COMP 161 - Lecture Notes - 06 - Functional Procedures

January 24, 2016

In COMP160, "How to Design Programs" was all about designing functions. Now that we're programming procedurally, we need to revisit our design process for statement based, functional procedures as well as effect oriented procedures. In these notes we recapture the functional design process with C++ procedural programming. You'll see that for functions, many things won't change beyond the obvious syntax shift from BSL Racket to C++. The core logic is the same, but how we express that logic is different.

Design and Development Process

It's always tempting to jump right in and start writing code. When we do this we're programming and planning at the same time. This often works out just fine[1], but as the complexity of the problem you're addressing with code increases, it's likely that you're vision for what needs to happen will get muddier and muddier. To address this we talk very purposefully about a design process. Your program and procedure designs help you establish a concrete plan and specification. A procedure's documentation and declaration clearly establish what your procedure is meant to accomplish and how it's expressed in code. The tests let you state several key cases of expected behavior. Sometimes you can implement the procedure without these in place, but if you cannot clearly and correctly state these things, then you shouldn't expect to be able to implement the procedure. If you can't write the documentation, a formal declaration, and some tests, they you probably don't have clear enough picture of what you're trying to accomplish in code.

[1] The act of programming really helps understand a problem

When designing and developing C++ procedures, we can stick to the same design process advocated by HtDP2e. Here's my updated take on that process with respect to our new set of programming tools.

1. (Design)Analyze the problem, decide on the data types[2] for the procedure's inputs and outputs, and document and declare the procedure in your library header.
2. (Design)Stub the procedure in your library implementation.
3. (Design)Compile the library to an object to check for syntax errors and warnings.
4. (Design)In your library's unit test file, write tests for the new procedure.

[2] more generally the data model

5. (DESIGN)Compile the tests to an object to check for syntax errors and warnings.

6. (DESIGN)Link the library object and the test object to make a test executable. Run tests to ensure you're tests and stubs are all setup and ready to go.[3]

[3] The test will fail. You're just checking to see that they run!

7. (IMPLEMENTATION)Finish the implementation of the procedure.

8. (IMPLEMENTATION)Recompile the library object to check for syntax errors and warnings.

9. (TESTING)Link the library object with tests object and run the tests. Debug as needed.

What you'll notice here is that I advocate for a lot of compiling checkpoints along the way. This can be tedious, but a Makefile and *make* will fix this by simplifying the build process. If we're especially lazy, we can have the default behavior of our Makefile build all our code to executable and just build everything each time we need to compile any one piece.[4]. The real goal here is to *never stray farm from code that compiles*. If it compiles without error, then you've passed the low bar of correctness: the computer recognizes your program as syntactically correct C++. If code doesn't compile, then it does nothing and is for all intents and purposes useless. As you build up a working program, you should always be adding new code to code you're certain compiles. If you don't know whether or not the code in front of you can compile or not, then you should stop what you're doing and ensure that it at least passes the compiler.

[4] This practice does not scale, but should ease you along for now

We want to have tests ready to go before we implement our procedures so that as soon as you complete an implementation you can verify that not only does it compile but it passes our basic benchmark for behavioral correctness: the tests. This means we need to ensure that tests are ready to compile *and* run them as soon as we finish the implementation. To get to this place we use a STUB DEFINITION. A stub is an implementation of a procedure that meets the signature but not the purpose. If your procedure is supposed to return a number, then you just return some random number. If it returns a boolean return then return true or false. The important thing to understand now is that there are simple ways to provide skeleton definitions that let us not only check the syntactic correctness of our declarations and tests, but let us ensure that everything will run when we're ready for it to run.

Static Typing in C++

BSL Racket didn't require you to explicitly identify what type of data your function took as input and returned as output. We of course would document in our function signatures this because passing the wrong type of data almost always leads to a RUN-TIME ERROR[5]. For example, a function that is meant t manipulate a string shouldn't be passed a number. Languages like BSL Racket are called DYNAMICALLY TYPED because the type of data associated with a particular name[6] isn't determined until run-time[7]. This type of language is fairly flexible but allows you to write something that looks syntactically correct but that contains obvious type errors.

[5] typically the program crashes

[6] or identifier

[7] dynamic→run-time

The other side of the coin is languages that are STATICALLY TYPED, like C++. Static typing means that the type of value associated with an identifier[8] is determined at compile time[9]. This means that *you must annotate your program with data types so that the compiler knows exactly what type of data is allowed for your procedures*. The reason for static typing is a stronger guarantee of correctness for programs that successful compile. With static types a compiler can catch obviously bad usage of data. If your code invokes a function that's meant to operate on numbers but you've passed it a string value, then the compiler can catch that because you've given it an explicit declaration of expected types for all your functions. All those run-time errors from BSL Racket because compile-time syntax errors in C++. This is good! The downside is you spend a lot of time annotating code and dealing with type errors at the compiler. So static types lead to better run-time correctness as the cost of the programmer's time. Dynamic types let the programmer make bad function calls, but often let you get code written quicker. It's all about trade-offs.

[8] or variable

[9] static→compile time

The last thing we need to be clear on before we move forward is what, exactly, do we mean when we say *data type*.

A TYPE is a set of values *and operations on those values.*

It's easy to lose track of the operations and forget that procedures and operators are all defined for specific data types. The only way a procedure or operator works on multiple types if it is defined for each of those types. This will quickly become painfully clear in C++.

