COMP 161 - Lecture Notes - 06 - How To Design Procedures for Atomic and Itemized Types

January 25, 2015

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. 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. We begin this process by basic forms of data and how they appear in C++.

Design and Development Process

Procedures are always designed with a particular project in mind. Before we get too deep into writing procedure code we should decide on the overall layout of our project in terms of libraries. Once we decided on the number and nature of our libraries we can get all the files setup and ready to go. Key in this process is getting a *Makefile* written. By doing this we'll be ready to easily compile and run code as it gets written. Don't start writing the real code of your project until you're ready to compile it. One important observation to make here is that we can compile and run all the boiler-plate code we need to write around our procedures. Any programs we build will obviously do nothing, but thats OK, we can build on top of this and it always give us a place to start.[1]

[1] Program from the place of adding on functionality to an existing, functioning set of code.

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.
5. (Design)Compile the tests to an object to check for syntax errors and warnings.
6. (Design)Link library object and test object to make test executable. Run tests to ensure you're tests and stubs are all setup and ready to go.[3]

[2] more generally the data model

[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 our pre-setup Makefile makes this super easy. 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].

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

As you get more comfortable with C++ you can try compiling less often. Just remember that waiting to compile can mean a long, scary looking list of errors. Its been my experience that people prefer fix just a few bugs at a time to many tens of bugs at a time. Additionally, errors beget errors so staying as close to a syntax error free program as possible is a good idea.

Another aspect of this process that may be new to you is the use of *stubs*. 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. We can expand on stubs to include a procedure template based on our analysis of the inputs and outputs. Just don't over-think the stub and start writing the real implementation. The goal of the stub is to have a functioning, complete, but incorrect definition that we can fix rather than nothing at all or something broken.[5].

[5] Once again. We're building onto a working program, not working from a blank slate

Kinds of Data, Data Models, and Basic Types in C++

The first thing a programmer must do is decide how best to MODEL real-world INFORMATION with COMPUTATIONAL DATA. You need to decide what DATA TYPES best capture the different parts of your problem and if those types aren't built in you either mimic their behavior or use things like structs and classes to define them. The big design lesson from COMP160 is that data tends to exhibit a few different structural patterns, and if we recognize the structure in our data, then we can base the structure of our program on those patterns. We wrote *data definitions* in COMP160 to explicitly state the type and structure of our problem data, then wrote templates as a reminder of the structural patterns inherent in our data. Those pattens then guide us towards an implementation for our function.[6].

[6] It's logical boilerplate!

Let's review the two most basic patterns you should have encountered in COMP160.

1. Primitive, Atomic Structures

These are the "low-level" data types that are built in to the system. The atoms. The smallest possible unit. They're things like numbers, booleans, letters, and from time to time, we'll treat non-atomic structures as if they were atomic.

2. Itemization Structures
   aka Either-Or data or Unions. We encounter this most often when we combine different kinds of data to form a new type of data or when you need to consider specific subsets or specific instances of atomic types. Numerical intervals and enumerations of values are classic examples of itemization.

One key thing to note is that itemizations are built from atomics and maybe other itemizations. In short, all roads lead to atomics and this means that our logical templates for non-atomic data tend to revolve around how we get at the underlying atomics. So without further adieu, let's see some templates.

Atomic: Compute. [7]

Itemization: Use conditionals to determine which variant of the itemization you're dealing with and then call [8] a helper procedure to process that variant.[9]

[7] Here is where we often need domain knowledge, facts and relations from the domain of our problem

[8] which often means design and write

[9] apply recursively as needed

Now that we've done some really high-level review, let's see what shape this takes in the concrete world of C++ data types.

Static Types 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[10]. Languages like BSL Racket are called dynamically typed because the type of data associated with a particular name[11] isn't determined until run-time[12].

[10] typically the program crashes

[11] or identifier

[12] 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[13] is determined at compile time[14]. 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*. Thus, 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. So 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

[13] or variable

[14] static→compile 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.

