Course Development Commitee

Chairman
Prof. (Dr.) Naresh Dadhich
Vice-Chancellor
Vardhaman Mahaveer Open University, Kota

Co-ordinator/Convener and Members

Convernor/Coordinator
Prof. (Dr.) D.S. Chauhan
Department of Mathematics
University of Rajasthan, Jaipur

Member Secretary/Coordinator
Sh. Rakesh Sharma
Assistant Professor (Computer Application)
V.M. Open University, Kota

Members

1. Prof. (Dr.) D.S. Chauhan
   Department of Mathematics
   University of Rajasthan, Jaipur
2. Prof. (Dr.) M.C. Govil
   Govt. Engineering College, Ajmer
3. Prof. (Dr.) A.K. Nagawat
   University of Rajasthan, Jaipur
4. Dr. (Mrs.) Madhavi Sinha
   BITS, Jaipur

Editing and Course Writing

Editor
Dr. Vijay Singh Rathore
Director Shree Karni College,
Jaipur

Writers

1. Sh. Ashish Swami
   Subodh P.G.. College
   Jaipur
2. Dr. Leena Bhatia
   Subodh P.G.. College
   Jaipur
3. Ms. Manisha Sharma
   AIM & ACT
   Banasthali University, Jaipur
4. Sh. Sanjeev Sharma
   Agra University, Agra
5. Mrs. Poonam Kshatriya
   AIM & ACT
   Banasthali University, Jaipur

Academic and Administrative Management

Prof. (Dr.) Naresh Dadhich
Vice-Chancellor
Vardhaman Mahaveer Open University,
Kota

Prof. (Dr.) M.K. Ghadoliya
Director (Academic)
Vardhaman Mahaveer Open University
Kota

Mr. Yogendra Goyal
Incharge
M.P.&D.

Course Material Production

Mr. Yogendra Goyal
Assistant Production Officer
Vardhaman Mahaveer Open University
Kota

Production : Feb. 2011, **ISBN No. : 13/978-81-8496-265-9**

All rights reserved. No part of this book may be reproduced in any form by mimeograph or any other means, without permission in writing from the V.M. Open University, Kota.

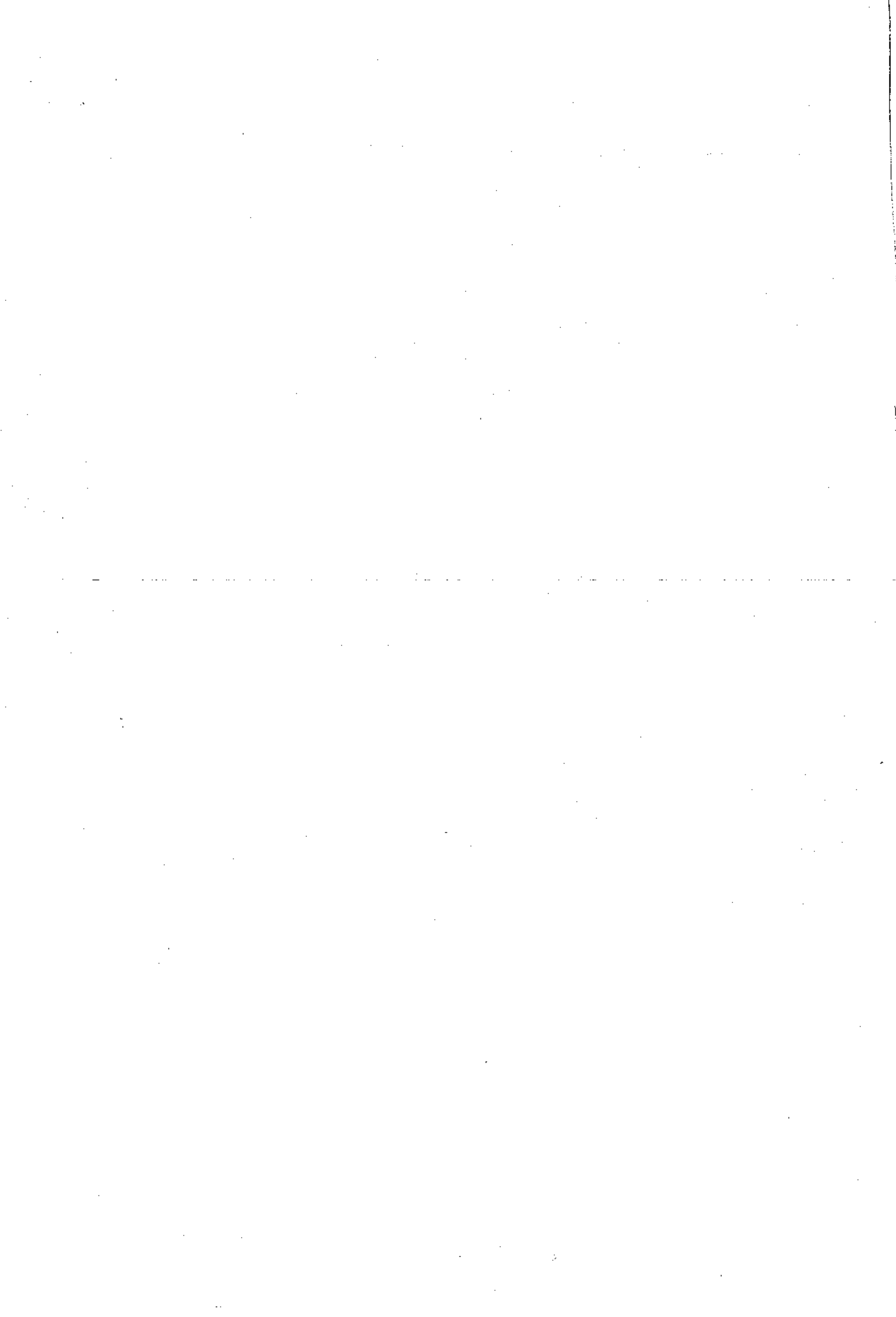
Printed and published on behalf of Registrar V. M. Open University, Kota.
Printed By : Kanchan Offset Printers, Kota. Qty. 500 Books

BCA-06

VARDHAMAN MAHAVEER OPEN UNIVERSITY, KOTA

Programming in C

Unit Number	Units	Page Number
Unit 1	Introducton to programming Language	1 - 15
Unit 2	Data Types and Keywords	16 - 28
Unit 3	Input and Output	29 -36
Unit 4	Decision Making & Conditions	37 - 51
Unit 5	Looping	52 - 60
Unit 6	Arrays	61 - 73
Unit 7	Introduction to Functions	74 - 85
Unit 8	Structures & Unions	86 - 100
Unit 9	Pointers	101 - 114
Unit 10	C Preprocessor	115 - 137
Unit 11	File Handling	138 - 168
Unit 12	Additional features of C	169 - 194



Unit 1 : Introduction

1.1 Introduction

1.2 Low Level Language

1.2.1 First Generation

1.2.2 Second Generation

1.2.3 Examples of Low Level Language

1.3 Middle Level Language

1.4 High Level Language

1.5 Algorithm

1.5.1 Expressing Algorithms

1.5.2 Computer Algorithms

1.5.3 Implementation

1.6 Flowchart

1.6.1 History

1.6.2 Flowchart Building Blocks

1.6.3 Types of Flowchart

After completing this chapter you will be able to understand the following:

1. Introduction
2. Low Level Programming Language
3. Middle Level Programming Language
4. High Level Programming Language
5. Algorithm
6. Flowcharts

1.1 Introduction

The first large computers, such as the ENIAC, incorporated logic that was part of the circuitry; therefore, they could only be "programmed" to a small extent by changing wiring. These were followed by programmable computers with von Neumann architecture. Instructions were independent of the circuitry. Early programs were entirely written in machine language; that is, each instruction gave the central processing unit (CPU) a specific task to do, typically one such as "move a word from one specified location to another" or "change a bit in a specified location from a 0 to a 1 unless 1 already." During the 1940s and the early 1950s, all computers were programmed in this kind of machine code. Such programming was a difficult and tedious task, requiring good knowledge of how the hardware of a computer works.

The introduction of interpreters was a major breakthrough. Interpreters translate a program written in an easier programming language into machine language. Particular locations in the CPU did not have to

be specified, and commands were more closely connected to operations. A big disadvantage of such an interpreter is that it translates the program line by line into machine language, slowing down execution.

A compiler offers a better method for translating programming language into machine language. A compiler translates a whole program into machine language first. When the compiled program is run on the computer, execution time is shortened since there is no translation step going on. Grace Murray Hopper developed one of the first such compilers during the early 1950s.

During World War II, Hopper had been working with Howard Aiken at the Bureau of Ordnance Computation Project at Harvard. Aiken had built the Mark I computer, a huge electromechanical calculator. Hopper designed programs for that computer so that it performed complex calculations for the military. In 1945 she and the team at Harvard started developing Mark II. It was in this computer that a moth, in the words of Hopper, "had been beaten to death" by a relay, stopping the computer. The moth ended up Scotch-taped in their lab logbook, with the note "First actual case of bug being found." Even then the word bug denoted something that caused the computer to fail or to produce incorrect calculations.

Hopper created the first compiler for UNIVAC. Hopper's compiler turned programs written in simple English into machine code. Saying that she was going to communicate with a computer in plain English produced general disbelief. Managers had accepted that computers understand numbers; but computers that understand English seemed inconceivable. Although it made the work of programmers much easier, convincing them to use the compiler was not an easy task. They had gotten used to writing machine code. Hopper developed distaste for the phrase "but we've always done it that way."

During the 1950s several experimental and mostly mathematically oriented languages appeared. The first one that gathered wide success was FORTRAN (FORmula TRANslator). Developed in 1956, it underwent several changes and is still used today in technical and scientific applications.

Several other languages with different orientations appeared. One of them, COBOL, was developed in 1959 by Grace Hopper for business applications. BASIC, a programming language developed by the mathematicians John Kemeny and Thomas Kurtz at Dartmouth University in 1965, was first used as an educational tool, but became in the late 1970s the most popular language for the personal computer.

BASIC, and to a lesser degree FORTRAN, was often criticized because of the use of loops and branches. These produced the inconvenience that when a program grew to any length, it became more and more entangled, a phenomenon called spaghetti. A minor change anywhere in the program could have uncontrollable repercussions in the whole program, requiring tedious searches and rewriting, a process called debugging.

Languages that allow structured programming were a solution to the spaghetti problem. ALGOL, developed by an international committee, and Pascal, written by Niklaus Wirth, are such languages. These languages require that the coding be organized in logical groups, making reading and amending programs much easier.

1.2 Low Level Language

In computer science, a **low-level programming language** is a programming language that provides little or no abstraction from a computer's instruction set architecture. The word "low" refers to the small or nonexistent amount of abstraction between the language and machine language; because of this, low-level languages are sometimes described as being "close to the hardware."

A low-level language does not need a compiler or interpreter to run; the processor for which the language was written is able to run the code without using either of these.

By comparison, a high-level programming language isolates the execution semantics of a computer architecture from the specification of the program, making the process of developing a program simpler and more understandable.

Low-level programming languages are sometimes divided into two categories: *first generation*, and *second generation*.

1.2.1 First Generation

The first-generation programming language, or *1GL*, is machine code. It is the only language a microprocessor can process directly without a previous transformation. Currently, programmers almost never write programs directly in machine code, because it requires attention to numerous details which a high-level language would handle automatically, and also requires memorizing or looking up numerical codes for every instruction that is used. For this reason, second generation programming languages provide one abstraction level on top of the machine code.

Example: A function in 32-bit x86 machine code to calculate the *n*th Fibonacci number:

```
8B542408 83FA0077 06B80000 0000C383
FA027706 B8010000 00C353BB 01000000
B9010000 008D0419 83FA0376 078BD98B
C84AEBF1 5BC3
```

1.2.2 Second generation

The second-generation programming language, or *2GL*, is assembly language. It is considered a second-generation language because while it is not a microprocessor's native language, an assembly language programmer must still understand the microprocessor's unique architecture (such as its *registers* and *instructions*). These simple instructions are then assembled directly into machine code. The assembly code can also be abstracted to another layer in a similar manner as machine code is abstracted into assembly code.

Example: The same Fibonacci number calculator as above, but in x86 assembly language using MASM syntax:

```
fib:
  mov edx, [esp+8]
  cmp edx, 0
  ja @f
  mov eax, 0
  ret

  @@:
  cmp edx, 2
  ja @f
  mov eax, 1
  ret

  @@:
  push ebx
  mov ebx, 1
  mov ecx, 1
```

```
@@:
  lea eax, [ebx+ecx]
  cmp edx, 3
  jbe @f
  mov ebx, ecx
  mov ecx, eax
  dec edx
jmp @b

@@:
pop ebx
ret
```

1.2.3 Examples of Low Level Language

The "machine language" and "assembly language" for each CPU architecture are the lowest-level programming languages. . .. .

The "Forth language" and the "C programming language" are perhaps the most popular non-CPU-specific low-level programming languages. They were once considered high-level programming languages, and certainly they are at a higher level than assembly language, but now they are considered low-level programming languages when compared to the much higher-level languages available today (Python, Java, C++, etc).

Low-level programming languages provide little or no abstraction from the CPU's instruction set architecture; typically they interact with the hardware directly.

1.3 Middle Level Language

Language which supports inline assembly language programs as well as high level language features is Middle Level Language. Middle Level Language are closely related to machine as well as human being. It is more user friendly as compared to previously available low level language.

Examples of Middle Level Language are C, C++. PHP etc.

1.4 High Level Language

A **high-level programming language** is a programming language with strong abstraction from the details of the computer. In comparison to low-level programming languages, it may use natural language elements, be easier to use, or be more portable across platforms. Such languages hide the details of CPU operations such as memory access models and management of scope.

Starting a development effort at a high level of abstraction often leads to shorter development time since it retains opportunities to specialize the design, e.g. to adapt it to unforeseen insights into the Application Domain or to incorporate changing requirements. It is usually harder to generalize a specific design than to specialize or extend the design, since unanticipated generalization may force the re-examination of many existing relations between the constituents of the system to identify hard-coded design decisions that may have been invalidated. However see Premature Generalization.

Working at a high level of abstraction does not necessarily preclude runtime efficiency of the implementation. Many modern languages (e.g. CeePlusPlus, CommonLisp, AdaLanguage) aim to provide access to low level elements of the implementation while retaining means to develop at a high level of abstraction. Most language implementations allow to *drop out* to a different language to implement specific parts of a system at a lower level of abstraction.

The term "High Level Language" was originally used to distinguish things like Fortran Language from things like assembly language. Therefore, originally "high level language" very much included Fortran, Basic, COBOL, PL/I, and a little later, C.

Observing that such languages are not very high level compared with e.g. Prolog, YACC, Lex, ML, Haskell, etc, some people started calling the older high level languages "low level languages", or qualifying them as "higher level languages", etc. This is is often erroneously thought to be revisionism but is the very basis of much of Computer Science, and such terminology while not universally accepted among all programmers, is at least understood by those with a broad understanding of the relevant foundations of the topic at hand.

This greater abstraction and hiding of details is generally intended to make the language user-friendly, as it includes concepts from the problem domain instead of those of the machine used. A high-level language isolates the execution semantics of a computer architecture from the specification of the program, making the process of developing a program simpler and more understandable with respect to a low-level language. The amount of abstraction provided defines how "high-level" a programming language is.

The first high-level programming language to be designed for a computer was Plankalkül, created by Konrad Zuse. However, it was not implemented in his time and his original contributions were isolated from other developments.

Features

The term "high-level language" does not imply that the language is superior to low-level programming languages—in fact, in terms of the depth of knowledge of how computers work required to productively program in a given language, the inverse may be true. Rather, "high-level language" refers to the higher level of abstraction from machine language. Rather than dealing with registers, memory addresses and call stacks, high-level languages deal with variables, arrays, objects, complex arithmetic or boolean expressions, subroutines and functions, loops, threads, locks, and other abstract computer science concepts, with a focus on usability over optimal program efficiency. Unlike low-level assembly languages, high-level languages have few, if any, language elements that translate directly into a machine's native codes. Other features such as string handling routines, object-oriented language features and file input/output may also be present.

Abstraction penalty

Stereotypically, high-level languages make complex programming simpler, while low-level languages tend to produce more efficient code. *Abstraction penalty* is the barrier that prevents high-level programming techniques from being applied in situations where computational resources are limited. High-level programming features like more generic data structures, run-time interpretation and intermediate code files often result in slower execution speed, higher memory consumption and larger binary program size. For this reason, code which needs to run particularly quickly and efficiently may require the use of a lower-level language, even if a higher-level language would make the coding easier.

However, with the growing complexity of modern microprocessor architectures, well-designed compilers for high-level languages frequently produce code comparable in efficiency to what most low-level programmers can produce by hand, and the higher abstraction may allow for more powerful techniques providing better overall results than their low-level counterparts in particular settings.

Relative meaning

The terms *high-level* and *low-level* are inherently relative. Some decades ago, the C language, and similar languages, were most often considered "high-level", as it supported concepts such as expression evaluation, parameterised recursive functions, and data types and structures, while assembly language was considered "low-level". Many programmers today might refer to C as low-level, as it lacks a large runtime-

system (no garbage collection etc.), basically supports only scalar operations, and provides direct memory addressing. It therefore readily blends with assembly language and the machine level of CPUs and microcontrollers.

Also note that assembly language may itself be regarded as a higher level (but often still one-to-one if used without macros) representation of machine code, as it supports concepts such as constants and (limited) expressions, sometimes even variables, procedures, and data structures. Machine code, in its turn, is inherently at a slightly higher level than the microcode or micro-operations used internally in many processors.

Execution models

There are three models of execution for modern high-level languages:

Interpreted

Interpreted languages are read and then executed directly, with no compilation stage. A program called an *interpreter* reads the program line by line and executes the lines as they are read.

Compiled

Compiled languages are transformed into an executable form before running. There are two types of compilation;

Machine code generation

Some compilers compile source code directly into machine code. This is the original mode of compilation, and languages that are directly and completely transformed to machine-native code in this way may be called "truly compiled" languages.

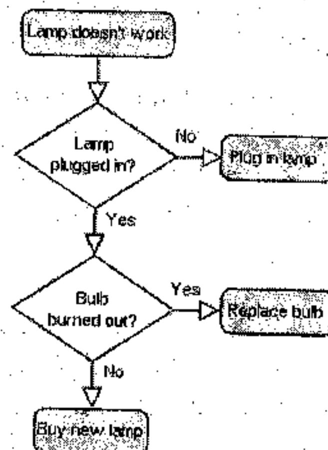
Intermediate representations

When a language is compiled to an intermediate representation, that representation can be optimized or saved for later execution without the need to re-read the source file. When the intermediate representation is saved it is often represented as byte code. The intermediate representation must then be interpreted or further compiled to execute it. Virtual machines that execute byte code directly or transform it further into machine code have blurred the once clear distinction between intermediate representations and truly compiled languages.

Translated

A language may be translated into a low-level programming language for which native code compilers are already widely available. The C programming language is a common target for such translators.

1.5 Algorithm



This is an algorithm that tries to figure out why the lamp doesn't turn on and tries to fix it using the steps. Flowcharts are often used to represent algorithms graphically.

In mathematics, computer science, and related subjects, an **algorithm** is an effective method for solving a problem expressed as a fi1nite sequence of steps. Algorithms are used for calculation, data processing, and many other fields. (In more advanced or abstract settings, the instructions do not necessarily constitute a finite sequence, and even not necessarily a sequence; see, e.g., "nondeterministic algorithm".)

Each algorithm is a list of well-defined instructions for completing a task. Starting from an initial state, the instructions describe a computation that proceeds through a well-defined series of successive states, eventually terminating in a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate randomness.

The adjective "continuous" when applied to the word "algorithm" can mean:

1). An algorithm operating on data that represents continuous quantities, even though this data is represented by discrete approximations – such algorithms are studied in numerical analysis;

or

2) An algorithm in the form of a differential equation that operates continuously on the data, running on an analog computer

Algorithms are essential to the way computers process information. Many computer programs contain algorithms that specify the specific instructions a computer should perform (in a specific order) to carry out a specified task, such as calculating employees' paychecks or printing students' report cards. Thus, an algorithm can be considered to be any sequence of operations that can be simulated by a Turing-complete system.

Typically, when an algorithm is associated with processing information, data is read from an input source, written to an output device, and/or stored for further processing. Stored data is regarded as part of the internal state of the entity performing the algorithm. In practice, the state is stored in one or more data structures.

For some such computational process, the algorithm must be rigorously defined specified in the way it applies in all possible circumstances that could arise. That is, any conditional steps must be systematically dealt with, case-by-case; the criteria for each case must be clear (and computable).

Because an algorithm is a precise list of precise steps, the order of computation will always be critical to the functioning of the algorithm. Instructions are usually assumed to be listed explicitly, and are described as starting "from the top" and going "down to the bottom", an idea that is described more formally by *flow of control*.

So far, this discussion of the formalization of an algorithm has assumed the premises of imperative programming. This is the most common conception, and it attempts to describe a task in discrete, "mechanical" means. Unique to this conception of formalized algorithms is the assignment operation, setting the value of a variable. It derives from the intuition of "memory" as a scratchpad. There is an example below of such an assignment.

1.5.1 Expressing Algorithms

Algorithms can be expressed in many kinds of notation, including natural languages, pseudocode, flowcharts, programming languages or control tables (processed by interpreters). Natural language expressions of algorithms tend to be verbose and ambiguous, and are rarely used for complex or technical algorithms. Pseudocode, flowcharts and control tables are structured ways to express algorithms that avoid many of the ambiguities common in natural language statements, while remaining independent of a

particular implementation language. Programming languages are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used as a way to define or document algorithms.

There is a wide variety of representations possible and one can express a given Turing machine program as a sequence of machine tables, as flowcharts, or as a form of rudimentary machine code or assembly code called "sets of quadruples".

Sometimes it is helpful in the description of an algorithm to supplement small "flow charts" (state diagrams) with natural-language and/or arithmetic expressions written inside "block diagrams" to summarize what the "flow charts" are accomplishing.

Representations of algorithms are generally classed into three accepted levels of Turing machine description:

- **1 High-level description:**

  "...prose to describe an algorithm, ignoring the implementation details. At this level we do not need to mention how the machine manages its tape or head."

- **2 Implementation description:**

  "...prose used to define the way the Turing machine uses its head and the way that it stores data on its tape. At this level we do not give details of states or transition function."

- **3 Formal description:**

  Most detailed, "lowest level", gives the Turing machine's "state table".

1.5.2 Computer Algorithms

In computer systems, an algorithm is basically an instance of logic written in software by software developers to be effective for the intended "target" computer(s), in order for the software on the target machines to *do something*. For instance, if a person is writing software that is supposed to print out a PDF document located at the operating system folder "/My Documents" at computer drive "D:" every Friday at 10 pm, they will write an algorithm that specifies the following actions: "If today's date (computer time) is 'Friday', open the document at 'D:/My Documents' and call the 'print' function". While this simple algorithm does not look into whether the printer has enough paper or whether the document has been moved into a different location, one can make this algorithm more robust and anticipate these problems by rewriting it as a formal CASE statement or as a (carefully crafted) sequence of IF-THEN-ELSE statements. For example the CASE statement might appear as follows (there are other possibilities):

CASE 1: IF today's date is NOT Friday THEN *exit this CASE instruction* ELSE

CASE 2: IF today's date is Friday AND the document is located at 'D:/My Documents' AND there is paper in the printer THEN print the document (and *exit this CASE instruction*) ELSE

CASE 3: IF today's date is Friday AND the document is NOT located at 'D:/My Documents' THEN display 'document not found' error message (and *exit this CASE instruction*).ELSE

CASE 4: IF today's date is Friday AND the document is located at 'D:/My Documents' AND there is NO paper in the printer THEN (i) display 'out of paper' error message and (ii) *exit*.

Note that CASE 4 includes two possibilities: (i) the document is NOT located at 'D:/My Documents' AND there's paper in the printer OR (ii) the document is NOT located at 'D:/My Documents' AND there's NO paper in the printer.

The sequence of IF-THEN-ELSE tests might look like this:

TEST 1: IF today's date is NOT Friday THEN *done* ELSE TEST 2:

TEST 2: IF the document is NOT located at 'D:/My Documents' THEN display 'document not found' error message ELSE TEST 3:

TEST 3: IF there is NO paper in the printer THEN display 'out of paper' error message ELSE print the document.

These examples' logic grants precedence to the instance of "NO document at 'D:/My Documents'". Also observe that in a well-crafted CASE statement or sequence of IF-THEN-ELSE statements the number of distinct actions—4 in these examples: do nothing, print the document, display 'document not found', display 'out of paper' – equals the number of cases.

Given unlimited memory, a computational machine with the ability to execute either a set of CASE statements or a sequence of IF-THEN-ELSE statements is Turing complete. Therefore, anything that is computable can be computed by this machine. This form of algorithm is fundamental to computer programming in all its forms.

Implementation

Most algorithms are intended to be implemented as computer programs. However, algorithms are also implemented by other means, such as in a biological neural network (for example, the human brain implementing arithmetic or an insect looking for food), in an electrical circuit, or in a mechanical device.

Example

An animation of the quicksort algorithm sorting an array of randomized values. The red bars mark the pivot element; at the start of the animation, the element farthest to the right hand side is chosen as the pivot.

One of the simplest algorithms is to find the largest number in an (unsorted) list of numbers. The solution necessarily requires looking at every number in the list, but only once at each. From this follows a simple algorithm, which can be stated in a high-level description English prose, as:

High-level description:

1. Assume the first item is largest.
2. Look at each of the remaining items in the list and if it is larger than the largest item so far, make a note of it.
3. The last noted item is the largest in the list when the process is complete.

(Quasi-)formal description: Written in prose but much closer to the high-level language of a computer program, the following is the more formal coding of the algorithm in pseudocode or pidgin code:

Algorithm LargestNumber

Input: A non-empty list of numbers L.

Output: The *largest* number in the list L.

largest •! L_0

for each *item* **in** the list *(Length(L)e"1)*, **do**

if the *item* > *largest*, **then**

largest •! the *item*

return *largest*

- "•!" is a loose shorthand for "changes to". For instance, "*largest* •! *item*" means that the value of *largest* changes to the value of *item*.
- "**return**" terminates the algorithm and outputs the value that follows.

Classification

There are various ways to classify algorithms, each with its own merits.

By implementation

One way to classify algorithms is by implementation means.

- **Recursion** or **iteration**: A recursive algorithm is one that invokes (makes reference to) itself repeatedly until a certain condition matches, which is a method common to functional programming. Iterative algorithms use repetitive constructs like loops and sometimes additional data structures like stacks to solve the given problems. Some problems are naturally suited for one implementation or the other. For example, towers of Hanoi is well understood in recursive implementation. Every recursive version has an equivalent (but possibly more or less complex) iterative version, and vice versa.
- **Logical**: An algorithm may be viewed as controlled logical deduction. This notion may be expressed as: **Algorithm = logic + control**. The logic component expresses the axioms that may be used in the computation and the control component determines the way in which deduction is applied to the axioms. This is the basis for the logic programming paradigm. In pure logic programming languages the control component is fixed and algorithms are specified by supplying only the logic component. The appeal of this approach is the elegant semantics: a change in the axioms has a well defined change in the algorithm.
- **Serial** or **parallel** or **distributed**: Algorithms are usually discussed with the assumption that computers execute one instruction of an algorithm at a time. Those computers are sometimes called serial computers. An algorithm designed for such an environment is called a serial algorithm, as opposed to parallel algorithms or distributed algorithms. Parallel algorithms take advantage of computer architectures where several processors can work on a problem at the same time, whereas distributed algorithms utilize multiple machines connected with a network. Parallel or distributed algorithms divide the problem into more symmetrical or asymmetrical subproblems and collect the results back together. The resource consumption in such algorithms is not only processor cycles on each processor but also the communication overhead between the processors. Sorting algorithms can be parallelized efficiently, but their communication overhead is expensive. Iterative algorithms are generally parallelizable. Some problems have no parallel algorithms, and are called inherently serial problems.
- **Deterministic** or **non-deterministic**: Deterministic algorithms solve the problem with exact decision at every step of the algorithm whereas non-deterministic algorithms solve problems via guessing although typical guesses are made more accurate through the use of heuristics.
- **Exact** or **approximate**: While many algorithms reach an exact solution, approximation algorithms seek an approximation that is close to the true solution. Approximation may use either a deterministic or a random strategy. Such algorithms have practical value for many hard problems.

