CHAPTER 12

ARRAYS OF ARRAYS

A TWO-DIMENSIONAL ARRAY

`scores[0][0]`	`scores[0][1]`	`scores[0][2]`
`scores[1][0]`	`scores[1][1]`	`scores[1][2]`
`scores[2][0]`	`scores[2][1]`	`scores[2][2]`
`scores[3][0]`	`scores[3][1]`	`scores[3][2]`
`scores[4][0]`	`scores[4][1]`	`scores[4][2]`
`scores[5][0]`	`scores[5][1]`	`scores[5][2]`
`scores[6][0]`	`scores[6][1]`	`scores[6][2]`
`scores[7][0]`	`scores[7][1]`	`scores[7][2]`
`scores[8][0]`	`scores[8][1]`	`scores[8][2]`
`scores[9][0]`	`scores[9][1]`	`scores[9][2]`

ioscores.h

(Figure 12.2 – Page 283)

```
/*
 * Prototypes for score manipulating functions.
 */
#define MAX_TESTS 3

extern int read_scores(int s[][MAX_TESTS], int maxrow);
extern void print_scores(int s[][MAX_TESTS], int maxrow);
```

ioscores.c

(Figure 12.3 – Page 283)

```
/*
 * Functions to read and print a 2-d array of student test scores.
 *   read_scores - read scores into array, one input score per line.
 *   print_scores - write scores from array, one line per student.
 */
#include <stdio.h>
#include "ioscores.h"

#define  MAXLEN   80

int read_scores(int s[][MAX_TESTS], int maxrow)
{
  int  getline(char *buf, int len);
  int  student, test;                 /* our indices */
  char line[MAXLEN + 1];

  for (student = 0; student < maxrow; student++)
    for (test = 0; test < MAX_TESTS; test++)
      if (getline(line, MAXLEN) == -1 ||
            sscanf(line, "%i", &s[student][test]) != 1)
        return -student;              /* negative indicates problem */
  return maxrow;                      /* successfully read all scores */
}

void print_scores(int s[][MAX_TESTS], int maxrow)
{
  int student, test;                  /* our indices */

  for (student = 0; student < maxrow ; student++)
    for (test = 0; test < MAX_TESTS; test++)
      printf("%i%c", s[student][test],
                 (test != MAX_TESTS - 1) ? ' ' : '\n');
}
```

tstscrs.c

(Figure 12.4 – Page 284)

```
/*
 * Read and print student scores.
 */
#include <stdio.h>
#include "ioscores.h"

#define MAX_STUDENTS 10

int main()
{
  int scores[MAX_STUDENTS][MAX_TESTS];

  if (read_scores(scores, MAX_STUDENTS) == MAX_STUDENTS)
    print_scores(scores, MAX_STUDENTS);
  else
    printf("Couldn't read all students successfully\n");
  return 0;
}
```

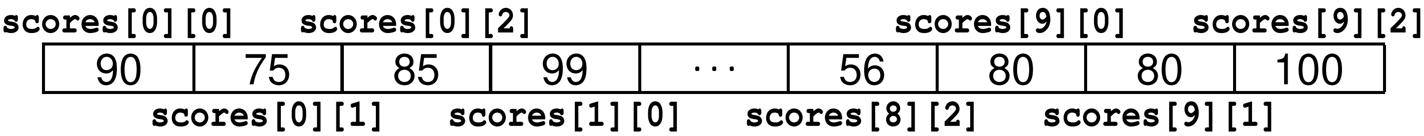
INITIALIZING A TWO-DIMENSIONAL ARRAY

`scores[0][0]`

90	75	85
99	99	95
65	69	66
100	100	100
56	60	70
78	85	90
87	82	97
65	80	70
75	78	56
80	80	100

