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

January 18, 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 and effectful procedures. Many things don't change beyond the obvious syntax shift from BSL Racket to C++. We begin this process by basic forms of data and how they appear in C++.

Design and Development Process

When designing and developing C++ procedures, we can stick to the same design procecess advocated by HtDP2e. Here we update things with respect to our new set of tools.

1. (Design)Analyze the problem, decide on your data model for the procedure's inputs and outputs, and document and declare the procedure in your library.
2. (Design)Write tests for the procedure in your test file.
3. (Design)Compile the tests to an object to check for syntax errors and warnings.
4. (Design)Stub the procedure in your library implementation. Write stubs to reflect template information whenever possible.
5. (Design)Compile the library 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 correctly. 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 is meant to keep the number of compiler errors from getting too large. 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 thing that might be new is the use of *stubs*. A stub is an implementation of a procedure that meets the signature but not the purpose. So 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.

Kinds of Data and Data Models

The most important part of program design happens before you write a single line of code. The first thing a programmer must do is decide how best to MODEL real-world INFORMATION with COMPUTATIONAL DATA. For example, if your program is a Hangman game, then you need to decide what DATA TYPES best capture the different parts of the game. 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.[1].

[1] 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. Things like numbers, booleans, and letters fall into this category. From time to time, we treat non-atomic structures as if they were atomic.

2. Itemization Structures
   aka Either-Or data or Unions. We encounter this when we combine different kinds of data to form a new type of data. It also pops up when you need to consider different subsets or specific instances of atomics. Numerical intervals and enumerations of values are classic examples.

One key thing to note is that Itemizations are built from atomics and maybe other itemizations. In short, all roads lead to atomics. What this means is that our templates for non-atomic data tend to

revolve around how we get at atomics, where we actually do the computation. For example, here's the templates for our two kinds of data.

> ATOMIC: Compute.
>
> ITEMIZATION: Use conditionals to determine which variant you're dealing with, then write a helper procedure to process the variant specifically.

With itemizations, we figure out which variation of the item we're dealing with and call a helper that works with that type of data. If that type is atomic, then we'll compute. If it's another itemization, then we'll repeat until we hit something atomic.

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

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 this because passing the wrong type of data almost always leads to a RUN-TIME ERROR[2]. Languages like BSL Racket are called DYNAMICALLY TYPED because the type of data associated with a particular name[3] isn't determined until run-time[4].

[2] typically the program crashes

[3] or identifier

[4] 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[5] is determined at compile time[6]. This means that *you must annotate your program with types so that the compiler knows exactly what type of data its dealing with*. The reason for static typing is correctness. 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 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.

[5] or variable

[6] static→compile time

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

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

The important thing to note is that both values and operations have types.

Common C++ atomic types

We'll begin our journey in to C++ types with four primitive types:

- *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 C++ number types are *int*[7] and *double*[8]. 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 that we'll look at as needed. The most important thing to point out right now is that you can either do double arithmetic or integer arithmetic but not a mix. Integer arithmetic produces integer values and double arithmetic produces doubles. The most common gotcha is with integer division. In math class 1/2 is 0.5, but in C++ 1/2 is 0. See, the result must be an integer, and it doesn't know to round up or down because integers have no fractional part[9].

The other big thing we need to be aware of is that they have limited precision. As we saw in homework 2, int types can't store certain numbers[10] because they're too big. The problem is that your limited to a fixed number of digits[11]. If I cut you off at three digits, then you can't do anything in the thousands. With doubles the problem is worse. There are more distinct numerical values between 0 and 1 then there are integer values[12]. So while doubles have twice the storage capacity of an int[13], they have infinite more values to attempt to represent! Furthermore, certain common values, like 0.1, are actually impossible to accurately represent in the binary system used by doubles. What you need to take away from all of this is:

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

[7] integers
[8] Double-Precision, Floating Point
[9] so basically it always rounds down
[10] `http://www.cplusplus.com/reference/climits/`
[11] 32 bits
[12] `http://en.wikipedia.org/wiki/Infinity#Cardinality_of_the_continuum`
[13] 64 bits

If you weren't a fan of BSL Racket's prefix notation[14] 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.

[14] 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:

```
3 + 4 * 15 - 7

3 % 2

3.2 * 5.9 / 0.002

5 + 3 / 4.0
```

The first two use *int* operators and the third uses double. The last is kind of tricky because the compiler will do the whole thing with double operators and convert[15] the *int* values to doubles before it does. In general, if one double is involved in the arithmetic, the whole thing will use double operators.

