Introduction to Programming in Turing

First Edition - Second Printing

J. N. P. Hume

Holt Software Associates, Inc.
Toronto, Canada

Publisher:
HOLT SOFTWARE ASSOCIATES INC.
203 College St., Suite 305
Toronto, Ontario, Canada M5T 1P9
(416) 978-6476 1-800-361-8324
http://www.holtsoft.com

ISBN: 0-921598-42-4

First Edition - Second Printing

Printed in Canada by the University of Toronto Press

Table of Contents

Preface

This textbook, *Introduction to Programming in Turing*, is intended to be used in a first computer science course. It emphasizes the basic concepts of programming and sets programming within the larger context of computer science and computer science within the context of problem solving and design.

Introduction to Programming in Turing focuses on computing concepts with the ultimate goal of facilitating the broadest possible coverage of the core computer science curriculum.

The programming language used in this book is Turing (OOT), which has an easy-to-learn syntax and is supported by student-friendly programming environments. By learning the fundamentals of programming using a language that does not intrude with a difficult syntax, students can make a relatively easy transition to any one of the currently popular object-oriented languages such as C++ or Java.

Overview

This book covers the material in standard curricula for first courses in computer science.

The list of chapter titles outlines the arrangement of materials.

1. Computing Essentials
2. The Turing Environment
3. Program Design and Style
4. Simple Programs
5. Variables and Constants
6. Repetition
7. Character Graphics
8. Pixel Graphics
9. Selection
10. Storing Data on the Disk

11. Handling Strings
12. Processing Text
13. Program Structure
14. Arrays and Other Data Types
15. Music
16. Subprograms
17. Subprograms with Array Parameters
18. Records and Files
19. Advanced Topics
20. Advanced Pixel Graphics
21. Animation and GUIs

Chapter 1 provides an overview of the history of modern computing, including hardware, software, programming languages, and number systems. By highlighting a number of key technological developments, it attempts to place Computer Engineering today in its scientific and social context. It also explores some of the current issues in computing such as employment, privacy, and access to information.

Chapter 2 describes how to use the **Turing** environment. The chapter also covers how syntax errors are reported and how they are corrected.

Chapter 3 introduces some of the key ideas in computer science. It explains the process of programming and provides suggestions for good programming style. It also introduces important design concepts such as the software lifecycle.

Chapter 4 introduces programs that consist of a single line. Arithmetic operations for both integers and real numbers are illustrated. Labels are shown to improve the understandability of results that are output. Programs of several lines are used to output a series of lines.

Chapter 5 shows how declarations are used to prepare locations in memory for storing information. Strings and numbers are input to set values of variables. The use of constants and comments to improve the program understandability is illustrated.

Prompts are also introduced as a way of informing users what is expected of them.

Chapter 6 introduces the repetition control structure used in structured programming. Both counted and conditional loops are discussed. The exit condition is explained in terms of providing a means for exiting from any point within the loop. Random integers are generated to control loop termination.

Chapter 7 shows how characters can be positioned in the Execution window to create various patterns. Manipulation of character colors and background colors is also demonstrated. The use of drawing and erasing a pattern to achieve the illusion of animation is also introduced.

Chapter 8 explains how to create graphics by plotting in pixel positions in the Execution window. The instructions for plotting points, lines, ovals, and arcs to allow the creation of simple line graphics is illustrated. Graphics with colored backgrounds, closed curves filled with color and sound are also explored.

Chapter 9 introduces the selection construct. Two-way, three-way, and multi-way selection are investigated. The **case** construct is presented as an alternative to the **if**..**then**..**elsif**..**else** structure for programming multi-way selection.

Chapter 10 shows how to redirect a program's output to a disk file rather than or in addition to displaying it in the Execution window. Using the disk file to provide input to a program is also explained and using the end-of-file automatically placed at the end of any disk file is introduced. Reading of lines of text from the disk is shown as an alternative to token-oriented input of string data.

Chapter 11 outlines methods to: find the length of a strong, join strings together, locate a substring, search for a pattern in a string, replace one pattern with another, and eliminate characters or patterns from strings. A method of preventing improper input to a program from causing a program crash is shown.

Chapter 12 summarizes various ways of inputting text material from a file by token, line, or number of characters. Ways of handling multiple files directly in a program rather than by

redirection are shown. Text formatting is described and a simple example of language translation is programmed.

Chapter 13 presents flow charts for the three basic control structures used in structured programming: linear program flow, repetition, and selection. Charts for nesting of structures and multi-way selection are also shown. The scope of declarations of variables and constants is discussed. Top-down program design is outlined.

Chapter 14 defines the declaration of the array type. The use of arrays is discussed with examples for sorting lists. The datatypes: subrange, boolean, and named are given with examples of their use. Two-dimensional arrays are illustrated by tables.

Chapter 15 describes how to play musical notes, both individually and as a series. Arranging the music to accompany graphic animation in the Execution window is illustrated.

Chapter 16 introduces the two types of Turing subprograms: functions that produce a value and procedures that produce some action. Both predefined and user-created examples of these two types of subprograms are presented. The notion of parameters, both value and variable, are explained. Type transfer functions and recursive subprograms are introduced.

Chapter 17 shows how both functions and procedures can have parameters that are of array type. The correspondence between formal and actual parameters is presented and the idea of dynamic formal parameters for arrays introduced. Local and global variables are differentiated.

Chapter 18 describes the declaration, input, and output for the data type record. Arrays and files of records are presented and the movement of records in memory is explained. Text and binary files are contrasted and the way that random access to records on disk is obtained is described. An example of maintaining a binary file of records is presented.

Chapter 19 introduces binary search as an efficient method of searching in a sorted file. The method of sorting by merging is detailed . Files stored as linked lists of records are described and an example of a highly interactive graphic is presented.

Chapter 20 examines programs for pixel graphic examples that require the use of more complex mathematics or subprograms with array parameters. Animation using a buffer is shown. Methods of displaying statistical information in a graphical format such as a pie or bar chart are given. Mathematical functions are plotted.

Chapter 21 shows programs involving animation requiring interaction with the mouse. Two advanced animation techniques are described. These include use of the Pic and Sprite modules available in the OOT library of predefined modules. Another module which facilitates the creation of graphical user interface (GUI) elements such as input boxes and buttons, as well as sound with animation, is introduced.

Flexibility

Introduction to Programming in Turing has been organized to provide an introduction to the fundamental concepts of computer science.

Differing course demands and student populations may require instructors to omit certain chapters or parts of chapters, or to insert additional material to cover some concepts in greater detail. For example, instructors wishing to address the historical information in Chapter 1 after students have more hands-on programming may choose to begin with a later chapter.

The Turing Programming Language

This book uses the language Turing. One of the main advantages of using Turing for instruction is that its simple syntax allows instructors to cover more computer science concepts in the limited teaching hours available in a course. This is possible because significantly less time needs to be devoted to teaching language details.

The Turing language was first developed in 1982 in response to the need for a programming notation that incorporates good structuring methods, supports a correctness approach, is easy for the student to learn, and is suited to interactive programming.

This language has gained considerable acceptance as a teaching medium.

The Turing language has undergone extensive enhancement, in terms of both software support and language notation, since its inception. In 1992 a superset of Turing, called Object Oriented Turing (OOT) was created by adding a number of features including objects, classes, and inheritance.

The programming environment for Turing is particularly helpful for the learning student. It provides good diagnostic messages for syntax errors and run-time errors. It provides strong run-time checking, with automatic detection of uninitialized variables and dangling pointers. It supports an on-line manual (lookup of language features by a button press), a directory browser, and, in the advanced mode, multi-window editing. Versions of the environment allow the student to use a wide variety of hardware platforms and operating systems.

Once programming concepts have been learned in Turing, it is relatively easy to transfer these ideas to new situations. In particular, data structuring, algorithms, and object-orientation learned using OOT transfer directly to industrial systems such as C++ and Java.

Comments

Your comments, corrections, and suggestions are very welcome. Please feel free to contact us at:

Distribution Manager
Holt Software Associates Inc.
203 College Street, Suite 305
Toronto, Ontario, Canada M5T 1P9
E-mail: books@hsa.on.ca
Phone: (416) 978-6476
USA or Canada Toll-Free: 1-800-361-8324
World Wide Web: http://www.holtsoft.com

Acknowledgments

As the author of *Introduction to Programming in Turing* it gives me great pleasure to acknowledge all the help we have had in its preparation. Chris Stephenson served as our curriculum expert and edited the text. Tom West provided excellent technical assistance. Paola Barillaro, Susan Heffernan, and Graham Smyth provided valuable suggestions for additions to the text. The text was entered with great care by Inge Weber. Harriet Hume produced a first-class index to the book. Catharine Kozuch did our final page check. The cover was designed by Brenda Kosky Communications.

It was my good fortune again to work with such a cooperative and competent team.

J.N.P. Hume
University of Toronto

Chapter 1

Computing Essentials

1.1 Introduction

Computers are now part of our everyday lives. We do our banking by computer. When we buy something at the grocery store, the clerk runs the bar code over an optical scanner which then inputs the price from a computer database. We also use computers for recreation, for example, we play games that use simulation and computer animation.

Despite the fact that most of us use computers every day, many people still do not know exactly what is inside the computer, what its various parts do, and how they all work together.

This chapter begins with a brief history of computers and then describes the parts of the computer and how they function. Although this chapter discusses the history of computing and the person most-commonly associated with each new development, it is important to remember that what appears to be a great leap forward is often the result of many tiny steps, both forward and backward, taken not by one, but by many people. Like most scientific endeavors, advances in computing are usually the result of work by many people over long periods of time.

1.2 A Brief History of Computer Hardware

The history of computing goes back to the earliest days of recorded civilization and humankind's desire to find ways to calculate more accurately and reason more systematically. The Greeks, for example, helped to systematize reasoning and developed axiomatic mathematics and formal logic. The Babylonians and Egyptians contributed enormously to computational science by developing multiplication tables, tables for squares and roots, exponential tables, and the formula for quadratic equations.

In the late sixteenth and early seventeenth centuries a number of major discoveries contributed to our ability to perform complex mathematical operations. These included the development of Algebra (using letters for unknowns), logarithms, the slide rule, and analytic geometry.

The development of mechanical machines to assist with calculations began in 1623 when Wilhelm Schickard designed and built what is believed to be the first digital calculator. This machine did addition and subtraction mechanically and partially automated multiplication and division. About twenty years later Blaise Pascal developed a gear-based machine that was capable of performing addition and subtraction. Gottfried Wilhelm Leibniz (who invented calculus along with Sir Issac Newton) invented a device called the **Leibniz Wheel** that did addition, subtraction, multiplication, and division automatically.

The nineteenth century saw major advances in computation. Charles Babbage (who was a founding member of the Royal Astronomical Society in Britain) developed the concepts for two steam powered machines, which he called the **Difference Engine** and the **Analytic Engine**. The **Difference Engine** could perform mathematical computations to eight decimal places.

Although it was never actually built (and would have been the size of a railway locomotive if it had been) the **Analytic Engine** was truly an ancestor of the modern computer. It was to be a machine capable of performing mathematical operations from instructions on a series of punched cards. It had a memory unit, a sequential control, and many other features of a modern computer.

In 1854 Swedish printer Pehr George Scheutz succeeded in building a **Difference Engine** for calculating mathematical tables. Scheutz's machine consisted of a memory (which he called a ÒstoreÓ) and a central processor (which he called a ÒmillÓ). It operated by a series of punched cards that contained a series of operations and data. His design was based on the **Jacquard loom** which used punched cards to encode weaving patterns.

What might be considered the first computer company was established in 1896 by Herman Hollerith. Hollerith was a

mechanical engineer who was helping to tabulate the census of the population of the United States. In 1890 he invented a new punched card technology which proved much faster than traditional manual methods of tabulating the results and this allowed the *United States Census Bureau* to gather more information by asking more questions. Hollerith's company, the *Tabulating Machines Company*, later became *International Business Machine*s**,** better known now as *IBM*.

In 1937 a British mathematician named Alan M. Turing made a significant contribution to computing with the development of an abstract machine called the **Turing machine**. During World War II, Turing was involved in top secret work for the British military and was responsible for breaking the German code, thus providing the allied forces with important information about Germany's war plans.

In the late 1930s and early 1940s *Bell Telephone Laboratories* began building more powerful machines for scientific calculations. In 1944, Howard T. Aiken, an electromechanical engineer, collaborated with a number of *IBM* engineers to design and build the **Mark I** computer. This machine could handle negative and positive numbers, carry out long calculations in their natural sequence, use a variety of mathematical functions, and was fully automatic. Rather than punched cards, the instructions for the **Mark I** were punched on a paper tape.

In 1945 John W. Mauchly and J. Presper Eckert Jr. designed and built the first large-scale electronic digital computer from 18,000 vacuum tubes and 1500 relays. This machine was called the **ENIAC** (electronic numerical integrator and calculator). **ENIAC** was a thousand times faster than the earlier electromechanical machines. It weighed 30 tons and took up 1500 square feet of floor space.

John von Neumann, who also worked on the **ENIAC** project, played a major role in the development of a machine that would improve on **ENIAC** in a number of ways. The new machine, called **EDVAC** (electronic discrete variable and calculator), was capable of storing a program in memory and breaking computations down into a sequence of steps that could be

performed one at a time. **EDVAC** used **binary notation** to store and manipulate numbers whereas earlier machines used decimal arithmetic. (See section 1.4 for a more complete explanation of binary.)

In 1951 Mauchly and Eckert developed the **UNIVAC I** (universal automatic computer) for the *Remington-Rand Corporation*. **UNIVAC I** was considered by many to be the first commercially viable electronic digital computer.

In 1952 the University of Toronto purchased a computer from *Ferranti Electric* and called it **FERUT** (Ferranti University of Toronto). This machine, which replaced the University's original calculating punch equipment, used a punched paper tape system based on the teletype machine. It was actually a copy of the **Mark I** developed at the University of Manchester by Alan M. Turing. It is important to note the large number of women who programmed for **FERUT**. These included Audrey Bates, Charlotte Frose, Jean McDonald, Jean Tucker, and Beatrice Worsley.

In 1953 *IBM* entered the commercial market with the **IBM 701**. This was followed a year later by the **IBM 650**, which was a decimal machine designed as a logical upgrade to punched-card machines.

The next major breakthrough in computer hardware came in the late 1950s and early 1960s during which the expensive and often unreliable **vacuum tubes** were replaced with **transistors**. The transistor allowed for the design of computers that were more reliable and more powerful.

Between 1965 and 1970 the introduction of **integrated circuits**, each of which contained many transistors, made the previous generation of computers virtually obsolete. The wires connecting the transistors were printed as copper on a sheet of insulating material. The transistors were plugged into the insulating sheet, making contact with the wires as required. In modern computers, the integrated circuit is achieved by a process of printing metal onto a tiny chip made of silicon which acts as an insulator. Adding certain other elements to the silicon creates a transistor. Multiple transistors are be combined to create circuits.

Between 1970 and 1980 the move to **large-scale integration** (LSI) and **very large-scale integration** (VLSI) continued this miniaturization trend. High-speed **semi-conductor** technology has also led to significant improvements in memory storage.

The late 1970s saw the development of personal computers, the first computers small enough and affordable enough for the average person to buy. The development of the **Altair 8800**, the **Apple II**, and the **IBM PC** marked the first steps in what has now become a world-wide mega-industry.

In the 1980s **reduced instruction-set computers** (RISC), parallel computers, neural networks, and optical storage were introduced.

Generation	Year	Hardware Development
1	1951-58	vacuum tubes card or paper tape input/output magnetic drum memory
2	1959-64	integrated circuits (printed) magnetic tape I/O magnetic core memory
3	1965-70	integrated circuits (photographic) minicomputers magnetic disk I/O
4	1970-80	LSI VLSI virtualstorage microcomputers
5	1980+	RISC parallel computers optical storage laser disk

1.3 A Brief History of Programming

The person believed to be the very first computer programmer was Ada Byron King, countess Lovelace, the daughter of the English romantic poet Lord Byron. As a woman in the early 1800s, Ada Byron King was largely self-taught in mathematics. This was both remarkable and necessary since at that time women were not allowed to even enter a library, let alone attend university. During her studies, Ada Byron King corresponded with many of the important English mathematicians of her day and eventually came to work with Charles Babbage. While Babbage concentrated on designing the computer hardware, Ada Byron King became the first person to develop a set of instructions, which came to be called a **computer program**, for Babbage's **Analytic Engine**.

As computer hardware evolved from those early prototypes or models to today's fast and powerful machines, the ways in which information and instructions were prepared for the computer also radically changed. Over time the emphasis shifted from making the instructions simple enough for the computer to understand to making them close enough to the spoken language for everyone else to understand.

Computers such as ENIAC and EDVAC stored information, both numbers and program instructions, as groups of binary digits; 0 and 1. Each instruction was often made up of two parts, a group of bits (binary digits) representing the **operation** to be performed, and a group representing the **operand**, that is, the machine **address** of data to be operated on. A program consisted of a series of such instructions written in this **machine code**.

The problem with machine language is that it required programmers to write very long series of numbered instructions and to remember the binary codes for the different commands in the machine's instruction set. They also had to keep track of the storage locations (addresses) of the data and the instructions. As a result, a single program often took months to write and was commonly full of hard-to-find errors.

A major improvement in programming languages began in the 1950s with the development of symbolic machine languages, called **assembly languages.** Assembly languages allowed the programmer to write instructions using letter symbols rather than binary operation codes. The **assembler** then translated these simple written instructions into machine language. A program written in assembly language is called a **source program**. After the source program has been converted into machine code by an assembler, it is called an **object program**.

Assembly languages have a number of advantages when compared to machine language. They are:

- easier and faster to write,
- easier to **debug** (to find and correct errors), and
- easier to change at a later date.

But assembly languages also have a major drawback. Because the communication is taking place at one step up from the machine, an assembly language is designed for a specific make and model of computer processor. This means a program written to run on one computer will not work on another. Assembly languages are therefore said to be **machine oriented**.

Symbolic addressing was another interesting programming language development. Symbolic addressing involves the use of symbols to represent the assignment of storage addresses to data. Programmers could thus create symbolic names to represent items of data, and these names could be used throughout a program. Programmers no longer had to assign actual machine addresses to symbolic data items. The processor automatically assigned storage locations when the program ran.

The development of **high-level languages** was the next important step in computing. Like a program written in assembly language, a program written in a high-level language still needed to be translated into machine code. High-level languages included their own translation software to perform this task. The translating program is called a **compiler**. Compilers often generated many machine instructions for each source code statement. Today, however, many personal computers use an

interpreter, rather than a compiler. A compiler translates the source program into object code and then executes it; the interpreter converts each source program statement into machine language every time the statement is executed. It does not save the object code.

High-level languages provide a number of benefits. They:

- free programmers from concerns about low-level machine details such as memory addressing and machine dependency,
- can be run on different makes of computers,
- are easier to use than assembly languages, and
- are easier for programmers to learn.

Here is an example of a simple instruction which shows the high-level language, assembly language and binary versions.

High Level Language (Turing)

```
a := 5

put a
```

8086 Assembly language program

```
mov  word ptr _a,0005H          [_a=824H]       c7 06 24 08
05 00

push  _a                                        ff 36 24 08

call  printf_               [printf=421CH]  e8 1c 42

add  p,0004H                                83 c4 04
```

Binary program

```
11000111 00000110 00100100 00001000 00000101 00000000 11111111
00110110

00100100 00001000 11101000 00011100 01000010 10000011 11000100
00000100
```

Until the late 1950s computers were still being used primarily for mathematical and scientific calculation. Computers used binary arithmetic and it was not until machines were developed to use decimal arithmetic that they were practical for business calculations. In 1958, J. N. Patterson Hume and C.C. Gotlieb from the University of Toronto published the first book on using the computer for business, called *High-Speed Data Processing* (McGraw Hill, 1958). Hume and Gotlieb are credited with publishing many of the computer terms still used today, including: compiler, data processing, and keyboard. In many ways Hume and Gotlieb pioneered today's wide-spread use of computers in business.

The major problem with programming in the early days was that it was time-consuming and difficult work, and programs often did not work. The growing recognition of the potential use of computers in many fields led to the evolution of modern programming languages that allowed for programs that were easier to develop, use, and modify over time.

In 1954 John Backus began leading an *IBM*-sponsored committee to develop a new scientific-mathematical programming language. In 1957 this committee introduced a new high-level language called **FORTRAN** (Formula Translator) for the **IBM 704** computer. FORTRAN gained wide acceptance, especially among scientists and statisticians because of its usefulness for expressing mathematical equations and over the years many versions of FORTRAN have been developed.

As more businesses began to appreciate the potential for computers, new languages were developed to meet their particular needs. In 1961, Grace Murray Hopper helped to invent **COBOL** (Common Business-Oriented Language). COBOL was designed to process business data and for many years it was used heavily by the insurance and banking industries among

others. The language **ALGOL** was also developed by an international committee for scientific use.

At the same time John McCarthy at the Massachusetts Institute of Technology developed **LISP** was designed to support research in the field of **artificial intelligence** (AI).

BASIC (Beginners All-purpose Symbolic Instruction Code) was developed by John Kemeny and Thomas Kurtz beginning in 1964 at Dartmouth College. Kemeny and Kurtz wanted to develop a language that undergraduate students could easily learn and use on a time-shared basis on a large computer. Although the original BASIC had a well-defined syntax (the grammar rules of a programming language), over the years many different non-compatible versions have been developed, so that today, there are many ÒdialectsÓ of BASIC.

LOGO was developed in the 1960s by Seymour Papert at the Massachusetts Institute of Technology. Papert's goal was to help young children explore a mathematical environment by using an on-screen ÒturtleÓ to draw figures and create simple animations.

In 1970, **Pascal** (named after the seventeenth-century French mathematician Blaise Pascal) was developed by Niklaus Wirth at the *Federal Institute of Technology* in Switzerland. One of the major benefits of Pascal over BASIC, was that it was designed to support the concepts of **structured programming**. In the case of earlier languages such as BASIC, a program's structure could be quite complex and trying to follow the sequence of execution (how a program runs) was like trying to untangle a plate of spaghetti.

At the heart of structured programming is the idea that within a program there are groups of statements that are used to control the flow of information. These groups, often referred to as **control constructs**, are:

- **linear sequence**: where statements are executed one after the other in the order in which they are written,
- **repetition**: where a group of statements is to be executed repeatedly, and

- **selection**: where one group of statements is selected for execution from a number of alternatives.

Within a structured program, these control constructs are organized so that the program can be read from top to bottom. Each structure allows only one entrance and one exit. Indenting the instructions in a program printout helps someone looking at a program to understand its logic more easily.

The term structured programming now relates not just to program control logic, but to how programs are designed and developed. Today when people speak of structured programming, they are often referring to a systematic way of analyzing computer problems and designing solutions called **top-down programming**. In top-down programming, a problem is broken down into a series of smaller problems, each of which is then solved. Once the smaller problems are solved, the solutions are combined to solve the larger problem.

In the early 1970s Dennis Ritchie and Brian Kernighan of *Bell Labs* developed a programming language they called **C**. This language allowed for easy access to the hardware and was very efficient. C was designed to program large systems and so it became a favorite programming language of systems programmers. It was used to produce the **UNIX** operating system.

In 1975 the United States Department of Defense began a series of studies intended to support the development of a new programming language to be used by computer vendors and military programmers. The language, called **Ada** (after Ada Byron King, countess Lovelace), was released by *CII-Honeywell-Bull* in 1979 and was used extensively by the United States military.

Year	Person	Developed
1800s	Ada Byron	first computer programs
1954	John Backus	FORTRAN

1960s	Grace Murray Hopper	COBOL
	John Kemeny Thomas Kurtz	BASIC
	Seymour Papert	LOGO
1970s	Niklaus Wirth	Pascal
	Dennis Ritchie	Ada
1980s	Brian Kernighan Dennis Ritchie	C
	Ric Holt James Cordy	Turing
1990s	Bjarne Stroustrup	C++
	James Gosling	Java

Turing (named after Alan M. Turing) was developed by Ric Holt and James Cordy at the University of Toronto in 1984. Like Pascal before it, Turing was designed to suit personal computers, where the instructions are typed in directly. Turing was more powerful than Pascal and was designed to have a mathematically specified syntax (preventing the development of numerous incompatible versions) and that its syntax was much easier to understand.

1.3.1 A New Way of Organizing Large Programs

As computer programs and the tasks they perform have become larger and more complex, programmers, scientists, and academics have worked to find new ways of thinking about, developing, and maintaining these programs over time. The term **programming paradigm** refers to the set of ideas which forms the basis of a particular way of programming.

The move from the unstructured to structured programming paradigm (see previous section) was a significant improvement in

how programs were written and developed. By following the rules associated with structured programming, programmers made the logic of their programs easier to follow. This meant that it was easier to locate and fix errors and to make changes to those programs as needed.

But the reality is that even structured programs are often time-consuming to write, hard to understand, and inflexible in that they are designed for a specific task.

Object-oriented programming is a way of designing and writing programs based on the concept that a program can be created as a collection of **objects** which work together. Each of these objects is a set of data and methods that operate on this set of data. The details of how the data is stored and how the methods work are hidden from the user.

An object is created from a class from which many such objects can be created. One class can **inherit** (or borrow) features from another class. Classes which perform specific tasks can also be reused in other programs, thus speeding up program development. Current object-oriented programming languages include **C++** (developed by Bjarne Stroustrup), **Java** (developed by James Gosling), and **Object Oriented Turing** (developed by Ric Holt).

1.4 What is a Computer?

Now that we have looked at how computers have evolved, it makes sense to look more closely at what exactly is meant by the term computer. A computer is a machine that takes in information, transforms it, and outputs it in a different form. Information used by the computer is called **data**. Data can take many forms. It can be text, numbers, graphics, and even sounds. More specifically, information put into the computer is called **input**. Information that comes out of a computer is called **output**.

While computers themselves can be very complex, there is a very simple model that represents how all computers work. There

must be something to take in information, something to process it, something to store it, and something to output the result. Here is a simple model of a computer.

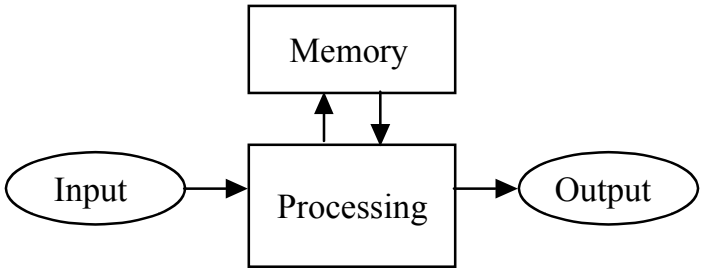


Figure 1.1 Simple Model of a Computer

The part of the computer performing the input function receives information in the form of data or programs. Common examples of **input devices** include keyboards, disk drives, scanners, and the mouse.

One of the things that can be very confusing is that, while many kinds of devices (microwave ovens, bathroom scales, toasters, even singing birthday cards) use computer technology such as computer chips and memory, they are not really general purpose computers because they can only be used to perform the task for which they have been built. They cannot be programmed to perform a variety of tasks. Computers, however, are defined by their ability to process information in a variety of ways.

1.4.1 The Central Processing Unit

The internal part of the computer that processes information is called the **Central Processing Unit** or **CPU**. The CPU does the work and determines the order in which operations are performed. There are three main sections of the CPU:

- the **primary storage** section,
- the **arithmetic-logic** section, and

- the **control** section.

The primary storage section or **main memory** is used four ways:

- the **input storage area** collects data that has been fed in until it is ready to be processed,
- the **program storage** area holds the processing instructions,
- the **working storage space** holds the data being processed and the intermediate results of that processing, and
- the **output storage area** holds the final results of the processing operations until they can be released.

The arithmetic-logic section is where all calculations are performed and decisions are made; in other words, where the data held in the storage areas is processed. Data moves from primary storage to the arithmetic logic unit and back again to storage repeatedly until processing is completed. Once the processing is completed, the final results are sent to the output storage section and from there to an **output device** such as the **computer screen** (also called a monitor) or the **printer**.

The control section maintains the order of the entire system by directing the data to the appropriate places. It is the traffic cop of the CPU. When processing begins, the control section retrieves the first instruction from the program storage area. The control section then interprets the instruction and signals the other sections to begin their tasks. Program instructions are selected and carried out in sequence unless an instruction is encountered to tell it to jump to another instruction or until the processing is completed.

1.4.2 Memory

Over the years the ways in which computers store information have changed remarkably. Initially information was stored using vacuum tubes. Vacuum tubes, which resemble light bulbs, are glass cylinders containing several filaments. A later generation of computers stored information using **magnetic core memory**, which consisted of donut shaped rings that were magnetized in

one direction or another. Today, main memory is usually on **silicon** chips (see Section 1.5.2) and the time required to access information in main memory (**access time**) is measured in nanoseconds (billionths of a second) instead of the microseconds (millionths of a second) magnetic core memory required.

Main memory is broken down into two main types, **RAM** (Random Access Memory) and **ROM** (Read Only Memory). RAM stores programs, data, or information from input and output devices. This stored information can be manipulated by computer programs or input from or output to devices such as printers and scanners. RAM memory is lost when the computer is shut off. Saving RAM memory in secondary storage is essential if this information is required later. Creating a **back-up** (saving a second copy of the same information) is a good programming practice.

Memory	Conversion
1 bit	either 1 or 0
1 nybble	4 bits
1 byte	8 bits or 2 nybbles
1 kilobyte	1000 bytes (approx.)
1 megabyte	1000 kilobytes
1 gigabyte	1000 megabytes
1 terabyte	1000 gigabytes

ROM on the other hand stores information that the system needs in order to function. The contents of this memory are recorded at the time the computer is manufactured and are not lost when the computer is shut off, if there is an interruption in power, or if an application stops working properly (referred to as ÒcrashingÓ).

In early personal computers, the overall amount of memory was very limited. For example, when the **Commodore Pets** were

introduced the total RAM was 8K and many people could not imagine needing any more memory than this. Today, however, personal computers contain far more RAM and sizes are now measured in gigabytes (millions of kilobytes).

Information that needs to be stored for future use can be kept in **secondary storage devices** such as floppy disks, hard drives, and Zip[a] drives.

1.4.3 Output Devices

Once data is input and processed it often needs to be output in various forms. When the computer is used for numerical computation, for example, the numbers are input, then processed, and the result is output. Output devices can include monitors (the computer screen), printers, and synthesizers. Over time the technology underlying computer monitors has changed dramatically, and these changes have often been driven by the need for higher resolution especially for computer graphics and smaller, more portable computers. Different types of display technologies include CRTs (cathode ray tubes), LCDs (liquid crystal), and gas plasma displays.

1.5 Number Systems: Decimal and Binary

At its most basic, a computer is a machine that processes information, but before you can understand more explicitly how it does that, you have to know something about how that information is expressed at the machine level. As mentioned earlier in this chapter, machine language is based on the binary number system rather than the decimal number system. We will now look more closely at both of these systems and how they differ.

When human beings count, they use a number system which is based upon combinations of ten different digits. These are:

0 1 2 3 4 5 6 7 8 9

All numbers are combinations of these symbols. This number system is called the **decimal number system** or **base 10** since it is based on a set of ten digits. At the machine level, however, computers use only combinations of two different digits to represent all information that can be stored. These are:

0 1

This two-digit system is called the **binary number system**. It is also called **base 2**. When people need to communicate with computers at their most fundamental (machine) level, they need to be able to convert between binary and decimal representation.

An important aspect of each number system is that it has **place value**. The base 10 number (the decimal number) 987 has 7 in the units column, 8 in the tens column, and 9 in the hundreds column. Moving from the right column to the left, each column is multiplied by 10.

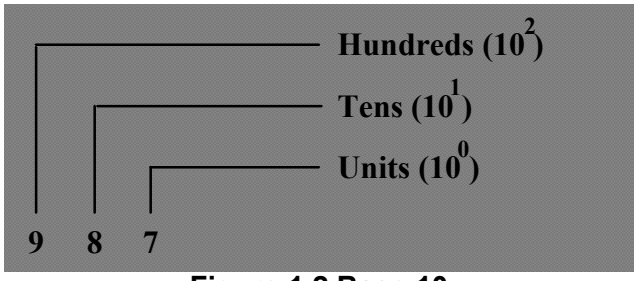


Figure 1.2 Base 10

Therefore the base 10 number 987 equals

$$9 \times 10^2 + 8 \times 10^1 + 7 \times 10^0$$

In mathematics 10^0 equals 1. The figures in superscript (smaller raised figures) represent the number of times the base number is multiplied by itself.

The binary number system also has place value. The binary number (base 2) 1101 has the right-most 1 in the units column, the 0 to its left is in the twos column, the 1 in the 2 squared (or fours column), and the left-most 1 in the 2 cubed column (eights column). Each column is multiplied by 2, moving from the right to the left.

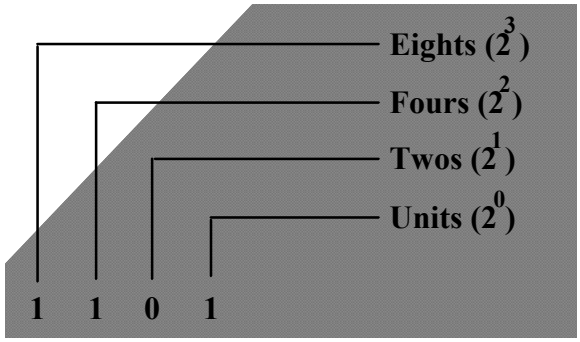


Figure 1.3 Base 2

For example, the binary number 1101 equals

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \text{ or } 8 + 4 + 0 + 1 = 13$$

therefore

$$1101_2 = 13_{10}$$

In mathematics 2^0 equals 1. The figures in **subscript** (smaller lowered figures) represent the number system being used.

All base 10 numbers have an equivalent binary number. Understanding the decimal and binary number systems is essential in order to understand how computers work.

Here is a chart showing the decimal and binary version of the first sixteen natural numbers in both systems.

Decimal or Base 10	Binary or Base 2
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111

Decimal or Base 10	Binary or Base 2
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

1.6 Hardware and Networks

Once you understand what computers are and how they store information it is important to look more closely at the actual components or parts that make up a computer system.

When you hear people talk about computers they often talk about **hardware** and **software**. By hardware, they mean the electronic and mechanical parts of the computer. Computer hardware can include things such as:

- the **hard disk**, which stores programs and data,
- the **keyboard**, **CD ROM, a**nd the **mouse** which are used for inputting data,
- **monitors**, **printers**, and **speakers** which are output devices,
- l**ight pens** and **barcode readers** which are input devices, and
- **disk drives** and **modems** which can be either input or output devices.

The term **peripheral** refers to the many devices which can be attached to the computer. These peripherals often include

printers, scanners, speakers, modems, and joysticks. Their primary purpose is to either provide input to, or output from the computer. Printers, for example, provide textual output from a file while barcode readers are used to scan information such as a price or part number into the computer. In later chapters we will investigate a variety of peripherals in more detail.

Figure 1.4 shows what a standard hardware setup might look like.

Figure 1.4 Standard Computer Hardware Setup

This setup includes input devices such as the mouse, keyboard, and CD-ROM and output devices such as speakers and the printer. The disk drive and modem are unique because they can be used for both input and output. The disk drive can be used to store and retrieve files from floppy disks which can be transferred from computer to computer, while modems send and retrieve information via telephone or cable lines.

Figure 1.5 provides a simplified illustration of some of the internal components of the computer.

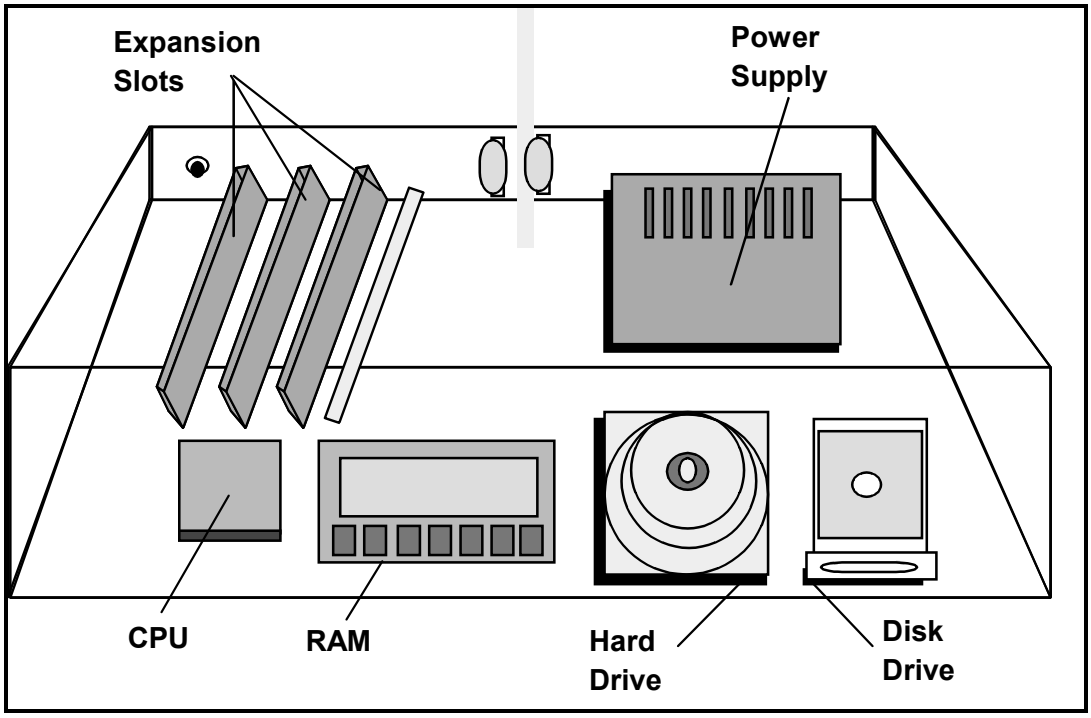


Figure 1.5 Computer Components

Many computers are also connected to **networks** which allow the computers (and their users!) to communicate and share information and resources. Networking facilitates two-way sharing of software and hardware locally or globally. Computer networks can take many forms, depending upon the type and number of computers in the network and the distance over which they must be linked. The hardware on the network may include: personal computers, mainframes, supercomputers, printers, fax machines, navigational control systems, and interactive entertainment centers. The software, however, will always include application software, desktop software, and networking operating systems.

The benefits of networking computers are enormous. Geographically remote areas can be connected to share information. Without actually transferring an entire file to all the people involved, several people can simultaneously share large

files. Also, within a networked environment the information generated by a single user can be shared world-wide instantaneously.

For example, a large company with a number of sales offices in different cities can use its networked computers to maintain a single file with all of their customer information. The sales people at each office would be able to get information from this file and add information to it without having to have a separate copy of the entire file on their computers. In this way, every sales person would have access to the most up-to-date information for the entire company. The problem of keeping the separate lists in each office current would be eliminated. And finally, fewer computer resources would be needed because there would be one shared file rather than many separate files. In this way, networking enables faster, more precise communication which should translate into greater accuracy, productivity, and cost savings.

Networking also allows different types of computers to communicate. Users choose specific computers and operating systems for many reasons. For example, a particular application might be better suited for a Macintosh than an IBM PC. Networking allows users to share resources even when their systems are different.

Users on a network can also share physical resources. Many individual computers can share one scanner, printer, or other expensive piece of hardware. Sharing hardware significantly reduces the expense of running a system.

1.6.1 Different Kinds of Computers for Different Needs

The use of computers to perform an ever-increasing number of jobs has led to the development of various kinds and sizes of computers. The largest computers are called **supercomputers**. These computers are used primarily by government, industry, and research organizations for projects which require huge amounts of computing power, often for fast calculation purposes.

Supercomputers have multiple CPUs and are known for their processing speed.

Many large businesses such as banks, airlines, insurance companies, and law enforcement agencies use **mainframe computers** to store and manipulate large amounts of information. Mainframe computers are capable of multiprocessing, that is, running many processes simultaneously. Their main advantage is the ability to store vast amounts of information and later to distribute it.

Mainframe computers are often used in a network of medium-sized computers called **minicomputers**. A minicomputer, despite its name, is a medium-sized computer. The most common computer, however, is the **personal computer**. It is also know as the **microcomputer** or **desktop computer**. This computer is designed for a single user and consists of a CPU, input/output units, and a power supply.

As more and more people are using their computers at work, at home, and on the road, **laptop** computers have become increasingly popular. These computers have been designed to provide most of the same features as a desktop or personal computer, but in a more portable size. These computers are often characterized by a smaller keyboard and a flip-top lid containing a monitor or screen.

One of the newest computers sold commercially is the **hand-held** or **palmtop computer**. These extremely small computers are often used as personal organizers, word processors, or for accessing local or global telecommunications such as **electronic mail**, commonly called **email**.

As computer power of even the smallest computers has increased, the distinctions between these kinds of computers have begun to blur, and it is important to note that the size of the computer is no longer an indication of how powerful or how fast it is.

1.6.2 The Silicon Chip

Computers today, regardless of what they are used for, are smaller, faster, and less expensive than they were twenty years ago. One of the main reasons for this is the development of the **silicon chip**. The importance of this technology to the computing industry can be seen in the fact the area in California where there is a high concentration of computing companies is often referred to as ÒSilicon ValleyÓ.

Silicon is found in common substances such as sand, but has unique properties which make it especially useful in the manufacturing of electronics equipment. While silicon alone acts as an insulator, when a line of metal is deposited on it, that metal acts as a wire **conductor** which transmits electric current. If the silicon is impregnated with certain other elements, it acts as a **semi-conductor**. These semi-conductors are used to create **transistors**. A transistor is an electronic switch that conducts electric current under certain conditions.

A series of switches and conducting wires that perform a specific function is called a **circuit** because it controls the path over which electricity can flow. A silicon chip is made up of a large number (potentially millions) of transistors which are in turn organized into circuits. Silicon chips are often called integrated circuits because each chip can contain many different circuits.

The silicon chip has also increased the speed at which computers can perform tasks, resulting in computers that perform an operation in a trillionth of a second, performing millions of calculations, or processing tens of thousands of instructions in a second.

Because silicon chips are extremely small, they can be used in many products. A silicon chip less than a centimeter (an eighth of an inch) square can store millions of bits of information and perform the work it once would have taken a room full of computers to perform. Many common products such as watches now contain silicon chips. Specialized chips are also used in many larger products such as cars, VCRs, microwave ovens, and washing machines.

Every computer from a supercomputer to a laptop computer contains silicon chips. Because modern digital computers tend to

isolate the user from the actual silicon chips that control the computer's functions, each containing huge numbers of elements, very few users actually understand the underlying logic of computer chips.

1.7 Software

The term software refers to all of the instructions that make the hardware work. Within the computer there are many different levels of software. In this section we will briefly examine four distinct kinds of software:

- operating systems,
- programming environments, and
- applications.

1.7.1 Operating Systems

The **operating system** is a collection of programs that the computer uses to manage itself and use its resources efficiently. Operating systems you might have already heard of include: UNIX, Linux, Microsoft Windows[a], DOS[a], and MacOS[a].

The operating system performs tasks such as accepting input from the keyboard and managing input and output between the computer and external devices (peripherals). Some of these devices such as printers are not part of the computer but can be used to send information to or receive information from the computer.

1.7.2 Programming Environments

In order to write your own program you need to use a **programming environment**. A programming environment is a set of tools that includes an **editor** for entering and changing the

program, a compiler or interpreter for translating programs into a machine language that the computer can understand, and sometimes a **debugger** for locating and fixing errors.

As noted earlier in this chapter, many different programming languages have been developed, each with its own set of features. Many of these programming languages now include their own programming environments.

1.7.3 Applications

Software can refer to a single computer program that does one task or a collection of programs, called an **application**, that perform many of the same kinds of tasks. Some common examples of large applications include:

- computer games,
- word processors,
- graphics packages,
- virtual reality software,
- presentation software,
- web browsers,
- database management systems, and
- spreadsheets.

Computer games are among the most popular software applications and are available from the simplest tic-tac-toe games to highly complex role playing games with advanced graphics, animation, and sound.

Word processing software enables the user to create, edit, and save textual documents. Almost all word processing software provides features for editing (cut and paste) and formatting (font styles and font sizes, tab bars, and headers and footers) text. Most also include an additional set of tools such as a spell checker, grammar checker, word counter, and a thesaurus.

Graphics applications provide many of the same basic functions as word processors, except that they are designed for

creating and manipulating images rather than text. There is a wide range of graphics packages, from simple paint programs that allow the user to create simple shapes in different colors to advanced tools that allow images to be created, modified, and displayed.

Virtual reality packages are programs that use computer-generated images, text, and sound to imitate real-world events. Simulation software creates a virtual environment which the user can observe and interact with. The aerospace industry, for example, uses flight simulators to test new aircraft designs before the product is released. Using simulation software to ÓvirtuallyÓ fly the aircraft helps the designers identify and correct potential problems before the plane is built. This saves time, money, and possibly even lives. The nuclear power industry also uses simulation software to demonstrate the operations of a nuclear power plant and simulate what might happen in the event of natural disasters (such as earthquakes). In this way, simulation software can be used to test safety and emergency procedures without requiring a real emergency. Simulation software is used in many fields, such as microcomputer design, global climatic studies, agricultural research, and black-hole research.

The term **presentation software** refers to an application that helps the user organize information so that it can be used for presentations or display. These applications often provide a set of templates or example formats into which the user can insert the information she or he wishes to display. Imported text can then be formatted in a variety of different styles. Most presentation packages allow the user to import graphical images, sound, and video. If the user has access to the appropriate display hardware and software, she or he can create a computer-based presentation run from either the keyboard or a hand-held device. If no such equipment is available, the presentation can also be printed directly onto acetates (clear plastic) to be displayed using an overhead projector.

Web browser applications allow the user to access, view, and **download** text and graphics from the **World Wide Web**. The **Internet** is a vast and complex network of computers which provides storage space for textual and graphical information

which can be shared. The term World Wide Web refers to specific kinds of files or collections of files where information is organized and displayed in **hypertext**. Hypertext is a non-linear organizational format. Rather than proceeding in a straight line from beginning to end like a book, it organizes the information in layers. In order to access the Web, users must be able to log into a computer network linked to the Internet, in other words they need an Internet account. Many large organizations such as corporations and universities provide accounts to their employees. Individuals with a home computer can also purchase an account (often on a monthly basis) from an **Internet Service Provider** (ISP) or through a telephone company or cable television provider.

Database applications help users store, organize, and use numbers, text and other forms of data. Databases tend to be used for more text-rich storage such as customer information.

S**preadsheet** applications also help users store, organize, and use numbers and text spread out in rows and columns. Spreadsheets usually provide features for manipulating numbers and performing calculations and so are used for financial plans, such as budgets.

1.8 The Social Impact of Computers

The impact of computers on our society is profound. Many books have been written on the social aspects of technology. In this section we will briefly examine some of the positive and negative impacts of computing.

The evolution of computer hardware and software has created a revolution in our society and can in some ways be seen as the latest extension of the Industrial Revolution. During the Industrial Revolution increasing mechanization led to the growth of industry. Since industry tended to concentrate in urban (city) rather than rural (country) areas, more and more people began moving into cities to work in factories and businesses. This has also been the

case with the computer revolution. Computers are now part of our everyday life and so in some subtle and not so subtle ways they affect how we work, play, and live.

1.8.1 Employment

One of the primary benefits of computing is that it has created thousands of new job opportunities. There is almost no industry now that does not require the skills of programmers. At the same time, the computerization of many traditional manufacturing industries, such as the auto industry, has led to the displacement of many workers. The growing use of computers has also eliminated many of the jobs formerly performed by unskilled workers or skilled tradespeople.

Increased computerization has allowed businesses, industries, and governments to streamline their operations, increase productivity, and provide services more efficiently. These improvements include easier access to goods and services such as Internet shopping, more reliable billing procedures, and faster and more detailed access to customer information. People in every profession can share information more easily. For example, scientists in different parts of the world can share their findings through email. They can also turn over the boring jobs, such as performing time-consuming calculations, to computers, thus increasing job satisfaction and improving their opportunity to find and share solutions.

At the same time, however, many people feel that computers are taking the human touch out of everything by making all our transactions so impersonal. Now, instead of dealing with people such as tellers in banks and service people at gas stations, we are dealing with machines. Instead of being people, we are personal identification numbers (pin), social insurance numbers (SIN), and account numbers. After a while this lack of human contact becomes extremely frustrating. The mayor of the city of Toronto, for example, banned voice mail from city offices because he said it was making the people of the city feel there were no real people to help them.

1.8.2 Privacy

Computer telecommunications such as email have opened the windows of communication world-wide. Through email, students in Canada and students in Australia can share information about themselves and their communities, and thus come to understand each other better. Many people who may be isolated by geography or disability can find others to communicate with via a chat room or listserv. For example, deaf people from around the world use a listserv called DEAF-L to discuss issues of concern to their community and to share information about available resources.

While computer technology has made it easier for us to gather information, it has also led to invasion of people's privacy. Information about every aspect of a person's life can be collected and sold to businesses and organizations without the person agreeing or even knowing. For example, grocery stores which use optical scanners to calculate the cost of food items and let customers use bank cards to pay can collect information about what kinds of products people buy and provide that information to the people who make or advertise those products. There is also growing concern about the availability of personal information such as medical and financial records.

1.8.3 Access to Information

The Internet is a tool people can use to gain access to many different sources of information on any topic imaginable. One problem, however, is that some people take everything they find on the Internet at face value. They do not consider that the people who are making that information available might be ill-informed or trying to mislead.

There is a saying that Òon the net, everyone is an expertÓ. What this means is that while everyone may seem to be an expert, some people simply do not know what they are talking about. When viewing material on the Internet, it is important to determine the validity of the information. Readers should consider

whether the person posting the information is likely to have the required knowledge and whether he or she might benefit from presenting false or misleading information. This does not mean that you should always trust someone of high social status, such as a university professor, while mistrusting a student. It simply means that you should always be careful to evaluate information based on its source.

There is also a debate about just how much of Òthe wrong kindÓ of information is available on the Internet. As the media have frequently reported, there is pornographic material available to those who go looking for it. For this reason many schools use some kind of blocking software on their networks to prevent students from gaining access to such information. Many companies also have very strict policies about firing employees who are caught using their office computers to download or store pornographic material.

Again, it is up to the user to access information in a responsible way. You should also keep in mind that no matter how clever you are, it is always possible for someone to track your use of a computer. Virtually every activity on a network is logged somewhere in the system and it is a relatively simple task for the network administrator to find out where you have been and what you have been doing.

1.8.4 Leisure Activities

For many people, computers have opened up a whole new realm of leisure activities. People can use their computers to create new computer programs, to play games, to use computer applications to master new skills, or to chat with people anywhere in the world. They can also take on-line courses. Unfortunately, like many other fascinating hobbies, computers can take over people's lives if they fall victim to computer addiction. On many university campuses, computer addiction is a major cause of failure, as students become so immersed in their hobby that they neglect their studies. Among many adults, home computers have also been known to put stress on their important relationships. Time on the computer just seems to fly by and somehow ÒI'll be there in ten minutesÓ turns into hours.

The most obvious sign of computer addiction is that addicts neglect other important aspects of their life such as family, work, friends, school, and sleep. The key to avoiding addiction is to ensure that, despite temporary indulgences, time spent at the computer is not time stolen from other important activities and people.

1.8.5 Illegal Activity

Like most human inventions, computers can be used in good ways and in bad ways. As computer use is increasing, so is the use of computers to commit criminal offenses. Computers are now being used to create illegal credit cards, to defraud banks, and to steal information from businesses. In this way, they have given rise to a new breed of white-collar criminals.

A new area of computer crime, called **software piracy**, involves people duplicating, using, and sharing, information (textual, graphical, and sound) and applications to which they have no right. Many people do not realize that it costs literally hundreds of thousands of dollars to create a software application. When people copy that application and share it with their friends,

they rob the people who spent the time and money to develop and distribute it.

In order to try to protect themselves from software piracy, many software companies attempt to protect their products with registration codes. While such measures can prove annoying to users, they are often the only way the company can protect itself from unauthorized duplication and distribution.

Another way that software companies are fighting back is through a process called a **software audit**. If a software vendor believes that a company, organization, or institution is using its software illegally, it can demand that the company determine every single software product it uses and prove that it is using it legally (has purchased the right to use it).

Many people do not realize that it is illegal to use copyrighted material without the permission of the copyright holder. People who post images, video, or sound on their web pages without finding out who they really belong to run the risk of being charged with **copyright infringement**.

1.8.6 Computers: Good or Bad?

Like most objects, computers themselves are neither good nor bad, but they can be used by good or bad people to do good or bad things. As responsible human beings we all have an obligation to look carefully at how we use technology. As a caring society we need to question whether the way in which we use our tools makes the world a better place for all of us, or just a more profitable place for some and a worse place for most.

Like most technologies, computers have the potential to improve life for everyone who owns one. At the same time, they have the potential to leave those who do not have access to these resources further and further behind.

1.9 Exercises

1. Describe the contribution to the development of modern computers made by the following people.
   (a) Charles Babbage
   (b) Pehr George Scheutz
   (c) Herman Hollerith
   (d) Alan M. Turing
   (e) Howard T. Aiken
   (f) John W. Mauchly and J. Presper Eckert Jr.
   (g) John von Neumann
   (h) Charlotte Frose
   (i) Ada Byron King
   (j) Grace Hopper
2. List the five generations of computers along with two significant features of each generation.
3. Define the following:
   (a) computer programming
   (b) programming language
   (c) syntax
   (d) debug
4. Explain two differences between machine code and assembly language.
5. List three advantages of assembly languages over machine code.
6. Differentiate between a compiler and an interpreter.
7. List five high-level programming languages.
8. State one feature of each of the five high-level languages identified in the Exercise 7 and list the most important person(s) involved with each language.

9. Machine code and assembly languages are both examples of Òlow-level languages.Ó List five advantages high-level languages have over low-level languages.
10. Define programming paradigm.
11. Explain the main differences between two programming paradigms.
12. Draw a simple model of a computer.
13. What does the acronym CPU stand for?
14. State the main function of the CPU.
15. State the three main sections of the CPU.
16. Explain the functions of each of the three sections of the CPU.
17. Explain the significance of each of the following terms relating to computer memory.
    (a) main memory
    (b) RAM
    (c) ROM
    (d) vacuum tubes
    (e) magnetic core
    (f) silicon chip
    (g) back-up
18. The binary number system is fundamental to really understanding how a computer functions. Explain.
19. Explain the difference between hardware and software.
20. List one significant feature (besides size) of each type of computer.
    (a) supercomputer
    (b) mainframe
    (c) minicomputer
    (d) personal computer

(e) laptop
(f) palmtop

21. What is a silicon chip?
22. List three features of silicon chips.
23. List three types of software and one feature of each type.
24. Computers can be used to obtain personal information without your knowledge. Explain.
25. Computers have created many new job opportunities. Explain.
26. Computers have eliminated many jobs. Explain.

1.10 Technical Terms

access time
Ada
algorithm
Analytic Engine
application software
artificial intelligence
arithmetic logic section
assembly language
back-up
base 2
base 10
BASIC
batch processing
binary notation
binary number system
binary representation
bit
C++
central processing unit (CPU)
classes
circuit
control constructs
COBOL
coding
control
compiler
computer game
computer programming
computer program
computer screen
conductor
control construct
copyright infringement

data
database
decimal number system
debug
debugger
Difference Engine
disk drive
download
editor
electronic mail (email)
execute
external documentation
FORTRAN
graphics package
hacker
hand-held/palmtop computer
hard disk
hardware
high-level language
hypertext
information technology
inherit
input
input device
input storage area
insulator
integrated circuit
interactive computing
internal documentation
Internet
interpreter
Internet Service Provider
invalid data
Java
keyboard
laptop
Large-scale integration
linear sequence
LISP

Logo
machine language
machine oriented
magnetic core memory
main memory
mainframe computer
maintaining
microcomputer/desktop computer
microprocessor
minicomputer
mouse
network
object oriented programming
Object Oriented Turing
object program
operating system
output
output device
output storage area
Pascal
peripheral
personal computer
place value
presentation software
primary storage
printer
problem solving
program storage
programmer
programming environment
programming language
RAM
reduced instruction-set computing (RISC)
repetition
ROM
secondary storage device
selection
silicon
silicon chip
simulation software
software
software audit
software piracy
source program
spreadsheet
structured programming
subscript

supercomputer
switch
symbolic addressing
syntax
top-down programming
transistor
Turing
UNIX
vacuum tube
valid data
very large-scale integration (VLSI)
virus
web browser
well-designed program
word processor
working storage space
World Wide Web

Chapter 2

The Turing Environment

2.1 Introduction

The Turing **programming environment** is the software that allows you to enter and edit Turing programs on the computer. This activity takes place in an **Editor window**. The programming environment provides the means for you to run a program and examine the results in an **Execution window**. The Turing programming environment also contains the software that **translates** (**compiles**) the program in the Turing language into a language that the computer can interpret and then execute.

During execution, the program may **prompt** the user to enter data that will be used by the program to produce its result. The Execution window will contain the output of prompts to the user and the final results. As information is provided by the user, what the user types in is echoed on the screen in the Execution window. For this reason, the Execution window could just as easily be called the **Input/Output window**.

Because these different components that allow you to develop a program are all part of the environment software, it is often called an **Integrated Development Environment** (IDE). Some IDEs, including Turing also contain a **debugger**, which is a tool programmers can use to help locate and fix errors (also called bugs) in programs.

2.2 The Editor Window

When the Turing environment is started, the Editor window is displayed on the screen. The process for starting Turing depends upon the type of personal computer and operating system being used. The process is different, for example, for a PC running a Windows operating system and a Macintosh using MacOS. The Editor window, however, is very similar for both platforms.

Across the top of the Editor window on the PC, or across the top of the screen on a Macintosh, is a **menu bar**, The menus in

the menu bar offer a selection of commands that can be executed by dragging the mouse down from a menu heading and releasing it when the desired command in the menu is reached.

On the PC, there is also a row of buttons below the menu bar. These buttons give instant access to some of the most common commands found in the menus. The PC also has a **status bar** at the bottom of the window. This bar is used to display messages to the user from the environment.

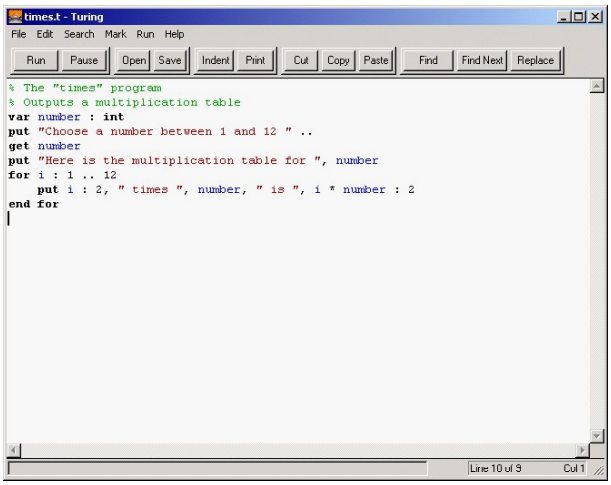


Figure 2.1 Editor window on the PC

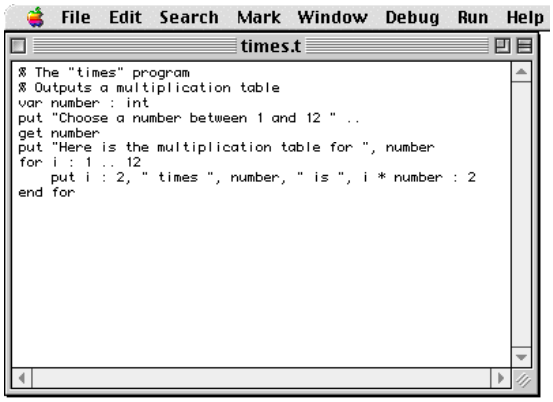


Figure 2.2 Editor window on the Macintosh

Programs are entered into the Editor window and edited just as text is entered using any word processing software. It is assumed that you have some experience:

- typing text,
- moving the cursor,
- deleting characters,
- inserting characters, and
- selecting portions of text.

2.2.1 Cut, Copy, and Paste

You will often wish to copy or move a block of text when writing a program. This can be done easily in the Turing editor using the **Cut**, **Copy**, and **Paste** commands.

To copy a block of text to a different location in the file, you select the portion of the text you wish to copy by clicking and dragging the mouse, or using the cursor keys with the Shift key pressed. You then choose **Copy** from the **Edit** menu. This command copies the selected text to the **clipboard**. The clipboard is a block of memory used for temporarily storing text. The Copy command does not change the text in the Editor window.

To place a copy of the text in another location, move the cursor to the desired place in the text and select **Paste** from the **Edit** menu. This copies the text from the clipboard into the text at the cursor location. The clipboard's contents are not changed, so you can continue to paste the same block of text multiple times.

The **Cut** command in the **Edit** menu is similar to the Copy command except that in addition to copying the selected text to the clipboard, it deletes the text from the Editor window. The Cut command is used to move a block of text to a new location. The **Clear** command in the **Edit** menu deletes the selected text without placing it in the clipboard.

