CS 40: Interfaces, Implementations, and Images (iii)

Check the course calendar for due dates. Your design submission, described in Part E: Designing Function Contracts, is due prior to your code submission.

Please read the entire assignment before starting work.

Contents

1 Purpose

This assignment has five goals:

1. To spur you to think more deeply about programming technique.
2. To give you practice designing your own interfaces, not just using interfaces designed by other people.
3. To give you practice thinking about what familiar algorithms and data structures you can use to solve new problems.
4. To build a set of abstractions that will help you represent and manipulate digital images on this and future assignments.
5. To lay a foundation for future assignments. In these future assignments:
   - You will learn about *locality*, its effects on performance, and how to change a program's locality.
   - You will understand how data structures in a high-level language map to machine structures, and how to improve space performance by programming directly with machine structures.
   - You will learn to improve the performance of programs by *code tuning*.

2 Preliminaries

- In Hanson's *C Interfaces and Implementations*, refresh your memory about exceptions (Section 4.1) and memory management (Section 5.1). Study the `Bit` abstraction defined in Chapter 13 and the `UArray` abstraction defined in the Hanson supplemental chapter.
- From wherever you intend to work, issue:

  ```
  pull-code iii
  ```

  You should now have two `.c` files and two executables. These are discussed below.
- You should also now have a `Makefile` that's used in a manner similar to the one we gave you for HW1, but that has targets for the code you will build in this assignment. The style of this `Makefile` is a little different, but all the key concepts are the same; indeed, this `Makefile` also introduces a few interesting new features of `make`.

3 Part A: Two-Dimensional, Polymorphic, Unboxed Arrays

In the Hanson supplemental chapter, Dave Hanson and Norman Ramsey provide an abstraction that implements unboxed one-dimensional arrays. For this part of the assignment, you'll adapt the unboxed-array abstraction to support *two*-dimensional arrays. Your adaptation will be called `UArray2` and should define the type `UArray2_T`. Your adaptation should include the following changes when compared to the original `UArray` interface:

- Instead of a *length*, a `UArray2_T` will have a *width* and a *height*.
- Instead of being identified by a single index, an element of a `UArray2_T` will be identified by *two* indices: the *column* or x index measures the distance between the element and the left edge of the array, while the *row* or y index measures the distance between the element and the top row of the array. Thus the top left element is always indexed by the pair *(0, 0)*.
- Omit the `resize` and `copy` operations.
- You must define *two* analogs of the `Bit_map` function described on Hanson page 201:
  - `UArray2_map_row_major` calls an `apply` function for each element in the array. Column indices vary more rapidly than row indices.

– `UArray2_map_col_major` calls an `apply` function for each element in the array. Row indices vary more rapidly than column indices.

The terms "row major" and "column major" may be found in Bryant and O'Halloran as well as on Wikipedia.

As in Hanson's code, an out of bounds reference or a failure to successfully allocate needed memory should result in a checked run-time error. You must also follow Hanson's conventions, e.g., that Hanson's abstract type name (`UArray2_T` in this case) denotes a pointer.

It is **not acceptable** to clone and modify Hanson's implementation. Your new code should be a *client* of Hanson's existing code, and you should rely on Hanson to do the heavy lifting as much as possible.

For Part A, the problem you are to solve is **define an interface and build an implementation** for `UArray2`. This file **must be self-contained**, i.e. it must all be contained in `uarray2.c`. Do not create separate files for any helper functions you might create. Richard Townsend's solution to this problem takes about 120 lines of C code (for both the interface and implementation combined).

Helper Code

We want you to have the experience of creating and implementing an interface, but we want to be sure the interface you create is at least reasonably close to what we're expecting. To help you get this right, we are giving you some tools to help you understand what's expected and to do some simple testing of your results.

When you pull the starter code (as instructed in Section 2) you will get two files relating to `UArray2`s:

1. `useuarray2.c`: this is the source to a C program that uses a `UArray2` implementation. Specifically, it does a `#include "uarray2.h"` and calls most of the methods in the interface.
2. `correct_useuarray2`: this is an executable program that is built from the `.c` program, but it's linked with our "correct" implementation of `Uarray2`. So, you can see what a correct implementation of `Uarray2` does.

