B.A Part-2 (Sem-3) **Paper: BAP-201**

Computer Science

C Programming and Data Structures

UNIT No.1

Center for Distance and Online Education, PunjabiUniversity,Patiala

Lesson No:

(Syllabus)

BAP-201 C Programming and Data Structures

Max Marks: 60 Maximum Time: 3 Hrs.
Min Pass Marks: 35%

A) Instructions for the paper setter

The question paper will consist of three sections: A, B & C. SECTIONs A & B will have four questions each from the respective sections of the syllabus and will carry 40% marks each. SECTION C will have 6-12 short-answer type questions which will cover the entire syllabus uniformly and will carry 20% marks in all.

B) Instructions for the candidates

1. Candidates are required to attempt two questions each from the sections A & B of the question paper and the entire section C.
2. Use of non-programmable calculator is allowed.

Section - A

Overview of C Language: C Fundamental: Introduction to C, character set, identifier, keywords, data types, constants, variable, user defined data type, arithmetic, unary, relational, logical, assignment and conditional operators and expression, Basic structure of a C Program, Data I/O statement: single character I/O, formatted I/O, String I/O functions.
Control Structure: sequencing, alteration (if-else, switch, break, continue, goto, iteration while, do-while, for) and nested loops.
Functions: defining and accessing a function, passing arguments to a function, specifying arguments data types, function prototypes, recursion.
Storage Classes; Automatic, External, Static, Register
Pointer and Structures: Character pointer, pointer to arrays, array of pointers, Structures and union: Defining and processing structure, Unions, Preposser directives.

Section - B

Basic Notations and Arrays (Data Structure): Basic concepts and notations, data structures, types of data structures and data structures operations, mathematic notations and functions, algorithmetic complexity, Big 'O' notation and time-space tradeoff, Arrays: Linear Array, respresentation of linear array in memory, traversing linear array, insertion and deletion in an array, Multi-dimensional array: Row-Major, Column Major order, space array.
Stacks: Push and Pop in stack, representation of stack in memory (Using Arrays)
Queues: Insertion and deletion operations
Searching Techniques: Linear and Binary search
Sorting Techniques: Insertion sort, selection sort, bubble sort, merge sort, quick sort.

Text Books:

1. Byron Gotfried, Programming with C, 2nd edition, Schaum's outline series
2. Shubhnandan S. Jamwal, Programming in C, Pearson education.

Reference Books:

1. Seymour Lipschutz, Theory and Practice of Data Structures, McGrawHill, 1998
2. B.W. Kernighan and D.M.Ritchie, The C Programming Language, PHI.

BA PART-2 (SEM-3) **PAPER : BAP-201**

C Programming and Data Structures

LESSON NO. 1 **AUTHOR : DR. KANWAL PREET SINGH**

CONVERTED INTO SLM BY: DR. VISHAL SINGH

PROBLEM ANALYSIS

1.1 Objectives

In this lesson, we will discuss how to analyse the problem for which we want to write the program. You will also see how to write an algorithm and draw a flowchart for a given problem.

1.2 Introduction

A computer is a machine that receives instructions and produces a result after performing an appropriate assignment. Since it is a machine, it expects good and precise directives in order to do something. The end result depends on various factors ranging from the particular capabilities of the machine, the instructions it received, and the expected result. As a machine, the computer cannot figure out what you want. The computer doesn't think and therefore doesn't make mistakes.

Computer programming is the art of writing instructions (programs) that asks the computer to do something and give a result. A computer receives instructions in many different forms, four of which are particularly important.

The first set of instructions is given by the manufacturers of various hardware parts such as the microprocessor, the motherboard, the floppy and the CD-ROM drives, etc. These parts are usually made by different companies setting different and various goals that their particular part can perform. The instructions given to the microprocessor, for example, tell it how to perform calculations, at what speed and under which circumstances. The instructions given to the motherboard allows information to flow from one section of the computer to another.

Once the instructions given to the hardware parts are known, software engineers use that information to give the second set of instructions to the computer. These instructions, known as an operating system are usually written by one company. These second instructions tell the computer how to coordinate its different components so the result will be a combination of different effects. This time, the computer is instructed about where the pieces of information it receives are coming from, what to do with them and then where to send the result. Some of the operating systems in the market are: Microsoft Windows XP, Apple Macintosh, Red Hat Linux etc. A particular OS (for example Microsoft Windows XP) depending on a particular processor (for example Intel Dual Core) is sometimes referred to as a platform. Some of the computer languages running on Microsoft Windows operating systems are C,C++, Java and their variants.

The actual third set of instructions is given to the computer by you, the programmer, using one or more of the languages that the operating system you are planning to use can understand. Your job is going to consist of writing applications. As a programmer, you write statements such as telling the computer, actually the operating system, that "If the user clicks this, do the following, but if he clicks that, do something else. If the user right clicks, display this; if he double-clicks that, do that." To write these instructions, called programs, you first learn to "speak" one of the languages of the OS. Then, you become more creative. Some of the application programs in the market are Microsoft Word, Microsoft Excel, Adobe Acrobat, etc.

The last instructions are given by whoever uses your program, or your application. For example, if you had programmed Microsoft Word, you would have told the computer that "If a user clicks the New button on the Standard toolbar, I want you to display a new empty document. But if the user clicks File -> New..., I want you to 'call' the New dialog and provide more options to create a new document. If the same user right-clicks on any button on any of the toolbars, I want you to show, from a popup menu, all the toolbars available so she can choose which one she wants. But if she right-clicks on the main document, here is another menu I want you to display."

Your interest here is on the computer languages, since you are going to write programs. There are various computer languages, for different reasons, capable of doing different things. Fortunately, the computer can distinguish between different languages and perform accordingly. These instructions are given by the programmer who is using compilers, interpreters, etc, to write programs. Examples of those languages are C, C++, Java, etc.

1.3 Problem Analysis

A computer can be used to solve a problem by following a set of stored instructions called the program. The problem solving process needs initial data, the operations that are to be performed and results in the form of output. The following steps are required for problem solving:

Define the problem:

The first step is to give a clear concise problem statement. The problem definition should clearly specify the desired input and output. This step demands that user should have full knowledge of the background of the problem. A stated goal will help in the organization of the remaining steps.

Examples of simple problems can be:

- To find average of two numbers.
- To determine a student's final grade and indicate whether it is passing or failing. The final grade is calculated as the average of four marks.

Develop an Algorithm:

The next step is to devise and describe a precise plan of what you want the computer to do. This plan, expressed as a sequence of steps to be taken or operations to be performed is called an **algorithm**. The algorithm works by breaking the process into a number of steps which are smaller and simpler than the entire process. Further the sub-algorithms can themselves be broken into a number of steps.

Write a code based on the algorithm using a programming language:

The computer cannot understand the above written algorithm. It must be written in a programming language which can be understood by the computer. Once the program is ready, it can be translated into machine code using a compiler or interpreter. When the machine code version of the program is ready, it can be executed on a computer.

Testing the Program:

If the program compiles correctly, use a simple set of test values to verify that the result is what you expected. If the results seem valid, test the program with a variety of real data sets. The mistakes if encountered in a program are called bugs and debugging the code may take longer than writing the code.

1.4 Algorithms

An algorithm is a detailed sequence of simple steps that are needed to solve a problem. It is an effective procedure for solving a problem in a finite number of steps. It is *effective*, which means that an answer is found and it finishes, that is it has a *finite* number of steps. A well-designed algorithm will always provide an answer, it may not be the answer you want but there will be an answer. A well-designed algorithm is also guaranteed to terminate.

The computers lack intuition or common sense to realize the full procedure for solving a problem. Therefore the programmer must describe the step by step procedure for solving a problem to the computer minutely. This sequence of steps written in a simple language, form an algorithm. **The characteristics of a good algorithm are:**

- An algorithm should have zero or more input.
- An algorithm should exhibit at least one output.
- An algorithm should be finite.
- Each instruction in an algorithm should be defined clearly.
- Each instruction used in an algorithm should be basic and easy to perform.

Following are some simple examples of algorithms:

Problem1: Write an algorithm to find the average of two numbers.

Algo 1:

1. Input the first value in x.
2. Input the second value in y.
3. Add the two numbers and put result in sum.
4. Divide sum by 2 and put the result in average.
5. Output value of average.

This is simple problem. So we write the complete algorithm in one go. But the problems may be complex also. In that case, we will first write an initial algorithm and then we refine that algorithm further. The second algorithm is a bit more complex and we use three stages to write the final algorithm.

Problem2: Write an algorithm for withdrawing money for a bank ATM.

Algo 2:

Stage1:

```
1.      Display a message asking how much money is to be withdrawn
2.      Input the withdrawal amount
3.      Deduct the withdrawal amount from the balance
4.      Give withdrawal amount of cash
5.      Stop
```

Stage2:

```
1.      Display a message asking how much money is to be withdrawn
2.      Input the withdrawal amount
3.      If the withdrawal amount is greater than the balance then
           Print "Insufficient funds."
        Otherwise
           Deduct the withdrawal amount from the balance
4.      Give withdrawal amount of cash
5.      Endif
6.      Stop
```

Stage3:

```
1.      Repeat
2.      Display a message asking how much money is to be withdrawn
3.      Input the withdrawal amount
4.      If the withdrawal amount is greater than the balance then
           Output "Insufficient funds."
        Otherwise
           Deduct the withdrawal amount from the balance
5.      Give withdrawal amount of cash
6.      Endif
7.      End Repeat
8.       Stop
```

Different ways of stating algorithms

One way of stating an algorithm has already been shown above. Let us call this the Step-Form. There can be three different ways of stating algorithms:

1. Step-Form
2. Pseudocode
3. Flowchart

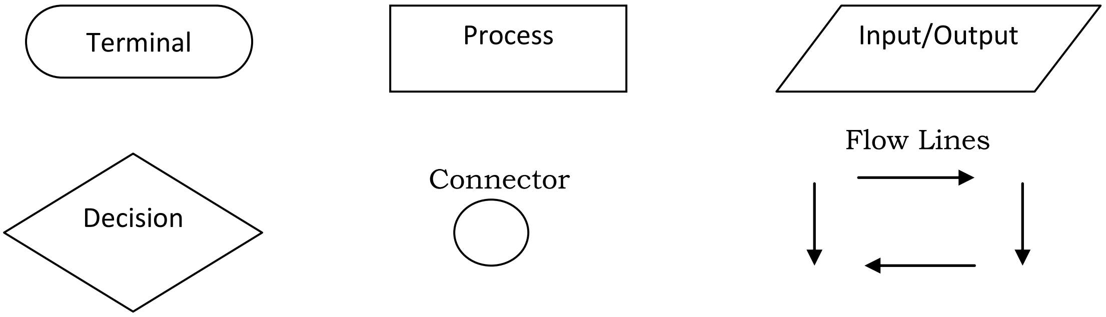
The first two are written forms. The above algorithms are in Step-Form and as you saw with the Step-Form (SF) the written form is just normal language. A problem with human language is that it can seem to be imprecise. In terms of meaning, what I write may not be the same as what you read. Pseudocode is also human language but tends toward more precision by using a limited vocabulary. The last one is graphically-oriented, that is it uses symbols and language to represent sequence, decision and repetition. We will be discussing only the Step-Form and the Flow Charts here.

1.5 Flow Charts

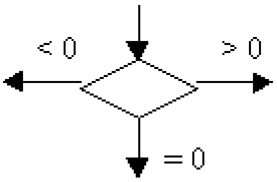
A flowchart is a graphical representation of an algorithm. A flowchart illustrates the steps in a process. By visualizing the process, a flowchart can quickly help identify bottlenecks or inefficiencies where the process can be streamlined or improved. A flowchart is a diagrammatic representation that illustrates the sequence of operations to be performed to get the solution of a problem. Flowcharts are generally drawn in the early stages of formulating computer solutions. Flowcharts facilitate communication between programmers and business people. These flowcharts play a vital role in the programming of a problem and are quite helpful in understanding the logic of complicated and lengthy problems. Once the flowchart is drawn, it becomes easy to write the program in any high level language. Often we see how flowcharts are helpful in explaining the program to others. Hence, it is correct to say that a flowchart is a must for the better documentation of a complex program. A flowchart can be compared to the blueprint of a building. As we know a designer draws a blueprint before starting construction on a building. Similarly, a programmer prefers to draw a flowchart prior to writing a computer program. As in the case of the drawing of a blueprint, the flowchart is drawn according to defined rules and using standard flowchart symbols prescribed by the American National Standard Institute (ANSI).

Flowchart Symbols

Flowcharts are usually drawn using some standard symbols; however, some special symbols can also be developed when required. Some standard symbols, which are frequently required for making flowcharts are shown below:



1. **Terminal:** The terminal symbol is used to begin and end each flow chart. The starting terminator has the word *start* while the ending terminator usually has the word *stop*.
2. **Process:** A processing symbol is used to represent arithmetic and data movement instructions. Hence, all the arithmetic processes of addition, subtraction, multiplication and division are shown by this symbol. Any type of assignment is also done in this box.
3. **Input/Output:** This box shows interaction with an outside entity. This may represent data being input into the algorithm or information being displayed to an outside entity.
4. **Decision or Condition:** This box is used to indicate a point at which decision has to be made, and a branch to one of the two or more alternative points is possible. The following figure shows the decision box with three alternative paths:



Depending on the condition one of the three paths will be followed i.e if the value generated by decision box is less than 0, the left path will be followed, if the value generated by decision box is greater than 0, the right path will be followed and if the value generated by decision box is equal to 0, the down path will be followed.

5. **Connectors:** A connector is used as a link between parts of a flowchart if a flowchart is large and cannot fit in a single page. We can also use it to represent a point at which the flowchart connects with another process. The name or reference for the other process should appear within the symbol.
6. **Flow Lines:** Flow lines with arrow heads are used to indicate the flow of operation, that is, the exact sequence in which the instructions are to be executed. The normal flow of flowchart is from top to bottom and from left to right.

Guidelines for Drawing a Flowchart

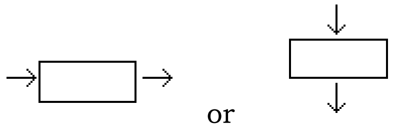
The following are some guidelines in flowcharting:

a. In drawing a proper flowchart, all necessary requirements should be listed out in logical order.

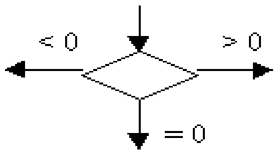
b. The flowchart should be clear, neat and easy to follow. There should not be any room for ambiguity in understanding the flowchart.

c. The usual direction of the flow of a procedure or system is from left to right or top to bottom.

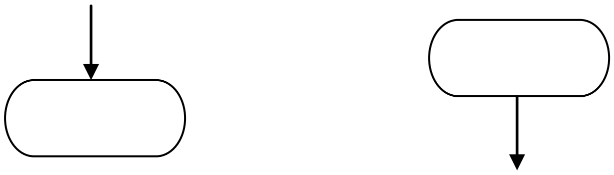
d. Only one flow line should come out from a process symbol.



e. Only one flow line should enter a decision symbol, but two or more flow lines, one for each possible answer, should leave the decision symbol.



f. Only one flow line is used in conjunction with terminal symbol.



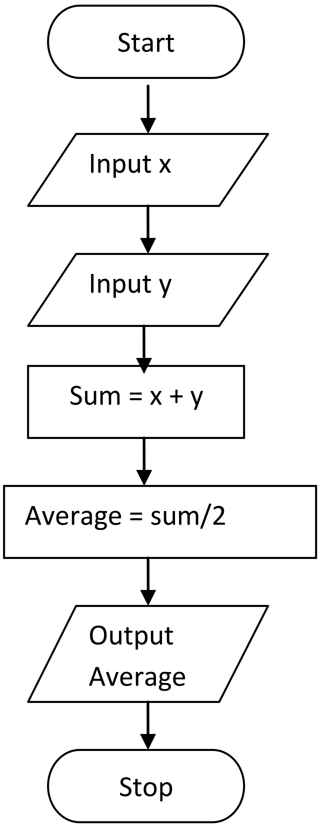
h. If the flowchart becomes complex, it is better to use connector symbols to reduce the number of flow lines. Avoid the intersection of flow lines if you want to make it more effective and better way of communication.

i. Ensure that the flowchart has a logical start and finish.

j. It is useful to test the validity of the flowchart by passing through it with a simple test data.

1.6 Sample Flowcharts

Example 1: Draw a flowchart to find the average of two numbers.
We have already written the algorithm for this problem. The flowchart is as follows:



Decisions (Switching logic)

Switching logic consists of two components - a condition and a *goto* command depending on the result of the condition test. The computer can determine the *truth value* of a statement involving one of six mathematical relations symbolized in the table below:

Symbol	**Meaning**
= =	Equals
!=	Not Equal
<	Less Than
<=	Less Than Equal to
>	Greater Than
>=	Greater Than Equal to

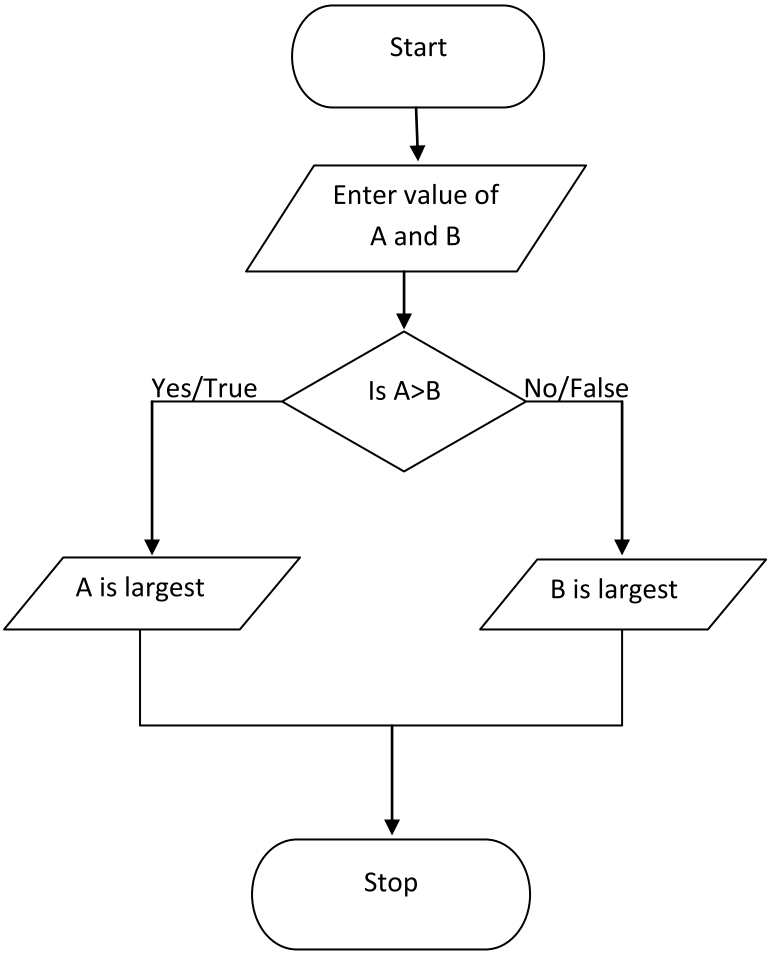
In practice, the computer is presented not with a true/false statement, but with a question having a "Yes" or "No" answer, for example if A = 10, B = 20, K = 5 and SALES = 10000, then:

Condition (Question)	**"Answer"**
Is A == B?	No

Is B > A?	Yes
Is K <= 25?	Yes
Is SALES >= Rs.5000.00?	Yes

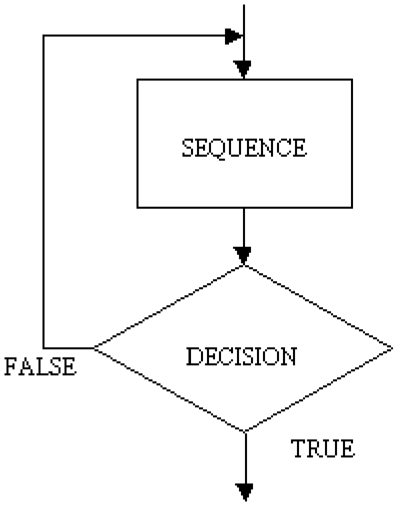
With each question, the computer can be programmed to take a different course of action depending on the answer. **A step in an algorithm that leads to more than one possible continuation is called a decision.** In flowcharting, the diamond-shaped symbol is used to indicate a decision. The question is placed inside the symbol and each alternative answer to the question is used to label the exit arrow which leads to the appropriate next step of the algorithm. The decision symbol is the only symbol that may have more than one exit.

Example 2: Draw a flowchart to find the larger of two numbers.



Loops

Most programs involve repeating a series of instructions over and over until some event occurs. This process of repeating a certain part of the program again and again until some condition is satisfied is called looping. For example, if we wish to read ten numbers and compute the average, we need a loop to count the number of numbers we have read. Consider the following figure:



This figure shows how we can implement loops in a flowchart. Note that the sequence is followed by a decision box and one branch of the decision box leads to the top of the sequence.

So, till the condition stated in decision box remains false, the control will be shifted to the top of the sequence and the sequence will be executed again and again.

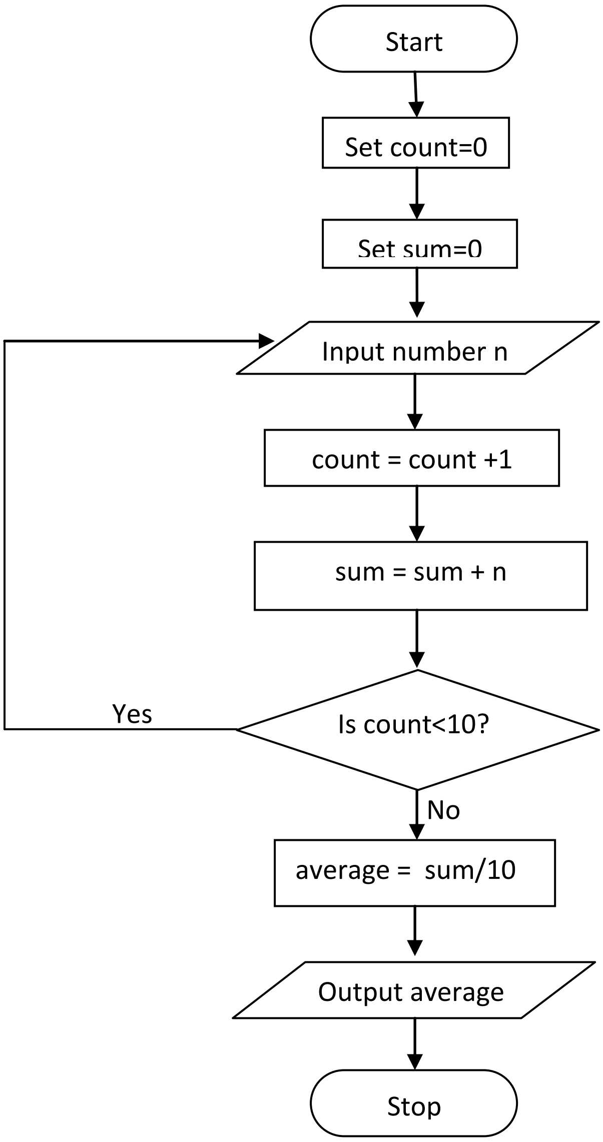
Example 3: Write an algorithm and draw a flowchart to input ten numbers and find their average.

Algo 3:

```
1.    Set count=0.
2.    Input a number
3.    Increment count by one.
4.    If count<10
        Goto  step 2
      Else
        Goto step 5
5.    Add the ten numbers and put the answer in sum.
6.    Divide sum by ten and put the result in average.
7.    Output Average.
```

In the above algorithm, we set value of count to zero. Every time you input a number, the value of count is incremented by one. Step 4 checks whether the value of count is less than 10 or not. If count<10, then the control moves back to step 2 and when the value of count becomes 10, the control moves to step 5. Therefore, the steps 2 through 4 are repeated until value of count becomes 10. In this way, we can implement a loop.

The flowchart of the above algorithm is as follows:



1.7 Summary:

A computer is a machine that receives instructions and produces a result after following those instructions. A computer can be used to solve a problem by following a set of stored instructions called the program. The problem solving process consists of defining the problem, developing an appropriate algorithm, writing a code based on the algorithm using a programming language and then testing the program. An algorithm is a detailed sequence of simple steps that are needed to solve a problem. There can be different ways of stating algorithms. One way of representing an algorithm is a flowchart. A flowchart is a diagrammatic representation of an algorithm that illustrates the sequence of operations to be performed to get the solution of a problem.

1.8 Keywords:

Computer: A computer is an electronic device that takes input from the user, stores, processes data and generates an output after processing the instructions given to it by the user.

Program: Set of instructions given to the computer to solve a particular problem in a language that is understood by the computer.

Algorithm: An algorithm is a detailed sequence of simple steps that are needed to solve a problem.

Flowchart: A flowchart is a diagrammatic representation of an algorithm that illustrates the sequence of operations to be performed to get the solution of a problem.

1.9 Short Answer Type Questions:

1. What is a computer?
2. Define an algorithm. What are the different ways of stating an algorithm?
3. What is a flowchart? Name some important symbols used in a flowchart.

1.10 Long Answer Type Questions:

1. Explain the problem solving process in detail.
2. Explain the guide lines for drawing a flowchart. Also explain how you can implement different programming strategies (simple sequence, decision making and looping) in a flowchart.
3. Draw a flow chart for stage 3 algo of problem 2 given in the lesson.
4. Write an algorithm and draw a flowchart to input numbers of a student in 6 subjects and to find the total marks obtained and his grade.

1.11 Suggested Readings:

1.	Computer Fundamentals	Pradeep K. Sinha, Preeti Sinha
2.	Windows Based Computer Courses	Gurvinder Singh, Rachhpal Singh

Lesson No. 2 **Author: Dr. DharamVeer Sharma**
Converted into SLM by: Dr. Vishal Singh

Introduction to C Language and Program Development

1. Objectives

In this lesson we shall learn about the origination and features of the C language. Let us begin with a quick introduction to C. Our aim is to show the essential elements of the language in real programs, but without getting bogged down in details, rules, and exceptions. At this point, we are not trying to be complete or even precise (save that the examples are meant to be correct). One needs to concentrate on the basics: variables and constants, arithmetic, control flow, functions, and the rudiments of input and output for starting to learn programming in any language. Some topics like pointers, structures, most of C's rich set of operators, several control-flow statements, and the standard library have not been touched upon in this lesson to keep the learning of the balanced in terms of complexity.

2. Introduction

Computer can understand language of 0s and 1s only, therefore, to interact with computer we should know the binary language, which is extremely difficult to learn and implement, because one wrong combination of 0s and 1s can mean entirely different thing. Human being, on the other hand can converse in their own language which is not directly understandable to computers. Therefore, some intermediate or translator is required to facilitate communication between humans and computers. These translators should be able to convert human language to computer language and vice-versa. Computer language and human language are two extremes in the hierarchy of languages. What we speak, at times, may mean differently for different persons. The same word or sentence may have different meaning. This is called ambiguity. Ambiguous languages constructs can not be correctly understood by computers. Therefore some language is required which is unambiguous, close to human language, whose words or sentences may be translated and represent precisely one meaning. Such languages are called programming languages. C is one such language, used extensively by programmers around the globe for writing computer programs, which are translated to binary language for computers' understanding and functioning.

3. Origin of C Language

The C language was developed in 1970s at Bell Laboratories by a system programmer named Dennis Ritchie. It derives its name from the fact that it is based on a language B, developed by Ken Thompson, another system programmer at Bell Laboratories.

C is a general-purpose programming language. It has been closely associated with the UNIX operating system where it was developed, since both the system and most of the programs that run on it are written in C. The language, however, is not tied to any one operating system or machine; and although it has been called a "system programming language" because it is useful for writing compilers and operating systems, it has been used equally well to write major programs in many different domains.

Many of the important ideas of C stem from the language BCPL (Basic Combined Programming Language), developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language B, which was written by Ken Thompson in 1970 for the first UNIX system on the DEC PDP-7.

BCPL and B are "typeless" languages, means data type of variable need not be declared in advance. By contrast, C provides a variety of data types. The fundamental types are characters, and integers and floating point numbers of several sizes. In addition, there is a hierarchy of derived data types created with pointers, arrays, structures and unions. Expressions are formed from operators and operands; any expression, including an assignment or a function call, can be a statement. Pointers provide for machine-independent address arithmetic.

4. Types of languages

In order to understand the features of C programming language we need to know the various types of programming languages and their features. The programming language can be divided into two categories:

i. Low level languages
ii. Middle Level Languages
iii. High level languages

i. Low level languages: These are the languages which are closer to the machine languages. These languages permit the efficient use of the machine. But these languages are hardware dependent, means programs written in one language may not run on other machines. Moreover, learning these languages is not an easy task. For learning the language programmers need to possess thorough knowledge of the hardware. These languages include machine languages and assembly languages, though assembly language is also referred to as middle level languages in some language literature.

ii. **Middle Level Languages:** These are the languages which are neither close to machine language nor near to human understandable languages. These are actually symbolic languages. Symbols representing, operations are the main building blocks. The symbols are mnemonics or acronyms of the operations to be performed. Programs written in middle level languages are very cryptic. Assembly language falls under this category. High level languages are sometimes converted to middle level languages before further translation to low level languages. In C language we can even write code in middle level language but in that case the middle level language code is translated by the respected language compiler, which needs to be installed in the machine and C environment must be configured to use that compiler.

iii. **High level languages:** These are the languages which are closer to the human understandable languages and include FORTRAN, BASIC, PASCAL, COBOL, PL/1 etc. These languages have been designed for better programming efficiency and have the following advantages:

- The syntax is like English language. This enables the programmer to easily learn the language. Additionally, the programs written in these languages are easily understandable.
- The programs written in these languages are portable, means the programs are not hardware dependent.

