

# PERFORMANCE, DATA STRUCTURES AND ALGORITHMS

Exercise 08

Exercise 08

searching using recursion in c

# Name: Minakov Nikita csse 1709r

PURPOSE

The purpose of this exercise is to give you practice at working with recursion in C and to use conditional compilation to control the output of a program.

In Exercise 06, you used compiler directives (#define, #ifndef) in a header file to control whether definitions associated with the library are included. You did this to ensure that the header file is included exactly once in a source file that includes that header file. This is conditional compilation.

In this exercise, you will use conditional compilation to control the output of your program. Any code in any source file (.h and .c) can be enclosed in a #ifdef/#endif pair or a #indef/#endif pair. The first checks to see if a constant is defined. The second checks to see if a constant is *not* defined. Thus, a constant (#define) can be used to control whether or not code is included in the final program. In this case, the #ifdef means, “if <constant> is defined, include the following code”. The #endif marks the end of the conditionally compiled code.

The constant that controls the conditional compilation can be defined in a source file, or it can be defined as a parameter to the compile command. The latter is much more convenient when there are multiple source files in your program, so that you can avoid having to edit multiple files to change whether or not the constant is defined. Also, every time you edit a source file, it increases the chance of introducing errors to the program, so it is preferable to avoid editing source files if possible.

The makefiles that you have used so far have a macro called CFLAGS. This defines parameters that are passed to the C compiler. You can define any #define constant as a parameter to the compiler by using the -D option. For example:

gcc -Wall -I. -DDEBUG

This defines DEBUG, which can be used in the source code of your program to control whether or not debug output is printed as part of the output of a program.

When you change CFLAGS in the makefile, and then type “make” to build your program, nothing will be compiled. This is because you are not changing your source file. When you change a file, it results in a changed modification time. You could make a change to the source file to ensure a recompile when using the “make” command, but a change to the source file was not necessary in this case, we just only needed to change the makefile. An easy way to solve this is to use the “touch” command on the source file. The “touch” command sets the modification time of a file to the current time. Thus, you can edit your make file, “touch” your source files, and when you run “make”, the file will be compiled even though you have not modified the file.

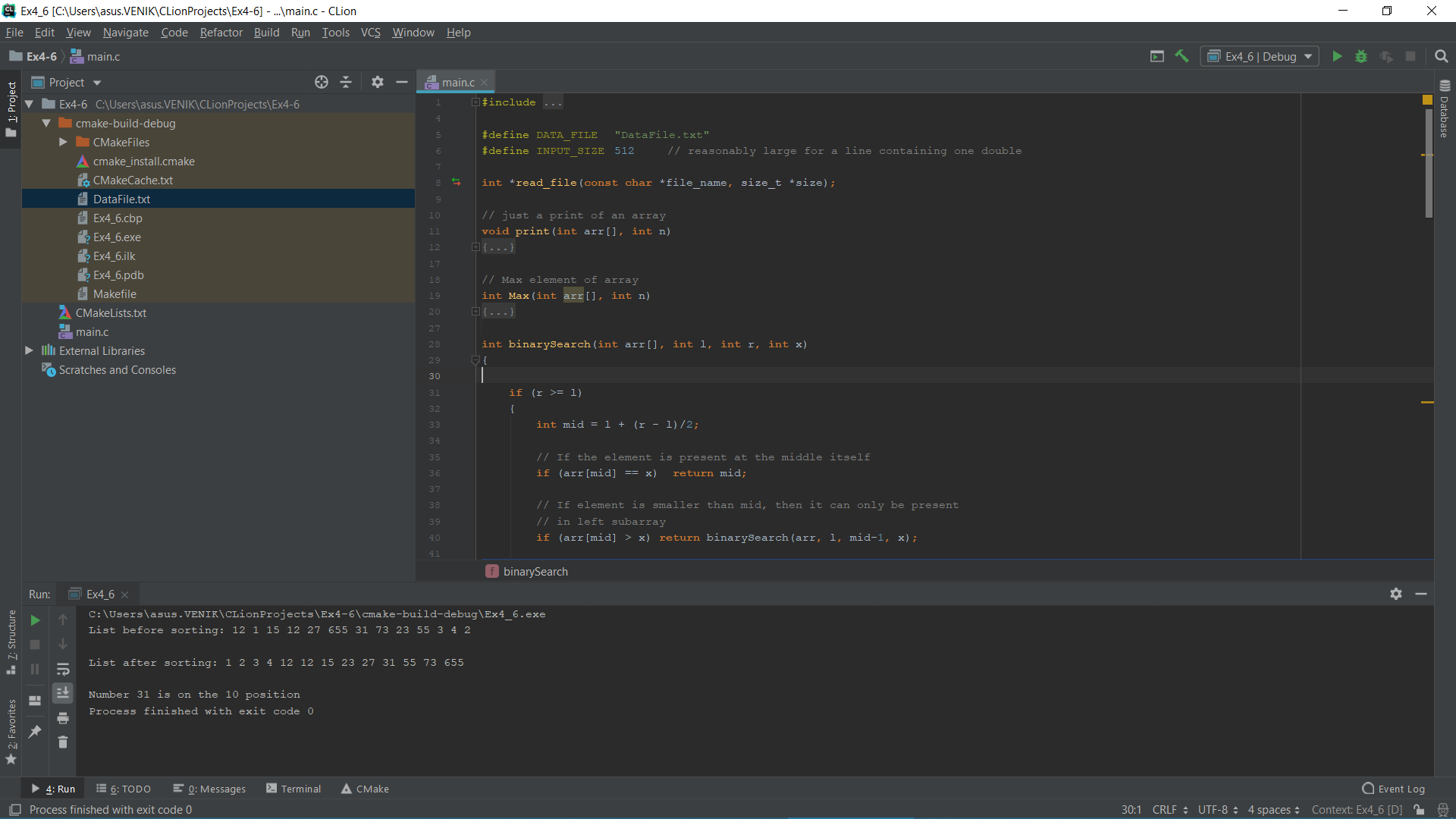
While touch allows selectively controlling which targets are made, an alternative way to force compilation of *all* targets in your makefile is to specify the -B option when you run make: “make -B”. Another useful make option is “-n”. This prints what would be built using make, without actually building it. The “man” page for make contains information about all the options that make recognizes.

