ECE 209 Program 2: `futo`

Spring 2019

Due Monday, March 06 @ 11:45pm

This programming assignment must be completed individually. Do not share your code with or receive code from any other student. Evidence of copying or other unauthorized collaboration will be investigated as a potential academic integrity violation. **The minimum penalty for cheating on a programming assignment is a grade of -100 on the assignment.** If you are tempted to copy because you're running late, or don't know what you're doing, you will be better off missing the assignment and taking a zero. Providing your code to someone is cheating, just as much as copying someone else's work.

DO NOT copy code from the Internet, or use programs found online or in textbooks as a "starting point" for your code. Your job is to design and write this program from scratch, on your own. Evidence of using external code from any source will be investigated as a potential academic integrity violation.

This program will solve a Futoshiki puzzle, a grid-based logic puzzle from Japan, also known as Unequal.

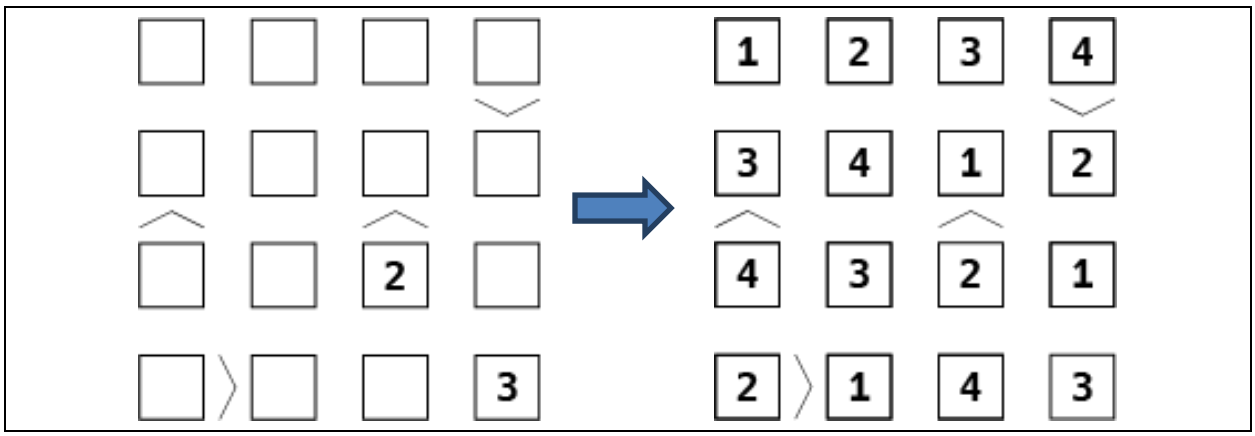
The learning objectives of the program are:

- Use C functions to solve a problem, including recursive functions.
- Read from a text file.
- Use dynamic memory allocation to create an array whose size is not known at compile time.

Futoshiki

Futoshiki involves an NxN square grid, in which the numbers 1 through N are placed. A number may not appear more than once in the same column, or more than once in the same row. In addition, there are greater-than/less-than relations specified between selected pairs of grid boxes, and these relations must be obeyed. The puzzle may start with some numbers already placed.

The figure below shows a 4x4 puzzle on the left, and its solution on the right.



For more information about Futoshiki puzzles, and for example puzzles, see **www.futoshiki.org**.

Solving the Puzzle: Recursive Backtracking

For a person, solving a Futoshiki puzzle requires complex and careful logical reasoning. However, it turns out that even the most challenging Futoshiki puzzles can be easily solved by a computer using *recursive backtracking*.

To use backtracking, we first choose a possible partial solution to the problem. Then we recursively try to solve the rest of the problem. If we succeed, we're done. If there's no solution compatible with our proposed partial solution, we "backtrack" and try a different solution. The use of a recursive function makes the code much simpler to manage than trying to remember where we are and how to backtrack.

