Programming Principles

1

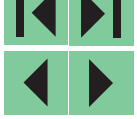
THIS CHAPTER summarizes important principles of good programming, especially as applied to large projects, and introduces methods such as object-oriented design and top-down design for discovering effective algorithms. In the process we raise questions in program design and data-storage methods that we shall address in later chapters, and we also review some of the elementary features of the language C++ by using them to write programs.

1.1 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.



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:

choice of 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 the files are arranged.

A second goal of this book, therefore, is to present several elegant, yet fundamentally simple ideas for the organization and manipulation of data. Lists, stacks, and queues are the first three such organizations that we study. Later, we shall develop 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 of algorithms

Contents
Index
Help

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 errors 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 errors, 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.

Even after a program is completed, fully debugged, and put into service, a great deal of work may be required to maintain the usefulness of the program. In time there will be new demands on the program, its operating environment will change, new requests must be accommodated. For this reason, it is essential that a large project be written to make it as easy to understand and modify as possible.

maintenance

C++

The programming language C++ is a particularly convenient choice to express the algorithms we shall encounter. The language was developed in the early 1980s, by Bjarne Stroustrup, as an extension of the popular C language. Most of the new features that Stroustrup incorporated into C++ facilitate the understanding and implementation of data structures. Among the most important features of C++ for our study of data structures are:

Highlights

- C++ allows ***data abstraction***: This means that programmers can create new types to represent whatever collections of data are convenient for their applications.
- C++ supports ***object-oriented design***, in which the programmer-defined types play a central role in the implementation of algorithms.
- Importantly, as well as allowing for object-oriented approaches, C++ allows for the use of the ***top-down approach***, which is familiar to C programmers.
- C++ facilitates ***code reuse***, and the construction of general purpose libraries. The language includes an extensive, efficient, and convenient standard library.
- C++ improves on several of the inconvenient and dangerous aspects of C.
- C++ maintains the efficiency that is the hallmark of the C language.

It is the combination of flexibility, generality and efficiency that has made C++ one of the most popular choices for programmers at the present time.

We shall discover that the general principles that underlie the design of all data structures are naturally implemented by the data abstraction and the object-oriented features of C++. Therefore, we shall carefully explain how these aspects of C++ are used and briefly summarize their syntax (grammar) wherever they first arise in our book. In this way, we shall illustrate and describe many of the features of C++ that do not belong to its small overlap with C. For the precise details of C++ syntax, consult a textbook on C++ programming—we recommend several such books in the references at the end of this chapter.

Contents
Index
Help

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 principles 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 principles that will prove their value in a range of practical applications. In later chapters we shall employ similar methods for 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.

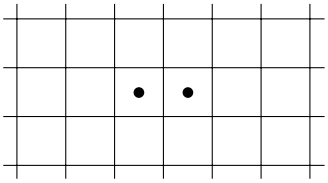
configuration

A particular arrangement of living and dead cells in a grid is called a ***configuration***. The preceding rules explain how one configuration changes to another at each generation.

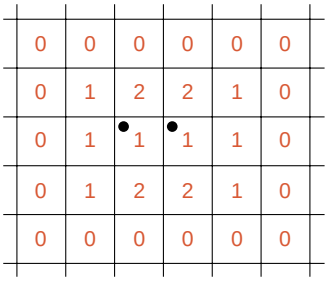
Contents
Index
Help

1.2.2 Examples

As a first example, consider the configuration



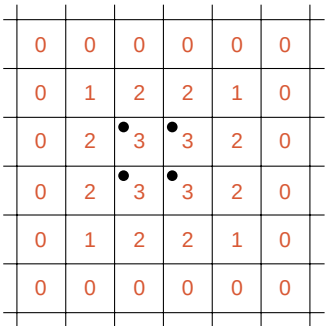
The counts of living neighbors for the cells are as follows:



moribund example

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 configuration dies out.

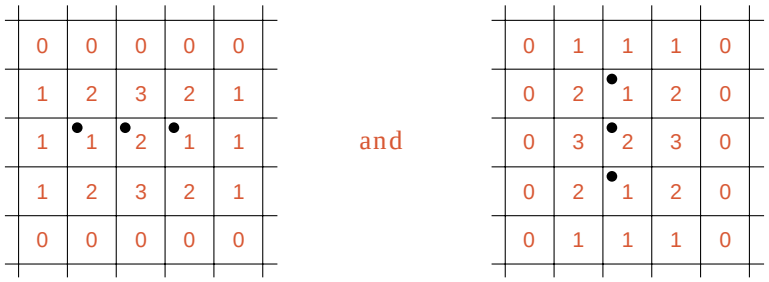
On the other hand, the configuration



stability

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 configurations



alternation

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 configurations can develop, lasting many generations, and it is usually not obvious what changes will happen as generations progress.

variety

Some very small initial configurations will grow into large configurations; 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 home microcomputers.

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

Contents
Index
Help

1.2.3 The Solution: Classes, Objects, and Methods

In outline, a program to run the Life game takes the form:

algorithm

Set up a Life configuration as an initial arrangement of living and dead cells.

Print the Life configuration.

While the user wants to see further generations:

    Update the configuration by applying the rules of the Life game.
    Print the current configuration.

class

object

The important thing for us to study in this algorithm is the Life configuration. In C++, we use a ***class*** to collect data and the methods used to access or change the data. Such a collection of data and methods is called an ***object*** belonging to the given class. For the Life game, we shall call the class Life, so that configuration becomes a Life *object*. We shall then use three methods for this object: initialize() will set up the initial configuration of living and dead cells; print() will print out the current configuration; and update() will make all the changes that occur in moving from one generation to the next.

C++ classes

Every C++ class, in fact, consists of ***members*** that represent either variables or functions. The members that represent variables are called the ***data members***; these are used to store data values. The members that represent functions belonging to a class are called the ***methods*** or ***member functions***. The methods of a class are normally used to access or alter the data members.

methods

clients

Clients, that is, user programs with access to a particular class, can declare and manipulate objects of that class. Thus, in the Life game, we shall declare a Life object by:

```
Life configuration;
```

member selection operator

We can now apply methods to work with configuration, using the C++ operator . (the member selection operator). For example, we can print out the data in configuration by writing:

```
configuration.print();
```

specifications

It is important to realize that, while writing a client program, we can use a C++ class so long as we know the ***specifications*** of each of its methods, that is, statements of precisely what each method does. We do not need to know how the data are actually stored or how the methods are actually programmed. For example, to use a Life object, we do not need to know exactly how the object is stored, or how the methods of the class Life are doing their work. This is our first example of an important programming strategy known as ***information hiding***.

information hiding

private and public

When the time comes to implement the class Life, we shall find that more goes on behind the scenes: We shall need to decide how to store the data, and we shall need variables and functions to manipulate this data. All these variables and functions, however, are *private* to the class; the client program does not need to know what they are, how they are programmed, or have any access to them. Instead, the client program only needs the *public* methods that are specified and declared for the class.

Contents
Index
Help

In this book, we shall always distinguish between methods and functions as follows, even though their actual syntax (programming grammar) is the same:

Convention

Methods of a class are public.
Functions in a class are private.

1.2.4 Life: The Main Program

The preceding outline of an algorithm for the game of Life translates into the following C++ program.

```
#include "utility.h"
#include "life.h"

int main()                    // Program to play Conway's game of Life.
/* Pre:  The user supplies an initial configuration of living cells.
   Post: The program prints a sequence of pictures showing the changes in the
         configuration of living cells according to the rules for the game of Life.
   Uses: The class Life and its methods initialize(), print(), and update().
         The functions  instructions(), user_says_yes(). */
{
  Life configuration;
  instructions();
  configuration.initialize();
  configuration.print();
  cout << "Continue viewing new generations? " << endl;
  while (user_says_yes()) {
    configuration.update();
    configuration.print();
    cout << "Continue viewing new generations? " << endl;
  }
}
```

utility package

The program begins by including files that allow it to use the class Life and the standard C++ input and output libraries. The utility function user_says_yes() is declared in `utility.h`, which we shall discuss presently. For our Life program, the only other information that we need about the file `utility.h` is that it begins with the instructions

```
#include <iostream>
using namespace std;
```

which allow us to use standard C++ input and output streams such as cin and cout. (On older compilers an alternative directive #include <iostream.h> has the same effect.)

program specifications

The documentation for our Life program begins with its ***specifications***; that is, precise statements of the conditions required to hold when the program begins and the conditions that will hold after it finishes. These are called, respectively, the ***preconditions*** and ***postconditions*** for the program. Including precise preconditions and postconditions for each function not only explains clearly the purpose of the function but helps us avoid errors in the interface between functions. Including specifications is so helpful that we single it out as our first programming precept:

Programming Precept

Include precise preconditions and postconditions with every program, function, and method that you write.

functions

A third part of the specifications for our program is a list of the classes and functions that it uses. A similar list should be included with every program, function, or method.

action of the program

The action of our main 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 simply update the Life configuration, print the configuration, and ask the user whether we should continue. Note that the Life methods, initialize, update, and print are simply called with the member selection operator.

In the Life program we still must write code to implement:

- The **class** Life.
- The method initialize() to initialize a Life configuration.
- The method print() to output a Life configuration.
- The method update() to change a Life object so that it stores the configuration at the next generation.
- The function user_says_yes() to ask the user whether or not to go on to the next generation.
- The function instructions() to print instructions for using the program.

The implementation of the class Life is contained in the two files `life.h` and `life.c`. There are a number of good reasons for us to use a pair of files for the implementation of any class or data structure: According to the principle of information hiding, we should separate the definition of a class from the coding of its methods. The user of the class only needs to look at the specification part and its list of methods. In our example, the file `life.h` will give the specification of the **class** Life.

Moreover, by dividing a class implementation between two files, we can adhere to the standard practice of leaving function and variable definitions out of files with a suffix `.h`. This practice allows us to compile the files, or compilation units, that make up a program separately and then link them together.

Contents
Index
Help

Each compilation unit ought to be able to include any particular `.h` file (for example to use the associated data structure), but unless we omit function and variable definitions from the `.h` file, this will not be legal. In our project, the second file `life.c` must therefore contain the implementations of the methods of the **class** Life and the function instructions().[1]

Another code file, `utility.c`, contains the definition of the function

user_says_yes().

utility package

We shall, in fact, soon develop several more functions, declarations, definitions, and other instructions that will be useful in various applications. We shall put all of these together as a ***package***. This package can be incorporated into any program with the directive:

#include "utility.h"

whenever it is needed.

Just as we divided the Life class implementation between two files, we should divide the utility package between the files `utility.h` and `utility.c` to allow for its use in the various translation units of a large program. In particular, we should place function and variable definitions into the file `utility.c`, and we place other sorts of utility instructions, such as the inclusion of standard C++ library files, into `utility.h`. As we develop programs in future chapters, we shall add to the utility package. Appendix C lists all the code for the whole utility package.

