Elegance & Simplicity

Scenario

Quinn was asked to write a function to check an array of floating point numbers to make sure none are zero. The function will be used to check a column vector of coefficients in a scientific computation, which would fail if any of the coefficients were zero.

The details of the function that Quinn needed to implement were

- The function would be passed the size of the array, and the array itself
- The function should return true if the array is okay, and false otherwise

A First Try

```
bool isAllNonzero (const int lengthOfColumn, double columnToCheck[])
{
    bool failedZerosCheck;                  // Variables used by this function
    int arrayPosition, lastElement;         // (a flag and two array indexes).

    if (lengthOfColumn == 0) {              // Optimization for the special case of
        return true;                        // an empty array.
    }

    lastElement = lengthOfColumn - 1;       // We'll stop when we reach the last
                                            // element of the array.

    if (columnToCheck[0] == 0.0) {          // Initialize flag to correct value.
        failedZerosCheck = true;
    } else {
        failedZerosCheck = false;
    }

    arrayPosition = 0;                      // Iterate through the array, note that
    while (failedZerosCheck != true         // we exit early if the flag is set.
            && arrayPosition != lastElement) {
        arrayPosition = arrayPosition + 1;
        if (columnToCheck[arrayPosition] == 0.0) {
            failedZerosCheck = true;        // Set the flag if we find a zero
        }                                   // element.
    }

    if (failedZerosCheck == false) {
        return true;                        // If we didn't fail, return 'okay'.
    } else {
        return false;                       // If we did fail, return 'failure'.
    }
}
```

Quinn believes that the code on the preceding page meets all the requirements for code in CS 70. Not only does the function seem to work, but it also uses descriptive variable names and is well commented. But Malory, a CS 70 grader, tells Quinn that the code is overcomplicated and needs to be simplified.

Revision 1

Quinn realizes that some of the **if** statements in the code are redundant, and can be replaced with simpler code. Specifically, the statements failedZerosCheck != true and failedZerosCheck == false can be rewritten as !failedZerosCheck, and similarly, the form

```
if (boolean-expression) {
    return true;
} else {
    return false;
}
```

can be rewritten as

```
return boolean-expression;
```

Thus the code becomes...

```
bool isAllNonzero (const int lengthOfColumn, double columnToCheck[])
{
    bool failedZerosCheck;                  // Variables used by this function
    int arrayPosition, lastElement;         // (a flag and two array indexes).

    if (lengthOfColumn == 0) {              // Optimization for the special case of
        return true;                        // an empty array.
    }

    lastElement = lengthOfColumn - 1;       // We'll stop when we reach the last
                                            // element of the array.

    failedZerosCheck = (columnToCheck[0] == 0.0);
                                            // Initialize flag to correct value.

    arrayPosition = 0;                      // Iterate through the array, note that
    while (!failedZerosCheck                // we exit early if the flag is set.
            && arrayPosition != lastElement) {
        arrayPosition = arrayPosition + 1;
        if (columnToCheck[arrayPosition] == 0.0) {
            failedZerosCheck = true;        // Set the flag if we find a zero
        }                                   // element.
    }

    return !failedZerosCheck;               // If we didn't fail, return 'okay'.
                                            // otherwise return 'failure'.
}
```

Revision 2

Quinn also notices that the comment next to the variable declarations doesn't say much, and that in any case the declarations can be moved to the point at which the variables are initialized. Quinn also observes that failedZerosCheck is always used in its negated form and so replaces it with an opposite flag.

```
bool isAllNonzero (const int lengthOfColumn, double columnToCheck[])
{
    if (lengthOfColumn == 0) {                  // Optimization for the special case
        return true;                            // of an empty array.
    }

    int lastElement = lengthOfColumn - 1;       // We'll stop when we reach the last
                                                // element of the array.

    bool isOkay = (columnToCheck[0] != 0.0);    // Initialize flag to correct value.

    int arrayPosition = 0;                      // Iterate through the array, note that
    while (isOkay                               // we exit early if the flag is set.
            && arrayPosition != lastElement) {
        arrayPosition = arrayPosition + 1;
        if (columnToCheck[arrayPosition] == 0.0) {
            isOkay = false;                     // If we find a zero, it's not okay.
        }
    }

    return isOkay;
}
```

Notice that Quinn ended up removing the comment from the final **return** statement. The meaning is now entirely obvious from the code.

Revision 3

Quinn realizes that the **while** loop is a somewhat hard to follow, because it performs a loop increment at the start of the loop, rather than at the end (the loop is technically correct but it doesn't match any familiar pattern). A small adjustment to the code makes the loop more conventional and allows Quinn to eliminate the lastElement variable.

