Function Documentation Resources: Assertions and Dataflow comments

Writing Assertions as Function Documentation

Although this topic is not often emphasized, in that it is sometimes explicitly covered in textbook chapter content and in other instances implied through the source code examples or in the Appendix, here is a bit of addtional information that may prove helpful while you begin exploring and implementing the concept of functional decomposition in your programs. The point of all this is .. how to properly document your functions.

When talking about functions, we can say informally that every C++ function has preconditions and post conditions as well as pass-by-value or pass-by-reference data parameters in many cases. From now on how about attempting to include **preconditions** and **postconditions** as well as **data flow** comments to document function interfaces.

Here's an example:

```
#include <iostream>
using namespace std;

void displayAverage(/*in*/float, /*in*/int) ; //function prototype indicates the position of two datatypes as value parameters
 and the direction of dataflow

int main()
{
  float someFloat = 0.0;
  int someInt =0;

  someFloat = 72.50; //assuming some input updates the value for this variable
  someInt = 10;        //assuming some input updates this counte type variable

   //here is a sample function call ...main is the calling function aka. as the caller
   displayAverage(someFloat, someInt); //from within int main() ...here is the function call with actual arguments...notice no
 data flow comments are needed here

    //other program code here....

    return 0;
}

//and here is the function definition

//************************************************************************************************************************
// This function computes and displays the average given two incoming value parameters of type float and int   *
// See Gaddis textbook pgs. 318-319 section on Passing Data by Value                                             *
// Note this function header comment is used in the Gaddis textook source code samples                           *
//************************************************************************************************************************
void displayAverage(/*in*/float total, /*in*/int count) //notice the data flow comments in the formal parameter list indicate
 the intended direction of the dataflow
// Pre:      total has been assigned and count is greater that 0
// Post:     The average has been output on one line
{
    cout <<"Average is "<< total / static_cast<float>(count) <<endl;
}
```

The precondition is an assertion describing everything that the function requires to be true at the moment when the caller invokes the function. The postcondition describes the state of the program at the moment when the function finishes executing.

In the preceding example, the precondition warns the calling function to make sure that total has been assigned a meaningful value and that count is positive. If this precondition is true, the function guarantees it will satisfy the postcondition. If count isn't positive when displayAverage is invoked or called, the effect of the module is undefined. (For example, if count equals 0, the postcondition surely isn't satisfied-any code that implements this module crashes). YIKES!

Sometimes the calling function doesn't need to satisfy any precondition before calling a function. In this case, the precondition can be written as the value true or simply omitted. In the following example, no precondition is necessary:

Here's another example:

```
#include <iostream>
using namespace std;

void getDimensions(/*out*/ int&, /*out*/ int&, /*out*/ int&); //function prototype indicates the position of three datatypes as
reference parameters and the direction of dataflow

int main()
{
    //three local variables declared and initialized

   int firstValue = 0;
   int secondValue = 0;
   int thirdValue = 0;

    //here is a sample function call ...main is the calling function aka. as the caller
    getDimensions(firstValue, secondValue, thirdValue); //from within int main() ...here is the function call with actual
arguments...notice no data flow comments are needed here

    //output statements below used to determine if the firstValue, secondValue and thirdValue changed after the previous
call to the getDimensions function.
    cout<<firstValue <<endl;
    cout<<secondValue<<endl;
    cout<<thirdValue<<endl;

    return 0;
}

//and here is the function definition

//******************************************************************************************************************************
// This function provides an input prompt and gets user input of three integer values which is then immediately passed
through *
// from these three reference variables and written to the memory address of the related actual argument or variable in
the          *
// calling code. See Gaddis textbook pgs. 350-354 section on Using Reference Variables as Parameters
                                  *
// Note this function header comment is used in the Gaddis textbook source code
samples                                              *
//******************************************************************************************************************************
void getDimensions(/*out*/ int& height, /*out*/ int& width, /*out*/ int& stages) //notice the data flow comments in the formal
 parameter list indicate the intended direction of the dataflow
//Post:    User has been prompted to enter three integers
//          height is the first input value
//          width is the second input value
//          stages is the third input value
//Include comments related to any data validation routines
{
    cout<< "Please enter three values (whole numbers) to represent height, width and stages of the rocket generator:"
<<endl;
    cin >>height>>width>>stages;
    }
```

Ada Lovelace? ….and a bit more on dataflow comments

Choosing between Passing by Value and Passing By Reference

Choosing between the two argument-passing mechanisms is one of the most persistent difficulties for programming students, often lasting well into the CS10 course (and, unfortunately, even beyond).