Contents
Index
Help

Exercises 1.2

IRM

Determine by hand calculation what will happen to each of the configurations 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.]

1.3 PROGRAMMING STYLE

Before we turn to implementing classes and 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), the LORD 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

[1] On some compilers the file suffix `.c` has to be replaced by an alternative such as `.C`, `.cpp`, `.cxx`, or `.cc`.

8

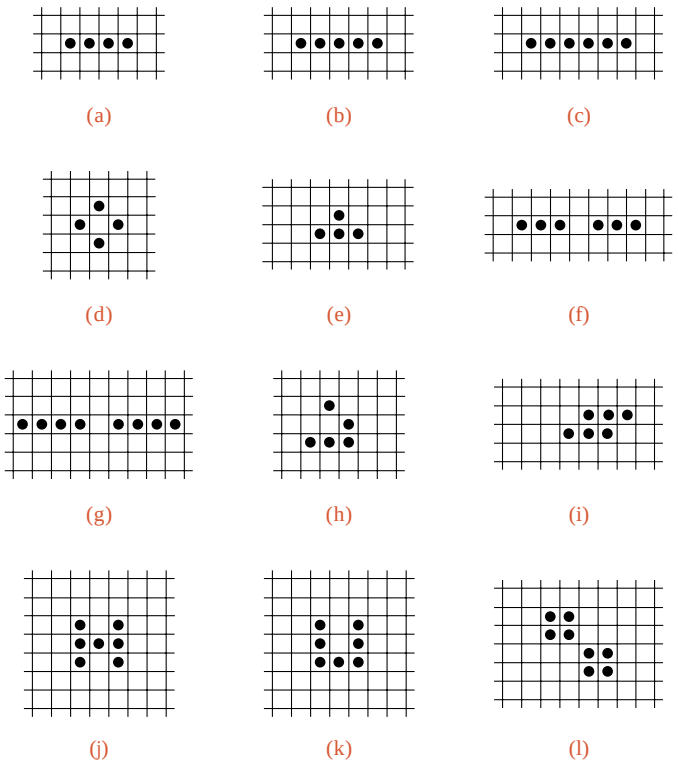


Figure 1.1. Simple Life configurations

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 class and variable represents and to know exactly what each function does. Documentation explaining the classes, variables, and functions should therefore always be included. The names of classes, variables, and functions should be chosen with care so as to identify their meanings clearly and succinctly. Finding good names is not always an easy task, but is important enough to be singled out as our second programming precept:

9

Programming Precept

Always name your classes, variables and functions with the greatest care, and explain them thoroughly.

C++ goes some distance toward enforcing this precept by requiring the declaration of variables and allows us almost unlimited freedom in the choice of identifying

names. Constants used in different places should be given names, and so should different data types, so that the compiler can catch errors that might otherwise be difficult to spot.

We shall see that types and classes play a fundamental role in C++ programs, and it is particularly important that they should stand out to a reader of our programs. We shall therefore adopt a capitalization convention, which we have already used in the Life program: We use an initial capital letter in the identifier of any class or programmer defined type. In contrast, we shall use only lowercase letters for the identifiers of functions, variables, and constants.

The careful choice of names can go a long way in clarifying a program and in helping to avoid misprints and common errors. Some guidelines are

guidelines

1. Give special care to the choice of names for classes, functions, constants, and all global variables used in different parts of the program. These names should be meaningful and should suggest clearly the purpose of the class, function, variable, and the like.
2. Keep the names simple for variables used only briefly and locally. Mathematicians usually use a single letter to stand for a variable, and sometimes, when writing mathematical programs, it may be permissible to use a single-letter name for a mathematical variable. However, even for the variable controlling a **for** loop, it is often possible to find a short but meaningful word that better describes the use of the variable.
3. Use common prefixes or suffixes to associate names of the same general category. The files used in a program, for example, might be called

   ```
   input_file   transaction_file   total_file   out_file   reject_file
   ```

4. Avoid deliberate misspellings and meaningless suffixes to obtain different names. Of all the names

   ```
   index     indx     ndex     indexx     index2     index3
   ```

   only one (the first) should normally be used. When you are tempted to introduce multiple names of this sort, take it as a sign that you should think harder and devise names that better describe the intended use.
5. Avoid choosing cute names whose meaning has little or nothing to do with the problem. The statements

   ```
   do {
      study();
   } while (TV.in_hock());
   if (!sleepy) play();
   else nap();
   ```

   may be funny but they are bad programming!

Contents
Index
Help

6. Avoid choosing names that are close to each other in spelling or otherwise easy to confuse.
7. Be careful in the use of the letter "l" (small ell), "O" (capital oh), and "0" (zero). Within words or numbers these usually can be recognized from the context and cause no problem, but "l" and "O" should never be used alone as names. Consider the examples

```
l = 1;    x = 1;    x = l;    x = O;    O = 0
```

1.3.2 Documentation and Format

the purpose of documentation

Most students initially regard documentation as a chore that must be endured after a program is finished, to ensure that the marker and instructor can read it, so that no credit will be lost for obscurity. The author of a small program indeed can keep all the details in mind, and so needs documentation only to explain the program to someone else. With large programs (and with small ones after some months have elapsed), it becomes impossible to remember how every detail relates to every other, and therefore to write large programs, it is essential that appropriate documentation be prepared along with each small part of the program. A good habit is to prepare documentation as the program is being written, and an even better one, as we shall see later, is to prepare part of the documentation before starting to write the program.

Contents
Index
Help

Not all documentation is appropriate. Almost as common as programs with little documentation or only cryptic comments are programs with verbose documentation that adds little to understanding the program. Hence our third programming precept:

Programming Precept

Keep your documentation concise but descriptive.

The style of documentation, as with all writing styles, is highly personal, and many different styles can prove effective. There are, nonetheless, some commonly accepted guidelines that should be respected:

guidelines

1. Place a prologue at the beginning of each function including
   (a) Identification (programmer's name, date, version number).[2]
   (b) Statement of the purpose of the function and algorithm used.
   (c) The changes the function makes and what data it uses.
   (d) Reference to further documentation external to the program.
2. When each variable, constant, or class is declared, explain what it is and how it is used. Better still, make this information evident from the name.

[2] To save space, programs printed in this book do not include identification lines or some other parts of the prologue, since the surrounding text gives the necessary information.

3. Introduce each significant section (paragraph or function) of the program with a comment stating briefly its purpose or action.
4. Indicate the end of each significant section if it is not otherwise obvious.
5. Avoid comments that parrot what the code does, such as

```
count++;                    //  Increase counter by 1.
```

   or that are meaningless jargon, such as

```
//  Horse string length into correctitude.
```

   (This example was taken directly from a systems program.)
6. Explain any statement that employs a trick or whose meaning is unclear. Better still, avoid such statements.
7. The code itself should explain *how* the program works. The documentation should explain *why* it works and *what* it does.
8. Whenever a program is modified, be sure that the documentation is correspondingly modified.

Contents
Index
Help

format

Spaces, blank lines, and indentation in a program are an important form of documentation. They make the program easy to read, allow you to tell at a glance which parts of the program relate to each other, where the major breaks occur, and precisely which statements are contained in each loop or each alternative of a conditional statement. There are many systems (some automated) for indentation and spacing, all with the goal of making it easier to determine the structure of the program.

prettyprinting

A ***prettyprinter*** is a system utility that reads a C++ program, moving the text between lines and adjusting the indentation so as to improve the appearance of the program and make its structure more obvious. If a prettyprinter is available on your system, you might experiment with it to see if it helps the appearance of your programs.

consistency

Because of the importance of good format for programs, you should settle on some reasonable rules for spacing and indentation and use your rules consistently in all the programs you write. Consistency is essential if the system is to be useful in reading programs. Many professional programming groups decide on a uniform system and insist that all the programs they write conform. Some classes or student programming teams do likewise. In this way, it becomes much easier for one programmer to read and understand the work of another.

Programming Precept

The reading time for programs is much more than the writing time. Make reading easy to do.

1.3.3 Refinement and Modularity

problem solving

Computers do not solve problems; people do. Usually the most important part of the process is dividing the problem into smaller problems that can be understood in more detail. If these are still too difficult, then they are subdivided again, and so on. In any large organization the top management cannot worry about every detail of every activity; the top managers must concentrate on general goals and problems and delegate specific responsibilities to their subordinates. Again, middle-level managers cannot do everything: They must subdivide the work and send it to other people. So it is with computer programming. Even when a project is small enough that one person can take it from start to finish, it is most important to divide the work, starting with an overall understanding of the problem, dividing it into subproblems, and attacking each of these in turn without worrying about the others.

subdivision

Let us restate this principle with a classic proverb:

Programming Precept

Don't lose sight of the forest for its trees.

Contents
Index
Help

top-down refinement

This principle, called ***top-down refinement***, is the real key to writing large programs that work. The principle implies the postponement of detailed consideration, but not the postponement of precision and rigor. It does not mean that the main program becomes some vague entity whose task can hardly be described. On the contrary, the main program will send almost all the work out to various classes, data structures and functions, and as we write the main program (which we should do first), we decide *exactly* how the work will be divided among them. Then, as we later work on a particular class or function, we shall know before starting exactly what it is expected to do.

specifications

It is often difficult to decide exactly how to divide the work into classes and functions, and sometimes a decision once made must later be modified. Even so, some guidelines can help in deciding how to divide the work:

Programming Precept

Use classes to model the fundamental concepts of the program.

For example, our Life program must certainly deal with the Life game and we therefore create a class `Life` to model the game. We can often pick out the important classes for an application by describing our task in words and assigning classes for the different nouns that are used. The verbs that we use will often signify the important functions.

Programming Precept

Each function should do only one task, but do it well.

Plain & Simple

That is, we should be able to describe the purpose of a function succinctly. If you find yourself writing a long paragraph to specify the preconditions or postconditions for a function, then either you are giving too much detail (that is, you are writing the function before it is time to do so) or you should rethink the division of work. The function itself will undoubtedly contain many details, but they should not appear until the next stage of refinement.

Programming Precept

Each class or function should hide something.

Middle-level managers in a large company do not pass on everything they receive from their departments to their superior; they summarize, collate, and weed out the information, handle many requests themselves, and send on only what is needed at the upper levels. Similarly, managers do not transmit everything they learn from higher management to their subordinates. They transmit to their employees only what they need to do their jobs. The classes and functions we write should do likewise. In other words, we should practice *information hiding*.

Contents
Index
Help

One of the most important parts of the refinement process is deciding exactly what the task of each function is, specifying precisely what its preconditions and postconditions will be; that is, what its input will be and what result it will produce. Errors in these specifications are among the most frequent program bugs and are among the hardest to find. First, the parameters used in the function must be precisely specified. These data are of three basic kinds:

parameters

- ***Input parameters*** are used by the function but are not changed by the function. In C++, input parameters are often passed by value. (Exception: Large objects should be passed by reference.[3] This avoids the time and space needed to make a local copy. However, when we pass an input parameter by reference, we shall prefix its declaration with the keyword **const**. This use of the type modifier **const** is important, because it allows a reader to see that we are using an input parameter, it allows the compiler to detect accidental changes to the parameter, and occasionally it allows the compiler to optimize our code.)

