1st Class

2016-2017

Fundamentals of Programming Techniques

أساسيات تقنيات البرمجة

أستاذة المادة : أ.م.د. يسرى حسين

Classification of Programming Techniques

A program was defined as a sequence of logical instructions that should be followed by a computer to solve some given problem. Most of the programs would consist of data definition and procedure calls.

In general, Programming Techniques could be classified into the following categories:

1) Unstructured programming.

2) Procedural programming.

3) Modular programming.

4) Object-oriented programming.

1. Unstructured Programming

Usually, people start to learn a programming language by writing small and simple programs that consist mainly of one main program. Here the term “main program” stands for a sequence of commands or statements, which modify global data, throughout the whole program. Figure 1 illustrates this concept.

Unstructured programming techniques provide tremendous disadvantages once a program gets sufficiently large. For example, if the same statement sequence is needed at different locations

within the program, the same sequence of statements must be repeatedly copied to the new locations.

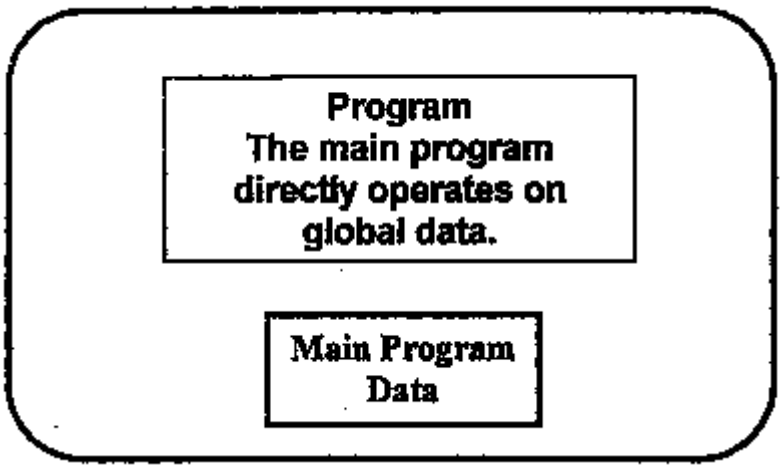


Figure1. Unstructured programming.

2. Procedural Programming

The natural improvement to the unstructured programming technique was the introduction of the Procedural Programming technique. In such a technique, a sequence of instructions used to perform a specific task are grouped together and then called a Program. The main program directly operates on global data needed. The grouping of such instructions is called a Procedure. After the call to the procedure and the execution of statements within the procedure, the program control would return back to. The statement in the main program that directly follows the calling statement, Figure2 illustrates this.

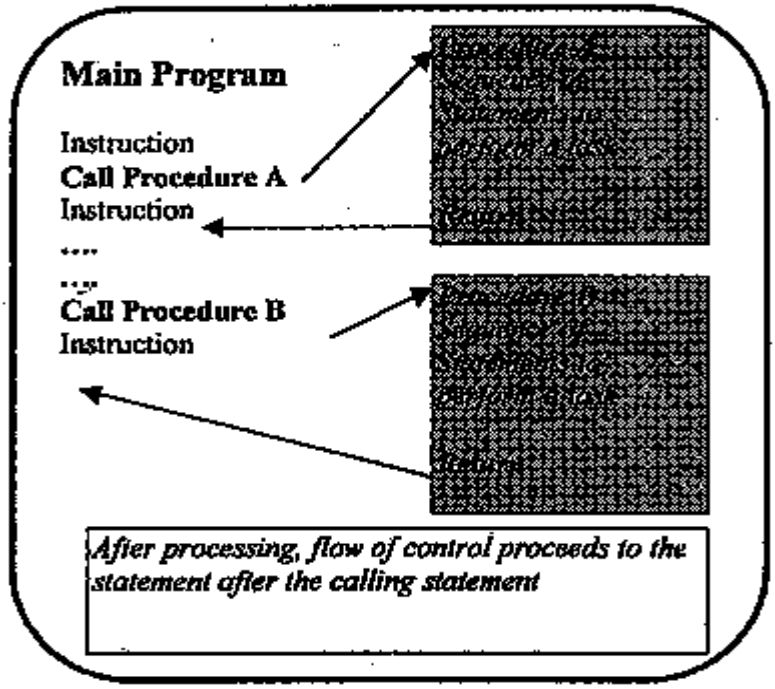


Figure 2. Procedural Programming.

With the introduction of parameters as well as procedures of procedures (sub-procedures) in the Procedural Programming technique, more structured and error free programs could be written. Maintaining such programs became easier as well.

In procedural programming, a program can be viewed as a sequence of procedure calls. The main program is responsible for passing data through each individual call to the corresponding procedure, the data is then processed by that procedure and, once the procedure execution finishes, the result is presented and passed back to the main program. Thus, the flow of data in such technique can be illustrated as a hierarchical graph like a tree, as shown in figure 3 for a program with no sub-procedures:

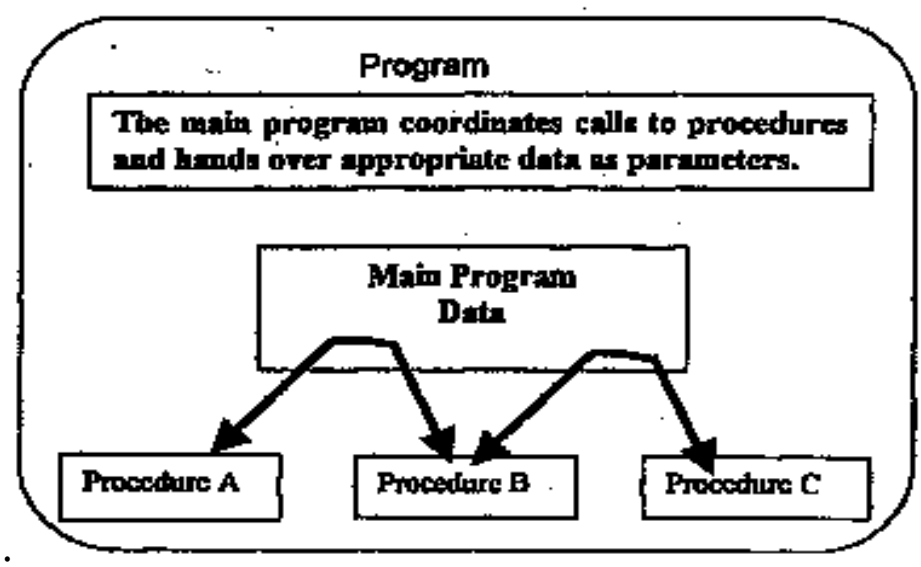


Figure3. Procedural programming

3 Modular Programming

In procedural programming, a single program is divided into smaller pieces called procedures. To enable the usage of general procedures, or groups of procedures, in other programs, such procedures must be separately available. Modular programming allows the grouping of procedures into what is known as modules.

With modular programming, procedures of common functionality are grouped together into separate modules. A program therefore no longer consists of only one single part rather is now divided into several smaller parts which interact with each other through procedure calls.

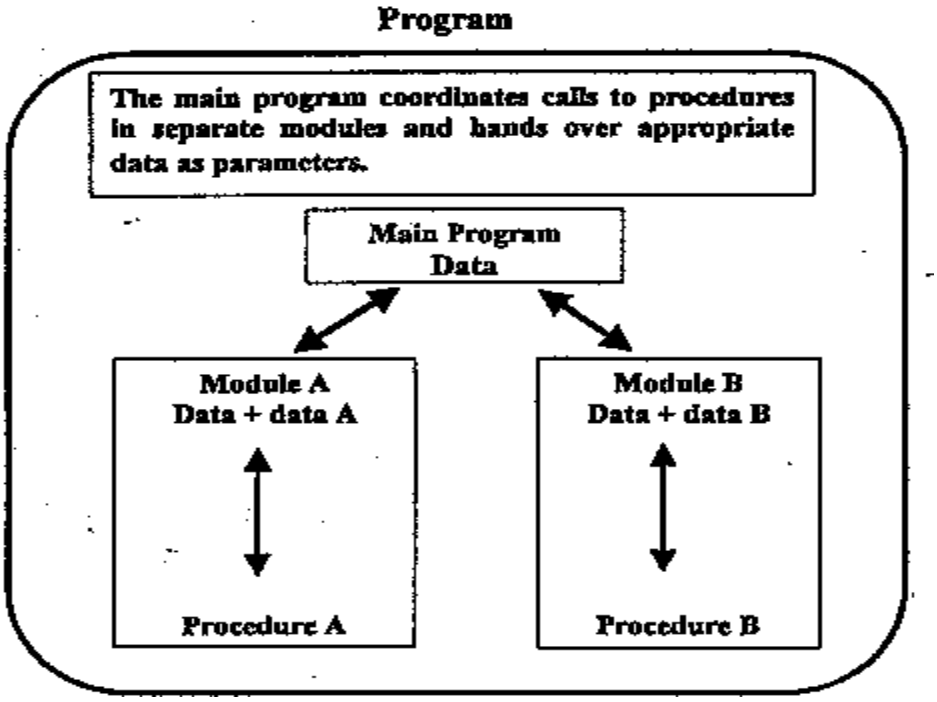


Figure 4. Modular programming.

Each module can have its own data. This allows each - module to manage an internal state that is modified by calls to procedures of this module. However, there is only one state per module and each module exists at most once in the whole program.

4 Object-Oriented Programming

Due to the problems rose with the former techniques of programming; especially, when programs get larger, the object-oriented programming was introduced as a new programming technique. Many object-oriented programming languages were Implemented and introduced for this technique; however, C++ was

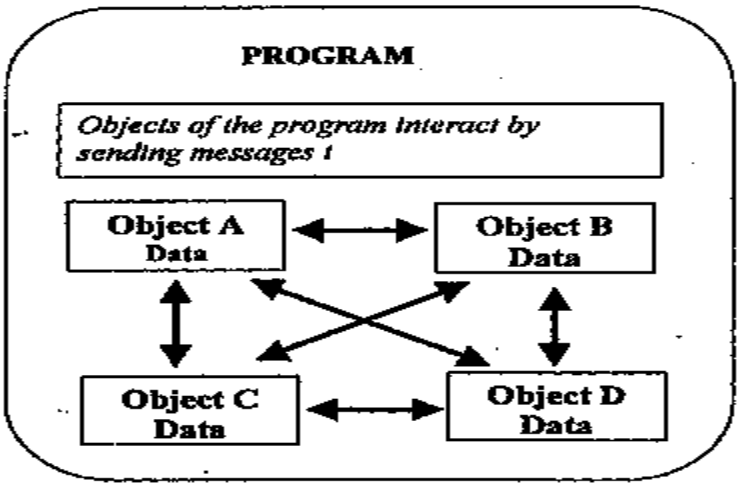
one of the most powerful true object-oriented languages that were introduced.

All object-oriented languages try to accomplish three things as a way of thwarting the problems inherent in large projects:

1) Object-oriented programming languages implement "data abstraction" in a clean way using the concept of "classes". So that all characteristics (attributes) and behaviors (functions) and their details are hidden within the class.

2) Object-oriented programming languages make parts of the programs easily reusable and extensible. Providing this feature usually reduces the development time required, especially, for large projects for which existing code can be inserted C into new projects easily.

3) Object-oriented programming languages provide unique and a very powerful concept to make existing code easily modifiable



Programming languages

A programming language allows a programmer or end user to develop the sets of instruction that constitute a computer program. Many different programming languages have been developed, each with its own unique vocabulary, grammar, and users. Programming languages can be grouped into the following categories as:-

1) Machine languages (or first generation languages) are the most basic level of programming languages. In the early stages of computer development, all program instructions had to be written using binary codes unique to each computer. This type of programming involves the difficult task of writing instruction in the form of strings of binary digits (ones and zeros) or other number systems. Programmers must have a detailed knowledge of the internal operations of the specific type of CPU they are using .They must write long series of detailed instructions to accomplish even simple processing tasks.

2) Assembler languages (or second generation languages) are the next level of programming languages. They were developed to reduce the difficulties in writing machine language programs. The use of assembler language requires language translator program called assembler that allows a computer to convert the instructions of such languages into machine instructions. Assembler languages

are frequently called symbolic languages because symbols are used to represent operation codes and storage locations

3) High level language (or third generation languages) uses instructions, which are called statements, that closely resemble language or standard notation of mathematics. Individual high level language statement are actually macroinstructions, that is ,each individual statement generates several machine instructions when translated into machine language by high level language translator.

4) Forth generation languages most forth generation languages are non procedural languages that encourage users and programmers to specify the result they want, while the computer determines the sequence if instructions that will accomplish those result. Users and programmers no longer have to spend a lot of time developing the sequence of instructions the computer must follow to achieve a result. Thus, 4GLs have helped simplify the programming process. Natural languages are 4GLs that are very close to English or other human languages.

Example

Machine language		High level language
1010	11001	Basic
1011	11010	Z=5 , Y=3
1100	11011	X=Y+Z

Assembler languages	Forth generation languages
LOD Y	Sum the following numbers
ADD Z	Z AND Y
STR X	put the result in X

5) Object oriented languages

Object-oriented programming languages are among the newest types of programming languages. Instead of separating variables, procedures, and data, as in traditional programming languages, object-oriented programs group all pieces together into "objects." An example of an object might be employee identification and payroll information and a set of corresponding rules for calculating monthly payroll for a variety of job classifications and tax rules. This process of grouping the data and instructions together into a single object is called ***encapsulation***.

