Data Structures and Program Design in C++

NAVIGATING THE DISK

For information on using the Acrobat toolbar and other Acrobat commands, consult the Help document within Acrobat. See especially the section "Navigating Pages."

Material displayed in green enables jumps to other locations in the book, to transparency masters, and to run sample demonstration programs. These come in three varieties:

- The green menu boxes in the left margin of each page perform jumps to frequently used parts of the book:
- Green material in the text itself will jump to the place indicated. After taking such a jump, you may return by selecting the ◁◁ icon (go back) in the Acrobat toolbar.
- The transparency-projector icon ( ) brings up a transparency master on the current topic. Return by selecting the ◁◁ icon (go back) in the Acrobat toolbar.
- The Windows ( ) icon in the left margin select and run a demonstration program, which will operate only on the Windows platform.

This CD contains a folder `textprog` that contains the source code for all programs and program segments appearing in the book. These files cannot be compiled directly, but they can be copied and used for writing other programs.

HINTS FOR PAGE NAVIGATION

- Each chapter (or other major section) of the book is in a separate `pdf` file, so you may start Acrobat directly on a desired chapter.
- To find a particular section in the current chapter, hit the **Home** key, or select |◁ in the Acrobat toolbar or in the green menu bar, which will jump to the first page of the chapter where there is a table of contents for the chapter.
- After jumping to a new location in the book, you can easily return to your previous location by selecting ◁◁ (go back) in the Acrobat toolbar.
- To find a particular topic, select the index icon ( ) in the left margin.
- To find a particular word in the current chapter, use the binoculars icon in the Acrobat toolbar.
- The **PgDown** and **Enter** (or **Return**) keys advance one *screenful*, whereas ▷, ↓, →, and advance one *page*. Of these, only will move from the last page of one chapter to the first page of the next chapter.
- To move backwards, **PgUp** and **Shift+Enter** move up one *screenful*, whereas ◁, ↑, ←, and move back one *page*. Of these, only will move from the first page of one chapter to the last page of the previous chapter.

Data Structures and Program Design in C++

Robert L. Kruse

Alexander J. Ryba

CD-ROM prepared by

Paul A. Mailhot

Prentice Hall

Upper Saddle River, New Jersey 07458

Library of Congress Cataloging–in–Publication Data

KRUSE, ROBERT L.
Data structures and program design in C++ / Robert L. Kruse, Alexander J. Ryba.
p. cm.
Includes bibliographical references and index.
ISBN 0–13–087697–6
1. C++ (Computer program language) 2. Data Structures (Computer Science) I. Ryba, Alexander J. II. Title.
QA76.73.C153K79 1998 98–35979
005.13'3—dc21 CIP

Publisher: *Alan Apt*
Editor in Chief: *Marcia Horton*
Acquisitions Editor: *Laura Steele*
Production Editor: *Rose Kernan*
Managing Editor: *Eileen Clark*
Art Director: *Heather Scott*
Assistant to Art Director: *John Christiana*
Copy Editor: *Patricia Daly*

Cover Designer: *Heather Scott*
Manufacturing Buyer: *Pat Brown*
Assistant Vice President of Production and Manufacturing: *David W. Riccardi*
Editorial Assistant: *Kate Kaibni*
Interior Design: *Robert L. Kruse*
Page Layout: *Ginnie Masterson (PreTEX, Inc.)*
Art Production: *Blake MacLean (PreTEX, Inc.)*

Cover art: *Orange*, 1923, by Wassily Kandinsky (1866-1944), Lithograph in Colors. Source: Christie's Images

The typesetting for this book was done with PreTEX, a preprocessor and macro package for the TEX typesetting system and the POSTSCRIPT page-description language. PreTEX is a trademark of PreTEX, Inc.; TEX is a trademark of the American Mathematical Society; POSTSCRIPT is a registered trademarks of Adobe Systems, Inc.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-087697-6

Prentice-Hall International (U.K.) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Contents

Preface

THE APPRENTICE CARPENTER may want only a hammer and a saw, but a master builder employs many precision tools. Computer programming likewise requires sophisticated tools to cope with the complexity of real applications, and only practice with these tools will build skill in their use. This book treats structured problem solving, object-oriented programming, data abstraction, and the comparative analysis of algorithms as fundamental tools of program design. Several case studies of substantial size are worked out in detail, to show how all the tools are used together to build complete programs.

