Lab 9: 2-(and Higher)D Arrays in C

"Out of the box", C treats multidimensional arrays (arrays of more than one dimension such as images or matrices) not as well as it should:

- There is no bounds checking. Certainly not within the compiler, but even `valgrind` won't detect the error when one index exceeds its bounds, as long as the result lies within the memory allocated for the array. For example, given an array a declared `int a[2][2]`, a reference to `a[0][3]` is not flagged.
- There is no way to declare an array of more than two dimensions when those dimensions are not known until run time. (GNU C has some help for this, but it's rather cumbersome.)

Fortunately, however, C does allow us to add some support features that allow us to work with multidimensional arrays more easily. In this lab, we'll use them to implement dynamically-sized, 2D arrays with bounds checking.

Files for this lab will be in

http://www.tricity.wsu.edu/~bobl/cpts360/lab09_arrays

(There is also a link to it in the schedule on the course web page.) Calling it up in a browser will allow you to download all of the files you need for this lab. `writeup.html` in that directory contains an HTML version of this page.

Every part after Part 1 depends on the part that went right before it, so be sure that part is working before you move on.

Part 1: Generating a 2D Image From a Height Field

Here's the working program `plotz2_pt1`:

```
/*
 * plotz2 -- 2D z-value image generator
 */

#include <stdio.h>
#include <math.h>

#define N_PIXEL_1D 7

void getExtrema(double z[N_PIXEL_1D][N_PIXEL_1D], double *zMin_p,
                double *zMax_p)
/* return the minimum and maximum values in a 2D double array */
{
    /*
     * input
     *    z[][] -- an N_PIXEL_1D x N_PIXEL_1D array of double values
     *
     * output
     *    *zMin_p - the minimum value in z[][]
     *    *zMax_p - the maximum value in z[][]
     */
```

```
    int i, j;

    (*zMin_p) = (*zMax_p) = z[0][0]; // initialize
    for (i = 0; i < N_PIXEL_1D; i++) { // for all rows i
        for (j = 0; j < N_PIXEL_1D; j++) { // for all columns j
            if (z[i][j] < (*zMin_p)) {
                (*zMin_p) = z[i][j]; // update minimum
            } else if ((*zMax_p) < z[i][j]) {
                (*zMax_p) = z[i][j]; // update maximum
            }
        }
    }
}

void printSquarePgm(double z[N_PIXEL_1D][N_PIXEL_1D])
/* print a double array on stdout as a PGM file with automatic scaling */
{
    /*
     * input
     *    z[][] -- an N_PIXEL_1D x N_PIXEL_1D array of double values
     *
     * This function first finds the maximum and minimum values in
     * z[][], zMin and zMax, with a call to getExtrema(). It then
     * prints out the values of the array as a PGM file. The PGM file
     * format is as follows:
     *
     * line #  contains
     * ------  --------
     *   1      "P2"
     *   2      "w h" where "w" is the width of the image and "h" is the
     *           height. In this case, both values are N_PIXEL_1D.
     *   3      "255"
     *   4      scaled (see below) values of z[0][...]
     *   5      scaled values of z[1][...]
     *  ...     ...
     *  N_PIXEL_1D+3     scaled values of z[N_PIXEL_1D-1][...]
     *
     * Before printing, each value of z[][] is scaled and converted to
     * an int "pixelValue" between 0 and 255 (inclusively).
     */
    double zMin, zMax;
    int i, j;
    int pixelValue;
    int maxVal = 255; /* should be at least this, greater is okay */

    getExtrema(z, &zMin, &zMax);

    printf("P2\n");
    printf("%d %d\n", N_PIXEL_1D, N_PIXEL_1D);
```

```
    printf("%d\n", maxVal);
    for (i = 0; i < N_PIXEL_1D; i++) {
        for (j = 0; j < N_PIXEL_1D; j++) {
            if (zMin == zMax) {
                pixelValue = 128;
            } else {
                pixelValue = maxVal * (z[i][j] - zMin) / (zMax - zMin);
            }
            printf("%3d ", pixelValue);
        }
        printf("\n");
    }
}

void sampleFunction(double (*f_p)(double x, double y),
                    double z[N_PIXEL_1D][N_PIXEL_1D])
// sample an N_PIXEL_1D x N_PIXEL_1D grid over the square
// [-1,1] x [-1,1] */
{
    /*
     * input:
     *    f -- pointer to a function that will be called over the grid
     *
     * output:
     *    z -- N_PIXEL_1D x N_PIXEL_1D array of values of f
     *         evaluated over [ -1, 1 ] x [ -1, 1 ]. z[0][0]
     *         corresponds to the upper left corner (x, y) = (-1, 1).
     *
     * This function evaluates an N_PIXEL_1D x N_PIXEL_1D grid that
     * fits into a 2 x 2 square centered on the origin.
     */
    double x, dx, y, dy;
    int i, j;

    dx = 2.0 / (N_PIXEL_1D - 1);
    dy = 2.0 / (N_PIXEL_1D - 1);
    for (i = 0; i < N_PIXEL_1D; i++) {
        y = 1.0 - dy * i;
        for (j = 0; j < N_PIXEL_1D; j++) {
            x = dx * j - 1.0;
            z[i][j] = (*f_p)(x, y);
        }
    }
}

double hemisphere(double x, double y)
/* return the height of a unit hemisphere or 0 if (x,y) lies outside */
{
    double rSqrd = 1.0 - x*x - y*y;
```

```
    if (rSqrd > 0)
        return sqrt(rSqrd);
    return 0.0;
}

double ripple(double x, double y)
/* return a radial, exponentially-damped cosine, or "ripple" */
{
    double r = sqrt(x*x + y*y);
    return exp(-2*r) * cos(20*r);
}

/*
 * Add your own function here, if you wish. Be creative!
 */

int main(int argc, char *argv[])
{
    /*
     * Windows won't let us declare local (i.e. stack) variables this
     * large (Linux will -- so there!), so we need to declare them
     * static (i.e. global) for portability.
     */
    static double z[N_PIXEL_1D][N_PIXEL_1D];

    // Instead of "hemisphere", try "ripple" or your own function's
    // name.
    sampleFunction(hemisphere, z);

    printSquarePgm(z);

    return 0;
}
```

