1.2.1 Rules for the Game of Life

Life is really a simulation, not a game with players. It takes place on an unbounded rectangular grid in which each cell can either be occupied by an organism or not. Occupied cells are called alive; unoccupied cells are called dead. Which cells are alive changes from generation to generation according to the number of neighboring cells that are alive, as follows:

1. The neighbors of a given cell are the eight cells that touch it vertically, horizontally, or diagonally.
2. If a cell is alive but either has no neighboring cells alive or only one alive, then in the next generation the cell dies of loneliness.
3. If a cell is alive and has four or more neighboring cells also alive, then in the next generation the cell dies of overcrowding.
4. A living cell with either two or three living neighbors remains alive in the next generation.
5. If a cell is dead, then in the next generation it will become alive if it has exactly three neighboring cells, no more or fewer, that are already alive. All other dead cells remain dead in the next generation.
6. All births and deaths take place at exactly the same time, so that dying cells can help to give birth to another, but cannot prevent the death of others by reducing overcrowding, nor can cells being born either preserve or kill cells living in the previous generation.

1.2.2 Examples

As a first example, consider the community

The counts of living neighbors for the cells are as follows:
large by that we sometimes general chapters

By rule 2 both the living cells will die in the coming generation, and rule 5 shows that no cells will become alive, so the community dies out.

On the other hand, the community has the neighbor counts as shown. Each of the living cells has a neighbor count of three, and hence remains alive, but the dead cells all have neighbor counts of two or less, and hence none of them becomes alive.

The two communities continue to alternate from generation to generation, as indicated by the neighbor counts shown.

It is a surprising fact that, from very simple initial configurations, quite complicated progressions of Life communities can develop, lasting many generations, and it is usually
variety  not obvious what changes will happen as generations progress. Some very small initial
configurations will grow into large communities; others will slowly die out; many will
reach a state where they do not change, or where they go through a repeating pattern
every few generations.

popularity  Not long after its invention, MARTIN GARDNER discussed the Life game in his column
in Scientific American, and, from that time on, it has fascinated many people, so that for
several years there was even a quarterly newsletter devoted to related topics. It makes
an ideal display for microcomputers.

Our first goal, of course, is to write a program that will show how an initial com-
munity will change from generation to generation.

1.2.3 The Solution

At most a few minutes’ thought will show that the solution to the Life problem is so
simple that it would be a good exercise for the members of a beginning programming
class who had just learned about arrays. All we need to do is to set up a large rectangular
array whose entries correspond to the Life cells and will be marked with the status of the
cell, either alive or dead. To determine what happens from one generation to the next,
we then need only count the number of living neighbors of each cell and apply the rules.
Since, however, we shall be using loops to go through the array, we must be careful not
to violate rule 6 by allowing changes made earlier to affect the count of neighbors for
cells studied later. The easiest way to avoid this pitfall is to set up a second array that
will represent the community at the next generation and, after it has been completely
calculated, then make the generation change by copying it to the original array.

Next let us rewrite this method as the steps of an informal algorithm.

algorithm  Initialize an array called map to contain the initial configuration of living cells.
Repeat the following steps for as long as desired:
For each cell in the array do the following:
  Count the number of living neighbors of the cell.
  If the count is 0, 1, 4, 5, 6, 7, or 8, then set the corresponding cell in
  another array called newmap to be dead; if the count is 3, then set
  the corresponding cell to be alive; and if the count is 2, then set the
  corresponding cell to be the same as the cell in array map (since the
  status of a cell with count 2 does not change).
Copy the array newmap into the array map.
Print the array map for the user.

