Algorithms

5. Data Types and their Representation in Memory

R K Shyamasundar

R K Shyamasundar is Professor of Computer Science at TIFR, Mumbai and has done extensive research in various foundation areas of computer science.

In this article, we show how the general abstraction of a program as being composed of data and algorithms aids in the understanding of the universality of computers and the development of programs. We describe a simple organization of the memory unit of a computer,and discuss number representation. Subsequently, we show the need and use of *types* in programming. Further, we describe simple data types and illustrate the use of simple data-structures in the development of programs.

Introduction

In the first article of this series, we had compared a *recipe* to an algorithm and the ingredients for the recipe to the input. On similar terms, in the context of programming the role of ingredients is played by the input or the *data*. The algorithm (which is a sequence of instructions) operates on the input and yields an output. We could treat a program as composed of two parts: algorithm and data. Such an abstraction makes it possible to equate the definitions of a computer and a program; in other words,a computer can be treated as consisting of an algorithmic part and a data part. The data part is nothing but the registers and the memory words; these registers or the memory words can store binary values. The algorithm part is the finite set of algorithms built into the computer corresponding to the logic-circuit rules that are used to interpret the stored data as instructions and fetch data. At the most fundamental level the computer works on only one kind of data, i.e., binary bits. The built-in algorithms are fixed and computers can only execute as per these instructions and nothing else.

At the most fundamental level the computer works on only one kind of data, i.e., binary bits.

Programmers rarely write programs in terms of bits. It is more often the case that people would like to use data in terms of the objects they are imagining such as integers, real numbers, character strings, list of items, matrices etc. Further, the algorithms written/contemplated are more varied than the fixed set of algorithms mentioned above. The question that arises is: How can a spectrum of problems be solved by a single machine that executes according to the fixed set of algorithms built into it? The answer is that a computer is a truly general purpose device;its behaviour gets transformed to the behaviour of the program given to it. In other words, the behaviour of the programs are mimicked. The underlying principle lies in the *Stored Program Concept* set forth by John von Neumann. In this model, a stream of information to the computer is interpreted as data at one instant and as an instruction (program) at another instant. It is easy to visualize a simple program that repeatedly takes a given program and its data and executes as per the given

A computer is a truly general purpose device;its behaviour gets transformed to the behaviour of the program given to it.

Stored Program Concept

The stored program concept can be seen in the memo written by J von Neumann proposing a stored-program computer called EDVAC (Electronic Discrete Variable Automatic Computer).

The intention of the proposal was to build a machine that was fully automatic in character, independent of the human intervention once the computation starts. For this purpose, it must be quite evident that the machine must be capable of storing in some manner not only the digital information of the input and intermediate values but also the instructions which govern the actual routine to be performed on the bit stream data. In a special purpose machine these instructions form an integral part of the device and constitute a part of its design structure. However, for a general purpose machine it must be possible to instruct the device to carry out any computation that can be formulated on bit streams (or numerical terms). Hence, the program instructions must be stored in some part of the machine and there should be a part in the machine that can interpret these instructions and execute them. In other words, a general-purpose machine has algorithms that take data and program (as data) and execute the instructions as per the program using the given data and storing the intermediate results in its store and indicating completion after the completion of the program.

The general basis of the universality of computers lies in the notion of Universal Turing Machines proposed by A M Turing.

program on the data given; as soon as the program terminates, it is ready to interpret the next data (which is again a program and its data). The general basis of the universality of computers lies in the notion of *Universal Turing Machines* proposed by A M Turing and will be discussed in forthcoming articles.