Many of the algorithms and data structures we study possess an intrinsic elegance, a simplicity that cloaks the range and power of their applicability. Before long the student discovers that vast improvements can be made over the naïve methods usually used in introductory courses. Yet this elegance of method is tempered with uncertainty. The student soon finds that it can be far from obvious which of several approaches will prove best in particular applications. Hence comes an early opportunity to introduce truly difficult problems of both intrinsic interest and practical importance and to exhibit the applicability of mathematical methods to algorithm verification and analysis.

Many students find difficulty in translating abstract ideas into practice. This book, therefore, takes special care in the formulation of ideas into algorithms and in the refinement of algorithms into concrete programs that can be applied to practical problems. The process of data specification and abstraction, similarly, comes before the selection of data structures and their implementations.

We believe in progressing from the concrete to the abstract, in the careful development of motivating examples, followed by the presentation of ideas in a more general form. At an early stage of their careers most students need reinforcement from seeing the immediate application of the ideas that they study, and they require the practice of writing and running programs to illustrate each important concept that they learn. This book therefore contains many sample programs, both short

functions and complete programs of substantial length. The exercises and programming projects, moreover, constitute an indispensable part of the book. Many of these are immediate applications of the topic under study, often requesting that programs be written and run, so that algorithms may be tested and compared. Some are larger projects, and a few are suitable for use by a small group of students working together.

Our programs are written in the popular object-oriented language C++. We take the view that many object-oriented techniques provide natural implementations for basic principles of data-structure design. In this way, C++ allows us to construct safe, efficient, and simple implementations of data-structures. We recognize that C++ is sufficiently complex that students will need to use the experience of a data structures courses to develop and refine their understanding of the language. We strive to support this development by carefully introducing and explaining various object-oriented features of C++ as we progress through the book. Thus, we begin Chapter 1 assuming that the reader is comfortable with the elementary parts of C++ (essentially, with the C subset), and gradually we add in such object-oriented elements of C++ as classes, methods, constructors, inheritance, dynamic memory management, destructors, copy constructors, overloaded functions and operations, templates, virtual functions, and the STL. Of course, our primary focus is on the data structures themselves, and therefore students with relatively little familiarity with C++ will need to supplement this text with a C++ programming text.

SYNOPSIS

Programming Principles

By working through the first large project (CONWAY's game of Life), Chapter 1 expounds principles of object-oriented program design, top-down refinement, review, and testing, principles that the student will see demonstrated and is expected to follow throughout the sequel. At the same time, this project provides an opportunity for the student to review the syntax of elementary features of C++, the programming language used throughout the book.

Introduction to Stacks

Chapter 2 introduces the first data structure we study, the stack. The chapter applies stacks to the development of programs for reversing input, for modelling a desk calculator, and for checking the nesting of brackets. We begin by utilizing the STL stack implementation, and later develop and use our own stack implementation. A major goal of Chapter 2 is to bring the student to appreciate the ideas behind information hiding, encapsulation and data abstraction and to apply methods of top-down design to data as well as to algorithms. The chapter closes with an introduction to abstract data types.

Queues

Queues are the central topic of Chapter 3. The chapter expounds several different implementations of the abstract data type and develops a large application program showing the relative advantages of different implementations. In this chapter we introduce the important object-oriented technique of inheritance.

Linked Stacks and Queues

Chapter 4 presents linked implementations of stacks and queues. The chapter begins with a thorough introduction to pointers and dynamic memory management in C++. After exhibiting a simple linked stack implementation, we discuss

destructors, copy constructors, and overloaded assignment operators, all of which are needed in the safe C++ implementation of linked structures.

Recursion

Chapter 5 continues to elucidate stacks by studying their relationship to problem solving and programming with recursion. These ideas are reinforced by exploring several substantial applications of recursion, including backtracking and tree-structured programs. This chapter can, if desired, be studied earlier in a course than its placement in the book, at any time after the completion of Chapter 2.