[15] aka CAST

Letters

A single letter can be represented by a *char*, or character type. By default, C++ uses the ASCII 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[16], '\n'. There are several other characters using the backslash escape.

[16] 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[17] 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 the operation you're looking for. The old C library *ctype* is a good place to start[18]. In C++ it's called *cctype*.

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

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

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

```
tolower('a')
```

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

The last two procedures are what we call PREDICATES. A PREDICATE evaluates its input for some logical property and therefore returns a boolean value.

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
==	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

We'll first look at functional procedures. These are procedures which take and return values and have no side effects[19].

[19] just like Racket Functions

Library Declarations

Before we get to writing the function, we first declare it in the library header. This mean making the function signature and purpose clear to the reader[20]. Declarations have two parts: the documentation and the function header.

[20] compiler and programmer

```
/**
 * Carry out some computation on anAtom
 * @param anAtom an integer
 * @return another integer
 */
int function_on_atom(int anAtom);
```

All the text between the /* and */ is a comment and ignored by the compiler. This is documentation of the programmer. 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[21]. The first occurrence of *int* indicates the return type, or our procedure. 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, *functionOnAtom*[22]. The procedure's argument is then given in parenthesis following the procedure name. The pattern here is:

[21] the non-comment line

[22] 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 atomic{

  /**
   * Carry out some computation on anAtom
   * @param anAtom an integer
   * @return another integer
   */
  int function_on_atom(int anAtom);

}
```

If our library had more functions, then we'd put them in the same block. This block declares the *atomic* 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:

```
atomic::function_on_atom(5)
```

So the pattern for calling functions is,

```
namespace_name::function_name( argument list..)
```

There are a couple of ways around the namespace specifier. We'll save them for later. For now, writing out the "full name" of your functions is good reinforcement of what's going on in the organization of the code.

Writing Tests

Once the library declarations are written we can start writing tests. The compiler knows enough about the function to at least recognize a properly written test. With that, we can build the object for our test file. In BSL Racket we'd use *check-expect* to write tests.

```
(check-expect (function-on-atom 5) 10)
(check-expect (function-on-atom 51) 13)
```

In this class we're using a testing framework written by Google for testing their C++ code. The basis for testing is the same, check the result of the function against an expected value. However, we need to put a little more effort in to organizing tests. For each procedure we'll typically define one test case and at least one test. The basic template for a test is:

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

Avoid underscores in case and test names. If we want to do a basic equal test, then we'll typically use one of two expect statements. The first is for non-double values and the second is for doubles.

```
EXPECT_EQ(10,atomic::function_on_atom(5));
EXPECT_FLOAT_EQ(10.0,atomic::function_on_double(5.0));
```

The pattern here is:

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

Notice the new convention to write expected values prior to actual computed values.

Putting this together, our BSL Racket tests from before now get written as this gTest test case:

```
TEST(funOnAtom,allTests){
  EXPECT_EQ(10,atomic::function_on_atom(5));
  EXPECT_EQ(13,atomic::function_on_atom(51));
}
```

Stubs for Atomics

If the input to your procedure is a primitive or atomic structure then you're free to compute with it. As a BSL Racket template, we might write something like this:

```
(define (function-on-atom anAtom)
   (... anAtom ...) )
```

Now in C++ we'd write something like this in our library implementation file:

```
int atomic::function_on_atom(int anAtom){
    return ...anAtom...;
}
```

Notice that we have to put the namespace name in front of the function name this time.

The body of the procedure is found within a set of curly braces[23]. The opening curly brace can also be written on the next line, but we'll prefer the style shown above in this class. For our template body we write the outline of a RETURN STATEMENT. These statements tell the computer to return, as output the value of the expression following them. The big thing to notice is that the statement is terminated with a semicolon. This is true of most statements, and given that procedures are sequences of statements, you need to get used to ending your statements with a semicolon.

[23] not parenthesis

In this class we'll prefer stubs over templates. A STUB is basically a template that compiles, and we always like to work with compilable code whenever possible. So leaving the ... sitting around isn't really something we want to do. When writing a stub, your goal is to write a procedure that passes the compiler, not a procedure that does what you need it to do. To turn the above to a stub we'd simply remove the dots.

