CS 40 Assignment: Locality and the costs of loads and stores

Overview and purpose

This assignment is all about the cache and locality. You'll implement blocked two-dimensional arrays, which you'll then use to evaluate the performance of image rotation using three different array-access patterns with different locality properties. You'll also see how to write code that is polymorphic in an array type.

The assignment has two parallel tracks:

1. On the **design and building track**, you will implement *blocked* two-dimensional arrays and *polymorphic* image rotation.

2. On the **experimental computer-science track**, you will predict the costs of image rotations, and later measure them. Your predictions will be based on knowledge of the cache as covered in Chapter 6 of Bryant and O'Halloran and as covered in class.

As described in Section 2, in this assignment we provide you with a *lot* of code and information. It will take time to assimilate. You can get the starter code by running the following command in the directory where you intend to work:

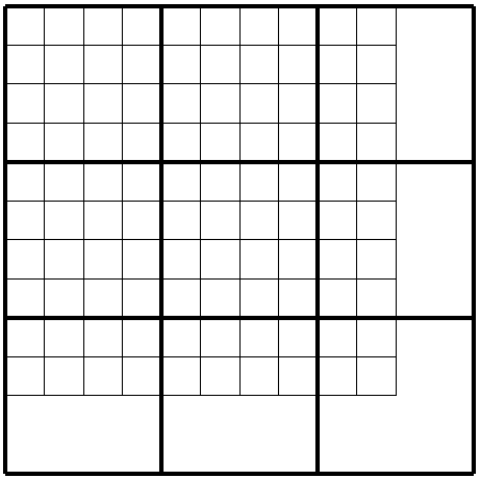
```
pull-code locality
```

Contents

1 Problems

1.1 Part A: Improving locality through blocking

In this part of the assignment, you will implement a standard technique for improving locality: *blocking*. The idea is best expressed in a picture. Here is a 10-by-10 array organized in 4-by-4 blocks:



The idea is simple: the blocked array has a similar *interface* to `UArray2`, but a different *cost model*. In particular,

- **Cells in the same block are located near each other in memory**.
- **Mapping is done by blocks**, not rows or columns. Mapping visits all cells in one block before moving on to the next block.
- **Some memory is wasted** at the right and bottom edges: not all the cells in those blocks are used. But if the array is large, then the wasted memory has size $O(\sqrt{n})$ and is unimportant. If the array is small, it probably fits in the cache and you shouldn't use blocking.

You have just one task for this part:

- **Implement blocked arrays** as described in the `UArray2b` interface below. Your solution must be submitted in a source file named `uarray2b.c`. Your solution to the problem must be self-contained in a single file.

Required interface

The interface you are to implement, to be called `UArray2b`, appears in Figure 1. The `blocksize` parameter to `UArray2b_new` counts the number of *cells* on one side of a block, so the actual number of cells in a block is `blocksize * blocksize`. The number of *bytes* in a block is `blocksize * blocksize * size`. The `blocksize` parameter has no effect on semantics, only on performance.

The `UArray2b_new_64K_block` allows you to default the blocksize; it is similar to `UArray2b_new`. It chooses a `blocksize` that is as large as possible while still allowing a block to fit in 64KB of RAM (here, a KB is 1024 bytes). If a single cell will not fit in 64KB, the block size should be 1. The reasonably fast L2 caches on most modern machines will hold between 128 KB and 256 KB of data, so if you create arrays using `UArray2b_new_64K_block`, you can fit two blocks in the cache at one time.

```
#ifndef UARRAY2B_INCLUDED
#define UARRAY2B_INCLUDED

#define T UArray2b_T
typedef struct T *T;

/*
 * new blocked 2d array
 * blocksize = square root of # of cells in block.
 * blocksize < 1 is a checked runtime error
 */
extern T    UArray2b_new (int width, int height, int size, int blocksize);

/* new blocked 2d array: blocksize as large as possible provided
 * block occupies at most 64KB (if possible)
 */
extern T    UArray2b_new_64K_block(int width, int height, int size);

extern void  UArray2b_free     (T *array2b);

extern int   UArray2b_width    (T  array2b);
extern int   UArray2b_height   (T  array2b);
extern int   UArray2b_size     (T  array2b);
extern int   UArray2b_blocksize(T  array2b);

/* return a pointer to the cell in the given column and row.
 * index out of range is a checked run-time error
 */
extern void *UArray2b_at(T array2b, int column, int row);

/* visits every cell in one block before moving to another block */
extern void  UArray2b_map(T array2b,
                          void apply(int col, int row, T array2b,
                                     void *elem, void *cl),
                          void *cl);

/*
 * it is a checked run-time error to pass a NULL T
 * to any function in this interface
 */

#undef T
#endif
```