`scores[9][2]`

INTERNAL LAYOUT OF A TWO-D ARRAY



init2d.c

(Figure 12.7 – Page 287)

```
/*
 * A program that uses two different ways to initialize a 2-d array.
 *   twod_init - initialize a two-d array using array subscripting.
 *   table_init - initialize a two-d array using pointers.
 */
#include <stdio.h>

#define ITERATIONS  50
#define MAXROWS    150
#define MAXCOLS    200

int main()
{
  void twod_init(int table[][MAXCOLS], int rows, int value);
  void table_init(int *ptr, int *endptr, int value);
  int t[MAXROWS][MAXCOLS];             /* table to initialize */
  int i;                               /* count */

  for (i = 0; i < ITERATIONS; i++)     /* first try slow way */
    twod_init(t, MAXROWS, 5);
  for (i = 0; i < ITERATIONS; i++)     /* then try faster way */
    table_init(&t[0][0], &t[MAXROWS - 1][MAXCOLS - 1], 5);
  return 0;
}

void twod_init(int table[][MAXCOLS], int rows, int value)
{
  int row, col;

  for (row = 0; row < rows; row++)
    for (col = 0; col < MAXCOLS; col++)
      table[row][col] = value;
}

void table_init(int *ptr, int *endptr, int value)
{
  for (; ptr < endptr; *ptr++ = value)
    ;
}
```

Sample Output – usetest

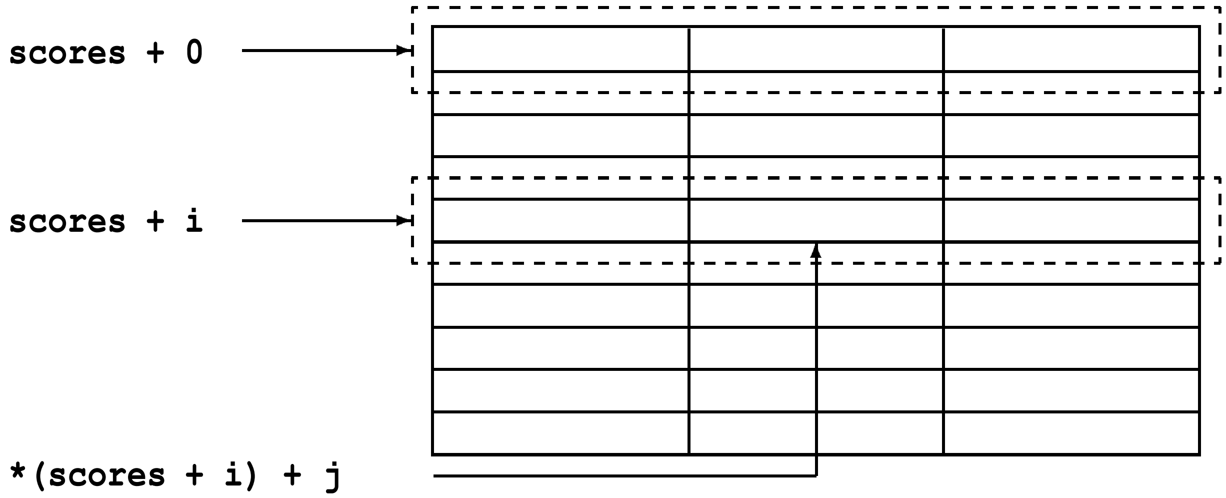
```
Average on test 0 is 79.50
Average on test 1 is 80.80
Average on test 2 is 82.90
```

usetest.c

(Figure 12.9 – Page 289)

```
/*
 * Print average on each test.  For simplicity, the initial scores
 * are hard-coded into the array rather than read.
 */
#include <stdio.h>
#include "ioscores.h"

int main()
{
  double test_avg(int s[][MAX_TESTS], int rows, int n);
  int scores[][MAX_TESTS] = {{90, 75, 85}, {99, 99, 95},
                             {65, 69, 66}, {100, 100, 100},
                             {56, 60, 70}, {78, 85, 90},
                             {87, 82, 97}, {65, 80, 70},
                             {75, 78, 56}, {80, 80, 100}};
  const int n = sizeof(scores) / (sizeof(int) * MAX_TESTS);
  int i;

  for (i = 0; i < MAX_TESTS; i++)
    printf("Average on test %i is %.2f\n", i, test_avg(scores, n, i));
  return 0;
}
```

POINTER ACCESS TO TWO-D ARRAYS



testavg1.c

(Figure 12.10 – Page 290)