- ***Output parameters*** contain the results of the calculations from the function. In this book, we shall use reference variables for output parameters. In contrast, C programmers need to simulate reference variables by passing addresses of variables to utilize output parameters. Of course, the C approach is still available to us in C++, but we shall avoid using it.

- ***Inout parameters*** are used for both input and output; the initial value of the parameter is used and then modified by the function. We shall pass inout parameters by reference.

[3] Consult a C++ textbook for discussion of call by reference and reference variables.

In addition to its parameters, a function uses other data objects that generally fall into one of the following categories.

variables

- ***Local variables*** are defined in the function and exist only while the function is being executed. They are not initialized before the function begins and are discarded when the function ends.

- ***Global variables*** are used in the function but not defined in the function. It can be quite dangerous to use global variables in a function, since after the function is written its author may forget exactly what global variables were used and how. If the main program is later changed, then the function may mysteriously begin to misbehave. If a function alters the value of a global variable, it is said to cause a ***side effect***. Side effects are even more dangerous than using global variables as input to the function because side effects may alter the performance of other functions, thereby misdirecting the programmer's debugging efforts to a part of the program that is already correct.

side effects

Programming Precept

Keep your connections simple. Avoid global variables whenever possible.

Programming Precept

Never cause side effects if you can avoid it.
If you must use global variables as input, document them thoroughly.

While these principles of top-down design may seem almost self-evident, the only way to learn them thoroughly is by practice. Hence throughout this book we shall be careful to apply them to the large programs that we write, and in a moment it will be appropriate to return to our first example project.

Contents
Index
Help

Exercises 1.3

IRM

E1. What classes would you define in implementing the following projects? What methods would your classes possess?

(a) A program to store telephone numbers.
(b) A program to play Monopoly.
(c) A program to play tic-tac-toe.
(d) A program to model the build up of queues of cars waiting at a busy intersection with a traffic light.

E2. Rewrite the following class definition, which is supposed to model a deck of playing cards, so that it conforms to our principles of style.

```
    class a {                  //   a deck of cards
int X;  thing Y1[52];   /* X is the location of the top card in the deck.  Y1 lists
the cards. */ public:  a();
void Shuffle();                //   Shuffle randomly arranges the cards.
thing d();                     //   deals the top card off the deck
}
                ;
```

E3. Given the declarations

```
int a[n][n], i, j;
```

where n is a constant, determine what the following statement does, and rewrite the statement to accomplish the same effect in a less tricky way.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[i][j] = ((i + 1)/(j + 1)) * ((j + 1)/(i + 1));
```

E4. Rewrite the following function so that it accomplishes the same result in a less tricky way.

```
void does_something(int &first, int &second)
{
  first = second - first;
  second = second - first;
  first = second + first;
}
```

E5. Determine what each of the following functions does. Rewrite each function with meaningful variable names, with better format, and without unnecessary variables and statements.

(a)
```
int calculate(int apple, int orange)
  { int peach, lemon;
  peach = 0; lemon = 0; if (apple < orange)
  peach = orange; else if (orange <= apple)
  peach = apple; else { peach = 17;
  lemon = 19; }
  return(peach);
  }
```

(b) For this part assume the declaration `typedef float vector[max];`

```
float figure (vector vector1)
{ int loop1, loop4; float loop2, loop3;
loop1 = 0; loop2 = vector1[loop1]; loop3 = 0.0;
loop4 = loop1; for (loop4 = 0;
loop4 < max; loop4++) { loop1 = loop1 + 1;
loop2 = vector1[loop1 - 1];
loop3 = loop2 + loop3; } loop1 = loop1 - 1;
loop2 = loop1 + 1;
return(loop2 = loop3/loop2); }
```

(c)
```
int question(int &a17, int &stuff)
  { int another, yetanother, stillonemore;
  another = yetanother; stillonemore = a17;
  yetanother = stuff; another = stillonemore;
  a17 = yetanother; stillonemore = yetanother;
  stuff = another; another = yetanother;
  yetanother = stuff; }
```

Contents
Index
Help

(d)
```
int mystery(int apple, int orange, int peach)
{ if (apple > orange) if (apple > peach) if
(peach > orange) return(peach); else if (apple < orange)
return(apple); else return(orange); else return(apple); else
if (peach > apple) if (peach > orange) return(orange); else
return(peach); else return(apple); }
```

E6. The following statement is designed to check the relative sizes of three integers, which you may assume to be different from each other:

```
if (x < z) if (x < y) if (y < z) c = 1; else c = 2; else
  if (y < z) c = 3; else c = 4; else if (x < y)
  if (x < z) c = 5; else c = 6; else if (y < z) c = 7; else
  if (z < x) if (z < y) c = 8; else c = 9; else c = 10;
```

(a) Rewrite this statement in a form that is easier to read.
(b) Since there are only six possible orderings for the three integers, only six of the ten cases can actually occur. Find those that can never occur, and eliminate the redundant checks.
(c) Write a simpler, shorter statement that accomplishes the same result.

E7. The following C++ function calculates the cube root of a floating-point number (by the Newton approximation), using the fact that, if y is one approximation to the cube root of x, then

$$z = \frac{2y + x/y^2}{3}$$

cube roots

is a closer approximation.

```
float function fcn(float stuff)
{ float april, tim, tiny, shadow, tom, tam, square; int flag;
tim = stuff; tam = stuff; tiny = 0.00001;
if (stuff != 0) do {shadow = tim + tim; square = tim * tim;
tom = (shadow + stuff/square); april = tom/3.0;
if (april*april * april - tam > -tiny) if (april*april*april - tam
    < tiny) flag = 1; else flag = 0; else flag = 0;
if (flag == 0) tim = april; else tim = tam; } while (flag != 1);
if (stuff == 0) return(stuff); else return(april); }
```

(a) Rewrite this function with meaningful variable names, without the extra variables that contribute nothing to the understanding, with a better layout, and without the redundant and useless statements.
(b) Write a function for calculating the cube root of x directly from the mathematical formula, by starting with the assignment y = x and then repeating

```
y = (2 * y + (x/(y * y)))/3
```

until abs(y * y * y - x) < 0.00001.
(c) Which of these tasks is easier?

statistics

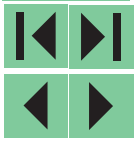
E8. The ***mean*** of a sequence of numbers is their sum divided by the count of numbers in the sequence. The (population) ***variance*** of the sequence is the mean of the squares of all numbers in the sequence, minus the square of the mean of the numbers in the sequence. The ***standard deviation*** is the square root of the variance. Write a well-structured C++ function to calculate the standard deviation of a sequence of n floating-point numbers, where n is a constant and the numbers are in an array indexed from 0 to $n - 1$, which is a parameter to the function. Use, then write, subsidiary functions to calculate the mean and variance.

plotting

E9. 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 function to plot one such pair of coordinates. The details of the function 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 function bounds that will read the whole file and determine these four maxima and minima. Afterward, another function 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 functions.
(b) Write the function bounds.
(c) Write the preconditions and postconditions for the remaining functions together with appropriate documentation showing their purposes and their requirements.

Contents
Index
Help



1.4 CODING, TESTING, AND FURTHER REFINEMENT

The three processes in the section title 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

early debugging and testing

After coding the main program, most programmers will wish to complete the writing and coding of the required classes and 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, the writing and coding will be such a large job that, by the time it is complete, many of the details of the main program and the classes and functions that were written early will have been forgotten. In fact, different people may be writing different functions, and some 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 class and function as soon as it is written than it is to wait until the project has been completely coded.

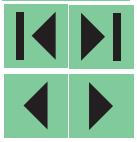
Even for smaller projects, there are good reasons for debugging classes and functions one at a time. We might, for example, be unsure of some point of C++ syntax that will appear in several places through the program. If we can compile each function separately, then we shall quickly learn to avoid errors 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.

stubs

To compile the main 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 little or nothing at all:

```
void instructions() { }
bool user_says_yes() { return(true); }
```

Contents
Index
Help



Note that in writing the stub functions we must at least pin down their associated parameters and return types. For example, in designing a stub for user_says_yes(), we make the decision that it should return a natural answer of **true** or **false**. This means that we should give the function a return type **bool**. The type **bool** has only recently been added to C++ and some older compilers do not recognize it, but we can always simulate it with the following statements—which can conveniently be placed in the utility package, if they are needed:

```
typedef int bool;
const bool false = 0;
const bool true = 1;
```

In addition to the stub functions, our program also needs a stub definition for the **class** Life. For example, in the file `life.h`, we could define this class without data members as follows:

```
class Life {
public:
   void initialize();
   void print();
   void update();
};
```

We must also supply the following stubs for its methods in `life.c`:

```
void Life::initialize() {}
void Life::print() {}
void Life::update() {}
```

Note that these method definitions have to use the C++ scope resolution operator :: [4] to indicate that they belong to the scope of the class Life.

Even with these minimal stubs we can at least compile the program and make sure that the definitions of types and variables are syntactically correct. Normally, however, each stub function should print a message stating that the function was invoked. When we execute the program, we find that it runs into an infinite loop, because the function user_says_yes() always returns a value of **true**. However, the main program compiles and runs, so we can go on to refine our stubs. For a small project like the Life game, we can simply write each class or function in turn, substitute it for its stub, and observe the effect on program execution.

1.4.2 Definition of the Class Life

1: living cell
0: dead cell

Each Life object needs to include a rectangular array,[5] which we shall call grid, to store a Life configuration. We use an integer entry of 1 in the array grid to denote a living cell, and 0 to denote a dead cell. Thus to count the number of neighbors of a particular cell, we just add the values of the neighboring cells. In fact, in updating a Life configuration, we shall repeatedly need to count the number of living neighbors of individual cells in the configuration. Hence, the class Life should include a member function neighbor_count that does this task. Moreover, since the member neighbor_count is not needed by client code, we shall give it **private** visibility. In contrast, the earlier Life methods all need to have **public** visibility. Finally, we must settle on dimensions for the rectangular array carried in a Life configuration. We code these dimensions as global constants, so that a single simple change is all that we need to reset grid sizes in our program. Note that constant definitions can be safely placed in .h files.