Lists and Strings

More general lists with their linked and contiguous implementations provide the theme for Chapter 6. The chapter also includes an encapsulated string implementation, an introduction to C++ templates, and an introduction to algorithm analysis in a very informal way.

Searching

Sorting

Chapter 7, Chapter 8, and Chapter 9 present algorithms for searching, sorting, and table access (including hashing), respectively. These chapters illustrate the interplay between algorithms and the associated abstract data types, data structures, and implementations. The text introduces the "big-O" and related notations for elementary algorithm analysis and highlights the crucial choices to be made regarding best use of space, time, and programming effort. These choices require that we find analytical methods to assess algorithms, and producing such analyses is a battle for which combinatorial mathematics must provide the arsenal. At an elementary level we can expect students neither to be well armed nor to possess the mathematical maturity needed to hone their skills to perfection. Our goal, therefore, is to help students recognize the importance of such skills in anticipation of later chances to study mathematics.

Tables and Information Retrieval

Binary Trees

Binary trees are surely among the most elegant and useful of data structures. Their study, which occupies Chapter 10, ties together concepts from lists, searching, and sorting. As recursively defined data structures, binary trees afford an excellent opportunity for the student to become comfortable with recursion applied both to data structures and algorithms. The chapter begins with elementary topics and progresses as far as such advanced topics as splay trees and amortized algorithm analysis.

Multiway Trees

Chapter 11 continues the study of more sophisticated data structures, including tries, B-trees, and red-black trees.

Graphs

Chapter 12 introduces graphs as more general structures useful for problem solving, and introduces some of the classical algorithms for shortest paths and minimal spanning trees in graphs.

Case Study: The Polish Notation

The case study in Chapter 13 examines the Polish notation in considerable detail, exploring the interplay of recursion, trees, and stacks as vehicles for problem solving and algorithm development. Some of the questions addressed can serve as an informal introduction to compiler design. As usual, the algorithms are fully developed within a functioning C++ program. This program accepts as input an expression in ordinary (infix) form, translates the expression into postfix form, and evaluates the expression for specified values of the variable(s). Chapter 13 may be studied anytime after the completion of Section 10.1.

The appendices discuss several topics that are not properly part of the book's subject but that are often missing from the student's preparation.

Mathematical Methods

Appendix A presents several topics from discrete mathematics. Its final two sections, Fibonacci numbers amd Catalan numbers, are more advanced and not

needed for any vital purpose in the text, but are included to encourage combinatorial interest in the more mathematically inclined.

Random Numbers

Appendix B discusses pseudorandom numbers, generators, and applications, a topic that many students find interesting, but which often does not fit anywhere in the curriculum.

Packages and Utility Functions

Appendix C catalogues the various utility and data-structure packages that are developed and used many times throughout this book. Appendix C discusses declaration and definition files, translation units, the utility package used throughout the book, and a package for calculating CPU times.

Programming Precepts, Pointers, and Pitfalls

Appendix D, finally, collects all the Programming Precepts and all the Pointers and Pitfalls scattered through the book and organizes them by subject for convenience of reference.

COURSE STRUCTURE

prerequisite

The prerequisite for this book is a first course in programming, with experience using the elementary features of C++. However, since we are careful to introduce sophisticated C++ techniques only gradually, we believe that, used in conjunction with a supplementary C++ textbook and extra instruction and emphasis on C++ language issues, this text provides a data structures course in C++ that remains suitable even for students whose programming background is in another language such as C, Pascal, or Java.

A good knowledge of high school mathematics will suffice for almost all the algorithm analyses, but further (perhaps concurrent) preparation in discrete mathematics will prove valuable. Appendix A reviews all required mathematics.

content

This book is intended for courses such as the ACM Course CS2 (*Program Design and Implementation*), ACM Course CS7 (*Data Structures and Algorithm Analysis*), or a course combining these. Thorough coverage is given to most of the ACM/IEEE knowledge units[1] on data structures and algorithms. These include:

AL1 Basic data structures, such as arrays, tables, stacks, queues, trees, and graphs;

AL2 Abstract data types;

AL3 Recursion and recursive algorithms;

AL4 Complexity analysis using the big Oh notation;

AL6 Sorting and searching; and

AL8 Practical problem-solving strategies, with large case studies.