By design paradigm

Another way of classifying algorithms is by their design methodology or paradigm. There is a certain number of paradigms, each different from the other. Furthermore, each of these categories will include many different types of algorithms. Some commonly found paradigms include:

- **Brute-force** or **exhaustive search**. This is the naïve method of trying every possible solution to see which is best.

- **Divide and conquer**. A divide and conquer algorithm repeatedly reduces an instance of a problem to one or more smaller instances of the same problem (usually recursively) until the instances are small enough to solve easily. One such example of divide and conquer is merge sorting. Sorting can be done on each segment of data after dividing data into segments and sorting of entire data can be obtained in the conquer phase by merging the segments. A simpler variant of divide and conquer is called a **decrease and conquer algorithm**, that solves an identical subproblem and uses the solution of this subproblem to solve the bigger problem. Divide and conquer divides the problem into multiple subproblems and so the conquer stage will be more complex than decrease and conquer algorithms. An example of decrease and conquer algorithm is the binary search algorithm.

- **Dynamic programming**. When a problem shows optimal substructure, meaning the optimal solution to a problem can be constructed from optimal solutions to subproblems, and overlapping subproblems, meaning the same subproblems are used to solve many different problem instances, a quicker approach called *dynamic programming* avoids recomputing solutions that have already been computed. For example, Floyd–Warshall algorithm, the shortest path to a goal from a vertex in a weighted graph can be found by using the shortest path to the goal from all adjacent vertices. Dynamic programming and memoization go together. The main difference between dynamic programming and divide and conquer is that subproblems are more or less independent in divide and conquer, whereas subproblems overlap in dynamic programming. The difference between dynamic programming and straightforward recursion is in caching or memoization of recursive calls. When subproblems are independent and there is no repetition, memoization does not help; hence dynamic programming is not a solution for all complex problems. By using memoization or maintaining a table of subproblems already solved, dynamic programming reduces the exponential nature of many problems to polynomial complexity.

- **The greedy method**. A greedy algorithm is similar to a dynamic programming algorithm, but the difference is that solutions to the subproblems do not have to be known at each stage; instead a "greedy" choice can be made of what looks best for the moment. The greedy method extends the solution with the best possible decision (not all feasible decisions) at an algorithmic stage based on the current local optimum and the best decision (not all possible decisions) made in a previous stage. It is not exhaustive, and does not give accurate answer to many problems. But when it works, it will be the fastest method. The most popular greedy algorithm is finding the minimal spanning tree as given by Huffman Tree, Kruskal, Prim, Sollin.

- **Linear programming**. When solving a problem using linear programming, specific inequalities involving the inputs are found and then an attempt is made to maximize (or minimize) some linear function of the inputs. Many problems (such as the maximum flow for directed graphs) can be stated in a linear programming way, and then be solved by a 'generic' algorithm such as the simplex algorithm. A more complex variant of linear programming is called integer programming, where the solution space is restricted to the integers.

- **Reduction**. This technique involves solving a difficult problem by transforming it into a better known problem for which we have (hopefully) asymptotically optimal algorithms. The goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithm's. For example, one selection algorithm for finding the median in an unsorted list involves first sorting the list (the expensive portion) and then pulling out the middle element in the sorted list (the cheap portion). This technique is also known as *transform and conquer*.

- **Search and enumeration**. Many problems (such as playing chess) can be modeled as problems on graphs. A graph exploration algorithm specifies rules for moving around a graph and is useful for

such problems. This category also includes search algorithms, branch and bound enumeration and backtracking.

1.6 Flowchart

A **flowchart** is a type of diagram, that represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows. This diagrammatic representation can give a step-by-step solution to a given problem. Data is represented in these boxes, and arrows connecting them represent flow / direction of flow of data. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.

1.6.1 History

The first structured method for documenting process flow, the "flow process chart", was introduced by Frank Gilbreth to members of the American Society of Mechanical Engineers (ASME) in 1921 in the presentation "Process Charts—First Steps in Finding the One Best Way". Gilbreth's tools quickly found their way into industrial engineering curricula. In the early 1930s, an industrial engineer, Allan H. Mogensen began training business people in the use of some of the tools of industrial engineering at his Work Simplification Conferences in Lake Placid, New York.

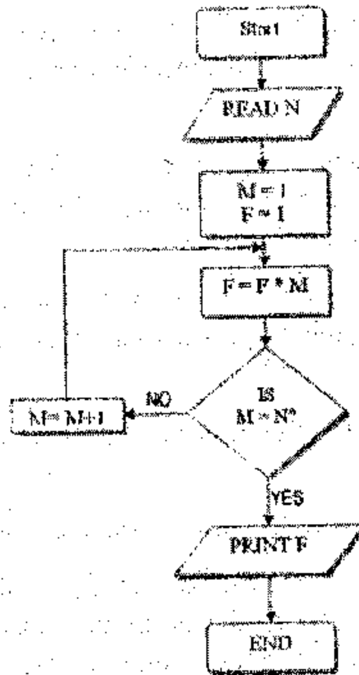
A 1944 graduate of Mogensen's class, Art Spinanger, took the tools back to Procter and Gamble where he developed their Deliberate Methods Change Program. Another 1944 graduate, Ben S. Graham, Director of Formcraft Engineering at Standard Register Corporation, adapted the flow process chart to information processing with his development of the multi-flow process chart to display multiple documents and their relationships. In 1947, ASME adopted a symbol set derived from Gilbreth's original work as the ASME Standard for Process Charts by Mishad, Ramsan & Raiaan.

Douglas Hartree explains that Herman Goldstine and John von Neumann developed the flow chart (originally, diagram) to plan computer programs. His contemporary account is endorsed by IBM engineersand by Goldstine's personal recollections The original programming flow charts of Goldstine and von Neumann can be seen in their unpublished report, "Planning and coding of problems for an electronic computing instrument, Part II, Volume 1" (1947), which is reproduced in von Neumann's collected works.

Flowcharts used to be a popular means for describing computer algorithms and are still used for this purpose. Modern techniques such as UML activity diagrams can be considered to be extensions of the flowchart. In the 1970s the popularity of flowcharts as an own method decreased when interactive computer terminals and third-generation programming languages became the common tools of the trade, since algorithms can be expressed much more concisely and readably as source code in such a language. Often pseudo-code is used, which uses the common idioms of such languages without strictly adhering to the details of a particular one.

1.6.2 Flowchart Building Blocks

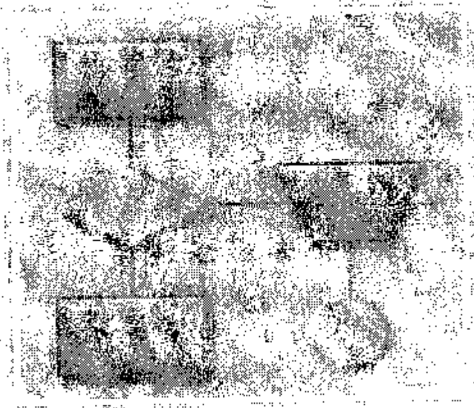
Examples



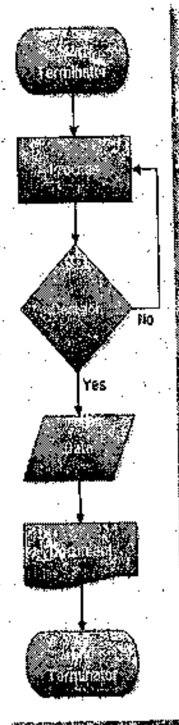
A simple flowchart for computing factorial N (10!)

A flowchart for computing factorial N (10!) where N! = (1*2*3*4*5*6*7*8*9*10), see image. This flowchart represents a "loop and a half" — a situation discussed in introductory programming textbooks that requires either a duplication of a component (to be both inside and outside the loop) or the component to be put inside a branch in the loop.

Basic Flowchart Symbols



Basic Flowchart



Flowcharts are the ideal diagrams for visually representing business processes. For example, if you need to show the flow of a custom-order process through various departments within your organization, you can use a flowchart. This paper provides a visual representation of basic flowchart symbols and their proposed usage in communicating the structure of a well-developed web site, as well as their correlation in developing on-line instructional projects. A typical flowchart from older Computer Science textbooks may have the following kinds of symbols:

- **Start** and **end** symbols, represented as lozenges, ovals or rounded rectangles, usually containing the word "Start" or "End", or another phrase signaling the start or end of a process, such as "submit enquiry" or "receive product".
- **Arrows**, showing what's called "flow of control" in computer science. An arrow coming from one symbol and ending at another symbol represents that control passes to the symbol the arrow points to.
- **Processing steps**, represented as rectangles. Examples: "Add 1 to X"; "replace identified part"; "save changes" or similar.
- **Input/Output**, represented as a parallelogram. Examples: Get X from the user; display X.
- **Conditional** (or **decision**), represented as a diamond (rhombus). These typically contain a Yes/No question or True/False test. This symbol is unique in that it has two arrows coming out of it, usually from the bottom point and right point, one corresponding to Yes or True, and one corresponding to No or False. The arrows should always be labeled. More than two arrows can be used, but this is normally a clear indicator that a complex decision is being taken, in which case it may need to be broken-down further, or replaced with the "pre-defined process" symbol.
- A number of other symbols that have less universal currency, such as:
  - A **Document** represented as a rectangle with a wavy base;
  - A **Manual input** represented by rectangle, with the top irregularly sloping up from left to right. An example would be to signify data-entry from a form;
  - A **Manual operation** represented by a trapezoid with the longest parallel side upmost, to represent an operation or adjustment to process that can only be made manually.
  - A **Data File** represented by a cylinder

Flowcharts may contain other symbols, such as connectors, usually represented as circles, to represent converging paths in the flowchart. Circles will have more than one arrow coming into them but only one going out. Some flowcharts may just have an arrow point to another arrow instead. These are useful to represent an iterative process (what in Computer Science is called a loop). A loop may, for example, consist of a connector where control first enters, processing steps, a conditional with one arrow exiting the loop, and one going back to the connector. Off-page connectors are often used to signify a connection to a (part of another) process held on another sheet or screen. It is important to remember to keep these connections logical in order. All processes should flow from top to bottom and left to right.

1.6.3 Types of flowchart

Sterneckert (2003) suggested that flowcharts can be modelled from the perspective of different user groups (such as managers, system analysts and clerks) and that there are four general types:

- *Document flowcharts*, showing controls over a document-flow through a system
- *Data flowcharts*, showing controls over a data flows in a system

- *System flowcharts* showing controls at a physical or resource level
- *Program flowchart*, showing the controls in a program within a system

Notice that every type of flowchart focusses on some kind of control, rather than on the particular flow itself.

However there are several of these classifications. For example Andrew Veronis (1978) named three basic types of flowcharts: the *system flowchart*, the *general flowchart*, and the *detailed flowchart*. That same year Marilyn Bohl (1978) stated "in practice, two kinds of flowcharts are used in solution planning: *system flowcharts* and *program flowcharts*...". More recently Mark A. Fryman (2001) stated that there are more differences: "Decision flowcharts, logic flowcharts, systems flowcharts, product flowcharts, and process flowcharts are just a few of the different types of flowcharts that are used in business and government"

Summery

In this unit we have learnt about the basic programming languages structures and their development. We have discussed regarding the development of programming languages from Low level to Middle Level to High level programming languages. We have also discussed regarding the use of algorithms in computer programming. After that we have learnt about the use of flow charts in the programming. We also have learnt that what are the symbols that are used in flow charts and how to prepare flow chargs for the programming.

Self Assignments

1. What do you mean by programming language?
2. What is Low level programming language?
3. Name few low level programming language?
4. What is Middle level programming language?
5. Why High level programing language is more use these days?
6. What is the use of algorithms in computer programming?
7. Explain the symbols used in flow charts.
8. Prepare a flow chart to substract higher number from smaller number.

Unit 2: Data Types and Keywords

2.1 Introduction

 2.1.1 First example

 2.1.2 Second example

2.2 Basic Data Types

 2.2.1 Types

 2.2.2 Constants

 2.2.3 Declarations

 2.2.4 Variable Names

2.3 C Keywords

2.4 C Standard Library (Header files)

After completing this unit you would be able to understand the following points We will learn following topics in this unit:

- Introduction to C Programming
- Basic Data Types
- Keywords
- Header Files

2.1 Introduction

C – Programming Language is a relatively small language, but one which wears well. C's small, unambitious feature set is a real advantage. There's less to learn. There isn't excess baggage in the way when you don't need it. It can also be a disadvantage, since it doesn't do everything for you, there's a lot you have to do yourself. (Actually, this is viewed by many as an additional advantage: anything the language doesn't do for you, it doesn't dictate to you, either, so you're free to do that something however you want.)

C is sometimes referred to as a "High-Level Assembly Language". Some people think that's an insult, but it's actually a deliberate and significant aspect of the language. If you have programmed in assembly language, you'll probably find C very natural and comfortable (although if you continue to focus too heavily on machine-level details, you'll probably end up with unnecessarily nonportable programs). If you haven't programmed in assembly language, you may be frustrated by C's lack of certain higher-level features. In either case, you should understand why C was designed this way so that seemingly simple constructions expressed in C would not expand to arbitrarily expensive (in time or space) machine language constructions when compiled. If you write a C program simply and succinctly, it is likely to result in a succinct, efficient machine language executable. If you find that the executable program resulting from a C program is not efficient, it's probably because of something silly you did, not because of something the compiler did behind your back which you have no control over. In any case, there's no point in complaining about C's low-level flavor C is what it is.

A programming language is a tool, and no tool can perform every task unaided. If you're building a house, and I'm teaching you how to use a hammer, and you ask how to assemble rafters and trusses into gables, that's a legitimate question, but the answer has fallen out of the realm of "How do I use a hammer?"

and into "How do I build a house?". In the same way, we'll see that C does not have built-in features to perform every function that we might ever need to do while programming.

As mentioned above, C imposes relatively few built-in ways of doing things on the programmer. Some common tasks, such as manipulating strings, allocating memory, and doing input/output (I/O), are performed by calling on library functions. Other tasks which you might want to do, such as creating or listing directories, or interacting with a mouse, or displaying windows or other user-interface elements, or doing color graphics, are not defined by the C language at all. You can do these things from a C program, of course, but you will be calling on services which are peculiar to your programming environment (compiler, processor, and operating system) and which are not defined by the C standard. Since this course is about portable C programming, it will also be steering clear of facilities not provided in all C environments.

C does not, in general, try hard to protect a programmer from mistakes. If you write a piece of code which will (through some oversight of yours) do something wildly different from what you intended it to do, up to and including deleting your data or trashing your disk, and if it is possible for the compiler to compile it, it generally will. You won't get warnings of the form "Do you really mean to...?" or "Are you sure you really want to...?". C is often compared to a sharp knife: it can do a surgically precise job on some exacting task you have in mind, but it can also do a surgically precise job of cutting off your finger. It's up to you to use it carefully.

This aspect of C is very widely criticized it is also used (justifiably) to argue that C is not a good teaching language because it means that C does not try to protect them from themselves when they know what they're doing, even if it's risky or obscure, they can do it. Students of C hate this aspect of C because it often seems as if the language is some kind of a conspiracy specifically designed to lead them into booby traps and "gotcha!s".

This is another aspect of the language, which it's fairly pointless to complain about. If you take care and pay attention, you can avoid many of the pitfalls. These notes will point out many of the obvious (and not so obvious) trouble spots.

2.1.1 First Example

The best way to learn programming is to dive right in and start writing real programs. This way, concepts, which would otherwise seem abstract, make sense, and the positive feedback you get from getting even a small program to work gives you a great incentive to improve it or write the next one.

Diving in with "real" programs right away has another advantage, if only pragmatic if you're using a conventional compiler, you can't run a fragment of a program and see what it does nothing will run until you have a complete (if tiny or trivial) program. You can't learn everything you'd need to write a complete program all at once, so you'll have to take some things ``on faith'' and parrot them in your first programs before you begin to understand them. (You can't learn to program just one expression or statement at a time any more than you can learn to speak a foreign language one word at a time. If all you know is a handful of words, you can't actually *say* anything: you also need to know something about the language's word order and grammar and sentence structure and declension of articles and verbs.)

Besides the occasional necessity to take things on faith, there is a more serious potential drawback of this "dive in and program" approach it's a small step from learning-by-doing to learning-by-trial-and-error, and when you learn programming by trial-and-error, you can very easily learn many errors. When you're not sure whether something will work, or you're not even sure what you could use that might work, and you try something, and it does work, you do *not* have any guarantee that what you tried worked for the right reason. You might just have "learned" something that works only by accident or only on your compiler, and it may be very hard to un-learn it later, when it stops working.

Therefore, whenever you're not sure of something, be very careful before you go off and try it "just to see if it will work." Of course, you can never be absolutely sure that something is going to work before you try it, otherwise we'd never have to try things. But you should have an expectation that something is going to work before you try it, and if you can't predict how to do something or whether something would work and find yourself having to determine it experimentally, make a note in your mind that whatever you've just learned (based on the outcome of the experiment) is suspect.

The first example program in any language: print or display a simple string, and exit. Here is my version of "hello, world" program:

```
#include <stdio.h>
main()
{
printf("Hello, world!\n");
return 0;
}
```

If you have a C compiler, the first thing to do is figure out how to type this program in and compile it and run it and see where its output went. (If you don't have a C compiler yet, the first thing to do is to find one.)

The first line is practically boilerplate it will appear in almost all programs we write. It asks that some definitions having to do with the "Standard I/O Library" be included in our program these definitions are needed if we are to call the library function printf correctly.

The second line says that we are defining a function named main. Most of the time, we can name our functions anything we want, but the function name main is special: it is the function that will be "called" first when our program starts running. The empty pair of parentheses indicates that our main function accepts no *arguments*, that is, there isn't any information which needs to be passed in when the function is called.

The braces {and} surround a list of statements in C. Here, they surround the list of statements making up the function main.

The line

```
printf("Hello, world!\n");
```

is the first statement in the program. It asks that the function printf be called printf is a library function, which prints formatted output. The parentheses surround printf's argument list the information that is handed to it, which it should act on. The semicolon at the end of the line terminates the statement.

(printf's name reflects the fact that C was first developed when Teletypes and other printing terminals were still in widespread use. Today, of course, video displays are far more common. printf's "prints" to the *standard output*, that is, to the default location for program output to go. Nowadays, that's almost always a video screen or a window on that screen. If you do have a printer, you'll typically have to do something extra to get a program to print to it.)

printf's first (and, in this case, only) argument is the string which it should print. The string, enclosed in double quotes " ", consists of the words "Hello, world!" followed by a special sequence \n. In strings, any two-character sequence beginning with the backslash \ represents a single special character. The sequence \n represents the "new line" character, which prints a carriage return or line feed or whatever it takes to end one line of output and move down to the next. (This program only prints one line of output, but it's still important to terminate it.)

The second line in the main function is

```
return 0;
```

In general, a function may return a value to its caller, and main is no exception. When main returns (that is, reaches its end and stops functioning), the program is at its end, and the return value from main tells the operating system (or whatever invoked the program that main is the main function of) whether it succeeded or not. By convention, a return value of 0 indicates success.

This program may look so absolutely trivial that it seems as if it's not even worth typing it in and trying to run it, but doing so may be a big (and is certainly a vital) first hurdle. On an unfamiliar computer, it can be arbitrarily difficult to figure out how to enter a text file containing program source, or how to compile and link it, or how to invoke it, or what happened after (if?) it run. The most experienced C programmers immediately go back to this one, simple program whenever they're trying out a new system or a new way of entering or building programs or a new way of printing output from within programs. As Kernighan and Ritchie say, everything else is comparatively easy.

How *you* compile and run this (or any) program is a function of the compiler and operating system you're using. The first step is to type it in, exactly as shown this may involve using a text editor to create a file containing the program text. You'll have to give the file a name, and all C compilers (that I've ever heard of) require that files containing C source end with the extension .c. So you might place the program text in a file called hello.c.

The second step is to compile the program. (Strictly speaking, compilation consists of two steps, compilation proper followed by linking, but we can overlook this distinction at first, especially because the compiler often takes care of initiating the linking step automatically.) On many Unix systems, the command to compile a C program from a source file hello.c is

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

You would type this command at the Unix shell prompt, and it requests that the cc (C compiler) program be run, placing its output (i.e. the new executable program it creates) in the file hello, and taking its input (i.e. the source code to be compiled) from the file hello.c.

The third step is to run (execute, invoke) the newly-built hello program. Again on a Unix system, this is done simply by typing the program's name:

```
hello
```

Depending on how your system is set up (in particular, on whether the current directory is searched for executables, based on the PATH variable), you may have to type

```
./hello
```

to indicate that the hello program is in the current directory (as opposed to some "bin" directory full of executable programs, elsewhere).

You may also have your choice of C compilers. On many Unix machines, the cc command is an older compiler which does not recognize modern, ANSI Standard C syntax. An old compiler will accept the simple programs we'll be starting with, but it will not accept most of our later programs. If you find yourself getting baffling compilation errors on programs which you've typed in exactly as they're shown, it probably indicates that you're using an older compiler. On many machines, another compiler called acc or gcc is available, and you'll want to use it, instead. (Both acc and gcc are typically invoked the same as cc; that is, the above cc command would instead be typed, say, gcc -o hello hello.c .)

Don't name your test programs test, because there's already a standard command called test, and you and the command interpreter will get badly confused if you try to replace the system's test command with your own, not least because your own almost certainly does something completely different.)

Under MS-DOS, the compilation procedure is quite similar. The name of the command you type will depend on your compiler (e.g. cl for the Microsoft C compiler, tc or bcc for Borland's Turbo C, etc.). You may have to manually perform the second, linking step, perhaps with a command named link or tlink. The executable file which the compiler/linker creates will have a name ending in .exe (or perhaps .com), but you can still invoke it by typing the base name (e.g. hello). See your compiler documentation for complete details; one of the manuals should contain a demonstration of how to enter, compile, and run a small program that prints some simple output, just as we're trying to describe here.

In an integrated or "visual" progamming environment, such as those on the Macintosh or under various versions of Microsoft Windows, the steps you take to enter, compile, and run a program are somewhat different (and, theoretically, simpler). Typically, there is a way to open a new source window, type source code into it, give it a file name, and add it to the program (or "project") you're building. If necessary, there will be a way to specify what other source files (or "modules") make up the program. Then, there's a button or menu selection that compiles and runs the program, all from within the programming environment. (There will also be a way to create a standalone executable file, which you can run from outside the environment.) In a PC-compatible environment, you may have to choose between creating DOS programs or Windows programs. (If you have troubles pertaining to the printf function, try specifying a target environment of MS-DOS. Supposedly, some compilers which are targeted at Windows environments won't let you call printf, because until you call some fancier functions to request that a window be created, there's no window for printf to print to.) Again, check the introductory or tutorial manual that came with the programming package. It should walk you through the steps necessary to get your first program running.

2.1.2 Second Example

Our second example is of little more practical use than the first, but it introduces a few more programming language elements:

```
#include <stdio.h>
/* print a few numbers, to illustrate a simple loop */
main()
{
int i;
for(i = 0; i < 10; i = i + 1)
        printf("i is %d\n", i);
return 0;
}
```

As before, the line #include <stdio.h> is boilerplate which is necessary since we're calling the printf function, and main() and the pair of braces {} indicate and delineate the function named main we're (again) writing.

The first new line is the line

/* print a few numbers, to illustrate a simple loop */

which is a *comment*. Anything between the characters /* and */ is ignored by the compiler, but may be useful to a person trying to read and understand the program. You can add comments anywhere you want to in the program, to document what the program is, what it does, who wrote it, how it works, what the various functions are for and how *they* work, what the various variables are for, etc.

The second new line, down within the function main, is

int i;

which *declares* that our function will use a variable named i. The variable's type is int, which is a plain integer.

Next, we set up a *loop*:

for(i = 0; i < 10; i = i + 1)

The keyword for indicates that we are setting up a "for loop." A for loop is controlled by three expressions, enclosed in parentheses and separated by semicolons. These expressions say that, in this case, the loop starts by setting i to 0, that it continues as long as i is less than 10, and that after each iteration of the loop, i should be incremented by 1 (that is, have 1 added to its value) details will be discussed in later chapters.

Finally, we have a call to the printf function, as before, but with several differences. First, the call to printf is within the *body* of the for loop. This means that control flow does not pass once through the printf call, but instead that the call is performed as many times as are dictated by the for loop. In this case, printf will be called several times: once when i is 0, once when i is 1, once when i is 2, and so on until i is 9, for a total of 10 times.

A second difference in the printf call is that the string to be printed, "i is %d", contains a percent sign. Whenever printf sees a percent sign, it indicates that printf is not supposed to print the exact text of the string, but is instead supposed to read another one of its arguments to decide what to print. The letter after the percent sign tells it what type of argument to expect and how to print it. In this case, the letter d indicates that printf is to expect an int, and to print it in decimal. Finally, we see that printf is in fact being called with another argument, for a total of two, separated by commas. The second argument is the variable i, which is in fact an int, as required by %d. The effect of all of this is that each time it is called, printf will print a line containing the current value of the variable i:

```
i is 0
i is 1
i is 2
...
```

After several trips through the loop, i will eventually equal 9. After that trip through the loop, the third control expression i = i + 1 will increment its value to 10. The condition i < 10 is no longer true, so no more trips through the loop are taken. Instead, control flow jumps down to the statement following the for loop, which is the return statement. The main function returns, and the program is finished.

2.2 Basic Data Types

The *type* of a variable determines what kinds of values it may take on. An *operator* computes new values out of old ones. An *expression* consists of variables, constants, and operators combined to perform some useful computation. In this chapter, we'll learn about C's basic types, how to write constants and declare variables of these types, and what the basic operators are.

As, "The type of an object determines the set of values it can have and what operations can be performed on it." This is a fairly formal, mathematical definition of what a type is, but it is traditional (and meaningful). There are several implications to remember:

1. The "set of values" is finite. C's int type can not represent *all* of the integers; its float type can not represent *all* floating-point numbers.
2. When you're using an object (that is, a variable) of some type, you may have to remember what values it can take on and what operations you can perform on it. For example, there are several operators which play with the binary (bit-level) representation of integers, but these operators are not meaningful for and may not be applied to floating-point operands.

3. When declaring a new variable and picking a type for it, you have to keep in mind the values and operations you'll be needing.

In other words, picking a type for a variable is not some abstract academic exercise; it's closely connected to the way(s) you'll be using that variable.

2.2.1 Types

2.2.2 Constants

2.2.3 Declarations

2.2.4 Variable Names

2.2.1 Types

There are only a few basic data types in C. The first ones we'll be encountering and using are:

- "Char" a character
- "int" an integer, in the range -32,767 to 32,767
- "long" int a larger integer (up to +-2,147,483,647)
- "float" a floating-point number
- "double" a floating-point number, with more precision and perhaps greater range than float

If you can look at this list of basic types and say to yourself, "Oh, how simple, there are only a few types, I won't have to worry much about choosing among them," you'll have an easy time with declarations. (Some masochists wish that the type system were more complicated so that they could specify more things about each variable, but those of us who would rather not have to specify these extra things each time are glad that we don't have to.)

The ranges listed above for types int and long int are the guaranteed minimum ranges: On some systems, either of these types (or, indeed, any C type) may be able to hold larger values, but a program that depends on extended ranges will not be as portable. Some programmers become obsessed with knowing exactly what the sizes of data objects will be in various situations, and go on to write programs which depend on these exact sizes. Determining or controlling the size of an object is occasionally important, but most of the time we can sidestep size issues and let the compiler do most of the worrying.

(From the ranges listed above, we can determine that type int must be at least 16 bits, and that type long int must be at least 32 bits. But neither of these sizes is exact; many systens have 32-bit ints, and some systems have 64-bit long ints.)

You might wonder how the computer stores characters. The answer involves a *character set*, which is simply a mapping between some set of characters and some set of small numeric codes. Most machines today use the ASCII character set, in which the letter A is represented by the code 65, the ampersand & is represented by the code 38, the digit 1 is represented by the code 49, the space character is represented by the code 32, etc. (Most of the time, of course, you have *no* need to know or even worry about these particular code values. They're automatically translated into the right shapes on the screen or printer when characters are printed out, and they're automatically generated when you type characters on the keyboard. Eventually, though, we'll appreciate, and even take some control over, exactly when these translations—from characters to their numeric codes—are performed.) Character codes are usually small—the largest code value in ASCII is 126, which is the ~ (tilde or circumflex) character. Characters usually fit in a byte, which is usually 8 bits. In C, type char is defined as occupying one byte, so it is usually 8 bits.