```
const int maxrow = 20, maxcol = 60;   // grid dimensions

class Life {
public:
   void initialize();
   void print();
   void update();
private:
   int grid[maxrow + 2][maxcol + 2];
                                 // allows for two extra rows and columns
   int neighbor_count(int row, int col);
};
```

We can test the definition, without writing the member functions, by using our earlier stub methods together with a similar stub for the private function neighbor_count.

[4] Consult a C++ textbook for discussion of the scope resolution operator and the syntax for class methods.

[5] An array with two indices is called ***rectangular***. The first index determines the ***row*** in the array and the second the ***column***.

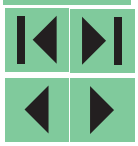
1.4.3 Counting Neighbors

function neighbor_count

Let us now refine our program further. The function that counts neighbors of the cell with coordinates row, col requires that we look in the eight adjoining cells. We shall use a pair of **for** loops to do this, one running from row − 1 to row + 1 and the other from col − 1 to col + 1. We need only be careful, when row, col is on a boundary of the grid, that we look only at legitimate cells in the grid. Rather than using several **if** statements to make sure that we do not go outside the grid, we introduce a ***hedge*** around the grid: We shall enlarge the grid by adding two extra rows, one before the first real row of the grid and one after the last, and two extra columns, one before the first column and one after the last. In our definition of the **class** Life, we anticipated the hedge by defining the member grid as an array with maxrow + 2 rows and maxcol + 2 columns. The cells in the hedge rows and columns will always be dead, so they will not affect the counts of living neighbors at all. Their presence, however, means that the **for** loops counting neighbors need make no distinction between rows or columns on the boundary of the grid and any other rows or columns. See the examples in Figure 1.2.

hedge

Contents
Index
Help



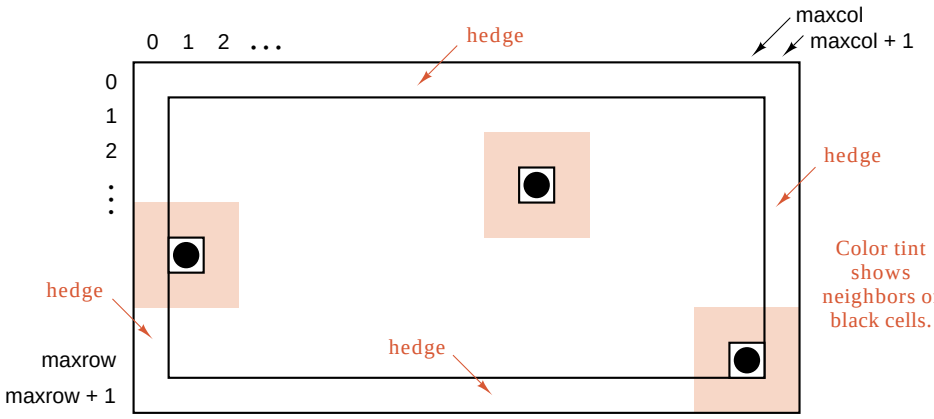


Figure 1.2. Life grid with a hedge

sentinel

Another term often used instead of hedge is ***sentinel***: A sentinel is an extra entry put into a data structure so that boundary conditions need not be treated as a special case.

```
int Life::neighbor_count(int row, int col)
/* Pre:  The Life object contains a configuration, and the coordinates row and col
         define a cell inside its hedge.
 Post:  The number of living neighbors of the specified cell is returned. */
```

```
{
   int i, j;
   int count = 0;
   for (i = row - 1; i <= row + 1; i++)
      for (j = col - 1; j <= col + 1; j++)
         count += grid[i][j];    // Increase the count if neighbor is alive.
   count -= grid[row][col];
                                 // Reduce count, since cell is not its own neighbor.
   return count;
}
```

1.4.4 Updating the Grid

method update

The action of the method to update a Life configuration is straightforward. We first use the data stored in the configuration to calculate entries of a rectangular array called new_grid that records the updated configuration. We then copy new_grid, entry by entry, back to the grid member of our Life object.

To set up new_grid we use a nested pair of loops on row and col that run over all non-hedge entries in the rectangular array grid. The body of these nested loops consists of the multiway selection statement **switch**. The function neighbor_count(row, col) returns one of the values 0, 1, . . . , 8, and for each of these cases we can take a separate action, or, as in our application, 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.


```
void Life::update()
/* Pre:  The Life object contains a configuration.
   Post: The Life object contains the next generation of configuration. */
{
   int row, col;
   int new_grid[maxrow + 2][maxcol + 2];
   for (row = 1; row <= maxrow; row++)
      for (col = 1; col <= maxcol; col++)
         switch (neighbor_count(row, col)) {
         case 2:
            new_grid[row][col] = grid[row][col];    // Status stays the same.
            break;
         case 3:
            new_grid[row][col] = 1;    // Cell is now alive.
            break;
         default:
            new_grid[row][col] = 0;    // Cell is now dead.
         }
   for (row = 1; row <= maxrow; row++)
      for (col = 1; col <= maxcol; col++)
         grid[row][col] = new_grid[row][col];
}
```

Contents
Index
Help

1.4.5 Input and Output

careful input and output

It now remains only to write the Life methods initialize() and print(), with the functions user_says_yes() and instructions() that do the input and output for our program. In computer programs designed to be used by many people, the functions performing input and output are often the longest. Input to the program must be fully checked to be certain that it is valid and consistent, and errors in input must be processed in ways to avoid catastrophic failure or production of ridiculous results. The output must be carefully organized and formatted, with considerable thought to what should or should not be printed, and with provision of various alternatives to suit differing circumstances.

Programming Precept

Keep your input and output as separate functions,
so they can be changed easily
and can be custom tailored to your computing system.

1. Instructions

stream output operators

The instructions() function is a simple exercise in use of the *put to* operator ≪ and the standard output stream called cout. Observe that we use the manipulator endl to end a line and flush the output buffer. The manipulator flush can be used instead in situations where we just wish to flush the output buffer, without ending a line. For the precise details of stream input and output in C++, consult a textbook on C++.

```
void instructions()
/* Pre:   None.
   Post:  Instructions for using the Life program have been printed. */
{
  cout ≪ "Welcome to Conway's game of Life." ≪ endl;
  cout  ≪ "This game uses a grid of size "
        ≪ maxrow ≪ " by " ≪ maxcol ≪ " in which" ≪ endl;
  cout ≪ "each cell can either be occupied by an organism or not." ≪ endl;
  cout ≪ "The occupied cells change from generation to generation" ≪ endl;
  cout  ≪ "according to the number of neighboring cells which are alive."
        ≪ endl;
}
```

2. Initialization

input method

The task that the Life method initialize() must accomplish is to set up an initial configuration. To initialize a Life object, we could consider each possible coordinate pair separately and request the user to indicate whether the cell is to be occupied or not. This method would require the user to type in

maxrow * maxrow = 20 * 60 = 1200

entries, which is prohibitive. Hence, instead, we input only those coordinate pairs corresponding to initially occupied cells.

```
void Life::initialize()
// Pre:  None.
//  Post: The Life object contains a configuration specified by the user. *//
{
  int row, col;
  for (row = 0; row <= maxrow + 1; row++)
    for (col = 0; col <= maxcol + 1; col++)
      grid[row][col] = 0;
  cout << "List the coordinates for living cells." << endl;
  cout << "Terminate the list with the the special pair -1 -1" << endl;
  cin >> row >> col;
  while (row != -1 || col != -1) {
    if (row >= 1 && row <= maxrow)
      if (col >= 1 && col <= maxcol)
        grid[row][col] = 1;
      else
        cout << "Column " << col << " is out of range." << endl;
    else
      cout << "Row " << row << " is out of range." << endl;
    cin >> row >> col;
  }
}
```

output

For the output method print() we adopt the simple method of writing out the entire rectangular array at each generation, with occupied cells denoted by * and empty cells by blanks.

```
void Life::print()
// Pre:  The Life object contains a configuration.
//  Post: The configuration is written for the user. *//
{
  int row, col;
  cout << "\nThe current Life configuration is:" << endl;
  for (row = 1; row <= maxrow; row++) {
    for (col = 1; col <= maxcol; col++)
      if (grid[row][col] == 1) cout << '*';
      else cout << ' ';
    cout << endl;
  }
  cout << endl;
}
```

Contents
Index
Help

response from user

Finally comes the function user_says_yes(), which determines whether the user wishes to go on to calculate the next generation. The task of user_says_yes() is to ask the user to respond yes or no. To make the program more tolerant of mistakes in input, this request is placed in a loop that repeats until the user's response is acceptable. In our function, we use the standard input function get() to process input characters one at a time. In C++, the function get() is actually just a method of the **class** istream: In our application, we apply the method, cin.get(), that belongs to the istream object cin.

```
bool user_says_yes()
{
  int c;
  bool initial_response = true;
  do {                              //  Loop until an appropriate input is received.
    if (initial_response)
      cout << " (y,n)? " << flush;
    else
      cout << "Respond with either y or n: " << flush;
    do {                            //  Ignore white space.
      c = cin.get();
    } while (c == '\n' || c == ' ' || c == '\t');
    initial_response = false;
  } while (c != 'y' && c != 'Y' && c != 'n' && c != 'N');
  return (c == 'y' || c == 'Y');
}
```

At this point, we have all the functions for the Life simulation. It is time to pause and check that it works.

1.4.6 Drivers

separate debugging

For small projects, each function is usually inserted in its proper place as soon as it is written, and the resulting program can then be debugged and tested as far as possible. For large projects, however, compilation of the entire project can overwhelm that of a new function being debugged, and it can be difficult to tell, looking only at the way the whole program runs, whether a particular function is working correctly or not. Even in small projects the output of one function may be used by another in ways that do not immediately reveal whether the information transmitted is correct.

driver program

One way to debug and test a single function is to write a short auxiliary program whose purpose is to provide the necessary input for the function, call it, and evaluate the result. Such an auxiliary program is called a ***driver*** for the function. By using drivers, each function can be isolated and studied by itself, and thereby errors can often be spotted quickly.

As an example, let us write drivers for the functions of the Life project. First, we consider the method neighbor_count(). In our program, its output is used but has not been directly displayed for our inspection, so we should have little confidence that it is correct. To test neighbor_count() we shall supply a Life object configuration, call neighbor_count for every cell of configuration, and write out the results.

Contents
Index
Help

The resulting driver uses configuration.initialize() to set up the object and bears some resemblance to the original main program. In order to call neighbor_count(), from the driver, we need to adjust its visibility temporarily to become **public** in the **class** Life.

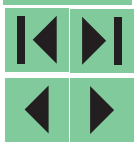
```
int main ()                          //  driver for neighbor_count()
/* Pre:  None.
   Post: Verifies that the method neighbor_count() returns the correct values.
   Uses: The class Life and its method initialize(). */
{
   Life configuration;
   configuration.initialize();
   for (row = 1; row <= maxrow; row++){
      for (col = 1; col <= maxrow; col++)
         cout << configuration.neighbor_count(row, col) << " ";
      cout << endl;
   }
}
```

Sometimes two functions can be used to check each other. The easiest way, for example, to check the Life methods initialize() and print() is to use a driver whose action part is