For this exercise you are given a program that reads a file containing a series of floating point values, each on one line in the file. The numbers in the file are already sorted in ascending order, so you do not need a sort function for this exercise.

ACTIVITIES

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

1. Create a function that searches an array of doubles that has been sorted in ascending order. Use a recursive binary search for your search implementation.
2. Modify main() to call your search function once for each item in the search\_values array.
3. Your program should display each search value and whether or not it was found in the array.
4. Instrument your code using conditional compilation so that at compile time, you can control whether or not additional information is included in the output. The additional output should include at a minimum the index into the array and the value in the array at that index of the value that is being compared to the search value.
5. Create a makefile to build your program.
6. Compile your program using the makefile you created.
7. Run your program without the conditional code, so that the additional information is *not* included in the output. Copy and paste the output of your program below:
8. Run your program *with* the conditional code, so that the additional information *is* included in the output. Copy and paste the output of your program below:
9. Document your code carefully.
10. When you are ready to submit your work, first remove all intermediate files from your src directory.
11. Save and archive your finished C program, your makefile and this document, and upload it to the LMS.



Code:

#include <stdio.h>  
#include <stdlib.h>  
#include <stdbool.h>  
  
#define DATA\_FILE "DataFile.txt"  
#define INPUT\_SIZE 512 // reasonably large for a line containing one double  
  
**int** \*read\_file(**const char** \*file\_name, size\_t \*size);  
  
// just a print of an array  
**void** print(**int** arr[], **int** n)  
{  
 **for** (**int** i = 0; i < n; i++)  
 printf("%d ", arr[i]);  
 printf("\n");  
}  
  
// Max element of array  
**int** Max(**int** arr[], **int** n)  
{  
 **int** maxD = arr[0];  
 **for** (**int** i = 1; i < n; i++)  
 **if** (arr[i] > maxD)  
 maxD = arr[i];  
 **return** maxD;  
}  
  
**int** binarySearch(**int** arr[], **int** l, **int** r, **int** x)  
{  
  
 **if** (r >= l)  
 {  
 **int** mid = l + (r - l)/2;  
  
 // If the element is present at the middle itself  
 **if** (arr[mid] == x) **return** mid;  
  
 // If element is smaller than mid, then it can only be present  
 // in left subarray  
 **if** (arr[mid] > x) **return** binarySearch(arr, l, mid-1, x);  
  
 // Else the element can only be present in right subarray  
 **return** binarySearch(arr, mid+1, r, x);  
 }  
  
 // We reach here when element is not present in array  
 **return** -1;  
}  
  
**void** Sort(**int** arr[], **int** n, **int** pw) {  
 **int** out[1000]; // output array  
 **int** i, position[1000] = {0};  
  
