COMP 161 - Lecture Notes - 07 - Procedures for Effect: I/O and State

Spring 2014

In these notes we look at something new: procedures for effect. We'll discuss how they might arise in the course of program design and how to properly implement and test the most common procedure forms.

Thinking about Architecture

When you designed Universe programs in BSL Racket, you used a particular architectural pattern to organize you code. First you did what we always do, *develop a computational* MODEL *of your problem's information*. You then wrote a series of functions to manage three things:

1. MODEL Function like your tick handler which update and work with the data model
2. *view* Functions to draw your model and display a user interface.
3. *controller* Functions to oversee the interaction of the model and the view/UI.

The real CONTROLLER was *big-bang* which glued all of these things together and managed the event-handling. We did provide some essentials for things like key presses and a stop condition. This SOFTWARE ARCHITECTURE is an industry standard in many programming situations and it is called MODEL-VIEW-CONTROLLER[1]. It allows you, the programmer, to maintain relative independence between your data model and your user interface. It is highly useful for several reasons:

[1] MVC

- we often need to employ non-obvious models and algorithms on those models. In these cases, the structure of the model and the logical "view" of the information do not match and this separation is vital[2].
- we often need to check, validate, and pre-process data coming from the UI and this architecture provides the breathing room in which to do that.

[2] the user cares about information, not clever models

So far we've really just been looking at writing functions with an eye towards our model. We'll now start looking at interfaces to and views of our model. The MVC architecture lets us maintain the all important *separation of concerns* when developing our complete program. If we do it right, then we can develop our model logic and interface logic SYSTEMS more or less independently of one another.

Interest Program

As soon as we start thinking about user interfaces[3], we start thinking about I/O and STATE. Let's say you're developing a program to allow someone to better understand the interest rate and interests associated with their bank account, and that the program provides the following functionality:

[3] UI

- Get the initial account balance from the user
- Deposit and withdraw some funds.
- Communicate the current interest rate for the current balance to the user
- Communicate the interest earned for the current balance to the user

The first reason you might need state is to act as a *communication channel* between the UI and the model. Our program can create a STATE VARIABLE that the UI can write data in to. The program can then read from that variable and react to the value. Changes in the state of the variable[4] thereby determine how the program executes.

[4] i.e. its value

This is our new systems oriented view of programming[5]. We've clearly associated parts of the computer with this problem. The user types at the keyboard and we write that data to memory[6]. We can then read from that place in memory to determine what the user wrote. Our program plays the role of the CPU in this story.

[5] as opposed to our old functional view

[6] our variable

State Variables

We previously encountered variables when dealing with the linux CLI and noted that three logical operations tend to govern our interaction with variables.

- INITIALIZATION
  Assigning a starting, initial value to the variable.
- MUTATION
  Assigning a new value to the variable
- ACCESS
  Viewing or getting the current value in the variable.

In C++, variables are typed. This means they can only hold values of that type. When we first add, or *declare*, a variable to our program we must state its type and name and initialize its value.[7]. Let's declare a double typed variable called balance and initialize it to zero.

[7] it's possible to declare variables and not initialize them. I'm not even going to show you how to do it because you shouldn't

```
double balance = 0.0;
```

Let's break this down.

You should recognize things like *double balance* from declaring procedure arguments. The purpose here is the same, to introduce a variable[8]. The difference is all about context. Here we're talking about introducing variables to arbitrary sections of our code. With procedure argument variables, we're only introducing a variable to the procedure, and nowhere else.

[8] we don't always treat procedure arguments like variables, but they are

After the declaration of type and name is the initialization. For this we use the ASSIGNMENT OPERATOR, `=`. This is not the = you know and love from mathematics. This is an operator which assigns a value from the right hand side to a variable on the left hand side. The = symbol you know from mathematics is more closely related to the == operator as it is used as an assertion of equality. *The C++ = is not a logical assertion its a* MUTATOR. Returning to our example, we see that we're assigning 0.0 to the newly created variable *double* and thereby initializing it.

There's another way of initializing variables. The benefit of this second syntax is that it is specifically designed for initialization. Our previous syntax used the assignment operator =, which we'll use for general purpose mutation as well as initialization. The following is equivalent to what we saw before.

```
double balance(0.0);
```

Here we put the initial value in parenthesis following the name of the variable.