Basic types in C++

We'll begin our journey in to C++ types with four primitive types. Primitive types are built into the language and require no external libraries in order to use them. Let's name them and look at some literal values[10].

[10] values expressed directly and readable by the compiler as such

- *int* Whole valued numbers

```
1 0 5 -34 19473 -878237
```

- *double* Decimal values

```
1.0  0.0 5.0 -0.2345 14.234932 -3.14159
```

- *char* letters and symbols

```
'a' 'A' 'c' '5' '+' '\0' '\n' '\t'
```

- *bool* boolean values

```
0 1 false true
```

There are many other primitive types available, but these will almost always get the job. If and when we need a primitive type other than these four, then it will typically be because we're dealing with a subset or variation of one of these types we'll address that type then.

Number Types

The two most common number types in C++ are *int*[11] and *double*[12]. The *int* type is used for whole valued numbers and the *double* type for real-valued number, or numbers with a decimal point. In BSL Racket numbers were numbers and there wasn't an obvious distinction between whole-valued or real-valued numbers. You even had the ability to express fractions, rational numbers, as such. In C++ we must choose one type of number or the other because *they use different circuits in the CPU*.

[11] integers

[12] Double-Precision, Floating Point Numerals

What these two number types clearly demonstrate is the typed nature of operators. The integer arithmetic works on integer values and produces an integer value *only* where double arithmetic works with and produces doubles. *There are no mixed operations.* A common gotcha that results from this is integer division. In math class 1/2 is 0.5. Notice that the operands are both integers but the result is a double. So, in C++ 1/2 ends up as 0[13]. If you want a double result at least one, and ideally both, of the operands must be doubles, 1.0/2.0. When you mix numerical types, the compiler will automatically convert them so that they're type is homogeneous. This can lead to problems if we're not careful.

[13] Remainders are always dropped which effectively rounds down the result

The other big thing we need to be aware of with computational number types is that they have limited precision. The root of the problem is that we're working with a limited number of bits. Imagine using only three digits for numbers. You could only deal with numbers in the hundreds or less. Thus, C++ numerical types have certain limits[14]. With doubles we also have to deal with values that are impossible to express in base 2 (binary). This is the same thing that we run into with base 10. You can't represent 1/3 as a decimal number without an infinite number of places. In binary, 1/10 or 0.1 is one of these numbers. You can't represent it in binary without an infinite number of bits, which we do not have.

[14] `http://www.cplusplus.com/reference/climits/`

As we carry out operations on these imperfect representations of numbers, we can run in to rounding errors, overflow[15], and underflow[16]. As programmers we must always be aware of the fact that our calculating machines are imperfect calculators.

[15] numbers too big to represent

[16] numbers too small to represent

> The math carried out by a computer is not always the math we learn in school. It's an approximation that often goes astray.

If you weren't a fan of BSL Racket's prefix notation[17] then you're in luck, C++ uses the same infix style you learned in math classes. The basic set of numerical operations are what you'd expect for the most part.

[17] operator before operands

+	int and double addition
-	int and double subtraction
*	int and double multiplication
/	int and double division
%	int remainder

Let's look at a few examples.

Here we see classic integer arithmetic. The computer will carry this out with standard order of operations so the value of this expression is 56.

```
3 + 4 * 15 - 7
```

This expression uses the integer remainder operation. You'll need to dredge up your long-division skills for this one. Recall that 3/2 is 1 with a remainder of 1. This expression takes on that remainder, 1, as its value. Put more formally, we can approach integer division and the remainder operation through a single equation. Where a/b is c with remainder r we can write $a = cb + r$. The integer division operator gives us c and the remainder operator gives us r.

```
3 % 2
```

This is an expression of double arithmetic. The value of this expression happens to be 9440.0. Notice we're keeping the 0 decimal value in order to keep the type explicit. As far as the computer is concerned 9440 and 9440.0 are two different things.

```
3.2 * 5.9 / 0.002
```

Our final example mixes integers and doubles. The truth is that the compiler will force the 3 to 3.0 in order to first carry out double division. The 5 is then converted to a double as well and the value of the complete expression is 5.75.

```
5 + 3 / 4.0
```

In general, if one double is involved in the arithmetic, the whole thing will use double operators. There are exceptions and they can cause some real headaches. This expression has a value of 2.0. Do you see why? Were you expecting 2.5?

```
1/2 + 6.0/3
```

Letters

A single letter can be represented by a *char*, or character type. By default, C++ uses the ASCII[18] encoding of letters and symbols. It's important to remember that a char value is only a single symbol. The characters that might make you think otherwise are the characters that use the escape character \. The most common example of this is the character for a newline[19], '\n'. There are several other characters using the backslash escape.

[18] `http://www.asciitable.com/`

[19] enter key

It's occasionally useful to recognize that ASCII characters have numerical values associated with them. This means that we can often trick the compiler[20] into doing unsigned integer arithmetic with characters. While this is fun and does have its uses, you shouldn't resort to this until after you've checked out the standard set of character libraries for you're desired character operation. The old C library *ctype* is a good place to start[21]. In C++ it's called *cctype* and we include it in our code with

[20] not really. it knows what's going on

[21] `http://www.cplusplus.com/reference/cctype/`

```
#include <cctype>
```

These "operators" are really just procedures, so using them requires a procedure call. Notice that procedure/function invocation in C++ is done like in mathematics[22].