```
configuration.initialize();
configuration.print();
```

Both methods can be tested by running this driver and making sure that the configuration printed is the same as that given as input.

Contents
Index
Help



1.4.7 Program Tracing

group discussion

After the functions have been assembled into a complete program, it is time to check out the completed whole. One of the most effective ways to uncover hidden defects is called a ***structured walkthrough***. In this the programmer shows the completed program to another programmer or a small group of programmers and explains exactly what happens, beginning with an explanation of the main program followed by the functions, one by one. Structured walkthroughs are helpful for three reasons. First, programmers who are not familiar with the actual code can often spot bugs or conceptual errors that the original programmer overlooked. Second, the questions that other people ask can help you to clarify your own thinking and discover your own mistakes. Third, the structured walkthrough often suggests tests that prove useful in later stages of software production.

print statements for debugging

It is unusual for a large program to run correctly the first time it is executed as a whole, and if it does not, it may not be easy to determine exactly where the errors are. On many systems sophisticated ***trace tools*** are available to keep track of function calls, changes of variables, and so on. A simple and effective debugging tool, however, is to take ***snapshots*** of program execution by inserting printing statements at key points in the main program; this strategy is often available as an option in a debugger when one is available. A message can be printed each time a

function is called, and the values of important variables can be printed before and after each function is called. Such snapshots can help the programmer converge quickly on the particular location where an error is occurring.

temporary scaffolding

Scaffolding is another term frequently used to describe code inserted into a program to help with debugging. Never hesitate to put scaffolding into your programs as you write them; it will be easy to delete once it is no longer needed, and it may save you much grief during debugging.

When your program has a mysterious error that you cannot localize at all, then it is very useful to put scaffolding into the main program to print the values of important variables. This scaffolding should be put at one or two of the major dividing points in the main program. (If you have written a program of any significant size that does not subdivide its work into several major sections, then you have already made serious errors in the design and structure of your program that you should correct.) With printouts at the major dividing points, you should be able to determine which section of the program is misbehaving, and then you can concentrate on that section, introducing scaffolding into its subdivisions.

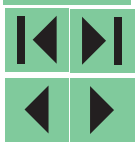
defensive programming

Another important method for detecting errors is to practice ***defensive*** programming. Put **if** statements at the beginning of functions to check that the preconditions do in fact hold. If not, print an error message. In this way, you will be alerted as soon as a supposedly impossible situation arises, and if it does not arise, the error checking will be completely invisible to the user. It is, of course, particularly important to check that the preconditions hold when the input to a function comes from the user, or from a file, or from some other source outside the program itself. It is, however, surprising how often checking preconditions will reveal errors even in places where you are sure everything is correct.

static analyzer

For very large programs yet another tool is sometimes used. This is a ***static analyzer***, a program that examines the source program (as written in C++, for example) looking for uninitialized or unused variables, sections of the code that can never be reached, and other occurrences that are probably incorrect.

Contents
Index
Help



1.4.8 Principles of Program Testing

choosing test data

So far we have said nothing about the choice of data to be used to test programs and functions. This choice, of course, depends intimately on the project under development, so we can make only some general remarks. First we should note the following:

Programming Precept

The quality of test data is more important than its quantity.

Many sample runs that do the same calculations in the same cases provide no more effective a test than one run.

Programming Precept

Program testing can be used to show the presence of bugs, but never their absence.

It is possible that other cases remain that have never been tested even after many sample runs. For any program of substantial complexity, it is impossible to perform exhaustive tests, yet the careful choice of test data can provide substantial confidence in the program. Everyone, for example, has great confidence that the typical computer can add two floating-point numbers correctly, but this confidence is certainly not based on testing the computer by having it add all possible floating-point numbers and checking the results. If a double-precision floating-point number takes 64 bits, then there are 2^{128} distinct pairs of numbers that could be added. This number is astronomically large: All computers manufactured to date have performed altogether but a tiny fraction of this number of additions. Our confidence that computers add correctly is based on tests of each component separately; that is, by checking that each of the 64 digits is added correctly and that carrying from one place to another is done correctly.

testing methods

There are at least three general philosophies that are used in the choice of test data.

1. The Black-Box Method

Most users of a large program are not interested in the details of its functioning; they only wish to obtain answers. That is, they wish to treat the program as a black box; hence the name of this method. Similarly, test data should be chosen according to the specifications of the problem, without regard to the internal details of the program, to check that the program operates correctly. At a minimum the test data should be selected in the following ways:

Contents
Index
Help

data selection

1. ***Easy values***. The program should be debugged with data that are easy to check. More than one student who tried a program only for complicated data, and thought it worked properly, has been embarrassed when the instructor tried a trivial example.
2. ***Typical, realistic values***. Always try a program on data chosen to represent how the program will be used. These data should be sufficiently simple so that the results can be checked by hand.
3. ***Extreme values***. Many programs err at the limits of their range of applications. It is very easy for counters or array bounds to be off by one.
4. ***Illegal values***. "Garbage in, garbage out" is an old saying in computer circles that should not be respected. When a good program has garbage coming in, then its output should at least be a sensible error message. Indeed, the program should provide some indication of the likely errors in input and perform any calculations that remain possible after disregarding the erroneous input.

2. The Glass-Box Method

The second approach to choosing test data begins with the observation that a program can hardly be regarded as thoroughly tested if there are some parts of its code that, in fact, have never been executed. In the ***glass-box*** method of testing, the logical structure of the program is examined, and for each alternative that may occur, test data are devised that will lead to that alternative. Thus care is taken to choose data to check each possibility in every **switch** statement, each clause of

path testing

every **if** statement, and the termination condition of each loop. If the program has several selection or iteration statements, then it will require different combinations of test data to check all the paths that are possible. Figure 1.3 shows a short program segment with its possible execution paths.

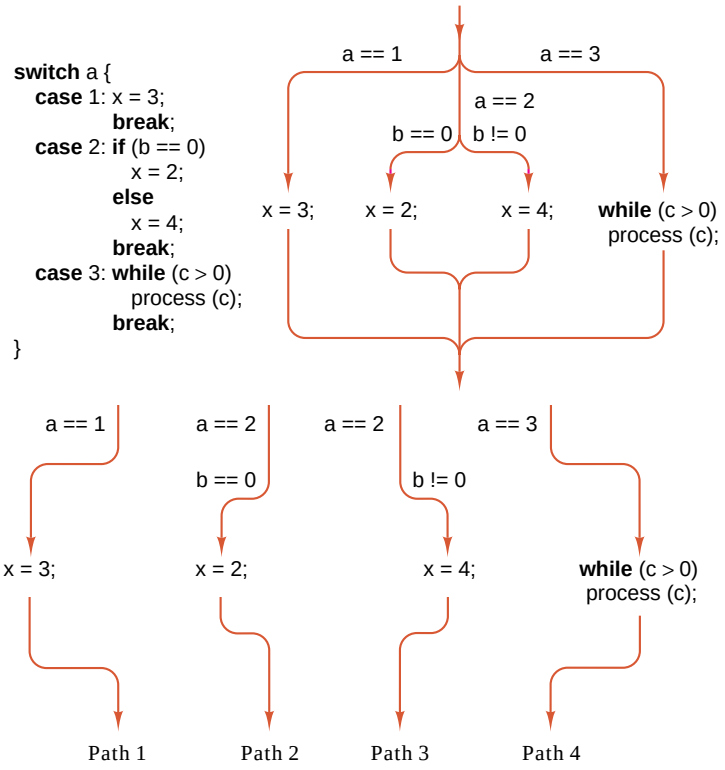


Figure 1.3. The execution paths through a program segment

For a large program the glass-box approach is clearly not practicable, but for a single small module, it is an excellent debugging and testing method. In a well-designed program, each module will involve few loops and alternatives. Hence only a few well-chosen test cases will suffice to test each module on its own.

modular testing

In glass-box testing, the advantages of modular program design become evident. Let us consider a typical example of a project involving 50 functions, each of which can involve 5 different cases or alternatives. If we were to test the whole program as one, we would need 5^{50} test cases to be sure that each alternative was tested. Each module separately requires only 5 (easier) test cases, for a total of $5 \times 50 = 250$. Hence a problem of impossible size has been reduced to one that, for a large program, is of quite modest size.

comparison

Before you conclude that glass-box testing is always the preferable method, we should comment that, in practice, black-box testing is usually more effective in uncovering errors. Perhaps one reason is that the most subtle programming errors often occur not within a function but in the interface between functions, in

interface errors

Contents
Index
Help

misunderstanding of the exact conditions and standards of information interchange between functions. It would therefore appear that a reasonable testing philosophy for a large project would be to apply glass-box methods to each small module as it is written and use black-box test data to test larger sections of the program when they are complete.

3. The Ticking-Box Method

To conclude this section, let us mention one further philosophy of program testing, a philosophy that is, unfortunately, quite widely used. This might be called the ***ticking-box*** method. It consists of doing no testing at all after the project is fairly well debugged, but instead turning it over to the customer for trial and acceptance. The result, of course, is a time bomb.

Exercises 1.4

E1. If you suspected that the Life program contained errors, where would be a good place to insert scaffolding into the main program? What information should be printed out?

E2. Take your solution to Section 1.3, Exercise E9 (designing a program to plot a set of points), and indicate good places to insert scaffolding if needed.

E3. Find suitable black-box test data for each of the following:

(a) A function that returns the largest of its three parameters, which are floating-point numbers.

(b) A function that returns the square root of a floating-point number.

(c) A function that returns the least common multiple of its two parameters, which must be positive integers. (The ***least common multiple*** is the smallest integer that is a multiple of both parameters. Examples: The least common multiple of 4 and 6 is 12, of 3 and 9 is 9, and of 5 and 7 is 35.)

(d) A function that sorts three integers, given as its parameters, into ascending order.

(e) A function that sorts an array `a` containing `n` integers indexed from 0 to `n – 1` into ascending order, where `a` and `n` are both parameters.

E4. Find suitable glass-box test data for each of the following:

(a) The statement