You can do some interesting things with the the `.c` source. The key is that when you build *your* `uarray2.h` and `uarray2.c`, link them with the `.o` resulting from `useuarray2.c`, and run the result *you should produce the same output* as `correct_useuarray2`. Knowing that, here are some things to consider doing:

- Carefully read and figure out what `useuarray2.c` is doing. It's not an otherwise useful or sensible program, but it's designed to make clear certain characteristics required of your `Uarray2` interface and implementation. The code is intentionally not heavily commented, as figuring things out from the code itself is part of the exercise. Of course, you'll want to compare the source with the output.
- When you think you've got a good `uarray2.h`, use the `Makefile` to try compiling `useuarray2.c` with your `.h` file to produce a `.o` file. If the compile doesn't work, your `.h` file has a problem.
- After you build your own `uarray2.c` (which is your implementation of the 2D array), use the `Makefile` to build `my_useuarray2`. You can do this by issuing the command:

  ```
  make my_useuarray2
  ```

  This will take the `.o` file you got in the step above and link it with *your* `uarray2.o` to produce an executable. If your code needs additional modules to run then you will need to modify the `Makefile`, but if you just need `uarray2.o` what we supply should work.
- Now, run your executable. If the output doesn't exactly match what's produced by `correct_useuarray2` then you surely have a problem in your implementation. Hint: use file redirection and the Linux `diff` command for this. Use the `man` command to find out about `diff` if you need a refresher.

Note: `useuarray2.c` **is *not* designed as a comprehensive test program**, though it does some very helpful limited testing. Writing good test code is your job. You are welcome to make a copy of `useuarray2.c` and hack it up to make better test programs. Indeed, one of the reasons the actual logic in that source looks a little odd and arbitrary is that we did not want to hand you a complete test framework. So, it includes just enough to highlight some features of the interface.

The `my_useuarray2` program should compile and run properly before you submit your work.

Hints

- The key to this problem is to set up an implementation in which the elements of your two-dimensional array are in one-to-one correspondence with elements of one or more one-dimensional `UArray_T`'s. The key question to answer is

  How do you relate each element in a two-dimensional array to a corresponding element in some one-dimensional array?

  Having a precise answer to this question will set you up for success. If your answer is not precise, it's easy to get lost chasing pointers.
- Representation is the essence of programming! Your major design decision will be how to represent a `UArray2_T` (and a `Bit2_T`, which is described in the following section). Two obvious alternatives, both of which are acceptable, are:
  - To represent a `UArray2_T` as an array of `UArray_T`'s.
  - To represent a `UArray2_T` as a single `UArray_T`.
- The indices into a two-dimensional array, regardless of how you name them (e.g., "x and y" or "column and row" or anything else), are always both integers. When passing them in the wrong order to a function, however, the compiler will not catch the error. A common mistake is to use different orderings in different parts of your code. Choose **one ordering** and use it *consistently* in all your code. Consistent use of the names "row" and "col" will also tend to reduce mistakes compared to use of single-letter names. It will also make your program clearer to your readers/graders.
- Don't worry about performance; aim for simplicity. If you feel compelled to worry about performance, you may make simple code improvements provided you *justify* them. Don't try anything radical; premature optimization is the root of much evil.
- Think carefully about what should be the prototype for the `apply` function passed to the functions `UArray2_map_col_major` and `UArray2_map_row_major`.
- The pixels in a portable gray map are stored in row-major order, so one way to test your `UArray2` mapping functions is to write a simple program that reads and writes a graymap by calling the `UArray2_map_row_major` function with an argument that calls `Pnmrdr_get` from the `Pnmrdr` interface. If you compare results with `diff -bu` you should be able to get the same output as `pnmtopnm -plain`. If you read with `UArray2_map_col_major` and write with `UArray2_map_row_major`, you should be able to duplicate the effect of `pamflip -transpose`.
- Think about other ways of putting data into your array that will make it easy to tell whether your implementation is working as intended. Sometimes, putting in some temporary debugging output can be helpful too.
- When working with void pointers, it's easy to get confused about the correct number of levels of indirection. **Draw diagrams**.

Your life will be much easier if you follow the programming idioms for Hanson's `UArray_T`s, which deal with most of these issues.