2.2.2 Undo

When editing a program, it is not uncommon to make mistakes such as accidentally overtyping selected text or replacing the wrong selection. The Turing environment allows you to undo the last action performed on the text.

On the PC, you can undo the last several actions, one at a time, using Undo several times in a row. To undo an action, select **Undo** from the **Edit** menu. This restores the text and the cursor position to where it was before the last change. On the Macintosh, however, you can only undo the last action.

To restore a change that was undone by you, you choose the **Redo** from the **Edit** menu on the PC, or **Undo** from the **Edit** menu on the Macintosh.

2.3 Saving Programs on Disk

After entering a program in the Editor window, you must save it to a file on disk. To save programs to the disk you must choose the **Save As** command from the **File** menu. A **Save File** dialog box appears asking you to type in a file name for the file you are saving.

Once you have entered the file name, you must click the **Save** button. If another file already exists by the same name, a second dialog box appears asking whether the contents of the old file are to be replaced by the contents of the file you are saving. This allows you to choose another name for the new file to be saved.

On the PC you can determine if a program has been changed since it was last saved by looking at the window title bar. If the file name is prefixed by a * then the file has been changed since it was last saved.

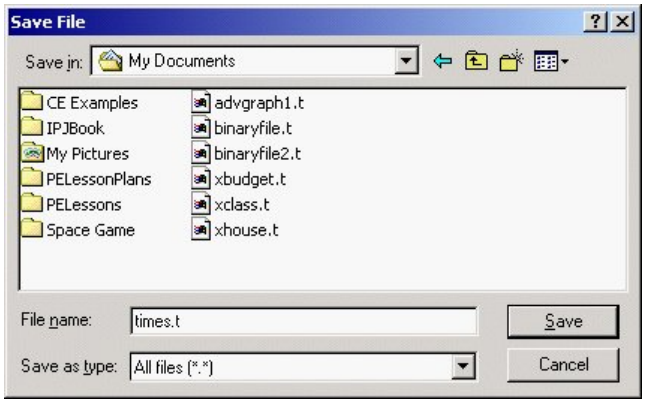


Figure 2.3 Save File Dialog Box on the PC

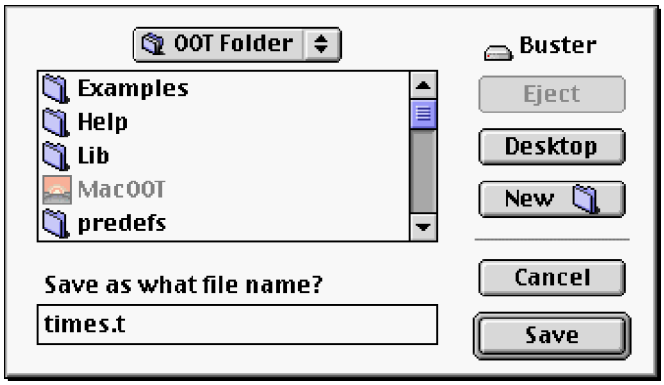


Figure 2.4 Save File Dialog Box on the Macintosh

Commonly Turing files are saved with .t added to the end of the file name. For example, a Turing file containing a program to draw a happy face might be given the file name:

Happy.t

The .t helps identify files as Turing files rather than other kinds of files. On the PC, the Turing environment automatically adds .t to the file name if no **file suffix** is specified. If you do not want any suffix added to the file, save the file with a period at the end of the file name.

2.4 Running Programs

To execute (run) a program, you select the **Run** command from the **Run** menu. If the Turing environment finds errors in the program during the translation (compilation) process, the Editor window highlights the lines on which any errors were found in gray. It highlights the part of the line where the error actually occurred in a different color. It also displays an **error message** giving a description of the error. On the PC, the error message is displayed in the status bar, while on the Macintosh, the error message is displayed in a separate window.

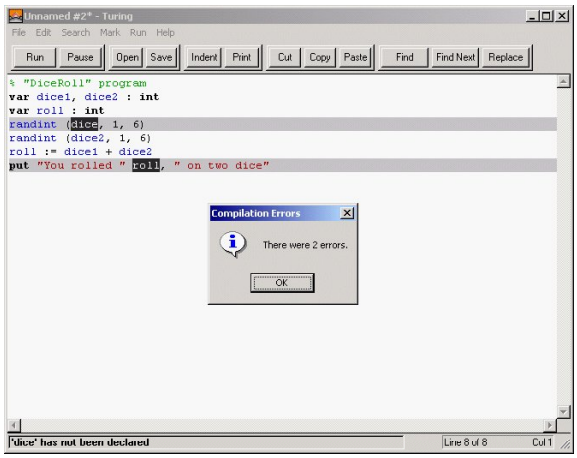


Figure 2.5 Program Error on the PC

Figure 2.6 Program Error on the Macintosh

As you correct a line containing an error, the highlighting disappears. You can move to the next line containing an error by selecting **Find Next Error** from the **Search** menu.

Errors that occur during compilation are called **syntax errors**. These errors are the result of improperly formed programs.

Programs can be run again any time using the **Run** command. Sometimes programmers find it helpful to run the program again even though they have not corrected all of the errors because it helps them make sure that the ones they have fixed are fixed properly.

If you wish to terminate a running program, select **Stop** from the **Run** menu.

2.4.1 Input/Output Redirection

When executing a Turing program, it is possible to specify that input come from a file instead of from the keyboard. Likewise, it is possible to specify that output be sent to a file instead of, or in addition to, the output window. The process of changing where input comes from, or output goes to, is called **input/output redirection**. To redirect input or output select **Run with Arguments** from the **Run** menu. This displays the **Run with Arguments** dialog box. The dialog box allows you to select where input is to come from and where output is to go to.

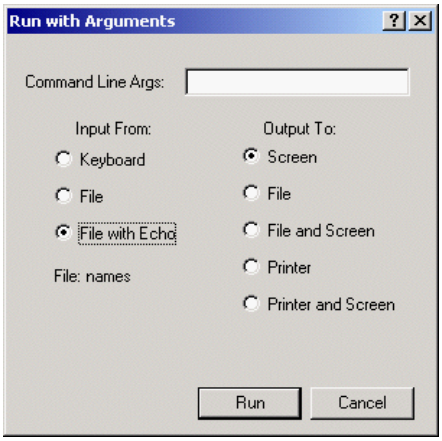


Figure 2.7 Run with Arguments Dialog Box on the PC

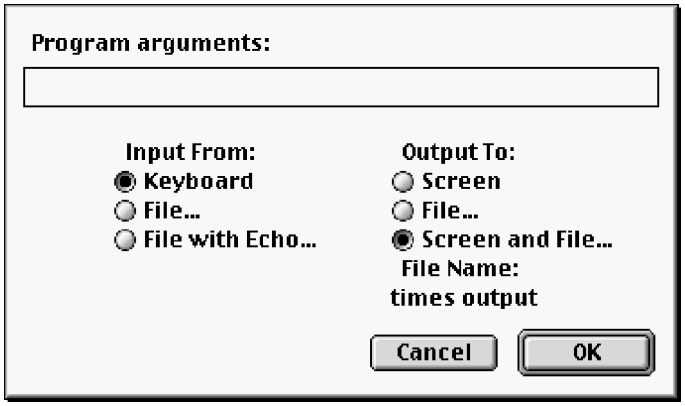


Figure 2.8 Run with Arguments Dialog Box on the Macintosh

To make input come from a file, select either the **File...** or **File with Echo...** radio button under **Input From:**. This displays a **File Open** dialog box which you use to select the file from which to read input. If **File** is selected, input from the file will not be displayed in the Execution window. If **File with Echo** is selected, input read from the file will be echoed to the Execution window.

To redirect output to go to a file, select either the **File...** or **Screen and File...** radio button. This displays a **File Save** dialog box which you use to select the file to which output will be sent. If **File** is selected, output will be sent to the file and not to the Execution window. If the program prompts for input, these prompts will not be seen by the user. As well, any keyboard input will appear in the Execution window, not in the file to which output has been redirected.

If **Screen and File** is chosen, then output will be sent to both the Execution window and to the selected file. On the PC, you can choose to send output to the printer instead of a file. On the Macintosh, however, you must send output to a file and then print that file.

Note that only textual output is sent to the file. Graphical output like rectangles and ovals are sent only to the Execution window.

2.5 Indenting Programs and Syntax Coloring

In programming, it is very important to make programs as clear and easy to understand as possible. This makes them easy for people other than the original programmer to understand and for any programmer to maintain and change over time.

2.5.1 Indenting Programs

Two important ways to make programs clearer are to list each instruction on a separate line and to indent specific groups of instructions from the left margin. This can be done by inserting spaces as you type the program or it can be done automatically by the Turing environment. Selecting the **Indent** command from the **Edit** menu automatically indents the program to its proper structure.

Automatic indentation also helps programmers discover errors in their programs because it provides a visual clue to missing elements of the program. For example, if, after indenting a program, a **loop** statement is aligned with an **end if** statement, then there is a problem with the program and it is likely that an **end loop** statement has been forgotten somewhere.

2.5.2 Syntax Coloring

On the PC, the Turing environment displays various elements of the program in color or in boldface type. This is called **syntax coloring**.

Here is how the elements in a program will appear:

keywords	boldface
comments	green
strings	red

identifiers blue

Syntax coloring can help programmers discover syntax errors in their programs because it provides a visual clue about misspellings or unterminated strings or comments. If a Turing keyword is not displayed in bold face, then it has been misspelled. If parts of a line after a string are displayed in red, then a quotation mark has been forgotten.

2.6 Starting and Stopping the Environment

The process for starting up the Turing environment depends upon the system on which the software is installed. Most schools will have their own particular method of logging on to the system and starting the software. Consult the teacher for details.

To exit the environment, Select **Exit** from the **File** menu on the PC, or **Quit** from the **File** menu on the Macintosh. If there are any Editor windows with unsaved programs in them, a dialog box will appear giving you the chance to save the contents of the window. If you select **Yes**, the program will be saved. If you select **Cancel**, then the Turing environment will not exit. If you select **No**, the contents of that Editor window will be discarded.

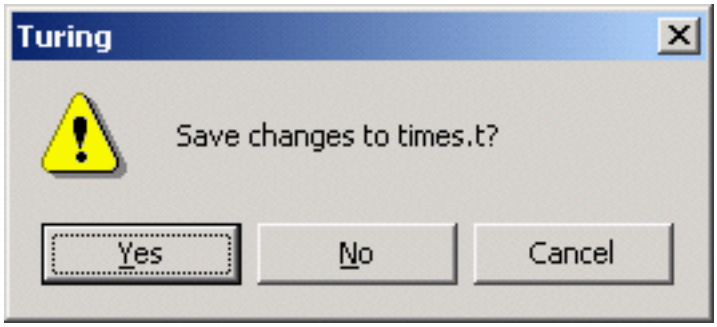


Figure 2.9 Save Program Changes Dialog Box on the PC

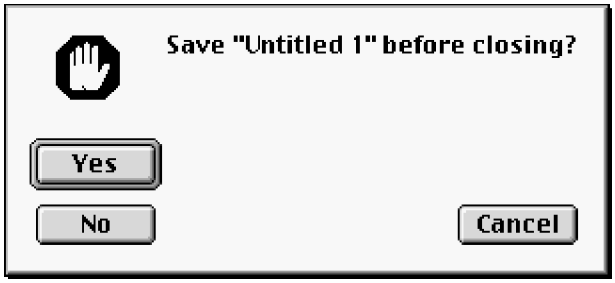


Figure 2.10 Save Program Changes Dialog Box on the Macintosh

On the PC, you can also exit the Turing environment by closing the open Editor window. If you do so, a **Confirmation** dialog box will appear. Clicking **Cancel** will stop the environment from closing.

Figure 2.11 Confirmation Dialog Box on the PC

2.7 Opening an Existing Program

To open an existing file, you select **Open** from the **File** menu. Turing displays the **Open File** dialog box with a list of existing files. Select the file you want to open from this list of files and click **Open**.

On the PC, you can also open a file that has recently been opened by selecting **Recent Files** from the **File** menu. A submenu then appears which contains a list of the last five files you have opened. To open a file, just select the file name from that list.

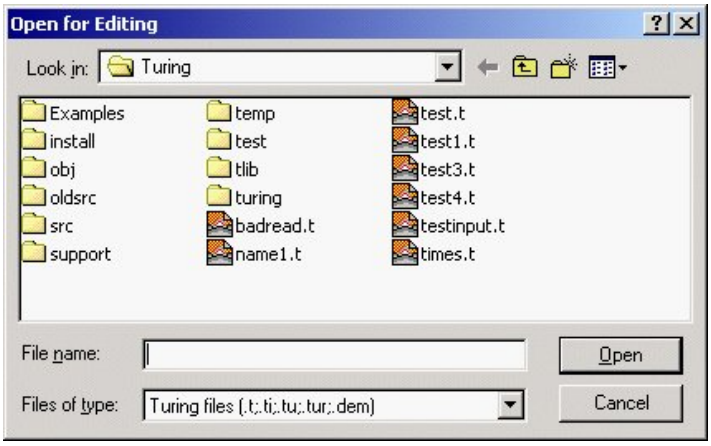


Figure 2.12 Open File Dialog Box on the PC

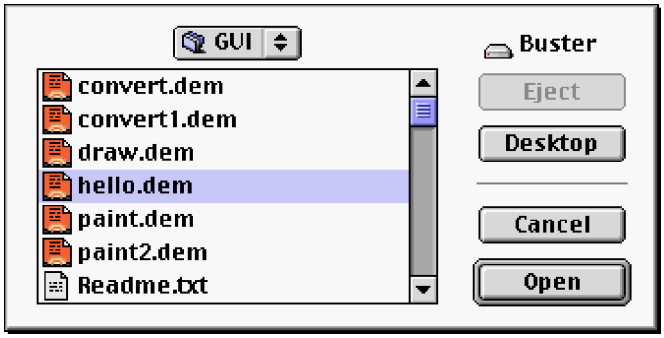


Figure 2.13 Open File Dialog Box on the Macintosh

2.8 Searching for Text in a File

Like many word processing programs, the Turing environment allows you to easily find pieces of text within a file. It also allows you to automatically substitute one piece of text for another. These operations are called **Find** and **Replace**.

2.8.1 Finding Text in a File

To find a piece of text in a file, you select **Find** from the **Search** menu. A **Find** dialog box is then displayed. You enter the text to be found in the **Find What?** text field. This text is called the **search text**. You then click the **Find Next** button. The next occurrence of the search text in the file is highlighted and displayed in the Editor window. If the text is not found, a message is displayed. To search for the previous occurrence of the search text in the window, select the **Up** radio button on the PC, or the **Search Backwards** check box on the Macintosh.

To find subsequent occurrences of the same search text without having to reopen the **Find** dialog box, press **F3** on the PC or **Command+G** on the Macintosh.

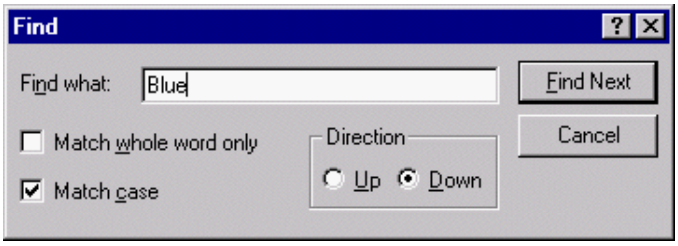


Figure 2.14 Find Dialog Box on the PC

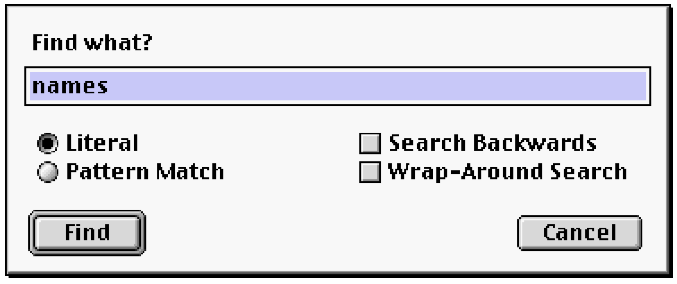


Figure 2.15 Find Dialog Box on the Macintosh

2.8.2 Replacing Text in the File

The editor is also capable of replacing found text. This can be useful when changing the name of a variable. To display the **Replace** dialog box, select **Replace** from the **Search** menu on the PC, or **Change** from the **Search** menu on the Macintosh.

Like the **Find** dialog, you enter the text to be found in the **Find What?** text field. You then enter the text with which it is to be replaced in the **Replace With** text field. You then click the **Find Next** button to highlight the next occurrence of the search text in the file. Unlike the **Find** dialog box, the **Replace** dialog box does not disappear when the search text is found and highlighted.

At this point, you should click the **Find Next** button if you want to skip to the next occurrence of the search text without replacing the currently highlighted text. To replace the currently selected text with the replacement text and search for the next occurrence of the search text, click the **Replace** button on the PC, or the **Change, then Find** button on the Macintosh. On the Macintosh, you can also click the **Change** button to replace the highlighted text without searching for the next occurrence of the search text.

The **Replace All** button causes every occurrence of the search text in the file to be replaced with the replacement text.

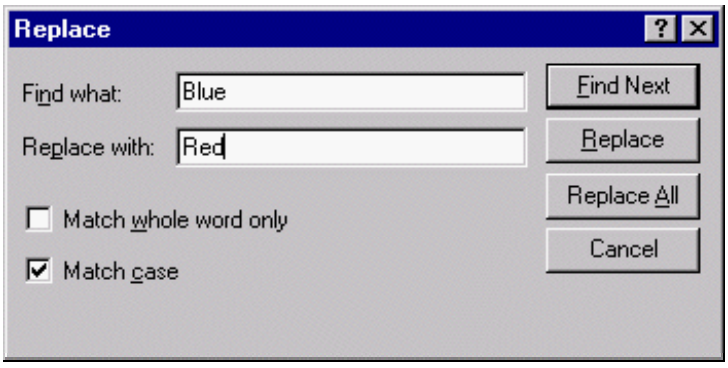


Figure 2.16 Replace Dialog Box on the PC

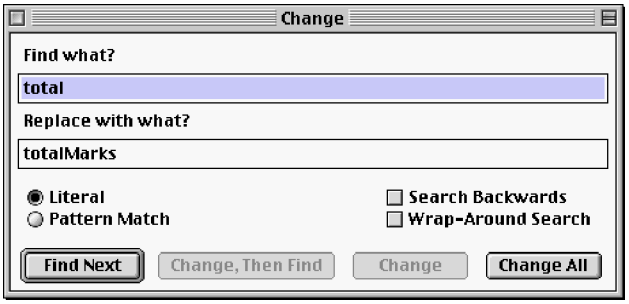


Figure 2.17 Replace Dialog Box on the Macintosh

2.9 Printing a Program

To print a program, select **Print** from the **File** menu. This displays the **Print** dialog box. This dialog box provides a variety of options such as selection of pages to print and, on the PC, whether to bold face keywords and italicize identifiers.

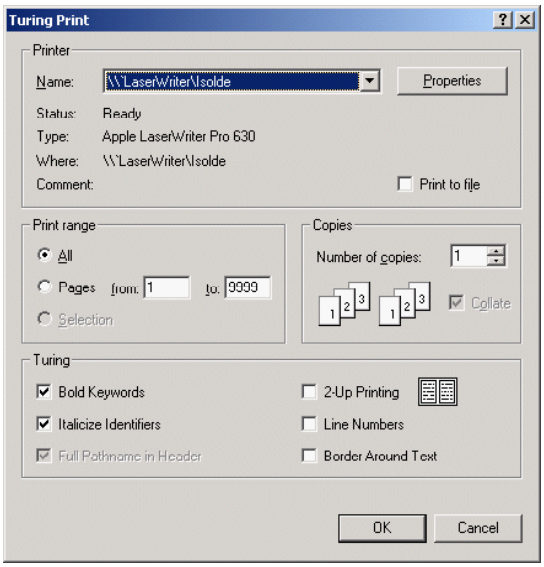


Figure 2.18 Print Dialog Box on the PC

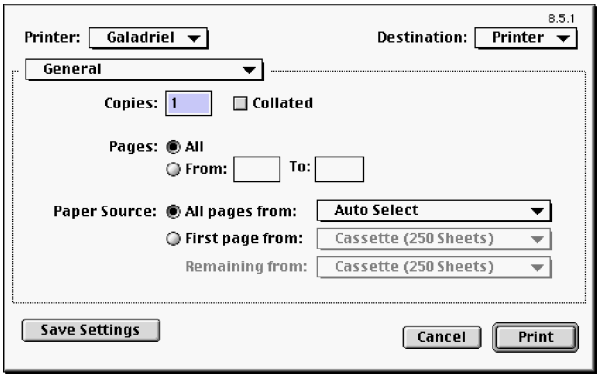


Figure 2.19 Print Dialog Box on the Macintosh

2.10 Example Turing Programs

Even though you have not yet studied how to create your own Turing programs, you can learn about the Turing environment by typing programs in and running them.

When you start the Turing environment it automatically provides a blank Editor window in which you can begin entering your program. If there is no blank Editor window, you simply select the **New** command from the **File** menu.

Here is a program for you to type in and store on the disk. When you type this program in, do not indent it. Instead, wait until you have typed in the entire program and select the **Indent** command. Notice how the automatic indenting structures the parts of the program.

```
% The "TimesTable" program
% Outputs a multiplication table
var number : int
put "Choose a number between 1 and 12 " ..
get number
put "Here is the multiplication table for ", number
for i : 1 .. 12
    put i : 2, " times ", number, " is ", i * number : 2
```

```
end for
```

Once you have entered and indented this program, select the **Run** command from the **Run** menu. The screen clears when the program begins execution. When the prompt line

```
Choose a number between 1 and 12
```

appears on the screen, type a number between 1 and 12 inclusive. When you finish typing, press Enter so the computer knows you have finished. As soon as you press Enter, the multiplication table appears for the number you have chosen. The number you type is shown here in boldface.

Here is a sample Execution window:

```
Choose a number between 1 and 12 6
Here is the multiplication table for 6
  1 times 6 is  6
  2 times 6 is 12
  3 times 6 is 18
  4 times 6 is 24
  5 times 6 is 30
  6 times 6 is 36
  7 times 6 is 42
  8 times 6 is 48
  9 times 6 is 56
 10 times 6 is 60
 11 times 6 is 66
 12 times 6 is 72
```

Suppose you wanted to have a table showing your number multiplied by the values 1 to 20 rather than 1 to 12. Can you guess what change in the program might do this? Try changing the program and then run it again. Store the changed program under a new file name, say *TimesTable20.t.*

Try changing the program by substituting a / for the * in the program. This produces a division table instead of a multiplication table. Can you fix the rest of the program to suit this change? Store this program as *Divide.t* on the disk.

Here is a slightly longer program. When you have finished typing it in, give the **Run** command and play the guessing game. After you have played the game try reading the program to see if you can understand some of it.

```
% The "GuessNumber" program
% Chooses a number at random between 1 and 99
% and allows you to guess it
var hidden, guess : int
var reply : string (1)
put "See if you can guess the hidden number"
put "It is between 1 and 99 inclusive"
loop
    var count : int := 0
    put "Do you want to play? Answer y or n " ..
    get reply
    exit when reply = "n"
    % Choose a random number between 1 and 99
    randint (hidden, 1, 99)
    loop
        put "Enter your guess (any number between 1 and 99) " ..
        get guess
        count := count + 1
        if guess < hidden then
            put "You are low"
        elsif guess > hidden then
            put "You are high"
        else
            put "You got it in ", count, " guesses"
            exit
        end if
    end loop
end loop
```

Save the program on the disk as *GuessNumber.t*

2.11 Exercises

1. The program window can be used to enter any kind of data. Clear the program window, then enter a short letter to your teacher telling her or him how exciting it is to be using the computer as a simple word processor. Print the letter if you have a printer. Store the letter on the disk under the file name *Teacher*. Check the directory to see that it is there.
2. Change the letter you wrote for question 1 so that an extra paragraph is added about how simple it is to edit text on a computer. Arrange to address this same letter to a friend as well as to your teacher. Print both new letters if you have a printer. Store them as files called *Teacher2* and *Friend*. Check the directory to see that all three files are there.
3. Here is a Turing program to type into the program window and run.

```
% The "Seesaw" program
% Makes saw tooth patterns
loop
    put "How many teeth do you want? (1–12) "
    var count : int
    get count
    put repeat ("*     ", count)
    put repeat (" *   *", count)
    put repeat ("  * * ", count)
    put repeat ("   *  ", count)
end loop
```

Try running the program. If you get tired of making saw tooth patterns you can stop the execution of the program by selecting **Stop** from the Run menu.

2.12 Technical Terms

Turing environment
Editor window
Execution window
translation
prompt
Input/Output window
Integrated Development Environment (IDE)
menu bar
status bar
cursor
indent
syntax coloring
cut
copy
paste
clipboard
file suffix
error message
syntax error
redirecting output to disk
redirecting input from disk

Chapter 3

Program Design and Style

3.1 Programming and Programmers

When a programmer creates a program she or he does much more than simply sit down at a computer and begin entering commands into the keyboard. There is a whole range of activities that the programmer must do in order to create a **well-designed program**, that is, a program that reliably solves the problem it was created to solve.

Programming is the activity of:

- Analyzing a problem to be solved.
- Preparing a design for the steps in a set of instructions (an algorithm) that together with the data, will solve the problem.
- Expressing the algorithm in a language that the computer can ultimately execute.
- Ensuring that there is adequate documentation so that other people, and not just the original programmer, can understand the computer program.
- Testing and validating the program to ensure that it will give correct results for the full range of data that it is to process.
- Maintaining the program over time so that any changes that are needed are incorporated.

Many years ago very few people knew how to program and those who did were often thought of as ÒgurusÓ. Because very few people really understood computers a number of negative stereotypes began to be applied to people who did. Programmers were sometimes thought of as people, almost always males, who spent all of their time with machines and did not relate very well to other people. Like many negative stereotypes, however, this one is false.

The term **hacker** is also used to describe people who program computers but, over time, it has taken on even more negative connotations. Today the term is used to describe people who use computers to play pranks such as gaining illegal access to other people's or organizations' computers and changing data, or creating destructive computer programs called **viruses**.

Computer viruses are actually hidden programs which, when downloaded by the unsuspecting user, perform unauthorized activities on any computer in which they reside. Some viruses can cause considerable damage, so many computer users run **virus checker software** to detect problems in any programs or files they load onto their computer.

Programmers come from all walks of life and programming itself can take many different forms. The field of **information technology**, which covers a wide variety of computing activities, offers many interesting and diverse career options. In truth, the only things most programmers share are their abilities to:

- solve problems,
- organize information,
- design solutions,
- express instructions in a logical sequence, and
- input them into a computer.

3.2 Programming Style

Often, there are two types of computer programs: programs that work but do not make sense to anyone except the person who created them, and programs that any programmer can read and understand. In the real world, no one has much use for messy, hard to read programs. As programs become bigger and more complex, they are almost always worked on by a number of different people. This means that a number of people will have to read parts of the program that they did not write. Also, over time, programs need to be changed. Often the person who changes the program is not the original programmer, but she or he still must be able to read it and understand how it works.

One of the ways to make sure that programs make sense is to follow a set of guidelines sometimes referred to as a **programming style guide**. A programming style guide provides a set of rules or expectations that every programmer must follow

to ensure that all programs are clearly written and easy to fix or change. While programming style guides may be different for different workplaces, they all have certain things in common. (These terms will become more familiar as you begin to develop larger programs.)

Quick Guide to Good Programming Style:

1. Create a header at the beginning of every program. A header is not part of the actual program, but identifies the programmer and briefly explains what the program does. It should contain:

   - programmer's name,
   - the date,
   - the name of the saved program,
   - project (teacher's) name, and
   - a brief description of what the program does.

2. Always include comments to explain what is happening in the program and what various parts of it do. Variables should include a brief description of how each variable is used.

3. Use names for variables that give a good indication of what they do (for example, use ÒpriceÓ rather than ÒpÓ).

4. Make the structure of programs obvious by indenting the contents of loops and if-then-else structures.

One of the most important aspects of good programming style is to provide information about how the various parts of a program work within the body of the program itself. This is usually done by including **comments** at various places throughout the program. These comments are written in English but are preceded by some kind of symbol which alerts the compiler or interpreter that they are not to be read as instructions. In Java, for

example, all comments are preceded by a // of /*....*/ sign. For example,

```
// This is a comment.
/* This is also a comment.
As is this line. */
```

Comments are usually placed before sections of the program to explain the purpose of the command or commands following the comment. This is especially important for long programs. The comments help the programmer to better understand the structure and purpose of the program as a whole as well as its various parts.

Efficiency is another important consideration in good programming style. An efficient program is one which meets the specified requirements, executes in the least possible amount of time, and is logically organized with clearly documented code so that it can be easily maintained.

3.3 The Software Development Process

Large software development projects can go through a sequence of these steps one after the other in a **software life cycle** as shown in Figure 3.1. This series of steps is sometimes called a **waterfall model** because it looks like water cascading down from one level to the next.

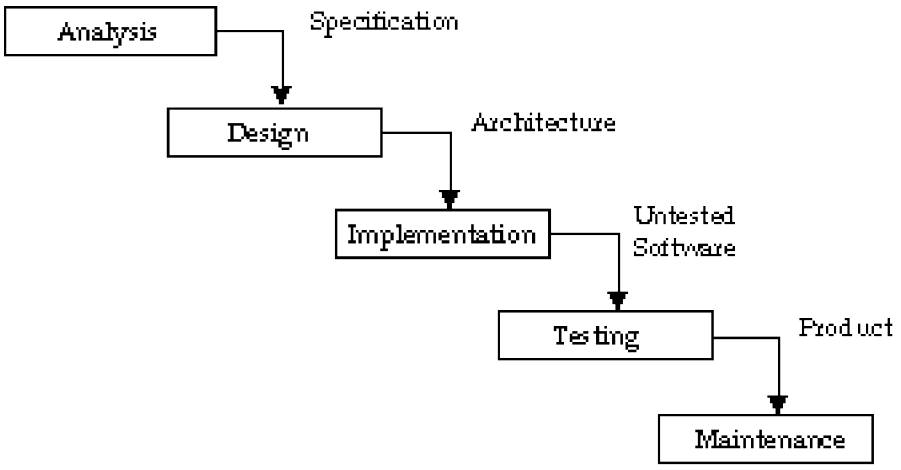


Figure 3.1 Waterfall Model of Software Development

In the early days of computing the first three parts of the programming activity were sometimes called analyzing, programming, and coding. Each of these activities might have been done by a separate person with special qualifications. Over time, however, the increasing complexity of the software being developed has required that greater emphasis be placed on the analysis of problems to be solved and the resulting design of programs and applications.

The implementation stage (once called coding because most early programs were written in machine code) has also evolved to meet the need for programs which can be understood by anyone with some knowledge of the nature of programming languages. Programmers today are much more likely to use a high-level programming language. The advantage of a high-level language is that it is easier to understand.

The waterfall model is an idealization of the programming process. In reality the process is closer to that shown in Figure 3.2.

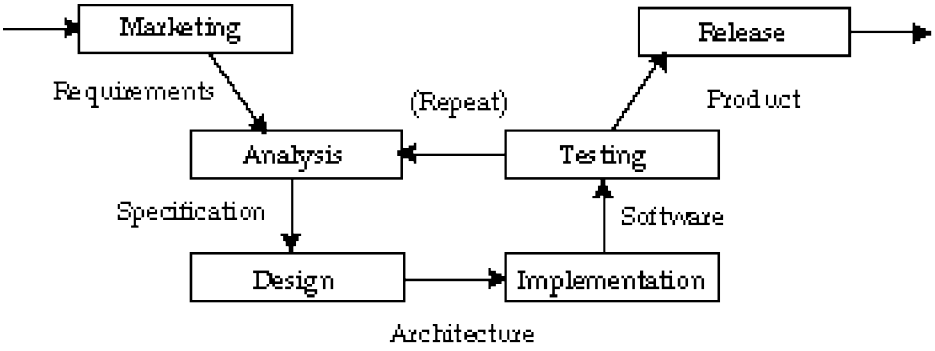


Figure 3.2 Iterative Model of Software Evolution

There is a certain amount of backtracking in the actual evolution of a piece of software of any complexity, particularly if the software is constantly being updated.

In many ways we try to prepare programs that are self-documenting by using variable names such as *netPay* to indicate that the variable contains information concerning the net pay of an employee.

The programming language Turing has a number of words called **keywords** that are used for various operations, such as **if** for selection or **while** for repetition. These have been chosen to be brief but expressive. When a programmer invents names to identify entities (**identifiers**) it is best if they are as brief as possible without sacrificing understandability. Abbreviations should be used only when they are well-understood, such as *GST* for Goods and Services Tax and *FBI* for Federal Bureau of Investigation.

Since the purpose of creating computer programs is to solve problems, before anything else, a programmer must be a problem solver. **Problem solving** is the process of breaking problems down into smaller more manageable parts. As the large problem is broken down into smaller parts, each part becomes more specific. These individual solutions to smaller problems are then combined to solve the larger problem. Problem solving is also referred to as analyzing the problem to be solved.

Once the programmer has analyzed the problem, he or she is ready to begin designing a solution. In programming, the design stage involves defining a set of instructions (called an **algorithm**) that, together with the data, will solve the problem. The next step is to express this algorithm in a programming language so that the computer can **execute** it (carry out the instructions). This stage is sometimes referred to as **coding** although this term is not as popular as it once was.

Because people other than the programmer must be able to understand, use, and often modify a program, the program must be understandable. This means ensuring that there is good documentation. There are commonly two kinds of documentation. The term **internal documentation** refers to the comments within a program that explain what the various parts do. The supplementary resources such as Reference Manuals and User's Manuals are **external documentation**. Often, an application will also provide internal **help files** that the user can access.

Once the program has been written it must be extensively tested to ensure that it performs the way it should. Testing often involves running the program with a full range of data. The programmer must also be sure to run it with both **valid data**, that is, the data the user should be inputting, and **invalid data**, that is, incorrect or unexpected data, to determine how the program will behave in as many situations as possible.

When computer programs control such profound aspects of our lives as the airplanes we fly in and the machines that provide radiation to cancer patients, the results of improper or insufficient testing can be wide-ranging and catastrophic.

The programming process also involves the job of **maintaining** a program over time so that any changes that are needed are incorporated. Many businesses, for example, must frequently change or enhance their programs to make them perform new tasks or perform old tasks more efficiently.

3.4 Procedure-Oriented Programming

In procedure-oriented programming, programs are designing in a **top-down** manner. The programmer starts by writing the specification of the problem to be solved and gradually refines the solution by breaking it into parts, each of which can in turn be broken into parts. These parts are often solved (implemented) as **procedures**.

Figure 3.3 is called a **structure chart**. It illustrates how the each task is broken down into subtasks. In a program, each subtask might consist of a procedure which would be **called** by the procedure above it in the structure chart.

This structure chart shows how a program is **refined step by step** starting at the top (1), with a statement of what is required, here sorting a list of names alphabetically. This top node in the diagram is expanded or refined into three nodes 2, 3, and 4. The computations represented by these three nodes, carried out sequentially, satisfy the top node's specification. The nodes 5, 6, and 7 are further refinements in the procedure-oriented programming language.

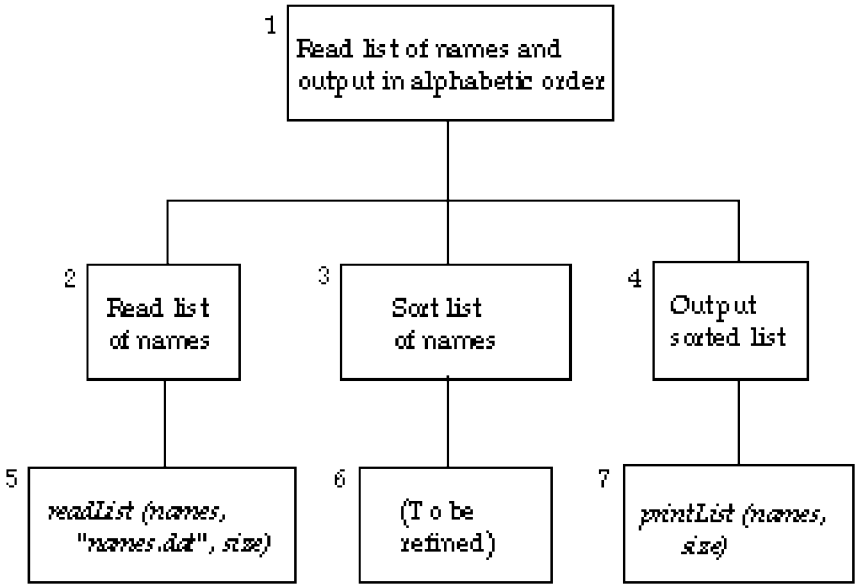


Figure 3.3 Top-Down Programming

3.5 Exercises

1. Define the following:
   (a) algorithm
   (b) hacker
   (c) virus
2. Explain the purpose of a programming style guide?
3. Explain the main differences between the waterfall model of software development and the iterative model of software development.
4. Describe the process by which a programmer designs a program.
5. What does a structure chart show?

3.6 Technical Terms

algorithm
called
coding
comments
documentation
execute
help files
keyword
identifier
information technology
invalid data
maintaining
problem solving
procedure
procedure-oriented programming
programming style guide
software lifecycle
step-by-step refinement
structure chart
top-down
valid data
virus
virus checker software
waterfall model
well-designed program

Chapter 4

Simple Programs

4.1 A One-Line Program

By following the examples in this chapter you can learn how to write programs in the Turing **programming language** and have them execute (run) on your computer. Here is a program for you to try. (Make sure that you have a clear window. If not, use the *new* command.)

```
put "Have a good day"
```

This is a one-line program. Type it into the window just as you did other text. When you finish typing the program, you can run it. The result of the execution will appear in the Execution window. The output will be

```
Have a good day
```

Concentrate on the output from your program. Notice that the **put** caused output. What is output is the **string** of characters that is enclosed in the double quotes following **put**. The quotes themselves are not output. The **put** is a **keyword** in the Turing language and is used whenever you want output. Keywords in this book are shown in **boldface** type. In the window they will not be boldface. Whenever a new technical word appears in the text it is also shown in boldface type so you will notice it. Usually the meaning of the new term is clear from the context. The next time it appears it will just be in ordinary type. Keywords from Turing will always be boldface.

4.2 Changing the Program

The execution of the first program resulted in the output of

```
Have a good day
```

We are going to change the program so that it will type a person's name. If the name is Pat, it will say

```
Have a great day, Pat
```

Go back to the Editor window if the Execution window is still showing. Now use the editing methods we already discussed to make the changes. We want to change the word *good* into the word *great* and add *Pat* at the end of the line. We cannot simply exchange letters by typing *great* over top of *good* because *great* has one more letter. See if you can do this. When you are done, run the program again to see that you get the expected result.

4.3 Substituting One String of Characters for Another

In our editing we changed the word *good* to the word *great*. This can also be done by using the **Replace** command (**Change** on the Macintosh) from the **Search** menu. This allows you to change a string, called the **source string,** to another string, called the **target string**. In the example above, use the Replace command to change *Pat* to *Chris*. Now run the program again.

4.4 A Program that Computes

The program that outputs "Have a good day" shows you how the computer can be programmed to display a message to the user. This is very important when you want other people to use your programs. You can tell them what you want them to do.

But computers got their name from the fact that their original purpose was to calculate, using numbers. They can, for instance, add 2 and 2. Here is a program to do this.

```
put 2 + 2
```

Try running this one-line program. It will output the number 4. Now change the program to this.

```
put 2 * 2
```

You will get 4 when you run it. The * means multiply 2 by 2. If you don't believe it multiplies, change it to

```
put 2 * 3
```

You get 6. When you try

```
put 2 + 3
```

you get 5. Try subtraction

```
put 6 – 2
```

If you want to divide, you use the slash / symbol. The program

```
put 9 / 3
```

will produce 3 which is 9 divided by 3.

4.5 Integers and Real Numbers

So far all the calculations we have shown involve integers, which are numbers without fractions. If you write this program

```
put 4 / 3
```

the result is 1.333333. The answer has a decimal point in it. We say it is a **real** number rather than an **integer** which has no decimal point. Try this one

```
put 2 / 3
```

The result is 0.666667. There are 7 digits in the answer. The true answer would have an infinite number of digits; the 6 would just keep repeating. When this real number is represented by a

limited number of digits, the number is rounded off. The rule for rounding is that the computer attempts to produce the closest answer to the exact result. That is why the last 6 is rounded up to a 7.

You can do calculations with real numbers as well as with integers. For example,

```
put 7.2 + 9.35
```

will give the output 16.55.

4.6 Arithmetic Expressions

We have been asking the computer to do arithmetic for us and have given it **arithmetic expressions** to be evaluated. These expressions involved two numbers, either real or integer combined in some operation such as addition, subtraction, multiplication, and division. We can also write more complicated arithmetic expressions and the computer can deal with these just as you would. For example,

```
put 5 + 6 * 3
```

will produce the output 23. You would perform the multiplication first because the multiplication operator * has **precedence** over the addition operator +. The normal rules of precedence apply also to the computer: multiplication and division before addition and subtraction, and left to right for equal precedence operators. You can change the precedence by using parentheses in the arithmetic expression. For example,

```
put (5 + 6) * 3
```

gives the output 33. The operations inside parentheses have precedence over those outside.

4.7 Combining Calculations and Messages

We can write output statements which have more than one output item, for example, a message in quotes followed by an arithmetic expression. Here is such a program.

```
put "2 + 2=", 2 + 2
```

It produces this output

```
2 + 2=4
```

The message in quotes is output exactly as typed, with the quotes removed, then the value of the arithmetic expression follows, namely 4. This kind of statement is used when we want to say what a calculated number represents. We **label** the results.

4.8 Output of a Series of Items

You can output a list of items provided they are separated from each other by commas. For example,

```
put 2 + 5, 3 * 7, 5 - 2
```

produces the output 7213. The output values are all run together so that it looks like one value. You can space items apart by including spaces (blanks in quotation marks) between them, like this

```
put 2 + 5, " ", 3 * 7, " ", 5 - 2
```

The output is now

```
7 21 3
```

Another way to separate output is to give a **field size** in which the item is to be output. This field size is placed after the output item referred to and preceded by a colon. For example,

```
put 2 + 5 : 5, 3 * 7 : 5, 5 – 2 : 5
```

Here each output item is assigned a field of size 5 spaces. It will be output right-justified in the size of field so that the output will look like this

```
°°°°7°°°21°°°°3
```

There is no output where the °s are. They are there just to show you how many spaces are left. The actual output is

```
7    21    3
```

Using field sizes or blanks in quotation marks allows us to space the output items along a line any way we want. We **format** the output.

When we format a message in quotes by specifying a field size it is left-justified (placed on the left end) in its field. Numbers are right justified. To get multiple blanks in the format we can use a single blank in quotes followed by a field size. For example, there will be 10 blanks between the 5 and 6 in the output from this program

```
put 5, " ": 10, 6
```

4.9 A Series of Output Statements

Each **put** statement starts a new line of output. For example, here is a program to output two lines.

```
put "To be or not to be"
put "That is the question"
```

You do not have to start a new line with every new **put** statement. If you want the following **put** statement to continue on the same line then just follow the output items of the first **put** by two dots. For example,

```
put "Albert" ..
put "Einstein"
```

produces output on a single line

```
AlbertEinstein
```

If you want a space between these you must include a separate blank output item or have a blank inside the quotes following *Albert* or preceding *Einstein*. To output a blank line we use

```
put " "
```

where what is output on the line is a string with no characters in it.

4.10 Exercises

1. Erase the window (or create a new one) using the *New* command in the *File* menu and enter a program that wishes you a happy birthday. Run it, then change it to a greeting that includes a friend's name. Try using the replace command and give birthday greetings to another friend.
2. Enter a program to output a number of lines of a song. Pick a repetitious song like "Old MacDonald Had a Farm" and enter the first verse and chorus. Run it. Now use the editing facilities of the Turing environment to add a second verse and chorus without entering the entire words; copy the program that produces the first verse and chorus and make the necessary changes. Store your program on the disk.
3. Write a program to output an address label, with name, street, city, province, and postal code. Copy the program several times using *Copy* and *Paste*. Run the program to produce a series of identical address labels. Add **put** statements between the ones producing the labels so that there are two blank lines between each label.
4. Write programs to calculate:
   a. the area of a circle of radius 10 m,

b. the annual interest payable on a loan of $5365.25 at 12.5%, and

c. the sales tax payable at 7% on a purchase of $12.50.

In the last two examples the answers are not rounded off to the nearest cent. You can force this by indicating not only the field size but the numbers of decimal places to be output. For example,

```
put 1.66666 : 5 : 2
```

will produce °1.67. The 2 indicates that you want 2 decimal places. The last decimal digit is rounded off. Change the programs for parts b and c to output values to the nearest cent.

5. Experiment to see what happens when you specify a field size which is too small to hold a number. For example, try this program.

```
put "How many digits ", 1 / 3 : 5
put "What about this?": 8
```

6. What happens when you work with very big numbers? Try this example

```
put 32516 * 578632.0
```

Try others. When the result is output, it is in the **exponent** form, often used in science. Try leaving the decimal point off the second number.

7. What happens when you work with very small numbers. Try this example

```
put 0.0000003 * 0.000006
```

Try others. The exponents or powers of 10 are negative.

8. Try a program with the ** arithmetic operator

```
put 7 ** 2, " is the same as ", 7 * 7
```

The ** is the operator for **exponentiation**. Try other examples.

9. Use the exponent form of real numbers in an arithmetic expression. For example,

   put 6e5 + 2e2, " is the same as ", 600000 + 200

   When is the answer output in exponent form? Experiment using other examples.

10. In Turing there are a number of mathematical functions predefined in the language. Try this

    put "The square root of 49 is ", sqrt (49)

    The *sqrt* function produces the square root of what follows in parentheses. Try this

    put sqrt (371 ** 2)

    Experiment with the square root function.

4.11 Technical Terms

programming language
put **instruction**
string
line editor
appending lines
substitute command
execute
arithmetic expression
arithmetic operator
precedence of operators
parentheses
real number
integer
field size
right justified
left justified
format
round off of number
exponent form
exponentiation operator
square root function
blank line
new page

Chapter 5

Variables and Constants

5.1 Storing Information in the Computer

So far the Turing programs you have seen just output messages or the results of a calculation. You can output a series of such items and format them so the output is easily understood. Nothing very exciting happens in programs until you learn how to input information and store it in the computer's memory.

The kind of information or **data** we store is mostly either numbers (real or integer) or strings of characters. When you store information in the computer's memory you must remember where you stored it. To do this you give each location where you will store data a **name** or **identifier**. What you store in a named memory location can change; it can vary. You call the locations that hold the data **variables**. In writing a program that uses variables you must declare the names you intend to give to your variables. You make up the names yourself. After you have established what the names will be, you can input information into the variables from the keyboard, change the values of the variables, and output their values into the window.

5.2 Declaring Variables

In naming variables you must declare what type of information they are to hold, that is, the **data type** of the variable. A variable to hold numbers can be either a **real** or **int** (for integer) data type. Strings of characters can be stored in variables of type **string**. Variables that you want to use in a program must be declared before you use them.

To declare a variable you write a line in your program which has the form

var name-of-variable : type-of-variable

A declaration begins with the keyword **var** (for variable). Here is a declaration which declares a variable called *age* which is to hold an integer.

```
var age: int
```

The name of the variable *age* is followed by a colon (:) then its data type which is **int**. The types are also keywords in the Turing language but the name is something you make up yourself.

5.2.1 Names of Variables

Names or identifiers of variables contain letters of the alphabet. After the first letter, you can use digits (0 to 9) and underscores (_), but no special characters such as spaces, hyphens, or periods. For example, *page3* is valid name but *3rd* is not because it does not have a letter as its first character. If you want to have a name which is really two or more words such as *this year,* you could write it as *this_year* but we will write it as one word *thisYear* using a capital letter to show where the second word begins. Here is a declaration for a variable to store a person's first name.

```
var firstName: string
```

There is a list of words in Turing that are **reserved** and may not be used as names of variables. These words are listed in the appendix.

5.3 Inputting Character Strings

Here is a program that reads a string and outputs a message with the string in it.

```
put "Enter your first name"
var firstName: string
get firstName
put "Hello ", firstName
```

The first line of the program outputs a message which **prompts** you to enter your first name. The next line is the declaration of the variable *firstName*. After that is the input instruction. It starts with the keyword **get** and then follows the name of the location where what you input is to be stored. The last statement outputs two items: the string in quotes, then the value of the variable *firstName*. Type the program into your computer and try it. After the prompt

```
Enter your first name
```

appears in the window the cursor will be at the beginning of the line following the prompt. Nothing will happen until you type a name followed by Return. If you do not press Return after typing your name the computer will not know that you have finished. For example, if your name were Diana you might type a nickname such as Di and expect the computer to go on and say *Hello Di* . But it will do nothing until you press Return. Run it again, this time giving your second name instead of your first name. The computer does not tell you that you are mistaken. It cannot tell the difference between a first name and a second name or even a last name. Here is how the Execution window might look after the program has run

```
Enter your first name
Albert
Hello Albert
```

The computer produces the first and third lines with the two **put** instructions; you type in the second. Both what you type and what the computer outputs are displayed in the Execution window in the same way so that when you are finished you cannot tell the **input** from the **output**.

5.3.1 Strings Containing Blanks

When you input a string in the normal way it is assumed that there are no blanks in it. For example, if you enter your first and last names in the previous program your Execution window will look like this

```
Enter your first name
Albert Einstein
Hello Albert
```

The last name is ignored. This is because the **get** operator for a string variable reads characters until it hits "white space" such as a blank or a Return. Whatever comes after that is not read. You could get it to read the whole name if you put it in quotation marks as here

```
Enter your first name
"Albert Einstein"
Hello Albert Einstein
```

We call a quoted string or a string surrounded by white space a **token** and say we are using token-oriented input. Turing includes a version of the **get** command that allows an entire line including spaces to be input. (See Chapter 10 for information on Òline-oriented inputÓ.)

5.4 Mistakes in Programs

All variables used in programs must be declared before you use them. If, by mistake, you omit a declaration the computer will give you an error message indicating that in a certain statement you are attempting to use an undeclared variable. This kind of error is called a **syntax error**. Turing, like the English language, has rules of grammar or syntax that must be obeyed. One of these syntax rules is that variables used in programs must be declared before use.

If, by mistake, you spell a variable's name differently in the declaration and the statement where it is used you get the same syntax error. You are using a variable that has not been declared. For example, if you forgot to capitalize the *N* in *firstName* in the **get** statement you would get a syntax error message. When you get such a message, change the program to correct the error and run it again.

5.5 Inputting Numbers

When you are inputting numbers you must remember that there are two kinds of numbers: real numbers and integers. A variable declared as **int** can hold an integer such as 23 but not a real number such as 8.47. A variable declared as **real** can hold a real number such as 8.47. It can also hold 23.0, which is a real number that we can consider to be the same as the integer 23. A **string** variable can hold a string of characters, such as *Albert*, but not a number as such.

Here is a program that reads an integer and outputs its square.

```
var number: int
put "Enter an integer"
get number
put "The square of ", number, " is ", number  * number
```

Here the declaration of the variable *number* precedes the prompt. It does not matter in what order these are given since the prompt does not use the variable. Here is a sample execution

```
Enter an integer
12
The square of 12 is 144
```

Try running the program yourself.

5.5.1 Mistakes in Data

If in the previous program you enter a real number the computer will report an error. This is called an **execution** or **run time** error since the error is not in the program itself but occurs during the running of the program. Here is an example where a data error occurs.

```
Enter an integer
8.47
Line 3: Illegal integer input
```

You should not enter a real number or a character string when the computer expects an integer.

5.6 Inputting Real Numbers

Here is an example using real numbers.

```
put "Enter the radius of a circle"
var radius: real
get radius
put "Area is ", 3.14159 * radius ** 2
```

Here is a sample Execution window.

```
Enter the radius of a circle
35
Area is 3848.44775
```

Notice that the input of an integer is legal for a real variable. At most 6 digits appear to the right of the decimal point in the area. If fewer appear it is because they are zero. Since exponentiation (**) has higher precedence than multiplication, the radius is squared before multiplying by 3.14159.

If the radius is large the area is output in the exponent form. Here is such an example Execution window.

```
Enter the radius of a circle
6754
Area is 1.433084e8
```

The digit 8 following the e is the power of 10 that is to multiply the significant digits part. For small values the exponent form is also used. For example,

```
Enter the radius of a circle
.0000897
Area is 2.527752e-8
```

The exponent form is used automatically by the computer when the number it has to output is too large or too small for its

normal form of real numbers. You can use real numbers in the exponent form in your program or input if you want to.

5.7 Constants

Sometimes we use memory locations to hold information that remains constant. Such constants can be declared just as variables are declared but in this case a value must be assigned to the constant at the time of declaration. For example, we could declare a constant *pi* by this declaration

```
const pi : real := 3.14159
```

Constants can also be strings. For example we could declare a constant *prompt* using this declaration

```
const prompt : string := "Enter your name "
```

Notice that the name of the constant is followed by the type then := and then the value. As with variable declarations, constant declarations can appear anywhere in the program as long as the declaration precedes the use of the constant. We could use such a constant in our area program in this way

```
const pi : real := 3.14159
var radius: real
put "Enter radius " ..
get radius
put "Area is ", pi * radius ** 2
```

Here is a sample Execution window.

```
Enter radius 5
Area is 78.53975
```

The result is basically the same as before. We give names to constants, such as 3.14159, to help make our programs more understandable. Notice that the prompt here has two dots after it so that the cursor remains on the same line for you to enter the

radius. The prompt has a blank at the end of it so that there will be a space before the radius (before 5).

Here is example of a program that uses a string constant.

```
const prompt : string := "Enter your first name"
var firstName : string
put prompt ..
get firstName
put "Hello ", firstName
```

Placing the text of the prompt in a constant declared at the top of the program allows you to change the text without having to search through the program. This becomes more important when you have large programs and want to use the same prompt several times.

5.8 Assignment of Values to Variables

So far we have just input values into the location of variables. We can also assign a value to a variable in an **assignment statement**. Here is an example program using an assignment statement.

```
const pi : real := 3.14159
var radius: real
var area: real
put "Enter radius " ..
get radius
area := pi * radius ** 2
put "Area is ", area
```

Because *radius* and *area* are both **real** variables they can, if you want, be declared in the same declaration. For example

```
var radius, area: real
```

instead of

```
var radius : real
var area : real
```

The second last statement assigns the value of the arithmetic expression

```
pi * radius ** 2
```

to the variable *area* . Then, in the following **put** statement, the value of *area* is output. This program does exactly the same thing as before except that now it is clear that you are computing the area of the circle.

Assignment statements are used when an intermediate value in a calculation has to be stored in memory, as well as to make the program easier to understand.

We can think of := as a left pointing arrow that moves the value into the variable. This is the same pair of symbols (:=) used when the value is assigned to a constant in a declaration.

Assignment statements can also be used with strings. Here is a program that reverses the order of two entered items.

```
var firstItem, secondItem : string
put "Enter the first item " ..
get firstItem
put "Enter the second item " ..
get secondItem
put "Here are the items reversed: " ..
put secondItem, "  ", firstItem
```

5.9 Understandable Programs

Several times we have mentioned that we want our programs to be understandable. This is so you yourself can understand what is happening in the program even as you create it. You are less likely to make mistakes this way. If you have a written record of a program and come back to it after a day or so it is much easier to see what it does. Also another person, such as a teacher, can quickly read the program and perhaps see what is wrong with it if it is not doing what you expect it to do. Understandability is so important that computer scientists believe

that a program which is not easily understood is virtually worthless even though it may give a correct answer.

One way we have of trying to make programs easy to read is to put each statement on a separate line. This does not matter to the computer. We could, if we liked, run the program all together like this

```
const pi := 3.14159 var radius, area: real put "Enter radius" ..
get radius area := pi * radius ** 2 put "Area is ", area
```

But this is much harder to read. Sometimes a program will have a rather long line and we must start a new line. This is fine provided we do not break the line in the middle of a keyword, or variable name, or a number, or a quoted string. If you must break a quoted string in a **put** statement insert a quote and a comma before the break and quotes to begin the continuation. For example

```
put "Here is a very long line that had to ",
    "be broken"
```

5.10 Comments in Programs

There are a number of ways to make a program more easily understood. We try to choose names for variables which tell the reader precisely what the values to be stored in a variable location are to represent. We choose good variable names.

Another way to make a program understandable is to add comments to it to explain what it does or how it does it if it is at all obscure. Most good programs require only a few comments but they can be helpful.

A comment in the program begins with a % sign and ends with a Return. We could add this comment to our *circle* program

```
% Computes the area of a circle given its radius
```

A comment can be placed anywhere you like in a program. It is ignored by the computer; it is just for the reader's benefit.

Sometimes we want to give a program a name so you can refer to it by name. We will be storing programs as files on the disk memory and files must have a name. A comment placed as the first line of a program can give its file name. We use a standard form which you may use. For example, as a first line in the *CircleArea* program we would have a comment

```
% The "CircleArea" program
```

Here is the *CircleArea* program with comments.

```
% The "CircleArea" program
% Computes the area of a circle given its radius
const pi : real := 3.14159
var radius: real
var area: real
% Ask the user for the radius
put "Enter radius " ..
get radius
% Calculate the circle's area
area := pi * radius ** 2
% Output the result
put "Area is ", area
```

Turing programs stored on disk should have .t added to their names to identify that they are Turing programs. This program would be stored as *CircleArea.t*.

5.11 Exercises

1. There are 2.54 cm in one inch. Write a program to input the length of a desk in inches and output its length in centimeters. Use a constant for the conversion factor. Be sure to prompt for the input and to label the output.
2. Write a program that asks for a person's year of birth and outputs their age in the current year. Write the program to work for whatever the current year happens to be.

3. Write a program that inputs the starting time and finishing time of an automobile trip as well as the distance in kilometers traveled and outputs the average speed in km/hr. The times are to be given in two parts: the hours, then the minutes.
4. Write a program that reads in four numbers and then outputs the four numbers all on one line with commas between the numbers.
5. Write a program to input a name and output a message that greets a name. For example, if the name is ÒSueÓ, then the message might be ÒHello Sue!Ó. Use constants for the greeting.
6. Write a program to calculate and output the product of three numbers entered via the keyboard. Also output the square of the product.
7. Write a program that inputs five full names and outputs the names in reverse order.
8. Write a program that inputs a first name and a last name then outputs this in the form

   last name, first name

   Try to write a version which inputs the two names from a single input line.
9. Experiment with programs where you purposely make syntax errors to see what the error messages are like.
10. Experiment with programs where you purposely input the wrong type of data to see what happens. For example, enter an integer when a string is expected.
11. See what happens when you run this program

```
var age: int
put "Enter age"
get age
age := age + 1
put "age is ", age
```

   How do you interpret the assignment statement

```
age := age + 1
```

Would another variable called *ageNextYear* make the program more understandable?

12. Experiment with adding comments to a program. See if you can add a comment to the end of a line. Can a comment be in the middle of a line?
13. The Prom Committee at your school will sell tickets at $65 per person. Expenses include the cost of the food, the DJ, the hall, the decorations, and the waiting staff. To these expenses, add $100 for miscellaneous expenditures. Write a program to ask for each of the expense totals incurred and then calculate and output to the committee how many tickets they must sell to break even.
14. A student wrote 5 tests. Ask the student for their name and then what each test is marked out of and what mark they received on each test. At the end of the run calculate and output the percentage for each test as well as the average on all five tests also as a percent. When querying the student, address each request with their name. Make sure that output statements include the name of the student.
15. Ask the user for a real number which expresses the area of a figure. Assume that the figure was a circle. Output the radius and then the circumference of the circle. Now assume that the figure was a square. Output the length and width and then the perimeter of the square. Use complete statements in each case.

5.12 Technical Terms

data
memory of computer
variable
declaration of variable
data type
name of variable
reserved word
type of variable
real
int
string
var

get **statement**

prompt
token
token-oriented input
quoted string
syntax error
execution error
run time error
constant
declaration of constant
assignment statement
comment
program name

Chapter 6

Repetition

6.1 Loops

One of the labor saving features of computers is that the same set of instructions can be used over and over with different input data. So far to get repetitious results we had to run our programs over and over. For example, we could run the *CircleArea* program and each time compute the area of a new circle. In Turing there is a way of having repetitions built into the program itself.

Here is the *ManyCircleAreas* program all set to work repetitiously.

```
% The "ManyCircleAreas" program
% Computes the areas of circles
var radius, area : real
const pi := 3.14159
loop
    put "Enter radius " ..
    get radius
    area := pi * radius ** 2
    put "Area is ", area
end loop
```

In this program there is a **loop** statement which starts with the keyword **loop** and ends with **end loop**. The **body** of the loop, which is the part we have indented, will be executed over and over. This means that we can calculate the areas of as many circles as we want. As soon as we enter the radius of a circle in response to the prompt, its area is calculated and output; then the execution of the body of the loop begins again with the output of the prompt. This is an infinite loop in that it will go on forever.

