CSC 411 Assignment: Interfaces, Implementations, and Images

Interfaces and design checklists(design-adt.pdf on Sakai) for parts A and B Friday, February 7th at 11:59PM. Full assignment (all of parts A, B, C, D, and E) due **Friday, February 14th** at 11:59PM.

Please read the entire assignment before starting work.

Purpose

This assignment has four 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 lay a foundation for future assignments. In these future assignments,
   - You will learn about *locality*, its effects on performance, and how to change the locality of a program.
   - 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*

The abstractions you build in this assignment will help you represent and manipulate digital images.

Preliminaries

- In Hanson's *C interfaces and implementations*, refresh your memory about exceptions (Section 4.1) and memory management (Section 5.1). Study the `Array` abstraction defined in Chapter 10 and the `Bit` abstraction defined in Chapter 13.

Part A: Two-Dimensional, Polymorphic, Unboxed Arrays

Hanson provides 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 an `Array`:

- Instead of a *length*, an `UArray2_T` will have a *width* and a *height*.
- Instead of being identified by a single index, an element of an `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 `UArray2_put`. Implement `UArray2_at` which is analogous to `Array_get`.
- Omit the `resize` and `copy` operations.
- You must define *two* analogs of the `Bit_map` function described on 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, a reference out of bounds should result in a checked run-time error.

For part A, the problem you are to solve is *define an interface and build an implementation for `UArray2`.*
My solution to this problem takes about 100 lines of C code.

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 `Array_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?*

  If you have a precise answer to this question, the code is pretty easy to get right. If not, it's easy to get lost chasing pointers.

- 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 `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 `UArray2_map_row_major` with a function 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 `pnmtoplainpnm`.
If you read with `UArray2_map_col_major` and write with `UArray2_map_row_major`, you should be able to duplicate the effect of `pnmflip -transpose`.

Unpleasantness to watch out for:

- When working with `void *` pointers, it's easy to get confused about the correct number of levels of indirection. **Draw diagrams.**
- If you assign the result of `Array_get(a, i)` to a pointer `p`, it *must* be the case that

  ```
  sizeof(*p) == Array_size(a)
  ```

  Violating this specification results in an *unchecked* run-time error. Sometimes, but not always, you can catch such errors using valgrind's default "memcheck tool". It's therefore good to check this property with an assertion, e.g.,

  ```
  element *p = Array_at(a, i);
  assert(sizeof(*p) == Array_size(a));
  *p = f();
  ```

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

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. Each bit is either black (1) or white (0). To save space, it is useful to have a *packed* representation of such images. For this part of the lab, you'll design `Bit2`: an interface to support *two*-dimensional arrays of bits.

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 not good sources for this operator, but if you are curious you will find a marvellous collection of simple transformations in a classic

article by Guibas and Stolfi. *There is no need to implement any of these transformations.*

For part B, the problem you are to solve is *define an interface and build an implementation for* `Bit2`.
My solution to this problem takes about 110 lines of C code.

Part C: Using the `UArray2` abstraction to identify Sudoku solutions

Write the test program `sudoku`. It takes as input a single portable graymap file, which may be named on the command line or may be given on standard input. Your program **must not print anything**, but if the graymap file represents a solved sudoku puzzle, your program should call exit(0); otherwise it should call exit(1). 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.
- When 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 in `/csc/411/images/sudoku/sudoku.pgm`):