4 Part B: Two-Dimensional Arrays of Bits

In some cases, particularly for documents scanned at high resolution, it can be useful to represent an image as an array of bits. To save space, it is useful to have a *packed* representation of such images. The `pbm` format provides this style of representation but (ANNOYINGLY) takes the "opposite" approach to the `pgm` format you experienced in HW1: in `pgm`, black is represented as 0 and white is some maximum value; in `pbm` (which is what you'll be using in Section 6), **black is represented as 1 and white is represented as 0**.

For this part of the assignment, you'll design `Bit2`: an interface to support *two*-dimensional arrays of bits.

Helper Code

We supply helper code for `Bit2` that's equivalent in function and intent to what you used above for `UArray2`. Adapt the instructions in the obvious way.

Hints

- Your interface should be very, very similar to your `UArray2` interface, with one possible exception: because it is not possible to create a pointer to a single bit, you cannot use the `at` idea; your only option is an interface like the `Bit` interface, which exports `put` and `get`.
- Your interface should include row-major and column-major mapping operations.
- Your interface should not contain anything analogous to the set operations in Chapter 13. These operations are quite useful when considering a one-dimensional bit vector as a set, but it is rare to require set operations over integer pairs. Indeed, the most useful transformations and computations over two-dimensional bitmaps involve an operator called "bit blit." Google and Wikipedia are fine sources for this operator, but if you are curious you will find a marvelous collection of simple transformations in a classic article by Guibas and Stolfi (Note that Tufts students have free access to ACM articles). *There is no need to implement any of these transformations.*
- You should provide for checked runtime errors in the cases equivalent to those allowed for `UArray2`.

For part B, the problem you are to solve is **define an interface and build an implementation** for `Bit2`. Similar to `UArray2`, `Bit2` must be self contained. Richard Townsend's solution to this problem takes about 110 lines of C code.

5 Part C: Using `UArray2` to identify Sudoku solutions

Background: Programs as Predicates

You have seen that C (and C++) programs produce a value, i.e., they return a value to the operating system either by returning from `main` or by calling `exit`. (Either method of terminating the program will be considered "exiting the program.") A value of 0 indicates success (think "no problem"). Any other value is considered an error indication. Programs can use non-zero return codes, aka *exit codes*, to indicate what went wrong.

> Happy families are all alike; every unhappy family is unhappy in its own way.
>
> —Leo Tolstoy, *Anna Karenina*

Fun fact: You can find out the return code of the last program you ran in the shell by typing `echo $?`

You've seen that C provides the names `EXIT_SUCCESS` and `EXIT_FAILURE` (via `stdlib.h`), defined to be `0` and `1`, respectively, to be used as a clearer indication of the intent of the exit codes.

Thus, we can write entire programs that function as *predicates;* they result in a true or false indication. A program that *succeeds* (exits with a code of `EXIT_SUCCESS`, aka 0) is treated as *true*; a program that *fails* (exits a non-zero code like `EXIT_FAILURE`) is treated as *false*. That's right — 0 means true!

The following `bash` shell program prints `Hurray!` if, for example, `foo.c` contains the word "struct" and `Aww...` otherwise:

```
if grep struct *.c >& /dev/null
then
     echo 'Hurray!'
else
     echo 'Aww...'
fi
```

The command `grep struct *.c >& /dev/null` succeeds (with an exit code of 0) if it finds the string "struct" in any of the C source files in the current directory. `grep` fails (returns a non-zero code) if it doesn't find the string. (There are different codes depending on whether it was actually able to search the file or not; the file may not exist or the permissions may not permit `grep` to read the contents.) The output redirection throws away both standard output and standard error because, in this example, we don't want to see the result; we just want to know whether the string is there.

The Problem

Write the test program `sudoku`, which functions as a predicate. The syntax of the sudoku command is:

```
./sudoku [ filename ]
```

where `filename` is the (optional) name of an input graymap file; if there is no filename argument, input data is on standard input.

We define *correct* input as a single portable graymap (i.e., `pgm`) file. If the input is not *correct*, your program should terminate with a checked run-time error (any one will do). Otherwise, if the input is *correct*, your program **must not print anything**, but:

- If the graymap file represents a solved sudoku puzzle, your program must exit with `EXIT_SUCCESS` (i. e., a code of 0).
- Otherwise it must exit with `EXIT_FAILURE` (i. e., a code of 1).