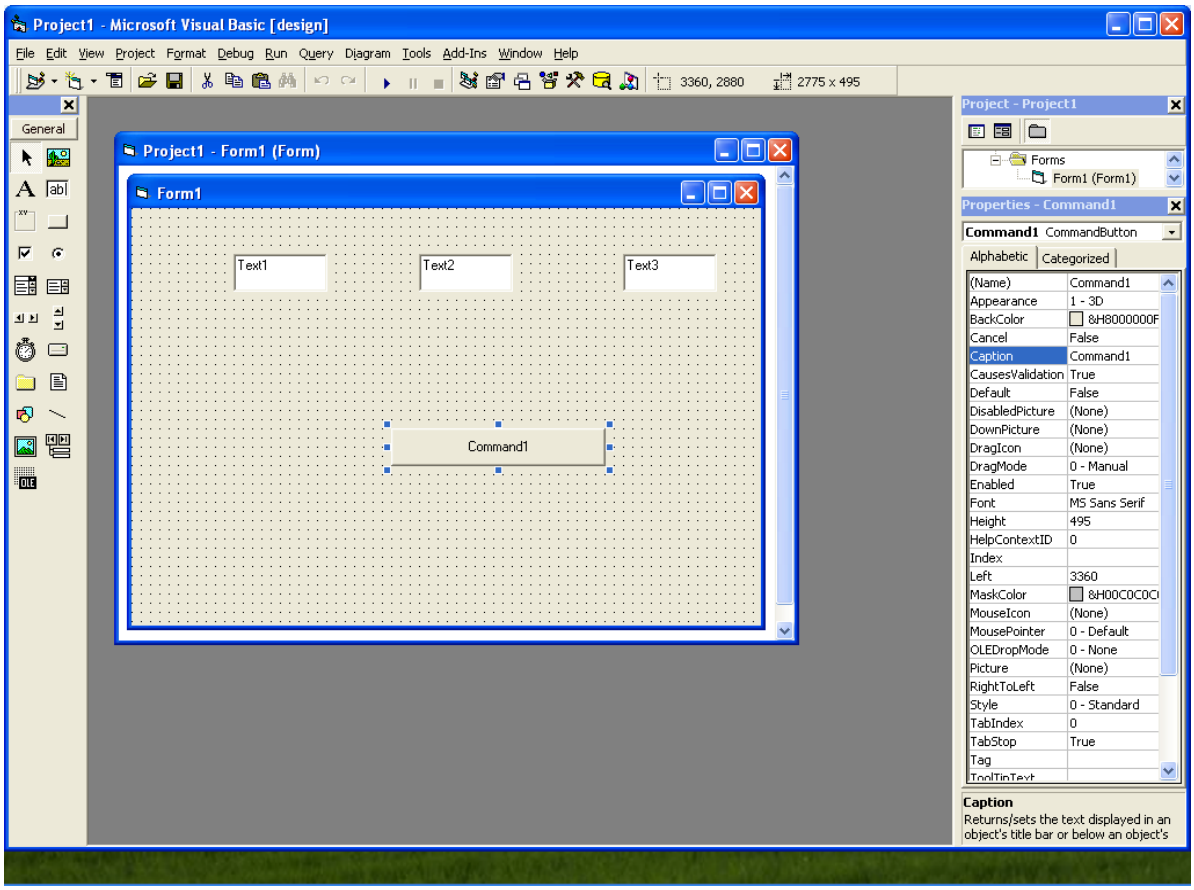
By ***encapsulating*** the instructions and data together, programs are easier to maintain because the things that are grouped together are protected or isolated from other parts of the program.

A second characteristic of object-oriented languages is ***inheritance,*** which means that all lower-level, or children, nodes in an inheritance hierarchy inherit the characteristics of the parent node. In addition to being object-oriented, programs and programming languages can also be ***event-driven***. Unlike programs written in procedural programming languages, programs written with the event-driven approach do not follow a sequential logic. The programmer does not determine the sequence of execution for the program. The user can press certain keys and click on various buttons and boxes presented to her.

6) Visual Programming Languages

Visual programming languages make programming easier and more intuitive. They allow the programmer to create the graphics-intensive applications that today's business user's demand. For example, to make a button appear for the user on a particular screen at a particular point in time, a programmer using a visual programming language only needs to bring up the screen where the button is to appear, choose the button from a palette of choices,

drag and drop the button to the proper location, size and style the but ton with a few mouse clicks, and click on the button's pop-up menu to set the properties that will control its behavior (see Figure bellow



Programming languages translator (PLT)

PLT are programs that translate other programs into machine language instruction codes that computers can execute and can be divided into the following:-

1) An assembler translates the symbolic instructions codes of program written in an assembler language into machine language instructions.
2) A compiler program that translate a high level programming language into machine language program.
3) An interpreter program that translates and executes each source language statement before translating and executing the next one.

Programming tools

Programming tools help programmers write programs by providing program creation and editing facilities.

Attributes or features of a Good Language

Sometimes indirectly, determines which languages live and die. Many reasons might be suggested to explain why programmers prefer one language over another. Let us consider some of these.

1. **Clarity, simplicity, and unity**. A programming language provides both a conceptual framework for thinking about algorithms and a means of expressing those algorithms. The language should be an aid to the programmer long before the actual coding stage. It should provide a clear, simple, and unified set of concepts that can be used as primitives in developing algorithms. To this end it is desirable to have a minimum number of different concepts, with the rules for their combination being as simple and regular as possible. We call this attribute conceptual integrity.

The syntax of a language affects the ease with which a program may be written, tested, and later understood and modified. The readability of programs in a language is a central issue here. A syntax that is particularly terse or cryptic often makes a program easy to write (for the experienced programmer) but difficult to read when the program must be modified later.

2. **Orthogonality**. The term orthogonality refers to the attribute of being able to combine various features of a language in all possible combinations, with every combination being meaningful. For example, suppose a language provides for an expression that can produce a value, and it also provides for a conditional statement that evaluates an expression to get a true or false value. These two features of the language, expression and conditional statement, are orthogonal if any expression can he used (evaluated) within the conditional

When the features of a language are orthogonal, then the language is easier r to learn and programs are easier write because there are fewer exceptions and special cases to remember. The negative aspect of orthogonality is that a program will often compile without errors even though it contains a combination of features that are logically incoherent or extremely inefficient to execute. Because of these opposing qualities. orthogonality as an attribute of a language design is still controversial, since some like it and others do not.

3. Naturalness for the application. A language needs a syntax that when properly used allows the program structure to reflect the underlying logical structure of the algorithm. Ideally it should he possible to translate such a program design directly into appropriate program statements that reflect the structure of the algorithm. Sequential algorithms, concurrent algorithms, logic algorithms, etc., all have differing natural structures' that are represented by programs in those languages.

The language should provide appropriate data structures, operations, control structures, and a. natural syntax for the problem to be solved, One of the major reasons for the proliferation of languages is just- this need for naturalness.

4 Support for abstraction. Even with the most natural programming language for an application, there is always a substantial gap remaining between the abstract data structures and operations that characterize the solution to a problem and the particular primitive data structures and operations built into a language. For example, C may he an appropriate language for constructing a program to do class scheduling for a university, but the abstract data structures of "student ,""c lass section,

“instructor,’ lecture room,” and the abstract of operations of “assign a student to a class section,” “schedule a class section in a lecture room” etc., that, are natural to the application are not provided directly by C.

.5. Ease of program ,verification. ‘the reliability of programs written in a language is always a central concern There are many techniques for verifying that a program correctly performs its required function .A program may be proven correct by a formal verification method it, may be informally proven correct by desk checking (reading and visually checking the program text), it may be tested by executing it with test input data and checking tile output results against the specifications, etc, For large programs some combination of all these methods is often used. A language that makes program verification difficult may be far more troublesome to use than one that supports and simplifies verification.

6. Programming environment. The technical structure of a programming language is only one aspect affecting its utility. The presence of an appropriate programming environment may make a technically weak language easier to work with than a stronger language that has little external support. A long list of factors might be included as part of the programming environment. The availability of a reliable, efficient, and well-documented implementation of the language must head the list. Special editors and testing packages tailored to the language may greatly speed the creation and testing of programs. Facilities for maintaining and modifying multiple versions of a program may make working with large programs much simpler.

7. Portability of programs. One important criterion for many programming projects is that of the transportability of the resulting programs from the computer on which they are developed to other computer systems. A language that is widely available and Whose definition is independent of the features of a particular machine forms a useful base for the production of transportable programs. C,

and c++ all have standardized definitions allowing for portable applications to be implemented.

8 Cost of use. Cost is certainly a major element in the evaluation of any programming language, but different cost measures are feasible:

(a) Cost of program execution. Research on the design of optimizing compilers, efficient register allocation, and the design of efficient run-time support mechanisms was important Cost of program execution. Although always of some importance in language design, is of primary importance for large production programs that will execute repeatedly. For many applications, speed of execution not of highest concern. With desktop machines running at several million instructions per second and sitting idle much of the time.

(b) Cost of program translation. When a language like C is used in teaching, the question of efficient translation (compilation) rather than efficient execution may be paramount. Typically, student programs are compiled many times while being debugged but are executed only a few times. In such a case it is important to have a fast and efficient compiler rather than a compiler that produces optimized executable code.

(c) Cost of program creation, testing, and use. Yet a third aspect of cost in a programming language is exemplified by the language Smalltalk. For a certain class of problems a solution may be designed, coded, tested, modified, and used with a minimum investment of programmer time and energy. Smalltalk is cost effective in that the overall time and effort expended in solving a problem on the computer is minimized. Concern with this sort of overall cost iii use of a language has become as important in many cases as the more traditional concern with efficient program execution and compilation.

(d) Cost of program maintenance. Many studies have shown that the largest cost involved in any program that is used over a period of years is not the cost of initial design, coding, and testing of the program, but total life cycle costs including development costs and the cost of maintenance of the program while it is in production use. Maintenance includes the repair of errors discovered after the program is put into use, changes in the program required as the underlying hardware or operating system is updated, and extensions.

The Structure And Operation Of A Computer

A computer is an integrated set of algorithms and data structures capable of storing and executing programs. A computer may be constructed as an actual physical device using wires, integrated circuits, circuit boards, and the like ,in which case it is termed an actual computer or hardware computer. However, it may also be constructed via software by programs running on another computer in which case it is a software-simulated computer.

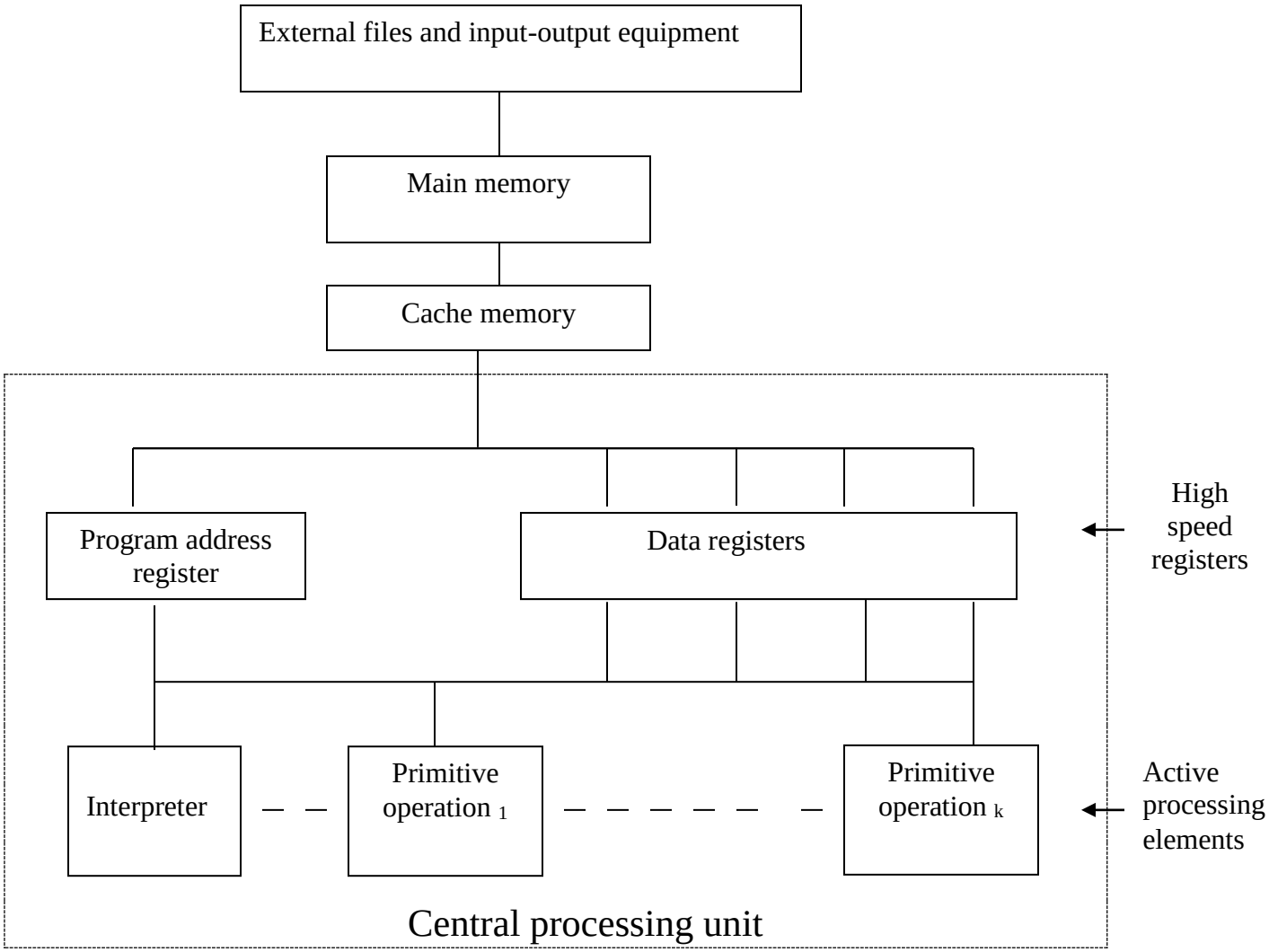


Figure 1 Organization of a conventional computer

A computer consists of six major components that correspond closely to the major aspects of a programming language:

1. Data. A computer must provide various kinds of elementary data items and data structures to be manipulated.

2. Primitive operations A computer must provide a set of primitive operations useful for manipulating the data.

3. Sequence control. A computer must provide mechanisms for controlling, the sequence in which the primitive operations are to be executed.

4. Data access .A computer must provide mechanisms for controlling the data supplied to each execution of an operation.

5. Storage management. A computer must provide mechanisms to control the allocation of storage for programs and data.

6. Operating environment. A computer must provide mechanisms for communication with an external environment containing programs and data to be processed.