1.2.4 Life: The Main Program

The preceding outline of an algorithm for the game of Life translates into the following
C program.
small initial PTER 1 n his column SECTION 1.2 tioning pattern, so that for cs. It makes n initial com-

The Game of life 7

it; many will ating pattern

* Simulation of Conway's game of Life on a bounded grid */
* Version 1 */
#include "general.h" /* common include files and definitions */
#include "lifedef.h" /* Life's defines and typedefs */
#include "calls.h" /* Life's function declarations */

void main (void)
{
int row, col;
Grid.type map;
Grid.type newmap;
/* current generation */
/* next generation */

initialization
Initialize (map);
WriteMap (map);
do {

calculate changes
for (row = 0; row < MAXROW; row++)
    for (col = 0; col < MAXCOL; col++)
        switch (NeighborCount (row, col, map)) {
            case 0:
                newmap [row] [col] = DEAD;
                break;
            case 1:
                newmap [row] [col] = map [row] [col];
                break;
            case 2:
                newmap [row] [col] = ALIVE;
                break;
            case 3:
                newmap [row] [col] = DEAD;
                break;
            case 4:
                case 5:
                case 6:
                case 7:
                case 8:
                newmap [row] [col] = DEAD;
                break;
        }
    CopyMap (map, newmap);
    WriteMap (map);
} while (Enquire ());

Before we discuss the C program above we need to establish what is included with the #include preprocessor command. There are three files: general.h, lifedef.h, and calls.h.

The file general.h contains the definitions and #include statements for the standard files that appear in many programs and will be used throughout this book. The file includes

external dependencies. The file lifedef.h contains the definitions of the data types used in the game, such as Grid and Map types. Finally, the file calls.h contains the function declarations for the functions that are called within the program.
#include <stdio.h>
#include <stdlib.h>
typedef enum boolean_tag { FALSE, TRUE } Boolean_type;
void Error(char *s);

The function Error is a simple function we use throughout the book. Error displays an error message and terminates execution. Here is the function.

/*@ Error: print error message and terminate the program. */
void Error(char *s)
{
    fprintf(stderr, "%s\n", s);
    exit(1);
}

The file lifedef.h contains the definitions for the Life program:

#define MAXROW 50 /* maximum number of rows */
#define MAXCOL 80 /* maximum number of columns */
typedef enum status_tag { DEAD, ALIVE } Status_type;
typedef Status_type Grid_type[MAXROW][MAXCOL];

functions

and calls.h contains the function prototypes for the Life program:

void CopyMap(Grid_type, Grid_type);
Boolean_type Enquire(void);
void Initialize(Grid_type);
int NeighborCount(int, int, Grid_type);
void WriteMap(Grid_type);

We create a new calls.h file with the function prototypes for each program we write. In this program we still must write the functions Initialize and WriteMap that will do the input and output, the function Enquire that will determine whether or not to go on to the next generation, the function CopyMap that will copy the updated grid, newmap, into map, and the function NeighborCount that will count the number of cells neighboring the one in row,col that are occupied in the array map. The program is entirely straightforward. First, we read in the initial situation to establish the first configuration of occupied cells. Then we commence a loop that makes one pass for each generation. Within this loop we first have a nested pair of loops on row and col that will run over all entries in the array map. The body of these nested loops consists of the one special statement

switch {...},

which is a multiway selection statement. In the present application the function NeighborCount will return one of the values 0, 1, ..., 8, and for each of these cases we can take a separate action, or, as in our program, some of the cases may lead to the same action. You should check that the action prescribed in each case corresponds correctly to the rules 2, 3, 4, and 5 of Section 1.2.1. Finally, after using the nested loops and switch
1.3 Programming Style

Before we turn to writing the functions for the Life game, let us pause to consider several principles that we should be careful to employ in programming.

In the story of creation (Genesis 2:19), God brought all the animals to Adam to see what names he would give them. According to an old Jewish tradition, it was only when Adam had named an animal that it sprang to life. This story brings an important moral to computer programming: Even if data and algorithms exist before, it is only when they are given meaningful names that their places in the program can be properly recognized and appreciated, that they first acquire a life of their own.

For a program to work properly it is of the utmost importance to know exactly what each variable represents, and to know exactly what each function does. Documentation
E5. The mean of a sequence of real numbers is their sum divided by the count of numbers in the sequence. The (population) variance of the sequence is the mean of the squares of all numbers in the sequence, minus the square of the mean of the numbers in the sequence. The standard deviation is the square root of the variance. Write a well-structured C function to calculate the standard deviation of a sequence of \( n \) numbers, where \( n \) is a constant and the numbers are in an array indexed from 0 to \( n - 1 \), where \( n \) is a parameter to the function. Write, then use, subsidiary functions to calculate the mean and variance.