The C language stands between these two types of languages. It has some features of the low level languages along with the features of high level languages. The C language has capability of directly interacting with the hardware.

A program written in high level language needs to be compiled for checking syntax or grammatical errors. If the program is free of error then it is translated to low level language which can be directly executed on the computer.

5. Feature of C Language

C provides the fundamental control-flow constructions required for well-structured programs: statement grouping, decision making (if-else), selecting one of a set of possible values (switch), looping with the termination test at the top (while, for) or at the bottom (do), and early loop exit (break).

C language is case sensitive. 'A' and 'a' mean differently in the language.

Functions may return values of basic types, structures, unions, or pointers. Any function may be called recursively. Local variables are typically ``automatic'', or created anew with each invocation. Function definitions may not be nested but variables may be declared in a block-structured fashion. The functions of a C program may exist in separate source files that are compiled separately. Variables may be internal to a function, external but known only within a single source file, or visible to the entire program.

A preprocessing step performs macro substitution on program text, inclusion of other source files, and conditional compilation.

C is a relatively "low-level" language. This characterization is not pejorative; it simply means that C deals with the same sort of objects that most computers do, namely characters, numbers, and addresses. These may be combined and moved about with the arithmetic and logical operators implemented by real machines.

C provides no operations to deal directly with composite objects such as character strings, sets, lists or arrays. There are no operations that manipulate an entire array or string, although structures may be copied as a unit. The language does not define any storage allocation facility other than static definition and the stack discipline provided by the local variables of functions; there is no heap or garbage collection. Finally, C itself provides no input/output facilities; there are no READ or WRITE statements, and no built-in file access methods. All of these higher-level mechanisms must be provided by explicitly called functions. Most C implementations have included a reasonably standard collection of such functions.

Similarly, C offers only straightforward, single-thread control flow: tests, loops, grouping, and subprograms, but not multiprogramming, parallel operations, synchronization, or co-routines.

Although the absence of some of these features may seem like a grave deficiency, ("You mean I have to call a function to compare two character strings?"), keeping the language down to modest size has real benefits. Since C is relatively small, it can be described in small space, and learned quickly. A programmer can reasonably expect to know and understand and indeed regularly use the entire language.

For many years, the definition of C was the reference manual in the first edition of *The C Programming Language*. In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C", was completed in late 1988. Most of the features of the standard are already supported by modern compilers.

The standard is based on the original reference manual. The language is relatively little changed; one of the goals of the standard was to make sure that most existing programs would remain valid, or, failing that, that compilers could produce warnings of new behavior.

For most programmers, the most important change is the new syntax for declaring and defining functions. A function declaration can now include a description of the arguments of the function; the definition syntax changes to match. This extra information makes it much easier for compilers to detect errors caused by mismatched arguments; in our experience, it is a very useful addition to the language.

There are other small-scale language changes. Structure assignment and enumerations, which had been widely available, are now officially part of the language. Floating-point computations may now be done in single precision. The properties of arithmetic, especially for unsigned types, are clarified. The preprocessor is more elaborate. Most of these changes will have only minor effects on most programmers.

A second significant contribution of the standard is the definition of a library to accompany C. It specifies functions for accessing the operating system (for instance, to read and write files), formatted input and output, memory allocation, string manipulation, and the like. A collection of standard headers provides uniform access to declarations of functions in data types. Programs that use this library to interact with a host system are assured of compatible behavior. Most of the library is closely modeled on the ``standard I/O library'' of the UNIX system. This library was described in the first edition, and has been widely used on other systems as well. Again, most programmers will not see much change.

Because the data types and control structures provided by C are supported directly by most computers, the run-time library required to implement self-contained programs is tiny. The standard library functions are only called explicitly, so they can be avoided if they are not needed. Most can be written in C, and except for the operating system details they conceal, are themselves portable.

Although C matches the capabilities of many computers, it is independent of any particular machine architecture. With a little care it is easy to write portable programs, that is, programs that can be run without change on a variety of hardware. The standard makes portability issues explicit, and prescribes a set of constants that characterize the machine on which the program is run.

C is not a strongly-typed language, but as it has evolved, its type-checking has been strengthened. The original definition of C frowned on, but permitted, the interchange of pointers and integers; this has long since been eliminated, and the standard now requires the proper declarations and explicit conversions that had already been enforced by good compilers. The new function declarations are another step in this direction. Compilers will warn of most type errors, and there is no automatic conversion of incompatible data types. Nevertheless, C retains the basic philosophy that programmers know what they are doing; it only requires that they state their intentions explicitly.

C, like any other language, has its blemishes. Some of the operators have the wrong precedence; some parts of the syntax could be better. Nonetheless, C has proven to ben an extremely effective and expressive language for a wide variety of programming applications.

6. Structure of a C Program

The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:
Print the words

hello, world

This is a big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy.

In C, the program to print ``hello, world" is

```
#include <stdio.h>

main()

{

        printf("hello, world\n");

}
```

To run this program first it needs to be compiled and translated to machine level language, which we shall discuss in the next section.After executing the program it will print

```
hello, world
```

Now, for some explanations about the program itself. A C program, whatever its size, consists of *functions* and *variables*. A function contains *statements* that specify the computing operations to be done, and variables store values used during the computation. C functions are like the subroutines and functions in Fortran or the procedures and functions of Pascal. Our example is a function named main. Normally you are at liberty to give functions whatever names you like, but ``main" is special - your program begins executing at the beginning of main. This means that every program must have a main somewhere.

main will usually call other functions to help perform its job, some that you wrote, and others from libraries that are provided for you. The first line of the program,

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

tells the compiler to include information about the standard input/output library; the line appears at the beginning of many C source files. The standard library is described in following lessons.

One method of communicating data between functions is for the calling function to provide a list of values, called *arguments*, to the function it calls. The parentheses after the function name surround the argument list. In this example, main is defined to be a function that expects no arguments, which is indicated by the empty list ( ).

```
#include <stdio.h>  include information about standard library

main()              define a function called main that received
                    no argument values statements of main are
```

```
    {                               enclosed in braces main calls library

        printf("hello, world\n");       function printf to print this sequence
                                        of characters \n represents the
                                    newline character

    }
```

The first C program

The statements of a function are enclosed in braces { }. The function main contains only one statement,

```
    printf("hello, world\n");
```

A function is called by naming it, followed by a parenthesized list of arguments, so this calls the function printf with the argument "hello, world\n". printf is a library function that prints output, in this case the string of characters between the quotes.

A sequence of characters in double quotes, like "hello, world\n", is called a *character string* or *string constant*. For the moment our only use of character strings will be as arguments for printf and other functions.

The sequence \n in the string is C notation for the *newline character*, which when printed advances the output to the left margin on the next line. If you leave out the \n (a worthwhile experiment), you will find that there is no line advance after the output is printed. You must use \n to include a newline character in the printf argument; if you try something like

```
    printf("hello, world

    ");
```

the C compiler will produce an error message.

printf never supplies a newline character automatically, so several calls may be used to build up an output line in stages. Our first program could just as well have been written

```
    #include <stdio.h>

    main()

    {

        printf("hello, ");

        printf("world");

        printf("\n");
    }
```

to produce identical output.

Notice that \n represents only a single character. An *escape sequence* like \n provides a general and extensible mechanism for representing hard-to-type or invisible characters. Among the others that C provides are \t for tab, \b for backspace, \" for the double quote, \a for producing a bell sound and \\ for the backslash itself.

7. Stages in Program Development

The following are the various stages involved in development of computer program ready to be executed on the computer:

i. **Developing the program:** The first and fore most task for develop the program for a particular problem is to understand the problem in hand to be solved. Analysis of the problem will reveal the input required and output produced by the problem solution. Some input is clearly visible from the problem statement itself and some other input may be hidden which is revealed while developing the solution. Output to be produced by the program is clearly stated by the problem it self. Some auxiliary output may also be produced, however, which may or may not be of some use. The next step is to prepare a detailed list of steps required to be carried out for solving the problem, which is called the algorithm. Once input, output and algorithm have been clearly defined, the next step is to translate these steps in a computer program using any high level language. The program in high level language is called the source code and is stored in a disk file. This program contains the logic or steps for solving the problem.

ii. **Compile the program:** The next step is to compile the program for checking syntax errors and translation. This is done by the C compiler. The output of compilation is the object code, if no syntax errors are reported by the compiler. The following figure shows the compilation process.

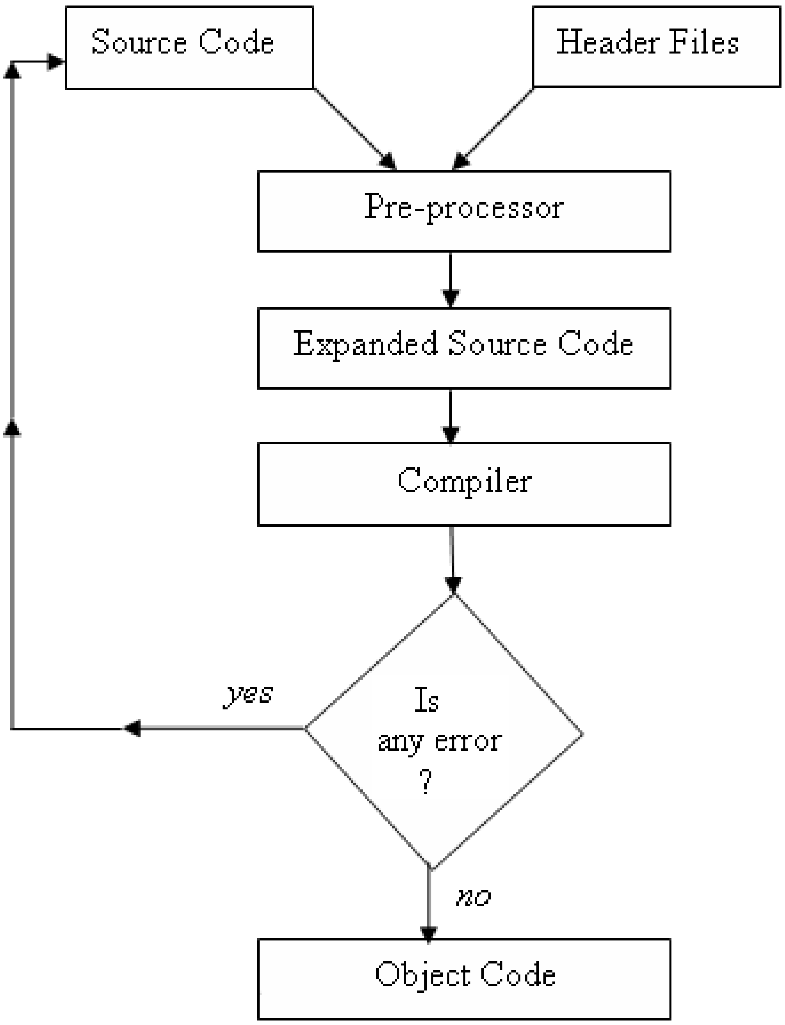


Figure 1: Stages in Compilation of a C program

C compiler has in-built pre-processor. The pre-processor processes the source code before it is passed to the compiler for compilation. Pre-processor commands, also known as directives, tell the pre-processor how to process the source code. Depending on the pre-processor directives, the pre-processor processes the source code and produces the expanded version of source code. The C complier takes expanded version of the source code as its input and if there are no errors in the source code, it produces a machine code (object code) version of the program which is saved on the disk file.

If there are some errors during compilation phase, known as syntax errors, then the compiler reports these errors in the form of diagnostic messages, which tell the cause and origin of errors and compilation process terminates. Having corrected the reported errors the source code is compiled again as shown in figure 1.

iii. **Linking the program:** After the compilation stage, the machine code version of the program but it can not be directly executed, as it may contain references to the library functions or user defined functions in other object modules which are compiled separately. In order to produce an executable code, these object codes are to be linked together and also with the system library. The process of linking is shown in figure 2 below:

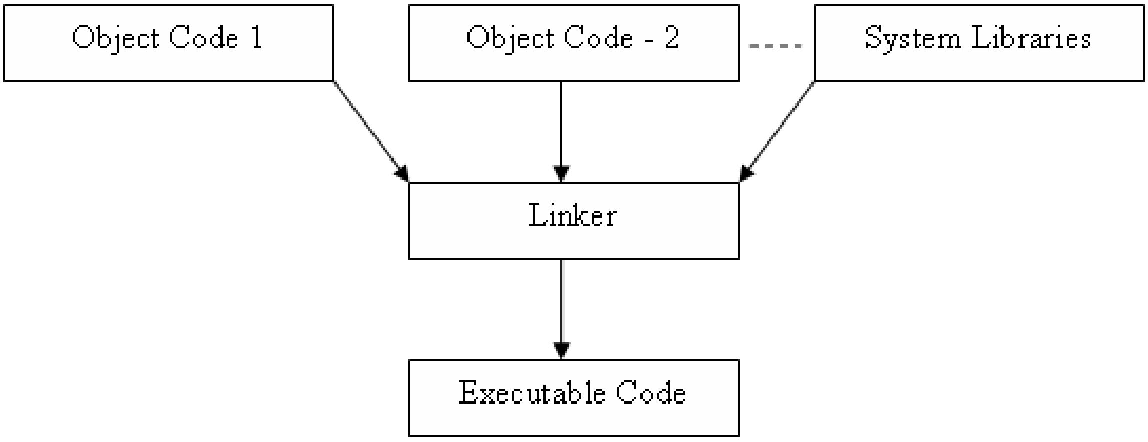


Figure 2: Linking of object code(s) with system libraries

Once the linking process is over, another disk, with the same name as of the program file, with extension exe is produced. This is the file that contains the code which can be directly executed on the computer.

iv. **Debugging the program:** The next stage in the program development is debugging of the program to make sure that the program is free of bugs and produces the desired outputs for all possible inputs. In this stage the program is executed with all possible values of input data for which result is known or can be computed manually. The program is declared correct if the output of the program matches the expected results. If the output does not match the expected results then the program is said to be semantically or logically incorrect and needs to be corrected by scanning the logic used in the source code. The logical or semantic errors are removed and the program is compiled and linked again.

v. **Documenting the program:** The final stage in the development of a program is documentation. The term documentation means recording the important information regarding the program. The documentation enables other users or programmers to understand the logic and purpose of the program. This facilitates maintenance and upgradation of the program.

Some compilers, like Turbo C, have integrated development environment (IDE), which facilitates program writing (editing), compilation, linking and execution of the program from one place. But in some other systems like in UNIX we need separate tools for editing, compilation, linking and execution of the program.

8. Summary

The C language was developed in Bell Laboratories by Dennis Ritchie and his associates. It is a refined version of BCPL with enhanced features. The C language is a high level language with capabilities of low level language as well. It has defined data types and program constructs like sequential, conditional and iterative flows. A program written in C language is first check for syntax correctness and then converted to object code, thereafter it is linked with other libraries and finally an executable file is produced.

9. Short Answer Type Questions

1. What are the various types of programming languages?

2. What are the various types of errors?

3. Name any four high level programming languages?

10. Long Answer Type Questions

1. Discuss in detail the origin of C language.

2. What are the various features of C programming language?

3. What are various parts of a C program?

4. What are various stages of program development using C language?

11. Suggested Books

1.	Application Programming in C	R. S. Salaria
2.	C Programming using Turbo C	Robert Lafore

Lesson No. 3 **Author: Dr. DharamVeer Sharma**
Converted into SLM by: Dr. Vishal Singh

Identifiers, Keywords, Data Types and Type Conversion

1. Objectives

In this lesson we will learn about the basic concepts used in the C programming language, which include, identifiers, variables and constants, keywords, data types and type conversion.

2. Introduction

Identifiers and data types are the basic building blocks of a programming language. A C program starts with the declaration of data types of the various identifiers to be used in the program. Then the behaviour of the identifiers also needs to be defined that whether these are variables or constants. Study of type conversion methodology is also important as it facilitates safe programming.

3. Character Set

The character set used to form words, numbers and expressions depend upon the computer on which the program runs. The characters in C are classified in the following categories:

i. Letters

ii. Digits

iii. White spaces

iv. Special characters

The C language character set is listed in the following table:

Letters	A to Z, a to Z
Digits	0 to 9
White Spaces	Blank, Horizontal tab, Vertical tab, new line, form feed
Special characters	All other characters available on standard keyboard.

4. Identifiers

Every program element must be named to distinguish it from other elements. The name assigned to the element should be meaningful, though it is not necessary, but it facilitates easy understanding of the program. Identifier is the name given to some program element. The program element is then identified by that name. The element may be some variable, constant, data structure, program block, function, pointer, file etc. The identifier naming rules are as follows:

a. Identifier name should begin with an alphabet or underscore (_) but never with a digit.
b. The following characters may be any combination of alphabets, digits and special symbol (underscore) but two consecutive underscores are not permitted.
c. In some C language compilers identifier length is restricted to some limit which varies from 32 to 48.
d. No other symbol or special character is permitted.

The following are valid C identifiers:

sum, factorial, number, first_name, permanent_address.

However, sum, SUM and Sum are different identifier because C language is case sensitive.

The following are invalid C identifiers:

5thdigit	(first letter should be alphabet or underscore)
first name	(Identifier can't contain spaces)
char	(It is a reserve word, discussed in next section)

It's wise to choose identifier names that are related to the purpose of the identifier, and that are unlikely to get mixed up typographically. We tend to use short names for local variables, especially loop indices, and longer names for external variables.

5. Keywords

There is a set of words whose meaning is predefined in the C language and these words can not be used as identifier. These words are also called reserve words. The following is the set of words used as reserve words in the C language.

void	int	char	float	double	long	short	signed
for	do	while	If	else	break	continue	goto
static	auto	extern	case	switch	default	return	struct
sizeof	enum	typedef	const	volatile	register	unsigned	union

6. Data Types

Since, the C language is a strongly typed language therefore data type of all the variables need to be declared in advance. There are only a few basic data types in C:

Char	a single byte, capable of holding one character in the local character set
Int	an integer, typically reflecting the natural size of integers on the host machine
Float	single-precision floating point
double	double-precision floating point

In addition, there are a number of qualifiers that can be applied to these basic types. short and long apply to integers:

```
short int sh;

long int counter;
```

The word int can be omitted in such declarations, and typically it is. The intent is that short and long should provide different lengths of integers where practical; int will normally be the natural size for a particular machine, short is often 16 bits long, and int either 16 or 32 bits. Each compiler is free to choose appropriate sizes for its own hardware, subject only to the the restriction that shorts and ints are at least 16 bits, longs are at least 32 bits, and short is no longer than int, which is no longer than long.

The qualifier signed or unsigned may be applied to char or any integer, unsigned numbers are always positive or zero, and obey the laws of arithmetic modulo 2^n, where n is the number of bits in the type. So, for instance, if chars are 8 bits, unsigned char variables have values between 0 and 255, while signed chars have values between -128 and 127 (in a two's complement machine.) Whether plain chars are signed or unsigned is machine-dependent, but printable characters are always positive.

The type long double specifies extended-precision floating point. As with integers, the sizes of floating-point objects are implementation-defined; float, double and long double could represent one, two or three distinct sizes.

The standard headers <limits.h> and <float.h> contain symbolic constants for all of these sizes, along with other properties of the machine and compiler.

Data type	**Size (in bytes)**	**Range**	**Format String**
char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
short or int	2	-32768 to 32767	%i or %d
unsigned int	2	0 to 65535	%u
long	4	-2147483648 to 2147483647	%ld
unsigned long	4	0 to 4294967295	%lu
float	4	3.4 e-38 to 3.4 e+38	%f or %g
double	8	1.7 e-308 to 1.7 e+308	%lf
long double	10	3.4 e-4932 to 1.1 e+4932	%lf

7. Type Conversion

Some times data type of values needs to be modified. For example, if two integer values are divided then the result may be required in float but since the variable are of type integer, the result produced will also be of type integer.

If a = 5 and b = 2 and both a and b are integer then

result = a/b;

will store 2 in result irrespective of the data type of variable result. However to obtain float value we need to modify the data type of the argument variables of the expression, which can be done as follows:

result = (float)a/b; and value of result will be 2.5

this is called type casting and is done explicitly. Implicit data type conversion is also done while evaluating expression containing mixed types of variable. This is called coercion. In this case the data type of the lower sized or ranged variable is converted to the upper sized or ranged variable, for example, if b is float in the above example then the value of result will be 2.5.

When an operator has operands of different types, they are converted to a common type according to a small number of rules. In general, the only automatic conversions are those that convert a ``narrower'' operand into a ``wider'' one without losing information, such as converting an integer into floating point in an expression like f + i. Expressions that don't make sense, like using a float as a subscript, are disallowed. Expressions that might lose information, like assigning a longer integer type to a shorter, or a floating-point type to an integer, may draw a warning, but they are not illegal.

A char is just a small integer, so chars may be freely used in arithmetic expressions. This permits considerable flexibility in certain kinds of character transformations. One is exemplified by this naive implementation of the function atoi, which converts a string of digits into its numeric equivalent.

```
/* atoi:  convert s to integer */

int atoi(char s[])

{

    int i, n;

    n = 0;

    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)

        n = 10 * n + (s[i] - '0');

    return n;

}
```

The expression

```
 s[i] - '0'
```

gives the numeric value of the character stored in s[i], because the values of '0', '1', etc., form a contiguous increasing sequence.

Another example of char to int conversion is the function lower, which maps a single character to lower case *for the ASCII character set*. If the character is not an upper case letter, lower returns it unchanged.

```
/* lower:  convert c to lower case; ASCII only */

int lower(int c)

{

    if (c >= 'A' && c <= 'Z')

        return c + 'a' - 'A';

    else
```

```
        return c;
  }
```

This works for ASCII because corresponding upper case and lower case letters are a fixed distance apart as numeric values and each alphabet is contiguous -- there is nothing but letters between A and Z. This latter observation is not true of the EBCDIC character set, however, so this code would convert more than just letters in EBCDIC.

The standard header <ctype.h>, defines a family of functions that provide tests and conversions that are independent of character set. For example, the function tolower is a portable replacement for the function lower shown above. Similarly, the test

```
  c >= '0' && c <= '9'
```

can be replaced by

```
  isdigit(c)
```

We will use the <ctype.h> functions from now on.

There is one subtle point about the conversion of characters to integers. The language does not specify whether variables of type char are signed or unsigned quantities. When a char is converted to an int, can it ever produce a negative integer? The answer varies from machine to machine, reflecting differences in architecture. On some machines a char whose leftmost bit is 1 will be converted to a negative integer (``sign extension''). On others, a char is promoted to an int by adding zeros at the left end, and thus is always positive.

The definition of C guarantees that any character in the machine's standard printing character set will never be negative, so these characters will always be positive quantities in expressions. But arbitrary bit patterns stored in character variables may appear to be negative on some machines, yet positive on others. For portability, specify signed or unsigned if non-character data is to be stored in char variables.

Relational expressions like i > j and logical expressions connected by && and || are defined to have value 1 if true, and 0 if false. Thus the assignment

```
  d = c >= '0' && c <= '9'
```

sets d to 1 if c is a digit, and 0 if not. However, functions like isdigit may return any non-zero value for true. In the test part of if, while, for, etc., ``true'' just means ``non-zero'', so this makes no difference.

Implicit arithmetic conversions work much as expected. In general, if an operator like + or * that takes two operands (a binary operator) has operands of different types, the ``lower'' type is *promoted* to the ``higher'' type before the operation proceeds. The result is of the integer type.

If there are no unsigned operands, however, the following informal set of rules will suffice:

- If either operand is long double, convert the other to long double.
- Otherwise, if either operand is double, convert the other to double.
- Otherwise, if either operand is float, convert the other to float.
- Otherwise, convert char and short to int.
- Then, if either operand is long, convert the other to long.

Notice that floats in an expression are not automatically converted to double; this is a change from the original definition. In general, mathematical functions like those in <math.h> will use double precision. The main reason for using float is to save storage in large arrays, or, less often, to save time on machines where double-precision arithmetic is particularly expensive.

Conversion rules are more complicated when unsigned operands are involved. The problem is that comparisons between signed and unsigned values are machine-dependent, because they depend on the sizes of the various integer types. For example, suppose that int is 16 bits and long is 32 bits. Then -1L < 1U, because 1U, which is an unsigned int, is promoted to a signed long. But -1L > 1UL because -1L is promoted to unsigned long and thus appears to be a large positive number.

Conversions take place across assignments; the value of the right side is converted to the type of the left, which is the type of the result.

A character is converted to an integer, either by sign extension or not, as described above.

Longer integers are converted to shorter ones or to chars by dropping the excess high-order bits. Thus in

```
int  i;

char c;

i = c;

c = i;
```

the value of c is unchanged. This is true whether or not sign extension is involved. Reversing the order of assignments might lose information, however.

If x is float and i is int, then x = i and i = x both cause conversions; float to int causes truncation of any fractional part. When a double is converted to float, whether the value is rounded or truncated is implementation dependent.

Since an argument of a function call is an expression, type conversion also takes place when arguments are passed to functions. In the absence of a function prototype, char and short become int, and float becomes double. This is why we have declared function arguments to be int and double even when the function is called with char and float.

Finally, explicit type conversions can be forced (``coerced'') in any expression, with a unary operator called a cast. In the construction

(*type name*) *expression*

the *expression* is converted to the named type by the conversion rules above. The precise meaning of a cast is as if the *expression* were assigned to a variable of the specified type, which is then used in place of the whole construction. For example, the library routine sqrt expects a double argument, and will produce nonsense if inadvertently handled something else. (sqrt is declared in <math.h>.) So if n is an integer, we can use

```
sqrt((double) n)
```

to convert the value of n to double before passing it to sqrt. Note that the cast produces the *value* of n in the proper type; n itself is not altered. The cast operator has the same high precedence as other unary operators, as summarized in the table at the end of this chapter.

If arguments are declared by a function prototype, as the normally should be, the declaration causes automatic coercion of any arguments when the function is called. Thus, given a function prototype for sqrt:

```
double sqrt(double)
```

the call

```
root2 = sqrt(2)
```

coerces the integer 2 into the double value 2.0 without any need for a cast.

8. Variables and constants

Variables and constants are the basic data objects manipulated in a program. Declarations list the variables to be used, and state what type they have and perhaps what their initial values are. Operators specify what is to be done to them. Expressions combine variables and constants to produce new values. The type of an object determines the set of values it can have and what operations can be performed on it.

Declaring variables:

The declaration of all variables to be used in a function should be declared in the variable declaration part of the function. All the variables must be declared before they can be used. A declaration specifies a type, and contains a list of one or more variables of that type. It also provides a variable name to the compiler and tells the data type of the variable which helps in determining the memory requirements for the variable. The syntax for variable declaration is as follows:

```
data_type variable_name
```

Example

```
int rollno;
char c;
float amount;
double d;
```

Commas in the variable declaration separate the variables of the same type, as in

```
  int  lower, upper, step;
  char c, line[1000];
```

Variables can be distributed among declarations in any fashion; the lists above could well be written as

```
  int  lower;
  int  upper;
  int  step;
  char c;
  char line[1000];
```

The latter form takes more space, but is convenient for adding a comment to each declaration for subsequent modifications.

A variable may also be initialized in its declaration. If the name is followed by an equals sign and an expression, the expression serves as an initializer, as in

```
  char  esc = '\\';
  int   i = 0;
  int   limit = MAXLINE+1;
  float eps = 1.0e-5;
```

If the variable in question is not automatic, the initialization is done once only, conceptionally before the program starts executing, and the initializer must be a constant expression. An explicitly initialized automatic variable is initialized each time the function or block it is in is entered; the initializer may be any expression.

External and static variables are initialized to zero by default. Automatic variables for which is no explicit initializer have undefined (i.e., garbage) values.

Declaring constants:

The qualifier const can be applied to the declaration of any variable to specify that its value will not be changed. For an array, the const qualifier says that the elements will not be altered.

```
const double e = 2.71828182845905;

const char msg[] = "warning: ";
```

The const declaration can also be used with array arguments, to indicate that the function does not change that array:

```
int strlen(const char[]);
```

The result is implementation-defined if an attempt is made to change a const.

An integer constant like 1234 is an int. A long constant is written with a terminal l (ell) or L, as in 123456789L; an integer constant too big to fit into an int will also be taken as a long. Unsigned constants are written with a terminal u or U, and the suffix ul or UL indicates unsigned long.

Floating-point constants contain a decimal point (123.4) or an exponent (1e-2) or both; their type is double, unless suffixed. The suffixes f or F indicate a float constant; l or L indicate a long double.

The value of an integer can be specified in octal or hexadecimal instead of decimal. A leading 0 (zero) on an integer constant means octal; a leading 0x or 0X means hexadecimal. For example, decimal 31 can be written as 037 in octal and 0x1f or 0x1F in hex. Octal and hexadecimal constants may also be followed by L to make them long and U to make them unsigned: 0XFUL is an *unsigned long* constant with value 15 decimal.

A character constant is an integer, written as one character within single quotes, such as 'x'. The value of a character constant is the numeric value of the character in the machine's character set. For example, in the ASCII character set the character constant '0' has the value 48, which is unrelated to the numeric value 0. If we write '0' instead of a numeric value like 48 that depends on the character set, the program is independent of the particular value and easier to read. Character constants participate in numeric operations just as any other integers, although they are most often used in comparisons with other characters.

