.1 INTRODUCTION

The greatest difficulties of writing large computer programs are not in deciding what the goals of the program should be, nor even in finding methods that can be used to reach these goals. The president of a business might say, "Let's get a computer to keep track of all our inventory information, accounting records, and personnel files, and let it tell us when inventories need to be reordered and budget lines are overspent, and let it handle the payroll." With enough time and effort, a staff of systems analysts and programmers might be able to determine how various staff members are now doing these tasks and write programs to do the work in the same way.

problems of large programs

This approach, however, is almost certain to be a disastrous failure. While interviewing employees, the systems analysts will find some tasks that can be put on the computer easily and will proceed to do so. Then, as they move other work to the computer, they will find that it depends on the first tasks. The output from these, unfortunately, will not be quite in the proper form. Hence they need more programming to convert the data from the form given for one task to the form needed for another. The programming project begins to resemble a patchwork quilt. Some of the pieces are stronger, some weaker. Some of the pieces are carefully sewn onto the adjacent ones, some are barely tacked together. If the programmers are lucky, their creation may hold together well enough to do most of the routine work most of the time. But if any change must be made, it will have unpredictable consequences throughout the system. Later, a new request will come along, or an unexpected problem, perhaps even an emergency, and the programmers' efforts will prove as effective as using a patchwork quilt as a safety net for people jumping from a tall building.

purpose of book

The main purpose of this book is to describe programming methods and tools that will prove effective for projects of realistic size, programs much larger than those ordinarily used to illustrate features of elementary programming. Since a piecemeal approach to large problems is doomed to fail, we must first of all adopt a consistent, unified, and logical approach, and we must also be careful to observe important principles of program design, principles that are sometimes ignored in writing small programs, but whose neglect will prove disastrous for large projects.

problem specification

The first major hurdle in attacking a large problem is deciding exactly what the problem is. It is necessary to translate vague goals, contradictory requests, and perhaps unstated desires into a precisely formulated project that can be programmed. And the methods or divisions of work that people have previously used are not necessarily the best for use in a machine. Hence our approach must be to determine overall goals, but precise ones, and then slowly divide the work into smaller problems until they become of manageable size.

program design

The maxim that many programmers observe, "First make your program work, then make it pretty," may be effective for small programs, but not for large ones. Each part of a large program must be well organized, clearly written, and thoroughly understood, or else its structure will have been forgotten, and it can no longer be tied to the other parts of the project at some much later time, perhaps by another programmer. Hence we do not separate style from other parts of program design, but from the beginning we must be careful to form good habits.

Even with very large projects, difficulties usually arise not from the inability to find a solution but, rather, from the fact that there can be so many different methods

and algorithms that might work that it can be hard to decide which is best, which may lead to programming difficulties, or which may be hopelessly inefficient. The greatest room for variability in algorithm design is generally in the way in which the data of the program are stored:

data structures

- How they are arranged in relation to each other.
- Which data are kept in memory.
- Which are calculated when needed.
- Which are kept in files, and how are the files arranged.

A second goal of this book, therefore, is to present several elegant, yet fundamentally simple ideas for the organization of data, and several powerful algorithms for important tasks within data processing, such as sorting and searching.

When there are several different ways to organize data and devise algorithms it becomes important to develop criteria to recommend a choice. Hence we devote attention to analyzing the behavior of algorithms under various conditions.

analysis

The difficulty of debugging a program increases much faster than its size. That is, if one program is twice the size of another, then it will likely not take twice as long to debug, but perhaps four times as long. Many very large programs (such as operating systems) are put into use still containing bugs that the programmers have despaired of finding, because the difficulties seem insurmountable. Sometimes projects that have consumed years of effort must be discarded because it is impossible to discover why they will not work. If we do not wish such a fate for our own projects, then we must use methods that will

testing and verification

program correctness

- Reduce the number of bugs, making it easier to spot those that remain.
- Enable us to verify in advance that our algorithms are correct.
- Provide us with ways to test our programs so that we can be reasonably confident that they will not misbehave.

Development of such methods is another of our goals, but one that cannot yet be fully within our grasp.