To stop the execution of a program that contains an infinite loop requires intervention on your part. Click the **Stop** button in the Execution window. On the Macintosh, select **Stop** from the **Run** menu. This will stop execution and return you to the Editor

window with the line that was being executed highlighted. Entering illegal input (for example a word when the computer expects a number) will also stop the program.

Notice that we have put comments in this program to give it a title and explain what it does.

6.2 Conditional Loops

The first loop we have shown is an infinite loop and requires your intervention to stop its execution. We will now see a different loop that will stop itself once a particular condition holds. This program stops when you enter a value for the radius which is an acceptable value to input but is an impossible value for a radius. A negative radius is not meaningful so we will use that as a signal to stop.

This signal is called a **sentinel**. Always indicate what the sentinel is so the user knows how to gracefully exit the loop. Immediately after inputting the variable, test to make sure it is not the sentinel. Your **exit when** statement should follow your **get** statement. If you do not test the variable you may be including your sentinel in calculations that it was not meant to affect.

Here is the program.

```
% The "ManyCircleAreas2" program
% Compute circle areas until you enter a negative radius
var radius, area : real
const pi := 3.14159
put "Enter a negative radius to stop execution"
loop
    put "Enter radius " ..
    get radius
    exit when radius < 0
    area := pi * radius ** 2
    put "Area is ", area
end loop
put "That's all folks"
```

This program is like the one containing the infinite loop except for different comments, and the addition of a new prompt and an **exit when** statement in the body of the loop. In the **exit when** statement, there is the **condition** *radius* < 0 which tests to see if the radius is less than zero (that is, negative). When this condition is true the computer leaves the loop and goes to the statement following the **end loop**. It will then output the sign off message.

Here is a sample Execution window.

```
Enter a negative radius to stop execution
Enter radius 5
Area is 78.53975
Enter radius .75
Area is 1.767144
Enter radius –1
That's all folks
```

The signal to stop is given after two actual calculations. If we wanted the signal for ending the loop to be –1 and not just any negative number, we would use this statement

exit when *radius* = –1

6.2.1 Comparisons

Simple conditions (also called **logical**, **Boolean** or true/false conditions) used in **exit when** statements involve comparing two things where the **comparison operator** can be < or =, as we have already seen. The complete set of these operators is

=	equal to
<	less than
>	greater than
not=	not equal to
<=	less than or equal
>=	greater than or equal

The form of a condition is

expression comparison-operator expression

Comparisons of numerical values have the usual mathematical meaning. For example, 6 < 7 is true, 2 + 2 = 4 is true, and 11 <= 16 – 8 is false. In all cases, the comparison (and thus the value of the simple condition) is either true or false. A **Boolean** value is one that can have only the values **true** or **false**.

A problem may arise when comparing real values Avoid using the Ò=Óor equals condition when comparing real numbers. When stored in a computer, real numbers may not be completely accurate. For example a computer cannot store the value one-third exactly. For this reason, it is better to use Ò>=Ó or Ò<=Ó instead of Ò=Ó when doing a comparison.

For example, instead of :

exit when *amount* = 0.1

use

exit when *amount* <= 0.1

or use

exit when *amount* >= 0.1

whichever fits the logic of the program.

6.2.2 Comparing Strings

With strings, the comparison operators take on a slightly different meaning. The meaning of < is that one string comes before the other string alphabetically, provided the strings are all letters of the same case: upper or lower. When digits or special characters are included in the strings the sequence followed cannot be alphabetic since they are not part of the alphabet. The actual sequence is called the **collating sequence** of the code for the characters. Microcomputers use the set of **ASCII characters** which is given in the appendix. Each character has a numerical

value, for example the character "A" is 65, "B" is 66, "C" is 67, and so on. Lower case letters have different values: "a" is 97, "b" is 98, etc. This means that the condition

"A" < "B"

is true and also that

"a" < "B"

is false.

If the first characters of two strings are the same then the comparison of the next two tells the relation between them. For example,

"Bob" > "Bill"

is considered to be true. In other words, "Bob" comes after "Bill". Two strings are not equal to each other unless they are the same length. For example,

"Stop" = "Stop "

is not true because there is a blank at the end of the second string and it is thus five characters long. However,

"Stop" **not** = "Stop "

is true.

6.2.3 An Example Conditional Loop

Here is a program that reads in the marks of a student and then outputs the student's average mark to the nearest integer.

```
% The "ComputeAverage" program
% Compute the average of a series of marks
% Give average to the nearest integer
put "Enter a series of marks"
const sentinel := – 1
put "End with ", sentinel
var mark : int
var count, sum : int := 0
```

```
loop
    get mark
    exit when mark = sentinel
    count := count + 1
    sum := sum + mark
end loop
put "Average mark is ", round (sum / count)
```

Here is an example execution.

```
Enter a series of marks
End with –1
68
74
83
90
–1
Average mark is 79
```

The signal that the series of marks is finished is set to –1 by a constant declaration. This means if you want to change the signal, all you have to do is change that one statement. The two variables *count* and *sum* are declared as **int** and **initialized** to be zero by the := 0 at the end of the declaration. These variables have been initialized to a value in their declarations. The declaration with initialization is equivalent to this.

```
var count, sum: int
count := 0
sum := 0
```

As each mark is read in, *count* is increased by 1 and the mark is added to *sum*. The final average, after the last mark is read in, is computed by dividing the *sum* by the *count*. The value of *sum/count* is a real value which may, by chance, be an integer but, in general, has a fractional part which must be rounded off. To do this we use the predefined Turing function round. If we had wanted just to discard the fractional part, that is **truncate** it, we could use the integer division

```
sum div count
```

The **div** operator is similar to / but it discards any fraction and produces an integer. The value of round (5/3) is 2 whereas 5 **div** 3 is 1.

6.2.4 Another Conditional Loop

This program reads words until the word "stop" is read, then it stops.

```
% The "Obey" program
% Read in a series of words
% until the word "stop" is read
var word : string
put "Enter a series of words, one to a line"
put "If you want to stop say so"
loop
    get word
    exit when word = "stop"
end loop
put "This is the end"
```

Here is a sample Execution window.

```
Enter a series of words, one to a line
If you want to stop say so
dog
bites
man
bites
dog
stop
This is the end
```

Although these examples do not show it, a loop can contain several **exit** statements.

6.3 Counted Loops

So far we have had two kinds of loops: the infinite loop and the conditional loop. There is another kind of loop that you can use, that is, the **counted loop**. These are loops that you want repeated a fixed number of times.

Here is a program that reads 5 student marks and computes the average mark.

```
% The "ComputeAverages" program
% Reads marks and computes average
var mark : int
var sum : int := 0
put "Enter marks"
for count : 1 .. 5
    put count
    get mark
    sum := sum + mark
end for
put "Average is ", round (sum / 5)
```

This program is similar to the *ComputeAverage* program which used a conditional loop. However this time we know that there will be exactly five marks to be averaged, and we use a counted loop. The counted loop begins with the keyword **for**. Following this is *count*, the name of the **index** or **counter** for the loop. The index of a counted loop is a variable of type **int** with very special properties. It cannot be declared. Its declaration is implied by its appearance in a **for** statement. In this program, it is assigned the value 1 when the **for** loop is first entered and is automatically incremented by 1 on each repetition. It has a value 2 the second time the loop is executed, 3 on the next, and eventually 5 on the last. The **range** of its values is given by 1 .. 5. The starting value of the index is given first, then two dots followed by the finishing value of the index. You are not allowed to assign a value to this counter yourself to interfere with the automatic action of the **for** loop, but you may use its value in

arithmetic expressions and output it. Outside the loop the counter's value is *not* available. We say it is only available within the **scope** of the loop. We could *not*, for example, have written the statement

put "Average is ", round (*sum* / *count*)

outside the loop because *count* is not available below **end for**. The **put** statement in the body of the loop is legal since the value of *count* is available there.

Here is a possible execution of the program.

```
Enter 5 marks
1
52
2
76
3
85
4
71
5
81
Average is 73
```

Here the 5 marks you enter are 52 76 85 71 81. When you press Return after each mark is entered, it is read by the **get** and the value of *count* is output on the next line. You can type all the marks on a single line followed by Return and then the window will look like this

```
Enter 5 marks
1
52  76  85  71  81
2
3
4
5
Average is 73
```

The **get** statement does not read any of the marks until you press Return.

It is good practice in programming to avoid using numbers, like 5, directly in a **for** loop. It is preferable to define a constant, say *numMarks*, with a value 5 and then use this instead. The program would then have these changes

```
. . .
const numMarks := 5
put "Enter ", numMarks," marks"
for i: 1 .. numMarks
  . . .
put "Average is ", round (sum / numMarks)
```

This means that if you wanted to modify the program to read six marks instead only one line need be changed, namely,

```
const numMarks := 6
```

Here is another program that allows the user to enter both a start and stop value for the counted loop.

```
var start, stop : int
put "Enter the initial and final values for the loop: " ..
get start, stop
for i : start .. stop
    put i
end for
```

6.3.1 Counting in Multiples

The loops in the previous section all increased the index by one each time through the loop. It is also possible to increase the index by any value. The **by** clause is used to increase the value by a number greater than one.

Here is an example of a loop that outputs even numbers from 2 to 10 using the **by** clause.

```
for count : 2 .. 10 by 2
    put count
end for
```

The index of the **for** loop starts at 2. At each repetition, the index increases by 2. If the index is greater than the second range value (in this case 10), the loop exits.

It is never necessary for the index value to equal the second range value. For example the loop

```
for count : 1 .. 10 by 5
    put count
end for
```

outputs the values 1 and 6.

6.3.2 Indenting the Body of Loops

Since we are trying to make programs easy to read it helps to indent the body of the loop several spaces more than the beginning and end of the loop. This is also called **paragraphing** the program. It makes the program easier to understand because you can actually see the **scope** of the loop.

The Turing Environment provides an **Indent** command from the Edit menu that automatically indents a program in the standard form. It also performs syntax coloring.

6.3.3 Loops that Count Backwards

The counted loop we have shown counts by ones, starting at the first value of the range up to and including the last value. If you want to count backwards by ones you can do it by this statement:

```
for decreasing count: 5 .. 1
    ... body of loop ...
end for
```

Here the index *count* will start at 5 and go down by 1 until it reaches its final value of 1.

A common error made when using a decreasing for loop is to reverse the order of the upper and lower limits of the loop. Be

careful because this will not give a syntax error and the problem may go unnoticed. For example:

```
for decreasing count : 1 .. 5
    ... body of loop ...
end for
```

The body of the loop will never be executed because 1 is already less than 5 and cannot be further decreased. Execution goes from the top of the **for** loop and jumps to the end of the loop. Similarly, the following program will not give an error but will do nothing since 5 is already greater than 3.

```
for count : 5 .. 3
    ... body of loop ...
end for
```

6.3.4 Counted Loop Examples

Here are some examples of counted loops and the output that they produce.

```
for count : 2 .. 10
    put count
end for
```

Outputs 2, 3, 4, 5, 6, 7, 8, 9, 10. The loop starts with 2 and counts up ending with 10.

```
for count : –4 .. 6
    put count
end for
```

Outputs –4, –3, –2, –1, 0, 1, 2, 3, 4, 5, 6. The loop starts with –4 and counts upward ending with 6.

```
for count : –4 .. –1
    put count
end for
```

Outputs –4, –3, –2, –1. The loop starts with –4 and counts upward ending with –1. The fact that the first and second value of

the range are both negative does not change how the loop operates.

```
for count : 2 .. 2
    put count
end for
```

Outputs 2. The loop starts with 2 and then exits the loop.

```
for count : 4 .. 1
    put count
end for
```

Outputs nothing. The body of the loop is never executed because the first value of the range is higher than the second value.

```
for count : 1 .. 20 by 5
    put count
end for
```

Outputs 1, 6, 11, 15. The loop exits when count is increased to 21 which is greater than 20.

```
for count : 5 .. 15 by 20
    put count
end for
```

Outputs 5. The loop exits when count is increased to 25 which is greater than 15.

Loops can also count downward by using the keyword **decreasing** in the **for** loop header.

```
for decreasing count : 7 .. 3
    put count
end for
```

Outputs 7, 6, 5, 4, 3. The loop starts with 7 and counts downward ending with 3.

```
for decreasing count : 1 .. 10
    put count
end for
```

Outputs nothing. The body of the loop is never executed because the first value of the range in the decreasing loop is lower than the second value.

```
for decreasing count : 17 .. 7 by 3
    put count
end for
```

Outputs 17, 14, 11, 8. The loop starts with 17 and counts downward by 3 ending when the index reaches 5 (which is lower than 7).

6.3.5 Counted Loops with Exits

The counted loop can have an **exit when** condition inside its body and this is often useful. Here is a program that reads words until it has read ten words or has read the word *stop*.

```
% The "ReadWords" program
% Read words until ten have been read
% or the word "stop" is read
var word : string
put "Enter ten words one to a line, or finish with 'stop' "
put ""
for count : 1 .. 10
    get word
    exit when word = "stop"
    put word
end for
```

Here is a possible execution.

```
Enter ten words one to a line, or finish with 'stop'

one
one
two
two
stop
```

Notice that after the word *stop* is read you exit from the loop even though the loop has not been executed ten times and the word *stop* is not output by the **put** as the other two are. A blank line is left after the prompt by the **put** with a **empty** string, that is, quotes with no characters inside.

6.4 Random Exit from Loop

In Turing random numbers can be generated using predefined **procedures**, randint for generating random integers and rand for generating random real numbers. If we use the statement

```
rand (number)
```

in the program and *number* has been declared as a real variable the value of *number* will be a real number randomly chosen between 0 and 1 exclusive. These random numbers will be uniformly distributed which means, for instance, that half of the time the number would be less or equal to .5 and half the time greater. Here is a program that counts from 1 to 10 but has a 25% chance of stopping after each count.

```
% The "CountUp" program
% Counts from 1 to 10 but has 25 percent chance of stopping
var number : real
for count : 1 .. 10
    rand (number)
    exit when number <= .25
    put count
end for
```

To use the randint procedure for producing pseudo-random integers we give a statement such as

```
randint (count, 1, 10)
```

This will assign to the variable *count*, which must have been declared as an integer variable, integer values uniformly distributed between 1 and 10 inclusively.

Here is a program that uses randint to control the number of times a loop is executed.

```
% The "CountDown" program
% Counts down from 10 to 1
% but stops at a random point
var last : int
randint (last, 1, 10)
for decreasing count : 10 .. last
    put count : 2 ..
end for
```

In this program if the value of *last* is assigned by randint to be 6 then the output is

```
10 9 8 7 6
```

Five numbers are output.

6.5 Compound Conditions

Sometimes an exit condition is more complicated than just a comparison. We can form **compound conditions** where two simple conditions are combined using the **logical** or **Boolean** operators called **and** and **or**. If two simple conditions are combined using **and** then both conditions must be true before the compound condition is true. For example, the compound condition

6 < 7 **and** 5 = 3 + 2

is true because both simple conditions are true.

When two simple conditions are combined using **or,** the compound condition is true if either one (or both) of the two is true. For example,

```
9 < 8 or 6 > 2
```

is true since the second simple condition is true even though the first is false.

There is a Boolean operator which does not connect two conditions but acts on only one. It is the **not** operator. For example,

```
not A > B
```

which is the same as

```
not (A > B)
```

is true when the value of *A* is less than or equal to the value of *B* . We can usually write conditions in such a way as to avoid using the **not** operator. For example, the condition we just had could be written as

```
A <= B
```

Conditions can be compounded several times by using **and** and **or** several times. When writing very complicated compound conditions it is best to use parentheses to make your meaning clear rather than depending on the precedence rules for the Boolean operators. These rules specify that **and** is done before **or**.

6.6 Exercises

1. Write a program containing an infinite loop which outputs the series of integers starting at 5 and going up by 5s. Revise this program to output the integers starting at 5 and decreasing by 10s.
2. Write a program that endlessly tells you to "Have a good day". Try stopping execution. Change it so that it is a program to wish you a good day only six times.

3. Write a program that reads words entered one to a line and counts how many words have been entered before you give the signal word "end" to stop execution of the program. Experiment to determine what happens if you put several words on a line as you enter them.
4. A series of marks is to be entered and averaged. Before you enter the series you are to have the program ask you how many marks there are in the series then read it in. Test your program to see that it works for series of different lengths, say four marks or six marks.
5. Write a program that announces at each repetition of a loop the number of times it has executed the loop. Have it stop at each execution with the message

   ```
   Type 'more' to continue
   ```

   A sample Execution window might be

   ```
   Loop execution number 1
   Type 'more' to continue
   more
   Loop execution number 2
   Type 'more' to continue
   more
   Loop execution number 3
   Type 'more' to continue
   stop
   ```

6.a Write a program to output a table of values of the integers starting at 1 and their squares. Label the table at the top of the columns. For example, your output might look like this

   ```
   Number        Square
      1          1
      2          4
      3          9
      4          16
      5          25
   ```

   Try to format the output so that it looks attractive. What happens as the numbers get larger and larger? Change the

program to output the first 100 integers rather than attempting to go on forever.

6.b Modify your program so that in one **for** loop you output the following

```
Num      Square    Num      Square     Num     Square     Num
      Square
1     1         21     441          41     1681      61     3721
2     4         22     484          42     1764      62     3864
```

continue through to

```
20    400          40     1600      60     3600      80     6400
```

7. Write a program using a loop counting backwards. Output the index of the loop on each execution so the output is the same as the count down for a rocket launch. Arrange the output so that it is all on one line like this

```
5  4  3  2  1
```

8. Write a program to output a backwards count by 5s from 100 down to 5. Modify it so that you count from 100 down to 50. Modify it so that before you start the count you can input a number between 100 and 50 so that the program will stop when the count would be less than the number input. For example the execution might be like this:

```
What number do I stop at? 82
Stop when count less than 82
100
 95
 90
 85
```

9. Write a program to find the sum of a number of terms of the infinite series

```
1 + x + x**2 + x**3 + x**4 + ...
```

where the number of terms *n* to be evaluated and the value of *x* are input before the summation begins. Experiment with different values of *n* and *x*.

10. Write a program to compute the bank balance at the end of each year for 10 years resulting from an initial deposit of $1000 and an annual interest rate of 6%. Output for each year end the number of the year, the initial balance, the interest for the year, and the balance at the end of the year.

11.a A homeowner takes out a mortgage for $120,000 at 7.75% per year. At the end of each year an amount of $24,000 is paid. Write a program to show how the mortgage is paid off, year by year, until nothing is owing.

11.b Assume that each month in the year has 30.5 days in it. Give the number of the month and the day in the month in which the mortgage is paid. (i.e. month #3, day #20)

12. Write a program to simulate the playing of a simple dice game (played with one die). Roll the die to get a value from 1 to 6. This we will call your point. Now keep rolling until you get the same value (your point) again and see how many rolls it takes. Program it so you can play this game repeatedly.

13. Ask the user for an integer between 1 and 50. Output all the factors of that integer. Next, modify the program so that it outputs the factors of each integer up to the value of the integer input by the user.

14. Ask the user for an integer. Output the number of digits in the integer. Then output the sum of the digits.(i.e. 1234 has 4 digits and their sum is 10).

15. Write a program to keep inputting integers until a perfect square (for example 64) between 40 and 100 is entered. (This is a difficult one!)

16. Write a program to generate 10 random real numbers between:

    a. 4 and 5

    b. 0 and 10

    c. 20 and 30

    d. *x* and *y* where *x* and *y* are integer inputs.

6.7 Technical Terms

repetition
loop **statement**
end loop
body of loop
stopping execution (Control-Break)
conditional loop
condition
exit when **statement**
stopping signal
comparison operators (<, >, =, >=, <=, not=)
collating sequence
ASCII character code
round function
initialization of variable value in declaration
truncation of real number
div **operator**
counted loop
for **loop**
end for
index or counter of loop
range of index
paragraphing program
scope of loop
backwards counted loop
decreasing
multiple exits from loop
random number
rand
randint
pseudo-random sequence
procedure
compound condition
empty string
logical operator
Boolean operator

and
or
not
true
false

Chapter 7

Character Graphics

7.1 Character Locations in the Execution Window

The output of Turing programs so far has been limited to a display of lines of characters in the Execution window. These characters have been values of expressions which are of real, integer, or string type. The output on the lines followed in sequence, one line after the next. To produce successful graphics we must be able to output characters anywhere in the window, in any order that we choose. We will be outputting one character at a time at a location in the window which is selected as the location of the cursor before the output instruction is executed.

The Execution window can have characters in any one of 25 lines (or rows) and on each line in any one of 80 columns. To output a character in a particular location in the Execution window we first place the cursor at that location by a locate predefined procedure in the form

```
locate (row, column)
```

The *row* can be any integer between 1 and 25 inclusive, and the *column* any integer between 1 and 80 inclusive. To place the letter T approximately at the center of the Execution window we would give the instructions

```
locate (13, 40)
% 13th row down, 40th column across
put "T" ..
```

Notice that there are two dots in the **put** instruction. Without the two dots the **put** instruction assumes that the line is complete and would make the rest of the output line blank. To prevent this happening we use the two dots which indicate that the cursor should be left where it was. If we give the instructions:

```
locate (13, 40)
```

```
put "T" ..
put "U" ..
```

the character U would appear after the T on the line in the location (13, 41).

The following program writes Fred in the four corners and in the middle of the Execution window.

```
locate  (1, 1)   % Not necessary.
put "Fred" ..
locate (1, 77)
put  "Fred" ..
locate (13, 38)
put "Fred" ..
locate (25, 1)
put "Fred" ..
locate  (25, 77)
put "Fred" ..
```

At the beginning of the program the cursor is always located at (1, 1) so it is not really necessary to include the first locate. However, if the program segment is used in another program, the cursor might not be located at (1, 1) so it is helpful to include the locate.

The .. after each **put** statement serves two purposes. First, it stops the rest of the line from being erased and second, it stops the window from scrolling when there is output on the last line. If you removed the .. and ran this program again, the window would scroll, causing the Freds at the top of the window to disappear and the two Freds at the bottom to appear on different lines.

7.2 Creating a Graphical Pattern with Characters

Now that you know how to output a character in the Execution window in any position you can use this facility to draw simple pictures. To begin a new graphic we clear the window so that we

are not drawing on top of some other output. To do this we use the procedure cls. The instruction

```
cls
```

will clear the window and then place the cursor at the location (1, 1), which is the top left-hand corner. Here is a program which draws a line of asterisks down the window from top to bottom in the column you specify.

```
% The "DrawVerticalLine" program
% Draws a vertical line of asterisks
var column : int
put "Choose a column for line ",
    "between 1 and 80 inclusive"
get column
cls
for row : 1 .. 25
    locate (row, column)
    put "*" ..
end for
```

Notice that after the prompt appears and the *column* value has been entered the window is cleared, using cls, so that the prompt disappears and does not interfere with the appearance of the asterisks.

7.2.1 Interactive Graphics

Suppose you wanted to have a graphics program with which you were going to interact. For example, you might want to draw a picture a little at a time. You would need to split the Execution window into two parts: part for the graphic and part for the prompting message and the input. We will call each part a **subwindow.** The graphics will be in the top part of the window from row 1 to row 20 and the prompting message and your input will be at the bottom in lines 23 and 24. The two subwindows will be separated by a line of minus signs in row 21.

Here is the program.

```
% The "DrawVerticalLines" program
% Draw a series of vertical lines, one at a time
cls
var column : int
% Draw line of minus signs between the two subwindows
locate (21, 1)
for count : 1 .. 80
    put "-" ..
end for
% Could instead use put repeat ("-", 80)
loop
    % Move cursor into prompt subwindow
    locate (23, 1)
    put " "      % Clear the line
    locate (23, 1)
    put "Choose a column for the line between 1 and 80, end with -1: "..
    get column
    exit when column = – 1
    for row : 1 .. 20
        locate (row, column)
        put "*" ..
    end for
end loop
```

7.2.2 Diagonal Lines and Patterns

Here is a program to draw a diagonal line in the Execution window starting at the point (1, 1).

```
% The "DrawDiagonalLine" program
% Draw a diagonal line in window
% starting in row 1, column 1
var row, column := 1
cls
loop
    % Stop when diagonal touches bottom of window
    exit when row > 25
```

```
    locate (row, column)
    put "*" ..
    % Move down diagonal
    row := row + 1
    column := column + 1
end loop
```

A more interesting program that is interactive can be obtained by letting the user choose the starting point for the diagonal. If we did this we would have to worry about the diagonal hitting the side of the window as well as the bottom and the exit condition would have to be

```
exit when row > 25 or column > 80
```

A still more interesting program is obtained if we let the diagonal "bounce" off the edge of the window and be reflected just as a hockey puck bounces off the boards. If it bounces off the bottom we must start decreasing the row number but keep the column number increasing. Here is the complete program for a bouncing (or reflected) motion. The reflection is not exactly accurate because the column change of one character width and row change of one line spacing are not exactly equal for the winsow.

```
% The "Bounce" program
% Simulates the action of a bouncing puck
var row, column : int
put "Enter starting row from 2-25 " ..
get row
put "Enter starting column from 2-80 " ..
get column
var rowChange, columnChange := 1
cls
loop
    % The next six lines are a Selection construct
    % See the Selection chapter if you do not
    % understand them
    if row = 25 or row = 1 then
        rowChange := – rowChange
```

```
    end if
    if column = 80 or column = 1 then
        columnChange := – columnChange
    end if
    locate (row, column)
    put "*" ..
    row := row + rowChange
    column := column + columnChange
end loop
```

If the drawing is formed too rapidly for you to enjoy the bouncing you can slow it down by including a time-wasting **for** loop right after the **put** statement that draws the asterisk.

Here is such a time wasting **for** loop.

```
var garbage : int
for i : 1 .. 10
    % Perform a time wasting calculation
    garbage := 237 * 26
end for
```

You can also use the built in procedure delay in order to slow execution down. It has the form

delay (*duration*)

where the duration is in milliseconds (thousandths of a second).

7.3 Drawing in Color

Graphics can be made more interesting by displaying the results in color. To choose a color for a character to be displayed use the color predefined procedure in the form

color (*chosenColor*)

The *chosenColor* can be one of Turing's predefined colors or a number. These colors are white, blue, green, cyan, red, magenta, brown, black, gray, brightblue, brightgreen, brightcyan, brightred, brightmagenta, yellow, and darkgray. If using a number, the number

can range from 0 to the maximum color number available in Turing, usually 255.

Note: in Turing you may spell color as colour and gray as grey if you prefer.

Here is a program that displays a box outlined by randomly colored asterisks that blink.

```
% The "Marquee" program.
% Draws a box outlined by randomly colored asterisks.
var depth, width : int
put "Enter the width of the box (less than 60)"
get width
put "Enter the depth of the box (less than 20)"
get depth
cls

% Arrange to center box in window.
const topRow := (25 – depth) div 2
const leftColumn := (80 – width) div 2
var colorNo : int

% Draw top of box
locate (topRow, leftColumn)
for count : 1 .. width
    % Choose a random color
    randint (colorNo, 0, 15)
    % Set the color.
    color (colorNo)
    put "*" ..
end for

% Draw bottom of box
locate (topRow + depth – 1, leftColumn)
for count : 1 .. width
    randint (colorNo, 0, 15)
    color (colorNo)
    put "*" ..
end for
```

```
% Draw left side of box
const sideTop := topRow + 1
const sideBottom := topRow + depth – 2
for row : sideTop .. sideBottom
    randint (colorNo, 0, 15)
    color (colorNo)
    locate (row, leftColumn)
    put "*" ..
end for

% Draw right side of box
const rightColumn := leftColumn + width – 1
for row : sideTop .. sideBottom
    randint (colorNo, 0, 15)
    color (colorNo)
    locate (row, rightColumn)
    put "*" ..
end for
```

7.4 Background Color

As well as controlling the color of the displayed characters, the background for each character can be set to a variety of colors by the *colorback* predefined procedure. It has the form

colorback (*colorNumber*)

If a blank is output, the background color is all that you see.

Here is a program to color the entire window blue.

```
% The "DrawSky" program
% Color the whole window blue
colorback (1)
for row : 1 .. 25
    for column : 1 .. 80
        locate (row, column)
        % Output a blank
        put " " ..
```

```
    end for
end for
```

Notice that the window is colored a character at a time.

Here is a program that colors the window green in a random fashion.

```
% The "LeafFall" program
% Color parts of the window randomly
cls
% Set to color background green
colorback (green)
var row, column : int
loop
    randint (row, 1, 25)
    randint (column, 1, 80)
    locate (row, column)
    put " " ..
end loop
```

Run the program to see how long it takes for the leaves to completely cover the window.

7.5 Hiding the Cursor

Sometimes the appearance of a graphical pattern can be confused by the display of the cursor which always appears following the last output or echoed input. This is particularly true in animated graphics.

To hide the cursor we would use the predefined procedure setscreen in the form setscreen ("nocursor"*).* To have the cursor show again use setscreen ("cursor"*).*

7.6 Animation with Graphics

The illusion of continuous motion can be created by having a series of still pictures in which an object is shown in slightly different positions from one to the next. We will color the window green then move a magenta asterisk around at random starting it at the center. When it reaches the edge of the window we will start it back at the center again.

```
% The "BrownianMotion" program
% Moves an asterisk around on
% the window like a smoke particle
% This is known as Brownian motion
% Color window green
colorback (green)
for row : 1 .. 25
    put repeat (" ", 80) ..
end for

% Set character color to magenta
color (magenta)
setscreen ("nocursor") % Hide cursor
loop
    % Start at center
    var row := 13
    var column := 40
    loop
        % Start over when asterisk goes off the window
        exit when column < 1 or column > 80
            or row < 1 or row > 25
        locate (row, column)
        put "*" ..
        delay (100)
        % Erase asterisk in old location by
        % outputting a blank there
        locate (row, column)
        put " " ..
        % Compute new value for row
        randint (row, row – 1, row + 1)
```

```
        % Compute new value for column
        randint (column, column – 1, column + 1)
    end loop
    play (">C<")     % Sound a note
    % This is explained in the Music chapter
end loop
```

7.7 Controlling the Speed of Animation

In the *BrownianMotion* program the particle moves about very rapidly. It is possible that you might want to introduce a delay between the **put** instruction which outputs the asterisk in a given place and the **put** instruction which erases it. You can do this as you have seen by inserting between the two **put** instructions a call to the delay predefined procedure. The object of the delay procedure is to just waste time. Nothing is actually being accomplished other than that. You can control the length of the delay by setting the duration to a variable like *duration* where *duration* is an integer value which is a measure of how much time you want to waste. Here is the time wasting call to delay.

```
delay (duration)
```

The statements for controlling the speed can be inserted into the program right after the first 5 lines of comments. These are

```
var duration: int
put "Choose delay time for animation: " ..
get duration
```

The choice of a good value for *duration* must be found by experimenting with various values on your computer to get the speed of animation you want.

7.8 Pausing for User Input

You can also instruct a program to pause until the user presses a key. This is done using the statements

```
var reply : string (1)
getch (reply)
```

The getch procedure causes Turing to wait until a key is pressed and the assigns the key to *reply*. The *reply* is a one-character string and it is used in the getch (get a character) command. The *reply* variable can only be declared once. Here is a program segment that prompts the user to hit a key to resume the execution.

```
put "Press any key to continue the program execution. " ..
getch (reply)
```

Once the user presses a key the execution will resume. The getch (*reply*) instruction is useful because the user indicates when they are ready to continue, whereas a delay waits for a period of time set by the programmer.

To stop the character the user presses from being output to the window, use

```
setscreen ("noecho")
```

To have the user input displayed on the window, use

```
setscreen ("echo")
```

7.9 Exercises

1. Write a program to plot a horizontal line of minus signs in any row of the window that you specify.
2. Make the program of question 1 interactive so that you can plot as many horizontal lines as you want, one after the other.

3. Write a program to fill the Execution window with diagonal lines of asterisks separated by diagonal blank lines. Note that this can be accomplished easily by outputting lines of alternating asterisks and blanks using, in turn, either

   ```
   put repeat (" *", 40)
   ```

   or

   ```
   put repeat ("* ", 40)
   ```

   (Note the space before and after the asterisk.) Try doing this in color with light green asterisks on a background of blue. Next, make the asterisks blink.
4. Write a program to draw a funny face in the window with its center at any point you specify. Make the face of a size 9 characters by 9 characters. Arrange so that the color of the face and the background can be input at the same time as you specify its location in the Execution window. Refuse to plot the face if it goes outside the window. You will need a selection construct for this. See the next chapter for how to draw a face in pixel graphics.
5. Write a program to plot a red box on a blue background. To do this wash the whole window with blue then change the background color to red and plot a box full of blanks.
6. Write a program to output a box which is tilted in the window so that its top and bottom follow diagonal lines like those in question 3. How would you fill the box with a color different from the background.
7. By modifying the *BrownianMotion* program, arrange that a picture of a funny face is moved randomly around the window.
8. Modify the *BrownianMotion* program so that you do not erase the image of the particle each time but leave it as a trail when it moves. Change the color of the trail randomly each time the particle starts at the center. Use a black background.
9. Write a program that outputs a cat's face at the center of the window, erases it when you type the letter *e*, then plots it again when you type the letter *p*.

10. Modify the *Bounce* program so that the bouncing asterisk changes color randomly on each bounce.
11. There are a number of characters available in the ASCII code of the computer besides the ones given in the appendix. You can explore these with this program

```
% The "CharValues" program
% See what characters are available
var value: int
loop
    put "Enter a character value, 1 to 255 inclusive " ..
    get value
    put "Character whose value is ", value, " is ", chr (value)
end loop
```

Try to see what different characters you get. One of the characters will produce a beep rather than appearing in the window. What is its value? What happens when you try the value 0?

12. Display the letters of the alphabet in different colors on one line. The letters should appear one at a time and each letter should appear slowly and then disappear when the next letter appears (like the floors lighting up on an elevator).
13. Ask the user for the upper left row and column and the bottom right row and column to represent the top left and bottom right corners of a rectangle. Ask the user which character they would like the rectangle to be created with. Create the outline of the rectangle from the top-left corner to the bottom-right corner given by the user in the character they indicated.
14. Ask the user for the upper left row and column and the bottom right row and column to represent the top left and bottom right corners of a rectangle. Ask the user for the number of the color they wish to use. Ask the user for the character they wish to use. Create a filled in rectangle in the color and character indicated by the user going from the top-left to the bottom-right co-ordinates.

7.10 Technical Terms

location of character in the Execution window
locate
graphics
cls
interactive graphics
window
color
background color
colorback
animation
delay

Chapter 8

Pixel Graphics

8.1 Pixel Positions in the Execution Window

So far we have plotted graphics using characters. In this part we will plot graphics using dots or **pixels** in much the same way as a television picture is plotted. With characters there were 2000 positions in the window where a character could be placed (25 rows with 80 character positions in each row). With pixel graphics there are many more positions where dots can be plotted, over 150,000 in a standard output window (300 rows with 640 pixel positions in each row). Pixel graphics have a much **higher resolution**.

Pixel positions are given in terms of **coordinates**. The *x*-coordinate is the position number along a line starting from the left-hand side of the window. The first position has number 0; the last position depends on the type of graphics. For a standard output window it is 639. The *y*-coordinate is the row number starting at the bottom of the window at 0 and numbering upward. In a standard output window the last row number is 299. The *x*- and *y*-coordinates correspond to the usual way of plotting graphs mathematically. The **origin** of the coordinates is in the lower left-hand corner of the window.

Turing Execution windows can be one of two types: text windows or graphics windows. Text windows only support the **put** and **get** commands. You cannot use the locate or graphics commands in a text window. All output sent to a text Execution window can be viewed using the window's scroll bar. Saving the text Execution window produces a text file with all the output.

Graphics windows allow all available output commands but any output that scrolls off the top of the screen is lost. Saving a graphics Execution window creates a graphics file.

Here are the commands to set the window to text or graphics.

```
setscreen ("text")
setscreen ("graphics")
```

They must appear before any output.

8.2 Plotting Dots in the Execution Window

To place a dot in the window at a point whose coordinates are (*x, y*) you use the statement

drawdot (*x*, *y*, *c*)

This calls the predefined Turing procedure drawdot. If the values of *x* or *y* are outside the allowed range, the dot is not plotted.

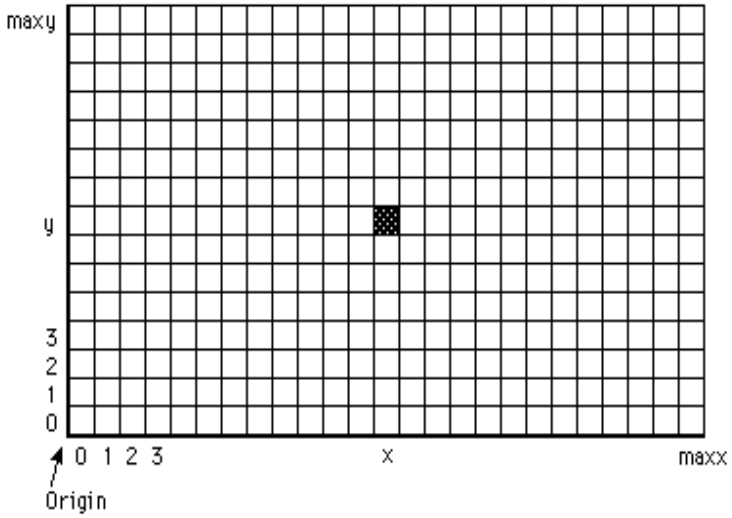


Figure 8.1 Pixel Locations

The *c* is an integer (or integer variable) that sets the color of the dot to be plotted. In graphics a zero value of *c* gives the same color as the background, normally white. Color values range from 0 up to 255, although it is preferable to use the predefined names for colors.

Because different computers may use differently sized output windows, rather than explicitly placing the size of the Execution window in a program, Turing allows you to use maxx and maxy.

The maxx predefined function represents the maximum *x*-coordinate available in the window and maxy represents the maximum *y*-coordinate available in the window. By using maxx and maxy, programs are resolution independent and will work regardless of Execution window size.

Here is a program like the *LeafFall* program in the section 7.4 on character graphics. Execution windows in Turing are normally in pixel graphics mode when the program begins. However, it is possible to set the Turing environment to start the Execution window in text mode. In text mode, output that scrolls off the end of the Execution window can be viewed using the scroll bars.

To set the screen to the pixel graphics mode, we use the statement

```
setscreen ("graphics")
```

This instruction is not absolutely necessary since a call to any pixel graphics procedure, such as drawdot, will set the Execution window to pixel graphics mode even if it was in text mode.

```
% The "Confetti" program
% Color pixels in the window randomly
setscreen ("graphics")
var x, y, c : int
loop
    randint (x, 0, maxx)
    randint (y, 0, maxy)
    randint (c, 1, 15)
    drawdot (x, y, c)
end loop
```

8.3 Changing the Execution Window Size

By default, Turing creates an Execution window that 25 rows by 80 columns. It is possible to change the size of the Execution window. This is done with the instruction

```
setscreen ("graphics:<width>;<height>")
```

where <width> and <height> are the desired width and the height in pixels of the window. For example, to set the Execution window to be 300 pixels wide by 500 pixels tall, the statement

```
setscreen ("graphics:300;500")
```

would be used. Note that there is a colon after the ÒgraphicsÓ and a semicolon between the width and the height. The window can also be sized to be the largest possible window that will fit on the screen. This can be set with the instruction

```
setscreen ("graphics:max;max")
```

An Execution window's size can also be set in terms of the number of rows and columns using the instruction

```
setscreen ("screen:<rows>;<columns>")
```

With this, the Execution window is in graphics mode and <rows> and <columns> are the desired number of rows and columns in the window.

To determine the maximum color number available, a program can use the maxcolor predefined function, which is gives you the maximum allowable color number.

When you change the size of a window, the values returned by maxx and maxy will also change. After the statement

```
setscreen ("graphics:300;500")
```

maxx would give 299 and maxy would give 499.

For example, a modification of the *Confetti* program to produce output in a smaller window could be written this way.

```
% The "Wedding" program
% Gives randomly colored dots with greater variety of
% color and higher resolution than CGA graphics
setscreen ("graphics:200;200")
var x, y, c : int
loop
    randint (x, 0, maxx)
    randint (y, 0, maxy)
    randint (c, 0, maxcolor)
    drawdot (x, y, c)
end loop
```

8.4 Drawing Lines

A line can be drawn from a point in the window with coordinates (*x1, y1*) to a point with coordinates (*x2, y2*) in color *c*

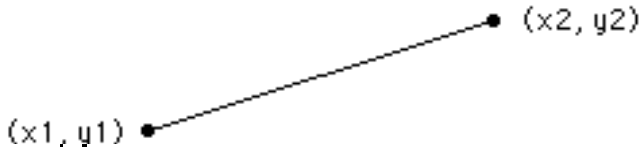


Figure 8.2 Drawing a Line with drawline

with the statement

drawline (*x1*, *y1*, *x2*, *y2*, *c*)

Here is a program that draws lines from the center of the window to points chosen randomly in random colors.

```
% The "StarLight" program
% Draws lines from center to random points
setscreen ("graphics")
var x, y, c : int
const centerx := maxx div 2
const centery := maxy div 2
```

```
loop
    randint (x, 0, maxx)
    randint (y, 0, maxy)
    randint (c, 0, maxcolor)
    drawline (centerx, centery, x, y, c)
end loop
```

Here is a program that draws a rectangle, or box, of width *w* and height *h* with its lower-left corner at (*x, y*).

The four drawline statements in the program could be replaced by the single statement

```
drawbox (x, y, x + w, y + h, c)
```

where (*x, y*) is the lower left corner and (*x + w, y + h*) the upper right corner of the box.

```
% The "DrawBox" program
% Draws a box of width, height,
% position, and color that you specify
var w, h, x, y, c : int
put "Enter width of box in pixels: " ..
get w
put "Enter height of box in pixels: " ..
get h
put "Enter x-coordinate of lower-left corner: " ..
get x
put "Enter y-coordinate of lower-left corner: " ..
get y
put "Enter color number: " ..
get c
setscreen ("graphics")
drawline (x, y, x + w, y, c)                % Base of box
drawline (x + w, y, x + w, y + h, c)    % Right side
drawline (x + w, y + h, x, y + h, c)     % Top
drawline (x, y + h, x, y, c)                % Left side
drawfill (x + 1, y + 1, c, c)               % Fill the box
```

The last statement of this program will fill the whole rectangle with the color number *c*. The form of this predefined procedure is

drawfill (*xinside*, *yinside*, *fillcolor*, *bordercolor*)

The area to be colored must contain the point (*xinside, yinside*) and be surrounded by the *bordercolor*. In this program the border color and the fill color are the same.

The four drawline statements and the drawfill statement could be replaced by the single statement

drawfillbox (*x*, *y*, *x* + *w*, *y* + *h*, *c*)

where (*x, y*) is the lower-left corner and (*x* + *w, y* + *h*) the upper-right corner of the box. The instruction drawfillbox draws a box and then fills it in with color *c*.

8.5 Drawing Circles and Ellipses

To draw an oval whose center is at (*x, y*), whose half-width is *xRadius*, and whose half-height is *yRadius* in color *c*

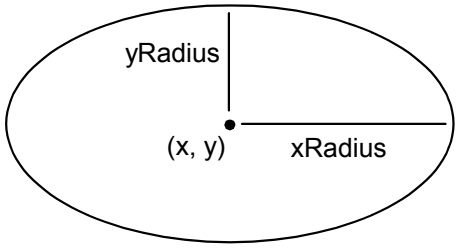


Figure 8.3 Drawing an Oval with drawoval

we use the statement

drawoval (*x*, *y*, *xRadius*, *yRadius*, *c*)

The resulting curve is an **ellipse**. To produce a circle we use equal values for *xRadius* and *yRadius.* Here is a program that draws a series of magenta circles radiating from the center of the window until the circles touch the boundary of the window.

```
% The "Pow" program
```

```
% Plots magenta circles of ever increasing radius centered
% in window until the edge of the window is reached
setscreen ("graphics")
const centerx := maxx div 2
const centery := maxy div 2
for radius : 1 .. centery
    drawoval (centerx, centery, radius, radius, magenta)
end for
```

In Turing, the predefined procedure drawfilloval draws a filled oval without the necessity of using drawfill. It has the form

drawfilloval (*x*, *y*, *xRadius*, *yRadius*, *c*)

and causes an oval to be drawn and filled with color *c*.

8.6 Animation

The effect of animation can be achieved with the *Pow* program by erasing each circle after a delay before plotting the next circle. To erase a circle you can re-plot it using the background color, number 0. The predefined procedure delay can be called to waste time between the plot of each circle in magenta and the plot of the same circle in black. It has the form

delay (*duration*)

where the duration is in milliseconds. The *Pow* program can thus be changed to the *Ripple* program in which a ripple starting at the center of the window appears to move out, by adding the two statements

```
delay (500)   % Delay half a second
drawoval (centerx, centery, radius, radius, 0)
```

after the drawoval that produces the magenta circle. This method of animation can be used for any simple form like a dot, line, box, or oval.

8.7 Drawing Arcs

A portion of an oval can be drawn using the predefined procedure drawarc in the form

drawarc (*x*, *y*, *xRadius*, *yRadius*, *initialAngle*, *finalAngle*, *c*)

where in addition to the parameters required to draw the oval you provide the initial and final angles in degrees measured counter clockwise that the lines from (*x, y*) to the end points of the arc make from the three o'clock position.

Figure 8.4 Drawing an Arc with drawarc

Here is a program that draws a series of circular arcs in green with their center at the center of the window and their radius changing by 1 pixel from 1 to 50. It is like the program *Pow* except that a portion of the circle is drawn rather than the whole circle. We will draw arcs of 60 degrees and a zero initial angle.

```
% The "CheeseSlice" program
% Draws a slice of green cheese
% that is one sixth of a round cheese
% by drawing circular arcs
setscreen ("graphics")
const maxRadius := 50
const xcenter := maxx div 2
const ycenter := maxy div 2
const theta := 60
for radius : 1 .. maxRadius
```

```
    drawarc (xcenter, ycenter, radius, radius, 0, theta, green)
end for
```

In Turing, you can also draw a filled in ÒsliceÓ by using the drawfillarc procedure. The call looks like

drawfillarc (*x*, *y*, *xRadius*, *yRadius*, *initialAngle*, *finalAngle*, *c*)

and draws a ÒsliceÓ filled with color *c*.

In the chapter on advanced pixel graphics we will use a different technique for drawing a **pie chart** but that requires more mathematics.

8.8 Plotting a Mathematical Function

To keep things simple we will plot a mathematical function that can be drawn with the origin of its coordinates at the lower left of the window, the bottom of the window as the *x*-axis, and the left side as the *y*-axis. We will plot the curve for the parabola

y = x ** 2

for values of *x* going from 0 to 14. We will let each unit in the *x*-direction be represented by 20 pixels. One pixel is thus .05 units. This means that in the x-direction we will use from pixel 0 to pixel 280. Each unit in the *y*-direction will be represented by 1 pixel since *y* will vary from 0 to 196. What we have done is to choose a proper **scale** for *x* and *y* so that the graph nearly fills the window.

```
% The "Parabola" program
% Draws the graph of the function y = x ** 2
% for x going from 0 to 14 in steps of .05
% Draw axes
drawline (0, 0, maxx, 0, blue)
drawline (0, 0, 0, maxy, blue)
for pixel : 0 .. 280
    const x := .05 * pixel
    drawdot (pixel, round (x ** 2), cyan)
```

```
end for
```

In the chapter on advanced pixel graphics we will show an all purpose mathematical graph plotting program.

8.9 Using Text with Pixel Graphics

Frequently we want to add text to a pixel graphics plot. The range of character colors is the same as for dots. The color number of characters is set by the statement

color (*chosenColor*)

The position of characters to be output is set using the statement

locatexy (*x*, *y*)

The next characters output by a **put** statement will begin approximately at the point (*x, y*). Note that the output will not necessarily appear at exactly (*x*, *y*). This is because the characters still appear in the 25 by 80 grid of character rows and columns. Instead, the character appears at the location closest to the *x* and *y* in the locatexy statement.

In the *Parabola* program we could have labelled the graph in magenta by adding these statements just before the last two statements.

```
color (magenta)
% Label x-axis
locatexy (160, 10)
put "x-axis"
% Label y-axis
locatexy (10, 100)
put "y-axis"
% Label graph of parabola
locatexy (100, 100)
put "Graph of parabola y = x**2"
```

8.10 Background Color

The color of the background of the Execution window can be set by using the statement

drawfillbox (0, 0, maxx, maxy, colorNumber)

This erases anything else in the window that is currently displayed.

Here is a program to change the color of the window randomly.

```
% The "FlashWindow" program
% Randomly changes the window color
setscreen ("graphics")
var c : int
loop
    randint (c, 0, maxcolor)
    drawfillbox (0, 0, maxx, maxy, c)
    delay (500)
end loop
```

8.11 Sound with Graphics

A simple sound is often all that is required in a graphic display. It can be obtained using the statement

sound (*frequency*, *duration*)

where the frequency is in hertz or cycles per second and the duration is in milliseconds. The frequency should generally be between 200 and 2000. Frequencies outside that range are too low or high to be reproduced on a computer speaker.

Often, computers require a sound card to process the sound or music commands.

Here is a program that draws graphics and plays a sound that goes up and then down.

```
% The "MakeSound" program
% Plays a rising then falling frequency along with colorful graphics
for i : 0 .. maxy
    drawfillbox (0, 0, i * 2, maxy – i, i mod 16)
    sound (i * 2 + 200, 50)
end for
for decreasing i : maxy .. 0
    drawfilloval (0, 0, i * 2, maxy – i, i mod 16)
    sound (i * 2 + 200, 50)
end for
```

8.12 Current Values of Graphic Parameters

You can find out the current values of various parameters when you are in pixel graphics mode. A number of functions provide such values.

```
whatdotcolor (x, y)
```

gives the color of the pixel at (*x, y*); the value of

```
whatcolorback
```

is the current background color number. Similarly the whatcolor function returns the current text color.

8.13 Exercises

1. Write a program to plot a horizontal band 5 pixels wide centered on any *y*-value you specify and with any color.

2. Change the program so that you can continue to add such horizontal lines as long as you want.
3. Write a program to draw a striped pattern of lines of random color at an angle of 45¡ to the bottom of the window. Arrange to stop execution when the window is completely covered.
4. Draw four circular arcs of width 5 pixels and radius 50 pixels centered on the four corners of the window using the drawarc procedure. What happens if you try to draw four complete circles using these centers and the drawoval procedure?
5. Draw a slice of watermelon with red flesh and green rind. Make it one quarter of a complete circular slice.
6. Write a program like the *BrownianMotion* program but instead of having an asterisk, use a small magenta ball. Can you arrange to display this on a window that is not black?
7. Write a program like the *Bounce* program only using a small circle instead of an asterisk to represent a puck bouncing off the boards in a hockey game. Make a sound as you bounce. Use delay rather than a time-wasting loop.
8. Write a program to draw a cartoon of a smiling face in the Execution window. Change the face to frown; now alternate between these two faces.
9. Draw the face of a clown centered on the window. Use maxx **div** 2, and maxy **div** 2 as the center of the face. Make the face in any shape and size oval that you wish. Add a nose in the center of the face. Add eyes so that they are symmetrically placed on either side of the face (they should be equidistant from the center of the face). Add a mouth, an arc. It's center coordinate should have maxx **div** 2 as its x. Add an ear on each side of the head. Make them so that their center has maxy **div** 2 for its y. Now that your clown is drawn, use animation to close the eye and open it again. The eye should close by reducing the size of the oval until it hits a size of 0. It should open by drawing an eye which grows from size 0 to the previous eye size. Use a delay to control the speed of the closing and opening of the eye. Which eye you animate is your choice.

10. Modify the clown above so that the eye closes in a different manner. The manner is of your choice. You may have it close from top to bottom, like a eyelid coming down, from left side and right side simultaneously until they hit the middle, or any method you choose. Again make the eye open. Use delays to control the speed of the closing and the opening.
11. Modify either clown#1 or clown#2. Have the eye close as well as the mouth close. The actions should appear to be simultaneous. Then have the eyes and mouth reopen, again simultaneously.
12. Draw an arrow on the bottom left of the window using drawlines. The point of the arrow should face right. Have the arrow move across the window from left to right using delay to control the speed. Once the arrow is off the window, make it reappear on the bottom- right corner with the point facing left.
13. Draw an oval at the bottom left of the window. Have the oval move from the bottom to the top of the window. Once it hits the top, make it change direction and come to the bottom. Once it hits the bottom, make it go diagonally to the top-right corner of the window and then drop vertically to the bottom of the window again (bottom- right).
14. Draw a ball in motion starting at the top of the window so that its center moves to the position it would move to under gravity in each step of the animation. The equation for the *y*-coordinate of the center is

    ```
    y = ymax – (1/2) * a * t ** 2
    ```

    where *a* is the acceleration of gravity, namely 9.8 m sec^{-2}, and *t* is the time in seconds. If each pixel in the *y*-direction represents 1 meter choose an appropriate time scale to watch the motion: in real time, in slow motion, and in time lapse motion.
15. Draw an animated graph to represent a ripple being reflected from the top of the window when the wave of the program *ripple* in this chapter strikes it. What happens when you let the original ripple go on out beyond the top of the window?

16. Write a program to plot either a cosine or a sine curve for values of the angle going from 0 to 720 degrees. Draw the *x*- and *y*-axes, and label the graph appropriately. Choose a scale for plotting that has the graph more or less fill the window. Remember that the values of cosine and sine vary between –1 and +1.

8.14 Technical Terms

pixel
dot
plot
graphics mode
resolution
coordinate
axis
origin of coordinates
drawdot
aspect ratio
palette
setscreen
maxx
maxy
maxcolor
drawline
drawfill
drawbox
drawoval
getch
delay
drawarc
pie chart
scale
parabola
sound
whatdotcolor
whatcolorback
whatcolor

Chapter 9

Selection

9.1 Simple Selection

So far we have looked at two of the three basic constructs in computer programs. The first was a simple **sequence** of statements, executed one after another, and the second was a **loop** construct in which the body of the loop is executed repeatedly. In this chapter we look at the **selection** construct by which you can get the computer to select one of a number of alternative courses of action.

Here is a program that reads in the mark in a course and outputs a pass or fail standing. If the mark is 50 or more the computer will say that it is a pass; if it is less than 50 it will say that it is a failure.

```
% The "PassFail" program
% Read a mark and decide whether it is a pass or failure
put "Enter a mark " ..
var mark : int
get mark
if mark >= 50 then
    put mark, " is a pass"
else
    put mark, " is a failure"
end if
```

Remember the >= sign means is greater than or equal to. The selection construct is sometimes called the **if...then...else** construct. The statement (or statements) between the **then** and the **else** are executed if the condition after the **if** is true. They are called the **then** clause. If the condition is false, the statement (or statements) between the **else** and the **end if** are executed. These are called the **else** clause. Only one of the **then** or **else** clauses is executed. Which one is selected depends on whether the condition is true or false.

If there is nothing to be done when the condition is false, the keyword **else** and the **else** clause are omitted. For example, here is a program where the **if** statement does not have an **else** clause.

```
% The "FindLargest" program
% Read 10 positive integers and find the largest
put "Enter 10 positive integers"
var number : int
var largest := – 1
for i : 1 .. 10
    get number
    assert number >= 0
    if number > largest then
        largest := number
    end if
end for
put "Largest integer of ten is ", largest
```

The variable *largest* is initialized to be –1 which will be smaller than any of the positive integers entered. Its type is implied to be **int** since its initial value –1 is an integer. Whenever a number that is bigger than the current value of *largest* is read, the value of the variable is changed to that new value. When the loop is finished the largest integer will have been stored in *largest*.

In the loop, after the **get** statement, there is a statement we have not seen before which says

```
assert number >= 0
```

If the condition after the keyword **assert** is false a run-time error will occur and execution of the program will be stopped. This prevents you from continuing if you enter a negative integer by mistake.

9.2 Three-way Selection

While simple selection allows a choice between two alternatives, often the solution to a given programming problem requires more choices. For example, determining the price of a movie ticket requires at least three alternatives based on the age of the movie-goer. Here is a program that calculates the price of a ticket based on age. This program calculates the ticket price for three alternatives: senior, adult, and child.

```
% The "MoviePrice" program
% Reads age and gives ticket price
var age : int
loop
    put "Please enter your age " ..
    get age
    assert age > 0
    if age >= 65 then
        put "Please pay $3.00"
        put "You are a senior"
    elsif age >= 14 then
        put "Please pay $6.00"
        put "You are an adult"
    else
        put "Please pay $2.50"
        put "You are a child"
    end if
end loop
```

This program will continue running until you interrupt it. The **assert** statement checks to see if the age is positive so that entering a negative age will stop it.

When an age is entered and has been passed by the **assert** statement, the program enters the three-way selection construct. This contains a new keyword **elsif**. If the condition *age >= 65* is true, the **then** clause is executed and the output is

```
Please pay $3.00
```

```
You are a senior
```

If the condition is false the person is either an adult or a child, and their age is less than 65. After the **elsif,** which is a special contraction of the two keywords **else** and **if,** we meet the second condition. If this condition is true, that is, *age* >= *14* the next two **put** statements are executed and the output is

```
Please pay $6.00
You are an adult
```

If the condition after the **elsif** is false, the statements after the **else** are executed with the output

```
Please pay $2.50
You are a child
```

Depending on the age of the person, one of the three possible alternatives is selected. We could call such a program construct an **if**...**then**...**elsif**...**then**...**else** construct. Like the **if**...**then**...**else** construct, the construct is terminated by the **end if**.

It is important to remember that the **else** must always come after the **elsifs** in a selection statement.

When using three-way selection, it is important to order the **if** statements correctly. Remember that as soon as the condition **if** tested true, then the execution goes to the **end if**. If we were to modify the program so that we switched the **if**'s as shown in the *BadMoviePrice* program

```
% The "BadMoviePrice" program.
...
    if age >= 14 then
        put "Please pay $6.00."
        put "You are an adult."
    elsif age >= 65 then
        put "Please pay $3.00."
        put "You are a senior."
    else
        put "Please pay $2.50."
        put "You are a child."
```

```
    end if
...
```

everyone would be adult or child. There would be no seniors. You must order from lowest to highest or highest to lowest.

It is not necessary to have an **else** clause. Here is a program that allows the user to buy three levels different products.

```
% The "Grocery" program
% Runs a cash register for a farmer's market stall selling a dozen corn
% for $2.25, bags of potatoes for $6.50, and artichokes for $1.75.
var total : real := 0
var product : int
loop
    put "Enter product (1=Corn, 2=Potatoes, 3=Artichokes, 4=Quit): " ..
    get product
    if product = 1 then
        total := total + 2.25
    elsif product = 2 then
        total := total + 6.50
    elsif product = 3 then
        total := total + 1.75
    elsif product = 4 then
        exit
    end if
    % The phrase "total : 0 : 2" outputs total to two decimal places
    put "The running total is $", total : 0 : 2
end loop
put "Final total = $", total : 0 : 2
```

9.3 Multi-way Selection

Here is a program that has more than three alternatives. It also uses a selection construct to test for improper data instead of using the **assert** condition. In this way you ask the person entering the data to correct it. This program reads a mark and gives the corresponding letter.

```
% The "Grade" program
% Read mark and find corresponding letter grade
put "Enter marks, end with 999"
var mark : int
loop
    put "Enter mark: " ..
    get mark
    exit when mark = 999
    if mark >= 0 and mark <= 100 then
        if mark >= 80 then
            put "A"
        elsif mark >= 70 then
            put "B"
        elsif mark >= 60 then
            put "C"
        elsif mark >= 50 then
            put "D"
        else
            put "Fail"
        end if
    else
        put "Improper mark"
    end if
end loop
```

The multi-way **if** selection construct is in the body of a conditional loop which starts with **loop** and finishes with **end loop**. The exit condition of the loop is true when you enter the dummy mark 999. After the first keyword **if** in the program is the compound condition

mark >= 0 **and** *mark* <= 100

The **and** means that both these simple conditions must be true: the mark must be between 0 and 100 inclusive. If it is true the **then** clause is executed otherwise the **else** clause is executed, and the words ÒImproper markÓ will be output.

The **then** clause contains a selection construct, a multi-way one at that. This multi-way selection construct is **nested** inside the **then** clause of the two-way selection construct. The nesting is

emphasized by the indentation of the lines of the program. A multi-way selection has one or more **elsif...then** clauses. Only one of the **then**, **elsif...then**, or **else** clauses is executed. The multi-way selection construct, like the two-way selection construct, ends with the keywords **end if**.

Ordering of the if statements is also crucial in mutli-selection. They must go from highest to lowest or from lowest to highest. Do not mix up the order of the conditions as it will influence the logic of the program. Examine the *BadGrade* program.

```
% The "BadGrade" program
...
        if mark >= 60 then
            put "C"
        elsif mark >= 50 then
            put "D"
        elsif mark >= 80 then
            put "A"
        elsif mark >= 70 then
            put "B"
        else
            put "F"
        end if
...
```

A mark of 99, 100, 75, 65, or 60 would all be Cs. This program would grant Ds to marks between 50 and 59 and Fs to all others.