The three most advanced knowledge units, AL5 (complexity classes, NP-complete problems), AL7 (computability and undecidability), and AL9 (parallel and distributed algorithms) are not treated in this book.

[1] See *Computing Curricula 1991: Report of the ACM/IEEE-CS Joint Curriculum Task Force*, ACM Press, New York, 1990.

Most chapters of this book are structured so that the core topics are presented first, followed by examples, applications, and larger case studies. Hence, if time allows only a brief study of a topic, it is possible, with no loss of continuity, to move rapidly from chapter to chapter covering only the core topics. When time permits, however, both students and instructor will enjoy the occasional excursion into the supplementary topics and worked-out projects.

two-term course

A two-term course can cover nearly the entire book, thereby attaining a satisfying integration of many topics from the areas of problem solving, data structures, program development, and algorithm analysis. Students need time and practice to understand general methods. By combining the studies of data abstraction, data structures, and algorithms with their implementations in projects of realistic size, an integrated course can build a solid foundation on which, later, more theoretical courses can be built. Even if it is not covered in its entirety, this book will provide enough depth to enable interested students to continue using it as a reference in later work. It is important in any case to assign major programming projects and to allow adequate time for their completion.

SUPPLEMENTARY MATERIALS

A CD-ROM version of this book is anticipated that, in addition to the entire contents of the book, will include:

- All packages, programs, and other C++ code segments from the text, in a form ready to incorporate as needed into other programs;
- Executable versions (for DOS or Windows) of several demonstration programs and nearly all programming projects from the text;
- Brief outlines or summaries of each section of the text, suitable for use as a study guide.

These materials will also be available from the publisher's internet site. To reach these files with `ftp`, log in as user `anonymous` to the site `ftp.prenhall.com` and change to the directory

```
pub/esm/computer_science.s-041/kruse/cpp
```

Instructors teaching from this book may obtain, at no charge, an instructor's version on CD-ROM which, in addition to all the foregoing materials, includes:

- Brief teaching notes on each chapter;
- Full solutions to nearly all exercises in the textbook;
- Full source code to nearly all programming projects in the textbook;
- Transparency masters.

BOOK PRODUCTION

This book and its supplements were written and produced with software called PreTEX, a preprocessor and macro package for the TEX typesetting system.[2] PreTEX, by exploiting context dependency, automatically supplies much of the typesetting markup required by TEX. PreTEX also supplies several tools that greatly simplify some aspects of an author's work. These tools include a powerful cross-reference system, simplified typesetting of mathematics and computer-program listings, and automatic generation of the index and table of contents, while allowing the processing of the book in conveniently small files at every stage. Solutions, placed with exercises and projects, are automatically removed from the text and placed in a separate document.

For a book such as this, PreTEX's treatment of computer programs is its most important feature. Computer programs are not included with the main body of the text; instead, they are placed in separate, secondary files, along with any desired explanatory text, and with any desired typesetting markup in place. By placing tags at appropriate places in the secondary files, PreTEX can extract arbitrary parts of a secondary file, in any desired order, for typesetting with the text. Another utility removes all the tags, text, and markup, producing as its output a program ready to be compiled. The same input file thus automatically produces both typeset program listings and compiled program code. In this way, the reader gains increased confidence in the accuracy of the computer program listings appearing in the text. In fact, with just two exceptions, all of the programs developed in this book have been compiled and succesfully tested under the g++ and Borland C++ compilers (versions 2.7.2.1 and 5.0, respectively). The two exceptions are the first program in Chapter 2 (which requires a compiler with a full ANSI C++ standard library) and the last program of Chapter 13 (which requires a compiler with certain Borland graphics routines).

ACKNOWLEDGMENTS

Over the years, the Pascal and C antecedents of this book have benefitted greatly from the contributions of many people: family, friends, colleagues, and students, some of whom are noted in the previous books. Many other people, while studying these books or their translations into various languages, have kindly forwarded their comments and suggestions, all of which have helped to make this a better book.