Quinn also decides that some of the variable names are overly long and aren't adding much to the meaning of the program, and decides to shorten them (but without losing any meaning).

```
bool isAllNonzero (const int length, double column[])
{
    if (length == 0) {                      // Optimization for the special case
        return true;                        // of an empty array.
    }

    bool isOkay = (column[0] != 0.0);       // Initialize flag to correct value.

    int index = 1;                          // Iterate through the array, note that
    while (isOkay && index != length) {     // we exit early if column isn't okay.
        if (column[index] == 0.0) {
            isOkay = false;                 // If we find a zero, it's not okay.
        }
        index = index + 1;
    }

    return isOkay;
}
```

It's also worth noting that Quinn's first comment is somewhat inaccurate. The test a for zero-length column isn't just "an optimization"—it's a necessity; without it the function would have undefined behaviour for an empty column (because the next line tries to read the first value in the column).

Revision 4

There isn't really any need to set the initial value of the isOkay flag by testing the first element of the array. Quinn decides to let the loop handle it, by having it start at the first element rather than the second.

```
bool isAllNonzero (const int length, double column[])
{
    if (length == 0) {                      // Optimization for the special case
        return true;                        // of an empty array.
    }

    bool isOkay = true;                     // Everything is fine until we find
                                            // otherwise.

    int index = 0;                          // Iterate through the array, note that
    while (isOkay && index != length) {     // we exit early if column isn't okay.
        if (column[index] == 0.0) {
            isOkay = false;                 // If we find a zero, it's not okay.
        }
        index = index + 1;
    }

    return isOkay;
}
```

This change also means that the initial **if** is now redundant. Although the program would be correct if this test was left in, there is no reason to suppose that this special-case code would make the function "more efficient". As unnecessary code, it should be removed.

```
bool isAllNonzero (const int length, double column[])
{
    bool isOkay = true;                     // Everything is fine until we find
                                            // otherwise.

    int index = 0;                          // Iterate through the array, note that
    while (isOkay && index != length) {     // we exit early if column isn't okay.
        if (column[index] == 0.0) {
            isOkay = false;                 // If we find a zero, it's not okay.
        }
        index = index + 1;
    }

    return isOkay;
}
```

Quinn could stop at this point—the code is elegant enough for full credit under CS 70's style rules.

Revision 5

While removing the "**return** true;" code on the previous page, Quinn realizes that there is a way to eliminate the isOkay variable—you can return immediately if a zero is found.

```
bool isAllNonzero (const int length, double column[])
{
    int index = 0;                          // Iterate through the array...
    while (index != length) {
        if (column[index] == 0.0) {
            return false;                   // If we find a zero, it's not okay.
        }
        index = index + 1;
    }

    return true;                            // Otherwise, it's all good!
}
```

The above code can also be simplified further because the **while** loop can now be replaced by an idiomatic (i.e., standard) **for** loop.

```
bool isAllNonzero (const int length, double column[])
{
    for (int i = 0; i < length; ++i) {
        if (column[i] == 0.0) {
            return false;                   // If we find a zero, it's not okay.
        }
    }

    return true;                            // Otherwise, it's all good!
}
```

As a final piece of showing off, Quinn applies the rule that the braces for loops and conditionals are optional when the block consists of a single statement. Thus Quinn can write:

```
bool isAllNonzero (const int length, double column[])
{
    for (int i = 0; i < length; ++i)
        if (column[i] == 0.0)
            return false;                   // If we find a zero, it's not okay.

    return true;                            // Otherwise, it's all good!
}
```

(Whether this version is more elegant than the one above is a matter of opinion.)

Lessons to Take Away

For this specific example, some students would have written the short and simple loop at the outset. But at some point in CS 70, many students will be in the situation of having written code that is more complex than it needs to be.

Complex code is harder to reason about, harder to debug, and requires more comments. For that reason, experienced programmers develop a sense of resistance to program complexity—it feels wrong. As a function gets longer, they start looking for ways to make it shorter. Possible techniques include

- Looking for redundancy, and eliminating it or factoring it out
- Trying to avoid conditional code
- Trying to use familiar patterns whenever possible

Finally, *do not* spend time worrying about low-level efficiency issues. For example, some students may think that the code at the bottom of page 5 is inefficient compared to the final version because it uses a flag variable. In fact, our compiler, g++, produces *identical* output code for both this code and the final four-line version when optimization is turned on. Don't second guess the compiler, just write the most elegant code you can.