know how to use it. However, this means that if you are going to write your own functions, it is crucial that (a) you give good names to the function name and arguments, and (b) you provide provide at least some basic documentation that shows what the description, precondition, postcondition, and output are of your function. Typically when we document user-defined functions, we do that in the corresponding header files. Below is an example of one way we can provide this documentation for a user-defined `sqrt` function in accordance with Dr. Booth's standards:

```
/* userDefinedMath.h */

/**
  * description: calculates the square root of a double x
  * return: the square root of x (double)
  * precondition: x is a non-negative double
  * postcondition: the square root of x is returned
 */
double sqrt(double x);
```

Now that we know how to properly use, write, and document functions in C++, you may be wondering what the limitations on functions are. Unlike some programming languages, C++ only permits either a single value of a specified type to be returned or no value at all (if the return type is declared as `void`). In practice, there may be instances where we may want to return more than one value from a function of one or more types. In C++ the solution to this dilemma is to use *arguments by reference*. To illustrate this, consider the following example:

```
/* doSomething.h */
...
/* (documentation goes here) */
void doSomething(int x);

/* (documentation goes here) */
void doSomethingElse(int x);
```

```
/* doSomething.cpp */
...
void doSomething(int x){
    x = 42;
}
```

```
6
7 void doSomethingElse(int& x){
8     x = -42;
9 }
```

```
1 /* main.cpp */
2 ...
3 int main(){
4     int x = 0;
5
6     // call doSomething with x:
7     doSomething(x);
8
9     cout << "After doSomething(), x is: " << x << endl;
10
11    // call doSomethingElse with x:
12    doSomethingElse(x);
13
14    cout << "After doSomethingElse(), x is: " << x << endl;
15
16    return 0;
17 }
```

In this example, the output is:

```
After doSomething(), x is: 0
After doSomethingElse(), x is: -42
```

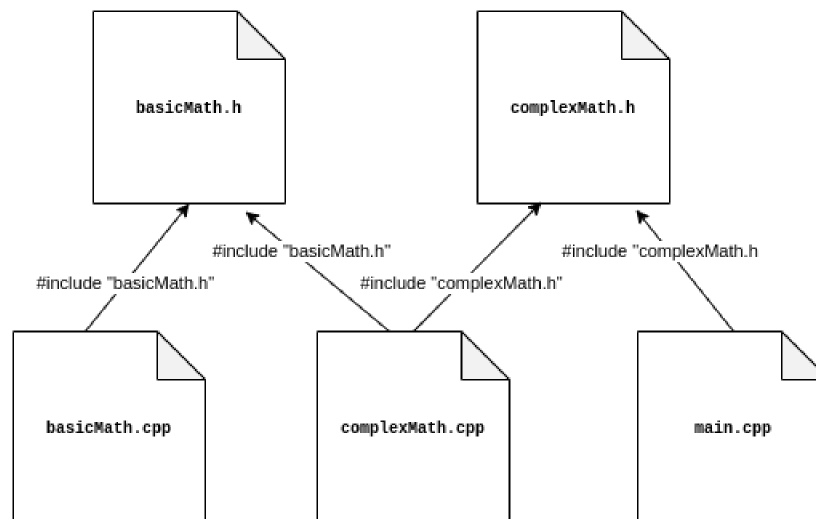
Notice that the primary difference between the `doSomething()` function and the `doSomethingElse()` function is the "`&`" character that appears before the parameter `x` in `doSomethingElse()`. In the code above, when a call is made to `doSomething(x)`, we notice that the actual value of the x that is declared in the `main()` function does not change. This is a consequence of the default behavior for function calls in C++, which is:

When a function is called, a copy is made of each of the parameters, and these copies are passed as the parameters within the called function.

In the call to `doSomething()`, notice that a copy is made of the value of x, and that copy is modified within the function body of `doSomething()`. The original variable x in the main function is left unmodified. However, when we call `doSomethingElse()`, which has the special "`&`" after the type of the parameter x, the default behavior above is overridden and we instead pass the variable `x` *by reference*. When we pass a variable by reference, it means that a copy is not made before passing the parameter into the function. As a result, any modifications made to that variable int the function body will modify the original value. This is why we get `-42` as the value of x after the call to `doSomethingElse()`.