Certain characters can be represented in character and string constants by escape sequences like \n (newline); these sequences look like two characters, but represent only one. In addition, an arbitrary byte-sized bit pattern can be specified by

```
'\ooo'
```

where *ooo* is one to three octal digits (0...7) or by

```
'\xhh'
```

where *hh* is one or more hexadecimal digits (0...9, a...f, A...F). So we might write

```
#define VTAB '\013'   /* ASCII vertical tab */
#define BELL '\007'   /* ASCII bell character */
```

or, in hexadecimal,

```
#define VTAB '\xb'   /* ASCII vertical tab */
#define BELL '\x7'   /* ASCII bell character */
```

The complete set of escape sequences is

\a	alert (bell) character	\\	backslash
\b	backspace	\?	question mark
\f	formfeed	\'	single quote
\n	newline	\"	double quote
\r	carriage return	*ooo*	octal number
\t	horizontal tab	\x*hh*	hexadecimal number
\v	vertical tab		

The character constant '\0' represents the character with value zero, the null character. '\0' is often written instead of 0 to emphasize the character nature of some expression, but the numeric value is just 0.

A *constant expression* is an expression that involves only constants. Such expressions may be evaluated at during compilation rather than run-time, and accordingly may be used in any place that a constant can occur, as in

```
#define MAXLINE 1000
char line[MAXLINE+1];
```

or

```
#define LEAP 1 /* in leap years */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

A *string constant*, or *string literal*, is a sequence of zero or more characters surrounded by double quotes, as in

```
"I am a string"
```

or

```
"" /* the empty string */
```

The quotes are not part of the string, but serve only to delimit it. The same escape sequences used in character constants apply in strings; \" represents the double-quote character. String constants can be concatenated at compile time:

```
"hello, " "world"
```

is equivalent to

```
"hello, world"
```

This is useful for splitting up long strings across several source lines.

Technically, a string constant is an array of characters. The internal representation of a string has a null character '\0' at the end, so the physical storage required is one more than the number of characters written between the quotes. This representation means that there is no limit to how long a string can be, but programs must scan a string completely to determine its length. The standard library function strlen(s) returns the length of its character string argument s, excluding the terminal '\0'. Here is our version:

```
/* strlen:  return length of s */
int strlen(char s[])
{
    int i;


    while (s[i] != '\0')
        ++i;
    return i;
}
```

strlen and other string functions are declared in the standard header <string.h>.

Be careful to distinguish between a character constant and a string that contains a single character: 'x' is not the same as "x". The former is an integer, used to produce the numeric value of the letter *x* in the machine's character set. The latter is an array of characters that contains one character (the letter *x*) and a '\0'.

There is one other kind of constant, the *enumeration constant*. An enumeration is a list of constant integer values, as in

```
enum boolean { NO, YES };
```

The first name in an enum has value 0, the next 1, and so on, unless explicit values are specified. If not all values are specified, unspecified values continue the progression from the last specified value, as the second of these examples:

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',

               NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };



enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,

              JUL, AUG, SEP, OCT, NOV, DEC };

                  /* FEB = 2, MAR = 3, etc. */
```

Names in different enumerations must be distinct. Values need not be distinct in the same enumeration.

Enumerations provide a convenient way to associate constant values with names, an alternative to #define with the advantage that the values can be generated for you. Although variables of enum types may be declared, compilers need not check that what you store in such a variable is a valid value for the enumeration. Nevertheless, enumeration variables offer the chance of checking and so are often better than #defines. In addition, a debugger may be able to print values of enumeration variables in their symbolic form.

9. Summary

Character set of a language specifies the valid set of characters using which words of the language are formed for identifier declaration.

Identifier is the name given to some program element. The element may be some variable, constant, data structure, program block, function, pointer, file etc. Identifier is the name given to some program element.

There is a set of words whose meaning is predefined in the C language and these words can not be used as identifier. These words are also called reserve words.

Since, the C language is a strongly typed language therefore data type of all the variables need to be declared in advance. The qualifier signed or unsigned may be applied to char or any integer, unsigned numbers are always positive or zero, and obey the laws of arithmetic modulo 2^n, where n is the number of bits in the type.

Type conversion facilitates the conversion of data type of the result of expression. It may be explicit, called type casting or implicit, called conversion.

Variables and constants are the basic data objects manipulated in a program. Declarations list the variables to be used, and state what type they have and perhaps what their initial values are. Operators specify what is to be done to them. Expressions combine variables and constants to produce new values. The type of an object determines the set of values it can have and what operations can be performed on it.

10. Short Answer Type Questions

1. What are valid characters in the C character set?
2. What do you mean by an identifier?
3. What do you mean by a reserve word?
4. What is the need of declaring the type of a variable?
5. What do you mean by type conversion?
6. What is the difference between variable and constant?

11. Long Answer Type Questions

1. Discuss the various identifier naming rules.
2. Write any 24 reserve words of C language.
3. What are the basic data types available in C language? What is the size of each data type?
4. What are the various types of constant declarations? Explain giving examples.

12. Suggested Books

Programming with ANSI and Turbo C	Ashok N. Kamthane
C Programming	E. Balagurusamy
Application Programming in C	R. S. Salaria

BA PART-2 (SEM-3) **PAPER : BAP-201**
C Programming and Data Structures

Lesson No. 4 **Author: Dr. DharamVeer Sharma**
Converted into SLM by: Dr. Vishal Singh

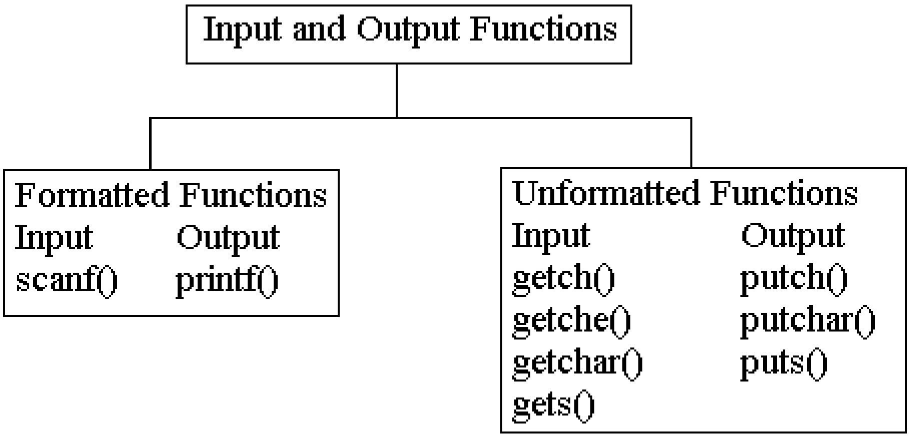
Performing Input Output Operations

1. Objectives

In this lesson we will discuss the role, types and usage of input and output statements.

2. Introduction

Input output statements facilitate interaction between program and the users. Through input statements user provide input to the program and through the output statements prompts and results are displayed. The following are the input output functions which we shall discuss in this lesson.



3. Unformatted Input Statements

Input and output are not part of the C language itself, so we have not emphasized them in our presentation thus far. Nonetheless, programs interact with their environment in much more complicated ways than those we have shown before. In this lesson we will describe the standard library, a set of functions that provide input and output, string handling, storage management, mathematical routines, and a variety of other services for C programs. We will concentrate on input and output

The ANSI standard defines these library functions precisely, so that they can exist in compatible form on any system where C exists. Programs that confine their system interactions to facilities provided by the standard library can be moved from one system to another without change.

The properties of library functions are specified in more than a dozen headers; we have already seen several of these, including <stdio.h>, <string.h>, and <ctype.h>. We will not present the entire library here, since we are more interested in writing C programs that use it.

Standard Input

As we said in, the library implements a simple model of text input and output. A text stream consists of a sequence of lines; each line ends with a newline character. If the system doesn't operate that way, the library does whatever necessary to make it appear as if it does. For instance, the library might convert carriage return and linefeed to newline on input and back again on output.

The simplest input mechanism is to read one character at a time from the *standard input*, normally the keyboard, with getchar:

```
int getchar(void)
```

getchar returns the next input character each time it is called, or EOF when it encounters end of file. The symbolic constant EOF is defined in <stdio.h>. The value is typically -1, bus tests should be written in terms of EOF so as to be independent of the specific value.

In many environments, a file may be substituted for the keyboard by using the < convention for input redirection: if a program prog uses getchar, then the command line

```
prog <infile
```

causes prog to read characters from infile instead. The switching of the input is done in such a way that prog itself is oblivious to the change; in particular, the string ``<infile'' is not included in the command-line arguments in argv. Input switching is also invisible if the input comes from another program via a pipe mechanism: on some systems, the command line

```
otherprog | prog
```

runs the two programs otherprog and prog, and pipes the standard output of otherprog into the standard input for prog.

Following are the unformatted input functions

a. **getchar():** This function reads character type data from the standard input. It reads one character at a time till the user presses the enter key.

b. **getch() and getche():** These functions read any character from the standard input device. The character entered is not displayed or echoed by getch() function. These functions are included in conio.h header file.

c. **gets():** This function is used for accepting any string through stdin (keyboard) until enter key is pressed. The header file stdio.h is needed for implementing this function.

4. Formatted Input – scanf

The function scanf is the input analog of printf, providing many of the same conversion facilities in the opposite direction.

```
int scanf(char *format, ...)
```

scanf reads characters from the standard input, interprets them according to the specification in format, and stores the results through the remaining arguments. The format argument is described below; the other arguments, *each of which must be a pointer*, indicate where the corresponding converted input should be stored. As with printf, this section is a summary of the most useful features, not an exhaustive list.

scanf stops when it exhausts its format string, or when some input fails to match the control specification. It returns as its value the number of successfully matched and assigned input items. This can be used to decide how many items were found. On the end of file, EOF is returned; note that this is different from 0, which means that the next input character does not match the first specification in the format string. The next call to scanf resumes searching immediately after the last character already converted.

There is also a function sscanf that reads from a string instead of the standard input:

```
int sscanf(char *string, char *format, arg1, arg2, ...)
```

It scans the string according to the format in format and stores the resulting values through arg1, arg2, etc. These arguments must be pointers.

The format string usually contains conversion specifications, which are used to control conversion of input. The format string may contain:

- Blanks or tabs, which are not ignored.
- Ordinary characters (not %), which are expected to match the next non-white space character of the input stream.

- Conversion specifications, consisting of the character %, an optional assignment suppression character *, an optional number specifying a maximum field width, an optional h, l or L indicating the width of the target, and a conversion character.

A conversion specification directs the conversion of the next input field. Normally the result is places in the variable pointed to by the corresponding argument. If assignment suppression is indicated by the * character, however, the input field is skipped; no assignment is made. An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, is specified, is exhausted. This implies that scanf will read across boundaries to find its input, since newlines are white space. (White space characters are blank, tab, newline, carriage return, vertical tab, and formfeed.)

The conversion character indicates the interpretation of the input field. The corresponding argument must be a pointer, as required by the call-by-value semantics of C. Conversion characters are shown in following table.

Basic scanf Conversions

Character	**Input Data; Argument type**
D	decimal integer; int *
i	integer; int *. The integer may be in octal (leading 0) or hexadecimal (leading 0x or 0X).
o	octal integer (with or without leading zero); int *
u	unsigned decimal integer; unsigned int *
x	hexadecimal integer (with or without leading 0x or 0X); int *
c	characters; char *. The next input characters (default 1) are placed at the indicated spot. The normal skip-over white space is suppressed; to read the next non-white space character, use %1s
s	character string (not quoted); char *, pointing to an array of characters long enough for the string and a terminating '\0' that will be added.
e,f,g	floating-point number with optional sign, optional decimal point and optional exponent; float *

%	literal %; no assignment is made.

The conversion characters d, i, o, u, and x may be preceded by h to indicate that a pointer to short rather than int appears in the argument list, or by l (letter ell) to indicate that a pointer to long appears in the argument list.

As a first example, the rudimentary calculator can be written with scanf to do the input conversion:

```
#include <stdio.h>

main()  /* rudimentary calculator */

{

        double sum, v;

        sum = 0;

        while (scanf("%lf", &v) == 1)

                printf("\t%.2f\n", sum += v);

        return 0;

}
```

Suppose we want to read input lines that contain dates of the form

```
25 Dec 1988
```

The scanf statement is

```
int day, year;

char monthname[20];

scanf("%d %s %d", &day, monthname, &year);
```

No & is used with monthname, since an array name is a pointer.

Literal characters can appear in the scanf format string; they must match the same characters in the input. So we could read dates of the form mm/dd/yy with the scanf statement:

```
int day, month, year;

scanf("%d/%d/%d", &month, &day, &year);
```

scanf ignores blanks and tabs in its format string. Furthermore, it skips over white space (blanks, tabs, newlines, etc.) as it looks for input values. To read input whose format is not fixed, it is often best to read a line at a time, then pick it apart with scanf. For example, suppose we want to read lines that might contain a date in either of the forms above. Then we could write

```
while (getline(line, sizeof(line)) > 0) {
        if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
                printf("valid: %s\n", line); /* 25 Dec 1988 form */
        else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
                printf("valid: %s\n", line); /* mm/dd/yy form */
        else
                printf("invalid: %s\n", line); /* invalid form */
}
```

Calls to scanf can be mixed with calls to other input functions. The next call to any input function will begin by reading the first character not read by scanf.

A final warning: the arguments to scanf and sscanf *must* be pointers. By far the most common error is writing

```
scanf("%d", n);
```

instead of

```
scanf("%d", &n);
```

This error is not generally detected at compile time.

5. Unformatted Output Statements

The function

```
int putchar(int)
```

is used for output: putchar(c) puts the character c on the standard output, which is by default the screen. putchar returns the character written, or EOF is an error occurs. Again, output can usually be directed to a file with >*filename*: if prog uses putchar,

```
prog >outfile
```

will write the standard output to outfile instead. If pipes are supported,

```
prog | anotherprog
```

puts the standard output of prog into the standard input of anotherprog.

Output produced by printf also finds its way to the standard output. Calls to putchar and printf may be interleaved - output happens in the order in which the calls are made.

Each source file that refers to an input/output library function must contain the line

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

before the first reference. When the name is bracketed by < and > a search is made for the header in a standard set of places (for example, on UNIX systems, typically in the directory /usr/include).

Following are the unformatted output functions:

a. **putchar():** This functions prints one character on the screen at a time which is read by the standard input.

b. **putch():** This function prints any character taken by the standard input devices.

c. **puts():** This function prints the string or character array.

Many programs read only one input stream and write only one output stream; for such programs, input and output with getchar, putchar, and printf may be entirely adequate, and is certainly enough to get started. This is particularly true if redirection is used to connect the output of one program to the input of the next. For example, consider the program lower, which converts its input to lower case:

```
#include <stdio.h>

#include <ctype.h>

main() /* lower: convert input to lower case*/

{

        int c

        while ((c = getchar()) != EOF)

                putchar(tolower(c));

        return 0;

}
```

The function tolower is defined in <ctype.h>; it converts an upper case letter to lower case, and returns other characters untouched. As we mentioned earlier, ``functions'' like getchar and putchar in <stdio.h> and tolower in <ctype.h> are often macros, thus avoiding the overhead of a function call per character. Regardless of how the <ctype.h> functions are implemented on a given machine, programs that use them are shielded from knowledge of the character set.

6. Formatted Output - printf

The output function printf translates internal values to characters. We have used printf informally in previous chapters. The description here covers most typical uses but is not complete; for the full story, refer the books given at the end of this lesson.

```
int printf(char *format, arg1, arg2, ...);
```

printf converts, formats, and prints its arguments on the standard output under control of the format. It returns the number of characters printed.

The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to printf. Each conversion specification begins with a % and ends with a conversion character. Between the % and the conversion character there may be, in order:

- A minus sign, which specifies left adjustment of the converted argument.
- A number that specifies the minimum field width. The converted argument will be printed in a field at least this wide. If necessary it will be padded on the left (or right, if left adjustment is called for) to make up the field width.
- A period, which separates the field width from the precision.
- A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits after the decimal point of a floating-point value, or the minimum number of digits for an integer.
- An h if the integer is to be printed as a short, or l (letter ell) if as a long.

Conversion characters are shown in the following table. If the character after the % is not a conversion specification, the behavior is undefined.

A width or precision may be specified as *, in which case the value is computed by converting the next argument (which must be an int). For example, to print at most max characters from a string s,

```
printf("%.*s", max, s);
```

Basic printf Conversions

Character	**Argument type; Printed As**
d,i	int; decimal number

o	int; unsigned octal number (without a leading zero)
x,X	int; unsigned hexadecimal number (without a leading 0x or 0X), using abcdef or ABCDEF for 10, ...,15.
u	int; unsigned decimal number
c	int; single character
s	char *; print characters from the string until a '\0' or the number of characters given by the precision.
f	double; [-]*m.dddddd*, where the number of *d*'s is given by the precision (default 6).
e,E	double; [-]*m.dddddd*e+/-*xx* or [-]*m.dddddd*E+/-*xx*, where the number of *d*'s is given by the precision (default 6).
g,G	double; use %e or %E if the exponent is less than -4 or greater than or equal to the precision; otherwise use %f. Trailing zeros and a trailing decimal point are not printed.
p	void *; pointer (implementation-dependent representation).
%	no argument is converted; print a %

Along with the conversion characters, precision can also be defined for reserving or limiting the space for output. Most of the format conversions have been illustrated in earlier sections. One exception is the precision as it relates to strings. The following table shows the effect of a variety of specifications in printing ``hello, world'' (12 characters). We have put colons around each field so you can see it extent.

:%s:	:hello, world:
:%10s:	:hello, world:
:%.10s:	:hello, wor:
:%-10s:	:hello, world:

:%.15s:	:hello, world:
:%-15s:	:hello, world :
:%15.10s:	: hello, wor:
:%-15.10s:	:hello, wor :

A warning: printf uses its first argument to decide how many arguments follow and what their type is. It will get confused, and you will get wrong answers, if there are not enough arguments of if they are the wrong type. You should also be aware of the difference between these two calls:

```
printf(s);          /* FAILS if s contains % */
printf("%s", s);   /* SAFE */
```

The function sprintf does the same conversions as printf does, but stores the output in a string:

```
int sprintf(char *string, char *format, arg1, arg2, ...);
```

sprintf formats the arguments in arg1, arg2, etc., according to format as before, but places the result in string instead of the standard output; string must be big enough to receive the result.

7. Escape Sequences

The printf() and scanf() statements follow the combination of characters called as escape sequences. The following are the escape sequences with their use and ASCII value.

Escape Sequence	Use	ASCII Value
\n	New line	10
\b	Backspace	8
\f	Form Feed	12
\'	Single Quote	39
\\	Backslash	92
\0	Null	0

\t	Horizontal Tab	9
\r	Carriage Return	13
\a	Alert	7
\"	Double Quote	34
\v	Vertical Tab	11
\?	Question Mark	63

Escape sequences facilitate formatting of output, generating alerts and printing some characters which can not be directly printed using output functions.

8. Summary

Input and output are not part of the C language itself, so we have not emphasized them in our previous lessons. Nonetheless, programs interact with their environment in much more complicated ways than those we have shown before. In this lesson we have described the standard library, a set of functions that provide input and output functions.

9. Short Answer Type Questions

1. What are the basic input output functions available in C?
2. What do you mean by formatted I/O?
3. What is the difference between getch() and getche() functions?
4. Why ampersand (&) is used in scanf while reading numeric or character data types?

10. Long Answer Type Questions

1. Discuss in detail the various types of input and output functions used in C language. Also discuss the syntax of each of the functions giving examples.
2. Which characters are used for conversion in printf and scanf statements?
3. What are the various escape sequences? Write their use as well.

11. Suggested Books

Programming with ANSI and Turbo C	Ashok N. Kamthane
C Programming	E. Balagurusamy
Application Programming in C	R. S. Salaria

BA PART-2 (SEM-3) **PAPER : BAP-201**
C Programming and Data Structures

Lesson No. 5 **Author: Dr. DharamVeer Sharma**
Converted into SLM by: Dr. Vishal Singh

Operators and Expressions

1. Objectives

In this lesson we shall discuss the various types of operators available in the C language. We shall also discuss the method of using these operators, their precedence and their order of evaluation in expressions.

2. Introduction

In order to perform different types of operations, C uses different kind of operators. An operator indicates an operation to be performed on data that yields a value. With the help of various operators available in C language, one can link the variables and constants. An operand is a data item on which operators perform the operations. C provides four classes of operators. They are 1) Arithmetic 2) Relational 3) Logical and 4) bitwise. Along with these operators there are other operators like unary, conditional, assignment and comma operator.

The following are the various types of operators available in C language:

Type of operator	**Symbolic representation**
Arithmetic	+, -, *, / and %
Relational	>, >, >=, <=, == and !=
Logical	&&, || and !
Increment and decrement	++ and --

Assignment	=
Bitwise	&, |, ^, >>, << and ~
Comma	,
Conditional	? :

3. Arithmetic Operators

The binary arithmetic operators are +, -, *, /, and the modulus operator %. Integer division truncates any fractional part. The expression

```
x % y
```

produces the remainder when x is divided by y, and thus is zero when y divides x exactly. For example, a year is a leap year if it is divisible by 4 but not by 100, except that years divisible by 400 *are* leap years.

Therefore

```
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)

    printf("%d is a leap year\n", year);

else

    printf("%d is not a leap year\n", year);
```

The % operator cannot be applied to a float or double. The direction of truncation for / and the sign of the result for % are machine-dependent for negative operands, as is the action taken on overflow or underflow.

The binary + and - operators have the same precedence, which is lower than the precedence of *, / and %, which is in turn lower than unary + and -. Arithmetic operators associate left to right.

Unary Operators

C provides two unusual operators for incrementing and decrementing variables. The increment operator ++ adds 1 to its operand, while the decrement operator -- subtracts 1. We have frequently used ++ to increment variables, as in

```
if (c == '\n')

    ++nl;
```

The unusual aspect is that ++ and -- may be used either as prefix operators (before the variable, as in ++n), or postfix operators (after the variable: n++). In both cases, the effect is to increment n. But the expression ++n increments n *before* its value is used, while n++ increments n *after* its value has been used. This means that in a context where the value is being used, not just the effect, ++n and n++ are different. If n is 5, then

```
x = n++;
```

sets x to 5, but

```
x = ++n;
```

sets x to 6. In both cases, n becomes 6. The increment and decrement operators can only be applied to variables; an expression like (i+j)++ is illegal.

In a context where no value is wanted, just the incrementing effect, as in

```
if (c == '\n')
    nl++;
```

prefix and postfix are the same. But there are situations where one or the other is specifically called for. For instance, consider the function squeeze(s,c), which removes all occurrences of the character c from the string s.

```
/* squeeze:  delete all c from s */
void squeeze(char s[], int c)
{
   int i, j;
   for (i = j = 0; s[i] != '\0'; i++)
       if (s[i] != c)
           s[j++] = s[i];
   s[j] = '\0';
}
```

Each time a non-c occurs, it is copied into the current j position, and only then is j incremented to be ready for the next character. This is exactly equivalent to

```
if (s[i] != c) {
    s[j] = s[i];
    j++;
}
```

Another example of a similar construction comes from the getline function that we wrote in lesson 1, where we can replace

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```

by the more compact

```
if (c == '\n')
   s[i++] = c;
```

As a third example, consider the standard function strcat(s,t), which concatenates the string t to the end of string s. strcat assumes that there is enough space in s to hold the combination. As we have written it, strcat returns no value; the standard library version returns a pointer to the resulting string.

```
/* strcat:  concatenate t to end of s; s must be big enough */
void strcat(char s[], char t[])
{
    int i, j;
    i = j = 0;
    while (s[i] != '\0') /* find end of s */
        i++;
    while ((s[i++] = t[j++]) != '\0') /* copy t */
        ;
}
```

As each member is copied from t to s, the postfix ++ is applied to both i and j to make sure that they are in position for the next pass through the loop.

4. Relational Operators and Logical Operators

The relational operators are

```
>   >=   <   <=
```

They all have the same precedence. Just below them in precedence are the equality operators:

```
== !=
```

Relational operators have lower precedence than arithmetic operators, so an expression like i < lim-1 is taken as i < (lim-1), as would be expected.

More interesting are the logical operators && and ||. Expressions connected by && or || are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known. Most C programs rely on these properties. For example, here is a loop from the input function getline:

```
for (i=0; i < lim-1 && (c=getchar()) != '\n' && c != EOF; ++i)
    s[i] = c;
```

Before reading a new character it is necessary to check that there is room to store it in the array s, so the test i < lim-1 *must* be made first. Moreover, if this test fails, we must not go on and read another character.

Similarly, it would be unfortunate if c were tested against EOF before getchar is called; therefore the call and assignment must occur before the character in c is tested.

The precedence of && is higher than that of ||, and both are lower than relational and equality operators, so expressions like

```
i < lim-1 && (c=getchar()) != '\n' && c != EOF
```

need no extra parentheses. But since the precedence of != is higher than assignment, parentheses are needed in

```
(c=getchar()) != '\n'
```

to achieve the desired result of assignment to c and then comparison with '\n'.

By definition, the numeric value of a relational or logical expression is 1 if the relation is true, and 0 if the relation is false.

The unary negation operator ! converts a non-zero operand into 0, and a zero operand in 1. A common use of ! is in constructions like

```
if (!valid)
```

rather than

```
if (valid == 0)
```

It's hard to generalize about which form is better. Constructions like !valid read nicely (``if not valid''), but more complicated ones can be hard to understand.

5. Bitwise Operators

C provides six operators for bit manipulation; these may only be applied to integral operands, that is, char, short, int, and long, whether signed or unsigned.

&	bitwise AND
|	bitwise inclusive OR
^	bitwise exclusive OR
<<	left shift
>>	right shift
~	one's complement (unary)

The bitwise AND operator & is often used to mask off some set of bits, for example

```
  n = n & 0177;
```

sets to zero all but the low-order 7 bits of n.

The bitwise OR operator | is used to turn bits on:

```
  x = x | SET_ON;
```

sets to one in x the bits that are set to one in SET_ON.

The bitwise exclusive OR operator ^ sets a one in each bit position where its operands have different bits, and zero where they are the same.

One must distinguish the bitwise operators & and | from the logical operators && and ||, which imply left-to-right evaluation of a truth value. For example, if x is 1 and y is 2, then x & y is zero while x && y is one.

The shift operators << and >> perform left and right shifts of their left operand by the number of bit positions given by the right operand, which must be non-negative. Thus x << 2 shifts the value of x by two positions, filling vacated bits with zero; this is equivalent to multiplication by 4. Right shifting an unsigned quantity always fits the vacated bits with zero. Right shifting a signed quantity will fill with bit signs (``arithmetic shift'') on some machines and with 0-bits (``logical shift'') on others.

The unary operator ~ yields the one's complement of an integer; that is, it converts each 1-bit into a 0-bit and vice versa. For example

```
  x = x & ~077
```

sets the last six bits of x to zero. Note that x & ~077 is independent of word length, and is thus preferable to, for example, x & 0177700, which assumes that x is a 16-bit quantity. The portable form involves no extra cost, since ~077 is a constant expression that can be evaluated at compile time.

As an illustration of some of the bit operators, consider the function getbits(x,p,n) that returns the (right adjusted) n-bit field of x that begins at position p. We assume that bit position 0 is at the right end and that n and p are sensible positive values. For example, getbits(x,4,3) returns the three bits in positions 4, 3 and 2, right-adjusted.

```
/* getbits:  get n bits from position p */

unsigned getbits(unsigned x, int p, int n)

{

   return (x >> (p+1-n)) & ~(~0 << n);

}
```

The expression x >> (p+1-n) moves the desired field to the right end of the word. ~0 is all 1-bits; shifting it left n positions with ~0<<n places zeros in the rightmost n bits; complementing that with ~ makes a mask with ones in the rightmost n bits.

6. Assignment Operator and Expression Evaluation

An expression such as

```
i = i + 2
```

in which the variable on the left side is repeated immediately on the right, can be written in the compressed form

```
i += 2
```

The operator += is called an *assignment operator.*

Most binary operators (operators like + that have a left and right operand) have a corresponding assignment operator *op*=, where *op* is one of

```
+  -  *  /  %  <<  >>  &  ^  |
```

If $expr_1$ and $expr_2$ are expressions, then

$expr_1$ *op*= $expr_2$

is equivalent to

$expr_1$ = ($expr_1$) *op* ($expr_2$)

except that $expr_1$ is computed only once. Notice the parentheses around $expr_2$:

```
x *= y + 1
```

means

```
x = x * (y + 1)
```

rather than

```
  x = x * y + 1
```

As an example, the function bitcount counts the number of 1-bits in its integer argument.

```
  /* bitcount:  count 1 bits in x */

  int bitcount(unsigned x)

  {

      int b;

      for (b = 0; x != 0; x >>= 1)

          if (x & 01)

              b++;

      return b;

  }
```

Declaring the argument x to be an unsigned ensures that when it is right-shifted, vacated bits will be filled with zeros, not sign bits, regardless of the machine the program is run on.

Quite apart from conciseness, assignment operators have the advantage that they correspond better to the way people think. We say ``add 2 to i'' or ``increment i by 2'', not ``take i, add 2, then put the result back in i''. Thus the expression i += 2 is preferable to i = i+2. In addition, for a complicated expression like

```
  yyval[yypv[p3+p4] + yypv[p1]] += 2
```

the assignment operator makes the code easier to understand, since the reader doesn't have to check painstakingly that two long expressions are indeed the same, or to wonder why they're not. And an assignment operator may even help a compiler to produce efficient code.

We have already seen that the assignment statement has a value and can occur in expressions; the most common example is

```
  while ((c = getchar()) != EOF)

      ...
```

The other assignment operators (+=, -=, etc.) can also occur in expressions, although this is less frequent.

In all such expressions, the type of an assignment expression is the type of its left operand, and the value is the value after the assignment.

