ALGORITHMS + DATA STRUCTURES = PROGRAMS

NIKLAUS WIRTH

ALGORITHMS + DATA STRUCTURES = PROGRAMS

NIKLAUS WIRTH

Eidgenossische Technische Hochschule
Zurich, Switzerland

PRENTICE-HALL, INC.

ENGLEWOOD CLIFFS, N.J.

Library of Congress Cataloging in Publication Data

WIRTH, NIKLAUS.
Algorithms + data structures = programs.

Bibliography: p.
Includes index.
1. Electronic digital computers--Programming.
2. Data structures (Computer science) 3. Algorithms.
I. Title.
QA76.6.W56 001.6'42 75-11599
ISBN 0-13-022418-9

Current printing (last digit):
19 18 17 16 15

Printed in the United States of America

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA, PTY., LTD., *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA (PTE.) LTD., *Singapore*

PREFACE

In recent years the subje
a discipline whose maste
many engineering projects and which is amenable to scientific treatment and presentation. It has advanced from a craft to an academic discipline. The initial outstanding contributions toward this development were made by E. W. Dijkstra and C. A. R. Hoare. Dijkstra's "Notes on Structured Programming"* opened a new view of programming as a scientific subject and an intellectual challenge, and it coined the title for a "revolution" in programming. Hoare's "Axiomatic Basis of Computer Programming"† showed in a lucid manner that programs are amenable to an exacting analysis based on mathematical reasoning. Both these papers argue convincingly that many programming errors can be prevented by making programmers aware of the methods and techniques which they hitherto applied intuitively and often unconsciously. These papers focused their attention on the aspects of composition and analysis of programs, or, more explicitly, on the structure of algorithms represented by program texts. Yet, it is abundantly clear that a systematic and scientific approach to program construction primarily has a bearing in the case of large, complex programs which involve complicated sets of data. Hence, a methodology of programming is also bound to include all aspects of data structuring. *Programs*, after all, are concrete formulations of abstract *algorithms* based on particular representations and structures of *data*. An outstanding contribution to bring order into the bewildering variety of terminology and concepts on data structures was made by Hoare through his "Notes on Data Structuring."‡ It made clear that decisions

*In *Structured Programming* by Dahl, Djkstra, and Hoare (New York: Academic Press, 1972), pp. 1–82.

†In *Comm. ACM*, **12**, No. 10 (1969), 576–83.

‡In *Structured Programming*, pp. 83–174

about structuring data cannot be made without knowledge of the algorithms applied to the data and that, vice versa, the structure and choice of algorithms often strongly depend on the structure of the underlying data. In short, the subjects of program composition and data structures are inseparably intertwined.

Yet, this book starts with a chapter on data structure for two reasons. First, one has an intuitive feeling that data precede algorithms: you must have some objects before you can perform operations on them. Second, and this is the more immediate reason, this book assumes that the reader is familiar with the basic notions of computer programming. Traditionally and sensibly, however, introductory programming courses concentrate on algorithms operating on relatively simple structures of data. Hence, an introductory chapter on data structures seems appropriate.

Throughout the book, and particularly in Chap. 1, we follow the theory and terminology expounded by Hoare* and realized in the programming language PASCAL.† The essence of this theory is that data in the first instance represent abstractions of real phenomena and are preferably formulated as abstract structures not necessarily realized in common programming languages. In the process of program construction the data representation is gradually refined—in step with the refinement of the algorithm—to comply more and more with the constraints imposed by an available programming system.‡ We therefore postulate a number of basic building principles of data structures, called the *fundamental structures*. It is most important that they are constructs that are known to be quite easily implementable on actual computers, for only in this case can they be considered the true elements of an actual data representation, as the *molecules* emerging from the final step of refinements of the data description. They are the *record*, the *array* (with fixed size), and the *set*. Not surprisingly, these basic building principles correspond to mathematical notions which are fundamental as well.

A cornerstone of this theory of data structures is the distinction between fundamental and "advanced" structures. The former are the molecules—themselves built out of atoms—which are the components of the latter. Variables of a fundamental structure change only their value, but never their structure and never the set of values they can assume. As a consequence, the size of the store they occupy remains constant. "Advanced" structures, however, are characterized by their change of value *and* structure during

*"Notes of Data Structuring."

†N. Wirth, "The Programming Language Pascal," *Acta Informatica*, **1**, No. 1 (1971), 35–63.

‡N. Wirth, "Program Development by Stepwise Refinement," *Comm. ACM*, **14**, No. 4 (1971), 221–27.

the execution of a program. More sophisticated techniques are therefore needed for their implementation.