Now that you've declared and initialized a variable, you'll probably want to access it. For example, you might like to compute the interest earned for the balance given that the interest rate is 4.5%. Accessing variables in C++ is as simple as using the name where you want the value. The following expression[9], computes the interest.

[9] not statement, expression. Know the difference?

```
balance * 0.045
```

Notice that *you do not restate the type of the variable*. That's a declaration thing, not a usage thing. You've already established that *balance* is of type *double*. This mechanic of associating name with value is familiar to you from BSL Racket. The big difference is that in BSL Racket, the value associated with a name was constant. In C++, we can change, or mutate, the value.

Maybe you want to assign a new value to *balance*. Well we've already seen the operator for this: =. To change *balance* to 4500.00 we'd use the following assignment statement.

```
balance = 4500.00;
```

Again, notice we do not restate the type of *balance.* This statement looks very similar to our first declaration and initialization statement. It's this similarly that makes the second initialization syntax appealing; there's no way to confuse that with this. It's also worth noting that in this context, the system interprets *balance* not as value but a location. The technical distinction is made by calling balance in this context an L-VALUE[10]. When we're using balance for it's stored value, then we call it an R-VALUE[11]. These terms get thrown around by the compiler from time to time, so do not disregard them as overly technical.

[10] location value

[11] thing to the right of =?

Let's try another common form of mutation, updating. Say you want to add 500.00 to the current balance. This very common form of mutation/assignment statement highlights the L-VALUE/R-VALUE distinction and the fact that = is not = from mathematics.

```
balance = balance + 500.0;
```

Here we see *balance* used for both its L-VALUE and its R-VALUE. We also see an expression that makes zero sense mathematically[12] but perfect sense as an expression of mutation. To the right of = we'd compute the current balance value plus 500.0 and then store that result back in balance.

[12] $2 = 2 + 1$ is just bananas

Let's recap. We have basic statements and operators to support the three fundamental usages of variables.

- INITIALIZE

  ```
  double balance(0.0);
  ```

- *Mutate*

  ```
  balance = 500.0;
  ```

- ACCESS

  ```
  balance
  ```

The first two forms are complete statements. We use them for effect. The final is an expression that shows up in statements like *balance = balance + 500.0*. We use it for value.

We also learned a new operator for variable assignment. Here it is along with some shortcut operators for updating assignment that you might find useful:

Operator	purpose
$=$	assign r-value to l-value
$+=$	$a+=b$ is the same as $a=a+b$
$*=$	like $+=$ but with $*$
$-=$	like $+=$ but with $-$
$/=$	like $+=$ but with $/$

Streaming I/O

Without a GUI system in place, we have two basic means of interfacing with our model.

1. the CLI
   Users interact with the program via the keyboard and monitor. This typically happens one of two ways:
   (a) Interactive
       While running, the program prompts the user for input and reacts according to their input
   (b) Command-line arguments
       User input is given as command-line arguments. The program then proceeds, without user intervention, based on command line arguments.[13]
2. files
   Program I/O is done via files. Files can be read/written by human or computer users.

[13] hybrid modes are also possible. Begin based on command line arguments, then ask for user intervention when needed

In C++ we have a generalized I/O framework that provides a fairly uniform I/O experience for both CLI and file based I/O: I/O Streams. A stream is an arbitrarily long sequence of data made available over time. The C++ *iostream* library captures I/O devices as streams and provides the programmer with operators for adding data to or taking it out of these streams. The *fstream* library allows us to create I/O streams from files. Finally, the *stringstream* library lets you create a stream from string data[14].

[14] which is how we receive command line arguments, so we'll come back to this one

Output on the Standard Output and Standard Error

The *iostream* library provides our two standard output streams, both are defined within the *std* namespace. The standard output is called *cout*[15] and the standard error output is called *cerr*[16]. The streaming output operator, $<<$. lets you add a typed value to the stream.