7. Conditional Expression

The statements

```
if (a > b)

    z = a;

else

    z = b;
```

compute in z the maximum of a and b. The *conditional expression*, written with the ternary operator ``?:'', provides an alternate way to write this and similar constructions. In the expression

expr$_1$? *expr*$_2$: *expr*$_3$

the expression *expr*$_1$ is evaluated first. If it is non-zero (true), then the expression *expr*$_2$ is evaluated, and that is the value of the conditional expression. Otherwise *expr*$_3$ is evaluated, and that is the value. Only one of *expr*$_2$ and *expr*$_3$ is evaluated. Thus to set z to the maximum of a and b,

```
z = (a > b) ? a : b;    /* z = max(a, b) */
```

It should be noted that the conditional expression is indeed an expression, and it can be used wherever any other expression can be. If *expr*$_2$ and *expr*$_3$ are of different types, the type of the result is determined by the conversion rules discussed earlier in the previous lesson. For example, if f is a float and n an int, then the expression

```
(n > 0) ? f : n
```

is of type float regardless of whether n is positive.

Parentheses are not necessary around the first expression of a conditional expression, since the precedence of ?: is very low, just above assignment. They are advisable anyway, however, since they make the condition part of the expression easier to see.

The conditional expression often leads to succinct code. For example, this loop prints n elements of an array, 10 per line, with each column separated by one blank, and with each line (including the last) terminated by a newline.

```
for (i = 0; i < n; i++)

    printf("%6d%c", a[i], (i%10==9 || i==n-1) ? '\n' : ' ');
```

A newline is printed after every tenth element, and after the n-th. All other elements are followed by one blank. This might look tricky, but it's more compact than the equivalent if-else. Another good example is

```
printf("You have %d items %s.\n", n, n==1 ? "" : "s");
```

8. Comma Operator

The comma operator is used to separate two or more expressions. The comma operator has the lowest priority among all the operators. It is not essential to enclose the expressions with comma operators within the parenthesis.

Example:

A=2,b=3,c=5; or (A=2,b=3,c=5;)

are valid statements.

9. Operator Precedence and Associativity

Table 2.1 summarizes the rules for precedence and associativity of all operators, including those that we have not yet discussed. Operators on the same line have the same precedence; rows are in order of decreasing precedence, so, for example, *, /, and % all have the same precedence, which is higher than that of binary + and -. The ``operator'' () refers to function call. The operators -> and . are used to access members of structures; they will be covered in lessons 11 and 12, along with sizeof (size of an object). Lesson 11 discusses * (indirection through a pointer) and & (address of an object).

Operators **(in order of their precedence)**	**Operation**	**Associativity**	**Priority**
() [] -> .	Function call Array expression or square bracket Structure operator Structure operator	left to right	1
! ~ ++ -- + -	Not operator Ones complement Increment Decrement Unary plus Unary minus	right to left	2

* & (*type*) sizeof	Pointer operator Address operator Type cast Size of an object		
* / %	Multiplication Division Modular division	left to right	3
+ -	Addition Subtraction	left to right	4
<< >>	Left shift Right shift	left to right	5
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	left to right	6
== !=	Equality Inequality	left to right	7
&	Bitwise AND	left to right	8
^	Bitwise XOR	left to right	9
|	Bitwise OR	left to right	10
&&	Logical AND	left to right	11
||	Logical OR	left to right	12
?:	Conditional operator	right to left	13
= += -= *= /= %= &= ^= |= <<= >>=	Assignment operators	right to left	14
,	Comma operator	left to right	15

Unary & +, -, and * have higher precedence than the binary forms.

Note that the precedence of the bitwise operators &, ^, and | falls below == and !=. This implies that bit-testing expressions like

```
if ((x & MASK) == 0) ...
```

must be fully parenthesized to give proper results.

C, like most languages, does not specify the order in which the operands of an operator are evaluated. (The exceptions are &&, ||, ?:, and `,'.) For example, in a statement like

```
x = f() + g();
```

f may be evaluated before g or vice versa; thus if either f or g alters a variable on which the other depends, x can depend on the order of evaluation. Intermediate results can be stored in temporary variables to ensure a particular sequence.

Similarly, the order in which function arguments are evaluated is not specified, so the statement

```
printf("%d %d\n", ++n, power(2, n));   /* WRONG */
```

can produce different results with different compilers, depending on whether n is incremented before power is called. The solution, of course, is to write

```
++n;

printf("%d %d\n", n, power(2, n));
```

Function calls, nested assignment statements, and increment and decrement operators cause ``side effects'' - some variable is changed as a by-product of the evaluation of an expression. In any expression involving side effects, there can be subtle dependencies on the order in which variables taking part in the expression are updated. One unhappy situation is typified by the statement

```
a[i] = i++;
```

The question is whether the subscript is the old value of i or the new. Compilers can interpret this in different ways, and generate different answers depending on their interpretation. The standard intentionally leaves most such matters unspecified. When side effects (assignment to variables) take place within an expression is left to the discretion of the compiler, since the best order depends strongly on machine architecture. (The standard does specify that all side effects on arguments take effect before a function is called, but that would not help in the call to printf above.)

The moral is that writing code that depends on order of evaluation is a bad programming practice in any language. Naturally, it is necessary to know what things to avoid, but if you don't know *how* they are done on various machines, you won't be tempted to take advantage of a particular implementation.

10. Summary

Operators are the building blocks of expressions. The C language uses different kind of operators. There are arithmetic, relational, logical, assignment, conditional, comma and bitwise operators available in C.

11. Short Answer Type Questions

1. What is the precedence of different arithmetic operators?
2. What is a ternary operator?
3. What are the various relational operators?
4. What is the role of comma operator?

12. Long Answer Type Questions

1. What is the difference between precedence and associativity?
2. What are the rule governing the use of logical operators?
3. How bitwise operators are used?

13. Suggested Books

Programming with ANSI and Turbo C	Ashok N. Kamthane
Programming using C	E. Balagurusamy
Application Programming in C	R. S. Salaria

BA PART-2 (SEM-3) **PAPER : BAP-201**
C Programming and Data Structures

Lesson No. 6 **Author: Dr. DharamVeer Sharma**
Converted into SLM by: Dr. Vishal Singh

Sequential and Conditional Control Statements

1. Objectives

In this lesson we shall discuss the various conditional control structures available in the C language. These constructs provide branching with in the program based on some condition.

2. Introduction

The control-flow of a language specifies the order in which computations are performed. In C language there are sequential, conditional and iterative control structures are available for program design. In this lesson we shall discuss the various conditional control structures and iterative control structure will be discussed in the next lesson. Conditional constructs are required for making decision and choosing some execution path based on the satisfied condition.

3. Statements and Blocks

An expression such as x = 0 or i++ or printf(...) becomes a *statement* when it is followed by a semicolon, as in

```
x = 0;

i++;

printf(...);
```

In C, the semicolon is a statement terminator, rather than a separator as it is in languages like Pascal.

Braces { and } are used to group declarations and statements together into a *compound statement*, or *block*, so that they are syntactically equivalent to a single statement. The braces that surround the statements of a function are one obvious example; braces around multiple statements after an if, else, while, or for are another. (Variables can be declared inside *any* block) There is no semicolon after the right brace that ends a block. A block can be created anywhere with in the program.

Example

```
    main()
    {
        int a = 10;
        {
            int a = 20;
            printf("Value of a inside the block is -> %d",a);
        }
        printf("Value of a outside the block is -> %d",a);
        return 0;
    }
```

In the above example, for the second declaration of a, a's scope is limited to the block only and output will be 20 for the first printf statement and for the second printf the output will be 10.

4. if ... else Construct

The if-else statement is used to express decisions. Formally the syntax is

```
    if (expression)
        statement1
    else
        statement2
```

where the else part is optional. The *expression* is evaluated; if it is true (that is, if *expression* has a non-zero value), *statement*$_1$ is executed. If it is false (*expression* is zero) and if there is an else part, *statement*$_2$ is executed instead.

Since an if tests the numeric value of an expression, certain coding shortcuts are possible. The most obvious is writing

```
    if (expression)
```

instead of

```
if (expression != 0)
```

Sometimes this is natural and clear; at other times it can be cryptic.

Because the else part of an if-else is optional, there is an ambiguity when an else if omitted from a nested if sequence. This is resolved by associating the else with the closest previous else-less if. For example, in

```
if (n > 0)
        if (a > b)
                z = a;
        else
                z = b;
```

the else goes to the inner if, as we have shown by indentation. If that isn't what you want, braces must be used to force the proper association:

```
if (n > 0) {
        if (a > b)
                z = a;
}
else
        z = b;
```

The ambiguity is especially pernicious in situations like this:

```
if (n > 0)
        for (i = 0; i < n; i++)
                if (s[i] > 0) {
                        printf("...");
                        return i;
                }
else        /* WRONG */
        printf("error -- n is negative\n");
```

The indentation shows unequivocally what you want, but the compiler doesn't get the message, and associates the else with the inner if. This kind of bug can be hard to find; it's a good idea to use braces when there are nested ifs.

By the way, notice that there is a semicolon after z = a in

```
if (a > b)

        z = a;

else

        z = b;
```

This is because grammatically, a *statement* follows the if, and an expression statement like ``z = a;'' is always terminated by a semicolon.

5. else ... if

The construction

```
if (expression)

        statement

else if (expression)

        statement

else if (expression)

        statement

else if (expression)

        statement

else

        statement
```

occurs so often that it is worth a brief separate discussion. This sequence of if statements is the most general way of writing a multi-way decision. The *expressions* are evaluated in order; if an *expression* is true, the *statement* associated with it is executed, and this terminates the whole chain. As always, the code for each *statement* is either a single statement, or a group of them in braces.

The last else part handles the ``none of the above'' or default case where none of the other conditions is satisfied. Sometimes there is no explicit action for the default; in that case the trailing

```
else

        statement
```

can be omitted, or it may be used for error checking to catch an ``impossible'' condition.

To illustrate a three-way decision, here is a binary search function that decides if a particular value x occurs in the sorted array v. The elements of v must be in increasing order. The function returns the position (a number between 0 and n-1) if x occurs in v, and -1 if not.

Binary search first compares the input value x to the middle element of the array v. If x is less than the middle value, searching focuses on the lower half of the table, otherwise on the upper half. In either case, the next step is to compare x to the middle element of the selected half. This process of dividing the range in two continues until the value is found or the range is empty.

```
/* binsearch:  find x in v[0] <= v[1] <= ... <= v[n-1] */
int binsearch(int x, int v[], int n)
{
        int low, high, mid;
        low = 0;
        high = n - 1;
        while (low <= high) {
                mid = (low+high)/2;
                if (x < v[mid])
                        high = mid + 1;
                else if (x  > v[mid])
                        low = mid + 1;
                else    /* found match */
                        return mid;
        }
        return -1;   /* no match */
}
```

The fundamental decision is whether x is less than, greater than, or equal to the middle element v[mid] at each step; this is a natural for else-if.

Utmost care should be taken while use conditional construct, as is evident from the following example.

```
if (day == 1)

        printf("Monday");

if (day == 2)

        printf("Tuesday");

if (day == 3)

        printf("Wednesday");

if (day == 4)

        printf("Thursday");

if (day == 5)

        printf("Friday");

if (day == 6)

        printf("Saturday");

else

        printf("Sunday");
```

The above use of if is wrong as for any value of day between 1 and 5 it will print the Sunday as well because in the last if statement it will always printf Sunday if value of day is not 6. Therefore, in the above example if else if construct should be used, as given below.

```
if (day == 1)

        printf("Monday");

else if (day == 2)

        printf("Tuesday");

else if (day == 3)

        printf("Wednesday");

else if (day == 4)

        printf("Thursday");

else if (day == 5)

        printf("Friday");
```

```
else if (day == 6)
        printf("Saturday");
else
        printf("Sunday");
```

In this case any condition will be checked only if its previous condition is false.

6. Using logical operators in if construct

Logical operators are indispensable part of if … else construct, but these should be use with utmost caution. There is a need of understanding the way these are evaluated.

```
if (condition1 && condition2)
        statement
```

In this construct condition2 is evaluated only if condition1 is true, otherwise condition2 is never reached. Therefore if some calculation is involved in condition2 then that calculation will also not be performed. Therefore, care should be taken while using && operator.

```
if (condition1 || condition2)
        statement1
```

In this construct conditon2 is evaluated only if condition1 is false, otherwise condition2 is never reached. Therefore the problem is the same as in the case of && operator.

7. switch … case Construct

The switch statement is a multi-way decision that tests whether an expression matches one of a number of *constant* integer values, and branches accordingly.

```
switch (expression) {
        case const-expr: statements
        case const-expr: statements
        default: statements
}
```

Important distinction between use of if and switch construct is that for each switch construct there is an equivalent if construct available. However, the reverse is not true. switch can replace only those if constructs where the value of only one variable is tested for different integer values.

Example

```
    if (day == 1)
        printf("Monday");
    else if (day == 2)
        printf("Tuesday");
    else if (day == 3)
        printf("Wednesday");
    else if (day == 4)
        printf("Thursday");
    else if (day == 5)
        printf("Friday");
    else if (day == 6)
        printf("Saturday");
    else
        printf("Sunday");
```

For this situation where value of day is checked, switch construct is the most suitable.

```
    switch (day)
    {
        case 1: printf("Monday");
            break;
        case 2: printf("Tuesday");
            break;
        case 3: printf("Wednesday");
            break;
        case 4: printf("Thursday");
            break;
        case 5: printf("Friday");
            break;
        case 6: printf("Saturday");
                break;
        case 7: printf("Sunday");
                break;
    }
```

Each case is labeled by one or more integer-valued constants or constant expressions. If a case matches the expression value, execution starts at that case. All case expressions must be different. The case labeled default is executed if none of the other cases are satisfied. A default is optional; if it isn't there and if none of the cases match, no action at all takes place. Cases and the default clause can occur in any order.

Following is a program to count the occurrences of each digit, white space, and all other characters, using a switch:

```
    #include <stdio.h>
    main()  /* count digits, white space, others */
    {
        int c, i, nwhite, nother, ndigit[10];
        nwhite = nother = 0;
        for (i = 0; i < 10; i++)
            ndigit[i] = 0;
        while ((c = getchar()) != EOF) {
            switch (c) {
                case '0': case '1': case '2': case '3': case '4':
                case '5': case '6': case '7': case '8': case '9':
                    ndigit[c-'0']++;
                    break;
                case ' ': case '\n': case '\t': nwhite++;
                    break;
                default:
                    nother++;
```

```
                        break;
                }
        }
        printf("digits =");
        for (i = 0; i < 10; i++)
                printf(" %d", ndigit[i]);
        printf(", white space = %d, other = %d\n", nwhite, nother);
        return 0;
}
```

Note that multiple cases can be combined as is in the case with the digits in the above program. This is similar to the multiple conditions combined using logical or (||) with in one if statement. Therefore, it can be said that if multiple conditions involving single variable but combined using logical or (||) are under one if then those can be safely converted to switch construct. But if the logical and has been used to combine multiple conditions or if the multiple conditions involve more than one variable then switch construct can not be used.

The break statement causes an immediate exit from the switch. Because cases serve just as labels, after the code for one case is done, execution *falls through* to the next unless you take explicit action to escape. break and return are the most common ways to leave a switch. A break statement can also be used to force an immediate exit from while, for, and do loops, as will be discussed later in this lesson.

Falling through cases is a mixed blessing. On the positive side, it allows several cases to be attached to a single action, as with the digits in this example. But it also implies that normally each case must end with a break to prevent falling through to the next. Falling through from one case to another is not robust, being prone to disintegration when the program is modified. With the exception of multiple labels for a single computation, fall-throughs should be used sparingly, and commented.

Default case is used when none of the conditions specified under cases are encountered and thus may be ignored or an appropriate action may be taken. It mean all those cases which have not been handled using case labels.

As a matter of good form, put a break after the last case (the default here) even though it's logically unnecessary. Some day when another case gets added at the end, this bit of defensive programming will save you.

8. goto and labels

C provides the infinitely-abusable goto statement, and labels to branch to. Formally, the goto statement is never necessary, and in practice it is almost always easy to write code without it. We have not used goto in this book.

Nevertheless, there are a few situations where gotos may find a place. The most common is to abandon processing in some deeply nested structure, such as breaking out of two or more loops at once. The break statement cannot be used directly since it only exits from the innermost loop. Thus:

```
for ( ... )
        for ( ... ) {
                ...
                if (disaster)
                goto error;
        }
        ...
error:
/* clean up the mess */
```

This organization is handy if the error-handling code is non-trivial, and if errors can occur in several places.

A label has the same form as a variable name, and is followed by a colon. It can be attached to any statement in the same function as the goto. The scope of a label is the entire function.

As another example, consider the problem of determining whether two arrays a and b have an element in common. One possibility is

```
for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
                if (a[i] == b[j])
                        goto found;
                /* didn't find any common element */
...
found:
/* got one: a[i] == b[j] */
```

```
    ...
```

Code involving a goto can always be written without one, though perhaps at the price of some repeated tests or an extra variable. For example, the array search becomes

```
    found = 0;

    for (i = 0; i < n && !found; i++)

        for (j = 0; j < m && !found; j++)

            if (a[i] == b[j])

                found = 1;

    if (found)

        /* got one: a[i-1] == b[j-1] */

        ...

    else

        /* didn't find any common element */

        ...
```

With a few exceptions like those cited here, code that relies on goto statements is generally harder to understand and to maintain than code without gotos. Although we are not dogmatic about the matter, it does seem that goto statements should be used rarely, if at all.

9. Summary

Conditional flow of program provides decision making ability. In C language there are if …else and switch constructs for implementing conditional flow. if statements can be nested. In case of switch statement value of some variable is checked against integer constants. Logical operators are to be used cautiously for combining conditions.

10. Short Answer Type Questions

1. What are the various conditional constructs available in C?
2. What is the purpose of switch statement?
3. Why break is needed after cases?
4. What is a default case?

11. Long Answer Type Questions

1. What is the difference between if and switch constructs?
2. In what type of situation switch will be preferred over if statements?
3. What are the rules of using logical operators with conditions?

4. Is it possible to replace all kinds of if constructs with switch? If not then why?
5. WAP to check whether the person is eligible for voting or not.
6. WAP to calculate total modes, percentage and division of the student using nested if Else.
7. WAP to show use of switch statement.

12. Suggested Books

Programming with ANSI and Turbo C	Ashok N. Kamthane
Programming using C	E. Balagurusamy
Application Programming in C	R. S. Salaria

BA PART-2 (SEM-3) **PAPER : BAP-201**

C Programming and Data Structures

Lesson No. 7 **Author: Dr. DharamVeer Sharma**

Converted into SLM by: Dr. Vishal Singh

Iterative Control Statements

1. Objectives

In this lesson we shall discuss the various iterative control structures available in the C language. These constructs provide repetitive execution of some set of statements.

2. Introduction

Iterative control structures provide repetitive computations. In case where some set of statements are to be executed repeatedly, the iterative control structures can be used. The C language provides three iterative control structures namely for, while and do ... while loops. for and while loops are entry control loops where as do ... while is an exit controlled loop.

3. while Loop

As discussed earlier while is an entry controlled loop, means if the condition is true only then statements under the loop will be executed. In

while (*expression*)

statement

the *expression* is evaluated, if it is non-zero, *statement* is executed and *expression* is re-evaluated. This cycle continues until *expression* becomes zero, at which point execution resumes after *statement*. However if the expression evaluates to zero then the statement is not executed. while loop is used in case the number of iterations are not known in advance.

The following variant of while loop produces an infinite loop

while (1)

In this case since expression always evaluates to non-zero value, the loop is not terminated by just checking the expression. Instead, some internal control breaking mechanism is required to come out of the loop. The mechanism has been discussed later in this lesson.

4. for Loop

Like while loop, for is also an entry controlled loop. The for statement

```
for (expr1; expr2; expr3)

        statement
```

is equivalent to

```
expr1;

while (expr2) {

        statement

        expr3;

}
```

except for the behaviour of break or continue.

A loop control variable is used for controlling the number of times for which the loop will be executed.

Grammatically, the three components of a for loop are expressions. Most commonly, *expr*$_1$ and *expr*$_3$ are assignments or function calls and *expr*$_2$ is a relational expression. Any of the three parts can be omitted, although the semicolons must remain. If *expr*$_1$ or *expr*$_3$ is omitted, it is simply dropped from the expansion. If the test, *expr*$_2$, is not present, it is taken as permanently true, so

```
for (;;) {

        ...

}
```

is an ``infinite'' loop, presumably to be broken by other means, such as a break or return.

The three expressions of the for loop are executed in the following way:

expr1: It is executed only once, primarily for initializing the loop control variable.

expr2: It is executed repeatedly for checking the truthfulness of condition, up to which the body of the loop is to be executed. When the condition evaluates to false the loop is terminated.

expr3: It is also repeatedly executed for incrementing or decrementing the value of the loop control variable.

In general, for loop is used when number of iterations is known in advance and while is used when iterations are not known. However these can be used interchangeably. Whether to use while or for largely becomes a matter of personal preference. For example, in

```
while ((c = getchar()) == ' ' || c == '\n' || c = '\t')
        ;  /* skip white space characters */
```

there is no initialization or re-initialization, so the while is most natural.

The for is preferable when there is a simple initialization and increment since it keeps the loop control statements close together and visible at the top of the loop. This is most obvious in

```
for (i = 0; i < n; i++)
        ...
```

which is the C idiom for processing the first n elements of an array, the analog of the Fortran DO loop or the Pascal for. The analogy is not perfect, however, since the index variable i retains its value when the loop terminates for any reason. Because the components of the for are arbitrary expressions, for loops are not restricted to arithmetic progressions. Nonetheless, it is bad style to force unrelated computations into the initialization and increment of a for, which are better reserved for loop control operations.

As a larger example, here is another version of atoi for converting a string to its numeric equivalent. It copes with optional leading white space and an optional + or - sign.

The structure of the program reflects the form of the input:

skip white space, if any

get sign, if any

get integer part and convert it

Each step does its part, and leaves things in a clean state for the next. The whole process terminates on the first character that could not be part of a number.

```
#include <ctype.h>

/* atoi:  convert s to integer; version 2 */

int atoi(char s[])
```

```
{
    int i, n, sign;
    for (i = 0; isspace(s[i]); i++)  /* skip white space */
    ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')  /* skip sign */
        i++;
    for (n = 0; isdigit(s[i]); i++)
        n = 10 * n + (s[i] - '0');
    return sign * n;
}
```

The standard library provides a more elaborate function strtol for conversion of strings to long integers.

The advantages of keeping loop control centralized are even more obvious when there are several nested loops. The following function is a Shell sort for sorting an array of integers.

The basic idea of this sorting algorithm, which was invented in 1959 by D. L. Shell, is that in early stages, far-apart elements are compared, rather than adjacent ones as in simpler interchange sorts. This tends to eliminate large amounts of disorder quickly, so later stages have less work to do. The interval between compared elements is gradually decreased to one, at which point the sort effectively becomes an adjacent interchange method.

```
/* shellsort:  sort v[0]...v[n-1] into increasing order */
void shellsort(int v[], int n)
{
    int gap, i, j, temp;
    for (gap = n/2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap) {
                temp = v[j];
                v[j] = v[j+gap];
```

```
                        v[j+gap] = temp;
                    }
    }
```

There are three nested loops. The outermost controls the gap between compared elements, shrinking it from n/2 by a factor of two each pass until it becomes zero. The middle loop steps along the elements. The innermost loop compares each pair of elements that is separated by gap and reverses any that are out of order. Since gap is eventually reduced to one, all elements are eventually ordered correctly. Notice how the generality of the for makes the outer loop fit in the same form as the others, even though it is not an arithmetic progression.

One final C operator is the comma ``,'', which most often finds use in the for statement. A pair of expressions separated by a comma is evaluated left to right, and the type and value of the result are the type and value of the right operand. Thus in a for statement, it is possible to place multiple expressions in the various parts, for example to process two indices in parallel. This is illustrated in the function reverse(s), which reverses the string s in place.

```
    #include <string.h>

    /* reverse:  reverse string s in place */

    void reverse(char s[])

    {

    int c, i, j;


        for (i = 0, j = strlen(s)-1; i < j; i++, j--) {

            c = s[i];

            s[i] = s[j];

            s[j] = c;

        }

    }
```

The commas that separate function arguments, variables in declarations, etc., are *not* comma operators, and do not guarantee left to right evaluation.

Comma operators should be used sparingly. The most suitable uses are for constructs strongly related to each other, as in the for loop in reverse, and in macros where a multistep computation has to be a single expression. A comma expression might also be appropriate for the exchange of elements in reverse, where the exchange can be thought of a single operation:

```
        for (i = 0, j = strlen(s)-1; i < j; i++, j--)
                c = s[i], s[i] = s[j], s[j] = c;
```

Forms of for loop

Syntax	Output	Remarks
for (;;)	Infinite loop	No arguments means condition is always true, therefore the loop executes for infinite number of times.
for (a=0;a<=20;)	Infinite loop	Value of 'a' is not modified, therefore, condition will always evaluate to true making the loop an infinite loop.

for (a=0;a<=10;a++)
printf("%d ",a);
	Displays values from 0 to 10	Since initial value of 'a' is 0 and it is incremented by one, the values from 0 to 10 will be printed. When 'a' will become 11 the condition will be false and loop will terminate.

for (a=10;a>=0;a--)	Displays values from 10 to 1	Since initial value of 'a' is 10 and it is decremented by one, the values from 10 to 0 will be printed. When 'a' will become -1 the condition will be false and loop will terminate.

5. do … while Loop

As we discussed in lesson 1, the while and for loops test the termination condition at the top. By contrast, the third loop in C, the do-while, tests at the bottom *after* making each pass through the loop body; the body is always executed at least once. Therefore, do … while loop is termed as exit controlled loop.

The syntax of the do is

```
do
        statement
while (expression);
```

The *statement* is executed, then *expression* is evaluated. If it is true, *statement* is evaluated again, and so on. When the expression becomes false, the loop terminates. Except for the sense of the test, do-while is equivalent to the Pascal repeat-until statement.

Experience shows that do-while is much less used than while and for. Nonetheless, from time to time it is valuable, as in the following function itoa, which converts a number to a character string (the inverse of atoi). The job is slightly more complicated than might be thought at first, because the easy methods of generating the digits generate them in the wrong order. We have chosen to generate the string backwards, then reverse it.

```
/* itoa:  convert n to characters in s */
void itoa(int n, char s[])
{
        int i, sign;
        if ((sign = n) < 0)  /* record sign */
                n = -n;          /* make n positive */
        i = 0;
        do {      /* generate digits in reverse order */
                s[i++] = n % 10 + '0';  /* get next digit */
        } while ((n /= 10) > 0);    /* delete it */
        if (sign < 0)
                s[i++] = '-';
        s[i] = '\0';
        reverse(s);
}
```

The do-while is necessary, or at least convenient, since at least one character must be installed in the array s, even if n is zero. We also used braces around the single statement that makes up the body of the do-while, even though they are unnecessary, so the hasty reader will not mistake the while part for the *beginning* of a while loop.

do … while construct is best suited for situations where some program or block is to be repeatedly executed but that must be executed at least once. Following is the example demonstrating the use of do … while construct:

Example

```
    /* program for finding sum any n numbers */

    #include <stdio.h>

    #include <conio.h>

    void main()

    {

        int i, x, n, sum;

        char choice;

        do {

            sum = 0;

            clrscr();

            printf("Enter the number of values -> ");

            scanf("%d",&n);

            for (i=0; i < n;i++)

            {

                printf("\nEnter some number -> ");

                scanf("%d",&x);

                sum += x;

            }

            printf("\nSum of entered numbers is -> %d",sum);

            printf("Want to run the program again <Y/N>");

            choice = getch();


        } while (choice == 'Y' || choice == 'y');
        printf("\n!!! That's all folks !!!");

        getch();

    }
```

In the above example the program runs for the first time and them asks the user if he wants to run the program again. Depending on the choice of the user the program either executes again or is exits.

6. Nested loops

Loops can be nested in any order within a program. If iterations with in iterations are to be performed then nested loops can be use. In such cases loop control variable, which controls the number of iterations to be performed, should be chosen separately for each of the inner loops.

This can be well demonstrated from the following example in which all possible outcomes of throwing three dice can be generated:

Example

```
    #include <stdio.h>

    #include <conio.h>

    void main()

    {

        int i,j,k;

        printf("Possible outcomes of throwing three dice are -> \n");

        for (i=1;i<=6;i++)

            for (j=1;j<=6;j++)

                for (k=1;k<=6;k++)

                    printf("%d %d %d\t",I,j,k);

    }
```

This program will produce all possible outcomes of throwing three dice simultaneously.

7. Sequence Breaking Control Statements

It is sometimes convenient to be able to exit from a loop other than by testing at the top or bottom. The break statement provides an early exit from for, while, and do, just as from switch. A break causes the innermost enclosing loop or switch to be exited immediately.

The following function, trim, removes trailing blanks, tabs and newlines from the end of a string, using a break to exit from a loop when the rightmost non-blank, non-tab, non-newline is found.

```
/* trim:  remove trailing blanks, tabs, newlines */
int trim(char s[])
{
    int n;
    for (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
    s[n+1] = '\0';
    return n;
}
```

strlen returns the length of the string. The for loop starts at the end and scans backwards looking for the first character that is not a blank or tab or newline. The loop is broken when one is found, or when n becomes negative (that is, when the entire string has been scanned). You should verify that this is correct behavior even when the string is empty or contains only white space characters.

The continue statement is related to break, but less often used; it causes the next iteration of the enclosing for, while, or do loop to begin. In the while and do, this means that the test part is executed immediately; in the for, control passes to the increment step. The continue statement applies only to loops, not to switch. A continue inside a switch inside a loop causes the next loop iteration.