  
 **for** (i = 0; i < n; i++)  
 {  
 position[(arr[i] / pw) % 10]++;  
 // Store count of occurrences in count[]  
 //print(position, n);  
 }  
  
 **for** (i = 1; i < 10; i++)  
 {  
 position[i] += position[i - 1];  
 // see the position of each number in digit  
 //print(position, n);  
 }  
  
 **for** (i = n - 1; i >= 0; i--)  
 {  
 out[position[(arr[i] / pw) % 10] - 1] = arr[i];  
 position[(arr[i] / pw) % 10]--;  
 // step by step proses of radix sort  
 //print(out, n);  
 }  
  
 // Saving "out" to our current list  
 **for** (i = 0; i < n; i++)  
 {  
 arr[i] = out[i];  
 }  
}  
  
// The main function to that sorts arr[] of size n using  
// Radix Sort  
**void** radix(**int** arr[], **int** n)  
{  
 **int** maxD = Max(arr, n); // Max value of list, needed for radix algorithm  
  
 //print(arr,n);  
 **for** (**int** pw = 1; maxD/pw > 0; pw \*= 10)  
 Sort(arr, n, pw);  
}  
  
  
// Driver program to test above functions  
**int** main()  
{  
 size\_t num; // the number of elements in the array created by read\_file  
  
 // Read in the values to sort from the data file  
 **int** \*arr= read\_file(DATA\_FILE, &num);  
 printf("List before sorting: ");  
 print(arr, num);  
  
 radix(arr, num);  
 printf("\nList after sorting: ");  
 print(arr, num);  
  
 **int** x = 31; //Searching number  
  
 **int** result = binarySearch(arr, 0,num-1, x); // binary search function  
  
 **if** (result == -1)  
 {  
 printf("\nNo this number in list");  
 } **else** {  
 printf("\nNumber %d is on the %d position", x, result + 1);  
 }  
  
  
 **return** 0;  
}  
  
**int** \*read\_file(**const char** \*file\_name, size\_t \*size)  
{  
 // open the file for reading  
 FILE \*fp = fopen(file\_name, "r");  
 **if** (fp == NULL) {  
 fprintf(stderr, "Unable to open %s for reading\n", file\_name);  
 **return** NULL;  
 }  
  
 // Determine the number of lines in the file (and therefore the number of double  
 // values, since the file is expected to have one value per line) by counting  
 // the lines. We need to know this so we can allocated an appropriate sized  
 // array for the values.  
 **char** input[INPUT\_SIZE]; // space to store the line read from the file  
 **int** num = 0; // number of items read  
 // fgets returns NULL at end of file or if there is an error  
 **while**(fgets(input, INPUT\_SIZE, fp) != NULL) {  
 num ++;  
 }  
 rewind(fp); // move the file pointer back to the start of the file  
  
 // returning the number of values in the file  
 \*size = num;  
  
 // Allocate space for the contents of the file. The size to allocate is the number  
 // of values in the file multiply by the size, in bytes, of each item (in this case,  
 // we are storing doubles in the array)  
 **int** \*array = (**int** \*)malloc(num \* **sizeof**(**double**));  
 **if**(!array) {  
 fprintf(stderr, "Unable to allocate %ld bytes for file buffer\n", (**long**) num);  
 **return** NULL;  
 }  
  
 // Read the file one line at a time and parse the value from the line using sscanf.  
 **int** i;  
 bool have\_error = false;  
 **for** (i=0; i<num && !have\_error; i++) {  
 // Get the line and put it into a string  
 **if**(fgets(input, INPUT\_SIZE, fp) == NULL) {  
 fprintf(stderr, "Unable to read values from %s\n", DATA\_FILE);  
 have\_error = true;  
 }  
 // Parse the value from the input line and store it in the array.  
 // sscanf returns the number of values in the string that match the pattern it is given,  
 // in this case "%lf" for one double value.  
 **if**(sscanf(input, "%d", &array[i]) != 1) {  
 fprintf(stderr, "Line %d of %s does not contain a valid floating point number\n", i+1, DATA\_FILE);  
 have\_error = true;  
 }  
 }  
  
 // close the input file  
 fclose(fp);  
  
 // check for an error - if there was an error parsing the file, free the space and return NULL  
 **if** (have\_error) {  
 free(array);  
 array = NULL;  
 \*size = 0;  
 }  
  
 **return** array;  
}