Think Like a Programmer

Errata, Clarifications, and Updates
Last Updated 2/21/2017

This lists errata and clarifications going back to the first printing; almost all of these issues are already corrected in the latest printing. The most recent errata are marked [NEW].

Chapter 1

[NEW] **Page 12.** In part 3 of Figure 1-12, the 7 is missing from the centermost square.

Chapter 2

Page 32. Second paragraph, "starting with each digit to the right of the check digit," should be "each digit to the left of the check digit."

Page 34. In first code listing, *doubledDigit > 10* should be *doubledDigit >= 10*.

Page 38. In the code listing, *else checksum += 2 * (digit - '0')* should be
else checksum += doubledDigitValue(digit - '0')

Page 41. In the third paragraph of the "Decode a Message" problem description, "...would yield the letter b because 57 modulo 27 is 2...", the 57 should be 56.

Pages 46, 47, and 48. In these listings, the output labels should be "Number entered" and not "Numbered entered."

Pages 53-54. Problems 2.1 through 2.3 require a third output statement, one that outputs a space (i.e., *cout << " ";*).

Change the text of 2.1 to:

> Using only single-character output statements that output a hash mark, a space, or an end-of-line, write a program that produces the following shape:

The shape itself, and questions 2.2 and 2.3, can then remain as they are.

Chapter 3

Page 60. Last two sentences of the paragraph that starts the page should dereference the variables intA and intB. "If **intA* > **intB*, for example, we want to return a positive number, and **intA* – **intB* will be positive if **intA* > **intB*. Likewise, **intA* – **intB* will be negative if **intB* > **intA* and will be zero when the two integers are equal."

Page 66. The sentence, "Therefore, we initialize it to 0 and not the value in *location[0]*." The word "location" is formatted as though it was the name of the array. What is meant is "in location [0]", that is, the location [0] in the *histogram* array.

Page 71. The first paragraph refers to the code on the previous page, but describes it as finding the record with the grade closest to the average, when it's just finding the record with the highest grade. It should read:

> Here, the variable *highPosition* (1) takes the place of highest. Because we aren't directly tracking the highest grade, when it's time to compare the highest grade against the current grade, we use *highPosition* as a reference into *studentArray* (2). If the grade in the current array position is higher, the current position in our processing loop is assigned to *highPosition* (3). Once the loop is over, we can access the name of the student with the highest grade using *studentArray[highPosition].name*, and we can also access any other data related to that student record.

Page 72. Third non-code paragraph, *numAgents* should be *NUM_AGENTS*. (

Page 72-73. In *arrayAverage* function, 0.5 should not be added to sum before dividing by *ARRAY_SIZE*. In the code that uses the function, *highestAverage* and *agentAverage* should be *double* instead of *int*.

The code at the bottom of page 72 should be (change highlighted):

```
double arrayAverage(int intArray[], int ARRAY_SIZE) {
    double sum = 0;
    for (int i = 0; i < ARRAY_SIZE; i++) {
        sum += intArray[i];
    }
    double average = sum / ARRAY_SIZE;
    return average;
}
```

The code at the top of page 73 should be:

```
double highestAverage = ❶arrayAverage(sales[0], 12);
for (int agent = 1; agent < NUM_AGENTS; agent++) {
    double agentAverage = ❷arrayAverage(sales[agent], 12);
    if (agentAverage > highestAverage)
        highestAverage = agentAverage;
```

```
}
cout << "Highest monthly average: " << highestAverage << "\n";
```

Page 77. First full paragraph, "Furthermore, reading all of the grades into the vector..." should be "reading all of the survey responses into the vector..."

Chapter 4

Page 89. The paragraph that begins at the bottom of the page is missing two of the references to the preceding code block. The first part of the paragraph should read:

> This code has a global variable (1) which in most cases is bad style, but here I need a value that persists throughout all of the recursive calls. As this variable is declared outside of the function, no memory is allocated for it in the function's activation record, nor are there any other local variables or parameters. All the function does is increment *count* (2) and make a recursive call (3). Recursion is discussed extensively in Chapter 6 but is used here simply to make the chain of function calls as long as possible. The activation record of a function remains on the stack until that function ends. So when the first call is made to *stackOverflow* from *main* (4), an activation record is placed on the runtime stack that cannot be removed until that first function call ends.