As an example, this fragment processes only the non-negative elements in the array a; negative values are skipped.

```
for (i = 0; i < n; i++)
    if (a[i] < 0)   /* skip negative elements */
        continue;
... /* do positive elements */
```

The continue statement is often used when the part of the loop that follows is complicated, so that reversing a test and indenting another level would nest the program too deeply.

8. Summary

Iterative control structures are used when some statements are to be executed repetitively. In C language, there are three iterative control structures available. for and while are entry controlled loops and can be used interchangeably. do …while is an exit controlled loop. for or while loops should not be used in place of do … while loop.

9. Short Answer Type Questions

1. What are the various iterative controlled structures available in C?
2. What is the difference between for and while loop?
3. Atleast how many times statements are executed in do … while loop?
4. What is the difference between break and continue?

10. Long Answer Type Questions

1. Discuss in detail the syntax and use of for and while loops.
2. Is it possible to use for or while loops in place of do … while loop? Explain your answer.
3. Explain the meaning of sequence breaking control statements.
4. WAP to print the Even numbers less than 100.
5. WAP to print the following :

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

11. Suggested Books

Programming with ANSI and Turbo C	Ashok N. Kamthane
Programming using C	E. Balagurusamy
Application Programming in C	R. S. Salaria

BA PART-2 (SEM-3) **PAPER : BAP-201**

C Programming and Data Structures

Lesson No. 8 **Author: Dr. DharamVeer Sharma**

Converted into SLM by: Dr. Vishal Singh

ARRAYS

1. Introduction

When multiple elements of the same data type are to be used, then we need some such identifier which can store these multiple elements. In the C language array is one such data structure that can store groups of similar data type elements. These elements are stored in contiguous memory area. The array elements are accessed by providing the name of the storage area or the array and a subscript representing the position of the element within array.

2. Objectives

We shall discuss the declaration, initialization, printing and manipulation of array elements in this lesson. We shall see how arrays are passed to functions. We shall also discuss multi-dimensional array and using pointers with arrays.

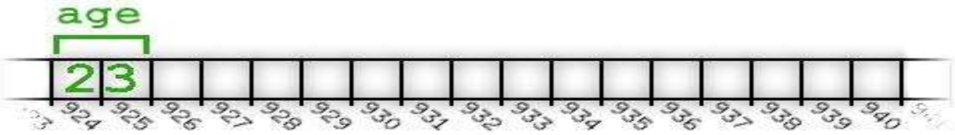
3. Array basics

Let's start by looking at a single variable used to store a person's age.

```
#include <stdio.h>

int main()
{
```

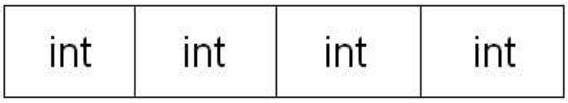
```
5:   short age;
6:   age=23;
7:   printf("%d\n", age);
8:   return 0;
9: }
```

Not much to it. The variable age is created at line (5) as a short. A value is assigned to age. Finally, age is printed to the screen.



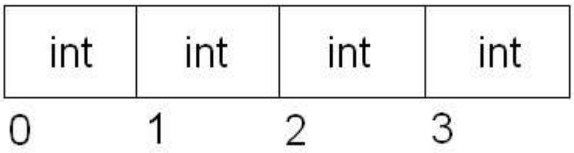
Now let's keep track of 4 ages instead of just one. We could create 4 separate variables, but 4 separate variables have limited appeal. (If using 4 separate variables **is** appealing to you, then consider keeping track of 1000 ages instead of just 4). Rather than using 4 separate variables, we'll use an array which can store a group of similar data type elements as single entity and whose each element is accessed by providing its offset with in the array.

An array is a simple sequence of objects. All of the objects in the sequence are of the same type. The following example presents an array of four integers.



Array of four integers

Each cell of the array is accessed through its index number. Arrays are zero-based. Thus the first cell is defined by index 0, the second by index 1 and so on.



Arrays are declared in the following manner:

type variable_name[array_size];

The following examples show different ways to declare various arrays.

int my_int_array[4]; *// An array of 4 integers*

```
double my_double_array[10]; // An array of 10 doubles
```

Accessing an element within an array

Elements within an array can be accessed (for reading or writing) with the subscript operator **[ ]**.

Example -

```
int my_array[10]; // create an array of 10 integers

my_array[3] = 15; // store the number 15 in the 4th element of my-array

cout << my_array[3]; // display the number 15 we just put in
```

NOTE: In C, arrays do not have boundary checking. This means that the programmer is responsible for knowing the number of cells in the array and thus, the last valid index which can be referenced. Accessing an element past the end of the array bounds will not cause a compiler error, but will crash your program at an unpredictable (but usually the worst possible) time. This is a very common bug which is found even in some of the most popular commercial software packages. Be careful – this is one of the most difficult bugs to track down.

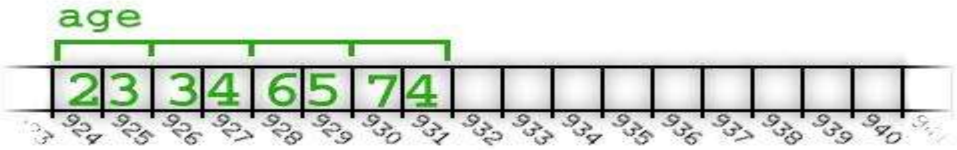
Example -

```
int the_array[2];           //array of 2 integers

the_array[0] = 42;          //valid access

the_array[1] = 1776;        //valid access

the_array[3] = 1492;        //oops! Error but not reported by compiler.
```

Here's how to create an array and one way to initialize an array:

```
1: #include <stdio.h>
2:
3: int main()
4: {
5:   short age[4];
6:   age[0]=23;
7:   age[1]=34;
8:   age[2]=65;
9:   age[3]=74;
10:   return 0;
11: }
```

On line (5), an array of 4 shorts is created. Values are assigned to each variable in the array on line (6) through line (9).



Accessing any single short variable or element, in the array is straightforward. Simply provide a number in square braces next to the name of the array. The number identifies which of the 4 elements in the array you want to access.

The program above shows that the first element of an array is accessed with the number 0 rather than 1. Later We'll discuss why 0 is used to indicate the first element in the array.

4. Array Initialization

When an array of any type is created, the cells within the array are not empty. Depending on your setup, they will either have an already initialized value depending on the type of array you created or the cells will contain garbage values left over from previous use of that piece of memory. In either case, it is usually a good idea to initialize the elements of the array prior to use. Two common ways of initializing an array is through an initializer list or through a loop. Array elements can be initialized at the time of declaration of the array as following.

Example (initializer list) -

```
int int_arr[] = { 34, 68, 7, 9, 20 };

double double_arr[] = { 22.78, 9.7, 3.1415, 2.71 };

char name[10] = "John";

char name[10]= { 'J', 'o','h','n','\0'};
```

In all the above declaration the array has been initialized there itself. If an array is partially initialized, then the remaining elements are automatically initialized to 0 in case of numeric arrays.

```
int a[10] = {0};
```

The above example is a handy way of initializing all elements of an array to 0.

NOTE: When initializer lists are used, the size of the array inside the brackets is not needed. You are free to explicitly place the number there, but if you do not, the compiler will know the size based on the number of elements you initialized the array with.

Example (loop) -

```
const int ARRAY_SIZE = 10;
```

```
int my_array[ARRAY_SIZE];
for (int i = 0; i < ARRAY_SIZE; i++)
{
        my_array[i] = 0;
}
```

This method is useful when arrays are to be initialized with in the program.

NOTE: A 'for' loop is used by convention to initialize the array. When the value to be used to initialize the array is the same for all the cells, a loop is usually the way to go.

Arrays and Loops

Most of the useful work done with an array requires some sort of searching through the array. While searching through the array, you are accessing every cell in that array. This is done with the same kind of 'for' loop. The following example 'traverses' (runs through) the entire array in order to check if the number 2 appears anywhere within the array.

Example:

```
const int ARR_SIZE = 7;
int my_array[ARR_SIZE] = { 60, 3, 2, 8, 19, 2, 9 };
for (int i = 0; i < ARR_SIZE; i++)
{
        if (my_array[i] == 2)
        {
                cout << "Number 2 appears in index " << i;
        }
}
```

5. Arrays and Functions

Arrays, by default, are not passed to functions in the same way as regular variables are. In C, regular variables are passed by value, meaning that any changes you make to those variables in that function will not persist after you leave the function. Arrays are passed by reference (this isn't technically true, but the effect is the same), meaning that any changes you make to array cells in the function are still in effect when the function exits.

NOTE: (For the more inquisitive of you): The name of the array is actually a pointer to the first cell of the array. When you pass an array to a function, what is passed is actually the address of the first cell. Because of this, access to other cells of the array is freely available from inside the function through the use of the subscript operator or through pointer arithmetic.

What follows is a complete program which declares an array, initializes its cells and increments the value in each cell in a separate function.

Example -

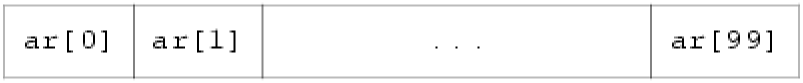
```
void increment_all(int an_array[], int size)
{
        for (int i = 0; i < size; i++)
                an_array[i] += 1;
}
int main()
{
        const int ARR_SIZE = 10;
        int my_arr[ARR_SIZE];
        for (int i = 0; i < ARR_SIZE; i++)
                my_arr[i] = i;
        increment_all(my_arr, ARR_SIZE);
        return 0;
}
```

Like other languages, C uses arrays as a way of describing a collection of variables with identical properties. The group has a single name for all of the members, with the individual members being selected by an index.

Here's an array being declared:

```
double ar[100];
```

The name of the array is ar and its members are accessed as ar[0] through to ar[99] inclusive, as the following figure shows.



ar[0]	ar[1]	. . .	ar[99]

100 element array

Each of the hundred members is a separate variable whose type is double. Without exception, all arrays in C are numbered from 0 up to one less than the bound given in the declaration. This is a prime cause of surprise to beginners—watch out for it. For simple examples of the use of arrays, look back at earlier lessons where several problems are solved with their help.

One important point about array declarations is that they don't permit the use of varying subscripts. The numbers given must be constant expressions which can be evaluated at compile time, not run time. For example, this function incorrectly tries to use its argument in the size of an array declaration:

```
f(int x){
        char var_sized_array[x];        /* FORBIDDEN */
}
```

It's forbidden because the value of x is unknown when the program is compiled; it's a run-time, not a compile-time, value.

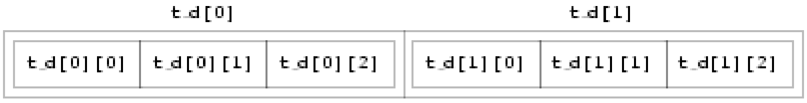
To tell the truth, it would be easy to support arrays whose first dimension is variable, but neither Old C nor the Standard permits it, although we do know of one Very Old C compiler that used to do it.

6. Multi-dimensional Arrays

Multidimensional arrays can be declared like this:

```
int three_dee[5][4][2];
int t_d[2][3]
```

The use of the brackets gives a clue to what is going on. The first declaration gives us a five-element array called three_dee. The members of that array are each a four element array whose members are an array of two ints. We have declared arrays of arrays, as the following figure shows for two dimensions.



Two-dimensional array, showing layout

In the diagram, you will notice that t_d[0] is one element, immediately followed by t_d[1] (there is no break). It so happens that both of these elements are themselves arrays of three integers. Because of C's storage layout rules, t_d[1][0] is immediately after t_d[0][2]. It would be possible (but very poor practice) to access t_d[1][0] by making use of the lack of array-bound checking in C and to use the expression t_d[0][3]. That is not recommended—apart from anything else, if the declaration of t_d ever changes, then the results will be likely to surprise you.

That's all very well, but does it really matter in practice? Not much it's true; but it is interesting to note that in terms of actual machine storage layout the rightmost subscript ‘varies fastest’. This has an impact when arrays are accessed via pointers. Otherwise, they can be used just as would be expected; expressions like these are quite in order:

```
three_dee[1][3][1] = 0;
three_dee[4][3][1] += 2;
```

The second of those is interesting for two reasons. First, it accesses the very last member of the entire array—although the subscripts were declared to be [5][4][2], the highest usable subscript is always one less than the one used in the declaration. Second, it shows where the combined assignment operators are a real blessing. For the experienced C programmer, it is much easier to tell that only one array member is being accessed, and that it is being incremented by two. Other languages would have to express it like this:

```
three_dee[4][3][1] = three_dee[4][3][1] + 2;
```

It takes a conscious effort to check that the same array member is being referenced on both sides of the assignment. It makes things easier for the compiler too: there is only one array indexing calculation to do and this is likely to result in shorter, faster code. (Of course a clever compiler would notice that the left- and right-hand sides look alike and would be able to generate equally efficient code—but not all compilers are clever and there are lots of special cases where even clever compilers are unable to make use of the information.)

It may be of interest to know that although C offers support for multidimensional arrays, they aren't particularly common to see in practice. One-dimensional arrays are present in most programs, if for no other reason than that's what strings are. Two dimensional arrays are seen occasionally and arrays of higher order than that are most uncommon. One of the reasons is that the array is a rather inflexible data structure, and the ease of building and manipulating other types of data structures in C means that they tend to replace arrays in the more advanced programs. We will see more of this when we look at pointers.

7. Summary

Arrays are used for storing multiple elements of the same data type. Elements of array can be accessed by providing a subscript enclosed in square brackets. Arrays can be initialized at the time of declaration or using a for loop. If array is partially initialized then the remaining elements are automatically initialized to 0, in case of numeric arrays. Arrays can be passed as arguments to functions. Multi-dimensional arrays are used for storing matrix and other higher dimensional data.

8. Short Answer Type Questions

1. Define arrays.

1. How array can be initialized at the time of declaration?

2. Define multi-dimensional array.

9. Long Answer Type Questions

1. What are the methods of arrays initialization? Explain giving examples.

1. How arrays are passed as arguments to functions? Explain.
2. What is the purpose of multi-dimensional arrays? Give an example of these.
3. WAP to find the transpose of a matrix.
4. WAP of matrix multiplication using Array's.

10. Suggested Books

1.	Application Programming in C	R. S. Salaria
2.	C Programming using Turbo C	Robert Lafore
3.	Programming with ANSI and Turbo C	Ashok N. Kamthane
4.	Programming using C	E. Balagurusamy
5.	Let Us C	Yashawant Kanetkar

Lesson No. 9

Author: Dr. DharamVeer Sharma
Converted into SLM by: Dr. Vishal Singh

Functions

1. Introduction

The C language is basically a functional programming language, in which the program is divided in small manageable pieces of code called functions, which break large computing tasks into smaller ones and enable people to build on what others have done instead of starting over from scratch. Appropriate functions hide details of operation from parts of the program that don't need to know about them, thus clarifying the whole and easing the pain of making changes.

C has been designed to make functions efficient and easy to use; C programs generally consist of many small functions rather than a few big ones. A program may reside in one or more source files. Source files may be compiled separately and loaded together, along with previously compiled functions from libraries. We will not go into that process here, however, since the details vary from system to system.

2. Objectives

In this lesson, we shall discuss the program structuring tool called function. We shall discuss function prototyping, function definition and function calling methods. The discussion will also include methods of parameter passing and recursive functions.

3. Basics of Functions

A function definition contains a function declaration and the body of a function. To begin with, let us design and write a program to print each line of its input that contains a particular ``pattern'' or string of characters. For example, searching for the pattern of letters ``ould'' in the set of lines :

Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!

will produce the output

Ah Love! could you and I with Fate conspire
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!

The job falls neatly into three pieces:

```
while (there's another line)
    if (the line contains the pattern)
        print it
```

Although it's certainly possible to put the code for all of this in main, a better way is to use the structure to advantage by making each part a separate function. Three small pieces are better to deal with than one big one, because irrelevant details can be buried in the functions and the chance of unwanted interactions is minimized. And the pieces may even be useful in other programs.

``While there's another line'' is getline, a function that we wrote in lesson 1, and ``print it'' is printf, which someone has already provided for us. This means we need only write a routine to decide whether the line contains an occurrence of the pattern.

We can solve that problem by writing a function strindex(s,t) that returns the position or index in the string s where the string t begins, or -1 if s does not contain t. Because C arrays begin at position zero, indexes will be zero or positive, and so a negative value like -1 is convenient for signaling failure. When we later need more sophisticated pattern matching, we only have to replace strindex; the rest of the code can remain the same. (The standard library provides a function strstr that is similar to strindex, except that it returns a pointer instead of an index.)

Given this much design, filling in the details of the program is straightforward. Here is the whole thing, so you can see how the pieces fit together. For now, the pattern to be searched for is a literal string, which is not the most general of mechanisms. We will return shortly to a discussion of how to initialize character arrays and will show how to make the pattern a parameter that is set when the program is run. There is also a slightly different version of getline; you might find it instructive to compare it to the one in lesson 2.

```
#include <stdio.h>
#define MAXLINE 1000 /* maximum input line length */
int getline(char line[], int max)
int strindex(char source[], char searchfor[]);
char pattern[] = "ould";   /* pattern to search for */
/* find all lines matching pattern */
main()
{
        char line[MAXLINE];
        int found = 0;
        while (getline(line, MAXLINE) > 0)
                if (strindex(line, pattern) >= 0) {
                        printf("%s", line);
                        found++;
                }
        return found;
}
/* getline:  get line into s, return length */
int getline(char s[], int lim)
{
        int c, i;
        i = 0;
        while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
                s[i++] = c;
        if (c == '\n')
                s[i++] = c;
        s[i] = '\0';
        return i;
}

/* strindex:  return index of t in s, -1 if none */
int strindex(char s[], char t[])
{
        int i, j, k;
```

```
            for (i = 0; s[i] != '\0'; i++) {
                    for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
                            ;
                    if (k > 0 && t[k] == '\0')
                            return i;
            }
            return -1;
    }
```

Each function definition has the form

```
    return-type function-name(argument declarations)
    {
            declarations and statements
    }
```

Various parts may be absent; a minimal function is

```
    dummy() {}
```

which does nothing and returns nothing. A do-nothing function like this is sometimes useful as a place holder during program development. If the return type is omitted, int is assumed.

A program is just a set of definitions of variables and functions. Communication between the functions is by arguments and values returned by the functions, and through external variables. The functions can occur in any order in the source file and the source program can be split into multiple files, so long as no function is split.

The return statement is the mechanism for returning a value from the called function to its caller. Any expression can follow return:

```
    return expression;
```

The *expression* will be converted to the return type of the function if necessary. Paresthesis are often used around the *expression*, but they are optional.

The calling function is free to ignore the returned value. Furthermore, there need to be no expression after return; in that case, no value is returned to the caller. Control also returns to the caller with no value when execution ``falls off the end'' of the function by reaching the closing right brace. It is not illegal, but probably a sign of trouble, if a function returns a value from one place and no value from another. In any case, if a function fails to return a value, its ``value'' is certain to be garbage.

The pattern-searching program returns a status from main, the number of matches found. This value is available for use by the environment that called the program

The mechanics of how to compile and load a C program that resides on multiple source files vary from one system to the other. Suppose that the three functions are stored in three files called main.c, getline.c and strindex.c. Then the command

```
cc main.c getline.c strindex.c
```

compiles the three files, placing the resulting object code in files main.o, getline.o, and strindex.o, then loads them all into an executable file called a.out. If there is an error, say in main.c, the file can be recompiled by itself and the result loaded with the previous object files, with the command

```
cc main.c getline.o strindex.o
```

The cc command uses the ``.c'' versus ``.o'' naming convention to distinguish source files from object files.

4. Functions Returning Non-integers

So far our examples of functions have returned either no value (void) or an int. What if a function must return some other type? many numerical functions like sqrt, sin, and cos return double; other specialized functions return other types. To illustrate how to deal with this, let us write and use the function atof(s), which converts the string s to its double-precision floating-point equivalent. atof if an extension of atoi, which we showed versions of in lessons 2 and 3. It handles an optional sign and decimal point and the presence or absence of either part or fractional part. Our version is *not* a high-quality input conversion routine; that would take more space than we care to use. The standard library includes an atof; the header <stdlib.h> declares it.

First, atof itself must declare the type of value it returns, since it is not int. The type name precedes the function name:

```
#include <ctype.h>
/* atof:  convert string s to double */
double atof(char s[])
{
        double val, power;
        int i, sign;
        for (i = 0; isspace(s[i]); i++)  /* skip white space */
                ;
        sign = (s[i] == '-') ? -1 : 1;
        if (s[i] == '+' || s[i] == '-')
                i++;
        for (val = 0.0; isdigit(s[i]); i++)
                val = 10.0 * val + (s[i] - '0');
        if (s[i] == '.')
```

```
                i++;
        for (power = 1.0; isdigit(s[i]); i++) {
                val = 10.0 * val + (s[i] - '0');
                power *= 10;
        }
        return sign * val / power;
}
```

Second, and just as important, the calling routine must know that atof returns a non-int value. One way to ensure this is to declare atof explicitly in the calling routine. The declaration is shown in this primitive calculator (barely adequate for check-book balancing), which reads one number per line, optionally preceded with a sign, and adds them up, printing the running sum after each input:

```
#include <stdio.h>
#define MAXLINE 100
/* rudimentary calculator */
main()
{
        double sum, atof(char []);
        char line[MAXLINE];
        int getline(char line[], int max);
        sum = 0;
        while (getline(line, MAXLINE) > 0)
                printf("\t%g\n", sum += atof(line));
        return 0;
}
```

The declaration

```
double sum, atof(char []);
```

says that sum is a double variable and that atof is a function that takes one char[] argument and returns a double.

The function atof must be declared and defined consistently. If atof itself and the call to it in main have inconsistent types in the same source file, the error will be detected by the compiler. But if (as is more likely) atof were compiled separately, the mismatch would not be detected, atof would return a double that main would treat as an int and meaningless answers would result.

In the light of what we have said about how declarations must match definitions, this might seem surprising. The reason a mismatch can happen is that if there is no function prototype, a function is implicitly declared by its first appearance in an expression, such as

```
sum += atof(line)
```

If a name that has not been previously declared occurs in an expression and is followed by a left parentheses, it is declared by context to be a function name, the function is assumed to return an int and nothing is assumed about its arguments. Furthermore, if a function declaration does not include arguments, as in

```
double atof();
```

that too is taken to mean that nothing is to be assumed about the arguments of atof; all parameter checking is turned off. This special meaning of the empty argument list is intended to permit older C programs to compile with new compilers. But it's a bad idea to use it with new C programs. If the function takes arguments, declare them; if it takes no arguments, use void.

Given atof, properly declared, we could write atoi (convert a string to int) in terms of it:

```
/* atoi:  convert string s to integer using atof */
int atoi(char s[])
{
        double atof(char s[]);
        return (int) atof(s);
}
```

Notice the structure of the declarations and the return statement. The value of the expression in

return *expression*;

is converted to the type of the function before the return is taken. Therefore, the value of atof, a double, is converted automatically to int when it appears in this return, since the function atoi returns an int. This operation does potentionally discard information, however, some compilers warn of it. The cast states explicitly that the operation is intended and suppresses any warning.

5. External Variables

A C program consists of a set of external objects, which are either variables or functions. The adjective ``external'' is used in contrast to ``internal'', which describes the arguments and variables defined inside functions. External variables are defined outside of any function and are thus potentionally available to many functions. Functions themselves are always external, because C does not allow functions to be defined inside other functions. By default, external variables and functions have the property that all references to them by the same name, even from functions compiled separately are references to the same thing. (The standard calls this property *external linkage*.) In this sense, external variables are analogous to Fortran COMMON blocks or variables in the outermost block in Pascal. We will see later how to define external variables and functions that are visible only within a single source file. Because external variables are globally accessible, they provide an alternative to function arguments and return values for communicating data between functions. Any function may access an external variable by referring to it by name, if the name has been declared somehow.

If a large number of variables must be shared among functions, external variables are more convenient and efficient than long argument lists. As pointed out in lesson 1, however, this reasoning should be applied with some caution, for it can have a bad effect on program structure and lead to programs with too many data connections between functions.

External variables are also useful because of their greater scope and lifetime. Automatic variables are internal to a function; they come into existence when the function is entered and disappear when it is left. External variables, on the other hand, are permanent, so they can retain values from one function invocation to the next. Thus if two functions must share some data, yet neither calls the other, it is often most convenient if the shared data is kept in external variables rather than being passed in and out via arguments.

6. Formal and Actual Arguments

The values passed to the function at the time of call are called actual parameters. The values of the actual parameters are received by the variable declared at the time of the function declaration. These receiving variable are called **formal parameters**.

Example:

```
#include <stdio.h>

unsigned long GetSquare(unsigned long x)

{

        return x * x;

}

void main()

{

        int a = 10;

        printf("Square of value %d is %ld", GetSquare(a));

}
```

In the above example, while calling the function GetSquare(a), the parameter passed is an actual parameter, while when this values is received by the function then it is stored in formal parameter x.

7. Scope Rules

The functions and external variables that make up a C program need not all be compiled at the same time; the source text of the program may be kept in several files, and previously compiled routines may be loaded from libraries. Among the questions of interest are

- How are declarations written so that variables are properly declared during compilation?
- How are declarations arranged so that all the pieces will be properly connected when the program is loaded?
- How are declarations organized so there is only one copy?
- How are external variables initialized?

Let us discuss these topics by reorganizing the calculator program into several files. As a practical matter, the calculator is too small to be worth splitting, but it is a fine illustration of the issues that arise in larger programs.

The *scope* of a name is the part of the program within which the name can be used. For an automatic variable declared at the beginning of a function, the scope is the function in which the name is declared. Local variables of the same name in different functions are unrelated. The same is true of the parameters of the function, which are in effect local variables.

The scope of an external variable or a function lasts from the point at which it is declared to the end of the file being compiled. For example, if main, sp, val, push, and pop are defined in one file, in the order shown above, that is,

```
main() { ... }

int sp = 0;

double val[MAXVAL];

void push(double f) { ... }

double pop(void) { ... }
```

then the variables sp and val may be used in push and pop simply by naming them; no further declarations are needed. But these names are not visible in main, nor are push and pop themselves.

On the other hand, if an external variable is to be referred to before it is defined, or if it is defined in a different source file from the one where it is being used then an extern declaration is mandatory.

It is important to distinguish between the *declaration* of an external variable and its *definition.* A declaration announces the properties of a variable (primarily its type); a definition also causes storage to be set aside. If the lines

```
int sp;

double val[MAXVAL];
```

appear outside of any function, they *define* the external variables sp and val, cause storage to be set aside and also serve as the declarations for the rest of that source file. On the other hand, the lines

```
extern int sp;

extern double val[];
```

declare for the rest of the source file that sp is an int and that val is a double array (whose size is determined elsewhere), but they do not create the variables or reserve storage for them.

There must be only one *definition* of an external variable among all the files that make up the source program; other files may contain extern declarations to access it. (There may also be extern declarations in the file containing the definition.) Array sizes must be specified with the definition, but are optional with an extern declaration. Initialization of an external variable goes only with the definition.

Although it is not a likely organization for this program, the functions push and pop could be defined in one file and the variables val and sp defined and initialized in another. Then these definitions and declarations would be necessary to tie them together:

in file1:

```
extern int sp;

extern double val[];

void push(double f) { ... }

double pop(void) { ... }
```

in file2:

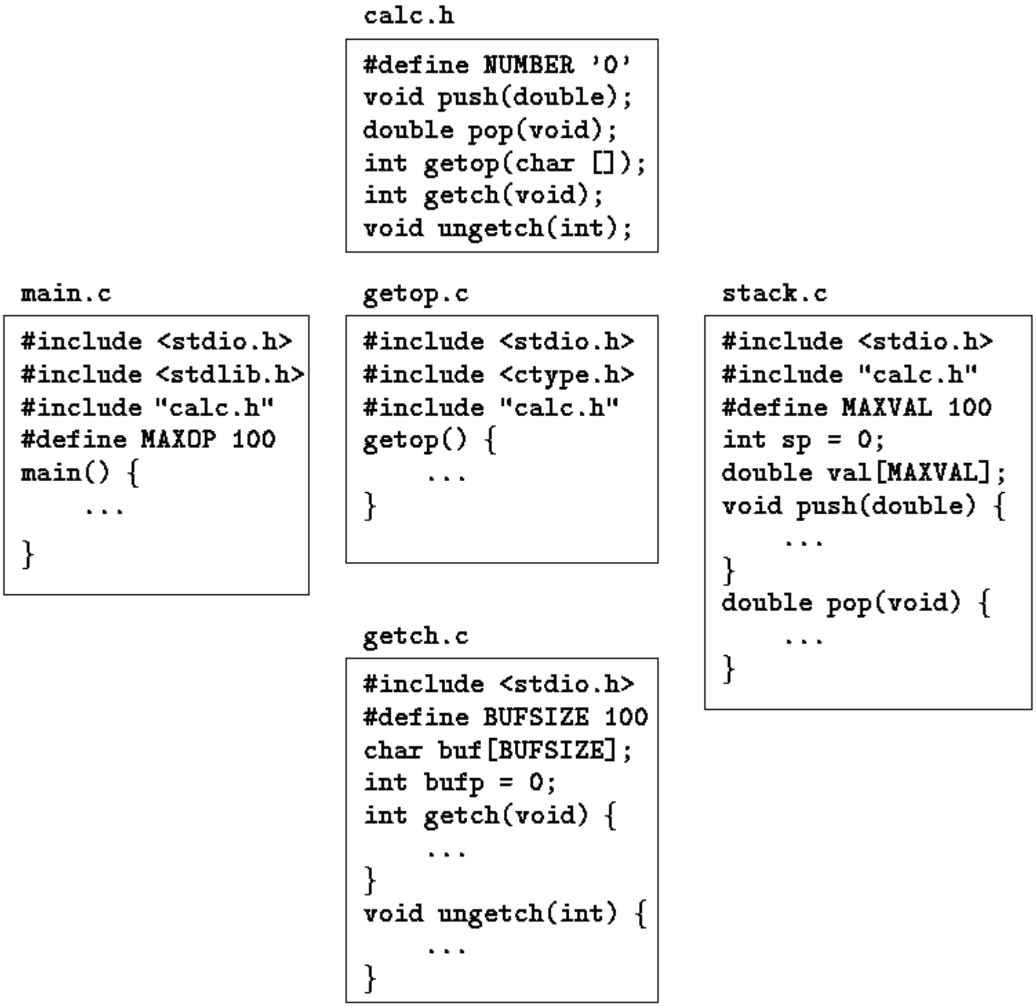
```
int sp = 0;

double val[MAXVAL];
```

Because the extern declarations in *file1* lie ahead of and outside the function definitions, they apply to all functions; one set of declarations suffices for all of *file1*. This same organization would also bee needed if the definition of sp and val followed their use in one file.