[22] *name*(*arguments*)

```
tolower('a')
toupper('a')
isdigit('5')
isdigit(' ')
```

The first procedure returns *'a'*, the second *'A'*, the third *true*[23], and the last *false*[24]. The last two procedures are what we call PREDICATES. A PREDICATE evaluates its input for some logical property and returns a boolean value.

[23] or 1

[24] or 0

Booleans

Booleans are, at first glance, dead simple. There's only two values: true and false. The problem is that in C++ the integer value 0 is equivalent to false and any non-zero integer is true. These days you don't have many good reasons to leverage this fact, but sometimes you run into it by accident. The standard boolean operators look a bit different in C++.

&&	boolean and
||	boolean or
!	boolean not

We also have standard comparison operators defined for built in types.

==	equal?
!=	not equal?
<=	less than or equal for numbers
>=	greater than or equal for numbers
<	less than for numbers
>	greater than for numbers

The biggest change coming from BSL Racket that you'll experience is with the use of *and* and *or*. Not only are the operators different and infix, but they're strictly binary. Here's a BSL Racket expression and the equivalent C++.

```
(and a b c)
a && b && c
```

Similarly, the numerical comparison operators are strictly binary. Here we see a ternary Racket comparison and the equivalent C++ expression.

```
(< 5 b 10)
5 < b && b < 10
```

Expressions vs Statements

Before

Simple Functional Procedures

We'll first look at functional procedures. These are procedures which take and return values and have no side effects[25]. Let's look at two really basic numerical functions as examples. For these examples we've already decided on our types.

[25] just like Racket Functions

1. Compute the cube of an integer
2. Given the slope, y-intercept of a line, and an x-coordinate on that line, compute the y-coordinate that goes with the given x[26]. We'll use doubles for all our values here.

[26] recall $y = mx + b$

We'll put these functions in a library named *practice* with a namespace called *practice*.

Declarations

First we declare our functions in the library header. This means making the function signature and purpose clear to the reader[27]. Procedure declarations have two parts: the documentation and the function header. Let's declare our two functions.

[27] compiler and programmer

```
/**
 * Cube an integer
 * @param x an arbitrary integer
 * @return the cube of x
 */
int cube(int x);

/**
 * Compute the y-coordinate for a point on a line.
 * @param m the line's slope
 * @param b the line's y-intercept
 * @param x the x coordinate of the point
 * @return y coordinate of the point
 */
double y_coordinate(double m, double b, double x);
```

All the text between the /* and */ is a comment and ignored by the compiler. This is documentation for programmers. Notice how the documentation style we'll be using in C++ has all the things we used in BSL Racket, but presents them differently. We start with a purpose statement. Next we document each input with and param tag. Finally, we document the return value with an return. We'll learn some other tags as we go along.

Next we notice the format for the function header[28]. The first thing you see is the type of the function's return value. Next we see the procedure name. The dash - is not allowed in C++ names so we either use the underscore _ or a style called camel case, *yCoordinate*[29]. The procedure's argument is then given in parenthesis following the procedure name. Multiple arguments are separated by commas. The pattern here is:

[28] the non-comment line

[29] see the camel-like humps?

```
RETURNTYPE NAME(ARGTYPE ARGNAME,...);
```

Our style of writing libraries puts function declarations inside namespace blocks. Let's see that:

```
namespace practice {

  /**
   * Cube an integer
   * @param x an arbitrary integer
   * @return the cube of x
   */
  int cube(int anInt);

  /**
   * Compute the y-coordinate for a point on a line.
   * @param m the line's slope
   * @param b the line's y-intercept
   * @param x the x coordinate of the point
   * @return y coordinate of the point
   */
  double y_coordinate(double m, double b, double x);
}
```

If our library had more functions, then we'd put them in the same block. This block delineates the definitions found within the *practice* namespace. That's just a name we choose. The importance of the namespace name is it adds another layer of naming to our functions. This seems like extra work and complexity at first, but it pays off in the long run. Calling functions declared in a namespace looks like this:

```
practice::cube(5)
```

or more generally.

```
NAMESPACENAME::FUNCTIONNAME(ARG,...)
```

You can[30] use a *using namespace* declaration within a function to direct the compiler towards namespaces being used. For example,

[30] you'll rarely see me do it though

```
using namespace practice;
```

will direct the compiler to check the practice namespace for any definition not in the global namespace. So when we call *cube(5)* it will check practice for *cube*. You'll see this as we start writing tests.

Stubs

Next we want to "stub out" our procedures in the library implementation file. The goal is to have something that compiles and runs.

That's it. For both our functions this means making them return a number of the correct type. If we do that, then the compiler has everything it needs to guarantee that the function signature is satisfied by the definition. Let's stub.

```
namespace practice{

  int cube(int x){
    return 0;
  }

  double y_coordinate(double m, double b, double x){
    return 0.0;
  }

}
```

Alternatively we can drop the namespace block and tag each definition with its namespace like this.

```
int practice::cube(int x){
   return 0;
}

double practice::y_coordinate(double m, double b, double x){
  return 0.0;
}
```

The advantage of the first is less typing. The advantage of the second is keeping the indentation of our code down. You can decide which you prefer[31]; just know how to read and interpret both when you see them. The important thing is that you connect the definition with the namespace!