Most of the simple variables in most programs are of types int, long int, or double. Typically, we'll use int and double for most purposes, and long int any time we need to hold integer values greater than 32,767. As we'll see, even when we're manipulating individual characters, we'll usually use an int variable,

for reasons to be discussed later. Therefore, we'll rarely use individual variables of type char, although we'll use plenty of arrays of char.

2.2.2 Constants

A ***constant*** is just an immediate, absolute value found in an expression. The simplest constants are decimal integers, e.g. 0, 1, 2, 123 . Occasionally it is useful to specify constants in base 8 or base 16 (octal or hexadecimal); this is done by prefixing an extra 0 (zero) for octal, or 0x for hexadecimal. The constants 100, 0144, and 0x64 all represent the same number. (If you're not using these non-decimal constants, just remember not to use any leading zeroes. If you accidentally write 0123 intending to get one hundred and twenty three, you'll get 83 instead, which is 123 base 8.)

We write constants in decimal, octal, or hexadecimal for our convenience, not the compiler's. The compiler doesn't care. It always converts everything into binary internally, anyway. (There is, however, no good way to specify constants in source code in binary.)

A constant can be forced to be of type long int by suffixing it with the letter L (in upper or lower case, although upper case is strongly recommended, because a lower case l looks too much like the digit 1).

A constant that contains a decimal point or the letter e (or both) is a floating-point constant: 3.14, 10., .01, 123e4, 123.456e7 . The e indicates multiplication by a power of 10. 123.456e7 is 123.456 times 10 to the 7th, or 1,234,560,000. (Floating-point constants are of type double by default.)

We also have constants for specifying characters and strings. (Make sure you understand the difference between a character and a string a character is exactly one character; a string is a set of zero or more characters a string containing one character is distinct from a lone character.) A character constant is simply a single character between single quotes: 'A', '.', '%'. The numeric value of a character constant is, naturally enough, that character's value in the machine's character set. (In ASCII, for example, 'A' has the value 65.)

A *string* is represented in C as a sequence or array of characters. (We'll have more to say about arrays in general, and strings in particular, later.) A string constant is a sequence of zero or more characters enclosed in double quotes: "apple", "hello, world", "this is a test".

Within character and string constants, the backslash character \ is special, and is used to represent characters not easily typed on the keyboard or for various reasons not easily typed in constants. The most common of these "character escapes" are:

Code	**Character**	**Description**
\\	\	Backslash
\'	'	Single Quote
\"	"	Double Quote
\?	?	Question Mark
\0	<NUL>	Binary 0
\a	<BEL>	Bell (Audible alert)
\b	<BS>	Back Space
\f	<FF>	Form Feed
\n	<NL>	New Line

\r	<CR>	Carriage Return
\t	<HT>	Horizontal Tab
\v	<VT>	Vertical Tab

Notes:

- These escape sequences can be used only within character or string literal constants. They each represent a single character.

For example, “he said \ “hi\” “ is a string constant which contains two double quotes, and “ \” is a character constant consisting of a (single) single quote. Notice once again that the character constant ‘A’ is very different from the string constant “A”.

2.2.3 Declarations

Informally, a *variable* (also called an *object*) is a place you can store a value. So that you can refer to it unambiguously, a variable needs a name. You can think of the variables in your program as a set of boxes or cubbyholes, each with a label giving its name; you might imagine that storing a value “in” a variable consists of writing the value on a slip of paper and placing it in the cubbyhole.

A *declaration* tells the compiler the name and type of a variable you'll be using in your program. In its simplest form, a declaration consists of the type, the name of the variable, and a terminating semicolon:

```
char c;
int i;
float f;
```

You can also declare several variables of the same type in one declaration, separating them with commas:

```
int i1, i2;
```

Later we'll see that declarations may also contain *initializers*, *qualifiers* and *storage classes*, and that we can declare *arrays*, *functions*, *pointers*, and other kinds of data structures.

The placement of declarations is significant. You can't place them just anywhere (i.e. they cannot be interspersed with the other statements in your program). They must either be placed at the beginning of a function, or at the beginning of a brace-enclosed block of statements (which we'll learn about in the next chapter), or outside of any function. Furthermore, the placement of a declaration, as well as its storage class, controls several things about its *visibility* and *lifetime*, as we'll see later.

You may wonder *why* variables must be declared before use. There are two reasons:

1. It makes things somewhat easier on the compiler it knows right away what kind of storage to allocate and what code to emit to store and manipulate each variable; it doesn't have to try to intuit the programmer's intentions.

2. It forces a bit of useful discipline on the programmer you cannot introduce variables willy-nilly; you must think about them enough to pick appropriate types for them. (The compiler's error messages to you, telling you that you apparently forgot to declare a variable, are as often helpful as they are a nuisance they're helpful when they tell you that you misspelled a variable, or forgot to think about exactly how you were going to use it).

Although there are a few places where declarations can be omitted (in which case the compiler will assume an implicit declaration), making use of these removes the advantages of reason 2 above, so I recommend always declaring everything explicitly.

Most of the time, I recommend writing one declaration per line. For the most part, the compiler doesn't care what order declarations are in. You can order the declarations alphabetically, or in the order that they're used, or to put related declarations next to each other. Collecting all variables of the same type together on one line essentially orders declarations by type, which isn't a very useful order (it's only slightly more useful than random order).

A declaration for a variable can also contain an initial value. This *initializer* consists of an equals sign and an expression, which is usually a single constant:

```
int i = 1;
int i1 = 10, i2 = 20;
```

2.2.4 Variable Names

Within limits, you can give your variables and functions any names you want. These names (the formal term is "identifiers") consist of letters, numbers, and underscores. For our purposes, names must begin with a letter. Theoretically, names can be as long as you want, but extremely long ones get tedious to type after a while, and the compiler is not required to keep track of extremely long ones perfectly. (What this means is that if you were to name a variable, say, supercalafragalisticespialidocious, the compiler might get lazy and pretend that you'd named it supercalafragalisticespialidocio, such that if you later misspelled it supercalafragalist icespialidociouz, the compiler wouldn't catch your mistake. Nor would the compiler necessarily be able to tell the difference if for some perverse reason you *deliberately* declared a second variable named supercalafragalistice spialidociouz.)

The capitalization of names in C is significant the variable names variable, Variable, and VARIABLE (as well as silly combinations like variAble) are all distinct.

A final restriction on names is that you may not use *keywords* (the words such as int and for which are part of the syntax of the language) as the names of variables or functions (or as identifiers of any kind).

Deep sentence:

- Don't begin variable names with underscore, however, since library routines often use such names.

If you happen to pick a name which "collides" with (is the same as) a name already chosen by a library routine, either your code or the library routine (or both) won't work. Naming issues become very significant in large projects, and problems can be avoided by setting guidelines for who may use which names. One of these guidelines is simply that user code should not use names beginning with an underscore, because these names are (for the most part) "reserved to the implementation" (that is, reserved for use by the compiler and the standard library).

Note that case is significant. Assuming that case is ignored (as it is with some other programming languages and operating systems) can lead to real frustration.

The convention that all-upper-case names are used for symbolic constants (i.e. as created with the #define directive) is arbitrary, but useful. Like the various conventions for code layout, this convention is a good one to accept (i.e. not get too creative about), until you have some very good reason for altering it.

Deep sentence:

- Keywords like if, else, int, float, etc., are reserved you can't use them as variable names.

2.3 C Keywords

C makes use of only 32 keywords or reserved words which combine with the formal syntax to the form the C programming language. Note that all keywords in C are written in lower case. A keyword may not be used as a variable name.

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

2.4 C standard library (Header Files)

The **C standard library** consists of a set of sections of the ISO C standard which describe a collection of headers and library routines used to implement common operations, such as input/output and string handling, in the C programming language. The C standard library is an interface standard described by a document. It is not an actual library of software routines available for linkage to C programs. No such implementation is properly called C standard library.

Note that there is also the term **C library**, which may refer either to an informal synonym for C standard library (e.g. "malloc is the name of a function in the C library"), a reference to a particular implementation which provides the C standard library features and other features (e.g. "this compiler comes with a very reliable C library for ISO C and POSIX programming"), or a generic term for a library which has an interface for linking to C programs (e.g. "this software company offers a C library of fast Fourier transform functions").

The term **C** runtime library is used on some platforms to refer to a set of base libraries, which may be distributed in dynamically linkable form with an operating system (with or without header files), or distributed with a C compiler. Another term sometimes used is libc. Not just any library is called the run-time library run time in this context means the run-time support package associated with a compiler which is understood to make a language complete. The run-time support provides not only the C standard library functions, but possibly other material needed to create an environment for the C program, such as initialization prior to the invocation of the main function, or subroutines to provide arithmetic operations missing from the CPU that are needed by code generated by the C compiler.

C library headers files

Name	Description
<assert.h>	Contains the assert macro, used to assist with detecting logical errors and other types of bug in debugging versions of a program.
<complex.h>	A set of functions for manipulating complex numbers.
<ctype.h>	Contains functions used to classify characters by their types or to convert between upper and lower case in a way that is independent of the used character set (typically ASCII or one of its extensions, although implementations utilizing EBCDIC are also known).

<errno.h>	For testing error codes reported by library functions.
<fenv.h>	For controlling floating-point environment.
<float.h>	Contains defined constants specifying the implementation-specific properties of the floating-point library, such as the minimum difference between two different floating-point numbers (_EPSILON), the maximum number of digits of accuracy (_DIG) and the range of numbers which can be represented (_MIN, _MAX).
<inttypes.h>	For precise conversion between integer types.
<iso646.h>	For programming in ISO 646 variant character sets.
<limits.h>	Contains defined constants specifying the implementation-specific properties of the integer types, such as the range of numbers which can be represented (_MIN, _MAX).
<locale.h>	For setlocale and related constants. This is used to choose an appropriate locale.
<math.h>	For computing common mathematical functions.
<setjmp.h>	Declares the macros setjmp and longjmp, which are used for non-local exits.
<signal.h>	For controlling various exceptional conditions.
<stdarg.h>	For accessing a varying number of arguments passed to functions.
<stdbool.h>	For a boolean data type.
<stdint.h>	For defining various integer types.
<stddef.h>	For defining several useful types and macros.
<stdio.h>	Provides the core input and output capabilities of the C language. This file includes the venerable printf function.
<stdlib.h>	For performing a variety of operations, including conversion, pseudo-random numbers, memory allocation, process control, environment, signalling, searching, and sorting.
<string.h>	For manipulating several kinds of strings.
<tgmath.h>	For type-generic mathematical functions.
<time.h>	For converting between various time and date formats.
<wchar.h>	For manipulating wide streams and several kinds of strings using wide characters - key to supporting a range of languages.
<wctype.h>	For classifying wide characters.

Summary

In this unit we have learnt about the basic concepts of C-Programming Language. We have discussed basic programming style in this unit through examples. Then we have learnt about the use of basic data types in C- Programming. After that we have discussed regarding various reserve keywords of C- Programming. Then we have learnt the use of header files and different types of header files available in C- Programming language.

Self - Assignments

1. WAP to read a number and calculate its factorial.
2. WAP to read temperature in Celsius and convert it into Fahrenheit.
3. WAP to read three numbers and calculate average.
4. WAP to swap two numbers without using third variable.
5. WAP to calculate the percentage of 5 subjects marks provided by user.
6. WAP to calculate the area of circle.

Unit 3 : Input and Output

3.0 Console Input/Output Functions

3.1 Formatted I/O Functions

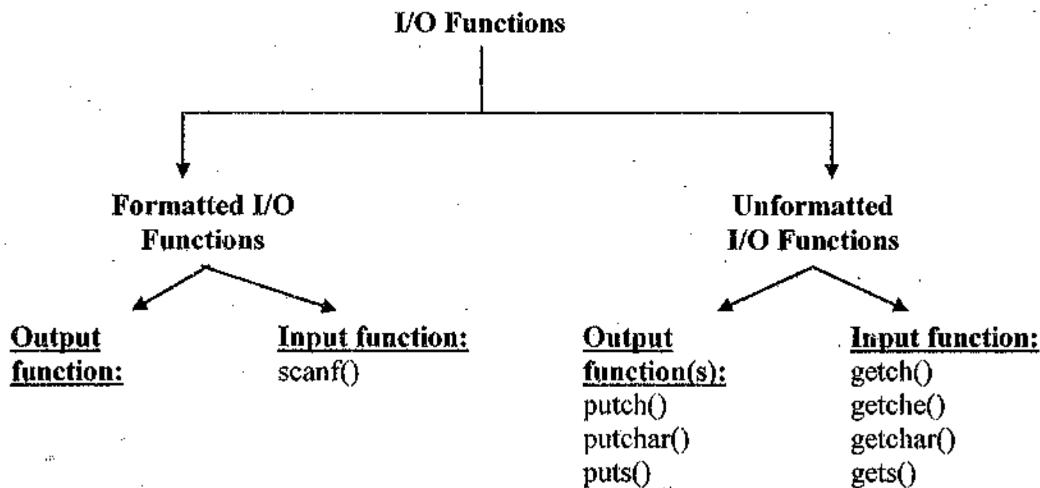
3.2 Escape Sequences

3.3 Scanset ([ ]) Specifier

3.4 Unformatted I/O Functions

Console Input / Output Functions

The screen and keyboard together are called a console. The functions that are used to get data from user through keyboard or display output on the screen are called Console I/O functions. Console functions can be divided as follows:



Formatted I/O functions

These functions allow us to supply the input in a fixed format and let us obtain the output in the specified format. There are a number of related functions used for formatted I/O, each one determining the format of the I/O from a *format string*.

For output, the format string consists of plain text, which is output unchanged, and embedded *format specifications* which call for some special processing of one of the remaining arguments to the function.

On the other hand on input, the plain text must match what is seen in the input stream; the format specifications again specify what the meaning of remaining arguments is. Each format specification is introduced by a % character, followed by the rest of the specification.

printf()

printf() is used to print or display data on the console in a formatted form. It returns the number of characters written or a negative value if an error occurs. The printf() function is declared in stdio.h.

Syntax:

int printf ("control string", list of variables)

The control string can contain:

- Characters that are simply printed as they are.
- Conversion specifications (format specifies for different data types) that specify the data types of variables being printed (starts with % sign)
- Escape sequences that begin with \ sign.

'C' Program to demonstrate use of printf() function:

```
/*Purpose: Print Hello! */
void main()
{
printf("Hello! From C");
}
```

In the above example, we are displaying Hello! From C, which is a string that has to be displayed as such that's why we have enclosed that in double quotes.

How to print values of variables:

To print value of a variable or variables, we need to specify *format specifier* in control string and then list of variables whose values has to be displayed (separated by commas).

Example(s):

To print value of character variable:

```
char ch='A';
printf("The value of ch is %c",ch);
```

output: The value of ch is A

To print value of integer variable:

```
int num=10;
printf("The value of num is %d",num);
```

output: The value of num is 10

%c specifier is used to print individual characters whereas, %d is used for integer variable. Value would be replaced at the position of format specifier. To print values of more than one variable:

```
char ch='A';

int num=10;

printf("The value of num is %d and ch is %c" ,num, ch);
```

output: The value of num is 10 and ch is A

Format Specifiers:

Data Type	Format String
Char	%c
unsigned char	%c
short or int	%i or %d
unsigned int	%u

Long	%ld
unsigned long	%lu
Float	%f or %g
double	%lf
long double	%lf
Scientific Notation	
(E in capital used with float or double type variables)	%E
Scientific Notation	
(e in lower case used with Float or double type variables)	%e
Octal Number (used with integer)	%o
Hexadecimal (used with integer)	%X
To print % sign	%%
Print String	%s

/*Purpose: Printf() example*/

```
void main()
{
        float num=798.54;
        int num2=1941;
        char ch='$';
        clrscr();
        printf("\nNumber in Signed integer:%d",num2);
        printf("\nNumber in Unsigned integer:%u",num2);
        printf("\nNumber in floatinf point:%f",num);
        printf("\nNumber in Scientific notation(E small case):%e",num);
        printf("\nNumber in Scientific Notation (E caps):%E",num);
        printf("\nNumber in octal:%o",num2);
        printf("\nNumber in Hexadecimal :%X",num2);
        printf("\nASCII value of character$:%d",ch);
        getch();
}
```

Output

```
Number in Signed integer:1941
Number in Unsigned integer:1941
Number in floatinf point:798.539978
Number in Scientific notation(E small case):7.985400e+02
Number in Scientific Notation (E caps):7.985400E+02
Number in octal:0
Number in Hexadecimal :0
ASCII value of character$:36
```

Other optional Specifiers:

Specifier	Description
Dd	Digits specifying field width
Dd	Digits specifying precision (number of digits after decimal) used with double or float.Minus sign is used for left justifying output.

/*Purpose: Printf() example with width and precision*/

```
void main()
{
        float num=100.5665;
        int num2=12;
        char st1[20]="Hello!";
        clrscr();
        printf("Float Value with precision 3 Number : %.3f\n\n",num);
        printf("The value of Integer with precision : %.3d\n\n",num2);
        printf("%s String with precision: %10.2s\n\n",st1,st1);
        printf("Right Justified Number : %12.2f\n\n",num);
        printf("Left Justified Number : %-12.2f\n\n",num);
}
```

Output:

```
Float Value with precision 3 Number : 100.566
The value of Integer with precision : 012
Hello! String with precision:         He
Right Justified Number :       100.57
Left Justified Number : 100.57
```

3.2 Escape Sequences:

These are mainly used for screen formatting of the output. They are always start from backslash (\). Since \ is considered as "escape character" these statements are called as escape sequences. They cause an escape from the normal interpretation of a string, so that the next character (after \) is recognized as having a special meaning. For example:

```
printf("Life\tis\tvery short\n&\nthere's\tno time\n");
```

output would be:

```
Life    is      very short
&
there's no time.
```

Escape sequences

Code	Meaning
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Horizontal Tab
\"	Double quotes
\'	Single quote
\0	Null
\\	Backslash
\v	Vertical tab

scanf()

It is general-purpose console input routine. It can read all the built-in data types and converts numbers into proper internal format. scanf() returns the number of data items which have successfully been assigned a value. In case of an error, it returns EOF.

Syntax:

scanf("control_string",argument_list);

control_string determines how values are read into the variables pointed to in the argument_list. The control_string consists of format specifier, white space characters and non-white space characters. The format specifiers are same as in case of printf().

Important points to be remembered while using scanf()

- The ampersand (&) sign precedes the variable to be read. This is compulsory while operating on the basic data types and optional while operating on arrays, strings etc.
- scanf() does not take any escape sequences such as \n in its first argument.
- We can specify the separator while getting values in more than one variable at a same time. For example if we want to use, as separator:
  scanf("%d,%d",&n1,&n2); /* , is separator*/
  scanf("%d%d",&n1,&n2); /* space is default separator*/
- It doesn't allow ant text to be printed in its control_statement. For example the following would be wrong:
  scanf(" Enter character:%c",&ch);

To input character value

char ch;

printf("Enter values for character");

```
scanf("%c",&ch);
```

To input integer value

```
int num=0;
printf("Enter a number");
scanf("%d",&num);
```

3.3 Scanset ([ ]) Specifier:

It defines a set of characters that may be read by scanf() and assigned to the corresponding character array. It is defined by putting the characters to be scanned for, inside the square brackets ([ ]). The beginning [must be prefixed with % sign. For example the following scanset would allow scanf() to read only the characters a b and c.

Example:

```
void main()
{
        char ch;
        clrscr();
        printf("Enter any Character(a,b or c):" );
        scanf("%[abc]",ch);
        puts(ch);
        getch();
}
```

Example:

```
/* C Program: Sum of two Numbers */
void main()
{
        int num1,num2,sum=0;
        clrscr();
        printf("Enter First Number:");
        scanf("%d",&num1);
        printf("Enter Second Number:");
        scanf("%d",&num2);
        sum=num1+num2;
        printf("The sum of %d and %d is %d",num1,num2,sum);
        getch();
}
```

3.4 Unformatted I/O Functions

getch(), getche() and getchar():

all these are input functions but used for single character only. We can use these functions when we want to get one character from the user.

getch()	getche()	getchar()
Accepts Single character only	Accepts Single character only	Accepts Single character only
It does not echo the character being entered by user.	It echoes the character being entered by user.	It echoes the character being entered by user.
It does not wait for enter key to be pressed.	It does not wait for enter key to be pressed.	It waits for enter key to be pressed.
Example: char ch;ch=getch();	Example: char ch;ch=getche();	Example: char ch;ch=getchar();

gets()

· It is also unformatted input function but used with strings. When we want to get string from user then we use this function.

Syntax : gets(<character array>);

When we give this command, It starts reading characters and stores in character array and terminates when user presses enter key. Spaces are also stored while using gets() but with scanf("%s",<array name>) spaces are not stored in array.

Example:

```
char nm[20];
gets(nm);
   or
scanf("%[^\n]",nm);  /* to store spaces too */
```

Example:

```
/* C program for getch(),getche() and getchar(), gets() */
void main()
{
        char ch,nm[20];
        clrscr();
        printf("Enter character getch():");
        ch=getch();
        printf("Enter character getche():");
        ch=getche();
        printf("Enter Character getchar():");
        ch=getchar();
        printf("Enter Name gets():");
        gets(nm);
        printf("Your Name is:%s",nm);
        getch();
}
```

puts(), putchar(), putch()

putchar() and putch() print a character on the screen whereas puts is used to print string on the screen.

Syntax :

```
putchar(<character variable>);
putch(<character variable>);
puts(<character array> or "constant string">);
```

```
void main()
{
    char ch,nm[20];
    clrscr();
    puts("Enter character getch():");
    ch=getch();
    puts("the character you typed:");
    putchar(ch);
    puts("Enter Name gets():");
    gets(nm);
    puts("Your Name is:");
    puts(nm);
    getch();
}
```

Summary

- There are basically two types of Console Input/output functions in 'C' i.e., Formatted and Unformatted I/O/ functions.
- printf() and scanf() functions are examples of Formatted I/O.
- gets(), getch(), getche(), getchar(), puts() and putchar() are examples of unformatted I/O.
- printf() is used to display data on the screen whereas scanf() is used to read data from the user. Both functions can be used with any datatype.
- getch(), getche() and getchar() functions are used to read one character and gets() function is used to read character array from the user.
- putch() and putchar() are used to print one character on the screen whereas to print character array puts() function can be used.

Questions:

1. Differentiate between
   a. scanf() and gets()
   b. formatted and unformatted I/O functions
   c. printf() and puts()
2. Write a C program to demonstrate formatted and unformatted I/O functions.

   Differentiate among getch(), getche() and getchar().

Unit 4 : Decision Making & Conditions

4.0 Introduction

4.1 Overview of compiler and Interpreters

4.2 C Program Structure

4.3 Control Instructions in C

 4.3.1 Sequence Control Instructions

 4.3.2 Selection or Decision Control Instructions

 4.3.3 Loop Control Instructions

4.0 Introduction

"C" was developed by Dennis Ritchie at the Bell Laboratories in the early 1970's. It was initially implemented on a system that used the UNIX operating System. C was the result of a development process, which started with an older language BCPL, developed by Martin Richards. BPCL influenced a language B, written by Ken Thompson, which was the predecessor of C. BPCL and B were typeless languages whereas C provides a variety of data types. Though, it has been closely associated with the UNIX system, C is not tied to any one operating system or machine. It has been used equally well to write major programs in many different domains.

C- A Middle Level Language

C is thought of as a middle level language because it combines elements of high level languages with functionalism of assemble language. C allows manipulation of bits, byte and addresses-the basic elements with which computer functions. Also C is very portable, that is software written on one machine can be adapted to work on another machine.

4.1 Overview of Compiler and Interpreters

A program is a set of instructions for performing particular task. These instructions are just like English words. The computer interprets the instructions as a 1's and 0's. The program written in high-level language (or assemble language) is called source program. To execute the source program it should be converted into machine language, which is called object code. Either Compiler or interpreter will do this activity.

Interpreters: Interpreters read one line of a source program at a time and converts it to object code. In case of any error, the same will be indicated instantly.

Compilers: A compiler reads the entire program and converts it to the object code. It provides errors not of one line but errors of whole program. Only error free programs are executed.

Compiling And Running A Program

Various stages of translation of a C program from the source code to the object code are as follows:

i) Source Code

The text of program, which the user can read and written in C language. It is the input for the C compiler.

ii) C Preprocessor

The source code is first passed through the C preprocessor. The preprocessor acts on special statements called directives, which start with a #. These directives are usually placed at the start of the program, though they can be placed anywhere else.

iii) Expanded C code

The C processor expands the shorthand of the directives and produces an output. This is called the expanded C source code and is passed to the C compiler.

iv) C Compiler

The C compiler translates the expanded code into the computer's assembly language, which can be understood by the computer.

v) Assembly Language Code

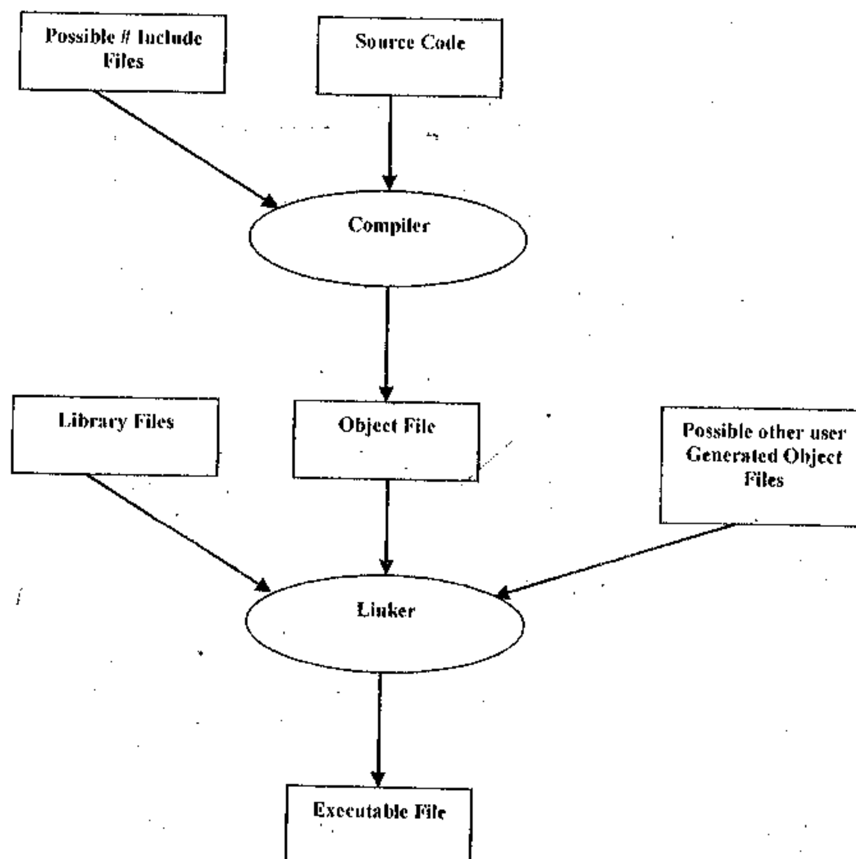
The system's assembler takes the code from the C compiler and produces object code. This code can be read and executed directly and provides input to the linker.

vi) Linker

The object code along with support routines from the standard library and any other separate compiled functions are linked together by the linker into an executable code.

vii) Loader

The executable code is run using the system's loader.



Steps for Compiling and Executing 'C' Code

4.2 C Program Structure

An important aspect of C is that it has minimal number of keywords, 32 to be precise. As mentioned earlier, Keywords are the word whose meaning has already been explained to the C compiler. These keywords are listed in the table below. These keywords combined with the formal C syntax form the C programming language. Many C compilers also have added keywords to exploit the memory organisation of certain preprocessors.

Some rules which are hold for all programs written in C are as follows:

- All keyword are lowercased.
- C is case sensitive i.e., do is different from DO.
- Keywords cannot be used for any other purpose, that is, they cannot be used as variable or function name.

main() is always the first function called when a program execution begins.

Auto	Double	int	struct
Break	Else	long	switch
Case	Enum	register	typedef
Char	Extern	return	Union
const	Float	short	unsigned
continue	For	signed	Void
default	Goto	sizeof	volatile
Do	If	static	While

Sample C Program Structure:

Include Header Section
Global Declaration Section

/* comments */
void main()
{
Declaration Part
Executable Part
}
user-defined functions
{
}

Include header file section: C program depends upon some header files for function definition that are used in program. Each header file by default has extension .h. The header files should be included using #include directive as given below.