The sequential file—or simply the sequence—appears as a hybrid in this classification. It certainly varies its length; but that change in structure is of a trivial nature. Since the sequential file plays a truly fundamental role in practically all computer systems, it is included among the fundamental structures in Chap. 1.

The second chapter treats *sorting algorithms.* It displays a variety of different methods, all serving the same purpose. Mathematical analysis of some of these algorithms shows the advantages and disadvantages of the methods, and it makes the programmer aware of the importance of analysis in the choice of good solutions for a given problem. The partitioning into methods for sorting arrays and methods for sorting files (often called internal and external sorting) exhibits the crucial influence of data representation on the choice of applicable algorithms and on their complexity. The space allocated to sorting would not be so large were it not for the fact that sorting constitutes an ideal vehicle for illustrating so many principles of programming and situations occurring in most other applications. It often seems that one could compose an entire programming course by selecting examples from sorting only.

Another topic that is usually omitted in introductory programming courses but one that plays an important role in the conception of many algorithmic solutions is recursion. Therefore, the third chapter is devoted to *recursive algorithms.* Recursion is shown to be a generalization of repetition (iteration), and as such it is an important and powerful concept in programming. In many programming tutorials it is unfortunately exemplified by cases in which simple iteration would suffice. Instead, Chap. 3 concentrates on several examples of problems in which recursion allows for a most natural formulation of a solution, whereas use of iteration would lead to obscure and cumbersome programs. The class of *backtracking* algorithms emerges as an ideal application of recursion, but the most obvious candidates for the use of recursion are algorithms operating on data whose structure is defined recursively. These cases are treated in the last two chapters, for which the third chapter provides a welcome background.

Chapter 4 deals with *dynamic data structures,* i.e., with data that change their structure during the execution of the program. It is shown that the recursive data structures are an important subclass of the dynamic structures commonly used. Although a recursive definition is both natural and possible in these cases, it is usually not used in practice. Instead, the mechanism used in its implementation is made evident to the programmer by forcing him to use explicit reference or *pointer* variables. This book follows this technique and reflects the present state of the art: Chapter 4 is devoted to programming with pointers, to lists, trees, and to examples involving even more complicated meshes of data. It presents what is often (and somewhat inappropriately) called "list processing." A fair amount of space is devoted to tree organizations, and in particular to search trees. The chapter ends with a presentation of scatter tables, also called "hash" codes, which are often preferred to search trees. This provides the possibility of comparing two fundamentally different techniques for a frequently encountered application.

The last chapter consists of a concise introduction to the definition of *formal languages* and the problem of *parsing*, and of the construction of a *compiler* for a small and simple language for a simple computer. The motivation to include this chapter is threefold. First, the successful programmer should have at least some insight into the basic problems and techniques of the compilation process of programming languages. Second, the number of applications which require the definition of a simple input or control language for their convenient operation is steadily growing. Third, formal languages define a recursive structure upon sequences of symbols; their processors are therefore excellent examples of the beneficial application of recursive techniques, which are crucial to obtaining a transparent structure in an area where programs tend to become large or even enormous. The choice of the sample language, called PL/0, was a balancing act between a language that is too trivial to be considered a valid example at all and a language whose compiler would clearly exceed the size of programs that can usefully be included in a book that is not directed only to the compiler specialist.

Programming is a constructive art. How can a constructive, inventive activity be taught? One method is to crystallize elementary composition principles out of many cases and exhibit them in a systematic manner. But programming is a field of vast variety often involving complex intellectual activities. The belief that it could ever be condensed into a sort of pure "recipe teaching" is mistaken. What remains in our arsenal of teaching methods is the careful selection and presentation of master examples. Naturally, we should not believe that every person is capable of gaining equally much from the study of examples. It is the characteristic of this approach that much is left to the student, to his diligence and intuition. This is particularly true of the relatively involved and long examples of programs. Their inclusion in this book is not accidental. Longer programs are the "normal" case in practice, and they are much more suitable for exhibiting that elusive but essential ingredient called style and orderly structure. They are also meant to serve as exercises in the art of program *reading*, which too often is neglected in favor of program writing. This is a primary motivation behind the inclusion of larger programs as examples in their entirety. The reader is led through a gradual development of the program; he is given various "snapshots" in the evolution of a program, whereby this development becomes manifest as a *stepwise refinement* of the details. I consider it essential that programs are shown in final form with sufficient attention to details, for in programming, the devil hides in the details. Although the mere presentation of an algorithm's principle and its mathematical analysis may be stimulating and challenging to the academic mind, it seems dishonest to the engineering practitioner. I have therefore strictly adhered to the rule of presenting the final programs in a language in which they can actually be run on a computer.