[31] I'll almost always use option 2

Procedure stubs are complete procedure definitions. They connect the header with a sequence of statements that execute when the procedure is called. The sequence of statements is called the procedure BODY OF THE PROCEDURE and is found within a set of curly braces[32] that follow the header line. The opening curly brace can also be written on the next line, but we'll prefer the style shown above in this class. Stubs typically have a single statement in the body. The *return statement*. In Racket, the return value was implicit. In C++ we must explicitly instruct the computer to return a value. The numbers following the return are the values to be returned. As is typical, a semi-colon ends the statement.

[32] not parenthesis

At this point you should stop and compile your library object file. This will catch any syntax errors and give us foundation of working code upon which we can build.

Tests

The next step is tests. Coming up with tests help us work out how the procedure should work and give us something concrete to check out implementation against when its done. In short, they force us to prove to ourselves that we known what a correct implementation of our procedure will do and they do so in a form that the computer can also check. Nothing bad comes from writing tests first. The investment of your time is worth it.

In this class we're using a testing framework written by Google for testing their C++ code. It functions on the same principles as Racket tests: check the return value of the function against an expected value. The Google testing framework requires us to put a little more effort in to organizing tests than Racket's testing framework.

For each procedure we write we'll typically define one test case and at least one test. The basic template for a test is[33]:

[33] *Avoid underscores in case and test names*

```
TEST(caseName,testName){
  // expect statements
}
```

This looks like vaguely like a procedure definition. It is not. This is a Macro. The C++ preprocessor will re-write this as C++[34].

[34] try just running some tests through the pre-processor and you'll see what I mean

There's no such thing as too many tests, but it is possible to write too few. For these simple functions we can probably get away with a single test. In general, you need to analyze the structure of the problem and determine what cases exist within the problem and test each case.

```
TEST(cube,allTests){



}

TEST(yCoordinate,allTests){
  using namespace practice;

}
```

I've avoided underscores and went ahead and put a using namespace statement in one test but not the other so you can see the difference.

The basic test we'll write is an equality check[35]. For non-double values, we can check exact equality. For doubles we'll need to check that the value is close enough to our expected value. Thankfully, Google wrote a test that does this for some fixed definition of "near"[36]. You also have the option to specify you're own nearness by giving a

[35] more here `https://code.google.com/p/googletest/wiki/Primer`

[36] `https://code.google.com/p/googletest/wiki/AdvancedGuide#Floating-Point_Comparison`

delta value below which the absolute difference between the expected and actual double values must fall[37]. Equality for doubles is fraught with problems because of the inexactness of the representation. Racket saved you from worrying about this. C++ does not.

[37] $|e - a| < DELTA$

Your first goal with tests is to come up with one or more tests that correctly capture the procedure's purpose and force every line of code in the procedure to execute. We call this CODE COVERAGE. Given that we haven't written a procedure, we'll have to think through the problem and imagine what tests will probably cover our code and add more later if needed. More generally, we should come up with a series of tests that cover a variety of situations ranging from simple to complex. A good check on simplicity is if you can do it yourself by hand or in your head. For our examples this means small numbers or numbers that make the arithmetic really easy.

```
TEST(cube,allTests){

  EXPECT_EQ(0,practice::cube(0));
  EXPECT_EQ(1,practice::cube(1));
  EXPECT_EQ(8,practice::cube(2));
  EXPECT_EQ(1000,practice::cube(10));
  EXPECT_EQ(-1,practice::cube(-1));
  EXPECT_EQ(-8,practice::cube(-2));
}

TEST(yCoordinate,allTests){
  using namespace practice;

  // constant functions
  // Using DOUBLE_EQ
  EXPECT_DOUBLE_EQ(0.0,y_coordinate(0.0,0.0,0.0) );
  EXPECT_DOUBLE_EQ(0.0,y_coordinate(0.0,0.0,3.5) );
  EXPECT_DOUBLE_EQ(2.2,y_coordinate(0.0,2.2,3.5) );
  // slope 1 that hits the origin
  // Using NEAR (the third argument is the delta
  EXPECT_NEAR(4.0,y_coordinate(1.0,0.0,4.0) , 0.00000001 );
  EXPECT_NEAR(-3.1,y_coordinate(1.0,0.0,-3.1) , 0.00000001 );
  EXPECT_NEAR(-0.014,y_coordinate(1.0,0.0,-0.014) , 0.00000001 );
  //slope 1 that doesn't hit the origin
  EXPECT_DOUBLE_EQ(7.0,y_coordinate(1.0,3.0,4.0) );
  EXPECT_DOUBLE_EQ(-1.75,y_coordinate(1.0,2.25.0,-4.0) );
  EXPECT_DOUBLE_EQ(4.0,y_coordinate(1.0,0.0,4.0) );
  // a general case
  EXPECT_DOUBLE_EQ(7.5,y_coordinate(2.5,5.0,1.0) );

}
```

When testing doubles, you can start with DOUBLE_EQ and see if your tests work within the implied correctness of that test. It's also often possible to choose test vales that are less likely to round off funny and fall short of the EQ tests. In the end, there will always be cases where you need to loosen things up a bit and use EXPECT_NEAR.