The last thing we need to be clear on before we move forward is what, exactly, is a 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.

Common C++ atomic types

We'll begin our journey in to C++ types with four primitive types. Let's name them and look at some literal values.

- *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*

```
0 1 false true
```

Number Types

The two most common number types in C++ are *int*[15] and *double*[16]. The *int* type is used for whole valued numbers and the *double* type for real-valued number, or numbers with a decimal point. There are many other numerical types and we'll check them out as needed.

[15] integers

[16] Double-Precision, Floating Point Numerals

What 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[17]. If you want a double result at least one, and

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

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.

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[18]. 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.

[18] `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[19], and underflow[20]. As programmers we must always be aware of the fact that our calculating machines are imperfect calculators.

[19] numbers too big to represent

[20] 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[21] 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.

[21] 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[22] 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[23], '\n'. There are several other characters using the backslash escape.

[22] http://www.asciitable.com/

[23] 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[24] 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[25]. In C++ it's called *cctype* and we include it in our code with

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

[25] http://www.cplusplus.com/reference/cctype/

```
#include <cctype>
```

These "operators" are really just procedures, so using them requires a procedure call.

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

The first procedure returns *'a'*, the second *'A'*, the third *true*[26], and the last *false*[27]. The last two procedures are what we call PREDICATES. A PREDICATE evaluates its input for some logical property and returns a boolean value.

[26] or 1

[27] 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
```

Functional Procedures for Atomics

We'll first look at functional procedures. These are procedures which take and return values and have no side effects[28]. Let's look at two really basic numerical functions as examples. For these examples we've already decided on our types.

[28] 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[29]. We'll use doubles for all our values here.

[29] 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[30]. Procedure declarations have two parts: the documentation and the function header. Let's declare our two functions.

[30] 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[31]. 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*[32]. The procedure's argument is then given in parenthesis following the procedure name. Multiple arguments are separated by commas. The pattern here is:

[31] the non-comment line

[32] 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,...)
```

We typically use a *using namespace* declaration within a function to direct the compiler towards namespaces being used. For example,

```
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; just know how to read and interpret both when you see them. The important thing is that you connect the definition with the namespace!

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[33] 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.

[33] 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[34]:

[34] *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++[35].

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

With atomics, it's likely we'll need just a single case and test. Let's setup tests for *cube* and *y_coordinate*.

```
TEST(cube,allTests){
  using namespace practice;

}

TEST(yCoordinate,allTests){
  using namespace practice;

}
```

I've avoided underscores and went ahead and put a using namespace statement in the test body. This keeps us from having to explicitly state our procedure namespace when writing tests.

The basic test we'll write is an equality check[36]. 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 taht does this for some fixed definition of "near"[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.

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

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

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){
  using namespace practice;
  EXPECT_EQ(0,cube(0));
  EXPECT_EQ(1,cube(1));
  EXPECT_EQ(8,cube(2));
  EXPECT_EQ(1000,cube(10));
  EXPECT_EQ(-1,cube(-1));
  EXPECT_EQ(-8,cube(-2));
}

TEST(yCoordinate,allTests){
  using namespace practice;

  // constant functions
  EXPECT_FLOAT_EQ(0.0,y_coordinate(0.0,0.0,0.0) );
  EXPECT_FLOAT_EQ(0.0,y_coordinate(0.0,0.0,3.5) );
  EXPECT_FLOAT_EQ(2.2,y_coordinate(0.0,2.2,3.5) );
  // slope 1 that hits the origin
  EXPECT_FLOAT_EQ(4.0,y_coordinate(1.0,0.0,4.0) );
  EXPECT_FLOAT_EQ(-3.1,y_coordinate(1.0,0.0,-3.1) );
  EXPECT_FLOAT_EQ(-0.014,y_coordinate(1.0,0.0,-0.014) );
  //slope 1 that doesn’t hit the origin
  EXPECT_FLOAT_EQ(7.0,y_coordinate(1.0,3.0,4.0) );
  EXPECT_FLOAT_EQ(-1.75,y_coordinate(1.0,2.25.0,-4.0) );
  EXPECT_FLOAT_EQ(4.0,y_coordinate(1.0,0.0,4.0) );

  // a general case
  EXPECT_FLOAT_EQ(7.5,y_coordinate(2.5,5.0,1.0) );

}
```

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

```
EXPECT_*EQ(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 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.

Procedures on atomic data can often be written with a single return statement. That is the case with our example procedures.