In general, a program is composed of objects familiar and convenient to the user. This process enables the design and construction of complex programs. A programmer arrives at a solution starting from a skeletal solution (which may not be suitable for execution on a computer) to an implementation. In other words, a programmer goes through a series of abstraction layers before arriving at the final program. In each layer, the programmer attends to the details of some part ignoring the presence of others. This process of abstraction is essential for the construction and understanding of the programs. The main purpose of abstraction is to concentrate on similarities and de-emphasize the differences. By doing so, one would be concentrating on the relevant features of the problem at hand, ignoring the irrelevant features of the problem. For instance, when trying to arrive at a payroll program the financial aspects of an employee are more important rather than his/her physical or metabolic features. In a similar way, while analyzing the engine-performance of an automobile, its colour and aesthetic features are irrelevant. It can also be seen that abstractions need not be unique. Depending on the representation/manipulation of data one could arrive at different abstractions. Abstraction is the key to gaining intellectual mastery over any complex system. Needless to say it requires great skill and experience to use abstraction effectively. Abstract notations at higher levels require translators that take source programs (in high-level languages/notations the users would have written)and translate them into programs in terms of bits that can be executed by the computer. Program abstractions can be broadly divided into control structure abstractions, procedural abstractions and data abstractions.In earlier articles of this series,we have had a look at control structure abstractions and procedural abstractions. In this article, we will consider data abstractions.

A variable is not a data item; it is rather a name.

Types

In the previous sections, we have used inputs, outputs and other intermediate computation objects, in a rather intuitive way without worrying about their structure. For instance, one could think of the procedure for summing *N* numbers as corresponding to summing the salaries of *N* employees. These items (numbers in this example) are generally referred to by the generic term *data*. Some of the questions that we need to look at are:How do we refer to the data? Is there a need for *type* information along with the data? How do we structure data? These aspects are discussed in the sequel.

Constants and Variables

In programming languages, facility of referring to an item through a name is very common. Some of these named objects will have the same value throughout the whole program execution. These items are referred to as *constants*. Some named objects take on new values as the computation progresses. These items are referred to as *variables*. For instance, in the example of 'sum-of-N-numbers', we used *sum* to accumulate the partial and final results. It must be emphasized that a *variable* is not a data item; it is rather a *name*. In short, the name of a constant or a variable is only a mnemonic aid to the programmer; it has no meaning to the computer. The identifiers are like names to locations — such as a number given to a post box in a post-office. However, there is one difference: there can possibly be more than one name for the same location (similar to *nicknames*). That is, *aliasing* is a possibility; there cannot be a hotel room with two room numbers but two *names* can refer to the same location. The translator of the program that maps the text into a bit stream merely associates with each identifier a unique *location* in the memory of the computer;the *location* is usually referred to as the *address* of the variable. Thus, when an instruction uses a constant or a variable named through an identifier say *sum*, the computer fetches whatever value is in the location corresponding to *sum*. Various operations can be performed on *expressions* formed from constants and variables. One could say " set x to $x + 1$ ", " set x to $x + y$" and so on. One can also test whether "$x = 100$ " or "$x > 100$ " holds.In other words, naming constants and variables are similar to the use of symbolic variables and expressions in algebra.However, one has to note an important difference. In the algebraic expression "$a + b$ ", it is implicitly understood that we can add numbers denoted by a and b. However, in the context of computers where *unambiguous representation is a must*, we need to use additional information (i.e., *type*) of the variables or constants used. For example, we can talk of the predecessor of a natural number but not of a real number. Similarly, if one is keeping track of the name of an employee, we know that there is no need to do any arithmetic operations on the names of the employees. In traditional algebra, we understand the type of the variable from the context implicitly. In the context of computers, it is essential to keep track of the type of the data *explicitly* so that the data can be interpreted unambiguously. This aspect will become clear if we look at the basic organization of memory and the internal representation of basic types such as integers,reals, and characters.

In the context of computers, it is essential to keep track of the type of the data explicitly so that the data can be interpreted unambiguously.

Memory Organization and Internal Representation of Data

Let us consider a simple organization of the main memory unit of a computer. A simple organization of computer memory has the following features:

- The memory unit is subdivided into sub units each of which can be individually accessed. The subunits are grouped into larger units each of which can store the same amount of information (some fixed number of bits). The organization is referred to as digit-or character-organized depending upon whether such a unit has the capacity to hold a digit or a character respectively. In the current day architectures,such an unit is usually referred to as *byte* and consists of eight bits. For the purpose of carrying out arithmetic or logical operations the memory is organized in terms of a fixed number of bytes (2, 4 etc.) referred to as *words*. Such an organization is usually referred to as word organized.

- Each of these subunits/bytes/words has an associated number called the *address* (or location) that can be used for accessing or locating it.