```
if (a < b) if (c > d) x = 1;  else if (c ==  d) x = 2;
else x = 3;  else if (a ==  b) x = 4;  else if (c ==  d) x = 5;
else x = 6;
```

(b) The Life method `neighbor_count(row, col)`.

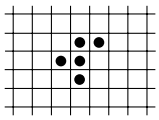
Programming Projects 1.4

P1. Enter the Life program of this chapter on your computer and make sure that it works correctly.

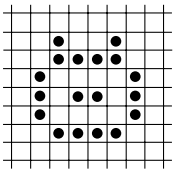
P2. Test the Life program with the examples shown in Figure 1.1.

P3. Run the Life program with the initial configurations shown in Figure 1.4. Several of these go through many changes before reaching a configuration that remains the same or has predictable behavior.

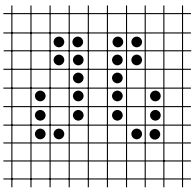
Contents
Index
Help



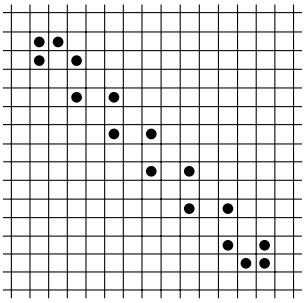
R Pentomino



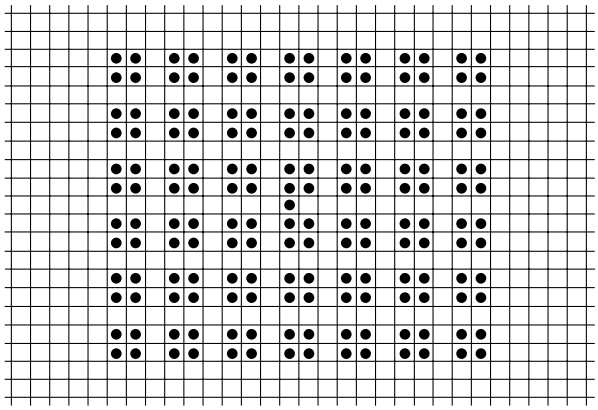
Cheshire Cat



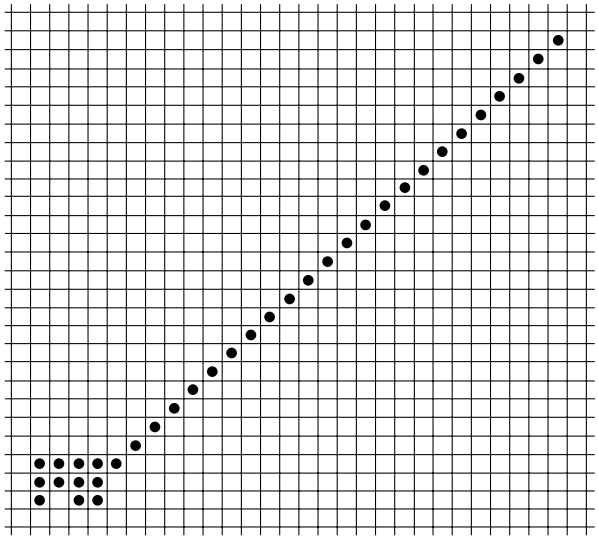
Tumbler



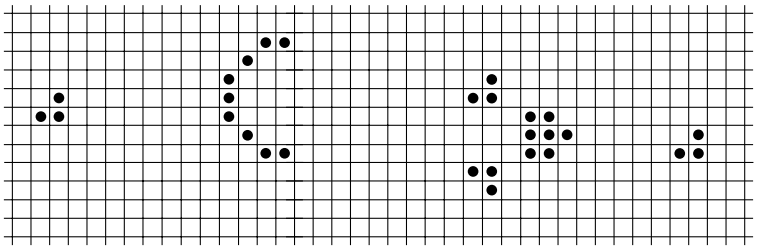
Barber Pole



Virus



Harvester



The Glider Gun

Figure 1.4. Life configurations

Contents
Index
Help

1.5 PROGRAM MAINTENANCE

Small programs written as exercises or demonstrations are usually run a few times and then discarded, but the disposition of large practical programs is quite different. A program of practical value will be run many times, usually by many different people, and its writing and debugging mark only the beginning of its use. They also mark only the beginning of the work required to make and keep the program useful. It is necessary to *review* and *analyze* the program to ensure that it meets the requirements specified for it, *adapt* it to changing environments, and *modify* it to make it better meet the needs of its users.

Maintenance of a computer program encompasses all this work done to a program after it has been fully debugged, tested, and put into use. With time and experience, the expectations for a computer program will generally change. The operating and hardware environment will change; the needs and expectations of users will change; the interface with other parts of the software system will change. Hence, if a program is to have continued usefulness, continuing attention must be given to keep it up to date. In fact, surveys show the following:

Contents
Index
Help

Programming Precept

For a large and important program, more than half the work comes in the maintenance phase, after it has been completely debugged, tested, and put into use.

1.5.1 Program Evaluation

The first step of program maintenance is to begin the continuing process of review, analysis, and evaluation. There are several useful questions we may ask about any program. The first group of questions concerns the use and output of the program (thus continuing what is started with black-box testing).

1. Does the program solve the problem that is requested, following the problem specifications exactly?
2. Does the program work correctly under all conditions?
3. Does the program have a good user interface? Can it receive input in forms convenient and easy for the user? Is its output clear, useful, and attractively presented? Does the program provide alternatives and optional features to facilitate its use? Does it include clear and sufficient instructions and other information for the user?

   The remaining questions concern the structure of the program (continuing the process begun in glass-box testing).

4. Is the program logically and clearly written, with convenient classes and short functions as appropriate to do logical tasks? Are the data structured into classes that accurately reflect the needs of the program?
5. Is the program well documented? Do the names accurately reflect the use and meaning of variables, functions, types, and methods? Are precise pre- and postconditions given as appropriate? Are explanations given for major sections of code or for any unusual or difficult code?
6. Does the program make efficient use of time and of space? By changing the underlying algorithm, could the program's performance be improved?

Some of these criteria will be closely studied for the programs we write. Others will not be mentioned explicitly, but not because of any lack of importance. These criteria, rather, can be met automatically if sufficient thought and effort are invested in every stage of program design. We hope that the examples we study will reveal such care.

1.5.2 Review of the Life Program

Contents
Index
Help

Let us illustrate these program-evaluation criteria by reconsidering the program for the Life game. Doing so, in one sense, is really overkill, since a toy project like the Life game is not, in itself, worth the effort. In the process, however, we shall consider programming methods important for many other applications. Let us consider each of the preceding questions in turn.

1. Problem Specification

problem: the boundary

If we go back to review the rules for the Life game in Section 1.2.1, we will find that we have not, in fact, been solving the Life game as it was originally described. The rules make no mention of the boundaries of the grid containing the cells. In our program, when a moving colony gets sufficiently close to a boundary, then room for neighbors disappears, and the colony will be distorted by the very presence of the boundary. That is not supposed to be. Hence our program violates the rules.

It is of course true that in any computer simulation there are absolute bounds on the values that may appear, but certainly the use of a 20 by 60 grid in our program is highly restrictive and arbitrary. It is possible to write a Life program without restricting the size of the grid, but before we can do so, we must develop several sophisticated data structures. Only after we have done so can we, in Section 9.9, write a general Life program without restrictions on the size of the grid.

On a first try, however, it is quite reasonable to restrict the problem being solved, and hence, for now, let us continue studying Life on a grid of limited size. It is, nevertheless, very important to say exactly what we are doing:

Programming Precept

Be sure you understand your problem completely.
If you must change its terms, explain exactly what you have done.

2. Program Correctness

Since program testing can show the presence of errors but not their absence, we need other methods to *prove* beyond doubt that a program is correct. Constructing formal proofs that a program is correct is often difficult but sometimes it can be done, as we shall do for some of the sophisticated algorithms developed in later chapters. For the Life game, let us be content with more informal reasons why our program is correct.

First, we ask which parts of the program need verification. The Life configuration is changed only by the method update, and only update and neighbor_count involve any calculation that might turn out to be wrong. Hence we should concentrate on the correctness of these two methods.

correctness of neighbor_count

The method neighbor_count looks only at the cell given as its parameters and at the neighbors of that cell. There are only a limited number of possibilities for the status of the cell and its neighbors, so glass-box testing of these possibilities is feasible, using a driver program for neighbor_count. Such testing would quickly convince us of the correctness of neighbor_count.

correctness of update

For update, we should first examine the cases in the **switch** statement to make sure that their actions correspond exactly to the rules in Section 1.2.1. Next, we can note that the action for each cell depends only on the status of the cell and it neighbor count. Hence, as for neighbor_count, we can construct a limited set of glass-box test data that verify that update performs the correct action in each possible case.

Contents
Index
Help

3. User Interface

problem: input

In running the Life program, you will have likely found that the poor method for input of the initial configuration is a major inconvenience. It is unnatural for a person to calculate and type in the numerical coordinates of each living cell. The form of input should instead reflect the same visual imagery that we use to print a configuration. At a minimum, the program should allow the user to type each row of the configuration as a line of blanks (for dead cells) and non-blank characters (for living cells).

file input and output

Life configurations can be quite complicated. For easier input, the program should be able to read its initial configuration from a file. To allow stopping the program to be resumed later, the program should also be able to store the final configuration in a file that can be read again later.

editing

Another option would be to allow the user to edit a configuration at any generation.

output improvements

The output from the program can also be improved. Rather than rewriting the entire configuration at each generation, direct cursor addressing should be used to change only the cells whose status has changed. Color or other features can be used to make the output both much more attractive and more useful. For example, cells that have newly become alive might be one color and those continuing alive other colors depending on how long they have been alive.

help screen

To make the program more self-contained, it would also be useful to have an optional display of a short description of the Life game and its rules, perhaps as a pop-up screen.

In general, designing a program to have an attractive appearance and feel to the user is very important, and in large programs a great deal of importance is given to the user interface, often more than to all other parts of the program combined.

Programming Precept

Design the user interface with the greatest care possible.
A program's success depends greatly on its attractiveness and ease of use.

4. Modularity and Structure

We have already addressed these issues in the original design. The decisions already made will continue to serve us well.

5. Documentation

Again, we have previously addressed issues of documentation, which need not be repeated here.

6. Efficiency

Where does the Life program spend most of its time? Surely it is not in the input phase, since that is done only once. The output too is generally quite efficient. The bulk of the calculation is in method update and in neighbor_count, which it invokes.

At every generation, update recalculates the neighbor counts of every possible cell. In a typical configuration, perhaps only five percent of the cells are living, often localized in one area of the grid. Hence update spends a great deal of time laboriously establishing that many dead cells, with no living neighbors, indeed have neighbor counts of 0 and will remain dead in the next generation. If 95 percent of the cells are dead, this constitutes a substantial inefficiency in the use of computer time.

poor speed

But is this inefficiency of any importance? Generally, it is not, since the calculations are done so quickly that, to the user, each generation seems to appear instantaneously. On the other hand, if you run the Life program on a very slow machine or on a busy time-sharing system, you may find the program's speed somewhat disappointing, with a noticeable pause between printing one generation and starting to print the next. In this case, it might be worthwhile to try saving computer time, but, generally speaking, optimization of the Life program is not needed even though it is very inefficient.

Programming Precept

Do not optimize your code unless it is necessary to do so.
Do not start to optimize code until it is complete and correct.

Most programs spend 90 percent of their time
doing 10 percent of their instructions.
Find this 10 percent, and concentrate your efforts for efficiency there.

Contents
Index
Help

Another reason to think carefully before commencing optimization of a program is that optimizations often produce more complicated code. This code will then be harder to debug and to modify when necessary.

Programming Precept