9.4 Case Construct

When the choice among alternatives of a multi-way selection is determined by an integer value you can use the **case** construct. Here is a program that counts the number of votes for three political parties called Left, Middle, and Right. To vote for one of these enter a 1, or a 2, or a 3 respectively. To end the voting procedure enter –1.

```
% The "Voting" program
% Read votes and count totals
const sentinel := – 1
put "Vote 1 for Left, 2 for Middle, 3 for Right, end with ", sentinel
var vote : int
var left, middle, right : int := 0  % initialize all three to 0
const leftVote := 1
const midVote := 2
const rightVote := 3
loop
    put "Enter vote " ..
    get vote
    case vote of
        label leftVote : left := left + 1
        label midVote : middle := middle + 1
        label rightVote : right := right + 1
        label sentinel : exit
        label : put "Invalid vote"
    end case
end loop
put "Left" : 8, "Middle" : 8, "Right" : 8
put left : 4, middle : 10, right : 7
```

In the **case** statement, if the value of *vote* is 1 the statement following **label** leftVote**:** will be executed and 1 will be added to the total *left*. If the value of *vote* is the signal then the **exit** causes the loop to stop. If the value of *vote* is not any one of those specified after the **label** keywords then the **label** with no value following it will be selected. This one acts like an **else** clause in the **if..then..else** construct and must be the last **label** in the **case** statement.

Notice the rather peculiar field sizes for the integers in the last **put** statement. These will result in the lining up of the total votes in each of the three categories with the headings.

Remember strings are left-justified in their fields and integers (or reals) are right-justified. Try changing the size of each field to 8, the same as for the headings, and see the difference.

The **case** construct does not allow you to do anything that the **if..then..else** construct cannot do but sometimes it seems more understandable. It is also more efficient.

9.5 Commands for Action

One of the important things you can do with the selection construct is to issue commands to the computer to take different actions depending on what the command is.

Here is an example program which uses the selection construct.

```
% The "ShowEmotion" program
% Respond to various commands
var command : string
loop
    put "Enter command: " ..
    get command
    if command = "stop" then
        exit
    elsif command = "sing" then
        put "la la la"
    elsif command = "cry" then
        put "boo hoo"
    elsif command = "laugh" then
        put "ha ha ha"
    else
        put "I don't understand"
    end if
end loop
put "That's all folks"
```

Here is a sample Execution window.

```
Enter command cry
boo hoo
Enter command laugh
```

```
ha ha ha
Enter command smile
I don't understand
Enter command stop
That's all folks
```

Notice that the response to the command ÒstopÓ is just **exit** which takes you out of the **loop** to the statement after **end loop**.

9.6 Selecting from a Menu of Commands

Sometimes it is easier to present a list or **menu** of all the possible commands available at a particular time and let the user choose one. If the commands are numbered you can choose by typing the number. If the command is a number we can use a **case** statement instead of an **if..then..elsif..else** statement.

Here is a program to produce the same results as before.

```
% The "ShowEmotion2" program
% Here we select commands by number from a menu
var command : int
loop
    put "Choose from 1-sing, 2-cry, 3-laugh, 4-stop: " ..
    get command
    case command of
        label 1 : put "la la la"
        label 2 : put "boo hoo"
        label 3 : put "ha ha ha"
        label 4 : exit
        label : put "I don't understand"
    end case
end loop
put "That's all folks"
```

You can use strings as well as integers in a case construct. Here is a version of the *ShowEmotion2* program that asks the user to enter a string to represent the action.

```
% The "ShowEmotion3" program
% Here we select commands by entering a string
var command : string
loop
    put "Choose from sing, cry, laugh, stop: " ..
    get command
    case command of
        label "sing" : put "la la la"
        label "cry" : put "boo hoo"
        label "laugh" : put "ha ha ha"
        label "stop" : exit
        label : put "I don't understand"
    end case
end loop
put "That's all folks"
```

9.7 Exercises

1. Write a program to divide a class of students in two groups: those whose last names begin with A to H and those that begin with I to Z. Ask a person to enter their last name and then output a message indicating which group they are in. Repeat for each student.
2. Prepare a check for someone eating lunch in a restaurant. If the meal costs more than $4.00 a 7% tax is to be added.
3. Write a program to classify athletes into three classes by weight. The categories are: over 80 kg - heavyweight, between 60 and 80 kg - medium weight, and less than 60 kg - lightweight. Prepare the program so that a team of 10 athletes can enter their weight one after another and be told what category they are in.
4. Write a program to read a series of first names of people. After reading the series (you will need a signal), output the name that is alphabetically last.
5. Write a program to display a multiple choice question with five different answers in the window and then depending on which

answer is chosen give a different comment. For example, your question might be

Turing is:

(1) a great programming language
(2) a kind of car
(3) a mathematician
(4) a machine
(5) all of the above

The choice will be an integer from 1 to 5. Use a **case** construct for this.

6. Write a program to give proper greetings to a person. Ask what the occasion is and then give the appropriate greeting. You can offer a menu of available occasions. Use a case construct with string input.
7. Federal income tax is to be levied in stages on taxable income. On the first $27,500 you pay 17%, on the next $27,500 you pay 24%, and on the rest 29%. Write a program to read in a taxable income and compute the federal tax payable to the nearest cent.
8. Write a program to read in a series of positive integers and output the range of the integers, that is, the interval from the smallest to the largest.
9. Compute values of the function

$$f(x) = 3x^2 - 2x+1$$

in steps of 0.1 between x=0 and x=1 inclusive and find the value of x for which f(x) is a minimum.

10. Rewrite the *Grade* program of this chapter to use an **assert** statement instead of a nested **if**...**then** construct. What happens when you enter a mark that is negative or greater than 100. Try misspelling **elsif** as **else if** and see what happens.
11. Marks in a test are given out of 10 where 9 or 10 is A grade, 7 is B grade, 6 is C grade, 5 is D grade, below 5 is F grade. Use a **case** construct to change the numerical test mark into the

appropriate letter grade. Note that if you want to give the same grade for both 9 and 10 you can label that case with two values. For example,

```
label 9, 10: put "This is an A"
```

Arrange to enter a number of test marks for a class ending with a mark of –1. At the end, output the percentage of the class with each letter grade. To do this you must keep track of the total numbers of marks and the numbers in each grade.

12. Write a program called *Mystery* which offers a menu of mysterious alternatives. Use a **case** construct to do quite different things in response to the different choices.
13. Write a program to input an integer from the user and output all its factors and whether or not it is a prime number. Remember that a prime number has only two factors, 1 and itself. Three is prime while four is not.
14. Write a program to output all the prime numbers from 1 to 50.
15. Write a program that asks the user for 10 marks(percents) and, at end of the run, outputs the highest mark and the lowest mark.
16. Write a program to input a a series of positive integers until a sentinel is entered. Use –1 as the sentinel. Output the product of all the integers input.
17. Write a program to read in 5 marks that should be between 0 and 100 inclusive. Output an error message if a mark is not between 0 and 100. For the valid marks, output ÒgoodÓ if the mark is between 70 and 100 inclusive and ÒsatisfactoryÓ if the mark is between 50 and 69 inclusive.

9.8 Technical Terms

selection construct
if...then...else **construct**
assert **statement**
two-way selection
three-way selection

Chapter 10

Storing Data on the Disk

10.1 Data Files on Disk

All the disk files in the examples so far have been programs. Data, such as numbers that are output from a program, can also be stored as files. We normally send our output data to the window but it is possible to redirect the output data to a file on the disk. Suppose we had this program in the Editor window.

```
% The "Counting" program
for Index : 10 .. 15
    put Index : 3 ..
end for
```

If you give the command to run the program the Execution window will look like this:

```
 10 11 12 13 14 15
```

To run this program and store its output as a disk file called *Count*, we give the output redirection command. Select the **Run with Args...** menu item from the **Run** menu. This displays a dialog box where you click the **Output to: File** button. Enter the name *Count* into the file dialog box that now appears and then click the **Run** button in this dialog box.

This time the results do not appear in the window but instead they are stored on the disk in the file called *Count*. Now bring the file *Count* into the Editor window using the *Open* command. The data file containing the numbers 10 to 15 will be in the window. You can list these numbers on the printer by selecting the **Print** menu item from the **File** menu.

This is one way of getting printed output for the program. You can also redirect the output to go to the printer directly in the **Run with Arguments** dialog box.

You can also have the output go to the Execution window and a file at the same time. You do this by selecting the **Output to: Screen and File** button in the **Run with Arguments** dialog box.

10.2 Input Data from Disk Files

You can use a data file as input for a program. Here is a program that will read and output the numbers in the data file *Count*.

```
% The "EchoInput" program
put "Enter five integers"
var number : int
for i : 1 .. 5
    get number
    put number
end for
```

Here the index or counter of the **for** loop has a one-letter name *i* . We often use letters like *i, j, k* as index variables when our imagination for making up good variable names fails us. We could have used *index* or *counter* but that gets boring.

This program expects five integers to be input. It will then echo these integers. Here is a example Execution window in which 10 is typed and then echoed, then 11 is typed and echoed, and so on.

```
Enter five integers
10
10
11
11
12
12
13
13
14
14
```

Now suppose instead we redirect input to come from the file *count* rather than the keyboard. Select the **Run with Args...** menu item from the **Run** menu, select the **Input from: File**

button, and choose the file *Count* from the file dialog that is displayed. The Execution window will now look like this

```
Enter five integers
10
11
12
13
14
```

The input does not appear in the window. The result of the **put** statement shows you what was input. If you wish to have the input echoed in the window when it is read from a file, select the **File with Echo** button in the **Run with Argument**s dialog box.

You can redirect both the input and the output at the same time. Select the **Run with Arguments...** menu item from the **Run** menu, select the **Input from: File** button, and select *Count* from the file dialog. Then click the **Output to: File** button and enter *list* in the file dialog that appears. Now nothing appears in the Execution window, but if you check the directory you can see that the file *List* is now there. Try bringing it into the Editor window using the *Open* command to see what it is like. Try printing it.

If you want to have the input read from a file and the output go to a file but still see the program executing, you can select Input: **File with Echo** and Output: **File and Screen**.

10.3 End-of-file for Data

We have looked at programs that read from input until a special signal such as –1 is found in the data. We will now look at programs that detect the end of the data by checking for the end-of-file marker. All files on a computer contain a marker after the last piece of data in a file. Programs can check for this marker. Using a check for end-of-file eliminates the need to add a special signal to the data.

Here is a program that reads a series of grades and averages them. It is very much like the *ComputerAverage* program of the

chapter on repetition but now the signal will be the end-of-file marker. Since we will be reading from a disk, a prompt is not appropriate. We need prompts only when we are interacting with a person at the keyboard.

```
% The "ComputeInputAverage" program
% Computes the average of a series of grades
% stored in a disk file
var grade : int
var count, sum : int := 0
loop
    % Skip over any white space to next grade or to eof
    get skip
    exit when eof
    get grade
    sum := sum + grade
    count := count + 1
end loop
put "Average mark is ", round (sum / count)
```

In the *ComputeInputAverage* program the variables *count* and *sum* are initialized to zero in their declaration. They could have also been declared and initialized with the statement

```
var count, sum := 0
```

In this case the variables *count* and *sum* are initialized to zero in their declaration but their type **int** is omitted. The type in a declaration can be omitted if you are initializing; the type of the initial value is automatically given to the variable. If the type is omitted in the declaration of a **real** variable, the initial value cannot be an integer (such as 0) but would need to be a real value (such as 0.0).

The first statement in the loop uses the special input statement **get skip** which moves to the next number or the end-of-file marker. The value of eof is true if we are at the end of the file, otherwise it is false. We need to use **get skip** in the program because white space after the last number can prevent eof from being true.

When entering input data from the keyboard the key combination *Control+Z* indicates the end of file. (On some computers it is *Control+D*.)

10.3.1 End-of-file with Strings

Here is a program that counts the number of words in a file *text*.

```
% The "WordCount" program
% Counts the number of words in a text
var count : int := 0
var word : string
loop
    get skip
    exit when eof
    get word
    count := count + 1
end loop
put "There are ", count, " words in the text"
```

Run the program taking input from a file called *Text* (redirect the input to come from the file *Text*). Notice that a **get skip** is needed with strings just as it is with numbers when you are looking for the end of file.

10.4 Reading Lines of Text from a File

So far in reading text we read in a token at a time, a token being a string of characters surrounded by white space or a string in quotes. You can read an entire line of text at a time. This form of input is known as **line-oriented** input (as opposed to **token-oriented** input).

Here is a program that reads lines of text and outputs each line along with its line number.

```
% The "ListInput" program
% Read text and output a line at a time
% numbering each line
var lineNumber := 0
var line : string
loop
    exit when eof
    get line : *      % Read an entire line
    lineNumber := lineNumber + 1
    put lineNumber : 4, " ", line
end loop
```

The important instruction is the **get** line : *. The colon followed by the asterisk causes an entire line of input to be read at once and stored in the string variable line. This is also how you can read in a complete name with spaces without having to force the user to put quotes around the name.

You can use a program as text. For example, if the *ListInput.t* program is stored on the disk as the file *ListInput.t* it can be listed with line numbers by redirecting input to *ListInput.t* to come from the file *ListInput.t*. The Execution window would look like this

```
1 % The "ListInput" program
2 % Read text and output a line at a time
   ... and so on ...
```

This is an interesting example in which a program, *ListInput.t* reads its own text.

10.5 Exercises

1. Prepare a file of data called *Marks* that can be used as input for the *ComputeAverages* program of the chapter on repetition. Run the program redirecting the input to be read from the file *Marks*.

   Two versions of the Execution window are shown for *ComputeAverages* in the repetition chapter: one for the case where the marks are entered from the keyboard one-to-a-line,

and one for the case where all five marks are entered before the Return is pressed. How does your Execution window compare with these? Try two versions of your input data file *Marks* to see what difference it makes. How could the *ComputeAverages* program be modified so that the Execution window is the same as one or both of the keyboard versions?

2. Try redirecting the output of the *DrawSky* program in Chapter 7 to a file named *Blue* . What do you see in the window? Now bring the output into the window by bringing *Blue* into the program window

   What happens? Is output redirection suitable for graphics programs?

3. Prepare a data file called *Final* that will contain marks of students on final exams in a suitable form for input to the *Grade* program of the chapter on selection (chapter 9). Bring the *Grade* program into the window (or enter it if you do not have it saved on the disk). Now run the program redirecting input to be read from the file *Final* and the output to be written to the file *Letters*.

   Compare the file *Letters* with what you would get if you entered the same exam results at the keyboard. How can the *Grade* program be modified so the two are identical?

4. Prepare a file of input data suitable for the *ShowEmotion* program from Chapter 9. Store it on the disk under the name *Examples*. Now bring the *ShowEmotion* program into the window and then give the command to run it with data coming from the *Examples* file.

   How does the Execution window compare in appearance to the one you would have if you typed in the same data from the keyboard? Can you change the *ShowEmotion* program so that the Execution window will be the same, that is, so that you know what the input values are?

5. Having changed the *ShowEmotion* program to echo the values of input data so that they appear on the output when the input is from disk, redirect the output to a file called *Answers*

Check the list of files stored. Read the *Answers* file into the Editor window. Now print it.

6. Enter the *ComputeInputAverage* program of this chapter. Try using it with the input data coming from the keyboard. Now prepare a data input file on disk called *Class* and run *ComputeInputAverage* with the input coming from the file *Class*.

   Try modifying the program by removing the **get skip** instruction and see if you detect any difference in performance. Try giving the eof right after the last number and also giving a Return followed by the eof.
7. Type in the *WordCount* program of this chapter and use it to count the words in the program itself by redirecting input to come from the file *WordCount.t* (in other words, itself).

   What gets counted as a word?
8. Type in the *ListInput* program of this chapter and list some of the programs and data files you have on the disk.
9. Create a *Story.txt* file. It can contain any fiction you like. Write a program to receive input from this file and output it to the window with only four words to a line.
10. Using the *Story.txt* file, write a program that reads the file for input and then outputs the story to the window so that the first word is sent to the Execution window the second is redirected to *Story2.txt.* This pattern should continue until the end of the file. At the end output to the window: the total number of words sent to the window, the total number of words sent to the file, and the total number of words handled from story.txt.

11 What happens when you direct output to a file that already exists? (What happens to the original data in the output file?) Try this with a program.

10.6 Technical Terms

data files on disk

Chapter 11

Handling Strings

11.1 Length of a String

In this chapter we will look at ways that you can work with strings. All the different operations that you need to perform can be accomplished by two basic string operations: joining two strings together and selecting a part of a string. As well, you need to use the predefined function length to tell you the number of characters in a string.

It is easy to find the length of any string stored in a variable using the Turing predefined function length.

Here is a program that outputs the length of words.

```
% The "WordLengths" program
% Read words and find their length
put "Enter one word to line, end with 'end' "
var word : string
loop
    put "Enter word: " ..
    get word
    exit when word = "end"
    put word, " has ", length (word), " letters"
end loop
```

Here is a sample Execution window.

```
Enter one word to line, end with 'end'
Enter word banana
banana has 6 letters
Enter word kangaroo
kangaroo has 8 letters
Enter word end
```

11.2 Joining Strings Together

The operation of joining strings together is called **catenation** and is accomplished using the operator +. When this operator is between two numbers it means that they are to be added. When the + operator is between two strings it means they are to be joined. Here is an example.

```
put "O" + "K"
```

It produces the output *OK*; the two strings are joined.

Here is a program that reads five words and joins them into a line of output.

```
% The "WordsOnLine" program
% Reads 5 words and outputs all on a line
var word : string
var line := ""    % Initialize line to the empty string
put "Enter 5 words"
for i : 1 .. 5
    get word
    line := line + word + " "
end for
put line
```

Here is a sample Execution window.

```
Enter 5 words
Why am I entering this many words
Why am I entering this
```

If the words were entered one to a line the output would be given right after you enter the 5th word and the window would look like this

```
Why
am
I
entering
```

```
this
Why am I entering this
```

Notice that in the declaration of the string variable *line* it is initialized to the empty string. In the **for** loop the most recently read *word* is catenated onto the line and a blank is catenated on to that; the result is stored in the variable *line*. If the blank were not catenated the output would be

```
WhyamIenteringthis
```

11.3 Selecting Part of a String

A part of a string is called a **substring**. We define a substring by placing in parentheses after the name of the string: the position of the first character in the substring, then two dots, followed by the position of the last character in the substring. The character positions of a string are numbered starting on the left from 1 to the length of the string. The output for this program

```
const word := "magnet"
put word (4 .. 6)
```

would be *net*. The substring *net* consists of the 4th to the 6th characters of *magnet*.

If the substring goes from some position to the end of the string then the last position can be written as an asterisk. So the program we just had would give the same result if the **put** statement were

```
put word (4 .. *)
```

Here is a program to read a series of words and output the last three characters of each word until the word *quit* is read.

```
% The "FinalThreeLetters" program
% Read words and if possible give last three characters
var word : string
```

```
put "Enter words one to a line, end with 'quit' "
loop
    put "Enter word: " ..
    get word
    exit when word = "quit"
    if length (word) >= 3 then
        put word (* – 2 .. *)
    else
        put "Word has fewer than 3 characters"
    end if
end loop
```

Notice that the third last character can be given as * – *2* in the substring specification.

You must not ask for a character position that is not present, for example, a character position whose number is less than 1 or greater than the length of the word or an execution time error will result. In general, the first character position's value must be between 1 and the last character position's value inclusive. In the particular case where it is one greater than the last character position the result is an empty string. For example, *word (*length *(word)+ 1 .. *) gives the empty string.

Here is a sample Execution window for the *FinalThreeLetters* program.

```
Enter words one to a line, end with 'quit'
Enter word speaking
ing
Enter word softly
tly
Enter word to
Word has fewer than 3 characters
Enter word quit
```

If we had not tested in the **if** statement to see if the *word* was at least 3 characters long and had asked for the last three characters of *to* we would have been told of an execution error. You should try this program and see what you get

```
const word := "to"
```

```
put word (* – 2 .. *)
```

There will be an execution error because * – 2 attempts to locate a character before the first character.

If the substring is to be a single character you can just put that character's position in parentheses; a range is not required.

Here is a program that outputs each letter of a word on a separate line.

```
% The "LetterAtATime" program
% Read a word and output it a letter-at-a-time
var word : string
put "Enter words, end with 'tired' "
loop
    put "Enter word: " ..
    get word
    exit when word = "tired"
    for i : 1 .. length (word)
        put word (i)
    end for
end loop
put "Why not take a rest?"
```

Here is a sample Execution window.

```
Enter words, end with 'tired'
Enter word sick
s
i
c
k
Enter word tired
Why not take a rest?
```

11.4 Searching for a Pattern in a String

We have seen several examples where we recognized a word such as *stop* or *quit* that we have read. We also need to be able to recognize if a certain pattern of characters is in a string or not. For example, we might want to look for words containing the pattern ÒieÓ. To do this we use the predefined Turing function index whose value is the character position where a pattern first matches a string. The value of

index (*string*, *pattern*)

is the first position from the left in *string* where the *pattern* matches. For example,

index ("getting", "t")

has a value 3. There is a second occurrence of *t* in *getting* at position 4. The value of

index ("dandelion", "lion")

is 6. If there is no match at all the value of index is zero.

Here is a program to detect words that contain the letter *s*.

```
% The "HaveAnS" program
% Test to see if a word contains an "s"
var word : string
put "Enter a series of words, end with 'last' "
loop
    put "Enter word: " ..
    get word
    exit when word = "last"
    if index (word, "s") not= 0 then
        put word, " contains an 's' "
    else
        put word, " does not contain an 's' "
    end if
end loop
```

Here is a similar program to see if a word contains two occurrences of a pattern. It is a more difficult process. This time the program will ask you to enter the pattern you want to test for.

```
% The "DoublePattern" program
% Test to see if a word has two occurrences of a pattern
var word, pattern : string
put "Enter the pattern you want to test for: " ..
get pattern
const size := length (pattern)
put "Enter a series of words"
put "Enter 'finis' to stop"
loop
    get word
    exit when word = "finis"
    const place := index (word, pattern)
    if place = 0 then
        put word, " contains no occurrences of ", pattern
    elsif index (word (place + size .. *), pattern) = 0 then
        put word, " contains one occurrence of ", pattern
    else
        put word, " contains at least two occurrences of ", pattern
    end if
end loop
```

In this program if you find that the word contains one occurrence of the pattern then you must see if there is a second occurrence. The second search for the pattern must be in the substring of *word* starting after the end of the first occurrence and going to the end.

To see whether or not you have the correct expressions for the conditions try working through a test case or two on paper rather than on the computer. For example, suppose the pattern is ÒsÓ and the word is ÒwallsÓ. The value of *size* is 1 and *place* is 5. The value of *place* + *size* is 6. The value of the substring *word (place + size .. *)* will be the empty string which is what you get if the value of the beginning of the substring range is one greater than the length of the string which is 5. Now try the word ÒglassÓ. The value of *place* will be 4 and *place* + *size* will be 5.

The substring *word (place+size .. *)* is then really *word (5 .. 5)* which is just the last character.

11.4.1 Counting Patterns in a Word

The previous program shows how to count the patterns in a word. If the word is ÒbananaÓ and the pattern is ÒaÓ then the program will tell you that there are at least two occurrences of ÒaÓ in ÒbananaÓ. It is also possible to modify the program so that the exact count will be outputt.

Here is the revised program that inputs the pattern and outputs the count.

```
% The "DoublePattern2" program.
var pos, count, size : int
count := 0
var word, pattern : string
put "Enter the pattern that you want to search for: " ..
get pattern
put "Enter the word you want to search through: " ..
get word
size := length (pattern)
loop
    pos := index (word, pattern)
    exit when pos = 0
    count := count + 1
    word := word (pos + size .. *)
end loop
put "The number of occurrences: ", count
```

You can see how *count* changes by tracing through the program. The following trace shows how the *DoublePattern2* program runs when *pattern* is set to ÒaÓ and *word* is set to ÒbananaÓ.

```
count=0
size=1
first time through loop
    pos=2
    count=1
```

```
    word=nana
second time through loop
    pos=2
    count=2
    word=na
third time through loop
    pos=2
    count=3
    word=""
fourth time through loop
    pos=0
loop exited and the count of 3 is output
```

11.5 Substituting One Pattern for Another

We showed examples where we looked for a pattern in a word. We could have substituted a different pattern after we found the one we were searching for. This would then be a search and substitute process. For example, in the program *HaveAnS*, in the previous section we could have substituted another letter such as ÒtÓ for the ÒsÓ.

In Turing, you cannot assign to a substring. In order to change part of a string, you must rebuild the string. For example, to change the letter ÒsÓ to ÒtÓ in a string, you would create a new string by concatenating the part of the string up to (but not including) the ÒsÓ with the ÒtÓ and then concatenating the result with the part of the string from just past the ÒsÓ to the end of the string.

Here is the assignment statement that would do it. Place this statement after the **put** in the **then** clause

```
word := word (1 .. index (word, "s") – 1) + "t" +
            word (index (word, "s") + 1 .. *)
```

The new *word* is made up of three pieces catenated. The first piece is the substring of the original *word* up to the position of the pattern. Next comes the substituted letter ÒtÓ, then the substring

of the original *word* starting after the pattern and going to the end. If the pattern is already at the end of the word this last substring will be a empty string.

Sometimes in programs the same function is evaluated several times. It is often more efficient to assign its value to a variable, then use the variable instead of reevaluating the function. For example, in the *twice* program we assigned to the variable *place* the value of index *(word, pattern)*.

11.6 Eliminating Characters from Strings

Sometimes it is useful to be able to eliminate a certain class of characters from a string. Here is a program which removes all the vowels from a word.

```
% The "RemoveVowels" program
% Eliminates the vowels from a word
var word : string
put "Enter a series of words, end with '*' "
const vowels := "aeiou"
loop
    get word
    exit when word = "*"
    var newWord := ""       % empty string
    for i : 1 .. length (word)
        if index (vowels, word (i)) = 0 then
            % Letter is not a vowel
            newWord := newWord + word (i)
        end if
    end for
    put "Word without vowels ", newWord
end loop
```

In the **for** loop each letter of *word*, namely *word (i)*, is tested as the pattern against the string of vowels. If it is found in the

string of vowels the *index* function will not be zero and we do not catenate it onto the new word we are forming.

Here is a sample Execution window with the keyboard input in bold.

```
Enter a series of words, end with '*'
jump
Word without vowels jmp
diagonal
Word without vowels dgnl
*
```

We can also remove a pattern from within a string using *word* by

word := *word* (1 .. *pos* – 1) + *word* (*pos* + *size* .. *)

where *pos* is the position of the pattern in the string and *size* is the size of the pattern. This redefines *word* so that it contained the part of the word before the pattern (*word* (1 .. *pos* – 1)) and the part of the word after the pattern (*word* (*pos* + *size* .. *)).

Here is a modification of the *DoublePattern2* program that outputs the string with all occurrences of a pattern removed.

```
% The "DeletePattern" program.
var pos, size : int
var word, pattern : string
put "Enter the pattern that you want to search for: " ..
get pattern
put "Enter the word you want to search through: " ..
get word
size := length (pattern)
loop
    pos := index (word, pattern)
    exit when pos = 0
    word := word (1 .. pos – 1) + word (pos + size .. *)
end loop
put "Here is the word with the pattern removed: ", word
```

11.7 Bullet-Proofing Programs

Another important consideration when designing programs is making sure that the user cannot crash the program by entering unexpected data. This is called Òbullet-proofingÓ your program. The most common kind of error that can occur is if the user enters a letter of the alphabet as input when asked for a number. When this happens, Turing is unable to read the input as a number and the program stops execution and outputs an error message. It creates a run-time error.

Reading all input as strings avoids this problem. The string is converted to an integer or real only after making certain that the string contains valid input. If the string does not contain valid input, the program can ask the user to reenter a proper value.

Here is an example of a program segment that gets an integer value from the user.

```
var input : string
var age : int
put "Enter your age: " ..
loop
    get input
    exit when strintok (input)
    put "Not a number. Please enter an integer: " ..
end loop
age := strint (input)
```

The program prompts the user for input and then enters the loop. In the loop, the user inputs their age into the string variable *input*. The program then checks whether *input* can be converted to an integer. The strintok (pronounced *strint-okay*) built-in subprogram examines *input* and returns true if *input* can be converted to an integer and false otherwise. If *input* cannot be converted to an integer, the program outputs an error message and loops back, asking for the another input. If the user enters valid input the program leaves the loop. The string is then converted into an integer using the strint function.

A similar program segment for reading in a real number can be created using the strrealok and strreal perdefined subprograms.

11.8 Exercises

1. Write a program to count the total number of characters in a series of 10 words that you enter, and compute the average word length.
2. Write a program to output the first and last letters of a series of words. A sample Execution window might be:

```
Enter a series of words one to a line, end with
'wow'
Enter word pig
pg
Enter word dog
dg
Enter word a
Word has only 1 character
Enter word wow
```

3. Write a program which produces a line of asterisks of a given length by catenating enough single asterisks in a **for** loop. Here is a sample Execution window:

```
Enter a negative number to stop
How many asterisks do you want? 8
********
How many asterisks do you want? 5
*****
How many asterisks do you want? -1
```

The *repeat* predefined function can be used to do this too. For example,

put repeat ("*", 5)

will result in the output *****. Patterns of more than one character can be repeated also, for example, repeat ("Hi", 3) produces three ÒHiÓs.

4. Rewrite the program of question 3 so that the pattern to be repeated by catenation is read into the computer. Try several patterns.
5. Write a program to change words made up of lower case letters into a secret code. The letter *a* is to be changed to *b*, *b* to *c*, and so on; *z* becomes *a*. To do this you must know a little about the ASCII code shown in the appendix. The predefined function ord has a value equal to the ASCII code of the letter which is its parameter. For example, ord ("a") has the value 97 which is the ASCII equivalent of the letter *a*. The letter *b* has the code 98. The predefined function chr can translate back from a value to a letter. For example, the function chr (97) has a value the character *a*. To change an *a* to a *b* you would use both functions, one after the other. The value of

   ```
   chr (ord ("a") + 1)
   ```

   is *b*. In this way you can convert to the secret code. Try your luck at this.
6. Write a program to read a series of words from the keyboard and output the reverse word with the letters backward. Keep all the letters of the word in the same case: upper or lower. Here is a sample window.

   ```
   Enter a word COW
   The reverse word is WOC
   Enter a word madam
   The reverse word is madam
   (etc.)
   ```

   If the reverse word is the same as the word, the word is a **palindrome**. If you find this to be the case output a line saying

   ```
   This is a palindrome
   ```
7. Read a series of words and output the middle letter of each word that has an odd number of letters or announce that the

word has an even number of letters. Here is a sample Execution window:

```
Enter word brine
The middle letter is i
Enter word bright
The word has an even number of letters
(etc.)
```

Use the end-of-file signal to stop the repetition. Try putting the input on a disk file called *list* and redirect the input to be from it. Does your output look the same as before? How could you change your program so that you see the word that is read.

8. Write a program which gives the user this menu.

```
Menu
1. Count a pattern
2. Eliminate a pattern
3. Substitute a pattern
4. Exit
```

If the user chooses #1 - ask for a word and a single letter pattern. Display the number of times the pattern occurs in the word. (banana, a, 3)

If the user chooses #2 - ask for a word and a single letter pattern. Display the word without the pattern (banana, a, bnn)

If the user chooses #3 - ask for a word, a single letter search pattern and a single letter replacement pattern. Display the word with the alterations. (banana, a, o, bonono)

If the user chooses #4 - quit the program

Use getch and clear the Execution window between options.

9. Modify alternative 3 in Exercise 8 so that it properly handles the case of replacing each o in moon by oo.
10. Using the *RemoveVowels* program as your guide, write a program which inputs a string and then outputs each character in the string and whether or not the character is a vowel, a consonant, a number or any other character. For example if Ò5te+Ó were input, then the output should be:

```
5 is a number
```

```
t is a consonant
e is a vowel
+ is any other character
```

11. Modify Exercise 10 so that it outputs the vowels, consonants, numbers, and other characters as words. For example if Òasdfert456u2~1?Ó were input then the output would be:

```
numbers - 45621
vowels - aeu
consonants - sdfrt
any other character - ~?
```

12. Use your ingenuity to come up with a different way of finding out whether or not a word is a palindrome.
13. Write a program that inputs a word, a letter, and a replacement letter. If the first letter exists in the word, all occurrences of it should be replaced by the replacement letter and the new ÒwordÓ printed . If the letter does not exist in the word, the message Òno replacement neededÓ should be printed. For example:

```
Enter a word: program
Enter a letter: r
Enter replacement letter: l
New "word" is plogram.
```

14. Repeat Exercise 8 but use patterns of more than 1 letter (replace a pattern of letters in a word with another pattern of letters).

11.9 Technical Terms

length of string
catenation
substring
range of substring
index function
searching for pattern
substitution of one string by another
deletion of characters
insertion of characters
repeat function
ord function
chr function
palindrome

Chapter 12

Processing Text

12.1 Token-oriented Input

We have already seen how to read and output strings of characters. The string of characters was read as a single input item free of blanks or enclosed quotes. In other words it was read as a token which is a string of characters surrounded by **white space**. This chapter explains other ways of reading characters, including a specified number of characters or a whole line of characters.

The normal input of strings in Turing is by tokens. Here is a program, just for review, which reads a series of words in a line ending with *stop* and outputs one word at a time.

```
% The "EchoWords" program
% Read a series of words, end with stop
var word : string
put "Enter a line of text, end with stop"
loop
    get word
    exit when word = "stop"
    put word
end loop
```

Here is a sample Execution window.

```
Enter a line of text, end with stop
Please don't stop
Please
don't
```

The word *Please* is the first token read; since it is not *stop* it is then output. On the next round of the loop the word *don't* is read and again output. When the token *stop* is read the exit condition is satisfied and you leave the loop and the program is finished.

No matter how many words you have after the word *stop*, and before you press Return, the result would be the same. For example, if you enter

```
Please don't stop doing what you're doing
```

the output is identical.

You can make a token out of a string of characters with a blank in it by surrounding the string with double quotes. Here is an example program.

```
% The "Quotes" program
% Read a string in quotes
var name : string
put "Enter a string in quotes"
get name
put name
```

Here is a sample window.

```
Enter a string in quotes
"Alan Turing"
Alan Turing
```

The quotes are not part of the token and are not stored in the variable *name*.

12.2 Inputting a Fixed Number of Characters

If you arrange input in specific places along a line you can read a certain number of characters rather than a token. It is important to note that the full number of characters will always be read. If there are not enough characters on the line of input, then the remaining characters will be read from the next line of input. Tabs can also be a problem because they appear as multiple spaces but are only read as a single character.

Here is a program that reads names and phone numbers.

```
% The "EchoData" program
% Read and output names and phone numbers
var name : string (20)
var phone : string (8)
put "Name" : 20, "Phone" : 8
loop
    get skip
    exit when eof
    get name : 20, phone : 8
    put name : 20, phone : 8
end loop
```

Here is a possible Execution window with the output shown in a fixed- spacing characters.

```
Name                 Phone
Chin Yat Tat Aaron   973-2761
Chin Yat Tat Aaron   973-2761
Maltby Charles Ward  875-8637
Maltby Charles Ward  875-8637
(end of file here)
```

We placed the headings *Name* and *Phone* as a prompt over the columns where the entries are to begin; this saves you counting characters as you type in the data.

In this example, as we have in several previous examples, we are using a special key (or keys) that you can press *Control+Z* or *Control+D* to indicate that you are finished with a series of entries. The Turing predefined function eof responds to this. Until you press *Control+Z* or *Control+D,* eof has a value **false**. When you press *Control+Z* or *Control+D* and all previous characters have been read, eof becomes **true**.

The eof function is called a **Boolean** function: it has values false or true just as a condition has. It is used in this program in an **exit when** statement. When you are using the end-of-file to indicate the end of a stream of tokens or a stream of numbers you should precede the test with a **get skip** statement. This is

necessary to skip past white space, including the end-of-line character or blanks, that may precede the end-of-file marker.

12.3 Line-oriented Input

To read a whole line of text at a time we use an asterisk (*) instead of a precise number of characters to be read. Here is an example.

```
% The "ReadWholeLines" program
% Read and output a line at a time
put "Enter a series of lines of text, end with eof"
var line : string
loop
    exit when eof
    % Read a whole line
    get line : *
    put line
end loop
```

Here is a sample window.

```
Enter a series of lines of text, end with eof
How now brown cow
How now brown cow
Out of the house
Out of the house
(end-of-file here)
```

When the program reads a line at a time you do not need **get skip** before the eof test. This is because the asterisk method of reading automatically skips over the end-of-line character that appears at the end of each line. If you have a **get** instruction that reads numbers, tokens, or a fixed number of characters that is followed by **get** that reads a line you must have a **get skip** between them. This makes sure that you are positioned at the start of a line for the line-reading **get**.

12.4 Files on Disk

You can see from the last two programs that it is sometimes confusing to have the input and the output of text material on the same window. We can arrange to read input from a file on disk or to write output to a file on disk. We saw how to do this by redirecting the input from the disk rather than having it come from the keyboard, or by redirecting the output to the disk rather than having it go to the window. This is done by Turing Environment commands.

You can arrange for the program itself to explicitly use files. This allows us to have several input or output files at a time. Each input or output file must be given a **stream number** as well as a name which we assign to it. This stream number is placed after a colon following a **get** or **put** keyword. Before a file is referred to in a program it must be **opened** using the **open** statement which assigns a stream number to the file. If the **open** statement is not successful a non-positive stream number (≤ 0) is assigned. We can detect this by an **assert** statement.

Here is a program that reads a file which is another program and outputs the program with the lines numbered.

```
% The "ListEchoFile" program
% Reads lines of text from file "echo" and numbers them
var streamNumber : int
% Open the file "Echo" for reading using "get"
open : streamNumber, "Echo", get
% Test that the open procedure has been successful
assert streamNumber > 0
var lineNumber := 0
var line : string
loop
    exit when eof (streamNumber)
    get : streamNumber, line : *
    lineNumber := lineNumber + 1
    put lineNumber : 3, " ", line
end loop
```

The sixth line of the program uses the **open** statement to establish that the file called *Echo* will be given a stream number; the ÒgetÓ which is the last entry of the statement means that you will be using the **get** statement to read from the file. If you wanted to output to the file using **put**, you would have used the word ÒputÓ here. When you are done with the file, you should close it with the statement

close : *streamNumber*

You can reopen it for other purposes later. The Turing statement **close** requires only one item after the colon, namely the stream number of the file to be closed. All files are closed automatically at the end of the execution of a program so it is not essential to close a file unless you plan to reopen it for a different purpose.

We will be looking first at **sequential files**; each thing you get (or output) from (or to) the file is the item following the previous item. Items are read or written in sequence starting from the first of the file.

In the loop of the *ListEchoFile* program the exit occurs when the end of file of stream *streamNumber* occurs. That is why *streamNumber* appears in parentheses after eof. Every file that you create and place on the disk, either using the editing system of the Turing Environment or by writing it in a program, automatically has an end-of-file marker placed after the last character of the file. In the **get** statement the stream number of the file is placed after a colon and followed by a comma. The single input item here is given as *line* :* which means as before that you want to input all the characters up to the end of the line (to the Return). These will be stored in the variable *line*.

After a line is read, the value of *lineNumber* is increased by 1 and then it and the line are output by the **put** statement. If you do not know the name of the file that you want to output with line numbers, or you do not want to call your file *Echo*, you can write:

```
% The "ListAnyFile" program
% Reads a file and outputs its lines numbered
```

```
var fileName : string
var streamNumber : int
put "Enter name of file: " ..
get fileName
open : streamNumber, fileName, get
 (the rest as before)
```

12.5 Reading one File and Writing Another

This program reads words from a file called *Script* and outputs all the four-letter words from *Script* onto a file called *Censor*. It then reads and displays in the window what is in the *Censor* file.

```
% The "Purge" program
% Find and output all four-letter words in script
% Assumes no punctuation marks in text
% Assign a stream number to the input file Script
var script : int
open : script, "Script", get
assert script > 0
% Assign a stream number to the output file Censor
var censor : int
open : censor, "Censor", put
assert censor > 0
var word : string
loop
    get : script, skip
    exit when eof (script)
    get : script, word
    if length (word) = 4 then
        put : censor, word
    end if
end loop
% Close censor file and open for reading
close : censor
open : censor, "Censor", get
assert censor > 0
% Read and output censor file to the window
```

```
loop
    get : censor, skip
    exit when eof (censor)
    get : censor, word
    put word
end loop
```

Notice that we had to close the *Censor* file after writing it then open it for reading. Again, when you are reading a token-at-a-time rather than a line-at-a-time from a file you must execute a **get skip** before a test for end-of-file in case there is blank space or a Return just before the end of the file. Here we used the same names for the stream numbers as for the files. This perhaps makes it easier to read but is not necessary.

Here is an example where the punctuation marks are removed from a text. It reads a file named *Text* and writes a file named *Expurge.*

```
% The "RemovePunctuation" program
% Eliminates punctuation marks from text
% Assumes no double quotes or parentheses
var text : int
open : text, "Text", get
assert text > 0
var expurge : int
open : expurge, "Expurge", put
assert expurge > 0
var line, output : string
loop
    exit when eof (text)
    get : text, line : *
    output := ""
    for place : 1 .. length (line)
        if index (",.:;'?!", line (place)) = 0 then
            output := output + line (place)
        end if
    end for
    put : expurge, output
end loop
```

The reason that the double quote sign was not included as one of the punctuation marks to be eliminated is that putting it inside double quotes in the index function is not the same as most other characters. It looks like the end of the quoted string if a double quote is included. We have used single quotes inside double quotes to avoid the problem. You can use a double quote inside provided you precede it by a backslash (\). For example, index (",.:;'\"?!", *word*). To include a backslash itself as a character you must precede it by another backslash (\\).

12.6 Text Formatting

You can use computers to rearrange the number of characters on a line in text. Here is an example that allows you to specify the maximum length of each line and then read text and output lines that are as long as possible within this limit.

```
% The "LeftJustify" program
% Reads file text and left-justifies it
var text : int
open : text, "Text", get
put "Enter length of output lines: " ..
var maxLength : int
get maxLength
var word : string
var lineLength : int := 0
loop
    get : text, skip
    exit when eof (text)
    get : text, word
    if lineLength + length (word) < maxLength then
        put " ", word ..
        lineLength := lineLength + length (word) + 1
    else
        put ""          % Start new line
        put word ..
```

```
        lineLength := length (word)
    end if
end loop
put ""   % End last line
```

This is a very simple text formatting program. Many programs arrange to make the characters at the ends of lines line up on the left and on the right. Ours only left justifies the line. When you right justify a line as well, extra spaces must be placed between words to make it come out. This can look peculiar. Often words are hyphenated to improve appearances. The hyphenation program is also rather tricky.

12.7 Simple Language Translation

Computers can be used to translate from one language to another. A Turing compiler translates programs written in the Turing language to the language of the particular computer that it is running on. Here is a simple translation program that changes English into Pig Latin. The translation rules are very simple: each word that begins with a vowel has *ay* added to the end of it; each word that begins with a consonant has the consonant moved to the end of the word and then has *ay* added.

Here is the program.

```
% The "PigLatin" program
% Reads a text and translates to Pig Latin
% Assumes no capital letters and no punctuation
var fileName : string
put "Enter name of file where English is stored"
get fileName
var inFile : int
open : inFile, fileName, get
assert inFile > 0
put "Enter name of file where Pig Latin is to be stored"
get fileName
```

```
var outFile : int
open : outFile, fileName, put
assert outFile > 0
var word : string
loop
    get : inFile, skip
    exit when eof (inFile)
    get : inFile, word
    if index ("aeiou", word (1)) = 0 then
        put : outFile, word (2 .. *) + word (1) + "ay "
    else
        put : outFile, word + "way "
    end if
end loop
```

12.8 Exercises

1. Write a Turing program to read a series of lines and count the number of words read before the word ÒstopÓ is reached. What does the output look like if you try to output a word at a time like the *EchoWords* program of this chapter? Use input redirection to take the input from a file called *Text* instead of from the keyboard.
2. Write a program that takes a text file stored on disk under the file name *NewText* and outputs the text with exactly 7 words on each line.
3. Using the same input text file (*NewText*) as the previous question, write a program that outputs words on a line until the next word would make the line longer than 60 characters then starts a new line. Use the *LeftJustify* program as a basis but arrange that the output lines are right justified rather than left justified.
4. Write a program that computes the average length of words in a text. Assume that there are no punctuation marks in the text.
5. Use the *RemovePunctuation* program of this chapter to remove punctuation marks from a text. Be sure the

RemovePunctuation program is stored on your disk as you may need it. Try to depunctuate the *RemovePunctuation* program itself.

6. Write a program to read one line of text at a time from a file on disk and output the line if it contains the word ÒvarÓ. In this way you can spot all the declarations in a program. Use it to find the declarations in the *RemovePunctuation* program.

   Write the program in such a way that you ask the user what word they want to search for. The lines containing the pattern should have a line number so they can be easily found in the original text.

7. Write a program to change any ÒorÓ endings of words to ÒourÓ. For example it should change ÒcolorÓ to ÒcolourÓ. Improve it so that it does not affect the word ÒorÓ itself.

8. Write a program to change all capital letters in a word to little letters. To do this we use the two predefined Turing functions ord and chr. These allow us to convert between the characters and their ASCII code values. A to Z have values 65 - 90, whereas a to z have values 97 - 122. To change *letter* from a capital to little use these statements:

```
% Test if letter is a capital letter
if "A" <= letter and letter <= "Z" then
    % Change to corresponding little letter
    letter := chr (ord (letter) + (ord ("a") – ord ("A")))
end if
```

9. Write a program to read the words in *NewText* and output the words ending with a vowel to a file called *NewTextVowels*. The program should then display the contents of *NewTextVowels* in the Execution window with double quotes (Ò Ó) around each word.

10. Write a program that reads text from a file specified by the user and outputs every other line of text, starting with the second line, to another file specified by the user.

11. A UFO was found circling our area and upon investigation it was found that the aliens on board were sending coded messages to their superiors on planet L-E-N. The top

investigating team found that they were using the following method.

a) If the character was a digit greater than 0, then output the letter in our alphabet that falls in that order number. For example Ò3Ó gives ÒcÓ, Ò5Ó gives ÒeÓ.

b) If the character was a letter, then it output the next letter in the alphabet. If the character was a ÒzÓ or ÒZÓ, then ÒaÓ or ÒAÓ was output.

c) Characters other than digits or letters were left as is.

Create a program that replicates this alien code. Use any text file you have and use it as input for the program. Output the coded information to a file called ÒAliensÓ.

12. Write a program to run an analysis of vowel usage in any file. Open any text file and analyze the data using the following criteria. Count the number of times each vowel is used in the file. Upper and lower case letters must both be counted. At the end of analysis, output the number of a's, e's, i's, o's and u's. Also output the percentage of usage based on the number of vowels and of letters in general. Lastly, give the percentage of consonants used. Look at the example for clarification.

It is the time for all good children to go to school and learn Turing.

a's = 2	10% of vowels	3.6% of all letters used
e's = 4	20% of vowels	7.3% of all letters used
i's = 5	25% of vowels	9.1% of all letters used
o's = 8	40% of vowels	14.5% of all letters used
u's = 1	5% of vowels	1.8% of all letters used
Percentage of consonants used		63.7% of all letters used

13. Modify the program in Exercise 12 to ask the user for any character and the name of a file. It should analyze the file to find the number of occurrences of that character in the file as well as the percentage use for this letter in the total text.

12.9 Technical Terms

token-oriented input
line-oriented input
text processing
text formatting
right justified
left justified
stream number of file
open **statement**
close **statement**
sequential file
echoing input data
backslash
language translation

Chapter 13

Program Structure

13.1 Structure Diagrams

All programs are made up of three basic constructs: the sequential execution construct, the repetition construct, and the selection construct. We can represent these three by structure diagrams.

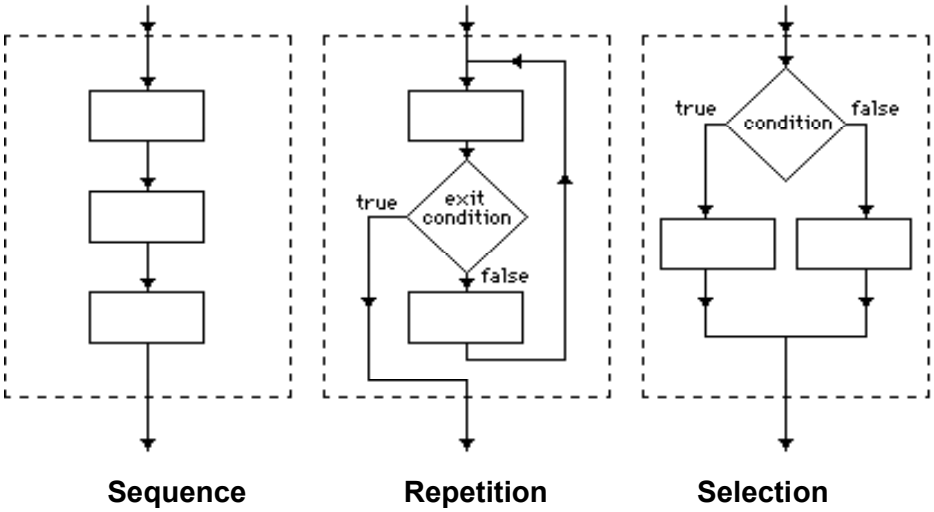


Figure 13.1 Structure Diagrams

As you can see the arrows on the lines joining the boxes give the sequence of execution. In the sequential execution diagram we show three boxes with an arrowed line coming into the first box, one between each of the next boxes, and one leaving the third box. We could think of another box, the one shown by the dotted outline, as containing the whole structure. There is one entrance to the structure and one exit. This is true of the other two constructs as well. This is the great secret of **structured programming**. Any one of the rectangular boxes in a structure diagram could have appeared inside another structure. They all fit inside each other.

In the diagrams for repetition and selection there is another kind of box beside the rectangular box. It has a diamond shape and from the diamond-shaped boxes there are two exits as well as one entrance. No nesting is possible for these boxes. You cannot put one inside the other.

13.2 Nested Structures

Here is a structure diagram for a nested structure of one repetition inside another or as we sometimes say one loop nested inside another.

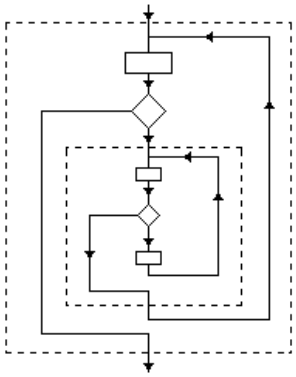


Figure 13.2 Structure Diagram for Nested Loops

You can see that structure diagrams can look quite complicated. It is easy enough to see the structure here because we have used a smaller scale for the inner loop. Usually diagrams like this are drawn so that all boxes are much the same size then the same diagram would look like Figure 13.3.

It is not so easy to see that it is a loop within a loop. Notice that in the diagram no arrowed lines ever cross each other. This is true of most diagrams for programs made of the three basic constructs.

Before about 1972 programmers were not aware of the enormous value of structured programming. In fact the languages available then like Fortran and Basic did not have the selection construct as part of the language. As well, they permitted exiting from loops to points other than the next statement after the loop.

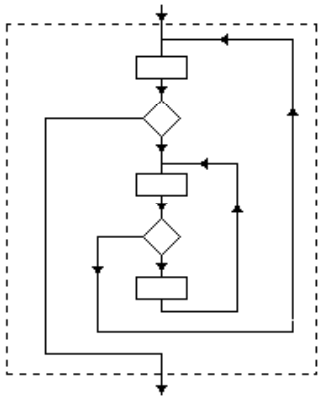


Figure 13.3 Structure Diagram with all Boxes Same Size

In those days people thought that structure diagrams were essential to understanding the operation of a program. And they may have been right because you could create very messy programs. Certainly the arrowed lines crossed over each other frequently in their diagrams. They were called **flow diagrams** or **flow charts**. The tangle of lines in these early diagrams led later enlightened programmers to call the early programs Òspaghetti programsÓ. Most programmers now do not draw structure diagrams because the structure is very evident from the program itself if you always indent the body of loops and other nested parts of the program.

13.2.1 A Loop Nested Inside a Loop

Here is a program with one loop nested inside another. It outputs a repeated triangular pattern.

```
% The "Border" program
```

```
% Outputs 5 triangles
% Each triangle has 6 lines
const row := "******"
for triangle : 1 .. 5
    for line : 1 .. 6
        put row (1 .. line)
    end for
end for
```

Notice that the body of the outer counted loop, the one with the index *triangle*, is indented four spaces. The body of the inner loop, the one with the index *line* is indented four more spaces. You can see right away which **end for** goes with which **for**. The output for this program is

```
*
**
***
****
*****
******
*
**
***
****
etc.
```

Each line of the pattern is simply a substring of the string constant *row*.

13.2.3 More Complicated Nesting of Structures

Here is a program that alternatively outputs a triangle with its wide part at the top then with its wide part at the bottom.

```
% The "FancyBorder" program
% Outputs alternating triangles
const row := "******"
for triangle : 1 .. 5
    if triangle mod 2 = 0 then
        for line : 1 .. 6
```

```
            put row (1 .. line)
        end for
    else
        for line : 1 .. 6
            put row (line .. *)
        end for
    end if
end for
```

You can see the structure clearly from the program. Nested inside the outer **for** loop is an **if..then..else** construct. Inside the **then** clause of the selection construct is a **for** loop which outputs the same triangle as in the *Border* program. Inside the **else** clause is nested a **for** loop which outputs a triangle like this

```
******
*****
****
***
**
*
```

The type of triangle output depends on the value of *triangle* **mod** 2**.** The operator **mod** gives us the remainder when one integer is divided by another. This will be zero for even values of *triangle* so the first triangle output will be the new type, the second the old type, and so on. Try drawing the structure diagram for this program. Does it help you to understand the structure?

13.3 Structure Diagram for elsif

The **elsif** is a relatively recent programming language feature. Before it was introduced you had to use nested **if..then..else** constructs.

Here is a program written as if **elsif** did not exist.

```
% The "RateMarks" program
```

```
% Reads in a series of marks, ending with -1
% Outputs comments about performance
var mark : int
put "Enter -1 to signal end of series of marks"
loop
    put "Enter mark: " ..
    get mark
    exit when mark = – 1
    if mark >= 80 then
        put "Excellent"
    else
        if mark >= 70 then
            put "Good"
        else
            if mark >= 50 then
                put "Satisfactory"
            else
                put "Poor"
            end if
        end if
    end if
end loop
```

Notice that we have nesting of one **if..then..else** inside another which in turn is nested in a third. There are three **end if**s. The **then** and **else** clauses of the inner **if..then..else** are indented four levels of indentation, once for the **loop** and three for the selection. You can draw the structure diagram for this if you like.

Here is the same program with **elsif**s.

```
% The "RateMarks2" program
% Read in a series of marks, ending with -1
% Output comments about performance
var mark : int
put "Enter -1 to signal end of series of marks"
loop
    put "Enter mark: " ..
    get mark
    exit when mark = – 1
    if mark >= 80 then
```

```
        put "Excellent"
    elsif mark >= 70 then
        put "Good"
    elsif mark >= 50 then
        put "Satisfactory"
    else
        put "Poor"
    end if
end loop
```

Here there are only two levels of indentation necessary instead of four and only one **end if**. Here is a structure diagram for the **elsif** structure.

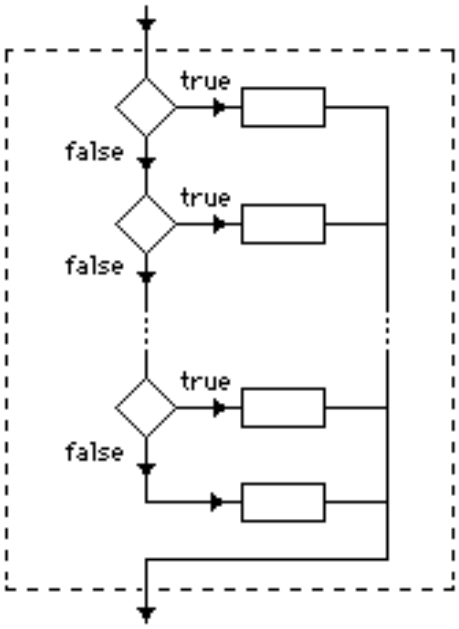


Figure 13.4 Structure of elsif

This is sometimes called a **cascaded if** statement.

13.4 Declaration of Variables and Constants Inside Constructs

We have so far declared our variables and constants outside of either repetition or selection constructs. They can then be used anywhere in a program. We have learned that the index of a counted loop is not known outside the scope of the loop.

We can arrange to have variables or constants known only within the scope of these constructs. All we need to do is declare them inside the construct. This is not something we do frequently but it is useful when we want to prevent access to the variable outside the construct.

If a variable or constant is declared inside a loop, then it is known until the end of the loop. If they are declared in the **then** clause of an **if..then..else** the scope is the **then** clause. Similarly for the **else** clause. Here is a program segment which swaps the numbers stored in variables *number1* and *number2* if the value stored in *number1* is larger than that in *number2:*

```
if number1 > number2 then
    const temp := number1
    number1 := number2
    number2 := temp
end if
```

Here the constant *temp* is declared in the **then** clause as having the value stored in *number1*. This permits the swap to take place. The value of the constant *temp* is not known outside the **then** clause. We could have declared it as a variable instead of a constant but since it does not change in this scope it is better to call it a constant.

Here is a program that calculates a user-specified Fibonacci number. Fibonacci numbers are a sequence of numbers where each number is the sum of the previous two elements in the sequence. The first two elements are set as 1 and 1. The third is 2 (1 + 1), the fourth 3 (1 + 2), the fifth 5 (2 + 3) and so on.

The declaration of the elements used to calculate the Fibonacci number is in the **else** clause and the declaration of the temporary variable used in the calculation is in the loop itself.

It is important to remember that one purpose of placing declarations inside a construct is to make the program easier to read and follow. Thus it is important to make it easy to find the declarations of variables that occur in the program. In practice, this means either placing the declarations near the top of the program or placing them within a page of the variable's use in the program.

```
% The "Fibonacci" program
% This outputs a fibonacci number
var number : int

put "Enter the element in the Fibonacci sequence to calculate: " ..
get number

if number <= 0 then
    put "The number should be positive"
else
    var element1 : int := 1
    var element2 : int := 1
    for i : 3 .. number
        var temp := element1 + element2
        element1 := element2
        element2 := temp
    end for
    put "Element number ", number, " is ", element2
end if
```

13.5 Design of Programs

So far the programs that we have written have been short. When we try to create larger programs it is best to develop the design of the program in stages. First of all you must decide what the program is to accomplish. You must work out detailed

specifications for the form the input data is to take and the form in which you want the output data. You will be **solving the problem** of transforming the input data into the output data.

The details of how this is to be done, what we call the **algorithm** for performing the transformation, will be worked out step-by-step. The result will be a Turing program by which the computer can accomplish what you want done.

The original specification of the problem will be in English. The solution will be in Turing. As you move to refine the process step by step, what you are creating is gradually transformed from the English language statement of what is to be done to the Turing language statement of how to do it.

At each stage our emerging design will be changing from English to Turing and at any intermediate stage can be a mixture, some things in English some in Turing. This is called **step-by-step refinement** or the **top-down approach** to program development.

In the gradual transformation it is common to leave the English parts in the final Turing program as comments. This lets you see the logic of the program development process.

13.5.1 Controlling Complexity

We learn how to do simple things well then try to reduce complicated things to a collection of simple things. In this way we find we can cope with very large programs without getting lost in a maze of detail. As well as using the top-down approach to the design of larger programs we do something else to keep programs small. We break them up into **subprograms**. Each subprogram is then a manageable smaller program.

This is sometimes referred to as **modular programming** since the final program consists of a number of components or modules connected together to make a unit. In Turing the word **module** is used in a special technical sense which must be understood if you want to write really large programs that will work and give correct results. Subprograms will be discussed later.

13.6 Exercises

1. Write a program to produce a repeated border running down the page. The pattern should be like this:

```
*****
    *
    *
*****
*
*
*****
    *
```

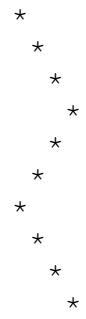
   and so on.

2. Write a program that produces a customized pattern. There are to be three basic components which can be selected by the user. As each component is output the computer asks which component is to be next. In order to keep the output pattern separate from the prompt and input information, output the pattern to a disk file called *Pattern*.
3. Try to create a program to output a repeated pattern across the page rather than down the page.
4. Write a program to draw a rectangular box outlined by asterisks where the number of asterisks across and the number down is input to the program. For example, a sample Execution window might be

```
How many asterisks across? 5
How many asterisks down? 6
Here is the pattern

*****
*   *
*   *
*   *
*   *
*****
```

5. Write a program to produce a zigzag pattern like this



   and so on where the lengths of the zig and the zag are input. Keep the zig bigger than the zag or the pattern will move off the page too quickly.
6. How do you decide whether to use an **elsif** construct or a **case** construct when a selection is to be made among more than 2 alternatives?
7. Write a program to exchange the values of the variables v1, v2, and v3 so that v2 holds the original value of v1, v3 holds the original value of v2, and v1 holds the original value of v3. Do this with 10 sets of numbers and for each set, the numbers should be entered by the user and new values for the variables should be printed after the exchange.