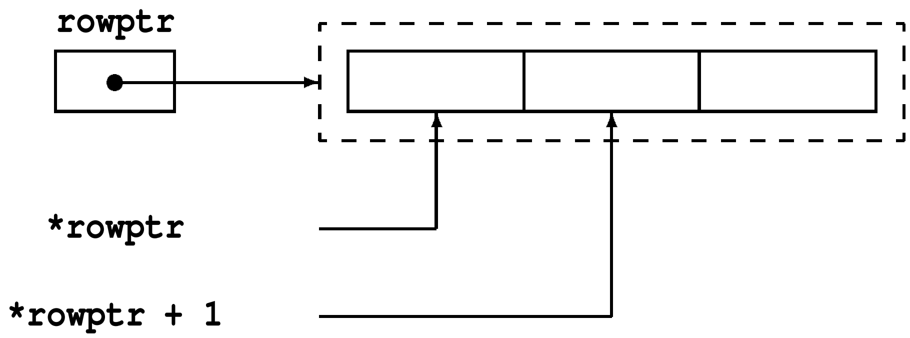
```
/*
 * Compute the average of a column in a 2-d array using subscripting.
 */
#include "ioscores.h"

double test_avg(int s[][MAX_TESTS], int rows, int n)
{
  double sum = 0.0;                           /* column total */
  int    i;                                   /* row index */

  for (i = 0; i < rows; i++)
    sum += s[i][n];
  return sum / rows;
}
```

testavg2.c

(Figure 12.11 – Page 290)

```
/*
 * Compute the average of a column in a 2-d array using pointers.
 */
#include "ioscores.h"

double test_avg(int (*rowptr)[MAX_TESTS], int rows, int n)
{
  double sum = 0.0;                         /* column total */
  int (*endptr)[MAX_TESTS];                 /* will point to last row */

  for (endptr = rowptr + rows; rowptr < endptr; rowptr++)
    sum += (*rowptr)[n];
  return sum / rows;
}
```

ACCESSING THRU A ROW POINTER



Sample Output – userow

```
83.33 97.67 66.67 100.00 62.00 84.33 88.67 71.67 69.67 86.67
```

userow.c

(Figure 12.14 – Page 293)

```
/*
 * Read scores and print average for each student.  As before,
 * for simplicity the test scores are hard-coded, not read.
 */
#include <stdio.h>
#include "ioscores.h"

int main()
{
  double student_avg(int s[][MAX_TESTS], int n);
  int i;
  int scores[][MAX_TESTS] = {{90, 75, 85}, {99, 99, 95},
                             {65, 69, 66}, {100, 100, 100},
                             {56, 60, 70}, {78, 85, 90},
                             {87, 82, 97}, {65, 80, 70},
                             {75, 78, 56}, {80, 80, 100}};
  int n = sizeof(scores) / (sizeof(int) * MAX_TESTS);

  printf("Student\t   Average\n");
  for (i = 0; i < n; i++)
    printf("%4i\t%10.2f\n", i, student_avg(scores, i));
  return 0;
}
```

rowavg1.c

(Figure 12.15 – Page 293)

```
/*
 * Compute row average using array subscripting.
 */
#include "ioscores.h"

double student_avg(int s[][MAX_TESTS], int n)
{
  int i = 0;                              /* index to next row */
  long sum = 0L;                          /* total so far */

  for (; i < MAX_TESTS; sum += s[n][i++])
    ;
  return (double) sum / MAX_TESTS;
}
```

rowavg2.c

(Figure 12.16 – Page 294)

```
/*
 * Compute row average using pointer indexing.
 */
#include "ioscores.h"

double student_avg(int s[][MAX_TESTS], int n)
{
  const int *colptr = &s[n][0];                 /* ptr to first item */
  const int *const endptr = colptr + MAX_TESTS;  /* ptr to last item */
  long sum = 0L;                                 /* total so far */

  for (; colptr < endptr; sum += *colptr++)
    ;
  return (double) sum / MAX_TESTS;
}
```