Keep your algorithms as simple as you can.
When in doubt, choose the simple way.

1.5.3 Program Revision and Redevelopment

As we continue to evaluate a program, asking whether it meets its objectives and the needs of its users, we are likely to continue discovering both deficiencies in its current design and new features that could make it more useful. Hence program review leads naturally to program revision and redevelopment.

As we review the Life program, for example, we find that it meets some of the criteria quite well, but it has several deficiencies in regard to other criteria. The most serious of these is that, by limiting the grid size, it fails to satisfy its specifications. Its user interface leaves much to be desired. Finally, its computations are inefficient, but this is probably not important.

With some thought, we can easily improve the user interface for the Life program, and several of the projects propose such improvements. To revise the program to remove the limits on grid size, however, will require that we use data structures and algorithms that we have not yet developed, and hence we shall revisit the Life program in Section 9.9. At that time, we shall find that the algorithm we develop also addresses the question of efficiency. Hence the new program will both meet more general requirements and be more efficient in its calculations.

Programming Precept

Sometimes postponing problems simplifies their solution.

Contents
Index
Help

Exercises 1.5

IRM

E1. Sometimes the user might wish to run the Life game on a grid smaller than 20×60. Determine how it is possible to make maxrow and maxcol into variables that the user can set when the program is run. Try to make as few changes in the program as possible.

E2. One idea for speeding up the function Life::neighbor_count(row, col) is to delete the hedge (the extra rows and columns that are always dead) from the arrays grid and new_grid. Then, when a cell is on the boundary, neighbor_count will look at fewer than the eight neighboring cells, since some of these are outside the bounds of the grid. To do this, the function will need to determine whether or not the cell (row, col) is on the boundary, but this can be done outside the nested loops, by determining, before the loops commence, the lower and upper bounds for the loops. If, for example, row is as small as allowed,

then the lower bound for the row loop is row; otherwise, it is row – 1. Determine, in terms of the size of the grid, approximately how many statements are executed by the original version of neighbor_count and by the new version. Are the changes proposed in this exercise worth making?

Programming Projects 1.5

P1. Modify the Life function initialize so that it sets up the initial Life::grid configuration by accepting occupied positions as a sequence of blanks and x's in appropriate rows, rather than requiring the occupied positions to be entered as numerical coordinate pairs.

P2. Add a feature to the function initialize so that it can, at the user's option, either read its initial configuration from the keyboard or from a file. The first line of the file will be a comment giving the name of the configuration. Each remaining line of the file will correspond to a row of the configuration. Each line will contain x in each living position and a blank in each dead position.

P3. Add a feature to the Life program so that, at termination, it can write the final configuration to a file in a format that can be edited by the user and that can be read in to restart the program (using the feature of Project P2).

P4. Add a feature to the Life program so, at any generation, the user can edit the current configuration by inserting new living cells or by deleting living cells.

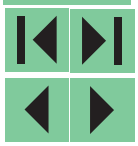
P5. Add a feature to the Life program so, if the user wishes at any generation, it will display a help screen giving the rules for the Life game and explaining how to use the program.

P6. Add a step mode to the Life program, so it will explain every change it makes while going from one generation to the next.

P7. Use direct cursor addressing (a system-dependent feature) to make the Life method print update the configuration instead of completely rewriting it at each generation.

P8. Use different colors in the Life output to show which cells have changed in the current generation and which have not.

Contents
Index
Help



1.6 CONCLUSIONS AND PREVIEW

This chapter has surveyed a great deal of ground, but mainly from a bird's-eye view. Some themes we shall treat in much greater depth in later chapters; others must be postponed to more advanced courses; still others are best learned by practice.

This section recapitulates and expands some of the principles we have been studying.

1.6.1 Software Engineering

Software engineering is the study and practice of methods helpful for the construction and maintenance of large software systems. Although small by realistic standards, the program we have studied in this chapter illustrates many aspects of software engineering.

Software engineering begins with the realization that it is a very long process to obtain good software. It begins before any programs are coded and continues as maintenance for years after the programs are put into use. This continuing process is known as the ***life cycle*** of software. This life cycle can be divided into phases as follows:

phases of life cycle

1. *Analyze* the problem precisely and completely. Be sure to *specify* all necessary user interface with care.
2. *Build* a prototype and *experiment* with it until all specifications can be finalized.
3. *Design* the algorithm, using the tools of data structures and of other algorithms whose function is already known.
4. *Verify* that the algorithm is correct, or make it so simple that its correctness is self-evident.
5. *Analyze* the algorithm to determine its requirements and make sure that it meets the specifications.
6. *Code* the algorithm into the appropriate programming language.
7. *Test* and *evaluate* the program on carefully chosen test data.
8. *Refine* and *repeat* the foregoing steps as needed for additional classes and functions until the software is complete and fully functional.
9. *Optimize* the code to improve performance, but only if necessary.
10. *Maintain* the program so that it will meet the changing needs of its users.

Most of these topics have been discussed and illustrated in various sections of this and the preceding chapter, but a few further remarks on the first phase, problem analysis and specification, are in order.

Contents
Index
Help

1.6.2 Problem Analysis

Analysis of the problem is often the most difficult phase of the software life cycle. This is not because practical problems are conceptually more difficult than are computing science exercises—the reverse is often the case—but because users and programmers tend to speak different languages. Here are some questions on which the analyst and user must reach an understanding:

specifications

1. What form will the input and output data take? How much data will there be?
2. Are there any special requirements for the processing? What special occurrences will require separate treatment?
3. Will these requirements change? How? How fast will the demands on the system grow?
4. What parts of the system are the most important? Which must run most efficiently?
5. How should erroneous data be treated? What other error processing is needed?
6. What kinds of people will use the software? What kind of training will they have? What kind of user interface will be best?

7. How portable must the software be, so that it can move to new kinds of equipment? With what other software and hardware systems must the project be compatible?
8. What extensions or other maintenance are anticipated? What is the history of previous changes to software and hardware?

1.6.3 Requirements Specification

For a large project, the phase of problem analysis and experimentation should eventually lead to a formal statement of the requirements for the project. This statement becomes the primary way in which the user and the software engineer attempt to understand each other and establishes the standard by which the final project will be judged. Among the contents of this specification will be the following:

1. *Functional requirements* for the system: what it will do and what commands will be available to the user.
2. *Assumptions* and *limitations* on the system: what hardware will be used for the system, what form must the input take, what is the maximum size of input, what is the largest number of users, and so on.
3. *Maintenance requirements*: anticipated extensions of the system, changes in hardware, changes in user interface.
4. *Documentation requirements*: what kind of explanatory material is required for what kinds of users.

The requirements specifications state *what* the software will do, not *how* it will be done. These specifications should be understandable both to the user and to the programmer. If carefully prepared, they will form the basis for the subsequent phases of design, coding, testing, and maintenance.

Contents
Index
Help

1.6.4 Coding

specifications complete

In a large software project it is necessary to do the coding at the right time, not too soon and not too late. Most programmers err by starting to code too soon. If coding is begun before the specifications are made precise, then unwarranted assumptions about the specifications will inevitably be made while coding, and these assumptions may render different classes and functions incompatible with each other or make the programming task much more difficult than it need be.

Programming Precept

Never code until the specifications are precise and complete.

Programming Precept

Act in haste and repent at leisure.
Program in haste and debug forever.

top-down coding

It is possible but unlikely, on the other hand, to delay coding too long. Just as we design from the top down, we should code from the top down. Once the specifications at the top levels are complete and precise, we should code the classes and functions at these levels and test them by including appropriate stubs. If we then find that our design is flawed, we can modify it without paying an exorbitant price in low-level functions that have been rendered useless.

The same thought can be expressed somewhat more positively:

Programming Precept

Starting afresh is often easier than patching an old program.

A good rule of thumb is that, if more than ten percent of a program must be modified, then it is time to rewrite the program completely. With repeated patches to a large program, the number of bugs tends to remain constant. That is, the patches become so complicated that each new patch tends to introduce as many new errors as it corrects.

Contents
Index
Help

An excellent way to avoid having to rewrite a large project from scratch is to plan from the beginning to write two versions. Before a program is running, it is often impossible to know what parts of the design will cause difficulty or what features need to be changed to meet the needs of the users. Engineers have known for many years that it is not possible to build a large project directly from the drawing board. For large projects engineers always build ***prototypes***; that is, scaled-down models that can be studied, tested, and sometimes even used for limited purposes. Models of bridges are built and tested in wind tunnels; pilot plants are constructed before attempting to use new technology on the assembly line.

software prototypes

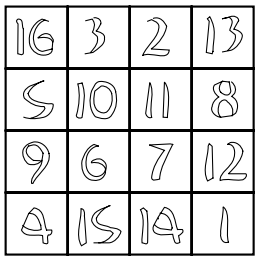