Figure 1: Interface for blocked arrays

One possible architecture for your implementation

If you wish, you may use your own design and architecture for the implementation of `UArray2b`, or you may use one of mine described as follows:

> Here is a simple architecture for `UArray2b`. Because of the many layers of abstraction, it does not perform very well, but it is relatively easy to implement.
>
> - A `UArray2b_T` can be represented as a `UArray2_T`, each element of which contains one block.
> - A block should be represented as a single `UArray_T`. This representation guarantees that cells in the same block are in nearby memory locations.
> - To find the cell at index (i, j), first find the block at index (`i / blocksize`, `j / blocksize`). Within that block, use the cell at index `blocksize * (i % blocksize) + (j % blocksize)`.
> - Your mapping function should visit all the cells of one block before moving onto the cells of the next. **Blocks on the bottom and right edges may have unused cells**, and your mapping function must **not** visit these cells.

If you implement this design successfully, it is not too difficult to modify the code such that your blocked array is stored in a single, contiguous area of memory. Once you have the address arithmetic right, you can get a substantial speedup by avoiding all the memory references involved in going indirectly through the `UArray2` and `UArray` abstractions. But the focus of this assignment is not primarily on performance. We expect your program to run at reasonable speed, but you will do fine with the `UArray2b` design outlined above (I.e. with the `UArray2_T` of `Uarray_T`s). If you prefer, you can try a different approach.

Our solutions

Norman Ramsey wrote two solutions to this problem. The one that uses the design sketched above is about 175 lines of C, 50 of which appear at the end of this assignment. He then wrote another, faster solution which is about 130 lines of C. The faster solution has a significantly more complicated invariant and was correspondingly more difficult to get right.

1.2 Part B: Supporting polymorphic manipulation of 2D arrays

You now have two different representations of two-dimensional arrays: `UArray2`, which supports column-major and row-major mapping, and `UArray2b`, which supports block-major mapping. In order not to duplicate code, we want to write image rotations that can operate on *either* kind of array. To achieve this kind of reuse, we resort again to polymorphism: we define an interface `A2Methods` that can represent either kind of two-dimensional array. You write *one* image-rotation program against this interface, and you can use it with *two* implementations.

- Because I have specified the exact interface for `UArray2b`, I can provide an implementation of `A2Methods` that uses `UArray2b`.
- Because you designed the `UArray2` interface yourself, *you* will provide an implementation of `A2Methods` that uses `UArray2`. My implementation is in file `a2blocked.c`; **your implementation goes into** `a2plain.c`, an initial template of which has been provided to you.

The `A2Methods` interface uses the same principles as the declaration of an abstract class in a language like C++, C#, Java, or Smalltalk. Instead of calling functions by name, you will call through *pointers* to functions. Those pointers live in a *method suite* of type `A2Methods_T`, which is a pointer to a record of function pointers (see file `a2methods.h` in `/comp/40/build/include`). For each of these function pointers, you will need to create a `static` function that calls into `UArray2`. To implement your method suite, you put pointers to those functions into a `struct`. Looking at `a2blocked.c` will show you a complete example, and we have already started your method suite for you in `a2plain.c`.

The interface you are to implement is defined in `a2plain.h`:

```
#include <a2methods.h>

extern A2Methods_T uarray2_methods_plain; /* functions for normal arrays */
```

You should **complete file `a2plain.c`, which implements this interface**. It will look quite similar to the implementation for blocked 2D arrays found in `a2blocked.c`.

Here is a summary of your obligations for this part:

- You submit a file `a2plain.c` which exports the single pointer `uarray2_methods_plain`.
- In the methods suite, you *must* implement all the methods from `new` through `at` (some are already done for you).
- Because `UArray2` does not support blocking, you *must not* implement `map_block_major` or `small_map_block_major`. These pointers must be `NULL`.