The overall pattern is to write expected values prior to actual values.

```
EXPECT_*(expected,actual);
```

Once you tests are written you can compile and run the tests. Odds are good they will all fail, but occasionally the stub gets it right. Our goal isn't for them to pass or fail it's to start the process of finishing the procedure from a place that quickly and easily allows us to check out work. So assuming our tests are correct[38], we can easily run the tests to see if your implementation is on the right check. After all, if you haven't actually run the code with real data, then you can't really claim that it works. The fact that it compiles without error means very little. We can make all kinds of garbage compile.

[38] Tests that don't represent correct behavior are always possible, so check and double check your thinking!

Completing the Definition

We've declared the procedures in our library header, stubbed them out in the library implementation, and written a well thought out set of tests. We've also compiled all of this code to rule out syntax errors and set ourselves up with a foundation of correctness from which we can work. Now let's make these procedures carry out their intended purpose by completing the definitions.

Knowing what code to write for functions on atomic data is largely a matter of understanding the problem domain. So be ready to research the problem. Use tests to explore the problem and strengthen your understanding with concrete examples because the procedure represents the solution for all possible inputs. Concrete examples help shed light on this abstraction by showing one specific case from many. After a few examples, you might see the pattern that results in the general solution.

Very simple procedures can often be written with a single return statement. That is the case with our example procedures. One definition is placed in a namespace block, the other declares its namespace in the header. You should pick one style and stick to it. I did them both only for demonstration purposes.

```
namespace practice{

  int cube(int x){
    return x*x*x;
  }
}

double practice::y_coordinate(double m, double b, double x){
    return m*x + b;
}
```

Functional Procedures With Conditionals

Let's say we needed to solve some kind of classic tax problem where for an income in 0 to 500 we pay 10% tax, for 501 to 1000 we pay 15%, and for an income above 1000 we pay 25%. Well what we have is an itemization. In this case we're dealing with a series of numeric intervals[39]

[39] notation reminder: [or] means the number is included and (or) means its excluded from the interval.

1. $[0, 500]$
2. $[501, 1000]$
3. $(1000, \infty)$

What we're looking at is a procedure that takes as input a double which is a number from one of these intervals and we'll need conditional logic to manage the problem. In Racket we had the *cond* expression. In C++ we'll mainly work with *if .... else if .. else* statements. It's important to remember that conditionals in C++ are statements not expressions!

Declarations

Really there's nothing new going on in the documentation and declaration. It would, however, be helpful if we documented the different cases.

```
namespace practice{

 /**
  * Compute the taxes for a given income.
  *   Income can fall into three brackets [0,500], [501,1000],
  *   and (1000,inf)
  * @param income The individual's income
  * @return taxes owed
  */
  double my_taxes(double income);
}
```

This declaration tells us more about the problem without crossing the line into describing a solution. The benefit to you is that you're forced to write down the variants of the input and this acts as a check on your thinking. The benefit to the reader is they know more about the problem. In a more immediate sense, when I'm reading the more complete documentation I can tell more about your understanding of the problem than I can with the first. More often than not, programming problems stem from misunderstanding the problem, not the program. I won't force you to use the more detailed documentation style, but you're doing yourself a big favor if you do. However, if you're getting help from me, then I might require you to write it if I'm not convinced you understand the problem you're trying to solve.

Stubs

You can stub your procedures for itemized data in the exact same way you do those for atomics.

```
double practice::my_taxes(double income){
   return 0.0;
}
```

We could potential start working out the logical template for the conditional, but that will tempt us to go too far and complete the implementation. The goal of the stub is to get the simplest possible definition that satisfies the signature. So once again, we simply return a literal value from our return type.

Tests

It helps if we think of procedures with conditionals as a set of related by disjoint procedures. Each variant does its own thing. In terms of testing, this implies that we write a set of tests for each variant. You can probably get away with a single set of tests for the whole thing, but when you run into a problem case, a variant that is trickier than the others, you'll end up having to stare at test results for all of the variants you don't care about. By writing a separate set of tests for each variant you have the chance to run the tests for each variant separately from the others. I won't make a big deal out of this *unless* you're having problems with a procedure for itemized data. I might then ask you to rewrite your tests to break out each case. Doing this can help narrow in on problems and forces you to think more deeply about the problem at hand.

Our tax problem has three variants so we'll write three sets of tests. When working with intervals we should be certain to test the boundary values.

```
TEST(myTaxes,from0to500){

   EXPECT_DOUBLE_EQ(0.0 ,practice::my_taxes(0.0) );
   EXPECT_DOUBLE_EQ(10.0 ,practice::my_taxes(100.0) );
   EXPECT_DOUBLE_EQ(20.0 ,practice::my_taxes(200.0) );
   EXPECT_DOUBLE_EQ(37.55 ,practice::my_taxes(375.5) );
   EXPECT_DOUBLE_EQ(50.0 ,practice::my_taxes(500.0) );
}

TEST(myTaxes,from501to1000){

   EXPECT_DOUBLE_EQ(75.15 ,practice::my_taxes(501.0) );
   EXPECT_DOUBLE_EQ(90.0 ,practice::my_taxes(600.0) );
   EXPECT_DOUBLE_EQ(112.5 ,practice::my_taxes(750.0) );
   EXPECT_DOUBLE_EQ(150.0 ,practice::my_taxes(1000.0) );



}

TEST(myTaxes,from1000up){

   EXPECT_DOUBLE_EQ(250.0025 ,practice::my_taxes(1000.01) );
   EXPECT_DOUBLE_EQ(500.0 ,practice::my_taxes(2000.0) );
   EXPECT_DOUBLE_EQ(2500.0 ,practice::my_taxes(10000.0) );
   EXPECT_DOUBLE_EQ(25000.0 ,practice::my_taxes(100000.0) );
}
```

