Functions for All Subtasks II

CS 211

Adam Sweeney

September 21, 2017

Wichita State University, EECS

Introduction

- Wrap up our discussion of functions

Agenda

- Finish Chapter 5
- Another Procedural Abstraction Example
- Testing and Debugging Functions
- General Debugging

Procedural Abstraction II

Procedural Abstraction II

- We want our functions to act as black boxes
- This does take work on our end to achieve this goal
- We'll go over another example

Supermarket Pricing

- Based on how long an item is expected to remain on the shelf at a supermarket, the wholesale price is marked up either by 5%, or 10%. If and item is expected to sell within one week (7 days max), the markup is 5%. Otherwise, the markup is 10%.
- Our function should take the wholesale price of an item and the expected shelf life and return an appropriate retail price

Breaking the problem down

- We'll be taking a slightly different approach, but it will serve to illustrate the principle of procedural abstraction
- First, we still restate the problem:
    - Problem: We need to figure out the markup of items based on their typical shelf life
    - Inputs: An item's wholesale price, and expected shelf life
    - Outputs: A properly calculated retail price

Sub-tasks

1. Input the data
    - Inputs: wholesale price (double), expected shelf life (int)
2. Compute the retail price
    - Apply the appropriate markup based on the expected shelf life
3. Output the results
    - This should be simple enough, output the retail price (double)

- In this example, each sub-task will be implemented as a function

Function declarations

```
// Parameters are actually values returned based on user inputs
void getInput(double& cost, int& turnover);

// Returns retail price based on appropriate shelf-life markup
double calculatePrice(double cost, int turnover);

// Presents previously acquired inputs & calculated output to the user
void showOutput(double cost, int turnover, double price);
```

Our `main` function

```
int main()
{
    double wholesalePrice, retailPrice;
    int shelfLife;

    getInput(wholesalePrice, shelfLife);
    retailPrice = calculatePrice(wholesalePrice, shelfLife);
    showOutput(wholesalePrice, shelfLife, retailPrice);

    return 0;
}
```

Notes on the Code

- I did skip through the planning phase of the function declarations
    - However, it is covered in lecture
- By choosing good names for our functions and their parameters, the comment only has to fill in the blanks
- The lesson here is the main function
- With only function declarations, we are able to write our main function, and have it make perfect sense as to what our program will do
- This is procedural abstraction

More Notes on the Code

- Personal notes:
    - I don't particularly like the `getInput()` function; there are cleaner ways to handle this that don't subvert our expectations of a function
    - We'll actually cover one method before the end of the semester
    - Our main functions can and should still do stuff, not just call functions, but it is important to recognize the need to break sub-tasks out into functions
    - This example takes the extreme route to illustrate its point

Debugging and Testing Functions

Debugging and Testing Functions

- One reason we divide our problem is so that once a part is done, it is done
- This means that when we finish a part, we should test it to make sure it works as intended
- Once tested and verified, we should feel confident that any bugs encountered cannot be from a finished sub-task
- How can we test our functions?

Drivers & Stubs

- Once we write a function, we should test it
- We should do this with code designed solely to test the function
- This program is called a driver
- Yes, this means writing code that would not be submitted or used in a final release
- However, it serves to validate that the function operates as expected

Another Way to Do Drivers

- The book suggests creating a program specifically for testing the function
- In industry, this is a good practice
- Hard to expect of a student, though
- Instead, write your function, then enough of your main function to perform tests on that function
- You will still be writing code that doesn't end up being used
- But you will also be writing some code that will stick around
- When you perform tests, edge cases are the best
- Typically, when we provide inputs that are expected, we get expected behavior; inputs at the edge of our expecations are the true test of our function's robustness

Stubs

- We will eventually get to a case where two functions are very integrated with each other
- We still want to test only one function at a time
- A stub is a stand-in for a function that simply provides the outputs necessary to test our single function
- Once we verify our function, we can expand the stub, test it individually, and re-test the interaction between the two completed functions
- It sounds tedious, but this eliminates a lot of potentially much harder to trace bad behavior
- Stubs also help us plan and build out our full program structure

General Debugging Techniques

General Debugging Techniques

- We keep talking about debugging
- It is important, and it is usually not easy
- This is why we revisit the topic after learning something new

Keep an Open Mind

- Don't assume the bug is where you initially think it is
- My time in lab has shown that about half the time, the bug lies elsewhere
- Do NOT throw code into the editor to see what happens
- Debugging is a controlled process
- When the decision comes down to taking a break and getting frustrated, take the break

Check the Basic Stuff

- Uninitialized variables?
- Off-by-one?
- Exceeding a data boundary? (Very imporant with arrays)
- Automatic type-conversion?
- Using $=$ instead of $==$?

Localize the Error

- Can't fix a bug if we don't know where it is!
- We can use comments as debugging tools, too
- We can comment out a line (or block) of code, and observe the outcomes
- We may not want to delete the code, because the problem may lie elsewhere, or we just need the reference
- We can trace variables with `cout`
- This can become tedious

assert

- A lot of cout statements can get tedious
- Especially when it comes time to clean them out of your code
- While we can't eliminate the need entirely, we can mitigate it quite a bit
- assert, from the <cassert> libary, can enforce conditions for us

`assert` usage

```
assert(BOOLEAN_EXPRESSION)
```

- `assert` will only allow code to continue executing if the the Boolean Expression evaluates to TRUE
- Consider the example on the following slide

Newton's Method of Calculating the Square Root

```
// All parameters must be positive
double calcNewtonSqrt(double n, int maxIterations)
{
    double answer = 1;
    int i = 0;

    assert((n > 0) && (maxIterations > 0));
    while (i < maxIterations) {
        answer = 0.5 * (answer + (n / answer));
        i++;
    }

    return guess;
}
```

So Now We've Got All These `asserts` Lying Around

- We do not have to comment them out or delete them
- We do not want to leave them in our "production" code, either
- There is a simple solution
- Above your include directive for <cassert>, place the following directive, `#define NDEBUG`

```
#define NDEBUG
#include <cassert>
```

- If we need to go back to testing, we can simply comment out the `#define NDEBUG` statement, and vice versa