[15] or std::cout
[16] std::cerr

Perhaps the biggest gotcha with output streams is that they'll output only what you add to the stream. So, if you wanted to print 3 numbers, each separated by a space, to *std::cout*, then you'd have to add both the numbers and the spaces. People often leave out the spaces. Here's an output statement for printing 3 space separated numbers.

```
std::cout << 4 << ' ' << 3.2 << ' ' << -56;
```

Notice that You need one operator per item added to the stream. The $<<$ operator is technically a binary operator. It takes an output stream on the left and some typed data to the right and effectively returns the stream again. Returning the stream is what allows us to keep adding more data.

The next big gotcha is assume that ending the statement line ends the printed line. The output statement above would effectively leave the output cursor after the -56, so the next thing you add to the output stream would show up right next to that number. People often assume that ending the statement ends the printed line such that the next thing printed shows up on a new line. That's not the case. You must add the newline character[17] to the stream if that's what you want. We can revise the above statement

[17] '\n'

```
std::cout << 4 << ' ' << 3.2 << ' ' << -56 << '\n';
```

Writing data to the standard error works the same way. You just swap out the streams!

```
std::cerr << 4 << ' ' << 3.2 << ' ' << -56 << '\n';
```

We learned a new operator. Let's take note of it.

Operator	Purpose
$<<$	Streaming output

Input form the Standard Input

The *iostream* library defines an input stream for the standard input[18] called *cin* within the *std* namespace. We can then use the streaming input operator $>>$ to read data from the stream and into a variable. Notice the mechanism requires variables. Let's say we wanted to get three numbers from our user. First we need three variables.

[18] keyboard

```
int x(0);
int y(0);
int z(0);
```

It is possible to combine multiple variable declarations, for variables of the same type, in to one statement. Don't go nuts with this shortcut though. It arguable decreases human readability and that's a bad thing.

```
int x(0),y(0),z(0);
```

Now that we have our variables we want to use input to write values to them. You again have options here. You can write input to the variable one at a time.

```
std::cin >> x;
std::cin >> y;
std::cin >> z;
```

Or you can do them all at once.

```
std::cin >> x >> y >> z;
```

Streaming input has its own quirks. By default, the system assumes that your input *tokens*[19] are separated by white space characters[20]. This includes space, tab, and the newline character. So the above examples work under the assumption that the user types the number then follows it with either spaces, tabs, or enter. Later we'll look at how to control the way in which data is read from the stream by setting things like the token delimiter.

[19] logical chunk of data

[20] white space is called the DELIMITER

We learned a new operator. Let's take note of it.

Operator	Purpose
>>	Streaming input

Designing for Effect

Let's return to our banking program. The system, as it stands, will use a single STATE VARIABLE to maintain the balance. The interface initializes the state through user input and then allows the user to modify state through deposits and withdraws. They then access the state when computing interest rates and amounts. Let's point out all the obvious I/O and Variable effects that will occur in this program:

1. State
   (a) Mutator(s) for adding and subtracting from the deposit state
   (b) Accessors that compute interest rate and amount based on the deposit state
2. I/O
   (a) Input initial value to deposit
   (b) Output interest rate and amount in human readable format[21]

[21] sometimes called pretty printing

Remember that whole MVC architecture we talked about? Well it seems we're getting pretty well setup for that. The MODEL is our state variable and the VIEW is carried out by various I/O procedures. Eventually we'll write code to control the interaction of these two things. This code makes ups the CONTROL and a good portion of it will be housed in our *main* procedure. As we proceed lets remember that one of the points of this architecture is to separate issues of the model from issues with the view. In our case that means keeping state procedures and I/O procedures as separate as possible.

Procedures for Effect

Thus far, we approached the design and develop procedures by doing the following:

1. Declare and Document the procedure
2. Set clear input/output expectations by writing unit tests
3. Stub the procedure to outline basic logic
4. Complete the procedure by filling in the stub
5. Test the procedure by running the unit tests.

We use basic tests for two reasons: *to help us understand the expected behavior of the procedure and to help us evaluate the correctness of our implementation.* The problem we face when procedures are run for effect is that effects do not occur through the usual function input, function output mechanic. So, we either need to write different forms of tests than what we're used to, or we need to find another way to design expected behaviors and test for correct behaviors.