The Hardware of the Computer

Hardware computers organizations vary widely, but Figure 1 illustrates a fairly typical conventional organization. A main memory contains programs and data to be processed. Processing is performed by an interpreter, which takes each machine language instruction in turn, decodes it, and calls the designated primitive operation the designated operands as input. The primitives manipulate the data in main memory and in high-speed registers and also may transmit programs or data in memory and the external operating environment. Let us consider the six major parts of the computer in more detail.

Data. The schematic of Figure1 shows three major d storage components: main memory, high-speed registers and external files. Main memory is usually organized as a linear sequence of bits subdivided into fixed-length words (typically 32 or 64 bits) or 8-bit bytes (typically 4 or 8 bytes per word). The high-speed register consist of word-length bit sequences and may have special subfields that are directly accessible the contents of a register may represent data or the address in main memory containing the data or next instruction. A high-speed cache memory is often situated

between main memory and the registers as a mechanism to speed up access to data from this main memory. External files, stored on magnetic disk, magnetic tape, or increasingly today, CD ROM, are usually subdivided into records, each of which is a sequence of bits or bytes

A computer has certain built in data types that can be manipulated directly by hardware primitive operations A common set might include integers, single precision (e g, one word) reals, also called floating point numbers, fixed length character strings, and fixed length bit strings (where the length is equal to the number of bits that fit into a single word of storage) besides these obvious hardware data elements, programs are also a form of data. As with the other built in data types there must be a built-in representation for programs, termed the machine language representation of the computer; Typical a machine language program would be structured as a sequence of memory locations, each containing. one or more instructions. Each instruction in turn is composed of an operation code and a set of operand designators.

***Operations*.** A computer must contain a set of built-in primitive operations, -usually one to one with the operation codes that may appear in machine language instructions. A typical set would include primitives for arithmetic on each built in numeric data type (e g, real and integer addition, subtraction, multiplication, and division), primitives for testing various properties of data items (e.g., test for zero, positive, and negative numbers), primitives for accessing and modifying various parts of a data item (e.g., retrieve or store a character in a word and retrieve Or store an operand address in an instruction), primitives for controlling input-output devices, and primitives for sequence control (e.g.,. unconditional and return jumps)

Sequence control. ‘the next instruction to be executed at any point during execution of a machine language program is usually determined by the contents of a special program address register(also called the Location counter), which always contains the memory address of the next instruction. Certain primitive operations are allowed to modify the program address register in order to transfer control to another part of the program, but. it is the interpreter that actually uses the program address register and guide the sequence of operation.

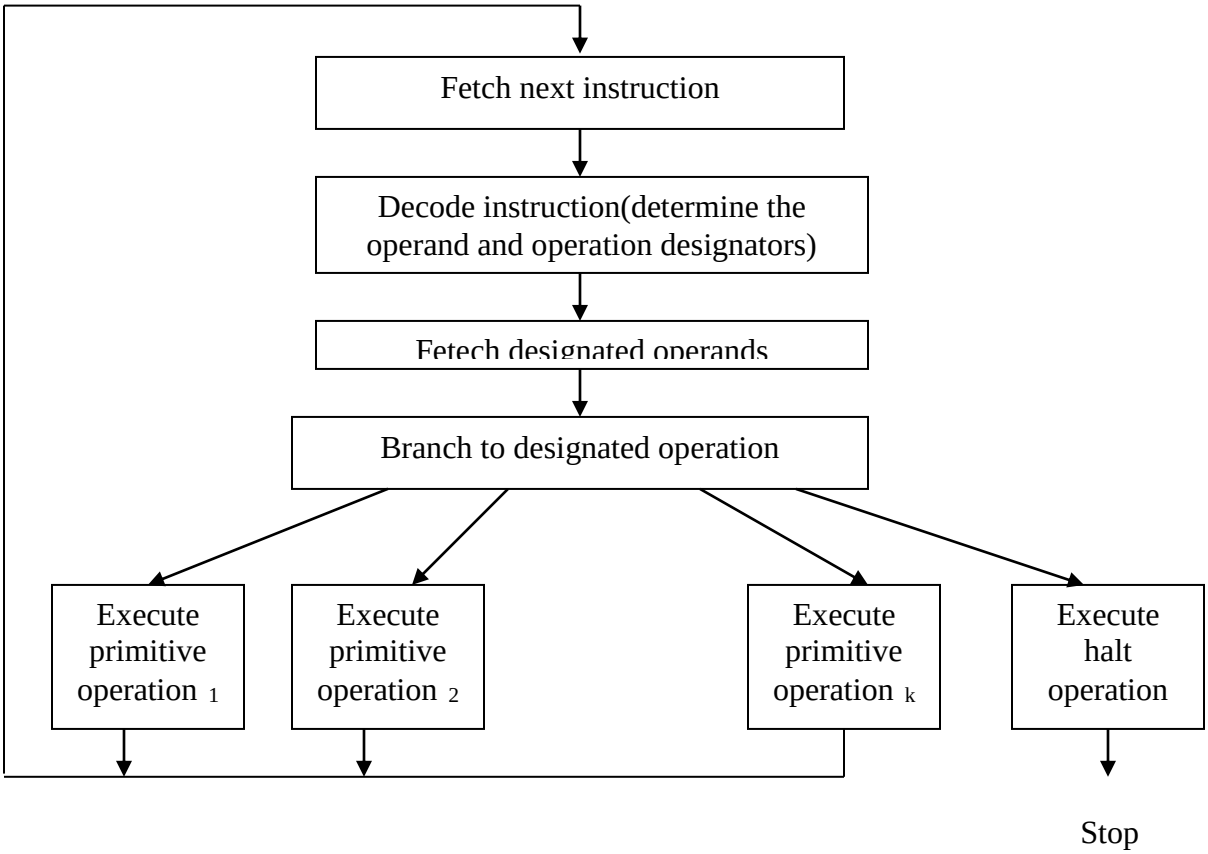


Figure 2. Program interpretation and execution.

The interpreter is central to the operation of a computer. Typically the interpreter executes the simple cyclic algorithm shown in Figure 2. During each cycle the interpreter gets the address of the

next instruction from the program address register (and increments the register value to be the next instruction address),

Fetches he designated instruction from memory, decodes the instruction into an operation code and a set of operand designators, fetches the designated operands (if necessary), and calls the designated operation with the designated operands as arguments. The primitive operation may modify data in memory or registers, access input-output devices, or change the execution sequence by modifying the contents of the program address register. After execution of the primitive the interpreter simply repeats the above cycle.

Data access. Besides an operation code, each machine instruction must specify the operands that the designated operation is to use Typically an operand might be in main memory or in a register. A computer must incorporate a means of designating operands and a mechanism for retrieving operands from a given operand designator. The result of a primitive operation must be stored in some designated location we term these facilities the data access control of the computer

The conventional scheme is to simply associate integer addresses with memory locations and provide operations for retrieving the contents of a location given its address (or alternatively. for storing a new value in a location whose address is given)

Storage management. One driving principle in machine design is to keep all resources of the computer (e.g., memory, central processor, external data devices)operating as much as possible The central conflict in achieving this principle is at operations within the CPU typically happen at the nanosecond level (e g, 10- 50 nanoseconds for a typical operation on a modern processor), accessing memory occurs at the microsecond level (1-2) microseconds or 100-200 nanoseconds), and external data operations occur at the millisecond level (15-30 milliseconds). This responds to a factor of 1,000,000 between the internal speed of a microprocessor and the speed o reading data from a disk In order to balance these speeds appropriately; various storage management facilities are employed.

For speeding up the imbalance between external data access and the central processor, multiprogramming is often used by the operating system. While waiting the many milliseconds for data to be read (e.g., from disk), the computer will execute another program in order to allow for many programs to be co-resident in memory at the same time.

For speeding up the imbalance between main memory and the central processor a cache memory is used. A cache memory is a small .high-speed data storage that is between main memory and the central processor (see Figure 2). This memory typically 1K (where K is standard computer jargon for 1,024 or 2K bytes to 256K bytes) contains the data and instructions most recently used by the central processor, and hence contain the data and instructions most likely to be needed again in the near future.

Operating environment The operating environment of a computer ordinarily consists of a set of peripheral storage and input-output devices These devices represent the “outside world” to the computer, and any communication with the computer must be by way of the operating environment Often there are hardware distinction between various classes of devices in the environment,

based on differences in use or in speed of access, e g high-speed storage (e g, extended memories), medium speed storage (magnetic disks, CD-ROM), low-speed storage (tapes), and input output devices (readers printers, displays data communication lines)

The store of identifiers

The store is conceptually a very simple device used for holding values as the processor executes a program. It consists of a series of storage cells as shown in figure 1.

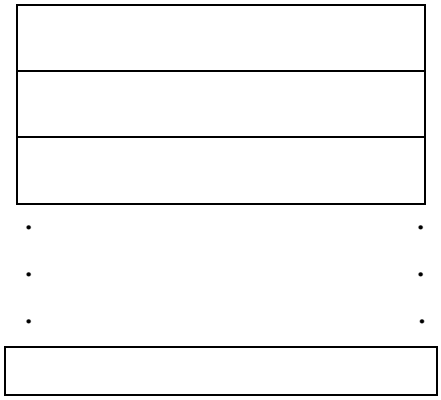


Figure1. The store of identifiers

Each cell in the store can hold a single value, such as 25, -57.3, 0, or 3.14159. In order to keep track of which storage cells contain which values, we are allowed to name individual cells. We can then refer to the contents of a particular cell by using the cell's name. Figure 2 illustrates the store when some of the cells have been given names and contain numeric values. In this figure the cell named cat has the value 45. Rather than saying "the cell named cat has the value 45" each time, we shorten this to "cat has the value 45" or, even more simply, "cat = 45." In this way the actual storage cell becomes transparent to us: we identify the cell and its contents with the cell name.

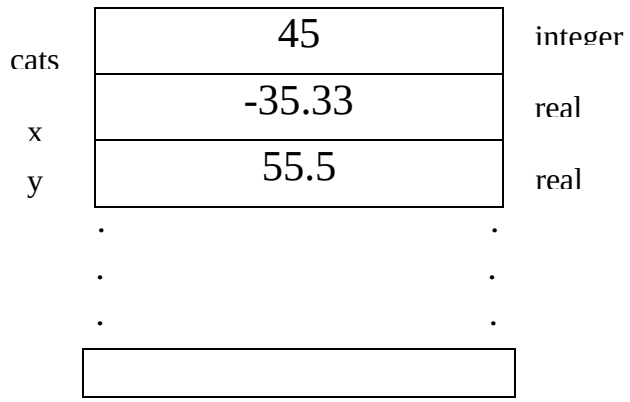


figure 2. the store with cell names and values

Notice also in figure 2 that some of the numeric values in the cells are whole numbers (numbers with no decimal point) and some are fractional numbers (numbers with a decimal point). In our model, as well as in the real computers we are modeling, these two types of numbers are handled differently, so we must be careful to distinguish between them. Formally, we refer to whole numbers as integers and to fractional numbers as reals.

The Variables Instruction

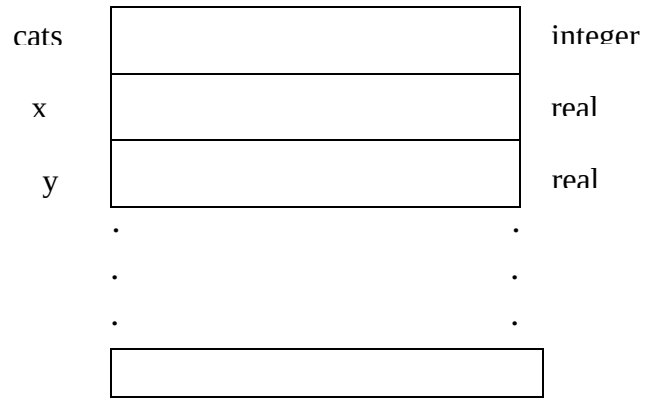
While each memory cell can hold either integer or real values, the processor must know which type of value each storage cell currently contains so that arithmetic on these values can be done properly. One of our responsibilities as programmers is to supply both the names of storage cells we wish to use and type of value we intend to store in each cell. We will do this in each procedure we write by using a variables instruction. For example, a typical declaration of the names and types of storage cells might be

Variables

cats: integer

x,y: real

When the processor encounters this instruction, it will name the first three cells of the store cats, x, and y. respectively. The cell named cats can hold only integer values, whereas the cells named x and y are designated to hold only real values. After processing this variables instruction, the processor would setup the store to look like



The integer and real designations after each storage cell indicate the type of value the cell can contain. The cell names cats, x, and y are called variables, because the storage cells they stand for can hold different values at different times; that is, the values associated with cats, x, and y can vary throughout the course of the program.

The Assignment Statement

To assign values to the storage cells established in a variables instruction, we can use an assignment statement. For example, the assignment statement

cats ← 20

assigns the value 20 to the storage cell cats. Similarly, the statement

y ← 3.14159

puts the value 3.14159 into the cell named y. In executing an assignment statement, the processor takes the value on the right of the ← symbol and stores it into the cell whose name is given on the left side of the ← symbol. After the two assignment statements shown above are executed, the store looks like

cats	20	integer
x		real
y	3.14159	real
	.	.
	.	.
	.	.

Notice that x still has no assigned value; we say that x is undefined. (Immediately after a variables instruction is executed, all of the newly named cells undefined values, since, the variables instruction only names, not values, to the cells.)

The assignment statement allows great flexibility. We can give the storage cells values that we will use in subsequent calculations. Later we can easily change the value in a cell by simply issuing another assignment statement. For example, if the store appears as last shown and we issue the statement

Cats ← - 30

The store would now appear as

cats	-30	integer
x		real
y	3.14159	real
	.	.
	.	.
	.	.

A natural question to ask about the use of assignment statements is What happen if we assign a real value to an integer cell? or What happens if we assign an integer value to a real cell? In the first case, when our model processor handles an assignment statement such as

Cats ← 12.721

it recognizes that the variable cats can contain only integer values, so it truncates the value 12.721 to 12 and stores 12 into the cell named cats. In other words if a real value is to be stored into an integer cell, the processor takes the whole number portion of the value and discards the fractional part. It does not round the value. If an integer value is to be assigned to a real cell, the processor simply adds a decimal point to the end of the integer before storing it into the real cell. For example,

x ← 25

Would result in 25.0 being stored into x. Thus, execution of the two assignment statements

Cats ← 12.721

x ← 25

cats	12	integer
x	25.0	real
y	3.14159	real
	.	.
	.	.
	.	.