Example: #include <stdio.h>

Or

#include "stdio.h"

Global Declaration Section: This section declares some variables that are used in more than one function or that are accessible in all the functions of the current program. These are called as global variables. These must be declared outside of all the functions.

main(): Every program written in C language must contain main() function. The main() is starting point of every C program. The execution of the program always starts with function main(). The opening curly braces i.e., { and closing curly brace i.e., } of main() includes the code that has to be executed.

Declaration Section: the declaration part declares the entire variables that are used in executable part. The initialization of variables is also done in this section. The initialization means providing initial value to the variables.

Executable part: This part includes single or multiple statements following the declaration of the variable. These statements would be executed for solving the problem.

User-defined functions: The functions defined by the user are called user-defined functions. These functions are generally defined after main(). They can also be defined before main(). This portion is optional.

Comments: Comments are important but not necessary in the program. To understand the program, the programmer can include the comments for documentation purposes. Comments are not complied by the compiler i.e., the compiler ignores comments at the time of compilation. They are used to specify the purpose of the program, reasons or clarification of complex calculations etc.

Comments are statements enclosed in /* & */ delimiters.

4.3 Control Instructions in C

Control instructions enable us to specify the order in which the various instructions in a program are to be executed by the computer. These determine the flow of control in the program. There are four types of control instructions present in C:

1. Sequence Control Instructions
2. Selection or Decision Control Instructions
3. Loop Control Instructions
4. Case Control Instructions

4.3.1 Sequence Control Instructions ensures that the instructions are executed in the same order in which they appear in the program.

Example:

```
/* C Program: Multiplication of two Numbers */
void main()
{
```

```
    int num1,num2,sum=0;
    clrscr();
    printf("Enter First Number:\n");
    scanf("%d",&num1);
    printf("Enter Second Number:\n");
    scanf("%d",&num2);
    sum=num1*num2;
    printf("The Product of %d and %d is %d",num1,num2,sum);\"
    getch();
}
```

All the statements would be executed sequentially in the above program.

Output:

Enter First Number:12

Enter Second Number:2

The Product of 12 and 2 is 24

4.3.2 Selection or Decision Control Instructions allow the computer to take a decision as to which instructions are to be executed.

If ..else constructs are used in these instructions.

- Simple If (if.... else)
- Multiple Ifs (if.... else if... else)
- Nested If (If...If...else...)

Simple If in C

In simple if there is only one condition if evaluates to true than it would do something else something else.

Syntax in C

```
if( <Condition>)
{
    <Statements to be executed if condition evaluates to true>;
}
else
{
<Statements to be executed if condition evaluates to false>;
}
```

Note: If there is only one statement in block ({ }) then we can ignore the curly braces but for more than one statements this is compulsory to type code in between pair of curly braces({ }).

Example: To check whether the given Number is Even or Odd in C?

```
void main()
{
 int num=0,result=0;
 printf("Enter Number to be Checked:");
 scanf("%d",&num);
 result=num%2;
        if(result==0)
        {
                printf( " The Given Number is Even");
else
{
                printf( " The Given Number is Odd");
}
 getch();
}
```

Multiple Ifs

In real life every situation may have multiple options and we have to choose only one option among all the available options. This situation may also arise in computer programs and we have multiple ifs for handling those situations. In multiple If, if one condition does not evaluate to true second condition may be tried and so on.

```
if(<Condition1>)
{
        <Statements to be executed if Condition1 evaluates to true>;
}
else
if<Condition2>
{
        <Statements to be executed if Condition2 evaluates to true>;
}
else
{
        <Statements to be executed if neither Condition1 nor Condition2 evaluates to true>;
}
```

Note: In multiple Ifs, if condition1 evaluates to true then it would not check the next condition.

Example: To Display Grade of the student according to the following conditions:

If percentage>=75 Grade would be "Distinction"

If percentage>=60 (but less than 75) Grade would be "1st Division"

If percentage>=50 (but less than 60) Grade would be "2nd Division"

If percentage>=45 (but less than 50) Grade would be "Pass"

Else Grade would be "Fail"

```
void main ()
{
int Per=0;
printf("Enter Percentage:");
scanf("%d", &Per);
if (Per>=75)
        printf( "Grade: Distinction");
else
 if( Per>=60)
        printf( "Grade:1st Division");
 else
        if (Per>=50)
         printf( "Grade:2nd Division");
  else
         if( Per>=45 )
         printf( "Grade: Pass");
    else
          printf( "Grade: Fail"); .
         getch();
}
```

Nested If:

Sometimes for doing something it is necessary to evaluate more than one condition. If all conditions evaluate to true than only that activity can be done. To handle such conditions in programming we use Nested Ifs.

```
if(<Condition1>)
{
  if( <Condition2> )
```

```
    {
        <Statements that will be executed if both Condition 1 and 2 is satisfied>;
    }
    else
    {
        <Statements that will be executed if Condition1 is true but Condition2 is false>;
    }
}
else
{
<Statements that will be executed if Condition1 is not satisfied>;
}
```

Example: Arranging 3 Different Numbers in Ascending order.

```
void main()
{
int N1=0;
int N2=0,N3=0;
clrscr();
printf( “Enter Three Numbers:”);
scanf(“%d %d %d”, &N1, &N2, &N3);
if (N1>N2)
{
 if( N1>N3)
 {
   if(N2>N3)
     printf(“%d%d%d”, N1, N2, N3);
   else
     printf(“%d%d%d”, N1, N3, N2);
 }
 else
  printf(“%d%d%d”, N3, N1, N2);
}
else
```

```
{
 if (N2>N3)
  {
   if(N3>N1)
   printf("%d%d%d", N2, N3, N1);
   else
        printf("%d%d%d", N2, N1, N3);
  }
 else
        printf("%d%d%d",N3, N2, N1);
}
getch();
}
```

Use of logical operators in nested ifs: We can use logical operators (&& or ||) instead of nested if.

Example: Finding maximum of 3 different numbers with logical operators.

```
void main()
{
        int n1,n2,n3;
        clrscr();
        printf( "Enter Three Numbers:");
        scanf("%d %d %d", &n1, &n2, &n3);
        if(n1>n2 && n1>n3)
        printf("n1 is maximum");
        else
        if(n2>n3 && n2>n1)
        printf("n2 is maximum");
        else
        if(n3>n1 && n3>n2)
        printf("n3 is maximum");
        getch();
}
```

4.3.3 Loop Control Instructions: Although we will discuss loops or iteration in the next chapter, but right now we will learn the syntax of all three loops supported by C.

- while loop
- for loop
- do…while loop

while loop:

General syntax of while loop in C is:

```
while( <Condition>)
{
  <Statement to be executed if condition is true>;
}
```

Features of while loop:

- It is pre-tested loop i.e., it first checks the condition and then executes the statements.
- Executes till condition evaluates to true.
- Terminates when condition evaluates to false.
- If first time condition evaluates to false, it would never be executed.

Example: Printing series of 100 numbers (1,2,3,………100)

```
void main()
{
 int N=1;
 while(N<=100)
 {
        printf("%d", N);
        N=N+1;
 }
getch();
}
```

for Loop

for loop has three parts:

Initialization part: Here we can initialize the values of variables being used in loop.

Condition Part: for loop in C executes according to the condition specified in this part.

increment/decrement part: After every iteration, change in values of variables could be mentioned here.

All these parts could be blanked but ; is compulsory. So we have to put two semicolons(;) compusorly.

Syntax:

```
for( <initialization part>;<condition part>;<increment/decrement part>)
```

```
{
        <Statement to be executed condition evaluates to true>;
}
```

Example: To print series of Odd numbers till 99(1,3,5….99).

```
void main()
{
 int n;
 for(n=1;n<=100;n+=2)
  {
        printf("%d,", n);
  }
getch();
}
```

Features of for loop:

- It is pre-tested loop.
- Executes till condition evaluates to true.
- Terminates when condition evaluates to false
- If first time condition evaluates to false, it would never be executed.

do…while Loop:

Syntax:

```
do
{
 <Statement to be executed if condition is true>;
}
while( <Condition>);
```

Same as while loop except it is post-tested loop i.e., it first executes the statement and then checks for the condition. That means it would definitely be executed for once whether the condition is false or true at very first time.

Features of do…while loop:

- It is post-tested loop.
- Executes till condition evaluates to true.
- Terminates when condition evaluates to false
- If first time condition evaluates to false still it would be executed for at-least once.

Example: Print series of Even Number till 100. (2 4 6….100)

```
void main()
{
 int N=1;
 do
 {
        printf("%d", N);
        N=N+1;
 } while(N<=100);
getch();
}
```

Nested Loops

When one loop is inserted in another loop, this condition is known as nested loops. In nested loop inner loop is executed for each change in outer loop. For example if we want to print triangle of stars like:

```
*
* *
* * *
* * * *
* * * * *
```

```
void main()
{
        int I,j;
        for(I=1;I<=5;I++)
        {
                for(j=1;j<=I;j++)
                {
                printf("* ");
                }
        printf("\n");
        }
getch();
}
```

Note: We shall discuss loops in the next unit in details.

4. Case Control Instructions:

The control statements that allow us to make a decision from the number of choices is called switch or more precisely a switch-case –default.

Syntax:

```
switch( <integer variable>)
{
        case constant1:
        <statements to be executed if integer variable= constant1>;
        break;
        case constant2:
        <statements to be executed if integer variable= constant2>;
        break;
        default:
        <finally, if all the statements are false do this>;
}
```

Use of break statement:

Consider the following example:

```
void main()
{
        int i=2;
        switch(i)
        {
                case 1:
                 puts("case 1\n");
                case 2:
                 puts("case 2\n");
                case 3:
                 puts("case 3\n");
                case 4:
                 puts("case 4\n");
                default:
                 puts("In default case\n ");
        }
```

```
}
```

output:

```
case 2
case 3
case 4
In default case
```

The output is not what we expected. This is because that switch executes the case where a match is found and all the subsequent statements and the default as well.

Break statements terminates the switch and transfer the control anyway. That's why we use break in switch so that when the match is found, it do the necessary action and exists from switch. Break could also be used with loops where it terminates the innermost loop.

Continue statement is used if we want to transfer the control to the loop conditions statement. Lets take an example, suppose I want to print all the numbers except that are divisible by 5 with continue from 1 to 100:

```
void main()
{
    int I=0;
    while(I<=100)
    {
        I++;
        if(I%5==0)
            continue;
        printf("%d",I);
    }
}
```

Whenever, I is found to be divisible by 5 then it would find continue and it will skip all the statements below it and transfer the control to while(I<=100) statement. Otherwise it would print the value of i.

Summary

- The program written in high-level language (or assemble language) is called source program.
- To execute the source program it should be converted into machine language, which is called object code. Either Compiler or interpreter will do this activity.
- Interpreters read one line of a source program at a time and converts it to object code. In case of any error, the same will be indicated instantly.
- A compiler reads the entire program and converts it to the object code. It provides errors not of one line but errors of whole program. Only error free programs are executed.
- The object code along with support routines from the standard library and any other separate compiled functions are linked together by the linker into an executable code.

- The preprocessor acts on special statements called directives, which start with a #.
- Control instructions enable us to specify the order in which the various instructions in a program are to be executed by the computer. These determine the flow of control in the program.
- The control statements that allow us to make a decision from the number of choices is called switch or more precisely a switch-case –default.

Questions:

1. Explain different types of control statements of C.
2. Differentiate between simple if and multiple if with suitable example.
3. What is a purpose of break statement?
4. Why do we use continue statement?
5. What is the purpose of default keyword in switch.
6. Switch can be used in place of multiple ifs. Proof this statement with suitable example.
7. Write a note on compiling and executing C program.
8. Write a "C" code to arrange three numbers in descending order.
9. Write a "C" code to check whether the given year is leap year or not.
10. Write a "C" code to check the zodiac sign of given date of birth.

Unit 5 : Looping

Structure of the Unit

5.0 Objective

5.1 Introduction

5.1.1 While loop

5.1.2 For loop

5.1.3 Do while loop

5.2 Break and continue

5.3 Programming examples

5.4 Self Learning Exercise

5.5 Summary

5.6 Further Readings

5.7 Answer to Self Learning Exercise

5.8 Unit End Questions

5.0 Objective

At the end of the chapter you should be well informed about different types of loops available in C and with the help of the given programming examples to implement the loops, you must be able to write various loop based programs in C.

5.1 Introduction

Many times we want to do something a lot of times. An example would be printing a character at the beginning of each line on the screen. To do this you would have to type out 24 printf commands because there are 25 rows on the screen and the 25th row will be left blank. We can use a loop to do this for us and only have to type one printf command.

So loops are used to repeat a block of code. Being able to have your program repeatedly execute a block of code is one of the most basic but useful tasks in programming — many programs or websites that produce extremely complex output (such as a message board) are really only executing a single task many times. (They may be executing a small number of tasks, but in principle, to produce a list of messages only requires repeating the operation of reading in some data and displaying it.) Now, think about what this means: a loop lets you write a very simple statement to produce a significantly greater result simply by repetition.

Most real programs contain some construct that loops within the program, performing repetitive actions on a stream of data or a region of memory.

Following are basic loops available in C:

1. **while loop**
2. **for loop**
3. **do while loop**

5.1.1 The while loop

The while loop is used when you don't know how many times you want the loop to run. You also have to always make sure you initialize the loop variable before you enter the loop. Another thing you have to do is increment the variable inside the loop. Following is the syntax for while loop.

```
while (expression)
  {
        ...block of statements to execute...
  }
```

The while loop continues to loop while some condition is true, which is given in the form of expression in the syntax. When the condition becomes false, the looping is discontinued. The condition is tested upon entering the loop.

Here is an example of a while loop that runs the amount of times that the user enters:

```
#include<stdio.h>
int main()
{
  int i,times;
  scanf("%d",&times);
  i = 0;
  while (i <= times)
  {
    i++;
    printf("%d\n",i);
  }
  return 0;
}
```

5.1.2 The for loop

The for loop lets you loop from one number to another number and increases by a specified number each time. This loop is best suited for the problems where we know exactly how many times we need to loop for. The for loop uses the following structure:

```
for (expression_1; expression_2; expression_3)
   {
        ...block of statements to execute...
   }
```

Where expression _1 is starting number, expression_2 is loop condition and expression_3 is increase variable

With loops you also have to put statements to be executed between curly brackets if there is more than one of them. Here is the solution to the problem that we had with the 24 printf commands:

```
#include <stdio.n)
int main()
{
```

```
  int i;
  for (i = 1;i <= 24;i++)
    printf("H\n");
  return 0;
}
```

A for loop is made up of three parts inside its brackets which are separated by semi-colons. The first part initializes the loop variable. The loop variable controls and counts the number of times a loop runs. In the example the loop variable is called i and is initialized to 1. The second part of the for loop is the condition a loop must meet to keep running. In the example the loop will run while i is less than or equal to 24 or in other words it will run 24 times. The third part is the loop variable incrementer. In the example i++ has been used which is the same as saying i = i + 1. This is called incrementing. Each time the loop runs i has 1 added to it. It is also possible to use i— to subtract 1 from a variable in which case it's called decrementing.

The for loop is a special case, and is equivalent to the following while loop:

```
expression_1;
while (expression_2)
{
        ...block of statements...
        expression_3;
}
```

For instance, the following structure is often encountered:

```
i = initial_i;

while (i <= i_max)
{
        ...block of statements...

        i = i + i_increment;
}
```

This structure may be rewritten in the easier syntax of the for loop as:

```
for (i = initial_i; i <= i_max; i = i + i_increment)
{
        ...block of statements...
}
```

5.1.3 The do while loop

The do while loop is the same as the while loop except that it tests the condition at the bottom of the loop.

```
#include<stdio.h>
int main()
{
  int i,times;
```

```
scanf("%d",&times);
 i = 0;
 do
 {
   i++;
   printf("%d\n",i);
 }
 while (i <= times);
 return 0;
}
```

The appropriate block of statements is executed according to the value of the expression, compared with the constant expressions in the case statement. The break statements insure that the statements in the cases following the chosen one will not be executed. If you would want to execute these statements, then you would leave out the break statements. This construct is particularly useful in handling input variables.

6.2 Break and continue

Infinite loops are possible (e.g. for(;;)), C permits you to write an infinite loop, and provides the break statement to breakout of the loop.

You can exit out of a loop at any time using the break statement. This is useful when you want a loop to stop running because a condition has been met other than the loop end condition.

```
#include<stdio.h>
int main()
{
  int i;
  while (i < 10)
  {
    i++;
    if (i == 5)
      break;
  }
  return 0;
}
```

You can use continue to skip the rest of the current loop and start from the top again while incrementing the loop variable again. The following example will never print "Hello" because of the continue.

```
#include<stdio.h>

int main()
{
  int i;
  while (i < 10)
```

```
  {
    i++;
    continue;
    printf("Hello\n");
  }
  return 0;
```

6.3 Programming examples

Following are some example programs to print the various patterns using loops :

Pattern 1 :

```
   1
  121
 12321
1234321
```

Program :

```
#include<stdio.h>
#include<conio.h>
Void main()
{       int i, j, n;
        printf("Please enter number of the rows to be print");
        scanf("%d", &n);
        for(i=1;i<=n;i++) /*for counting the rows */
        {       for(j=1;j<=n-i;j++) /*for counting the spaces */
                printf(" ");
                for(j=1;j<=i;j++) /* for the printing of first half */
                printf("%2d",j);
                for(j=i-1;j>=1;j—) /* for the printing of second half */
                printf("%2d",j);
                printf("\n");
}
getch();
}
```

Pattern 2 :

```
   1
  212
 32123
4321234
```

Program :

```
#include<stdio.h>
#include<conio.h>
Void main()
```

```
{       int i, j, n;
        printf("Please enter number of the rows to be print");
        scanf("%d", &n);
        for(i=1;i<=n;i++) /*for counting the rows */
        {       for(j=1;j<=n-i;j++) /*for counting the spaces */
                printf(" ");
                for(j=i;j>=1;j—)
                printf("%d",j);
                for(j=2;j<=1;j++)
                printf("%d",j);
                printf("\n");
}
getch();
}
```

Pattern 3 :

```
1
121
12321
1234321
```

Program :

```
#include<stdio.h>
#include<conio.h>
Void main()
{       int i, j, n;
        printf("Please enter number of the rows to be print");
        scanf("%d", &n);
        for(i=1;i<=n;i++) /*for counting the rows */
        {       for(j=1;j<=i;j++)
                printf("%d ",j);
                for(j=i-1;j>=1;j—)
                printf("%d",j);
                printf("\n");
}
getch();
}
```

Pattern 4 :

```
*****
*
*
*
*
*****
```

Program :

```
#include<stdio.h>
#include<conio.h>
Void main()
{       int i, n;
        printf("Please enter number of the rows to be print");
        scanf("%d", &n);
        for(i=1;i<n;i++) /*for counting the rows */
                printf("*");
        for(i=1;i<=n;i++)
        {       printf("*");
printf("\n");
        }
for(i=1;i<=n;i++)
                printf("*");
getch();
}
```

Pattern 5 :

```
*
**
***
****
*****
```

Program :

```
#include<stdio.h>
#include<conio.h>
Void main()
{       int i, j, n;
        printf("Please enter number of the rows to be print");
        scanf("%d", &n);
        for(i=1;i<=n;i++) /*for counting the rows */
        {       for(j=1;j<=i;j++)
                printf("*");
                printf("\n );
        }
        getch();
}
```

5.4 Self Learning Exercise

Multiple choice questions

1. Execution of a **break** statement in the body of a **while** loop

   [a.] causes the program to terminate.

   [b.] causes the current iteration of the loop to terminate and the next iteration to begin.

   [c.] causes the loop to terminate and the statement after the body of the while loop to be executed next.

   [d.] None of the above.

2. Execution of a **continue** statement in the body of a **for** loop

   [a.] causes the program to terminate.

   [b.] causes the current iteration of the loop to terminate and the next iteration to begin.

   [c.] causes the loop to terminate and the statement after the body of the while loop to be executed next.

   [d.] None of the above.

3. Determine the output of the following code.

```
int a = 5, b = 4, c = 1 ;
if ( a < b + 1 ) {
  printf("Tea, Earl Grey, Hot!\n") ;
} else if ( a > b + c ) {
  printf("Ahead warp factor 9. Engage!\n") ;
} else if ( a % b >= c ) {
  printf("Warp core breach in 20 seconds!\n") ;
} else {
  printf("I sense a lot of anxiety in this room.\n") ;
}
```

   [a.] Tea, Earl Grey, Hot!

   [b.] Ahead warp factor 9. Engage!

   [c.] Warp core breach in 20 seconds!

   [d.] I sense a lot of anxiety in this room.

4. What is the effect of the following code?

```
int i, total ;
for (i = 1 ; i <= 15 ; i = i + 1)
{
 if ( i % 3 == 0)
  {
   printf("%d ", i) ;
  }
}
```

printf("\n\n");

[a.] It prints out the integers from 3 to 15.

[b.] It prints out the multiples of 3 from 3 to 15.

[c.] It prints out the sum of the integers from 3 to 15.

[d.] It prints out the sum of the multiples of 3 from 3 to 15.

5.5 Summary

This chapter starts with the introduction of the loops. After this, various loops, while..do, for and do..while are discussed with break and continue statements with their syntax and programming examples in C. In next section various example programs are given for the better understanding of the loops. And at the end questions are given for the self assessment.

5.6 Further Readings

- Programming in C :- Balguruswamy, Tata McGraw-Hill Publication.
- The C programming Lang., Pearson Ecl :- Dennis Ritchie, Prentice HAll, Delhi.
- Let us C :- Yashwant Kanetkar, BPB publication

5.7 Answer to Self Learning Exercises

Question	Answer	Question	Answer
1	c	3	c
2	b	4	b

5.8 Unit End Questions

1. Write short note on

   (a) for loop

   (b) continue-break statement

2. State the difference between while and do-while.

3. Write a C program to calculate factorial of a given number.

4. Write a C program to print Armstrong numbers from 1 to 500.

Unit 6 : Arrays

Structure of the Unit

6.0 Objective

6.1 Introduction

6.1.1 Declaration of arrays

6.1.2 Initialization of arrays

6.2 Multi dimensional arrays

6.2.1 Row-major and column-major order

6.2.2 Initialization of multidimensional arrays

6.3 Sorting

6.3.1 Bubble Sort

6.3.2 Selection Sort

6.4 Strings

6.5 String Operations

6.5.1 string.h

6.5.2 strlen function

6.5.3 strcat function

6.5.4 strcmp function

6.5.5 strcpy function

6.6 Self Learning Exercise

6.7 Summary

6.8 Further Readings

6.9 Answer to Self Learning Exercise

6.10 Unit End Questions

6.0 Objective

At the end of the chapter you should be well informed about arrays in C and the declaration and use of one and multidimensional arrays in C. With the help of the given programming examples to implement the arrays, you must be able to write various array based programs in C.

6.1 Introduction

The C language provides a capability that enables the user to define a set of ordered data items known as an array.

Suppose we had a set of grades that we wished to read into the computer and suppose we wished to perform some operations on these grades, we will quickly realize that we cannot perform such an

operation until each and every grade has been entered since it would be quite a tedious task to declare each and every student grade as a variable especially since there may be a very large number.

In C we can define variable called grades, which represents not a single value of grade but a entire set of grades. Each element of the set can then be referenced by means of a number called as index number or subscript.

6.1.1 Declaration of arrays:

Like any other variable arrays must be declared before they are used. The general form of declaration is:

Data type array-name[size];

The type specifies the type of the elements that will be contained in the array, such as int float or char and the size indicates the maximum number of elements that can be stored inside the array for ex:

float weight[20];

Declares the weight to be an array containing 20 real elements. Any subscripts 0 to 19are valid. In C the array elements index or subscript begins with number zero. So weight [0] refers to the first element of the array and the declaration will reserve 20 continuous memory allocations and can be referred as weight[0], weight[1], weigh[2] and so on.

As individual array element can be used anywhere that a normal variable with a statement such as

g = grade [50];

The statement assigns the value stored in the 50th index of the array to the variable g.

More generally if I is declared to be an integer variable, then the statement g=grades [I];

Will take the value contained in the element number I of the grades array to assign it to g. so if I were equal to 5 when the above statement is executed, then the value of grades [5] would get assigned to g.

The C system needs declaration information to identify the type of the values stored in an array and the information in order to determine how much memory space to reserve for the particular array.

The declaration int values[10]; would reserve enough space for an array called values that could hold up to 10 integers. Refer to the below given picture to conceptualize the reserved storage space.

values[0]
values[1]
values[2]
values[3]
values[4]
values[5]
values[6]
values[7]
values[8]
values[9]

6.1.2 Initialization of arrays:

We can initialize the elements in the array in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is:

type array_name[size]={list of values};

The values in the list care separated by commas, for example the statement

int number[3]={0,0,0};

Will declare the array size as a array of size 3 and will assign zero to each element if the number of values in the list is less than the number of elements, then only that many elements are initialized. The remaining elements will be set to zero automatically.

In the declaration of an array the size may be omitted, in such cases the compiler allocates enough space for all initialized elements. For example the statement

int counter[]={1,1,1,1};

A value stored into an element in the array simply by specifying the array element on the left hand side of the equals sign. In the statement

grade [100]=50;

The value 50 is stored into the element number 100 of the grade array.

The ability to represent a collection of related data items by a single array enables us to develop concise and efficient programs. For example we can very easily sequence through the elements in the array by varying the value of the variable that is used as a subscript into the array. So the for loop

```
For(i=0;i<50;++i)
sum = sum + grades [i];
```

will sequence through the first 50 elements of the array grades (elements 0 to 49) and will add the values of each grade into sum. When the for loop is finished, the variable sum will then contain the total of first 50 values of the grades array (Assuming sum were set to zero before the loop was entered).

The initialization of arrays in c suffers two draw backs

1. There is no convenient way to initialize only selected elements.
2. There is no shortcut method to initialize large number of elements.

```
/* Program to count the no of positive and negative numbers*/
#include< stdio.h >
void main( )
{
int a[50],n,count_neg=0,count_pos=0,i;
printf("Enter the size of the arrayn");
scanf("%d",&n);
printf("Enter the elements of the arrayn");
for i=0;i < n;i++)
```

```
scanf("%d",&a[i]);
for(i=0;i<n;i++)
{
if(a[i]<0)
count_neg++;
else
count_pos++;
}
printf("There are %d negative numbers in the arrayn",count_neg);
printf("There are %d positive numbers in the arrayn",count_pos);
}
```

6.2 Multi dimensional Arrays:

Often there is a need to store and manipulate two dimensional data structure such as matrices and tables. Here the array has two subscripts. One subscript denotes the row and the other the column. The declaration of two dimension arrays is as follows:

```
data_type     array_name[row_size][column_size];
int m[10][20]
```

Here m is declared as a matrix having 10 rows(numbered from 0 to 9) and 20 columns(numbered 0 through 19). The first element of the matrix is m[0][0] and the last row last column is m[9][19]

6.2.1 Row-major and Column-major order

In computing, **row-major order** and **column-major order** describe methods for storing multidimensional arrays in linear memory. Following standard matrix notation, rows are identified by the first index of a two-dimensional array and columns by the second index.

In row-major storage, a multidimensional array in linear memory is accessed such that rows are stored one after the other. It is the approach used by the C programming language as well as many other languages.