13.7 Technical Terms

structure of program
structure diagram
nesting of constructs
structured programming
flow diagram or flow chart
mod **operator**
declarations within construct
scope of constant or variable
swapping values
design of program
problem solving
algorithm

step-by-step refinement
top-down approach
transformation of data
controlling complexity
subprogram
modular programming

Chapter 14

Arrays and Other Data Types

14.1 Array Data Types

So far we have seen three types of data for single variables: **real** and **int** for numbers and **string** for strings of characters. In Turing there are a number of other data types. One of these types called the **array** type allows you to group single variables into a **data structure**. Often you are dealing with a lot of similar items of information, like the names of your friends. Usually you keep this sort of information in a **list** somewhere. To store such a list in the computer you could use a series of variables named *friend1*, *friend2*, *friend3,* and so on. In this case the ideal data type is not a series of single variables but one variable of **array** data type. For example, if you declare a variable *friend* as an array type by the declaration

```
var friend: array 1..100 of string
```

you would have established memory locations for 100 friend's names. The different individual locations are referred to as *friend* (1), *friend* (2), *friend* (3), and so on. This makes the declaration simpler than writing

```
var friend1, friend2, ..., friend100 : string
```

It also makes it easier to manipulate the list in the computer.

Arrays can be of any data type. Here is the declaration for an array of ten integers.

```
var number : array 1 .. 10 of int
```

Here is the declaration for an array of ten real numbers.

```
var decimal : array 1 .. 10 of real
```

The upper and lower bounds of an array can be integers. The upper bound can also be an integer variable so that the size of the array can be changed at run time.

Here is a program that declares an array with a user-specified size.

```
var size : int
```

```
put "Enter the array size : " ..
get size
var numbers : array 1 .. size of int
```

14.2 Manipulating Lists

Here is a program that reads a list of names and outputs the list in reverse order.

```
% The "ReverseNames" program
% Reads a list of names and outputs in reverse
var howMany : int
put "How many names are there? " ..
get howMany
var name : array 1 .. howMany of string (20)
put "Enter ", howMany, " names, one to a line"
for i : 1 .. howMany
    get name (i) : *
end for
put "Here are the names in reverse order"
for decreasing i : howMany .. 1
    put name (i)
end for
```

Notice that the variable *name* is an array of strings of maximum length 20. This saves space since the default length of strings is 256. The number of elements in the array is not known until the program executes. At that time you tell it how many there are. This is known as dynamic declaration of an array.

Here is an Execution window.

```
How many names are there?
4
Enter 4 names, one to a line
Mark Mendell
Steve Perelgut
Chris Stephenson
```

```
Inge Weber
Here are the names in reverse order
Inge Weber
Chris Stephenson
Steve Perelgut
Mark Mendell
```

When the names are read into the array the **get** statement reads a line at a time. This is what the :* is for. The array element *name (1)* will have *Mark Mendell* as a value, *name (2)* will have *Steve Perelgut,* and so on. The second **for** loop is one where the index decreases by one for each iteration so the names are output in reverse order. It is important to note that the range of an array does not have to start with 1. For example, the range could be

var *name* : **array** 3 .. 10 **of string**

or

var *name* : **array** –2 .. 12 **of string**

14.3 When to Use an Array

Arrays are more complicated than single variables and, since our goal in programming is to keep things as simple as possible, we should not use an array unless we need to. In the first example we needed to read in an entire list of names before we could start to output the names in reverse order.

The next program uses an array but the array is not necessary. The program finds the average mark for a class in a computer science exam.

```
% The "ArrayAverage" program
% Reads a list of marks and computes their average
var count : int
put "How many students in class? " ..
get count
var mark : array 1 .. count of int
var sum := 0
```

```
put "Enter the marks for the students"
for i : 1 .. count
    get mark (i)
end for
for i : 1 .. count
    sum := sum + mark (i)
end for
const average := sum / count
put "Average mark is ", average : 6 : 2
```

In this example the two **for** loops can clearly be combined into one

```
for i : 1 .. count
    get mark (i)
    sum := sum + mark (i)
end for
```

Since the mark is added into the sum as soon as it is read there is absolutely no need to store it in an array. The program without an array would have the declaration

```
var mark : int
```

and the single **for** loop would be

```
for i : 1 .. count
    get mark
    sum := sum + mark
end for
```

Always be on the alert for cases when a data item can be processed as soon as it is read and need not be stored in the memory. In cases like that, an array is unnecessary. One of the most common uses of arrays is for data that is to be available in the memory so that any piece of it can be retrieved on request.

To make retrieval easy it is sometimes appropriate to sort the data into a systematic order, for example, into alphabetic order for a list of names. Arrays are frequently used to hold a list that is to be sorted.

14.4 Initialization of Arrays

Just as variables can be initialized to a value in their declaration so also can arrays. The declaration

```
var list : array 1 .. 5 of int := init (2, 7, 8, 6, 5)
```

will initialize the array values so that *list* (1) is 2, *list* (2) is 7, *list* (3) is 8, and so on.

Here are two more examples of array initialization.

```
var names : array 1 .. 5 of string := init ("Fred","Barney",
    "Wilma", "Betty", "Dino")
var decimal : array 1 .. 4 of real := init (0, 0, 0, 0)
```

If the array is large and the initial value of each element is the same then a **for** loop can be used to initialize the array.

```
var names: array 1.. 100 of string
var numbers : array 1 .. 100 of int
var decimals : array 1 .. 100 of real
for i : 1..100
    names (i) := ""
    numbers (i) := 0
    decimals (i) := 0
end for
```

14.5 Sorting an Array

There are many ways to sort a list. Some ways are much more efficient than others. Also some ways use very little space other than the space the original array takes.

Here is a program for sorting an array of integers less than 999 into ascending order. It uses a second array to do this and it is not an efficient method.

```
% The "SortArray" program
```

```
% Reads a list of positive integers and
% sorts it into ascending order
var count : int
put "How many integers in list? " ..
get count
var list : array 1 .. count of int
% Read list into array
put "Enter the numbers in the list"
for i : 1 .. count
    get list (i)
end for
% Declare second array for sorted list
var sortList : array 1 .. count of int
for i : 1 .. count
    % Find smallest remaining in list
    var smallest := 999
    var where : int
    for j : 1 .. count
        if list (j) < smallest then
            smallest := list (j)
            where := j
        end if
    end for
    % Store smallest in sortList array
    sortList (i) := smallest
    % Replace smallest in array list by 999
    list (where) := 999
end for
% Output sorted list
put "Here are the numbers in ascending order"
for i : 1 .. count
    put sortList (i)
end for
```

Notice that when the smallest is selected from the array *list*, it is replaced by a large integer, namely 999. This means it will never be selected as smallest again. When you have finished sorting, all the elements of the array *list* will be 999. The elements of the array *sortList* will be the integers in ascending order. This method of sorting is called **sorting by selection**.

14.6 Related Lists

Sometimes associated with one list there is another list. For example, a list of friends could have a list of phone numbers related to it. The first name in the one list would correspond to the first telephone number in the other list, and so on. In this case we would use two separate arrays to store the information.

Here is a program to look up a telephone number of a friend. The phone directory is two arrays: a series of names and their corresponding numbers. It is stored in a disk file called *Direct*.

```
% The "PhoneDirectory" program
% Reads a phone directory
% then looks up numbers in it
const maxEntries := 50
var name : array 1 .. maxEntries of string (20)
var phoneNumber : array 1 .. maxEntries of string (8)
var count : int := 0
% Assign a stream number to directory
var directory : int
open : directory, "Direct", get
assert directory > 0
loop
    get : directory, skip
    exit when eof (directory)
    count := count + 1
    assert count <= maxEntries
    get : directory, name (count) : 20,
        phoneNumber (count) : 8
    % All names will be of length 20
end loop
close : directory
put "There are ", count, " entries in directory"
var friend : string
loop
    put "Enter friend's name: " ..
    exit when eof
    get friend : *
```

```
    if length (friend) < 20 then
        % Pad with blanks to make string friend
        % of length 20 because strings must have
        % same length to be equal
        friend := friend + repeat (" ", 20 – length (friend))
    end if
    var place : int := 0
    % Search list for name
    for i : 1 .. count
        if name (i) = friend then
            place := i
            exit
        end if
    end for
    if place not= 0 then
        put "Phone number is ", phoneNumber (place)
    else
        put "Not in list"
    end if
end loop
```

In this example the names and phone numbers in the directory are formatted with 20 character positions for the name and 8 for the phone number. The **get** : *directory*, **skip** statement is needed to skip the end-of-line character between pairs of entries. As the entries of the directory are read into the two arrays *name* and *phoneNumber,* they are counted. The number of names is stored in the variable *count*. The arrays have been declared as having *maxEntries* entries so *count* must not be bigger than *maxEntries*. Notice that we are reading 20 characters for each name in the directory so that, if we expect the friend's name to match one of these, it must be padded with blanks to make a total of 20 characters.

14.7 Subrange Data Types

There is another way in Turing besides using an **assert** statement to make sure in the *PhoneDirectory* program that *count* has values only between 0 and *maxEntries* inclusive. This way is to declare *count* as a **subrange** type by this declaration

```
var count : 0 .. maxEntries := 0
```

The value of *count* is automatically restricted to the subrange and is also initialized to zero as before. If in the execution of the program an attempt is made to go beyond *maxEntries* an execution error is reported and the program stops. Subrange types can be used as ranges in a **for** loop or in an array declaration.

14.8 Boolean Data Types

The lookup process can perhaps be made clearer using a variable *found* which is a **Boolean variable**. Boolean variables can have only two values: **true** and **false**. You can assign either of these values to such a variable but you cannot input or output the value of a Boolean variable. They can be used anywhere a condition can be used: in an **if** statement or an **exit when** statement.

Here are some declarations and statements using Boolean variables.

```
var stat : boolean
stat := true
stat := not stat      % makes stat false
stat := not stat      % makes stat true
```

Here are two examples of a Boolean variable being used in place of a true/false condition.

```
exit when stat
exit when stat = true      % These two statements are
equivalent.

exit when not stat
exit when stat = false  % These two statements are equivalent.
```

Here is an example program using a Boolean variable. This program reads a list of names and eliminates all duplicate names. As each new name is read into the array *nameList* it is compared with all those already read in and if it is not found in the list it is added. The list without duplicates is then output. Here is the program.

```
% The "RemoveDuplicates" program
% Reads a series of names and eliminates duplicates
var nameCount : int
put "Enter number of names " ..
get nameCount
put "Enter a series of ", nameCount, " names"
var nameList : array 1 .. nameCount of string (30)
var arrayCount := 1
for i : 1 .. nameCount
    get nameList (arrayCount) : *
    var found : boolean := false
    for j : 1 .. arrayCount – 1
        if nameList (arrayCount) = nameList (j) then
            found := true
            exit
        end if
    end for
    if not found then
        arrayCount := arrayCount + 1
    end if
end for
% Output list without duplicates
put "Here are the names without duplicates"
for i : 1 .. arrayCount – 1
    put nameList (i)
end for
```

14.9 Tables

When all the entries in a number of related lists are of the same data type you can use a single **two-dimensional array** or **table** to store the information. Suppose for a particular city you have a list of the airfares (super economy of course) to a number of other cities. You have such lists for every city that the airline flies to. Each entry in all the lists is a number of dollars. There will be as many lists as there are entries in each list. Usually we arrange lists like this as a table with rows and columns.

Here is such a table shown in fixed-spacing characters.

```
To      Chicago(1) New York(2)   Boston(3)  Toronto(4)
From

Chicago(1)      0         110         120          80
New York(2)   110           0          50         100
Boston(3)     120          50           0          90
Toronto(4)     80         100          90           0
```

Usually the order of the headings on the columns is the same as the order of the headings on the rows. Chicago is the heading of the first row and the first column. The entries on the diagonal are all zero; it does not cost anything to fly to your own city. This table is symmetric about the diagonal of zeros. We can declare such a two-dimensional array by this declaration.

```
var airfare : array 1 .. 4, 1 .. 4 of int
```

The first range 1 .. 4 refers to the number of rows, the second 1 .. 4 to the number of columns. To refer to the array element that is from Boston to New York we use

```
airfare (3, 2)
```

It is the element in the table in the 3rd row and 2nd column.

To initialize this two-dimensional array so all elements arte zero, you use nested **for** loops.

```
for row : 1 .. 4
    for column : 1 .. 4
        airfare (row, column) := 0
    end for
end for
```

Here is a program that would read in a table of airfares and store it in a file on disk called *Fare*. You enter the table a row at a time.

```
% The "FareTable" program
% Reads and stores an airfare table
put "Enter number of cities in table: " ..
var cityCount : int
get cityCount
var airfare : array 1 .. cityCount, 1 .. cityCount of int
% Read table into two-dimensional array
for i : 1 .. cityCount
    put "Enter next row of table"
    for j : 1 .. cityCount
        get airfare (i, j)
    end for
end for
% Output two-dimensional array to disk
% Assign a stream number to fare
var fare : int
open : fare, "Fare", put
put : fare, cityCount
for i : 1 .. cityCount
    for j : 1 .. cityCount
        put : fare, airfare (i, j) : 8 ..
    end for
    put : fare, ""
end for
```

You can also create three or even four dimensional arrays. This is done by adding more ranges in the declaration. Here is the declaration for a three dimensional array.

```
var statistics : array 1 .. 10, 1 .. 5, 1 .. 8 of real
```

14.10 Named Data Types

Sometimes a particular complex data type such as an **array** type or a subrange type occurs several times in the same program. It is possible to give the type a name and then just refer to this **named type** each time you use it. To declare a named type you use the form:

type name : description

In our *Fares* program we could have declared a named subrange type called *airports* by this declaration

type *airports* : 1 .. *cityCount*

if *cityCount* was a constant, rather than a variable. *cityCount* must be a constant because both bounds of a subrange must be known at **compile time** (in other words, before the program starts execution). Each time this subrange is used it can be referred to by the name *airports*. For example, the declaration for *airfare* could now be

var *airfare* : **array** *airports*, *airports* **of int**

Even the **for** loops could use the named subrange type to declare the range of indexes. For example,

for *i* : *airports*

Here is a program that uses a named range called *monthType* as both an index to an array and the range of a **for** loop.

```
% The "CalculateDaysInYear" program
% Calculate the number of days in a non-leap year
type monthType : 1 .. 12
var days : array monthType of int := init (31, 28, 31, 30, 31, 30, 31,
31,
    30, 31, 30, 31)
var totalDays : int := 0

for month : monthType
```

```
    totalDays := totalDays + days (month)
end for
put "There are ", totalDays, " days in a year"
```

Named types are used to greater effect when we have subprograms.

14.11 Exercises

1. Write a program to read in a list of names, sort the list, and then output the list in sorted order. This program could be divided into three steps as illustrated by the comments

   ```
   % Read list of names
     ...
   % Sort list of names
     ...
   % Output sorted list of names
   ```

   We will not worry about whether or not the sorting algorithm we use is an efficient algorithm. Perhaps you can have two arrays: one for the unsorted list and a second for the sorted list. Look for the alphabetically least in the unsorted list and put it in the first place in the sorted list. Then look for the next. If you blot out the first one you choose with some string like 'ZZZZ' it will never get chosen again. Do this until you have placed the whole list in order.
2. Write a program to read in a series of heights of people and output all those that are above average in height for the group.
3. The median value in a list of values is the value which has as many entries in the list with smaller values as it does with larger values. Write a program to determine the median value of a list of integers.

4. An inventory of the articles you have in a drawer is stored on a disk file under the file name *Drawer*. Write a program to read the file into memory then, when asked, to tell you whether or not there is a certain item in the drawer. A sample Execution window might look like this.

```
What are you looking for?
pen
There is a pen in the drawer
What are you looking for?
dime
There is no dime in the drawer
... and so on ...
```

5. Test the subrange feature of Turing by declaring that an integer must be between 1 and 10 then trying to read an integer outside this range.
6. Adapt the *RemoveDuplicates* program in this chapter to read a file from disk called *Family* which contains the first names of your relatives. (Be sure there are some duplicates.) The program is then to store the list, with duplicates eliminated, in a file called *Names*.
7. Use the program *Fares* in this chapter to store an airfare table for five cities in a file called *Fare*. Now write a program of your own to read the table into memory and use it to answer customers' questions about airfares. A sample Execution window might be for the four-city table in the chapter.

```
Where are you flying from?
Chicago
Where are you flying to?
New York
The fare is $110
```

8. Create a data file called *Marks* which has 10 records in it. Each record consists of a name of a student, and four integer values which are the marks of the student. The first mark is out of 25, the second is out of 10, the third is out of 50 and the fourth is out of 20. Write a program that reads the file into 5 related arrays:

```
var names : array 1..10 of string
var mark1, mark2, mark3, mark4 : array 1..10 of int
```

Declare three more arrays.

```
var evenlyWeighted, weighted, assigned : array 1..10 of real
```

These three will contain the evenly-weighted, weighted, and assigned averages for each of the students. To get the evenly weighted average, convert each mark to a percent and then take an average of the percents. If the marks are 10/25, 8/10, 30/50 and 14/20 then the percents are 40, 80, 60, and 70 respectively. The evenly-weighted average would be 62.5%, so the corresponding array element would be assigned 62.5.

To get the weighted average, take the sum of the marks and the sum of the total test marks available and express this as a percent. For the above example the calculation would be

100 * (10+8+30+14)/(25+10+50+20) or 59.0%.

The weighted average would be 59.0%, so 59.0 is assigned to the corresponding array element. Assign the better of the two above averages to the *assigned* array. Now display all the arrays in chart form in the window.

9. Modify the program in Exercise 8 so that the records are displayed in descending order according to the assigned average. Make sure that swapping the assigned average has not mixed up your data. All fields must be swapped to ensure that the integrity of your records has not been compromised.
10. Write a program that asks the user how many numbers they want generated. Randomly create and display these numbers. At the end of the run display the mean, median, and the mode. The mean is the average (sum of all numbers/number of numbers). The median is the center value of a sorted list (if there are an even number of numbers then the median is the average of the two middle numbers, if there are an odd number of numbers, the median is a single value). The mode is the number or numbers which occur the most often. In the list below 3 and 4 are the modes as they both occur 4 times.

```
1 2 2 3 3 3 3 4 4 4 4 5 6 7 8 9 9 9
```

11. Write a program to output the song ÒOld MacDonald Had a FarmÓ with 10 verses. Use two arrays, one for the animals and one for the sounds they make and initialize them both. Use a getch between each verse to clear the window. If the first three animals were cow, ass, and pig and the first three sounds were moo, heehaw, and oink then this is how the first three verses should appear. Take note of the ÒaÓ or ÒanÓ in front of the animal or its sound.

(verse 1)

Old MacDonald had a farm, e-i-e-i-o
And on that farm he had a cow, e-i-e-i-o
With a moo-moo here and a moo-moo there
Here a moo, there a moo, everywhere a moo-moo
Old MacDonald had a farm, e-i-e-i-o

(getch and verse 2)

Old MacDonald had a farm, e-i-e-i-o
And on that farm he had an ass, e-i-e-i-o
With a hee-haw here and a hee-haw there
Here a hee, there a hee, everywhere a hee-haw
With a moo-moo here and a moo-moo there
Here a moo, there a moo, everywhere a moo-moo
Old MacDonald had a farm, e-i-e-i-o

(getch and verse 3)

Old MacDonald had a farm, e-i-e-i-o
And on that farm he had a pig, e-i-e-i-o
With an oink-oink here and an oink-oink there
Here an oink, there an oink, everywhere an oink-oink
With a hee-haw here and a hee-haw there
Here a hee, there a hee, everywhere a hee-haw
With a moo-moo here and a moo-moo there
Here a moo, there a moo, everywhere a moo-moo
Old MacDonald had a farm, e-i-e-i-o

12. Ask the user for a series of words or tokens and have a sentinel to mark the end of the session. For each word output only the letters that appear only once in the word. Treat upper and lower case letters as equivalent.

sample input	sample output
apple	ale
Rory	oy
AaBbc	c
NNN	none is unique

13. Initialize an array with 26 single character elements to be the 26 letters of the alphabet. Randomly generate integers from 1 to 26 to indicate which of the letters you will be using. For example if 2 is generated you would use the letter b. Use this method to make a series of four-letter words. Ask the user how many words they want created and then output them in the window.
14. Modify the program in Exercise 13 to keep a count of which letters are chosen. Do this with one extra array. Output the number of times each letter has been chosen to create the words in the program.
15. Write a program to simulate the throw of a die 50 times and output the number of times each number was thrown. (Use the randint procedure .)
16. Repeat Exercise 4 but this time, the file should be named *Drawer2* and contain the quantity of each item. A sample Execution window might look like this:

```
What are you looking for?
pen
There are 4 pens in the drawer.
```

Modify the program to ouput

```
There is one pen in the drawer.
```

for the case of a value 1.

17. Write a program to input 10 integers and once all 10 integers are input, output a message indicating whether or not each of the 10 integers was even or odd.

14.12 Technical Terms

array **data type**
element of array
index of array
list
init
initialization of array
retrieval of data
sorting of array
sorting by selection
lookup in list or table
subrange data type
boolean **data type**
table
two-dimensional array
named data type
type **declaration**

Chapter 15

Music

15.1 Playing Musical Notes

Turing programs can be written to output music. The music can be played without anything in the Execution window or it can accompany a graphic display.

Musical notes can be played one-at-a-time or as a series of notes. To play a single note we use the predefined procedure play in the form

```
play (value-of-note)
```

where the value of note is its duration followed by its pitch. The pitch of a note is given by the letters A to G inclusive which represent the corresponding musical notes. (Little letters a to g can be used equally well.) If the letter is followed by a plus sign, the note is sharp; if by a minus sign, the note is flat. For example, we can play the note D and then an F sharp this way:

```
play ("D")
play ("F+")
```

These notes are played as quarter notes. The duration of a note is given by a single digit according to this table:

1 whole note
2 half note
4 quarter note
8 eighth note
6 sixteenth note

So this plays C as an eighth note:

```
play ("8C")
```

All subsequent notes will be eighth notes unless explicitly set to a different duration. The pitch is presumed to be in the middle octave which starts at middle C. The middle octave notes are

C D E F G A B

To represent C in the octave above the middle octave, we use the notation >C; to represent it in the octave below the middle, we use <C. Once either > or < have been used all future notes in that play instruction, or in subsequent play instructions, are shifted to the same octave. To prevent the notation for one note from influencing the value of the next, we should make a practice of returning to the middle octave after the note is played. For example, to play an eighth note with pitch A in the octave above the middle octave we would use

```
play ("8 > A<")
```

In these examples the value of a note has been a string of characters which are enclosed in quotes. We could also write

```
var note : string := "8 >A<"
play (note)
```

Now the parameter of the play procedure is a string variable.

Here is a program which plays random sixteenth notes in the C-scale in the middle octave.

```
% The "RandomNotes" program
% Plays sixteenth notes in C scale at random
const CScale := "CDEFGAB"
play ("6") % Set to play sixteenth notes
var note : int
loop
    randint (note, 1, 7)     % Pick a note at random
    play (CScale (note))
end loop
```

Since all notes are to be with the same duration we can give a play statement to set this for all following notes. Notice that the parameter of play giving the value of the note is a one-character substring of the string *CScale*.

15.1.2 Resting for a While

Sometimes we want an interval of silence in our music. This is called a **rest** or **pause** and is achieved by playing a ÒnoteÓ whose ÒpitchÓ is P (or p). The duration of the rest is given in the same way as the duration of a note. A half note rest is given as Ò2pÓ.

15.2 Playing a Series of Notes

A series of notes is given by catenating the notations for individual notes and placing the resulting string in a play instruction. If the duration is the same from one note to the next it need not be repeated. If the octave is the same you do not need to return to the middle and then go back to the same octave. For example, the instruction

```
play (">8A< >8B<")
```

which plays two notes, both eighth notes in the octave above the middle octave could be written more simply as

```
play (">8AB<")
```

Notice that we make a practice of returning to the middle octave even after a series of notes. Here are two instructions that play ÒMary had a little lambÓ

```
play ("4EDCDEE2E4DD2D4EG2G")
play ("4EDCDEEEEDDED2C")
```

15.3 Using the Keyboard to Make Music

Here is a program that lets you play music using the keyboard of the computer. It plays the notes of the C-scale when you press the keys 1–8. When you press 1 you get middle C, when you

press 2 you get D, and so on. When you press 8 you get C in the octave above the middle octave. All the notes have the same duration. Before you begin you will be asked to enter the digit that corresponds to the duration you want.

```
% The "Piano" program
% Makes the computer keyboard into a musical keyboard
cls
var duration : string (1)
put "Enter one digit to control duration"
put "You can enter 1, 2, 4, 8, or 6: " ..
get duration
play (duration)
put "You can now begin to play"
put "Play notes by pressing keys 1 to 8, any other key to stop"
var note : string (1)
loop
    getch (note)     % Wait until a key is pressed
    if note = "1" then
        play ("C")
    elsif note = "2" then
        play ("D")
    elsif note = "3" then
        play ("E")
    elsif note = "4" then
        play ("F")
    elsif note = "5" then
        play ("G")
    elsif note = "6" then
        play ("A")
    elsif note = "7" then
        play ("B")
    elsif note = "8" then
        play (">C<")
    else
        exit
    end if
end loop
```

Try playing a simple folk tune on your keyboard. Record the notes so you can use them to write a program to play the tune automatically.

15.4 Animation with Music

In this section we will try a more difficult problem. We will combine animated graphics with music. It is important that the music and the graphics be in time with each other; they must be synchronized. The particular problem we will attempt is a sing-along animation which displays the words of a song, one-line-at-a-time, and has a ball bounce from one syllable of the words to the next, as the corresponding note to be sung is played. This used to be a very common way of leading a sing song for a group of people sitting in a theatre.

The problem will be simplified by treating all syllables as strings of length 7 characters (including any hyphen). This means that the Òbouncing ballÓ can be placed above the first character of the string. If the line consists of 5 syllables with 5 corresponding notes, then the ball will be in column 1 for the first note, 8 for the next, 15 for the next, and so on, for 5 different positions.

We will not try to animate the ball's motion between these positions in our simple program. It will just disappear from one position and reappear at the next when the note is finished playing. Here there is no need to have a delay in the animation; that is accomplished by waiting for playdone to be **true**.

The predefined function playdone is a boolean function whose value is true if the last note of the preceding play procedure is finished. We will store the songs in a library of old favorites on the disk.

These data files will have this form

```
number of lines
{number-of-syllables-in-line syllables-separated-by-blanks
 notes-separated-by-blanks }
```

The individual line specifications are preceded by an integer which tells how many lines there are. Each line is preceded by the number of syllables in the line followed by those syllables, separated by blanks, then the corresponding notes, separated by blanks.

Here is what the data would look like for the first verse of the song ÒMary had a Little LambÓ

```
5
7  Ma- ry  had  a  lit- tle  lamb 4E  4D  4C  4D  4E  4E  2E
3  Lit- tle  lamb  4D  4D  2D
3  Lit- tle  lamb  4E  4G  2G
7  Ma- ry  had  a  lit- tle  lamb 4E  4D  4C  4D  4E  4E  4E
6  Its  fleece  was  white  as  snow 4E  4D  4D  4E  4D  2C
```

The description of the notes is given so that each note is specified independently of the previous notes or the following notes. This means that each will have a duration and a pitch. If the note is not in the middle octave then the pitch is preceded by the sign (or signs) necessary to shift octaves and followed by the complementary sign (or signs) to return to the middle octave.

The maximum length of the string needed to specify a note is seven characters, one for duration, one or two for pitch, two or four for octave shift and return. If there is a pause, a syllable with no characters (an empty string) is inserted in the line.

To do this you must use two double quotes with no characters between. The corresponding note will be a *p* (for pause) prefaced by a duration. Since the line of the song must fit onto the line in the window we will allow for a maximum of 11 syllables which would take up 77 character positions. Longer lines can be split into two.

The syllables of the lines will be stored in a two-dimensional array called *lineSyll* where each element is

lineSyll (*line*, *syllNo*)

The 3rd syllable of the 2nd line would be

lineSyll (2, 3)

We will read in the whole song from the disk before playing it so that any delays in getting it from the disk will not interrupt the playing of the song. Because the input is from the disk there are no prompts.

Here is a sketch of the program.

```
% The "singSong" program
% Plays songs for a sing-a-long
% Read song from disk
for line : 1 .. numLines
    % Display words of line on window row 10
    for syll : 1 .. numberOfSyll (line)
        % Plot ball over syllable
        % Play corresponding note
        % When playdone = true erase ball
    end for
    % Erase words of line
end for
```

The number of syllables in the lines will be stored in a one-dimensional array called *numberOfSyll* where the element *numberOfSyll* (3) would be the number of syllables in the third line.

Here is the complete program.

```
% The "SingASong" program
% Plays songs for a sing-a-long
% Read song from disk
var numLines : int
get numLines
var lineSyll : array 1 .. numLines, 1 .. 7 of string (7)
var lineNote : array 1 .. numLines, 1 .. 7 of string (7)
var numberOfSyll : array 1 .. numLines of int
for line : 1 .. numLines
    % Read number of syllables in line
    get numberOfSyll (line)
    % Read syllables in line
    for syllNo : 1 .. numberOfSyll (line)
```

```
        get lineSyll (line, syllNo)
    end for
    % Read notes in line
    for noteNo : 1 .. numberOfSyll (line)
        get lineNote (line, noteNo)
    end for
end for

cls
for line : 1 .. numLines
    % Display words of line on window row 10
    locate (10, 1)
    for syllNo : 1 .. numberOfSyll (line)
        put lineSyll (line, syllNo) : 7 ..
    end for
    % Plot ball over syllable
    for syllNo : 1 .. numberOfSyll (line)
        % Display ball over syllable in window row 9
        locate (9, 1 + (syllNo – 1) * 7)
        put "*" ..
        % Play corresponding note
        play (lineNote (line, syllNo))
        % When note is finished playing, erase ball
        locate (9, 1 + (syllNo – 1) * 7)
        put " " ..
    end for
    % Erase words of line
    locate (10, 1)
    put " "
end for
```

If the song is stored in the file called *Mary* the program could be executed using the run command that redirects the input to be from that file.

The combination of music and graphics can be used to enhance many computer applications.

15.5 Exercises

1. Write a program to read in a string of symbols that represent a song you know and then play it repeatedly.
2. Arrange that there is a pause of 10 full notes duration between repetitions of the song of question 1.
3. Arrange that you can interrupt the repetitions of the song of question 1 between repetitions by typing the letter *q* for quit.
4. Write a program to play a song and display the words a line-at-a-time (for a song of at least four lines). Do not display the words of the next line until the notes of the previous line have finished playing.
5. Prepare the data (notes and words) for a song to be in the sing-a-long library for the program *SingASong*. Store it on the disk. Now run the *SingASong* program using your song as input data.
6. Modify the *Piano* program to use a **case** statement instead of a cascaded **if**...**then**...**elsif**...**else** statement.
7. Write a program like the *Piano* program that lets you sound any of the 12 notes on the piano (including the black keys) in the middle octave at any duration that you want. This is not to be a program for playing at a normal rate but rather one that will provide you with a note to start a group singing without accompaniment.
8. Modify the *BrownianMotion* program in the chapter graphics to make a more interesting noise when the particle reaches the edge of the window.
9. Write a program that sounds notes in the scale of three flats at random. This scale is E-flat major. The E-flat scale can be played with the instruction

```
play ("E–FGA–B–>CDE–<")
```

10. Write a program that plays Beethoven's ÒOde to JoyÓ and at the same time changes the color of the window every third note. Here is a string that represents the ode:

```
8bb>cddc<baggabb6p
a4a8bb>cddc<baggaba6p
g4g8aabga6b>c<8bga6b>c
<8baga4d
8bb>cddc<baggaba6pg4g
```

    Store the song on the disk in the one-note-at-a-time form. Read it into an array before you play it.

11. Modify the program of question 10 to change window color randomly so that the number of notes (or pauses) between color changes varies between 2 and 6.
12. Improve the *SingASong* program by inserting another position of the bouncing ball between syllables. Put this on row 8, half way between syllable positions.
13. Write a program to play the C major scale in 8th notes in 4 octaves ascending and descending.

15.6 Technical Terms

octave
pitch
duration
scale
play
rest
pause
flat
sharp
playdone

Chapter 16

Subprograms

16.1 Functions

So far all the programs shown in the book have been reasonably small. When you try to write a longer program there are more opportunities to get confused and to have programs that are not easy to understand. So we try always to keep our programs small by subdividing larger programs into a number of subprograms.

In Turing there are two kinds of subprograms: **functions** and **procedures**. A function is a subprogram that produces a single value, like the square root of a number or an amount of money to the nearest cent. A procedure can do much more such as read in an array or plot a graph.

16.1.1 Predefined Functions

We have already used functions in many of our programs, not functions that we programmed ourselves, but functions that are part of the Turing system. These have been **predefined** and their definitions stored in the computer. Some of these functions require real parameters and produce real values.

For example, the function for square root would have a definition that would begin with a header line such as

```
function sqrt (r: real): real
```

This function returns the positive square root of *r*, where *r* is a non-negative value. If we wrote a program such as

```
put sqrt (36)
```

the output would be 6. Integer values are acceptable as actual parameters where the formal parameter *r* is real. The value of the square root is computed as a real value but, because in this case it is an integer, it is output as such by the **put** statement. This is the same as when we write

```
put 100 / 4
```

The result of the division is real but it is output as 25.

Whenever you call a function, you must use the result. In other words, the line

```
sqrt (36)
```

is illegal, just as having the line

```
5 + 10
```

is illegal. You must use the value of the function in some way. The value can be as part of an expression, assigned to a variable, or output as the result of the function call.

When you are passing a parameter to a function or a procedure, you must be able to assign the parameter's data type to the formal parameter's data type. For example, you could not pass a **string** to the *sqrt* function because you cannot assign a **string** to a **real**. You can, however, pass an integer to the *sqrt* function because you can assign an **int** to a **real**.

In the appendix of this book there is a list of all the predefined functions in Turing. A number of these are mathematical functions which are very useful for scientific calculations. These are sin, cos, arctan, ln, exp, and so on.

Other functions are used to convert a parameter of one data type into a value of another data type. They are called **type transfer functions**. For example,

```
put round (5 / 3)
```

will produce the result 2. A real value, the result of division, is converted to an integer. We can use one function in another function's definition provided that the first function is either predefined or its definition precedes the second function's definition in the program.

16.1.2 Type Transfer Functions

Type transfer functions are used to convert one data type to another data type. For example, the *round* function is used to convert a real number to an integer. Turing contains a complete set of these functions.

Here are the type transfer functions covered earlier in the book.

round	round to nearest integer
ceil	round up to nearest integer
floor	round down to nearest integer
ord	convert a single character to an integer
chr	convert an integer to a single character
strint	convert a string to an integer

Here are some other useful type transfer functions.

intstr	convert an integer to a string
intreal	convert an integer to a real
strreal	convert a string to a real
realstr	convert a real to a string

There are also two functions to help convert strings to integers and reals. These are strintok and strrealok. Both of these functions take a string and return **true** if the string can be converted to a number.

Various functions allow you to change between integers and strings. For example, chr (7) produces a one-character string, namely the digit 7. In ASCII code chr (65) has a value A; chr (90) is Z. The function intstr (*i, w*) yields the string of length *w* corresponding to the integer *i* (right-justified). The function strint (*s*) yields the integer equivalent of string *s*. These are all defined in the appendix.

Here is a program that illustrates the use of type transfer functions. The result of the put instructions are shown as comments on the lines where they occur.

```
put round (1.55)                % 2
```

```
put ceil (1.33)              % 2
put floor (1.789)            % 1
put ord ("A")                % 65
put ord (65)                 % A
put strintok ("123")         % true
put strintok ("12ab")        % false
put strealok ("58.7")        % true
put strealok ("?")           % false
var word, neword: string
word := "34"
var num, newnum : int
num := strint (word)
put word + word              % 3434
put num + num                % 68
neword := intstr (num)
put neword + neword          % 3434
word := "3.14"
put word + word              % 3.143.14
var deci : real
deci := strrea l (word)
put deci + deci              % 6.28
neword := realstr (deci, 5)
put neword + neword          % ^3.14^3.14
```

16.1.3 User-created Functions

Here is an example of a function subprogram that we call *square* that gives the square of a number.

```
function square (number : real): real
    result number ** 2
end square
```

This is a very small subprogram. The first line of the **function definition** is a header that indicates that it is a function and gives its name. In parentheses after the name are listed the **formal parameters** of the function with their data types. Parameters are a way that a subprogram communicates with the main program. Through the parameters, information flows into the function subprogram from the main program. Here there is only one

formal parameter *number* and it is of type **real**. After the parentheses is a colon then the data type that the value of the function will have. Here the square will be **real**.

The last line of a function definition must have the keyword **end** followed by the function's name. In between the header and the **end** is the **body** of the function subprogram. Here the body is only one statement and it gives the value of the function using the keyword **result** in front of the value. This is such a small subprogram that it is hardly worth bothering about but it shows all the basic ideas.

Now we must use the function in a program to show how that is done. This program outputs the squares of the numbers from 1 to 10.

```
% The "TableOfSquares" program
% Outputs a table of squares

function square (number : real) : real
    result number ** 2
end square

for i : 1 .. 10
    put i : 2, square (i) : 4
end for
```

Here is the Execution window shown in fixed-spacing characters.

```
 1   1
 2   4
 3   9
 4  16
 5  25
 6  36
 7  49
 8  64
 9  81
10 100
```

To use the function you just give its name followed in parentheses by the **actual parameter** whose square is wanted. Here the parameter is the loop index *i*. The value of the actual parameter *i* is used as the value for the formal parameter *number* in the function definition. The function definition is included in the main program anywhere as long as the definition appears before any use is made of it. We will put blank lines around a subprogram definition to make the program easy to read. You can see at a glance where each piece begins and ends.

Here is a program using another function called *roundCent*.

```
% The "ComputeInterest" program
% Computes bank interest to nearest cent

function roundCent (amount : real) : real
    % Round amount to nearest cent
    result round (amount * 100) / 100
end roundCent

var balance, interestRate : real
put "Enter balance: " ..
get balance
put "Enter current interest rate in percent: " ..
get interestRate
const interest := roundCent (balance * interestRate / 100)
balance := balance + interest
put "New balance = ", balance
put "Interest = ", interest
```

Here is a sample Execution window.

```
Enter balance 576.37
Enter current interest rate 7.8
New balance = 621.33
Interest = 44.96
```

Here is another version of the interest program using a second function *calcInterest*. The *calcInterest* function uses the *roundCent* function. This is possible because *roundCent* is declared before *calcInterest*.

```
% The "ComputeInterest2" program
% Computes bank interest to nearest cent

function roundCent (amount : real) : real
    % Round amount to nearest cent
    result round (amount * 100) / 100
end roundCent

function calcInterest (amount, interestRate : real) : real
    result roundCent (amount * interestRate / 100)
end calcInterest

var balance, interestRate : real
put "Enter balance: " ..
get balance
put "Enter current interest rate in percent: " ..
get interestRate
const interest := calcInterest (balance, interestRate)
balance := balance + interest
put "New balance = ", balance
put "Interest = ", interest
```

16.1.4 A String-valued Function

In this section we will look at the definition of a function that has exactly the same purpose as the predefined function repeat in order to let you see how the repeat function might be defined. In the appendix the repeat function's description is:

> repeat (*s*: **string**, i: **int**): **string**
> If *i* > 0, returns *i* copies of *s*
> joined together, else returns the empty string.
> Note that if *j* > 0, length *(*repeat *(t, j)) = j* * length *(t).*

Here is the definition of a function called *copy* that does the same thing as repeat.

```
function copy (s : string, i : int): string
    var copies : string := ""
```

```
    if i > 0 then
        for j : 1 .. i
            copies := copies + s
        end for
    end if
    result copies
end copy
```

The variable *copies* is local to the function definition and is used to build the string that is returned as the result. If this definition is included in the program the result of the statement

```
put copy ("HiHo ", 3)
```

will be

```
HiHo HiHo HiHo
```

the same as you would get for this statement

```
put repeat ("HiHo ", 3)
```

16.2 A Procedure with No Parameters

Functions return a value and use their parameters to compute their result. Procedures can return many values and usually have one or more parameters which are used in computing the values returned. There are other possibilities. For example, a procedure can cause output and may not have any parameters at all.

Here is a procedure which will output a triangle shape.

```
procedure triangle
    % Outputs a triangle of asterisks
    for i : 1 .. 5
        put repeat ("*", i)
    end for
end triangle
```

Notice that the header of the procedure begins with the keyword **procedure** followed by the name *triangle* that we are giving to the

procedure. There is no data type of a procedure as there is to a function. At the end of the definition is the keyword **end** followed by the name of the procedure. Between the header and the **end**, with the name *triangle* after it, is the body of the procedure which consists here of a comment explaining what the procedure does and a **for** loop. We will leave blank lines around a procedure definition as we did for a function definition to make our program more readable.

Here is a program that uses this procedure.

```
% The "Jagged" program
% Outputs a number of triangles

procedure triangle
    % Outputs a triangle of asterisks
    for i : 1 .. 5
        put repeat ("*", i)
    end for
end triangle

var howMany : int
put "How many triangles do you want? " ..
get howMany
for i : 1 .. howMany
    triangle
end for
```

Notice that the definition of the procedure *triangle* is included in the program. It can be placed, as a unit, anywhere in the program as long as it precedes its use by the program. This means the definition must come before the statement *triangle* that causes it to be used.

The call to a procedure is a statement. It is not part of an assignment or an output statement. The call appears by itself on a line.

Here is a sample Execution window.

```
How many triangles do you want? 3
*
**
```

```
***
****
*****
*
**
***
****
*****
*
**
***
****
*****
```

Here we have made a procedure *triangle* and used it in the main program. In the *triangle* program we use the loop index *i* and the call to *triangle* in the main program is nested inside a loop using the index *i*. If we tried to write a single program containing such a sequence namely:

```
for i: 1 .. howMany
    for i: 1 .. 5
        put repeat("*", i)
    end for
end for
```

we would be given a syntax error message and would have to change the first *i* to some other name such as *j* to get the program to work. When the inner **for** loop is embedded in the *triangle* procedure there is no problem. The index *i* of the procedure is a **local** variable in the procedure and is not known outside at all. Thus there is no problem having the loop contain the statement *triangle* controlled by an index called *i*.

This is an important point about procedures and functions: name coincidence between a variable local to the procedure or function and other variables in the main program, or in other subprograms, causes no conflict. The main program, however, does not have any access to local variables in a subprogram.

16.3 A Procedure with One Parameter

Here is a procedure that is similar to the procedure *triangle* but which requires one piece of information to be fed into it from the main program, namely how big the triangle to be output is. We will call this procedure *triangles* to indicate that various sizes are possible.

Here is its definition.

```
procedure triangles (size : int)
    % Outputs a triangle of size asterisks
    for i : 1 .. size
        put repeat ("*", i)
    end for
end triangles
```

In the header of the procedure definition the parameter *size* and its data type **int** are given in parentheses after the name of the procedure.

Here is a program that uses the *triangles* procedure.

```
% The "RandomTriangles" program
% Outputs 5 triangles of random size between 3 and 6

(copy triangles procedure here)

var howBig : int
for i : 1 .. 5
    randint (howBig, 3, 6)
    triangles (howBig)
end for
```

In this example the actual parameter for triangles is *howBig*. This is in correspondence with the formal parameter *size*. Both are of type **int**.

This program uses the predefined procedure randint. Here is the signature for *randint*.

randint (**var** *i* : **int**, *low*, *high* : **int**)

The *randint* procedure sets *i* to the next value of a sequence of pseudo random integers that approximate a uniform distribution over the range *low* ² *i* and *i* ² *high*. It is required that *low* ² *high*.

When randint is called in the program *RandomTriangles* it is called by

randint (*howBig*, 3, 6)

The actual parameter 3 corresponds to the formal parameter *low*; the actual parameter 6 corresponds to the formal parameter *high*. This means that, when randint executes, the actual parameter *howBig,* which corresponds to the formal parameter *i*, is set to the next value in a sequence of pseudo-random integers from the range 3 to 6 inclusive. The variable *howBig* has been altered by the action of the procedure; we say it is a **variable parameter**. In the definition of a procedure any parameter whose value is altered by that procedure must be indicated by having the keyword **var** placed in front of its type definition. Notice the **var** in front of the *i* in the definition for randint.

We use the term pseudo-random because the results from *randint* are not truly random. Because the computer is not capable of basing its results on any truly random events, it simulates random numbers by generating a sequence of numbers according to a mathematical formula. The results of this formula appear to be random and are suitable for use in programs.

Note that because the first parameter in randint is a variable parameter (we also say that the parameter is **passed by reference**), you must pass a variable as the first parameter. The statement

randint (4, 3, 6)

is illegal because it attempts to pass an integer as the first variable.

Variables that are not passed by reference are **passed by value**. This means that the value of the parameter is assigned to

the parameter in the subprogram. This also means that you cannot change the value of the parameter in the subprogram. The subprogram

```
procedure badSwap (x, y : int)
    var temp : int := x
    x := y
    y := temp
end badSwap
```

generates compilation errors on the lines assigning values to *x* and *y*.

16.4 Variable Parameters in Procedures

Here is another example of a procedure with a variable parameter. This procedure will use the balance in a bank account and the interest rate and return the new balance and the interest paid. Compare this with the *ComputeInterest* program at the beginning of the chapter.

```
% The "ComputeSavings" program
% Computes bank interest to nearest cent
var balance, interestRate, interest : real

function roundCent (amount : real) : real
    % Round amount to nearest cent
    result round (amount * 100) / 100
end roundCent

procedure banker (var balance, interest : real, interestRate : real)
    interest := roundCent (balance * interestRate / 100)
    balance := balance + interest
end banker

put "Enter balance " ..
get balance
put "Enter current interest rate " ..
```

```
get interestRate
banker (balance, interest, interestRate)
put "New balance = ", balance
put "Interest = ", interest
```

The parameters of the procedure *banker* that are variable parameters are *balance* and *interest* because *banker* changes these values. Their names are preceded by the keyword **var**. The procedure *banker* does not change the value of *interestRate* so it is not declared as a variable parameter.

Here the names of the actual parameters and the formal parameters are identical. This is perfectly all right; the names that are in correspondence may be different but do not have to be.

As it happens if we are using identical names we can omit all references to the parameters in the definition of the procedure and the call to it. This is possible because any subprogram has access to the variables declared in the main program. We say these variables are **global**. They are known both in the main program and in any subprogram defined in the program after they have been declared. We could then write our program with the procedure header.

```
procedure banker
```

and the call to it simply as

```
banker
```

Note that in the procedure *banker*, the variables *balance*, *interest*, and *interestRate* refer to the local variables declared in the procedure rather than the global variables declared at the beginning of the program.

16.4.1 Procedures to Bullet-proof Input

We have already shown a bullet-proof program segment that gets an integer value from the user. You can make this segment even more useful by making it a procedure so that it can be called from any place in a program.

Here is the *getInt* procedure.

```
procedure getInt (var number : int)
    var input : string
    loop
        get input
        exit when strintok (input)
        put "Not a number. Please enter an integer: " ..
    end loop
    number := strint (input)
end getInt
```

Here is an example program that uses the *getInt* procedure to compare two ages.

```
% The "GetTwoAges" program.

(copy getInt function here)

var age1, age2 : int
put "Enter the age of the first person: " ..
getInt (age1)
put "Enter the age of the second person: " ..
getInt (age2 )
if age1 > age2 then
    put "The first person is older."
else
    put "The second person is older."
end if
```

16.5 Predefined Procedures and Functions

As we have already said certain functions and procedures in Turing are predefined and can be used without actually including them in your own program. For example, the predefined functions length, index or repeat for strings or sqrt, abs and min for numbers. There are, as you know, Turing procedures for producing random numbers. The predefined procedure randint whose header line is

randint (**var** *i* : **int**, *low*, *high* : **int**)

sets the variable *i* to a pseudo random integer from a uniform distribution between *low* and *high* inclusive.

Here is another example using randint. In it we simulate the throw of a die (one of two dice) using

randint (*throw*, 1, 6)

This will produce an integer value for *throw* between 1 and 6 inclusive. Here is a program to produce a series of throws for two dice.

```
% The "RollDice" program
% Simulate a sequence of dice throws
var die1, die2 : int
loop
    put "Do you want to roll (y or n)? " ..
    var reply : string (1)
    get reply
    if reply = "y" then
        randint (die1, 1, 6)
        randint (die2, 1, 6)
        put "You rolled ", die1 + die2
    elsif reply = "n" then
        exit
    else
        put "Reply y or n"
    end if
end loop
```

In this program randint is used. Other predefined procedures for random numbers are

rand (**var** *r* : **real**)

which sets *r* to a pseudo random real number from a uniform distribution in the range between 0 and 1 (not inclusive).

If you have a subprogram of your own that you want to use in a program, store it in a file under its name then in the location

where its declaration should be in the main program place the line

include "filename"

This will cause the subprogram to be included at this point of the main program. You can also copy subprograms from a separate file into a program using the cut and paste commands of the Turing editor.

16.6 Recursive Subprograms

In Turing it is permitted to have a subprogram call itself. We say then that the subprogram is **recursive**. Here is a function that produces the value of factorial *n*. The definition of factorial *n* is the product of all the integers from *1* to *n* inclusive. A recursive definition of factorial *n* that is equivalent to the other definition is

factorial (n) = n * factorial (n – 1)

and that factorial (1) = 1. This means that you can work out the factorial of the next higher integer if you know the factorial of the previous integer.

Here is the recursive function.

```
function factorial (n: int): int
    pre n > 0
    if n > 1 then
        result n * factorial (n – 1)
    else
        result 1
    end if
end factorial
```

Seems like magic but it works. The function factorial keeps calling itself for the factorial of a smaller value as long as *n* is greater than 1.

In the line after the function header line there is a **pre** condition. This is like the **assert** statement used in programs but is used in subprograms to state conditions that must be true upon

entry into the subprogram. If the condition is not true, execution will be stopped. You can also have a **post** condition which states a condition that must be true as you exit from a subprogram.

16.7 Functions versus Procedures

When creating a subprogram to perform a task, it is important to decide whether the subprogram should be a procedure or a function. There are a few guidelines that are used to determine which type of subprogram should be used, depending on the characteristics of the subprogram.

If the subprogram calculates a single value, then the subprogram should be made a function and the calculated value should be returned as the value of the function. If the subprogram can change multiple values, then the subprogram should be a procedure and the values to be changed passed as **var** parameters to the procedure. If the subprogram does not change any values, then it should also be made a procedure. If a subprogram changes global variables, then the subprogram should also be a procedure.

Here are some sample subprograms, along with the reasoning for making them a funtion or a procedure.

A subprogram that calculates the interest given an initial balance and an interest rate – function. There is only one value calculated and no changes are made to global variables.

A subprogram that draws a flag – procedure. There is no returned value and no parameters are changed.

A subprogram that determines if an answer to a multiple choice question was legal – function. There is a single returned value (**true** or **false**). The return value depends on whether the user's input was from a single letter from a to e.

A subprogram that returns the coordinates of a mouse click – procedure. The mouse click coordinates consist of two values, the *x*- and *y*-coordinates. The subprogram has two **var** parameters to return the two values.

A subprogram that calculates the mean of an array of numbers – function. The subprogram has an array as a parameter and returns the mean as a single value.

16.8 Exercises

1. Write and test a function subprogram called *circleArea* whose value is to be the area of a circle whose radius is given as the parameter of the function. Compare the results with those of the *ManyCircleAreas* program in Chapter 6.
2. Write and test a function subprogram called *interest* whose value is the yearly interest to the nearest cent on a bank balance. The amount of the balance and the yearly interest rate as a percentage are to be parameters of the function.
3. Write and test a procedure subprogram called *swap* for swapping the values between two **string** variables given as parameters of the procedure.
4. Write and test a procedure subprogram called *rotate* which will rotate the integer values of three parameters. If the parameters A, B, and C originally had values 1 2 3 before the operation of the procedure, then afterwards the values would be 3 1 2.
5. Write and test a procedure subprogram called *inflate* that, given as parameters an amount of money and a constant annual rate of inflation, will produce a list of the inflated values of the original amount at the end of each year for 10 years.
6. Use the predefined procedure randint to produce a pattern of a random number of asterisks between 1 and 5 along the page. Have the total number of lines in the pattern as a parameter. For example, the pattern might look like this

```
**
*
***
**
*****
```

```
*
* * * *
```

and so on

7. Experiment with the predefined functions floor, ceil, intreal, intstr, and strint to try to understand their usefulness.
8. Write and test a subprogram called *midLetter* that returns the middle letter of a word given as a parameter. If the word has an even number of letters, the first letter in the second half of the word is considered the middle letter.
9. Write and test a procedure subprogram called *card* that randomly generates suit and value of a card (for example the Jack of Hearts) and returns them through parameters.
10. Write and test a procedure subprogram that plays the music to ÒHappy BirthdayÓ.
11. Write and test a recursive function that returns the values of x^n, where x and n are parameters and n is positive.
12. Write a procedure called *blank* that works like cls. Do not use ÒclsÓ as a statement in the procedure.
13. Write a function called *rounding* which does what the predefined function round does. Make sure that it takes a real parameter and returns an integer.
14. Write a function called *reverse* which takes one value string parameter and returns a string that is the reverse of the parameter string. Incorporate this function in a main program which tests to see if a word is a palindrome.
15. Write a function called *toUpper* which has one value parameter of string type. It should return a string which has all the letters in the parameter string converted to uppercase. For example

    put *toUpper* ("Fred")

    should output FRED.
16. Write a function called *absolute* which takes one real parameter and returns the absolute value of that real parameter. Do not use the predefined function *abs*.

17. Write a subprogram called *butterfly* which takes 6 value parameters, all of integer type. The first four parameters represent the bottom-left corner and top-right corner of the box in which the butterfly is to be located. The last two parameters are for the color of the wings, and the color of the body respectively. No matter how small or large the box is, the butterfly (large or small) should retain its shape.
18. Write a procedure called *clean* which takes one variable parameter of string type. This procedure should take the string and change it so that it contains only letters. For example,

```
var word := "9u%Tre?."
clean (word)
put word    %---> should output "uTre"
```

19. Write a subprogram called *filename* which takes a value string parameter which is the name of a file. The function should return true if the file exists and false if it does not.

16.9 Technical Terms

subprogram
function
function definition
body of function
procedure
procedure definition
body of procedure
header
parameter
formal parameter
actual parameter
result **statement**
variable parameter
local variable
global variable
predefined function
predefined procedure
randint procedure
rand procedure

Chapter 17

Subprograms with Array Parameters

17.1 Functions with Array Parameters

In this chapter we will look at subprograms that have parameters that are of the array data type. Here is a function which will have a value that is the largest element of an array of 5 integers.

```
function maxArray (list : array 1 .. 5 of int) : int
    var biggest := list (1)
    for i : 2 .. 5
        if list (i) > biggest then
            biggest := list (i)
        end if
    end for
    result biggest
end maxArray
```

Here is a program that uses the maxArray function.

```
% The "FindBestMark" program
% Finds the highest mark in a class of five

 (copy maxArray function definition here)

put "Enter five marks"
var mark : array 1 .. 5 of int
for i : 1 .. 5
    get mark (i)
end for
put "The best mark is ", maxArray (mark)
```

In choosing names for functions or procedures (or variables) we must avoid using those of predefined functions or procedures which, like keywords, are reserved in the Turing language. A list of reserved words can be found in the appendix. Notice that *max* and *min* are in this list.

17.2 Array Parameters in Procedures

This procedure will sort a list of ten names of maximum length 30 characters into alphabetic order. The method of sorting is called a **bubble sort**. The sorting is accomplished by comparing the first two items in the list and interchanging their positions if they are not in ascending order of size. This process is repeated using the second and third items, then the third and fourth. When you are finished the largest item will be in the last position in the list. The whole process is then repeated on a list that excludes the last item. In this way the complete list gradually gets sorted. In carrying out the algorithm, if at any time there is no need to exchange a pair of elements, the list must be sorted.

```
procedure bubble (var list : array 1 .. 10 of string (30))
    % Sort list of 10 names into alphabetic order
    for decreasing last : 10 .. 2
        var sorted : boolean := true
        % If there is any swapping, list is not sorted
        % Swaps elements so largest in last
        for i : 1 .. last – 1
            if list (i) > list (i + 1) then
                sorted := false
                % swap elements in list
                const temp := list (i)
                list (i) := list (i + 1)
                list (i + 1) := temp
            end if
        end for
        exit when sorted
    end for
end bubble
```

The parameter *list* will be altered by the procedure *bubble* so it must be declared as a variable parameter. The variable *sorted* is a variable that is local to the procedure *bubble* as of course are the index variables *last* and *i* and the constant *temp*. Just as the value of an index variable is not available outside the **for** loop

that it controls, the value of *sorted* is not available outside the procedure in which it is defined.

Here is a program that uses the procedure *bubble*. There are as well, two other procedures, one for reading a list of 10 names and one for outputting the list.

```
% The "SortNames" program
% Reads, sorts, and outputs a list of 10 names

procedure readList (var people : array 1 .. 10 of string (30))
    % Read 10 names
    put "Enter 10 names one to a line"
    for i : 1 .. 10
        get people (i)
    end for
end readList

(copy bubble procedure here)

procedure writeList (name : array 1 .. 10 of string (30))
    % Output 10 names
    put "Here is sorted list"
    for i : 1 .. 10
        put name (i)
    end for
end writeList

var friend : array 1 .. 10 of string (30)
readList (friend)
bubble (friend)
writeList (friend)
```

Notice that the formal parameter *name* in the *writeList* procedure is not a variable parameter because it is not altered by the procedure. The formal parameters for the list of names are: *list* in *bubble, people* in *readList*, and *name* in *writeList*. These are each in turn in correspondence with the actual parameter *friend*.

17.3 Dynamic Formal Parameters

In the procedures *bubble, readList,* and *writeList* the parameter involved is an array of ten strings of length 30 characters. This means that their use is restricted to this particular length of array and length of string. We could have left these two unspecified in the procedures and then the procedure would work for any length of array or string.

For example, the procedure *readList* might be given as

```
procedure readList (var people: array 1 .. * of string (*))
    % Read names
    put "How many names are there? " ..
    var number
    get number
    for i : 1 .. number
        get people (i)
    end for
end readList
```

Here in the header the upper limit of the array is given by an asterisk as is the length of the string. We say that the array parameter and the string parameter are **dynamic**; they can change. The declaration of *friend* in the main program cannot be given in this ÒdynamicÓ way. It must be declared with the actual string length to be used and an array size large enough to hold the longest array you want to read in.

The *writeList* procedure can similarly be written in this manner.

```
procedure writeList (name: array 1 .. * of string (*))
    % Output as many names as in array
    put "Here is sorted list"
    for i : 1 .. upper (name)
        put name (i)
    end for
```

```
end writeList
```

Here the actual upper limit of the array that is in correspondence with the formal parameter *name* is given when you ask for the value of

upper (*name*)

The lower limit would also be available using

lower (*name*)

Note that you can only use the * to represent the array size only when the array is being passed as a parameter. This is because the array size has already been set.

17.3.1 Another Example of a Procedure

Here is a procedure that finds the minimum and the maximum of a list of real values.

```
procedure minmax (list : array 1 .. * of ageRange,
        var minimum, maximum : ageRange, number : int)
    minimum := list (1)
    maximum := list (1)
    for i : 2 .. number
        if list (i) > maximum then
            maximum := list (i)
        elsif list (i) < minimum then
            minimum := list (i)
        end if
    end for
end minmax
```

To review: a procedure declaration begins with the keyword **procedure** followed by the name you choose for it. In parentheses are the names of the formal parameters with their types. With functions information can only flow into the subprogram through the parameters. It is through the value of the function itself that information flows out. No other **side effects** are permitted. With procedures information can flow both ways

through the parameters and this means that for procedures there is another kind of formal parameter. If the procedure changes the value of any parameter, which means information is flowing out, its name must be preceded by the keyword **var**. It is said to be a variable parameter. The *minmax* procedure will change the values of the parameters *minimum* and *maximum* so they must be declared as variable. Since *minmax* will only be looking at *list* and not changing any value, *list* is not preceded by the keyword **var**. Thus *list* cannot be changed. There is no data type of the procedure itself as there is with a function. The body of procedure ends with the keyword **end** followed by the name of the procedure.