1.3 Part C: `ppmtrans`, a program with straightforward locality properties

Using the `A2Methods` abstraction, implement program `ppmtrans`, which is modelled on `jpegtran` and performs some simple image transformations. Program `ppmtrans` offers a subset of `jpegtran`'s functionality. The image-transformation options you may support are as follows:

```
-rotate 90
       Rotate image 90 degrees clockwise.

-rotate 180
       Rotate image 180 degrees.

-rotate 270
       Rotate image 270 degrees clockwise (or 90 ccw).

-rotate 0
       Leave the image unchanged.

-flip horizontal
       Mirror image horizontally (left-right).

-flip vertical
       Mirror image vertically (top-bottom).

-transpose
       Transpose image (across UL-to-LR axis).

-time <timing_file>
       Create timing data (see Section 1.5 below) and store
       the data in the file named <timing_file>.
```

You must implement 0-degree, 90-degree and 180-degree rotations and the `-time` option. Other options **may be implemented for extra credit**; if you choose not to implement them, reject the unimplemented options with a suitable error message written to `stderr` and a nonzero exit code. As usual, successful execution of an implemented option should result in an exit code of EXIT_SUCCESS, which is 0. If `ppmtrans` is run with no options, it defaults to a 0-degree rotation. If a filename is supplied and

that filename cannot be opened for reading, the result must be an explanatory message on `stderr` and an exit with code EXIT_FAILURE. No output may be written to `stdout` or `stderr` except where instructions elsewhere in this specification specifically allow for or require it.

Significant requirements:

- Your program must also **recognize and implement these options**:

```
-row-major
       Copy pixels from the source image using map_row_major
-col-major
       Copy pixels from the source image using map_col_major
-block-major
       Copy pixels from the source image using map_block_major
```

  If none of these options is provided, follow the example of the argument handling code below, which uses `methods->map_default`.

- You **must not call `UArray2` functions or `UArray2b` functions directly**. Instead you must call *indirectly* through the function pointers in a methods suite.

- You **must use the mapping functions defined in `a2methods.h` to perform the transform**, *not* nested `for` loops. (Reading the PPM file is always done in row-major order, because that's how the pixels are stored in the file.)

- For row-major and column-major mapping, you will use the methods suite `uarray2_methods_plain` that you will have created in Part B of this homework. For block-major mapping, you will use the methods suite `uarray2_methods_blocked` that we provide in interface `a2blocked.h`.

Your `ppmtrans` should read a single `ppm` image either from standard input or from a file named on the command line.

Your `ppmtrans` should write the transformed image to standard output. Note that your image will be written in the binary (not plain) `ppm` format, and that we give you code (described below) that will do this correctly. Specifically, you must use the services defined in `pnm.h` to read, write and free the image files. Also, you do not need to do any explicit error handling relating to incorrect image file formats: you may rely on the services defined in `pnm.h`, and you should not be catching the exceptions it raises. For help handling command-line options, see the suggested code at the end of this assignment.

Why this problem is interesting from a cache point of view:

> *If cells in a row are stored in adjacent memory locations, processing cells in a row has good spatial locality, but it's not clear about processing cells in a column. If cells in a column are stored in adjacent memory locations, processing cells in a column has good spatial locality, but it's not clear about processing cells in a row. In a 90-degree rotation, processing a row in the source image means processing a column in the destination image, and vice versa. Thus, the locality properties of 90-degree rotation are not immediately obvious.*
>
> *In a 180-degree rotation, rows map to rows and columns map to columns. Thus, whatever locality properties are enjoyed by the source-image processing are enjoyed equally by the destination-image processing. If you understand how your data structure works, then, you should find it easier to predict the locality of 180-degree rotation.*
>
> *In a blocked representation, the mapping of blocks to blocks is not obvious. To understand the locality properties of blocked array processing, you will have to think carefully.*

My solution to this problem is about 150 lines of code.

1.4 Part D (experimental): Analyze locality and predict performance

This part of the assignment is to be completed and turned in at the same time as your design submission. Please **estimate the expected cache hit rate for reads** of each of the six operations in the table below. Assume that the images being rotated are **much too large to fit in the cache**.

	row-major access (UArray2)	column-major access(UArray2)	blocked access (UArray2b)
90-degree rotation			
180-degree rotation			

The first two columns should be your estimate of how your `UArray2` implementation would perform with row-major and column-major access respectively. The third column should be for block-major access, which obviously is for your `UArray2B` implementation.

