

```
//////////////////////////////////// isLeapYear()  
// Determines whether a given year is a leap year.  
//  
// Parameters:  
//     Year    year to be tested  
//  
// Returns:   true if Year is a leap year, false otherwise  
//  
// Pre:      Year has been initialized  
// Post:    specified value has been returned  
//  
// Called by: buildCalendar()  
// Calls:   none  
//  
bool isLeapYear (int Year) {  
    return ( ( Year % 400 == 0 ) ||  
            ( ( Year % 4 == 0 ) &&  
              ( Year % 100 != 0 ) ) );  
} // end isLeapYear
```

If a function needs to communicate just one value to the caller you may accomplish that in either of two ways:

- use a `void` function with a reference parameter for communication
- use a typed function with an appropriate `return` statement

The latter approach is generally preferred because it makes the effect of the function call clearer.

Remember that in C++ there is no indication in the function call of which actual parameters are passed by reference (and hence at risk of being modified) and which are passed by value or constant reference (and hence guaranteed not to be modified by the function call).

```
void SquareIt(double Value, double& Square) {  
    Square = Value * Value;  
}
```

... could be invoked in the following manner:

```
cin >> xval;  
SquareIt(xval, xsquared);    // call is a separate statement  
cout << setw(10) << setprecision(4) << xsquared;
```

```
double SquareIt(double Value) {  
    double Square = Value * Value;  
    return Square;  
}
```

... could be invoked in the following manner:

```
cin >> xval;  
xsquared = SquareIt(xval); // called as part of an expression  
cout << setw(10) << setprecision(4) << xsquared;  
                                     // . . . or even as:  
cout << setw(10) << setprecision(4) << SquareIt(xval);
```

If a function needs to communicate more than one value to the caller you must use reference parameters to accomplish the communication (pending the introduction of structured variables).

```
void readItem(istream& In, string& SKU, int& Units, int& Dollars,
              int& Cents) {

    getline(In, SKU, ':');
    In >> Units;
    In >> Dollars;
    In.ignore(INT_MAX, '.');
    In >> Cents;
    In.ignore(INT_MAX, '\n');

    return;
}
```

Note: stream variables must always be passed by reference. In fact, the Standard mandates that stream variables cannot be copied!

The placement of the declaration (prototype) of a function determines the scope of the function name, just as with other identifiers.

Placing the declaration outside all function bodies gives the function name file scope. This allows any function defined later (after the declaration) in the same file to invoke that function.

Placing the declaration inside the body of another function gives the function name scope local to the compound statement in which the declaration is placed. This can be used to restrict which functions are allowed to invoke that function.

The declaration for a function may occur more than once.

The definition (implementation) of a function can occur only one time.

Consider designing a function to determine if three given integer values are potentially the sides of a valid triangle.

Geometry review: there exists a triangle with sides A, B and C if the sum of any two sides is larger than the third side.

Initial Considerations

The decision can be made entirely on the basis of the three values that are given. The function does not require any additional information, so it will not be provided with any.

We will delegate all input and output operations to the client code. The function will take the three integer values as parameters.

The function will return a `bool` value to indicate the result of the test. This could be implemented via a reference parameter, but it is cleaner to use a typed function.

The test and interface seem clear enough... here's an implementation:

```
bool validTriangle(int A, int B, int C) {  
    return (A + B > C);  
}
```

Function Testing

The function implementation must be tested. The specification gave no indication of how the input will be supplied to the program, nor how the function will be used. However we may still test the function by supplying a "driver" and suitable test data.

The "driver" will read a sequence of data sets from an input file and generate a report file summarizing the results obtained from the function implementation.

The report must be examined by a human to determine whether any cases were handled incorrectly.

The driver is straightforward because the function interface is simple and the function performs a single, constrained task:

```
. . .
int main() {
    . . .
    int A, B, C;
    In >> A >> B >> C;
    In.ignore(INT_MAX, '\n');
    while (In) {
        cout << setw(5) << A
             << setw(5) << B
             << setw(5) << C
             << boolalpha
             << "      "
             << validTriangle(A, B, C) << endl;
        In >> A >> B >> C;
        In.ignore(INT_MAX, '\n');
    }
    . . .
}
```

Of course, this is of no use without test data.

Designing good test data is one of the hardest tasks a developer will face.

Usually the logic of the problem will imply a number of cases; these may be distinct or they may overlap.

A test designer must identify these cases and create data corresponding to them.

The test designer must also determine what the correct results would be, usually by hand, since the implementation being tested obviously cannot be used to determine correctness.

Case Identification

Obviously there are two main cases here: valid triangles and invalid triangles.

Obviously there are an infinite number of possible test cases; we cannot try all of them. This leads to the Fundamental Rules of Testing:

No amount of testing can prove an implementation is entirely correct.

The goal of testing is to discover flaws, not to verify correctness.

Despite the first rule, do not conclude testing is unimportant or pointless.

In view of the second, remember your goal is to "break" the implementation being tested.

Valid triangles can be classified as acute or right or obtuse. It's not clear that those distinctions are relevant here, but it wouldn't hurt to be sure that all are included in the test data.

Invalid triangles will have one side that's "too long" for the other two.

3	6	7	acute
5	12	13	right
5	12	15	obtuse

3	6	10	bad
6	10	3	bad
10	3	6	bad
3	5	8	boundary case
5	8	3	boundary case
8	3	5	boundary case

The test design has identified a basic issue the function designer missed. What?

It has also missed a basic issue also missed by the function designer. What is that?

When the driver and function are executed on the given test data the results are correct for the given valid triangle data. However, the results are incorrect for several of the invalid data cases:

3	6	10	false
6	10	3	true
10	3	6	true
3	5	8	false
5	8	3	true
8	3	5	true

By examining the test results we can determine the responsible logical flaw in the current design. Knowing that, we can revise the design to eliminate the flaw.

Here, it is obvious the test designer considered the effect of permuting (reordering) the input values and the developer did not. This sort of oversight during program design is all too common and often not so easily detected or repaired.

If the test designer also overlooked this issue then the flaw would have slipped through.

In view of the test results the function must be redesigned. There are at least two sensible approaches. We could first determine the maximum of the three values and then test whether the sum of the other two exceeds the maximum. We could consider all three possible tests, not just whether $A + B > C$.

```
bool validTriangle(int A, int B, int C) {  
    return ( (A + B > C) &&  
            (A + C > B) &&  
            (B + C > A) );  
}
```

The revision fixes the detected problem.

What problems remain?

The revised function produces the following output on the test data:

3	6	7	true
5	12	13	true
5	12	15	true
3	6	10	false
6	10	3	false
10	3	6	false
3	5	8	false
5	8	3	false
8	3	5	false

Note well:

- You cannot simply assume that the revision corrected the problem --- retest!
- You cannot simply assume that the revision has not introduced new problems. You must retest on ALL of the test data.
- The amount of test data used here is certainly inadequate.