An Elementary Introduction to Scientific Computing [1]

Charles C. Dyer and Peter S. S. Ip
Division of Physical Sciences
University of Toronto at Scarborough

Revised January 3, 2000

[1]This is a preliminary draft for CSCA57 students only, not to be distributed.

Chapter 1

Introduction to Computers

1.1 A Brief History of Computers

Through the centuries humans have used various devices to assist with mathematical calculations. Such aids to counting and calculation have included simple count recorders, such as stones in one-to-one correspondence to objects to be counted, such as cattle. In the last few centuries, with the invention of logarithms by Napier, analog methods based on logarithms were developed, such as that illustrated in 1.1. The use of tables of logarithms and trigonometric functions and highly automated methods for their use in navigation and related areas, where the human user is oblivious to the underlying principles, eventually led to one of the most widely used computational devices, the slide rule. Until the advent of small hand-held electronic calculators, the slide rule was the constant companion of engineers, scientists, technicians, and university students in related areas.

The modern computer differs from the slide rule and its predecessors in that it is not an analog device, but a discrete device. The slide rule functions by transforming a multiplication, for example, into a simple addition, where the quantities to be added are the logarithms of the multiplicands. The addition is performed by placing two lengths end-to-end, and comparing the total length to a scale of lengths corresponding to the logarithms of real numbers. Analog computers were developed in the first half of the twentieth century, using operational amplifiers based on vacuum tube electronics. These proved useful in specialized tasks, but they were difficult to program, and had very limited precision.

In contrast, the modern computer performs its arithmetic in a more exact fashion, at least for operations involving integers. Since, as we shall see, there is a finite set of integers which can be stored in a computer memory location, there is finite limit on the number of unique answers that can result from any operations performed on that computer. Thus the modern digital computer is in some sense a return to the earlier discrete counting approach to computation.

Beyond its capability to perform arithmetic operations, the major advance achieved with modern computational devices is that they can be programmed to perform highly repetitive tasks. The idea of a programmable computing machine is not new, being described in some detail by Charles Babbage, starting in about 1823. The realization of a practical programmable computing machine had to wait until the invention of the transistor in the 1950's.

The analog computer devices used the actual value of various physical parameters, such

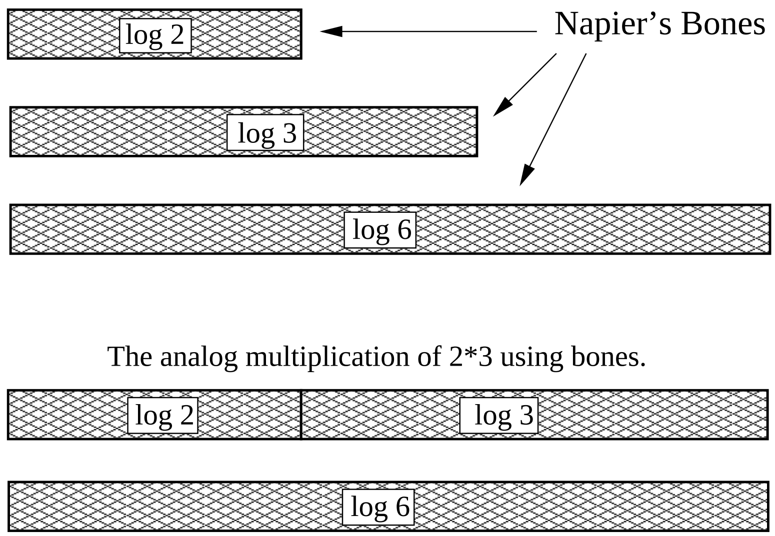

Figure 1.1: A simple analog computation based on *Napier's Bones*, a term derived from the appearance of the ivory rods, of logarithmic progression in length, used by Napier as the earliest form of what became the *slide rule*. In this example the product of 2 times 3 is computed by laying end-to-end the $log2$ and $log3$ bones and then finding another bone, in this case the $log6$ bone that matches.

as the output voltage of some electronic circuit, which thus required highly stable circuits in significant numbers. Any drift in the temperature of circuits, for example, would result in errors in the computed results. The major breakthrough of the digital computer involved choosing two distinct states of an electronic circuit, and using that circuit to only store one binary digit (known as a bit) of information, according to which of the distinct states the circuit occupied. With careful choice of the circuit, it is possible to ensure that the distinction between the two states is clear.

Thus the digital computer trades away the multitude of values one could store in an analog circuit, in return for having a highly deterministic binary digit stored. In terms of the development of the electronic circuits, the need for highly stable circuits capable of high precision was traded away in return for accepting the requirement of having many more circuits, each of which required less precise characteristics. This is a valuable trade-off, especially when the requirements for a very high degree of linearity in the analog circuits is taken into account, for it is very difficult to design analog circuits that remain linear over a large dynamic range. In contrast, the circuits for digital logic have quite modest requirements, since they are only required to maintain significant separation between two distinct states, with little requirement for linearity.

A further advantage of the digital approach to computers depends on the speed with which a typical digital logic circuit can change from one of its binary states to the other, the switching speed. In the case of an analog computer, severe constraints are imposed on operating speed when one also has the requirement of a high degree of linearity.

This book will deal only with digital computers. All representation of numbers and other data will be based on this binary digital model. The representation of numbers will thus be in a base of 2, rather than the more familiar decimal base of 10. The representation of textual information, for example, will involve establishing a one-to-one correspondence between the characters of the alphabet, the punctuation marks, etc., and a sequence of binary digit patterns, also known as bit patterns. For most text applications, the characters can be mapped onto either 128 or 256 different patterns, so that each character consists of a pattern of 8 bits (binary digits), which is usually referred to as a byte. The most frequently used encoding scheme is the *American Standard Code for Information Interchange*, or ASCII, for short. This scheme uses 7 bits and thus encodes 128 characters, the majority of which are actual printable characters. The remainder are so-called *control characters* such as the *carriage return* character, the *line feed* character, or the *tab* character. Since the ASCII code uses only 7 bits, this leaves one more bit free in the usual 8-bit byte, the *high bit*, which many manufacturers use to encode up to 128 special characters, frequently of a graphic nature, such as musical notes, mathematics symbols, etc.

1.2 Programming of Computers

The computer as described above is essentially just a calculator, and a very simple one at that. To be of any significant use in scientific problem solving, the computer must become programmable, in the sense that the sequence of calculations and decision making criteria to direct the calculations can be set out and made to control the sequence of steps taken by the calculator. The set of instructions which describe the process is called a program, and any programmable computer must have circuitry to allow it to execute the instruction sequence described in such a program.

At the lowest level, the program is written in machine code, which is in a format appropriate for direct interpretation by the computer circuitry as the computer reads the instruction sequence from some source, such as the computer's memory. This language level is very far from any human language, and thus quite inappropriate for the description of the methods to solve scientific problems.

While the computer can only execute instructions in its own machine language, we are free to describe our scientific problems in another language, which we will call a *higher level language*, that is more appropriate, provided we have available some device to translate the instructions from this high level language to the machine code appropriate to the computer being used. In fact, we can make an association between a language level and an abstract computer that executes instructions of that language, so that the choice of a programming language implicitly imposes the abstract machine on which that language will execute. Since one can construct translators from one language level to another, it is quite possible for a user to write programs at one language level which will actually be executed on a real machine of a much lower language level. In a well designed environment, the details of the actual machine on which the user's program executes should be of no direct interest to the user.

In the solution of many scientific problems, the natural language is that of conventional mathematics, with the usual sequence of assignments, using the = sign, involving expressions with addition, multiplication, and similar operations, as well as function evaluations. One of the first computer languages to provide significant facilities in this manner was the language FORTRAN (FORmula TRANslator) developed in the mid 1950's. While still used extensively, FORTRAN has been followed by a number of more modern languages ranging from BASIC to

ALGOL, PASCAL, and C. In addition to this class of languages, others such as LISP (also from the mid 1950's) and PROLOG have been developed to process information of a more abstract nature. The choice of language should normally be dictated by the problem to be solved, and at least one measure by which a particular language should be judged is the ease with which the language can be used to solve the greatest breadth of problems – having to learn a new language to solve a slightly different problem is not an efficient way to use human time!

While the choice among FORTRAN, PASCAL, and C is not clear, recent trends have been towards the use of C (the youngest of these three languages). Much scientific and engineering programming is still done in FORTRAN, in large part because there is a significant body of well designed and tested library routines in FORTRAN, but the trend is towards the use of C. C is now highly available, and in fact is available at no charge for almost all computers in use today through the GNU C compiler environment available through the Free Software Foundation (reference...). We will use C in this book.

While we will concentrate on the more numerical aspects of scientific computing, we will also discuss the representation and visualization of the results of observation and computations with graphical systems. As well, we will discuss some aspects of Computer Algebra systems which can be used to perform many of the more formal, or symbolic, calculations of interest to scientists. Each of these areas will require appropriate languages, with different facilities available to describe and manipulate data having properties quite different than data appropriate to C or FORTRAN.

1.3 The Programming Environment

To effectively use a computer for the solution of scientific problems requires an environment that provides easy access to the power of the computing system at a level appropriate to the user. Such an environment is provided by a number of pieces of software, including an *operating system*, a *text editor*, and either a programming language *interpreter* or *compiler* as the minimum. In addition, the environment should include many other software components to assist the user, such as extensive online documentation, communications tools, etc.

1.3.1 The Operating System

The operating system can be considered to be the *root* program executing on the computer. It is responsible for managing many services for all other programs executing, including the effective allocation of system resources, such as main memory, secondary storage space (typically on disk drives), access to various hardware subsystems (such as terminal ports, network connections, or printers), and of course, the fair distribution of access to the actual computer processor itself so that useful work can be done.

It is clear that an effective operating system must be able to perform a number of tasks in a manner which appears to allow simultaneous execution of these tasks. Since the computer processor can only execute one instruction at a time (this is slightly modified in modern processors which can perform a small number of instructions at once in a very restricted manner), the operating system must appear to provide a multi-tasking environment. This is accomplished by sequentially allocating access to the processor to various tasks (including itself!) in an appropriately prioritized manner.

The history of the development of operating systems involves an intriguing mix of careful analysis and design, and the market place where operating systems are used. This has led to the survival of operating systems that, while of relatively high quality at their inception, are far behind the state-of-the-art operating systems for the same hardware. A particularly interesting example is the MS-DOS operating system. This operating system is essentially a single-tasking system with limited provision for background execution of special services like printing. In addition, this operating system imposes a rigid limitation on the memory size and layout, reflecting the high cost of computer memory at the time of the design of MS-DOS. Even with these limitations, MS-DOS remains one of the highly dominant operating systems in the market, for reasons that are related to software vending dynamics. The large installed base for MS-DOS, combined with the greatly increased expectations of computer users, has led to numerous attempts to overcome the serious constraints of this operating system (while still using MS-DOS at the base). Examples of these attempts include the many memory extenders and multi-tasking add-on environments. In parallel with the development of these many ways of propping up the original MS-DOS, we have seen the development of other operating systems for the same hardware which provide true multi-tasking, resource sharing, multi-user environments. A particularly interesting example of these systems is the UNIX family of operating systems which are reasonably hardware independent, thus providing easy migration among hardware designs.

We will presume the availability of an operating system with at least the capabilities of MS-DOS, and sometimes assume the capabilities of a multi-tasking system like UNIX. While the operating system is responsible for handling many of the input/output tasks, the scientific user usually confronts many of these issues from the programming environment. Hence we will postpone discussions of some of the features of the input/output system until we consider a programming language in some detail.

1.3.2 The Text Editor

Unless we are content to enter instructions one by one for each step of our calculations, we will require some way to enter instruction sequences that will allow their re-use. Similarly, we will require some way to enter and store input data, and to examine the output data. These requirements are usually met by providing the computer system with a *mass storage* system, such as one or more magnetic disk drives. The allocation of space on these drives is the responsibility of the operating system, but the entry and manipulation of the content of *data files* on these drives is the rôle of the user in most cases.

One of the primary ways to create and manipulate files is through the use of a *text editor*, which allows the entry of textual data into a file that can be stored on a disk for later use. If we wished only to enter text that would be printed on paper, then the text editor can be visualized as a typewriter with memory. Of course, a typewriter with memory that cannot be modified does not allow the correction of the usually frequent typographical errors caused by “large fingers” or the occasional spelling error. Hence it is useful to develop a piece of software that can act not only as a text input device, but also as a text correction device – the basic rôle of the modern text editor. Once these two functions have been developed, there are many more functions that can be added, such as automated search and replace operations, simple text formatting, and spell checking. In addition, one can imagine that there are features of particular interest to the entry and editing of specific languages. This is particularly useful for the entry of the program text for languages such as C or LISP.

While there is great variety in the choice of editor, the fact that it is usually the most used piece of software suggests that it be chosen with a view to reasonably extensive facilities and great portability across all hardware configurations and operating systems where possible. There are many choices for the editor but in the interest of choosing editors that are widely available on many systems, those dominant in the UNIX environment are probably the best choice. The most used editors in UNIX environments are VI and EMACS. While they offer similar capabilities, their approach to the user-editor interaction is quite different. We will choose VI for its simplicity and portability (this results partly from the fact that it is distributed with every UNIX system, and there now exist a number of excellent public-domain versions for almost all operating systems), and because we have to make some choice! VI is quite an old editor and derives from the days when most computer terminals had no graphical input device, such as a *mouse* or *track-ball*, and when the standards dictating the control methods for video display terminals were highly varied, being proprietary to individual manufacturers. Thus it was necessary to design VI to be a very general editor which depended only on the most basic facilities, particularly the standard keyboard without any *function keys*. Of course VI allows the user to *map* any function keys or mouse actions to the standard command interface, allowing VI to become highly customized to any particular users hardware. For instance, most people would want to have *cursor* (or *arrow*) keys functional, and to have various shortcuts to perform frequently used operations.

The comments about VI indicate a significant difference from the *word processor* editors frequently found on the most popular personal computers, where the word processor software is highly dependent on the machine processor hardware type, the display screen control hardware, and so on. If the aim is portability to any work environment, it is usually best to choose the software that makes minimal assumptions about the hardware platform or the host operating system. It can come as somewhat of a shock when one realizes that one of the most popular word processing packages runs on only two operating systems (MS-DOS, and MS-Windows and derivatives), both of which can be characterized as relatively low-end operating systems available on only a small subgroup of all the different hardware platforms and operating systems available. The investment of significant effort learning packages that are not portable to higher performance systems may be a poor investment compared to learning a more portable package.

1.3.3 The Interpreter and Compiler

The first phase of preparing a program for the computer to execute is the entry of the program instructions at an appropriate language level, and this will usually be done with a text editor as already described. The language used to describe the procedure to be followed is almost always quite different from the machine language level. Thus some form of translation from one language level to a lower level is required. In the early days of electronic computers, this was done by hand assembling the machine instructions to execute each high level instruction. One of the first routine tasks to be automated was the process of such a translation, resulting in either an *interpreted* or a *compiled* execution environment for the program.

The distinction between the terms *interpret* and *compile* arises when we consider that most useful tasks in scientific computing involve highly repetitive instruction sequences, usually called *loops* of one form or another. For a program which involves significant looping, the process of translating from the high level language to the machine level for each iteration can consume a large fraction of the computer time, while for a program that does not have to execute many repetitive instructions, the cost of interpreting is relatively low.

When efficiency is critical, the compiled language approach is usually better, in part because of the elimination of the repeated translation task after the first cycle through a loop. A further significant improvement to be gained from the compiled approach is that the translation program, usually called the *compiler*, can be allowed to analyze the whole program, or at least large parts of it, to determine details of its structure which then allow optimization of the program. This usually results in significant improvement in efficiency, without the detailed intervention of the user. This feature cannot be implemented in the interpreted approach, since the interpreter cannot, by its definition, have significant information concerning the global structure of the program.

It is frequently useful to use both the interpreted and compiled approaches during program development, since the interpreted approach allows very fast incremental improvements to the program, which can then be compiled for actual production use. A prime example of such an approach is with the LISP language, where development is done in a LISP interpreted environment, and well tested modules can then be compiled by the LISP compiler and integrated back into the LISP interpreter. This hybrid approach forms the basis of many LISP applications, such as computer algebra systems, to be discussed later. While this approach can be used with languages like C and FORTRAN, there are frequently complications in the transition from interpreted to compiled programs. Of course the BASIC language was initially designed as an interpreted language, and the resulting low performance was later improved by adding the ability for the language to be compiled.