Each estimate should be a **rank between 1 and 6**, with 1 being the best hit rate and 6 being the worst hit rate. If you think two operations will have about the same hit rate, give them the same rank. For example, if you think that both column-major rotations will have the most cache misses and will have about the same number of cache misses, rank them both 5 and rank the other entries 1 to 4.

Justify your estimates on the grounds of **expected cache misses** and **locality**. *Your justifications will form a significant fraction of your grade for this part.*

Unfortunately, measuring (versus estimating) hit rates is not so easy. In industry, they would measure performance against simulators (`valgrind` has cache simulation tools, for example) and then collect data from the actual hardware. We will not do that here, but you will measure your program's performance in Part E below.

To complete this problem successfully, you will need to understand the material presented in class and in Chapter 6 of Bryant and O'Hallaron.

1.5 Part E (experimental): Measure and explain improvements in locality

To help understand the performance of your program, you have implemented the optional `-time <filename>` command line option. If this argument is given, then your program will produce timing data that can later be analyzed.

- **Carefully follow the instructions at our locality timing tutorial to learn about and integrate our timing library into your `ppmtrans.c`**
- You will find a very small supply of large images in `/comp/40/bin/images/large`.
- You can create your own large image by using any JPEG file with `djpeg` and `pnmscale`. Experiment until you get something of reasonable size, I.e. one that takes enough CPU time to show interesting results, and is not too big to be practical. A few seconds of CPU time is ideal.

  Example command lines:

  ```
  djpeg /comp/40/bin/images/from-wind-cave.jpg | pnmscale 3.5 |
     ./ppmtrans -rotate 90 | display -
  djpeg /comp/40/bin/images/wind-cave.jpg | pnmscale 1.2 |
     ./ppmtrans -rotate 90 | display -
  ```

- If you need to store a large image, you can create files and directories in the `/data` area, and they will not count against your disk quota. Note that `/data` directories tend to be private per machine, so if you switch machines you might not find information you left in the `/data` directory of another.
- **Remember, the instructions at `http://www.cs.tufts.edu/comp/40/docs/locality-timing.html` have told you how to instrument `ppmtrans.c` so that it appends accurate timing information to a named timing file. Be sure that you write your timing information to the file named with the `-timing` command line switch.**

Choose an image at a given scale and be sure to time *all* rotations on that image at the **same scale**. If you like, you can also do the same for other images of several different sizes: small ones may fit in caches and larger ones might not.

Note that our timing library records **CPU time**, which is the time the computer spends actually running your program. The time reported should, for the most part, be independent of whether anyone else is using the machine at the same time. Do make sure to do all your reported tests on the same machine, and indicate in your report which machine it is. Some computers are faster than others, of course, and those will report lower CPU times for a given input.

Always record not only the total time taken for each rotation, but also compute the time per input pixel. Since the number of total input pixels is `width * height` of your input image it should be trivial to have your `ppmtrans` output the number of pixels and time per pixel along with the total time. DO NOT SEPARATELY START AND STOP THE TIMER FOR EACH PIXEL! Just measure the time for the whole transformation, and then divide by total pixels to compute the average time per pixel.

There's an interesting lesson here: there is a little unavoidable fixed overhead in starting and stopping the timers. If you try to time the very small amount of work for moving one pixel, that overhead becomes relatively significant. If you time the work to transform the entire image, and then divide to get the average time per pixel, then the overhead of starting and stopping is amortized over the much longer computation being measured.

Why bother to compute the time per pixel at all? Why not just report the total for the whole image? The latter will obviously vary a lot between small and large images, but the average time to move a pixel can be directly compared. Think a bit: do you expect the time per pixel to be the same for small and large images?

2 Infrastructure that we provide

This section identifies infrastructure you can use for this assignment.

2.1 A polymorphic interface to two-dimensional arrays

1. The file `a2methods.h` in `/comp/40/build/include` (which should not be copied or moved) gives a polymorphic interface that describes a method suite for two-dimensional arrays. You will provide a method suite that works with your design and implementation of `UArray2`; we provide an implementation that works with `UArray2b`. For a rather simple example of how to use the 2D-array methods, see sample file `a2test.c` (provided to you in your starter code).

2.2 Other interfaces we have designed for you

The interfaces below appear in `/comp/40/build/include`. **Do not copy these files**. You should be able to compile against any of these interfaces by using the option `-I/comp/40/build/include` with `gcc`.