The size of the array that will be examined to find the *maximum* and *minimum* is passed as a parameter *number*. The array *list* itself is declared as an array whose elements range from 1 .. *. The use of the asterisk means that the size of the array can vary; it is a dynamic array size. When the procedure executes, the value of *number* will be the actual length of the array that is in correspondence with the formal array parameter *list*.

Now we must use this procedure in a main program. Here is the program that computes the difference between the largest and smallest elements of an array.

```
% The "AgeSpan" program
% Computes the difference between the largest
% and smallest element in an array of ages
type ageRange : 1 .. 120
put "Enter number of elements in array: " ..
var number : int
get number
var age : array 1 .. number of ageRange
% Read in array with number elements
put "Enter ", number, " ages"
for i : 1 .. number
    get age (i)
end for

(copy procedure minmax here changing real parameters to the
```

```
 named type ageRange)

var largest, smallest : ageRange
minmax (age, smallest, largest, number)
put "Age span in group is ", largest – smallest
```

To use the procedure *minmax* in this *AgeSpan* program you simply write its name followed in parentheses by the actual parameters you are using. Here the array of values is called *age*; this actual parameter corresponds to the formal parameter *list*. Then the actual parameters *smallest* and *largest* correspond with the formal parameters *minimum* and *maximum* respectively. When the procedure changes the value of *minimum* it is actually changing the value of *smallest*. When it changes *maximum* it is really changing *largest*. Actual parameters that are going to be changed by the procedure, that is variable parameters, must always be variables. Other parameters may be values such as 100 rather than variables.

The variable *age* is declared as an **array of** *ageRange*. This declares that it is to be an integer between 1 and 120 inclusive. This is a subrange of all integers. If you try to read an age into the array *age* that is outside this range an execution error will be reported. It is a way of protecting yourself against absurd values of the data.

Since *largest* and *smallest* are chosen from the array *age* they too are given the same subrange data type. The type of the formal parameters *maximum* and *minimum* of *minmax* must be changed to *ageRange* to match the type of *largest* and *smallest*. The declaration of the procedure can be anywhere in the program as long as it precedes its use and is not nested inside a construct such as a loop.

17.3.2 An Example Using Both a Function and Procedures

This example uses one function and two procedures; in it the marks in a class of students are displayed in a bar chart (or histogram) to show the statistical distribution.

```
% The "TextBarChart" program
% Plot a bar chart for class marks
var count : int

procedure readFreq (var freqList : array 1 .. * of int)
    for i : 1 .. upper (freqList)
        get freqList (i)
    end for
end readFreq

function maximum (list : array 1 .. * of int) : int
    var biggest : int := list (1)
    for i : 2 .. upper (list)
        if list (i) > biggest then
            biggest := list (i)
        end if
    end for
    result biggest
end maximum

procedure graph (grafList : array 1 .. * of int, width : int)
    const scale := width / maximum (grafList)
    for i : 1 .. upper (grafList)
        put repeat ("*", round (scale * grafList (i)))
    end for
    put "Maximum is ", maximum (grafList)
    put "Scale is ", scale : 10 : 2
end graph

put "Enter number of entries in list: " ..
get count
var entries : array 1 .. count of int
readFreq (entries)
graph (entries, 60)
```

The procedure *readFreq* reads in the array of marks. The procedure *graph* plots the bar chart of asterisks. It uses the function *maximum* to scale the graph so the longest bar is 60 asterisks long. The declaration of the function *maximum* is not placed inside the declaration of *graph* even though it is used only

by it because, in Turing, procedures and functions cannot be **nested**. Nesting means that one is inside the other.

We could use a function for *maximum* because it produces a single value. The *minmax* procedure produced two values. Notice that the actual parameter in *graph* that corresponds to the formal parameter *width* is 60. A value can be used since it is not a variable parameter.

In the function *maximum* the actual length of the array corresponding to the formal parameter *list* is not given as a parameter the way that we gave the *number* of elements as a parameter in the *minmax* procedure earlier in the chapter. We can use the predefined function *upper* to give the actual size of the array. The value of *upper(list)* will be the length of the array of the actual parameter that corresponds to the formal parameter *list*.

17.4 Local and Global Variables and Constants

In the subprograms that we have shown so far all the variables and constants used inside the subprogram have been declared in the subprogram, either in the list of formal parameters or in the body of the subprogram. These variables or constant names are not accessible outside the subprograms. They are **local variables** or **constants** just as the index of a counted **for** is local to the body of the **for** loop. You cannot use it outside.

In the subprogram there may be declared a variable that has the same name as one used in the main program or in other subprograms used by the main program. For example, in the *barChart* program the variable *i* is used in *readFreq*, *maximum,* and *graph*. This does not matter because in any one subprogram the *i* is a local variable.

The variables or constants declared in the main program before the definition of subprograms are all accessible to those

subprograms. We say they are **global variables** or **constants**. For example, in the main program *barChart* the variable *count* is declared before the subprograms *readFreq* and *graph* are defined. If we included the statement **put** *count* in either of the procedures *readFreq* or *graph*, the value of *count* would be output.

If we included this **put** statement in the function *maximum* it would work but we would be warned that the function *maximum* had the side effect of changing the output. Functions, remember, can only have a value and should not change any variables, input anything, or output anything.

We sometimes make use of the fact that variables and constants declared in the main program are global to all its subprograms to avoid long lists of parameters. This is true of programs which contain subprograms that are intended only for use by that particular main program. Subprograms intended for general use should not use global variables.

Here is an example where global variables are used.

```
% The "UseGlobals" program
% Read names and output list of names in order
put "How many names are there? " ..
var number : int
get number
var name : array 1 .. number of string (40)

procedure readNames
    put "Enter the names"
    for i : 1 .. number
        get name (i)
    end for
end readNames

procedure sortNames
    % Sort list of names alphabetically
    % The sorting algorithm is a Shell sort
    % See if you can follow it
    % It is a modification of the bubble sort
    var space : int := number
```

```
    loop
        space := floor (space / 2)
        exit when space <= 0
        for i : space + 1 .. number
            var j : int := i – space
            loop
                exit when j <= 0
                if name (j) > name (j + space) then
                    const temp := name (j)
                    name (j) := name (j + space)
                    name (j + space) := temp
                    j := j – space
                else
                    j := 0                  % Signal exit
                end if
            end loop
        end for
    end loop
end sortNames

procedure outputNames
    % Output sorted list
    put "Here are the names in order"
    for i : 1 .. number
        put name (i)
    end for
end outputNames

readNames
sortNames
outputNames
```

The global variables *number* and *name* are used in each one of the subprograms. There are no parameters in any one of them. There are however local variables in the subprograms. For example, *sortNames* declares its local variables *space* and *j* as well as a local constant *temp*. Because *j* is declared inside a **for** loop it is local to that **for** loop. It is not accessible even outside that. Similarly the constant *temp* is local to the **if..then..else** construct. When you leave that construct its value is inaccessible.

Each time you enter the **if** construct *temp* is redeclared as a constant with a new value. We could have used a **var** here but calling it a constant is more precise.

The procedure *sortNames* uses a method (or **algorithm**) of sorting invented by Donald Shell. It is called a **Shell sort**. Perhaps you can see how the list of names is sorted by this method. On the other hand you do not need to know how a procedure actually works to use it in your own program. The predefined function floor gives the integer just less than its real actual parameter *space*/2.

To be more generally useful the *sortNames* procedure should not use the global variables *name* and *number* but have instead parameters. Its header line would then be

```
procedure sortNames (var name:
    array 1 .. * of string(*), number: int)
```

Then the variables *name* and *number* would be parameters. Notice that we declare the upper bound of the range of the array *name* by an asterisk and the length of the **string** by an asterisk. The length of the string is also dynamic, it can change. The calling statement in the main program for *sortNames* would be

```
sortNames (name, number)
```

The actual parameters here have the same names as the formal parameters in the declaration of the procedure *sortNames*. There is nothing wrong with having it this way if it happens to suit you. We try to show examples with different names for the formal and actual parameters so you can see that they need not be the same. If they are all identical then global variables would have worked just as well.

17.5 Maintaining a Sorted List

The program in the previous chapter sorted an unsorted list all at once. However, once a sorted list is obtained, it is often

preferable to maintain the list's order as elements are added or deleted rather than to re-sort the list from scratch. A list is sorted in ascending order if each value in the list is no larger than the one that comes after it in the list.

When a list is sorted. maintaining the order when inserting or deleting values requires more work. Think of the list as being laid out from left to right. Before a value can be inserted, the appropriate location for it must be found in the list; then the value in that location, together with all the values to the right of that location, must be shifted to the right to make room for the new value. Before a value can be deleted, its location must be found in the list then all the values to the right of that location must be shifted to the left.

In Turing, only one element of an array can be manipulated at a time. To shift a part of a list left or right, the values must be shifted one at a time. To shift to the right, the last element must be moved to the right and the previous element must then be moved into the vacated position, and so on. In Figures 17.1 and 17.2, the numbers indicate the order in which the elements must be copied.

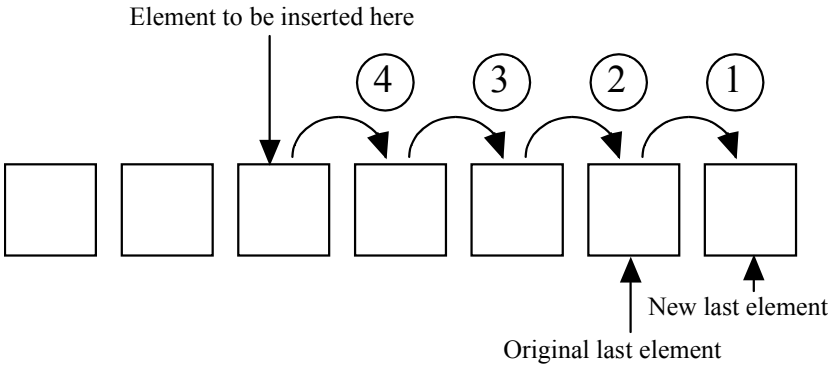


Figure 17.1 Inserting an Element in an Array

To move to the left, the first element must be moved to the left and the second element must then be moved into the vacated position, and so on.

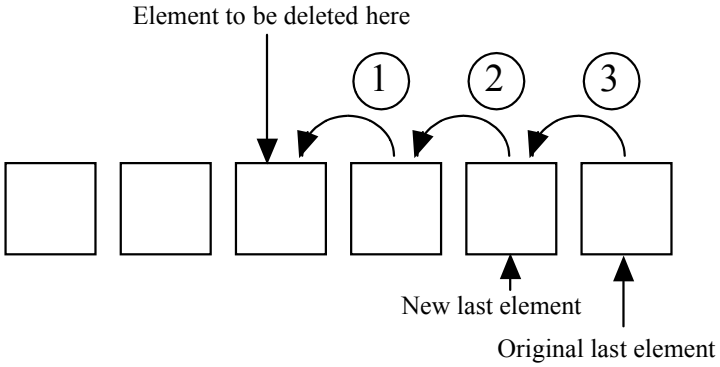


Figure 17.2 Deleting an Element in an Array

Here is a program that maintains a list of names in sorted order in an array. This program does not save the list of names between executions. It also does not check the number of names in the list compared to the maximum array size.

```
% The "SortedList" program
% Maintains a sorted list of names

% Add a name to the sorted list in the correct position
procedure AddName (name : string, var list : array 1 .. * of string (*),
        var listSize : int)
    % The array index where the name should be placed
    var insertPoint : int := 1

    % Determine the location in the array where the name should be
    % placed
    loop
        exit when insertPoint > listSize or list (insertPoint) > name
        insertPoint := insertPoint + 1
    end loop

    % Shift the remaining items over to make room at insertPoint
    for decreasing i : listSize .. insertPoint
        list (i + 1) := list (i)
    end for
```

```
    % Insert the name at insertPoint
    list (insertPoint) := name

    listSize := listSize + 1

    put name, " added. The list now has ", listSize, " name(s)"
end AddName

% Delete a name in the sorted list
procedure DeleteName (name : string, var list : array 1 .. * of
        string (*), var listSize : int)
    % The array index holding the name to be deleted
    var deletePoint : int := 1

    % Determine the location in the array of the name to be deleted
    loop
        exit when deletePoint >= listSize or list (deletePoint) >= name
        deletePoint := deletePoint + 1
    end loop

    % Make certain the name was found in the list
    if list (deletePoint) not= name then
        put name + " not found in the list"
        return
    end if

    % Shift the remaining items over to remove the element at
    % deletePoint
    for i : deletePoint + 1 .. listSize
        list (i – 1) := list (i)
    end for

    listSize := listSize – 1

    put name, " deleted. The list now has ", listSize, " name(s)"
end DeleteName

% Display the sorted list of names
procedure ListNames (list : array 1 .. * of string (*), listSize : int)
    for i : 1 .. listSize
        put i : 3, " ", list (i)
    end for
```

```
end ListNames

% Main program
var listOfNames : array 1 .. 100 of string (40)
var sizeOfList : int := 0
var choice : int
var name : string (40)
loop
    put "What is your command:"
    put " (1) Add a name"
    put " (2) Delete a name"
    put " (3) List the names"
    put " (4) Exit"
    put "Choice: " ..
    get choice
    if choice = 1 then
        put "Enter the name to be added: " ..
        get name : *
        AddName (name, listOfNames, sizeOfList)
    elsif choice = 2 then
        put "Enter the name to be deleted: " ..
        get name : *
        DeleteName (name, listOfNames, sizeOfList)
    elsif choice = 3 then
        ListNames (listOfNames, sizeOfList)
    elsif choice = 4 then
        exit
    else
        put "Illegal command"
    end if
end loop
```

17.6 Exercises

1. Write and test a function subprogram whose value is the average rounded to the nearest integer of a series of marks stored in an array of integers.

2. Write and test a function subprogram called *maxi* whose value is the maximum of an array of real values.
3. Write and test a subprogram called *range* whose value is the difference between the smallest and largest of an array of real values. This function may use the function *maxi* and a similar function *mini* as developed for Exercise 2.
4. Write and test a subprogram called *stats* which produces the average and the variance of an array of real values. The variance is computed by taking the sum of the squares of the differences between the average and the individual values, and dividing this by the number of values.
5. Try running the *TextBarChart* program of this chapter using sample data which you store in a file called *Student*.
6. Try to follow the shell sort algorithm included in the program *UseGlobals* given in this chapter by inserting output statements in the *sortNames* procedure to look at the sorting process as it proceeds.
7. Write and test a subprogram called *bubbleSort* that uses the bubble sort method to sort a series of integers in descending order.
8. Repeat Exercise 7 but write a subprogram called *shellSort* that uses the shell sort method rather than the bubble sort.
9. Ask the user for a list of names. Use a sentinel to stop the inputting of names. Output the names in alphabetical order getting rid of any duplicate names.
10. The clock procedure is used to give the current processor time in milliseconds. To time something use the *clock* procedure twice, once before and then after the process you are timing. The difference is the time elapsed for the process to occur. Here is an example.

```
var x, y : int
clock (x)
% The body of what you are timing.
clock (y)
put "The time elapsed is ", y – x, " ms."
```

Create files of 1000, 2000, and 3000 randomly created integers with values from 1 to 500. Use bubble sort, shell sort, and selection sort to sort each of the files and time and record the length of time required for each sort. Use a barchart program to make three graphs which you should display on separate windows separated by calls to getch. On the first chart, display the times the three sorts required to sort the 1000 integer file. The second chart should do the same for the 2000 integer file and the third chart should display the 3000 integer file. Which sort is the most time-efficient?

17.7 Technical Terms

bubble sort
dynamic array size
side effect of function
histogram
graph
Shell sort algorithm
dynamic string size

Chapter 18

Records and Files

18.1 Declaration of Records

Frequently we want to store information in files where each record in the file is a number of pieces of information about a single entity. For example, we might want a file of records of employees of a company. Each individual record might have stored: a name, an address, an employee number, a birth date, and so on. In Turing we use a **record** data type to store such data. Each record is a collection of **fields**.

Here is the declaration of a record called *student* used to hold a student's record.

```
var student:
    record
        name: string (20)
        address: string (40)
        phone: string (8)
    end record
```

In the declaration the **record** data type begins with the keyword **record** and ends with **end record**. In between is a list of the fields and their data types. Each of the three fields is of the type **string** but of different maximum length.

If in the program where such a record is declared you wanted to refer to a particular field you would use its full name which consists of the name of the record followed by a period then the name of the field. For example, the name for the phone number field is

student.phone

18.2 Inputting and Outputting Records

To input a record, each field of the record is read separately. Similarly on output each field is output separately.

Here is a program to read a student record and output it, just to show you how it is done.

```
% The "ReadAndOutputRecord" program
% Reads a record and outputs it
var student :
    record
        name : string (20)
        address : string (40)
        phone : string (8)
    end record
loop
    % Read in record
    put "Enter name"
    exit when eof
    get student.name : *
    put "Enter address"
    get student.address : *
    put "Enter phone"
    get student.phone : *
    % Output record
    put student.name : 20, student.address : 40,
        student.phone : 8
end loop
```

Each of the fields is entered and read as a line so that they can contain blanks. Since input is by lines a **get skip** is not necessary before the eof test. On output all three fields are on the same line.

18.3 Arrays of Records

Very frequently we have not just one record but an array of records. For example, we might want to have an array of student records for a whole class. Such an array might be a **file** of student records. Here is a program to read in student records into an array from a data file of at most 100 records called *Students*. This file is prepared so that on a line are the three fields of a single record. The name is left justified in the first 20 character positions and filled in on the right with blanks if necessary. Similarly the address is in the next 40 positions and the phone number in the last 8.

Here is the program.

```
% The "ReadStudentFile" program
% Reads records from student file into array
% then allows you to read individual records at random
type studentType :
    record
        name : string (20)
        address : string (40)
        phone : string (8)
    end record

const maxFile := 100
var studentFile : array 1 .. maxFile of studentType
% Read records from student file
var students : int
open : students, "Students", get
assert students > 0
var count := 0
loop
    get : students, skip
    exit when eof (students)
    count := count + 1
    get : students, studentFile (count).name : 20,
        studentFile (count).address : 40,
        studentFile (count).phone : 8, skip
```

```
end loop
put "There are ", count, " students in file"
var number : int
put "Enter negative record number to exit"
loop
    put "Give number of record you want: " ..
    get number
    exit when number < 0
    assert number <= count
    put studentFile (number).name : 20,
        studentFile (number).address : 40,
        studentFile (number).phone : 8
end loop
```

18.4 Binding to Records

In order to simplify the input and output instructions for records we can use the **bind** construct. For example, for the input of the student record we could replace the **get** by these two lines:

```
bind var which to studentFile (count)
get: students, which.name: 20, which.address: 40,
                    which.phone: 8
```

The bind construct makes programs more efficient and easier to read. In this example *which* becomes the name of *studentFile(count)*.

18.5 An Example using a File of Records

Suppose that you have a file of records of houses for sale by a real estate firm in disk file called *Sale*. For each house we have its color, its location, and its price in dollars. This file called *Sale* is to be read into the computer's memory so that information can be obtained from the file. For instance, you can ask for a listing of

all houses satisfying particular criteria, such as costing less then $100,000.

```
% The "HouseLocator" program
% Reads series of records for houses
% Allows questions about records in file
type houseType :
    record
        color : string (10)
        location : string (20)
        price : int
    end record
const maxHouses := 100
var houseFile : array 1 .. maxHouses of houseType

procedure readFile (var entry : array 1 .. * of houseType, var count :
int)
    % Read houseType records into an array
    var sale : int
    open : sale, "HouseInfo", get
    assert sale > 0
    count := 0
    loop
        get : sale, skip
        exit when eof (sale)
        count := count + 1
        bind var house to entry (count)
        get : sale, house.color : 10, house.location : 20,
            house.price
    end loop
end readFile

procedure outputRecord (h : houseType)
    put h.color : 10, h.location : 20, h.price : 10
end outputRecord

function wanted (house : houseType, desiredColor, desiredLocation :
        string (*),
        desiredUpperPrice : int) : boolean
    result (house.color = desiredColor or
        desiredColor = "any       ")
        and (house.location = desiredLocation
```

```
        or desiredLocation = "any                "
        and (house.price <= desiredUpperPrice)
end wanted
procedure readSpecs (var Color : string (*), var location : string (*),
        var price : int)
    put "Answer 'any' for color or location when it doesn't matter."
    put "What color of house do you want?"
    get Color           % must pad with blanks
    Color := Color + repeat (" ", upper (Color) – length (Color))
    put "What location of house do you want?"
    get location        % must pad with blanks
    location := location + repeat (" ", upper (location) – length (location))
    put "What is your highest price in dollars?"
    get price
end readSpecs

var howMany : int
readFile (houseFile, howMany)
var wantedColor : string (10)
var wantedLocation : string (20)
var upperPrice : int
readSpecs (wantedColor, wantedLocation, upperPrice)
for i : 1 .. howMany
    if wanted (houseFile (i), wantedColor,
            wantedLocation, upperPrice) then
        outputRecord (houseFile (i))
    end if
end for
```

A type of data is defined called *houseType* which is a **record type**. It has three fields: *color, location*, and *price*. The variable *houseFile* is declared to be an array of 100 such records. The procedure *readFile* reads records of type *houseType* into an array until it gets an end-of-file. It returns the *count* of the number of records read. In the *readFile* procedure the formal parameter *entry* is the array name for the records. The record *entry (count)* is the one being read at any time. The three fields of *entry (count)* record are referred to as *entry (count).color, entry (count).location*, and *entry (count).price*. Similarly the

procedure *outputRecord* outputs its record called *house* one field at a time, namely *house.color, house.location,* and *house.price*.

The function *wanted* is a **boolean** function meaning that its value is either **true** or **false**. In the procedure *readFile* we want to refer to fields of a record whose name is *entry (count).* It makes a more efficient program if we **bind** this name to another name *house* then, within the **scope** of the *readFile* function, anytime we say *house* we mean *entry (count)*. This is particularly true if what we are binding is an array element. What we bind to must be a variable.

18.6 Moving Records in Memory

One of the principal advantages of the record data structure is that an entire record can be moved (copied) from one memory location to another with a single assignment statement. (A record cannot however be input or output as a single item.) This is very useful when sorting records.

Suppose we wanted to sort the real estate records of the previous example in ascending values of their *price* fields. Here is a program that reads such records from a disk file, sorts them, and stores the sorted list in another disk file.

```
% The "SortHouseInfo" program
% Reads file of records from disk file
% Sorts records in order of ascending price
% Stores sorted records on disk
type houseType :
    record
        color : string (10)
        location : string (20)
        price : int
    end record
const maxHouses := 100
var houseFile : array 1 .. maxHouses of houseType
var count : int
```

```
procedure readDiskFile (fileStream : int,
        var houseFile : array 1 .. * of houseType,
        var count : int)
    % Read records from file
    count := 0
    loop
        get : fileStream, skip
        exit when eof (fileStream)
        count := count + 1
        bind var house to houseFile (count)
        get : fileStream, house.color : 10,
            house.location : 20, house.price
    end loop
end readDiskFile

procedure outputFileToDisk (fileStream : int,
        houseFile : array 1 .. * of houseType, count : int)
    % Output records from memory to disk
    for i : 1 .. count
        const house := houseFile (i)
        put : fileStream, house.color : 10,
            house.location : 20, house.price
    end for
end outputFileToDisk

procedure sortFile (var houseFile :
        array 1 .. * of houseType, count : int)
    % Sort list into ascending order of price
    % Use Shell sort algorithm
    var space : int := count
    loop
        space := floor (space / 2)
        exit when space <= 0
        for i : space + 1 .. count
            var j : int := i – space
            loop
                exit when j <= 0
                if houseFile (j).price >
                        houseFile (j + space).price then
                    const temp : houseType := houseFile (j)
                    houseFile (j) := houseFile (j + space)
```

```
                houseFile (j + space) := temp
                j := j – space
            else
                j := 0
                % Signal exit
            end if
        end loop
    end for
end loop
end sortFile

var filename : string
put "What is the file name for the unsorted records? " ..
get filename
var inStream : int
open : inStream, filename, get
assert inStream > 0
readDiskFile (inStream, houseFile, count)
close : inStream
sortFile (houseFile, count)
put "What is the file name for the sorted records to be? " ..
get filename
var outStream : int
open : outStream, filename, put
assert outStream > 0
outputFileToDisk (outStream, houseFile, count)
```

The *sortFile* procedure uses the Shell sort algorithm but this time the items being swapped are records. They are moved around in memory just as easily as single variables. The record type *houseType* is defined in the main program and is thus global to the three subprograms. Other than this, no global variables or constants are used. The names of the actual parameters are the same as the formal parameters. Notice that the **bind** statement is used in *readDiskFile* and that the fields of each record must be input or output separately.

The files that are created this way are **text files** and may be edited in the Turing Environment if desired. For example, the fields of any of the records may be altered, new records added, or records deleted using the editor.

18.7 Text Files

So far we have output records to disk files one field at a time using a **put** statement. A file of records had to be output sequentially, one record after the next, until it was closed. It could then be opened and read sequentially using **get** statements, where the fields of the record were input one at a time. Files of this nature can be examined in the editor and changes made directly to records by the Turing editing system. They are called **text files**; the program and data files that we have prepared in the editing system or output from a program so far are all text files.

18.8 Binary Files

There is an entirely different way of storing records in files. It is in **binary** (or **internal**) form. Files stored in binary form cannot be displayed in the window and edited in the Turing editor. They are produced as files on the disk by using the **write** output statement (instead of **put**) and input using the **read** statement (instead of **get**).

Records that are stored as text files are of variable length, each field taking up the space it actually requires. For example, a 15 character name takes less space than a 25 character name. Records stored in binary form to be accessed randomly are all of the same length. This length is the length specified in the definition of the corresponding record data type.

Storing records in binary form may take more or less space on the disk than storing them in text form but it has distinct advantages. First, records may be output as a whole and input as a whole.

Second, because the size is fixed, you can replace a record in the file with a modified record without it being of a different length than the original. This means that a file can be opened for both

reading and writing, whereas the text files that we have used were opened either for input or output but not for both at the same time.

The binary file is opened, as are text files, using an **open** statement. The **open** statement has the form

open : fileNumber, fileName, capability {, capability}

where *fileNumber* is a variable that has been declared as of **int** type and *fileName* is either a string constant which gives the name of the file on the disk or is a string variable whose value is the file's disk name. If the file is able to be opened properly a positive integer value is assigned to *fileNumber*. For binary files, the various values for *capability* are **read**, **write**, **seek**, or **mod**. The first two indicate whether the file is opened for binary reading, writing, or both. The **seek** indicates that the position in the file for the next read or write can be set by a **seek** statement or accessed by a **tell** statement. The **mod** is used when you are opening to write and not read but do not want to erase the current file.

We will describe a somewhat simplified use of binary input and output. The **read** statement has the form

read : fileNumber, readItem {, readItem}

where the *readItem* is a variable to be read in binary form using the size of the item that has been declared.

The **write** statement has the form

write : fileNumber, writeItem {, writeItem}

where a *writeItem* has a value to be written in binary form. The size of items to be written must be kept uniform from record to record.

There is also a **close** statement which has the form

close : fileNumber

When a file is opened, the operating system allocates space for it. Most operating systems have a limit to the number of files that may be opened at any one time, so it is important to close a

file when it is no longer needed. Note that Turing does automatically close all open files when the program is finished executing, however, it is still good practice to have your program close all open files.

18.9 Random Access to Records on Disk

To position for input/output in the disk file we use a statement of the form

seek : fileNumber, filePosition

where *filePosition* has an integer value giving the number of bytes from the beginning of the file to the point where you want to begin reading or writing. The start of the file is position zero. To position at the end of the file you use

seek : fileNumber, *

The position of this offset can be determined by using the **tell** statement which has the form

tell : fileNumber, filePosition

where *filePosition* is a variable of type **int** which is set by the **tell** to the offset in bytes.

18.9.1 An Example of Random Access to a Binary File on Disk

Here is a program that stores a number of student records in binary form in a disk file called *StudentData*. The information for the records is read in from the keyboard. As well, it keeps track of where each student's record is located in the file in an array in the main memory. Then the records on the disk are accessed randomly.

```
% The "SchoolDirectory" program
% Prepares a binary disk file of student records
```

```
% input from keyboard and stored in disk file " StudentData "
% As records are stored their position in the file is
% recorded in an array called "directory"
% Records are then accessed randomly
type studentType :
    record
        name : string (30)
        address : string (40)
        year : int
    end record
var student : studentType
% Open file called "StudentData" for writing
var fileNumber : int
open : fileNumber, "StudentData", read, write, seek
assert fileNumber > 0
const maxStudents := 500
type whereAbouts :
    record
        name : string (30)
        location : int
    end record
type directoryType :
    array 1 .. maxStudents of whereAbouts
var directory : directoryType
var filePosition : int
var count : int
put "How many students are there? " ..
get count
put "Enter ", count, " student records"
put " "

procedure find (var name : string (30), var position : int)
    % Uses global variables directory and count
    % See lookup program in chapter on arrays.
    if length (name) < 30 then
        % Pad with blanks
        name := name + repeat (" ", 30 – length (name))
    end if
    position := – 1
    % Search list for name by linear search
    for i : 1 .. count
        if name = directory (i).name then
```

```
            position := directory (i).location
            exit
        end if
    end for
end find
% Place labels to show user where to enter data
put "name" : 30, "address" : 40, "year"
for i : 1 .. count
    % Determine where next record will start in file
    tell : fileNumber, filePosition
    % Store filePosition as location in directory
    get skip, student.name : 30, student.address : 40,
        student.year
    directory (i).name := student.name
    directory (i).location := filePosition
    write : fileNumber, student
end for
% Access the records randomly
var wantedStudent : string (30)
loop
    var place : int
    put "Enter name of student"
    get skip, wantedStudent : *
    find (wantedStudent, place)
    if place not= – 1 then
        seek : fileNumber, place
        read : fileNumber, student
        put "Here is the information you want"
        put "name" : 30, "address" : 40, "year"
        put student.name : 30, student.address : 40,
            student.year
    else
        put "Student not in file"
    end if
end loop
```

18.10 Modification of Records on Disk

The major advantage of binary files over text files is that a record takes the same amount of space on the disk, regardless of the actual contents of the record. This means that unlike text files, you can change the contents of a single record in a binary file without having to change the contents of the entire file. For large files containing millions of records, this is the only practical technique for making data bases.

Because each record is stored on disk with a fixed length, it is possible to jump to a specified record without having to read the records ahead of it. For example, if a given record is 400 bytes in size, to skip to the beginning of the 8th record, you move to the 2800th byte in the file using

```
seek : f, 2800
```

The byte position in the file is 2800 rather than 3200 because the position of the first record is always 0. Thus the file position for the beginning of the *n*th record in the file is (*n* – 1) * record size.

To change a record in a file, you get the position in the file of the beginning of the record using the **tell** statement. You then read the contents of the record into memory. The record in memory is then changed as needed. To rewrite the data onto the disk, the program does a **seek** to the beginning of the record on disk and then writes the record. There is no danger of writing over top of the next record because, in binary files, the size of the record on disk does not depend on its contents. A record holding a name and phone number that takes 400 bytes of space on disk takes that space regardless of whether the name is ÒTom WestÓ or ÒArthur Frederick Schumacher IIIÓ.

Note that a binary file is usually larger than a text file holding the same data. That is because each binary record stored on disk reserves space for the maximum size of the data being written. For example, an integer in a binary file is always stored in 4 bytes. In a text file, it requires 1 byte per digit. A string always

uses space corresponding to the string's largest possible size plus one extra byte. For example, in

```
var a : string (20)
var b : string
```

the variable *a* has a maximum length of 20 and is stored in 21 bytes. The variable *b*, because it has no declared size, has a maximum length of 255 characters and is stored in 256 bytes.

To determine the size of a record, you use the sizeof function. This function returns the size of a record in memory.

```
var r : record
    name : string
    age : int
end r
put sizeof (r)
```

The next program creates an address book that contains names and addresses and allows changes to be saved on disk between runs. It allows you to add records and to change the address associated with a name. We will handle deleting records in the next section although we have Òleft roomÓ for it in the program.

The program keeps an array of the names corresponding to the records in the file. The first element in the *names* array is the name of the first record in the file, and so on. All lookup operations are done using the name array in memory. Once the name is found, then the corresponding record in the file is read and the full record is displayed.

We say the name field of the record is the key to the lookup operation. An array of keys is kept in memory while the whole record reamins on disk. In our example there are only two fields in the record so the fraction of the storage space required in memory, compared to what is required by the whole record, is large. In a real-life data base, there might be dozens of fields associated with each name.

Here is the header for the program. It defines the type of record used for each entry in the file and determines the size of the record using the **sizeof** function. It also defines the name of

the file where the records are stored as a constant. By using the constant rather than a file name in quotes, it reduces the chance that a misspelling occurs in the **open** statement.

```
% The "AddressBook" program
% Creates and maintains an addressbook of names and addresses.

% The type of each record in the file
type entryType :
    record
        name : string
        address : string
    end record
% Calculate the record size
const recordSize : int := sizeof (entryType)
const maxEntries : int := 100
const bookFileName : string := "address.dat"

var numNames : int % The number of names in the file

% The array stores the names in the same order as
% those in the records in the disk file.
var names : array 1 .. maxEntries of string
```

The *readNames* procedure reads the data file and placesl the names into an array. The rest of the data of the record is not stored. In this way, a database on disk can be gigabytes in size, while the array of keys in memory is much smaller.

```
% Read records in the file, initializing the
% array of names and determining the number of records.
procedure readNames
    var f : int
    var entry : entryType
    numNames := 0
    open : f, bookFileName, read
    if f not= 0 then
        loop
            exit when eof (f)
```

```
            read : f, entry
            numNames := numNames + 1
            names (numNames) := entry.name
        end loop
        close : f
    end if
    put numNames, " names read in"
end readNames
```

The *findName* function examines the array of names and returns the index of the array where the match occurs. This gives the position of the record in the file. If it does not find a match, it returns –1.

```
% Find a matching name in the array of names, returning
% -1 if no matching name is found.
function findName (name : string) : int
    for i : 1 .. numNames
        if name = names (i ) then
            result i
        end if
    end for
    result – 1
end findName
```

The *lookUpName* procedure is used to output the data in the address book that corresponds with a given name. It uses *findName* to determine the location in the data file of the name. If the name is found, then it opens the file and moves to the beginning of the record on disk using

```
seek : f, (recordNumber – 1) * recordSize
```

It then reads in the record and outputs the data (in this case, the address). In the procedure, the line

```
assert f > 0
```

occurs after the **open** statement. This is because we have already determined that the we are reading a record where we have already matched a name.

If the **open** returns a non-positive stream number, it would mean that for some reason the file became unreadable between the initial reading of the file in the *readNames* procedure and this procedure, or we have a bug in our program. In cases like hardware failure an assertion failure is an appropriate response.

When dealing with errors that occur because of incorrect user response, an assertion failure is not appropriate. Instead the user should get an error message and a chance to correct the input (as we do in the main section of the program).

```
% Output the address associated with the name.
procedure lookUpName (name : string)
    var recordNumber : int := findName (name)
    var f : int
    var entry : entryType
    if recordNumber = – 1 then
        put name, " not in address book"
    else
        % Open the file, seek to the beginning of the
        % record and read the entry.
        open : f, bookFileName, read, seek
        assert f > 0
        seek : f, (recordNumber – 1) * recordSize
        read : f, entry
        put "Address: ", entry.address
        close : f
    end if
end lookUpName
```

The *changeAddress* procedure reads the record associated with the name in the same manner as *lookUpName*. It gets the new address from the user. It then seeks back to the beginning of the record and writes the revised record into the data file. When the file is opened, it is going to be both read and written to. This means that the file must be opened for **read**, **write**, and **mod**. If the **mod** is forgotten, the file will be deleted if it already exists. The file must also be opened for **seek** as we use the **seek** command to move to the appropriate location in the file before reading and writing the record.

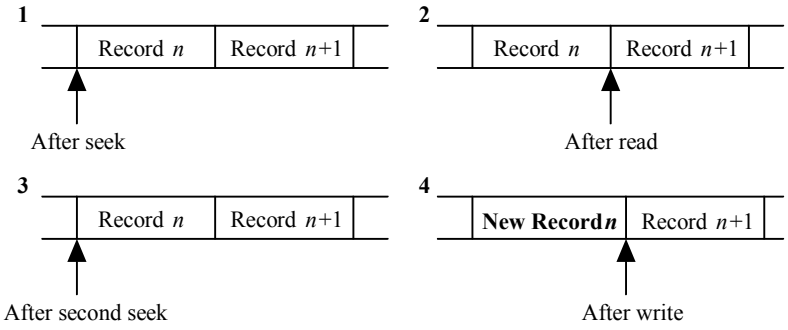


Figure 18.1 Changing a Record

```
% Change the address associated with the name
procedure changeAddress (name : string)
    var recordNumber : int := findName (name)
    var entry : entryType
    var f : int

    if recordNumber = – 1 then
        put name, " not in address book"
    else
        % Open the file, seek to the beginning of the
        % record and read the entry.
        open : f, bookFileName, read, write, mod, seek
        assert f > 0
        seek : f, (recordNumber – 1) * recordSize
        read : f, entry
        % Now get the new address, seek back to the
```

```
        % beginning of the record and write the new entry.
        put "Old Address: ", entry.address
        put "New Address: " ..
        get entry.address : *
        seek : f, (recordNumber – 1) * recordSize
        write : f, entry
        close : f
    end if
end changeAddress
```

The *addEntry* procedure must add the new name to the array and add the record to the data file. To open the file, we open the file for **write**, **mod**, and **seek**.

Once the file has been opened, we must move to the end of the file. To do so, we use

```
seek : f, *
```

which moves the file pointer to the end of the file. We could also have used

```
seek : f, (numNames – 1) * recordSize
```

as that is also the location of the end of the file.

```
% Add an entry to the end of the array and to the end of
% the file.
procedure addEntry (name, address : string)
    var f : int
    var entry : entryType
    % Add to the array of entries
    numNames := numNames + 1
    names (numNames) := name
    % Add the record to the file
    entry.name := name
    entry.address := address
    % Open the file, seek to the end of the file and
    % write the record.
    open : f, bookFileName, write, mod, seek
    assert f > 0
    seek : f, *
```

```
    write : f, entry
    close : f
end addEntry
```

The *deleteEntry* procedure is a place holder. When designing a program for future expansion, it is worthwhile to create the procedures that will be needed, even if the actual procedure has not yet been created. This makes it much easier to add in the needed section without having to worry about changing every aspect of the program. The place holder *deleteEntry* procedure is in the spirit of a step-wise refinement where we leave the further refinements for a later time.

```
% We do not implement this yet. However, we do leave
% a place for it to make it easy to add to this program
% later.
procedure deleteEntry (name : string)
    put "Deletion is not yet supported"
end deleteEntry
```

The main program reads in a user choice and then, if necessary gets a name and/or address from the user. It then calls the relevant procedure.

```
% Main Program
var name, address : string
var choice : int

% Read the names in the array
readNames

loop
    % Get the user's choice
    put "Enter 1 .. Look up a name"
    put "      2 .. Change an address"
    put "      3 .. Add a name"
    put "      4 .. List all names"
    put "      5 .. Delete a Name"
```

```
    put "      6 .. Exit"
    put "Choice: " ..
    get choice

    if choice = 1 then
        % Look up a name.
        put "Enter the name: " ..
        get name : *
        lookUpName (name)
    elsif choice = 2 then
        % Change the address associated with a name.
        put "Enter the name: " ..
        get name : *
        changeAddress (name)
    elsif choice = 3 then
        % Add an entry to the address book.
        put "Enter the new name: " ..
        get name : *
        put "Enter the address: " ..
        get address : *
        addEntry (name, address)
    elsif choice = 4 then
        % List all the names in the address book.
        put "Names in address book"
        for i : 1 .. numNames
            put names (i)
        end for
    elsif choice = 5 then
        % Delete an entry in the address book.
        put "Enter the name: " ..
        get name : *
        deleteEntry (name)
    elsif choice = 6 then
        % Exit the loop
        exit
    else
        % Invalid choice
        put "Choice must be between 1 and 6"
    end if
end loop
```

18.11 Deletion of Records on Disk

We have seen how records can be added to a binary file on disk. We now deal of deleting records from a file.

In the previous examples, when we added records, we simply appended the records to the end of the file. To delete records is somewhat more complicated. When dealing with an array in memory, if you want to delete an element in the middle, you move all the array elements above that element down one (see Section 17.5). However, when dealing with a large file on disk, the same technique is not practical. The time needed to read each record in the file from disk and write the record back onto the disk may well be prohibitive.

Instead, we overwrite the record, marking it as deleted. When we next add a record, we go through the file looking for records that we previously marked deleted and write the new record at that location. If there are no records marked deleted, then we add the new record to the end of the file.

To add deletion to the previous example, we add a constant and change two procedures: the *addEntry* and *deleteEntry* procedures. The constant is the name used when we want to mark an entry deleted.

```
const deletedEntry : string := "[Deleted Entry]"
```

The string is chosen to make certain that we will never have a legitimate entry in the address book with that name.

The *deleteEntry* procedure searches for the name in the array using the *findName* function. If it find it, it modifies the name in the array to Ò[Deleted Entry]Ó. It then seeks to the beginning of the corresponding record in the data file and writes a record with the name and address of Ò[Deleted Entry]Ó.

```
% Delete an entry in the address book by
% overwriting it with a deleted entry marker.
procedure deleteEntry (name : string)
    var recordNumber : int := FindName (name)
```

```
    var entry : entryType
    var f : int

    if recordNumber = – 1 then
        put name, " not in address book"
    else
        % Change the entry in the names array
        names (recordNumber) := deletedEntry

        % Open the file, seek to the beginning of the
        % record and write the entry.
        entry.name := deletedEntry
        entry.address := deletedEntry
        open : f, bookFileName, write, mod, seek
        assert f > 0
        seek : f, (recordNumber – 1) * recordSize
        write : f, entry
        close : f
    end if
end deleteEntry
```

The *addEntry* procedure is modified so that it looks for the name Ò[Deleted Entry]Ó. If it finds it, it writes the new name in the array and writes the new name and address at the corresponding location in the file. If it doesn't find an entry with the name Ò[Deleted Entry]Ó, it increments the number of names in the address book and sets the name to be inserted at the end of the array and the records to be written at the end of the file.

```
% Add an entry to the end of the array and over a record
% marked deleted or to the end of the file.
procedure addEntry (name, address : string)
    var recordNumber : int := findName (deletedEntry)
    var f : int
    var entry : entryType
    % Add the record to the file
    entry.name := name
    entry.address := address

    if recordNumber = – 1 then
```

```
            % There are no deleted entries, add the new
            % entry the array and indicate that the new record
            % be written at the end of the file.
            numNames := numNames + 1
            recordNumber := numNames
        end if

        % Add to the array of entries
        names (recordNumber) := name
        % Open the file, seek to location in the file and
        % write the record.
        open : f, bookFileName, write, mod, seek
        assert f > 0
        seek : f, (recordNumber – 1) * recordSize
        write : f, entry
        close : f
    end addEntry
```

In programs where a large number of records have been deleted, it is common to write a compressor program. This program opens up the old data file for reading and a new data file for writing. It then reads records from the old data file and writes the records to the new data file only if they are not marked deleted. In this way, the new data file has no deleted records.

18.12 Exercises

1. Write a program to prepare a disk file of records called *OnHand* suitable for use in the *HouseLocator* program of this chapter. In the *HouseLocator* program the records are read from the file *HouseInfo* by the procedure *readFile*. Change this procedure so that records for *houses* are read from the disk file *OnHand*. You can use the Turing editor if you are having difficulty prepare the file *OnHand*.
2. Use the *SortHouseInfo* program to sort the records you stored on the disk file *OnHand* in question 1.

3. A simple record consists of two fields: a name and a phone number. Write a program to prepare a disk file called *Mine* of such records and put your personal telephone list in it. Adapt the *SortHouseInfo* program of the chapter to sort your file *Mine* alphabetically.
4. Add a new entry to file *Mine* of the previous question so that the file is still sorted.
5. Delete an entry from the file *Mine* of personal phone numbers so that there are no deleted records in the file.
6. Write and test a subprogram that uses the shell sort method to sort the records in an existing binary file called *Inventory*. The records, which have fields *item* and *price* should be sorted in descending order according to the price.
7. Write a procedure to input a series of records for the above record structure into the binary file *Sample*.
8. Write a function that calculates the number of records in the file *Sample*. Does this correspond to the number of records you created when you ran Exercise #8?
9. Modify the *AddressBook* program to check that the number of records in the file never exceeds the maximum size of the array in memory.

18.13 Technical Terms

record
field of record
record declaration
file of records
text file
binary file
bind **to record**
write **statement**
read **statement**
seek **statement**
tell **statement**
close **statement**
record **data type**
input of record
output of record
array of records
access to record
random access to record

Chapter 19

Advanced Topics

19.1 Binary Search

In this chapter we include a number of longer, more advanced programs which involve some important ideas. The first is a method of searching for information in an array of records that are kept in alphabetical order called **binary search**. The next is a method of sorting an array of records using a recursive method called **successive merge sort**. The third is an efficient method of keeping a list of records in alphabetical order even though there are many changes in the file, both insertions and deletions. This is in a **linked list**.

The last example is one where a user is interacting with a character graphic display in a rapid fashion as is needed in many computer games. This we call **highly interactive graphics**.

A file of student records is stored in an array in such a way that the records are in alphabetic order of the *name* fields of the records. We say the *name* field is the **key** to the ordering. In the chapter on arrays we showed a method of searching that compared a name you were looking for in an array in turn with each of the elements, starting at the first until the search was over. This is a **linear search** method and is fine for short lists. If you have a list of 50 names, on the average you would have to compare the name you sought with 25 names before you found it.

The method of binary search is much more efficient for long lists that have been sorted into order. To search you first compare the name sought with the name in the middle record of the array. If the name sought comes alphabetically before the name in the middle then we discard the second half of the array; the name sought must be in the first half. In this way we cut the size of the array to be searched in half using one comparison. Next we compare the name sought with the middle record of the upper half of the array, again discarding the portion where the name could not possibly be.

Here is a program for searching using the binary search technique.

```
% The "BinarySearch" program
% Finds a particular record in a sorted file
% using the binary search method
type studentType :
    record
        name : string (30)
        address : string (40)
        phone : string (8)
    end record
const maxFile := 500
var studentFile : array 1 .. maxFile of studentType
var count : int

procedure readFile
    % Uses global variables
    var students : int
    open : students, "SortedStudents", get
    assert students > 0
    count := 0
    loop
        get : students, skip
        exit when eof (students)
        count := count + 1
        get : students, studentFile (count).name : 30,
            studentFile (count).address : 40,
            studentFile (count).phone : 8
    end loop
    put "There are ", count, " students in file"
end readFile

procedure search (studentFile : array 1 .. * of studentType,
        var key : string (*), count : int, var place : int)
    var first, last, middle : int
    if length (key) <= 30 then
        % Pad with blanks
        key := key + repeat (" ", 30 – length (key))
    end if
```

```
    % Initialize the binary search
    first := 1
    last := count
    % Search until one element is left
    % If key sought is in list this will be it
    loop
        middle := (first + last) div 2
        if studentFile (middle).name >= key then
            % Discard last half of list
            last := middle
        else
            % Discard first half of list including middle
            first := middle + 1
        end if
        % exit when only one record left
        exit when first >= last
    end loop
    if studentFile (first).name = key then
        place := first
    else
        place := 0
    end if
end search

readFile
var nameSought : string (30)
loop
    var place : int
    put "Enter name of student to be found (\"stop\" to exit)"
    get nameSought : *
    exit when nameSought = "stop"
    search (studentFile, nameSought, count, place)
    if place not= 0 then
        bind sought to studentFile (place)
        put "name" : 30, "address" : 40, "phone" : 8
        put sought.name : 30, sought.address : 40,
            sought.phone : 8
    else
        put "Not in file"
    end if
end loop
```

The binary search is very efficient for large files. A list of 256 records can be searched using 8 comparisons each comparison dividing it into half until only 1 record is left. The linear search would on the average require 128 comparisons. Quite a difference!

19.2 Sorting by Merging

If two lists are each sorted into alphabetic order then a single ordered list can be produced by merging the two. We will assume that the two lists are stored in a single array of records called *studentFile*; the one list goes from element *first* to element *middle* of the array and the second list from *middle + 1* to *last*.

Here is a procedure for merging the two lists. It uses a working area called *temp* which is an array of the same length as the total list.

```
procedure merge (var studentFile : array 1 .. * of studentType,
        first, middle, last : int)
    % Merges two sorted lists of records one going from
    % first to middle the other from middle + 1 to last
    var temp : array 1 .. last of studentType
    % Initialize pointers to the two sorted lists
    var point1 := first
    var point2 := middle + 1
    for point3 : first .. last
        %  point3 locates item in  merged list
        if point1 < middle + 1 and (point2 > last or
                studentFile (point2).name >
                studentFile (point1).name) then
            % Move record from first half
            temp (point3) := studentFile (point1)
            % Advance pointer 1
            point1 := point1 + 1
```

```
        else
            % Move record from second half
            temp (point3) := studentFile (point2)
            % Advance pointer 2
            point2 := point2 + 1
        end if
    end for
    % Copy merged array back to original place
    for point3 : first .. last
        studentFile (point3) := temp (point3)
    end for
end merge
```

This *merge* procedure can be used by a recursive procedure called *mergesort* to sort an unsorted file stored in the array *studentFile*. Here is the *mergeSort* procedure.

```
procedure mergesort (var studentFile : array 1 .. * of studentType,
        first, last : int)
    if last > first then
        const middle := (first + last) div 2
        mergesort (studentFile, first, middle)
        mergesort (studentFile, middle + 1, last)
        merge (studentFile, first, middle, last)
    end if
end mergesort
```

The *mergesort* procedure calls itself until in a call the *last* is equal to *first* and then there is only one element in the file and the file is sorted.

Here is a program that uses *merge* and *mergesort*.

```
% The "MergeSort" program
type studentType :
    record
        name : string (30)
        address : string (40)
        phone : string (8)
```

```
    end record
var maxFile := 100
var studentFile : array 1 .. maxFile of studentType
var count : int

(copy procedure readFile of BinarySearch opening file "UnsortedStudents")

(copy merge procedure here)

(copy mergesort procedure here)

procedure outFile
    % Uses global variables
    % Output file in order
    put "name" : 30, "address" : 40, "phone" : 8
    for i : 1 .. count
        put studentFile (i).name : 30,
            studentFile (i).address : 40,
            studentFile (i).phone : 8
    end for
end outFile

% This is the main program
readFile
mergesort (studentFile, 1, count)
outFile
```

19.3 Files of Records in Linked Lists

Sometimes records must be kept constantly in a sorted order even though records are being added or deleted repeatedly. One way to maintain such a sorted file, without constantly resorting the records, is to use the Turing Environment editor and insert or delete records as suggested in the chapter on records. Another way is to keep the file in memory in the form of a **linked list**.

In a linked list the actual physical order of records in an array of records does not matter. But each record has an extra field that indicates where the next record in the ordered sequence is to

be found. This is called the **link** or **pointer** to the next record. The last record in the linked list has a link which is zero (null) to indicate that there are no more records.

Here is a program that allows insertions and deletions to a linked list of name and address records that is maintained in alphabetical order. It also allows you to list the file in order. The list could be used to print mailing labels. In the program there are two linked lists being maintained: one of the label records, the other of vacant record spaces in the array. The variable *first* points to the first record in the list of labels; the variable *vacant* points to the first vacant space.

```
% The "MaintainLinkedList" program
% Allows insertions and deletions to list
% Permits listing in order
type labelType :
    record
        name : string (20)
        address : string (25)
        link : int
    end record
const null := 0
% Avoid using keyword label
const maxLabels := 100
var labels : array 1 .. maxLabels of labelType
% Initialize pointers to beginning of linked lists
% The label record list is empty
var first := null
% Initialize links of vacant array
var vacant : int := 1
% Each vacant record in array points to next
for i : 1 .. maxLabels – 1
    labels (i).link := i + 1
end for
% Last record in array has null link
labels (maxLabels).link := null

procedure insert (newLabel : labelType)
    % Uses global variables
    % Insert a new label record
```

```
    % Obtain a vacant space for new label record
    assert vacant not= null
    var spot := vacant
    vacant := labels (vacant).link
    % Place new label record in vacant space
    labels (spot).name := newLabel.name
    labels (spot).address := newLabel.address
    % See if new label goes first in list
    if first = null or newLabel.name < labels (first). name then
        labels (spot).link := first
        first := spot
    else
        % Find place to insert new label record
        var previous := first
        var next := labels (first).link
        loop
            exit when next = null or
                newLabel.name < labels (next).name
            previous := next
            next := labels (next).link
        end loop
        % Fix links to make insertion
        labels (previous).link := spot
        labels (spot).link := next
    end if
end insert

procedure delete (oldLabel : labelType)
    % Uses global variables
    % Find label to be deleted from linked list
    var old : int := first
    var previous : int
    loop
        exit when old = null or labels (old).name = oldLabel.name
        previous := old
        old := labels (old).link
    end loop
    % Remove label record from linked list
    if old = null then
        put oldLabel.name, " not found"
        return
    elsif first = old then
```

```
            first := labels (old).link
        else
            labels (previous).link := labels (old).link
        end if
        % Return unused record to vacant list
        labels (old).link := vacant
        vacant := old
    end delete

    procedure outputList
        % Output linked list in order
        var current := first
        loop
            exit when current = null
            put labels (current).name : 20,
                labels (current).address : 25
            current := labels (current).link
        end loop
    end outputList

    % Respond to commands
    var newLabel, oldLabel : labelType
    var reply : int
    const inserts := 1
    const deletes := 2
    const lists := 3
    const stops := 4
    loop
        put "Do you want to 1 - insert, 2 - delete, 3 - list, 4 - stop? " ..
        get reply
        case reply of
            label inserts :
                put "Enter name of label to be inserted"
                get newLabel.name
                put "Enter address"
                get newLabel.address
                insert (newLabel)
            label deletes :
                put "Enter name of record to be deleted"
                get oldLabel.name
                delete (oldLabel)
```

```
        label lists :
            outputList
        label stops :
            exit
        label :
            put "Bad command, try again"
    end case
end loop
```

This is a rather long example and you may not be able to follow it all. Remember: there are two linked lists, one with valid data, the other vacant. As a record is inserted, an element is removed from the vacant list and added to the valid list. Deletion produces the reverse effect.

In the *delete* procedure, if we come to the end of the linked list and do not find the name that is to be deleted, the value of *old* will be *null*. In this case we output an error message and say that the name you asked to have deleted was not found in the list. After the error message, a **return** statement occurs. This causes an exit from the *delete* procedure to the statement in the main program following its calling statement.

19.4 Highly Interactive Graphics

Many computer games are based on animated graphics. In these games the person playing is required to respond rapidly to some event pictured in the window such as a ghost, or a monster, or a witch approaching who must be warded off with the correct magic spell. (Unfortunately many computer games are far more violent.) Your response must be to press one of three keys on the keyboard before it is too late. To ward off the ghost a *g* is correct, *m* for monster, and *w* for the witch. If you press the wrong key or do not depress a key at all before the menace reaches the bottom of the window you lose the game.

We will use three procedures called *ghost, monster*, and *witch* to draw a picture whose top center is at the location (row, column). When we, for example, call the *witch* procedure with

```
witch (5, 20)
```

the picture of a witch is drawn with its top center at location (5, 20) in the window. The height of each picture is 8 rows. The width is 7 columns. The picture will be animated beginning with its top center in row 1 and in a randomly chosen column between 4 and 76 inclusive. The picture will move straight down the window. When the top center reaches row 18 the picture will be at the bottom of the window.

Here is the game program. It uses the predefined procedure getch which will input a single character. As well the boolean function hasch is used. The value of hasch is true if a character has been typed since the last getch was executed.

```
% The "Spooky" program
% A game for a dark night

procedure witch (row, column : int, symbol : string (1))
    locate (row, column)
    put symbol ..
    locate (row + 1, column – 1)
    put repeat (symbol, 3) ..
    locate (row + 2, column – 2)
    put repeat (symbol, 5) ..
    locate (row + 3, column – 4)
    put repeat (symbol, 8) ..
    locate (row + 4, column – 2)
    put symbol, repeat (" ", 3), symbol ..
    locate (row + 5, column – 2)
    put symbol, repeat (" ", 3), symbol ..
    locate (row + 6, column – 1)
    put symbol, " ", symbol ..
    locate (row + 7, column)
    put symbol ..
end witch
```

```
procedure monster (row, column : int, symbol : string (1))
    % you fill this in
    locate (row, column)
    put repeat (symbol, 2) ..
end monster

procedure ghost (row, column : int, symbol : string (1))
    % you fill this in
    locate (row, column)
    put repeat (symbol, 3) ..
end ghost

procedure animate (Time, column : int,
        procedure menace (row, column : int,
        symbol : string (1)),
        magic : string (1), var win : boolean)
    var finished : boolean := false
    var spell : string (1)
    %Suppress cursor and echo of input character
    setscreen ("nocursor")
    setscreen ("noecho")
    win := false
    % Move menace from top to bottom of window
    for line : 1 .. 17
        % Plot menace using asterisks
        menace (line, column, "*")
        % Test to see if a character has been typed
        if hasch then
            % Input a single character and assign to spell
            getch (spell)
            if spell = magic then
                win := true
            end if
            % Prepare to stop the animation
            finished := true
        end if
        delay (Time)        % Delay before erasing
        % Erase plot of menace
        menace (line, column, " ")
        % Stop animation if finished
        exit when finished
    end for
```

```
    % Unsuppress cursor and echo of input character
    setscreen ("cursor,echo")
end animate

loop
    put "Do you want to play this scary game? (y or n): " ..
    var reply : string (1)
    get reply
    exit when reply not= "y"
    var win : boolean
    cls
    const blue := 9
    color (blue)      % Draw figures in blue
    var count := 0
    var Delay : int
    put "Choose a delay time for animation: " ..
    get Delay
    loop
        % Choose a column at random
        var column : int
        randint (column, 4, 76)
        % Choose a spook at random
        var spook : int
        randint (spook, 1, 3)
        case spook of
            label 1 :
                animate (Delay, column, witch, "w", win)
            label 2 :
                animate (Delay, column, monster, "m", win)
            label 3 :
                animate (Delay, column, ghost, "g", win)
        end case
        if win then
            count := count + 1
        else
            exit
        end if
    end loop
    locate (24, 1)
    put "You warded off ", count, " menaces"
end loop
```

19.5 Exercise

1. Replace the binary search procedure *search* in the *BinarySearch* program by a linear search procedure and compare the speed of operation of the two.
2. How many comparisons are required using a binary search if there are 1024 records in the file.
3. Trace the action of the *MergeSort* program for a file of 10 records where the names are originally in this order

   Bill Teresa Win Inge Wayne
   Penny Ric Mark Chris Harriet

4. Use a procedure like *merge* to merge two files that are stored on the disk under the names *Master* and *Update*.
5. Experiment with a linked list of phone number records by adapting the *MaintainLinkedList* program.
6. Use the *MaintainLinkedList* program as it stands to work with house records. Add an extra command to let you store your linked list of records on disk when you want.
7. Modify the *Spooky* program so that the menace's picture wiggles back and forth at random about the chosen column as it moves down the window. You will have to allow more space for the wiggle so perhaps the column that the menace moves down should be limited to 15 to 65.

19.6 Technical Terms

linked list
link or pointer
null pointer
insertion of record
deletion of record
return
getch
hasch
merge sort
binary search
linear search

echo or suppress echo of input character

setscreen ("echo")

setscreen ("noecho")

draw cursor or suppress cursor

setscreen ("cursor")

setscreen ("nocursor")

Chapter 20

Advanced Pixel Graphics

20.1 Advanced Graphics Concepts

Certain features of pixel graphics cannot be explored until you understand a little more mathematics and the use of arrays and subprograms. In this chapter we will look at how to store complex images in memory so that an image can be repeated. This is particularly important for the animation of images more complex than dots, boxes, or ovals.

Also we will work out subprograms that will be generally useful for displaying statistical information by bar charts or pie charts, and for drawing curves from mathematical formulas or experimental data.

20.2 Drawing a Tilted Box

The box drawn using drawbox is oriented so that its sides are parallel to the sides of the window. We can draw a box at an angle, say *theta* to the sides of the window. This program uses some trigonometry.