Recently developed languages that have been designed to be interpreted include JAVA, T-cL/Tk, and PERL, among others. Some of these languages have been designed from the outset to allow a simpler form of compiling called *byte compiling*. The target in this form of compiling is not a specific hardware machine, but a simple and well defined intermediate level machine which is a *virtual machine*. While this virtual machine is not implemented in hardware, its specification is available to software designers, leading to the development of the implementation of the virtual machine on any hardware platform and any operating system. This approach provides many advantages at the possible cost of slightly slower execution. An important influence on the design of these byte-compiled languages is the *device-independence* required for programs that must run properly on any machine connected to the Internet. A program written and byte-compiled in JAVA, for instance, can be run on any computer and operating system that have the *byte-code interpreter* specified in the JAVA design. The implementation of this interpreter is the responsibility of the local host computer. This local host can thus control access to its own resources by carefully watching the byte-codes as they arrive, and allowing execution of only those instructions that do not threaten the integrity of the local system. Good design can provide high security from outside attack by *hackers*, but sloppy design can lead to a computer

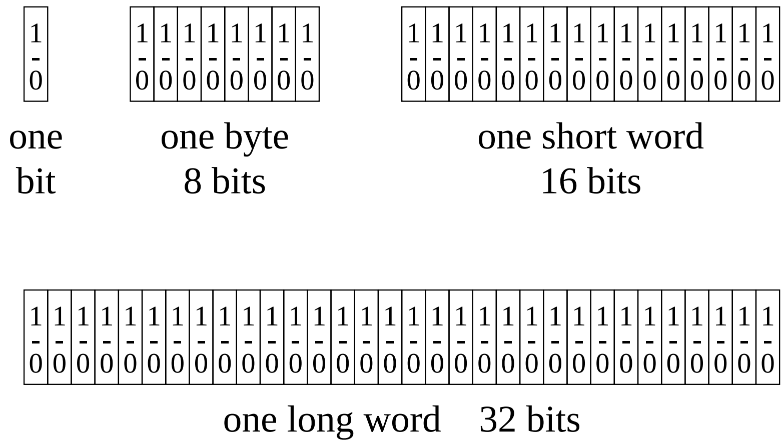

Figure 1.2: The *bit*, holding a value of 1 or 0, and typical groups of bits of practical use in computer memory and for computation.

system that is easy prey for outside hackers.

1.3.4 Data Representation – Data Types

While the basic memory unit in a digital computer is the *bit*, holding a value of 1 or 0, or equivalently *ON* or *OFF*, most information of interest to the user requires more than a simple bi-valued representation. Thus it is practical to form groups of bits, each consisting of some carefully chosen number of bits, and design the computer hardware to deal with these groups as complete units. In modern computers the most basic group consists of 8 bits, called the *byte*. Most computers implement groupings of two bytes, 4 bytes, and 8 bytes for purposes to be discussed below. Figure 1.2 illustrates the most frequently used word sizes. This is a useful approach from the users' point of view, and also from the technical implementation point of view. If the user requires information in chunks of 8, 16, or 32 bits, than there is no reason not to build the hardware so that it can transfer 8, 16, or 32 bits in one cycle rather than in sequence, and similar advantage can be gained by building the basic *arithmetic and logic unit* so that it can perform the basic operations of arithmetic and logic on these large chunks as well. This is the first and by far the most significant step towards implementing parallelism in computers. Most computers are now designed to handle *long words*, consisting of 32 bits.

The task of the computer is to process information, which it can only represent by various bit patterns, in a manner defined by the user's computer program. The significance of these bit patterns as data lies with the user and programmer, who has significant freedom in choosing the representations. In fact, in most programs, different memory locations will be defined to have completely different interpretations for any bit pattern. This leads to the possibility of assigning a *data type* to each *variable* in a program. The usual data types would include (we will use the names familiar to the C language – other languages have equivalents) `unsigned integer` (positive), `signed integer`, `float` (a finite subset of the rational numbers), as well as `char` for single characters. Most numeric data types can exist in at least two *sizes*. The `integer` data type usually has a `short` (usually 16 bits) and a `long` size. Similarly, the `float` type usually contains 32 bits, and its large counterpart is the `double` with 64 bits.

Address	123	124	125	126	127	128	129	130	131	132	133
Numeric value	69	76	76	69	78	32	68	89	69	82	0
Character value	'E'	'L'	'L'	'E'	'N'	' '	'D'	'Y'	'E'	'R'	'\0'

Figure 1.3: A simple array of bytes in memory. This example illustrates an array as a sequence of cells of the same data type, in this case interpreted as a *vector*, since it is a one dimensional array. This vector is also an example of a *character string*, since the basic data type is the byte, used here to hold one character, with the additional property that it is terminated by a special character, the *null character* which has the value 0, and is represented by the notation '\0'. The array is shown twice, the first copy showing the numerical value (in decimal) of each byte, and the second with the interpretation of these values as characters in the ASCII alphabet. Above the array of cells the address of each byte is shown, and it is clear that the bytes are contiguous in memory. The starting address of 123 is of no particular interest, and could lie anywhere in memory.

Of course, it is straightforward to develop the idea of a sequence of memory cells holding the same data type, and sharing a common name, with only a sequential index to distinguish them. This then allows the use of *arrays*, *vectors*, and *character strings*. In scientific computation the use of matrices in calculations is thus quite easy to implement.

While the implications of the finite number of bits used in each date type is easy to understand for the integer types, they are quite complicated for the `float` and `double` data types. In particular, there is always a temptation to treat the `float` and `double` data types as if they are *real numbers*, which they are most definitely not. In fact some languages use the data type name `real`, which adds to the confusion.

1.3.5 Implications of Finite Word Size

In the design of a practical computer, it is necessary to choose a finite word size that will be handled by the hardware as a unit. The usual choices are now 32 bits or 64 bits, while only a few years ago, most personal computers and many mini-computers used a word size of only 16 bits. The choice of the word size has significant impact on the usefulness of the computer for reasons that we will discuss here.

Before we consider the issues related to the accuracy of arithmetic operations, we will look at the impact of a limited word size on the amount of memory that can be accessed. If we define the word size of the computer to be w bits, these w bits can be chosen in 2^w different patterns, so that we can represent at most 2^w unique pieces of information. If we are using one computer word to describe the unique location of a piece of data within the computer's memory, we can only discriminate among 2^w unique locations. Thus, while an 8 bit computer may be useful for some arithmetic tasks, its limited word size allows the addressing of only 256 (ie. 2^8) unique locations in the computer's memory. This may be sufficient for a simple calculator, but it is clearly quite inadequate for most computational purposes. If we increase the word size to 16 bits, then since 2^{16} is 65536, it is quite adequate for many computational tasks. In fact, 16 bit computers were used for many years to perform significant tasks in scientific and engineering computation, as well as in other applications. For many applications involving experimental data and its analysis, a 16 bit word size is sufficient, since most experimental measurement apparatus does not have a dynamic range that exceeds a factor of 65536, though the use of highly linear solid state detectors has challenged this in recent years.

The rapid increase in the availability and the dramatic drop in the cost of graphics display devices has resulted in an ever increasing level of expectations for graphical user interface systems. Display screens with 1024×1024 pixels (picture elements), each having a dynamic range of 256 or higher, require memory addressing of millions of unique memory locations for the generation of images. Clearly a word size of 16 bits is no longer adequate for this task, and modern computer computers typically use a word size of 32 bits. This allows 2^{32} (somewhat more than 4 billion) unique patterns which is sufficient for many requirements. The choice of 32 is convenient because it can be subdivided into the two dominant older word sizes, ie. 16 bit words (sometimes called *short* words), and 8 bit words (which we have already called bytes). There are many situations where it is preferable to be able to refer directly to the subdivions of the 32 bit word, such as in text processing, where a byte can hold one character in many alphabets and the short 16 bit word can hold a character from almost any alphabet system.

The choice of word size has a significant impact on the arithmetic operations that can be performed easily. Starting with the case of integer arithmetic, the result of any integer operation must be one of 2^{32} possible results which would normally lie in the interval $[0, 4294967295]$ if we limit consideration to non-negative integers, or in the interval $[-2147483647, 2147483648]$ if we use one of the 32 bits to represent the sign of the integer. This range is quite sufficient for representing most experimental scientific data (with appropriate scaling factors) due to the limits on the achievable dynamic range of measurement instruments.

The limitations of 32 bit integers can be significant when computers are used to do exact mathematics, since it is quite easy to consider problems where the final result of a calculation is either 0 or non-zero, while the calculations to obtain that result might involve taking high powers of integers at intermediate stages of the calculation. A simple example is the demonstration that Pythagoras' theorem holds for a triangle with sides greater than 2^{16}, where the operation of squaring will lead to overflow, in the sense that the intermediate result lies outside the range of 32 bit integers. In such cases it is necessary to use multiple words to represent large integers. If we considered the case of the 3:4:5 right-angled triangle, and limited ourselves to a representation of integers that used only one base-10 digit, then we could not check that $3^2 + 4^2 = 5^2$ without introducing the notions of multi-digit arithmetic. The problems of large integer arithmetic on the computer can be solved in the same manner, but with a consequent increase in the effort required.

The discussion has been limited to arithmetic operations that map integer operands to integer results. This limitation is very stringent, since it does not include the division operation (except in special cases). It is clear that an extension to include rational numbers (essentially the quotients of integers) is required for most scientifically useful calculations. One simple method to represent some of these numbers is to represent each rational number as the ratio of two of the integers which we can represent. Such a scheme could be implemented, and is even sometimes preferable to the scheme we will discuss here, but it has many properties that limit its general usefulness. One limitation is that the largest number that can be represented is the simple 32 bit integer and the smallest is the reciprocal of that 32 bit integer, so that the dynamic range is equivalent to that of a double length integer.

While it is quite normal to adjust the scaling of the quantities in a scientific calculation, it is frequently quite inconvenient to do so in the intermediate stages of a calculation in the computer, since it not always easy to anticipate the scale of some intermediate results. Thus it is frequently worthwhile to consider a somewhat different distribution of rational numbers based on the familiar notion of *scientific notation* for numbers, where the rational number is represented in the form

$$mmmmm \times 10^{eeee},$$

where $mmmmm$ represents the significant digits of the *mantissa* of the number, and $eeee$ represents the *exponent* to which to raise the base (10 in this case). As an example, consider the number 0.000278356 which we can write in any of the forms:

$$0.00278356 \times 10^{-1}, 0.0278356 \times 10^{-2}, \ldots 2.78356 \times 10^{-4}.$$

While all these forms correctly represent the value, and there clearly are an arbitrary number of such expressions of the value 0.000278356, one particular choice stands out – that with exactly *one digit before the decimal point*, and all other digits after the decimal point. This *canonical* form is what we usually use in the so-called scientific notation. We also note that in any of the forms illustrated here, there is always the same precision represented by the number of *significant digits*. If we suppose for the moment that we can only compute with 5 significant digits, we would *round* the number to the representation:

$$2.7836 \times 10^{-4},$$

having reduced our precision from 6 digits to 5 digits. It is important to note that there are 10 numbers of 6 digit precision that, when reduced to 5 digit precision, all take the above value. When we limit precision by rounding (or by truncating), we also reduce the number of possible outcomes from any calculation in the process.

We can use the ideas discussed for base-10 scientific notation to develop a representation scheme for rational numbers in a computer with finite word size. The result is the *floating point* representation which partitions the computer word into two parts, one representing the mantissa and one representing the exponent, both with reference to the base 2. Both the mantissa and the exponent will carry a *sign bit* so that we can represent positive and negative numbers with magnitudes greater than or less than unity. There have been a number of approaches to the details of this form of representation, and we shall describe the modern IEEE version:

s	mmmmmmmmmmmmmmmmmmmmmmm	eeeeeeee

This uses one bit to represent the sign of the number, 8 bits to represent the exponent for the scaling factor, and 23 bits to store the mantissa.

We can invoke the requirement that the leading digit of the mantissa is always 1, in a manner similar to that used in the base-10 scientific notation, simply by adjusting the exponent. In the case of base-10 notation, the first digit must be one of the digits chosen from the set {1,2,3,4,5,6,7,8,9}, ie. the non-zero digits in base-10 arithmetic, but in base-2 arithmetic, there

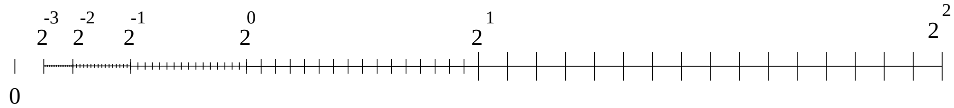

Figure 1.4: The distribution of points in each interval of factors of 2, using a 4-bit mantissa, so each interval has 16 points. Since the intervals have lengths varying by powers of 2, the density of points that are representable in the scheme also varies by powers of 2.

is only *one* non-zero digit, ie. 1. Thus if we impose this canonical form, we know that the leading digit of every non-zero number is exactly 1, so there is no need to store that digit. Thus instead of 23 bits, the mantissa has effectively 24 bits of precision, equivalent to about 7 base-10 digits (since $2^{24} \approx 1.7 \times 10^7$).

Since the exponent for the scaling factor has 8 bits, it can take on 256 unique values. In order that number less than 1.0 can be represented, these values are taken to be in the interval $[-128, +127]$. The scaling factor can thus range from 2^{-128} to 2^{+127} so the scaling factor can be as large as 10^{38} or as small as 10^{-38}.

It is important to realise that the floating point scheme for a 32 bit word size *can store no more information* than the 32 bit integers already discussed. The only change is in the *interpretation* of the bit patterns. In the integer representation, the points on the number line that are represented are uniformly distributed at the integer points, or any *fixed* rescaling of those points, leading to what are called *fixed-point* numbers. In the floating point representation, the points represented become exponentially more and more sparse for larger values. With 24 bits in the effective mantissa, there are 2^{23}, or about 8 million, points between $\frac{1}{2}$ and 1, and 2^{23} points between 1 and 2, and so on as the intervals over which the 2^{23} points are spread increases. A similar effect occurs for the decreasing intervals from $\frac{1}{2}$ to $\frac{1}{4}$ and from $\frac{1}{4}$ to $\frac{1}{8}$, and so on with the approach to 0, which is special in that it never lies in such an interval. The varying density of points in intervals is illustrated in Figure 1.4. Of course, the closest a result can be to 0, without being 0, is about 10^{-38}. Thus the density of points in the floating point number representation decreases logarithmically for larger scale factors and increases exponentially for smaller scale factors, which results in a very useful trade-off for many scientific calculations.

A close examination of the layout described will indicate that there are some bit pattern possibilities that have not yet been included – these are reserved to represent special values, including 0, and the so-called *NaN* (not-a-number) results that can arise from an attempt to divide by 0, or an overflow from multiplying two numbers which are so large that the resulting scale factor exceeds the limits discussed.

If we consider the multiplication of two integers, which we shall represent in binary numbers, then if the first of these has m significant bits counting from the rightmost bit, and the other has n significant bits, then the result of the multiplication will be a binary number with either $m + n - 1$ bits or $m + n$ bits. As an example, consider the multiplication of the two base-10 numbers 99 and 47, which in binary are 1100011 and 101111 respectively, as illustrated in Figure 1.3.5. The result in base-10 is 4653, which is 1001000101101 in binary, containing 13 bits. Both 99 and 47, having 7 and 6 significant bits respectively, fit nicely in a word size of 8 bits, what we have called the byte. The result of the multiplication requires 13 bits, so that it would not fit in

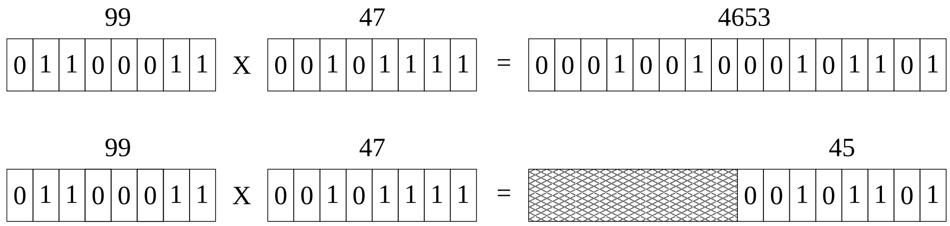

Figure 1.5: Multiplication of a 6 bit integer and a 7 bit integer, each represented within a byte, that is, 8 bits. In the top line, the full result is shown in a 16 bit word, with 13 significant bits. The second line shows the effect of discarding the top 8 bits and forcing the result into just the low 8 bits. The result of this last operation is to give a value which is the product *mod* 256, the remainder from dividing the full result by 256, that is 2^8.

the same 8-bit word size, but would fit nicely in a word of size 16 bits. If this multiplication is performed on a machine that can only handle 8-bit words, the result of this operation would be the low (ie. rightmost) 8 bits of the full result, that is 00101101, which in base-10 is 45. Thus the limited word size (8 bits in this example) results in arithmetic which is different than we would expect. In fact the arithmetic is modular arithmetic, where in this case it arithmetic *mod* 256, as can be verified by noting that 4653/256 yields 18 plus remainder 45.