Procedures on State Variables

Let's say we have a procedure called *foo* which takes a double as its input. Now, let's say we also have a double variable called *aDub*. How should we interpret the following function call expression?

```
foo(aDub)
```

We actually have two options:

1. pass the value, as in treat *aDub* as a R-VALUE
2. pass the variable, as in treat *aDub* as a L-VALUE

We call the first option, PASS-BY-VALUE, and the second PASS-BY-REFERENCE. C++ allows you to make this distinction for each input of a procedure.

Pass-by-Value

The default behavior of C++ is PASS-BY-VALUE. The procedures you've been working with thus far are all pass-by-value. Let's say *foo* has the following signature:

```
double foo(double arg1);
```

Then the function call expression *foo(aDub)* from before would initialize the variable *arg1* with the current value of *aDub)*. This is the essence of PASS-BY-VALUE and is the exact same function call mechanic you're used to from BSL Racket[22]. Using more technical terminology, we say that pass-by-value parameters can take, as inputs, any thing with an R-VALUE. This includes both variables and literals. So, we could also make the call *foo(5.0)* and the compiler would have no problem with this.

[22] SUBSTITUTE every occurrence of the variable for its value in the body of the function

When it comes to working with our state variables, we'll prefer *pass-by-value procedures for our* ACCESSORS. By copying values rather than sharing variables we can be certain that we only access and don't get tempted to mutate.

Pass-by-Reference

If we truly want a procedure to act on one of our variables, then we need the *l-value* of the procedure's argument to be the same as the *l-value* of our variable. Put another way, we need the function's argument to be an ALIAS for our variable such that the two procedures have SHARED STATE. A subtle change in the procedure's signature will cause this to happen:

```
double foo(double &arg1);
```

Now, when we call *foo(aDub)* the system will effectively set the *l-value* of *arg1* to be that of *aDub* thus causing *arg1* to be an ALIAS to *aDub*. It's important to note that this means a by-reference parameter *must have an* R-VALUE. This means *you cannot pass in literal values for by-reference parameters, you must pass variables!*

PASS-BY-REFERENCE is the required mechanism for MUTATOR procedures. The called procedure cannot mutate a state variable without having direct access to that variable. That's what pass-by-reference accomplishes and pass-by-value does not. A systems oriented view of this new mechanic might describe this shared state as a *communication channel* between the caller of the procedure and the called procedure[23]. Prior to this option, we could only communicate between procedures by passing in values as input, and returning values as output.

[23] the callee

Why make a big deal out of this communication centric view? Well, just like BSL Racket functions, we want out C++ procedures to be as independent as possible[24]. We can essential view them in the same way we did CLI commands. We construct more complex programs by combining these procedures. Just like on the CLI, we sometimes need these procedures to communicate either through state[25] or by handing off results of one command to the next [26].

[24] this is why pass-by-value is the default

[25] like how cd, pwd, and ls communicate

[26] like using pipes or expansions

This new shared state mechanism is game-changing. Sharing state is very powerful, but easily leads to problems. We'll try to be very careful and purposeful with pass-by-reference. Right now, we'll confine our usage of it to state mutator procedures.

Designing Mutation Procedures

If you want a procedure to change the value stored in a variable then you need to pass it by reference. Typically, such a procedure requires no output as the desired behavior is a side-effect.[27]. To declare that a procedure returns nothing, we give it the *void* return type. When procedures have effects, like state mutation, then we need to clearly document the side-effect as a POST-CONDITION of the procedure.

[27] the "output" is placed in the variable to be read as needed

Let's think about the problem of depositing and withdrawing from our bank account. We could write two procedures, one for withdraw and one for deposit. However, it's easy enough to kill to birds with one stone and just add a negative amount when withdrawing. So, lets write a general *updateBalance* procedure. First we declare and document:

```
namespace banking{

 /**
   * Update the current account balance
   * @param upVal is the update value as a double
   * @param balRef is a reference to the double balance state
   * @return none
   * @pre balRef is positive. and upVal will not overdraw the account
   *    (balRef >=0 and upVal+balRef >= 0)
   * @post balance state has increase/decreased by upVal
   */
 void updateBalance(double upVal, double &balRef);
}
```

I stuck something new in there. The *pre* tag is used to document the procedure's PRE-CONDITION. A PRE-CONDITION is an assumption the programmer makes about inputs to the procedure and sometimes the state of the system. In this case, we're assuming that the update value won't cause a negative value and that the current balance is positive. This frees us from having to check for overdraws here but probably means some other part of the system needs to do this for us. Pre-conditions and post-conditions are important parts of program correctness. They are particularly important when state enters the picture as they document state change across the system. Finally, I want to point out a stylistic convention used. Notice I used "Ref"[28] in the name of the alias. This is a nice naming style that helps us remember that this thing is not just a variable, but a reference to another variable somewhere else in the system.