avgtab.c

(Figure 12.17 – Page 295)

```
/*
 * Print average for each student, using our earlier table_average
 * function (again, scores are hard-coded rather than read).
 */
#include <stdio.h>
#include "ioscores.h"

int main()
{
  double table_average(int *ptr, int n);
  int i;
  int scores[][MAX_TESTS] = {90, 75, 85, 99, 99, 95, 65, 69, 66, 100,
                             100, 100, 56, 60, 70, 78, 85, 90, 87, 82,
                             97, 65, 80, 70, 75, 78, 56, 80, 80, 100};
  const int n = sizeof(scores) / (sizeof(int) * MAX_TESTS);

  printf("Student\t   Average\n");
  for (i = 0; i < n; i++)
    printf("%4i\t%10.2f\n", i, table_average(scores[i], MAX_TESTS));
  return 0;
}
```

classes.c

(Figure 12.18 – Page 296)

```
/*
 * Read and print student scores, one student per line.
 */
#include <stdio.h>
#include "ioclass.h"             /* for read_scores/write_scores */

#define MAX_CLASSES  2

int main()
{
  int classes[MAX_CLASSES][MAX_STUDENTS][MAX_TESTS];

  if (read_classes(classes, MAX_CLASSES) == MAX_CLASSES)
    print_classes(classes, MAX_CLASSES);
  else
    printf("Couldn't read all classes successfully\n");
  return 0;
}
```

A THREE-DIMENSIONAL ARRAY

1000	x[0][1][0]	x[0][1][1]	x[0][1][2]	x[0][1][3]	*1st 2-d array*
1008	x[0][1][0]	x[0][1][1]	x[0][1][2]	x[0][1][3]	
1016	x[1][0][0]	x[1][0][1]	x[1][0][2]	x[1][0][3]	*2nd 2-d array*
1024	x[1][1][0]	x[1][1][1]	x[1][1][2]	x[1][1][3]	
1032	x[2][0][0]	x[2][0][1]	x[2][0][2]	x[2][0][3]	*3rd 2-d array*
1040	x[2][1][0]	x[2][1][1]	x[2][1][2]	x[2][1][3]	

ioclass.h

(Figure 12.19 – Page 297)

```
/*
 * Prototypes for functions to read and print class scores.
 */
#include "ioscores.h"

#define MAX_STUDENTS 10

int read_classes(int c[][MAX_STUDENTS][MAX_TESTS], int n);
void print_classes(int c[][MAX_STUDENTS][MAX_TESTS], int n);
```

ioclass1.c

(Figure 12.20 – Page 297)

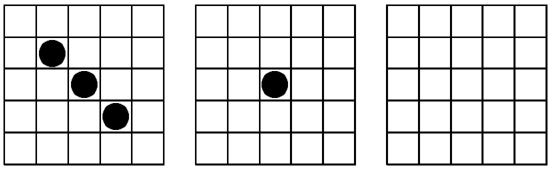
```
/*
 * Functions to read/print a 3-d array.
 *   read_classes - use read_scores to read scores for each class.
 *   print_classes - use print_scores to print scores for each class.
 */
#include <stdio.h>
#include "ioclass.h"

int read_classes(int c[][MAX_STUDENTS][MAX_TESTS], int n)
{
  int class;

  for (class = 0; class < n; class++)
    if (read_scores(c[class], MAX_STUDENTS) != MAX_STUDENTS)
      return -class;
  return class;
}

void print_classes(int classes[][MAX_STUDENTS][MAX_TESTS], int n)
{
  int class;

  for (class = 0; class < n; class++)
  {
    print_scores(classes[class], MAX_STUDENTS);
    putchar('\n');
  }
}
```