Of course, this raises the problem of finding a form which at the same time is both machine executable and sufficiently machine independent to be included in such a text. In this respect, neither widely used languages nor abstract notations proved to be adequate. The language PASCAL provides an appropriate compromise; it had been developed with exactly this aim in mind, and it is therefore used throughout this book. The programs can easily be understood by programmers who are familiar with some other high-level language, such as ALGOL 60 or PL/1, because it is easy to understand the PASCAL notation while proceeding through the text. However, this not to say that some preparation would not be beneficial. The book *Systematic Programming** provides an ideal background because it is also based on the PASCAL notation. The present book was, however, not intended as a manual on the language PASCAL; there exist more appropriate texts for this purpose.†

This book is a condensation—and at the same time an elaboration—of several courses on programming taught at the Federal Institute of Technology (ETH) at Zürich. I owe many ideas and views expressed in this book to discussions with my collaborators at ETH. In particular, I wish to thank Mr. H. Sandmayr for his careful reading of the manuscript, and Miss Heidi Theiler for her care and patience in typing the text. I should also like to mention the stimulating influence provided by meetings of the Working Groups 2.1 and 2.3 of IFIP, and particularly the many memorable arguments I had on these occasions with E. W. Dijkstra and C. A. R. Hoare. Last but not least, ETH generously provided the environment and the computing facilities without which the preparation of this text would have been impossible.

N. WIRTH

*N. Wirth (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1973.)

†K. Jensen and N. Wirth, "PASCAL—User Manual and Report" *Lecture Notes in Computer Science*, Vol. 18 (Berlin, New York; Springer-Verlag, 1974).

CONTENTS

1 FUNDAMENTAL DATA STRUCTURES

1.1. INTRODUCTION

The modern digital computer was invented and intended as a device that should facilitate and speed up complicated and time-consuming computations. In the majority of applications its capability to store and access large amounts of information plays the dominant part and is considered to be its primary characteristic, and its ability to compute i.e., to calculate, to perform arithmetic, has in many cases become almost irrelevant.

In all these cases, the large amount of information that is to be processed in some sense represents an *abstraction* of a part of the real world. The information that is available to the computer consists of a selected set of *data* about the real world, namely, that set which is considered relevant to the problem at hand, that set from which it is believed that the desired results can be derived. The data represent an abstraction of reality in the sense that certain properties and characteristics of the real objects are ignored because they are peripheral and irrelevant to the particular problem. An abstraction is thereby also a simplification of facts.

We may regard a personnel file of an employer as an example. Every employee is represented (abstracted) on this file by a set of data relevant either to the employer or to his accounting procedures. This set may include some identification of the employee, for example, his name and his salary. But it will most probably not include irrelevant data such as the color of hair, weight, and height.

In solving a problem with or without a computer it is necessary to choose an abstraction of reality, i.e., to define a set of data that is to represent the real situation. This choice must be guided by the problem to be solved. Then follows a choice of representation of this information. This choice is guided by the tool that is to solve the problem, i e , by the facilities offered by the computer In most cases these two steps are not entirely independent

The *choice of representation* of data is often a fairly difficult one, and it is not uniquely determined by the facilities available It must always be taken in the light of the operations that are to be performed on the data A good example is the representation of numbers, which are themselves abstractions of properties of objects to be characterized If addition is the only (or at least the dominant) operation to be performed, then a good way to represent the number *n* is to write *n* strokes The addition rule on this representation is indeed very obvious and simple The Roman numerals are based on the same principle of simplicity, and the adding rules are similarly straightforward for small numbers On the other hand the representation by Arabic numerals requires rules that are far from obvious (for small numbers) and they must be memorized However, the situation is inverse when we consider either addition of large numbers or multiplication and division The decomposition of these operations into simpler ones is much easier in the case of representation by Arabic numerals because of its systematic structuring principle that is based on positional weight of the digits.