Prototyping is especially helpful for computer software, since it can ease the communication between users and designers early in a project, thereby reducing misunderstandings and helping to settle the design to everyone's satisfaction. In building a software prototype the designer can use programs that are already written for input-output, for sorting, or for other common requirements. The building blocks can be assembled with as little new programming as possible to make a working model that can do some of the intended tasks. Even though the prototype may not function efficiently or do everything that the final system will, it provides an excellent laboratory for the user and designer to experiment with alternative ideas for the final design.

Programming Precept

Always plan to build a prototype and throw it away.
You'll do so whether you plan to or not.

Programming Projects 1.6

P1. A ***magic square*** is a square array of integers such that the sum of every row, the sum of every column, and sum of each of the two diagonals are all equal. Two magic squares are shown in Figure 1.5.[6]



17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

sum = 34 sum = 65

Figure 1.5. Two magic squares

(a) Write a program that reads a square array of integers and determines whether or not it is a magic square.

(b) Write a program that generates a magic square by the following method. This method works only when the size of the square is an odd number. Start by placing 1 in the middle of the top row. Write down successive integers 2, 3, ... along a diagonal going upward and to the right. When you reach the top row (as you do immediately since 1 is in the top row), continue to the bottom row as though the bottom row were immediately above the top row. When you reach the rightmost column, continue to the leftmost column as though it were immediately to the right of the rightmost one. When you reach a position that is already occupied, instead drop straight down one position from the previous number to insert the new one. The 5×5 magic square constructed by this method is shown in Figure 1.5.

P2. ***One-Dimensional Life*** takes place on a straight line instead of a rectangular grid. Each cell has four neighboring positions: those at distance one or two from it on each side. The rules are similar to those of two-dimensional Life except (1) a dead cell with either two or three living neighbors will become alive in the next generation, and (2) a living cell dies if it has zero, one, or three living neighbors. (Hence a dead cell with zero, one, or four living neighbors stays dead; a living cell with two or four living neighbors stays alive.) The progress of sample communities is shown in Figure 1.6. Design, write, and test a program for one-dimensional Life.

[6] The magic square on the left appears as shown here in the etching *Melancolia* by ALBRECHT DÜRER. Note the inclusion of the date of the etching, 1514.

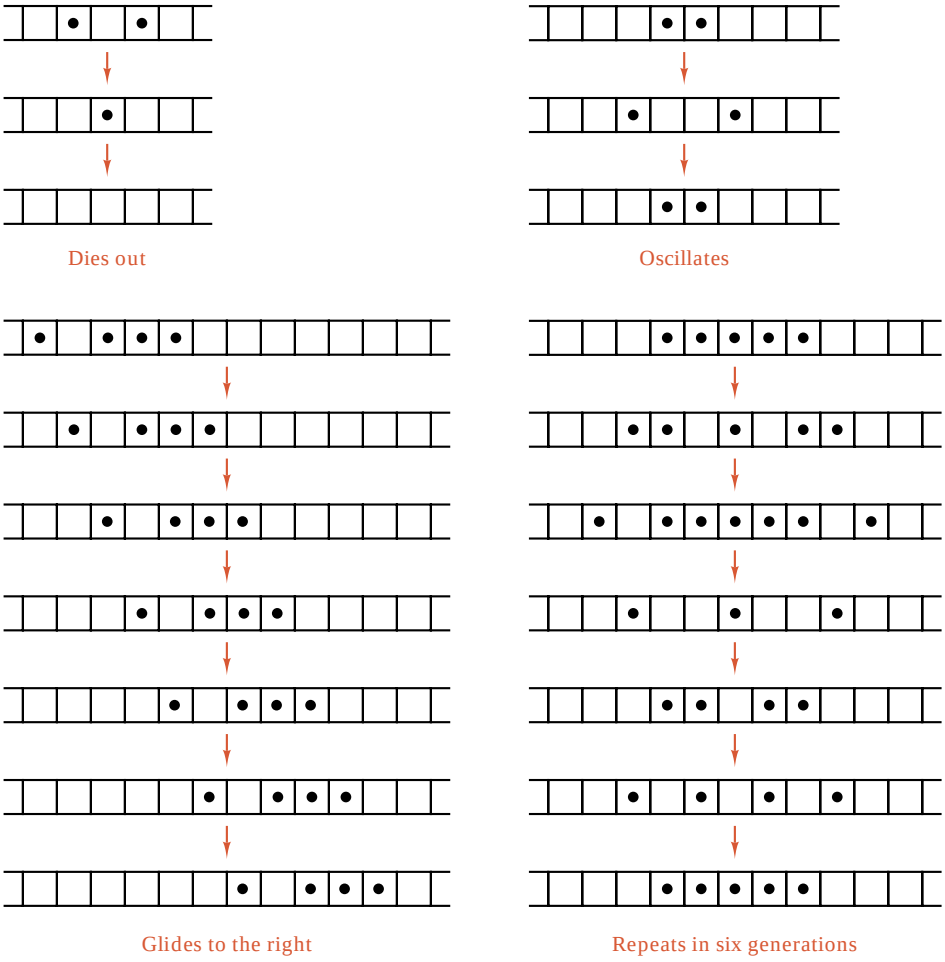


Figure 1.6. One-dimensional Life configurations

P3. **(a)** Write a program that will print the calendar of the current year.

(b) Modify the program so that it will read a year number and print the calendar for that year. A year is a leap year (that is, February has 29 instead of 28 days) if it is a multiple of 4, except that century years (multiples of 100) are leap years only when the year is divisible by 400. Hence the year 1900 is not a leap year, but the year 2000 is a leap year.

(c) Modify the program so that it will accept any date (day, month, year) and print the day of the week for that date.

(d) Modify the program so that it will read two dates and print the number of days from one to the other.

(e) Using the rules on leap years, show that the sequence of calendars repeats exactly every 400 years.

(f) What is the probability (over a 400-year period) that the 13th of a month is a Friday? Why is the 13th of the month more likely to be a Friday than any other day of the week? Write a program to calculate how many Friday the 13ths occur in this century.

POINTERS AND PITFALLS

1. To improve your program, review the logic. Don't optimize code based on a poor algorithm.
2. Never optimize a program until it is correct and working.
3. Don't optimize code unless it is absolutely necessary.
4. Keep your functions short; rarely should any function be more than a page long.
5. Be sure your algorithm is correct before starting to code.
6. Verify the intricate parts of your algorithm.
7. Keep your logic simple.
8. Be sure you understand your problem before you decide how to solve it.
9. Be sure you understand the algorithmic method before you start to program.
10. In case of difficulty, divide a problem into pieces and think of each part separately.
11. The nouns that arise in describing a problem suggest useful classes for its solution; the verbs suggest useful functions.
12. Include careful documentation (as presented in Section 1.3.2) with each function as you write it.
13. Be careful to write down precise preconditions and postconditions for every function.
14. Include error checking at the beginning of functions to check that the preconditions actually hold.
15. Every time a function is used, ask yourself why you know that its preconditions will be satisfied.
16. Use stubs and drivers, black-box and glass-box testing to simplify debugging.
17. Use plenty of scaffolding to help localize errors.
18. In programming with arrays, be wary of index values that are off by 1. Always use extreme-value testing to check programs that use arrays.
19. Keep your programs well formatted as you write them—it will make debugging much easier.

Contents
Index
Help

20. Keep your documentation consistent with your code, and when reading a program make sure that you debug the code and not just the comments.
21. Explain your program to somebody else: Doing so will help you understand it better yourself.

REVIEW QUESTIONS

IRM

Most chapters of this book conclude with a set of questions designed to help you review the main ideas of the chapter. These questions can all be answered directly from the discussion in the book; if you are unsure of any answer, refer to the appropriate section.

1.3

1. When is it appropriate to use one-letter variable names?

2. Name four kinds of information that should be included in program documentation.

3. What is the difference between *external* and *internal* documentation?

4. What are pre- and postconditions?

5. Name three kinds of parameters. How are they processed in C++?

6. Why should side effects of functions be avoided?

1.4

7. What is a program stub?

8. What is the difference between stubs and drivers, and when should each be used?

9. What is a structured walkthrough?

10. What is *scaffolding* in a program, and when is it used?

11. Name a way to practice *defensive* programming.

12. Give two methods for testing a program, and discuss when each should be used.

13. If you cannot immediately picture all details needed for solving a problem, what should you do with the problem?

14. What are preconditions and postconditions of a subprogram?

15. When should allocation of tasks among functions be made?

1.6

16. How long should coding be delayed?

17. What is *program maintenance*?

18. What is a *prototype*?

19. Name at least six phases of the software life cycle and state what each is.

20. Define software engineering.

21. What are requirements specifications for a program?

Contents
Index
Help

REFERENCES FOR FURTHER STUDY

C++

The programming language C++ was devised by BJARNE STROUSTRUP, who first published its description in 1984. The standard reference manual is

B. STROUSTRUP, *The C++ Programming Language*, third edition, Addison-Wesley, Reading, Mass., 1997.

Many good textbooks provide a more leisurely description of C++, too many books to list here. These textbooks also provide many examples and applications.

For programmers who already know the language, an interesting book about how to use C++ effectively is

SCOTT MEYERS, *Effective C++*, second edition, Addison-Wesley, Reading, Mass., 1997.

Programming Principles

Two books that contain many helpful hints on programming style and correctness, as well as examples of good and bad practices, are

BRIAN KERNIGHAN and P. J. PLAUGER, *The Elements of Programming Style*, second edition, McGraw-Hill, New York, 1978, 168 pages.

DENNIE VAN TASSEL, *Program Style, Design, Efficiency, Debugging, and Testing*, second edition, Prentice Hall, Englewood Cliffs, N.J., 1978, 323 pages.

EDSGER W. DIJKSTRA pioneered the movement known as structured programming, which insists on taking a carefully organized top-down approach to the design and writing of programs, when in March 1968 he caused some consternation by publishing a letter entitled "Go To Statement Considered Harmful" in the *Communications of the ACM* (vol. 11, pages 147–148). DIJKSTRA has since published several papers and books that are most instructive in programming method. One book of special interest is

EDSGER W. DIJKSTRA, *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, N.J., 1976, 217 pages.

A full treatment of object oriented design is provided by

GRADY BOOCH, *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, Redwood City, Calif., 1994.

The Game of Life

The prominent British mathematician J. H. CONWAY has made many original contributions to subjects as diverse as the theory of finite simple groups, logic, and combinatorics. He devised the game of Life by starting with previous technical studies of cellular automata and devising reproduction rules that would make it difficult for a configuration to grow without bound, but for which many configurations would go through interesting progressions. CONWAY, however, did not publish his observations, but communicated them to MARTIN GARDNER. The popularity of the game skyrocketed when it was discussed in

Contents
Index
Help

MARTIN GARDNER, "Mathematical Games" (regular column), *Scientific American* 223, no. 4 (October 1970), 120–123; 224, no. 2 (February 1971), 112–117.

The examples at the end of Sections 1.2 and 1.4 are taken from these columns. These columns have been reprinted with further results in

MARTIN GARDNER, *Wheels, Life and Other Mathematical Amusements*, W. H. Freeman, New York and San Francisco, 1983, pp. 214–257.

This book also contains a bibliography of articles on Life. A quarterly newsletter, entitled *Lifeline*, was even published for a few years to keep the real devotees up to date on current developments in Life and related topics.

Software Engineering

A thorough discussion of many aspects of structured programming is found in

EDWARD YOURDON, *Techniques of Program Structure and Design*, Prentice-Hall, Englewood Cliffs, N. J., 1975, 364 pages.

A perceptive discussion (in a book that is also enjoyable reading) of the many problems that arise in the construction of large software systems is provided in

FREDERICK P. BROOKS, JR., *The Mythical Man–Month: Essays on Software Engineering*, Addison-Wesley, Reading, Mass., 1975, 195 pages.

A good textbook on software engineering is

IAN SOMMERVILLE, *Software Engineering*, Addison-Wesley, Wokingham, England, 1985, 334 pages.

algorithm verification

Two books concerned with proving programs and with using assertions and invariants to develop algorithms are

DAVID GRIES, *The Science of Programming*, Springer-Verlag, New York, 1981, 366 pages.

SUAD ALAGIĆ and MICHAEL A. ARBIB, *The Design of Well-Structured and Correct Programs*, Springer-Verlag, New York, 1978, 292 pages.

Keeping programs so simple in design that they can be proved to be correct is not easy, but is very important. C. A. R. HOARE (who invented the quicksort algorithm that we shall study in Chapter 8) writes: "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult." This quotation is from the 1980 Turing Award Lecture: "The emperor's old clothes," *Communications of the ACM* 24 (1981), 75–83.

Two books concerned with methods of problem solving are

problem solving

GEORGE PÓLYA, *How to Solve It*, second edition, Doubleday, Garden City, N.Y., 1957, 253 pages.

WAYNE A. WICKELGREN, *How to Solve Problems*, W. H. Freeman, San Francisco, 1974, 262 pages.

The programming project on one-dimensional Life is taken from

JONATHAN K. MILLER, "One-dimensional Life," *Byte* 3 (December, 1978), 68–74.