We have seen that the usual integer multiplication scheme with a finite word size results in bits which would overflow the word size being discarded, resulting in a form of modular arithmetic. In the floating point system, the discarding of bits is reversed, so that the least significant bits are discarded with the consequence that the result of an floating point addition or multiplication is usually a either a rounded or truncated result. One useful way to see this is to consider the multiplication of two floating point numbers. The result of multiplying the two mantissas, each of 24 bits (say) will normally be a number with more than 24 bits of precision, but when picking a number from the possible 24 bit representations to return as the result, at best one can choose the closest, either by truncating the excess bits, or (at a higher cost) rounding the final bit that can fit in the 24 bit representation. Thus floating point arithmetic has sacrificed the uniform spacing of the possible results and the precision of the low-order bits to allow for a much larger and dynamic range of possible results. The corresponding weakness of the integer representation is the much more limited range of results, and the loss of precision in the most significant bits when the result would overflow the word size.

With either the integer or floating point representations, it is possible to extend the effective word size, and such a facility is always provided with the floating point scheme, usually referred to as *double precision* arithmetic. This comes at a cost in execution time, and in memory requirements, but it is important in many applications. The integer representation can be extended quite easily, and it is quite normal for LISP systems to implement integer and rational arithmetic to arbitrary range, by using multiple words to store the number, in a fashion that could be termed meta-digits. This can become quite expensive in computer resources, but without such a facility, computer algebra would be essentially impossible since, for example, proving that a polynomial has a particular set of factors depends on exact arithmetic, with no room for round-off error, or other approximations.

Chapter 2

The Python Programming Language

The use of the computer for scientific problem solving requires one or more languages to communicate our procedures and algorithms in a manner usable by the computer hardware. Since it is unreasonable to require the vast majority to use a language specific to a particular machine design, we normally use a high level language that promises to be widely available on most computers. Languages that have been popular for many scientific purposes include FORTRAN, BASIC, PASCAL, and C. FORTRAN has been dominant in science and engineering until recently, when the C language has been adopted for use in many applications. The Python language is a more recent development, combining many of the advantages of compiled languages like C and FORTRAN with the rapid development capabilities of interpreted languages like BASIC and the "shell programming" languages like TCL, PERL, of BASH. While we will choose to use Python in this book, there is quite a close correspondence in syntax to C and other structured languages.

2.1 Program Development

Before we begin a detailed examination of Python programs, it is useful to consider the typical procedure involved in developing a simple program. Figure 3.1 shows a flow diagram of this procedure, starting with the human input using a text editor, such as `vi` or `emacs` on a computer with a `UNIX` operating system. Once this file of Python instructions is complete, the file is written to disk like an ordinary text file. Exiting from the editor, the Python interpreter is started, and the new Python program is loaded and run. Normally there will be some errors reported by the Python system, in which case one can exit from Python by typing 'CTRL-D', that is by holding down the 'Ctrl' key and pressing the 'D' key.

The Python source that the user supplies does not constitute a complete program since it always requests services from the computer's operating system, such as access to various input/output devices, including disks, the keyboard, or the video screen. In addition, most of our programs will require the system to supply mathematical functions, such as the square root function, or trigonometric functions, that the user does not wish to rewrite each time a new

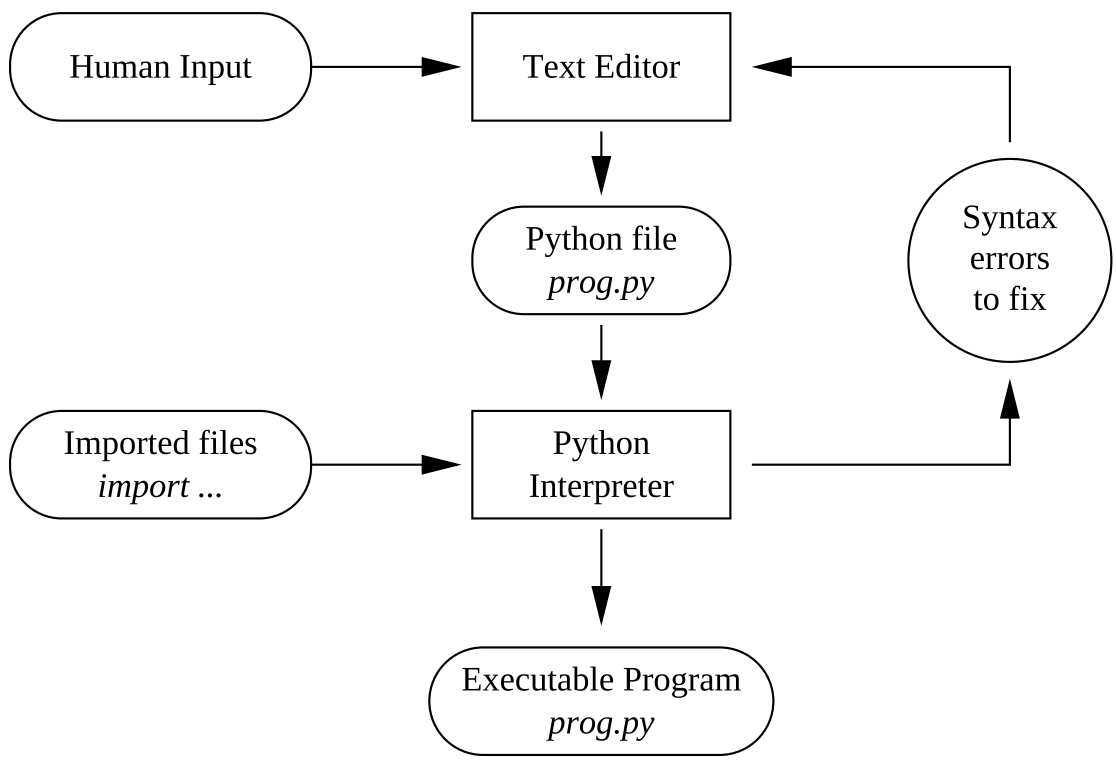

Figure 2.1: The simple program development loop for a program `prog.py`, showing the typical activities that eventually lead to a running program.

program is written. These services are usually made available through *libraries* or user loadable *modules* such as the mathematical library, or the basic system services library. Each of these libraries consists of a collection of *modules*, each of which performs some simple and well-defined task or set of tasks sharing common properties. Examples of such modules would include the *math* module which includes mathematical functions like the sin function or useful numbers like *pi*.

2.2 A Simple Python Program

We begin our study of the Python language with a very simple, and immensely popular program that prints a very simple message. We will assume that the program is stored in the text file `hello.py` on the computer disk, as:

```
#!/usr/bin/env python

# A VERY simple Python program ...

'''
    This is some text, called comments to describe the
    operation of this program. Note that this commentary text
    is delimited by three single quote marks in a row to start
    the commentary text, and another triplet of single quote
    marks at the end of the comment.

    While this particular comment has been delimited by three
    single quote marks, one can also use matched triplets of
    double quote marks in the same way.
'''
# And now for the actual statement to executed by Python:

print "Hello world!"    # This is a comment as well.

"""
    Note that I can use ''' in this text, since I used three
    double quotes to delimit this comment, rather than using
    three ' marks as I used in the previous long comment.
"""
```

When executed by Python, this program will simply write the text string `Hello world!` and return the computer systems prompt. The user simply types `python hello.py` followed by the ENTER key, to load and execute the program.

Before we analyze the program in the file `hello.py`, we need to introduce the idea of *comments*, meaning descriptive text which forms no part of the Python language to be executed. The purpose of this text is to explain various points about the Python program in order to make it more easily understood by the initial programmer and by those who have to maintain or enhance the program at a later time. Python has two basic methods of delimiting commentary text in a program. The best approach for short comments is to use the form of commentary text delimited by the `#` opening sequence. Once the `#` character appears on a line, it and all subsequent text on that line are ignored by Python. This is very useful for short comments on lines by themselves, and particularly useful for comments written after executable Python statements on a line, to indicate the purpose or restrictions of that Python statement.

2.3 The Second Python Program

We now consider a somewhat more complicated Python program which introduces a number of new ideas. The program `my_echo` echoes back to the user the input typed on the `my_echo` command line. Executing the command:

```
my_echo Ministry of Silly Walks
```

would result in the output to the screen of:

```
Program name = ./my_echo with 5 arguments given.
Ministry
```

while executing the command

```
my_echo "Ministry of Silly Walks"
```

would result in the output to the screen of:

```
Program name = ./my_echo with 2 arguments given.
Ministry of Silly Walks
```

The difference is caused by the fact that the first attempt passes a total of 5 arguments to the `my_echo` program (including the program name), while the use of the double-quotes around the character string results in it being passed to `my_echo` as a single argument. The difference is quite apparent in the difference in the two values reported for the argument count, `nargc`. In fact, the two values, 5 and 2, initially appear quite odd, until one learns that the name of the program (in this case `my_echo`) is always considered to be the first argument, so `nargc` is always at least 1.

```
'''
    FILE = my_echo:

    This is a simple echo program that echoes one character string, when
    a character string is given. If the character string consists of more
    than one word (as delimited by white-space) then the string should be
    "bound together" by enclosing it in double-quotes or in single-quotes.

    If no arguments are given to the program, it indicates this by
    responding with:

    **** Nothing to echo!

'''
#!/usr/bin/env python

import sys                 # Import the sys module, to get argv, etc.
```

```
nargs = len(sys.argv)      # The number of words on the command line.

print "Program name = %s with %d arguments given." % (sys.argv[0],nargs)

if nargs > 1:
  print sys.argv[1]
else:
  print "**** Nothing to echo!"
```

Note that we did not have to use the `.py` extension on the file name, using simply `my_echo` instead, to simplify the typing on the command line. How does the computer know that this is a Python program rather than a C program, or a FORTRAN program? The very first line of the Python program file:

```
#!/usr/bin/env python
```

provides the operating system with the required information. First we note that the line has the form of a Python comment, since it begins with a `#` character. The special sequence `#!` tells the operating system that this file should be handed to the program that immediately follows the `#!` pair. In this case the program indicated is `env` which is stored in the directory `/usr/bin` on most UNIX systems. `env` is useful in that it runs the progam specified (in this case python, the next word on the line), but only after finding where this program is located by using the `PATH` variable familiar on most operating systems, including UNIX and MS-DOS. Once the program (here python) has been started, it is handed the remainder of the file to execute.

While this may seem like a very involved approach, it relieves the user from having to know where `python` is located. This location can vary from system to system. On UNIX systems, `python` is usually found in one of `/usr/bin` or `/usr/local/bin`. Suppose that `python` is located in `/usr/bin` on one system you use. You could then replace the first line of the `my_echo` program file with the line:

```
#!/usr/bin/python
```

If you then moved your program to another system where `python` was found only in the `/usr/local/bin` directory, your program would not execute. The approach using:

```
#!/usr/bin/env python
```

would avoid this annoying dependence on the location of `python`.

In analyzing the structure of the `my_echo` text file, we must introduce some new symbols and their meanings in a Python program. The line:

```
import sys                  # Import the sys module, to get argv, etc.
```

tells Python to load (or "import") an additional *module* called `sys` so that its contents will be available. Loaded in this fashion, the resources contained in `sys` will be referred to with a preceding `sys.` to indicate that the particular resource is in the "name-space" defined in the `sys` module.

The `sys` module is particularly useful since it gives us access to information from the operating system, and allows us to use features of the operating system. In this Python program, we use the variable sys.argv which is actually an ordered *list* or an *array* of character strings containing the *arguments* passed to the Python program from the command line used to start the Python program. Since sys.argv is a list of items, it makes sense to be able to obtain its length, as the number of items, as:

```
nargs = len(sys.argv)      # The number of words on the command line.
```

The list sys.argv will always contain at least the name by which this Python program was invoked, so that in this case, sys.argv[0] contains the character string "./my_echo". We see that we can select a particular item from the list sys.argv by simply putting the "index" of the item in the list between [and] immediately following the name of the list, in this case sys.argv. A few lines further in the my_echo program the second item in the sys.argv list is used, referred to by with sys.argv[1].

Once the number of arguments, which we chose to call nargs here, has been determined, the print statement is used to report the program calling name and the number of arguments. We first saw the print statement in the previous program, where it was used to print a fixed text string, "Hello world!", to the screen. print is used in the same manner in the statement:

```
print "Program name = %s with %d arguments given." % (sys.argv[0],nargs)
```

but now the string is no longer a simple fixed string, but one in which some variables have been inserted, the program calling name and the number of arguments given on the command line. This line really combines two different actions, one being the printing of a string on the screen, and the other the *formatting* of that string to be printed. These two operations can be split apart, giving the equivalent Python code:

```
out_str = "Program name = %s with %d arguments given." % (sys.argv[0],nargs)
print out_str
```

The first line invokes a special *operator*, %, that takes two operands, one to the left and one to the right (similar to the division operator / which also takes two operands).

The operand preceding the % operator is the *format specifier* string, all of which will be printed, except for the sequences beginning with %. These *format control strings* are derived from those used in the fprintf functions of the C language. They are very flexible, and due to the complexity of all the possible options, the reader should get full details by using the "man fprintf" command in UNIX.

The operand following the % operator is a Python data structure called a *tuple*, denoted by enclosing a sequence of comma-separated Python data items in a (and) pair. For the present purposes, this *tuple* simply contains the names of all the variables to be inserted into the *format control string* in a one-to-one correspondence with the *format specifiers*, as indicated by the % characters in that *format control string*. In the present case, the specifier *%s* indicates that the variable sys.argv[0] should be printed as a character string, while the specifier *%d* indicates that the next item in the *tuple*, nargs should be printed as a base-10 integer.

In examining out the equivalence of the single line version and the split two line version in this particular case, one can see that the % operator can be used for many purposes beyond just printing things on the screen.

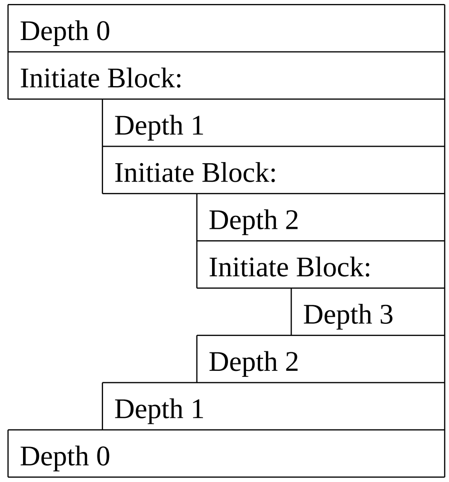

Figure 2.2: The nesting to successive depths of Python code blocks. Each new depth of nesting begins with a statement, such as an `if` or a `while` statement, that initiates a new block, indicated by the `:` and an increased level of intentation. The end-of-block is indicated by returning to the previous level of indentation.

Once the information about the arguments passed to the `my_echo` program is printed, the remainder of the program branches to one of two actions based on the outcome of a conditional test. This is a simple case of the more general multiple branching structure of the form:

```
if <condition_A>:           # Only one of these!
    <statements_A.1>            # block of statements to execute
    <statements_A.2>            # if condition_A is true
elif: <condition_B>:        # Zero or more of these, as required.
    <statements_B.1>            # block of statements to execute
    <statements_B.2>            # if condition_B is true
    <statements_B.3>
elif: <condition_C>:
    <statements_C.1>            # block of statements to execute
    <statements_C.2>            # if condition_C is true
else:                       # At most, one of these!
    <statements_D.1>            # block of statements to execute if
    <statements_D.2>            # none of the previous conditions are true
```

It is very important that the `:` be present where indicated, and that the indentation of conditional blocks of line be present and maintained. In this aspect, Python is very different from languages such as C, where "white-space" is quite unimportant, with lots of braces like { and } being used to indicate the block structure intended by the programmer.

The first condition, `condition_A`, is evaluated, and if it returns *true*, the subsequent indented block of statements are executed. If `condition_A` returns *false*, the next condition, `condition_B` is evaluated, and if true, the subsequent indented block of statements are executed. This process is continued until a condition evaluates to *true* in which case its subsequent indented block of statements are executed, or if all conditions evaluate to *false*, then the `else` block of statements is executed. Once one condition evaluates to *true*, or the `else` block is reached, and the indented block of statements is executed, the remainder of the `if`, `elif`, ... `else` blocks are skipped and control passes to the next statement following this structure.