Here is the program.

```
% The "TiltedBox" program
% The square box of length size is tilted so that its bottom
% makes an angle theta degrees with x-axis
% The upper left corner of the box is at the center
var theta : real
put "Enter angle of tilt in degrees   " ..
get theta
var size : int
put "Enter size of square box in pixels   " ..
get size
% Draw box
const xcenter := maxx div 2
```

```
const ycenter := maxy div 2
const c := cosd (theta)
const s := sind (theta)
% Draw box in cyan
const x1 := xcenter
const y1 := ycenter
const x2 := x1 + round (size * c)
const y2 := y1 – round (size * s)
const x3 := x2 + round (size * s)
const y3 := y2 + round (size * c)
const x4 := x1 + round (size * s)
const y4 := y1 + round (size * c)
drawline (x1, y1, x2, y2, cyan)
drawline (x2, y2, x3, y3, cyan)
drawline (x3, y3, x4, y4, cyan)
drawline (x4, y4, x1, y1, cyan)
var reply : string (1)
getch (reply)
```

20.3 Repeating a Pattern

Here is a subprogram called *drawflag* that will draw a pattern that is *w* pixels wide and *h* pixels high with its lower left corner at (*x, y)*

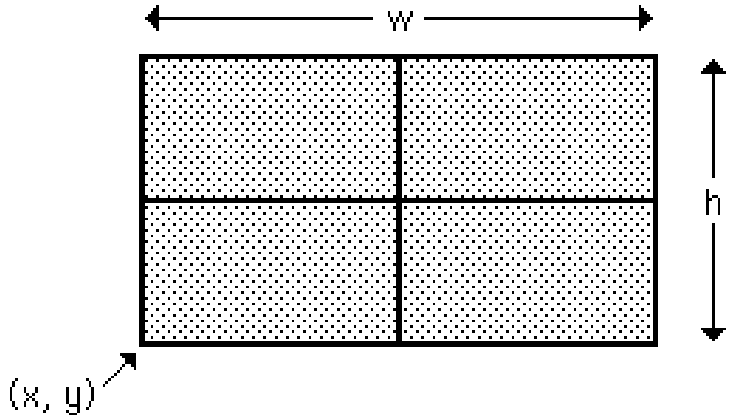


Figure 20.1 Output of the *drawflag* Procedure

in two colors *color1* and *color2*.

```
procedure drawflag (x, y, w, h, color1, color2 : int)
    const halfw := round (w / 2)
    const halfh := round (h / 2)
    drawbox (x, y, x + w, y + h, color2)
    drawfill (x + 1, y + 1, color1, color2)
    drawline (x + halfw, y, x + halfw, y + h, color2)
    drawline (x, y + halfh, x + w, y + halfh, color2)
end drawflag
```

Here is a program that produces this wallpaper design

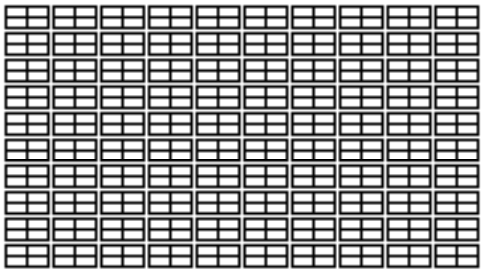


Figure 20.2 Output of the *DrawDesign* Program

using the *drawflag* procedure to repeat the flag pattern.

```
% The "DrawDesign" program
% Repeats a pattern in window
setscreen ("graphics")

(include procedure drawflag here)

% Draw a magenta and green pattern
const width := 20
const height := 10
for xcount : 0 .. 9
    for ycount : 0 .. 9
        const x := xcount * (width + 2)
        const y := ycount * (height + 2)
        drawflag (x, y, width, height, green, magenta)
    end for
end for
var reply : string (1)
getch (reply)
```

Another way of programming this design would be to draw the flag once then use the predefined procedure takepic to record the pattern in a buffer array and then use drawpic repeatedly to place the pattern anywhere you want in the window.

The procedure takepic records the pixel values that are in a rectangle with bottom left and top right corners (*x1, y1*) and (*x2, y2*) in an integer array called *buffer*. The form of takepic is

takepic (*x1*, *y1*, *x2*, *y2*, *buffer*)

To determine what size the integer array *buffer* should be you can use the predefined function sizepic in the form

sizepic (*x1*, *y1*, *x2*, *y2*)

To reproduce a picture stored in *buffer* by takepic with its lower left corner at (*x, y*) you use *drawpic* in the form

drawpic (*x*, *y*, *buffer*, *picmode*)

where *picmode* has a value 0 if you want to reproduce the original picture exactly, or 1 if you want to superimpose the picture onto the pixels it is covering. Picmode 1, also called **XOR mode**, combines the picture with the covered pixels in a way that

effectively shows both the picture and what it is covering. Picmode 1 is used mainly when you want to erase a picture you have drawn, as you do in animation. Drawing the same picture with picmode 0 first then with picmode 1 draws the picture then erases it.

Here is a program to make the same pattern as the *DrawDesign* program.

```
% The "DrawDesign2" program
% Repeats a pattern in window
setscreen ("graphics")

(include procedure drawflag here)

const width := 20
const height := 10
% Draw flag in bottom left-hand corner
drawflag (0, 0, width, height, magenta, green)
var snapshot : array 1 .. sizepic (0, 0, width, height) of int
takepic (0, 0, width, height, snapshot) % record the image
drawpic (0, 0, snapshot, 1) % erase the original

for xcount : 0 .. 9
    for ycount : 0 .. 9
        const x := xcount * (width + 2)
        const y := ycount * (height + 2)
        drawpic (x, y, snapshot, 0)
    end for
end for
var reply : string (1)
getch (reply)
```

20.4 Animation Using a Buffer

Here is the *FlagRise* program that raises the flag up a flagpole using the procedure *drawflag* as well as sizepic, takepic, and drawpic.

```
% The "FlagRise" program
% Raises a flag up a pole

(include procedure drawflag here)

setscreen ("graphics")
cls

% Draw brown flagpole
const poleleft := 150
const polelow := 50
const poleright := poleleft + 3
const polehigh := polelow + 100
drawbox (poleleft, polelow, poleright, polehigh, brown)
drawfill (poleleft + 1, polelow + 1, brown, brown)

% Draw flag at bottom of pole
const flagheight := 20
const flagwidth := 40
const flagleft := poleright + 1
const flaglow := polelow
const flagright := flagleft + flagwidth
const flaghigh := flaglow + flagheight
drawflag (flagleft, flaglow, flagwidth, flagheight, green, red)

% Record flag picture in buffer
% First declare buffer named camera
var camera : array 1 .. sizepic (flagleft, flaglow,
    flagright, flaghigh) of int
takepic (flagleft, flaglow, flagright, flaghigh, camera)
for y : polelow + 1 .. polehigh – flagheight
    % Erase former picture
    drawpic (flagleft, y – 1, camera, 1)
    % Draw picture in higher position
    drawpic (flagleft, y, camera, 0)
    sound (y * 25, 1)
end for
var reply : string (1)
getch (reply)
```

20.5 Bar Charts

Statistical data is often represented by bar charts or pie charts. When we are displaying information in bar charts we try to show as much contrast as possible between the height of the bar that represents the largest value and the one representing the smallest.

Figure 20.3 Output of the *BudgetBarChart* Program

Here is a program to represent the relative amounts a person spends on various expenses in a month in a bar chart.

```
% The "BudgetBarChart" program
% Displays monthly expenses graphically

function maxlist (list : array 1 .. * of int, listlength : int) : int
    % Finds the largest element of list
    var maximum := list (1)
    for i : 2 .. listlength
        maximum := max (maximum, list (i))
    end for
    result maximum
end maxlist

procedure barchart (item : array 1 .. * of int, itemcount : int)
    % Draws a bar chart representing array item
    const chartleft := maxx div 2
```

```
    const chartlow := maxy div 2 + 20
    const chartwidth := maxx div 2
    const chartheight := maxy div 2 – 30
    const barspacing := round (chartwidth / itemcount)
    const barwidth := barspacing – 3
    const barheightscale := chartheight / maxlist (item, itemcount)

    for i : 0 .. itemcount – 1
        const barleft := chartleft + i * barspacing
        const barheight := round (item (i + 1) * barheightscale)
        const Color := i mod maxcolor + 1
        drawfillbox (barleft, chartlow,
            barleft + barwidth, chartlow + barheight, Color)
    end for
end barchart

setscreen ("graphics")
var expense : array 1 .. 8 of int
put "Enter monthly expenses to nearest dollar"
put "Rent =" ..
get expense (1)
put "Food = " ..
get expense (2)
put "Clothing = " ..
get expense (3)
put "Insurance = " ..
get expense (4)
put "Car cost = " ..
get expense (5)
put "Taxes = " ..
get expense (6)
put "Utilities = " ..
get expense (7)
put "Entertainment = " ..
get expense (8)
barchart (expense, 8)
var reply : string (1)
getch (reply)
```

20.6 Pie Charts

We will now give procedures that can replace the procedure *barchart* in the previous program so that a pie chart such as

```
Enter monthly expenses to nearest dollar
Rent = 275
Food = 150
Clothing = 65
Insurance = 10
Car cost = 80
Taxes = 80
Utilities = 20
Entertainment = 100
```

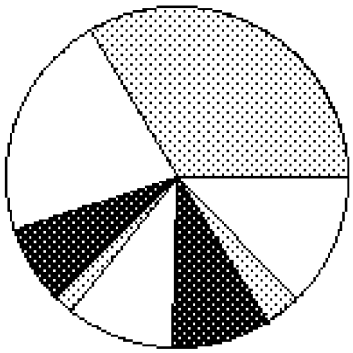


Figure 20.4 Output of the *BudgetPieChart* Program

can be drawn from the same data by the single statement

piechart (*expense*, 8)

We will use a procedure called *piechart* along with procedures called *fillslice* and *sumlist* .

```
% The "BudgetPieChart" program
% Displays monthly expenses graphically

function sumlist (list : array 1 .. * of int, listlength : int) : int
    % Sum the elements of list
    var sum := list (1)
    for i : 2 .. listlength
        sum := sum + list (i)
    end for
    result sum
end sumlist
procedure piechart (item : array 1 .. * of int, itemcount : int)
```

```
    % Draw pie chart
    const xcenter := 3 * maxx div 4
    const ycenter := maxy div 2
    const radius := maxy div 2
    var initialAngle : int
    var finalAngle := 0
    const itemscale := 360 / sumlist (item, itemcount)
    var sumItems := 0.0
    for i : 1 .. itemcount
        sumItems := sumItems + item (i)
        initialAngle := finalAngle
        finalAngle := round (sumItems * itemscale)
        const Color := i mod maxcolor + 1
        drawfillarc (xcenter, ycenter, radius, radius,
            initialAngle, finalAngle, Color)
    end for
end piechart

setscreen ("graphics")
var expense : array 1 .. 8 of int
put "Enter monthly expenses to nearest dollar"
put "Rent =" ..
get expense (1)
put "Food = " ..
get expense (2)
put "Clothing = " ..
get expense (3)
put "Insurance = " ..
get expense (4)
put "Car cost = " ..
get expense (5)
put "Taxes = " ..
get expense (6)
put "Utilities = " ..
get expense (7)
put "Entertainment = " ..
get expense (8)
piechart (expense, 8)
var reply : string (1)
getch (reply)
```

20.7 Graphing Mathematical Equations

In the introductory chapter on graphics we showed how to draw the graph of a simple curve, a parabola whose equation was

```
y = x ** 2
```

for values of *x* going from 0 to 14. We had to work out what scale was appropriate so that the graph would take up most of the window.

Now we will show a graph plotting program that uses pixels and works out on an appropriate scale for you. To use the program you must first calculate the values of *x* and *y* to be plotted and then call the procedure *graphplot* in the form.

```
graphplot (x, y, pointcount, graphtitle)
```

where *x* and *y* are the arrays of real values for *x* and *y*. There are *pointcount* such values. The graph will be labelled by *graphtitle.*

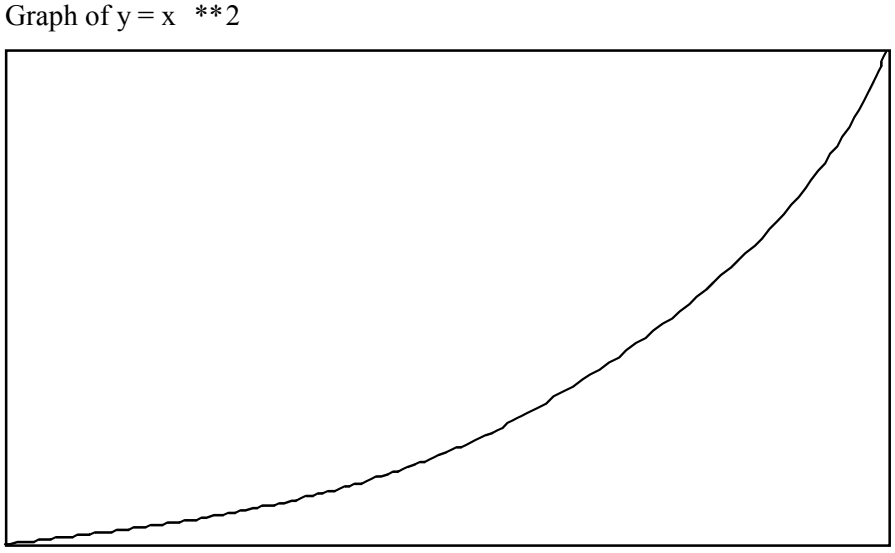


Figure 20.5 Output of the *DrawParabola* Program

Here is the program that would plot the parabola we had before.

```
% The "DrawParabola" program
% Plots a graph of y = x ** 2
% for values of x from 0 to 14
% Number of points to be plotted is 100
setscreen ("graphics")

(include the following procedures minmax and graphplot here)

const pointcount := 100
var x, y : array 1 .. pointcount of real
const xleft := 0.0
const xright := 14.0
const xdifference := (xright – xleft) / (pointcount – 1)
% Compute corresponding values of x and y
for i : 1 .. pointcount
    x (i) := xleft + (i – 1) * xdifference
    y (i) := x (i) ** 2
end for
graphplot (x, y, pointcount, "Graph of y = x ** 2")
```

Here are the procedures that you must include.

```
procedure minmax (list : array 1 .. * of real,
        listlength : int, var listmin, listmax, listrange : real)
    % Find the minimum, maximum and range of the list
    listmin := list (1)
    listmax := list (1)
    for i : 2 .. listlength
        listmin := min (listmin, list (i))
        listmax := max (listmax, list (i))
    end for
    listrange := listmax – listmin
end minmax

procedure graphplot (x, y : array 1 .. * of real,
        pointcount : int, title : string)
```

```
    % Label graph at top of window
    locate (1, 1)
    put title
    % Layout plotting area
    const plotleft := 0
    const plotlow := 0
    const plotright := maxx
    const plothigh := maxy – 20
    drawbox (plotleft, plotlow, plotright, plothigh, 1)

    var xmin, xmax, xrange : real
    var ymin, ymax, yrange : real
    minmax (x, pointcount, xmin, xmax, xrange)
    minmax (y, pointcount, ymin, ymax, yrange)
    const xscale := (plotright – plotleft) / xrange
    const yscale := (plothigh – plotlow) / yrange

    % Draw axes if in plotting area
    if ymin <= 0 and ymax > 0 then
        % Plot x-axis
        const yzero := round ( – ymin * yscale) + plotlow
        drawline (plotleft, yzero, plotright, yzero, 2)
    end if
    if xmin <= 0 and xmax > 0 then
        % Plot y-axis
        const xzero := round ( – xmin * xscale) + plotleft
        drawline (xzero, plotlow, xzero, plothigh, 2)
    end if

    % Plot graph
    for i : 1 .. pointcount
        const xplot := round ( (x (i) – xmin) * xscale) +
            plotleft
        const yplot := round ( (y (i) – ymin) * yscale) +
            plotlow
        drawdot (xplot, yplot, 3)
    end for
end graphplot
```

Notice that in putting the label on the graph in the *graphplot* procedure we used the predefined procedure locate rather than

locatexy. This works the same way as *locate* for character graphics in the form

```
locate (row, column)
```

except that for pixel graphics there are only 40 columns.

20.8 Exercises

1. Create a subprogram from the *TiltedBox* program of this chapter with parameters that let you change its color, size, and tilt. Now write a main program that draws a series of such boxes, with one common corner, tilted at angles ranging from 0 to 360 degrees at intervals of 10 degrees and where the size of the box increases as you go to larger angles. Choose a scale that keeps the final box completely in the window.
2. Try making a basic pattern and replicating it all over the window. Can you devise a more complicated replication than just an all-over pattern?
3. Create a bar chart using the *barchart* procedure in this chapter to represent the relative lengths of the various chapters in the this book.
4. Create a pie chart using the *piechart* procedure in this chapter to represent the relative amount of time you spend on the different courses you are studying this year.
5. Modify the program *BudgetBarChart* to plot a graph of the parabola *y* = *x***2 from 0 to 14. Make the first bar represent *x*=0, the second *x*=1, and so on. There will be 15 bars in your chart.
6. Use the *graphplot* procedure to graph the equation *y* = *sin*(*x*) for *x* going from 0 to 720 degrees.
7. Modify the graph plotting procedure *graphplot* so that instead of plotting closely spaced dots you plot a series of straight lines between more widely spaced points. Use this modified procedure to plot a graph of experimental data that you invent where the values for *x* are not necessarily uniformly spaced.

8. The *graphplot* procedure assumes that the graph has a single value of *y* for each value of *x* . Many graphs have two values, for example, a circle. How would you plot such graphs? Notice that for a circle of radius *r* that

   x = *r* * sind (*theta*)
   y = *r* * cosd (*theta*)

   Try forming dots from these equations expressing *x* and *y* in terms of *r* and the angle *theta* that the radius from the center makes counterclockwise from the three o'clock position.
9. Animate an apple falling from a tree.
10. A beach ball is made of six pieces of plastic: 2 red, 2 green, and 2 brown. The ball is assembled so that no two pieces of the same color touch each other except at the two end points. Write a program to draw a picture of the inflated beach ball from an end-on point of view. Can you program a side view of the ball? Try it.
11. Write a program with an animated ball bouncing up and down several times.
12. Repeat Exercise 11 but this time, each bounce should be lower than the previous bounce.

20.9 Technical Terms

takepic
sizepic
drawpic
buffer
picmode
XOR mode
bar chart
scaling a graph

Chapter 21

Animation and GUIs

21.1 The Mouse in Turing

This chapter covers a wide variety of graphics topics for use by students writing advanced graphically oriented programs. The chapter also introduces the Turing module, that allows students to write programs that have Graphical User Interface (GUI) components such as buttons, check boxes, and text fields.

Turing has several predefined subprograms to help handle the mouse. These include:

mousewhere,
buttonmoved, and
buttonwait

The mousewhere procedure

```
mousewhere (var x, y, b : int)
```

is used to determine the current location of the mouse cursor. It sets *x* and *y* to the current location of the mouse and sets *b* to 1 if the mouse button is currently pressed, otherwise *b* is set to 0.

Here is an example program called *MouseTrails* that use the mousewhere procedure. It draws a small filled circle around the mouse if the button is pressed. Each time through the program's loop, mousewhere is called to set *x*, *y*, and *b*. If the button is pressed and the mouse location is moved, then a circle is drawn in the new location in a new color.

The *MouseTrails* program only draws to the window when the mouse location has changed. This is a very common technique used to reduce flashing. Without it, on fast computers the program might update the window too frequently, producing an obvious flash.

```
% The "MouseTrails" program.
var x, y, oldx, oldy, b : int := – 1
var c : int := 0
loop
    % Get the current location of the mouse
    mousewhere (x, y, b)
    % if the mouse has moved and the button is pressed.
    if (x not= oldx or y not= oldy) and b = 1 then
        % Draw a circle around the mouse location
        drawfilloval (x, y, 20, 20, c)
        % Change the color
        c := (c + 1) mod 16
        oldx := x
        oldy := y
    end if
end loop
```

Here is what the output of the *MouseTrails* program looks like.

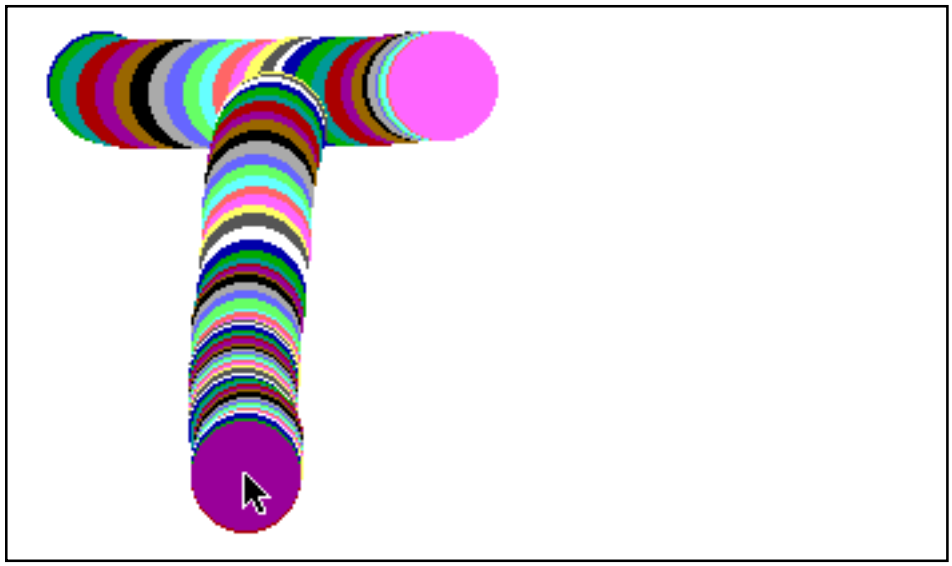


Figure 21.1 The Execution window from *MouseTrails*

The buttonmoved function

```
buttonmoved (direction : string) : boolean
```

also handles mouse activity. It returns **true** if a mouse press or release has occurred. If the *direction* parameter is set to ÒdownÓ, buttonmoved returns **true** if the mouse button has been pressed. If *direction* is set to ÒupÓ, buttonmoved returns **true** if the mouse button has been released. This function can be thought of as hasch for the mouse button.

The buttonwait procedure

```
buttonwait (direction : string, var x, y, buttonnumber,
    buttonupdown : int)
```

waits until a mouse button press has occurred and then returns the location of the mouse in *x* and *y*. It also returns the number of the button pressed and whether the button was pressed or released. We will not be using the last two parameters in our example. This procedure can be thought of as getch for mouse buttons.

The next example program is more complicated. It is a game called *MouseWhack*. In the game, a series of colored squares is displayed in the window in random locations. Each square disappears after a short interval. The object of the game is to click on a colored square before the square disappears. We call the buttonmoved procedure to check if a mouse press has occurred before we call buttonmoved. Calling buttonmoved and then buttonwait prevents execution from pausing until the user presses the mouse button.

The program is organized as a loop within a loop. At the beginning of the outer loop, a random location, color, and duration for the square are chosen. Execution then enters the inner loop. The inner loop checks to see whether the square has been visible for the prescribed duration and exits the inner loop if it has. If it does not exit, the program checks if a mouse button has been pressed by using buttonmoved. If a mouse button has been pressed, it reads the location where the mouse was pressed by using buttonwait.

Because we know from buttonmoved that a mouse button press has occurred, buttonwait returns immediately with the location of the mouse button press. The location is compared to the location of the square and if the button press is within the square, the window is cleared and a message is displayed. The program then terminates by calling **return** in the main program. If the mouse missed the square, the variable holding the number of failed clicks is increased by 1 using the statement

failedClicks += 1

This statement is a short form for

failedClicks = *failedClicks* + 1

Each time the program leaves the inner loop, the variable holding the number of squares missed is incremented by 1. Execution then returns to the top of the outer loop and a new square is drawn.

```
% The "MouseWhack" program.
var failedClicks, missedBlocks : int := 0
var exitLoopTime, currentTime : int
var x, y, clr, duration : int
var mx, my, dummy1, dummy2 : int
const SIZE : int := 20
% The outer loop
loop
    % Set a random location, color, and duration for the square
    randint (x, 0, maxx – SIZE)
    randint (y, 0, maxy – SIZE)
    randint (clr, 1, 15)

    % Draw the square
    drawfillbox (x, y, x + SIZE, y + SIZE, clr)

    clock (exitLoopTime)
    randint (duration, 300, 800)
    exitLoopTime := exitLoopTime + duration
    % The inner loop
    loop
```

```
            % Has the duration of the square expired
            clock (currentTime)
            exit when currentTime > exitLoopTime
            % If a button press has occurred
            if buttonmoved ("down") then
                % get the location of the button press
                buttonwait ("down", mx, my, dummy1, dummy2)
                % Is it within the square
                if x <= mx and mx <= x + SIZE and
                        y <= my and my <= y + SIZE then
                    % You clicked the square!
                    cls
                    put "You WON!  You missed ", failedClicks,
                        " clicks and ", missedBlocks, " blocks."
                    return
                else
                    failedClicks += 1
                end if
            end if
        end loop
        missedBlocks += 1
        % Erase the square by drawing it in the background color
        drawfillbox (x, y, x + SIZE, y + SIZE, colorbg)
    end loop
```

21.2 Animation using the Pic Module

In the previous chapter, we discussed how to animate an object using takepic. The method we used, however, does not work well when drawing an object over a complicated background. In this section, we show how to move an irregularly-shaped object over a complicated background. We will also show how to load an image from a file on disk.

We will be using predefined subprograms with a different naming convention. All the predefined subprograms we have used so far have had names that were lowercase, for example, drawbox. The predefined subprograms used in this section have a two-part name: a module name and a subprogram name

separated by a period. The subprograms belonging to a particular module are all related to a particular purpose. For example, the Pic module handles pictures, the Dir module creates directories and lists their contents, and the Window module manipulates windows. A complete list of the modules and the subprograms within each one can be found in Appendix C in the back of this book. Note that the module name and each word in the subprogram name start with a capital letter, for example Pic.FileNew, Dir.Get, and Window.Open.

The Pic module contains subprograms for creating, drawing, and disposing of pictures. The Pic.FileNew and Pic.New functions are used to create new pictures. Unlike the takepic procedure that saves an area of the window in an integer array buffer that you declare, the Pic.FileNew and Pic.New functions allocate a buffer automatically and return an integer called a **picture ID** that is used in the other Pic subprograms.

The Pic.Draw procedure draws a picture in the window at a specified location. When a picture is no longer needed, Pic.Free must be called to free up the allocated buffer. If too many pictures are allocated without being freed, a program can run out of memory in which to store additional pictures.

The signatures for the Pic subprograms in the program that we will show are:

- Pic.FileNew (*fileName* : **string**) : **int**

  Opens a graphic file on disk and returns the picture ID.

- Pic.ScreenLoad (*fileName* : **string**, *x*, *y*, *mode* : **int**)

  Opens a graphic file on disk and draws it in the window.

- Pic.New (*x1*, *y1*, *x2*, *y2* : **int**) : **int**

  Saves a rectangle with bottom left and top right corners (*x1, y1*) and (*x2, y2*) and returns the picture ID.

- Pic.Draw (*pictureID*, *x*, *y*, *mode* : **int**)

  Draws the picture specified by *pictureID* in the window with lower left corner at (*x, y*).

- Pic.Free (*pictureID* : **int**)

Frees up a picture's buffer when the picture is no longer needed.

Turing can load graphic files from disk. These graphic files can be created using various types of paint programs. On the Windows platform, Turing can load files that are stored in BMP format. Such files have a file suffix of Ò.bmpÓ. On the Macintosh, Turing can load files in PICT format. Graphics files in different formats can be converted to either BMP files or PICT files using a variety of freeware or shareware utilities.

The example program, *MoveLogo*, animates a small picture over top of a larger background picture. The smaller picture (in this case, a Turing logo) moves diagonally, bouncing off the edges of the window. This program needs a small graphic saved as *logo.bmp* and a larger background picture saved as *forest.bmp*.

Figure 21.2 The Execution window from *MoveLogo*

```
% The "MoveLogo" program
% This program animates a graphic around the window.

var logoPicID, bgPicID : int
```

```
% The picture IDs of the logo and the background
var logoWidth, logoHeight : int
% The width and height of the logo picture
var x, y : int := 0
% The lower-left corner of the logo
var dx, dy : int := 3
% The direction of movement of the logo

% Load the logo from file and get its height and width
logoPicID := Pic.FileNew ("logo.bmp")
logoWidth := Pic.Width (logoPicID)
logoHeight := Pic.Height (logoPicID)
% Load the forest picture from the file and draw it in the window
Pic.ScreenLoad ("forest.bmp", 0, 0, picCopy)

loop
    % Take the picture of the background
    bgPicID := Pic.New (x, y, x + logoWidth, y + logoHeight)
    % Draw the logo
    Pic.Draw (logoPicID, x, y, picMerge)
    delay (50)
    % Draw the background over the logo, effectively erasing the logo
    Pic.Draw (bgPicID, x, y, picCopy)
    % Free up the memory used by the background picture
    Pic.Free (bgPicID)
    % Check if the logo bounced off the left or right edges of the
window
    if x + dx < 0 or x + dx + logoWidth > maxx then
        dx := – dx
    end if
    % Check if the logo bounced off the top or bottom edges
    if y + dy < 0 or y + dy + logoHeight > maxy then
        dy := – dy
    end if
    % Change the location of the logo
    x := x + dx
    y := y + dy
end loop
```

After the variable declarations, the program loads in the logo from a BMP file called *logo.bmp* and determines the height and

width of the picture. Pic.FileNew loads the BMP picture from the disk and stores it as a picture. The next two lines specify the picture's height and width. The program then loads the forest image directly to a window using Pic.ScreenLoad. It does not save the picture in memory.

The animation of the logo bouncing around the window occurs in the main loop. With each iteration of the loop, the program saves the part of the window where the logo will be drawn as a picture using Pic.New. It then draws the logo using Pic.Draw over the portion of the window that just had its picture taken. It delays 50 milliseconds and then draws the small background picture over top of the logo using Pic.Draw again, erasing the logo. It then frees up the memory used by the small background picture using Pic.Free. Finally, it calculates the next location for the logo.

Note that when the picture is drawn over the background with Pic.Draw, the picMerge mode is used. The picMerge mode has the effect of not drawing the parts of the picture that are the background color, which is usually white (see Figure 21.3). This is very useful when drawing irregularly shaped objects over a background. Instead of the picture being drawn in a white rectangle, only the non-background pixels are drawn to the window. When the background is redrawn in the second call to Pic.Draw, the picCopy mode is used. This ensures that the original background picture is restored.

Here is a small example of the two drawing modes.

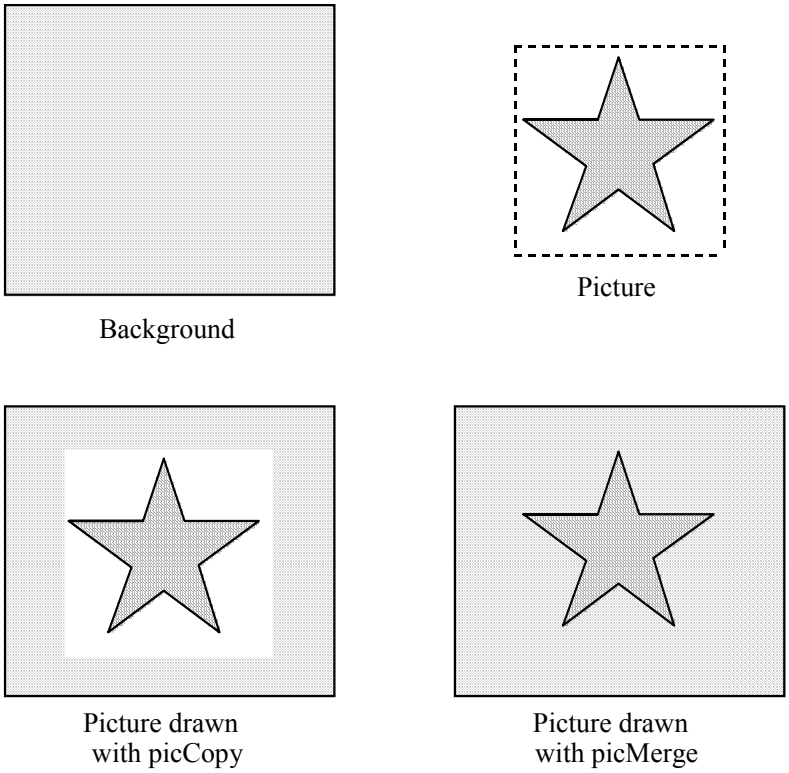


Figure 21.3 The picCopy and picMerge drawing modes

21.3 Animation using the Sprite Module

Sprites are small graphical objects created to move around the window. Turing supports the use of sprites with the Sprite module. Sprites have two advantages over using the Pic module in a Turing program.

The first advantage is that the background does not have to be redrawn by the program. When a sprite is moved, the background in the previous location is automatically restored.

The second advantage is that sprites can be given a priority. When two sprites are drawn over top of each other, the sprite with the higher priority is automatically drawn over top of the sprite with the lower priority. Any non-sprite drawing to the window is considered to be at priority 0. Sprites with negative priorities appear behind the background. That is, the sprite is drawn only on the parts of the window that appear in color 0.

The subprograms used in the following example program are:

- Sprite.New (*picture* : **int**) : **int**
  Creates a new sprite from a picture.
- Sprite.Show (*sprite* : **int**)
  Displays the sprite at its current location and height.
- Sprite.SetPriority (*sprite*, *height* : **int**) : **int**
  Sets the priority of the specified sprite.
- Sprite.SetPosition (*sprite*, *x*, *y* : **int**, *centered* : **boolean**)
  Moves the sprite to a new location with lower-left corner at (*x*, *y*). If *centered* is true, the sprite is centered at the location instead.

Our first example program is called *SpriteLogo*, and is similar to the *MoveLogo* program. The *SpriteLogo* program uses the Sprite module to perform the animation instead of the Pic module.

```
% The "SpriteLogo" program
% This program animates a graphic around the window using
% sprites.

var logoPicID, spriteID : int
% The picture IDs of the logo and the background
var logoWidth, logoHeight : int
% The width and height of the logo picture
var x, y : int := 0     % The location of the logo
var dx, dy : int := 3   % The direction of movement of the logo

% Load the logo from file and get its height and width
logoPicID := Pic.FileNew ("logo.bmp")
```

```
logoWidth := Pic.Width (logoPic)
logoHeight := Pic.Height (logoPic)

% Create the sprite
spriteID := Sprite.New (logoPicID)
Sprite.SetPriority (spriteID, 3)
Sprite.Show (spriteID)

% Load the forest picture from the file and draw it to the window
Pic.ScreenLoad ("forest.bmp", 0, 0, picCopy)

loop
    Sprite.SetPosition (spriteID, x, y)
    delay (50)
    if x + dx < 0 or x + dx + logoWidth > maxx then
        dx := – dx
    end if
    if y + dy < 0 or y + dy + logoHeight > maxy then
        dy := – dy
    end if
    x := x + dx
    y := y + dy
end loop
```

The initial few lines of *SpriteLogo* are identical to the *MoveLogo* program. After the picture is created, the sprite is created using Sprite.New with the picture as a parameter. This returns an integer representing the sprite. After the sprite is created, the priority of the sprite is set with Sprite.SetPriority. After this, the sprite is made visible in its default location using Sprite.Show.

In the loop, it is no longer necessary to store a small segment of the background. The loop contains calls to Sprite.SetPosition to move the sprite to a new location.

Sprites are especially useful when there are several to animate at a time. For example, in the *SpriteLogos* program, arrays are used to store the sprite number, location, and velocity of several logos. Each logo is given a particular priority and will always be drawn above any logo of a lower priority.

This program uses the Rand.Int predefined function of the Rand module. Rand.Int is very similar to randint in that it returns a random integer between its two parameters except that it is a function rather than a procedure. Almost all of the lowercase subprograms have a module.subprogram equivalent.

Figure 21.4 The Execution window from *SpriteLogos*

```
% The "SpriteLogos" program
% This program animates a graphic around the window using sprites

var logoPicID, spriteID : int
% The picture IDs of the logo and the background
var logoWidth, logoHeight : int
% The width and height of the logo picture
var spriteID : array 1 .. 6 of int
var x, y : array 1 .. 6 of int
var dx, dy : array 1 .. 6 of int

% Load the logo from file and get its height and width
logoPicID := Pic.FileNew ("logo.bmp")
logoWidth := Pic.Width (logoPic)
logoHeight := Pic.Height (logoPic)

% Create the sprites
```

```
for i : 1 .. 6
    spriteID (i) := Sprite.New (logoPicID)
    Sprite.SetPriority (spriteID (i), i)
    Sprite.Show (spriteID (i))
    x (i) := Rand.Int (logoWidth, maxx – logoWidth)
    y (i) := Rand.Int (logoWidth, maxy – logoWidth)
    dx (i) := Rand.Int ( – 3, 3)
    dy (i) := Rand.Int ( – 3, 3)
end for

% Load the forest picture from the file and draw it to the window
Pic.ScreenLoad ("forest.bmp", 0, 0, picCopy)

loop
    for i : 1 .. 6
        Sprite.SetPosition (spriteID (i), x (i), y (i))
        if x (i) + dx (i) < 0 or x (i) + dx (i) + logoWidth > maxx then
            dx (i) := – dx (i)
        end if
        if y (i) + dy (i) < 0 or y (i) + dy (i) + logoHeight > maxy then
            dy (i) := – dy (i)
        end if
        x (i) := x (i) + dx (i)
        y (i) := y (i) + dy (i)
    end for
    delay (50)
end loop
```

21.4 The *GUI* Module

Most user interfaces include **buttons**, **check boxes**, **scroll bars**, **text boxes**, and more. Such interfaces are known as **GUI**s or **Graphical User Interfaces**. Individual components of a GUI such as a button or a check box are often called **widgets**. Turing includes a *GUI* module to help create programs that use widgets. We will give three example programs using the *GUI* module in the Turing library. For more information on all the GUI widgets available in Turing, consult the *Turing Reference Manual*.

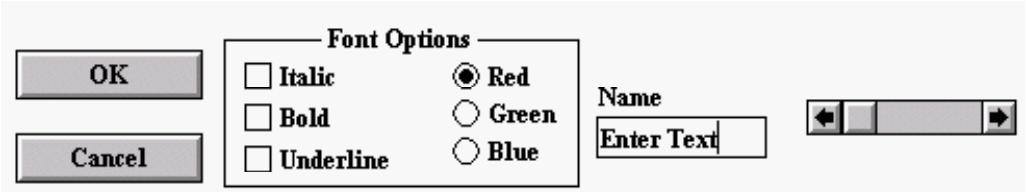


Figure 21.5 Some Turing GUI Widgets

Unlike all the other predefined subprograms we have used so far, the subprograms in the *GUI* module are not automatically included in Turing programs. To make them available, the *GUI* module must be imported explicitly with the line:

```
import GUI
```

This loads the *GUI* module in the Turing library.

Programs that use the *GUI* module have certain common elements. When each widget is created, an **action procedure** is specified. The action procedure is called when the widget is activated. For example, a button has an action procedure that is called when the user presses the button. A check box has an action procedure that is called whenever the user switches the check box from checked to unchecked or back again. A text box has an action procedure that is called whenever the user enters text in the box and presses Enter.

A program using the *GUI* module must have this program segment in order to allow the widgets to respond to the user.

```
loop
    exit when GUI.ProcessEvent
end loop
```

Each time through this loop, *GUI.ProcessEvent* is called to check for user input and handle it appropriately. For example, when the user clicks on a button, *GUI.ProcessEvent* draws the pressed button. When the user releases the mouse, *GUI.ProcessEvent* draws the original appearance of the button and calls the action procedure. If a user enters a keystroke when

a text field is active, *GUI.ProcessEvent* displays the key in the text field.

GUI programs have a very different structure than the Turing programs we have created so far. Rather than a loop containing much of the program logic, GUI programs tend to create all the widgets in the window and then use the *GUI.ProcessEvent* loop to respond to the user interaction by calling various action procedures. Individual mouse clicks, keystrokes, and so on are often called **events**. This model of programming is called **event-driven programming** and is common to most modern GUI programs. The program segment containing the loop that processes the user input is called the **event loop**.

We will now look at a GUI program called *GUICircles*. This program uses two widgets, both buttons. When the user clicks on the first button, the program draws a circle in a random location with a random radius and color. When the user clicks the second button, the program exits out of the event loop and terminates.

The new *GUI* subprograms we will use here are:

- *GUI.CreateButton* (*x*, *y*, *width* : **int**, *text* : **string**,
  actionProc : **procedure** *x* ()) : **int**

  Creates a button with lower-left corner at (*x*, *y*) with a width of *width*. The button is labelled with *text*. The *actionProc* parameter must be the name of a procedure without any parameters. In the parameter declaration, the *x* is a place holder and should be ignored. The empty brackets after the *x* indicate that the action procedure has no parameters.

- *GUI.ProcessEvent* : **boolean**

  Handles any user interaction. Returns **true** until *GUI.Quit* is called after which it returns **false**.

- *GUI.Quit*

  Causes the next call to *GUI.ProcessEvent* to return false, thus causing the event loop to exit. This procedure should be called when the program is to terminate.

In the *GUI.CreateButton* call, the *width* parameter can be set to 0, in which case the button's width is scaled to fit the width of

the label. In general, if there are several related buttons, they will look best if they are all given the same width. For single buttons, setting *width* to 0 allows the system to determine the appropriate size of the button. The function returns an integer that can be used to identify the widget, commonly called the **widget ID**. The widget ID is passed as a parameter to various subprograms that can show, hide, move. or dispose of the widget.

Note that the action procedure has no parameters. If the program has several buttons pointing to the same action procedure, the only way to determine which button was responsible for calling the action procedure is using the *GUI.GetEventWidgetID* function. The Turing Reference Manual has more details about this and other subprograms.

For the sake of consistency, this program uses the Draw.FillCircle procedure, which is the exact equivalent of the drawfillcircle procedure.

Here is the *GUICircles* program which draws a circle each time the user presses the ÒDraw CircleÓ button.

```
% The "GUICircles" program
import GUI

procedure DrawCircle
    var x, y, r, clr : int
    x := Rand.Int (0, maxx)
    y := Rand.Int (70, maxy)
    r := Rand.Int (20, 50)
    clr := Rand.Int (1, maxcolor)
    Draw.FillOval (x, y, r, r, clr)
end DrawCircle

var button : int := GUI.CreateButton (maxx div 2 – 30, 3, 60,
    "Draw Circle", DrawCircle)
loop
    exit when GUI.ProcessEvent
end loop
```

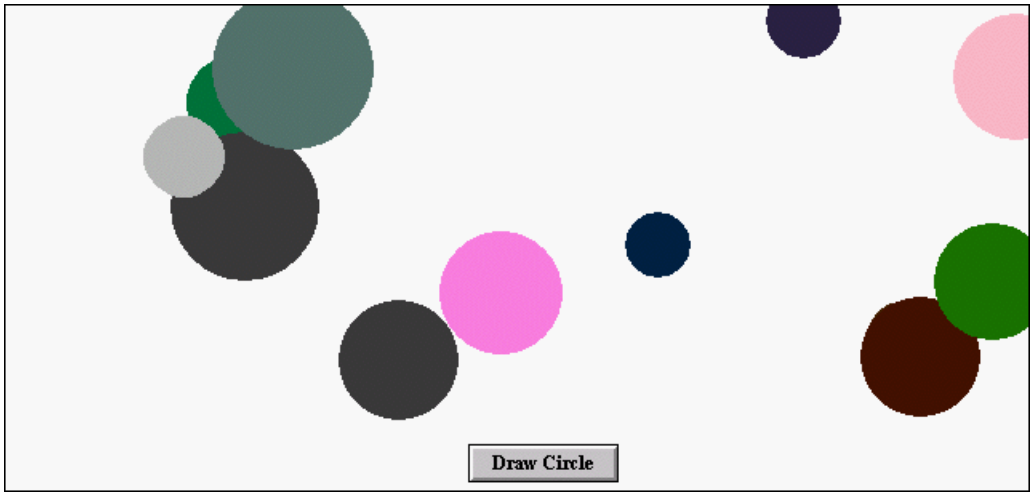


Figure 21.6 The Execution window from *GUICircles*

We will now look at a more complicated *GUI* example that uses widgets. The *Convert* program converts temperatures from Fahrenheit to Celsius and back. The program contains two text fields labeled Fahrenheit and Celsius for entering the temperatures and two buttons for specifying the direction of the conversion. The user can perform conversions by typing in the temperature to be converted into the appropriate text field and then, either pressing Enter or pressing the appropriate button. The result of the conversion appears in the other text field.

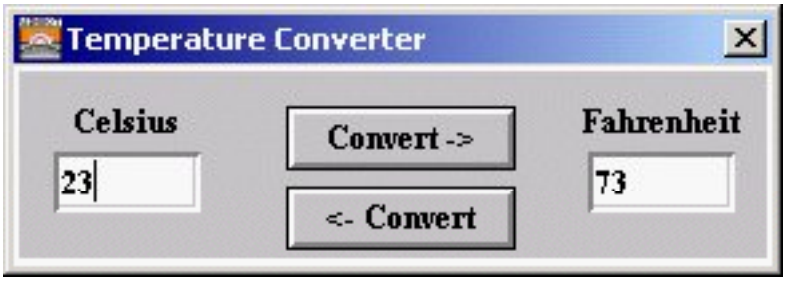


Figure 21.7 The Execution window from *Convert*

The *Convert* program contains six widgets: two buttons, two text fields, and two labels. A label is a string that is generally used to label other widgets. A text field is used for entering a single line of data, in this case, the temperature to be converted.

Each of the buttons and the text fields has its own action procedure. The action procedures for the Fahrenheit to Celsius button and the Fahrenheit text field both convert the contents of the Fahrenheit text field into a Celsius temperature. Unfortunately, an action procedure for a button has no parameters and the action procedure for a text field has a string as a parameter, so we need to create two separate action procedures. The text field's action procedure (*FahrenheitToCelsius1*), however, simply calls the button's action procedure (*FahrenheitToCelsius*).

The actual Fahrenheit to Celsius conversion occurs as follows:

- The contents of the Fahrenheit text field is read into a string using *GUI.GetText*.
- The program checks whether a valid conversion of the string to a number can be done using strintok and which returns **true** if the string can be converted.
- If it can be safely converted, the string is converted into a number.
- The number is then converted into a Celsius temperature using the regular temperature conversion formula.
- Finally, the number is converted back to a string using *strint* and the string is passed as a parameter to *GUI.SetText* which sets the contents of the Celsius text field to the string.

The action procedures and the procedures for conversion from Celsius to Fahrenheit are essentially the same.

There are a few extra aspects of the program. Near the beginning of the program is the line

```
View.Set ("graphics:280;70")
```

This resizes the current window to the specified size. The next statement

```
Window.Set (defWinId, "title:Temperature Converter")
```

changes the title of the output window to ÒTemperature ConverterÓ. Near the end of the program is the statement

```
GUI.SetBackgroundColor ("gray")
```

This procedure sets the background color of the window in which the widgets appear to gray. This is necessary if any of the widgets (such as the text fields) are to have a 3-D appearance. The color gray is the traditional color for dialog boxes.

The two text fields are created using *GUI.CreateTextFieldFull*. The *CreateÉFull* procedures allow for more variables relating to the widget being created to be specified. In this case, we want to specify the type of border around the text field in order to give the text field a 3-D appearance. The default value is for the text field to have a simple rectangular box around it. The *GUI.INDENT* parameter specifies the widget to have an indented border.

To place the labels in the correct position above the text fields, they are created with *GUI.CreateLabelFull*. The *GUI.CENTER* parameter specifies that the label be placed in the center of a box with the same left side and width as the text field. This ensures that the label is centered over the text field.

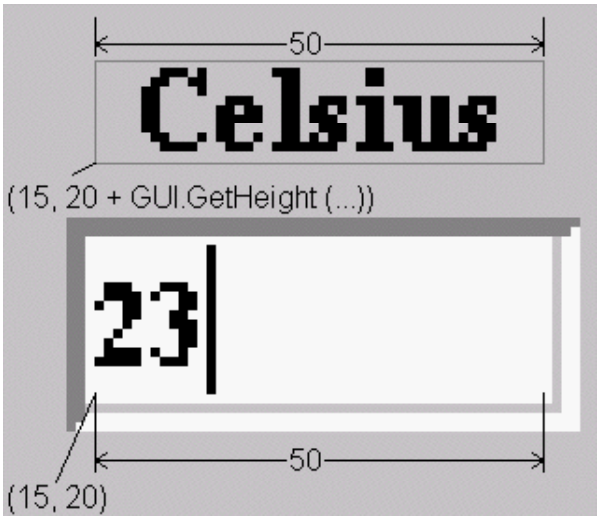


Figure 21.8 Positioning the labels above the text field

Here is the *Convert* program.

```
% The "Convert" program
import GUI

View.Set ("graphics:280;70")
Window.Set (defWinId, "title:Temperature Converter")

var celsiusTextField, fahrenheitTextField : int

procedure CelsiusToFahrenheit
    var celsius : string := GUI.GetText (celsiusTextField)
    if strintok (celsius) then
        var fahrenheit : int := round (strint (celsius) * 9 / 5 + 32)
        GUI.SetText (fahrenheitTextField, intstr (fahrenheit))
    end if
end CelsiusToFahrenheit

procedure CelsiusToFahrenheit1 (dummy : string)
    CelsiusToFahrenheit
end CelsiusToFahrenheit1
```

```
procedure FahrenheitToCelsius
    var fahrenheit : string := GUI.GetText (fahrenheitTextField)
    if strintok (fahrenheit) then
        var celsius : int := round ( (strint (fahrenheit) – 32) * 5 / 9)
        GUI.SetText (celsiusTextField, intstr (celsius))
    end if
end FahrenheitToCelsius

procedure FahrenheitToCelsius1 (dummy : string)
    FahrenheitToCelsius
end FahrenheitToCelsius1

% Create a text field for the celsius reading
celsiusTextField := GUI.CreateTextFieldFull (15, 20, 50, "",
    CelsiusToFahrenheit1, GUI.INDENT, 0, 0)

% Create a label for the celsius field
var d1 : int := GUI.CreateLabelFull (15, 20 +
    GUI.GetHeight (celsiusTextField),
    "Celsius", 50, 0, GUI.CENTER, 0)

% Create a text field for the fahrenheight reading
fahrenheitTextField := GUI.CreateTextFieldFull (215, 20, 50, "",
    FahrenheitToCelsius1, GUI.INDENT, 0, 0)

% Create a label for the fahrenheit field
var d2 : int := GUI.CreateLabelFull (215, 20 +
    GUI.GetHeight (celsiusTextField),
    "Fahrenheit", 50, 0, GUI.CENTER, 0)

% Create the button to convert from celsius to fahrenheit
var toFahrenheit : int := GUI.CreateButton (100, 33, 80, "Convert ->",
    CelsiusToFahrenheit)

% Create the button to convert from fahrenheit to celsius
var toCelsius : int := GUI.CreateButton (100, 3, 80, "<- Convert",
    FahrenheitToCelsius)

GUI.SetBackgroundColour (gray)

loop
    exit when GUI.ProcessEvent
```

```
end loop
```

21.5 Playing Music from Sound Files

Turing has the ability to play music from a sound file. (At the time of writing, this feature only exists under Windows. Check the on-line help for the Macintosh version of Turing to determine if the feature is available.)

To play sound and music on a PC requires a sound card and speakers. The internal speaker of a PC (the speaker that makes the initial beep when the machine is turned on) is not sufficient for playing music.

Turing uses two predefined subprograms for playing music.

```
Music.PlayFile (filename : string)
Music.PlayFileStop
```

The *Music.PlayFile* procedure loads the music file specified and starts playing it. The music file can be one of the following formats:

- a WAV file, in which case the file name ends with Ò.wavÓ,
- a MIDI file, in which case the file name ends with Ò.midÓ,
- a CD, in which case the file name is simply ÒcdÓ, or
- a CD track, in which case the file name is simply Òcd:<number>Ó where <number> is the track number to be played. For example Òcd:3Ó plays the third track on the CD.

Execution halts while playing a music file. This means that if you want music to play simultaneously with animation, you must use a technique that lets multiple parts of the program execute at the same time. This ability is called **concurrency** and in Turing is accomplished using **processes**. Concurrency is a very advanced topic and beyond the scope of this book. We will only provide enough of an introduction to allow the use of processes to play music.

In Turing, a process is like a procedure except a process is called using the **fork** statement. When the **fork** statement is called, the process starts execution, but unlike a procedure, execution also continues after the **fork** statement at the same time as the process is executing. Thus the process and the rest of the program are executing simultaneously. Like procedures, processes can have arguments.

To play a music file at the same time as performing other tasks, place the call to Music.PlayFile in a process and fork the process when the music is to be played.

```
process PlayMusic
    Music.PlayFile ("sonata.mid")
end PlayMusic

…
fork PlayMusic
…
```

To play a piece of music continuously, place the Music.PlayFile statement in a loop. Windows has the ability to play a MIDI file and a WAV file at the same time. This means that a MIDI file can be used for background music and a WAV file used for sound effects.

Playing a second WAV file before the first WAV file has completed terminates the first sound immediately.

In Turing, a program does not terminate while any processes are still executing. This means that to have a program that is playing background music in a loop terminate properly, a method for finishing the process must be added. This can be done by adding a global variable to specify when the program is to terminate. This global variable is checked each time through the loop in the process. When the program is to terminate, the global variable is set and then the Music.PlayFileStop procedure is called. This causes the music to stop and all Music.PlayFile statements to return immediately. Without it, the music might play for several more minutes before finishing and then exiting the loop.

```
var finished : boolean := false
…
```

```
process PlayMusic
    loop
        Music.PlayFile ("sonata.mid")
        exit when finished
    end loop
end PlayMusic

…
fork PlayMusic
…
% Quit playing music
finished := true
Music.PlayFileStop
```

The *MoveLogoWithMusic* program uses the Pic example but modifies it to play background music continuously and produce a sound effect every time the logo hits a wall. As well, the program terminates when a key is pressed.

Note that the sound effect is a WAV file and the background music is a MIDI file.

```
% The "MoveLogoWithMusic" program
% This program animates a graphic around the window

var logoPicID, bgPicID : int    % The picture Ids of the logo and the
bkgrnd
var logoWidth, logoHeight : int % The width and height of the logo
picture
var x, y : int := 0             % The location of the logo
var dx, dy : int := 3           % The direction of movement of the logo
var finished : boolean := false % Set to true when program finished

% Process to continuously play background music
process PlayBackgroundMusic (filename : string)
    loop
        exit when finished
        Music.PlayFile (filename)
    end loop
end PlayBackgroundMusic
```

```
% Process to play a sound file once
process PlaySoundEffect (filename : string)
    Music.PlayFile (filename)
end PlaySoundEffect

% Load the logo from file and get its height and width
logoPicID := Pic.FileNew ("logo.bmp")
logoWidth := Pic.Width (logoPic)
logoHeight := Pic.Height (logoPic)

% Load the forest picture from the file and draw it to the window
Pic.ScreenLoad ("forest.bmp", 0, 0, picCopy)

% Start the background music
fork PlayBackgroundMusic ("canyon.mid")

loop
    exit when hasch
    bgPicID := Pic.New (x, y, x + logoWidth, y + logoHeight)
    Pic.Draw (logoPicID, x, y, picMerge)
    delay (50)
    Pic.Draw (bgPicID, x, y, picCopy)
    Pic.Free (bgPicID)
    if x + dx < 0 or x + dx + logoWidth > maxx then
        fork PlaySoundEffect ("boing.wav")
        dx := – dx
    end if
    if y + dy < 0 or y + dy + logoHeight > maxy then
        fork PlaySoundEffect ("boing.wav")
        dy := – dy
    end if
    x := x + dx
    y := y + dy
end loop

finished := true
Music.PlayFileStop
```

21.6 Exercises

1. Write a program using mousewhere that draws a continuous line made up of a series of line segments. The program should add to the line whenever the mouse button is pressed adding a line segment from the end of the line to the current mouse location. Decide how to handle the first time the mouse button is pressed.
2. Write a program using buttonmoved that allows the user to draw rectangles. The user draws a rectangle by clicking the mouse for one corner and then clicking the mouse a second time to specify the opposite corner. Draw a dot when the first corner is specified and draw the rectangle when the second corner is specified.
3. Using the mouse subprograms, write a program to allow the user to draw filled rectangles in a random color by clicking in the window to specify one corner of the rectangle, dragging the mouse to the desired diagonally opposite corner, and releasing the mouse button. While the mouse is being dragged, draw an outline of the rectangle to indicate its current size, then draw a filled rectangle when the mouse button is released.
4. Modify the *MoveLogo* program so that instead of moving a Turing logo loaded from disk, it moves a small graphic created with Turing *Draw*... commands.
5. Write a program that animates two logos around the screen. Whenever the two logos would overlap, move one of the logos to a random, non-overlapping location.
6. Using sprites, write a program that draws a simple city scene and then moves cars, trucks and airplanes across the scene.
7. Create a ÒGuess my numberÓ program. The program should have five buttons numbered 1 through 4 and Quit. The program should choose a random number from 1 to 4. Each time the user selects a button, the program should display a message telling the user whether the guess was accurate and choose a new number.

8. [Project] Write a program that uses GUI buttons and a text field to simulate a calculator. The calculator should have buttons from each of the 10 digits, a button for plus, minus, multiply, and divide, and a button for equals. It should display the user's entered number in the text field as well as the output from the calculation.

21.7 Technical Terms

mousewhere
buttonmoved
buttonwait
Pic module
picture ID
Pic.FileNew
Pic.New
Pic.ScreenLoad
Pic.Draw
Pic.Free
BMP file
PICT file
picMerge
picCopy
priority
Sprite.New
Sprite.Show
Sprite.SetPriority
Sprite.SetPosition
sprite
buttons
check boxes
scroll bars
text boxes
GUI
Graphical User Interfaces
components
widgets
action procedure
event
event-driven programming
event loop

GUI.CreateButton
GUI.ProcessEvent
GUI.Quit
widget ID
WAV file
MIDI file
Music.PlayFile
Music.PlayFileStop
concurrency
process
fork

Appendices

For more information, consult the *Turing Reference Manual* available from Holt Software. More information can also be found at **http://www.holtsoft.com/turing**.

Appendix A : Simplified Turing Syntax

To describe any language you must know its grammar or **syntax**. The following is a simplified version of the syntax of the Turing programming language. For a full Turing syntax see the textbook *Introduction to Computer Science using the Turing Programming Language* by J.N.P. Hume and R.C. Holt published by Holt Software Inc.

The syntax given here has been simplified in several ways. A number of the advanced features of the Turing language have not been discussed in this book. These have been omitted from this syntax. The syntax description given here is a formal description and many seem somewhat strange to you. But here it is:

- Turing programs are constructed by applying the **production rules** given here.
- Each production rule defines a **syntactic variable** such as a *program* in terms of other syntactic variables and strings of characters (tokens) that will ultimately form part of the Turing program.
- For example, the first production rule listed is

  A *program* is:

  {declarationOrStatementInMainProgram}

  The syntactic variable *program* is defined in terms of *declarationOrStatementInMainProgram* which is another syntactic variable. Around this variable's name are curly braces which mean that this variable can occur zero or more times in a *program*.
- To **produce** a Turing program we begin by looking at the production rule for *declarationOrStatementInMainProgram* and substitute for it.
- When you have a rule which produces a word in boldface type you have produced part of the final program and no more

production rules can be applied to that part. It is a **terminal** token.

- The syntactic variables are called non-terminals and you must continue to replace them by their definition.
- In these syntax rules all the terminals are keywords in the Turing language.
- In a program you must replace the syntactic variable *id*, which stands for identifier, by a word you invent which follows the rules for identifiers listed under ñIdentifiers and Explicit Constantsîgiven at the end of the syntax. Such words are terminal tokens but are not keywords in the language.
- In production rules anything enclosed in square brackets [ ] is optional. Anything in curly braces { } can occur zero or more times. To summarize:

  [item] means that item is optional, and
  {item} means that item can occur zero or more times.

