

# PERFORMANCE, DATA STRUCTURES AND ALGORITHMS

Exercise 09

Exercise 09

Find a Defect in a C program

# Name: Minakov Nikita

PURPOSE

The purpose of this exercise is to give you practice at finding errors in a C program and to explore different approaches to debugging code.

“Defensive coding” is important in minimizing potential errors in code. Some strategies for this include the following coding practices:

1. Using constants in your code removes significant risk associated with making code changes, especially when constant values are used in more than one place. The preprocessor directive #define can be used for this. The ability to change your code in multiple places by changing a constant value in one place greatly reduces the risk of errors, especially with things like sizes of arrays.
2. Using named constants (such as INPUT\_SIZE) also makes code much more readable so that others can more easily understand what the code does. When, 6 months later, you have to go back to enhance your code, you will also more easily understand how your program was intended to work when you initially wrote the code.
3. C also has the *const* keyword. This tells the compiler (and others reading your code) that a variable will not be changed. It is generally good practice when writing functions to use the *const* keyword in the parameter list for your function to indicate which parameters the function can and cannot modify. Consider the following function definition:

int function(const char \*input, const int size, int \*value)

This tells the compiler and others reading your code that the value that “input” points to should not be changed, nor should “size” be changed. Since “value” does not have the constant qualifier keyword, the item that value points to can be changed. An example call of this function might look like this:

function(input\_array, INPUT\_SIZE, &value\_to\_set)

Most (but not all) compilers will detect an attempt from within the function to change the value of a parameter that has the const qualifier and will flag it as an error.

* Using printf to display information about a program while it is running is a powerful tool for finding errors. Building debugging features into your code while you create it can save effort later during development of a program. The conditional compilation in Exercise 08 is an example of this.

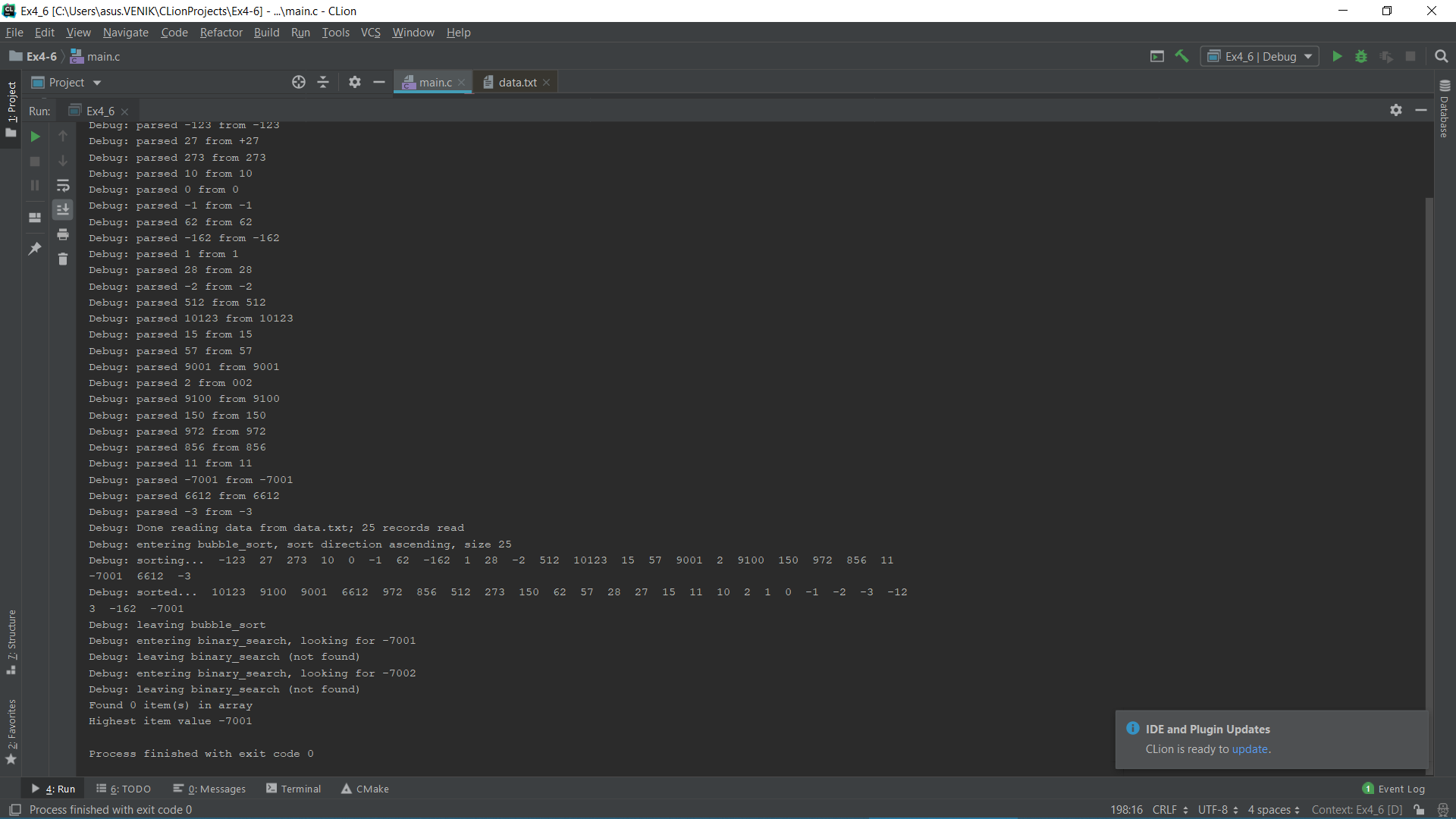
For this exercise, you are given a program that contains several errors. Your task is to use the debugging tools built in to the program to find and correct the errors. Note that the compiler directives in this program not only check to see if a constant is defined, but check the value of that constant to control conditional compilation. Thus, you can control the amount of debugging output. Keep in mind that if you set the debugging level too high for the problem you are trying to solve, you may end up inundated with a lot of information that you don’t need, which will make it harder for you to pinpoint the problem.