2. Files `a2plain.h` and `a2blocked.h` define two interfaces that promise method suites. We implement `a2blocked.h`; you should be able to link against our implementation using the options `-L/comp/40/build/lib -l40locality` with `gcc`.

3. File `uarray2b.h` defines the `UArray2b` interface. (You write the implementation.)

4. File `pnm.h` defines functions you can use to read, write, and free portable pixmap (PPM) files (see `/comp/40/build/include/pnm.h` for the interface). It defines a representation for colored pixels. The pixmap itself is represented as type `void *`; you will use this code with the `A2Methods_T` methods.

   The `Pnm` interface uses the `A2Methods` interface.

   You should be able to link against our implementation of `pnm.h` by using the options

```
-L/comp/40/build/lib -l40locality -lnetpbm
```

with `gcc`. (`Pnm_ppmread`, etc., have implementations in the `40locality` library we provide. These implementations in turn require things in the `netpbm` library.)

2.3 Test code for two-dimensional arrays

5. As usual when implementing polymorphism in C, it is possible to make a mistake with `void *` pointers. You will therefore want to **run small test cases using `valgrind`** in order to flush out potential memory errors. We provide one sample test case in file `a2test.c`; it tests the `cell` and `at` methods as well as row-major mapping, if present. Before you can use it you will need to implement `uarray2b.c` or `a2plain.c` or both.

   Once you can build `a2test`, run `valgrind ./a2test`.

2.4 Other source code

6. We provide C source code for `a2blocked.c`, which you don't need to compile, but you might find useful to study. We also provide incomplete versions of `a2plain.c` and `ppmtrans.c`.

2.5 Where to get what

7. The starter code we provide also contains a `Makefile` that builds `a2test`; you will need to extend the script to build `ppmtrans`.

   As described at the start of this document, you get all these sources by running:

   ```
   pull-code locality
   ```

   We recommend you *begin* the assignment by creating a new directory to use for this assignment.

2.6 Geometric calculations we have done for you

What's important about this assignment is how locality stores affects performance, not how to rotate images. We therefore inform you that we believe

8. If you have an original image of size $w \times h$, then when the image is rotated 90 degrees, pixel (i, j) in the original becomes pixel $(h - j - 1, i)$ in the rotated image.

9. When the image is rotated 180 degrees, pixel (i, j) becomes pixel $(w - i - 1, h - j - 1)$.

3 What we expect from your preliminary submission

In your first assignment, we provided you with a partial implementation plan for `restoration`. We are now turning the full implementation planning over to you. Your preliminary submission will consist of two parts:

1. A detailed **implementation plan** for Part C, `ppmtrans` (1 - 2 pages). Our expectation is that you will have already thought about the representations you will be using and the ways in which different pieces of your code will interact. This should be evident in your implementation plan. We expect to see you planning for a modular design that can be incrementally tested. For each step of your implementation plan, please describe the testing that will be done before you move on to the next step (you do **not** need to provide time estimates for each step).

2. All of your estimates and explanations for Part D. Please refer to Part D for the detailed specification.

Please submit your implementation plan and estimates in a file called `design.pdf` via the command `submit40-locality-desi`

4 What we expect from your final submission

Your **implementation**, to be submitted using `submit40-locality`, should include

1. 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
   - Identifies what has been correctly implemented and what has not
   - Documents the architecture of your solutions.
   - **Gives measured performance** for Part E and **explains** your results. Be sure to record in a table the following:
     - The sizes of the images used for each of your tests
     - The total CPU time and the time per input pixel for each rotation on each image you report. Making the simplifying assumption that our computers execute approximately 1,000,000,000 instructions per second, estimate for each rotation the number of instructions executed per input pixel rotated.
     - The name and model of the computer you ran your tests on as well as the CPU type and clock rate. (`more /proc/cpuinfo`)

     Below your table, discuss patterns you see in the results: for example, do certain rotations run faster when using blocked arrays vs. plain? If you try different sized images (not required but useful), is the number of instructions per pixel similar regardless of image size? If not, why not?
   - Says **approximately how many hours you have spent** completing the assignment

2. (Optional) If you happen to know how to use a spreadsheet or GNUPlot to create bar charts, you're welcome to include a few PDFs with charts of the interesting data from your tables. This is NOT required, but it might help you tell your story.