ioclass2.c

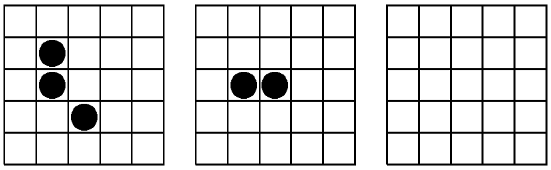
(Figure 12.22 – Page 299)

```c
/*
 * Pointer version of 3-d array reading and printing functions.
 *   read_classes - use read_scores to read scores for each class.
 *   print_classes - use print_scores to print scores for each class.
 */
#include <stdio.h>
#include "ioclass.h"

int read_classes(int (*ptr)[MAX_STUDENTS][MAX_TESTS], int n)
{
  int (*saveptr)[MAX_STUDENTS][MAX_TESTS] = ptr;
  int (*endptr)[MAX_STUDENTS][MAX_TESTS];

  for (endptr = ptr + n - 1; ptr <= endptr; ptr++)
    if (read_scores(*ptr, MAX_STUDENTS) != MAX_STUDENTS)
      return saveptr - ptr;   /* negative value on error */
  return ptr - saveptr;
}

void print_classes(int (*ptr)[MAX_STUDENTS][MAX_TESTS], int n)
{
  int (*endptr)[MAX_STUDENTS][MAX_TESTS];

  for (endptr = ptr + n - 1; ptr <= endptr; ptr++)
  {
    print_scores(*ptr, MAX_STUDENTS);
    putchar('\n');
  }
}
```

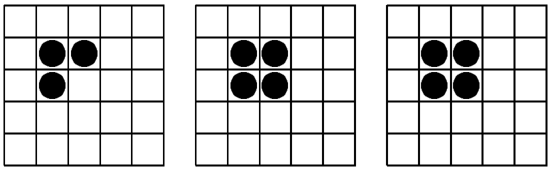
SOME SAMPLE LIFE GAMES



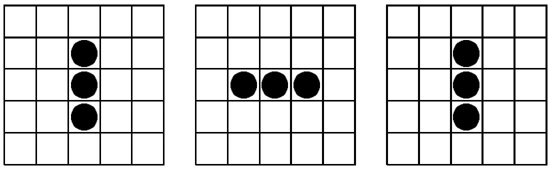
(a) A game that dies out quickly.



(b) A game that dies a little more slowly.



(c) A game that rapidly reaches a stable state.



(d) A game that reaches an oscillating state.

life.h

(Figure 12.24 – Page 302)

```
/*
 * Header file for Life game.
 */
#include <stdio.h>

#define  BORDER   'X'                /* border around world */
#define  MARKER   '#'                /* occupied cell marker */
#define  MAXROW   20                 /* legal rows: 1..20 */
#define  MAXCOL   78                 /* legal columns: 1..78 */
#define  NUMROWS  22                 /* extra cells surrounding */
#define  NUMCOLS  80                 /* (simplify neighbors) */
#define  MAXLEN   80                 /* longest input line */

typedef  short BOOLEAN;
typedef  BOOLEAN WORLD[NUMROWS][NUMCOLS];

typedef  BOOLEAN (*PTR_TO_WORLD)[NUMCOLS];
typedef  const BOOLEAN (*PTR_TO_CONST_WORLD)[NUMCOLS];

#define  TRUE     1
#define  FALSE    0

extern BOOLEAN GetWorld(PTR_TO_WORLD x);
extern int NextWorld(PTR_TO_CONST_WORLD x, PTR_TO_WORLD y);
extern void PutWorld(PTR_TO_CONST_WORLD x);

extern int Rows, Cols, Gen, EndGen, Inhabs;
```

life.c

(Figure 12.25 – Page 303)

```
/*
 * Main program for the Game of Life.
 */
#include "life.h"

int Rows;                                    /* # of rows used */
int Cols;                                    /* # of columss used */
int Inhabs;                                  /* # of inhabitants */
int Gen;                                     /* current generation */
int EndGen;                                  /* last generation */

int main()
{
  static WORLD world1,                       /* one is current world */
               world2;                       /* other is future world */
  PTR_TO_WORLD currptr = world1,             /* pointers to current and */
               nextptr = world2;             /*   future worlds */
  int          status;

  if (!GetWorld(currptr))                    /* bad input */
  {
    printf("No input world or input errors -- stopping\n");
    status = 1;                              /* error exit */
  }
  else
  {
    PutWorld(currptr);                       /* write initial world */
    for (Gen = 2; Inhabs && Gen <= EndGen; Gen++)
    {
      PTR_TO_WORLD temp;
```

```
      if (!NextWorld(currptr, nextptr))
        break;                                    /* no changes! */
      PutWorld(nextptr);                          /* write next world */
      temp = currptr;                             /* exchange worlds */
      currptr = nextptr;
      nextptr = temp;
    }
    if (!Inhabs)
      printf("All inhabitants are dead\n");
    else if (Gen <= EndGen)
      printf("Reached stable state\n");
    else
      printf("Computed requested generations.\n");
    status = 0;
  }
  return status;
}
```