ACTIVITIES

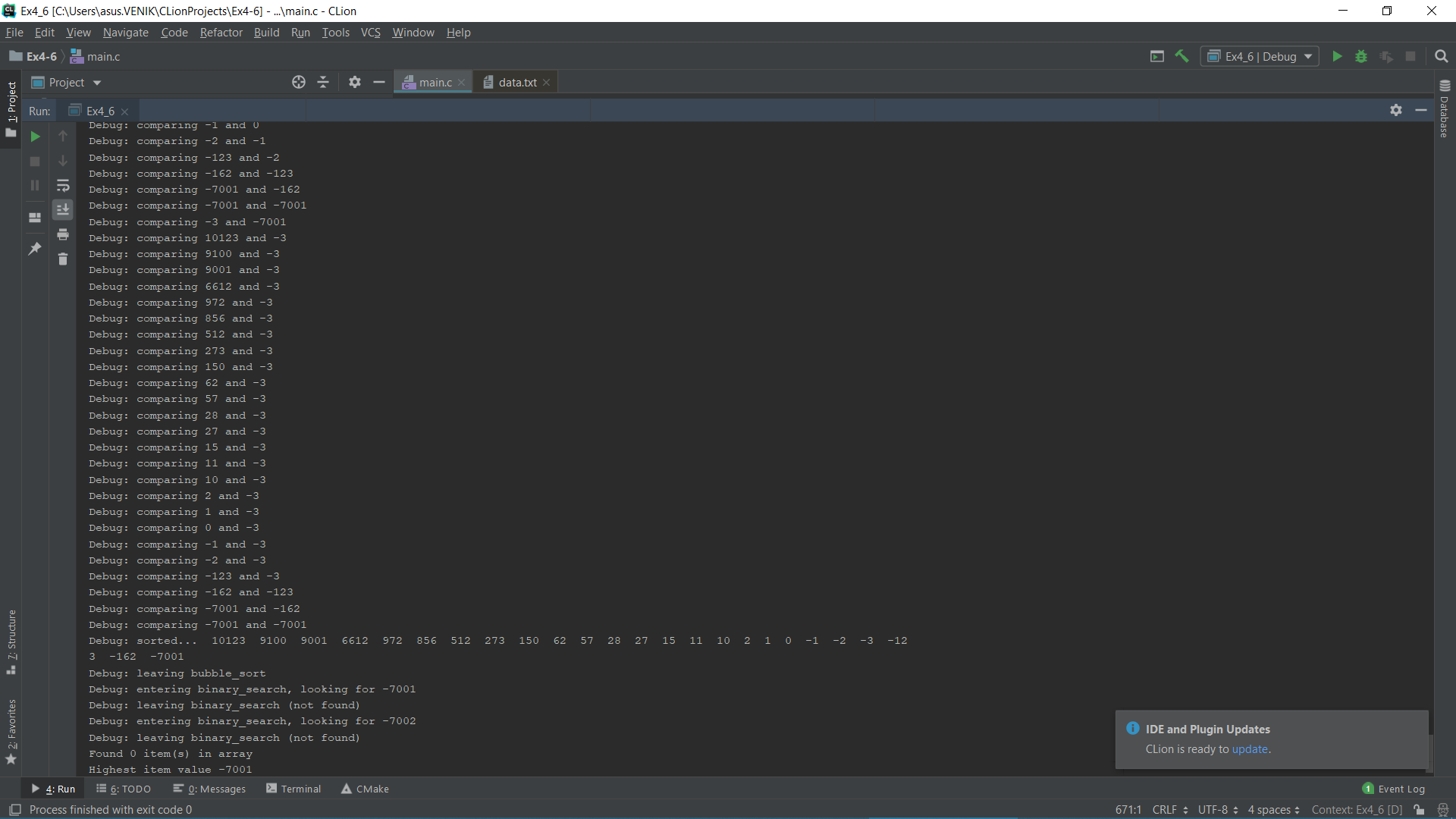
Perform each of the following activities. If you have questions, issues, or doubts, please ask for help and do not just guess.