This program will create a 2D image that visualizes any function given to it over the [-1, 1] x [-1, 1] square.

This creates an image file in the PGM (Portable Gray Map) format, a very easy-to-implement, well-established format for grayscale graphics.

Download this file `plotz2_pt1.c` and develop a `Makefile` for it. Compile it and run it. You should see output like this:

```
P2
7 7
255
  0   0   0   0   0   0   0
  0  84 169 190 169  85   0
  0 169 224 240 224 170   0
  0 190 240 255 240 190   0
  0 169 224 240 224 170   0
```

```
  0  85 170 190 170  85   0
  0   0   0   0   0   0   0
```

This is actually an encoding of a gray image of a hemisphere (see the `hemisphere()` function), although to see it better, do the following:

- Change the `#define` of `N_PIXEL_1D` from 7 to 512.
- Re-`make` the executable.
- If you're running in the lab on a Linux platform or remotely on `cslab` or over with an X11 connection (e.g. via MobaXTerm), run the following:

```
$ ./plotz2_pt1 >image.pgm
$ eog image.pgm
```

Cool, eh? (Well, *I* thought so.) This will work for any function `f(x,y)` defined over that 2 x 2 square centered on the origin. There's another function `ripple()` in the package that you can visualize by changing the first argument in the `sampleFunction()` call. Try creating your own 2D function, if you like.

One shortcoming of this program is that in order to change the size of the image, you have to modify the source and recompile. The goal of this lab is to make that size selectable by a command line option. We'll do this in several steps.

For grading purposes, set the function back to `hemisphere()` and `N_PIXEL_1D` back to 7 in your modified `plotz2_pt1.c` when you put it and your makefile in the tarball.

Part 2: Allowing a User-Selectable Image Size

The first step is to replace the "`#define N_PIXEL_1D`" with a global `int nPixel1D;`. Copy `plotz2_pt1.c` to `plotz2_pt2.c` and make these changes to the latter, organized by the part of the file they take place in:

globals

- Include "`assert.h`".
- Replace:

```
#define N_PIXEL_1D 7
```

  with:

```
int nPixel1D = 512;
```

  so that the default remains the same (512). (It's a good idea to keep it at 7 for debugging.)

everywhere

- Change all references to `N_PIXEL_1D` to refer to `nPixel1D`.

in `main()`

- Replace the `z` array declaration with `double *z;`, so that `z` becomes a pointer.
- Use the `ALLOC_ARRAY()` macro (see `allocarray.h`) to allocate `nPixel1D * nPixel1D` `doubles` for `z` to point to.

in `getExtrema()`, `printSquarePgm()`, and `sampleFunction()`

- Replace the 2D array declaration in the parameter list with pointer declarations. This means replacing:

  ```
  double z[N_PIXEL_1D][N_PIXEL_1D]
  ```

  with:

  ```
  double *_z
  ```

  The "`_`" prefix is a convention: a sort of warning to future programmers that they should not use that variable arbitrarily (see next item).

- At the very beginning of each function, right after the prototype, enter this:

  ```
  #define z(i,j) _z[nPixel1D*(i) + (j)]
  ```

  Note that this `z(i,j)` can be set and used in an expression just like `z[i][j]`, except that it now allows `nPixel1D` to set the number of columns rather than requiring it to be a constant.

  This is how you define variably-sized multidimensional arrays in C.

- Throughout the rest of the function, replace all references to `z[i][j]` (where `i` and `j` are any expressions) with (the macro invocation) `z(i,j)`.
- If the original `z` appears without "`[`" and "`]`" in a function call, replace it with `&z(0,0)`.
- At the very end of each function, add:

  ```
  #undef z
  ```

  as it's couth to make your `z(i,j)` definitions local to a single function.

in the `Makefile`

- Enhance the `Makefile` to build `plotz2_pt2` as well as `plotz2_pt1`.