When using row-major order, the difference between addresses of array cells in increasing rows is larger than addresses of cells in increasing columns. For example, consider this 2×3 array:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

An array declared in C as

```
int A[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

would be laid out contiguously in linear memory as:

1 2 3 4 5 6

Column-major order is a similar method of flattening arrays onto linear memory, but the columns are listed in sequence. The programming languages FORTRAN uses column-major ordering. The array

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

if stored contiguously in linear memory with column-major order would look like the following:
1 4 2 5 3 6

Elements of multi dimension arrays:

A two dimensional array marks [4][3] is shown below figure. The first element is given by marks [0][0] contains 35.5 and second element is marks [0][1] and contains 40.5 and so on.

marks [0][0] 35.5	marks [0][1]40.5	marks [0][2]45.5
marks [1][0] 50.5	marks [1][1]55.5	marks [1][2]60.5
marks [2][0]	marks [2][1]	marks [2][2]
marks [3][0]	marks [3][1]	marks [3][2]

6.2.2 Initialization of multidimensional arrays:

Like the one dimension arrays, 2 dimension arrays may be initialized by following their declaration with a list of initial values enclosed in braces.

Example:

```
int table[2][3]={0,0,0,1,1,1};
```

Initializes the elements of first row to zero and second row to 1. The initialization is done row by row. The above statement can be equivalently written as

```
int table[2][3]={{0,0,0},{1,1,1}}
/* Program to read  the matrix*/
#include< stdio.h >
void main( )
{
int a[3][3],n,m,i,j;
printf("Enter the number of rows in the matrix");
scanf("%d",&n);
printf("Enter the number of columns in the matrix");
scanf("%d",&m);
printf("Enter the matrix elements");
for (i=0;i < n;i++)
   for(j=0;j<m;j++)
       scanf("%d",&a[i][j]);
}
```

6.3 Sorting

Sorting is the process of arranging data into meaningful order so that you can analyze it more effectively. For example, you might want to order sales data by calendar month so that you can produce a graph of sales performance. You can use Discoverer to sort data as follows:

- sort text data into alphabetical order
- sort numeric data into numerical order
- group sort data to many levels, for example, you can sort on City within Month within Year

6.3.1 Bubble sort

The simplest sorting algorithm is bubble sort. The bubble sort works by iterating down an array to be sorted from the first element to the last, comparing each pair of elements and switching their positions if necessary. This process is repeated as many times as necessary, until the array is sorted. Since the worst case scenario is that the array is in reverse order, and that the first element in sorted array is the last element in the starting array, the most exchanges that will be necessary is equal to the length of the array. Here is a simple example: Given an array 23154 a bubble sort would lead to the following sequence of partially sorted arrays: 21354, 21345, 12345. First the 1 and 3 would be compared and switched, then the 4 and 5. On the next pass, the 1 and 2 would switch, and the array would be in order.

The basic code for bubble sort looks like this, for sorting an integer array:

```
for(int x=0; x<n; x++)
{
        for(int y=0; y<n-1; y++)
        {
                if(array[y]>array[y+1])
                {
                        int temp = array[y+1];
                        array[y+1] = array[y];
                        array[y] = temp;
                }
        }
}
```

Note that this will always loop n times from 0 to n, so the order of this algorithm is O(n^2).

6.3.2 Selection Sort

Selection sort is the most conceptually simple of all the sorting algorithms. It works by selecting the smallest (or largest, if you want to sort from big to small) element of the array and placing it at the head of the array. Then the process is repeated for the remainder of the array; the next largest element is selected and put into the next slot, and so on down the line.

Because a selection sort looks at progressively smaller parts of the array each time (as it knows to ignore the front of the array because it is already in order), a selection sort is slightly faster than bubble sort, and can be better than a modified bubble sort.

Here is the code for a simple selection sort:

```
for(int x=0; x<n; x++)
{
        int index_of_min = x;
        for(int y=x; y<n; y++)
        {
                if(array[index_of_min]>array[y])
                {
                        index_of_min = y;
                }
        }
        int temp = array[x];
        array[x] = array[index_of_min];
        array[index_of_min] = temp;
}
```

The first loop goes from 0 to n, and the second loop goes from x to n, so it goes from 0 to n, then from 1 to n, then from 2 to n and so on. The multiplication works out so that the efficiency is n*(n/2), though the order is still O(n^2).

6.4 Strings

A string is a sequence of characters. Any sequence or set of characters defined within double quotation symbols is a constant string. In C, many times it is required to do some meaningful operations on strings they are:

- Reading and displaying strings
- Combining or concatenating strings
- Copying one string to another.
- Comparing string and checking whether they are equal
- Extraction of a portion of a string

Strings are stored in memory as ASCII codes of characters that make up the string appended with '\0'(ASCII value of null). Normally each character is stored in one byte, successive characters are stored in successive bytes.

Character	a	b	c	d	e	f	g	\0

The last character is the null character having ASCII value zero.

Initializing Strings

Following the discussion on characters arrays, the initialization of a string must the following form which is simpler to one dimension array.

char month1[ ]={'j','a','n','u','a','r','y'};

Then the string month is initializing to January. This is perfectly valid but C offers a special way to initialize strings. The above string can be initialized char month1[]="January"; The characters of the string are enclosed within a part of double quotes. The compiler takes care of string enclosed within a pair of a double quotes. The compiler takes care of storing the ASCII codes of characters of the string in the memory and also stores the null terminator in the end.

```
/*String.c string variable*/
#include < stdio.h >
main()
{
char month[15];
printf ("Enter the string");
gets (month);
printf ("The string entered is %s", month);
}
```

In this example string is stored in the character variable month the string is displayed in the statement.

printf("The string entered is %s", month");

String is one dimension array. Each character occupies a byte. A null character (\0) that has the ASCII value 0 terminates the string. The figure shows the storage of string January in the memory recall that \0 specifies a single character whose ASCII value is zero.

J
A
N
U
A
R
Y
\0

Character string terminated by a null character '\0'. A string variable is any valid C variable name and is always declared as an array. The general form of declaration of a string variable is char string_name[size];

The size determines the number of characters in the string name.

Example:

char month[10]; char address[100];

The size of the array should be one byte more than the actual space occupied by the string since the complier appends a null character at the end of the string.

Reading Strings from the terminal:

The function scanf with %s format specification is needed to read the character string from the terminal.

Example:

```
Char    address[15];

scanf("%s",address);
```

Scanf statement has a draw back it just terminates the statement as soon as it finds a blank space, suppose if we type the string new york then only the string new will be read and since there is a blank space after word "new" it will terminate the string.

Note that we can use the scanf without the ampersand symbol before the variable name. In many applications it is required to process text by reading an entire line of text from the terminal.

The function getchar can be used repeatedly to read a sequence of successive single characters and store it in the array.

We cannot manipulate strings since C does not provide any operators for string. For instance we cannot assign one string to another directly.

For example:

```
String="xyz";

String1=string2;
```

Are not valid. To copy the characters in one string to another string we may do so on a character to character basis.

Writing strings to screen:

The printf statement along with format specifier %s to print strings on to tue screen. The format %s can be used to display an array of characters that is terminated by the null character for example printf("%s",name); can be used to display the entire contents of the array name. Function puts is also used to display the string, but it prints unformatted string only. Like : puts("hello");

Arithmetic operations on characters:

We can also manipulate the characters as we manipulate numbers in C language. When ever the system encounters the character data it is automatically converted into a integer value by the system. We can represent a character as a interface by using the following method.

```
X='a';
Printf("%d\n",X);
```

will display 97 on the screen. Arithmetic operations can also be performed on characters for example x='z'-1; is a valid statement. The ASCII value of 'z' is 122 the statement the **therefore will** assign 121 to variable x.

It is also possible to use character constants in relational expressions for example ch>'a' && ch < = 'z' will check whether the character stored in variable ch is a lower case letter. A character digit can also be converted into its equivalent integer value suppose un the expression a=character-'1'; where a is defined as an integer variable and character contains value 8 then a= ASCII value of 8 ASCII value '1'=56-49=7.

We can also get the support of the C library function to converts a string of digits into their equivalent integer values the general format of the function in x=atoi(string) here x is an integer variable and string is a character array containing string of digits.

6.5 String operations

6.5.1 string.h

C language recognizes that string is a different class of array by letting us input and output the array as a unit and are terminated by null character. C library supports a large number of string handling functions that can be used to array out many of the string manipulations such as:

- Length (number of characters in the string).
- Concatentation (adding two are more strings)
- Comparing two strings.
- Substring (Extract substring from a given string)
- Copy(copies one string over another)

To do all the operations described here it is essential to include string.h library header file in the program.

6.5.2 strlen() function:

This function counts and returns the number of characters in a string. The length does not include a null character.

Syntax n=strlen(string);

where n is integer variable. Which receives the value of length of the string.

Example

length=strlen("Hollywood");

The function will assign number of characters 9 in the string to a integer variable length.

```
/*write a c program to find the length of the string using strlen() function*/
#include < stdio.h >
include < string.h >
void main()
{
char name[100];
int length;
printf("Enter the string");
gets(name);
length=strlen(name);
printf("\nNumber of characters in the string is=%d",length);
}
```

6.5.3 strcat() function:

when you combine two strings, you add the characters of one string to the end of other string. This process is called concatenation. The strcat() function joins two strings together. It takes the following form

strcat(string1,string2)

string1 and string2 are character arrays. When the function strcat is executed string2 is appended to string1. the string at string2 remains unchanged.

Example

```
strcpy(string1,"sri");

strcpy(string2,"Bhagavan");

Printf("%s",strcat(string1,string2);
```

From the above program segment the value of string1 becomes sribhagavan. The string at str2 remains unchanged as bhagawan.

6.5.4 strcmp function:

In c you cannot directly compare the value of 2 strings in a condition like if(string1==string2)

Most libraries however contain the strcmp() function, which returns a zero if two strings are equal, or a non zero number if the strings are not the same. The syntax of strcmp() is given below:

strcmp(string1,string2)

String1 and string2 may be string variables or string constants. String1, and string2 may be string variables or string constants some computers return a negative if the string1 is alphabetically less than the second and a positive number if the string is greater than the second.

Example:

strcmp("Newyork","Newyork") will return zero because both strings are equal.

strcmp("their","there") will return a 9 which is the numeric difference between ASCII 'i' and ASCII 'r'.

strcmp("The", "the") will return 32 which is the numeric difference between ASCII "T" and ASCII "t".

6.5.5 strcpy() function:

C does not allow you to assign the characters to a string directly as in the statement name="Robert";

Instead use the strcpy() function found in most compilers the syntax of the function is illustrated below.

strcpy(string1,string2);

strcpy function assigns the contents of string2 to string1. string2 may be a character array variable or a string constant.

strcpy(Name,"Robert");

In the above example Robert is assigned to the string called name.

6.6 Self Learning Exercise

Multiple choice questions

1. What will happen if in a C program you assign a value to an array element whose subscript exceeds the size of array?

   [1] The element will be set to 0.

   [2] The compiler would report an error.

   [3] The program may crash if some important data gets overwritten.

   [4] The array size would appropriately grow.

2. int a[20];

   In the above declaration index will vary from

   [a] 0 to 19

   [b] 1 to 20

   [c] 1 to 19

   [d] None of the above.

3. At the end string stores

   [a] '/0'

   [b] '\0'

   [c] ' '

   [d] None of the above.

4. Bubble sort arranges array elements in

   [a] only in ascending order.

   [b] only in descending order

   [c] in both the orders

   [d] none of the above

6.7 Summary

This chapter starts with the introduction of the arrays. After this, declaration and type of arrays are discussed with example, in addition to this how array elements can be accessed is also discussed. In next section various sorting methods are given for the better understanding of the arrays. At the end strings are discussed with various built-in string functions.

7.8 Further Readings

- Programming in C :- Balguruswamy, Tata McGraw-Hill Publication.
- The C programming Lang., Pearson Ecl :- Dennis Ritchie, Prentice HAll, Delhi.
- Let us C :- Yashwant Kanetkar, BPB publication

6.9 Answer to Self Learning Exercises

Question	Answer	Question	Answer
1	c	3	b
2	a	4	c

6.10 Unit End Questions

1. Write short note on
   (a) Multi-dimensional array
   (b) strings
2. State the difference between strcmp and strcpy functions.
3. Write a C program to find the maximum number from a list of numbers.
4. Write a C program to find the frequency of a word 'is' in a string.
5. Write a C program to find the sum of two matrices.

Unit 7 : Introduction to Functions

Structure of the Chapter

7.0 Objective

7.1 Introduction

7.1.1 Function prototype and declaration

7.1.2 Calling a Function

7.1.3 Return statement

7.2 Scope of function variables

7.3 Storage Classes

7.3.1 Automatic

7.3.2 External

7.3.3 Static

7.3.4 Register

7.4 Sample Programs using functions

7.5 Functions with array

7.6 Self Learning Exercise

7.7 Summary

7.8 Further Readings

7.9 Answer to Self Learning Exercise

7.10 Unit End Questions

7.0 Objective

The objective of the chapter is to introduce the concept of modular programming. After completing this chapter you will be familiar with functions in C. In addition to introduction you will also be informed about function declaration, parameter passing and calling of function etc. With the help of the given programming example to implement the functions, you will be able to write various function based programs in C.

7.1 Introduction

Functions allow complicated programs to be divided into small blocks, each of which is easier to write, read, and maintain. Programs in real life applications are very large in terms of size, so it becomes very complex to handle these programs. Functions are solution to this, they provide modularity to the program. Complex programs can be divided into smaller parts known as subprograms, and specifically in C these are known as function. This method of dividing complete program into smaller modules is known as modular programming.

Advantages of modular programming :

1. It is easier to test and debug a long program by testing and debugging the individual modules in the program.
2. A modular program is easier to understand.
3. It is easier to modify a modular program by modification or replacement of individual modules without upsetting the entire program.
4. The programmer's output increases manifold, because each individual module can be developed separately in isolation. Thus, parallel designing of the modules speeds up the program design jobs.
5. The modules approach of designing the programs enhances the readability of the programs.
6. Modules can be developed in a general way so that these can be used with other programs. In this way, a library of standard and commonly used modules can be developed for future use.

Functions are self contained program segments that carry out some specific, well defined task. Every C program must have a function. One of the functions must be main().

Execution of program will always begin by carrying out the instructions in function main(). Additional functions will be subordinate to main() and also to one another.

There are two types of functions in C:

Library Function : These functions are predefined in C library and need is just to include the library.

User Defined Function : These needs to be developed by the programmer at the time of program writing as per the requirement.

7.1.1 Function declaration and prototype

There are two main parts of the function.

The function header and the function body.

```
int sum(int x, int y)
{
        int ans = 0;        //holds the answer that will be returned
        ans = x + y;        //calculate the sum
        return ans //return the answer
}
```

Function Header

In the first line of the above code

int sum(int x, int y)

It has three main parts

1. The name of the function i.e. *sum*
2. The parameters of the function enclosed in paranthesis
3. Return value type i.e. *int*

Function Body

What ever is written with in { } in the above example is the body of the function.

Function Prototypes

The prototype of a function provides the basic information about a function which tells the compiler that the function is used correctly or not. It contains the same information as the function header contains. The prototype of the function in the above example would be like

```
int sum (int x, int y);
```

The only difference between the header and the prototype is the semicolon ; there must the a semicolon at the end of the prototype.

7.1.2 Calling a Function

The call to a function in C simply entails referencing its name with the appropriate arguments. The C compiler checks for compatibility between the arguments in the calling sequence and the definition of the function.

Library functions are generally not available to us in source form. Argument type checking is accomplished through the use of header files (like stdio.h) which contain all the necessary information. For example, as we saw earlier, in order to use the standard mathematical library you must include math.h via the statement

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

at the top of the file containing your code. The most commonly used header files are

< stdio.h> -> defining I/O routines

< ctype.h> -> defining character manipulation routines

< string.h> -> defining string manipulation routines

< math.h> -> defining mathematical routines

< stdlib.h> -> defining number conversion, storage allocation and similar tasks

< stdarg.h> -> defining libraries to handle routines with variable numbers of arguments

< time.h> -> defining time-manipulation routines

In addition, the following header files exist:

< assert.h> -> defining diagnostic routines

< setjmp.h> -> defining non-local function calls

< signal.h> -> defining signal handlers

< limits.h> -> defining constants of the int type

< float.h> -> defining constants of the float type

Writing Your Own Functions

A function has the following layout:

```
return-type function-name ( argument-list-if-necessary )
{
```

```
    ...local-declarations...
    ...statements...
    return return-value;
}
```

If return-type is omitted, C defaults to int. The return-value must be of the declared type.

A function may simply perform a task without returning any value, in which case it has the following layout:

```
void function-name ( argument-list-if-necessary )
{
    ...local-declarations...
    ...statements...
}
```

As an example of function calls, consider the following code:

```
#include <stdio.h>
int add (int x, int y) {
    int z;
    z = x + y;
    return (z);
}
main ()
{
    int i, j, k;
    i = 10;
    j = 20;
    k = add(i, j);
    printf ("The value of k is %d\n", k);
}
```

Arguments are always passed *by value* in C function calls. This means that local "copies" of the values of the arguments are passed to the routines. Any change made to the arguments internally in the function are made only to the local copies of the arguments.

As an example, consider exchanging two numbers between variables. Lets illustrate what happen how the variables are passed by value:

```
#include < stdio.h>
void exchange(int a, int b);
void main()
```

```
{
   int a, b;
   a = 5;
   b = 7;
   printf("From main: a = %d, b = %d\n", a, b);
   exchange(a, b);
   printf("Back in main: ");
   printf("a = %d, b = %d\n", a, b);
}
void exchange(int a, int b)
{
   int temp;
   temp = a;
   a = b;
   b = temp;
   printf(" From function exchange: ");
   printf("a = %d, b = %d\n", a, b);
}
```

Run this code and observe that a and b are NOT exchanged! Only the copies of the arguments are exchanged. The RIGHT way to do this is of course to use pointers(will be discussed in next chapter).

7.1.3 Return statement

Data is returned from the function to the calling portion of the program via return statement. It causes control to be returned to the point from where the function was accessed. The return statement takes the following form :

Return(expression).

7.2 Scope of Function Variables

Only a limited amount of information is available within each function. Variables declared within the calling function can't be accessed unless they are passed to the called function as arguments. The only other contact a function might have with the outside world is through global variables.

Local variables are declared within a function. They are created anew each time the function is called, and destroyed on return from the function. Values passed to the function as arguments can also be treated like local variables.

Static variables are slightly different, they don't die on return from the function. Instead their last value is retained, and it becomes available when the function is called again.

7.3 Storage Classes :

Storage class defined for a variable determines the accessibility and longevity of the variable. The accessibility of the variable relates to the portion of the program that has access to the variable. The longevity of the variable refers to the length of time the variable exists within the program.

Types of Storage Class Variables in C:

- Automatic
- External
- Static
- Register

7.3.1 Automatic:

Variables defined within the function body are called automatic variables. Auto is the keyword used to declare automatic variables. By default and without the use of a keyword, the variables defined inside a function are automatic variables.

For instance:

```
void exforsys( )
{
auto int x;
auto float y;
.........
............
}
```

is same as

```
void exforsys( )
{
int x;
float y;          //Automatic Variables
.........
............
}
```

In the above function, the variable x and y are created only when the function exforsys( ) is called. An automatic variable is created only when the function is called. When the function exforsys() is called, the variable x and y is allocated memory automatically. When the function exforsys( ) is finished and exits the control transfers to the calling program, the memory allocated for x and y is automatically destroyed. The term automatic variable is used to define the process of memory being allocated and automatically destroyed when a function is called and returned. The scope of the automatic variables is only within the function block within which it is defined. Automatic variable are also called local variables.

7.3.2 External:

External variables are also called global variables. External variables are defined outside any function, memory is set aside once it has been declared and remains until the end of the program. These variables are accessible by any function. This is mainly utilized when a programmer wants to make use of a variable and access the variable among different function calls.

7.3.3 Static:

The static automatic variables, as with local variables, are accessible only within the function in which it is defined. Static automatic variables exist until the program ends in the same manner as external variables. In order to maintain value between function calls, the static variable takes its presence.

For example:

```
#include <iostream.h>
int exforsys(int);
void main( )
{
int in,out;
while(in!=0)
{
cout<<"Enter input value:";
cin>>in;
out=exforsys(in);
cout<<"\nResult:"<<out;
}
cout<<"\n End of Program"<<out;
}
int exforsys(int x)
{
static int a=0;
static int b=0;
a++;
b=b+x;
return(b/a);
}
```

In the above program, the static variables a and b are initialized only once in the beginning of the program. Then the value of the variables is maintained between function calls.

When the program begins, the value of static variable a and b is initialized to zero. The value of the input in is 5 (which is not equal to zero) and is then passed to the function in variable x. The variable a is incremented thus making a as equal to 1. Variable b becomes equal to 5 and thus, the return of value from function exforsys( ) for the first time is 5, which is printed in the called function.

The second time the value of the input in is 7 (which is not equal to zero) and is passed to the function in variable x. The variable a (which is declared as static) has the previous value of 1. This is incremented and the value of a is equal to 2. The value of b is maintained from the previous statement as 5 and new value of b now is b=5+7 = 12 and thus, the return value from the function is 12/2=6 which is printed in the called function.

7.3.4 Register Variable

Compiler can be informed that a variable should be kept in one of the machine's register, instead of keeping in the memory. Since a register access is much faster than a memoy access and keeping the frequently accessed variables in the register for the faster execution of the program.

7.4 Sample programs using functions

Following are some example programs using function :

Program 1: To check whether a given number is prime or not.

```
#include<stdio.h>
#include<conio.h>
#define prime 1
#define no_prime 0
int isprime(int number)
{       int count = 2;
        if (number==1)
                return no_prime;
        while (count<= number/2)
        {       if (number%count==0)
                        return no_prime;
                count++;
        }
        return prime;
}
void main()
{       int number;
        printf("enter a number");
        scanf("%d", &number);
        if (isprime(number))
                printf("%d is a prime number\n",number)
        else
                printf("%d is not a prime number\n",number);
}
```

Program 2: To reverse a number.

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

```
void main()
{       int number;
        long rnumber;
long reverse(int);
        printf("enter a number to be reversed");
        scanf("%d", &number);
        rnumber = reverse(number);
        printf("the reverse of %d is %ld\n", number, rnumber);
}
long  reverse(int n)
{       long r;
        r = 0;
        while (n>0)
        {       r = r * 10 + (n%10);
                n = n/10;
        }
        return r;
}
```

Program 3: To calculate the sum of first n odd numbers.

```
#include<stdio.h>
#include<conio.h>
int sumodd(int number)
{       int i, sum=0;
        for(i=0;i<number;i++)
                Sum = sum + (2*i+1);
        return sum;
}
void main()
{       int n;
        printf("enter how many odd numbers are to be added");
        scanf("%d", &n);
        printf("the sum of first %d odd numbers  is %d\n",n, sumodd(n));
}
```

Program 4: To calculate the power of a given number.

```
#include<stdio.h>
void main()
{       int a,b;
double p, power(int,int);
        printf("enter the value of a and b");
        scanf("%d %d ", &a, &b);
        p = power(a,b);
        printf("%lf",p);
}
double power (int x, int y)
{       int i;
        double p=1;
        for (i=1;i<=y;i++)
                p = x*p;
        return(p);
}
```

Program 5: To calculate the sum of Fibbonacci series of first n numbers.

```
#include<stdio.h>
#include<conio.h>
int fibonacci(int n)
{       static fn;
        fn = (n<3)?1:fibonacci(n-1) + fibonacci(n-2);
        return(fn);
}
void main()
{       int number, f ;
        printf("enter a number");
        scanf("%d", &number);
        f=fibonacci(number);
        printf("Sum is =>%d ",f);
}
```

7.5 Functions with array

At some moment it is needed to pass an array to a function as a parameter. In order to accept arrays as parameters the only thing that we have to do when declaring the function is to specify in its parameters the element type of the array, an identifier and a pair of void brackets [] with the size of the array . For example, the following function:

```
void procedure (int arg[], int n)
```

accepts a parameter of type "array of int" called arg and n is the size of array. In call by value, we pass the value of array element while in call by reference we pass the name of array, without any subscript, and size of array elements to the function.

Parameter passing by call by value :

```
#include<stdio.h>
#include<conio.h>
void display(int);
void main()
{       int i;
        static int num [ ] = {10,20,30,40,50};
        for(i=0;i<=4;i++)
                display(num(i));
                getch();
}
void display(int n)
{
        printf("%d",n);
}
```

7.6 Self Learning Exercise

State True/False

1. Global variables are declared inside the function.
2. Length of a program can be reduced using function.
3. Register variables are used for the fast access.
4. The scope of automatic variable is not confined to that function in which it is declared.

7.7 Summary

This chapter starts with the introduction of the functions. After this, function declaration, calling of function, scope of function variables is discussed. In next section various example programs are given for the better understanding of the functions. And at the end questions are given for the self assessment.

7.8 Further Readings

- Programming in C :- Balguruswamy, Tata McGraw-Hill Publication.
- The C programming Lang., Pearson Ecl :- Dennis Ritchie, Prentice HAll, Delhi.
- Let us C :- Yashwant Kanetkar, BPB publication

7.9 Answer to Self Learning Exercises

Question	Answer	Question	Answer
1	False	3	True
2	True	4	False

7.10 Unit End Questions

1. Write short note on
   (a) Library functions
   (b) User defined functions
2. Write the difference between global and local parameters.
3. Write a C program to calculate sum of first n even numbers using function.
4. Write a C program to find the sum of the digits of a number using function.

Unit 8 : Structures & Unions

8.0 Objectives

8.1 Introduction

8.2 Definition of Structure

8.3 Declaration of Structure Variables

8.3.1 With Structure Definition

8.3.2 Using Structure Tag

8.4 Initialization of Structure Variable

8.5 Accessing Fields Using Dot Operator

8.6 Union

8.6.1 Defining a Union

8.6.2 Declaration of Union Variable

8.6.3 Accessing of Union Members

8.6.4 Comparison with Structure

8.7 Array of User Define Data Types

8.8 Passing Structure Variables to Function

8.8.1 Passing Structure Variables as Arguments

8.8.2 Passing Structure Members as Arguments

8.9 Other User Define Data Type

8.9.1 Enumeration

8.9.2 typedef

8.0 Objectives

Array is a collection of same type of elements but in many real life applications we may need to group different types of logically related data. To store related fields of different data types we can use a structure, which is capable of storing heterogeneous data. Data of different types can be grouped together under a single name using structures.

8.1 Introduction

A structure can be considered as a template used for defining a collection of variables under a single name. Structures help a programmer in grouping elements of different data types into a single logical unit. This is unlike arrays which permit a programmer to group only elements of same data type. In some programming contexts, we need to access multiple data types under a single name for easy manipulation; for example if we want to refer to address with multiple data like house number, street, zip code, country C supports structure which allows us to wrap one or more variables with different data types. A structure can contain any valid data types like int, char, float even arrays and other structures. Each variable in structure is called a structure member.

8.2 Definition of a Structure

Definition of a structure creates a template of format that describes the characteristics of its members. All the variables that would be declared of this structure type, will take the form of this template. The general syntax of a structure definition is-

```
struct tagname {
        datatype member1;
        datatype member2;
        ........................
        ........................
        dataype member2;
};
```

Here **struct** is a keyword, which tells the compiler that a structure is being defined. Member1, member2, ..., memberN are known as members of structure and are declared inside curly braces. They should be a semicolon at the end of curly braces. These members can be of any data type like int, char, float, array, pointers or another structure type. tagname is the name of the structure and it is used further in the program to declare variables of this structure type.

It is important to remember that definition of a structure template does not reserve any space in memory for the members; space is reserved only when actual variables of this structure type are declared. Although the syntax of declaration of members inside the template is identical to the syntax we use in declaring variable but these members are not variables, they don't have any existence until they are attached with a structure variable. The member names inside a structure should be different from one another but these names can be similar to any other variable name declared outside the structure. The member names of two different structures may also be same.

Consider an example for declaring a structure template:

```
struct student {
        char name[20];
        int rollno;
        float marks;
};
```

Here, student is the structure tag and there are three members of this structure (name, rollono and marks). Structure template can be defined globally or locally i.e. it can be placed before all functions in the program or it can be locally present in a function. If the template is global then it can be used by all functions while if it is local then only the function containing it can use it.

8.3 Declaration of Structure Variable

We can declare structure variables in two ways:

1. With structure definition
2. Using the structure tag

8.3.1 With structure definition

```
struct student {
        char name[20];
        int rollno;
        float marks;
    } std1, std2, std3;
```

Here std1, std2 and std3 are variables of type struct student. When we declare a variable while defining the structure template, the tagname is optional. So we can also declare them as:

```
struct {
        char name[20];
        int rollno;
        float marks;
    } std1, std2, std3;
```

If we declare variable in this way, then we'll not be able to declare other variable of this structure type anywhere else in the program nor can we send these structure variable to functions. So although the tagname is optional it is better to specify a tagname for the structure.

8.3.2 Using structure tag

```
struct student {
        char name[20];
        int rollno;
        float marks;
    };
struct student std1, std2;
struct student std3;
```

Here std1, std2 and std3 are structure variables that are declared using the structure tag student. Declaring a structure variable reserves space in memory. Each structure variable declared to be of type struct student has three members (name, rollno and marks). The compiler will reserve space for each variable sufficient to hold all the members. For this example each variable of type struct student will occupy 26 (20+2+4) bytes.

8.4 Initialization of structure variables

The syntax of initializing structure variable is similar to that of arrays. All the values are given in curly braces and the number, order and type of these values should be same as in the structure template definition. The initializing values can only be constant expressions.