1. Compile the program that you are given for this exercise using the provided makefile.
2. Why does the compiler give you an error message?
3. Change the program to remove the cause of the error message. (It is NOT correct to change the function definition.)
4. Compile the program again. If you have correctly fixed the problem, the program will compile with no errors or warnings.
5. Now run the program. Copy and paste the output below:
6. Try different settings for DEBUG at the top of the program and find the cause of this problem. Describe the cause of the problem:
7. Correct the problem (you can do this by changing the data file).
8. Run the program again. Describe what happens:
9. Experiment with different settings for DEBUG until you find the problem. Describe what you did to fix the problem:
10. Set DEBUG to 0 and compile and run the corrected program. Copy and paste the output here:
11. Is this the correct output for the program?
12. Increase the DEBUG level until you get enough information to tell you where the problem is, if not exactly what the problem is. In which function is the problem (main, binary\_search, or bubble\_sort)?
13. Now increase DEBUG to 3. Does the additional debugging output help you to pinpoint the problem? What would be a better way to debug this particular problem?
14. In the time you have left, discover the exact problem in the code and correct it. What was the error?
15. Save and upload this document to the LMS.

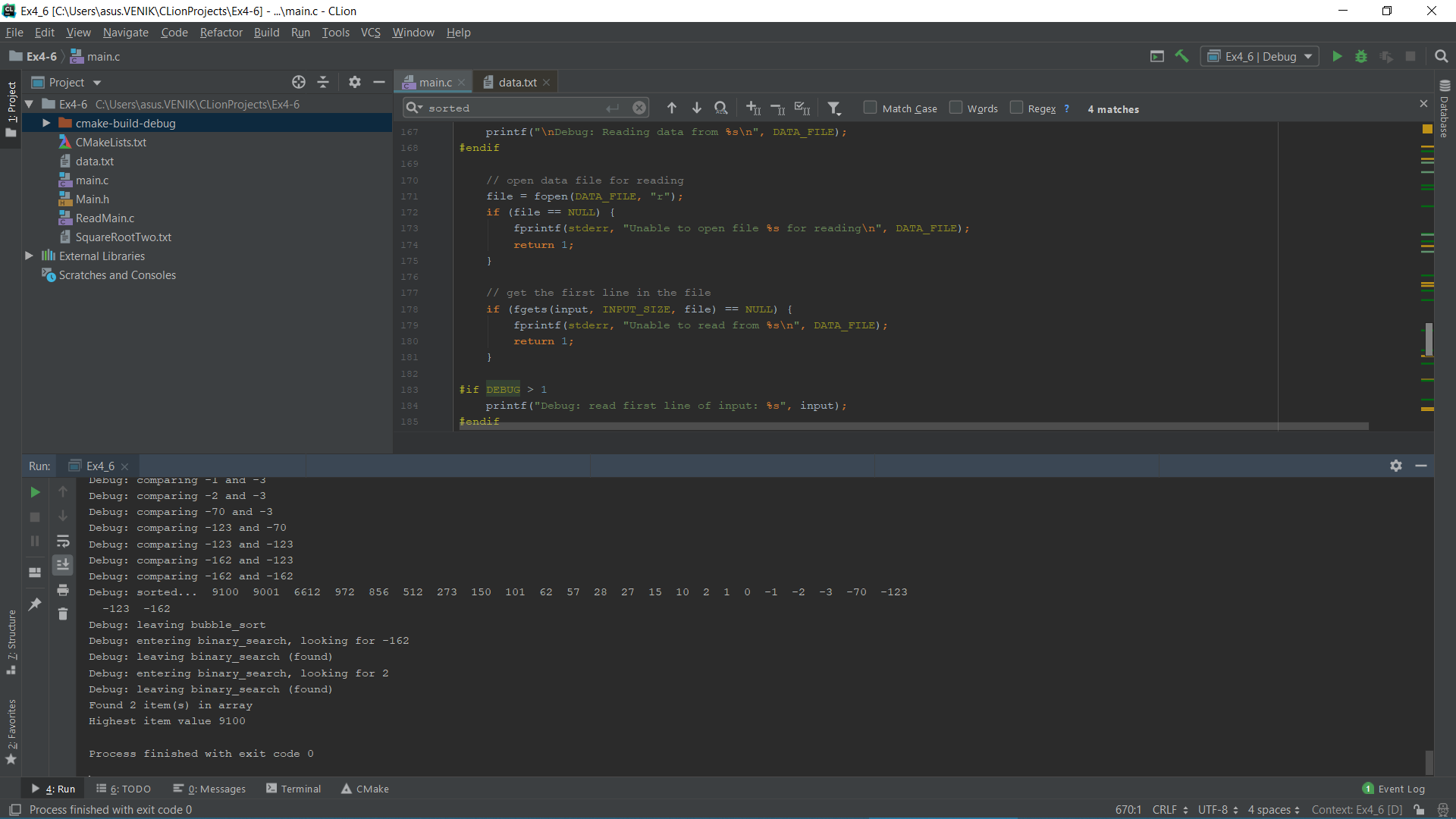
Debug 2:



Debug 3:



Final result:



Code:

#define \_CRT\_SECURE\_NO\_WARNINGS  
  
#include <stdio.h>  
#include <stdbool.h>  
  
#define DEBUG 0  
  
#define DATA\_FILE "data.txt" // Name of a data file containing the data to search (short ints)  
#define INPUT\_SIZE 7 // big enough for a short int with an optional sign  
  
#define ASCENDING 1 // sort in ascending order  
#define DESCENDING 2 // sort in descending order  
  
/\*\*\*\*\*\*\*\*\*  
\* bubble\_sort - sorts an array into ascending order using the (inefficient)  
\* bubble sort algorithm. The array is sorted in place (the  
\* caller's array is changed).  
\*  
\* input:  
\* array - an array of short integer numbers to sort  
\* array\_size - the number of items in array  
\*  
\* output:  
\* array is modified (sorted)  
\*  
\* return value: none  
\*\*\*\*\*\*\*\*\*\*/  
  
  
**void** bubble\_sort( **int** data[], **const int** size, **int** direction) {  
 **int** tmp; // temporary value storage  
 **int** i, j;  
 bool swap;  
  
#if DEBUG > 0  
 printf("Debug: entering bubble\_sort, sort direction %s, size %d\n", direction == ASCENDING ? "ascending" : "descending", size);  
#endif  
#if DEBUG > 1  
 printf("Debug: sorting...");  
 **for** (i = 0; i<size; i++) {  
 printf(" %d", data[i]);  
 }  
 printf("\n");  
#endif  
  
 // Starting from the beginning of the array, go through every item in the array...  
 **for** (i = 0; i<size; i++) {  
  
 // the item at i is the current item, now search the entire array, starting  
 // at the beginning, looking for the first value that is greater than or less  
 // than (depending on direction) the current item. Once found, swap them.  
 **for** (j = 0; j<size; j++) {  
#if DEBUG > 2  
 printf("Debug: comparing %d and %d\n", data[j], data[i]);  
#endif  
 **if** (direction == ASCENDING) {  
 **if** (data[j] < data[i])  
 swap = true;  
 **else** swap = false;  
 }  
 **else** { // direction is DESCENDING  
 **if** (data[j] > data[i])  
 swap = true;  
 **else** swap = false;  
 }  
 **if** (swap) {  
 tmp = data[i];  
 data[i] = data[j];  
 data[j] = tmp;  
 }  
 }  
 }  
  
#if DEBUG > 1  
 printf("Debug: sorted...");  
 **for** (i = 0; i<size; i++) {  
 printf(" %d", data[i]);  
 }  
 printf("\n");  
#endif  
#if DEBUG > 0  
 printf("Debug: leaving bubble\_sort\n");  
#endif  
  