Page 94. Last sentence of first paragraph, "Including the allocated memory..." should be "Including the deallocated memory..."

Page 101. "Tracking an Unknown Quantity of Student Records" problem description. The addRecord description should say; "This function takes a pointer to a collection of student records, a student number, and a grade, and it adds a new record with this data to the collection."

Page 102. Second paragraph, "Again, *studentCollection* is synonymous with *node* *." This should read "...with *listNode* *."

Chapter 5

Page 124. There's a mismatch between the code and the description. Change the sentence in parentheses in the second full paragraph on page 124, from:

> (You might wonder how, if we're validating grades as they are assigned, we could ever have an invalid grade, but remember that our default constructor assigns –1 to signal that no legitimate grade has been assigned yet.)

to

(You might wonder how, if we're validating grades as they are assigned, we could ever have an invalid grade, but we might decide to assign an invalid grade in the constructor to signal that no legitimate grade has been assigned yet.)

Page 127. In first paragraph, reference to "*studentRecord struct* type" should be "*studentRecord* class." I.e., *studentCollection* data member *studentData* is an instance of the *studentRecord* class.

Page 128. Third paragraph begins, "Now we can turn our attention to the last of the three member functions, *recordWithNumber*." This is the second function in the list, not the third.

Page 140. In sentence continued at the top of the page: "of student records, we're still responsible for deleting the nodes in the list when we we're through with them," the word "we" should be deleted.

Chapter 6

Page 156. In the declaration of the "bad example" function with too many parameters, the function should be named *arraySumRecursiveExtraParameters* to match the recursive call and the call in the code block later on the page.

Page 157. The *zeroCountIterative* function should not declare the local variable *sum*.

That line in the code should be deleted:

```
int zeroCountIterative(int numbers[], int size) {
    int sum = 0;
    ❶int count = 0;
    for (int i = 0; i < size; i++) {
       if (numbers[i] == 0) count ++;
    }
    return count;
}
```

Page 158. In the second code listing, the declaration of the *struct listNode* is mislabeled *listNnode*.

Page 163. *treePtr* in code listing should be a pointer to *binaryTreeNode*. See corrections for 163-166 below. (Style issue only).

Page 165. In the first sentence, "...private data member *root*," should be "...private data member *_root*." In the next paragraph, "It calls *privateCountLeaves*, passing the private

data member *root*," there's the same issue--should be *_root*. See corrections for 163-166 below.

Page 166. This is a style issue only--the stack-based leaf-counting function should declare nodes as *stack<treePtr>* instead of *stack<binaryTreeNode *>*. See corrections for 163-166 below.

Pages 163-166. The "countLeaves" methods should be given a "noun" name (like "leafCount") to match prior naming conventions.

Page 163, code listing. Changes in yellow.

```
class binaryTree {
    public:
        ❶int leafCount();
    private:
        struct binaryTreeNode {
            int data;
            binaryTreeNode * left;
            binaryTreeNode * right;
        };
        typedef binaryTreeNode * treePtr;
        treePtr _root;
};
```

and

```
int numLeaves = bt.leafCount();
```

Page 164, code listing.

```
struct binaryTreeNode {
    int data;
    binaryTreeNode * left;
    binaryTreeNode * right;
};
typedef binaryTreeNode * treePtr;
int leafCount(treePtr rootPtr) {
    if (rootPtr == NULL) return 0;
    if (rootPtr->right == NULL && rootPtr->left == NULL)
```

```
        return 1;
    int leftCount = leafCount(rootPtr->left);
    int rightCount = leafCount(rootPtr->right);
    return leftCount + rightCount;
}
```

Page 165, code listing.

```
class binaryTree {
    public:
        int publicLeafCount();
    private:
        struct binaryTreeNode {
            int data;
            binaryTreeNode * left;
            binaryTreeNode * right;
        };
        typedef binaryTreeNode * treePtr;
        treePtr _root;
        int privateLeafCount(treePtr rootPtr);
};
❶int binaryTree::privateLeafCount(treePtr rootPtr) {
    if (rootPtr == NULL) return 0;
    if (rootPtr->right == NULL && rootPtr->left == NULL)
        return 1;
    int leftCount = privateLeafCount(rootPtr->left);
    int rightCount = privateLeafCount(rootPtr->right);
    return leftCount + rightCount;
}
❷int binaryTree::publicLeafCount() {
    ❸return privateLeafCount(_root);
}
```