1.2 Code Organization

Now that we have begun to introduce user-defined functions, we have begun to organize our source code into separate files, where each source file contains a group of functions. However, sometimes the C++ compiler is not as smart as we are at organizing our code and compiling it down into machine instructions; sometimes, if a function has two conflicting declarations (e.g. has two different definitions with the same name, return type, and parameter types) our code will not compile. As an example, consider the code project below, where we have a `basicMath` library and a `complexMath` library that uses the functions provided by `basicMath` to perform more complex operations. Finally, we have a main file (`main.cpp`) where our main function lives:



Notice that both of the header files `basicMath.h` and `complexMath.h` are included in two source files; thus, when the source files are compiled, the compiler will go searching for function prototypes found in the header files. However, if we are not careful, it is possible for the compiler to include the same header file twice (e.g. if it is `#included` in different source files). This can potentially create problems when we try to compile our program. The solution to this problem is to use a preprocessor directive called an *include guard*, which sets a special flag in the compiler to indicate that a header file has already been included. We will only want to add include guards to our header files, since we will never need to (and never should) include `.cpp` files. The syntax for an include guard is given below:

```
1 // Replace 'HEADER_FILENAME' with the name of the header file:
2 #ifndef HEADER_FILENAME_H
3 #define HEADER_FILENAME_H
4
```

```
5 ...
6
7 // The comment below is optional, but best practice:
8 #endif /* HEADER_FILENAME_H */
```

2 Programming Example Problem

To illustrate how we can use arrays and vectors to solve problems, consider the following example programming problem:

Many times, it is helpful to separate any general purpose "utility" functions into their own source file. This helps to reduce clutter in your coding project. In this project, you will create your own utility function source and header file for performing some common vector operations.

Create two files, `vectorUtil.h` and `vectorUtil.cpp`, and write implementation for the following functions, which operate on C++ int vectors. Be sure to use best practices, include guards, etc. You may also want to write `main.cpp` file with a `main()` that demonstrates usage of the following functions:

```
// Concatenates the contents of b onto the end of a, leaving b unchanged
// (note the 'const' keyword):
void concatenate(vector<int>& a, const vector<int>& b)

// calculates the sum of the elements of a vector:
int sum(const vector<int>& a)

// calculates the min/max of the vector a and stores the resulting values
// in min/max respectively:
void range(const vector<int>& a, int& min, int& max)

// calculates the mean of a vector:
double mean(const vector& a)
```

My solution to this problem is on the next page. You can look at my answer if you are stuck, but I would strongly encourage you to sketch out a design first, and then proceed to write your solution in C++. This exercise is intended to check your understanding of the concepts from this week and last week.

```cpp
/* vectorUtil.h */
#ifndef VECTORUTIL_H
#define VECTORUTIL_H

#include <vector>
using namespace std;

/* (description/return/precondition/postcondition here) */
void concatenate(vector<int>& a, const vector<int>& b);

/* (description/return/precondition/postcondition here) */
int sum(const vector<int>& a);

/* (description/return/precondition/postcondition here) */
void range(const vector<int>& a, int& min, int& max);

/* (description/return/precondition/postcondition here) */
double mean(const vector<int>& a);

#endif /* VECTORUTIL_H */
```

```cpp
/* vectorUtil.cpp */
#include <cmath>
#include <vector>
#include "vectorUtil.h"

using namespace std;

void concatenate(vector<int>& a, const vector<int>& b){

    // this is not needed; just an optimization :)
    a.reserve(a.size() + b.size());

    // append the elements of b to a:
    for(int i = 0; i < b.size(); ++i){
        a.push_back(b[i]);
    }
}

int sum(const vector<int>& a){

    // calculate the sum of elements of a:
    int sum = 0;
    for(int i = 0; i < a.size(); ++i){
        sum += a[i];
    }

    return sum;
}

```

```
30 void range(const vector<int>& a, int& min, int& max){
31 
32     // only calculate the range if the vector is
33     // nonempty:
34     if(a.size() >= 0){
35         int currentMin = a[0];
36         int currentMax = a[0];
37 
38         // update the min and max as we iterate
39         // over each element:
40         for(int i = 1; i < a.size(); ++i){
41             if(a[i] > currentMax){
42                 currentMax = a[i];
43             } else if (a[i] < currentMin) {
44                 currentMin = a[i];
45             }
46         }
47 
48         // set the max and min:
49         min = currentMin;
50         max = currentMax;
51     }
52 }
53 
54 double mean(const vector<int>& a){
55 
56     // calculate the total using our sum function:
57     double total = static_cast<double>(sum(a));
58 
59     // return the mean:
60     return total / static_cast<double>(a.size());
61 }
```

```
1  /*
2   main.cpp
3   (your proper header comments go here :) )
4  */
5 
6  #include <iostream>
7  #include <vector>
8 
9  #include "vectorUtil.h"
10 
11 using namespace std;
12 
13 int main(){
14     // (your tests go here)
15     return 0;
16 }
```