We are happy to acknowledge the suggestions of the following reviewers, who have helped in many ways to improve the presentation in this book: KEITH VANDER LINDEN (Calvin College), JENS GREGOR (University of Tennessee), VICTOR BERRY (Boston University), JEFFERY LEON (University of Illinois at Chicago), SUSAN

[2] TEX was developed by DONALD E. KNUTH, who has also made many important research contributions to data structures and algorithms. (See the entries under his name in the index.)

HUTT (University of Missouri–Columbia), FRED HARRIS (University of Nevada), ZHI-LI ZHANG (University of Minnesota), and ANDREW SUNG (New Mexico Institute of Technology).

ALEX RYBA especially acknowledges the helpful suggestions and encouraging advice he has received over the years from WIM RUITENBURG and JOHN SIMMS of Marquette University, as well as comments from former students RICK VOGEL and JUN WANG.

It is a special pleasure for ROBERT KRUSE to acknowledge the continuing advice and help of PAUL MAILHOT of PreTEX, Inc., who was from the first an outstanding student, then worked as a dependable research assistant, and who has now become a valued colleague making substantial contributions in software development for book production, in project management, in problem solving for the publisher, the printer, and the authors, and in providing advice and encouragement in all aspects of this work. The CD-ROM versions of this book, with all their hypertext features (such as extensive cross-reference links and execution of demonstration programs from the text), are entirely his accomplishment.

Without the continuing enthusiastic support, faithful encouragement, and patience of the editorial staff of Prentice Hall, especially ALAN APT, Publisher, LAURA STEELE, Acquisitions Editor, and MARCIA HORTON, Editor in Chief, this project would never have been started and certainly could never have been brought to completion. Their help, as well as that of the production staff named on the copyright page, has been invaluable.

ROBERT L. KRUSE

ALEXANDER J. RYBA

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

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.

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

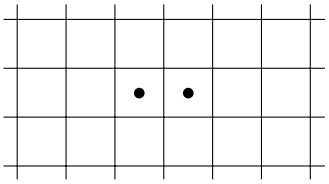
1. The neighbors of a given cell are the eight cells that touch it vertically, horizontally, or diagonally.
2. If a cell is alive but either has no neighboring cells alive or only one alive, then in the next generation the cell dies of loneliness.
3. If a cell is alive and has four or more neighboring 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

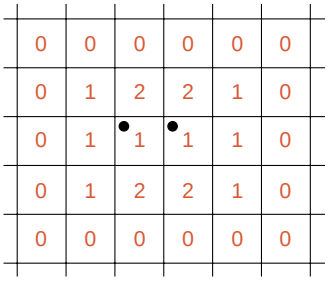
A particular arrangement of living and dead cells in a grid is called a ***configuration***. The preceding rules explain how one configuration changes to another at each generation.

1.2.2 Examples

As a first example, consider the configuration



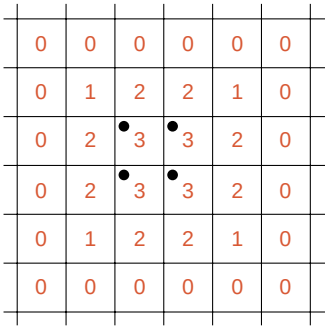
The counts of living neighbors for the cells are as follows:



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.

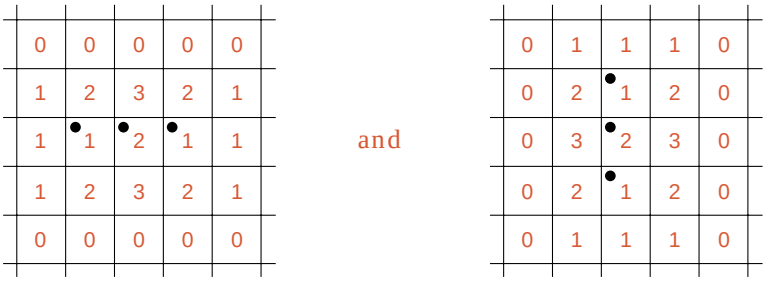
On the other hand, the configuration



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.

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

methods

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.

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.