getworld.c

(Figure 12.26 – Page 304)

```
/*
 * Life input handler (quits on first error)
 */
#include <stdio.h>
#include "life.h"

static int inrange(int min, int max, int value);

BOOLEAN GetWorld(PTR_TO_WORLD world)
{
  int getline(char *ptr, int len);
  char buf[MAXLEN + 1];                       /* input line */
  int r, c;                                   /* next position */

  if (getline(buf, MAXLEN) == -1 ||
      sscanf(buf, "%i %i %i", &Rows, &Cols, &EndGen) != 3)
  {
    printf("Didn't provide ROWS, COLS, FINAL GENERATION\n");
    return FALSE;
  }
```

getworld.c

(Figure 12.26 – Page 304)

```
  printf("Rows=%i Cols=%i, Final Generation=%i\n",
         Rows, Cols, EndGen);

  if (!inrange(1, MAXROW, Rows))
    printf("Invalid Rows, set to %i\n", MAXROW);
  if (!inrange(1, MAXCOL, Cols))
    printf("Invalid Columns, set to %i\n", MAXCOL);

  while (getline(buf, MAXLEN) != -1)
    if (sscanf(buf, "%i %i", &r, &c) != 2)
      printf("Non-numeric cell position -- ignored\n");
    else if (!inrange(1, Rows, r) || !inrange(1, Cols, c))
      printf("%i,%i is out of range -- ignored\n", r, c);
    else if (!world[r][c])
    {
      printf("Cell %i,%i is now occupied\n", r, c);
      world[r][c] = TRUE;
      Inhabs++;
    }
  return TRUE;
}

static int inrange(int min, int max, int value)
{
  return min <= value && value <= max;
}
```

nxtworld.c

(Figure 12.27 – Page 305)

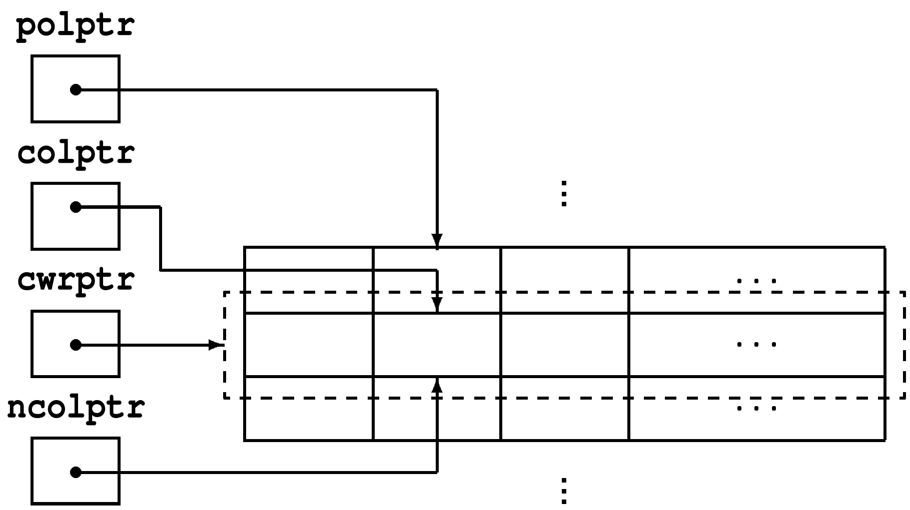
```
/*
 * Compute the next generation.
 */
#include "life.h"

int NextWorld(PTR_TO_CONST_WORLD curr, PTR_TO_WORLD next)
{
  int changed = 0;                     /* cells that changed */
  int r, c;                            /* row, column index */

  Inhabs = 0;
  for (r = 1; r <= Rows; r++)
    for (c = 1; c <= Cols; c++)
    {
      int neighbors =  curr[r - 1][c - 1] + curr[r - 1][c] +
                       curr[r - 1][c + 1] + curr[r + 1][c - 1] +
                       curr[r + 1][c]     + curr[r + 1][c + 1] +
                       curr[r][c - 1]     + curr[r][c + 1];

      if ((neighbors == 3) || (neighbors == 2 && curr[r][c]))
      {
        next[r][c] = TRUE;
        Inhabs++;
      }
      else
        next[r][c] = FALSE;

      changed += next[r][c] != curr[r][c];
    }
  return changed;
}
```