Let us look at the internal representation of some of the basic data items assuming the memory of a computer to be word organized where each word is comprised of four bytes and each byte consists of 8 bits. We will confine ourselves to this simple organization while discussing the internal representation unless otherwise stated. In this context, let us see how common data types such as numbers, characters and real numbers are represented.

Memory is organized in terms of a fixed number of bytes (2, 4 etc.) referred to as words.

Representation of Numbers

Everyone is familiar with radix notations; in practice we use radix 10 and in the binary system of representation the radix used is 2.For example, an integer 75 (without any sign) can be represented in 32 bits by (the space between bytes has been used only for ease of reading):

$$75 \equiv 00000000\ 00000000\ 00000000\ 01001011$$

For representing integers, we should have the ability to represent positive as well as negative numbers. There are various methods of representing negative numbers; the choice influences the way the arithmetic operations are carried out. The simplest one is to represent the magnitude and the sign explicitly. For example,

$$-75 \equiv -00000000\ 00000000\ 00000000\ 01001011$$

The major difference between signed magnitude and 2's complement notation in practice is that shifting right does not divide the magnitude by 2.

This is called the *signed-magnitude* representation. Naturally, the representation is appealing as it coincides with the classical notational conventions. A potential disadvantage is that minus zero and plus zero can both be represented while they should usually mean the same number; this requires care particularly when one is computing through mechanical or automatic machines.

One of the commonly used notations for representing negative numbers on the electronic computers is the *two's complement notation*. For instance, in the context of 32 bits as above, if we subtract 1 from 00000000 00000000 00000000 00000000 we get 11111111 11111111 11111111 11111111 which denotes –1 without an additional sign and computations are done modulo 2^{32}. Thus, –75 in this notation would be represented as

$$-75 \equiv 11111111\ \ 11111111\ \ 11111111\ \ 10110101$$

The major difference between signed magnitude and 2's complement notation in practice is that shifting right does not divide the magnitude by 2. A disadvantage of 2's complement notation is that it is not symmetric about zero;the largest negative number representable in p digits is not the negative of any p – digit positive number. Another notation that is used is the one's complement notation. The one's complement of a number is the result of changing each zero to one and one to zero. The two's complement can be defined as the one's complement plus a one in the least significant position. In fact, the two's complement and one's complement correspond to the ten's complement and the nine's complement used in decimal notation respectively.Computers use these notations for reasons connected with circuit economy and efficiency.

Real Numbers

Real numbers are approximated as decimal fractions. Numbers are represented in two forms: fixed-point and floating-point representations. A fixed-point number in a computer is one for which the computer always assumes the binary or the decimal point to be at the same place for numbers. This place is either at the left-hand end of the number (all numbers considered to be less than 1 in magnitude) or at the right-hand end (all numbers are treated as integers). For understanding the problems of this representation, consider the addition of two decimal numbers, -65.31 and 78.42. Now, if we assume that the decimal points are assumed at the right-hand end, then adding the two numbers, we get 1311;the decimal point can be placed without any problem and we get the result 13.11 as anticipated. Now consider the addition of -65.31 and 784.2 . Again,assuming the decimal point to be at the right-hand end,we get the same answer (i.e., 1311) as in the previous case which is clearly wrong. To overcome this problem, it can be easily seen that one number has to be shifted relative to the other so that the decimal points are lined up one above the other (which we do in the paper-pencil method);this process is referred to as *scaling*. Scaling is somewhat tedious in lengthy computations. Another problem that arises in this representation is the difficulty in handling numbers that differ very widely. For instance,in a 36-bit word machine (sign and 35 bits), if one wants to add 10^8 and 10^{-4}, it is not possible to do it in fixed-point arithmetic because no-matter where the imaginary scaled binary point is, the two numbers can never differ by more than 2^{35}. As a result of these scaling and magnitude problems, a new representation called the *floating-point* representation was developed.

A floating-point number is a sequence of bits which is interpreted to have two distinct parts, one called the exponent, E and the other called the fractional part (often referred to as the mantissa), M.