[28] short for reference

Now we want to document expected behavior in the form of tests. But wait! This procedure produces no output and can't be tested like we're used to. No worries. We're not testing for functional output anyway, we need to test for variable mutation effect. To do this right we should initialize a variable, check it before mutation, then check it after mutation to see if the desired change has occurred. When doing this right, our tests should not be dependent on one another. This can mean doing a fair amount of resetting between tests[29].

[29] gTest provides some ways of automating this reset process, but its usage involves some C++ we haven't encountered. We'll get there.

```
TEST(updateBalance,deposit){
  // declare and initialize a variable
  double bal(0.0);

  bal = 0.0; //reset
  EXPECT_FLOAT_EQ(0.0,bal); //before
  banking::updateBalance(50.0,bal); //mutate
  EXPECT_FLOAT_EQ(50.0,bal); //after

  bal = 0.0; //reset
  EXPECT_FLOAT_EQ(0.0,bal); //before
  banking::updateBalance(0.0,bal); //mutate
  EXPECT_FLOAT_EQ(0.0,bal); //after

  bal = 0.0; //reset
  EXPECT_FLOAT_EQ(0.0,bal); //before
  banking::updateBalance(125.45,bal); //mutate
  EXPECT_FLOAT_EQ(125.45,bal); //after
}

TEST(updateBalance,withdraw){
  // declare and initialize a variable
  double bal(0.0);

  bal = 100.0; //reset
  EXPECT_FLOAT_EQ(100.0,bal); //before
  banking::updateBalance(-50.0,bal); //mutate
  EXPECT_FLOAT_EQ(50.0,bal); //after

  bal = 100.0; //reset
  EXPECT_FLOAT_EQ(100.0,bal); //before
  banking::updateBalance(0.0,bal); //mutate
  EXPECT_FLOAT_EQ(100.0,bal); //after

  bal = 100.0; //reset
  EXPECT_FLOAT_EQ(0.0,bal); //before
  banking::updateBalance(12.50,bal); //mutate
  EXPECT_FLOAT_EQ(87.50,bal); //after
}
```

Notice that we honored the multiple purposes of the procedure[30] by grouping tests for each purpose together and not just lumping them all together in to one place. Next, you should notice that we not only tested that desired changes occurred, but also when change shouldn't occur[31]. We probably could have gotten away with leaving out the before tests for most of these. However, these tests make the expected behavior of the procedure crystal clear and clarity is always good. So, the time it takes to write these test is worth it.

[30] deposit and withdraw

[31] when the update value is 0.0

Next we move on to the stub.

```
void banking::updateBalance(double upVal, double &balRef){
    // upVal
// balRef

return;
}
```

Here we tossed in the names of our variables as comments so that we remember to start working with them when we get to coding this thing. Otherwise, stubs for void return types are easy enough, just return. In this context you can think of *return* not as a command to send back a value, but to just go back to where the call happened.

We can now compile and run out tests to verify we're setup for development. Once that's done we can go ahead and knock this thing out.

```
void banking::updateBalance(double upVal, double &balRef){
balRef += upVal;
return;
}
```

If we generalize everything we've seen so far, we see the following templates. First the declaration and documentation:

```
namespace LIBNAME{
  /**
   * A mutator procedure for the a state variable
   * @param aVar a reference to the state variable
   * @return none
   * @pre Stuff we're assuming about inputs
   * @post The state variable has changed
   */
  void MUTATOR_NAME(IN_TYPE IN_NAME, ... , VAR_TYPE &ALIAS_NAME, ...);
}
```

Notice that we prefer to list regular, by value, input parameters before the reference parameters, which can act as both input and output.