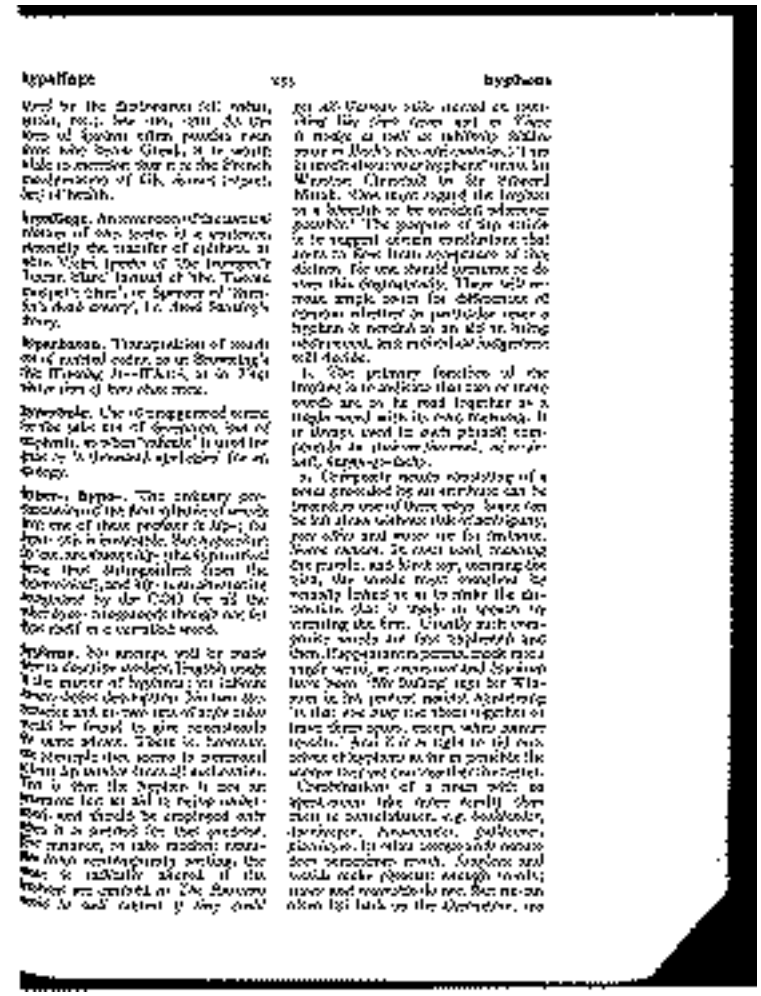
```
P2
9 9
# portable graymap representing a sudoku solution
9
1 2 3   4 5 6   7 8 9
4 5 6   7 8 9   1 2 3
7 8 9   1 2 3   4 5 6

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
```

My 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.

If `sudoku` is used in a way that violates its specification, it should terminate with a checked run-time error (any one will do). Read the specification carefully!

Part D: Using the `Bit2` abstraction to remove black edges

Write the test program `unblackedges`, which removes black edges from a scanned image. Example:



Before

After

The program `unblackedges` takes at most one argument:

- If an argument is given, it should be the name of a portable bitmap file (in `pbm` format).
- If no argument is given, `unblackedges` reads from standard input, which should contain a portable bitmap.
- If more than one argument is given, `unblackedges` halts with an error message.
- If a portable bitmap is promised but not delivered, `unblackedges` should halt with some sort of error message on `stderr`.

Program `unblackedges` should print, on standard output, a portable

bitmap file which is identical to the original file except that all *black edge pixels* should changed to white. You can find some sample images in `/csc/411/images/bitonal`; try, for example,

```
pngtopnm /csc/411/images/bitonal/hyphen.png | ./unblackedges | display -
```

** Note that to display images over ssh, you will need an X11 server installed on your own machine (XQuartz on MacOS) and to ssh with the `-Y` option **

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 i, row j is a black edge pixel if it is black, if $i>0$ and $i<w-1$, if $j>0$ and $j<h-1$, and any *neighboring pixel* is a black edge pixel.
- The neighboring pixels of the pixel in column i, row j are
  - The pixel in column $i-1$, row j
  - The pixel in column $i+1$, row j
  - The pixel in column i, row $j-1$
  - The pixel in column i, row $j+1$

My solution to this problem takes about 110 lines of C code for the main problem, plus about 40 lines of code that I can reuse for other problems. A colleague suggested an even simpler solution that requires less than 70 lines of code for the main part, and his solution runs about 30% faster than mine. It borrows ideas from garbage-collection algorithms.