Although we can change the values stored in the cells, we cannot change the cell types nor can we rename the cells once we have set them up with a variables instruction.

The assignment statement is actually more flexible than we have so far described. We are allowed to write statements of the form

Variable ⟵ *arithmetic expression*

Where on the left of the ⟵ symbol we have a variable that has been previously declared in a variables instruction and on the right of the ⟵ symbol we have any arithmetic expression. For example, having declared x to be a real variable, we can issue the instruction

x ⟵ (3.0 + 5.0 + 7.0)/2.0

The processor first evaluates the expression

(3.0 + 5.0 + 7.0)/2.0

to get 7.5, and then the 7.5 is stored into x. The evaluation chart for this assignment statement looks like

x ⟵ (3.0 + 5.0 + 7.0)/2.0

x ⟵ (8.0 + 7.0)/2.0

x ⟵ 15.0/2.0

x ⟵ 7.5

7.5 is stored into x

If the store contained the values last shown, then it would contain the following values after execution of the above assignment statement:

cats	12	integer
x	7.5	real
y	3.14159	real
	.	.
	.	.
	.	.

If the expression on the right side of the assignment statement evaluates to a real number, but the variable on the left side has type integer, then, as usual, the real number is truncated before it is stored into the variable. Similarly, if the result of the arithmetic expression is an integer and it is to be stored into a real variable, a decimal point is added to the integer number.

If we mix integer and real values in an arithmetic expression to the right as in an example bellow

x ← 7 /4 * 3.2

In the model computer and the model programming language we are developing, we will assume that every operation results in the

most accurate possible answer. Thus, the evaluation tree for this expression is

$$x \leftarrow 7/4 * 3.2$$

$$x \leftarrow 1.75 * 3.2$$

$$x \leftarrow 5.6$$

5.6 is stored into x

As a result of this assignment statement, the store would now look like

cats	12	integer
x	5.6	real
y	3.14159	real
	.	.
	.	.
	.	.

The character variables

We need a way to read in character strings one character at a time. This requires introduction of character variables. If a variable is declared to be of type character, it can hold just a single character.

.Example: A that we have the following declaration:

variables

x: integer

a, b, c : character

Then we can make assignments like

a ← ‘*’

b ← ‘b’

c ← ‘ ’

which assigns an asterisk to variable a, the letter ‘b” to variable b, and the blank to variable c. Notice that we still need the quote marks around the characters on the right side. This is required in assignment statements, otherwise a statement like

a ← c

would be ambiguous. Would this mean “assign the letter ‘c’ to the variable a” or “assign the contents of the variable c to the variable a”? Assigning a character to a character variable in an assignment statement thus requires the use of quotes.

When reading in characters with the input statement, however no quotes are required around the characters in the input, if the input contains

-30 2ab

and the statement

input x, a, b, c

is executed, the value in x will be-30, a will contain the character blank, b the character 2, and c the letter “a”. This happens as follows: Since x is of type integer, the processor scans past all blanks in the input line until the first nonblank symbol is found (the “-“). The processor then assumes that between the “-“ and the next blank symbol there is a number (if there isn’t a number here, an error occurs), so the -30 is input into x. At this point the input pointer is at the blank between 0 and 2; the input pointer points to the character position following the last item input so that the next input operation can take up from that point. Since a is of type character, the character pointed to by the input pointer (the blank) is placed into a, then the input pointer is moved to the right one space to point to the 2. Thus, the character 2 is stored into the variable b and the pointer is moved again. Finally, the letter ‘a” is stored into the variable c and the pointer is moved to the right one more time. After this input statement has been executed, the input pointer is left pointing at the “b’. The 2 in this example is treated as a character rather than a number, since it was input into a character variable.

If we are going to use character variables to input strings character by character, we need a mechanism to join characters arid strings together to form larger strings. This operation is called concatenation.

Example:

‘ab’ || ‘def’
is equal to
‘abdef’

To build up strings a character at a time, we need to start with a special string called the null string. The null string is the string that contains no characters at all, not even a blank.

Example: If str has been declared to be of type string, the assignment

str ← ‘‘

assigns the null string to str; there is nothing in between the two quote marks. Notice that this is not the same as

str ← ‘ ‘

which assigns a single blank character to str or

str ← ‘’’‘

which assigns a single quote mark to str.

The null string is the string of length 0 (in contrast, the string containing a single blank has length 1). This means, among other things, that the null string cannot be assigned to a variable of type character, since all character variables must be assigned a single character.

We can use the null string and concatenation to build up a string from the input one character at a time.

Example:

Assume that the input pointer is pointing to the first character of a word and the word is flowed by a blank. You should convince yourself that the following segment will input the word into the variable called word, assuming that word has type string and variable ch has type character:

```
word ←—— ‘’
input ch
loop while ch ≠’ ‘
word ←—— word || ch
input ch
endloop while
```

To try this segment, suppose the input is

the quick brown fox

with the input pointer currently pointing to the “t” of the. When the loop is entered, word will contain the null string and ch will contain the first letter, “t”. Within the loop, execution of the statements

```
word ←—— word || ch
input ch
```

will leave word containing “t” and ch containing “h”. The next iteration of the loop will assign “th” to word and “e” to ch. The third loop iteration will assign “the” to word and “ “to ch, which will cause termination of the loop.

The Processor

The processor derives its name from its function: based on instructions supplied by the programmer, the processor processes given data values to produce new values. As shown in figure 1, the processor has a number of different units for accomplishing these tasks. The first is the program unit, where the list of instructions (the program) is kept. There are other units for adding, subtracting, multiplying, and dividing two values. There is also a unit for exponentiating (raising a value to some power, for example, $10^{3)}$. Finally, there is a unit for comparing two values.

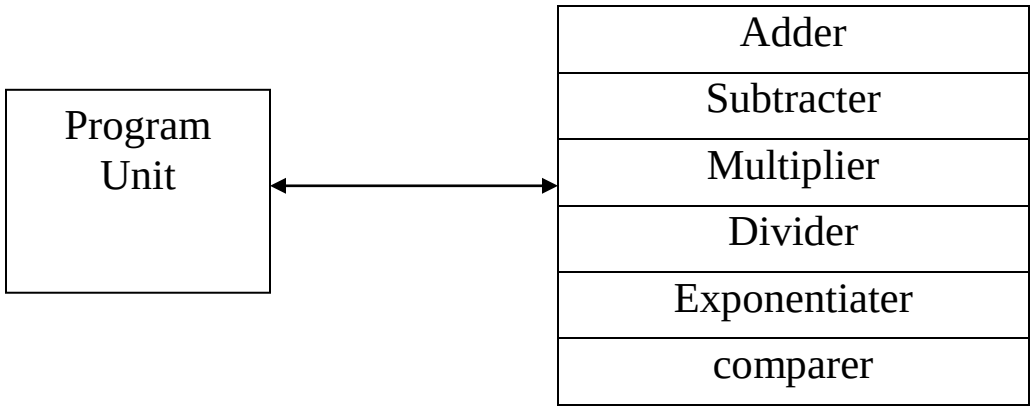


Figure1. The Processor

In order to understand how a computer is programmed, we must see how the arithmetic units in the processor function .The processor uses these five arithmetic units to evaluate arithmetic expressions For example the arithmetic expression 2 + 2 is

evaluated by the adder to produce 4 The + operator signals the processor to enter the values on either side of the + into the

adder to produce the result. The other operators work similarly. Table 1 gives the list of operators that the processor recognizes and can process. The+ and- symbols are familiar to us. The others may not be so familiar. The * is used to represent multiplication instead of the symbol X, which could be confused with the letter x. The symbol for divide is /; this allows us to write expressions involving division on a single line, for example,

1/2 instead of ½

The exponentiation symbol ↑is used for the same reason :we can write 10^3 as 10 ↑ 3 on a single line.

<u>Table 1 arithmetic operators</u>

+	Addition
-	Subtraction
*	Multiplication
/	Division
↑	Exponentiation

Example Table 2 gives a series of simple arithmetic expressions and their values as computed by the arithmetic units of the processor.

Table 2. Arithmetic Expressions And Values

Expression	Computed Value	Expression	Computed Value
3+5	8	-2*3	-6
5-3	2	-2*-3	6
3-5	-2	4/2	2
2*13	26	4↑2	16
13*2	26	3.2*1.1	3.52
7/4	1.75	2↑2	4

These expressions are all evaluated as we would expect. Notice that the processor can distinguish whether the symbol - is used to mean negative number" or "subtract." For example, the expression -2 * 3 in the above table evaluates to -6 (we read this expression as "multiply negative two times three").

Actually, the processor is able to process more complex arithmetic expressions such as

2+3+4

which is evaluated in the processor to produce 9. Since the adder can add only two numbers at a time, however, each + operator is handled separately: first the 2 is added to the 3 to get 5, and then this 5 is added to the 4 to get 9. The processor evaluates this more

complicated expression from left to right. If we did not want the processor to handle the expression from left to right, we could use parentheses to cause the processor to evaluate the expression in a different order. We simply rewrite the expression as

2 + (3 + 4)

using parentheses to indicate which sub expression is to be evaluated first. In this case, the processor first adds the 3 and 4 to get 7 and then adds the 2 to this 7 to get 9.

The use of parentheses in the above example does not change the final re suit of the expression; in both cases the computed value is 9. Consider the arithmetic expression

2 +3*4

If we rewrite this expression as

(2 + 3) * 4

then the processor will first use the adder on 2 and 3 to produce 5; this 5 and the 4 will then be processed by the multiplier to produce the value 20. On the other hand, if we rewrite the expression as

2+ (3 * 4)

the 3 and the 4 will first be multiplied to produce 12, and then the 2 and this 12 will be added to produce 14. In this case, the differing

use of parentheses causes the processor to evaluate the expressions in different orders and gives different results

What happens if we direct the processor to evaluate the expression2+ 3 * 4 without parentheses? In this case the processor does not evaluate the expression from left to right; precedence is given to the multiplication operator Over addition operator, and the expression is evaluated as if it had been parenthesized as 2+ (3 * 4) to yield the result 14.

The processor has precedence rules that it follows when evaluating arithmetic expressions. Expressions in parentheses are evaluated first. (If a parenthesized sub expression also contains parentheses, the sub expression surrounded by the innermost set of parentheses is evaluated first.) Exponentiation the operation with the highest precedence is performed next. Multiplication and division have the same precedence, just below exponentiation. Final addition and subtraction have the same precedence but are lower than that of multiplication or division. When there are two or more operators at the same parentheses level and with the same precedence, the leftmost of these operators is evaluated first. Table 3 shows the precedence rules used by the processor when evaluating arithmetic expressions.

<u>Table 3 precedence chart for arithmetic expressions</u>.

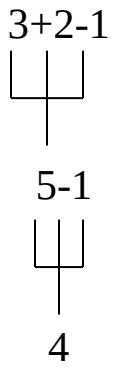
1- (,) expressions in parentheses are evaluated first; innermost parentheses have priority.

2- ↑ exponentiation is the arithmetic operation with highest precedence.

3- *, / multiplication and division have the same precedence.

4- +, - addition and subtraction have the same precedence.

Example: The expression 3 + 2 - 1 is processed to yield the value 4. Since + and - have the same precedence level, the 3 and 2 are first added to get 5, then the 1 is subtracted to yield 4. This can be illustrated as



This graphical representation of the evaluation of an expression is called an evaluation chart; it is a particularly convenient way of describing the order of evaluation, We will use evaluation charts in subsequent examples.

Example The evaluation chart for (3 + 2) * 2 / 5 is

(3 + 2) * 2 / 5

5*2/5

10 / 5

2

Example The evaluation chart for 2 * (3 + 7)/2 is

2*(3 + 7) / 2

2*10/2

20 / 2

10

The PROCESSOR COMPARISON UNIT

We now understand better how the computer performs arithmetic. The processor is capable of evaluating arithmetic expressions by using the five arithmetic units it contains: the adder, subtracter, multiplier, divider, and exponentiater. Computed values are assigned to cells of the store as the processor executes assignment statements. Each of these computations requires comparing values. This is done in the comparison unit of the processor, which operates in a way similar to the way the arithmetic units operate.

The average of four scores was computed with the assignment statement bellow

Average — (score 1 + score2 + score3 + score4) / 4

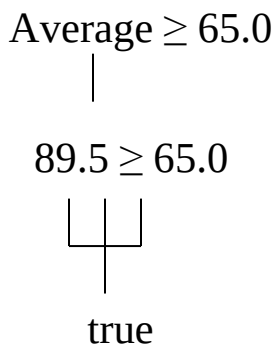
Afterwards the store looked like

average	89.5	real
score1	75	integer
Score2	100	integer
Score3	88	integer
Score4	95	integer
	. . .	. . .

Now we would like to determine whether the computed average is a passing score. If we have previously decided that 65.0 or above is passing, then we can have the processor decide whether average is greater than or equal to 65.0 by having it evaluate the expression

Average ≥ 65.0

We can think of the processor working this way: the value in average, the value 65.0, and the greater than or equal to symbol (≥) are each supplied to the comparison unit. The comparison unit then determines whether average is greater than or equal to 65.0. The evaluation chart for this expression is



The comparison unit in this case returns the value true because 89.5 is indeed greater than or equal to 65.0.

If the value in average were 57.3 instead of 89.5, the evaluation chart for

Average ≥ 65.0

Would be

Average ≥ 65.0

57.3 ≥ 65.0

false

In this case the comparison unit would return the value false because 57.3 is not greater than or equal to 65.0.

The symbol ≥ in the above example is called a relational operator, and the relation

Average ≥ 65.0

is called a conditional expression. Simple conditional expressions have the

value1 **relational operator** *value2*

where value 1 and value 2 are either variables or constants. Evaluation of conditional expressions by the comparison unit results in either true or false in every instance, because valuel either stands in the expressed relation to value2(**true**) or it does not (**false**).

The relational operators that can be processed by the comparison unit are

Given in able1. We can compare two values to see if the first value is less than or equal to, equal to, not equal to, greater than, or greater than or equal to the second value.