Notice that we cover the boundary values in each variant. It's also worth noting that our double based function can and will spit out numbers that aren't dollar values. If you're writing a real application involving money, you might do well to remember that doubles are not money and you should expect to spend a lot of time managing the difference if you choose to represent the former with the later[40]

[40] look for libraries our if you can't find one build your own money type

Completing the Function

Before we get to the conditionals, let's talk about the helper procedures. You don't always have to write helpers. You do yourself some favors if you do. If it turns out one variant has some really non-obvious logic, then making a helper lets you logically and physically set that apart from the big picture of the itemization. You can set this problem case aside, finish the rest, and then go on to the complex case. In short, always using helpers will keep you sane with things get rough. It forces you to break the problem in to tiny, manageable pieces and helps to avoid getting overwhelmed by large, complex problems. I won't force you to always write helpers, but if you get stuck, I will. Our example problem is simple enough that helpers are needed, but I will talk about how to approach the task of writing and designing helpers after we do it without them.

The C++ conditional statement we'll focus on is very similar to the Racket cond expression. It lets you identify a series of cases such that if the condition on the first is false, it will move on to the next and so forth. You can also add an *else* case that catches everything that does not meet any of the cases proceeding it.

Let's start by writing a skeleton for the conditional. We have three variants so we need three cases. This first is the *if* case, the second is the *else if*, and the last can be the *else*[41]. It's helpful to label each case with a comment describing the variant you plan to associate with the case.

[41] else is tricky here. more on that later

```
double practice::my_taxes(double income){

   if(  ){ //[0,500]

   }
   else if(){ //[501,1000]

   }
   else{ //(1000,inf)

   }
}
```

We can now go in and fill in the logical expressions that check the procedure argument *income* for it's variant type. These expressions go within the parenthesis following the *if* and *else if* in the expression. No logical check accompanies *else* because it's "everything that's not one of the above things". You'll notice the strict use of binary operations that C++ forces upon us. This is also a good time to re-stub the procedure. The skeleton wouldn't compile. We don't like to stray too far from code that compiles.

```
double practice::my_taxes(double income){

   if( income >= 0.0 && income <= 500.0 ){ //[0,500]
      return 0.0;
   }
   else if( income > 500.0 && income <= 1000.0){ //[501,1000]
      return 0.0;
   }
   else{ //(1000,inf)
      return 0.0;
   }
}
```

At this point we have a completely defined function again. It's a good time to compile. You can also change the return values such that they're different for each case. By doing this you can see if each case is getting caught correctly by the return value[42].

[42] Run your test. If the value that causes a failure is the value returned in the else if and that's case you were aiming for, then you're at least picking that up correctly

Now let's finish this thing out.

```
double practice::my_taxes(double income){

   if( income >= 0.0 && income <= 500.0 ){ //[0,500]
      return income * 0.1;
   }
   else if( income > 500.0 && income <= 1000.0){ //[501,1000]
      return income * 0.15;
   }
   else{ //(1000,inf)
      return income * 0.25;
   }
}
```

We're done. Run your tests. If they fail then either the test is wrong or your code is wrong. Check both. There is one problem with this version. It's kind of wrong. If, for some reason *income* is a negative number, then the return value is 25% of that negative number. It's fair to assume that *income* is never negative. Within the problem domain it doesn't really make sense, does it? If we're making some assumption about our data, then *we must document it*. These assumptions are called PRECONDITIONS. We document them in the header documentation. So if this is our final version of *my_taxes*, then our documentation should be revised as follows:

```
 /**
```

```
  * Compute the taxes for a given income.
  *   Income can fall into three brackets [0,500], [501,1000],
  *   and (1000,inf)
  * @param income The individual's income
  * @return taxes owed
  * @preconditions income >= 0.0
  */
  double my_taxes(double income);
```

So our solution was simple, "User be warned! We do not guarantee correct functionality if preconditions are not met!". We passed the buck to whom ever uses this procedure. Alternatively we could make the reasonable assumption that negative income results in 0.0 taxes.[43]. Let's quickly review the conditional statement in general and then explore this revised version of our function.

[43] Eventually we'll explore the option of generating a run-time error

In general, these *if* based conditional can take lots of shapes. The *else* and *else if*s are optional. So, you can have just an *if*. You can have as many *else if*s as you need. If we try to mimic the notation we saw in the Bash command manuals, then we can capture the pattern as follows:

```
if( BOOL-EXP ){
  STATEMENT-SEQ
}
[else if( BOOL-EXP){
  STATEMENT-SEQ
}
... ]
[else {
  STATEMENT-SEQ
}]
```

In plain English: "Conditionals are built from an *if* statement followed by zero or more else ifs and at most one else." There are some other variations on this structure that we'll for the most part ignore[44]. For example, the curly braces are optional if the statement sequence is a single statement. There is one rule we need to be aware of for functions. There must be a *return* statement in an else or outside of the conditional. Put another way, the compiler must be able to guarantee that the function will return the appropriate data type. To see what I mean, let's look at a version of our function that handles negative numbers.

