Comp 40 Homework 2: Interfaces, Implementations, and Images

Interfaces and design checklists for parts A and B are due earlier; the full assignment (all of parts A, B, C, D, and E) is due 6 days after that. **Check the course calendar for due dates.**

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 `Bit` abstraction defined in Chapter 13 and the `UArray` abstraction defined in the supplemental chapter.
- From wherever you intend to work, issue:

    ```
    git clone /comp/40/git/iii
    ```

    This should create a directory named `iii` in which you will do your work. Look in it and you should see two .c files and two executables. These are discussed below.
- The `git clone` will also create a `Makefile` that's similar in style to the one you used for HW1, but that has targets for the code you will build in this assignment. The `Makefile` also introduces some new features of `make`. p

Part A: Two-Dimensional, Polymorphic, Unboxed Arrays

In the supplement to Hanson, 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*, 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 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. Also, you must follow Hanson's conventions, in particular, note that Hanson's abstract type name (`UArray2_T` in this case) denotes a pointer.

For part A, the problem you are to solve is *define an interface and build an implementation for `UArray2`*. Norman Ramsey's 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 `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?*

  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`.
- 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.

Unpleasantness to watch out for:

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

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

  Violating this aspect of the 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 = UArray_at(a, i);
assert(sizeof(*p) == UArray_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.

Helper code

New for Spring 2014 (some TAs may not yet have seen this!): 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 do the `git clone` as instructed above you will get two files relating to two dimensional UArrays:

1. `useuarray2.c`: is the source to a C program that uses a UArray2 implementation. Specificially, 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 on otherwise useful or sensible program, but *it's designed to make clear certain characteristics requried 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 `useuarrauy2.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 the 2D array), use the `make` 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 compile script, 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: redirect the output of each program to a file. Use the `man` command to find out about the Linux `diff` and `cmp` commands; they may be useful!

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_useuarray` (and `my_usebit2`) programs should compile and run properly before you submit your project.

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 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 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 (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`. Norman Ramsey's solution to this problem takes about 110 lines of C code.

Part C: Using the `UArray2` abstraction 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 considerd “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*

You can find out the return code of the last program you ran in the shell by typing “echo $?”

Thus, we can write entire programs that function as *predicates*, which result in a true or false indication. A program that exits with a code of 0 is treated as *true*, a program that exits a non-zero code is *false*. That's right — 0 means true! The following 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`” returns 0 if it finds the string “struct” in any of the C source files in the current directory. `grep` 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. Correct input is a single portable graymap file. The syntax of the sudoku command is:

```
sudoku [ filename ]
```

where filename is the (optional) name of the input graymap file; if there is no filename argument, input data is on standard input. 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 a code of 0.
- Otherwise it must exit with a code of 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.
- 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
```

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! Note: it is true that the default handler for checked runtime errors writes an error message but that output is allowed.

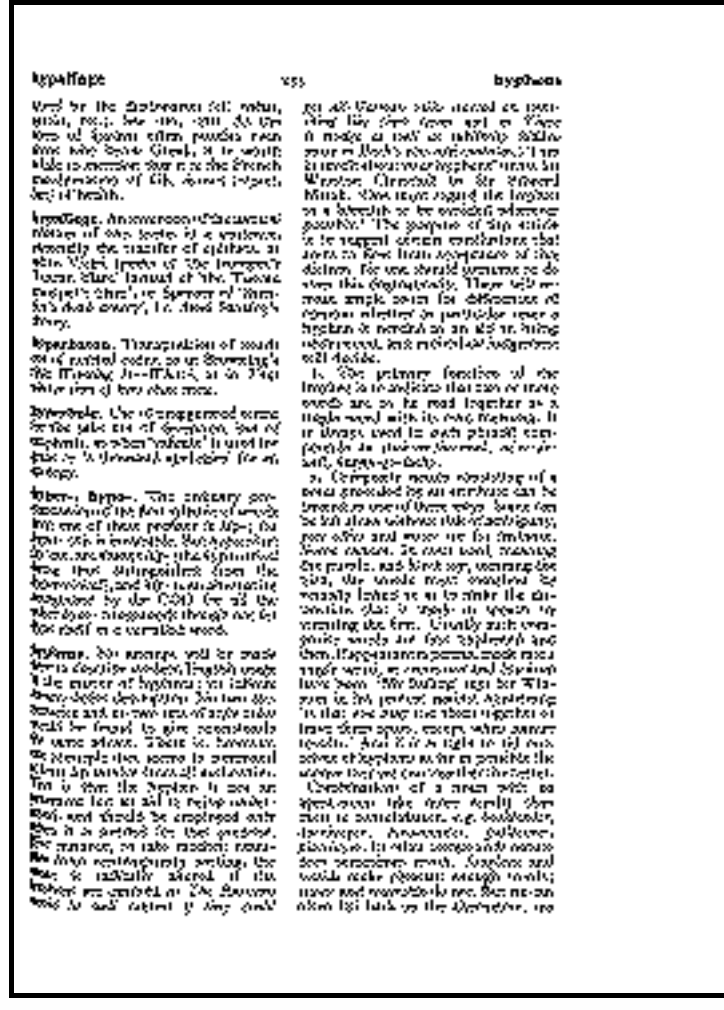
If the input file has additional data following data that is otherwise correct input, that additional data should be ignored (there is no need for you to look for it or attempt to read it at all; `pnmrdr` will silently ignore it).

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.

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 (on `stderr`).
- 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 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; the comment line begins with a # and the comment continues to the end of the line. You may 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/images/bitonal` try, for example,

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

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*

Wherever the specification above calls for the program to terminate with a message to `stderr`, you may instead raise an exception and rely on the default handler to write to `stderr`. Do *not* use an assertion.

If unblackedges is used any any other way that violates its specification, it MUST terminate with a reasonable exception;

Norman Ramsey's solution to this problem takes about 110 lines of C code for the main problem, plus about 40 lines of code 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:

- 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 (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).

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 the start of the term.
- 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 COMP 15 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 to use a word processor, 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 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`.
- 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. 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 orders in different parts of your code. Choose one order or the other and put it first **consistently** in all your code. By the way: Noah Mendelsohn thinks that consistent use of the names "row" and "col" will also tend to reduce mistakes compared to use of "i, j" or "x, y".
- 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 `UArray_at` 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 design

- For the design submission, 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, 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.

- To submit your design, change to the directory that contains these files and run the `submit40-iii-design` command. For this to work, you will have had to run `use comp40` either by hand or in your `.cshrc` or your `.profile`.

Before submitting your code

Remember that the main purpose of these assignments is not to get the code running. It's to teach a variety of concepts and techniques, some of which are summarized at the start of this assignment. Although you work in pairs, it's essential that *each* of you separately review what you have learned. For this assignment that includes:

- If you had an assignment of similar complexity in the future, could you (if necessary) succeed without a partner?
- Do you have a good sense of how existing ADT implementations (UArray) can be composed to build higher level abstractions, without those implementation details leaking through to the user?
- Do you have a clear mental picture of what is stored in memory as your Soduku and

unblackedges programs run?

- Do you understand the conceptual differences, the implementation differences and performance tradeoffs between boxed and unboxed object representations?

These are things we expect of *each* of you right now; they also may be the subject of questions on the midterm or final. There's a lot to learn, and we welcome you coming to us whenever have questions or need help. If you are unclear on any of the above, this is a good time to either discuss with your partner (if he or she might help you learn), or to seek out a TA or instructor.

Organizing and submitting your code and technique document

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

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

Mark A. Sheldon (msheldon@cs.tufts.edu)
Last Modified 4 February 2016