E6. Design a program that will plot a given set of points on a graph. The input to the program will be a text file, each line of which contains two numbers that are the \( x \) and \( y \) coordinates of a point to be plotted. The program will use a routine to plot one such pair of coordinates. The details of the routine involve the specific method of plotting and cannot be written since they depend on the requirements of the plotting equipment, which we do not know. Before plotting the points the program needs to know the maximum and minimum values of \( x \) and \( y \) that appear in its input file. The program should therefore use another routine Bounds that will read the whole file and determine these four maxima and minima. Afterward, another routine is used to draw and label the axes; then the file can be reset and the individual points plotted.

a. Write the main program, not including the routines.
b. Write the function Bounds.
c. Write the header lines for the remaining functions together with appropriate documentation showing their purposes and their requirements.

1.4 CODING, TESTING, AND FURTHER REFINEMENT

The three processes in the title above go hand-in-hand and must be done together. Yet it is important to keep them separate in our thinking, since each requires its own approach and method. Coding, of course, is the process of writing an algorithm in the correct syntax (grammar) of a computer language like C, and testing is the process of running the program on sample data chosen to find errors if they are present. For further refinement, we turn to the functions not yet written and repeat these steps.

1.4.1 Stubs

After coding the main program, most programmers will wish to complete the writing and coding of the functions as soon as possible, to see if the whole project will work. For a project as small as the Life game, this approach may work, but for larger projects, writing and coding all the functions will be such a large job that, by the time it is complete, many of the details of the main program and functions that were written early will have been forgotten. In fact, different people may be writing different functions, and so
of those who started the project may have left it before all functions are written. It is much easier to understand and debug a program when it is fresh in your mind. Hence, for larger projects, it is much more efficient to debug and test each function as soon as it is written than it is to wait until the project has been completely coded.

Even for smaller projects, there are good reasons for debugging functions one at a time. We might, for example, be unsure of some point of C syntax that will appear in several places through the program. If we can compile each function separately, then we shall quickly learn to avoid errors in syntax in later functions. As a second example, suppose that we have decided that the major steps of the program should be done in a certain order. If we test the main program as soon as it is written, then we may find that sometimes the major steps are done in the wrong order, and we can quickly correct the problem, doing so more easily than if we waited until the major steps were perhaps obscured by the many details contained in each of them.

To compile the program correctly, there must be something in the place of each function that is used, and hence we must put in short, dummy functions, called stubs. The simplest stubs are those that do nothing at all:

```c
/* Initialize: initialize grid map. */
void Initialize (Grid_type map) 
{} 

/* WriteMap: write grid map. */
void WriteMap (Grid_type map) 
{} 

/* NeighborCount: count neighbors of row, col. */
int NeighborCount (int row, int col, Grid_type map) 
{ 
    return 1;
}
```

Even with these stubs we can at least compile the program and make sure that the declarations of types and variables are syntactically correct. Normally, however, each stub should print a message stating that the function was invoked. When we execute the program, we find that some variables are used without initialization, and hence, to avoid these errors, we can add code to function Initialize. Hence the stub can slowly grow and be refined into the final form of the function. For a small project like the Life game, we can simply write each function in turn, substitute it for its stub, and observe the effect on program execution.

### 1.4.2 Counting Neighbors

Let us now refine our program further. The function that counts neighbors of the cell in row, col requires that we look in the eight adjoining positions. We shall use a pair of for loops to do this, one running usually from row - 1 to row + 1 and the other usually from col - 1 to col + 1. We need to be careful, when row, col is on a boundary of the grid, that we look only at legitimate positions in the grid. To do so we introduce four
After\n
parameters, which smallest integer
into ascending
variable \( n - 1 \)

5;

that it works

1.3.

it.

ogram.

art separately.

be more than

dbugging.

Always use
d debugging
g a program

and it better

Figure 1.3. Life configurations