Table 1. Relational Operators

<	Less Than
≤	Less Than or Equal To
=	Equal To
≠	Not Equal To
>	Greater Than
≥	Greater Than or Equal To

Example 1: Assume again that the store looks like

average	89.5	real
score1	75	integer
Score2	100	integer
Score3	88	integer
Score4	95	integer
	. . .	. . .

If we wish to assign a letter grade to the average we have computed instead of just deciding whether it is a passing score , the situation is somewhat more complex. For example, assume that the letter grade B is to be awarded if the average is at least 85.0 but not 92.0 or larger. In other words, we wish to test the condition that 85.0 is less than or equal to average and average is less than 92.0. This would be written mathematically as

$85.0 \leq$ average <92.0

However, this expression cannot be evaluated by the comparison unit, which can only handle two values at a time. If we tried to write the evaluation chart for this expression to see how the comparison unit would attempt to process it we would get

85.0 ≤ average <92.0

85.0≤ 89.5 <92.0

true < 92.0

???

since comparison unit can handle only two values at a time, it evaluates the first two to begin with. This results in the expression

true < 92.0

which makes no sense at all: true has absolutely no relationship to 92.0.

to write these more involved conditional expressions, then, we need a different method. Remember how we stated this expression in words: “85.0 is less than or equal to average and average is less than 92.0.” This statement has two parts, which we can separate as

85.0 is less than or equal to average

and

average is less than 92.0

in this case and is called a logical operator because (logically) if the first expression is true and the second expression is true, then the entire compound expression is true.

In order for average to lie between 85.0 and 92.0, both expressions must be true. However, if either expression (or both) is false, then (logically) the entire expression is false. We can rewrite the above expression as

(85.0 ≤ average) and (average <92.0)

This expression can now be evaluated by the comparison unit as

(85.0 ≤ average) and (average < 92.0)

(85.0 ≤ 89.5) **and** (89.5 <92.0)

true **and** (89.5 < 92.0)

true **and** true

true

Notice that when the **and** is finally evaluated (recall that expressions in parentheses are evaluated first) the comparison unit no longer remembers" what the two expressions were, but it has computed the fact that both expressions were true. Since the first expression is true and the second expression is true it follows (regardless of what those two expressions were) that the entire compound expression is true. The result computed by the comparison unit for true **and** true is thus true.

What if the value of average were 96.3 instead of 89.5? Then the evaluation chart would be

(85.0 ≤ average) and (average < 92.0)

(85.0 ≤ 96.3) **and** (96.3 <92.0)

true **and** (96.3 < 92.0)

true **and** false

false

Whenever at least one of the values on either side of the **and** is false, then the entire compound expression is false.

This example shows how compound conditional expressions can be constructed using the logical operator and two conditional expressions joined by **and** will be evaluated by the processor to the value true if and only if both of the conditional expressions first evaluate to true. If either or both of the conditional expressions evaluate to false, then the entire expression is false. We can show this symbolically using evaluation charts. If we let exp1 and exp2 be any valid conditional expressions, then we have:

(exp1) **and** (exp2)

true **and** true

true

(exp1) **and** (exp2)

false **and** false

false

(exp1) **and** (exp2)

true **and** false

false

(exp1) **and** (exp2)

false **and** true

false

These four charts represent all possible situations in which the logical operator **and** may appear.

Another logical operator that can be processed by the comparison Unit is **or.** Often we will not need to know whether both of two conditional expressions are true but only whether one or the other is true (or both).

Example: 2 suppose that a program is being designed to process patients' records for a doctor so that a notice can be sent to parents of 5- and 10-year-olds when it is time for their children to receive immunization shots. As each child's record is processed, we must ask the question is this child's age 5 or 10? If it is either 5 or 10, then the immunization no be sent. In terms of conditional expressions we would write this as

(age =5) **or** (age=10) where the variable age was previously declared in a variables statement as

variables
integer: age

if the value of age is 10 (i.e., the child whose record is being processed is 10 years old),then the evaluation chart for this expression is

(age = 5) **or** (age = 10)

(10 = 5) **or** (10 = 10)

false **or** (10 = 10)

false **or** true

true

The comparison unit evaluates this compound conditional expression to true because at least one of the simple conditional expressions on either side of the **or** is true.

What if age is 7? In this case the evaluation chart would be

(age = 5) **or** (age = 10)

(7 = 5) **or** (7 = 10)

false **or** (7 = 10)

false **or** false

false

Here, the entire expression evaluates to false because neither the first simple conditional expression nor the second is true.

There are four situations that can arise whenever the logical operator or is applied to two conditional expressions. If the expressions on both sides of the or evaluate to true, then the entire expression connected by the or would be true (this could happen, for example, if we are checking whether a number is even or greater than zero; clearly a number could satisfy both of these

conditions). If the expressions on both sides of the or evaluate to false, the entire expression connected by the or should be false. Finally, there are two situations in which the expression on one side (left or right) of the or evaluates to true but the expression on the other side (right or left) evaluates to false; in these cases the entire expression connected by the or is true. This can be shown by evaluation charts:

(exp1) **or** (exp2)

true **or** true

true

(exp1) **or** (exp2)

false **or** false

false

(exp1) **or** (exp2)

true **or** false

true

(exp1) **or** (exp2)

false **or** true

true

A final logical operator that is sometimes needed is **not**.

Example: 3

Using the setting of the previous example, suppose that for some reason letters is to be sent to all patients who are not 5 or 10 years old. Then for each patient we could have the processor evaluate the expression

not(age=5) **or** (age= 10))

the not is placed in front of the expression to be negated. If the age of a patient is 7, the evaluation chart for this expression would be

not(age = 5) **or** (age = 10)

not (7 = 5) **or** (7 = 10)

not(false) **or** (7 = 10))

not(false) **or** **(**false)

not false

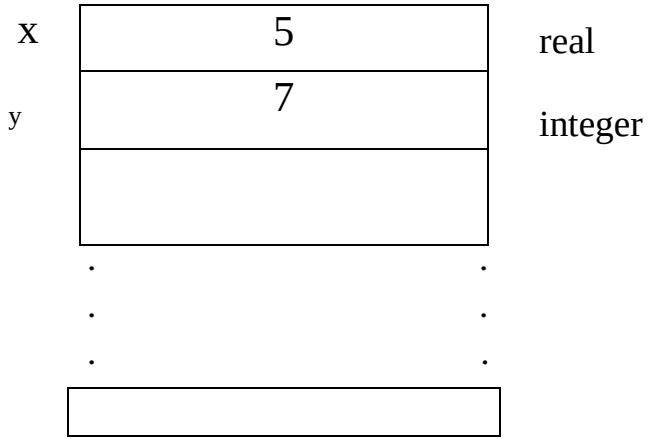
true

in other words, it is true that 7 is not 5 or 10, so this entire expression evaluates to true.

As shown in this example, the not logical operator negate the computed value of a conditional not true is (logically) false, whereas not false is (logically) true. The evaluation charts for not are quite simple and are shown bellow where exp is any conditional expression :

Example: 3

If the store looks like



We can write the expression

x * y+2 < 31 +x

which will be properly evaluated by the processor. Notice that before x*y+ 2 and 31 + x can be compared, their values must be

computed. In our model computer we will assume that the processor gives precedence to arithmetic operators over relational and logical operators. Then the evaluation chart for this expression is

```
x * y + 2 < 31 + x
|   |           |
5 * 7 + 2 < 31 + 5
 \_|_/
  |
  35 + 2 <31 + 5
   \_|_/
     |
     37 < 31 + 5
             \_|_/
             |
     37   <   36
      \_____|_____/
            |
          false
```

Table 2. Complete Operator Precedence Chart For Mpl

1) **(,)** expressions in parentheses are evaluated first; innermost parentheses take priority.
2) **↑** exponentiation is the operator with the highest precedence.
3) *** /** multiplication and division have the same precedence level.
4) **+, -** addition and subtraction are the arithmetic operators with the lowest precedence.
5) **<, > ,=, ≠, ≤, ≥ and**, **or**, **not** relational operators and logical operators have the same precedence; they are the last operators to be executed in an expression.

The Input output devices

These devices provide the processor with links to the outside world. Values can be read from the input device and placed into storage cells by the processor for use in later calculation. Also, values currently in storage cells, such as the results of calculations performed in the processor, can be printed at the output device the operation that the processor performs when reading a value from the input device and placing it into the store called an input operation. Printing the value of a storage cell at the output device is called an output operation.

The Input Device

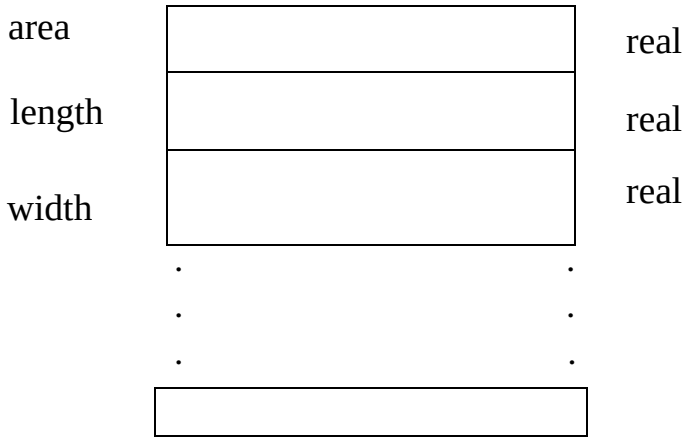
An **input** statement directs the processor to read a value from the input device and place it into a storage location.

Example 1: Assume that the processor has processed the statement

variables

area, length, width :real

causing the store to look like



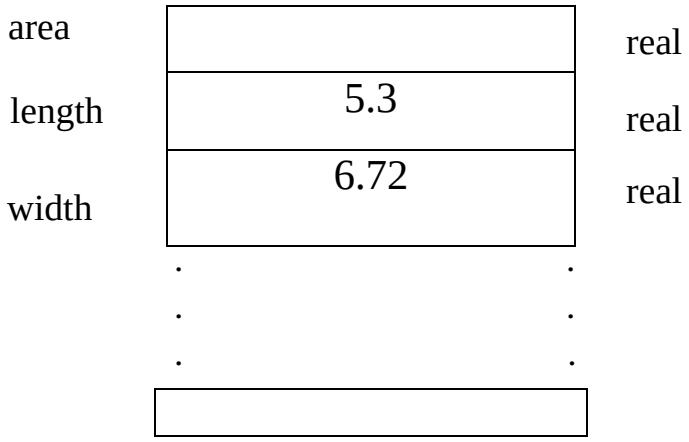
Then upon encountering the statement

Input length, width

the processor will read two values from the input device. The first value will be assigned to the variable length, and the second value will be assigned to the variable width. So if

5.3 6.72

are the values at the input device when the input statement is executed by the processor, the store will become



If the processor then executes the statement

area ← length * width

the variable area would take on the value 35.6 16 and the store would look like

area	35.6 16	real
length	5.3	real
width	6.72	real
	. . .	. . .

At this point you may be somewhat puzzled about the use of the input statement. If its only purpose is to assign values to storage cells, how is it different from the assignment statement? The difference is crucial the input statement is what makes our programs general. For example, if we wrote a program that calculated the area of a rectangle using the statements

length ← 3.0

width ← 2.5

area ← length * width

We would have to change the program every time we wished to compute the area of a different rectangle. That is, we would have to replace the statements

length ← 3.0

width ← 2.5

with different ones, for example,

length ← 7.2

width ← 7.1

There are many reasons why changing the program like this is undesirable: it takes time, it is a source of human error, and, most important, there is a better way.

Consider our first example in this section, where we used the statements

input length, width

area ← length * width

To calculate the area of the rectangle with length 3.0 and width 2.5 we would only need to supply the values 3.0 and 2.5 at the input device as the processor executes the program. If we later wished to calculate the area of the rectangle with length 7.2 and width 7.1,

we could have the processor execute the same program, but this time we would supply the values 7.2 and 7.1 at the input device. In other words, the **input** statement allows us to apply our program to different sets of data without making any modifications to the program. This is what makes a program general.

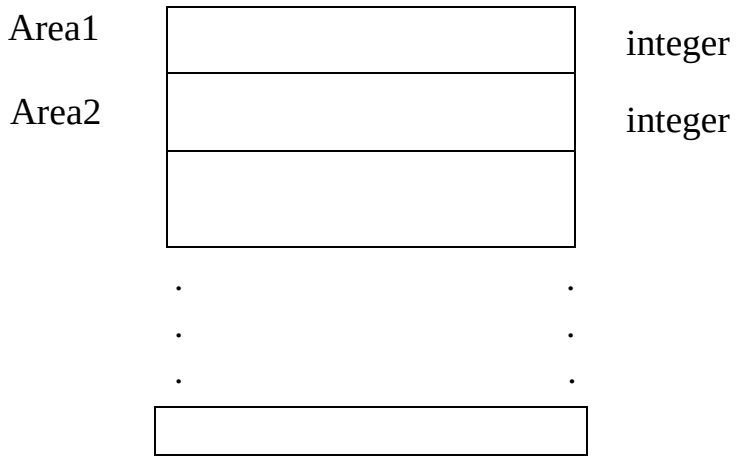
Except for the generality added by the input statement, it acts just like the assignment statement. If a value is placed into a variable by an **input** statement, it remains there until an assignment statement or another input statement changes that value.