Informal surveys show that, once a large and important program is fully debugged and in use, then less than half of the programming effort that will be invested altogether in the project will have been completed. ***Maintenance*** of programs, that is, modifications needed to meet new requests and new operating environments, takes, on average, more than half of the programming investment. For this reason, it is essential that a large project be written to make it as easy to understand and modify as possible.

maintenance

1.2 THE GAME OF LIFE

If we may take the liberty to abuse an old proverb:

> *One concrete problem is worth a thousand unapplied abstractions.*

case study Throughout this chapter we shall concentrate on one case study that, while not large by realistic standards, illustrates both the methods of program design and the pitfalls that we should learn to avoid. Sometimes the example motivates general principles; sometimes the general discussion comes first; always it is with the view of discovering general methods that will prove their value in a range of practical applications. In later chapters we shall employ similar methods for much larger projects.

The example we shall use is the game called ***Life***, which was introduced by the British mathematician J. H. CONWAY in 1970.

1.2.1 Rules for the Game of Life

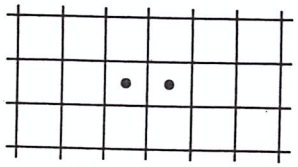
definitions Life is really a simulation, not a game with players. It takes place on an unbounded rectangular grid in which each cell can either be occupied by an organism or not. Occupied cells are called ***alive***; unoccupied cells are called ***dead***. Which cells are alive changes from generation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

transition rules

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.

.2.2 Examples

As a first example, consider the community



The counts of living neighbors for the cells are as follows:

moribund example

0	0	0	0	0	0
0	1	2	2	1	0
0	1	1	1	1	0
0	1	2	2	1	0
0	0	0	0	0	0

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

stability

0	0	0	0	0	0
0	1	2	2	1	0
0	2	3	3	2	0
0	2	3	3	2	0
0	1	2	2	1	0
0	0	0	0	0	0

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

alternation

0	0	0	0	0
1	2	3	2	1
1	1	2	1	1
1	2	3	2	1
0	0	0	0	0

and

0	1	1	1	0
0	2	1	2	0
0	3	2	3	0
0	2	1	2	0
0	1	1	1	0

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 community 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 *method* 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.


```
/* 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;                      /* current generation                     */
    Grid_type newmap;                   /* next generation                        */

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

```
#include <stdio.h>
#include <stdlib.h>
typedef enum boolean_tag { FALSE, TRUE } Boolean_type;
void Error(char *);
```

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

statement to set up the array newmap, the function CopyMap copies array newmap into array map, and the function WriteMap writes out the result.

Exercises 1.2 Determine by hand calculation what will happen to each of the communities shown in Figure 1.1 over the course of five generations. [*Suggestion*: Set up the Life configuration on a checkerboard. Use one color of checkers for living cells in the current generation and a second color to mark those that will be born or die in the next generation.]

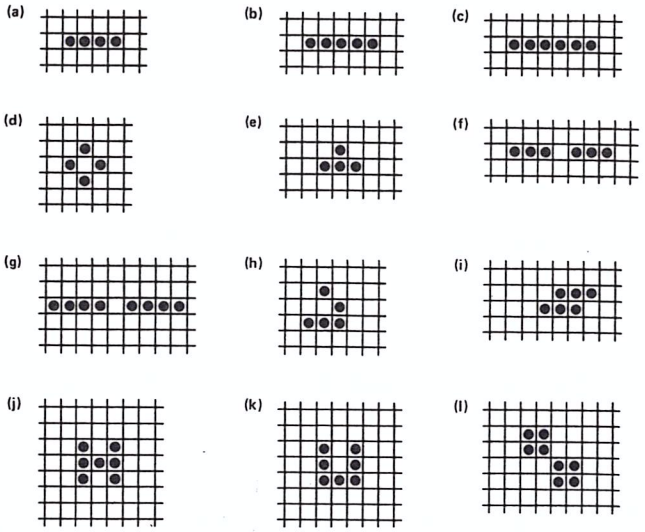


Figure 1.1. Life configurations

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.

1.3.1 Names

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.

purpose of careful naming

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