```
namespace practice{

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

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

}
```

Functional Procedures for Itemized Data

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. Each of these cases is called a VARIANT of the itemization. So, this double is not atomic, it's an itemization. The recipe for managing itemization is all about narrowing down possibilities.

> If the procedure has an input which is an itemization with *n* variants, then use an *n* branch conditional statement to determine to which of the *n* variants your input belongs.

In Racket we had the *cond* expression. In C++ we'll mainly work with *if .... else if .. else* statements.

Declarations

Declaring procedures for itemized data is in most respects the same as for atomic data.

```
namespace practice{

 /**
  * Compute the taxes for a given income.
  * @param income The individual income which can fall
  *    into three tax brackets
  * @return taxes owed
  */
  double my_taxes(double income);
}
```

The key difference is that we need at least pay some mention to the fact that we're dealing with an itemization. The declaration above is minimally acceptable. We can tell there is a conditional coming but know nothing more. We'd do ourselves and our readers a favor if we documented the conditions.

```
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 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 for itemized data as something like 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){
   using namespace practice;

   EXPECT_FLOAT_EQ(0.0 ,my_taxes(0.0) );
   EXPECT_FLOAT_EQ(10.0 ,my_taxes(100.0) );
   EXPECT_FLOAT_EQ(20.0 ,my_taxes(200.0) );
   EXPECT_FLOAT_EQ(37.55 ,my_taxes(375.5) );
   EXPECT_FLOAT_EQ(50.0 ,my_taxes(500.0) );
}

TEST(myTaxes,from501to1000){
   using namespace practice;

   EXPECT_FLOAT_EQ(75.15 ,my_taxes(501.0) );
   EXPECT_FLOAT_EQ(90.0 ,my_taxes(600.0) );
   EXPECT_FLOAT_EQ(112.5 ,my_taxes(750.0) );
   EXPECT_FLOAT_EQ(150.0 ,my_taxes(1000.0) );


}

TEST(myTaxes,from1000up){
   using namespace practice;

   EXPECT_FLOAT_EQ(250.0025 ,my_taxes(1000.01) );
   EXPECT_FLOAT_EQ(500.0 ,my_taxes(2000.0) );
   EXPECT_FLOAT_EQ(2500.0 ,my_taxes(10000.0) );
   EXPECT_FLOAT_EQ(25000.0 ,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

Let's recall the logic template for itemized data:

> ITEMIZATION: Use conditionals to determine which variant of the itemization you're dealing with and then call [41] a helper procedure to process that variant.[42]

[41] which often means design and write

[42] apply recursively as needed

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*[43]. It's helpful to label each case with a comment describing the variant you plan to associate with the case.

[43] 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[44].

Now let's finish this thing out.

[44] 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

```
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
 * @precondition: 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.[45]. Let's quickly review the conditional statement in general and then explore this revised version of our function.

[45] 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[46]. 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.

[46] 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[47] 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.

[47] 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){
   using namespace practice;

   EXPECT_FLOAT_EQ(0.0,my_taxes(-0.001) );
   EXPECT_FLOAT_EQ(0.0 ,my_taxes(-2000.0) );
   EXPECT_FLOAT_EQ(0.0 ,my_taxes(-1234.56) );
   EXPECT_FLOAT_EQ(0.0 ,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 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. We could focus on the variants and write a procedure for itemized data like this:

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

becomes,

```
bool 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[48].

[48] 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.