 **return**;  
}  
  
/\*\*\*\*\*\*\*\*\*  
\* binary\_search - Uses a binary search algorithm to search for an item in  
\* an array. The array is assumed to be sorted in  
\* ascending order.  
\*  
\* input:  
\* search\_value - the value to search for  
\* array - an array of int  
\* array\_size - the number of items in array  
\*  
\* output: none  
\*  
\* return value: true if the item was found, else false  
\*\*\*\*\*\*\*\*\*\*/  
bool binary\_search(**const int** search\_value, **const int** \*array[], **const int** array\_size) {  
 **int** low = 0;  
 **int** high = array\_size - 1;  
 **int** mid;  
  
 // array[0] = -32768;  
  
  
#if DEBUG > 0  
 printf("Debug: entering binary\_search, looking for %d\n", search\_value);  
#endif  
  
 **while** (low <= high) {  
 mid = (low + high) / 2;  
 **if** (array[mid] < search\_value) {  
 low = mid + 1;  
 }  
 **else if** (array[mid] > search\_value) {  
 high = mid - 1;  
 }  
 **else** {  
#if DEBUG > 0  
  
 printf("Debug: leaving binary\_search (found)\n");  
#endif  
 **return** true;  
 }  
 }  
  
#if DEBUG > 0  
 printf("Debug: leaving binary\_search (not found)\n");  
#endif  
  
 **return** false;  
}  
  
/\*\*\*\*\*\*\*\*\*  
\* main - Read data from a file, sort that data into ascending order and use a binary  
\* search to find two values in the data, where one value is in the data and  
\* one value is not. Correct output indicates that one data value was found.  
\* In addition, the largest number in the data file is displayed.  
\*  
\* input: none  
\* output: none  
\*  
\* return value: 1 if there was an error, else 0  
\*\*\*\*\*\*\*\*\*\*/  
  
  
**int** main(**void**) {  
  
 FILE \*file;  
 **char** input[INPUT\_SIZE];  
 **const int** num\_items = 25;  
  
#if DEBUG > 0  
 printf("\nDebug: Reading data from %s\n", DATA\_FILE);  
#endif  
  
 // open data file for reading  
 file = fopen(DATA\_FILE, "r");  
 **if** (file == NULL) {  
 fprintf(stderr, "Unable to open file %s for reading\n", DATA\_FILE);  
 **return** 1;  
 }  
  
 // get the first line in the file  
 **if** (fgets(input, INPUT\_SIZE, file) == NULL) {  
 fprintf(stderr, "Unable to read from %s\n", DATA\_FILE);  
 **return** 1;  
 }  
  
#if DEBUG > 1  
 printf("Debug: read first line of input: %s", input);  
#endif  
  
 // the first line is an integer representing the number of lines of data to sort  
 **if** (sscanf(input, "%d", &num\_items) != 1) {  
 fprintf(stderr, "Invalid count of data items in first line of %s\n", DATA\_FILE);  
 fclose(file);  
 **return** 1;  
 }  
  
#if DEBUG > 1  
 printf("Debug: num\_items %d\n", num\_items);  
#endif  
  
 **int** data[25];  
  
 // (scanf returns the number of items it parses. If the values and types are not  
 // what it is expecting ("%hd" indicates exactly one short integer) it will return  
 // the number of values it was able to parse, which will not match the number of  
 // values specified in the format string ("%hd"). In this case, if the format  
 // of the input is what we are expecting, scanf will return 1.  
 //  
 // while we are not at the end of the file and we have valid data on an input line...  
 **int** i = 0;  
 **while** (fgets(input, INPUT\_SIZE, file) != NULL && sscanf(input, "%d", &data[i]) == 1) {  
#if DEBUG > 1  
 printf("Debug: parsed %d from %s", data[i], input);  
#endif  
 i++;  
 }  
  
 // make sure our data matches up with what we expect  
 **if** (num\_items != i) {  
 fprintf(stderr, "Number of data lines in %s (%d) is less than specified in first line in %s (%d)\n", DATA\_FILE, i, DATA\_FILE, num\_items);  
 fclose(file);  
 **return** 1;  
 }  
  
 // done with the input file  
 fclose(file);  
  
#if DEBUG > 0  
 printf("Debug: Done reading data from %s; %d records read\n", DATA\_FILE, num\_items);  
#endif  
  
 **int** found = 0;  
 bubble\_sort(data, 25, ASCENDING);  
  
 **if** (!binary\_search(data[num\_items - 1], data, num\_items))  
 found++;  
 **if** (!binary\_search(data[num\_items - 10], data, num\_items))  
 found++;  
  
 printf("Found %d item(s) in array\n", found);  
 printf("Highest item value %d\n", data[0]);  
 // for (int j=0; j<25; j++){  
 // printf("%d ", data[j]);  
 //}  
 // system("pause");  
  
 **return** 0;  
}