[44] see `http://www.cplusplus.com/doc/tutorial/control/`

```
double practice::my_taxes(double income){

   if( income >= 0.0 && income <= 500.0 ){ //[0,500]
```

```
      return income * 0.1;
   }
   else if( income > 500.0 && income <= 1000.0){ //[501,1000]
      return income * 0.15;
   }
   else if( income > 1000.0 ){ //(1000,inf)
      return income * 0.25;
   }

   return 0.0;
}
```

This version uses a sequence of two statements: the conditional and an unconditional return. If any of the conditions in the conditional are met, then the value indicated within that condition will be returned. So if income is 400, our function will return 40.0. *This means only one return statement is ever executed. Once a function returns, it stops executing at the return and the program continues at the place where the function was called.* Now, imagine the *return 0.0* were not there. What happens if income is -5.0? Right, you just don't know. Nothing is the best answer, and that's not acceptable because the function must return a double. To guarantee something gets returned we add that final return. This satisfies the compiler[45] and guarantees the return of a double value. While this version works, it's not my favorite style for this kind of situation. I would, instead, use an else.

[45] which knows nothing about actual values seen at when the code is executing

```
double practice::my_taxes(double income){

   if( income >= 0.0 && income <= 500.0 ){ //[0,500]
      return income * 0.1;
   }
   else if( income > 500.0 && income <= 1000.0){ //[501,1000]
      return income * 0.15;
   }
   else if( income > 1000.0 ){ //(1000,inf)
      return income * 0.25;
   }
   else{ // this shouldn't happen?!
     return 0.0;
   }
}
```

The else has no condition on it, so it would catch the -5.0 case. This version is nice because it keeps all the logic about income variants in one place. I prefer this style and encourage you to use it. The previous style is pretty common though; I call it an *implied else* as the final return is encountered if and only if all the conditions in the conditional are false. This is the exact situation that causes else blocks to execute. As we move to more complex code, we'll find good times to use an *implied else*. I just don't think this is one of those times.

Technically, we changed our problem a bit. Let's revise the documentation before we move on.

```
namespace practice{

 /**
  * Compute the taxes for a given income.
  *   Income can fall into four brackets (-inf,0),[0,500], [501,1000],
  *   and (1000,inf)
  * @param income The individual's income
  * @return taxes owed
  */
  double my_taxes(double income);
}
```

We need to test the new variant as well. This test is super easy.

```
TEST(myTaxes,negatives){


   EXPECT_DOUBLE_EQ(0.0,practice::my_taxes(-0.001) );
   EXPECT_DOUBLE_EQ(0.0 ,practice::my_taxes(-2000.0) );
   EXPECT_DOUBLE_EQ(0.0 ,practice::my_taxes(-1234.56) );
   EXPECT_DOUBLE_EQ(0.0 ,practice::my_taxes(-5.0) );


}
```

It's not uncommon to encounter something you missed when working out the logic of a function. Just be certain to go back and revisit your documentation and tests to reflect your new thinking about the problem and its solution.

OK. Before we walk away, let's look at one more way of writing this function.

```
double practice::my_taxes(double income){

   if( income < 0.0 ){
     return 0.0;
   }
   else if( income <= 500.0 ){ //[0,500]
      return income * 0.1;
   }
   else if( income <= 1000.0 ){ //[501,1000]
      return income * 0.15;
   }
   else{ //(1000,inf)
      return income * 0.25;
   }
}
```

By re-ordering how we check our variants we can leverage the implicit conditions built in the statement and simplify the boolean expressions used at each step. For example, any negative value gets caught by the *if* clause. So, if we're looking at the first *else if*, then income must implicitly be greater than or equal to 0.0 or the if clause would have been true and the function would have returned 0.0. There's no need for us to check for *income* $>=$ 0.0, we've already determined that much is true of *income*. This logic continues as we move down the conditional. Finally, the else is really a true else statement. When you hit the else, all other clauses were false and the value of *income* must be greater than 1000.0.

This *revision* of our function merits discussion. Does if offer any benefits when compared to the version that checks for negative values after checking all the original variants. Is it better? They're both equally correct. The first is arguably simpler because the exact conditions for each variant are explicitly covered in the boolean expression. The reader does not have to pickup on the implied conditions that occur as you move down the conditional statement. However, a few comments and proper documentation make this clear and the overall logic isn't too complicated. So perhaps they're both pretty simple in terms of capturing the logic needed to solve our problem. The last thing we might compare is efficiency. In this regard our revision has a slight advantage. Previously we'd check both the upper and lower boundary for each case. Now we only check one boundary. This also means we can drop the boolean *&&* operators with each case. So, it would seem that we've saved on work, but just a little. In truth, these are pretty much the same on the efficiency front. They're so close that we won't worry about it until it's a problem. That means until we have empirical evidence that this procedure is slowing things down and worth optimizing, we don't need to quibble over the differences here. Instead, you the program, can choose the option that makes the most sense to you. I prefer the second because to me, it more clearly lays out the logic of the problem as we understand it now. The first is as much a reflection of how we came to understand the problem as it is the problem itself; the negatives are tacked on after the fact and that forces us to use more verbose boolean expressions. Put in general terms, the final versions seems to me to be a function that was not only written, but revised for clarity. It's a clean second draft where the first is a less clear first draft. So, let's see that last version one more time:

```
double practice::my_taxes(double income){

   if( income < 0.0 ){
     return 0.0;
   }
   else if( income <= 500.0 ){ //[0,500]
      return income * 0.1;
   }
   else if( income <= 1000.0 ){ //[501,1000]
      return income * 0.15;
   }
   else{ //(1000,inf)
      return income * 0.25;
   }
}
```