It is well-known that computers use an internal representation based on binary digits (bits) This representation is unsuitable for human beings because of the usually large number of digits involved, but it is most suitable for electronic circuits because the two values 0 and 1 can be represented conveniently and reliably by the presence or absence of electric currents, electric charge and magnetic fields

From this example we can also see that the question of representation often transcends several levels of detail Given the problem of representing, say, the position of an object, the first decision may lead to the choice of a pair of real numbers in, say, either Cartesian or polar coordinates The second decision may lead to a floating-point representation, where every real number x consists of a pair of integers denoting a fraction *f* and an exponent *e* to a certain base (say, $x = f \cdot 2^e$). The third decision, based on the knowledge that the data are to be stored in a computer, may lead to a binary, positional representation of integers, and the final decision could be to represent binary digits by the direction of the magnetic flux in a magnetic storage device Evidently, the first decision in this chain is mainly influenced by the problem situation, and the later ones are progressively dependent on the tool and its technology Thus, it can hardly be required that a programmer decide on the number representation to be employed or even on the storage device characteristics These "lower-level decisions" can be left to the designers of computer equipment, who have the most information available on current technology with which to make a sensible choice that will be acceptable for all (or almost all) applications where numbers play a role.

In this context, the significance of *programming languages* becomes apparent. A programming language represents an abstract computer capable of understanding the terms used in this language, which may embody a certain level of abstraction from the objects used by the real machine. Thus, the programmer who uses such a "higher-level" language will be freed (and barred) from questions of number representation, if the number is an elementary object in the realm of this language.

The importance of using a language that offers a convenient set of basic abstractions common to most problems of data processing lies mainly in the area of reliability of the resulting programs. It is easier to design a program based on reasoning with familiar notions of numbers, sets, sequences, and repetitions than on bits, "words," and jumps. Of course, an actual computer will represent all data, whether numbers, sets, or sequences, as a large mass of bits. But this is irrelevant to the programmer as long as he does not have to worry about the details of representation of his chosen abstractions and as long as he can rest assured that the corresponding representation chosen by the computer (or compiler) is reasonable for his purposes.

The closer the abstractions are to a given computer, the easier it is to make a representation choice for the engineer or implementor of the language, and the higher is the probability that a single choice will be suitable for all (or almost all) conceivable applications. This fact sets definite limits on the degree of abstractions from a given real computer. For example, it would not make sense to include geometric objects as basic data items in a general-purpose language, since their proper representation will, because of its inherent complexity, be largely dependent on the operations to be applied to these objects. The nature and frequency of these operations will, however, not be known to the designer of a general-purpose language and its compiler, and any choice he makes may be inappropriate for some potential applications.

In this book these deliberations determine the choice of notation for the description of algorithms and their data. Clearly, we wish to use familiar notions of mathematics, such as numbers, sets, sequences, and so on, rather than computer-dependent entities such as bitstrings. But equally clearly we wish to use a notation for which efficient compilers are known to *exist*. It is equally unwise to use a closely machine-oriented and machine-dependent language, as it is unhelpful to describe computer programs in an abstract notation which leaves problems of representation widely open.

The programming language PASCAL has been designed in an attempt to find a compromise between these extremes, and it is used throughout this book [1.3 and 1.5]. This language has been successfully implemented on several computers, and it has been shown that the notation is sufficiently close to real machines that the chosen features and their representations can be clearly explained. The language is also sufficiently close to other languages, particularly ALGOL 60, that the lessons taught here may equally well be applied in their use.

1.2. THE CONCEPT OF DATA TYPE

In mathematics it is customary to classify variables according to certain important characteristics. Clear distinctions are made between real, complex, and logical variables or between variables representing individual values, or sets of values, or sets of sets, or between functions, functionals, sets of functions, and so on. This notion of classification is equally important, if not more important, in data processing. We will adhere to the principle that *every constant, variable, expression, or function is of a certain type*. This type essentially characterizes the set of values to which a constant belongs, or which can be assumed by a variable or expression, or which can be generated by a function.

In mathematical texts the type of a variable is usually deducible from the typeface without consideration of context; this is not feasible in computer programs. For there is usually one typeface commonly available on computer equipment (i.e., Latin letters). The rule is therefore widely accepted that the associated type is made explicit in a *declaration* of the constant, variable, or function, and that this declaration textually precedes the application of that constant, variable, or function. This rule is particularly sensible if one considers the fact that a compiler has to make a choice of representation of the object within the store of a computer. Evidently, the capacity of storage allocated to a variable will have to be chosen according to the size of the range of values that the variable may assume. If this information is known to a compiler, so-called dynamic storage allocation can be avoided. This is very often the key to an efficient realization of an algorithm.

The primary characteristics of the concept of type that is used throughout this text, and that is embodied in the programming language PASCAL, thus are the following [1.2]:

1. A data type determines the set of values to which a constant belongs, or which may be assumed by a variable or an expression, or which may be generated by an operator or a function.
2. The type of a value denoted by a constant, variable, or expression may be derived from its form or its declaration without the necessity of executing the computational process.
3. Each operator or function expects arguments of a fixed type and yields a result of a fixed type. If an operator admits arguments of several types (e.g., + is used for addition of both integers and real numbers), then the type of the result can be determined from specific language rules.