Hints:

- Like the one-dimensional array, a 2-dimensional array can wear multiple hats in the world of ideas–that is, it can represent multiple abstractions. Besides the obvious, what other data structure is lurking within this problem? (The key data structure is suggested not by the mere fact of having a two-dimensional array, but by what we are asking you to do with it.)
- What algorithms have you seen that might be relevant?
- What data structure might you use to represent the set of black edge pixels in row i?
- You may find it useful to define auxiliary functions with these prototypes:

  ```
  Bit2_T pbmread(const char *path);
  void pbmwrite(FILE *fp, Bit2_T bitmap);
  ```

  You can write 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.

- 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).

If `unblackedges` is used in a way that violates its specification, it should terminate with a checked run-time error (any one will do).

Part E: Programming technique

Meet with your partner and identify *one* programming technique that meets *either* of these two criteria:

- One of you has incorporated it into your programming practice since January.
- One of you would like to incorporate it into your programming practice by the end of this term.

Your assignment is to **describe the technique in enough detail that a student halfway through CSC 212 could put it into practice**. Use at most one page.

You should submit your description in a plain text file called `TECHNIQUE`, or if you prefer, a PDF file called `technique.pdf`.

Expectations for your solutions

Your course instructor has thought of several ways to solve parts A and B. *Representation is the essence of programming!* Your major design decision will be how to represent a `UArray2_T` and a `Bit2_T`. Several obvious alternatives, all of which acceptable, are

- To represent a `UArray2_T` as an array of `Array_T`'s.
- To represent a `UArray2_T` as a single `Array_T`.
- To represent a `Bit2_T` as an array of `Bit_T`'s.
- To represent a `Bit2_T` as a single `Bit_T`.

What is not acceptable is 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. Reuse his code as much as possible.

Another significant design decision is the type (prototype) of the `apply` function you'll use in your row-major and column-major traversals.

> *If you're concerned about performance, don't worry—we'll make a careful study of it throughout the term, and you'll have a chance to revisit and improve your implementation.*

Common mistakes

Avoid these common mistakes:

- The indices into a two-dimensional array, whether they are called "x and y" or "i and j" or "n and m" or "row and column," are always both integers. The only way to distinguish them is which one appears first. A common mistake is to use different orders in different parts of your code. Choose one order or the other and put it first **consistently** in all your code.
- As in the previous assignment, don't forget to **initialize each C pointer variable**. (For a review of programming with pointers, we recommend Pointer Fun with Binky.)
- When getting a value from `Array_get` or a `UArray2_T_at` function, assign the value to a pointer of a suitable type. Then read or write the element by dereferencing the pointer. When the element is itself a pointer, it is easy to mistake the *address* of the pointer for the pointer itself. This mistake leads to valgrind errors and core dumps. Look at the C programming idioms.

Organizing and submitting your solutions

- On Monday, submit two interfaces and a DESIGN file which describes your design:

  ```
  uarray2.h
  bit2.h
  DESIGN
  ```

  If you prefer to use a word processor you can submit `design.pdf` instead of `DESIGN`. *Please make sure that any file called* ***`design.pdf`*** *is actually a PDF file.*
  Your `DESIGN` file should contain two design checklists (see design-adt.pdf on Sakai), one for each abstraction. **We are especially interested in items 1, 2, and 4** from the checklist. If you are confident of your design, you can scant the other items—but if you run into difficulty, we will ask you to complete the full checklist before asking for help.

Submit your design to the iii-design assignment on Sakai.

- In your final submission on the following Monday, 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 (ask me sometime about Otis the debugging dog)
  - Identifies what has been correctly implemented and what has not
  - Contains the critical parts of your design checklists

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

- Your submission should include at least these files:

  ```
  README
  compile
  uarray2.h
  uarray2.c
  bit2.h
  bit2.c
  sudoku.c
  unblackedges.c
  TECHNIQUE (or technique.pdf)
  ```

  Submit them to the iii assignment on Gradescope.