(You may define your own constants for the values 0 and 1 if you have clearer names for them.)

A solved sudoku puzzle is a nine-by-nine graymap with these properties:

- The maximum pixel intensity (aka the denominator for scaled integers) is nine.
- No pixel has zero intensity.
- In each row, no two pixels have the same intensity.
- In each column, no two pixels have the same intensity.
- If the nine-by-nine graymap is divided into nine three-by-three submaps (like a tic-tac-toe board), in each three-by-three submap, no two pixels have the same intensity.

Here's an example (which you can also view as an image):

```
P2
9 9
# portable graymap representing a sudoku solution
9
1 2 3   4 5 6   7 8 9
4 5 6   7 8 9   1 2 3
7 8 9   1 2 3   4 5 6

2 3 4   5 6 7   8 9 1
5 6 7   8 9 1   2 3 4
8 9 1   2 3 4   5 6 7

3 4 5   6 7 8   9 1 2
6 7 8   9 1 2   3 4 5
9 1 2   3 4 5   6 7 8
```

Norman Ramsey's solution to this problem takes about 120 lines of C code. There is a significant opportunity for abstraction; a Very Good solution will identify such opportunities and use them to avoid repeating code.

6 Part D: Using `Bit2` to remove black edges

Write the program `unblackedges`, which removes black edges from a scanned image. The program takes at most one argument:

- If an argument is given, it should be the name of a portable bitmap file (in `pbm` format; remember the difference from `pgm` discussed in Section 4?).
- If no argument is given, `unblackedges` reads from standard input, which should contain a portable bitmap.
- If more than one argument is given, or if a portable bitmap is promised but not delivered, or if the supplied bitmap has a width and/or height of 0, `unblackedges` should either raise a Checked Runtime Error (using Hanson assertions or a Hanson Exception) OR halt with some sort of error message (on `stderr`) and terminate with a return value of EXIT_FAILURE.

The program `unblackedges` should print, on standard output, a plain (P1) format portable bitmap file which has width, height and pixel values identical to the original file except that all *black edge pixels* are changed to white. You may note that PBM allows a comment field on the line after the P1 code; feel free to put a comment of your choice into the comment field of the output file if you like.

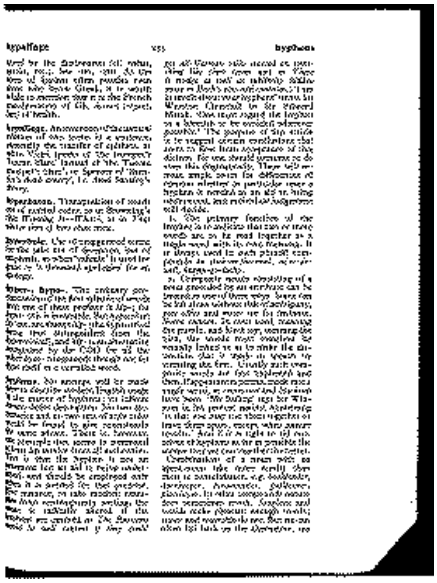
You can find some sample images in `/comp/40/bin/images/bitonal`. Try, for example,

```
pngtopnm /comp/40/bin/images/bitonal/hyphen.png | ./unblackedges | display -
```

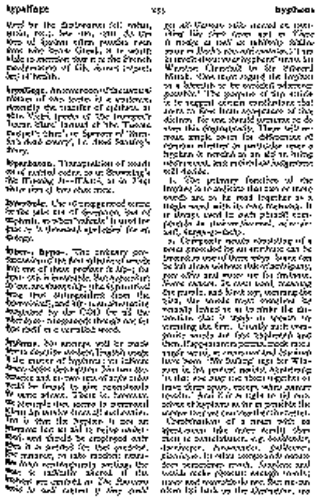
Here's an example of removing black edges from a scanned image:

For a bitmap of size w by h, a black edge pixel is defined inductively as follows:

- A pixel is a black edge pixel if it is black and if it appears in column 0, in row 0, in column w - *1*, or in row h - *1*.
- A pixel that appears in column c, row r is a black edge pixel if it is black, if $c > 0$ and $c < w$ - *1*, if $r > 0$ and $r < h$ - *1*, and any *neighboring pixel* is a black edge pixel.