```
struct student {
        char name[20];
        int rollno;
```

```
                float marks;
        } std1={"Manoj", 25, 98};
struct student std2={"Deep", 24, 70.5};
```

We cannot initialize members while defining the structure.

```
struct student {
                char name[20];
                int rollno;
                float marks=99;       /* Invalid */
        } std1;
```

This is invalid because there is no variable called marks, and no memory is allocated for structure definition.

If the number of initializers is less than the number of members then the remaining members are initialized with zero. For example if we have this initialization:

```
struct student std1={"Manoj"};
```

Here the members rollno and marks of std1 will be initialized to zero. This is equivalent to the initialization:

```
struct student std1={"Manoj", 0, 0};
```

8.5 Accessing fields using dot operator

For accessing any member of a structure variable, we use the dot (.) operator which is also known as the period or membership operator. The format for accessing a structure member is:

structurevariable.member

For example, consider the following structure:

```
struct student {
                char name[20];
                int rollno;
                float marks;
        };
struct student std1, std2;
```

name of std1 is given by- std1.name

rollno of std1 is given by- std1.rollno

marks of std1 is given by- std1.marks

name of std2 is given by- std2.name

rollno of std2 is given by- std2.rollno

marks of std2 is given by- std2.marks

We can use std1.name, std1.marks, std2.marks etc like any other ordinary variables in the program. They can be read, displayed, processed, assigned values or can be sending to functions as arguments. We can't use student.name or student.rollno because student is not a structure variable, it is a structure tag.

```
/* Program to display the values of structure members */
#include <stdio.h>
#include <string.h>
structure student {
                    char name[20];
                    int rollno;
                    float marks;
                };
main()
{
        struct student std1={"Manoj",25,79.5};
        struct student std2,std3;
        strcpy (std2.name, "Deep");
        std2.rollno=24;
        std2.marks=98;
        printf("Enter name, rollno and marks for std3 : ");
        scanf(" %s %d %f", std3.name, &std3.rollno, &std3.marks);
        printf("std1 : %s %d %f\n", std1.name, std1.rollno, std1.marks);
        printf("std2 : %s %d %f\n", std2.name, std2.rollno. std2.marks);
        printf("std3 : %s %d %f\n", std3.name, std3.rollno, std3.marks);
}
```

Output:

```
Enter name, rollno and marks for std3 : Anil 27 87.9
    std1 : Manoj 25 79.5
    std2 : Deep 24 98
    std3 : Anil 27 87.9
```

Note that since std2.name is an array so we can't assign a string to it using assignment operator, hence we have used the strcpy () function.

The dot operator is one of the highest precedence operators; its associativety is from left to right. Hence it will take precedence over all other unary, relational, logical, arithmetic and assignment operators. So in an expression like ++std.marks, first std.marks will be accessed and then its value will be increased by 1.

8. 4 Union

Union is a derived data type like structure and it can also contain members of different data types. The main difference between union and structure is in the way memory is allocated for the members. In a structure each member has its own memory location, whereas members of union share the same memory location. When a variable of type union is declared, compiler allocates sufficient memory to hold the largest memory in the union. Since all members share the same memory location hence we can use only one member at a time. Thus union is used for saving memory. The concept of union is useful when it is not necessary to use all members of union at a time.

8.6.1 Defining a Union

The syntax of definition of a union is:

```
union union_name {
                datatype member1;
                datatype member2;
                .......................
                .......................
                datatype memberN;
        }
```

8.6.2 Declaration of union variable

Like, structure variables, the union variables can be declared along with the definition or separately. For example:

```
union union_name {
                datatype member1;
                datatype member2;
                .......................
                .......................
                datatype memberN;
        } variable_name;
```

This can also be declared as:

```
union union_name variable_name;
```

8.6.3 Accessing of union members

We can access the union members using the same syntax used for structure. For example:

```
/* program for accessing union members */
#include <stdio.h>
main()
```

```
{
    union result {
                    int marks;
                    char grade;
                    float per;
            } res;
    res.marks=90;
    printf(" Marks : %d \n ", res.marks);
    res.grade='A';
    printf("Grade : %c \n ", res.grade);
    res.per= 85.5;
    printf("Percentage : %f\n ", res.per);
}
```

Output:

Marks : 90

Grade : A

Percentage : 85.5

Before the first printf, the value 90 is assigned to the union member marks, so other members grade and per contain garbage value. After first printf, the value 'A' is assigned to the union member grade. So now the other two members marks and per contain garbage value. Only one member of union can hold value at a time, don't try to use all the members simultaneously. So a union variable of type result can be treated as either an int variable or char variable or float variable. It is the responsibility of the programmer to keep track of member that currently hold the value.

Union variables can also be initialized, but there is a limitation. We know that due to sharing of memory, all the members can't hold values simultaneously. So during initialization also only one member can be given an initial value, and this privilege is given to the first member. Hence only the first member of a union can be given an initial value. The type of the initializer should match with the type of the first member. For example, we can initialize the above union variable as:

```
union result res = { 76 };
```

8.6.4 Comparison with structure

The main difference between union and structure is in the way memory is allocated for the members. The syntax used for definition of a union, declaration of union variables and for accessing members is similar to that used in structures, but here keyword union is used instead of struct. Now we'll consider a program to compare the memory allocated for a union and structure variable.

```
/* program to compare the memory allocated for a union and structure variable */
#include <stdio.h>
struct stag {
```

```
        char c;
        int i;
        float f;
    };
union utag {
        char c;
        int i;
        float f;
    };
main ()
{
    union utag uvar;
    struct stag svar;
    printf (" Size of svar = %u \n ", sozeof (svar));
    printf (" Address of svar : %u \t ", &svar);
    printf (" Address of members : %u %u %u \n ", &svar.c,&svar.i, &svar.f);
    printf (" Size of uvar = %u \n ", sozeof (uvar));
    printf (" Address of uvar : %u \t ", &uvar);
    printf (" Address of members : %u %u %u \n ", &uvar.c,&uvar.i, &uvar.f);
}
```

Output:

```
    Size of svar = 7
    Address of svar : 65514 Address of members : 65514 65515 65517
    Size of uvar = 4
    Address of uvar : 65522 Address of members : 65522 65522 65522
```

The addresses of members of a union are same while the addresses of members of a structure are different. The difference in the size of variables svar and uvar also indicates that union is very economical in the use of memory.

8.7 Array of User Define Datatype

We know that array is a collection of elements of same datatype. We can declare array of structures where each element of array is of structure type. We can be declared as:

struct student std[10];

Here std is an array of 10 elements, each of which is a structure of type struct student, means each element of std has members, define in structure. These structures can be accessed through subscript notation. To access the individual members of these structures we'll use the dot operator as usual. For example:

```
/* Program to understand array of structures */
#include <stdio.h>
struct student {
                    char name[20];
                    int rollno;
                    float marks;
                };
main ()
{
    int i;
        struct student std[10];
        for (i=0; i<10; i++)
        {
            printf(" Enter name, rollno and marks : ");
            scanf(" %s %d %f", std[i].name, &std[i].rollno, &std[i].marks);
        }
        for (i=0; i<10; i++)
        {
            printf (" %s %d %f\n", std[i].name, std[i].rollno, std[i].marks);
        }
}
```

8.8 Passing Structure variables to Functions

Structures may be passed as arguments to function in different way. We can pass individual members, whole structure variable to the function. Similarly a function can return either a structure member or whole structure variable.

1.6.1 Passing structure variable as arguments

We can pass structure variables as arguments to function. For example:

```
#include <stdio.h>
struct student {
                    char name[20];
                    int rollno;
                    float marks;
                };
display (struct student);
```

```
main()
{
        struct student std1={"Manoj",12,87);
        struct student std2={"Deep",18,90);
        display (std1);
        display (std2);
}
display (struct student std)
{
        printf(" Name - %s \t ", std.name);
        printf(" Rollno - %d \t ", std.rollno);
        printf(" Marks - %f \n ", std.marks);
}
```

Output:

```
        Name - Manoj Rollno - 12 Marks - 87
        Name - Deep Rollno - 18 Marks – 90
```

Here it is necessary to define the structure template globally because it is used by both functions to declared variables.

8.8.2 Passing structure members as arguments

We can pass individual structure members as arguments to functions like any other ordinary variable

```
/* Program to understand how structure members are sent to a function */
#include <stdio.h>
#include <string.h>
structure student {
                        char name[20];
                        int rollno;
                        float marks;
                    };
display (char name[ ], int rollno, float marks);
main ()
{
        struct student std1 = {"Manoj", 12, 87};
        struct student std2;
        strcpy (std2.name, "Deep");
```

```
        std2.rollno=18;
        std2.marks=90;
        display (std1.name, std1.rollno, std1.marks);
        display (std2.name, std2.rollno, std2.marks);
    }
display (char name[ ], int rollno, float marks)
  {
        printf(" Name - %s \t ", name);
        printf(" Rollno - %d \t ", rollno);
        printf(" Marks - %f \n ", marks);
    }
```

Output:

```
        Name - Manoj Rollno - 12 Marks - 87
        Name - Deep Rollno - 18 Marks – 90
```

Here we have passed members of the variables std1 and std2 to the function display(). The names of the formal arguments can be similar to the names of the members. We can pass the arguments using call by reference also so that the changes made in the called function will be reflected in the calling function. In that case we'll have to send the address of the members. It is also possible to return a single member from a function.

8.9 User Define Data Types

We have already seen the data types that are defined by the user: the structures and the union. But in addition to these there are other kinds of user defined data types:

8.9.1 Enumeration

Enumerated data types are a user defined ordinal data type. The main purpose of the enumerated data type is to allow numbers to be replaced by words. This is intended to improve the readability of programs. The general format of definition is:

```
enum data_type_name { word1, word2, …, word(n-1), word(n) };
```

OR

```
enum data_type_name { word1 = integer1, word2 = integer2, etc… };
```

here enum is a keyword, data_type_name is an identifier that sepecifies the name of the new enumeration type being defined, and word1 ,word2,……..,word(n) are identifiers which represent integer constants and are called enumeration constants or enumerators. The list of these enumerators is called enumerator list. Note that unlike structure and union, here the members inside the braces are not variables, they are named integer constants.

After the definition, we can declare variables of this new data type as:

```
enum data_type_name var1, var2,……., varN;
```

Here var1, var2, ...,varN are variables of type enum. These variables can take values only from the enumerator list.

The variables can also be declared with the definition as:

```
enum data_type_name {
                    word1;
                    word2;
                    ......
                        wordN;
                } var1, var2,.... varN;
```

Here the data_type_name is optional. Consider an example:

```
enum month { Jan, Feb, Mar, Apr, May, Jun};
```

here a new data type month is defined and the enumerator list contains six enumerators.

Initially the compiler treats enumerators as integer constants. These are automatically assigned intiger values beginning from 0, 1, 2,.....etc till the last member of the enumeration. In the above example these enumerators will take following values:

Jan	0
Feb	1
Mar	2
Apr	3
May	4
Jun	5

These are the default values assigned to the enumerators. It is also possible to explicitly assign any value to enumerators but in this case, the successive unassigned enumerators will take values one greater than the value of the previous enumerator. For example:

```
enum month { Jan, Feb=4, Mar, Apr, May=11, Jun};
```

Now the enumerators will take following values:

Jan	0
Feb	4
Mar	5
Apr	6
May	11
Jun	12

We can assign any signed integer value to enumerators, provided the value is within the range of int. It is also possible to assign same value to more than one enumerator.

```
enum month { Jan, January=0,Feb=1, February=1, Mar=2, March=2};
```

the enumerated variables can be processed like other integer variables. We can assign values to them from the enumerator list or they can be compared to other variables and values of the same type. For example:

```
enum month { Jan, Feb, Mar, Apr, May, Jun} m1, m2;
m1=Mar;
m2=May;
```

Now m1 has integer value 2 and m2 will take the value 4.

```
/* Program to print the value of enum variables */
#include <stdio.h>
main()
{
    enum month { Jan, Feb, Mar, Apr, May, Jun } m1, m2;
    m1=Mar;
    printf(" m1 = %d \n", m1);
    printf(" Enter value for m2 : ");
    scanf("%d", &m2);
    printf(" m2 = %d \n", m2);
}
```

Output:

```
m1 = 2
Enter the value for m2 : 5
m2 = 5
```

8.9.2 typedef

The typedef definition facility allow us to define a new name for an existing data type. the general syntax is:

```
typedef data_type new_name;
```

here typedef is a keyword, data_type ia any existing data type that may be a standard data type or a user defined type, new_name is an identifier, which is a new name of existing data type. For example, we can define a new name for int type by writing:

```
typedef int marks;
```

now marks is a synonym for int and we can use marks instead of int anywhere in the program, for example:

```
marks sub1, sub2;
```

Here sub1, sub2 are actually int variables and are similar to any variable declared using int keyword.

The above declaration is equivalent to:

```
int sub1, sub2;
```

We can give more than one name to a single data type using only one typedef statement. For example;

typedef int age, marks, units;

Now we'll see how typedef can be used to define new names for arrays, functions and structures.

1. Arrays

typedef int intarr[10];

After this statement intarr is another name for integer arrays of size 10. Now consider this declaration statement;

intarr a, b, c[15]; (Equivalent to ': int a[10], b[10], c[15][10])

Here a, b are declared as 1-D arrays of size 10, and c is declared as 2-D array of size 15x10.

2. Function

typedef float funct(float, int);

Here funct is any function that take two values, one float and one int and return a float value.

Now consider this declaration statement:

funct add, sub,mul,div;

Here add, sub, mul, div are declared as functions that take a float and int value and return a float value.

The above statement is equivalent to the following declaration statements:

float add (float, int);

float sub (float, int);

float mul (float, int);

float div (float, int);

3. Structure

Similarly we can also use typedef for defining a new name for structures. Suppose we have this structure definition;

```
struct studentrec{
        char name[20];
        int marks;
    };
```

Now whenever we want to use this structure we have to write struct studentrec. We can give a short and meaningful name to this structure by typedef.

typedef struct studentrec Student;

Now we can declare variable like this;

Student std1, std2; (Equivalent to ': struct studentrec std1, std2;)

We can also combine typedef and structure definition. The syntax is as:

```
typedef struct tagname{
                    datatype mamber1;
                    ................
                    ................
              } newname;
```

Here tagname can be same as the newname. we can omit the tagname if the structure is not self refrential.

```
typedef struct {
                    char name[20];
                    int age;
            } person;

person student, teacher, emp;
```

Here person is a new name for this structure and we have define three structure variables, which have the format of the above definition.

Advantages of using typedef

1. It makes our programs more readable and understandable since we can document our program by giving meaningful and descriptive names for existing types.
2. In structures it is important since we can give a single name to the structure, we need not write struct keyword repeatedly.
3. It makes our programs more portable. When program is run on a different machine on which standard data types are represented by different number of bytes, only typedef statement has to be changed.

Unit 9 : Pointers

9.0 Objectives

9.1 Introduction

9.2 Definition of Pointer

9.3 Declaration of Pointer

9.4 Memory Allocation to Pointers

9.5 Pointers to an Array

9.6 Pointer Arithmetic

9.7 Pointer to data type define by user using structure

9.8 Pointers and function

9.8.1 Call by value

9.8.2 All by reference

9.9 Dynamic Memory Allocation

9.9.1 Memory allocations process

9.9.2 Allocating a block of memory

9.9.3 Allocating multiple blocks of memory

9.9.4 Releasing the used space

9.9.5 To alter the size of memory

9.10 Summary

9.11 Unit end questions

9.0 Objectives

C is a very powerful language and the real power of C lies in pointers. The concept of pointers is interesting as well as challenging. It is very simple to use pointers provided the basics are understood thoroughly. So it is necessary to visualize every aspect of pointers instead of just having a superficial knowledge about their syntax and usage. The use of pointers makes the code more efficient and compact. Some of the uses of pointers are:

(i) Accessing array elements.

(ii) Returning more than one value from a function.

(iii) Accessing dynamically allocated memory.

(iv) Implementing data structures like linked list, trees and graphs.

9.1 Introduction

Pointers are an extremely powerful programming tool. They can make some things much easier, help improve our program's efficiency, and even allow us to handle unlimited amounts of data. For example, using pointers is one way to have a function modify a variable passed to it. It is also possible to use pointers to dynamically allocate memory, which means that we can write programs that can handle nearly unlimited amounts of data on the fly. We don't need to know, when we write the program, how much memory we need. In C a pointer is a variable that points to or references a memory location in which data is stored.

Each memory cell in the computer has an address that can be used to access that location so a pointer variable points to a memory location we can access and change the contents of this memory location via the pointer.

9.2 Definition of Pointer

To understand pointers, it helps to compare them to normal variables.

A "normal variable" is a location in memory that can hold a value. For example, when we declare a variable **i** as an integer, four bytes of memory are set aside for it. In our program, we refer to that location in memory by the name **i**. At the machine level that location has a memory address. The four bytes at that address are known to us, the programmer, as **i**, and the four bytes can hold one integer value.

A pointer is different. A pointer is a variable that **points** to another variable. This means that a pointer holds the memory address of another variable. Put another way, the pointer does not hold a value in the traditional sense; instead, it holds the address of another variable. A pointer "points to" that other variable by holding a copy of its address.

Because a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address. That addresses points to a value. There is the pointer and the value pointed to.

9.3 Declaration of Pointer

Like other variables, pointer variables should also be declared before being used. The general syntax of declaration is:

```
data_type *pointer_name;
```

Here pointer_name is the name of pointer variable, which should be a valid C identifier. The asterisk '*' preceding this name informs the compiler that the variable is declared as a pointer. Here data_type is known as the base type of pointer.Let us take some pointer declarations:

```
int *iptr;

float *fptr;

char *cptr,ch1,ch2;
```

here iptr is a pointer that should point to variables of type int, similarly fptr and cptr should point to variables of float and char type respectively. Here typeof variable iptr is 'pointer to int' or (int *), or we can say that base type of iptr is int. we can also combine the declaration of simple variables and pointer variables as we have done in the thired declaration statement where ch1 and ch2 are declared as variables of type char.

Pointers are also variables so compiler will reserve space for them and they will also have some address. All pointers irrespective of their base type will occupy same space in memory since all of them contain addresses only. Generally 2 bytes are used to store an address (may vary in different computers)., so the compiler allocates 2 bytes for a pointer variable.

9.4 Memory Allocation to Pointers

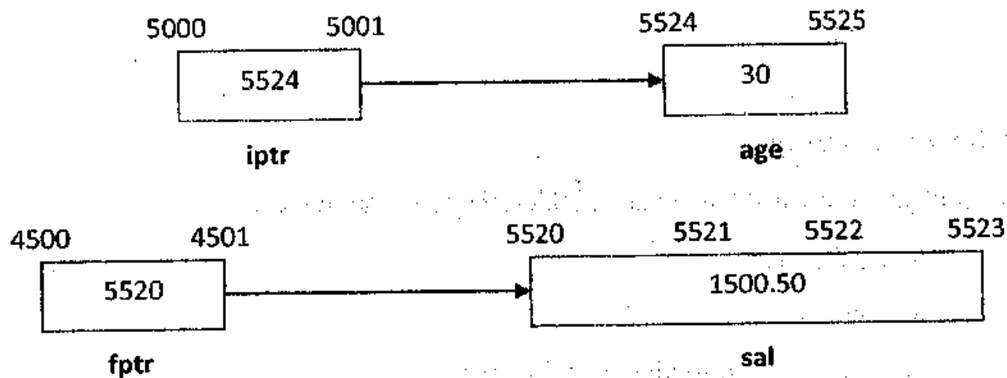
When we declare a pointer variable it contains garbage value i.e. it may be pointing anywhere in the memory. So we should always assign an address before using it in the program. The use of an unassigned pointer may give unpredictable results and even cause the program to crash. Pointers may be assigned the address of a variable using assignment statement. For example:

```
int *iptr, age = 30;

float *fptr, sal = 1500.50;
```

```
iptr = &age;
fptr = &sal;
```

Now iptr contains the address of variable age i.e. it points to variable age, similarly fptr points to variable sal. Since iptr is declared as a pointer of type int, we should assign address of only integer variables to it. If we assign address of some other data type then compiler won't show any error but the output will be incorrect.



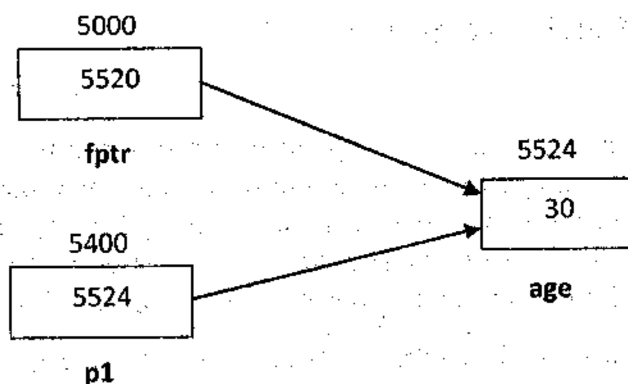
We can also initialize the pointer at the time of declaration. But in this case the variable should be declared before the pointer. For example:

```
int age = 30, *iptr = &age;
float sal = 1500.50, *fptr = &sal;
```

It is also possible to assign the value of one pointer variable to the other, provided their base type is same. For example, if we have an integer pointer p1 then we can assign the value of iptr to it as:

```
p1=iptr;
```

Now both pointer variable iptr and p1 contain the address of variable age and point to the same variable age.



We can assign constant zero to a pointer of any type. a symbolic constant NULL is defined in stdio.h, which denotes the value zero. The assignment of NULL to a pointer guarantees that it does not point to any valid memory location. This can be done as:

```
ptr = NULL;
/* Program to understand the use of pointer variable */
#include <stdio.h>
main()
```

```
{
  int x;     /* A normal integer*/
  int *p;    /* A pointer to an integer ("*p" is an integer, so p must be a pointer to an integer) */
  p = &x;    /* Read it, "assign the address of x to p" */
  scanf( " Enter the value of x = %d", &x );      /* Put a value in x, we could also use p here */
  printf( "The value of pointer p is : %d\n", *p );    /* Note the use of the * to get the value */
      getchar();
}
```

9.5 Pointer to an Array

Array elements are just like other variables: they have addresses.

```
int ar[20], *ip;
ip = &ar[5];
*ip = 0;      /* equivalent to ar[5] = 0; */
```

The address of ar[5] is put into ip, then the place pointed to has zero assigned to it. By itself, this isn't particularly exciting. What is interesting is the way that pointer arithmetic works. Although it's simple, it's one of the cornerstones of C.

Adding an integral value to a pointer results in another pointer of the same type. Adding n gives a pointer which point n elements further along an array than the original pointer did. (Since n can be negative, subtraction is obviously possible too.) In the example above, a statement of the form

```
*(ip+1) = 0;
```

would set ar[6] to zero, and so on. Again, this is not obviously any improvement on 'ordinary' ways of accessing an array, but the following is.

```
int ar[20], *ip;
    for (ip = &ar[0]; ip < &ar[20]; ip++)
    *ip = 0;
```

That example is a classic fragment of C. A pointer is set to point to the start of an array, then, while it still points inside the array, array elements are accessed one by one, the pointer incrementing between each one. The Standard endorses existing practice by guaranteeing that it's permissible to use the address of ar[20] even though no such element exists. This allows us to use it for checks in loops like the one above. The guarantee only extends to one element beyond the end of an array and no further.

Why is the example better than indexing? Well, most arrays are accessed sequentially. Very few programming examples actually make use of the 'random access' feature of arrays. If we do just want sequential access, using a pointer can give a worthwhile improvement in speed. In terms of the underlying address arithmetic, on most architecture it takes one multiplication and one addition to access a one-dimensional array through a subscript. Pointers require no arithmetic at all—they nearly always hold the store address of the object that they refer to. In the example above, the only arithmetic that has to be done is in the for loop, where one comparison and one addition are done each time round the loop. The equivalent, using indexes, would be this:

```
int ar[20], i;
```

```
for (i = 0; i < 20; i++)
    ar[i] = 0;
```

The same amount of arithmetic occurs in the loop statement, but an extra address calculation has to be performed for every array access.

Efficiency is not normally an important issue, but here it can be. Loops often get traversed a substantial number of times, and every microsecond saved in a big loop can matter. It isn't always easy for even a smart compiler to recognize that this is the sort of code that could be 'pointerized' behind the scenes, and to convert from indexing (what the programmer wrote) to actually use a pointer in the generated code.

To be honest, C doesn't really 'understand' array indexing, except in declarations. As far as the compiler is concerned, an expression like x[n] is translated into *(x+n) and use made of the fact that an array name is converted into a pointer to the array's first element whenever the name occurs in an expression. That's why, amongst other things, array elements count from zero: if x is an array name, then in an expression, x is equivalent to &x[0], i.e. a pointer to the first element of the array. So, since *(&x[0]) uses the pointer to get to x[0], *(&x[0] + 5) is the same as *(x + 5) which is the same as x[5]. A curiosity springs out of all this. If x[5] is translated into *(x + 5), and the expression x + 5 gives the same result as 5 + x (it does), then 5[x] should give the identical result to x[5]. Let us see an example that compiles and runs successfully:

```
#include <stdio.h>
#include <stdlib.h>
#define ARSZ 20
main()
{
    int ar[ARSZ], i;
    for(i = 0; i < ARSZ; i++)
    {
    ar[i] = i;
    i[ar]++;
  printf("ar[%d] now = %d\n", i, ar[i]);
    }
    printf("15[ar] = %d\n", 15[ar]);
    exit (EXIT_SUCCESS);
}
```

9.6 Pointer Arithmetic

All types of arithmetic operations are not possible with pointers. The only valid operations that can be performed are as:

(1) Addition of an integer to a pointer and increment operation.

(2) Subtraction of an integer from a pointer and decrement operation.

(3) Subtraction of a pointer from another pointer of same type.

Pointer arithmetic is somewhat different from ordinary arithmetic. Here all arithmetic is performed relative to the size of base type of pointer. For example if we have an integer pointer pi which contains address 1000 then on incrementing we get 1002 instead of 1001. This is because the size of int data type is 2. Similarly on decrementing pi, we will get 998 instead of 999. The expression (pi+3) will represent the address 1006. Let us see pointer arithmetic for int, float and char pointers.

```
int a =5, *pi = &a;
float b = 2.2, *pf = &b;
char c ='x', *pc = &c;
```

suppose the address of variables a, b and c are 1000, 4000, 5000 respectively, so initially values of p1, p2, p3 will be 1000, 4000 and 5000.

```
pi++; or ++pi;  pi = 1000 + 2 = 1002 (since int is of 2 bytes)
pi = pi - 3;    pi = 1002 - 3*2 = 996
pi = pi + 5;    pi = 996 + 5*2 = 1006
pi—; or —pi;    pi = 1006 - 2 = 1004

pf++; or ++pf;  pf = 4000 + 4 = 4004 (since float is of 4 bytes)
pf = pf - 3;    pf = 4004 - 3*4 = 3992
pf = pf + 5;    pf = 3992 + 5*4 = 4012
pf—; or —pf;    pf = 4012 - 4 = 4008
pc++; or ++pc;pc = 5000 +1 = 5001 (since char is of 1 bytes)
pc = pc - 3;    pc = 5001 - 3*1 = 4998
pc = pc + 5;    pc = 4998 + 5*1 = 5003
pc—; or —pc; pc = 5003 - 1 = 5002
```

The compiler scales all this arithmetic automatically since it knows the base type of pointer. The arithmetic in the case of char pointer seems to be like ordinary arithmetic because the size of a char The addresses of variables a, b and c are not affected by these operations, only the pointer moves ahead of backward.