================WORK-IN-PROGRESS========================

Thus the sequence

`if( 37 )` { *action1* } `else` { *action2* }

would always result in *action1* being taken, while

`if( 2*0 )` { *action1* } `else` { *action2* }

would always result in *action2* being taken since the condition always evaluates to zero. In the simple case at hand, such a test is used to determine if there is any data worth printing (`argc > 1` means `argc` greater than 1), and if not, an informative message is printed.

Each of the actions, such as *action1*, can consist of a block of any number of C statements separated by the semicolon character in the usual way as in the example:

```
if( x > 0 ) {
    w = sqrt(x);
    v = sqrt(y + x);
    }
else {
    w = sqrt(-x);
    v = sqrt(y - x);
    }
```

This has the effect of setting `w` to be the square root of the absolute value of `x`, and `v` to be the square root of the sum of `y` and the absolute value of `x`.

While we have already included some arithmetic expressions, we need to introduce the arithmetic operations commonly used in C. For sicientific calculations, the most important arithmetic operators are listed in the table:

C arithmetic operator	Operation	unary or binary
++	increment	unary
--	decrement sign	unary
-	change sign	unary
*	multiplication	binary
/	division	binary
%	remainer	binary
+	addition	binary
-	subtraction	binary
=	assign value	binary
+=	add to value	binary
-=	subtract from value	binary

These are listed in their order of precedence, beginning with the highest. The operators `++`, `--`, and unary `-` are of equal priority. The operators `*`, `/`, and `%` are equal and next in priority, followed by `+` and binary `-`. The assignment operators `=`, `+=`, and `-=` have equal, and lowest pririty.

When the action consists of only one C statement, the enclosing {...} is not required, so we could write:

```
if( x >= 0 ) printf("Absolute value is %d\n",x);
else printf("Absolute value is %d\n",-x);
```

Frequently we only need to check a condition, which if satisfied requires some action and otherwise the execution should just continue. This eliminates the need for the *else* statement, as in the example:

```
if( count > maximum ) {
    printf("Maximum count exceeded: Count = %d\n",count);
    exit(1);        /* EXIT - serious difficulties! */
    }
printf("About to process %d records ...\n",count);
... more statements ...
```

In many situations we need to use a multi-way branch decision structure based on testing a sequence of conditions. This can be done by simply stringing together a number of *if ... else* statements in the general form:

```
if( condition1 ) { action1 }         /* first condition/action pair    */
else if ( condition2 ) { action2 }   /* second condition/action pair   */
else if ( condition3 ) { action3 }   /* third condition/action pair    */
else { action4 }                     /* default if all conditions fail */
```

The multi-way branch must be distinguished from the *nested* `if` statements where the intention is to have conditional branching inside a block of C statements comprising the action of a higher level conditional branch structure. Consider the example:

```
if( y > 0 ) {
    if( x > 0 ) root = sqrt(y/x);
    else if( x < 0 ) root = sqrt(-y/x);
    else root = -1;
    }
else if( y < 0 ) {
    if( x < 0 ) root = sqrt(y/x);
    else if( x > 0 ) root = sqrt(-y/x);
    else root = -1;
    }
else root = 0;
```

This returns the positive square root of the absolute value of y/x unless $x = 0$, when it returns the negative value -1. Of course it is not the easiest way to get this result, but it does illustrate the use of nested multi-way branching. This nesting requires a clear understanding of the scope implied for each block of C statements by the matching `{...}` pairs.

We have used a few *relational operators*, such as `<` or `>`, in our examples so far. These are members of a set of relational operators that can be used in C programs which are summarised in the following table:

C notation for operator	Relation
`<`	less than
`>`	greater than
`<=`	less than or equal
`>=`	greater than or equal
`==`	equal
`!=`	not equal

The use of the `!` character to indicate the *negation* of a relation is general in C, but it is important to note that `!` is a *unary* operator (operates on a single entity) in all cases. In some circumstances, the arithmetic minus sign, $-$, acts as a *unary* operator, as in the expression `exp(-1.3)`, but the interpretation of $-$ depends on the context. In the expression `exp(x - y)`, it is clear that the $-$ acts as a *binary* operator (note that there is no *unary* $+$ in C). The case of the *binary* operator `!=` uses this sense of negation, but specific to the test for inequality. Thus it would not be correct to use a binary operator in the form `!>` to mean *not greater than*, but of course, the `<=` operator does mean *not greater than*. It is important to note that the *equal to* relational operator is the 2-character string "==" and not the single character "=". The single "=" character will always mean *value assignment* in C. Thus the result of the statements:

```
x = 5;
k = x;
```

is to give both `x` and `k` the value 5, while the result of the statements:

```
x = 5;
k = (x == 3);
```

is to give `x` the value 5 and `k` the value 0, since the relational test `x == 3` is false (`x` is 5, not 3). Since an expression like `x == 3` returns a value based on the result of the test, this value could be assigned to `k` in this example. Thus the result of the statement `k = (x < 10)` would be to set `k` to one, while `k = (x >= 7)` would assign `k` the value zero.

It is often useful to use multiple conditional tests to reach a branching decision. This is facilitated by the availability of the *logical operators* given in the table:

C notation for operator	Relation	Binary or Unary
`&&`	logical AND	Binary
`||`	logical OR	Binary
`!`	logical NOT	Unary

Again it is important to note that the 2-character strings "`&&`" and "`||`" are quite distinct from the 1-character strings "`&`" and "`|`". The 1-character strings represent *bit-wise* operations, and are not used that frequently in the kind of computing applications considered here.

The following C sequence of C statements uses compound tests to handle the conditions under which the square root of y/x can be computed and returns -1 when it cannot be computed with real numbers.

```
if( ( y >= 0 && x > 0 ) || ( y <= 0 && x < 0 ) ) {
    root = sqrt(y/x);
    }
else {
    root = -1;
    }
```

2.4 Simple Input/Output

For most scientific programming purposes, friendly interaction with the program through well designed and flexible input/output mechanisms is of some importance. The next example of C programming in the file `hypot.c` calculates the hypotenuse of a right angle triangle, given the other two sides. The program has been designed to allow the user to give a simple command like:

```
hypot 3 4
```

which will print the well-known result, 5, or to allow the user to simply start the program as:

```
hypot
```

whereupon the program will prompt the user for the appropriate input in the form:

```
Enter X and Y:
```

to which the user replies with the values of X and Y, the two sides opposite the hypotenuse.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <math.h>

main(int argc, char *argv[])
{
    float x, y, h;
    int count;
    if( argc == 3 ) {
        x = atof(argv[1]);
        y = atof(argv[2]);
        }
    else {
        fprintf(stderr,"Enter X and Y: ");
        count = fscanf(stdin,"%f%f",&x,&y);
        if( count != 2 ) {
            fprintf(stdout,"Invalid or incomplete data entered - exit!\n");
            return(1);
            }
        }
    h = sqrt(x*x + y*y);
    fprintf(stdout,"The hypotenuse on %f and %f is %f\n",x,y,h);
    return(0);
    }
```

This program introduces a very useful, but simple, approach to supplying input data to a program and getting output from a program. The model being used here is that of a `filter` in a simple picture of a useful process that takes input into some standard receptacle labelled *input* and sends the processed data out some standard receptacle labelled *output*. The standard names adopted for these receptacles or connectors (like the input and output connectors in a home stereo system or a VCR) are `stdin` and `stdout`. If all goes well, the programs we write will usually accept their input data and produce valid output data, but sometimes things go wrong, and it is necessary to have another output channel through which to send error messages or other special messages to the user. While the messages may not always be *error messages*, we usually refer to this second output channel as the `stderr` stream of output. One frequent use of `stderr`, other than error messages, is for prompting the user for input parameters, when we do not want the text of the prompt to the user to appear interleaved with the valid output of the program that is going to `stdout`. The file `stdio.h` contains the definitions of `stdin`, `stdout`, and `stderr`, and these three streams are opened by the operating system when it starts the program.

The function `printf` is already familiar for simple output, and we now have an example of a related function, `fprintf`. In fact the function `printf` is just a synonym for the `fprintf` function with its first argument set to be `stdout`. In the `hypot.c` file the `fprintf` function is invoked with `stderr` as its first argument, so that the prompt to the user for input of `x` and `y` appears on the user's display, but does not go to `stdout`. This distinction between `stderr` and `stdout` is not obvious when the program is invoked from a terminal and the output is viewed on the terminal, but we shall see below that the *redirection* of input and output can make the distinction important and very useful.

The `fscanf` function is invoked to get the values of `x` and `y` when the user is prompted for input. This function reads input from `stdin`, and a synonym for this more general function is

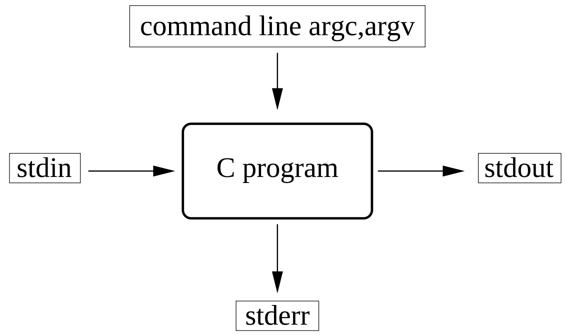

Figure 2.3: The simple 3-stream view of a process, consisting of one input stream, `stdin`, and two output streams, tt stdout for useful and normal output, and `stderr` for error messages or status reports that should not usually be part of the *pipe* structures. Note that one additional source of input is shown – the command line typed to invoke the program. This input is available to the C program through the arguments to the `main` function definition discussed in the text.

`scanf`, when the first argument is `stdin`. The first argument to `fscanf` is the input stream from which to read the data (thus `scanf` is to `fscanf` what `printf` is to `fprintf`). In a manner similar to the `fprintf` function, `fscanf` takes a format string which describes the types of data to be read using the % template notation, like `fprintf`. There is one very important distinction to be made between the output process (`fprintf`) and the input process (`fscanf`). In the output process, one specifies **what** (ie. value) to print, while for the input process, one can only specify **where** (ie. location in memory) to store the value read from the input. This distinction must be made clear to the C compiler, and this distinction is shown in the notation illustrated in `hypot.c`. Consider the variable `x`, declared to be of type `float`. When we deal with the value of `x`, we use the simple name for `x`, ie `x` stands for the *value of* `x`. When we deal with the memory location where the *value of* `x` *is stored*, we use the notation `&x` to refer to the *location* or *address* of `x`. Thus if we wish to assign a new value to the `float` variable `x`, its previous value is of little interest to us, but the address of the piece of memory in which the value of `x` is stored is what we need.

The files included at the beginning of the file `hypot.c` now include two more header files, `stdlib.h` and `math.h`, both from the standard include directories. The first of these includes definitions of the type of values returned by many functions – in our case it is the function `atof` that is of interest. Similarly, `math.h` includes information about the type of values returned by many mathematical functions – in our case it is the square root function `sqrt` that is of interest.

This program decides on the input mode the user is using by testing the value of the argument count, `argc`. If the user enters the program name and two values on the command line, the number of arguments is 3, and the program can compute the hypotenuse directly. If the number of arguments is not 3, the program prompts the user for the values of `x` and `y` using the `scanf` statement.

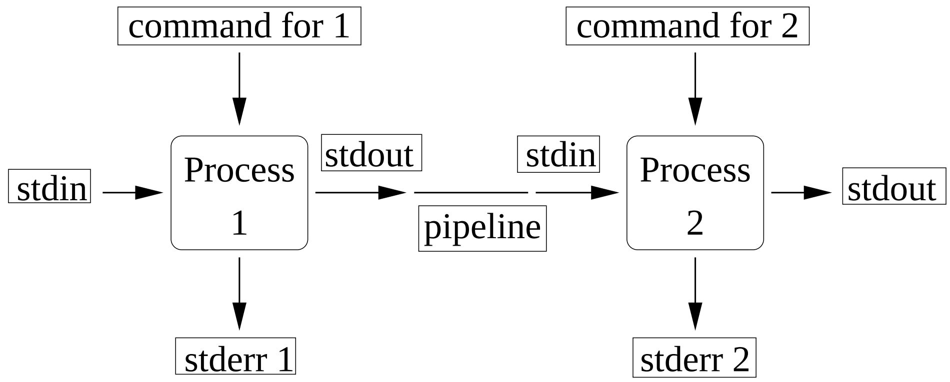

Figure 2.4: The *piping* together of processes by connecting the `stdout` output stream of Process 1 to the `stdin` input stream of Process 2. The `stderr` output streams of Process 1 and Process 2 remain distinct so that error messages from the process can handled differently. Similarly, the command lines for each process in the pipeline are distinct so that each process can be controlled. This type of structure can be extended to as many processes in the sequential stream as are required.

2.5 Input/Output Redirection – or Virtual Plumbing

The filter model for the design of many useful programs becomes a very powerful facility when we build composite filters by connecting the output stream of one filter program to the input stream of another filter program. This facility is provided by the operating system by *I/O redirection*. An analogy with simple plumbing is useful with the operating system providing a small set of connectors. The first of these is a simple straight connector which allows the feeding of the output from one filter into the input of another filter, and is denoted by:

```
process_1 | process_2
```

so that `process_1` provides its output (via `stdout`) as the input (via `stdin`) to `process_2`. The symbol | is called a *pipe* for obvious reasons.

The input data for a filter will frequently come from a data file already existing on the computer, so it is useful to provide a mechanism for feeding the contents of a file into a filter process, so that the `stdin` stream is fed from the file rather than from the keyboard of the user's terminal. The notation that sets up such a connection takes the form:

```
process < file_of_data
```

where the < indicates that the `stdin` stream for `process` should consist of the contents of `file_of_data`. A similar notation exists for redirecting the tt stdout stream from a process, indicated by the use of the > symbol instead of the < symbol.

As an example that we might find useful in managing our own files and directories, we might want to know which our subdirectories of files are the *disk hogs*. As simple way to do this is the *stream* of processes:

```
du -k /u/my_name | sort -n > hogs_list
```

The utility `du` computes the disk-usage for each subdirectory in the directory listed as its argument, in this case `my_name`. The `-k` switch simply asks `du` to give the directory sizes in *kilobytes* (ie. 1024 bytes), since some systems report the sizes in units of 512 bytes instead. The output from `du` goes to `stdout`, which is connected by default to our video screen. Here the `stdout` stream from `du` has been *piped* to the `stdin` of another program, `sort`, instead. The general purpose sorting program, `sort`, has many options specified by switches, but here we only use the switch `-n` which indicates that the sort should be numerical rather than alphabetic. The `stdout` output from `sort` is redirected by the `>` operator into the *file* `hogs_list` (where the choice of name is, of course, arbitrary). This file will contain a list of all your subdirectory sizes (in kilobytes), sorted numerically so that the largest subdirectory will be at the end.

The example illustrates the process of *pasting* together standardly available processes to build a more complex process as required by the user, but without the user having to write a complex program specifically designed for the task. The ease of doing this in *UNIX* and its derivatives (such as MS-DOS) is one of its most significant advantages, leading to much more rapid development of software. One aspect of this approach is that the *modular* design this allows results in fewer programming errors, since the persons responsible for the design of the `sort` program, for example, are free to concentrate their analysis on this task alone, and not all the uses humans may find for their module. Thus other programmers do not have to "re-invent" the program for sorting each time they need to sort data. The *redirection* mechanism allows many more complex possibilities, but the simple use of `|`, `<`, and `>` described here will handle most common situations.

2.6 Data Types Available in C

We have discussed the basic data-types available on modern computers in an earlier chapter, so we now turn to invetigating how these are made available in the C language. The simplest data-types are what are usually called *scalars*, which can be used to represent the basic building blocks for more complicated composite data-types. Integers, restricted to be small enough to fit in one computer word (say 32 or 64 bits), are the most basic data types beyond bits themselves. In C one can select from a limited number of classes of integers, based on the number of bits used to represent them. In the previous examples of C programs, we have seen a few examples of integers being *declared* as `int` as in:

```
int x, y;     /* Declare both x and y to be signed integers */
```

which tells the C compiler that `x` and `y` are to be taken to be the *natural* integer for that class of computer hardware. Normally this would a 32-bit signed integer, meaning that 31 bits are used for the binary representation of the absolute value of the integer, with one bit being reserved to represent the sign of the integer.

The *natural* integer for a particular computer system is fixed by the hardware manufacturer and does vary across the range of hardware commonly available. At present the natural word size on most computers is one of 16 bits, 32 bits, or 64 bits, with the typical word size becoming larger in the more recent machines. As a consequence of this variation in word size, C allows a more precise specification of the size characteristics of an integer through the use of additional data-types. In the declarations:

```
int x, y;       /* Declare x and y to be signed integers */
short s;        /* Integer s is a 16-bit signed integer  */
long t;         /* Integer t is a 32-bit signed integer  */
```

`x` and `y` are the natural integers for the machine, while `s` is declared to behave exactly as a 16-bit signed integer in all its operations, and `t` is declared to behave exactly as a 32-bit signed integer in all its operations.

In real applications, it is important to understand that declaring a variable as `int` should be thought of as declaring it to behave like a 16-bit signed integer, but implemented on the particular hardware in the most efficient way for that hardware. The declaration of the variable as `short` makes that behaviour explicit, but may result in some inefficiency on hardware with a natural word size of larger than 16 bits. Consequently it always best to restrict the use of the `int` declaration to those variables which you know will always fit within the `short` size limits, but where you want to leave the compiler free to implement the program in the most efficient way for that hardware.

The `long` declaration should be used when integers that can exceed the `short` limitations, regardless of the hardware word size, since we should worry first about the correctness of the calculation and then about the possible efficiency enhancements.

In many applications, there are integers that will always be non-negative, so that it is useful to specify that some variables should always be considered to have this property. One simple advantage is that the use of what would have been the sign-bit for a mantissa bit doubles the range of integer values that can be represented. More importantly, this added information can sometimes be used by the compiler to detect inconsistent use of the variable or functions involving this variable. The C keyword `unsigned` is prepended to the declaration in this case:

```
unsigned int x, y;      /* Declare x and y to be unsigned integers */
unsigned short s;       /* Integer s is a 16-bit unsigned integer  */
unsigned long t;        /* Integer t is a 32-bit unsigned integer  */
```

We have already encountered the `char` data-type in our earlier discussion of *character strings* which we typicall use to represent text. The `char` data-type normally comprises 8 data bits, and can be declared as `unsigned` if required. This data-type is usually too small to store the integers we would use in most programs, but it is useful to store *flags* or other special status information which is by definition of a very limited dynamic range, ie. in the range [0,255]. In situations where there are only a small number of flags to be stored, there no advantage to the use of `char` variables, but when it is necessary to store large arrays of flags, then the use of `char` variables can lower the storage requirements.

The *floating point* representation of a subset of the rational numbers was discussed previously. We saw that useful sizes for floating point numbers are 32 bits and 64 bits, with extended precision to 128 bits now becoming more available. The C declarations for such variables follows the pattern that we have seen for integers, taking the form:

```
float x, y;     /* Declare x and y as 32-bit floating point numbers */
double u, w;    /* Declare u and w as 64-bit floating point numbers */
```

2.7 Arrays in C

In most applications, there is a need to store *arrays* of C variables of identical data-type, and this is made available in C through the use of variables declared using notation illustrated by the following example:

```
float xdat[100];
float ydat[100];
```

These statements indicate that there should be 100 sequential locations in memory set aside for the array `xdat`, and similarly for `ydat`. It is important that the 100 `float` locations be sequential, and this is guaranteed to be the case. Each of these sequential locations can be treated as a normal `float` variable in arithmetic expressions, etc.

The general form of an array variable declaration is then:

```
data_type_keyword var_name[number_of_elements];
```

It is important to note that the `number_of_elements` value must be completely known at the time that the code is given to the C compiler. Thus the declarations:

```
int size;
...
double shift[2*size + 10];
float epsilon[3*size];
```

would not be valid, since the number of elements in each case is variable, and cannot be known at compile time. On the other hand, the declarations:

```
#define SIZE 50
      ...
      double shift[2*SIZE + 10];
      float epsilon[3*SIZE];
```

are acceptable, since the value of the symbol `SIZE` is known at compile time, due to the definition of `SIZE` in the line:

```
#define SIZE 50
```

This definition of `SIZE` is used *before* the C code is passed to the C compiler, since the C code is first handled by the `C preprocessor`, where all lines that begin with # in the first column are acted upon.

Each *element* of array is referred to by its *index* in the format illustrated in the statements:

```
#define SIZE 100
      float xdat[SIZE];              /* allocate memory for SIZE floats */
      float ydat[SIZE];
      int i;                         /* integer to use as an array index */
      float scale = 2.337;           /* declare and initialize variable scale */
      float shift = -1.4597;         /* declare and initialize variable shift */
      i = 0;
```

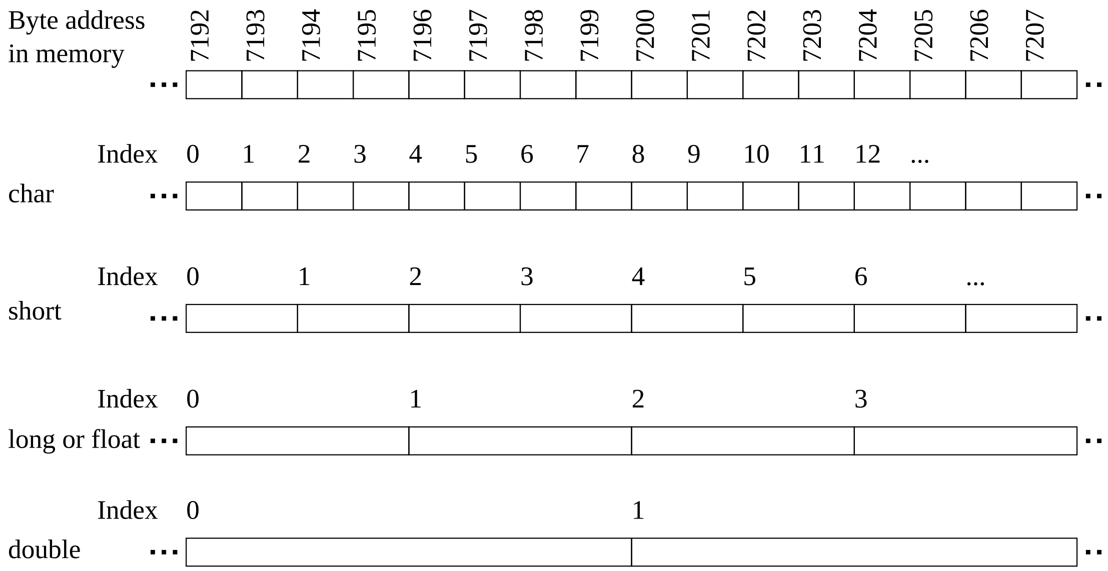

Figure 2.5: Layout of various data types in memory, showing the normal alignment of each data type, so that for a data type of size N bytes, each element starts on a multiple of N.

```
while( i < SIZE ) {           /* initialize the xdat array values */
    xdat[i] = i*i;
    i = i + 1;                /* increment the array indexing variable */
    }
for( i = 0 ; i < SIZE ; i = i + 1 ) {         /* initialize ydat array */
    ydat[i] = scale*xdat[i] + shift;
    }
```

Presuming that each element of the `xdat` array has a value, this C code sets each value of the array `ydat` to a scaled and shifted copy of the corresponding element of the `xdat` array. This example illustrates two new *control structures* in C – the *while loop* and the *for loop*.

The structure of the `while` loop is:

```
while( condition ) { statements; }
```

The *statements* are executed as long as the *condition* continues to be logically true. If the *condition* is not initially true, then the *statements* will not be executed at all. If the *condition* is initially true, then there must be some statement in the block of *statements* which will change the status of the condition to false – otherwise the `while` loop will never terminate!

The `for` control structure is essentially a concise way to implement a control structure that we could already construct based on the `while` and `if` statements. The structure can be written as:

```
for( initialize_statements ; continue_condition ; step_statements ) {
    statement_body;
    }
```

This is best understood by writing it in the form:

```
initialize_statements;
while( continue_condition ) {
    statement_body;
    step_statements;
    }
```

2.8 Initialization of Array Variables

There are many situations where it is useful to be able to not only allocate memory for an array variable, but to place specific values in each of the elements of that array. Consider an application which requires knowing the number of days in each month of the year. The following program is intended to print a table of the number of days in each month.

```
#include <stdio.h>
main() {
    int mdays[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
    char *month[] = {"January","February","March","April","May","June","July",
                     "August","October","September","November","December"};
    int nmonths = sizeof(mdays)/sizeof(mdays[0]);
    int i;
    for( i = 0 ; i < nmonths ; ++i ) {
        printf("%-10s has %d days\n",month[i],mdays[i]);
        }
    }
```

The output takes the form:

```
January   has 31 days
February  has 28 days
March     has 31 days
April     has 30 days
May       has 31 days
June      has 30 days
July      has 31 days
August    has 31 days
October   has 30 days
September has 31 days
November  has 30 days
December  has 31 days
```

Each of the arrays `mdays` and `month` has 12 elements, though of very different data types. The size specification of 12 for `mdays` is actually redundant, since the C compiler is presented with explicit data to place in each successive element of `mdays`. In fact, when an array is going to be initialized with explicit data, as in this case, it best *not* to specify the size of the array, but simply leave it empty, as done here for the declaration of the array `month`.

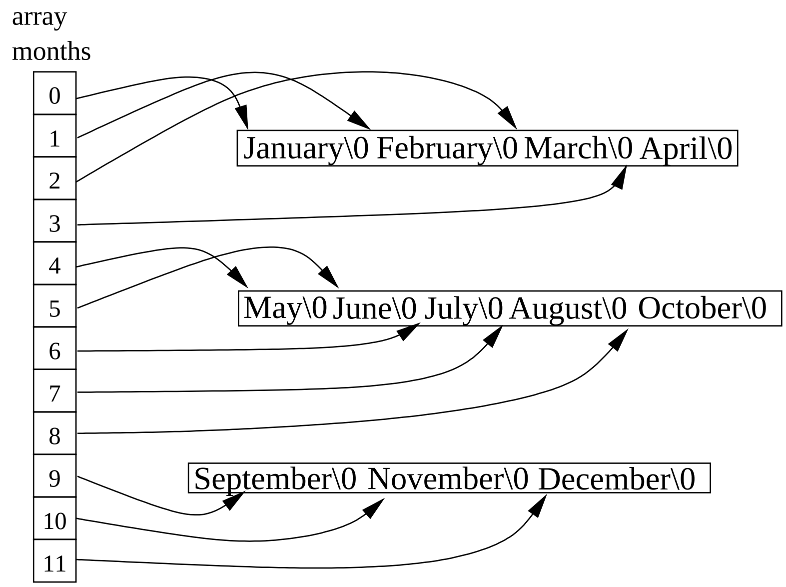

Figure 2.6: The use of a pointer array, `month`, to access the strings representing the names of the months, each stored somewhere in memory. There is no requirement that these 12 character strings be stored anywhere near each other in memory.

In fact, there is no need to ever specify a size of 12 in the program, since we can calculate the number of data elements in an array through the careful use of the `sizeof` operator. This operator returns the size of the argument in bytes, so that `sizeof float` returns 4, `sizeof char` returns 1, and `sizeof "specification"` returns 14, since the string "specification" contains 13 real characters, followed by the termination character, the *null* character, '\0'. Thus the expression `sizeof(mdays)/sizeof(mdays[0])` returns the number of bytes used to store `mdays` (12×4) divided by the number of bytes used to store `mdays[0]` (4), where we are using the example of a 32-bit integer. This yields 12 for the number of elements in `mdays`, regardless of the actual size of the integer data type. The number of months could also have been computed using the expression `sizeof(month)/sizeof(month[0])` instead. It is sometimes useful to compute the size of such *parallel* declarations of arrays, supposed to be in one-to-one correspondence, and checking to ensure that there are really the correct number of elements in each array.

The `sizeof` operator can be used without parentheses in many cases, particular when applied to a single named variable, as in `sizeof nmonths`. The `sizeof` operator can be applied to a data type keyword, as in `sizeof(double)`, but the parentheses are *required* in this case. Similarly, the parentheses are required when `sizeof` is applied to an expression. Since there is no problem with a few too many balanced parentheses, it is usually safest to always use the parentheses, and treat `sizeof` as if it is a *function call*.

The names of the months are stored in memory *pointed at* by the elements of the array `month`. It is important to note that the actual character strings, such as the array of characters "November", are not stored in the array `month`, but at some location in memory decided by the C compiler. Each element of the array `month` contains the *memory address* of the relevant string. Thus we frequently refer to an array used in this manner as a *pointer array*.

The use of pointers makes some operations particularly easy. In fact, the above program is correct C, but it has an error of fact in it. Note that the months of September and October are out of order. Rather than actually exchanging the characters of the two strings "September" and "October", which is not particularly easy since they have different numbers of characters, we need only exchange the *pointers* to these two strings. The following lines are a simple way to do that exchange. We declare another pointer, called `temp`, which will be a *scratch* variable needed for the exchange. Note that the declaration of `temp` looks very much like the declaration of `month`, except that we are only allocating one such location, so there is no array involved.

```
{           /* Start a block to provide scope for a new auto variable */
char *temp = month[8];    /* save the pointer located in month[8]     */
month[8] = month[9];      /* copy the pointer in month[9] to month[8] */
month[9] = temp;          /* copy saved pointer value into month[9]   */
}           /* End block, and scope of temporary auto variable, temp  */
```

This code segment could be placed inside any block of C code where the array `month` was already defined because the only additional variable required, ie. `temp`, is defined locally within the *scope* defined by the enclosing balanced braces, as indicated in the comments in the code segment. We will discuss the concept of *scope* more completely in a later section.

The revised program to correctly print the table of months and their number of days is then:

```
#include <stdio.h>
main() {
    int mdays[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
    char *month[] = {"January","February","March","April","May","June","July",
                         "August","October","September","November","December"};
    int nmonths = sizeof(mdays)/sizeof(mdays[0]);
    int i;
    printf("This listing is in error - two months are reversed!\n");
    for( i = 0 ; i < nmonths ; ++i ) {
        printf("%-10s has %d days\n",month[i],mdays[i]);
        }

    /* We had better fix that error, the easy way! */
    {           /* Start a block to provide scope for a new auto variable */
    char *temp = month[8];    /* save the pointer located in month[8]     */
    month[8] = month[9];      /* copy the pointer in month[9] to month[8] */
    month[9] = temp;          /* copy saved pointer value into month[9]   */
    }           /* End block, and scope of temporary auto variable, temp  */

    printf("FIXED IT! The two months are now in the correct order!\n");
    for( i = 0 ; i < nmonths ; ++i ) {
        printf("%-10s has %d days\n",month[i],mdays[i]);
        }
    }
```

While this use of pointer manipulation is rather simple, it clearly illustrates the ease with which one can do what would have been a complicated task if we had only the option of actually copying each string to do the exchange.

2.9 Storage Class and Scope of Variables

Until this point, our declarations of variables have not addresses the issue of where and how long variables are stored and available. These issues are related to the *storage class* and *scope* of variables. The C language provides four storage classes: *auto, static, extern,* and *register*, denoted by the C keywords `auto, static, extern,` and `register`.

Most of our variable declarations to this point have been implicitly of storage class `auto`, recognizable by the lack of an explicit storage class keyword. Thus the declaration `int count` is a short form of `auto int count`, since `auto` is the default storage class. Any variable declared inside a block of C statements surrounded by a balanced pair {...} without a storage class specified is of storage class `auto` and also *local* to that block of C statements. Thus the balanced pair {...} defines the *scope* of that variable. This variable is unknown and undefined in any C code lying outside that *scope*.

The notion of the *scope* of a variable provides powerful features that ensure the protection of the environment in which a given block of C statements are executed. This gives a significant level of *modularization*, so that once a block of C code is developed with a well-defined scope, etc., it can be viewed as somewhat of a *black box* from the outside. This scoping of variables is recursive in the sense that within any scope, one can define substructure with its own scope, and within that scope, there is protection from the enclosing scope.

Consider the block of C code:

```
#include <stdio.h>
main() {
    int i, j, k;
    i = 12;
    j = 14;
    printf("Outer scope: I = %d, J = %d\n",i,j);
    {                        /* The inner scope for variable i starts here.   */
        int i = j*j;         /* This i is completely unrelated to the outer i */
        j = j + i;
        printf("Inner scope: I = %d, J = %d\n",i,j);
        }                    /* The end of the scope for the local variable i */
    printf("Outer scope: I = %d, J = %d\n",i,j);
    k = i + j;
    printf("Final: K = %d\n",k);
    }
```

That the two variables with name `i` are really completely unrelated is shown by the following output of this program.