Before



After

- The neighboring pixels of the pixel in column c, row r are
  - The pixel in column c - *1*, row r
  - The pixel in column c + *1*, row r
  - The pixel in column c, row r - *1*
  - The pixel in column c, row r + *1*

Norman Ramsey's solution to this problem takes about 110 lines of C code for the main problem, plus about 40 lines of code that could be reused for other problems. John Dias suggested an even simpler solution that requires less than 70 lines of code for the main part, and John's solution runs 30% faster.

Hints

- Your solution is expected to scale to images of reasonably large size and complexity. However, the Halligan servers impose a strict and unusually low limit on the size of the stack allotted to any given program. Think carefully about perfectly reasonable strategies that this limit might preclude.
- You may find it useful to define auxiliary functions with these prototypes:

```
Bit2_T pbmread (FILE *inputfp);
void   pbmwrite(FILE *outputfp, Bit2_T bitmap);
```

  You can read or write pixels using an explicit loop or a row-major mapping function. To learn the correct output format for a PBM file, run

```
man 5 pbm
```

  and look for the **"plain" format** described at the bottom of the page. NOTE: programs like `display` may be forgiving of some errors when displaying `PBM` files, but your output is required to be correct per the man page. Make sure that your output conforms to *all* the rules or you may lose credit for all of your results.
- There is at least one opportunity to exploit one of your `map` functions.
- If you find yourself in difficulty, try writing a simpler program that merely inverts the image in a bitmap (change white to black and vice versa).

7 Part E: Designing Function Contracts

Your `filesofpix` design process focused on architecture, implementation, and testing. These are still important elements to consider in this assignment. For example, if you seek help from a TA, you must be able to clearly describe the representation you are using for your 2-dimensional array and your testing plan. However, your design *submission* for `iii` will focus on a new aspect of design: function contracts.

A good function contract is comprised of two parts. The first is a brief (1–2 sentence) description of what your function does. For example, the `readaline` function in your `filesofpix` submission might be described as: "Reads one line of text from the file specified in the first parameter into the character buffer supplied by the second parameter."

The second part of your function contract lists the logical expectations of your function's input and output. These are expectations that the compiler cannot verify, and thus you must ensure that they are met. Often, each of these expectations will translate into an `assert` statement in your code. For example, one of the expectations of `readaline` is that the first parameter, `inputfd`, is non-null. This would translate to the following `assert` statement:

```
assert(inputfd != NULL);
```

For this design submission, you will submit your two interfaces, `uarray2.h` and `bit2.h` (that is, the text from these two files should be copied, as is, into your design document). The `uarray2.h` interface you submit should specify function contracts for *every* function in that interface; the `bit2.h` interface does not need to provide them.

8 Organizing and submitting your solutions

8.1 Submitting your design document

Your document should be a `pdf` file named `design.pdf`. This should be an actual `pdf` file and not, for example, a Word file renamed with a `.pdf` extension; export an actual `pdf`. When you are ready to submit, put your design document on the server (this can be done with the `scp` command or through VS Code). Then, `cd` into the directory containing your design document and run the following command:

```
submit40-iii-design
```

Just like last time, please check Gradescope to make sure that the correct document was submitted. Do **not** upload your design document to Gradescope manually. If you do not see the document on Gradescope after a few minutes, check the log that printed out when you submitted (using `prolog40` if necessary). If the submission was accepted but not uploaded and your partner's login is correct, then post a note on Piazza and we will look into it.

8.2 Submitting your completed code

- In your final submission, don't forget to include a `README` file which:
  - Identifies you and your programming partner by name
  - Acknowledges help you may have received from or collaborative work you may have undertaken with classmates, programming partners, course staff, or others
  - Identifies what has been correctly implemented and what has not

  - Says **approximately how many hours you have spent** completing the assignment

- Your submission should include at least these files:

```
README
Makefile
uarray2.h
uarray2.c
bit2.h
bit2.c
sudoku.c
unblackedges.c
```

- When you get everything working, `cd` into the directory contianing your submission and type `submit40-iii` to submit your work.