A floating-point number is a sequence of bits which is interpreted to have two distinct parts, one called the *exponent*, E and the other called the fractional part (often referred to as the mantissa), M. The number is interpreted as $M \times 2^E$ in the binary internal representation; in the decimal representation the number is interpreted as $M \times 10^E$. For example, a 10-digit representation of 86.3 is

02 86300000

which is interpreted to have $E = 2$ and $M = .863$, i.e., 0.863

$\times 10^2$ where it is assumed that the decimal point for the fractional part is at the left-end of the mantissa. It can be easily seen that $E = 3$ and $M = .0863$ is also an equally correct representation. For this reason, it is required that the first digit (or the bit) to the right of the binary or the decimal point is nonzero unless the mantissa is itself zero. Since we have two parts of the numbers, we should have the ability to place signs to both the parts. In the floating-point representation one bit is used for representing the sign of the mantissa while for representing the sign of the exponent a different technique is used. Assuming 8-bits for exponent, the range of numbers that can be represented in 8 bits is from 0 to $2^8 - 1$. Thus,arbitrarily one assigns 10000000 to correspond to the zero exponent. Then the range of exponents is from -128(denoted by 00000000) to $+127$ (denoted by 11111111). This notation is referred to as the *excess* -128 notation. For instance, an exponent part of 10000110 denotes an exponent of 6 (i.e., $134 - 128$).

Characters and Digits:

It easily follows that by a proper encoding (or mapping) one can store digits/characters if the memory devices used are all binary devices. Characters are generally given seven- or eight-bit values specified by the American Standard Code for Information Exchange (ASCII) code. For instance,some of the typical codes for some digits and characters are shown below:

ASCII Code	Digit/Character	Base10 Representation
00100000	<space>	32
00110011	3	51
00111000	8	56
01000101	E	69
01010011	S	83
01010100	T	84

IEEE Floating Point Standard

We need numbers of various types such as integers, rationals, reals, irrationals etc. There is a variety of ways of representing nonintegers. One such method is the fixed point method described in the article. In fixed point representation,one essentially uses integer arithmetic operators assuming the binary point to be at some point other than the rightmost point. Though there are several possible representations, floating point representation is widely used. One of the difficulties with floating point representation is that the meanings of the operations are not as simple as that of integer operations. Further, the operational understanding and implementation depends on the number of bits for the mantissa and exponent (e.g., typical notions such as round-off, overflow, underflow etc.).

The main purpose of standardization is to aid *software designers* to develop efficient and reliable software, *computer designers* to develop techniques for implementing efficient computation of the operations on hardware and *computer manufacturers* to arrive at accelerators for floating point evaluations. Towards providing a unified account of the floating point representation, a standard format has been specified by IEEE standard 754-1985. The standard has gained wide acceptability from software designers to computer manufacturers. The merit of using a variant of the standard floating point representation is similar in spirit to that of using floating point representation over other methods of representation.

An important feature of the above standard is that *computation* continues despite *exceptional* situations like dividing by zero or taking a square root of a negative number etc. For instance, the result of taking the square root of a negative number is equated with a special sentinel value N a N (Not a Number) corresponding with a bit pattern for which there is no valid proper number in the domain. Such a representation has the advantage of portability (i.e., the same program can be used over different systems) since at the program level one can detect as to when the result goes outside the range of the computation. The actual action to be taken in such a situation, called "exceptions"(such as divide by zero etc.), is anyway language and operating-system dependent.

The round-off specification is another feature which when developed as per the standard, leads to unambiguous hardware designs. For instance, consider multiplying two numbers 22.1×0.5 (equal to 11.05) which is needed to be rounded off to two digits. The question is: Should it be rounded to 11.0 or to 11.1. As per the standard, such halfway cases are rounded to 11.0 and not 11.1. The standard has actually four *rounding modes*; the default mode is round to nearest and round to an even number in case of ties.The other modes are: round toward 0, round toward $+\infty$ and round toward $-\infty$. Details or references to the standard specifications can be found in the references given at the end.