A pseudocode description of the process for the Futoshiki puzzle is described below. The **solve** routine takes a partially-solved puzzle as its input, and returns True (1) or False (0), depending on whether a solution is found.

```
solve(p, N): p is the puzzle, and N is the size of the puzzle
  Find the first empty square in p.
  If there is no empty square, return True.  (puzzle is solved)
  For i = 1 to N:
    If i is not a legal solution for the square,
      Go back to the top of the loop to pick the next i
    Else:
      Write i in the square, which changes p
      If solve(p, N) is True, return True   (recursive)
      Else, erase i, making the square blank  (backtracking)
  End loop
  Return False  (no solution possible for p)
```

For each empty square, we try a number that follows all the rules -- this is a partial solution. Then we call **solve** again to see if this choice allows us to solve the entire puzzle. If not, then we try a different number (a different partial solution).

If we cannot find a number that leads to a complete solution, we return False to indicate that a solution could not be found. (And we undo our work, so that the puzzle is the same as when we were called.) Our caller will then try a different partial solution, or will give up if there are no more solutions to be tried. If we return True, then our caller also returns True, up to the original "root" caller.

To solve this problem, we need to (a) write code that evaluates whether a given number is legal in a particular square, and (b) write the **solve** function. We also need to decide how to represent the puzzle in a machine-readable way, and how to represent the puzzle and its solution using data structures.

File Format: Representing the Puzzle as Text

We are going to use a text file, that just contains ASCII characters, to represent the puzzle. For each grid row, we will have two rows of text. One row contains the content of the boxes (number or blank), as well as relationships between boxes in the row. The next row contains relationships between boxes in this row and the row below. The description below refers to the puzzle shown earlier in the figure.

The first line of the file contains a number, between 4 and 9, which is the size of the puzzle.

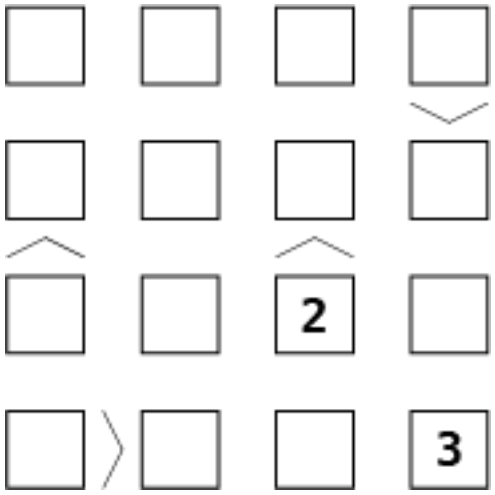
The next line is the representation of the first row of boxes. We start and end the line with a vertical bar ('|') -- this helps verify consistent spacing between the rows. For each box, we put either a digit (e.g., '1') or a hyphen ('-') if the box is blank. Next is either an empty space, if there is no relation specified,

or a greater-than ('>') or less-than ('<'). This repeats for each box in the row. Note that there is no space or symbol after the last box.

Next comes a row that contains only the "vertical" relationships between boxes on different rows. There are no hyphens, just spaces. The carat ('^') represents that the upper box is less than the lower box, and the lower-case 'v' indicates that the upper box is greater than the lower box. (Look at it sideways!)

This repeats for each row in the puzzle. For the final row, there is only the row of boxes, because there can't be a relation specified for a lower box.

In the figure below, the same puzzle as before is shown on the left, along with its text file representation on the right.




```
4
|- - - -|
|       v|
|- - - -|
|^    ^  |
|- - 2 -|
|       |
|->- - 3|
```

Data Structure: Puzzle and Constraints Arrays

Next, we need to define a data structure that will allow us to represent the puzzle and a partial (complete) solution in the program. You might think that we would use a two-dimensional array to represent a grid, but there are reasons why a "regular" one-dimensional array will work better for us. (We can talk, if you are curious.)

We need two kinds of information -- the boxes and the additional relational constraints. The boxes are easy: we'll use zero to represent an empty box, and numbers to represent the boxes with numbers. For the constraints, we'll need to encode the different greater/less-than options.