It is crucial that you see the choice not as a language issue but as a design decision. In designing the function interface, each parameter must be classified as an In, Out, or Inout parameter according to the data flow. Once this classification has occurred, the choice of mechanism is trivial —- each In parameter is passed by value, and each Out and Inout parameter is passed by reference.

Requiring each parameter to be labeled with the comment /* in */, /* out */, or /* inout */ (a useful and concise terminology borrowed from the Ada language) in your function's formal parameter list will help you in thinking carefully about the data flow.

For a given parameter, then, determining the direction of data flow is ultimately the hard part. Here is a thumbnail guide you might consider saving in your C++ toolkit as you work with functions:

Use of the Parameter	Data Flow
Inspected only	In
Modified only	Out
Inspected and modified	Inout

To give you practice in identifying the data flow of parameters, take a look at this fill-in-the-blank function definition:

```
void DoThis( _________ int__ alpha,_________ float__ beta )
{
beta = 3.8 * float(alpha);
}
```

For each parameter, do the following in the order shown:
1.Examine how the parameter is used within the function.
2.Fill in the first blank with /* in */, /* out */, or /* inout */.
3.Fill in the second blank with either an ampersand (&) or nothing.

For the example above, we could explain that alpha is inspected only, so it's an In parameter and there should be no ampersand after int (a pass by value). The parameter beta is modified only, so it's an out parameter and there should be an ampersand after float (a pass by reference).

As noted in the syllabus ...Throughout this course, we encourage and suggest the use of good programming style and documentation. In addition to programming style elements introduced in Chapter#2 (Gaddis textbook) and demonstrated in the numerous source code examples throughout the text, your instructor recommends that we also implement the examples of programming style, formatting, and documentation provided in one of the Appendices of Programming and Problem Solving with C++ by Dale and Weems (a reference and prior textbook used in this course) **- see Appendix F : Programming Style, Formatting and Documentation on pg.1003 (Dale & Weems).** (1.54MB)

Let's also remember Ada's contribution ...and not forget to include those data flow comments in your function parameter lists in all your upcoming programming project....thanks..sujan!

MATTERS OF STYLE - Other Function Documentation examples included here for your reference....

Preconditions and postconditions, when well written, will provide a concise but accurate description of the behavior of a function. A person reading your function should be able to see at a glance how to use the function simply by looking at its interface (the heading and the precondtion and postcondition). The reader should never have to look into the function body to understand its purpose or use.

A functon interface describes what the function does, not the details of how it works its algorithms. For this reason, the postcondition should mention (byname) each outgoing parameter and its value but should not mention any local variables. Local variables are implementation details; they are irrelevant to the module's interface.

In some textbooks (for example Gaddis, Starting out with C++ series and Dale & Weems, Programming and Problem Solving with C++,) pre/post conditions are informally written as comments, and often implied in the function header. However some programmers use a very formal notation to express them. For example get3Ints might be documented as follows:

```
//Postcondition:
//    User has been prompted to enter three integers
//    && height == first input value
//    && width == second input value
//    && stages == third input value
```

Some programmers place comments next to the parameters to explain how each parameter is used and use embedded comments to indicate which of the data flow categories each parameter belongs to.

```
void display (/*In*/        string  word,         // Word to be displayed
              /*In/out*    int wordCounter    // Number of words displayed so far
```

To write a postcondition that refers to parameter values that existed at the moment the function was called, you can (as some programmers do) attach the @entry to the end of the variable name. Below is an example of the use of this notation. The changeValues function exchanges the contents of its two parameters.

```
void changeValues (/*In/out*/ int& firstIntValue,
                   /*In/out*/ int& secondIntValue)
//Precondition:
//       firstIntValue and secondIntValue are assigned
//Postcondition:
//       firstIntValue == secondIntValue@entry
// &&  secondIntValue == firstIntValue@entry
{
    int temporaryValue;

    temporaryValue = firstIntValue;
    firstIntValue = secondIntValue;
    secondIntValue = temporaryValue;
}
```

*note this information sheet includes partial source code and content from the following academic textbooks adopted for CS10 course, Gaddis -Starting out with C++: From Control Structures to Objects (7E) as well as Dale and Weems - Programming and Problem Solving with C++ (5E).