Example 2: assume that the store has been organized by the statement

variables

area1, area2: integer

causing the store to look like

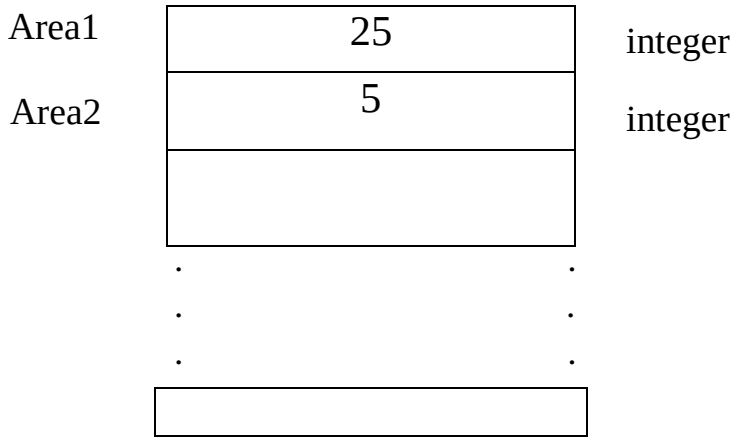


Then the statement

Input area1,area2

Would cause the processor to pick up two values supplied at the input device and place them into area1,area2 respectively. If the two values at the input device were

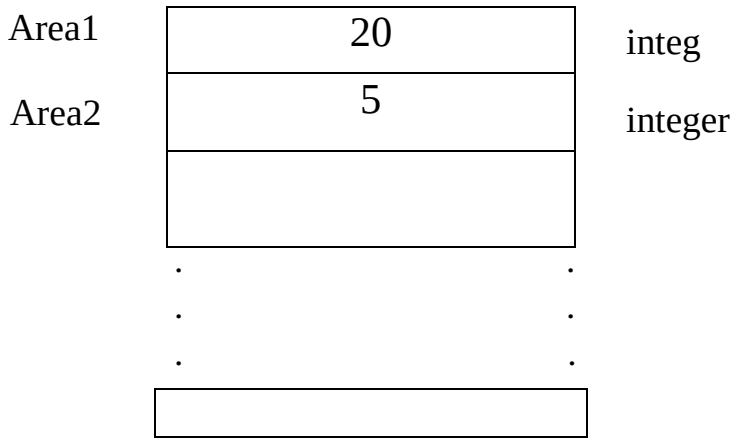
25 5 then after execution of the input statement the store would look like



Afterwards, execution of the statement

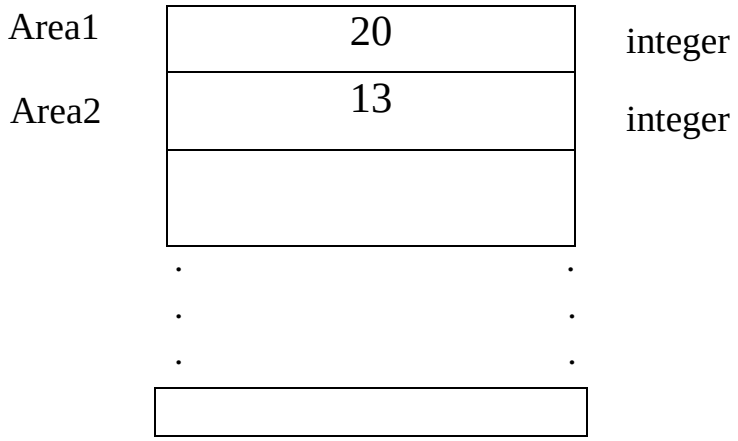
area1 ← area1- area2

would change the store to look like



Subsequent execution of the statement input area2

would cause the processor to pick up the next value supplied at the input device and place it into area2 If this value were 13, then the store would become



in summary, the input statement has the form

Input variable1, variable2… variable N

Where variable1, variable2,…, variable N must be variable names that ha been previously declared in a variables statement. Upon encountering such a input statement, the processor retrieves the first value it can find at the input device and places it into variable1, the second value is placed into variable2 and so on. It is the responsibility of the programmer to ensure that there are enough values at the input device and that these values match the type (e.g. integer or real) of the variables into which they are to be read.

The Output Device

The output statement directs the processor to print messages and the values certain variables at the output device. we pretend that we can see variable values by drawing a diagram of the store and listing the contents of each storage cell. Of course, this is not possible when dealing with real computers. It is also not desirable. What we really want is to have just those values that interest us displayed at a screen or printed on paper. The output statement allows us to do this.

Example 3 : Assume that the store currently looks like

length	3.0	real
width	2.5	real
area	7.5	real

. .
. .
. .

If we wish the processor to print the value of area we use the statement

Output area

The processor will then print the value currently in the cell named area at the output device. That is,

7.5

will appear on the output device. If we wished to see the values of length, width, and area together, we could write

Output length, width, area

This would result in

3.0 2.5 7.5

being printed at the output device.

Thus we see how the output statement is used to print variable values at the output device. The form of the output statement is

Output variable1, variable2… variable N

The results printed on the output device as a result of executing an output statement of this form are

Value1 Value2 ... valueN

Where value1 is the value of variablel, value2 is the value of variable2, and so on.

One problem remains in our use of the output statement. Since just the variable values are printed, we have no way of correlating the values with they stand for. The output

3.0 2.5 75

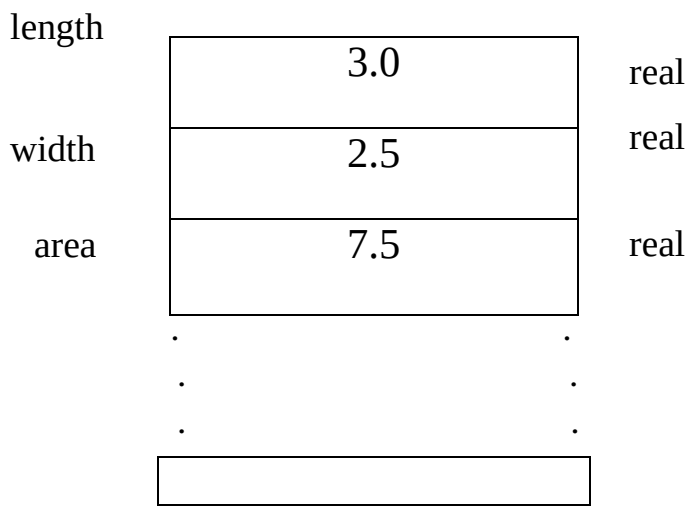
gives no clue to what these values mean. But, you might think, if we know the values were printed by

Output length.,width, area

Then we know that 3.0 is the length, 2.5 is the width, and 7.5 is the area. This is true it requires us to look back in our program to find the statement that printed these values, and this may not be a simple task if there are other output statements in the program.

Example 4:

If the store looks like



then the statement

Output ‘length ‘, length, ‘times width ‘,width,’ = ‘,area

when executed by the processor will cause

length 3.0 times width 2.5= 7.5

to be printed on the output device. The messages in the single quotation marks are printed just as they appear. Names not in quotes are assumed to be variables and their values are printed.

The output statement thus has the more general form

Output item 1, item2,…, item N

Where each item is either a message (any combination of characters) in single quotes or the name of a variable. Each message is printed literally, character by character. When a variable appears in the output statement, the value (not the

name) of the variable is printed.

At this point we should clarify two common misconceptions about the output statement. One concerns the use of messages to label variable values. Some beginning students believe that the computer can correlate the labels and the values. For example, the statement

Output ‘length ‘, length, ‘times width ‘, width, ‘= ‘area

was used in the previous example to demonstrate the use of message labels. We might think that somehow the computer knows that a word in quotes (e.g., ‘length’) corresponds to the variable length of the same name (e.g., length). This statement produced the output

length 3.0 times width 2.5 =7.5

as we saw earlier. On the other hand, the statement

Output ‘width’, length, ‘times length’, area,’ = ‘width

would cause

width 3.0 times length 7.5= 2.5

to be printed, and

Output ‘width’, area, times length ‘, width,’ = ‘length

would cause

width 7.5 times length 2.5 = 3.0

to appear at the output device. Of course, these last two printed messages make no sense to us, but the processor executes these output instructions without any problem it prints the first item in the output list, then the second item, then the third, and so on, just as it was instructed. Again we must stress this point: we as programmers are responsible for supplying meaningful and proper instructions to the processor.

A second misconception concerns the contents of the store after an output statement has been executed. Remember this: the output statement does not erase the variable values in the store when it prints them; it only prints a copy value. If the store looked like

length	3.0	real
width	2.5	real
area	7.5	real
	. . .	. . .

And the processor executed the statement

Output ‘length’, length, ‘times width ‘, width, ‘= ‘,area

Are printed

length 3.0 times width 2.5 = 7.5 .at the output device, the store would be unchanged:

length	3.0	real
width	2.5	real
area	7.5	real

Sequence Control

Control structures in a programming language provide the basic framework within which operations and data are combined into programs and sets of programs.

Implicit and Explicit Sequence Control

Sequence-control structures may be conveniently categorized in three groups:

1. Structures used in expressions (and thus within statements, since expressions form the basic building blocks for statements), such as precedence rules and Parentheses.

2. Structures used between statements or groups of statements, such as conditional and iteration statements.

3. Structures used between subprograms, such as subprogram calls and co routines.

Sequencing with Arithmetic Expressions

Consider the formula for computing roots of the quadratic equation:

Root = $- B \pm \sqrt{B^2 - 4 * A * C} \; /2*A$

This apparently simple formula actually involves at least 15 separate operations (assuming a square-root primitive and counting time various data references). Coded in a typical assembly or machine language it would require at least 15 instructions, and

probably far more. The formula for one of the roots can be coded as a single expression almost directly

ROOT = (-B + SQRT (B^2-4*A*C)) / (2*A)

The sequence-control mechanisms that operate to determine the order of operations within this expression are in fact rather complex and subtle

Tree-Structure Representation

The basic sequence-control mechanism in expressions is functional composition. An operation and its operands are specified the operands may he either constant, data objects, or other operations, whose operands in turn may be constants, data objects or still other operations, to any depth. Functional composition gives an expression the characteristic structure of a tree, where the root node of the tree represents the main operation, nodes between the root and the leaves represent intermediate-level operations and the leaves represent data references. For example, the expression for the quadratic for may be represented (using - to represent the unary minus operation) by the tree of Figure 1.

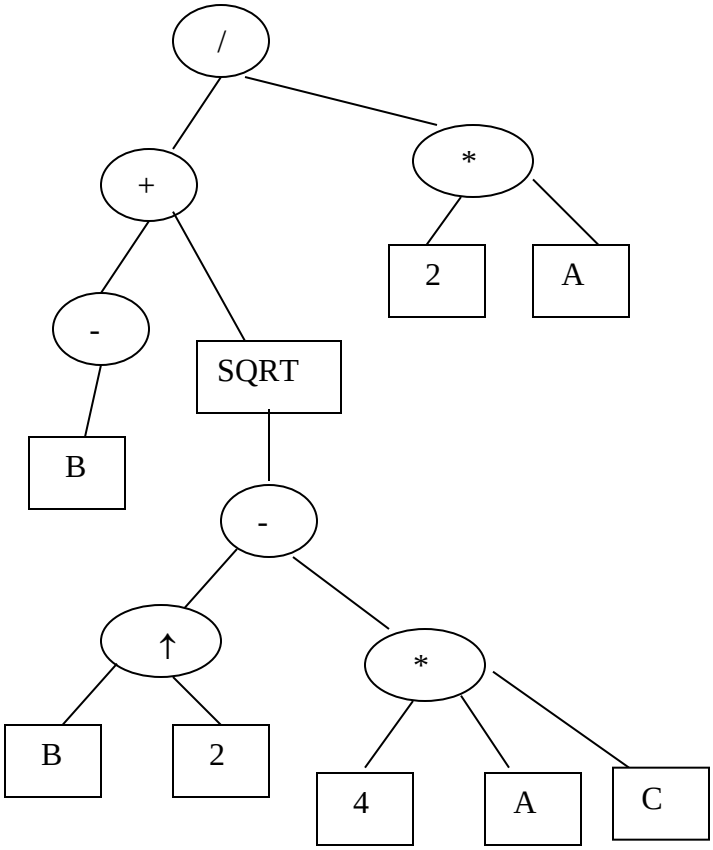


Figure 1: The tree representation of quadratic formula.

The tree representation leaves part of the order of evaluation undefined. For example, in the tree in Figure1 it is not clear whether B should be evaluated before or after B * *2, nor is it clear whether the two data references to the identifier B may be combined into a single reference.

Syntax for Expressions

If we take expressions as characteristically represented by trees, then in order to use expressions within programs some linearization of trees is required; i.e., one must have a notation for writing trees as linear sequences of symbols. Let us look at the most common notations:

1) Prefix (polish) notation. In prefix notation one writes the operation symbol first, followed by the operands in order from left to right. If an operand is itself an operation with operand, then the same rules applies. The tree of Figure 2 then becomes *+ a b- c a. Since + is a dyadic operator (requires two arguments), it is clear that the arguments for + are a and b. Similarly, the arguments for - must be c and a. Finally, the arguments for x must then be the + -term and the - -term. There is no ambiguity, and no parentheses were needed in specifying exactly how to evaluate the expression.

the expression for the quadratic for may be represented in prefix notation (using - to represent the unary minus operation) by the tree of Figure 1 as

/+-b √-↑ b 2 ** 4 a c *2 a

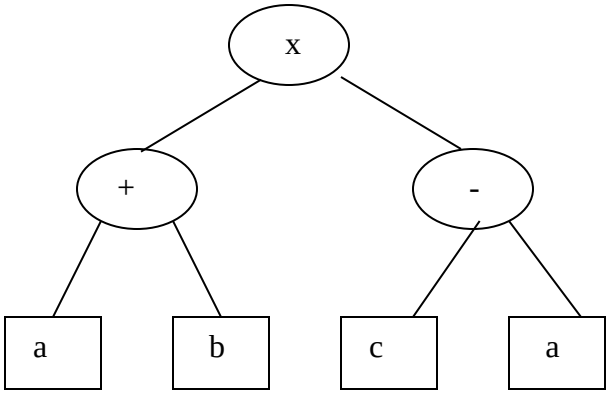


Figure 2. Tree form for simple expression: (a + b) * (c –a)

2) Postfix (suffix) notation. Postfix notation is similar to prefix notation except that the operation symbol follows the list of operands. For example, the expression in figure 2 is represented as a b + c a -*

3) **Infix notation**. Infix notation is most suitable for binary operations. in infix notation the operator symbol is written between the two operands. Because infix notation for basic arithmetic, relational, and logical operations is so commonly used in ordinary mathematics, the notation for these operations has been widely adopted in programming languages. In infix form the tree in figure 2 is represented as (a + b) * (c - a).

Forms of Statement-Level Sequence Control

three main forms of statement level sequence control are usually distinguish

1) Composition. Statements may be placed in a textual sequence so that they are executed in order whenever the larger program structure containing the sequence is executed.

2) Alternation. Two sequences of statements may form alternatives so that one or the other sequence is executed, but not both, whenever the larger program structure containing both sequences is executed.