Need for Data Types

From the above representation of numbers and characters, it should be clear that unless the type of the memory word is known, the data cannot be interpreted properly;in fact, from the table shown it can be easily seen that it leads to misinterpretation. Thus, there is a need for explicit *type* information while writing programs.In general, the type information becomes useful for

- Interpreting the data (and also the instruction).
- Deciding about the memory requirements

Indian Script Code for Information Interchange (ISCII)

There are 17 officially recognized languages in India: Hindi, Marathi, Sanskrit,Punjabi, Gujarati, Oriya, Bengali, Assamese, Telugu, Kannada, Malayalam, Tamil,Konkani, Manipuri,Urdu,Sindhi and Kashmiri. Each language is written in its own script. With the exception of languages such as Urdu, Sindhi and Kashmiri all other languages are written in the following scripts: Devanagari, Punjabi, Gujarati, Oriya, Bengali, Assamese, Telugu, Kannada, Malayalam and Tamil. The above scripts have evolved from the ancient Brahmi script. The official language of India, Hindi is written in the Devanagari script; Marathi and Sanskrit are also written in the Devanagari script; Devanagari script is also the official script of Nepal. Though Urdu, Sindhi and Kashmiri get written in Devanagari (Sindhi gets written in Gujarati script as well), these are primarily written in Perso-Arabic scripts.

ISCII has been adopted by the Bureau of Indian Standards and is intended for use in all computer communication media that requires the usage of 7 or 8-bit character. In an 8-bit environment, the lower 128 characters are the same as the ASCII character set. The top 128 characters cater to all the 10 Indian scripts based on the ancient Brahmi script. A different standard is envisaged for the Perso-Arabic scripts. An optimal keyboard overlay for all the scripts based on Brahmi script has been made possible by the phonetic nature of the alphabet. Such a standard has the advantage of having a common code and keyboard for all the Indian scripts and hence,transliteration between different Indian scripts becomes straightforward. Further details can be found from the document IS 13194: 1991 from the Bureau of Indian Standards.

Some of the basic data types that are commonly used in high level-languages are given below :

- Integers: The internal representation is as discussed above.

- Boolean: The domain of Boolean has two values usually denoted *true* and *false*. The internal representation of this type requires just one bit (1 denoting *true* and 0 denoting *false*). In a word of four bytes, one can also use the representation of all 0's and all 1's for representing *true* and *false* respectively.

- Reals: The internal representation is as discussed above.

- Characters: The internal representation is as discussed above.

- Strings: Strings are nothing but a sequence of characters. Thus, one way of representation is to use a length designator that indicates the length of the string followed by that many characters. For instance, the string "SET <blank space> 83"is represented by 00000110 01010011 01000101 01010100 00100000 00111000 00110011 where the first 8 bits are the length qualifier; in this case the length qualifier has value 6 which is indeed the length of the string (note that <blank space> denotes one character).

- Sets: Consider a 4 byte word each consisting of 8 bits. We can use a word to represent a set consisting of 32 elements; we could map the elements to 0 to 31 positions in the bit representation say, from the least significant to the most significant position. For instance, a set {31,8,4,2,1} is represented by10000000 00000000 00000001 00010110 where a 1 in the ith position indicates that element *i* is in the set; if it is 0 it indicates that the element is not in the set. That is, a map between the element and the position of the bit in the word is to be maintained. Under such a representation, several operations on sets can be performed efficiently using logical operations.

In addition to the reasons for the type information cited above, the type information is generally useful to associate entities used in the program with their types, for the following reasons:

- It provides clarity to the program by specifying the potential collection of values that may be assumed by the entities.
- It delineates the set of meaningful operations that can be performed on entities of such type.
- It enables the programmer to arrive at a proper representation and manipulation of the entities.

It assists the programmer to prevent or detect errors in the program well in advance during the program design process.

In the next article, we illustrate the use of data structures in designing algorithms for sorting and searching.

Address for Correspondence
R K Shyamasundar
Computer Science Group
Tata Institute of Fundamental Research
Homi Bhabha Road
Mumbai 400 005, India
email:shyam@tcs.tifr.res.in
Fax: 022-215 2110

Suggested Reading

◆ R G Dromey. *How to Solve it by Computer*. Prentice Hall International, 1982.
◆ V Rajaraman. *Fundamentals of Computers*. 2nd Edition. Prentice Hall of India Pvt. Ltd, 1996.

No science is immune to the infection of politics and the corruption of power

Jacob Bronowski