Next tests:

```
TEST(MUTATOR_NAME,testCase){
  VAR_TYPE VAR_NAME(init_value);

  VAR_NAME = value; //reset
  EXPECT_EQ(value,VAR_NAME); //before
  MUTATOR(VAR_NAME); //mutate
  EXPECT_EQ(new_value,VAR_NAME); //after

  ...
}
```

And then stubs:

```
void LIBNAME::MUTATOR_NAME(VAR_TYPE &ALIAS_NAME){
// ALIAS_NAME
return;
}
```

Access Procedures

Accessor procedures aren't anything new. By using pass-by-value for our these procedures we can treat them exactly the same as our functional procedures. Design them according to the same guidelines we've worked with previously.

Procedures for I/O

Procedures for output provide a new challenge. If something is in memory[32] or the CPU[33] then we can compare it to another value. But we don't have commands to tell the computer to check what it just wrote to some output device. However, we all have two eyes and are very capable of verifying that our procedure did, in fact, produce the expected output. The key, of course, is that *our procedure design process clearly and firmly established an expectation of the output.*[34].

[32] variables
[33] functions

[34] the correct output is not whatever the computer wrote, it's what you intended for it to write

Input procedures are really just mutation procedures in disguise. So, from that perspective, we just design them with the same pattern as we did with MUTATORS. Where they differ is in testing and usage. We'll have to get involved in testing and hit the keyboard a bit. The more subtle difference comes when we consider input procedures for some or all of our model.

Output Procedures

Output procedures are often about displaying computational data in ways that tie it back to the information the user is expecting. From this perspective, they're just like your drawing function from BSL Racket universe programs. There you drew a picture of the model. Here, you'll print out a text-based message about the model. The problem is still the same though: the model is computational data, not real world information. With our banking program, the problem is that the dollar values we're tossing around aren't dollar values, they're *double* values. So, when we need to report back how much interest someone will earn for their balance, we need to report that double value in a way that doesn't make the average user confused.

First things first, let's recognize that computing the interest amount is a MODEL problem and we should just worry about writing a procedure to report some pre-computed value. We'll plug the two pieces together with control code later. This has the added benefit of removing any temptation to pass state variables by reference to an output procedure. Let's get started with the declaration and documentation.

```
namespace bankingUI{

  /**
   * Report the amount of interest earned to the standard output.
   * @param interestAmt is the interest earned as a double
   * @return none
   * @pre interestAmt is a valid interest value (positive)
   * @post none
   */
  void reportInterest(double interestAmt);

}
```

So far so good. Of course, this procedure has no return type, it's meant to cause a write effect to the standard output. Next step in the design process is tests. Here things take a different turn. Recall that the goal of writing tests at this point is about making the expected behavior of the procedure crystal clear in our minds and not about actual testing[35]. What's needed now is for us to decide what the output should look like and document that for ourselves. We can still use gTest as a vehicle for this even though no strict testing will occur.

[35] can't test what you haven't written

```
TEST(reportInterest,all){

   reportInterest(3.0);
   // You'll earn $3.00 in interest.

   reportInterest(0.25);
   // You'll earn $0.25 in interest.

   reportInterest(135.72);
   // You'll earn $135.72 in interest.
}
```

There are not actual tests in this test case. It does, however, call our output procedure three times and give the expected output, as a comment, below each call. In doing this we get to decide exactly what we want from our procedure as well as document it for other programmers to see.[36].

[36] and graders to see that you've thought about what you want this thing to do

Now we stub.

```
void bankingUI::reportInterest(double inerestAmt){
std::cout << "Stub! " << interestAmt << '\n';
return;
}
```

For this stub, I chose to go ahead and print something and include the input value in that something. This is a good reminder that this procedure will write to standard out and that the output written is related to the input.[37] Now it's time to code *reportInterest* out. To do so we'll need to explore some more of *iostream* and the *iomanip* library. The *iomanip* library allows us to manipulate the way in which read and write values.

[37] do you know the exact output produced by that statement?