3) Iteration. A sequence of statements may be executed repeatedly, zero or more times (zero meaning execution may be omitted altogether), whenever the larger program structure containing the sequence is executed.

Explicit Sequence Control

Early programming languages were modeled after the underlying actual machine that would execute the program. Since machines consisted of memory locations, early languages (e.g., FORTRAN) modeled these with simple data types directly translatable into machine objects (e.g., C float and Pascal real into hardware floating point, C int into hardware integer) and with simple statements consisting of labels and branches. The transfer of

control is most often indicated by use of a goto statement to an explicit statement with a given label name.

1) **Goto** statement. Two forms of **goto** statement are often present in many languages:

1.1) Unconditional goto. Within a sequence of statements, an unconditional **goto**, such as:

goto NEXT

transfers control to the statement labeled NEXT. The statement following the **goto** is not executed as part of the sequence.

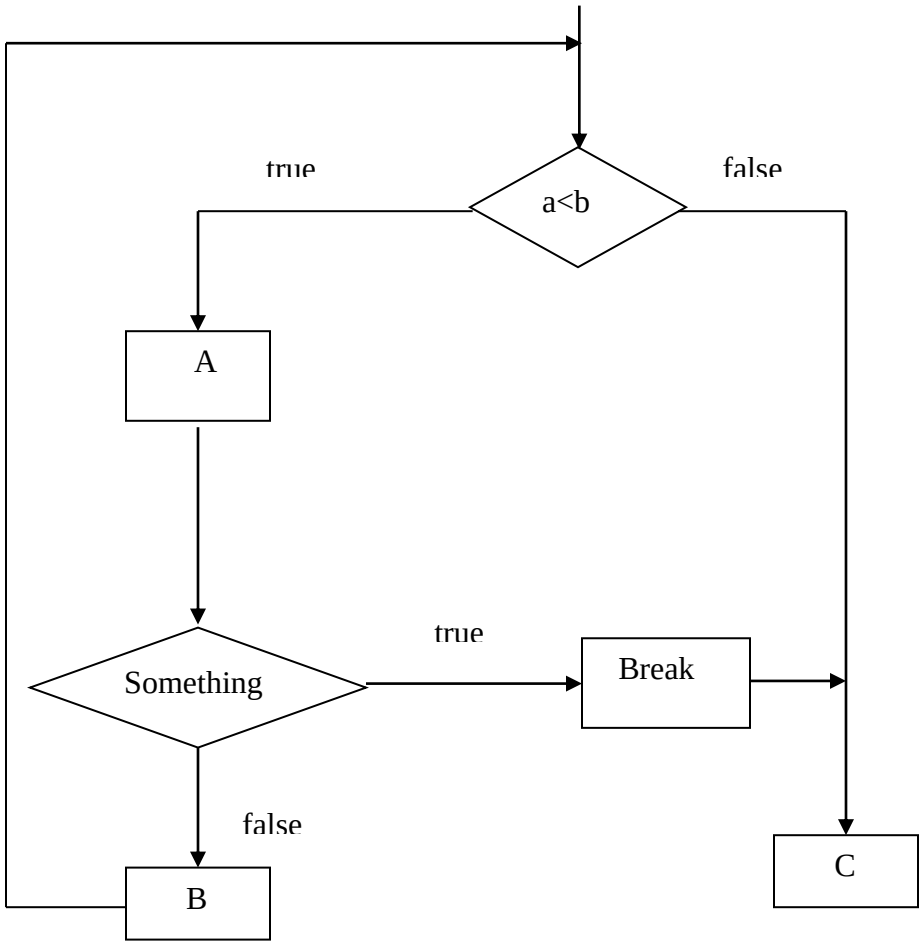
1.2) Conditional goto. Within a sequence of statements, a conditional goto such as:

if A = 0 then goto NEXT

transfers control to the statement labeled NEXT only if the specified condition holds.

2) Break statement. Some languages, such as C, include a **break** statement as a form of structured explicit control. Usually the **break** causes control to move forward in the program to an explicit point at the end of a given control structure. Thus **break** in C causes control to exit the immediately enclosing while, for, or switch statement. This still gives a one-in, one-out control structure that permits formal properties of a program to be developed as in figure 3

```
while (a<b)
{…
A;
…
if (something)
break;
…
B;
…
}
c;
```



a) syntax (b) Execution of break statement

Figure 3: Structured break statement

Structured sequence control

Most languages provide a set of control statements for expressing the basic control forms of composition, alternation, and iteration.

1) Compound Statements

A compound statement is a sequence of statements that may be treated as a single statement in the construction of larger statements. Often a compound statement is written:

Begin

… Sequence of statements (one or more)

End

In C it is written simply as {. ..}

Within the compound statement, statements are written in the order in which they are to be executed. Thus the compound statement is the basic structure for representing the composition of statements. Because a compound statement is itself a statement, groups of statements representing single conceptual units of computation may be kept together as a unit by the **begin ... end** bracketing, and hierarchies of such groups may be constructed.

A compound statement is implemented in a conventional computer by placing the blocks of executable code representing each constituent statement in sequence in memory. The order in which

they appear in memory determines the order in which they are executed.

2) Conditional Statements

A conditional statement is one that expresses alternation of two or more statements, or optional execution of a single statement: where statement means either a single basic statement, a compound statement, or another control statement. The choice of alternative is controlled by a test on some condition, usually written as an expression involving relational and Boolean operations. The most common forms of conditional statement are the **if** and **case** statements.

1) If statements. The optional execution of a statement is expressed as a single branch **if**:

if *condition* **then** *statement* **endif**

while a choice between two alternatives uses a twobranch **if**:

if *condition* **then** *statement*1**else** *statement* 2 **endif**

In the first case, a condition evaluating to true causes the statement to be executed, while a false condition causes the *statement* to be skipped. In the two branch **if**, *statement*1 or *statement*2 is executed depending upon whether condition true or false.

A choice among many alternatives may be expressed by nesting additional **if:**

if *condition1* **then** *statement1*

else if *eondifion2* **then** *statement2*

…

else if *eondifion n* **then** *statement n*

else *eondifion n+1* **then** *statement n+1*

2) Case statements. The conditions in a multibranch if often take the form of repeated testing of the value of a variable, such as:

if Tag = 0 **then** *statement0*

else if Tag = 1 **then** *statement1*

elseif Tag = 2 **then** *statement2*

else *statement3*

endif

This common structure is expressed more concisely as a case statement, such as

case Tag is

when 0 => begin

statement0

end;

when 1 => begin

statement1

end:

when 2 => begin

statement2

end:

when others => begin

statement3

end;

end case

3) Iteration Statements

The basic structure of an iteration statement consists of a head and a body. The head controls the number

of times that the body will be executed, while the body is usually a (compound statement that provides the action of the statement. Although the bodies of iteration statements are fairly unrestricted, only a few variants of head structure are usually used. Let us look at some typical ones.

3.1 Simple repetition. The simplest type of iteration statement head specifies that the body is to be executed some fixed number of times. Many languages are typical of this construct:

perform **body k times**

which causes K to be evaluated and then the body of the statement to be executed that many times.

3.2) Repetition while condition holds. A somewhat more complex iteration may constructed using a repeat while head. A typical form is:

while *test* **do** *body*

In this form of iteration statement the test expression is reevaluated each times after the body has been executed.

3.3 Repetition while incrementing a counter. The third alternative form of iteration statement is the statement whose head specifies a variable that serves as a counter or index during the iteration. An initial value, final value, and increment are specified in the head, and the body is executed repeatedly using first the initial value as the value of the index variable, then the initial value plus the increment, then initial value plus twice the increment, and so on, until the final value is reached.

Either **For** i=1 **to** final value **do** *body(increment)*

Or **For** i= final value down 1 **do** *body(decrement)*

3.4 indefinite repetitions. Where the condition for loop exit are complex and not easily expressible in the usual loop head, a loop with no explicit termination test in the head is pften used as

loop

…

exit when *condition*

…

end loop

Some C sample iteration loops can be specified as

1) Simple counter from 1 to 10

for (i=1;i<=10;i++){*body*}

2) Infinite loop

for (;;){*body*}

3) Counter with exit condition

for (i=1;i<=10 && Not End file ;i++){*body*}

Lecture Ten
Introduction to Structured Programming

1- Introduction programming:

One of the most important concepts of programming is the ability to control a program so that different lines of code are executed or that some lines of code are executed many times. The mechanisms that allow us to control the flow of execution are called control structures. Flowcharting is a method of documenting (charting) the flow (or paths) that a program would execute. There are four main categories of control structures:

- **Sequence:** Very boring. Simply do one instruction then the next and the next. Just do them in a given sequence or in order listed. Most lines of code are this.
- **Selection**: This is where you select or choose between two or more flows. The choice is decided by asking some sort of question. The answer determines the path (or which lines of code) will be executed.
- **Iteration**: Also known as repetition, it allows some code (one too many lines) to be executed (or repeated) several times. The code might not be executed at all (repeat it zero times), executed a fixed number of times or executed indefinitely until some condition has been met. Also known as looping because the flowcharting shows the flow looping back to repeat the task.
- **Branching**: A control structure that allows the flow of execution to jump to a different part of the program. This category is rarely used in modular structured programming.

1- Selection Control Structures: The basic attribute of a selection control structure is to be able to select between two of more alternate paths. This is described as either **two-way selection** or **multiway selection**.

We have mentioned that the **if then else** control structure belongs to the selection category and is a **two-way selection**.

Example: if then else control structure

```
 if (age > 17)
{
out << "You can vote.";
}
else
{
cout << "You can't vote.";
}
```

2- Iteration Control Structures: The basic attribute of an iteration control structure is to be able to repeat some lines of code. The visual display of iteration creates a circular loop pattern when flowcharted, thus the word **"loop"** is associated with iteration control structures. Iteration can be accomplished with **test before loops**, **counting loops**, and **test after loops.**

We have mentioned that the **while** control structure belongs to the iteration category and is a test before loop.

Example: while control structure

```
    counter = 0;
while (counter < 5)
{
cout << "\nI love computers!";
counter ++;
}
```

Definitions:

Control structures: Mechanisms that allow us to control the flow of execution within a program.

Sequence: A control structure where you do the items in the sequence listed.

Selection: A control structure where you select between two or more choices.

Iteration: A control structure that allows some lines of code to be executed many times.

Branching: A control structure that allows the flow of execution to jump to a different part of the program.

Structured programming: A method of planning programs that avoids the branching category of control structures.

2- Algorithm

The first part of the problem-solving process is understanding the problem. This consists of determining the input (what is given), the output what is to be produced), and the relationship between the two. For ex ample, if the task is to produce a payroll, it is necessary to know the employees' identifications; means of determining wages (hourly pay or salary); deductions and other factors (overtime pay rates, bonuses, taxes); what is to be produced (checks, year-to-date summaries, etc.); and finally how the givens are used to produce the results. This is a matter of analysis. At this stage it is essential that the person asking for a program and the person who will be writing the program understand each other.

From this example, the need for understanding is obvious. It is also necessary for programmers to communicate with each other. In the early stages of a program's development, this is not done with a program. Usually some less formal means are used for communication.

3- Pseudocode (structured English, algorithmic language, program design language or PDL) is an adaptation of English to express algorithms.

After the programmer understands the problem, he or she begins to formulate an algorithm for the solution of the problem. This is a matter of synthesis. English is generally inadequate for the expression of the algorithm, so pseudocode is used. An example of an algorithm in pseudocode is given in figure 1.

```
Begin Weekly Pay
Get EmployeeName, HoursWorked, HourlyRate
If HoursWorked > 40 Then { Overtime, bonus 1.5 }
 Pay ←— HourlyRate * 40 +1.5 * HourlyRate * (HoursWorked - 40)
Else { Regular Pay }
Pay ←— HourlyRate * HoursWorked
Print EmployeeName, Pay
End Weekly Pay
```

Figure 1: Algorithm for Computing Weekly Pay for Hourly Employee

This algorithm is designed to compute an employee's pay. A leftward arrow indicates that the quantity computed to the right of the arrow is given to the quantity on the left. Since any hours worked over 40 hours are considered overtime, it is necessary to test whether this has happened and compute the pay accordingly. As the algorithm is developed it is necessary to examine what is emerging. It may become evident that some necessary pieces of information are missing. Then the programmer returns to the problem step and may ask for more instructions from the program requestor.

When the programmer is convinced that the algorithm is complete, it is time to translate the algorithm into a program.

Coding is the process of transforming an algorithm into a program in a specific computer language.

A bug (in a computer program) is an error, usually one that is difficult to correct.

It is sometimes claimed that if the algorithm is constructed well enough, coding should be a relatively easy matter.

However, by testing a program, bugs frequently come to light and are eliminated. For example, we might mistakenly have typed or written 400 instead of 40 in the algorithm above in the line computing Pay (as in figure 2). Some test data probably would have shown that there was an error in the algorithm. (If this error was not caught the firm would issue some very large paychecks.) This might seem to be an unlikely and obvious error, but many rather minor typing errors cause major problems with programs and sometimes are not very easily uncovered.

```
Begin Weekly Pay

Get EmployeeName, HoursWorked, HourlyRate

If HoursWorked > 40 Then { Overtime, bonus 1.5 }

 Pay ←— HourlyRate * 400+1.5 * HourlyRate * (HoursWorked - 40)

Else { Regular Pay }

Pay ←— HourlyRate * HoursWorked

Print EmployeeName, Pay

End Weekly Pay
```

Figure 2: Algorithm for Computing Weekly Pay for Hourly Employee

4- Flowchart

The flowchart method is the oldest, most controversial, most widely used, and most misunderstood tool of program design. It is almost certain that each reader will have used flowcharts and have formed an opinion about them. We will begin by defining what a flowchart is and showing how it can be used as a design and documentation tool.

The standard symbols generally employed in flowcharting are given in Figure3. As an example, flowcharts are used to design a program to compute the heading-angle response of a ship to a step input of left rudder, follows by an equal step input of right rudder

One of the legitimate problems with using flowcharts is that when the code is changed, we often have to change the flowchart as well. If we abstain as was recommend from dealing with detailed flowcharts, then not all the changes in code will necessitate changes in the flowchart. All those code changes which impact the higher-level flowchart, however, will necessitate redrawing. This is analogous to the situation of the mechanical or electrical designer who has to change the mechanical drawing or electrical schematic whenever a significant change is made.