Be sure to include `plotz2_pt2.c` in the tarball.

Part 3: Getting Rid of the Global Array Size

In this part, we'll allow `nPixel1d` to be set from the commaond line. Also, although it's not doing much harm here, let's get rid of the `nPixel1D` global to make our code easier to reuse: Each function will now depend only on its parameters and no globals. This is couth.

Copy `plotz2_pt2.c` to `plotz2_pt3.c` and make these changes to the latter, organized by the part of the file they take place in:

globals

- Delete the "`int nPixel1D;`" global definition.

in `main()`

- Add a declaration "`int nPixel1D = 512;`". (This sets the default value.)
- If `argv[1]` is provided (check `argc`), convert it (a `char *`) to (an `int`), which you assign to `nPixel1D` (hint: `atoi(3)`) to permit the user to specify a different value for `nPixel1D` on the command line. You may add error checking if you like, but it is not required.

in `getExtrema()`, `printSquarePgm()`, and `sampleFunction()`

- Add an `int nPixel1D` parameter right after the `_z` parameter. (By C convention, size parameters typically follow the things they're the size of.)

everywhere

- Make all calls to the above functions pass `nPixel1D`.

in the `Makefile`

- Enhance the `Makefile` to build `plotz2_pt3` as well as `plotz2_pt2` and `plotz2_pt1`.

Part 4: Painlessly Adding Bounds Checking

Even though the code we have works, there's one other thing you should add to make it "bulletproof": bounds checking. This guarantees that none of the indices exceed the permitted values for its dimension: Given a (macro) reference `a(i,j)` to an `N` by `M` array, `i` should be between `0` and `N-1` and `j` should be between `0` and `M-1` (both inclusive). The way things stand now, C doesn't check this at all, and even `valgrind` won't say anything as long as "`M*i + j`" is between `0` and `N*M - 1` (inclusive).

A good way to do this is with the macro `I2D()`, which is defined in the `i2d.h` file. Here it is:

```
#include <assert.h>

/*
 *  I2D_C_STD(i, j, m) converts the 2D index (i,j) into a 1D index
 *  into a (1D) C array representing a 2D {anything} x m array using
 *  the normal C multidimensional conventions.  It does not perform
```

```
 *  bounds checking.
 */
#define I2D_C_STD(i, j, m) ((m)*(i) + (j))

/*
 *  I2D is our basic 2D indexing macro.  Use it to index all 1D
 *  representations of 2D arrays (usually within other macros).
 *  It includes bounds checking.
 */
#ifdef __GNUC__
/*
 *  If GNU C is available, take advantage of its ability to encode
 *  statements within expressions (note the "({" and "})"), which is
 *  non-standard C, but a very handy idea!
 */
#define I2D(i, n, j, m) ({ \
    /* make temporary copies in case of expressions or side effects */ \
    int _i = (i); \
    int _n = (n); \
    int _j = (j); \
    int _m = (m); \
\
    assert(0 <= _i && _i < _n); \
    assert(0 <= _j && _j < _m); \
    I2D_C_STD(_i, _j, _m); })
#else
/*
 *  Doing bounds checking in inline functions, which are in the C99
 *  standard, is less satisfactory than doing it in macros because
 *  assertion failure yields the location of the assert() in the
 *  inline definition, not the place in the "calling" code where the
 *  bounds check was violated.
 */
static inline int I2D(int i, int n, int j, int m)
{
    assert(0 <= i && i < n);
    assert(0 <= j && j < m);
    return I2D_C_STD(i, j, m);
}
#endif
```

As you can see, `I2D()` wraps the same calculation you're doing now to compute a one-dimensional index from two indices `i` and `j` with assertion checks on the individual bounds. Note the use of the `__GNUC__` define to detect when you're compiling with GNU C and can therefore make use of its handy `({ ... })` feature.

Copy `plotz2_pt3.c` to `plotz2_pt4.c` and make these changes to the latter, organized by the part of the file they take place in:

globals

- Include "`i2d.h`", of course.

in `getExtrema()`, `printSquarePgm()`, and `sampleFunction()`

- Modify the `z` definitions to invoke `I2D()` when they compute the index of `_z`, like this:

```
#define z(i,j) _z[I2D(i, nPixel1D, j, nPixel1D)]
```

- Temporarily insert a bounds check violation into the code, compile it, and run it to make sure it was detected. Be sure to remove it in your submission.

in the `Makefile`

- Enhance the `Makefile` to build `plotz2_pt4` as well as `plotz2_pt3`, `plotz2_pt2`, and `plotz2_pt1`.

There's really little to do here to get bounds checking. That's sort of the point.

Include `plotz2_pt4.c`, `plotz2_pt3.c`, `plotz2_pt2.c`, `plotz2_pt1.c`, `i2d.h`, `allocarray.h` and the final comprehensive version of the `Makefile` in the tarball and submit the tarball via Canvas.