Double values have decimal precision beyond two places. Dollar values do not. We need to inject some information in to the stream in order to tell the standard output how to write doubles.

```
void bankingUI::reportInterest(double interestAmt){
std::cout << "You’ll earn $" <<
    std::fixed << std::showpoint << std::precision(2) <<
    interestAmt << " in interest\n";
return;
}
```

The manipulators *std::fixed* and *std::showpoint* actually come from *iostream*. The former tells the stream that floating point values should be treated as fixed point values. Combined with *std::precision(2)* from *iomanip*, we get exactly two places after the decimal point. The manipulator *std::showpoint* guarantees that the decimal place is always show, even when there isn’t one. If we leave out the *std::fixed*, then *std::precision(2)* tells the stream to deal with only two digits total[38]. Leaving out *std::showpoint* could mean that values like 3.0 show up as 3 instead.

[38] before and after the decimal place

There are other useful I/O manipulators out there, we’ll check them out as needed. If you just can’t wait until then check out the documentation for *iostream* and *iomanip*.[39].

[39] look for things like setfill and setw

Input Procedures

Input procedures are just mutation procedures in disguise. Recall that input require a variable. So, if you’re getting input then you’re mutating a variable. The only difference is the mutation mechanism. Let’s look at a procedure to initialize the balance for our banking problem.

At first glance, you might imagine passing the balance state variable directly to our initializer procedure. This makes sense, if you completely trust your user to honor your PRE-CONDITIONS about your model’s state. What if they enter a negative balance? What if they decide to type their name instead of a dollar figure? What if they attempt some kind of code-injection attack on your software?[40]? So what is a poor, paranoid programmer to do?

[40] `http://en.wikipedia.org/wiki/Code_injection`

The answer is temporary state. First we get user input for the initial balance and stick it in a variable for safe-keeping. This is the real purpose of the input function. We can then use some CONTROLLER procedures to error check and *validate* the input. So, let’s proceed with our super basic input function.

```
namespace bankingUI{
  /**
   * Get an initial balance value from the user.
   * @param iValRef a reference to a double variable for user's value
   * @return none
   * @pre user types a number
   * @post variable referenced by iValRef contains unchecked user input
   */
void getUserBal(double &iValRef);
}
```

Notice that I made a point to mention that this procedure results in unchecked user input. We're also going to work on the assumption that the user type a single numerical value and not get too detailed in our input checking and validation just yet. From here on out, it's just like designing a MUTATOR. Testing, however, will require you to press some keys. So we can setup before/after tests like we did with our mutator, but will need to mark as a comment what you're planning to press. Here's one example, I'll let you imagine more.

```
TEST(getUserBal,all){
 double userIn(0.0);

 userIn = 0.0;
 EXPECT_FLOAT_EQ(0.0,userIn);
 getUserBal(userIn); // Type 350.0
 EXPECT_FLOAT_EQ(350.0,userIn);

}
```

Next we stub.

```
void bankingUI::getUserBal(double &iValRef){
//iValRef
return;
}
```

Not much new here. Do a simple return and at least comment out the variable name as a reminder for later. At this point we can compile and run our tests to verify that we're ready to roll with development. Once we've cleaned out errors and warnings, we move on to the coding stage:

```
void bankingUI::getUserBal(double &iValRef){
std::cout << "Please enter the initial value: ";
std::cin >> iValRef;
return;
}
```

Hey now! There's output in that procedure. True. If you expect the user to give you input, you should PROMPT them for it. We could write an output procedure for this, but that might be a bit too far. The point wasn't to only do input operations, it was to *carry out a user input task*, which should involve prompting as well.

Recap

We covered a lot of things in these notes. Let's break the big picture stuff down.

1. Procedure Design Process for Effect-Based Procedures
   (a) State Mutator Procedures
   (b) Output Procedures
   (c) Input Procedures
2. Programming Concepts
   (a) L-VALUES and R-VALUES
   (b) PASS-BY-VALUE and PASS-BY-REFERENCE procedures
   (c) MODEL-VIEW-CONTROLLER architectures
   (d) DATA STREAMS
3. C++
   (a) I/O Streams *std::cout*, *std::cerr*, and *std::cerr*
   (b) Streaming I/O operators << and >>
   (c) Stream manipulators *std::fixed*, *std::setprecision(n)*, and *std::showpoint*
   (d) Variable declaration and initialization.
   (e) The assignment operator =