Paragraph that follows:

> Although C++ would allow both functions to have the same name, for clarity, I've used different names to distinguish between the public and private "leaf

count" functions. The code in `privateLeafCount` ❶ is exactly the same as our previous, independent function `leafCount`. The wrapper function `publicLeafCount` is simple. It calls `privateLeafCount`, passing the private data member `root`, and returns the result ❸. In essence, it "primes the pump" of the recursive process. Wrapper functions are very helpful when writing recursive functions inside classes, but they can be used anytime a mismatch exists between the parameter list required by a function and the desired parameter list of a caller.

Page 167. The sentence reading "Second, look how many function calls *stackBasedCountLeaves* makes--for each visit to an interior node (i.e., not a leaf), this function makes four function calls: one each to *empty* and *top*, and two to *push*" is imprecise. It should say, "Second, look how many function calls *stackBasedCountLeaves* makes--for each visit to an interior node (i.e., not a leaf), this function makes up to five function calls: one each to *empty*, *top*, and *pop*, and one or two to *push*."

Chapter 7

Page 178. In the first sentence in the second paragraph, *lowerStudent* should be *lowerStudentNumber*.

Page 183. In the paragraph that continues at the top of the page, the sentence that begins "If *integerList* is a *list<int>*..." should begin "if *intList*..."

Page 190. End of first full paragraph. In the sentence beginning "If *sra* is an array containing *arraysize* objects" it should read "*arraySize*."

Page 192. Note: the secondary array *sortArray* must be declared dynamically if *arraySize* isn't a *const*:

```
❶studentRecord * sortArray = new studentRecord[arraySize];
❷int sortArrayCount = 0;
for (int i = 0; i < arraySize; i++) {
   ❸if (sra[i].grade() != -1) {
      sortArray[sortArrayCount] = sra[i];
      sortArrayCount++;
   }
}
❹qsort(sortArray, sortArrayCount, sizeof(studentRecord), compareStudentRecord);
❺sortArrayCount = 0;
```

```
❻for (int i = 0; i < arraySize; i++) {
   ❼if (sra[i].grade() != -1) {
      sra[i] = sortArray[sortArrayCount];
      sortArrayCount++;
   }
}
```

Chapter 8

Pages 213-214. These functions make use of a *const_iterator* object (*iter*) which is passed to the *erase* method of the *list* class. In some implementations of the standard library this will result in an error; to be safe *iter* should be declared simply as *iterator*.

The functions starting at the bottom of 213 should be (changes highlighted):

```
void removeWordsWithLetter(list<string> & wordList, char forbiddenLetter) {
	list<string>::iterator iter;
	iter = wordList.begin();
	while (iter != wordList.end()) {
		if (iter->find(forbiddenLetter) != string::npos) {
			iter = wordList.erase(iter);
		} else {
			iter++;
		}
	}
}

void removeWordsWithoutLetter(list<string> & wordList, char requiredLetter) {
	list<string>::iterator iter;
	iter = wordList.begin();
	while (iter != wordList.end()) {
		if (iter->find(requiredLetter) == string::npos) {
			iter = wordList.erase(iter);
		} else {
			iter++;
		}
	}
}
```

Page 221: The declaration of integerListClass in the Java code (middle of page) requires parentheses:

```
integerListClass numberList = new integerListClass();
```

General: The chapter doesn't show the code for the *reduceByPattern* function; it's very similar to the other word-list functions:

```
list<string> reduceByPattern(const list<string> & wordList, char letter, list<int>
pattern) {
	list<string> newList;
```

```
	list<string>::const_iterator iter;
	iter = wordList.begin();
	while (iter != wordList.end()) {
		if (matchesPattern(*iter, letter, pattern)) {
			newList.push_back(*iter);
		}
		iter++;
	}
	return newList;
}
```