```
    Outer scope: I = 12, J = 14
```

```
Inner scope: I = 196, J = 210
Outer scope: I = 12, J = 210
Final: K = 222
```

The variable `i` was initially assigned the value 12, and was unchanged when `k` was finally assigned the value `i + j`. On the other hand, `j` was initially assigned the value 14, which was used in the inner scope to assign `i` the value 14×14, so that the final value of `j` became $14 \times 14 + 14 = 210$. Thus the final of `k` was $210 + 12 = 222$ since in the outer scope, `i` retained its original value of 12.

Chapter 3

The C Language

The use of the computer for scientific problem solving requires one or more languages to communicate our procedures and algorithms in a manner usable by the computer hardware. Since it is unreasonable to require the vast majority to use a language specific to a particular machine design, we normally use a high level language that promises to be widely available on most computers. Languages that have been popular for many scientific purposes include FORTRAN, BASIC, PASCAL, and C. FORTRAN has been dominant in science and engineering until recently, when the C language has been adopted for use in many applications. While we will choose to use C in this book, there is quite a close correspondence among C, FORTRAN, and PASCAL.

3.1 Program Development

Before we begin a detailed examination of C programs, it is useful to consider the typical procedure involved in developing a simple program. Figure 3.1 shows a flow diagram of this procedure, starting with the human input using a text editor, such as `vi` or `emacs` on a computer with a `UNIX` operating system. Following the entry of a preliminary version of the C program, the C language file is submitted to the C language compiler, typically called `cc`. Usually this first compilation will result in numerous *compiler errors* and the user will use the editor to edit the C language file to correct the statements that have caused the various errors. This process of compilation and error correction will continue until the compiler ceases to complain about the C language file.

The C source that the user supplies does not constitute a complete program since it always requests services from the computer's operating system, such as access to various input/output devices, including disks, the keyboard, or the video screen. In addition, most of our programs will require the system to supply mathematical functions, such as the square root function, or trigonometric functions, that the user does not wish to rewrite each time a new program is written. These services are usually made available through *libraries* such as the mathematical library, or the basic system services library. Each of these libraries consists of a collection of *modules*, each of which performs some simple and well-defined task. Examples of such modules would include the sin function, or the `printf` function used to send nicely formatted output to the user's screen or to a disk file.

The modules defined in the various libraries each have their own structure for receiving their

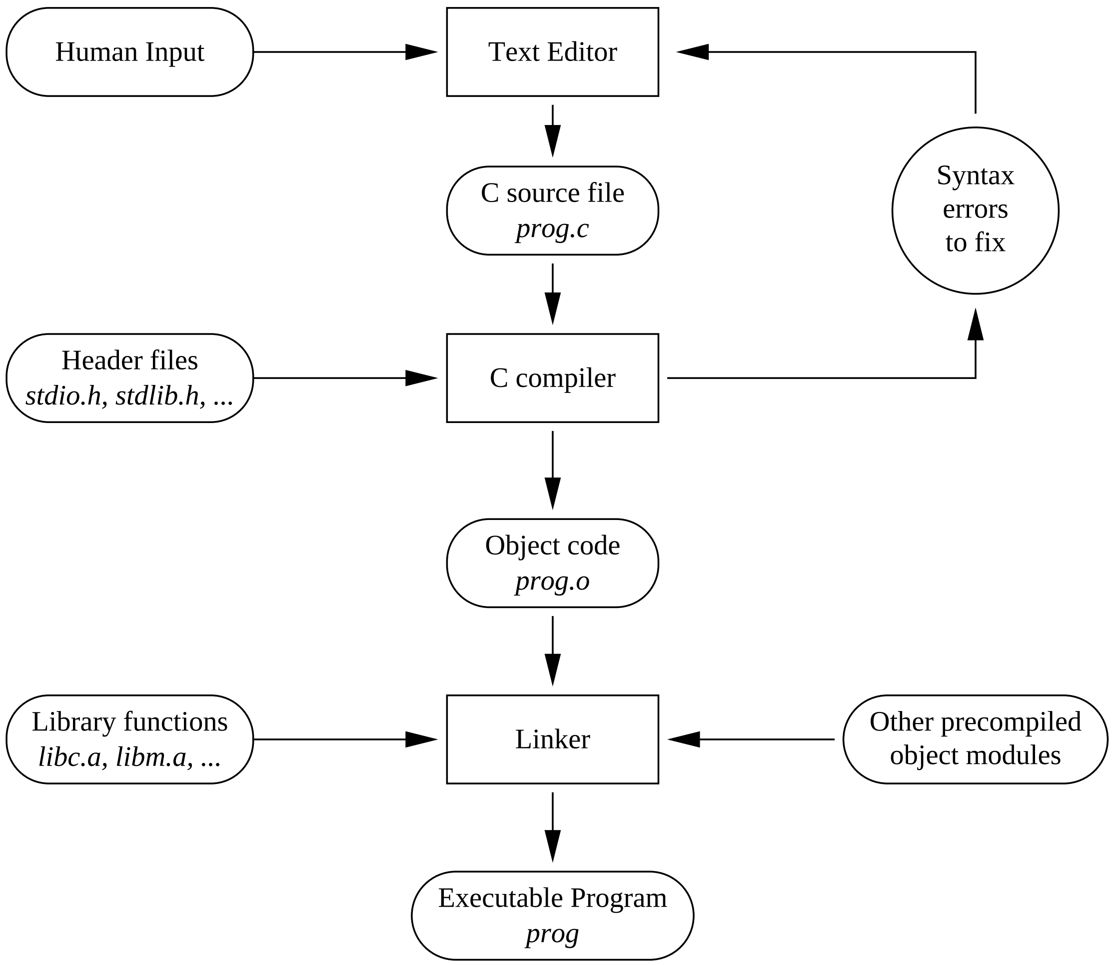

Figure 3.1: The simple program development loop for a program `prog.c`, showing the typical activities that eventually lead to a running program.

input from the user's program, and their own structure for returning their results, and success status. It is important that the C compiler have some knowledge of this structure at the time of compilation of the user's C source file, but the actual source code of each module is not required. Thus it is appropriate to capture this structure in a form that is sufficient for the C compiler at compile time, and this is accomplished through the use of *header files*. Header files are usually distinguished from C source files by having file names which end in `.h` rather than `.c`. Examples of these header files include the standard system supplied header files, such as `stdio.h`, `stdlib.h`, and `math.h`. It is useful to note that it is quite common to use the starting characters `std` for many of the standard headed files provided on almost systems. One possible confusion that can arise relates to the choice of standard header file in which to place many frequently used modules. In particular, as the C language standards have evolved, there has been some migration of modules from `math.h` to `stdlib.h`. If there is any doubt, then it is best to use the *man* facility to determine the correct header file to use, and to obtain a complete definition of the calling arguments and return value of a module. As an illustration of the use of `man`, the command:

```
man sin
```

would result in the output:

```
NAME
    sin - sine function

SYNOPSIS
    #include <math.h>
    double sin(double x);

DESCRIPTION
    The sin() function returns the sine of x, where x is given in radians.

RETURN VALUE
    The sin() function returns a value between -1 and 1.

SEE ALSO
    acos(3), asin(3), atan(3), atan2(3), cos(3), tan(3)
```

It is thus clear that the `sin` function expects its argument to be a `double`, and it will return a `double`. The inclusion of the `math.h` header file is mandatory, for otherwise the compiler will not know that the `sin` function will be returning a `double` value, and will thus allocate storage for a return value of type `int`, which is typically half the number of bits as a `double`.

3.2 A Simple C Program

We begin our study of the C language with a very simple, and immensely popular program that prints a very simple message. We will assume that the program is stored in the text file `hello.c` on the computer disk, as:

```
#include <stdio.h>                 /* Standard definitions */

main()                             /* Name of the function = main */
{                                  /* Start a block to define main */
    printf("Hello world!\n");      /* Rather simple in this case */
    }                              /* End the block defining main */
```

When compiled by the C compiler, this program will simply write the text string `Hello world!` and return the computer systems prompt. The compilation command is the line:

```
cc -o hello hello.c
```

This causes the computer to use the C compiler, normally called simply `cc`, to compile the file `hello.c` and to produce an output file `hello`, as directed by the `-o hello` part of the `cc` command. When the compilation process is complete, the user simply types `hello` followed by the ENTER key, to load and execute the newly created command `hello`.

Let us now return to the contents of the text file `hello.c`. While the text is shown in a readable format, there is considerable freedom in the use of *white-space*, ie. blanks, tab-characters, or new-lines, for the program layout. White-space inside *balanced* double-quote characters, ie. inside literal text strings, is of significance, and in the program `hello.c`, replacing the single space between `Hello` and `world` by 4 spaces, say, will alter the layout of the text printed to the screen.

Before we analyze the program in the file `hello.c`, we need to introduce the idea of *comments*, meaning descriptive text which forms no part of the C language to be compiled. The purpose of this text is to explain various points about the C program in order to make it more easily understood by the initial programmer and by those who have to maintain or enhance the program at a later time. Commentary text is delimited by the `/*` opening sequence and the `*/` closing sequence. While some C language systems are very forgiving in their handling of comments, one should always ensure that the `/* ...  */` sequences are properly balanced – very confusing errors can occur otherwise.

The first line of `hello.c`, ie. `#include <stdio.h>` is typical of a special class of statements in a C program. These are frequently called preprocessor statements or directives, since they specify some action to be taken, such as inclusion of all the lines from some other file, before the whole program file is passed to the C language compiler. This allows the easy inclusion of other C program files, frequently files containing very standard definitions of commonly used symbols, etc.

The decision that a line is a preprocessor directive is based on the appearance of the `#` character in the *first position on the line*. While the remainder of the preprocessor text may be shifted to the right by blanks or tab-characters, the `#` must not be preceded by any white-space.

In the directive `#include <stdio.h>`, the `<` and `>` characters (sometimes called angle brackets) have a special significance in the way they *bracket* the name of the file to be included. The string `<stdio.h>` implies that the preprocessor should look in a standard place or places in the operating system files for the file `stdio.h`, but if the file name `stdio.h` was enclosed in balanced double-quote characters, ie. `"stdio.h"`, then the preprocessor would only look in the local directory to find the file to be included. The `<...>` notation allows the definition of a set of *include* directories of useful files, known to the C compiler, so that the normal user never needs to know the actual location of the file to be included.

The program example, `hello.c`, contains the definition of only one function which is a named body of executable instructions. The name of this function is `main`, which must be defined in all normal user programs. In fact, the function `main` is the *entry point* into the whole program, and is the point at which the operating system initiates the execution of user-specified instructions. In the simple case at hand, the function `main` takes no arguments, which means that it does not require or use any information given by the user to the operating system, and subsequently passed to the program. The lack of any expected arguments to `main` is indicated by the empty argument list, `()`.

The body of instructions to be executed and named `main` is enclosed by a balanced {...} pair. The body of a function usually contains many statements, each terminated by a semi-colon. Except for the case of literal character strings (ie. inside double-quote characters), each statement may be broken across a number of lines, as long as the line-breaks come where any other white-space (blanks, etc.) could be placed. It is thus very important that the real end of a statement be indicated by the semi-colon, for otherwise the C compiler has no way of knowing when it has reached the end of an executable statement.

Turning now to the single executable statement in the body of the function `main`, we see that in this case it is a statement which will result in the printing of a certain character string. The statement consists of a call of the `printf` function, to print a single string. We shall soon find that the printf function has capabilities that go far beyond the printing of a single character string, including the capability to format numbers and other information. This particular call of `printf` requests the printing of the string `Hello world!\n`, where all the characters have their usual meanings, except for the special 2-character sequence `\n`. In this case, the `\` character removes the literal interpretation of the `n` character, and replaces it by a special meaning, in this case the *newline* character. The *newline* indicates that the print position should move down the page by one line and move to the far left of the print page. There are a number of such 2-character sequences, with frequently used sequences being `\r` for a *carriage return* (no implied *line-feed*), and `\t` for a *tab-character*. To produce a single \ character, it is necessary to use the special sequence \\.

3.3 The Second C Program

We now consider a somewhat more complicated C program which introduces a number of new ideas. The program `my_echo.c` echoes back to the user the input typed on the `my_echo` command line. Executing the command:

```
my_echo Ministry of Silly Walks
```

would result in the output to the screen of:

```
Program name = my_echo    Found argc = 5
Ministry
```

while executing the command

```
my_echo "Ministry of Silly Walks"
```

would result in the output to the screen of:

```
Program name = my_echo    Found argc = 2
Ministry of Silly Walks
```

The difference is caused by the fact that the first attempt passes a total of 5 arguments to the `my_echo` program (including the program name), while the use of the double-quotes around the character string results in it being passed to `my_echo` as a single argument. The difference is quite apparent in the difference in the two values reported for the argument count, `argc`. In fact, the two values, 5 and 2, initially appear quite odd, until one learns that the name of the program (in this case `my_echo`) is always considered to be the first argument, so `argc` is always at least 1.

```
/*
FILE = my_echo.c:

This is a simple echo program that echoes one character
string, when a character string is given. If the character
string consists of more than one word (as delimited by
white-space) then the string should be "bound together" by
enclosing it in double-quotes or in single-quotes.

If no arguments are given to the program, it indicates this
by responding with:

**** Nothing to echo!

*/
#include <stdio.h>

main(argc,argv)
int argc;       /* number of arguments (words) in call */
char *argv[];   /* argv[0], argv[1], ... are the arguments */
{
    printf("Program name = %s   Found argc = %d\n",argv[0],argc);
    if( argc > 1 ) {                /* Check if argc is greater than 1 */
        printf("%s\n",argv[1]);     /* There is something to print */
        }
    else {                          /* argc must have been 1 or less */
        printf("**** Nothing to echo!\n");
        }
    }
```

In analyzing the structure of the `my_echo.c` text file, we must introduce some new symbols and their meanings in a C program. The `main` function definition now includes some arguments, as indicated in the `main(argc,argv)` statement. This statement indicates that this `main` function expects two arguments to be passed to it. The types of these two arguments is indicated in the two subsequent lines. The statement `int argc;` indicates that the first argument to `main` will be treated as an integer, as opposed to a number with a fractional part. In the case at hand, this parameter is a count of the total number of words typed on the command line (including the command name itself).

The second argument to the `main` function introduces two new ideas. The first idea is quite simple - the idea of an array of character strings, as indicated by the declaration `char *argv[];`. This tells the compiler (and us!) that `main` expects to be given an array, `argv[]`, of character strings. These character strings are referred to as `argv[0]` (the name of the program), `argv[1]` (the first argument beyond the program name), etc. Since each of these array elements, such as `argv[2]`, represents a sequence of characters forming a character string, each `argv[]` must simply *point* to the location of the first character of the character string in question.

We point out that the form of the declaration of the types of the arguments to a function used above can be replaced by a slightly more recent form, adopted in the ANSI definition of the C language. The ANSI declaration for `main` would become:

```
main(int argc, char *argv[])
```

which would replace the three line form discussed above, though the interpretation remains unchanged.

The function with the special name `main` is supplied its arguments by the operating system using the text entered on the command line by the user. This allows a user to supply meaningful information to be easily passed to the program. The function `main` is the only function that will normally have arguments supplied in this way. Other functions will be passed their arguments by the C statements which call them. Thus the `main` function can be viewed as the *root* of every program the user may write and run.

The first executable statement of `my_echo.c` simply reports the name of the program, and the number of arguments passed to it. The `printf` statement is similar to that used in the `hello.c` program, but with significant additions. The first argument to `printf` consists of a string, all of which will be printed, except for the sequences beginning with `%`. These *format control strings* are very flexible, and due to the complexity of all the possible options, the reader should get full details by using the "`man fprintf`" command in UNIX. In the case at hand, the `%s` sequence directs the `printf` function to replace that `%s` by the next unused argument to the `printf` function, in this case `argv[0]`, which is the name of the program. Similarly, the `%d` directs `printf` to replace the `%d` sequence by the next unused argument to `printf`, interpreting it as a decimal integer.

Once the information about the arguments passed to the `my_echo` program is printed, the remainder of the program branches to one of two actions based on the outcome of a conditional test of the form:

`if(` *condition* `)` { *action1* } `else` { *action2* }

The statement *condition* is evaluated and if it returns a value which is *true*, then *action1* is taken, while if *condition* evalutes to *false*, then *action2* is taken, resulting in what is called a *two-way branch.* The value of *condition* is taken to be *true* if *condition* evaluates to any *non-zero* value and taken to be *false* if it evaluates to zero. Thus the sequence

`if( 37 )` { *action1* } `else` { *action2* } would always result in *action1* being taken, while

`if( 2*0 ) {` *action1* `} else {` *action2* `}`

would always result in *action2* being taken since the condition always evaluates to zero. In the simple case at hand, such a test is used to determine if there is any data worth printing (`argc > 1` means `argc` greater than 1), and if not, an informative message is printed.

Each of the actions, such as *action1*, can consist of a block of any number of C statements separated by the semicolon character in the usual way as in the example:

```
if( x > 0 ) {
    w = sqrt(x);
    v = sqrt(y + x);
    }
else {
    w = sqrt(-x);
    v = sqrt(y - x);
    }
```

This has the effect of setting `w` to be the square root of the absolute value of `x`, and `v` to be the square root of the sum of `y` and the absolute value of `x`.

While we have already included some arithmetic expressions, we need to introduce the arithmetic operations commonly used in C. For sicientific calculations, the most important arithmetic operators are listed in the table:

C arithmetic operator	Operation	unary or binary
`++`	increment	unary
`--`	decrement sign	unary
`-`	change sign	unary
`*`	multiplication	binary
`/`	division	binary
`%`	remainer	binary
`+`	addition	binary
`-`	subtraction	binary
`=`	assign value	binary
`+=`	add to value	binary
`-=`	subtract from value	binary

These are listed in their order of precedence, beginning with the highest. The operators `++`, `--`, and unary `-` are of equal priority. The operators `*`, `/`, and `%` are equal and next in priority, followed by `+` and binary `-`. The assignment operators `=`, `+=`, and `-=` have equal, and lowest pririty.

When the action consists of only one C statement, the enclosing {...} is not required, so we could write:

```
if( x >= 0 ) printf("Absolute value is %d\n",x);
else printf("Absolute value is %d\n",-x);
```

Frequently we only need to check a condition, which if satisfied requires some action and otherwise the execution should just continue. This eliminates the need for the *else* statement, as in the example:

```
if( count > maximum ) {
    printf("Maximum count exceeded: Count = %d\n",count);
    exit(1);        /* EXIT - serious difficulties! */
    }
printf("About to process %d records ...\n",count);
... more statements ...
```

In many situations we need to use a multi-way branch decision structure based on testing a sequence of conditions. This can be done by simply stringing together a number of *if ... else* statements in the general form:

```
if( condition1 ) { action1 }          /* first condition/action pair    */
else if ( condition2 ) { action2 }    /* second condition/action pair   */
else if ( condition3 ) { action3 }    /* third condition/action pair    */
else { action4 }                      /* default if all conditions fail */
```

The multi-way branch must be distinguished from the *nested* `if` statements where the intention is to have conditional branching inside a block of C statements comprising the action of a higher level conditional branch structure. Consider the example:

```
if( y > 0 ) {
    if( x > 0 ) root = sqrt(y/x);
    else if( x < 0 ) root = sqrt(-y/x);
    else root = -1;
    }
else if( y < 0 ) {
    if( x < 0 ) root = sqrt(y/x);
    else if( x > 0 ) root = sqrt(-y/x);
    else root = -1;
    }
else root = 0;
```

This returns the positive square root of the absolute value of y/x unless $x = 0$, when it returns the negative value -1. Of course it is not the easiest way to get this result, but it does illustrate the use of nested multi-way branching. This nesting requires a clear understanding of the scope implied for each block of C statements by the matching $\{\ldots\}$ pairs.

We have used a few *relational operators*, such as `<` or `>`, in our examples so far. These are members of a set of relational operators that can be used in C programs which are summarised in the following table:

C notation for operator	Relation
`<`	less than
`>`	greater than
`<=`	less than or equal
`>=`	greater than or equal
`==`	equal
`!=`	not equal

The use of the `!` character to indicate the *negation* of a relation is general in C, but it is important to note that `!` is a *unary* operator (operates on a single entity) in all cases. In some circumstances, the arithmetic minus sign, $-$, acts as a *unary* operator, as in the expression `exp(-1.3)`, but the interpretation of $-$ depends on the context. In the expression `exp(x - y)`, it is clear that the $-$ acts as a *binary* operator (note that there is no *unary* $+$ in C). The case of the *binary* operator `!=` uses this sense of negation, but specific to the test for inequality. Thus it would not be correct to use a binary operator in the form `!>` to mean *not greater than*, but of course, the `<=` operator does mean *not greater than*. It is important to note that the *equal to* relational operator is the 2-character string "==" and not the single character "=". The single "=" character will always mean *value assignment* in C. Thus the result of the statements:

```
x = 5;
k = x;
```

is to give both `x` and `k` the value 5, while the result of the statements:

```
x = 5;
k = (x == 3);
```

is to give `x` the value 5 and `k` the value 0, since the relational test `x == 3` is false (`x` is 5, not 3). Since an expression like `x == 3` returns a value based on the result of the test, this value could be assigned to `k` in this example. Thus the result of the statement `k = (x < 10)` would be to set `k` to one, while `k = (x >= 7)` would assign `k` the value zero.

It is often useful to use multiple conditional tests to reach a branching decision. This is facilitated by the availability of the *logical operators* given in the table:

C notation for operator	Relation	Binary or Unary
`&&`	logical AND	Binary
`||`	logical OR	Binary
`!`	logical NOT	Unary

Again it is important to note that the 2-character strings "`&&`" and "`||`" are quite distinct from the 1-character strings "`&`" and "`|`". The 1-character strings represent *bit-wise* operations, and are not used that frequently in the kind of computing applications considered here.

The following C sequence of C statements uses compound tests to handle the conditions under which the square root of y/x can be computed and returns -1 when it cannot be computed with real numbers.

```
if( ( y >= 0 && x > 0 ) || ( y <= 0 && x < 0 ) ) {
    root = sqrt(y/x);
```

```
        }
    else {
        root = -1;
        }
```

3.4 Simple Input/Output

For most scientific programming purposes, friendly interaction with the program through well designed and flexible input/output mechanisms is of some importance. The next example of C programming in the file `hypot.c` calculates the hypotenuse of a right angle triangle, given the other two sides. The program has been designed to allow the user to give a simple command like:

```
hypot 3 4
```

which will print the well-known result, 5, or to allow the user to simply start the program as:

```
hypot
```

whereupon the program will prompt the user for the appropriate input in the form:

```
Enter X and Y:
```

to which the user replies with the values of X and Y, the two sides opposite the hypotenuse.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

main(int argc, char *argv[])
{
    float x, y, h;
    int count;
    if( argc == 3 ) {
        x = atof(argv[1]);
        y = atof(argv[2]);
        }
    else {
        fprintf(stderr,"Enter X and Y: ");
        count = fscanf(stdin,"%f%f",&x,&y);
        if( count != 2 ) {
            fprintf(stdout,"Invalid or incomplete data entered - exit!\n");
            return(1);
            }
        }
    h = sqrt(x*x + y*y);
    fprintf(stdout,"The hypotenuse on %f and %f is %f\n",x,y,h);
    return(0);
    }
```

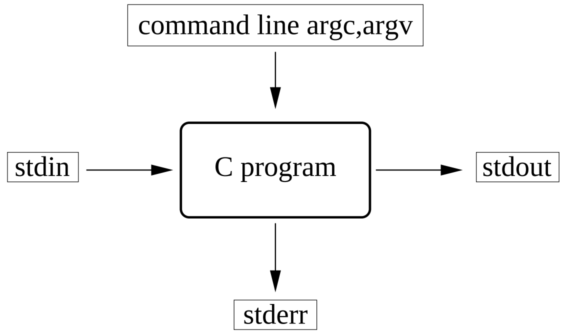

Figure 3.2: The simple 3-stream view of a process, consisting of one input stream, `stdin`, and two output streams, tt stdout for useful and normal output, and `stderr` for error messages or status reports that should not usually be part of the *pipe* structures. Note that one additional source of input is shown – the command line typed to invoke the program. This input is available to the C program through the arguments to the `main` function definition discussed in the text.

This program introduces a very useful, but simple, approach to supplying input data to a program and getting output from a program. The model being used here is that of a `filter` in a simple picture of a useful process that takes input into some standard receptacle labelled *input* and sends the processed data out some standard receptacle labelled *output*. The standard names adopted for these receptacles or connectors (like the input and output connectors in a home stereo system or a VCR) are `stdin` and `stdout`. If all goes well, the programs we write will usually accept their input data and produce valid output data, but sometimes things go wrong, and it is necessary to have another output channel through which to send error messages or other special messages to the user. While the messages may not always be *error messages*, we usually refer to this second output channel as the `stderr` stream of output. One frequent use of `stderr`, other than error messages, is for prompting the user for input parameters, when we do not want the text of the prompt to the user to appear interleaved with the valid output of the program that is going to `stdout`. The file `stdio.h` contains the definitions of `stdin`, `stdout`, and `stderr`, and these three streams are opened by the operating system when it starts the program.

The function `printf` is already familiar for simple output, and we now have an example of a related function, `fprintf`. In fact the function `printf` is just a synonym for the `fprintf` function with its first argument set to be `stdout`. In the `hypot.c` file the `fprintf` function is invoked with `stderr` as its first argument, so that the prompt to the user for input of `x` and `y` appears on the user's display, but does not go to `stdout`. This distinction between `stderr` and `stdout` is not obvious when the program is invoked from a terminal and the output is viewed on the terminal, but we shall see below that the *redirection* of input and output can make the distinction important and very useful.

The `fscanf` function is invoked to get the values of `x` and `y` when the user is prompted for input. This function reads input from `stdin`, and a synonym for this more general function is `scanf`, when the first argument is `stdin`. The first argument to `fscanf` is the input stream from which to read the data (thus `scanf` is to `fscanf` what `printf` is to `fprintf`). In a manner similar to the `fprintf` function, `fscanf` takes a format string which describes the types of data to be read using the % template notation, like `fprintf`. There is one very important distinction to be made between the output process (`fprintf`) and the input process (`fscanf`). In the output process, one specifies **what** (ie. value) to print, while for the input process, one can only specify **where** (ie. location in memory) to store the value read from the input. This distinction must be made clear to the C compiler, and this distinction is shown in the notation illustrated in `hypot.c`. Consider the variable `x`, declared to be of type `float`. When we deal with the value of `x`, we use the simple name for `x`, ie `x` stands for the *value of* `x`. When we deal with the memory location where the *value of* `x` *is stored*, we use the notation `&x` to refer to the *location* or *address* of `x`. Thus if we wish to assign a new value to the `float` variable `x`, its previous value is of little interest to us, but the address of the piece of memory in which the value of `x` is stored is what we need.

The files included at the beginning of the file `hypot.c` now include two more header files, `stdlib.h` and `math.h`, both from the standard include directories. The first of these includes definitions of the type of values returned by many functions – in our case it is the function `atof` that is of interest. Similarly, `math.h` includes information about the type of values returned by many mathematical functions – in our case it is the square root function `sqrt` that is of interest.

This program decides on the input mode the user is using by testing the value of the argument count, `argc`. If the user enters the program name and two values on the command line, the number of arguments is 3, and the program can compute the hypotenuse directly. If the number of arguments is not 3, the program prompts the user for the values of `x` and `y` using the `scanf` statement.

3.5 Input/Output Redirection – or Virtual Plumbing

The filter model for the design of many useful programs becomes a very powerful facility when we build composite filters by connecting the output stream of one filter program to the input stream of another filter program. This facility is provided by the operating system by *I/O redirection*. An analogy with simple plumbing is useful with the operating system providing a small set of connectors. The first of these is a simple straight connector which allows the feeding of the output from one filter into the input of another filter, and is denoted by:

```
process_1 | process_2
```

so that `process_1` provides its output (via `stdout`) as the input (via `stdin`) to `process_2`. The symbol `|` is called a *pipe* for obvious reasons.

The input data for a filter will frequently come from a data file already existing on the computer, so it is useful to provide a mechanism for feeding the contents of a file into a filter process, so that the `stdin` stream is fed from the file rather than from the keyboard of the user's terminal. The notation that sets up such a connection takes the form:

```
process < file_of_data
```

where the `<` indicates that the `stdin` stream for `process` should consist of the contents of `file_of_data`. A similar notation exists for redirecting the tt stdout stream from a process, indicated by the use of the `>` symbol instead of the `<` symbol.

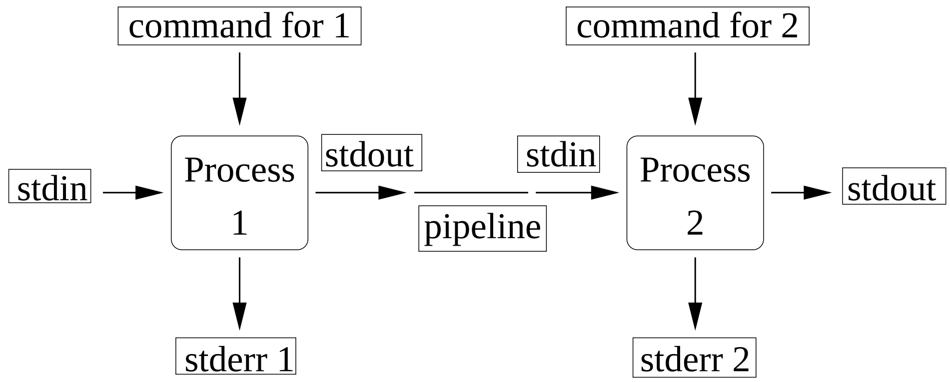

Figure 3.3: The *piping* together of processes by connecting the `stdout` output stream of Process 1 to the `stdin` input stream of Process 2. The `stderr` output streams of Process 1 and Process 2 remain distinct so that error messages from the process can handled differently. Similarly, the command lines for each process in the pipeline are distinct so that each process can be controlled. This type of structure can be extended to as many processes in the sequential stream as are required.

As an example that we might find useful in managing our own files and directories, we might want to know which our subdirectories of files are the *disk hogs*. As simple way to do this is the *stream* of processes:

```
du -k /u/my_name | sort -n > hogs_list
```

The utility `du` computes the disk-usage for each subdirectory in the directory listed as its argument, in this case `my_name`. The `-k` switch simply asks `du` to give the directory sizes in *kilobytes* (ie. 1024 bytes), since some systems report the sizes in units of 512 bytes instead. The output from `du` goes to `stdout`, which is connected by default to our video screen. Here the `stdout` stream from `du` has been *piped* to the `stdin` of another program, `sort`, instead. The general purpose sorting program, `sort`, has many options specified by switches, but here we only use the switch `-n` which indicates that the sort should be numerical rather than alphabetic. The `stdout` output from `sort` is redirected by the `>` operator into the *file* `hogs_list` (where the choice of name is, of course, arbitrary). This file will contain a list of all your subdirectory sizes (in kilobytes), sorted numerically so that the largest subdirectory will be at the end.

The example illustrates the process of *pasting* together standardly available processes to build a more complex process as required by the user, but without the user having to write a complex program specifically designed for the task. The ease of doing this in *UNIX* and its derivatives (such as MS-DOS) is one of its most significant advantages, leading to much more rapid development of software. One aspect of this approach is that the *modular* design this allows results in fewer programming errors, since the persons responsible for the design of the `sort` program, for example, are free to concentrate their analysis on this task alone, and not all the uses humans may find for their module. Thus other programmers do not have to "re-invent" the program for sorting each time they need to sort data. The *redirection* mechanism allows many more complex possibilities, but the simple use of `|`, `<`, and `>` described here will handle most common situations.

3.6 Data Types Available in C

We have discussed the basic data-types available on modern computers in an earlier chapter, so we now turn to invetigating how these are made available in the C language. The simplest data-types are what are usually called *scalars*, which can be used to represent the basic building blocks for more complicated composite data-types. Integers, restricted to be small enough to fit in one computer word (say 32 or 64 bits), are the most basic data types beyond bits themselves. In C one can select from a limited number of classes of integers, based on the number of bits used to represent them. In the previous examples of C programs, we have seen a few examples of integers being *declared* as `int` as in:

```
int x, y;      /* Declare both x and y to be signed integers */
```

which tells the C compiler that `x` and `y` are to be taken to be the *natural* integer for that class of computer hardware. Normally this would a 32-bit signed integer, meaning that 31 bits are used for the binary representation of the absolute value of the integer, with one bit being reserved to represent the sign of the integer.

The *natural* integer for a particular computer system is fixed by the hardware manufacturer and does vary across the range of hardware commonly available. At present the natural word size on most computers is one of 16 bits, 32 bits, or 64 bits, with the typical word size becoming larger in the more recent machines. As a consequence of this variation in word size, C allows a more precise specification of the size characteristics of an integer through the use of additional data-types. In the declarations:

```
int x, y;      /* Declare x and y to be signed integers */
short s;       /* Integer s is a 16-bit signed integer  */
long t;        /* Integer t is a 32-bit signed integer  */
```

`x` and `y` are the natural integers for the machine, while `s` is declared to behave exactly as a 16-bit signed integer in all its operations, and `t` is declared to behave exactly as a 32-bit signed integer in all its operations.