Programs and Declarations

A *program* is:

{*declarationOrStatementInMainProgram*}

A *declarationOrStatementInMainProgram* is one of:

a. *declaration*
b. *statement*
c. *subprogramDeclaration*

A *declaration* is one of the following:

a. *constantDeclaration*
b. *variableDeclaration*
c. *typeDeclaration*

A *constantDeclaration* is one of:

a. **const** *id* := *expn*
b. **const** *id* [: *typeSpec*]:= *initializingValue*

An *initializingValue* is one of:

a. *expn*
b. **init** (*initializingValue* {, *initializingValue*})

A *variableDeclaration* is one of:

a. **var** *id* {, *id* } := *expn*
b. **var** *id* {, *id* } : *typeSpec* [:= *initializingValue*]

Types

A *typeDeclaration* is:

type *id* : *typeSpec*

A typeSpec is one of the following:

a. *standardType*
b. *subrangeType*
c. *arrayType*
d. *recordType*
e. *namedType*

A *standardType* is one of:

a. **int**
b. **real**
c. **boolean**
d. **string** [(*compileTimeExpn*)]

A *subrangeType* is:

compileTimeExpn .. *expn*

An *arrayType* is:

array *indexType* {, *indexType* } **of** *typeSpec*

An *indexType* is one of:

a. *subrangeType*
b. *namedType*

A *recordType* is:

record
    id {, *id* } : *typeSpec*
    {*id* {, *id* } : *typeSpec* }
end record

A namedType is:
id

Subprograms

A *subprogramDeclaration* is:
subprogramHeader
subprogramBody

A subprogramHeader is one of:
a. **procedure** *id* [(*parameterDeclaration*
{, *parameterDeclaration* })]
b. **function** id [(*parameterDeclaration*
{, *parameterDeclaration* })] : *typeSpec*

A parameterDeclaration is:
[**var**] *id* {, *id* } : *parameterType*

A parameterType is one of:
a. *typeSpec*
b. **string** (*)
c. **array** *compileTimeExpn* .. * {, *compileTimeExpn* ..*}
of *typeSpec*
d. **array** compileTimeExpn .. * {, *compileTimeExpn* .. * }
of string (*)

A *subprogramBody* is:
declarationsAndStatements
end *id*

Statements and Input/Output

DeclarationsAndStatements is:
{*declarationOrStatement* }

A *declarationOrStatement* is one of:
a. *declaration*
b. *statement*

A *statement* is one of the following:

a. *variableReference* := *expn*
b. *procedureCall*
c. **assert** *booleanExpn*
d. **result** *expn*
e. *ifStatement*
f. *loopStatement*
g. **exit** [**when** *booleanExpn*]
h. *caseStatement*
i. *forStatement*
j. *putStatement*
k. *getStatement*
l. *openStatement*
m. *closeStatement*

A *procedureCall* is a:

reference

An *ifStatement* is:

if *booleanExpn* **then**
    declarationsAndStatements
{**elsif** *booleanExpn* **then**
    declarationsAndStatements }
[**else**
    declarationsAndStatements]
end if

A *loopStatement* is:

loop
    declarationsAndStatements
end loop

A *caseStatement* is:

case *expn* **of**
    label *compileTimeExpn* {, *compileTimeExpn* } :
            declarationsAndStatements
    {**label** *compileTimeExpn* {, *compileTimeExpn* } :
            declarationsAndStatements}

[**label** : *declarationsAndStatements*]
end case

A *forStatement* is one of:

a. **for** *id* : *expn* .. *expn* [**by** *expn*]
declarationsAndStatements
end for
b. **for decreasing** *id* : *expn* .. *expn* [**by** *expn*]
declarationsAndStatements
end for

The *id* may be omitted but is then not accessible.

A *putStatement* is:

put [: *streamNumber* ,] *putItem* {, *putItem* } [..]

A *putItem* is one of:

a. *expn* [: *widthExpn* [: *fractionWidth* [: *exponentWidth*]]]
b. **skip**

A *getStatement* is:

get [: *streamNumber* ,] *getItem* {, *getItem* }

A *getItem* is one of:

a. *variableReference*
b. **skip**
c. *variableReference* : *
d. *variableReference* : *widthExpn*

An *openStatement* is:

open: *fileNumberVariable*, *fileName*,
capability {, *capability* }

A *capability* is one of:

a. **get**
b. **put**

A *closeStatement* is:

close: *fileNumber*

A streamNumber, *widthExpn*, *fractionWidth*, *exponentWidth* , or

fileNumber is an:
expn

References and Expressions

A *variableReference* is a:
reference

A reference is one of:
a. *id*
b. *reference componentSelector*

A componentSelector is one of:
a. (*expn* {, *expn* })
b. . *id*

A booleanExpn or *compileTimeExpn* is an:
expn

An *expn* is one of the following:
a. *reference*
b. *explicitConstant*
c. *substring*
d. *expn infixOperator expn*
e. *prefixOperator expn*
f. (*expn*)

An *explicitConstant* is one of:
a. *explicitUnsignedIntegerConstant*
b. *explicitUnsignedRealConstant*
c. *explicitStringConstant*
d. **true**
e. **false**

An *infixOperator* is one of:
a. + (integer and real addition; string concatenation)
b. – (integer and real subtraction)
c. * (integer and real multiplication)
d. / (real division)

e. **div** (truncating integer division)
f. **mod** (integer remainder)
g. ** (integer and real exponentation)
h. < (less than)
i. > (greater than)
j. = (equal to)
k. <= (less than or equal to)
l. >= (greater than or equal to)
m. **not=** (not equal)
n. **and** (boolean and)
o. **or** (boolean inclusive or)

A *prefixOperator* is one of:

a. + (integer and real identity)
b. – (integer and real negation)
c. **not** (boolean negation)

All infix operators (including **) associate left-to-right. The precedence of all the operators is as follows, in decreasing order of precedence (tightest binding to loosest binding):

1. **
2. prefix +, –
3. *, /, **div**, **mod**
4. infix +,–
5. <, >, =, <=, >=, **not=**
6. **not**
7. **and**
8. **or**

A *substring* is:

reference (*substringPosition* [.. *substringPosition*])

A *substringPosition* is one of:

a. *expn*
b. * [– *expn*]

Identifiers and Explicit Constants

An *identifier* consists of a sequence of at most 50 letters, digits, and underscores beginning with a letter. All these characters are significant in distinguishing identifiers. Upper and lower case letters are considered to be distinct in identifiers and keywords; hence *j* and *J* are different identifiers. The keywords must be in lower case. Keywords and predefined identifiers must not be redeclared (they are reserved words).

An *explicit string constant* is a sequence of zero or more characters surrounded by double quotes. Within explicit string constants, the back slash character (\) is an escape to represent certain characters as follows: \" for double quote, \n or \N for end of line character, \t or \T for tab, \f or \F for form feed, \r or \R for return, \b or \B for backspace, \e or \E for escape, \d or \D for delete, and \\ for back slash. Explicit string constants must not cross line boundaries.

Character values are ordered by the ASCII collating sequence.

An *explicit integer constant* is a sequence of one or more decimal digits, optionally preceded by a plus or minus sign.

An *explicit real constant* consists of three parts: an optional plus or minus sign, a *significant figures part*, and an *exponent part*. The significant figures part consists of a sequence of one or more digits optionally containing a decimal point. The exponent part consists of the letter *e* (or *E*) followed optionally by a plus or minus sign followed by one or more digits. If the significant figures part contains a decimal point then the exponent part is optional. The following are examples of explicit real constants.

2.0 0. .1 2e4 −56.1e+27

An explicit integer or real constant that begins with a sign is called a *signed* constant; without the sign, it is called an *unsigned* constant.

The explicit boolean constants are **true** and **false**.

Appendix B : Predefined Subprograms

Predefined Functions

eof (*i* : **int**): **boolean**
: Accepts a non-negative stream number (see description of **get** and **put** statements) and returns true if, and only if, there are no more characters in the stream. This function must not be applied to streams that are being written to (via **put**). The parameter and parentheses can be omitted, in which case it is taken to be the default input stream.

length (*s* : **string**): **int**
: Returns the number of characters in the string. The string must be initialized.

index (*s , patt* : **string**): **int**
: If there exists an *i* such that *s(i .. i + length(patt) – 1) = patt,* then the smallest such *i* is returned, otherwise zero is returned. Note that 1 is returned if *patt* is the null string.

repeat (*s* : **string**, *i* : **int**): **string**
: If $i > 0$, returns *i* copies of *s* joined together, else returns the null string. Note that if $j \geq 0$, *length*(*repeat*(*t, j*)) = *j * length*(*t*).

hasch : **boolean**
: Value is true if single character has been read by procedure *getch*.

playdone : **boolean**
: Value is true if the execution of the preceding *play* procedure is finished.

whatcolor : **int**
: Value is the current color number in which characters will be displayed in pixel graphics.

whatcolorback : **int**
: Value is the current background color number in pixel graphics.

maxx : **int**
Maximum value of *x* in current pixel graphics mode. For CGA, *maxx* = 319.

maxy : **int**
Maximum value of *y* in current pixel graphics mode. For CGA, *maxy* = 199.

maxcolor : **int**
Value is the maximum color number in current pixel (or character) graphics mode. For CGA graphics, *maxcolor* = 3. For character graphics, *maxcolor* = 15.

Mathematical Functions

abs (*expn*)
Accepts an integer or real value and returns the absolute value. The type of the result is **int** if the *expn* is of root type **int**; otherwise it is real.

max (*expn , expn*)
Accepts two numeric (real or integer) values and returns their maximum. If both are of root type **int**, the result is an **int**; otherwise it is real.

min (*expn, expn*)
Accepts two numeric (real or integer) values and returns their minimum. If both are of root type **int**, the result is an **int**; otherwise it is real.

sign (*r* : **real**): –1 .. 1
Returns –1 if *r* < 0, 0 if *r* = 0, and 1 if *r* > 0.

sqrt (*r* : **real**): **real**
Returns the positive square root of *r*, where *r* is a non-negative value.

sin (*r* : **real**): **real**
Returns the sine of *r*, where *r* is an angle expressed in radians.

cos (*r* : **real**): **real**

Returns the cosine of *r*, where *r* is an angle expressed in radians.

arctan (*r* : **real**): **real**
Returns the arctangent (in radians) of *r*.

sind (*r* : **real**): **real**
Returns the sine of *r*, where *r* is an angle expressed in degrees.

cosd (*r* : **real**): **real**
Returns the cosine of *r*, where *r* is an angle expressed in degrees.

arctand (*r* : **real**): **real**
Returns the arctangent (in degrees) of *r*.

ln (*r* : **real**): **real**
Returns the natural logarithm (base e) of *r*.

exp (*r* : **real**): **real**
Returns the natural base *e* raised to the power *r*.

Type Transfer Functions

floor (*r* : **real**): **int**
Returns the largest integer less than or equal to *r*.

ceil (*r* : **real**): **int**
Returns the smallest integer greater than or equal to *r*.

round (*r* : **real**): **int**
Returns the nearest integer approximation to *r*. Rounds to larger value in case of tie.

intreal (*i* : **int**): **real**
Returns the real value corresponding to *i*. No precision is lost in the conversion, so *floor*(*intreal*(*j*)) = *ceil*(*intreal*(*j*)) = *j*. To guarantee that these equalities hold, an implementation may limit the range of *i*.

chr (*i* : **int**): **string** (1)
Returns a string of length 1. The *i*-th character of the ASCII sequence is returned, where the first character corresponds to 0, the second to 1, etc. See ASCII code for characters. The

selected character must not be *uninitchar* (a reserved character used to mark uninitialized strings) or *eos* (a reserved character used to mark the end of a string).

ord (*expn*)

Accepts a string of length 1 and returns the position of the character in the ASCII sequence.

intstr (*i, width* : **int**): **string**

Returns a string equivalent to an integer *i*, padded on the left with blanks as necessary to a length of *width*; for example, *intstr* (14,4) = "*bb*14" where *b* represents a blank. The width parameter is optional. If it is omitted, it is taken to be 1. The width parameter must be non-negative. If width is not large enough to represent the value of *i*, the length is automatically increased as needed. The string returned by *intstr* is of the form:

{blank}[−]digit{digits}

The leftmost digit is non-zero, or else there is a single zero digit.

strint (*s* : **string**): **int**

Returns the integer equivalent to string *s*. String *s* must consist of a possibly null sequence of blanks, then an optional plus or minus sign, and finally a sequence of one or more digits. Note that for integer *i*, and for non-negative *w, strint*(*intstr*(*i, w*)) = *i*.

strintok (*s* : **string**): **boolean**

Returns **true** if string *s* can be successfully converted to an integer using *strint*. This function can be used to test user input before calling *strint*, avoiding a run-time error if the user entered non-integer input.

erealstr (*r* : **real**,*width, fractionWidth, exponentWidth* : **int**): **string**

Returns a string (including exponent) approximating *r*, padded on the left with blanks as necessary to a length of *width*; for example, *erealstr*(2.5e1,9,2,2) = "*b*2.50e+01" where *b* represents a blank. The *width* must be non-negative **int** value. If the *width* parameter is not large enough to represent the value of *r*, it is implicitly increased as needed. The

fractionWidth parameter is the non-negative number of fractional digits to be displayed. The displayed value is rounded to the nearest decimal equivalent with this accuracy, with ties rounded to the next larger value. The *exponentWidth* parameter must be non-negative and gives the number of exponent digits to be displayed. If *exponentWidth* is not large enough to represent the exponent, more space is used as needed. The string returned by *erealstr* is of the form:

{blank}[–]digit,{digit}e sign digit {digit}

where ñsignî is a plus or minus sign. The leftmost digit is non-zero, unless all the digits are zeroes.

frealstr (*r* : **real**, *width, fractionWidth*: **int**): **string**

Returns a string approximating *r*, padded on the left with blanks if necessary to a length of *width*. The number of digits of fraction to be displayed is given by *fractionWidth*; for example, *frealstr*(2.5e1,5,1) = "*b*25.0" where *b* represents a blank. The *width* must be non-negative. If the *width* parameter is not large enough to represent the value of *r*, it is implicitly increased as needed. The *fractionWidth* must be non-negative. The displayed value is rounded to the nearest decimal equivalent with this accuracy, with ties rounded to the next larger value. The result string is of the form:

{blank} [–] digit{digit}.{digit}

If the leftmost digit is zero, then it is the only digit to the left of the decimal point.

realstr (*r* : **real**, *width* : **int**): **string**

Returns a string approximating *r*, padded on the left with blanks if necessary to a length of *width*, for example, *realstr*(2.5e1,4) = "*bb*25" where *b* represents blank. The width parameter must be non-negative. If the *width* parameter is not large enough to represent the value of *r*, it is implicitly increased as needed. The displayed value is rounded to the nearest decimal equivalent with this accuracy, with ties rounded to the next larger value. The string *realstr*(*r, width*) is the same as the string *frealstr*(*r, width, defaultfw*) when *r* = 0 or when 1*e* –3 ? *abs*(*r*) < 1*e* 6, otherwise the same as

erealstr(*r, width, defaultfw, defaultew*), with the following exceptions. With *realstr*, trailing fraction zeroes are omitted and if the entire fraction is zero, the decimal point is omitted. (These omissions take place even if the exponent part is output.) If an exponent is output, any plus sign and leading zeroes are omitted. Thus, whole number values are in general displayed as integers. *Defaultfw* is an implementation defined number of fractional digits to be displayed; for most implementations, *defaultfw* will be 6. *Defaultew* is an implementation defined number of exponent digits to be displayed; for most implementations, *defaultew* will be 2.

strreal (*s* : **string**): **real**

Returns a real approximation to string *s*. String *s* must consist of a possibly null sequence of blanks, then an optional plus or minus sign and finally an explicit unsigned real or integer constant.

Predefined Procedures

rand (**var** *r* : **real**)

Sets *r* to the next value of a sequence of pseudo random real numbers that approximates a uniform distribution over the range 0 < *r* < 1.

randint (**var** *i* : *int, low, high* : **int**)

Sets *i* to the next value of a sequence of pseudo random integers that approximate a uniform distribution over the range *low* ? *i* and *i* ? *high*. It is required that *low* ? *high.*

Graphics Procedures

locate (*row, column*: **int**)

Places the cursor at the point whose screen coordinates are (row, column).

cls

Clears the screen and places cursor at point whose screen coordinates are (1, 1). In pixel graphics mode, clears the screen and changes screen to current background color.

color (*colorNumber*: **int**)
Sets color for text to be displayed.

colorback (*colorNumber*: **int**)
Sets color of the background on which text is to be displayed.

getch (**var** *character*: **string**(1))
Reads a single character from the keyboard.

play (*music*: **string**)
Plays notes according to the *music*. See details in music chapter.

setscreen (*s* : **string**)
Changes to mode designated by string *s*. If *s* is "graphics" changes to graphics mode. If *s* is "text" changes to text mode which does not allow graphics.

colorback (*colorNumber* : **int**)
Sets current background color of text displayed. The default background color is white.

drawdot (*x, y, color* : **int**)
Sets a dot (pixel) of *color* at point (*x, y*).

drawline (*x1, y1, x2, y2, color* : **int**)
Draws a line in *color* from (*x1, y1*) to (*x2, y2*).

drawbox (*x1, y1, x2, y2, color* : **int**)
Draws a rectangle in *color* with sides parallel to the axes, bottom left corner at (*x1, y1*), and upper right corner at (*x2, y2*).

drawfillbox (*x1, y1, x2, y2, color* : **int**)
Draws a filled in rectangle in *color* with sides parallel to the axes, bottom left corner at (*x1, y1*), and upper right corner at (*x2, y2*).

drawoval (*xCenter, yCenter, xRadius, yRadius, color* : **int**)
Draws an oval in *color* with center at (*xCenter, yCenter*), horizontal distance to oval *xRadius*, vertical distance *yRadius*.

drawfilloval (*xCenter, yCenter, xRadius, yRadius, color* : **int**)

Draws a filled in oval in *color* with center at (*xCenter, yCenter*), horizontal distance to oval *xRadius*, vertical distance *yRadius*.

drawarc (*xCenter, yCenter, xRadius, yRadius* : **int**,
initialAngle, finalAngle, color : **int**)
Draws a part of an oval whose specifications are given (as in *drawoval*) between two lines from the center that make angles in degrees: *initialAngle* and *finalAngle* , as measured counterclockwise from the three o'clock position as zero.

drawfillarc (*xCenter, yCenter, xRadius, yRadius* : **int**,
initialAngle, finalAngle, color : **int**)
Draws a filled in ñpiece of pieî shaped wedge whose specifications are given (as in *drawoval*) between two lines from the center that make angles in degrees: *initialAngle* and *finalAngle*, as measured counterclockwise from the three o'clock position as zero.

drawfill (*xInside, yInside, fillColor, borderColor* : **int**)
Starting from a point (*xInside, yInside*) fills an area surrounded by *borderColor* with *fillColor* .

locatexy (*x, y* : **int**)
Changes the cursor position in pixel graphics (the cursor is not visible) to be in the nearest character position to point (*x, y*).

delay (*duration* : **int**)
Causes a delay of length *duration* milliseconds. A delay of duration 500 is half a second.

sound (*frequency, duration* : **int**)
Emits a sound of any *frequency* (cycles per second) for *duration* in milliseconds.

Appendix C : Predefined Subprograms by Module

This is the list of predefined modules available as part of Turing. Their names cannot be used as identifiers. This list may change with new releases of Turing.

Modules

Brush*	GUI	Rand
CheckBox*	Input	RGB
Comm*	Keyboard	Sound*
Concurrency	Limits	Sprite
Config	ListBox*	Str
ConfigNo	Math	Stream
Dir	Menu*	Student
Draw	Mouse	Sys
DropBox*	Music	Text
EditBox*	Network*	Time
Errno	Obsolete	Typeconv
Error	PC	Video*
Event	Pen*	View
Exceptions	Pic	Window
File	Print*	
Font	RadioButton*	

* = Module name reserved for future use

Descriptions

This is the list of exported subprograms available for use in each module, along with a short description. Subprograms listed without the module name are exported unqualified and can be called using just the subprogram name. All others must use the module name in the call to the subprogram. For complete details on any subprogram consult the *Turing Reference Manual*.

Concurrency

empty (*variableReference* : **condition**) : **boolean**
Return true if no processes are waiting on the condition queue.

getpriority : **nat**
Return the priority of the current process.

setpriority (*p* : **nat**)
Set the priority of the current process.

simutime : **int**
Return the number of simulated time units that have passed.

Config

Config.Display (*displayCode* : **int**) : **int**
Return information about display attached to computer.

Config.Lang (*langCode* : **int**) : **int**
Return information about the language and implementation limitations.

Config.Machine (*machineCode* : **int**) : **int**
Return information about the computer on which the program is running.

Dir

Getting Directory Listings

Dir.Open (*pathName* : **string**) : **int**
Open a directory stream in order to get a listing of the directory contents.

Dir.GetLong (*stream* : **int**) : **string**
Return the next file name in the directory listing.

Dir.GetLong (*stream* : **int**, **var** *entryName* : **string**, **var** *size*, *attribute*, *fileTime* : **int**)
Get the next file name in the directory listing along with the file size, attributes, and the last modification time of the file.

Dir.Close (*stream* : **int**)
Close the directory stream.

Disk Directory Manipulation

Dir.Create (*pathName* : **string**)
Create a directory.

Dir.Delete (*pathName* : **string**)
Delete a directory.

Execution Directory Manipulation

Dir.Change (*pathName* : **string**)
Change the current execution directory.

Dir.Current : **string**
Return the current execution directory.

Draw

Draw.Arc (*x, y, xRadius, yRadius, initialAngle, finalAngle, clr* : **int**)
Draw an arc on screen centered at (*x*, *y*).

Draw.Box (*x1, y1, x2, y2, clr* : **int**)
Draw a box on screen.

Draw.Cls
Clear the screen.

Draw.Dot (*x, y, clr* : **int**)
Draw a dot on screen at (*x*, *y*).

Draw.Fill (*x, y, fillColor, borderColor* : **int**)
Fill in a figure of color *borderColor*.

Draw (cont...)

Draw.FillArc (*x, y, xRadius, yRadius, initialAngle, finalAngle, clr* : **int**)
Draw a filled pie-shaped wedge on screen centered at (*x*, *y*).

Draw.FillBox (*x1, y1, x2, y2, clr* : **int**)
Draw a filled box on the screen.

Draw.FillMapleLeaf (*x1, y1, x2, y2, clr* : **int**)
Draw a filled maple leaf on the screen.

Draw.FillOval (*x, y, xRadius, yRadius, clr* : **int**)
Draw a filled oval on screen centered at (*x*, *y*).

Draw.FillPolygon (*x, y* : **array** 1 .. * **of int**, *n* : **int**, *clr* : **int**)
Draw a filled polygon on the screen.

Draw.FillStar (*x1, y1, x2, y2, clr* : **int**)
Draw a filled star on the screen.

Draw.Line (*x1, y1, x2, y2, clr* : **int**)
Draw a line on the screen.

Draw.MapleLeaf (*x1, y1, x2, y2, clr* : **int**)
Draw a maple leaf on screen.

Draw.Oval (*x, y, xRadius, yRadius, clr* : **int**)
Draw an oval on screen centered at (*x*, *y*).

Draw.Polygon (*x, y* : **array** 1 .. * **of int**, *n* : **int**, *clr* : **int**)
Draw a polygon on the screen.

Draw.Star (*x1, y1, x2, y2, clr* : **int**)
Draw a star on the screen.

Error

Error.Last : **int**
Return the (integer) error code from the last subprogram call.

Error.LastMsg : **string**
Return the error message from the last subprogram call.

Error.LastStr : **string**
Return the constant name for the error code from the last subprogram call.

Error.Msg (*errorCode* : **int**) : **string**
Return the error message corresponding to the error code.

Error.Str (*errorCode* : **int**) : **string**
Return the constant name corresponding to the error code.

Error.Trip (*errorCode* : **int**)
Abort execution with the specified error code.

File

File.Status (*pathName* : **string**, **var** *size*, *attribute*, *fileTime* : **int**)
Get the size, attributes, and the last modification time of the file.
File.Copy (*srcPathName*, *destPathName* : **string**)
Copy a file to another location.
File.Rename (*srcPathName*, *destName* : **string**)
Rename a file or directory.
File.Delete (*pathName* : **string**)
Delete a file.
File.DiskFree (*pathName* : **string**) : **int**
Return the free disk space in which the file or directory resides.
File.Exists (*pathName* : **string**) : **boolean**
Return true if the file exists.

Font

Font.New (*fontSelectStr* : **string**) : **int**
Return the fontID for a specified font string.
Font.Free (*fontID* : **int**)
Free a font.
Font.Draw (*textStr* : **string**, *x*, *y*, *fontID*, *clr* : **int**)
Draw text at position (*x*, *y*) with a font.
Font.Width (*textStr* : **string**, *fontID* : **int**) : **int**
Return the width of the string when displayed with a font.
Font.Sizes (*fontID* : **int**, **var** *height*, *ascent*, *descent*, *internalLeading* : **int**)
Return information about a font.
Font.Name (*fontID* : **int**) : **string**
Return the name of the font associated with *fontID*.

Font Enumeration

Font.StartName
Prepare to start listing the names of fonts on the system.
Font.GetName : **string**
Return the next font name.
Font.GetStyle (*fontName* : **string**, **var** *bold*, *italic*, *underline* : **boolean**)
Get the available styles for a font.

Font.StartSize (*fontName* : **string**, *fontStyle* : **string**)
Prepare to start listing the sizes of a font and style.
Font.GetSize : **int**
Return the next size of the font.

Input

getch (**var** *ch* : **char**)
Get a single character from the keyboard.
getchar : **char**
Return the next keystroke in the keyboard buffer.
hasch : **boolean**
Return true if there is a keystroke in the keyboard buffer.
Input.Pause
Wait for any keystroke.

Limits

minint : **int**
Return minimum int type value.
maxint : **int**
Return maximum int type value.
minnat : **nat**
Return minimum nat type value.
maxnat : **nat**
Return maximum nat type value.
Limits.DefaultFW : **int**
Return the default fraction width used in printing using ñputî.
Limits.DefaultEW : **int**
Return the default exponent width used in printing using ñputî.
Limits.Radix : **int**
Return the "radix" (usually 2) of real numbers.
Limits.NumDigits : **int**
Return the number of radix digits in a real number.
Limits.MinExp : **int**
Return the smallest (base radix) exponent allowed.
Limits.MaxExp : **int**
Return the largest (base radix) exponent allowed.

Limits.GetExp (*f* : **real**) : **int**
Return the (base radix) exponent of *f*.

Limits.SetExp (*f* : **real**, *e* : **int**) : **real**
Return the value of *f* with the (base radix) exponent replaced.

Limits.Rreb : **real**
Return the relative round-off error bound.

Math

abs (*x* : **int**) : **int**
abs (*x* : **real**) : **real**
Return the absolute value of *x*.

arctan (*x* : **real**) : **real**
Return the arctangent with result in radians.

arctand (*x* : **real**) : **real**
Return the arctangent with result in degrees.

cos (*angle* : **real**) : **real**
Return the cosine of angle in radians.

cosd (*angle* : **real**) : **real**
Return the cosine of angle in degrees.

exp (*r* : **real**) : **real**
Return the exponentiation function e^r.

ln (*r* : **real**) : **real**
Return the natural logarithm function $log_e r$.

max (*x*, *y* : **int**) : **int**
max (*x*, *y* : **nat**) : **nat**
max (*x*, *y* : **real**) : **real**
Return the maximum value of *x* and *y*.

min (*x*, *y* : **int**) : **int**
min (*x*, *y* : **nat**) : **nat**
min (*x*, *y* : **real**) : **real**
Return the minimum value of *x* and *y*.

sign (*r* : **real**) : **–1 .. 1**
Return the sign of the argument.

sin (*angle* : **real**) : **real**
Return the sine of *angle* in radians.

sind (*angle* : **real**) : **real**
Return the sine of *angle* in degrees.

sqrt (*r* : **real**) : **real**
Return the square root of *r*.

Mouse

Mouse.Where (**var** *x*, *y*, *button* : **int**)
Get the current location of the mouse cursor and the status of the mouse buttons.

Mouse.Hide
Hide the mouse cursor.

Mouse.Show
Show the mouse cursor.

Mouse.ButtonMoved (*motion* : **string**) : **boolean**
Return whether the mouse button has been pressed.

Mouse.ButtonWait (*motion* : **string**, **var** *x*, *y*, *buttonnumber*, *buttonupdown* : **int**)
Get information about a mouse button being pressed.

Mouse.ButtonChoose (*choice* : **string**)
Select the mode for the mouse (either single or multiple button).

Music

Music.Play (*music* : **string**)
Play a series of notes.

Music.PlayFile (*pathName* : **string**)
Play music from a file. File must be in an allowable format.

Music.Sound (*frequency*, *duration* : **int**)
Play a specified frequency for a specified duration.

Music.SoundOff
Immediately terminate any sound playing.

Pic

Pic.New (*x1*, *y1*, *x2*, *y2* : **int**) : **int**
Create a picture from a portion of the screen.

Pic.Draw (*picID*, *x*, *y*, *mode* : **int**)
Draw a picture on the screen at location (*x*, *y*).

Pic.Free (*picID* : **int**)
Free up a picture created by **Pic.New** or **Pic.FileNew.**
Pic.FileNew (*pathName* : **string**) : **int**
Read a picture in from a file.

Pic (cont...)

Pic.Save (*picID* : **int**, *pathName* : **string**)
Save a picture to a file for use with **Pic.FileNew** or **Pic.ScreenLoad.**
Pic.ScreenLoad (*fileName* : **string**, *x*, *y*, *mode* : **int**)
Load a picture stored in a file straight to the screen.
Pic.ScreenSave (*x1*, *y1*, *x2*, *y2* : **int**, *pathName* : **string**)
Save a portion of the screen to a file for use with **Pic.FileNew** or **Pic.ScreenLoad.**

Rand

Rand.Real : **real**
Return a random real number from 0.0 to 1.0.
Rand.Int (*low*, *high* : **int**) : **int**
Return a random integer from *low* to *high* inclusive.
Rand.Next (*seq* : 1 .. 10) : **real**
Return a random real number from 0.0 to 1.0 from a sequence.
Rand.Seed (*seed* : **int**, *seq* : 1 .. 10)
Set the random seed in a sequence.
Rand.Set
Set the random seed in the default sequence to the default value.

RGB

RGB.GetColor (*colorNumber* : **int**, **var** *redComp*, *greenComp*, *blueComp* : **real**)
RGB.GetColour (*colorNumber* : **int**, **var** *redComp*, *greenComp*, *blueComp* : **real**)
Get the red, green, and blue values for a color.
RGB.SetColor (*colorNumber* : **int**, *redComp*, *greenComp*, *blueComp* : **real**)
RGB.SetColour (*colorNumber* : **int**, *redComp*, *greenComp*, *blueComp* : **real**)
Set the red, green, and blue values for a color.

RGB.AddColor (*redComp*, *greenComp*, *blueComp* : **real**) : **int**
RGB.AddColour (*redComp*, *greenComp*, *blueComp* : **real**) : **int**
Create a new color with specified red, green, and blue values.

maxcolor : **int**
maxcolour : **int**
Return the maximum color number available.

RGB (cont...)

Color Names

black, **blue**, **green**, **cyan**, **red**, **magenta**, **purple**, **brown**, **white**, **gray**, **grey**, **brightblue**, **brightgreen**, **brightcyan**, **brightred**, **brightmagenta**, **brightpurple**, **yellow**
Names of colors.

brightwhite
Exists only under DOS. Under Windows, X-Windows and Mac, this is **darkgray**.

darkgray, **darkgrey**
Exists only under Windows, X-Windows, and Mac. Under DOS this is **brightwhite**.

colorfg, **colourfg**
Foreground color. Under DOS this is **white**. Under Windows, X-Windows, and Mac this is **black**.

colorbg, **colourbg**
Background color. Under DOS this is **black**. Under Windows, X-Windows, and Mac this is **white**.

Str

index (*s*, *pattern* : **string**) : **int**
Return the position of *pattern* within string *s*.

length (*s* : **string**) : **int**
Return the length of the string.

repeat (*s* : **string**, *i* : **int**) : **string**
Return the string *s* concatenated *i* times.

Stream

eof (*stream* : **int**) : **boolean**
Return **true** if end of file of a stream has been reached.

Stream.Flush (*stream* : **int**)
Flush an output stream.

Stream.FlushAll
Flush all open output streams.

Sys

Sys.GetEnv (*symbol* : **string**) : **string**
Return the environment string.

Sys.GetPid : **int**
Return the process id number of the current task.

Sys.Exec (*command* : **string**) : **int**
Execute a program using the operating system.

Command Line Arguments

Sys.Nargs : int
Return the number of command line arguments.

Sys.FetchArg (*i* : **int**) : **string**
Return the specified command line argument.

Text

maxrow : **int**
Return the number of screen text rows.

maxcol : **int**
Return the number of screen text columns.

Text.Locate (*row, col* : **int**)
Place cursor at character position (*row, col*).

Text.LocateXY (*x, y* : **int**)
Place cursor as close to pixel position (*x, y*) as possible.

Text.WhatRow : **int**
Return the current cursor row.

Text.WhatCol : **int**
Return the current cursor column.

Text.Cls
Clear the screen, setting it to all spaces.

Text.Color (*clr* : **int**)
Text.Colour (*clr* : **int**)
Set text color.

Text.ColorBack (*clr* : **int**)
Text.ColourBack (*clr* : **int**)
Set text background color.

Text (cont...)

Text.WhatColor : **int**
Text.WhatColour : **int**
Return the currently-active text color.

Text.WhatColorBack : **int**
Text.WhatColourBack : **int**
Return the currently-active text background color.

Text.WhatChar (*row*, *col* : **int**, **var** *ch* : **char**, **var** *foreColor*, *backColor* : **int**)
Return the character and text colors at a cursor position.

Time

Time.Sec : **int**
Return number of seconds since 1/1/1970 00:00:00 GMT.

Time.Date : **string**
Return the current date and time as a string.

Time.SecDate (*timeInSecs* : **int**) : **string**
Convert the number of seconds into a date/time string.

Time.DateSec (*dateString* : **string**) : **int**
Convert a date/time string into a number of seconds.

Time.SecParts (*timeInSecs* : **int**, **var** *year*, *month*, *day*, *dayOfWeek, hour*, *min*, *sec* : **int**)
Convert the number of seconds since 1/1/1970 00:00:00 GMT into a year, month, day, day of week, hour, minute, and seconds.

Time.PartsSec (*year*, *month*, *day*, *hour*, *minute*, *second* : **int**) : **int**
Convert the year, month, day, hour, minute, and seconds integers into the number of seconds since 1/1/1970 00:00:00 GMT.

Time.Elapsed : **int**
Return milliseconds since the program started to run.

Time.ElapsedCPU : **int**
Return milliseconds of CPU time since the program started to run.

Time.Delay (*duration* : **int**)
Sleep for a specified number of milliseconds.

Typeconv

From Int

intreal (*i* : **int**) : **real**
Convert an integer to a real.

intstr (*i* : **int**) : **string**
Convert an integer to a string.

From Real

round (*r* : **real**) : **int**
Convert a real to an integer (rounding to closest).

floor (*r* : **real**) : **int**
Convert a real to an integer (rounding down).

ceil (*r* : **real**) : **int**
Convert a real to an integer (rounding up).

realstr (*r* : **real**, *width* : **int**) : **string**
Convert a real to a string.

erealstr (*r* : **real**, *width*, *fractionWidth*, *exponentWidth* : **int**) : **string**
Convert a real to a string (exponential notation).

frealstr (*r* : **real**, *width*, *fractionWidth* : **int**) : **string**
Convert a real to a string (no exponent).

From String

strint (*s* : **string** [, *base* : **int**]) : **int**
Convert a string to an integer.

strintok (*s* : **string** [, *base* : **int**]) : **boolean**
Return whether a string can be converted to an integer.

strnat (*s* : **string** [, *base* : **int**]) : **nat**
Convert a string to a natural number.

strnatok (*s* : **string** [, *base* : **int**]) : **boolean**
Return whether a string can be converted to a natural number.

strreal (*s* : **string**) : **real**
Convert a string to a real.

strrealok (*s* : **string**) : **boolean**
Return whether a string can legally be converted to a real.

Typeconv (cont...)

From Nat

natreal (*n* : **nat**) : **real**
Convert a natural number to a real.

natstr (*n* : **nat**) : **string**
Convert a natural number to a string.

To/From ASCII

chr (*i* : **int**) : **char**
Return a character with the specified ASCII value.

ord (*ch* : **char**) : **int**
Return the ASCII value of a specified character.

View

View.ClipSet (*x1*, *y1*, *x2*, *y2* : **int**)
Set the clipping region to the specified rectangle.

View.ClipAdd (*x1*, *y1*, *x2*, *y2* : **int**)
Add another rectangle to the clipping region.

View.ClipOff
Stop all clipping.

View.Set (*setString* : **string**)
Change the configuration of the screen.

View.WhatDotColor (*x*, *y* : **int**) : **int**
View.WhatDotColour (*x*, *y* : **int**) : **int**
Return the color of the pixel at location (*x*, *y*).

maxx : **int**
Return the maximum x coordinate (width – 1).

maxy : **int**
Return the maximum y coordinate (height – 1).

Window

Window.Open (*setUpString* : **string**) : **int**
Open a new execution window.

Window.Close (*winID* : **int**)
Close an execution window.

Window (cont...)

Window.Select (*winID* : **int**)
Select an execution window for output.

Window.GetSelect : **int**
Return the currently-selected execution window.

Window.SetActive (*winID* : **int**)
Select and activate (make front-most) an execution window.

Window.GetActive : **int**
Return the currently-active execution window.

Window.Hide (*winID* : **int**)
Hide an execution window.

Window.Show (*winID* : **int**)
Show an execution window.

Window.Set (*winID* : **int**, *setUpString* : **string**)
Set the configuration of an execution window.

Window.SetPosition (*winID* : **int**, *x*, *y* : **int**)
Set the current position of an execution window.

Window.GetPosition (*winID* : **int**, **var** *x*, *y* : **int**)
Get the current position of an execution window.

Appendix D : Reserved Words

This is the list of reserved words in Turing. They cannot be used as identifiers.

addressint	**all**	**and**	**anyclass**
array	**assert**	**begin**	**bind**
body	**boolean**	**by**	**case**
char	**checked**	**class**	**close**
collection	**const**	**decreasing**	**deferred**
div	**else**	**elsif**	**end**
enum	**exit**	**export**	**external**
false	**fcn**	**for**	**fork**
forward	**free**	**function**	**get**
if	**implement**	**import**	**in**
include	**inherit**	**init**	**int**
int1	**int2**	**int4**	**invariant**
label	**loop**	**mod**	**module**
monitor	**nat**	**nat1**	**nat2**
nat4	**new**	**nil**	**not**
of	**opaque**	**open**	**or**
pause	**pervasive**	**pointer**	**post**
pre	**proc**	**procedure**	**process**
put	**quit**	**read**	**real**
real4	**real8**	**record**	**register**
rem	**result**	**return**	**seek**
set	**shl**	**shr**	**signal**
skip	**string**	**tag**	**tell**
then	**to**	**true**	**type**
unchecked	**union**	**unit**	**var**
wait	**when**	**write**	**xor**

Appendix E : Operators

This is a list of the operators available in Turing and their precedence.

Mathematical Operators

Operator	Operation	Result Type
Prefix **+**	Identity	Same as Operands
Prefix **–**	Negative	Same as Operands
+	Addition	Same as Operands
–	Subtraction	Same as Operands
*****	Multiplication	Same as Operands
/	Division	Same as Operands
div	Integer Division	**int**
mod	Modulo	**int**
rem	Remainder	**int**
******	Exponentiation	Same as Operands
<	Less Than	**boolean**
>	Greater Than	**boolean**
=	Equals	**boolean**
<=	Less Than or Equal	**boolean**
>=	Greater Than or Equal	**boolean**
not=	Not Equal	**boolean**

Boolean Operators

Operator	Operation	Result Type
Prefix **not**	Negation	**boolean**
and	And	**boolean**
or	Or	**boolean**
=>	Implication	**boolean**

Set Operators

Operator	Operation	Result Type
+	Union	**set**
-	Set Subtraction	**set**
*	Intersection	**set**
=	Equality	**boolean**
not=	Inequality	**boolean**
<=	Subset	**boolean**
<	Strict Subset	**boolean**
>=	Superset	**boolean**
>	Strict Superset	**boolean**

Operators on Members and Sets

Operator	Operation	Result Type
in	Member of Set	**boolean**
not in	Not Member of Set	**boolean**
xor	Exclusive Or	**set**

Bit Manipulation Operators

Operator	Operation	Result Type
shl	Shift left	**nat**
shr	Shift right	**nat**
xor	Exclusive Or	**nat**

Pointer Operators

Operator	Operation	Result Type
^	Follow pointer	Target type

Type Cheats

Operator	Operation	Result Type

#	Type cheat	**nat**

Operator Short Forms

These can be used in place of the above notation.

not	**~**
not=	**~=**
not in	**~in**
and	**&**
or	**|**

Operator Precedence

Highest precedence operators first.

(1)	****, ^, #**
(2)	prefix **+** and **–**
(3)	***** , **/** , **div** , **mod** , **rem** , **shl** , **shr**
(4)	**+** , **–** , **xor**
(5)	**<** , **>** , **=** , **<=** , **>=** , **not=** , **in** , **not in**
(6)	**not**
(7)	**and**
(8)	**or**
(9)	**=>**

Appendix F : File Statements

File Commands

open	open a file
close	close a file
put	write alphanumeric text to a file
get	read alphanumeric text from a file
write	binary write to a file
read	binary read from a file
seek	move to a specified position in a file
tell	report the current file position
eof	check for end of file

File Command Syntax

open : *streamNo*, *fileName*, *ioCapability* **{,** *ioCapability* **}**

ioCapability is one of **get, put, read, write, seek, mod**

put or **write** capability will cause any existing file to be truncated to zero length unless the **mod** capability is also specified.
seek capability is needed to use **seek** or **tell** commands.

close : *streamNo*
get : *streamNo* **,** *getItem* **{ ,** *getItem* **}**
put : *streamNo* **,** *putItem* **{ ,** *putItem* **}**
read : *streamNo* **[:** *fileStatus* **] ,** *readItem* **{ ,** *readItem* **}**
write : *streamNo* **[:** *fileStatus* **] ,** *writeItem* **{,** *writeItem* **}**
seek : *streamNo* **,** *filePosition* or **seek :** *streamNo* **, ***
tell : *streamNo* **,** *filePositionVar*

eof (*streamNo*) **: boolean** (This is a function.)

Appendix G : Control Constructs

FOR

```
for [ decreasing ] variable : startValue .. endValue
                    [ by increment ]
        ... statements ...
        exit when expn
        ... statements ...
end for
```

LOOP

```
loop
        ... statements ...
        exit when expn
        ... statements ...
end loop
```

IF

```
if condition then
        ... statements ...
{ elsif condition then
        ... statements ... }
[ else
        ... statements ... ]
end if
```

CASE

```
case expn of
        { label expn { , expn } :
                ... statements ... }
        [ label :
                ... statements ... ]
end case
```

Any number of **exit** and **exit when** constructs can appear at any place inside **for .. end for** constructs and **loop .. end loop** constructs.

Appendix H : Colors in Turing

Color Number	Color	Color Number	Color
0	White	8	Dark Grey
1	Dark Red	9	Red
2	Dark Green	10	Green
3	Dark Yellow	11	Yellow
4	Dark Blue	12	Blue
5	Dark Magenta	13	Magenta
6	Dark Cyan	14	Cyan
7	Grey	15	Black

Appendix I : Keyboard Codes for Turing

This table gives the ASCII values for characters typed at the keyboard. These values are read by *getch* and *getchar*. To obtain the ASCII value of a character, use the *ord* predefined function.

	0	(space)	32	@	64	`	96
Ctrl-A	1	!	33	A	65	a	97
Ctrl-B	2	"	34	B	66	b	98
	3	#	35	C	67	c	99
Ctrl-D	4	$	36	D	68	d	100
Ctrl-E	5	%	37	E	69	e	101
Ctrl-F	6	&	38	F	70	f	102
Ctrl-G	7	'	39	G	71	g	103
Ctrl-H / BS	8	(	40	H	72	h	104
Ctrl-I / TAB	9	)	41	I	73	i	105
Ctrl-J/Enter	10	*	42	J	74	j	106
Ctrl-K	11	+	43	K	75	k	107
Ctrl-L	12	,	44	L	76	l	108
Ctrl-M	13	-	45	M	77	m	109
Ctrl-N	14	.	46	N	78	n	110
Ctrl-O	15	/	47	O	79	o	111
Ctrl-P	16	0	48	P	80	p	112
Ctrl-Q	17	1	49	Q	81	q	113
Ctrl-R	18	2	50	R	82	r	114
Ctrl-S	19	3	51	S	83	s	115
Ctrl-T	20	4	52	T	84	t	116
Ctrl-U	21	5	53	U	85	u	117
Ctrl-V	22	6	54	V	86	v	118
Ctrl-W	23	7	55	W	87	w	119
Ctrl-X	24	8	56	X	88	x	120
Ctrl-Y	25	9	57	Y	89	y	121
Ctrl-Z	26	:	58	Z	90	z	122
Ctrl-[/ ESC	27	;	59	[	91	{	123
Ctrl-\	28	<	60	\	92	|	124
Ctrl-]	29	=	61	]	93	}	125
Ctrl-^	30	>	62	^	94	~	126

Ctrl-_	31	?	63	_	95	Ctrl-BS	127

Alt-9	128	Alt-D	160	F6	192	Ctrl-F3	224
Alt-0	129	Alt-F	161	F7	193	Ctrl-F4	225
Alt--	130	Alt-G	162	F8	194	Ctrl-F5	226
Alt-=	131	Alt-H	163	F9	195	Ctrl-F6	227
Ctrl-PgUp	132	Alt-J	164	F10	196	Ctrl-F7	228
	133	Alt-K	165		197	Ctrl-F8	229
	134	Alt-L	166		198	Ctrl-F9	230
	135		167	Home	199	Ctrl-F10	231
	136		168	Up Arrow	200	Alt-F1	232
	137		169	PgUp	201	Alt-F2	233
	138		170		202	Alt-F3	234
	139		171	Left Arrow	203	Alt-F4	235
	140	Alt-Z	172		204	Alt-F5	236
	141	Alt-X	173	Right Arrow	205	Alt-F6	237
	142	Alt-C	174		206	Alt-F7	238
Back TAB	143	Alt-V	175	End	207	Alt-F8	239
Alt-Q	144	Alt-B	176	Dn Arrow	208	Alt-F9	240
Alt-W	145	Alt-N	177	PgDn	209	Alt-F10	241
Alt-E	146	Alt-M	178	Ins	210		242
Alt-R	147		179	Del	211	Ctrl-L Arrw	243
Alt-T	148		180	Shift-F1	212	Ctrl-R Arrw	244
Alt-Y	149		181	Shift-F2	213	Ctrl-End	245
Alt-U	150		182	Shift-F3	214	Ctrl-PgDn	246
Alt-I	151		183	Shift-F4	215	Ctrl-Home	247
Alt-O	152		184	Shift-F5	216	Alt-1	248
Alt-P	153		185	Shift-F6	217	Alt-2	249
	154		186	Shift-F7	218	Alt-3	250
	155	F1	187	Shift-F8	219	Alt-4	251
	156	F2	188	Shift-F9	220	Alt-5	252
	157	F3	189	Shift-F10	221	Alt-6	253
Alt-A	158	F4	190	Ctrl-F1	222	Alt-7	254
Alt-S	159	F5	191	Ctrl-F2	223	Alt-8	255

Appendix J : ASCII Character Set

This table shows the ASCII character set. The characters for ASCII codes 0 to 31 and 128 to 255 are dependent on the output font.

32	Space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~

63	?	95	_	127	not shown

Appendix K : Glossary

Abstraction the essence of an idea or object without details.

Active window in a screen display system with several windows, the window that is currently referred to by menu selection, mouse pointing, or keyboard entry.

Actual parameter the value or variable name that appears in parentheses after the name of a procedure or function when it is used in a program.

Algorithm a step-by-step procedure by which certain results may be produced.

Analogy describing one thing by reference to another. The earth is round like an orange is an analogy.

And a logical operator connecting two conditions, each of which must be true if the compound condition is to be true.

Animation achieving the effect of movement in a graphic by displaying a sequence of graphics, one after the other, with small changes from one to the next.

Application program computer software that can accomplish a particular function, such as one to process text or handle business accounting.

Arithmetic operator symbol standing for operation of addition (+), subtraction (–), multiplication (*), or division (/).

Array a group of variables that share a name and are distinguished from each other by each having a particular index value.

Arrow key a key on the keyboard that causes the cursor to move in the direction of the arrow by one character space.

Artificial intelligence the simulation on a computer of activities that are considered to require human intelligence.

ASCII code a binary code chosen as the American Standard Code for Information Interchange.

Assertion a statement in a program that causes an interruption in execution if the condition following the keyword **assert** is false.

Assignment statement a statement that assigns a value to a variable.

Average the value obtained by adding up the values of a number of items and dividing by that number.

Background color the color filling a screen, or window, against which other colored symbols or objects can be displayed.

Backslash the symbol \, used often as an escape character in a computer.

Backspace key the key on the keyboard that causes the cursor to move back one space erasing the symbol it moves back over.

Binary form a way of representing numerical values using only two digits, rather then the decimal form which uses ten digits.

Binary operation a mathematical operation involving two values.

Binary search searching for a particular item in a sorted list of items by successively dividing the list in two equal parts and discarding the half which cannot possibly contain the sought item. The process stops when only one item remains.

Bit a binary digit.

Body of loop the declarations and statements that are contained between the beginning of a loop and the end of the loop.

Boolean variables that can have one of two values, true or false.

Byte a group of binary digits, often eight, that are treated as a unit.

Case construct a program construct used for multi-way selection when the various alternatives are characterized by different integer values.

Case sensitive a language like Turing, that distinguishes between capital letters and little letters.

Cascaded selection a selection construct that contains one or more **elsif**s. It is used in multi-way selection.

CGA graphics one form of graphics system for PCs.

Character a symbol displayed on the screen or stored in the computer memory. It may be a letter, a digit, or a special symbol.

Character graphics obtaining a picture on the screen by the placement of character symbols in various rows and columns.

Clicking mouse pressing the button on the top of the mouse and releasing it quickly.

Color number the integer value that corresponds to a particular color that can be displayed.

Column the spaces across the screen in which symbols can be displayed. Commonly there are 80 columns across a PC screen.

Command an instruction to the operating system of the computer.

Comment a line (or part of a line) of a program that is used to identify it or describe what the program does in order to make it more understandable to someone reading it.

Commutative operation a binary operation which results in the same value if the two operands are interchanged.

Comparison operator an operator used in a condition to compare two values.

Compile-time error a mistake in a program that can be discovered at the time the program is being translated before execution starts.

Compiling translating a program in a high-level language into one that can be understood by a computer.

Computational complexity a measure of the amount of calculation required to execute a particular algorithm.

Conditional loop a repetition that is terminated when a particular condition holds.

Constant a ñvariableî whose value never changes during the execution of a program.

Control-break pressing the control and break keys simultaneously. This is often used to interrupt program execution.

Control structure the sequence in which statements of a program are to be executed.

Coordinates a specification of the position of a point on the screen.

Copyright the ownership of rights and privileges to a particular intellectual property such as a book or computer software.

Correctness of a program a program is correct if it produces the correct output for any possible input. Testing a program on a sample of possible inputs cannot prove correctness.

Counted loop a repetition that is terminated when a counting index reaches a certain value. The index is altered by a fixed amount on each repetition.

Cursor the symbol displayed on the screen that indicates where the next symbol input will be displayed.

Data the information that is processed by a program.

Database a set of related pieces of information that is stored in the computer in a systematic way.

Data file on disk a sequence of data items stored in secondary (disk) memory.

Data type the kind of data represented, for example, numbers (real or integer) or strings of characters.

Debugging looking for errors in a program and correcting them.

Decimal form the usual way of representing numbers in the scale of ten.

Declaration of variable a line in a program that causes memory to be reserved for the variable. The data type of the variable is given in the declaration since different types require different amounts of memory.

Desk top a term used in a window system to show the files that are open or available.

Desk top publishing using the computer (on your own desk) to create camera ready copy for printing.

Dialog box a box which appears, usually in an operating system, and asks you to enter further information.

Digital computer a computer that works on devices that have two states (say on or off) rather than one that represents information continuously as a light meter does.

Directory of files a list of the names of all files on disk that can be accessed at that time.

Disk a circular disk that can hold digital magnetic recordings and is used as a means of storing information on a long term basis.

Disk drive the mechanism in the computer that spins the disk and has read and write heads for ñplayingî and recording.

Disk file name each file on a disk must have a unique name by which it may be retrieved.

Documentation descriptions in natural language and/or mathematics that help make programs understandable to the reader.

Drag an operation with a mouse of pressing the button and holding it down as you move to another screen location.

Echo of input when data is input from a keyboard it is usual for it to be displayed (or echoed) on the screen as it is being typed.

Edit make changes in data or a program.

Editor the systems program that allows you to enter and edit programs or data.

Efficiency of algorithm a measure of the computer time it takes to execute an algorithm. This usually is a function of the amount of data processed.

Encryption of data changing the symbols that represent data to other symbols so as to encode it. Unless the secret of decoding is known noone can understand it.

End-of-file value a special symbol stored automatically at the end of any file of data.

Enter a term also used for the return key of a keyboard.

Error message output from the system to inform the user that an error has occurred. The type and location of the error is given in the message.

Exact divisor an integer that divides a number evenly with no remainder.

Execution the operation of a program after it has been translated.

Expert system an application program that simulates the behavior of an expert in a narrow field in providing answers to questions posed by the user.

Exponent the integer that gives the position of the point (binary or decimal) to accompany another number that gives the actual sequence of significant digits.

Exponential algorithm an algorithm who execution time rises in proportion to a power that is dependent on number of items to be processed, for example, 2^N if there are N items.

Exponentiation raising a number to a power, for example, 2^3.

Field of record one of the items that make up the record structure. Fields need not all be of the same data type.

Field size the number of character positions reserved for outputting an item.

Forgiving program a program that asks the user to try again when an incorrect action is taken rather than stopping execution when an input error occurs.

Formal parameter in the definition of subprograms (either functions or procedures) after the subprogram's name follows, in parentheses, a list of formal parameters each with a colon and a data type after its name. These parameters represent information that is being given to the subprogram for processing or, for procedures, also output that is expected.

Formal relationship an expression, often mathematical, that relates the unknown in a problem to the knowns.

Format the layout of information on input or output.

Function a subprogram that produces a value.

Generalization expressing a relationship in abstract terms that applies in many particular instances.

Global variable a variable declared in the main program that can be used in any subprogram. In general, it is not advisable to make use of this capability inside the subprogram.

Hardware the electronic and mechanical parts of a computer.

Heuristic principle a principle that may be used to make reasonable guesses about something that cannot be easily calculated or proved.

I/O input and output.

I/O window a window that displays input and output during a program's execution.

Icon a small sketch displayed by the computer. Icons may be pointed to by a mouse and dragged from one place to another.

Index of array the integer that identifies a particular element of an array such as a list. Tables require two index values to identify an element. (Sometimes called a subscript.)

Index of counted loop the integer variable that is set to a value upon entry into the loop and changed on each repetition. The loop it terminated when the index reaches a certain value. The index variable must not be declared or altered in the body of the loop.

Initialization setting a starting value of a variable or values of an array.

Initialization in declaration setting the initial value of a variable or an array in the declaration of its data type.

Input instruction an instruction that causes data to be read into the computer.

Insertion sort a method of sorting where each new entry is inserted in its proper position in an ordered sequence.

Join operator the operator used to join strings together (in Turing the +).

Justified output that is brought to line up along one edge of the page or screen.

Knowledge base a systematic grouping of information in machine readable form.

Known information that is given in the specification of a problem.

Linear search a technique for looking for a particular item in an unordered list. Each item is examined in turn until the sought for item is found.

List a sequence of items arranged one after the other.

Local variable a variable that is defined in a subprogram definition and is accessible only to that subprogram. Variables can be local to loop constructs as well.

Logical operator an operator that connects two simple conditions so as to create a compound condition.

Lower case small letters.

Median the value that divides a sorted list of items in two equal halves.

Menu bar the screen display of available headings to the menus of individual possible commands in an operating system.

Memory the part of the computer that stores information.

Merge to blend two or more sorted lists of items into a single sorted list.

Microcomputer a computer that is small in size and usable by a single person at a time.

Mixed number a number that has both an integer and fractional part.

Mouse a device for controlling the movement of a pointer on the display screen and giving input through clicks.

Mouse button a device for giving input to the computer by pressing and releasing it.

Multi-way selection a program construct that selects one of several possible alternatives depending on conditions.

Nested loops one loop body wholly contained inside another loop's body.

Nested selections one selection construct wholly contained inside one or other of the alternative clauses of another selection construct.

Operand the entity that is subject to an operator.

Origin of coordinates the base point from which distances are measured in the x-and y-directions to locate a point.

Output information that is displayed, printed, or stored in secondary memory by a program.

Output item an individual piece of information that is output.

Otherwise clause the alternative in a case construct that is selected if none of the other labels is matched.

Paragraphing a program indenting the body of a loop, selection alternative, or subprogram definition beyond the key line that begins the construct.

Parameter a value passed to a subprogram when the subprogram is called.

Pixel a dot on the display screen.

Pixel position the coordinates on the screen of the point where a pixel is located. Each graphics system limits the number of possible pixel positions on the screen.

Pointer an arrow icon displayed on the screen in response to the positions of the mouse. Also a way of storing the location of a piece of information in the memory.

Polynomial time an algorithm that is executed in a time proportional to a polynomial in the number of items being processed, for example, N^2+2N+1 if there are N items.

Precedence of operations the sequence in which operations are performed in evaluating an expression.

Predefined subprograms subprograms such as *sqrt* that are part of the Turing language.

Prime number a number that has no other exact divisors but 1 and itself.

Procedure a subprogram that performs a task when called.

Program a set of instructions for a computer to perform a particular task or job.

Program window the part of the display screen used to enter the instructions of a program.

Prompt output from a program to get the user to take some action such as entering information from the keyboard.

Pseudo-random numbers a sequence of apparently random numbers that are in fact generated by an algorithm. Each number in the sequence is created from the previous one. The beginning of the sequence is called the seed.

Radix notation a display of digits in which the position of the digits indicates the power of the number base that is to multiply it.

Random numbers see pseudo-random numbers.

Read only memory (ROM) memory used to record information that cannot be changed by the user.

Real number a number with a decimal point (or binary point). Integers are real numbers but not all real numbers are integers.

Record data type a data type that permits a group of items of possibly different types, to be treated as a unit.

Recursive merge sort a method of sorting whereby a list is divided in two and each half sorted by the recursive merge sort and then the two sorted halves merged.

Recursive subprogram a subprogram that calls itself.

Redirecting input or output arranging to obtain standard input from a disk file rather than the keyboard or sending standard output to a disk file rather than to the screen.

Redundant information duplicated information in problem specification, for example, John is 16 and in two years John will be 18.

Related lists two lists which may be of different data types that are in a one-to-one relationship with each other for example one list of person's names and a related one of their phone numbers.

Relational operator the operator connecting two items in a condition, for example, in the condition 6 > 5, the > is a relational operator.

Repetition one of the three basic constructs of all programs. The body of the repetition is to be executed until a condition is met or a counting index reaches a certain value.

Reserved word either a keyword in the Turing language or the name of a predefined subprogram or constant.

Round off to change a mixed number to the integer that is nearest to its value.

Run to start the execution of a program.

Run-time error an error that is not discovered until execution begins, for example, attempting to divide a number by zero.

Saving a file recording information from memory to the disk (secondary memory).

Scroll the movement of the display of characters on the screen either up or down as a unit.

Search space the number of items of information out of which one particular item is sought.

Selecting a command using a mouse to choose an operating system command from a menu of possible commands.

Selecting text choosing a portion of the display in the program window to be deleted or copied. In some systems this is done by dragging a mouse, in others by marking lines by a command.

Selection one of the three basic constructs that make up all programs. Alternative sets of instructions are selected for execution depending on some condition.

Sentinel a mark or symbol used to indicate the end of a file.

Side effect an action taken by a subprogram that is not its purpose. Functions are not permitted to have side effects in Turing.

Significant digits the string of digits that describe a number without its point (binary or decimal) in its proper place.

Simple condition two expressions separated by a relational operator.

Software computer programs.

Sorting placing a list of items in ascending or descending order of a particular key belonging to the items.

Special character any character other than a letter or a digit.

Spreadsheet a display of information often a table used to make projections and financial plans.

Startup disk a disk containing the operating system and compiler necessary to enter and run programs.

Statement the basic elements of a program. See also instruction and declaration.

Step-by-step refinement a technique for developing a program that moves in steps from a statement of the problem specification in English to a program in Turing to solve the problem.

Step size the amount by which the index of a counted loop is altered on each repetition.

String a sequence of characters.

String constant a sequence of characters contained in quotation marks.

Stub an incomplete subprogram, generally one with a missing body, used in program development where other subprograms are being tested first.

Subscript see index of array.

Substring a portion of a string.

Syntax error an error in the grammar, or form, of a statement.

Table a display of values in rows and columns.

Test data sample data, typical of that which will occur in practice, used to find errors in programs.

Testing a program running the program using a range of input data typical of what it is expected to process and comparing the output with values obtained ñ by handî.

Text editor a program that permits you to change text by deletion, addition, and substitution.

Text processing changing text both by editing and formatting the output.

Token a sequence of characters surrounded by white space, that is, blanks or returns.

Token-oriented input using an instruction that will read text a token at a time.

Tracing execution carrying out ñby handî the actions that a computer must go through during the running of a program. The values of all variables must be kept track of as the trace proceeds. Used for finding errors that are not syntax errors.

Translation changing a Turing program into one that can be executed by the computer.

Type definition a statement in the program that gives a name to a particular non-simple data type, for example, a record type.

Type font the style and size of a set of characters, letters and digits used for output.

Unknown the information that is being sought when you solve a problem.

Upper case capital letters.

User-friendly program a program that prompts input, labels output, provides help to the user, and often is forgiving when the user makes a wrong entry.

VGA graphics a commonly used color graphics system for the PC.

Variable a memory location where information can be stored.

Variable parameter a parameter of a procedure whose value will be altered by the procedure.

White space blanks or return.

Window an area on the screen (possibly the whole screen) where certain kinds of information appear.

Word processor application software used to enter text, edit it, format it, save it, and so on.

Index