A Recursive Function

Recursive functions require conditionals and demonstrate function calls within a function definition as they must at least call themselves. Below is the basic recursive factorial function from lab 2. Notice that there's not much new going on in terms of syntax. We do make a recursive call in the definition, but we've actually already done lots of function calls in our tests.

Documentation and Declaration:

```
namespace ver1{
  /**
   *  Compute the factorial of n
   *  @param n integer
   *  @return the factorial of n
   *  @preconditions n>=0
   */
  int factorial (int n);
}
```

Stub:

```
int ver1::factorial(int n){
   return 0;
}
```

Tests:

```
TEST(ver1,factorial){
   // base case
   EXPECT_EQ(1,ver1::factorial(0));
   EXPECT_EQ(1,ver1::factorial(1));
   // recursive case
   EXPECT_EQ(2,ver1::factorial(2));
   EXPECT_EQ(6,ver1::factorial(3));
   EXPECT_EQ(120,ver1::factorial(5));
}
```

Implementation:

```
int ver1::factorial(int n){

  if( n == 0 ){
    return 1;
  }
  else{
    return n * ver1::factorial(n-1);
  }
}
```

A Note About Predicates

Functions that return a *bool* value are often called PREDICATES or boolean-valued functions. Boolean expressions can always be replaced by predicate functions, and since boolean expressions are an integral part of control structures like our *if...else if...else* statements, we'll stop and talk about a clean, concise style of writing predicates. It's often a good idea to write predicates helpers to clear out long and hard to parse boolean expressions.

In many cases, predicates can, and should, be written without the use of conditional statements. The temptation is to view it as a two-variant itemization: all the values for which the condition is true and all the others for which it is false. For example, let's say we need a predicate *isEven* which takes an *int* type and returns true if it is even and false otherwise. Here's some tests that demonstrate it's functionality and how to test boolean values.

```
TEST(isEven,all){

EXPECT_TRUE(practice::isEven(2));
EXPECT_TRUE(practice::isEven(0));
EXPECT_TRUE(practice::isEven(-2));
EXPECT_TRUE(practice::isEven(12348));

EXPECT_FALSE(practice::isEven(1));
EXPECT_FALSE(practice::isEven(17));
EXPECT_FALSE(practice::isEven(-1));
EXPECT_FALSE(practice::isEven(1327));
}
```

To implement this We could focus on the variants and write a procedure for itemized data like this:

```
bool practice::isEven(int n){

  if( n % 2 == 0 ){
    return true;
  }
  else{
    return false;
  }

}
```

However, notice that the *boolean expression n % 2 == 0* takes on exactly the value we want to return. So, a better implementation is to simply return the value of that expression.

```
bool practice::isEven(int n){

  return n % 2 == 0;

}
```

Maybe you were thinking check for odds then let evens be the else case? That's OK, you can always use the boolean negation operator ! to "flip" the result. This

```
bool practice::isEven(int n){
  if( n % 2 == 1 ){
    return false;
  }
  else{
    return true;
  }
}
```

becomes,

```
bool practice::isEven(int n){

  return !(n % 2 == 1);

}
```

I prefer the brevity of predicates that do not use conditionals. It is, again, more or less a stylistic choice. You should be able to do both, but can choose which sits better with you in the end.

A Note on Naming and Documentation for Functions

Names should be descriptive and make sense within the domain of the problem you're solving. For our functions we need to name the function and it's arguments. All our names should describe information from the problem that our function is meant to address. Argument names should describe the information they represent and function names should describe the information represented by the return value[46].

[46] if you understand why this might be then you've got a good conceptual grasp of functions

We got away with simple one letter names for arguments with our mathematical examples because that's what mathematicians use. They make sense within the problem domain. On the other hand, we used the more descriptive *income* in our tax problem. At the end of the day, you should err on the side of descriptive.

The documentation we provide in our library header file should also be focused on the details about the problem we're addressing and the information we're representing. Rarely should we provide concrete details about *how* we're solving the problem. The biggest mistake I see students make is to write purpose statements as an English translation of the function code. This is wrong. Just wrong. You can avoid this with functions by focusing on the high-level relationship between the inputs and the output and ultimately describing the function output.

A Note on char data

We didn't see an example that uses char type data. There isn't much new going on with them. They're tested just like integers. Here are the *cctype* examples from before written as tests to illustrate the point:

```
EXPECT_EQ('a',tolower('a'));
EXPECT_EQ('A',toupper('a'));
EXPECT_TRUE(isdigit('5'));
EXPECT_FALSE(isdigit(' '));
```

The only other thing worth pointing out here is that old C libraries, like *cctype*, don't use namespaces. Their definitions are technically in the *global namespace* which doesn't require a namespace specifier or using namespace declaration. We don't put our definitions there as a matter of good style and best practices.