outworld.c

(Figure 12.28 – Page 306)

```
/*
 * Handle printing the generation.
 */
#include "life.h"

static void putborder(int n);

void PutWorld(PTR_TO_CONST_WORLD world)
{
  int c, r;

  printf("Generation %i out of %i\n", Gen, EndGen);
  putborder(1 + Cols + 1);
  for (r = 1; r <= Rows; r++)
  {
    putchar(BORDER);
    for (c = 1; c <= Cols; c++)
      putchar(world[r][c] ? MARKER : ' ');
    putchar(BORDER);
    putchar('\n');
  }
  putborder(1 + Cols + 1);
  putchar('\n');
}

static void putborder(int n)
{
  int i;

  for (i = 1; i <= n; i++)
    putchar(BORDER);
  putchar('\n');
}
```

THE POINTERS IN LIFE



nxtwrld2.c

(Figure 12.30 – Page 308)

```
/*
 * Compute next generation, this time using pointers.
 */
#include "life.h"

int NextWorld(PTR_TO_CONST_WORLD curr, PTR_TO_WORLD next)
{
  PTR_TO_CONST_WORLD cwrptr = curr + 1;         /* 1st row curr world */
  PTR_TO_WORLD       nwrptr = next + 1;         /* 1st row next world */
  PTR_TO_CONST_WORLD endcwrptr = cwrptr + Rows; /* last row curr world */
  int changed = 0;                              /* changes to world */
  int neighbors;                                /* # of cell neighbors */

  for (Inhabs = 0; cwrptr <= endcwrptr; cwrptr++, nwrptr++)
  {
    const BOOLEAN *colptr    = cwrptr[0]  + 1;   /* current col */
    const BOOLEAN *pcolptr   = cwrptr[-1] + 1;   /* previous row */
    const BOOLEAN *ncolptr   = cwrptr[1]  + 1;   /* next row */
    const BOOLEAN *endcolptr = colptr + Cols;    /* last col */
    BOOLEAN *nxtptr          = nwrptr[0] + 1;    /* col in next */

    for (; colptr <= endcolptr; colptr++, nxtptr++, pcolptr++, ncolptr++)
    {
      neighbors =  pcolptr[-1] + pcolptr[0] + pcolptr[1] + colptr[-1] +
                   ncolptr[-1] + ncolptr[0] + ncolptr[1] + colptr[1];
      if ((neighbors == 3) || (neighbors == 2 && *colptr))
      {
        *nxtptr = TRUE;
        Inhabs++;
      }
      else
        *nxtptr = FALSE;
      changed += *nxtptr != *colptr;
    }
  }
  return changed;                          /* nonzero if not stable */
}
```

REMINDERS

- The name of a two-dimensional array is a pointer to its first row, not its first element.

- Dereferencing a pointer to a row gives us the row itself, not its first element.

CHAPTER SUMMARY

- C lets us have the elements of an array be arrays themselves.

- When we pass two-dimensional arrays as parameters, we have to specify the number of elements in each row, but not the number of rows.

- The name of a two-dimensional array is a pointer to its first element, which happens to be its first row.

- We can have pointers to rows and use them to traverse a two-dimensional array a row at a time. This lets us efficiently access a column.

- When we traverse a sequence of elements within a two-dimensional array, we can use pointers to items as if it were a one-dimensional array. This allows us to efficiently access each item in a row.

CHAPTER SUMMARY

- We can often process rows of a two-dimensional array using existing functions designed to work with a one-dimensional array.

- All of the techniques for accessing two-dimensional arrays can be generalized and used for higher-dimensional arrays.

- It's reasonable to use normal subscripting rather than pointers to traverse the elements of multidimensional arrays. Then, if the resulting programs needs to be sped up, these traverals should be rewritten to use pointers.