8. Header Files

Let us now consider dividing the calculator program into several source files, as it might be if each of the components were substantially bigger. The main function would go in one file, which we will call main.c; push, pop and their variables go into a second file, stack.c; getop goes into a third, getop.c. Finally, getch and ungetch go into a fourth file, getch.c; we separate them from the others because they would come from a separately-compiled library in a realistic program.

There is one more thing to worry about - the definitions and declarations shared among files. As much as possible, we want to centralize this, so that there is only one copy to get and keep right as the program evolves. Accordingly, we will place this common material in a *header file*, calc.h, which will be included as necessary. (The #include line is described in) The resulting program then looks like this:



calc.h

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
    ...
}
```

getop.c

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop() {
    ...
}
```

stack.c

```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
    ...
}
double pop(void) {
    ...
}
```

getch.c

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
    ...
}
void ungetch(int) {
    ...
}
```

There is a tradeoff between the desire that each file have access only to the information it needs for its job and the practical reality that it is harder to maintain more header files. Up to some moderate program size, it is probably best to have one header file that contains everything that is to be shared between any two parts of the program; that is the decision we made here. For a much larger program, more organization and more headers would be needed.

9. Recursion

Calling a function by itself is known as recursion. Any function can call any function including itself. C functions may be used recursively; that is, a function may call itself either directly or indirectly. Consider printing a number as a character string. As we mentioned before, the digits are generated in the wrong order: low-order digits are available before high-order digits, but they have to be printed the other way around.

There are two solutions to this problem. On is to store the digits in an array as they are generated, then print them in the reverse order, as we did with itoa in lesson 7. The alternative is a recursive solution, in which printd first calls itself to cope with any leading digits, then prints the trailing digit. Again, this version can fail on the largest negative number.

```
#include <stdio.h>
/* printd:  print n in decimal */
void printd(int n)
{
        if (n < 0) {
                putchar('-');
                n = -n;
        }
        if (n / 10)
                printd(n / 10);
        putchar(n % 10 + '0');
}
```

When a function calls itself recursively, each invocation gets a fresh set of all the automatic variables, independent of the previous set. Thus in printd(123) the first printd receives the argument n = 123. It passes 12 to a second printd, which in turn passes 1 to a third. The third-level printd prints 1, then returns to the second level. That printd prints 2, then returns to the first level. That one prints 3 and terminates.

Recursion may provide no saving in storage, since somewhere a stack of the values being processed must be maintained. Nor will it be faster. But recursive code is more compact, and often much easier to write and understand than the non-recursive equivalent. Recursion is the natural solution for some of the problems like factorial computation, Fibonacci series generation, quick sort, binary search etc.

While using recursion, care should be taken that recursive function always return some value or recursion has some terminating point.

10. Summary

Functions decompose a large program into manageable smaller units, which can be tested independently. Functions also facilitate breaking the program into logical units. While calling the functions, any parameter passed is the actual parameter and the variable receiving the value is called formal parameter. Recursive functions are those which call themselves directly or indirectly. Recursive functions should have a terminating point.

11. Short Answer Type Questions

1. What is the syntax of declaring a function?
2. What is the difference between function declaration and function definition?
3. What is the meaning of function prototyping?
4. Define recursion.

12. Long Answer Type Questions

1. What is the advantage of decomposing program into functions?
2. What type of values can be returned by functions?
3. What is the difference between formal and actual parameters?

13. Suggested Books

1.	Application Programming in C	R. S. Salaria
2.	C Programming using Turbo C	Robert Lafore
3.	Programming with ANSI and Turbo C	Ashok N. Kamthane
4.	Programming using C	E. Balagurusamy
5.	Let Us C	Yashwant Kantekar

BA PART-2 (SEM-3) **PAPER : BAP-201**
C Programming and Data Structures

Lesson No. 10 **Author: Dr. DharamVeer Sharma**
Converted into SLM by: Dr. Vishal Singh

STRINGS

1. Introduction

The character string is one of the most useful and important data types in C. You have been using character strings all along, but there still is much to learn about them. The C library provides a wide range of functions for reading and witting strings, copying strings, comparing strings, combining strings, searching strings, and more. This lesson will add these capabilities to your programming skills.

2. Objectives

You will learn about the following in this chapter:

- Functions: gets(), puts(), strcat(), strncat(), strcmp(), strncmp(), strcpy(), strncpy(), sprintf(), strchr()
- Creating and using strings
- Using several string and character functions from the C library and creating your own string functions
- Using command-line arguments

3. Representing Strings and String I/O

Of course, you already know the most basic fact: A character string is a char array terminated with a null character ('\O'). Therefore, what you've learned about arrays and pointers carries over to character strings. But because character strings are so commonly used, C provides many functions specifically designed to work with strings. This lesson discusses the nature of strings, how to declare and initialize strings, how to get them into and out of programs and how to manipulate strings.

The following listing presents a busy program that illustrates several ways to set up, read, and print strings. It uses two new functions—gets(), which reads a string and puts(), which prints a string. (You probably notice a family resemblance to getchar() and putchar().) The rest of the program should look fairly familiar.

The strings.c Program

```
// strings.c -- stringing the user along
#include <stdio.h>
#define MSG "You must have many talents. Tell me some."
                        // a symbolic string constant
#define LIM  5
#define LINELEN 81      // maximum string length + 1
int main(void)
{
      char name[LINELEN];
      char talents[LINELEN];
      int i;
      // initializing a one dimensioned char array
      const char m1[40] = "Limit yourself to one line's worth.";
                        // letting the compiler compute the
                        // array size
      const char m2[] = "If you can't think of anything, fake it.";
                        // initializing a pointer
      const char *m3 = "\nEnough about me - what's your name?";
                        // initializing an array of
                        // string pointers
      const char *mytal[LIM] = {  // array of 5 pointers
```

```
        "Adding numbers swiftly",
        "Multiplying accurately", "Stashing data",
        "Following instructions to the letter",
        "Understanding the C language"
    };
    printf("Hi! I'm Clyde the Computer."
        " I have many talents.\n");
    printf("Let me tell you some of them.\n");
    puts("What were they? Ah, yes, here's a partial list.");
    for (i = 0; i < LIM; i++)
        puts(mytal[i]);   // print list of computer talents
    puts(m3);
    gets(name);
    printf("Well, %s, %s\n", name, MSG);
    printf("%s\n%s\n", m1, m2);
    gets(talents);
    puts("Let's see if I've got that list:");
    puts(talents);
    printf("Thanks for the information, %s.\n", name);
    return 0;
}
```

To show you what this program does, here is a sample run:

```
Hi! I'm Clyde the Computer. I have many talents.
Let me tell you some of them.
What were they? Ah, yes, here's a partial list.
Adding numbers swiftly
Multiplying accurately
Stashing data
Following instructions to the letter
Understanding the C language
```

```
Enough about me – what's your name?
Nigel Barntwit
Well, Nigel Barntwit, You must have many talents. Tell me some.
Just limit yourself to one line's worth.
If you can't think of anything, fake it.
Fencing, yodeling, malingering, cheese tasting, and sighing.
Let's see if I've got that list:
Fencing, yodeling, malingering, cheese tasting, and sighing.
Thanks for the information, Nigel Barntwit.
```

Rather than going through the listing line-by-line, let's take a more encompassing approach. First, you will look at ways of defining a string within a program. Then you will see what is involved in reading a string into a program. Finally, you will study ways to output a string.

4. Defining Strings within a Program

As you probably noticed when you read the above listing, there are many ways to define a string. The principal ways are using string constants, using char arrays, using char pointers and using arrays of character strings. A program should make sure there is a place to store a string and we will cover that topic, too.

Character String Constants (String Literals)

A string constant, also termed a string literal, is anything enclosed in double quotation marks. The enclosed characters, plus a terminating '\O' character automatically provided by the compiler, are stored in memory as a character string. The program uses several such character string constants, most often as arguments for the printf() and puts() functions. Note, too, that you can use #define to define character string constants.

Recall that ANSI C concatenates string literals if they are separated by nothing or by whitespace. For example,

```
char greeting[50] = "Hello, and"" how are"  " you"
                               " today!";
```

is equivalent to this:

```
char greeting[50] = "Hello, and how are you today!";
```

If you want to use a double quotation mark within a string, precede the quotation mark with a backslash, as follows:

```
printf("\"Run, Spot, run!\" exclaimed Dick.\n");
```

This produces the following output:

```
"Run, Spot, run!" exclaimed Dick.
```

Character string constants are placed in the static storage class, which means that if you use a string constant in a function, the string is stored just once and lasts for the duration of the program, even if the function is called several times. The entire quoted phrase acts as a pointer to where the string is stored. This action is analogous to the name of an array acting as a pointer to the array's location. If this is true, what kind of output should the program in the following listing produce?

The quotes.c Program

```
/* quotes.c -- strings as pointers */

#include <stdio.h>

int main(void)

{

        printf("%s, %p, %c\n", "We", "are", *"space farers");

        return 0;

}
```

The %s format should print the string We. The %p format produces an address. So if the phrase "are" is an address, then %p should print the address of the first character in the string. (Pre-ANSI implementations might have to use %u or %lu instead of %p.) Finally, *"space farers" should produce the value of the address pointed to, which should be the first character of the string "space farers". Does this really happen? Well, here is the output:

```
We, 0x0040c010, s
```

Character String Arrays and Initialization

When you define a character string array, you must let the compiler know how much space is needed. One way is to specify an array size large enough to hold the string. The following declaration initializes the array m1 to the characters of the indicated string:

```
const char m1[40] = "Limit yourself to one line's worth.";
```

The const indicates the intent to not alter this string.

This form of initialization is short for the standard array initialization form:

```
const char m1[40] = {  'L', 'i', 'm', 'i', 't', ' ', 'y', 'o', 'u', 'r', 's', 'e', 'l',
                        'f', ' ', 't', 'o', ' ', 'o', 'n', 'e', ' ',
                    'l', 'i', 'n', 'e', '\'', 's', ' ', 'w', 'o', 'r',
                    't', 'h', '.', '\0'
        };
```

Note the closing null character. Without it, you have a character array, but not a string.

When you specify the array size, be sure that the number of elements is at least one more (that null character again) than the string length. Any unused elements are automatically initialized to 0 (which in char form is the null character, not the zero digit character).

Often, it is convenient to let the compiler determine the array size; recall that if you omit the size in an initializing declaration, the compiler determines the size for you:

```
        const char m2[] = "If you can't think of anything, fake it.";
```

Initializing character arrays is one case when it really does make sense to let the compiler determine the array size. That's because string-processing functions typically don't need to know the size of the array because they can simply look for the null character to mark the end.

Note that the program had to assign a size explicitly for the array name:

```
        #define LINELEN 81        // maximum string length + 1
        ...
        char name[LINELEN];
```

Because the contents for name are to be read when the program runs, the compiler has no way of knowing in advance how much space to set aside unless you tell it. There is no string constant present whose characters the compiler can count, so we gambled that 80 characters would be enough to hold the user's name. When you declare an array, the array size must evaluate to an integer constant. You can't use a variable that gets set at runtime. The array size is locked into the program at compile time. (Actually, with C99 you could use a variable-length array, but you still have no way of knowing in advance how big it has to be.)

```
        int n = 8;
        char cakes[2 + 5];  /* valid, size is a constant expression
        char crumbs[n];     /* invalid prior to C99, a VLA after C99
```

The name of a character array, like any array name, yields the address of the first element of the array. Therefore, the following holds for the array m1:

```
        m1 == &m1[0] , *m1 == 'L', and *(m1+1) == m1[1] == 'i'
```

Indeed, you can use pointer notation to set up a string. For example, the following declaration uses the pointer:

```
const char *m3 = "\nEnough about me -- what's your name?";
```

This declaration is very nearly the same as this one:

```
char m3[] = "\nEnough about me -- what's your name?"
```

Both declarations amount to saying that m3 is a pointer to the indicated string. In both cases, the quoted string itself determines the amount of storage set aside for the string. Nonetheless, the forms are not identical.

Array Versus Pointer

What is the difference, then, between an array and a pointer form? The array form (m3[]) causes an array of 38 elements (one for each character plus one for the terminating '\0') to be allocated in the computer memory. Each element is initialized to the corresponding character. Typically, what happens is that the quoted string is stored in a data segment that is part of the executable file; when the program is loaded into memory, so is that string. The quoted string is said to be in static memory. But the memory for the array is allocated only after the program begins running. At that time, the quoted string is copied into the array. Hereafter, the compiler will recognize the name m3 as a synonym for the address of the first array element, &m3[0]. One important point here is that in the array form, m3 is an address constant. You can't change m3, because that would mean changing the location (address) where the array is stored. You can use operations such as m3+1 to identify the next element in an array, but ++m3 is not allowed. The increment operator can be used only with the names of variables, not with constants.

The pointer form (*m3) also causes 38 elements in static storage to be set aside for the string. In addition, once the program begins execution, it sets aside one more storage location for the pointer variable m3 and stores the address of the string in the pointer variable. This variable initially points to the first character of the string, but the value can be changed. Therefore, you can use the increment operator. For instance, ++m3 would point to the second character (E).

In short, initializing the array copies a string from static storage to the array, whereas initializing the pointer merely copies the address of the string.

Are these differences important? Often they are not, but it depends on what you try to do. See the following discussion for some examples.

Array and Pointer Differences

Let's examine the differences between initializing a character array to hold a string and initializing a pointer to point to a string. (By "pointing to a string," we really mean pointing to the first character of a string.) For example, consider these two declarations:

```
char heart[] = "I love Tillie!";

char *head = "I love Millie!";
```

The chief difference is that the array name heart is a constant, but the pointer head is a variable. What practical difference does this make?

First, both can use array notation:

```
for (i = 0; i < 6; i++)

        putchar(heart[i]);

putchar('\n');

for (i = 0; i < 6; i++)

        putchar(head[i]));

putchar('\n');
```

This is the output:

```
I love

I love
```

Next, both can use pointer addition:

```
for (i = 0; i < 6; i++)

        putchar(*(heart + i));

putchar('\n');

for (i = 0; i < 6; i++)

        putchar(*(head + i));

putchar('\n');
```

Again, the output is as follows:

```
I love

I love
```

Only the pointer version, however, can use the increment operator:

```
while (*(head) != '\0')  /* stop at end of string            */

putchar(*(head++));  /* print character, advance pointer */
```

This produces the following output:

```
I love Millie!
```

Suppose you want head to agree with heart. You can say

```
head = heart;  /* head now points to the array heart */
```

This makes the head pointer point to the first element of the heart array.

However, you cannot say

```
heart = head;  /* illegal construction */
```

The situation is analogous to x = 3; versus 3 = x;. The left side of the assignment statement must be a variable or, more generally, an lvalue, such as *p_int. Incidentally, head = heart; does not make the Millie string vanish; it just changes the address stored in head. Unless you've saved the address of "I love Millie!" elsewhere, however, you won't be able to access that string when head points to another location.

There is a way to alter the heart message—go to the individual array elements:

```
heart[7]= 'M';  or  *(heart + 7) = 'M';
```

The elements of an array are variables (unless the array was declared as const), but the name is not a variable.

Let's go back to a pointer initialization:

```
char * word = "frame";
```

Can you use the pointer to change this string?

```
word[1] = 'l';  // allowed??
```

Your compiler probably will allow this, but, under the current C standard, the behavior for such an action is undefined. Such a statement could, for example, lead to memory access errors. The reason is that a compiler can choose to represent all identical string literals with a single copy in memory. For example, the following statements could all refer to a single memory location of string "Klingon":

```
char * p1 = "Klingon";

p1[0] = 'F';    // ok?

printf("Klingon");

printf(": Beware the %ss!\n", "Klingon");
```

That is, the compiler can replace each instance of "Klingon" with the same address. If the compiler uses this single-copy representation and allows changing p1[0] to 'F', that would affect all uses of the string, so statements printing the string literal "Klingon" would actually display "Flingon":

```
Flingon: Beware the Flingons!
```

In fact, several compilers do behave this rather confusing way, whereas others produce programs that abort. Therefore, the recommended practice for initializing a pointer to a string literal is to use the const modifier:

```
const char * pl = "Klingon";  // recommended usage
```

Initializing a non-const array with a string literal, however, poses no such problems, because the array gets a copy of the original string.

5. Inbuilt String Functions

The C library supplies several string-handling functions; ANSI C uses the string.h header file to provide the prototypes. We'll look at some of the most useful and common ones: strlen(), strcat(), strncat(), strcmp(), strncmp(), strcpy() and strncpy(). We'll also examine sprintf(), supported by the stdio.h header file. For a complete list of the string.h family of functions.

The strlen() Function

The strlen() function, as you already know, finds the length of a string. It's used in the next example, a function that shortens lengthy strings:

```
/* fit.c — truncation function */
void fit(char * string, unsigned int size)
{
        if (strlen(string) > size)
                *(string + size) = '\0';
}
```

This function does change the string, so the function header doesn't use const in declaring the formal parameter string.

The ANSI string.h file contains function prototypes for the C family of string functions, which is why this example includes it.

The strcat() Function

The strcat() (for string concatenation) function takes two strings for arguments. A copy of the second string is tacked onto the end of the first and this combined version becomes the new first string. The second string is not altered. Function strcat() is type char * (that is, a pointer-to-char). It returns the value of its first argument—the address of the first character of the string to which the second string is appended.

```
/* str_cat.c -- joins two strings */
#include <stdio.h>
#include <string.h>  /* declares the strcat() function */
#define SIZE 80
int main(void)
{
        char flower[SIZE];
        char addon[] = "s smell like old shoes.";
        puts("What is your favorite flower?");
        gets(flower);
        strcat(flower, addon);
        puts(flower);
        puts(addon);
        return 0;
}
```

This is the output:

```
What is your favorite flower?
Rose
Roses smell like old shoes.
s smell like old shoes.
```

The strncat() Function

The strcat() function does not check to see whether the second string will fit in the first array. If you fail to allocate enough space for the first array, you will run into problems as excess characters overflow into adjacent memory locations. Of course, you can use strlen() to look before you leap. Note that it adds 1 to the combined lengths to allow space for the null character. Alternatively, you can use strncat(), which takes a second argument indicating the maximum number of characters to add. For example, strncat(bugs, addon, 13) will add the contents of the addon string to bugs, stopping when it reaches 13 additional characters or the null character, whichever comes first. Therefore, counting the null character (which is appended in either case), the bugs array should be large enough to hold the original string (not counting the null character), a maximum of 13 additional characters and the terminal null character. The following listing uses this information to calculate a value for the available variable, which is used as the maximum number of additional characters allowed.

The join_chk.c Program

```
/* join_chk.c -- joins two strings, check size first */
#include <stdio.h>
#include <string.h>
#define SIZE 30
#define BUGSIZE 13
int main(void)
{
        char flower[SIZE];
        char addon[] = "s smell like old shoes.";
        char bug[BUGSIZE];
        int available;
        puts("What is your favorite flower?");
        gets(flower);
        if ((strlen(addon) + strlen(flower) + 1) <= SIZE)
                strcat(flower, addon);
        puts(flower);
        puts("What is your favorite bug?");
        gets(bug);
        available = BUGSIZE - strlen(bug) - 1;
        strncat(bug, addon, available);
        puts(bug);
        return 0;
}
```

Here is a sample run:

```
What is your favorite flower?
Rose
```

```
Roses smell like old shoes.
What is your favorite bug?
Aphid
Aphids smell
```

The strcmp() Function

Suppose you want to compare someone's response to a stored string, as shown in the following listing.

The nogo.c Program

```
/* nogo.c -- will this work? */
#include <stdio.h>
#define ANSWER "Grant"
int main(void)
{
        char try[40];
        puts("Who is buried in Grant's tomb?");
        gets(try);
        while (try != ANSWER)
        {
                puts("No, that's wrong. Try again.");
                gets(try);
        }
        puts("That's right!");
        return 0;
}
```

As nice as this program might look, it will not work correctly. ANSWER and try really are pointers, so the comparison try != ANSWER doesn't check to see whether the two strings are the same. Rather, it checks to see whether the two strings have the same address. Because ANSWER and try are stored in different locations, the two addresses are never the same and the user is forever told that he or she is wrong. Such programs tend to discourage people.

What you need is a function that compares string contents, not string addresses. You could devise one, but the job has been done for you with strcmp() (for string comparison). This function does for strings what relational operators do for numbers. In particular, it returns 0 if its two string arguments are the same. The revised program is shown in the following listing.

The compare.c Program

```
/* compare.c -- this will work */
#include <stdio.h>
#include <string.h>   /* declares strcmp() */
#define ANSWER "Grant"
#define MAX 40
int main(void)
{
        char try[MAX];
        puts("Who is buried in Grant's tomb?");
        gets(try);
        while (strcmp(try,ANSWER) != 0)
        {
                puts("No, that's wrong. Try again.");
                gets(try);
        }
        puts("That's right!");
        return 0;
}
```

One of the nice features of strcmp() is that it compares strings, not arrays. Although the array try occupies 40 memory cells and "Grant" only six (one for the null character), the comparison looks only at the part of try up to its first null character. Therefore, strcmp() can be used to compare strings stored in arrays of different sizes.

What if the user answers "GRANT" or "grant" or "Ulysses S. Grant"? The user is told that he or she is wrong. To make a friendlier program, you have to anticipate all possible correct answers. There are some tricks you can use. For example, you can use #define to define the answer as "GRANT" and write a function that converts all input to uppercase. That eliminates the problem of capitalization, but you still have the other forms to worry about. We leave that as an exercise for you.

The strcmp() Return Value

What value does strcmp() return if the strings are not the same? The following listing shows an example.

The compback.c Program

```
/* compback.c -- strcmp returns */
#include <stdio.h>
#include <string.h>
int main(void)
{
        printf("strcmp(\"A\", \"A\") is ");
        printf("%d\n", strcmp("A", "A"));
        printf("strcmp(\"A\", \"B\") is ");
        printf("%d\n", strcmp("A", "B"));
        printf("strcmp(\"B\", \"A\") is ");
        printf("%d\n", strcmp("B", "A"));
        printf("strcmp(\"C\", \"A\") is ");
        printf("%d\n", strcmp("C", "A"));
        printf("strcmp(\"Z\", \"a\") is ");
        printf("%d\n", strcmp("Z", "a"));
        printf("strcmp(\"apples\", \"apple\") is ");
        printf("%d\n", strcmp("apples", "apple"));
        return 0;
}
```

Here is the output on one system:

```
strcmp("A", "A") is 0
strcmp("A", "B") is -1
strcmp("B", "A") is 1
strcmp("C", "A") is 1
strcmp("Z", "a") is -1
strcmp("apples", "apple") is 1
```

Comparing "A" to itself returns 0. Comparing "A" to "B" returns -1 and reversing the comparison returns 1. These results suggest that strcmp() returns a negative number if the first string precedes the second alphabetically and that it returns a positive number if the order is the other way. Therefore, comparing "C" to "A" gives a 1. Other systems might return 2—the difference in ASCII code values. The ANSI standard says that strcmp() returns a negative number if the first string comes before the second alphabetically, returns 0 if they are the same and returns a positive number if the first string follows the second alphabetically. The exact numerical values, however, are left open to the implementation. Here, for example, is the output for another implementation, one that returns the difference between the character codes:

```
strcmp("A", "A") is 0
strcmp("A", "B") is -1
strcmp("B", "A") is 1
strcmp("C", "A") is 2
strcmp("Z", "a") is -7
strcmp("apples", "apple") is 105
```

What if the initial characters are identical? In general, strcmp() moves along until it finds the first pair of disagreeing characters. It then returns the corresponding code. For instance, in the very last example, "apples" and "apple" agree until the final s of the first string. This matches up with the sixth character in "apple", which is the null character, ASCII 0. Because the null character is the very first character in the ASCII sequence, s comes after it and the function returns a positive value.

The last comparison points out that strcmp() compares all characters, not just letters, so instead of saying the comparison is alphabetic, we should say that strcmp() goes by the machine collating sequence. That means characters are compared according to their numeric representation, typically the ASCII values. In ASCII, the codes for uppercase letters precede those for lowercase letters. Therefore, strcmp("Z", "a") is negative.

Most often, you won't care about the exact value returned. You just want to know if it is zero or nonzero—that is, whether there is a match or not—or you might be trying to sort the strings alphabetically, in which case you want to know if the comparison is positive, negative or zero.

Incidentally, sometimes it is more convenient to terminate input by entering an empty line—that is, by pressing the Enter key or Return key without entering anything else. To do so, you can modify the while loop control statement so that it looks like this:

The strncmp() Variation

The strcmp() function compares strings until it finds corresponding characters that differ, which could take the search to the end of one of the strings. The strncmp() function compares the strings until they differ or until it has compared a number of characters specified by a third argument. For example, if you wanted to search for strings that begin with "astro", you could limit the search to the first five characters. The following listing shows how.

The starsrch.c Program

```
/* starsrch.c -- use strncmp() */
#include <stdio.h>
#include <string.h>
#define LISTSIZE 5
int main()
{
    const char * list[LISTSIZE] =
    {
        "astronomy", "astounding",
        "astrophysics", "ostracize",
        "asterism"
    };
    int count = 0;
    int i;
    for (i = 0; i < LISTSIZE; i++)
        if (strncmp(list[i],"astro", 5) == 0)
        {
            printf("Found: %s\n", list[i]);
            count++;
        }
    printf("The list contained %d words beginning with astro\n",
    count);
    return 0;
}
```

Here is the output:

```
Found: astronomy
Found: astrophysics
The list contained 2 words beginning with astro.
```

The strcpy() and strncpy() Functions

We've said that if pts1 and pts2 are both pointers to strings, the expression

```
pts2 = pts1;
```

copies only the address of a string, not the string itself. Suppose, though, that you do want to copy a string. Then you can use the strcpy() function. The following listing asks the user to enter words beginning with q. The program copies the input into a temporary array and if the first letter is a q, the program uses strcpy() to copy the string from the temporary array to a permanent destination. The strcpy() function is the string equivalent of the assignment operator.

The copy1.c Program

```
/* copy1.c -- strcpy() demo */
#include <stdio.h>
#include <string.h>  /* declares strcpy() */
#define SIZE 40
#define LIM 5
int main(void)
{
        char qwords[LIM][SIZE];
        char temp[SIZE];
        int i = 0;
        printf("Enter %d words beginning with q:\n", LIM);
        while (i < LIM && gets(temp))
        {
                if (temp[0] != 'q')
                        printf("%s doesn't begin with q!\n", temp);
                else
```

```
            {
                strcpy(qwords[i], temp);
                i++;
            }
        }
        puts("Here are the words accepted:");
        for (i = 0; i < LIM; i++)
            puts(qwords[i]);
        return 0;
    }
```

Here is a sample run:

```
    Enter 5 words beginning with q:
    quackery
    quasar
    quilt
    quotient
    no more
    no more doesn't begin with q!
    quiz
Here are the words accepted:
    quackery
    quasar
    quilt
    quotient
    quiz
```

Note that the counter i is incremented only when the word entered passes the q test. Also note that the program uses a character-based test:

```
    if (temp[0] != 'q')
```

That is, is the first character in the temp array not a q? Another possibility is using a string-based test:

```
if (strncmp(temp, "q", 1) != 0)
```

That is, are the strings temp and "q" different from each other in the first element? Note that the string pointed to by the second argument (temp) is copied into the array pointed to by the first argument (qword[i]). The copy is called the target and the original string is called the source. You can remember the order of the arguments by noting that it is the same as the order in an assignment statement (the target string is on the left):

```
char target[20];
int x;
x = 50;                  /* assignment for numbers */
strcpy(target, "Hi ho!");  /* assignment for strings */
target = "So long";       /* syntax error           */
```

It is your responsibility to make sure the destination array has enough room to copy the source. The following is asking for trouble:

```
char * str;
strcpy(str, "The C of Tranquility"); /* a problem */
```

The function will copy the string "The C of Tranquility" to the address specified by str, but str is uninitialized, so the copy might wind up anywhere!

In short, strcpy() takes two string pointers as arguments. The second pointer, which points to the original string, can be a declared pointer, an array name, or a string constant. The first pointer, which points to the copy, should point to a data object, such as an array, roomy enough to hold the string. Remember, declaring an array allocates storage space for data; declaring a pointer only allocates storage space for one address.

The sprintf() Function

The sprintf() function is declared in stdio.h instead of string.h. It works like printf(), but it writes to a string instead of writing to a display. Therefore, it provides a way to combine several elements into a single string. The first argument to sprintf() is the address of the target string. The remaining arguments are the same as for printf()—a conversion specification string followed by a list of items to be written.

The following listing uses sprintf() to combine three items (two strings and a number) into a single string. Note that it uses sprintf() the same way you would use printf(), except that the resulting string is stored in the array formal instead of being displayed onscreen.

The format.c Program