To represent the two-dimensional grid with a one-dimensional array, we will store the information in "row-major" order. We store the elements of each row contiguously, one after the other. For an NxN grid, to calculate the array index for the box on row *i* and column *j*:

$$index(i, j) = i \times N + j$$

The array below shows the representation of a the 4x4 grid shown earlier; note the zeroes that represent empty boxes. The array index is shown below each array element.

0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The first grid row is in elements 0-3. The next row is in 4-7, followed by 8-11, and finally 12-15.

For the additional constraints, we will use a separate array. For each box, we store a value that encodes the relationship with the box to its right, and the box below. We'll use a bitwise encoding, as follows:

- Bit 0 = this box must be less than the box to the right (<)
- Bit 1 = this box must be greater than the box to the right (>)
- Bit 2 = this box must be less than the box below (^)
- Bit 3 = this box must be greater than the box below (v)

More than one bit can be set. For example, the puzzle could specify that a particular box is both less than the box to the right AND greater than the box below. In that case, the value stored in the constraints array would be 1 + 8 = 9 (which is 1001 in binary).

The table below shows the constraints for the sample 4x4 puzzle.

0	0	0	8	4	0	4	0	0	0	0	0	2	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Element 3 is 8, because box 3 (0,3) must be greater than the box below. Element 4 is 4, because box 4 (1,0) must be less than the box below it. Likewise for element 6 (1,2). Finally, element 12 is 2, because box 12 (4,0) must be greater than the box to its right.

Some things to consider as you are implementing your functions:

- For a box (*i*,*j*), what is the array index of the box to its right?
- For a box (*i*,*j*), what is the index of the box below?
- For a box (*i*,*j*), what is the index of the first (leftmost) box in its row?
- For a box (*i*,*j*), what is the index of the first (top) box in its column?

Program Specification

The entire program must be written in a single source code file, named **futo.c**. I am giving you the **main** function, and you must implement at least four additional functions, described below. You may choose to implement additional functions as part of your implementation.

The complete program does the following:

1. Prompt the user to enter the name of a file that specifies a Futoshiki puzzle, as described earlier.
2. Read the text file. Allocate the arrays and fill in the data that represent the puzzle boxes and constraints.
3. Print the unsolved puzzle.
4. Solve the puzzle, filling in the boxes appropriately.
5. Print the solved puzzle, or print a message if there is no solution.

The **main** function performs this general flow. Your functions will implement the fun parts!

Details: Functions

This section gives the prototype (declaration) for each function[1] that you must implement, as well as a description about what that function must do. You may not change the prototype, and the function must

[1] In case the prototype in the document does not match the prototype in the code that I provide, go with the code. This is a prime location for a specification error to creep in, because sometimes I adjust the code without coming back to change this spec. If there's a discrepancy, ask on Piazza, but assume that the code version is correct.

behave as specified. We will be testing each function individually. You may declare and define other functions.

```
int readPuzzle(const char *filename, int **puzzle, int **constraints,
               int *size);
```

We start with probably the most confusing spec, because it is very likely the first function you will implement. The job of this function is to (a) open the file, (b) read the puzzle description and create the data structures, (c) use the pointer arguments to "return" those data structure to the caller, and (d) close the file. The return value is 1 if the file was read successfully, and 0 otherwise.

For this assignment, you should assume that the formatting of the file is always correct. You don't have to check for errors in how the puzzle is represented. Therefore, the only error that you might see is a failure to open the file. A brief discussion of file I/O is in the Appendix, but we will certainly discuss this in class well before the program is due.

The caller provides pointers that readPuzzle uses to communicate the new data structures and size information. There are three pieces of information needed:

- **size** is the size of the puzzle. The value is an integer, so the caller provides a pointer to an integer variable, where the value must be stored.
- **puzzle** is the puzzle box array, described above. Since the size of the array is not known until the file is read, it is the job of **readPuzzle** to dynamically allocate the array. (Use **malloc** -- see the Appendix.) What you end up with is a pointer to the first array element. The caller provides a pointer to an `int*` variable (hence the type `int**` -- pointer to an `int` pointer), where the pointer must be stored. The array must contain zeroes for blank boxes, and numbers for any boxes specified by the file.
- **constraints** is the puzzle constraints array, described above. This also needs to be dynamically allocated and stored using the `int**` provided by the caller. The array must contain zeroes for boxes with no constraints, and the encoded constraints for those boxes for which relational constraints are specified in the file.