```
/**
 * Carry out some computation on anAtom
 * @param anAtom an integer
 * @return another integer
 */
int function_on_atom(int anAtom){
    return anAtom;
```

```
}
```

This is now an actual C++ procedure that will return the value of anAtom. Alternatively we could just return 0[24], but it's nice to keep the reminder that we probably need or want to do something with anAtom.

[24] or any int literal

Stubs for Itemization

Procedures on itemizations work from the following template:

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

Let's say we needed to solve some kind of classic tax problem where for numbers from 0 to 500 we do one thing, from 501 to 1000 we do something else, and for numbers above 1000 we do yet another thing. Well, then we're dealing with an itemization and as we know, we need a CONDITIONAL. Here's a BSL Racket skeleton:

```
(define (my-tax-func income)
   (cond [(<= 0 income 500) ...]
         [(<= 501 income 1000) ...]
         [(> income 1000) ...]))
```

In C++ we might go with the following stub[25].

[25] let's assume this is declared in our library header

```
int atomic::myTaxFunc(int income){

  if( 0 <= income && income <= 500 ){
     return 0;
  }
  else if( 501 <= income && income <= 1000){
     return 1;
  }
  else if( income > 1000 ){
     return 2;
  }

  return -1;
}
```

The *if...else if* statement is our general purpose conditional statement in C++. You can probably quickly guess how it works from the above example. Here I choose to return different values for each case; it might help with debugging and testing later. We could also have returned *income* for each case as the real goal here was get the variant

checks setup. The final *return -1;* is required by the compiler, which must be able to guarantee that an integer is, in fact, returned. If all of your returns are behind and *if* or and *else if*, then the compiler can guarantee one of them will be reached.

But what if we're not sure about the variant check? What then? Abandon the conditional? No Way!

```
int atomic::myTaxFunc(int income){

  if(true){
     return 0;
  }
  else if(false){
     return 1;
  }
  else if( false){
     return 2;
  }
  return -1;
}
```

This stub still compiles and let's us get the conditional scaffolding up without having to worry about how to actually check each variant. The moral of the story is: *there's zero reason to not stub at least the conditional scaffold.*

Now one quick note on the use of *else*. We could have approached the tax function like this:

```
int myTaxFunc(int income){

  if( 0 <= income && income <= 500 ){
     return 0;
  }
  else if( 501 <= income && income <= 1000){
     return 1;
  }
  else{
     return 2;
  }
}
```

The appealing part of this is that an unconditional *else* saves us the odd *return -1* from before[26]. But notice there's a fundamental difference between this version and the one we had before. Do you see it? The *int* type allows for positive and negative numbers. So, in the version that uses *else*, our else clause would catch negative numbers

[26] do you see why?

as well. There are several ways to deal with this, but the main point is this: *don't use else unless you truly mean all other possible inputs.*

Completing the Function

Once we've got declarations, tests, and stubs in place, we can go ahead with implementing the function. Let's consider an example:

> Convert a Fahrenheit temperature to Celsius

What might the design setup look like?

The header for our library contains:

```
namespace temperature{


  /**
   * Convert Fahrenheit to Celsius
   * @param ftemp is a Fahrenheit temperature as a double
   * @return the Celsius equivalent of ftemp, as a double
   */
   double FtoC(double ftemp);
}
```

And some tests for our test file:

```
TEST(FtoC,alltests){
   EXPECT_FLOAT_EQ(0.0,temperature::FtoC(32.0));
   EXPECT_FLOAT_EQ(-27.0,temperature::FtoC(-16.6));
   EXPECT_FLOAT_EQ(27.0,temperature::FtoC(80.6));
}
```

This is an atomic data problem, so we have an easy stub:

```
double temperature::FtoC(double ftemp){
return ftemp;
}
```

Now to implement we "simply" replace the stub value with an expression to compute our desired result.

```
(ftemp - 32.0)*(5.0/9.0)
```

Notice I used *double* literals to avoid any operator type confusion. So, putting this in the return statement we'd see:

```
double temperature::FtoC(double ftemp){
return (ftemp - 32.0)*(5.0/9.0);
}
```

We probably shouldn't expect all of our function to come out so simply. None, the less, these simple functions give us a good place to start and let us work the kinks out of basic C++ syntax and library development.

Predicates

Functions[27] that return a *bool* value are called PREDICATE or boolean-valued functions. Boolean expressions can always be replaced by predicate functions, and since boolean expressions are an integral part of loop structures, we'll stop and talk about a clean, concise style of writing predicates.

[27] procedures sans side-effects

In many cases, predicates can, and should, be written without the use of conditional statements. 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 do 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 return the value of that expression.

```
bool isEven(int n){

  return n % 2 == 0 ;

}
```

Maybe you tested to see if it was odd? That's OK, you can always use ! to negate a boolean value. This implementation,

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

becomes,

```
bool isEven(int n){

  return !(n % 2 == 1) ;

}
```