```
/* format.c -- format a string */
```

```
#include <stdio.h>
#define MAX 20
int main(void)
{
        char first[MAX];
        char last[MAX];
        char formal[2 * MAX + 10];
        double prize;
        puts("Enter your first name:");
        gets(first);
        puts("Enter your last name:");
        gets(last);
        puts("Enter your prize money:");
        scanf("%lf", &prize);
        sprintf(formal, "%s, %-19s: $%6.2f\n", last, first, prize);
        puts(formal);
        return 0;
}
```

Here's a sample run:

```
Enter your first name:
Teddy
Enter your last name:
Behr
Enter your prize money:
2000
Behr, Teddy              : $2000.00
```

The sprintf() command took the input and formatted it into a standard form, which it then stored in the string formal.

Other String Functions

The ANSI C library has more than 20 string-handling functions and the following list summarizes some of the more commonly used ones:

- char *strcpy(char * s1, const char * s2);

  This function copies the string (including the null character) pointed to by s2 to the location pointed to by s1. The return value is s1.

- char *strncpy(char * s1, const char * s2, size_t n);

  This function copies to the location pointed to by s1 no more than n characters from the string pointed to by s2. The return value is s1. No characters after a null character are copied and, if the source string is shorter than n characters, the target string is padded with null characters. If the source string has n or more characters, no null character is copied. The return value is s1.

- char *strcat(char * s1, const char * s2);

  The string pointed to by s2 is copied to the end of the string pointed to by s1. The first character of the s2 string is copied over the null character of the s1 string. The return value is s1.

- char *strncat(char * s1, const char * s2, size_t n);

  No more than the first n characters of the s2 string are appended to the s1 string, with the first character of the s2 string being copied over the null character of the s1 string. The null character and any characters following it in the s2 string are not copied and a null character is appended to the result. The return value is s1.

- int strcmp(const char * s1, const char * s2);

  This function returns a positive value if the s1 string follows the s2 string in the machine collating sequence, the value 0 if the two strings are identical and a negative value if the first string precedes the second string in the machine collating sequence.

- int strncmp(const char * s1, const char * s2, size_t n);

  This function works like strcmp(), except that the comparison stops after n characters or when the first null character is encountered, whichever comes first.

- char *strchr(const char * s, int c);

  This function returns a pointer to the first location in the string s that holds the character c. (The terminating null character is part of the string, so it can be searched for.) The function returns the null pointer if the character is not found.

- char *strpbrk(const char * s1, const char * s2);

  This function returns a pointer to the first location in the string s1 that holds any character found in the s2 string. The function returns the null pointer if no character is found.

- char *strrchr(const char * s, int c);

This function returns a pointer to the last occurrence of the character c in the string s. (The terminating null character is part of the string, so it can be searched for.) The function returns the null pointer if the character is not found.

- char *strstr(const char * s1, const char * s2);

  This function returns a pointer to the first occurrence of string s2 in string s1. The function returns the null pointer if the string is not found.

- size_t strlen(const char * s);

  This function returns the number of characters, not including the terminating null character, found in the string s.

Note that these prototypes use the keyword const to indicate which strings are not altered by a function. For example, consider the following:

char *strcpy(char * s1, const char * s2);

It means s2 points to a string that can't be changed, at least not by the strcpy() function, but s1 points to a string that can be changed. This makes sense, because s1 is the target string, which gets altered, and s2 is the source string, which should be left unchanged.

6. Summary

The C language does not support string data type. String must be terminated by appending a null character. But string can be simulated either using character array or pointer to character. The C language has a rich set of string functions, which operate on both representations of the string.

7. Short Answer Type Questions

1. What is the difference between character array and character pointer?
2. What is the purpose of strlen() function?
3. What is the purpose of strcat() function?

8. Long Answer Type Questions

1. Define and distinguish between character array and character pointer modes of string representation.
2. What are the various ways of storing strings?
3. What are the different ways of copying string?
4. What are the different ways of comparing strings?

9. Suggested Books

1.	Application Programming in C	R. S. Salaria
2.	C Programming using Turbo C	Robert Lafore
3.	Programming with ANSI and Turbo C	Ashok N. Kamthane

4.	Programming using C	E. Balagurusamy
5.	Let Us C	Yashwant Kantekar

BA PART-2 (SEM-3) **PAPER : BAP-201**

C Programming and Data Structures

Lesson No. 11 **Author: Dr. DharamVeer Sharma**

Converted into SLM by: Dr. Vishal Singh

Structures

1. Introduction

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. (Structures are called ``records'' in some languages, notably Pascal.) Structures help to organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities.

One traditional example of a structure is the payroll record: an employee is described by a set of attributes such as name, address, social security number, salary, etc. Some of these in turn could be structures: a name has several components, as does an address and even a salary. Another example, more typical for C, comes from graphics: a point is a pair of coordinate, a rectangle is a pair of points, and so on.

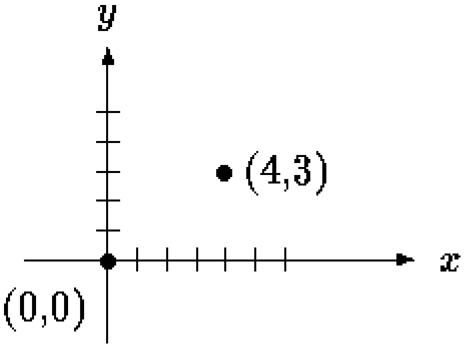
The main change made by the ANSI standard is to define structure assignment - structures may be copied and assigned to, passed to functions, and returned by functions. This has been supported by most compilers for many years, but the properties are now precisely defined. Automatic structures and arrays may now also be initialized.

2. Objectives

Structures and unions are the subject matter of this lesson. We shall delve into details of the declaration, use and importance of structures and unions.

3. Basics of Structures

A structure is a collection o variable referenced under are name, providing a convenient means of keeping related information together. Let us create a few structures suitable for graphics. The basic object is a point, which we will assume has an *x* coordinate and a *y* coordinate, both integers.



The two components can be placed in a structure declared like this:

```
struct point {
        int x;
        int y;
};
```

The keyword struct introduces a structure declaration, which is a list of declarations enclosed in braces. An optional name called a *structure tag* may follow the word struct (as with point here). The tag names this kind of structure and can be used subsequently as a shorthand for the part of the declaration in braces.

The variables named in a structure are called *members*. A structure member or tag and an ordinary (i.e., non-member) variable can have the same name without conflict, since they can always be distinguished by context. Furthermore, the same member names may occur in different structures, although as a matter of style one would normally use the same names only for closely related objects.

A struct declaration defines a type. The right brace that terminates the list of members may be followed by a list of variables, just as for any basic type. That is,

```
struct { ... } x, y, z;
```

is syntactically analogous to

```
int x, y, z;
```

in the sense that each statement declares x, y and z to be variables of the named type and causes space to be set aside for them.

A structure declaration that is not followed by a list of variables reserves no storage; it merely describes a template or shape of a structure. If the declaration is tagged, however, the tag can be used later in definitions of instances of the structure. For example, given the declaration of point above,

```
struct point pt;
```

defines a variable pt which is a structure of type struct point. A structure can be initialized by following its definition with a list of initializers, each a constant expression, for the members:

```
struct maxpt = { 320, 200 };
```

An automatic structure may also be initialized by assignment or by calling a function that returns a structure of the right type.

A member of a particular structure is referred to in an expression by a construction of the form

structure-name.member

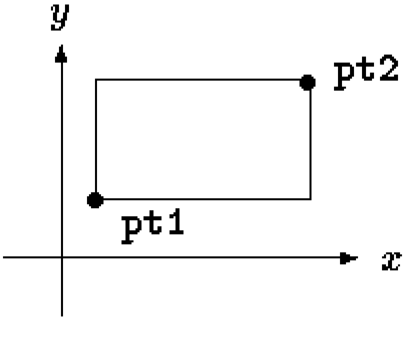
The structure member operator ``." connects the structure name and the member name. To print the coordinates of the point pt, for instance,

```
printf("%d,%d", pt.x, pt.y);
```

or to compute the distance from the origin (0,0) to pt,

```
double dist, sqrt(double);
dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

Structures can be nested. One representation of a rectangle is a pair of points that denote the diagonally opposite corners:



```
struct rect {
        struct point pt1;
        struct point pt2;
};
```

The rect structure contains two point structures. If we declare screen as

```
struct rect screen;
```

then

```
screen.pt1.x
```

refers to the *x* coordinate of the pt1 member of screen.

4. Structures and Functions

The only legal operations on a structure are copying it or assigning to it as a unit, taking its address with &, and accessing its members. Copy and assignment inpclude passing arguments to functions and returning values from functions as well. Structures may not be compared. A structure may be initialized by a list of constant member values; an automatic structure may also be initialized by an assignment.

Let us investigate structures by writing some functions to manipulate points and rectangles. There are at least three possible approaches: pass components separately, pass an entire structure or pass a pointer to it. Each has its good points and bad points.

The first function, makepoint, will take two integers and return a point structure:

```
/* makepoint:  make a point from x and y components */
struct point makepoint(int x, int y)
{
        struct point temp;
        temp.x = x;
        temp.y = y;
        return temp;
}
```

Notice that there is no conflict between the argument name and the member with the same name; indeed the re-use of the names stresses the relationship.

makepoint function can now be used to initialize any structure dynamically, or to provide structure arguments to a function:

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);
screen.pt1 = makepoint(0,0);
screen.pt2 = makepoint(XMAX, YMAX);
```

```
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
(screen.pt1.y + screen.pt2.y)/2);
```

The next step is a set of functions to do arithmetic on points. For instance,

```
/* addpoints:  add two points */
struct addpoint(struct point p1, struct point p2)
{
        p1.x += p2.x;
p1.y += p2.y;
        return p1;
}
```

Here both the arguments and the return value are structures. We incremented the components in p1 rather than using an explicit temporary variable to emphasize that structure parameters are passed by value like any others.

As another example, the function ptinrect tests whether a point is inside a rectangle, where we have adopted the convention that a rectangle includes its left and bottom sides but not its top and right sides:

```
/* ptinrect:  return 1 if p in r, 0 if not */
int ptinrect(struct point p, struct rect r)
{
        return p.x >= r.pt1.x && p.x < r.pt2.x && p.y >=
        r.pt1.y &&    p.y < r.pt2.y;
}
```

This assumes that the rectangle is presented in a standard form where the pt1 coordinates are less than the pt2 coordinates. The following function returns a rectangle guaranteed to be in canonical form:

```
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))
/* canonrect: canonicalize coordinates of rectangle */
struct rect canonrect(struct rect r)
{
        struct rect temp;
        temp.pt1.x = min(r.pt1.x, r.pt2.x);
        temp.pt1.y = min(r.pt1.y, r.pt2.y);
        temp.pt2.x = max(r.pt1.x, r.pt2.x);
        temp.pt2.y = max(r.pt1.y, r.pt2.y);
        return temp;
    }
```

If a large structure is to be passed to a function, it is generally more efficient to pass a pointer than to copy the whole structure. Structure pointers are just like pointers to ordinary variables. The declaration

```
    struct point *pp;
```

says that pp is a pointer to a structure of type struct point. If pp points to a point structure, *pp is the structure, and (*pp).x and (*pp).y are the members. To use pp, we might write, for example,

```
    struct point origin, *pp;
    pp = &origin;
    printf("origin is (%d,%d)\n", (*pp).x, (*pp).y);
```

The parentheses are necessary in (*pp).x because the precedence of the structure member operator . is higher then *. The expression *pp.x means *(pp.x), which is illegal here because x is not a pointer.

Pointers to structures are so frequently used that an alternative notation is provided as a shorthand. If p is a pointer to a structure, then

p->*member-of-structure*

refers to the particular member. So we could write instead

```
    printf("origin is (%d,%d)\n", pp->x, pp->y);
```

Both . and -> associate from left to right, so if we have

```
    struct rect r, *rp = &r;
```

then these four expressions are equivalent:

```
    r.pt1.x
    rp->pt1.x
    (r.pt1).x
    (rp->pt1).x
```

The structure operators . and ->, together with () for function calls and [] for subscripts, are at the top of the precedence hierarchy and thus bind very tightly. For example, given the declaration

```
    struct {
        int len;
        char *str;
        } *p;
then
        ++p->len
```

increments len, not p, because the implied parenthesization is ++(p->len). Parentheses can be used to alter binding: (++p)->len increments p before accessing len, and (p++)->len increments p afterward. (This last set of parentheses is unnecessary.)

In the same way, *p->str fetches whatever str points to; *p->str++ increments str after accessing whatever it points to (just like *s++); (*p->str)++ increments whatever str points to; and *p++->str increments p after accessing whatever str points to.

5. Arrays of Structures

Consider writing a program to count the occurrences of each C keyword. We need an array of character strings to hold the names and an array of integers for the counts. One possibility is to use two parallel arrays, keyword and keycount, as in

```
char *keyword[NKEYS];

int keycount[NKEYS];
```

But the very fact that the arrays are parallel suggests a different organization, an array of structures. Each keyword is a pair:

```
char *word;

int count;
```

and there is an array of pairs. The structure declaration

```
struct key {

        char *word;

        int count;

} keytab[NKEYS];
```

declares a structure type key, defines an array keytab of structures of this type, and sets aside storage for them. Each element of the array is a structure. This could also be written

```
struct key {

        char *word;

        int count;

};

struct key keytab[NKEYS];
```

Since the structure keytab contains a constant set of names, it is easiest to make it an external variable and initialize it once and for all when it is defined. The structure initialization is analogous to earlier ones - the definition is followed by a list of initializers enclosed in braces:

```
struct key {
        char *word;
        int count;
} keytab[] = {
                "auto", 0,
                "break", 0,
                "case", 0,
                "char", 0,
                "const", 0,
                "continue", 0,
                "default", 0,
                /* ... */
                "unsigned", 0,
                "void", 0,
                "volatile", 0,
                "while", 0
};
```

The initializers are listed in pairs corresponding to the structure members. It would be more precise to enclose the initializers for each "row" or structure in braces, as in

```
{ "auto", 0 },
{ "break", 0 },
{ "case", 0 },
...
```

but inner braces are not necessary when the initializers are simple variables or character strings and when all are present. As usual, the number of entries in the array keytab will be computed if the initializers are present and the [] is left empty.

The keyword counting program begins with the definition of keytab. The main routine reads the input by repeatedly calling a function getword that fetches one word at a time. Each word is looked up in keytab with a version of the binary search function. The list of keywords must be sorted in increasing order in the table.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100
int getword(char *, int);
int binsearch(char *, struct key *, int);
/* count C keywords */
main()
{
          int n;
          char word[MAXWORD];
          while (getword(word, MAXWORD) != EOF)
                    if (isalpha(word[0]))
                              if ((n = binsearch(word, keytab, NKEYS)) >= 0)
                                        keytab[n].count++;
          for (n = 0; n < NKEYS; n++)
                    if (keytab[n].count > 0)
                              printf("%4d %s\n",
          keytab[n].count, keytab[n].word);
          return 0;
}
/* binsearch:  find word in tab[0]...tab[n-1] */
int binsearch(char *word, struct key tab[], int n)
{
          int cond;
          int low, high, mid;
          low = 0;
          high = n - 1;
          while (low <= high) {
                    mid = (low+high) / 2;
```

```
                if ((cond = strcmp(word, tab[mid].word)) < 0)
                        high = mid - 1;
                else if (cond > 0)
                        low = mid + 1;
                else
                        return mid;
        }
        return -1;
}
```

We will show the function getword in a moment; for now it suffices to say that each call to getword function finds a word, which is copied into the array named as its first argument.

The quantity NKEYS is the number of keywords in keytab. Although we could count this by hand, it's a lot easier and safer to do it by machine, especially if the list is subject to change. One possibility would be to terminate the list of initializers with a null pointer, then loop along keytab until the end is found.

But this is more than is needed, since the size of the array is completely determined at compile time. The size of the array is the size of one entry times the number of entries, so the number of entries is just

size of keytab / *size of* struct key

C provides a compile-time unary operator called sizeof that can be used to compute the size of any object. The expressions

sizeof *object*

and

sizeof (*type name*)

yield an integer equal to the size of the specified object or type in bytes. (Strictly, sizeof produces an unsigned integer value whose type, size_t, is defined in the header <stddef.h>.) An object can be a variable or array or structure. A type name can be the name of a basic type like int or double or a derived type like a structure or a pointer.

In our case, the number of keywords is the size of the array divided by the size of one element. This computation is used in a #define statement to set the value of NKEYS:

```
#define NKEYS (sizeof keytab / sizeof(struct key))
```

Another way to write this is to divide the array size by the size of a specific element:

```
#define NKEYS (sizeof keytab / sizeof(keytab[0]))
```

This has the advantage that it does not need to be changed if the type changes.

A size of can not be used in a #if line, because the preprocessor does not parse type names. But the expression in the #define is not evaluated by the preprocessor, so the code here is legal.

Now for the function getword. We have written a more general getword than is necessary for this program, but it is not complicated. getword fetches the next ``word'' from the input, where a word is either a string of letters and digits beginning with a letter or a single non-white space character. The function value is the first character of the word, or EOF for end of file or the character itself if it is not alphabetic.

```
/* getword:  get next word or character from input */
int getword(char *word, int lim)
{
        int c, getch(void);
        void ungetch(int);
        char *w = word;
        while (isspace(c = getch()))
        ;
        if (c != EOF)
                *w++ = c;
        if (!isalpha(c)) {
                *w = '\0';
                return c;
        }
        for ( ; --lim > 0; w++)
                if (!isalnum(*w = getch())) {
                        ungetch(*w);
                        break;
                }
        *w = '\0';
        return word[0];
}
```

getword uses the getch and ungetch. When the collection of an alphanumeric token stops, getword has gone one character too far. The call to ungetch pushes that character back on the input for the next call. getword also uses isspace to skip whitespace, isalpha to identify letters and isalnum to identify letters and digits; all are from the standard header <ctype.h>.

6. Pointers to Structures

To illustrate some of the considerations involved with pointers to and arrays of structures, let us write the keyword-counting program again, this time using pointers instead of array indices.

The external declaration of keytab need not change, but main and binsearch do need modification.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100
int getword(char *, int);
struct key *binsearch(char *, struct key *, int);
/* count C keywords; pointer version */
main()
{
    char word[MAXWORD];
    struct key *p;
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((p=binsearch(word, keytab, NKEYS)) != NULL)
                p->count++;
    for (p = keytab; p < keytab + NKEYS; p++)
    if (p->count > 0)
        printf("%4d %s\n", p->count, p->word);
    return 0;
}
/* binsearch: find word in tab[0]...tab[n-1] */
struct key *binsearch(char *word, struck key *tab, int n)
{
    int cond;
```

```
            struct key *low = &tab[0];
            struct key *high = &tab[n];
            struct key *mid;
            while (low < high) {
                    mid = low + (high-low) / 2;
                    if ((cond = strcmp(word, mid->word)) < 0)
                            high = mid;
                    else if (cond > 0)
                            low = mid + 1;
                    else
                            return mid;
            }
            return NULL;
    }
```

There are several things worthy of note here. First, the declaration of binsearch must indicate that it returns a pointer to struct key instead of an integer; this is declared both in the function prototype and in binsearch. If binsearch finds the word, it returns a pointer to it; if it fails, it returns NULL.

Second, the elements of keytab are now accessed by pointers. This requires significant changes in binsearch.

The initializers for low and high are now pointers to the beginning and just past the end of the table.

The computation of the middle element can no longer be simply

```
            mid = (low+high) / 2    /* WRONG */
```

because the addition of pointers is illegal. Subtraction is legal, however, so high-low is the number of elements and thus

```
    mid = low + (high-low) / 2
```

sets mid to the element halfway between low and high.

The most important change is to adjust the algorithm to make sure that it does not generate an illegal pointer or attempt to access an element outside the array. The problem is that &tab[-1] and &tab[n] are both outside the limits of the array tab. The former is strictly illegal and it is illegal to dereference the latter. The language definition does guarantee, however, that pointer arithmetic that involves the first element beyond the end of an array (that is, &tab[n]) will work correctly.

In main we wrote

```
for (p = keytab; p < keytab + NKEYS; p++)
```

If p is a pointer to a structure, arithmetic on p takes into account the size of the structure, so p++ increments p by the correct amount to get the next element of the array of structures and the test stops the loop at the right time.

Don't assume, however, that the size of a structure is the sum of the sizes of its members. Because of alignment requirements for different objects, there may be unnamed ``holes'' in a structure. Thus, for instance, if a char is one byte and an int four bytes, the structure

```
struct {
        char c;
        int i;
};
```

might well require eight bytes, not five. The sizeof operator returns the proper value.

Finally, an aside on program format: when a function returns a complicated type like a structure pointer, as in

```
struct key *binsearch(char *word, struct key *tab, int n)
```

the function name can be hard to see, and to find with a text editor. Accordingly an alternate style is sometimes used:

```
struct key *
binsearch(char *word, struct key *tab, int n)
```

This is a matter of personal taste; pick the form you like and hold to it.

7. Self-referential Structures

Suppose we want to handle the more general problem of counting the occurrences of *all* the words in some input. Since the list of words isn't known in advance, we can't conveniently sort it and use a binary search. Yet we can't do a linear search for each word as it arrives, to see if it's already been seen; the program would take too long. (More precisely, its running time is likely to grow quadratically with the number of input words.) How can we organize the data to copy efficiently with a list or arbitrary words?

One solution is to keep the set of words seen so far sorted at all times, by placing each word into its proper position in the order as it arrives. This shouldn't be done by shifting words in a linear array, though - that also takes too long. Instead we will use a data structure called a *binary tree*.

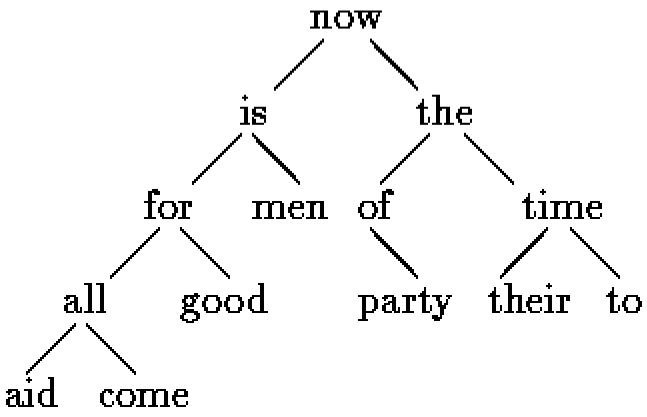
The tree contains one ``node'' per distinct word; each node contains

- A pointer to the text of the word,

- A count of the number of occurrences,
- A pointer to the left child node,
- A pointer to the right child node.

No node may have more than two children; it might have only zero or one.

The nodes are maintained so that at any node the left subtree contains only words that are lexicographically less than the word at the node and the right subtree contains only words that are greater. This is the tree for the sentence ``now is the time for all good men to come to the aid of their party'', as built by inserting each word as it is encountered:



To find out whether a new word is already in the tree, start at the root and compare the new word to the word stored at that node. If they match, the question is answered affirmatively. If the new record is less than the tree word, continue searching at the left child, otherwise at the right child. If there is no child in the required direction, the new word is not in the tree and in fact the empty slot is the proper place to add the new word. This process is recursive, since the search from any node uses a search from one of its children. Accordingly, recursive routines for insertion and printing will be most natural.

Going back to the description of a node, it is most conveniently represented as a structure with four components:

```
struct tnode {     /* the tree node: */
        char *word;           /* points to the text */
        int count;            /* number of occurrences */
        struct tnode *left;   /* left child */
        struct tnode *right;  /* right child */
};
```

This recursive declaration of a node might look chancy, but it's correct. It is illegal for a structure to contain an instance of itself, but

```
struct tnode *left;
```

declares left to be a pointer to a tnode, not a tnode itself.

Occasionally, one needs a variation of self-referential structures: two structures that refer to each other. The way to handle this is:

```
struct t {
        ...
        struct s *p;   /* p points to an s */
};
struct s {
        ...
        struct t *q;   /* q points to a t */
};
```

8. typedef

C provides a facility called typedef for creating new data type names. For example, the declaration

```
typedef int Length;
```

makes the name Length a synonym for int. The type Length can be used in declarations, casts, etc., in exactly the same ways that the int type can be:

```
Length len, maxlen;
Length *lengths[];
```

Similarly, the declaration

```
typedef char *String;
```

makes String a synonym for char * or character pointer, which may then be used in declarations and casts:

```
String p, lineptr[MAXLINES], alloc(int);
int strcmp(String, String);
p = (String) malloc(100);
```

Notice that the type being declared in a typedef appears in the position of a variable name, not right after the word typedef. Syntactically, typedef is like the storage classes extern, static, etc. We have used capitalized names for typedefs, to make them stand out.

As a more complicated example, we could make typedefs for the tree nodes shown earlier in this chapter:

```
typedef struct tnode *Treeptr;
typedef struct tnode { /* the tree node: */
        char *word;          /* points to the text */
        int count;           /* number of occurrences */
        struct tnode *left;   /* left child */
        struct tnode *right;  /* right child */
} Treenode;
```

This creates two new type keywords called Treenode (a structure) and Treeptr (a pointer to the structure). Then the routine talloc could become

```
Treeptr talloc(void)
{
        return (Treeptr) malloc(sizeof(Treenode));
}
```

It must be emphasized that a typedef declaration does not create a new type in any sense; it merely adds a new name for some existing type. Nor are there any new semantics: variables declared this way have exactly the same properties as variables whose declarations are spelled out explicitly. In effect, typedef is like #define, except that since it is interpreted by the compiler, it can cope with textual substitutions that are beyond the capabilities of the preprocessor. For example,

```
typedef int (*PFI)(char *, char *);
```

creates the type PFI, for ``pointer to function (of two char * arguments) returning int,'' which can be used in contexts like

```
PFI strcmp, numcmp;
```

in the sort program.

Besides purely aesthetic issues, there are two main reasons for using typedefs. The first is to parameterize a program against portability problems. If typedefs are used for data types that may be machine-dependent, only the typedefs need change when the program is moved. One common situation is to use typedef names for various integer quantities, then make an appropriate set of choices of short, int, and long for each host machine. Types like size_t and ptrdiff_t from the standard library are examples.

The second purpose of typedefs is to provide better documentation for a program - a type called Treeptr may be easier to understand than one declared only as a pointer to a complicated structure.

9. Union

A *union* is a variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements. Unions provide a way to manipulate different kinds of data in a single area of storage, without embedding any machine-dependent information in the program. They are analogous to variant records in pascal.

As an example such as might be found in a compiler symbol table manager, suppose that a constant may be an int, a float, or a character pointer. The value of a particular constant must be stored in a variable of the proper type, yet it is most convenient for table management if the value occupies the same amount of storage and is stored in the same place regardless of its type. This is the purpose of a union - a single variable that can legitimately hold any of one of several types. The syntax is based on structures:

```
union u_tag {
        int ival;
        float fval;
        char *sval;
} u;
```

The variable u will be large enough to hold the largest of the three types; the specific size is implementation-dependent. Any of these types may be assigned to u and then used in expressions, so long as the usage is consistent: the type retrieved must be the type most recently stored. It is the programmer's responsibility to keep track of which type is currently stored in a union; the results are implementation-dependent if something is stored as one type and extracted as another.

Syntactically, members of a union are accessed as

union-name.member

or

union-pointer->member

just as for structures. If the variable utype is used to keep track of the current type stored in u, then one might see code such as

```
if (utype == INT)
        printf("%d\n", u.ival);
if (utype == FLOAT)
        printf("%f\n", u.fval);
if (utype == STRING)
        printf("%s\n", u.sval);
else
        printf("bad type %d in utype\n", utype);
```

Unions may occur within structures and arrays and vice versa. The notation for accessing a member of a union in a structure (or vice versa) is identical to that for nested structures. For example, in the structure array defined by

```
struct {
        char *name;
        int flags;
        int utype;
        union {
                int ival;
                float fval;
                char *sval;
        } u;
} symtab[NSYM];
```

the member ival is referred to as

```
symtab[i].u.ival
```

and the first character of the string sval by either of