As a consequence, a compiler may use this information on types to check the compatibility and legality of various constructs. For example, the assignment of a Boolean (logical) value to an arithmetic (real) variable may be detected without executing the program. This kind of redundancy in the program text is extremely useful as an aid in the development of programs, and it must be considered as the primary advantage of good high-level languages over machine code (or symbolic assembly code). Evidently, the data will ultimately be represented by a large number of binary digits, irrespective of whether or not the program had initially been conceived in a high-level language using the concept of type or in a typeless assembly code. To the computer, the store is a homogeneous mass of bits without apparent structure. But it is exactly this abstract structure which alone is enabling human programmers to recognize meaning in the monotonous landscape of a computer store.

The theory presented in this book and the programming language PASCAL specify certain methods of defining data types. In most cases new data types are defined in terms of previously defined data types. Values of such a type are usually conglomerates of *component values* of the previously defined *constituent types*, and they are said to be *structured.* If there is only one constituent type, that is, if all components are of the same constituent type, then it is known as the *base type*.

The number of distinct values belonging to a type T is called the *cardinality* of T. The cardinality provides a measure for the amount of storage needed to represent a variable x of the type T, denoted by $x : T$.

Since constituent types may again be structured, entire hierarchies of structures may be built up, but, obviously, the ultimate components of a structure must be atomic. Therefore, it is necessary that a notation is provided to introduce such primitive, unstructured types as well. A straightforward method is that of *enumeration* of the values that are to constitute the type. For example, in a program concerned with plane geometric figures, there may be introduced a primitive type called *shape*, whose values may be denoted by the identifiers *rectangle*, *square*, *ellipse*, *circle*. But apart from such programmer defined types, there will have to be some *standard types* that are said to be predefined. They will usually include *numbers* and *logical values*. If an ordering exists among the individual values, then the type is said to be ordered or *scalar*. In PASCAL, all unstructured types are assumed to be ordered; in the case of explicit enumeration, the values are assumed to be ordered by their enumeration sequence.

With this tool in hand, it is possible to define primitive types and to build conglomerates, structured types up to an arbitrary degree of nesting. In practice, it is not sufficient to have only one general method of combining constituent types into a structure. With due regard to practical problems of representation and use, a general-purpose programming language must offer several *methods of structuring*. In a mathematical sense, they may all be equivalent; they differ in the operators available to construct their values and to select components of these values. The basic structuring methods presented here are the *array*, the *record*, the *set*, and the *sequence* (*file*). More complicated structures are not usually defined as "static" types, but are instead "dynamically" generated during the execution of the program during which they may vary in size and shape. Such structures are the subject of Chap. 4 and include lists, rings, trees, and general finite graphs.

Variables and data types are introduced in a program in order to be used for computation. To this end, a set of *operators* must be available. As with data types, programming languages offer a certain number of primitive, standard (atomic) operators, and a number of structuring methods by which composite operations can be defined in terms of the primitive operators. The task of composition of operations is often considered the heart of the art of programming. However, it will become evident that the appropriate composition of data is equally fundamental and essential.

The most important basic operators are *comparison* and *assignment*, i.e., the test for equality (and order in the case of ordered types) and the command to enforce equality. The fundamental difference between these two operations is emphasized by the clear distinction in their denotation throughout this text (although it is unfortunately obscured in such widely used programming languages as Fortran and PL/I, which use the equal sign as assignment operator).

Test for equality: $x = y$

Assignment to x: $x := y$

These fundamental operators are defined for most data types, but it should be noted that their execution may involve a substantial amount of computational effort if the data are large and highly structured.

Apart from test of equality (or order) and assignment, a class of fundamental and implicitly defined operators are the so-called *transfer operators*. They are mapping data types onto other data types. They are particularly important in connection with structured types. Structured values are generated from their component values by so-called *constructors*, and the component values are extracted by so-called *selectors*. Constructors and selectors are thus transfer operators mapping constituent types into structured types and vice versa. Every structuring method owns its particular pair of constructors and selectors that clearly differ in their denotation.

Standard primitive data types also require a set of standard primitive operators. Thus, along with the standard data types of numbers and logical values, we also introduce the conventional operations of arithmetic and propositional logic.

1.3. PRIMITIVE DATA TYPES

In many programs integers are used when numerical properties are not involved and when the integer represents a choice from a small number of