Don't forget to close the file after reading it. You will lose points if you don't.

```
void printPuzzle(const int puzzle[], const int constraints[], int size);
```

This function prints the puzzle to the standard output stream (use **printf**). The size is <u>not printed</u>, but the rest is the same as the file format described above. The `puzzle` and `constraints` arrays may not be changed; that is the reason for the `const` part of the parameter specification.

```
int solve(int puzzle[], const int constraints[], int size);
```

This function attempts to solve a puzzle, represented by the puzzle array and constraints array, with the given size. It solves the puzzle by filling in the `puzzle` array with numbers. The function returns 1 if the solution is successful, and 0 if there is no solution. If there is no solution, the state of the puzzle array <u>must</u> be unchanged from its original state when the function returns.

The solution is successful if every element of the `puzzle` array is non-zero, and all values are consistent with the row, column, and relational constraints described earlier. The `constraints` array is never changed by the solver. That's the reason for the `const` part of the parameter specification.

The operation of this function is described in the earlier part of the specification. It is intended that you use recursive backtracking. If you choose to solve the puzzle in some other way, that is allowed. (But I don't recommend it. Don't fear recursion!)

```
int isLegal(int row, int col, int num,
            const int puzzle[], const int constraints[], int size);
```

This function determines whether it is allowed to place a particular number (`num`) into a particular box of the puzzle grid, represented by row and column. The function returns 1 if the number is legal, and 0 otherwise.

This function is not called by the **main** function in the code provided to you. It is intended that your **solve** function will call it. If you choose to not use this function as part of your implementation, that's ok. But the function must be defined and will be tested for correctness, even if you never call it.

For a number to be legal, it must satisfy ***all three*** criteria:

- The number does not match any other number in the same row.
- The number does not match any other number in the same column.
- The number satisfies any relational (greater/less-than) constraints specified by the box or the surrounding boxes.

This function must not change either the `puzzle` or `contraints` array. That is the reason for the `const` part of the parameter specification.

Testing the Program

Several puzzles will be provided for you. Remember that every function will be tested individually; your job is to correctly implement all of the functions. If you do so, then the program will work correctly. But it would be useful for you to specifically test your functions using puzzle arrays that exercise all aspects of the specification.

Hints and Suggestions

- Don't overcomplicate the program. Do not use anything that we have not covered in class.
- Work incrementally. The starter code you were given has dummy versions of each function, so that it will compile and run. I recommend that you implement **readPuzzle** and **printPuzzle** first. Get them working correctly before starting on the other functions.
- You must include **stdio.h** for **printf/scanf** and **stdlib.h** for **malloc**.
- For compiler errors, look at the source code statement mentioned in the error. Try to figure out what the error is telling you. Try to fix the first error first, and then recompile. Sometimes, fixing the first error will make all the other errors go away. (Because the compiler got confused after the first error.)
- Use a source-level debugger to step through the program if the program behavior is not correct. If you are using CLion on your own computer, the debugger is integrated with the editor and compiler, so there's no excuse for not using it.

- For general questions or clarifications, use the Message Board, so that other students can see your question and the answer. For code-specific questions, email your code (as an attachment) to the support list: ece209-sup@wolfware.ncsu.edu.

Administrative Info

Updates or clarifications on Piazza:

Any corrections or clarifications to this program spec will be posted on the Piazza. It is important that you read these postings, so that your program will match the updated specification.

What to turn in:

- Source file – it must be named **futo.c**. Submit via Moodle to the Program 1 assignment.

Grading criteria:

20 points: File submitted with the proper name. (This is all or nothing. Either you will get all 20 points, or you will get zero. Be sure your file is named correctly, and remember that case matters. No exceptions!!!)