```
*symtab[i].u.sval
symtab[i].u.sval[0]
```

In effect, a union is a structure in which all members have offset zero from the base, the structure is big enough to hold the ``widest'' member and the alignment is appropriate for all of the types in the union. The same operations are permitted on unions as on structures: assignment to or copying as a unit, taking the address and accessing a member.

A union may only be initialized with a value of the type of its first member; thus union u described above can only be initialized with an integer value.

The storage allocator in shows how a union can be used to force a variable to be aligned on a particular kind of storage boundary.

10. Summary

Structures facilitate declaration of composite data types of heterogeneous variable. A structure may contain variables of different data types. These represent the logical organization of distinct variables as one unit. Structures may be self referential, where one structure may contain instances of itself. Unions are like structure. The only difference is that for union the memory allocated is equal to the largest sized data types declared in that union.

11. Short Answer Type Questions

1. Define structure.
2. What is the purpose of using a structure?
3. How an array of structures is declared and used?
4. How a pointer to a structure is declared and used?

12. Long Answer Type Questions

1. How structures are declared and used ? Explain by giving examples.
2. Give an example of array of structures.
3. What are self referential structures?
4. What is the difference between structure and union?

13. Suggested Books

1.	Application Programming in C	R. S. Salaria
2.	C Programming using Turbo C	Robert Lafore
3.	Programming with ANSI and Turbo C	Ashok N. Kamthane
4.	Programming using C	E. Balagurusamy
5.	Let Us C	Yashwant Kantekar

BA PART-2 (SEM-3) **PAPER : BAP-201**
C Programming and Data Structures

Lesson No. 12 **Author: Dr. DharamVeer Sharma**
Converted into SLM by: Dr. Vishal Singh

POINTERS

1. Introduction

A pointer is a variable that contains the address of a variable. Pointers are much used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways. Pointers and arrays are closely related; this chapter also explores this relationship and shows how to exploit it.

Pointers have been lumped with the goto statement as a marvelous way to create impossible-to-understand programs. This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected. With discipline, however, pointers can also be used to achieve clarity and simplicity. This is the aspect that we will try to illustrate.

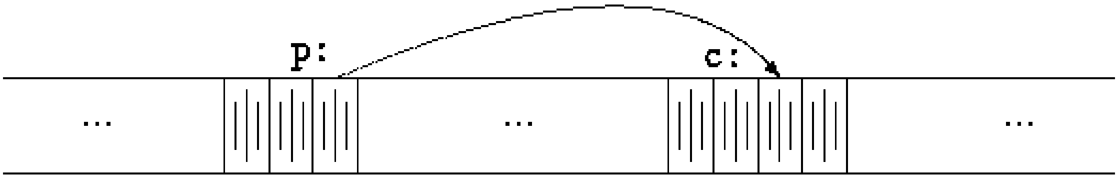
The main change in ANSI C is to make explicit the rules about how pointers can be manipulated, in effect mandating what good programmers already practice and good compilers already enforce. In addition, the type void * (pointer to void) replaces char * as the proper type for a generic pointer.

2. Objectives

Pointers play important role in efficient programming but are very dangerous if proper care is not taken while using them. Through pointers we can directly access the memory. In this lesson we shall discuss the concept of pointers and how to use pointers, including pointer arithmetic, passing pointers as function arguments, relation between arrays and pointers.

3. Pointers and Addresses

Let us begin with a simplified picture of how memory is organized. A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups. One common situation is that any byte can be a char, a pair of one-byte cells can be treated as a short integer, and four adjacent bytes form a long. A pointer is a group of cells (often two or four) that can hold an address. So if c is a char and p is a pointer that points to it, we could represent the situation this way:



The unary operator & gives the address of an object, so the statement p = &c; assigns the address of c to the variable p, and p is said to ``point to'' c. The & operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants or register variables. So, we can say that a **pointer** is a variable that points to (or contains) address of another variable.

The unary operator * is the *indirection* or *dereferencing* operator; when applied to a pointer, it accesses the object the pointer points to. Suppose that x and y are integers and ip is a pointer to int. This artificial sequence shows how to declare a pointer and how to use & and *:

```
int x = 1, y = 2, z[10];
int *ip;        /* ip is a pointer to int */
ip = &x;        /* ip now points to x */
y = *ip;        /* y is now 1 */
*ip = 0;        /* x is now 0 */
ip = &z[0];     /* ip now points to z[0] */
```

The declaration of x, y, and z are what we've seen all along. The declaration of the pointer ip,

```
int *ip;
```

is intended as a mnemonic; it says that the expression *ip is an int. The syntax of the declaration for a variable mimics the syntax of expressions in which the variable might appear. This reasoning applies to function declarations as well. For example,

```
double *dp, atof(char *);
```

says that in an expression *dp and atof(s) have values of double and that the argument of atof is a pointer to char.

You should also note the implication that a pointer is constrained to point to a particular kind of object: every pointer points to a specific data type. (There is one exception: a ``pointer to void'' is used to hold any type of pointer but cannot be dereferenced itself.)

If ip points to the integer x, then *ip can occur in any context where x could, so

```
*ip = *ip + 10;
```

increments *ip by 10.

The unary operators * and & bind more tightly than arithmetic operators, so the assignment

```
y = *ip + 1
```

takes whatever ip points at, adds 1 and assigns the result to y, while

```
*ip += 1
```

increments what ip points to, as do

```
++*ip
```

and

```
(*ip)++
```

The parentheses are necessary in this last example; without them, the expression would increment ip instead of what it points to, because unary operators like * and ++ associate right to left.

Finally, since pointers are variables, they can be used without dereferencing. For example, if iq is another pointer to int,

```
iq = ip
```

copies the contents of ip into iq, thus making iq point to whatever ip pointed to.

4. Pointers and Function Arguments

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. For instance, a sorting routine might exchange two out-of-order arguments with a function called swap. It is not enough to write

```
swap(a, b);
```

where the swap function is defined as

```
void swap(int x, int y)  /* WRONG */
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}
```

Because of call by value, swap can't affect the arguments a and b in the routine that called it. The function above swaps *copies* of a and b.

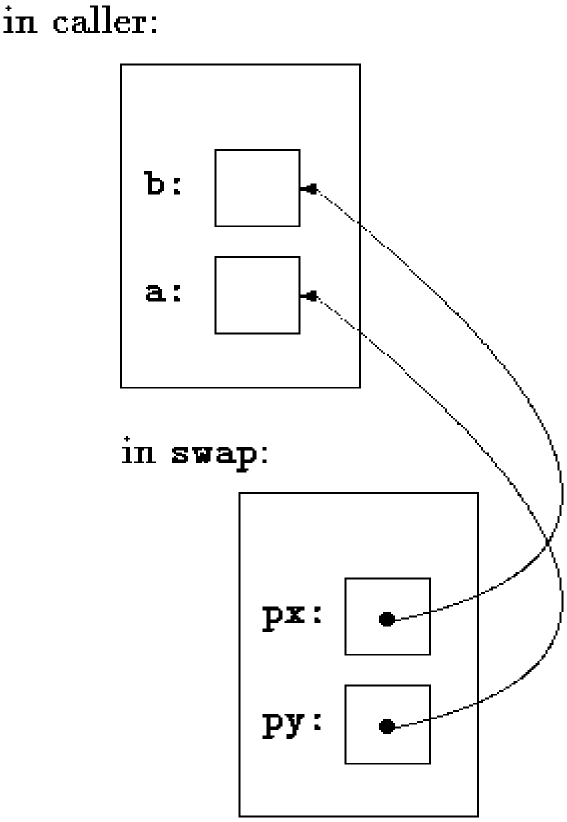
The way to obtain the desired effect is for the calling program to pass *pointers* to the values to be changed:

```
swap(&a, &b);
```

Since the operator & produces the address of a variable, &a is a pointer to a. In swap itself, the parameters are declared as pointers and the operands are accessed indirectly through them.

```
void swap(int *px, int *py)  /* interchange *px and *py */
{
        int temp;
        temp = *px;
        *px = *py;
        *py = temp;
}
```

Pictorially:



Pointer arguments enable a function to access and change objects in the function that called it. As an example, consider a function getint that performs free-format input conversion by breaking a stream of characters into integer values, one integer per call. getint has to return the value it found and also signal end of file when there is no more input. These values have to be passed back by separate paths, for no matter what value is used for EOF, that could also be the value of an input integer.

One solution is to have getint return the end of file status as its function value, while using a pointer argument to store the converted integer back in the calling function. This is the scheme used by scanf as well.

The following loop fills an array with integers by calls to getint:

```
int n, array[SIZE], getint(int *);

for (n = 0; n < SIZE && getint(&array[n]) != EOF; n++)

;
```

Each call sets array[n] to the next integer found in the input and increments n. Notice that it is essential to pass the address of array[n] to getint. Otherwise there is no way for getint to communicate the converted integer back to the caller.

Our version of getint returns EOF for end of file, zero if the next input is not a number, and a positive value if the input contains a valid number.

```
#include <ctype.h>

int getch(void);
```

```
void ungetch(int);
/* getint:  get next integer from input into *pn */
int getint(int *pn)
{
    int c, sign;
    while (isspace(c = getch()))   /* skip white space */
  ;
    if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetch(c);  /* it is not a number */
        return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getch();
    for (*pn = 0; isdigit(c), c = getch())
    *pn = 10 * *pn + (c - '0');
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return c;
}
```

Throughout getint, *pn is used as an ordinary int variable. We have also used getch and ungetch so the one extra character that must be read can be pushed back onto the input.

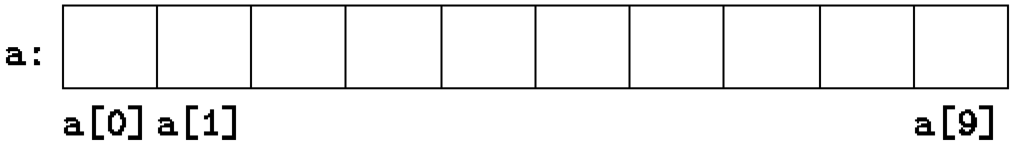
5. Pointers and Arrays

In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand.

The declaration

```
int a[10];
```

defines an array of size 10, that is, a block of 10 consecutive objects named a[0], a[1], ...,a[9].



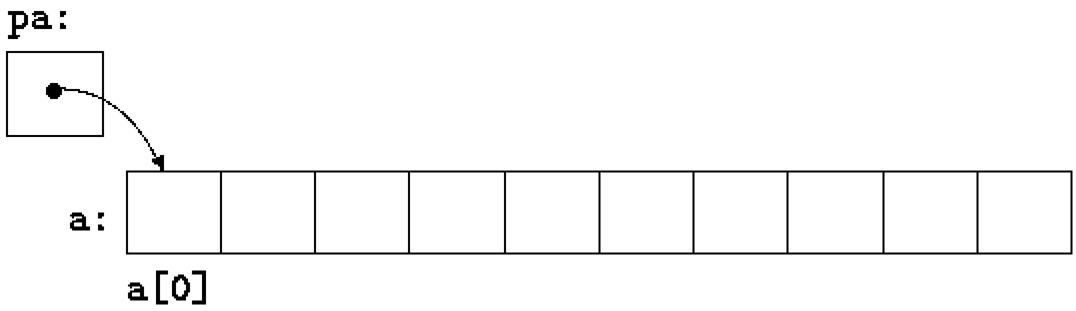
The notation a[i] refers to the i-th element of the array. If pa is a pointer to an integer, declared as

```
int *pa;
```

then the assignment

```
pa = &a[0];
```

sets pa to point to element zero of a; that is, pa contains the address of a[0].



Now the assignment

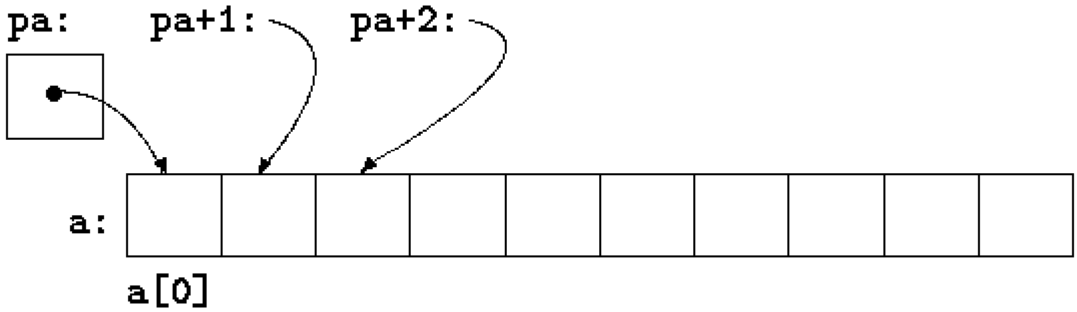
```
x = *pa;
```

will copy the contents of a[0] into x.

If pa points to a particular element of an array, then by definition pa+1 points to the next element, pa+i points i elements after pa and pa-i points i elements before. Thus, if pa points to a[0],

```
*(pa+1)
```

refers to the contents of a[1], pa+i is the address of a[i] and *(pa+i) is the contents of a[i].



These remarks are true regardless of the type or size of the variables in the array a. The meaning of ``adding 1 to a pointer,'' and by extension, all pointer arithmetic, is that pa+1 points to the next object, and pa+i points to the i-th object beyond pa.

The correspondence between indexing and pointer arithmetic is very close. **By definition, the value of a variable or expression of type array is the address of element zero of the array**. Thus after the assignment

```
pa = &a[0];
```

pa and a have identical values. Since the name of an array is a synonym for the location of the initial element, the assignment pa=&a[0] can also be written as

```
pa = a;
```

Rather more surprising, at first sight, is the fact that a reference to a[i] can also be written as *(a+i). In evaluating a[i], C converts it to *(a+i) immediately; the two forms are equivalent. Applying the operator & to both parts of this equivalence, it follows that &a[i] and a+i are also identical: a+i is the address of the i-th element beyond a. As the other side of this coin, if pa is a pointer, expressions might use it with a subscript; pa[i] is identical to *(pa+i). In short, an array-and-index expression is equivalent to one written as a pointer and offset.

There is one difference between an array name and a pointer that must be kept in mind. **A pointer is a variable, so pa=a and pa++ are legal. But an array name is not a variable; constructions like a=pa and a++ are illegal**.

When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable and so an array name parameter is a pointer, that is, a variable containing an address. We can use this fact to write another version of strlen, which computes the length of a string.

```
/* strlen:  return length of string s */
int strlen(char *s)
{
        int n;
        for (n = 0; *s != '\0', s++)
                n++;
        return n;
    }
```

Since s is a pointer, incrementing it is perfectly legal; s++ has no effect on the character string in the function that called strlen, but merely increments strlen's private copy of the pointer. That means that calls like

```
        strlen("hello, world");   /* string constant */
        strlen(array);            /* char array[100]; */
        strlen(ptr);              /* char *ptr; */
```

all work.

As formal parameters in a function definition,

```
char s[];
```

and

```
char *s;
```

are equivalent; we prefer the latter because it says more explicitly that the variable is a pointer. When an array name is passed to a function, the function can at its convenience believe that it has been handed either an array or a pointer and manipulate it accordingly. It can even use both notations if it seems appropriate and clear.

It is possible to pass part of an array to a function, by passing a pointer to the beginning of the subarray. For example, if a is an array,

```
f(&a[2])
```

and

```
f(a+2)
```

both pass to the function f the address of the subarray that starts at a[2]. Within f, the parameter declaration can read

```
f(int arr[]) { ... }
```

or

```
f(int *arr) { ... }
```

So as far as f is concerned, the fact that the parameter refers to part of a larger array is of no consequence.

If one is sure that the elements exist, it is also possible to index backwards in an array; p[-1], p[-2] and so on are syntactically legal and refer to the elements that immediately precede p[0]. Of course, it is illegal to refer to objects that are not within the array bounds.

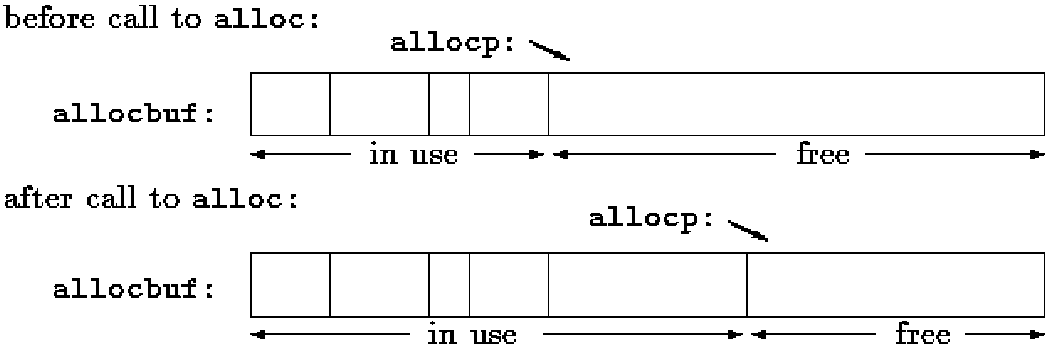
6. Address Arithmetic

We can perform certain arithmetic operations on pointer variables themselves. If p is a pointer to some element of an array, then p++ increments p to point to the next element, and p+=i increments it to point i elements beyond where it currently does. Note that, p++ causes p to point to the next element, not to the next memory location. These and similar constructions are the simples forms of pointer or address arithmetic.

C is consistent and regular in its approach to address arithmetic; its integration of pointers, arrays, and address arithmetic is one of the strengths of the language. Let us illustrate by writing a rudimentary storage allocator. There are two routines. The first, alloc(n), returns a pointer to n consecutive character positions, which can be used by the caller of alloc for storing characters. The second, afree(p), releases the storage thus acquired so it can be re-used later. The routines are ``rudimentary'' because the calls to afree must be made in the opposite order to the calls made on alloc. That is, the storage managed by alloc and afree is a stack or last-in, first-out. The standard library provides analogous functions called malloc and free that have no such restrictions.

The easiest implementation is to have alloc hand out pieces of a large character array that we will call allocbuf. This array is private to alloc and afree. Since they deal in pointers, not array indices, no other routine need know the name of the array, which can be declared static in the source file containing alloc and afree, and thus be invisible outside it. In practical implementations, the array may well not even have a name; it might instead be obtained by calling malloc or by asking the operating system for a pointer to some unnamed block of storage.

The other information needed is how much of allocbuf has been used. We use a pointer, called allocp, that points to the next free element. When alloc is asked for n characters, it checks to see if there is enough room left in allocbuf. If so, alloc returns the current value of allocp (i.e., the beginning of the free block), then increments it by n to point to the next free area. If there is no room, alloc returns zero. afree(p) merely sets allocp to p if p is inside allocbuf.



```
#define ALLOCSIZE 10000 /* size of available space */
static char allocbuf[ALLOCSIZE]; /* storage for alloc */
static char *allocp = allocbuf;  /* next free position */
char *alloc(int n)    /* return pointer to n characters */
{
        if (allocbuf + ALLOCSIZE - allocp >= n) {  /* it fits */
                allocp += n;
                return allocp - n; /* old p */
        } else      /* not enough room */
                return 0;
}

void afree(char *p)  /* free storage pointed to by p */
{
        if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
                allocp = p;
}
```

In general a pointer can be initialized just as any other variable can, though normally the only meaningful values are zero or an expression involving the address of previously defined data of appropriate type. The declaration

```
static char *allocp = allocbuf;
```

defines allocp to be a character pointer and initializes it to point to the beginning of allocbuf, which is the next free position when the program starts. This could also have been written

```
static char *allocp = &allocbuf[0];
```

since the array name *is* the address of the zeroth element.

The test

```
if (allocbuf + ALLOCSIZE - allocp >= n) {  /* it fits */
```

checks if there's enough room to satisfy a request for n characters. If there is, the new value of allocp would be at most one beyond the end of allocbuf. If the request can be satisfied, alloc returns a pointer to the beginning of a block of characters (notice the declaration of the function itself). If not, alloc must return some signal that there is no space left. C guarantees that zero is never a valid address for data, so a return value of zero can be used to signal an abnormal event, in this case no space.

Pointers and integers are not interchangeable. Zero is the sole exception: the constant zero may be assigned to a pointer, and a pointer may be compared with the constant zero. The symbolic constant NULL is often used in place of zero, as a mnemonic to indicate more clearly that this is a special value for a pointer. NULL is defined in <stdio.h>. We will use NULL henceforth.

Tests like

```
if (allocbuf + ALLOCSIZE - allocp >= n) {  /* it fits */
```

and

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

show several important facets of pointer arithmetic. First, pointers may be compared under certain circumstances. If p and q point to members of the same array, then relations like ==, !=, <, >=, etc., work properly. For example,

```
p < q
```

is true if p points to an earlier element of the array than q does. Any pointer can be meaningfully compared for equality or inequality with zero. But the behavior is undefined for arithmetic or comparisons with pointers that do not point to members of the same array. (There is one exception: the address of the first element past the end of an array can be used in pointer arithmetic.)

Second, we have already observed that a pointer and an integer may be added or subtracted. The construction

```
p + n
```

means the address of the n-th object beyond the one p currently points to. This is true regardless of the kind of object p points to; n is scaled according to the size of the objects p points to, which is determined by the declaration of p. If an int is four bytes, for example, the int will be scaled by four.

Pointer subtraction is also valid: if p and q point to elements of the same array, and p<q, then q-p+1 is the number of elements from p to q inclusive. This fact can be used to write yet another version of strlen:

```
/* strlen:  return length of string s */
int strlen(char *s)
{
        char *p = s;
        while (*p != '\0')
                p++;
        return p - s;
    }
```

In its declaration, p is initialized to s, that is, to point to the first character of the string. In the while loop, each character in turn is examined until the '\0' at the end is seen. Because p points to characters, p++ advances p to the next character each time, and p-s gives the number of characters advanced over, that is, the string length. (The number of characters in the string could be too large to store in an int. The header <stddef.h> defines a type ptrdiff_t that is large enough to hold the signed difference of two pointer values. If we were being cautious, however, we would use size_t for the return value of strlen, to match the standard library version. size_t is the unsigned integer type returned by the sizeof operator.

Pointer arithmetic is consistent: if we had been dealing with floats, which occupy more storage that chars and if p were a pointer to float, p++ would advance to the next float. Thus we could write another version of alloc that maintains floats instead of chars, merely by changing char to float throughout alloc and afree. All the pointer manipulations automatically take into account the size of the objects pointed to.

The valid pointer operations are assignment of pointers of the same type, adding or subtracting a pointer and an integer, subtracting or comparing two pointers to members of the same array and assigning or comparing to zero. All other pointer arithmetic is illegal. It is not legal to add two pointers or to multiply or divide or shift or mask them, or to add float or double to them or even, except for void *, to assign a pointer of one type to a pointer of another type without a cast.

7. Character Pointers and Functions

A *string constant*, written as

```
"I am a string"
```

is an array of characters. In the internal representation, the array is terminated with the null character '\0' so that programs can find the end. The length in storage is thus one more than the number of characters between the double quotes.

Perhaps the most common occurrence of string constants is as arguments to functions, as in

```
printf("hello, world\n");
```

When a character string like this appears in a program, access to it is through a character pointer; printf receives a pointer to the beginning of the character array. That is, a string constant is accessed by a pointer to its first element.

String constants need not be function arguments. If pmessage is declared as

```
char *pmessage;
```

then the statement

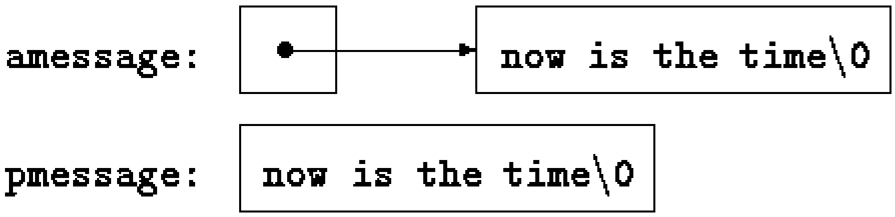
```
pmessage = "now is the time";
```

assigns to pmessage a pointer to the character array. This is *not* a string copy; only pointers are involved. C does not provide any operators for processing an entire string of characters as a unit.

There is an important difference between these definitions:

```
char amessage[] = "now is the time"; /* an array */
char *pmessage = "now is the time"; /* a pointer */
```

amessage is an array, just big enough to hold the sequence of characters and '\0' that initializes it. Individual characters within the array may be changed but amessage will always refer to the same storage. On the other hand, pmessage is a pointer, initialized to point to a string constant; the pointer may subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents.



We will illustrate more aspects of pointers and arrays by studying versions of two useful functions adapted from the standard library. The first function is strcpy(s,t), which copies the string t to the string s. It would be nice just to say s=t but this copies the pointer, not the characters. To copy the characters, we need a loop. The array version first:

```
/* strcpy:  copy t to s; array subscript version */
void strcpy(char *s, char *t)
{
        int i;
        i = 0;
        while ((s[i] = t[i]) != '\0')
                i++;
}
```

For contrast, here is a version of strcpy with pointers:

```
/* strcpy:  copy t to s; pointer version */
void strcpy(char *s, char *t)
```

```
{
        int i;
        i = 0;
        while ((*s = *t) != '\0') {
                s++;
                t++;
        }
```

Because arguments are passed by value, strcpy can use the parameters s and t in any way it pleases. Here, they are conveniently initialized pointers, which are marched along the arrays a character at a time, until the '\0' that terminates when has been copied into s.

In practice, strcpy would not be written as we showed it above. Experienced C programmers would prefer

```
/* strcpy:  copy t to s; pointer version 2 */
void strcpy(char *s, char *t)
{
        while ((*s++ = *t++) != '\0');
}
```

This moves the increment of s and t into the test part of the loop. The value of *t++ is the character that t pointed to before t was incremented; the postfix ++ doesn't change t until after this character has been fetched. In the same way, the character is stored into the old s position before s is incremented. This character is also the value that is compared against '\0' to control the loop. The net effect is that characters are copied from t to s, up and including the terminating '\0'.

As the final abbreviation, observe that a comparison against '\0' is redundant, since the question is merely whether the expression is zero. So the function would likely be written as

```
/* strcpy:  copy t to s; pointer version 3 */
void strcpy(char *s, char *t)
{
        while (*s++ = *t++)   ;
}
```

Although this may seem cryptic at first sight, the notational convenience is considerable, and the idiom should be mastered, because you will see it frequently in C programs.

The strcpy in the standard library (<string.h>) returns the target string as its function value.

The second routine that we will examine is strcmp(s,t), which compares the character strings s and t, and returns negative, zero or positive if s is lexicographically less than, equal to, or greater than t. The value is obtained by subtracting the characters at the first position where s and t disagree.

```
/* strcmp:  return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
        int i;
        for (i = 0; s[i] == t[i]; i++)
                if (s[i] == '\0')
                        return 0;
        return s[i] - t[i];
}
```

The pointer version of strcmp:

```
/* strcmp:  return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
        for ( ; *s == *t; s++, t++)
                if (*s == '\0')
                        return 0;
        return *s - *t;
}
```

Since ++ and -- are either prefix or postfix operators, other combinations of * and ++ and -- occur, although less frequently. For example,

```
*--p
```

decrements p before fetching the character that p points to. In fact, the pair of expressions

```
*p++ = val;  /* push val onto stack */
val = *--p;  /* pop top of stack into val */
```

are the standard idiom for pushing and popping a stack.

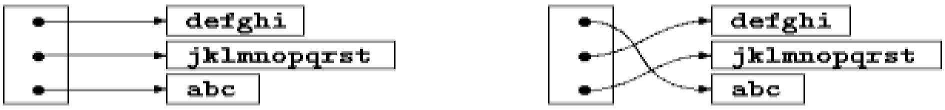
The header <string.h> contains declarations for the functions mentioned in this section, plus a variety of other string-handling functions from the standard library.

8. Pointer Arrays; Pointers to Pointers

Since pointers are variables themselves, they can be stored in arrays just as other variables can.

If we have to deal with lines of text, which are of different lengths, and which, unlike integers, can't be compared or moved in a single operation, we need a data representation that will cope efficiently and conveniently with variable-length text lines.

This is where the array of pointers enters. If the lines to be sorted are stored end-to-end in one long character array, then each line can be accessed by a pointer to its first character. The pointers themselves can be stored in an array. Two lines can be compared by passing their pointers to strcmp. When two out-of-order lines have to be exchanged, the pointers in the pointer array are exchanged, not the text lines themselves.



This eliminates the twin problems of complicated storage management and high overhead that would go with moving the lines themselves.

```
char *lineptr[MAXLINES];  /* pointers to text lines */
```

The main new thing is the declaration for lineptr:

```
char *lineptr[MAXLINES]
```

says that lineptr is an array of MAXLINES elements, each element of which is a pointer to a char. That is, lineptr[i] is a character pointer and *lineptr[i] is the character it points to, the first character of the i-th saved text line.

Since lineptr is itself the name of an array, it can be treated as a pointer in the same manner as in our earlier examples and writelines can be written instead as

```
/* writelines:  write output lines */
void writelines(char *lineptr[], int nlines)
{
        while (nlines-- > 0)
                printf("%s\n", *lineptr++);
}
```

Initially, *lineptr points to the first line; each element advances it to the next line pointer while nlines is counted down.

9. Pointers to Functions

In C, a function itself is not a variable, but it is possible to define pointers to functions, which can be assigned, placed in arrays, passed to functions, returned by functions and so on.

Example

```
#include <stdio.h>
#include <conio.h>
main()
{
        int show();     /*Function prototype*/
        int (*p)();     /*Pointer to function declaration*/
        p=show;         /*Assigning address of show to p*/
        (*p)();         /Function call using pointer*/
        printf("%u",show);   /*Displays address of function*/
}
int show()
{
        clrscr();
        printf("Inside show function whose address is -> ")
}
```

In the above program the variable p is pointer to function. Address of show() is assigned to pointer p. Using function pointer, the function show() is invoked. The output of the program is as under:

```
Inside show function whose address is -> 554
```

10. Summary

C Language facilitates direct access to memory through pointers. A pointer store the address of the variable of type for which pointer is defined. C language also provide parameter passing by address for modifying the values of the actual parameters while making the function calls. Arrays and pointers have strong relation between themselves. Use of pointers makes the programs run faster. Pointer are of great help but should be used very cautiously. If some memory has been allocated to a pointer then it must be freed if it is no longer required because otherwise the memory allocated will not be available for other uses. It will be available only after rebooting of the system. C language does not support string data type but character pointers can solve the purpose. Each string must be terminated by a null otherwise there will be overlapping in the memory read using a character pointer.

11. Short Answer Type Questions

1. What is the difference between int *A[20] and int (*A)[10];
2. What is the difference between array and pointer?
3. What do you mean by pointer arithmetic?
4. What is the data type of a pointer variable?

12. Long Answer Type Questions

1. What is the relation between pointer and addressing? Explain.
2. What is the difference between parameter passing by address and by value?
3. What is the relation between pointer and array?
4. Write the code for string copy and string comparison functions using arrays?

13. Suggested Books

1.	Application Programming in C	R. S. Salaria
2.	C Programming using Turbo C	Robert Lafore
3.	Programming with ANSI and Turbo C	Ashok N. Kamthane
4.	Programming using C	E. Balagurusamy
5.	Let Us C	Yashwant Kantekar

Last Updated on May 2023

Mandatory Student Feedback Form

https://forms.gle/KS5CLhvpwrpgjwN98

Note: Students, kindly click this google form link, and fill this feedback form once.