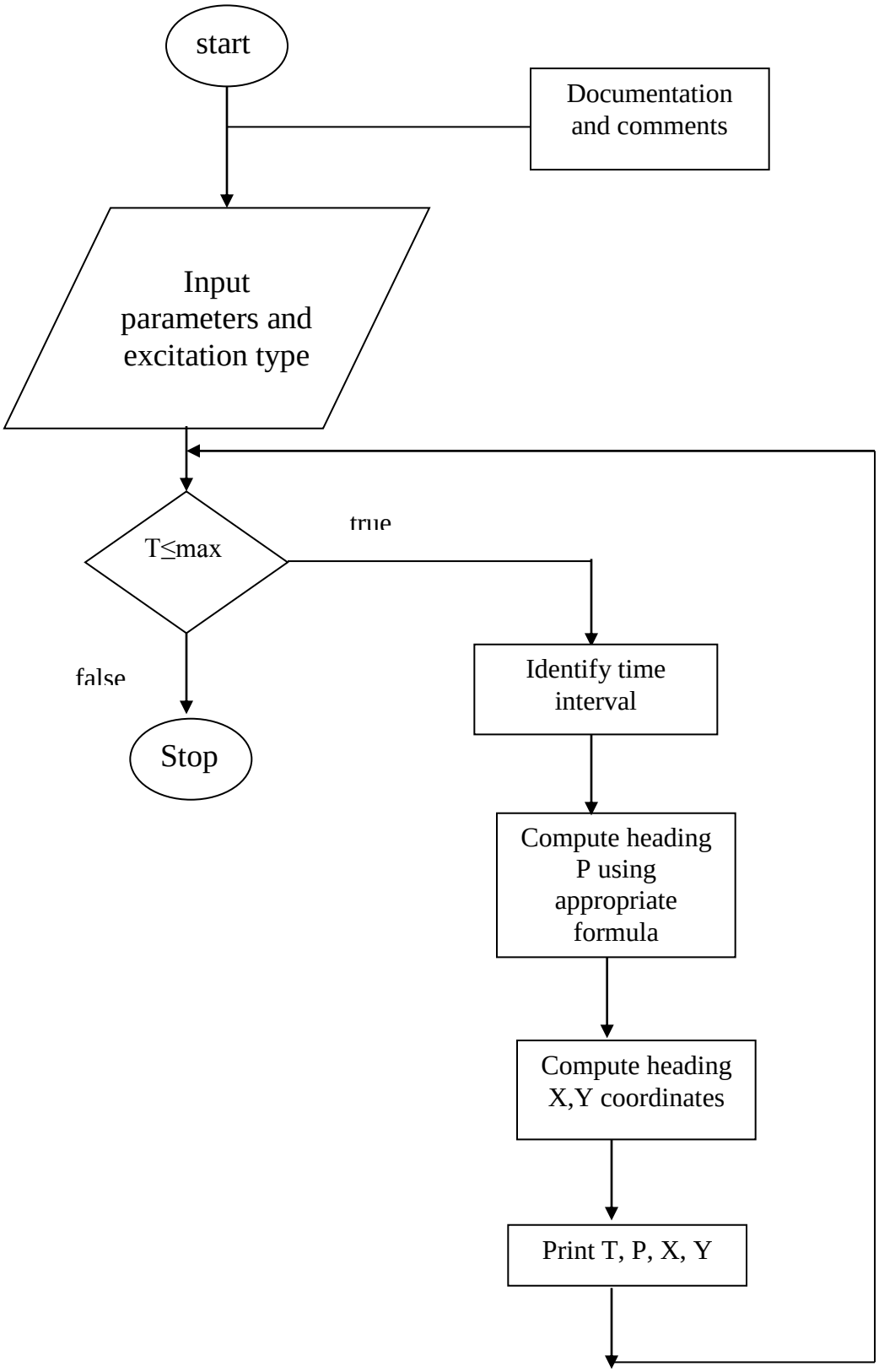


Figure 3: higher level flow chart

Pseudo-code

Suppose we were faced with the task of writing a 25-page instruction manual describing the use of a text-editing system. We would start the design with an outline. Then, we could begin to write specific sections immediately; however, a better plan would be to annotate the outline with a few brief sentences or a list of topics contained in each section. These notes could be written in standard English sentences, or abbreviated to just a few words, phrases, and mathematical symbols. Pseudo-code (sometimes called metacode) is a shorthand notation for the control structures and certain other elements of a programming language

A designer who is best versed in a particular programming language will probably write pseudo-code that is an abbreviated version of that language; however, it should still be possible to implement the design in any language desired. An example of the pseudo-code for a program which reads text records, identifies different words, counts the frequency of occurrence, and prints out a table of this information is given in figure 4.

```
Initialize the program
Read the first text record
Do while there are more lines in the text record
  Do while there are more words in the text record
    Extract the next text word
    Search the word-table for the extracted word
    If the extracted word is found
      Increment the word's occurrence count
    Else
```

```
        Insert the extracted word into the table
      End if
    Increment the words-processed count
   End do at the end of the text record
  Read the next text record
End do when all text records have been read
Print the table and summary information
Terminate the program
```

Figure 4: Pseudo-code for a word-frequency counting program

Example 1:-suppose we have an algorithm

```
Count:      procedure options (main);
/* this program counts positive numbers,            */
 /*negative numbers, and sums positive numbers. */
/* it stops if either input is zero or if the sum exceeds 1000.*/

k = 0; L=0 ;  total 0;    /*k counts positive numbers */
                          /*L counts negative numbers */
Repeat:  Get list (A);
If A = 0 then go to print;        /*if A = 0 we are finished */
If A >0 then go to update;       /*for a positive A we must increase
                                          k and total */
L=L+1                             /* increase negative count */
Go to repeat;
Update:    k=k+ 1;
Total =total + A;
If  total < 1000 then go to repeat;     /*repeat if not done*/
```

```
Print:   put list (k, L, total);          /*results of the program*/
End counts;
```

Figure 5: program algorithm

The flowchart (Figure5) has two exits from the main program loop (GETA, A=0, A>0, L=L+l, GETA...). The first exits if A=0 and the second is if TOTAL> 1000. Thus, the single-entry and single-exit criteria of classical structured programs are also violated.

We can easily design a structured solution to this problem by using IF THEN ELSE.

Example 1:-suppose we have an algorithm

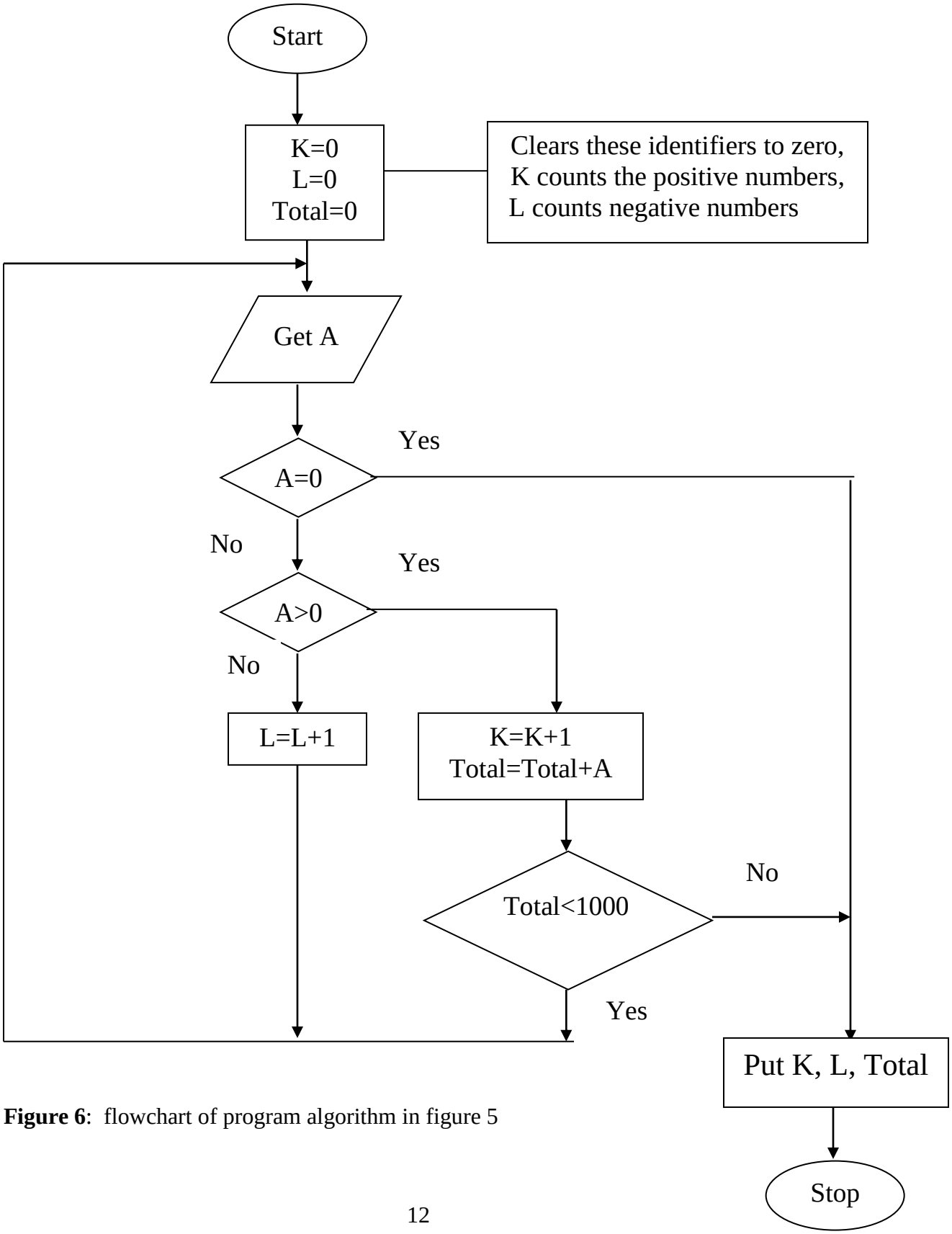


Figure 6: flowchart of program algorithm in figure 5

Example 2:-suppose we have the flow chart bellow that print the six numbers from 0 to 5 depend on selection statement

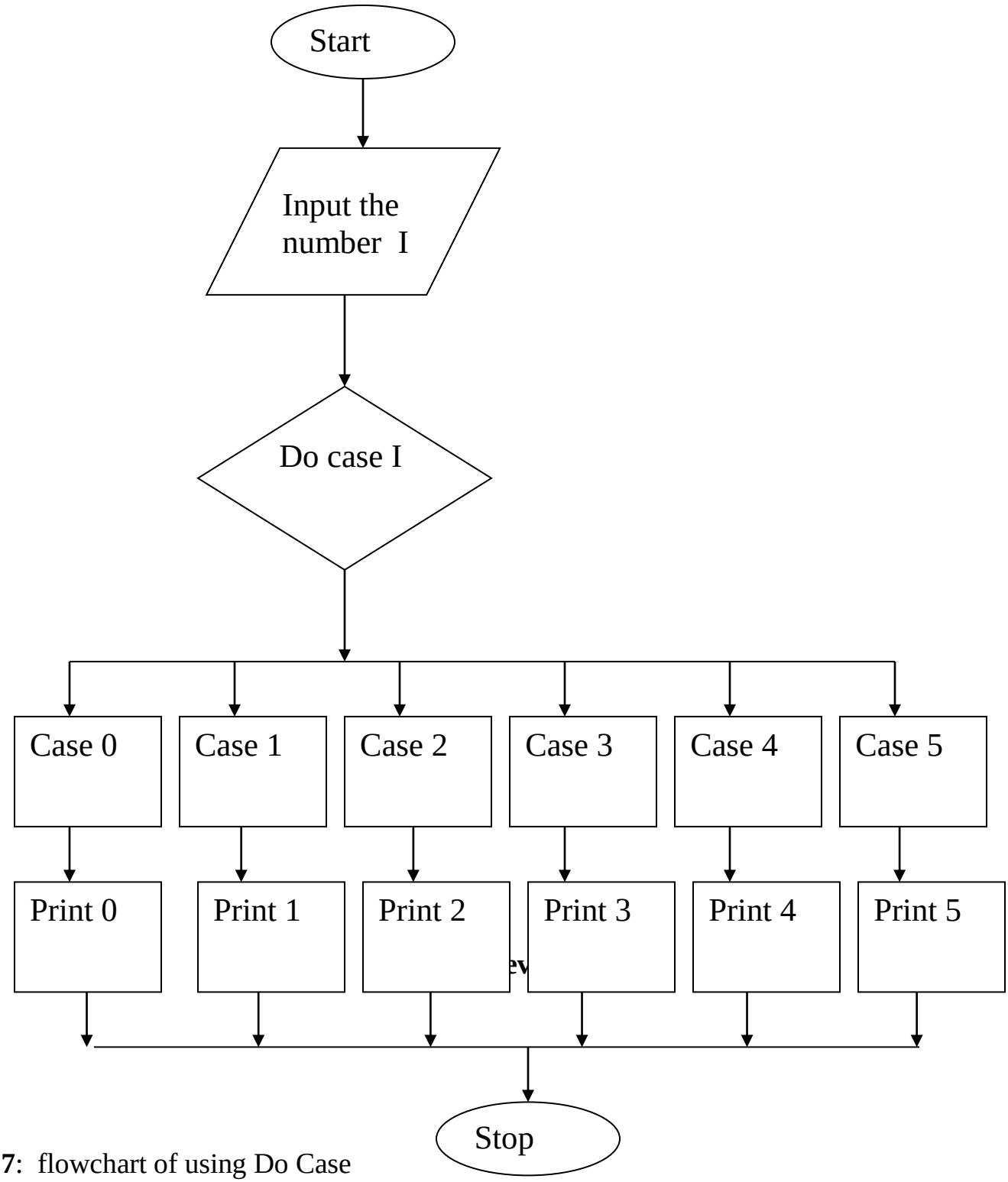


Figure 7: flowchart of using Do Case