10 points: Compiles with no warnings and no errors (using –Wall and –pedantic). (If the program is not complete, then only partial credit is available here. In other words, you won't get 20 points for compiling a few trivial lines of code.)

10 points: Proper coding style, comments, and headers. No unnecessary global variables. No goto. (See the Programming Assignments section on Moodle for more information.) For this assignment, there is no reason to use any global variable.

15 points: **readPuzzle** works correctly. Must close file for full credit.

10 points: **printPuzzle** works correctly.

15 points: **isLegal** works correctly.

20 points: **solve** works correctly. There may be more than one correct solution. Our testing will verify that your solution is correct, rather than comparing to one "golden" solution.

7.5 pts -- solves puzzles with no relational constraints.

7.5 pts -- solves puzzles with relational constraints.

5 pts -- correctly identifies puzzles that cannot be solved. (No points if your code always says that a puzzle can't be solved. Even a broken clock is correct twice a day.)

Appendix: File I/O

This section gives a quick overview of file-based I/O. This will be discussed in class. For more info, see Chapter 18 of Patt & Patel, and the two videos in the Macros and File I/O section of the Moodle site.

Opening a file

To open a file, use the fopen function. The first argument is a string that gives the file name. The second argument is a string that specifies the mode: "r" for reading and "w" for writing.

```
FILE* fopen(const char *name, const char *mode);
```

The return type is `FILE*`, which is the type used for an *I/O stream*. If the return value is zero (`NULL`), then there was an error opening the file. This usually means that a file of the given name was not found, but it can also mean that you don't have permission to open the file for the specified mode.

Below is a typical sequence for opening a file named "foo.txt":

```
FILE* in;  /* I/O stream for reading file */

in = fopen("foo.txt", "r");  /* open for reading */
if (!in) return 0;           /* if error, return an error code */
```

Note that the arguments don't need to be literal strings. If the name of a file is specified by a variable, it can still be used.

```
FILE* in;  /* I/O stream for reading file */
char *fname;  /* assume this will be set by other code... */

in = fopen(fname, "r");  /* open for reading */
```

Closing a file

When you are finished using a file, you should close it. This "cleans up" information used by the operating system. Your program will probably still work if you forget, and the file will be closed when your program terminates. But if you keep opening files and never close them, eventually you will run out of file descriptors and the OS will prevent you from opening more. It's good practice.

```
int fclose(FILE*);
```

The return value is an error code. We typically ignore it. (What would we do if a file doesn't close properly?)

Example: To close the file that we opened in the previous section:

```
fclose(in);
```

Reading and writing from a file

To read and write text from a file, we use special versions of **scanf** and **printf**, in which we specify an I/O stream as the first argument.

The **scanf** function always reads from the standard input stream (`stdin`). To specify the stream explicitly, we use **fscanf**. For example, if we had previously opened a file to create a stream called `in`:

```
fscanf(in, "%d", &x);  /* read decimal integer, store in x */
```

The **printf** function always prints to the standard output stream (`stdout`). To specify the stream explicitly, we use **fprintf**. For example, if we had previously opened a file to create a stream called `output`:

```
fprintf(output, "The answer is %d.\n", x);
```

That's all you need to know about file I/O for now. We will learn more about these functions in class. As always, you are free to ask questions on Piazza.

Appendix: Dynamic Memory Allocation

Sometimes we need an array, but we don't know the size of the array until runtime. For example, in this program, we don't know the size of the puzzle until we read the file. The compiler can't allocate this array for us, because it doesn't know how much space to leave. Therefore, we call a system routine named **malloc** (from "memory allocator") to allocate space for us.

We need to tell **malloc** the amount of memory to allocate, in *bytes*. Since each processor platform (LC-3, x86, Arm, etc.) might use different sizes for different data types, C provides the `sizeof` operator, implemented by the compiler, that tells how many bytes are needed for a particular data type. (Note: It looks like a function, but it's an operator.)

If I need an array that can hold *n* integers, I tell **malloc** that I need `n * sizeof(int)` bytes.