3. The file `ppmtrans.c`.

4. File `uarray2b.c`, which implements the `UArray2b` interface. This file should include internal documentation explaining your representation and its invariants.

5. File `a2plain.c`, which provides a method suite as described by the `a2methods.h` interface.

6. Any other files you may have created as useful components.

7. A `Makefile`, which, when run using

   ```
   make
   ```

   encounters no errors and builds *three* executable binaries: `a2test`, `timing_test` and `ppmtrans`.

   - `ppmtrans` should be linked with `ppmtrans.o`, `uarray2.o`, `uarray2b.o`, `a2plain.o`, `a2blocked.o`, and probably with other relocatable object files and libraries.

5 Avoid common mistakes

Here are the mistakes most commonly made on this project:

- It's a mistake to submit, in place of a succint function contract or a description of a single point of truth, a narrative description of a sequence of events.

- It's a mistake to try to explain a complex representation choice (e.g., where are elements stored in a UArray2?) in informal English.
- It's a mistake to analyze a rotation experiment if the rotation completes in less than a second or so.
- When two programs perform very differently, and the programs have very different loop structures, it's a mistake to try to explain performance differences by appealing to locality.

6 Code to handle command-line options and choose methods

To deal with command-line options in `ppmtrans.c`, consider the code below. This code does not help you decide if a file has been named on the command line, which determines whether you read from that file or from standard input. To make this decision, you will need to examine the values of `i` and `argc`.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "assert.h"
#include "a2methods.h"
#include "a2plain.h"
#include "a2blocked.h"
#include "pnm.h"



/*
 * SET_METHODS is a macro that will set the methods variable
 * and map variable in main to the proper to the proper values,
 * which depends on the arguments.
 * If you're curious about how this works, look up "C Macros" for
 * more information.
 */
#define SET_METHODS(METHODS, MAP, WHAT) do {                            \
                methods = (METHODS);                                    \
                assert(methods != NULL);                                \
                map = methods->MAP;                                     \
                if (map == NULL) {                                      \
                        fprintf(stderr, "%s does not support "          \
                                        WHAT "mapping\n",               \
                                argv[0]);                               \
                        exit(1);                                        \
                }                                                       \
        } while (0)


static void usage(const char *progname)
{
        fprintf(stderr, "Usage: %s [-rotate <angle>] "
                "[-{row,col,block}-major] [filename]\n",
                progname);
        exit(1);
}

int main(int argc, char *argv[])
{
        char *time_file_name = NULL;
        int   rotation = 0;
        int   i;
```

```
        /* default to using UArray2 methods */
        A2Methods_T methods = uarray2_methods_plain;
        assert(methods);

        /* default to using best map */
        A2Methods_mapfun *map = methods->map_default;
        assert(map);

/* parse through arguments */
        for (i = 1; i < argc; i++) {
                if (strcmp(argv[i], "-row-major") == 0) {
                        SET_METHODS(uarray2_methods_plain, map_row_major,
                                    "row-major");
                } else if (strcmp(argv[i], "-col-major") == 0) {
                        SET_METHODS(uarray2_methods_plain, map_col_major,
                                    "column-major");
                } else if (strcmp(argv[i], "-block-major") == 0) {
                        SET_METHODS(uarray2_methods_blocked, map_block_major,
                                    "block-major");
                } else if (strcmp(argv[i], "-rotate") == 0) {
                        if (!(i + 1 < argc)) {      /* no rotate value */
                                usage(argv[0]);
                        }
                        char *endptr;
                        rotation = strtol(argv[++i], &endptr, 10);
                        if (!(rotation == 0 || rotation == 90
                              || rotation == 180 || rotation == 270)) {
                                fprintf(stderr, "Rotation must be "
                                        "0, 90 180 or 270\n");
                                usage(argv[0]);
                        }
                        if (!(*endptr == '\0')) {    /* Not a number */
                                usage(argv[0]);
                        }
                } else if (strcmp(argv[i], "-time") == 0) {
                        time_file_name = argv[++i];
                } else if (*argv[i] == '-') {
                        fprintf(stderr, "%s: unknown option '%s'\n", argv[0],
                                argv[i]);
                } else if (argc - i > 1) {
                        fprintf(stderr, "Too many arguments\n");
                        usage(argv[0]);
                } else {
                        break;
                }
        }

/* time to start rotating */
         ...
}
```