In real applications, it is important to understand that declaring a variable as `int` should be thought of as declaring it to behave like a 16-bit signed integer, but implemented on the particular hardware in the most efficient way for that hardware. The declaration of the variable as `short` makes that behaviour explicit, but may result in some inefficiency on hardware with a natural word size of larger than 16 bits. Consequently it always best to restrict the use of the `int` declaration to those variables which you know will always fit within the `short` size limits, but where you want to leave the compiler free to implement the program in the most efficient way for that hardware.

The `long` declaration should be used when integers that can exceed the `short` limitations, regardless of the hardware word size, since we should worry first about the correctness of the calculation and then about the possible efficiency enhancements.

In many applications, there are integers that will always be non-negative, so that it is useful to specify that some variables should always be considered to have this property. One simple advantage is that the use of what would have been the sign-bit for a mantissa bit doubles the range of integer values that can be represented. More importantly, this added information can sometimes be used by the compiler to detect inconsistent use of the variable or functions involving this variable. The C keyword `unsigned` is prepended to the declaration in this case:

```
unsigned int x, y;       /* Declare x and y to be unsigned integers */
unsigned short s;        /* Integer s is a 16-bit unsigned integer  */
unsigned long t;         /* Integer t is a 32-bit unsigned integer  */
```

We have already encountered the `char` data-type in our earlier discussion of *character strings* which we typicall use to represent text. The `char` data-type normally comprises 8 data bits, and can be declared as `unsigned` if required. This data-type is usually too small to store the integers we would use in most programs, but it is useful to store *flags* or other special status information which is by definition of a very limited dynamic range, ie. in the range [0,255]. In situations where there are only a small number of flags to be stored, there no advantage to the use of `char` variables, but when it is necessary to store large arrays of flags, then the use of `char` variables can lower the storage requirements.

The *floating point* representation of a subset of the rational numbers was discussed previously. We saw that useful sizes for floating point numbers are 32 bits and 64 bits, with extended precision to 128 bits now becoming more available. The C declarations for such variables follows the pattern that we have seen for integers, taking the form:

```
float x, y;     /* Declare x and y as 32-bit floating point numbers */
double u, w;    /* Declare u and w as 64-bit floating point numbers */
```

3.7 Arrays in C

In most applications, there is a need to store *arrays* of C variables of identical data-type, and this is made available in C through the use of variables declared using notation illustrated by the following example:

```
float xdat[100];
float ydat[100];
```

These statements indicate that there should be 100 sequential locations in memory set aside for the array `xdat`, and similarly for `ydat`. It is important that the 100 `float` locations be sequential, and this is guaranteed to be the case. Each of these sequential locations can be treated as a normal `float` variable in arithmetic expressions, etc.

The general form of an array variable declaration is then:

```
data_type_keyword var_name[number_of_elements];
```

It is important to note that the `number_of_elements` value must be completely known at the time that the code is given to the C compiler. Thus the declarations:

```
int size;
...
double shift[2*size + 10];
float epsilon[3*size];
```

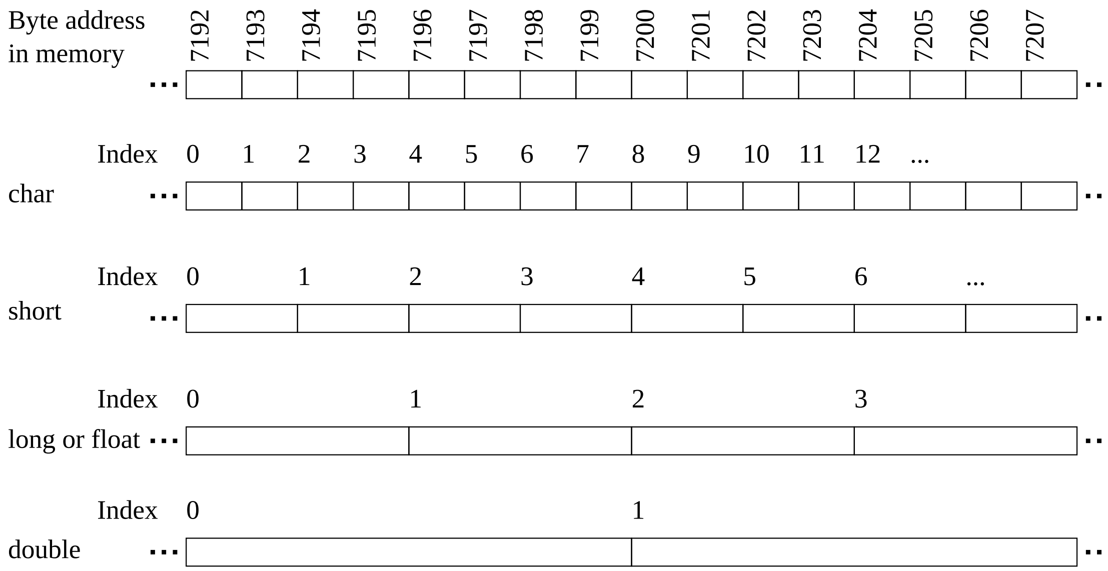

Figure 3.4: Layout of various data types in memory, showing the normal alignment of each data type, so that for a data type of size N bytes, each element starts on a multiple of N.

would not be valid, since the number of elements in each case is variable, and cannot be known at compile time. On the other hand, the declarations:

```
#define SIZE 50
      ...
      double shift[2*SIZE + 10];
      float epsilon[3*SIZE];
```

are acceptable, since the value of the symbol `SIZE` is known at compile time, due to the definition of `SIZE` in the line:

```
#define SIZE 50
```

This definition of `SIZE` is used *before* the C code is passed to the C compiler, since the C code is first handled by the `C preprocessor`, where all lines that begin with # in the first column are acted upon.

Each *element* of array is referred to by its *index* in the format illustrated in the statements:

```
#define SIZE 100
      float xdat[SIZE];            /* allocate memory for SIZE floats */
      float ydat[SIZE];
      int i;                       /* integer to use as an array index */
      float scale = 2.337;         /* declare and initialize variable scale */
      float shift = -1.4597;       /* declare and initialize variable shift */
      i = 0;
      while( i < SIZE ) {          /* initialize the xdat array values */
          xdat[i] = i*i;
          i = i + 1;               /* increment the array indexing variable */
```

```
    }
for( i = 0 ; i < SIZE ; i = i + 1 ) {          /* initialize ydat array */
    ydat[i] = scale*xdat[i] + shift;
    }
```

Presuming that each element of the `xdat` array has a value, this C code sets each value of the array `ydat` to a scaled and shifted copy of the corresponding element of the `xdat` array. This example illustrates two new *control structures* in C – the *while loop* and the *for loop*.

The structure of the `while` loop is:

```
while( condition ) { statements; }
```

The *statements* are executed as long as the *condition* continues to be logically true. If the *condition* is not initially true, then the *statements* will not be executed at all. If the *condition* is initially true, then there must be some statement in the block of *statements* which will change the status of the condition to false – otherwise the `while` loop will never terminate!

The `for` control structure is essentially a concise way to implement a control structure that we could already construct based on the `while` and `if` statements. The structure can be written as:

```
for( initialize_statements ; continue_condition ; step_statements ) {
    statement_body;
    }
```

This is best understood by writing it in the form:

```
initialize_statements;
while( continue_condition ) {
    statement_body;
    step_statements;
    }
```

3.8 Initialization of Array Variables

There are many situations where it is useful to be able to not only allocate memory for an array variable, but to place specific values in each of the elements of that array. Consider an application which requires knowing the number of days in each month of the year. The following program is intended to print a table of the number of days in each month.

```
#include <stdio.h>
main() {
    int mdays[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
    char *month[] = {"January","February","March","April","May","June","July",
                     "August","October","September","November","December"};
    int nmonths = sizeof(mdays)/sizeof(mdays[0]);
    int i;
    for( i = 0 ; i < nmonths ; ++i ) {
```

```
        printf("%-10s has %d days\n",month[i],mdays[i]);
        }
    }
```

The output takes the form:

```
    January    has 31 days
    February   has 28 days
    March      has 31 days
    April      has 30 days
    May        has 31 days
    June       has 30 days
    July       has 31 days
    August     has 31 days
    October    has 30 days
    September  has 31 days
    November   has 30 days
    December   has 31 days
```

Each of the arrays `mdays` and `month` has 12 elements, though of very different data types. The size specification of 12 for `mdays` is actually redundant, since the C compiler is presented with explicit data to place in each successive element of `mdays`. In fact, when an array is going to be initialized with explicit data, as in this case, it best *not* to specify the size of the array, but simply leave it empty, as done here for the declaration of the array `month`.

In fact, there is no need to ever specify a size of 12 in the program, since we can calculate the number of data elements in an array through the careful use of the `sizeof` operator. This operator returns the size of the argument in bytes, so that `sizeof float` returns 4, `sizeof char` returns 1, and `sizeof "specification"` returns 14, since the string "specification" contains 13 real characters, followed by the termination character, the *null* character, '\0'. Thus the expression `sizeof(mdays)/sizeof(mdays[0])` returns the number of bytes used to store `mdays` (12×4) divided by the number of bytes used to store `mdays[0]` (4), where we are using the example of a 32-bit integer. This yields 12 for the number of elements in `mdays`, regardless of the actual size of the integer data type. The number of months could also have been computed using the expression `sizeof(month)/sizeof(month[0])` instead. It is sometimes useful to compute the size of such *parallel* declarations of arrays, supposed to be in one-to-one correspondence, and checking to ensure that there are really the correct number of elements in each array.

The `sizeof` operator can be used without parentheses in many cases, particular when applied to a single named variable, as in `sizeof nmonths`. The `sizeof` operator can be applied to a data type keyword, as in `sizeof(double)`, but the parentheses are *required* in this case. Similarly, the parentheses are required when `sizeof` is applied to an expression. Since there is no problem with a few too many balanced parentheses, it is usually safest to always use the parentheses, and treat `sizeof` as if it is a *function call.*

The names of the months are stored in memory *pointed at* by the elements of the array `month`. It is important to note that the actual character strings, such as the array of characters "November", are not stored in the array `month`, but at some location in memory decided by

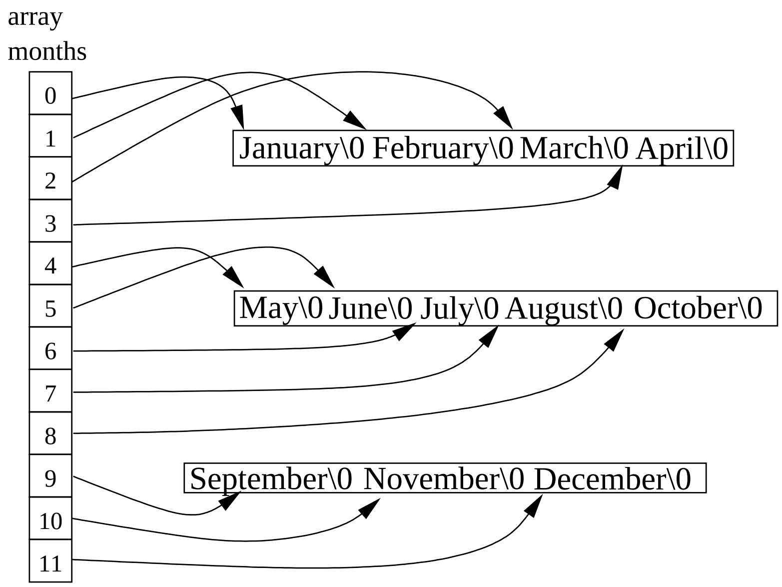

Figure 3.5: The use of a pointer array, `month`, to access the strings representing the names of the months, each stored somewhere in memory. There is no requirement that these 12 character strings be stored anywhere near each other in memory.

the C compiler. Each element of the array `month` contains the *memory address* of the relevant string. Thus we frequently refer to an array used in this manner as a *pointer array*.

The use of pointers makes some operations particularly easy. In fact, the above program is correct C, but it has an error of fact in it. Note that the months of September and October are out of order. Rather than actually exchanging the characters of the two strings "September" and "October", which is not particularly easy since they have different numbers of characters, we need only exchange the *pointers* to these two strings. The following lines are a simple way to do that exchange. We declare another pointer, called `temp`, which will be a *scratch* variable needed for the exchange. Note that the declaration of `temp` looks very much like the declaration of `month`, except that we are only allocating one such location, so there is no array involved.

```
{           /* Start a block to provide scope for a new auto variable */
char *temp = month[8];    /* save the pointer located in month[8]     */
month[8] = month[9];      /* copy the pointer in month[9] to month[8] */
month[9] = temp;          /* copy saved pointer value into month[9]   */
}           /* End block, and scope of temporary auto variable, temp  */
```

This code segment could be placed inside any block of C code where the array `month` was already defined because the only additional variable required, ie. `temp`, is defined locally within the *scope* defined by the enclosing balanced braces, as indicated in the comments in the code segment. We will discuss the concept of *scope* more completely in a later section.

The revised program to correctly print the table of months and their number of days is then:

```
#include <stdio.h>
main() {
```

```
int mdays[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
char *month[] = {"January","February","March","April","May","June","July",
                     "August","October","September","November","December"};
int nmonths = sizeof(mdays)/sizeof(mdays[0]);
int i;
printf("This listing is in error - two months are reversed!\n");
for( i = 0 ; i < nmonths ; ++i ) {
    printf("%-10s has %d days\n",month[i],mdays[i]);
    }

/* We had better fix that error, the easy way! */
{           /* Start a block to provide scope for a new auto variable */
char *temp = month[8];    /* save the pointer located in month[8]     */
month[8] = month[9];      /* copy the pointer in month[9] to month[8] */
month[9] = temp;          /* copy saved pointer value into month[9]   */
}           /* End block, and scope of temporary auto variable, temp  */

printf("FIXED IT! The two months are now in the correct order!\n");
for( i = 0 ; i < nmonths ; ++i ) {
    printf("%-10s has %d days\n",month[i],mdays[i]);
    }
}
```

While this use of pointer manipulation is rather simple, it clearly illustrates the ease with which one can do what would have been a complicated task if we had only the option of actually copying each string to do the exchange.

3.9 Storage Class and Scope of Variables

Until this point, our declarations of variables have not addresses the issue of where and how long variables are stored and available. These issues are related to the *storage class* and *scope* of variables. The C language provides four storage classes: *auto, static, extern,* and *register*, denoted by the C keywords `auto, static, extern,` and `register`.

Most of our variable declarations to this point have been implicitly of storage class `auto`, recognizable by the lack of an explicit storage class keyword. Thus the declaration `int count` is a short form of `auto int count`, since `auto` is the default storage class. Any variable declared inside a block of C statements surrounded by a balanced pair $\{\ldots\}$ without a storage class specified is of storage class `auto` and also *local* to that block of C statements. Thus the balanced pair $\{\ldots\}$ defines the *scope* of that variable. This variable is unknown and undefined in any C code lying outside that *scope*.

The notion of the *scope* of a variable provides powerful features that ensure the protection of the environment in which a given block of C statements are executed. This gives a significant level of *modularization*, so that once a block of C code is developed with a well-defined scope, etc., it can be viewed as somewhat of a *black box* from the outside. This scoping of variables is recursive in the sense that within any scope, one can define substructure with its own scope, and within that scope, there is protection from the enclosing scope.

Consider the block of C code:

```
#include <stdio.h>
main() {
    int i, j, k;
    i = 12;
    j = 14;
    printf("Outer scope: I = %d, J = %d\n",i,j);
    {                        /* The inner scope for variable i starts here.   */
        int i = j*j;         /* This i is completely unrelated to the outer i */
        j = j + i;
        printf("Inner scope: I = %d, J = %d\n",i,j);
        }                    /* The end of the scope for the local variable i */
    printf("Outer scope: I = %d, J = %d\n",i,j);
    k = i + j;
    printf("Final: K = %d\n",k);
    }
```

That the two variables with name `i` are really completely unrelated is shown by the following output of this program.

```
Outer scope: I = 12, J = 14
Inner scope: I = 196, J = 210
Outer scope: I = 12, J = 210
Final: K = 222
```

The variable `i` was initially assigned the value 12, and was unchanged when `k` was finally assigned the value `i + j`. On the other hand, `j` was initially assigned the value 14, which was used in the inner scope to assign `i` the value 14×14, so that the final value of `j` became $14 \times 14 + 14 = 210$. Thus the final of `k` was $210 + 12 = 222$ since in the outer scope, `i` retained its original value of 12.