If my array will hold *n* floating-point values: `n * sizeof(double)` bytes.

In order to use the memory, I need to know where it is. Therefore, the return value for **malloc** is a pointer. We typically cast that pointer to the type that is appropriate for the array that we're allocating. If **malloc** was unable to allocate the amount of memory requested, it returns zero (`NULL`).[2]

As an example, suppose we want to use a variable a to point to a dynamically-allocate array of integers. And the variable n (an integer) specifies how many integers are needed in the array. The type of a is int*, because it will point to the first element of the array.

```
int n;   /* gives the size of the array */
int *a;  /* will point to the array */

/* ...code... */

/* allocate an array of n integers */
a = (int*) malloc(n * sizeof(int));
```

For this program, the **readPuzzle** function is responsible for allocating the puzzle and constraints arrays. If the puzzle size is *n*, then the grid is *n*×*n* and each array requires n^2 integers. We use malloc to allocate the array. We also use the `int**` parameter (puzzle) to store the array pointer in the location specified by the caller.

```
int *p;  /* will point to the puzzle box array */

p = (int*) malloc(n*n*sizeof(int));  /* array for n-by-n puzzle */
*puzzle = p;  /* store ptr in location specified by caller */
```

Appendix: Example Run

You will be given several example puzzle files. Below is the result of running the program with the file named puz4.txt, which is the 4x4 example used throughout this specification document.

[2] To be absolutely safe, it's a good idea to check the return value of **malloc**, and to take some error handling action if the allocation was not performed. For this class, we will assume that **malloc** will always succeed and will never return `NULL`.

Remember: There could be multiple solutions to a given puzzle. As long as your solution is correct, it does not have to exactly match the solution found by our implementation.

```
Puzzle file: puz4.txt
PUZZLE:
|- - - -|
|       v|
|- - - -|
|^    ^  |
|- - 2 -|
|        |
|->- - 3|

SOLUTION:
|1 2 3 4|
|       v|
|3 4 1 2|
|^    ^  |
|4 3 2 1|
|        |
|2>1 4 3|
```

Appendix: CLion Tips

If you are using CLion as your development environment, here are some tips to make your life easier for this program.

Using the provided source code

I am giving you a source code file named **futo.c**, which contains the main function, declarations, and dummy implementations of all the functions. To use this in your CLion project:

1. Create a new CLion project (C executable, C90 standard). You can call the project anything you want. The project name is also the name of the directory that will be used to hold the project files.
2. Open the project in CLion.
3. Download **futo.c** into the project directory.
4. Edit the **CMakeLists.txt** file, and change all mentions of "main.c" to "futo.c". Now the project will compile **futo.c**, and you won't have to remember to change the file name when you submit your code.
5. While you're in there, you should also set the compiler flags:
   `set(CMAKE_C_FLAGS "-Wall -pedantic")`
6. Delete the **main.c** file. You don't need this anymore. You can do this by right-clicking on the file in the lefthand column, or you can go to the directory and delete the file.

Downloading puzzle files

To keep things simple, we want to download the puzzle files into the same directory as your source code files -- the main project directory. The problem: If you do that, then your program won't be able to find the file when you open it, because it is looking in the directory where the executable resides.

To fix this, you should set the working directory for your project. That will tell the program to look in that directory, instead of the default.

1. With your project open, select Run > Edit Configurations...
2. Next to Working Directory, click the box with ...
3. Navigate to and select your project directory.

You can choose a different directory, if you want, but to me it's convenient to have the files in the same directory as the source code for this program.

Creating new puzzle files

You should create your own puzzle files for testing. If you want to do so:

1. Right-click on the project name.
2. Select New > File. Give it a name.
3. The file should open in the editor. Be sure your file is formatted correctly, because your code is not designed to deal with file formatting errors.

Appendix: Linux

This is not new information, but here again is how to compile and run a program on the EOS Linux system. (This is where we will compile and test your code.)

Compile

```
gcc -o futo -Wall -pedantic futo.c
```

Run

```
./futo
```

The puzzle file must be in the same directory as your executable.