```
/* Program to show pointer arithmetic */
#include <stdio.h>
main()
{
    int a=5, *pi=&a;
    char b='x', *pc=&b;
    float c=5.5, *pf=&c;
    printf("value of pi = Address of a = %u \n", pi);
```

```
printf("value of pc = Address of a = %u \n", pc);
printf("value of pf = Address of a = %u \n", pf);
pi++;
pc++;
pf++;
printf("Now value of pi = %u \n", pi);
printf("Now value of pc = %u \n", pc);
printf("Now value of pf = %u \n", pf);
printf("value of pi = %u \n" ++pi);
printf("value of pi = %u \n" ++pi);
printf("value of pi = %u \n" pi++);
printf("value of pi = %u \n" —pi);
printf("value of pi = %u \n" pi—);
printf("value of pi = %u \n" pi);
}
```

Output:

```
Value of pi = Address of a = 1000
Value of pi = Address of b = 4000
Value of pi = Address of c = 8000
now value of pi = 1002
now value of pc = 4001
now value of pf = 8004
value of pi = 1004
value of pi = 1004
value of pi = 1004
value of pi = 1004
```

ontaining addresses 3000 and 3010 respectively then ptr2-ptr1 will give 5. (Science size of int is 2). This operation is generally performed when both pointer variables point to the elements of same array.

Subtraction of a pointer of a pointer from another pointer of same base type returns an integer, which denotes the number of elements between two pointers. If we have two int pointers ptr1 and ptr2, containing addresses 3000 and 3010 respectively then ptr2-ptr1 will give 5. (Science size of int is 2). This operation is generally performed when both pointer variables point to the elements of same array.

The arithmetic operations that can never be performed on pointers are:

(1) Addition, multiplication, division of two pointers.

(2) Multiplication between pointer and any number.

(3) Division of a pointer by any number.

(4) Addition of float or double values to pointers.

9.7 Pointer to data type define by user using structure

We have studied that pointer is a variable which holds the starting address of another variable of any data type like int, float or char. Similarly we can have pointer to structure, which can point to the starting address of a structure variable. These pointers are called structure pointers and can be declared as:

```
struct student{
        char name[20];
        int rollno;
        int marks;
        };
struct student std, *ptr;
```

Here ptr is a pointer variable that can point to a variable of type struct student. We'll use the & operator to access the starting address of a structure variable, so ptr can point to std by writing;

```
ptr = 7std;
```

There are two ways of accessing the members of structure through the structure pointers.

As we know ptr is a pointer to a structure, so by dereferencing it we can get the contents of structure variable. Hence 8ptr will give the contents of std. so to access members of a structure variable std we can write:

```
(*ptr).name
(*ptr).rollno
(*ptr).marks
```

Here parentheses are necessary because dot operator has higher precedence that the * operator. This syntax is confusing so C has provided another facility of accessing structure members through pointers. We can use the arrow operator (->) which is formed by hyphen symbol and greater than symbol. So we can access the members as:

```
ptr->name
ptr->rollno
ptr->marks
```

The arrow operator has same precedence as that of dot operator and it also associates from left to right.

```
/* program to understand pointers to structures */
#include<stdio.h>
struct student{
        char name[20];
```

```
            int rollno;
            int marks;
        };
main()
{
    struct student std = {"Manoj", 25, 68};
    struct student *ptr = &std;
    printf(" Name - %s \t", ptr->name);
    printf(" Rollno - %s \t", ptr->rollno);
    printf(" Marks - %s \t", ptr->marks);
}
```

We can also have pointers that point to individual members of a structure variable. For example:

```
int *p = &std.rollno;
float *ptr = &std.marks;
```

The expression &std.rollno is equivalent to &(std.rollno) because the precedence of dot operator is more than that of address operator.

A pointer can also be used as amember of structure. For example, we can define a structure like this:

```
struct student {
            char name[20];
            int *ptr;
        };
struct student std, 8stdptr = &std;
```

Here ptr is a pointer to int and is a member of the structure student.

To access the value of ptr, we'll write:

```
std.ptr or stdptr->ptr
```

To access the value pointed to by std.ptr, we'll write:

```
*std.ptr or *stdptr->ptr
```

Since the priority of dot and arrow operator is more than that of dereference operator, hence the expression 8std,ptr is equivalent to *(std.ptr), and the expression 8std->ptr is equivalent to *(stdptr->ptr).

9.8 Pointers and function

The pointers are very much used in a function declaration. Sometimes only with a pointer a complex function can be easily represented and success. The usage of the pointers in a function definition may be classified into two groups.

1. Call by value.
2. Call by reference.

9.8.1 Call by value

We have seen that a function is invoked there will be a link established between the formal and actual parameters. A temporary storage is created where the value of actual parameters is stored. The formal parameters picks up its value from storage area the mechanism of data transfer between actual and formal parameters allows the actual parameters mechanism of data transfer is referred as call by value. The corresponding formal parameter represents a local variable in the called function. The current value of corresponding actual parameter becomes the initial value of formal parameter. The value of formal parameter may be changed in the body of the actual parameter. The value of formal parameter may be changed in the body of the subprogram by assignment or input statements. This will not change the value of actual parameters.

```
/* Program to Understand call by value using pointers */
#include < stdio.h >
main()
{
        float add(int, float),result;
        float (*fp) (int, float);
        fp=add;         /* Assign address of function add() to pointer fp */
        /*Invoking a function directly using function's name */
        result = add(5, 6.2);
        printf("%f\n", result);
/* Invocking a function indirectly by dereferencing function pointer */
        result = (*fp) (5, 6.2);
        printf("%f\n", result);
}
float add(int a, float b)
{
        return (a + b);
}
Output:
        11.600000
        11.600000
```

9.8.2 Call by Reference

When we pass address to a function the parameters receiving the address should be pointers. The process of calling a function by using pointers to pass the address of the variable is known as call by reference. The function which is called by reference can change the values of the variable used in the call.

```
/* Program to understand the concept of call by reference using pointers*/
#include < stdio.h >
```

```
main()
{
        int x, y;
        x=20;
        y=30;
        printf("\n Value of a and b before function call = %d %d", a, b);
        fncn(&x, &y);
                printf("\n Value of a and b after function call = %d %d", a, b);
}
fncn(p,q)
int p,q;
{
        *p = *p + *p;
        *q = *q + *q;
}
```

9.9 Dynamic memory allocation

The process of allocating memory at run time is known as dynamic memory allocation. Although C does not inherently have this facility there are four library routines which allow this function.

Many languages permit a programmer to specify an array size at run time.

Such languages have the ability to calculate and assign during executions, the memory space required by the variables in the program. But C inherently does not have this facility but supports with memory management functions, which can be used to allocate and free memory during the program execution. The following functions are used in C for purpose of memory management.

Function	**Task**
malloc	Allocates memory requests size of bytes and returns a pointer to the Ist byte of allocated space
calloc	Allocates space for an array of elements initializes them to zero and returns a pointer to the memory
free	Frees previously allocated space
realloc	Modifies the size of previously allocated space.

9.9.1 Memory allocations process

According to the conceptual view the program instructions and global and static variable in a permanent storage area and local area variables are stored in stacks. The memory space that is located between these two regions in available for dynamic allocation during the execution of the program. The free memory region is called the heap. The size of heap keeps changing when program is executed due to creation and death of variables that are local for functions and blocks. Therefore it is possible to encounter memory overflow during dynamic allocation process. In such situations, the memory allocation functions mentioned above will return a null pointer.

9.9.2 Allocating a block of memory

A block mf memory may be allocated using the function malloc. The malloc function reserves a block of memory of specified size and returns a pointer of type void. This means that we can assign it to any type of pointer. It takes the following form:

ptr = (cast-type*) malloc (byte-size);

ptr is a pointer of type cast-type the malloc returns a pointer (of cast type) to an area of memory with size byte-size. For example:

x = (int *) malloc (100 * sizeof(int));

On successful execution of this statement a memory equivalent to 100 times the area of int bytes is reserved and the address of the first byte of memory allocated is assigned to the pointer x of type int.

```
/* Program to understand dynamic allocation of memory */
#include<stdio.h>
#include<alloc.h>
main()
{
        int *p, n, i;
        printf("Enter the number of integers to be entered : ");
        scanf("%d", &n);
        p= (int *) malloc(n*sizeof(int));
        if(p==NULL)
        {
                                printf("Memory not available \n");
                exit(1);
        }
        for(i=0;i<n;i++)
        {
                printf("Enter an integer : ");
                scanf("%d", p+i);
        }
        for(i=0;i<n;i++)
                printf("%d\t", *(p+i));
}
```

9.9.3 Allocating multiple blocks of memory

Calloc is another memory allocation function that is normally used to request multiple blocks of storage each of the same size and then sets all bytes to zero. The general form of calloc is:

ptr = (cast-type *) calloc (n, elem-size);

The above statement allocates contiguous space for n blocks each size of elements size bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space a null pointer is returned.

9.9.4 Releasing the used space

Compile time storage of a variable is allocated and released by the system in accordance with its storage class.

With the dynamic runtime allocation, it is our responsibility to release the space when it is not required. The release of storage space becomes important when the storage is limited. When we no longer need the data we stored in a block of memory and we do not intend to use that block for storing any other information, we may release that block of memory for future use, using the free function. free(ptr);

ptr is a pointer that has been created by using malloc or calloc.

9.9.5 To alter the size of allocated memory

The memory allocated by using calloc or malloc might be insufficient or excess sometimes in both the situations we can change the memory size already allocated with the help of the function realloc. This process is called reallocation of memory. The general statement of reallocation of memory is:

ptr = realloc(ptr, newsize);

This function allocates new memory space of size newsize to the pointer variable ptr ans returns a pointer to the first byte of the memory block. The allocated new block may be or may not be at the same region.

```
/* Program to understand the reallocation*/
#include< stdio.h >
#include< stdlib.h >.
define NULL 0
main()
{
    char *buffer;
    /*Allocating memory*/
    if((buffer=(char *) malloc(10))==NULL)
    {
        printf("Malloc failed\n");
        exit(1);
    }
    printf("Buffer of size %d created \n,_msize(buffer));
```

```
        strcpy(buffer,"Bangalore");
        printf("\nBuffer contains:%s\n",buffer);
        /*Reallocation*/
        if((buffer=(char *)realloc(buffer,15))==NULL)
        {
                printf("Reallocation failed\n");
                exit(1);
        }
        printf("\nBuffer size modified.\n");
        printf("\nBuffer still contains: %s\n",buffer);
        strcpy(buffer,"Mysore");
        printf("\nBuffer now contains:%s\n",buffer);
        /*freeing memory*/
        free(buffer);
}
```

9.10 Summary

It is very simple to use pointers provided the basics are understood thoroughly. Some of the uses of pointers are Accessing array elements, Returning more than one value from a function, Accessing dynamically allocated memory, Implementing data structures like linked list, trees and graphs.

9.11 Unit end questions

1. What do you mean by ponters?
2. Explain pointer arithmetic with example.
3. Differentiate between call by value and call by reference.
4. What do you mean by pointer to array?
5. What is dynamic memory allocation?

Unit 10 : C Preprocessor

Structure of the Unit

10.0 Objective

10.1 Introduction

10.2 Definition of Preprocessor

10.3 Macro Substitution directives

10.3.1 Simple macro substitution

10.3.2 Argument macro substitution

10.3.3 Nested macro substitution

10.4 File inclusion directives

10.5 Conditional compilation

10.5.1 #if, #elif, #endif

10.5.2 #ifdef, ifndef

10.6 Summary

10.7 Glossary

10.8 Further Readings

10.9 Unit end questions

10.0 Objective

Students who complete this unit should be able to understand the following tasks:

- Identify the role of Preprocessor
- Understand the types of preprocessor directives:
  - o Macro substitution directives
  - o File inclusion directives
  - o Compiler control directives

10.1 Introduction

This chapter focuses on the C Language - The Preprocessor, Preprocessor directives, Macros, #define identifier string, Simple macro substitution, Macros as arguments, Nesting of macros, Undefining a macro and File inclusion.

10.2 Definition of Preprocessor

Making programming versatile.

A unique feature of C language is the preprocessor, making 'C' an unusual language. A program can use the tools provided by preprocessor to make the program easy to read, modify, portable and more efficient. This comes from its Unix origins. As its name might suggest, the preprocessor is a phase which occurs prior to compilation of a program. Pre-processor commands are distinguished by the hash (number)

symbol '#'. When we compile the program, before the source code passes to the compiler it is examined by the C preprocessor for any macro definitions. When it sees the '#' directive, it goes through the entire program in search of the macro templates; whenever it finds one, it replaces the macro template with the appropriate macro expansion. Only after this procedure has been completed is the program handed over to the compiler.

Note: In C programming, it is customary to use capital letters for macro template. This makes it easy for programmers to pick out all the macro templates when reading through the program.

The C preprocessor, is used automatically by the C compiler to transform your program before compilation. Many times you will need to give special instructions to your compiler. This is done through inserting **preprocessor directives** into your code. When you begin compiling your code, a subprogram called the preprocessor scans the source code and performs simple substitution of tokenized strings for others according to predefined rules. Preprocessor is a program that processes the code before it passes through the compiler. It operates under the control of preprocessor command lines and directives. Preprocessor directives are placed in the source program before the main line before the source code passes through the compiler it is examined by the preprocessor for any preprocessor directives. If there is any, appropriate actions are taken then the source program is handed over to the compiler. In C language, all preprocessor directives begin with the hash character (#).

Directives

Directives are special instructions directed to the preprocessor (preprocessor directive) or to the compiler (compiler directive) on how it should process part or all of your source code or set some flags on the final object and are used to make writing source code easier (more portable for instance) and to make the source code more understandable. Directives are handled by the preprocessor, which is either a separate program invoked by the compiler or part of the compiler itself.

The preprocessor has two main uses: it allows external files, such as header files, to be included and it allows macros to be defined. This useful feature traditionally allowed constant values to be defined in Kernighan and Ritchie C, which had no constants in the language.

Other directives include #pragma compiler settings and macros. The result of the preprocessing stage is a text string. One thing to remember is that these directives are NOT compiled as part of your source code.

Preprocessor directives:

Directive	**Function**
#define	Defines a macro substitution
#undef	Undefines a macro
#include	Specifies a file to be included
#ifdef	Tests for macro definition
#endif	Specifies the end of #if
#ifndef	Tests whether the macro is not def
#if	Tests a compile time condition
#else	Specifies alternatives when # if test fails

The Preprocessor directives can be divided into three categories

1. Macro substitution division
2. File inclusion division
3. Compiler control division

10.3 Macro Substitution directives

When you write programs, you can create what is known as a *macro*, so the preprocessor replaces every occurrence of the identifier in the source code with the specified expression. The definition should start with the keyword '#define' and should follow on identifier and a string with at least one blank space between them. The string may be any text and identifier must be a valid C name. It also allows you to define *macros*, which are brief abbreviations for longer constructs, that we will discuss.

There are different forms of macro substitution. The most common form is

1. Simple macro substitution
2. Argument macro substitution
3. Nested macro substitution

10.3.1 Simple Macro Substitution

Simple string replacement is commonly used to define constants. Here's an example. If you write

```
#define RATE 8.50
```

Now, when you want to print or use the value of RATE, you use the word RATE instead of the number 8.50, the preprocessor will replace all instances of RATE with 8.50, which the compiler will interpret as the literal double 8.50. The preprocessor performs substitution, that is, RATE is replaced by 8.50 so this means there is no need for a semicolon. It is important to note that #define has basically the same functionality as the "find-and-replace" function in a lot of text editors/word processors. In case of change of RATE value, one has to define/change the value of RATE just once, and the new value will be flashed everywhere the constant is used. So, you need not go through the whole program and manually change each occurrence of the constant. It is more meaningful and would make a program read more naturally than if the raw number were used.

NOTE: Writing macro definition in capitals is a convention not a rule a macro definition can include more than a simple constant value it can include expressions as well.

Following are valid examples:

```
#define AREA 12.34
#define PIE  3.14
#define GREET "Good Morning.."
```

A **#define** directive is many a times used to define operators as shown below.

```
#define AND &&
#define OR ||
main()
{
        int f = 1, x = 4, y = 90 ;
```

```
        if ( ( f < 5 ) AND ( x <= 20 OR y <= 45 ) )
        printf ( "\nYour PC will always work fine..." ) ;
        else
        printf ( "\nIn front of the maintenance man" ) ;
}
```

A **#define** directive could be used even to replace a condition, as shown below.

```
#define AND &&
#define INRANGE ( a > 25 AND a < 50 )
main( )
{
        int a = 30 ;
        if ( INRANGE )
        printf ( "within range" ) ;
        else
        printf ( "out of range" ) ;
}
```

A **#define** directive could be used to replace even an entire C statement. This is shown below.

```
#define FOUND printf ( "The Yankee Doodle Virus" ) ;
main( )
{
        char signature ;
        if ( signature == 'Y' )
        FOUND
        else
        printf ( "Safe... as yet !" ) ;
}
```

10.3.2 Argument macro substitution

Function-like macros can take *arguments*, just like true functions. To define a macro that uses arguments, you insert *parameters* between the pair of parentheses in the macro definition that make the macro function-like. The parameters must be valid C identifiers, separated by commas and optionally whitespace.

To invoke a macro that takes arguments, you write the name of the macro followed by a list of *actual arguments* in parentheses, separated by commas. The invocation of the macro need not be restricted to a single logical line—it can cross as many lines in the source file as you wish. The number of arguments you give must match the number of parameters in the macro definition. When the macro is expanded, each

use of a parameter in its body is replaced by the tokens of the corresponding argument. (You need not use all of the parameters in the macro body.)

The preprocessor permits us to define more complex and more useful form of replacements it takes the following form. Macros can have arguments, just as functions can :

```
# define identifier(f1,f2,f3.....fn) string.
```

Notice that there is no space between identifier and left parentheses and the identifier f1,f2,f3.....fn is analogous to formal arguments in a function definition. There is a basic difference between simple replacement discussed above and replacement of macro arguments is known as a macro call. A simple example of a macro with arguments is

```
# define CUBE (x) (x*x*x)
```

If the following statements appears later in the program,

```
volume=CUBE(side);
```

The preprocessor would expand the statement to

```
volume =(side*side*side)
```

Here is another example that illustrates this fact.

```
#define AREA(x) ( 3.14 * x * x )
main()
{
float r1 = 6.25, r2 = 2.5, a ;
a = AREA ( r1 ) ;
printf ( "\nArea of circle = %f", a ) ;
a = AREA ( r2 ) ;
printf ( "\nArea of circle = %f", a ) ;
}
```

Here's the output of the program...

```
Area of circle = 122.656250
Area of circle = 19.625000
```

As an example, here is a macro that computes the minimum of two numeric values, as it is defined in many C programs, and some uses.

```
#define min (X, Y)          ((X) < (Y) ? (X) : (Y))
x = min(a, b);              ==> x = ((a) < (b) ? (a) : (b));
y = min(1, 2);              ==> y = ((1) < (2) ? (1) : (2));
z = min(a + 28, *p);        ==> z =((a + 28) < (*p)?(a + 28):(*p));
```

In this small example you can already see several of the dangers of macro arguments. For detailed explanations read more about some special rules that apply to macros and macro expansion, and point out certain cases in which the rules have counter-intuitive consequences that you must watch out for.

- Misnesting
- Operator Precedence Problems
- Swallowing the Semicolon
- Duplication of Side Effects
- Self-Referential Macros
- Argument Prescan
- Newlines in Arguments

Leading and trailing whitespace in each argument is dropped, and all whitespace between the tokens of an argument is reduced to a single space. Parentheses within each argument must balance; a comma within such parentheses does not end the argument. However, there is no requirement for square brackets or braces to balance, and they do not prevent a comma from separating arguments. Thus,

```
macro (array[x = y, x + 1])
```

passes two arguments to macro: array[x = y and x + 1]. If you want to supply array[x = y, x + 1] as an argument, you can write it as array[(x = y, x + 1)], which is equivalent C code.

All arguments to a macro are completely macro-expanded before they are substituted into the macro body. After substitution, the complete text is scanned again for macros to expand, including the arguments. This rule may seem strange, but it is carefully designed so you need not worry about whether any function call is actually a macro invocation.

For example, min (min (a, b), c) is first expanded to

```
min (((a) < (b) ? (a) : (b)), (c))
```

and then to

```
((((a) < (b) ? (a) : (b))) < (c)
? (((a) < (b) ? (a) : (b)))
: (c))
```

(Line breaks shown here for clarity would not actually be generated.)

You can leave macro arguments empty; this is not an error to the preprocessor (but many macros will then expand to invalid code). You cannot leave out arguments entirely; if a macro takes two arguments, there must be exactly one comma at the top level of its argument list. Here are some silly examples using min:

```
min(, b)      ==> ((  ) < (b) ? (  ) : (b))
min(a, )      ==> ((a  ) < ( ) ? (a  ) : ( ))
min(,)        ==> ((  ) < ( ) ? (  ) : ( ))
min((,),)     ==> (((,)) < ( ) ? ((,)) : ( ))
min()   error—> macro "min" requires 2 arguments, but only 1 given
min(,,)  error—> macro "min" passed 3 arguments, but takes just 2
```

Whitespace is not a preprocessing token, so if a macro foo takes one argument, foo () and foo ( ) both supply it an empty argument. Previous GNU preprocessor implementations and documentation were incorrect on this point, insisting that a function-like macro that takes a single argument be passed a space if an empty argument was required.

Macro parameters appearing inside string literals are not replaced by their corresponding actual arguments.

```
#define foo(x) x, "x"

foo(bar)      ==> bar, "x"
```

10.3.3 Nested macro substitution

We can also use one macro in the definition of another macro. That is macro definitions may be nested. Consider the following macro definitions

```
# define SQUARE(x)((x)*(x))
```

Undefining and Redefining Macros

If a macro ceases to be useful, it may be *undefined* with the '#undef' directive. '#undef' takes a single argument, the name of the macro to undefine. You use the bare macro name, even if the macro is function-like. It is an error if anything appears on the line after the macro name. '#undef' has no effect if the name is not a macro.

```
#define FOO 4

x = FOO;      ==> x = 4;

#undef FOO

x = FOO;      ==> x = FOO;
```

Once a macro has been undefined, that identifier may be *redefined* as a macro by a subsequent '#define' directive. The new definition need not have any resemblance to the old definition.

However, if an identifier which is currently a macro is redefined, then the new definition must be *effectively the same* as the old one. Two macro definitions are effectively the same if:

- Both are the same type of macro (object- or function-like).
- All the tokens of the replacement list are the same.
- If there are any parameters, they are the same.
- Whitespace appears in the same places in both. It need not be exactly the same amount of whitespace, though. Remember that comments count as whitespace.

These definitions are effectively the same:

```
#define FOUR (2 + 2)

#define FOUR       (2   +   2)

#define FOUR (2 /* two */ + 2)
```

but these are not:

```
#define FOUR (2 + 2)

#define FOUR ( 2+2 )

#define FOUR (2 * 2)

#define FOUR(score,and,seven,years,ago) (2 + 2)
```

If a macro is redefined with a definition that is not effectively the same as the old one, the preprocessor issues a warning and changes the macro to use the new definition. If the new definition is effectively the same, the redefinition is silently ignored. This allows, for instance, two different headers to define a common macro. The preprocessor will only complain if the definitions do not match.

Here are some important points to remember while writing macros with arguments:

(a) Be careful not to leave a blank between the macro template and its argument while defining the macro. For example, there should be no blank between **AREA** and **(x)** in the definition, #define AREA(x) (3.14 * x * x) : If we were to write **AREA (x)** instead of **AREA(x)**, the **(x)** would become a part of macro expansion, which we certainly don't want. What would happen is, the template would be expanded to

```
( r1 ) ( 3.14 * r1 * r1 )
```

which won't run. Not at all what we wanted.

(b) The entire macro expansion should be enclosed within parentheses. Here is an example of what would happen if we fail to enclose the macro expansion within parentheses.

```
#define SQUARE(n) n * n
main( )
{
    int j ;
    j = 64 / SQUARE ( 4 ) ;
    printf ( "j = %d", j ) ;
}
```

The output of the above program would be:

```
j = 64
```

whereas, what we expected was j = 4.

What went wrong? The macro was expanded into

```
j = 64 / 4 * 4 ; which yielded 64.
```

(c) Macros can be split into multiple lines, with a '\' (back slash) present at the end of each line. Following program shows how we can define and use multiple line macros.

```
#define HLINE for ( i = 0 ; i < 79 ; i++ ) \
printf ( "%c", 196 ) ;
#define VLINE( X, Y ) {\
gotoxy ( X, Y ) ; \
printf ( "%c", 179 ) ; \
}
main( )
{
```

```
int i, y ;
clrscr( ) ;
gotoxy ( 1, 12 ) ;
HLINE
for ( y = 1 ; y < 25 ; y++ )
VLINE ( 39, y ) ;
}
```

This program draws a vertical and a horizontal line in the center of the screen.

(d) If for any reason you are unable to debug a macro then you should view the expanded code of the program to see how the macros are getting expanded. If your source code is present in the file PR1.C then the expanded source code would be stored in PR1.I. You need to generate this file at the command prompt by saying:

```
cpp pr1.c
```

Here CPP stands for C PreProcessor. It generates the expanded source code and stores it in a file called PR1.I. You can now open this file and see the expanded source code. Note that the file PR1.I gets generated in C:\TC\BIN directory. The procedure for generating expanded source code for compilers other than

Turbo C/C++ might be a little different.

10.4 File inclusion directives

This directive causes one file to be included in another. The preprocessor command for file inclusion looks like this:

```
#include "filename"
```

and it simply causes the entire contents of **filename** to be inserted into the source code at that point in the program. Of course this presumes that the file being included is existing. When and why this feature is used? It can be used in two cases:

(a) If we have a very large program, the code is best divided into several different files, each containing a set of related functions. It is a good programming practice to keep different sections of a large program separate. These files are **#included** at the beginning of main program file.

(b) There are some functions and some macro definitions that we need almost in all programs that we write. These commonly needed functions and macro definitions can be stored in a file, and that file can be included in every program we write, which would add all the statements in this file to our program as if we have typed them in.

#include

C has some features as part of the language and some others as part of a **standard library**, which is a repository of code that is available alongside every standard-conformant C compiler. When the C compiler compiles your program it usually also links it with the standard C library. For example, on encountering a #include <stdio.h> directive, it replaces the directive with the contents of the stdio.h header file. When you use features from the library, C requires you to *declare* what you would be using. The first line in the program is a **preprocessing directive** which should look like this:

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

'#include <stdio.h>' is a command which tells the preprocessor to treat the file stdio.h as if it were the actually part of the program text, in other words to include it as part of the program to be compiled.The above line causes the C declarations which are in the stdio.h header to be included for use in your program. Usually this is implemented by just inserting into your program the contents of a **header file** called stdio.h, located in a system-dependent location. The location of such files may be described in your compiler's documentation. A list of standard C header files is listed below in the Headers table. The stdio.h header contains various declarations for input/output (I/O) using an abstraction of I/O mechanisms called **streams.** For example there is an output stream object called stdout which is used to output text to the standard output, which usually displays the text on the computer screen.

Note about '#include'

When an include statement is written into a program, it is a sign that a compiler should merge another file of C programming with the current one. However, the #include statement is itself valid C, so this means that a file which is included may contain #includes itself. The includes are then said to be "nested". This often makes includes simpler.

Macros are words which can be defined to stand in place of something complicated: they are a way of reducing the amount of typing in a program and a way of making long ungainly RATEces of code into short words. For example, the simplest use of macros is to give constant values meaningful names: e.g.

The **C standard library** is a standardized collection of header file's and library routines used to implement common operations, such as input/output and character string handling, in the C programming language. Unlike other languages such as COBOL, Fortran, and PL/I, C does not include builtin keywords for these tasks, so nearly all C programs rely on the standard library to function.

Design

The name and characteristic of each function are in a computer file called a header file but the actual implementation of functions are separated into a library file. The naming and scope of headers have become common but the organization of libraries still remains diverse. The standard library is usually shipped along with a compiler. Since C compilers often provide extra functionalities that are not specified in ANSI C, a standard library with a particular compiler is mostly incompatible with standard libraries of other compilers.

Design quality

Much of the C standard library has been shown to have been well-designed. A few parts, with the benefit of hindsight, are regarded as mistakes. The string input functions gets() (and the use of scanf() to read string input) are the source of many buffer overflows, and most programming guides recommend avoiding this usage. Another oddity is strtok(), a function that is designed as a primitive lexical analyser but is highly "fragile" and difficult to use.

History

The C programming language, before it was standardized, did not provide built-in functionalities such as I/O operations (unlike traditional languages such as Cobol and Fortran). Over time, user communities of C shared ideas and implementations of what is now called **C standard libraries** to provide that functionality. Many of these ideas were incorporated eventually into the definition of the standardized C programming language.

Both Unix and C were created at AT&T's Bell Laboratories in the late 1960s and early 1970s. During the 1970s the C programming language became increasingly popular. Many universities and organizations began creating their own variations of the language for their own projects. By the beginning of the 1980s compatibility problems between the various C implementations became apparent. In 1983 the

American National Standards Institute (ANSI) formed a committee to establish a standard specification of C known as "ANSI C". This work culminated in the creation of the so-called **C89** standard in 1989. Part of the resulting standard was a set of software libraries called the **ANSI C standard library**.

Later revisions of the C standard have added several new required header files to the library. Support for these new extensions varies between implementations.

The headers **<iso646.h>**, **<wchar.h>**, and **<wctype.h>** were added with Normative Addendum 1 (hereafter abbreviated as **NA1**), an addition to the C Standard ratified in 1995. The headers **<complex.h>**, **<fenv.h>**, **<inttypes.h>**, **<stdbool.h>**, **<stdint.h>**, and **<tgmath.h>** were added with **C99**, a revision to the C Standard published in 1999.

ANSI Standard

The ANSI C standard library consists of 24 C header files which can be included into a programmer's project with a single directive. Each header file contains one or more function declarations, data type definitions and macros. The contents of these header files follows. In comparison to some other languages (for example Java) the standard library is minuscule. The library provides a basic set of mathematical functions, string manipulation, type conversions, and file and console-based I/O. It does not include a standard set of "container types" like the C++ Standard Template Library, let alone the complete graphical user interface (GUI) toolkits, networking tools, and profusion of other functionality that Java provides as standard. The main advantage of the small standard library is that providing a working ANSI C environment is much easier than it is with other languages, and consequently porting C to a new platform is relatively easy.

Many other libraries have been developed to supply equivalent functionality to that provided by other languages in their standard library. For instance, the GNOME desktop environment project has developed the GTK+ graphics toolkit and GLib, a library of container data structures, and there are many other well-known examples. The variety of libraries available has meant that some superior toolkits have proven themselves through history. The considerable downside is that they often do not work particularly well together, programmers are often familiar with different sets of libraries, and a different set of them may be available on any particular platform.

ANSI C library header files

<assert.h> Contains the assert macro, used to assist with detecting logical errors and other types of bug in debugging versions of a program.

<complex.h> A set of functions for manipulating complex numbers. (New with **C99**)

<ctype.h> This header file contains functions used to classify characters by their types or to convert between upper and lower case in a way that is independent of the used character set (typically ASCII or one of its extensions, although implementations utilizing EBCDIC are also known).

<errno.h> For testing error codes reported by library functions.

<fenv.h> For controlling floating-point environment. (New with **C99**)

<float.h> Contains defined constants specifying the implementation-specific properties of the floating-point library, such as the minimum difference between two different floating-point numbers (_EPSILON), the maximum number of digits of accuracy (_DIG) and the range of numbers which can be represented (_MIN, _MAX).

<inttypes.h> For precise conversion between integer types. (New with **C99**)

<iso646.h>	For programming in ISO 646 variant character sets. (New with **NA1**)
<limits.h>	Contains defined constants specifying the implementation-specific properties of the integer types, such as the range of numbers which can be represented (_MIN, _MAX).
<locale.h>	For setlocale() and related constants. This is used to choose an appropriate locale.
<math.h>	For computing common mathematical functions
<setjmp.h>	Declares the macros setjmp and longjmp, which are used for non-local exits
<signal.h>	For controlling various exceptional conditions
<stdarg.h>	For accessing a varying number of arguments passed to functions.
<stdbool.h>	For a boolean data type. (New with **C99**)
<stdint.h>	For defining various integer types. (New with **C99**)
<stddef.h>	For defining several useful types and macros.
<stdio.h>	Provides the core input and output capabilities of the C language. This file includes the admired printf function.
<stdlib.h>	For performing a variety of operations, including conversion, pseudo-random numbers, memory allocation, process control, environment, signalling, searching, and sorting.
<string.h>	For manipulating several kinds of strings.
<tgmath.h>	For type-generic mathematical functions. (New with **C99**)
<time.h>	For converting between various time and date formats.
<wchar.h>	For manipulating wide streams and several kinds of strings using wide characters - key to supporting a range of languages. (New with **NA1**)
<wctype.h>	For classifying wide characters. (New with **NA1**)

The C standard library in C++

The C++ programming language includes the functionality of the ANSI C standard library, but makes several modifications, such as changing the names of the header files from <xxx.h> to <cxxx> (however, the C-style names are still available, although deprecated), and placing all identifiers into the std namespace.

Common support libraries

While not standardized, C programs may depend on a runtime library of routines which contain code the compiler uses at runtime. The code that initializes the process for the operating system, for example, before calling main(), is implemented in the C Run-Time Library for a given vendor's compiler. The Run-Time Library code might help with other language feature implementations, like handling uncaught exceptions or implementing floating point code.

The C standard library only documents that the specific routines mentioned in this article are available, and how they behave. Because the compiler implementation might depend on these additional implementation-level functions to be available, it is likely the vendor-specific routines are packaged with the C Standard Library in the same module, because they're both likely to be needed by any program built with their toolset.

Though often confused with the C Standard Library because of this packaging, the C Runtime Library is not a standardized part of the language and is vendor-specific.

NOTE: You should check the documentation of the development environment you are using for any vendor specific implementations of the #include directive.

A *library* in C is merely a group of functions and declarations. The library has an *interface* expressed in a file with a '.h' extension and an *implementation* expressed in a file with a .c extension (which may be precompiled or otherwise inaccessible).

The C90 standard headers list:

assert.h	ctype.h	errno.h	float.h	limits.h
locale.h	math.h	setjmp.h	signal.h	stdarg.h
stddef.h	stdio.h	stdlib.h	string.h	time.h

Headers added since C90:

complex.h	fenv.h	inttypes.h	iso646.h	stdbool.h
stdint.h	tgmath.h	wchar.h	wctype.h	

Header files and what to put into them

We have talked a lot in other parts of this course about header files. A header file is a file which is included (with a #include statement) in a C program. You have already encountered a number of library header files: <stdio.h> <string.h> <math.h> <stdlib.h>. Basically, any code *can* be put in a header file – after all, a #include simply shoves the code into your source at that point. However, here are the rules that we recommend for what to include in header files:

1) Defined constants (enum or #define).

2) Prototypes.

3) Only those #include statements which are *necessary* for the prototypes (for eg, if one of your prototypes includes FILE * then you *must* include stdio.h (otherwise FILE * won't make any sense to the compiler).

4) extern statements for global variables. (see later)

5) typedef and struct statements.

By convention, header files which you have written are included using double quotes rather than anglebrackets like so: #include "myheader.h" The reason for this rule is that it means that people reading your code can readily tell which files you have written and which are system files for the compiler you are using. This can be important, for example, the first few lines of a program are:

Egs.:

```
#include <stdio.h>
#include <Xlib.h>
#include <sys/types.h>
#include "protos.h"
#include "defs.h"
```

without the <> and "" notation it would be hard for all but the most experienced programmers to know which files were part of the program and which were part of the *operating system*. It is conventional to put system header files before user header files.

10.5 Conditional compilation

10.5.1 *#if, #elif* and *#endif* Directives

The **#if** command checks whether a controlling conditional expression evaluates to zero or nonzero, and excludes or includes a block of code respectively. For eg:

```
#if 1

/* This block will be included */

#endif

#if 0

/* This block will not be included */

#endif
```

The conditional expression could contain any C operator except for the assignment operators, the increment and decrement operators, the address-of operator, and the sizeof operator. One unique operator used in preprocessing and nowhere else is the **defined** operator. It returns 1 if the macro name, optionally enclosed in parentheses, is currently defined; 0 if not.

The **#elif** command is similar to #if, except that it is used to extract one from a series of blocks of code. E.g.:

```
#if /* some expression */

:

:

#elif /* another expression */

:

/* imagine many more #elifs here ... */

:

#else

/* The optional #else block is selected if none of the previous #if or

#elif blocks are selected */

:

:

#endif /* The end of the #if block */
```

The **#if** directive can be used to test whether an expression evaluates to a nonzero value or not. If the result of the expression is nonzero, then subsequent lines upto a **#else, #elif** or **#endif** are compiled, otherwise they are skipped. A simple example of **#if** directive is shown below:

```
main()
```

```
{
#if TEST <= 5
statement 1 ;
statement 2 ;
statement 3 ;
#else
statement 4 ;
statement 5 ;
statement 6 ;
#endif
}
```

If the expression, **TEST <= 5** evaluates to true then statements 1, 2 and 3 are compiled otherwise statements 4, 5 and 6 are compiled. In place of the expression **TEST <= 5** other expressions like **(LEVEL == HIGH || LEVEL == LOW)** or **ADAPTER == CGA** can also be used.

If we so desire we can have nested conditional compilation directives. An example that uses such directives is shown below.

```
#if ADAPTER == VGA
code for video graphics array
#else
#if ADAPTER == SVGA
code for super video graphics array
#else
code for extended graphics adapter
#endif
#endif
```

The above program segment can be made more compact by using another conditional compilation directive called **#elif**. The same program using this directive can be rewritten as shown below. Observe that by using the **#elif** directives the number of **#endifs** used in the program get reduced.

```
#if ADAPTER == VGA
code for video graphics array
#elif ADAPTER == SVGA
code for super video graphics array
#else
code for extended graphics adapter
#endif
```

There are a handful more preprocessor commands which can largely be ignored by the beginner. They are commonly used in "include" files to make sure that things are not defined twice.

#if

This is followed by some expression on the same line. It allows *conditional*

compilation. It is an advanced feature which can be used to say: only compile the code between #if and #endif if the value following #if is true, else leave out that code altogether. This is different from not executing code—the code will not even be compiled.

10.5.2 #ifdef and #ifndef Directives

The **#ifdef** command is similar to #if, except that the code block following it is selected if a macro name is defined. In this respect,

#ifdef NAME

is equivalent to

#if defined NAME

The **#ifndef** command is similar to **#ifdef**, except that the test is reversed:

#ifndef NAME

is equivalent to

#if !defined NAME

We can, if we want, have the compiler skip over part of a source code by inserting the preprocessing commands **#ifdef** and **#endif**, which have the general form:

```
#ifdef macroname

statement 1 ;

statement 2 ;

statement 3 ;

#endif
```

If **macroname** has been **#defined**, the block of code will be processed as usual; otherwise not.

Where would **#ifdef** be useful? When would you like to compile only a part of your program? In three cases:

(a) To "comment out" obsolute lines of code. It often happens that a program is changed at the last minute to satisfy a client. This involves rewriting some part of source code to the client's satisfaction and deleting the old code. But veteran programmers are familiar with the clients who change their mind and want the old code back again just the way it was.

Now you would definitely not like to retype the deleted code again.

One solution in such a situation is to put the old code within a pair of /* */ combination. But we might have already written a comment in the code that we are about to "comment out". This would mean we end up with nested comments. Obviously, this solution won't work since we can't nest comments in C.

Therefore the solution is to use conditional compilation as shown below.

```
main( )
```

```
{
#ifdef OKAY
statement 1 ;
statement 2 ; /* detects virus */
statement 3 ;
statement 4 ; /* specific to stone virus */
#endif
statement 5 ;
statement 6 ;
statement 7 ;
}
```

Here, statements 1, 2, 3 and 4 would get compiled only if the macro OKAY has been defined, and we have purposefully omitted the definition of the macro OKAY. At a later date, if we want that these statements should also get compiled all that we are required to do is to delete the **#ifdef** and **#endif** statements.

(b) A more sophisticated use of **#ifdef** has to do with making the programs portable, i.e. to make them work on two totally different computers. Suppose an organization has two different types of computers and you are expected to write a program that works on both the machines. You can do so by isolating the lines of code that must be different for each machine by marking them off with **#ifdef**. For example:

```
main()
{
#ifdef INTEL
code suitable for a Intel PC
#else
code suitable for a Motorola PC
#endif
code common to both the computers
}
```

When you compile this program it would compile only the code suitable for a Intel PC and the common code. This is because the macro INTEL has not been defined. Note that the working of **#ifdef - #else - #endif** is similar to the ordinary **if - else** control instruction of C.

If you want to run your program on a Motorola PC, just add a statement at the top saying,

```
#define INTEL
```

Sometimes, instead of **#ifdef** the **#ifndef** directive is used. The **#ifndef** (which means 'if not defined') works exactly opposite to **#ifdef**. The above example if written using **#ifndef**, would look like this:

```
main()
{
#ifndef INTEL
code suitable for a Intel PC
#else
code suitable for a Motorola PC


#endif
code common to both the computers
}
```

(c) Suppose a function **myfunc( )** is defined in a file 'myfile.h' which is **#included** in a file 'myfile1.h'. Now in your program file if you **#include** both 'myfile.h' and 'myfile1.h' the compiler flashes an error 'Multiple declaration for **myfunc**'. This is because the same file 'myfile.h' gets included twice. To avoid this we can write following code in the header file.

```
/* myfile.h */
#ifndef __myfile_h
#define __myfile_h
myfunc()
{
/* some code */
}
#endif
```

First time the file 'myfile.h' gets included the preprocessor checks whether a macro called **__myfile_h** has been defined or not. If it has not been then it gets defined and the rest of the code gets included. Next time we attempt to include the same file, the inclusion is prevented since **__myfile_h** already stands defined. Note that there is nothing special about **__myfile_h**. In its place we can use any other macro as well.

To summarize :

#undef : This undefines a macro, leaving the name free.

#ifdef: This is followed by a macro name. If that macro is defined then this is true.

#ifndef: This is followed by a macro name. If that name is not defined then this is true.

#else : This is part of an #if, #ifdef, #ifndef preprocessor statement.

#endif: The **#endif** command ends a block started by #if, #ifdef, or #ifndef. This marks the end of a preprocessor statement.

#line : Has the form:

#line *constant* filename

This is for debugging mainly. This statement causes the compiler to believe that the next line is line number (constant) and is part of the file (filename).

Example

```
/***********************************************************/
/* To compile or not to compile */
/***********************************************************/
#define SOMEDEFINITION 6546
#define CHOICE 1 /* Choose this before compiling */
/***********************************************************/
#if (CHOICE == 1)
#define OPTIONSTRING "The programmer selected this"
#define DITTO "instead of .... "
#else
#define OPTIONSTRING "The alternative"
#define DITTO "i.e. This! "
#endif
/***********************************************************/
#ifdef SOMEDEFINITION
#define WHATEVER "Something was defined!"
#else
#define WHATEVER "Nothing was defined"
#endif
/***********************************************************/
main ()
{
printf (OPTIONSTRING);
printf (DITTO);
}
```

#error

This is a part of the proposed ANSI standard. It is intended for debugging. It forces the compiler to abort compilation. The **#error** directive halts compilation. When one is encountered the standard specifies that the compiler should emit a diagnostic containing the remaining tokens in the directive. This is mostly used for debugging purposes.

```
#error message
```

There are a few more, less useful, features.

Except for expansion of predefined macros, all these operations are triggered with *preprocessing directives*. Preprocessing directives are lines in your program that start with '#'. Whitespace is allowed before and after the '#'. The '#' is followed by an identifier, the *directive name*. It specifies the operation to perform. Directives are commonly referred to as '*#name*' where *name* is the directive name. For example, '#define' is the directive that defines a macro.

The '#' which begins a directive cannot come from a macro expansion. Also, the directive name is not macro expanded. Thus, if foo is defined as a macro expanding to define, that does not make '#foo' a valid preprocessing directive.

The set of valid directive names is fixed. Programs cannot define new preprocessing directives.

Some directives require arguments; these make up the rest of the directive line and must be separated from the directive name by whitespace. For example, '#define' must be followed by a macro name and the intended expansion of the macro.

A preprocessing directive cannot cover more than one line. The line may, however, be continued with backslash-newline, or by a block comment which extends past the end of the line. In either case, when the directive is processed, the continuations have already been merged with the first line to make one long line.

Once-Only Headers

If a header file happens to be included twice, the compiler will process its contents twice. This is very likely to cause an error, e.g. when the compiler sees the same structure definition twice. Even if it does not, it will certainly waste time.

The standard way to prevent this is to enclose the entire real contents of the file in a conditional, like this:

```
/* File foo. */
#ifndef FILE_FOO_SEEN
#define FILE_FOO_SEEN
the entire file
#endif     /* !FILE_FOO_SEEN */
```

This construct is commonly known as a *wrapper #ifndef*. When the header is included again, the conditional will be false, because FILE_FOO_SEEN is defined. The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

The macro FILE_FOO_SEEN is called the *controlling macro* or *guard macro*. In a user header file, the macro name should not begin with '_'. In a system header file, it should begin with '__' to avoid conflicts with user programs. In any kind of header file, the macro name should contain the name of the file and some additional text, to avoid conflicts with other header files.

Standard Predefined Macros

The standard predefined macros are specified by the relevant language standards, so they are available with all compilers that implement those standards. Older compilers may not provide all of them. Their names all start with double underscores.

__FILE__

This macro expands to the name of the current input file, in the form of a C string constant. This is the path by which the preprocessor opened the file, not the short name specified in '#include' or as the input file name argument. For example, "/usr/local/include/myheader.h" is a possible expansion of this macro.

__LINE__

This macro expands to the current input line number, in the form of a decimal integer constant. While we call it a predefined macro, it's a pretty strange macro, since its "definition" changes with each new line of source code.

__FILE__ and __LINE__ are useful in generating an error message to report an inconsistency detected by the program; the message can state the source line at which the inconsistency was detected. For example,

```
fprintf(stderr, "Internal error: "
        "negative string length "
        "%d at %s, line %d.",
    length, __FILE__, __LINE__);
```

An '#include' directive changes the expansions of __FILE__ and __LINE__ to correspond to the included file. At the end of that file, when processing resumes on the input file that contained the '#include' directive, the expansions of __FILE__ and __LINE__ revert to the values they had before the '#include' (but __LINE__ is then incremented by one as processing moves to the line after the '#include').

A '#line' directive changes __LINE__, and may change __FILE__ as well.

C99 introduces __func__, and GCC has provided __FUNCTION__ for a long time. Both of these are strings containing the name of the current function (there are slight semantic differences; see the GCC manual). Neither of them is a macro; the preprocessor does not know the name of the current function. They tend to be useful in conjunction with __FILE__ and __LINE__, though.

__DATE__

This macro expands to a string constant that describes the date on which the preprocessor is being run. The string constant contains eleven characters and looks like "Feb 12 1996". If the day of the month is less than 10, it is padded with a space on the left.

If GCC cannot determine the current date, it will emit a warning message (once per compilation) and __DATE__ will expand to "??? ?? ????".

__TIME__

This macro expands to a string constant that describes the time at which the preprocessor is being run. The string constant contains eight characters and looks like "23:59:01".

If GCC cannot determine the current time, it will emit a warning message (once per compilation) and __TIME__ will expand to "??:??:??".

10.6 Summary

1. The preprocessing language consists of *directives* to be executed and *macros* to be expanded. Its primary capabilities are:

- Inclusion of header files. These are files of declarations that can be substituted into your program.
- Macro expansion. You can define *macros*, which are abbreviations for arbitrary fragments of C code. The preprocessor will replace the macros with their definitions throughout the program. Some macros are automatically defined for you.
- Conditional compilation. You can include or exclude parts of the program according to various conditions.
- Line control. If you use a program to combine or rearrange source files into an intermediate file which is then compiled, you can use line control to inform the compiler where each source line originally came from.
- Diagnostics. You can detect problems at compile time and issue errors or warnings.

2. To Undefine a macro - A defined macro can be undefined using the statement

undef identifier.

This is useful when we want to restrict the definition only to a particular part of the program.

3. There are three general reasons to use a conditional.

- A program may need to use different code depending on the machine or operating system it is to run on. In some cases the code for one operating system may be erroneous on another operating system; for example, it might refer to data types or constants that do not exist on the other system. When this happens, it is not enough to avoid executing the invalid code. Its mere presence will cause the compiler to reject the program. With a preprocessing conditional, the offending code can be effectively excised from the program when it is not valid.
- You may want to be able to compile the same source file into two different programs. One version might make frequent time-consuming consistency checks on its intermediate data, or print the values of those data for debugging, and the other not.
- A conditional whose condition is always false is one way to exclude code from the program but keep it as a sort of comment for future reference.

Simple programs that do not need system-specific logic or complex debugging hooks generally will not need to use preprocessing conditionals.

10.7 Glossary

ANSI (American National Standards Institute) An organization that creates standards for the computer industry. Responsible for the ANSI C standard.

ANSI C An international standard for the C programming language.

C An advanced programming language used for programming advanced computer applications.

C++ (C Plus Plus) The same as C with added object-oriented functions.

C# (C Sharp) A Microsoft version of C++ with added Java-like functions.

Case Sensitive A term used to describe if it is of importance to use upper or lower case letters.

Character set A finite set of different characters used for the representation, organisation or control of data.

Directive A directive is an instruction by the programmer for a compiler. Most programming languages have directives. In C, **#include** is used to include header files. You can use the #define **directive** to give a meaningful name to a constant in your program

Macro In C and C++, a Macro is a piece of text that is expanded by the preprocessor part of the compiler. This is used in to expand text before compiling.

Preprocessor The preprocessor is the part of the compiler in C and C++ that reads the source code files and expands text wherever it finds a # in column one.

NaN (not a number)

A value that can be stored in a floating type but that is not a valid floating point number. Not all systems support the NaN value.

10.8 Further Readings

1. Programming in 'C' – Balaguruswamy, Tata McGraw-Hill Publications
2. The C programming language – Dennis Ritchie, Prentice Hall, Delhi
3. Let us C – Yashwant Kanetkar, BPB Publication

10.9 Unit end questions

1. Define a macro called "birthday" which describes the day of the month upon which your birthday falls.
2. Write a command to the preprocessor to include to maths library math.h.
3. A macro is always a number. True or false?
4. A macro is always a constant. True or false?
5. Write a Program to find the area and perimeter of geometrical figure using macro definition.
6. What are the differences between header file and library file?
7. What is the difference between #include <stdio.h> and #include "stdio.h"?

Unit 11 : File Handling

Structure of the Unit

11.0 Objective

11.1 Introduction

11.2 Definition of File and its types

11.3 Operations on the Files

- 11.3.1 Opening
- 11.3.2 Closing
- 11.3.3 Read from a File
- 11.3.4 Write into a File

11.4 Standard file functions

- 11.4.1 fopen
- 11.4.2 fclose
- 11.4.3 feof
- 11.4.4 fseek, ftell
- 11.4.5 rewind
- 11.4.6 fgetc, fputc, fread, fwrite
- 11.4.7 fscanf, fprintf
- 11.4.8 Other file access functions
- 11.4.9 Error Handling Functions

11.5 File handling through programs

11.6 Summary

11.7 Glossary

11.8 Further Readings

11.9 Unit end questions

11.0 Objective

Students who complete this unit should be able to understand the following tasks:

- Identify the File and file operations
- Understand and use standard file handling functions
- Understand and use file access functions
- Understand and use error handling functions

11.1 Introduction

This chapter focuses on the File handling in C Language - The File and file operations, use of standard file handling functions, use of file access functions, use of error handling functions.

All files, irrespective of the data they contain or the methods used to process them, have certain important properties. They have a name. They must be opened and closed. They can be written to, or read from, or appended to. Conceptually, until a file is opened nothing can be done to it. When it is opened, we may have access to it at its beginning or end. To prevent accidental misuse, we must tell the system which of the three activities (reading, writing, or appending) we will be performing on it. When we are finished using the file, we must close it. If the file is not closed the operating system cannot finish updating its own housekeeping records and data in the file may be lost.

Essentially there are two kinds of files that programmers deal with – *text* files and *binary* files. These two classes of files will be discussed in the following sections.

11.2 Definition of File and its types

At some times it becomes necessary to store the data in a manner that can be later retrieved and displayed either in part or in whole. This medium is usually a 'file' on the disk. This chapter discusses how file I/O operations can be performed.

Data Organization

Before we start doing file input/output let us first find out how data is organized on the disk. All data stored on the disk is in binary form. How this binary data is stored on the disk varies from one OS to another. However, this does not affect the C programmer since he has to use only the library functions written for the particular OS to be able to perform input/output. It is the compiler vendor's responsibility to correctly implement these library functions by taking the help of OS. This is illustrated here :

Our program	OS	C Library functions	Disk

There are different operations that can be carried out on a file. These are:

- Creation of a new file
- Opening an existing file
- Reading from a file
- Writing to a file
- Moving to a specific location in a file (seeking)
- Closing a file

Files are places for reading data from and writing data into. This includes disk files and it includes devices such as the printer or the monitor of a computer. C treats all information which enters or leaves a program as though it were a stream of bytes: a file. The most commonly used file streams are stdin (the keyboard) and stdout (the screen), but more sophisticated programs need to be able to read or write to files which are found on a disk or to the printer etc. An operating system allows a program to see files in the outside world by providing a number of channels or 'portals' ('inlets' and 'outlets') to work through. In order to examine the contents of a file or to write information to a file, a program has to *open* one of these portals. The reason for this slightly indirect method of working is that channels/portals hide operating system dependent details of filing from the programmer. Think of it as a protocol. A program which writes information does no more than passing that information to one of these portals and the operating system's filing subsystem does the rest. A program which reads data simply reads values from its file portal and does

not have to worry about how they got there. This is extremely simple to work in practice. To use a file then, a program has to go through the following routine:

- ?Open a file for reading or writing. (Reserve a portal and locate the file on disk or whatever.)
- ?Read or write to the file using file handling functions provided by the standard library.
- ?Close the file to free the operating system "portal" for use by another program or file.

A program opens a file by calling a standard library function and is returned a file pointer, by the operating system, which allows a program to address that particular file and to distinguish it from all others.The ANSI C <stdio.h> library is based on the original Unix file I/O primitives but casts a wider net to accommodate the least-common denominator across varied systems.

Let us now look at a program that reads a file and display its contents on the screen.

```
#include "stdio.h"
main( )
{
FILE *test;
char c;
test = fopen("TENLINES.TXT", "r"); /* open in 'r'ead mode */
if (test == EOF) printf("File doesn't exist\n");
else {
do {
c = getc(test); /* get one character from the file */
putchar(c); /* display it on the screen */
} while (c != EOF); /* repeat until EOF (end of file) */
}
fclose(test);
}
```

In this program we check to see that the file exists, and if it does, we execute the main body of the program. If it doesn't, we print a message and quit. If the file does not exist, the system will set the pointer equal to NULL which we can test. The main body of the program is one do while loop in which a single character is read from the file and output to the monitor until an EOF (end of file) is detected from the input file. The file is then closed and the program is terminated.

Streams

Input and output, whether to or from physical devices such as terminals and tape drives, or whether to or from files supported on structured storage devices, are mapped into logical data streams, whose properties are more uniform than their various inputs and outputs. Two forms of mapping are supported: text streams and binary streams.

A text stream is an ordered sequence of characters composed into lines, each line consisting of zero or more characters plus a terminating new-line character. Whether the last line requires a terminating new-line character is implementation-defined. Characters may have to be added, altered, or deleted on

input and output to conform to differing conventions for representing text characters in a stream and those in the external representation.

Data read in from a text stream will necessarily compare equal to the data that were earlier written out to that stream only if the data consist only of printable characters and the control characters horizontal tab and new-line, no new-line character is immediately preceded by space characters, and the last character is a new-line character. Whether space characters that are written out immediately before a new-line character appear when read in is implementation-defined.

Unix adopted a standard internal format for all text streams. Each line of text is terminated by a new-line character. That's what any program expects when it reads text, and that's what any program produces when it writes text. If such a convention doesn't meet the needs of a text-oriented peripheral attached to a Unix machine, then the fixup occurs out at the edges of the system. None of the code in the middle needs to change.

A binary stream is an ordered sequence of characters that can transparently record internal data. Data read in from a binary stream shall compare equal to the data that were earlier written out to that stream under the same implementation. Such a stream may, however, have an implementation-defined number of null characters appended to the end of the stream.

Nothing in Unix prevents the program from writing arbitrary 8-bit binary codes to any open file, or reading them back unchanged from an adequate repository. Thus, Unix obliterated the long-standing distinction between text streams and binary streams.

Files Generally

C provides two levels of file handling; these can be called high level and low level. High level files are all treated as text files. In fact, the data which go into the files are exactly what would be seen on the screen, character by character, except that they are stored in a file instead. This is true whether a file is meant to store characters, integers, floating point types. Any file, which is written to by high level file handling functions, ends up as a text file which could be edited by a text editor.

High level text files are also read back as character files, in the same way that input is acquired from the keyboard. This all means that high level file functions are identical in concept to keyboard/screen input/output. The alternative to these high level functions, is obviously low level functions. These are more efficient, in principle, at filing data as they can store data in large lumps, in raw memory format, without converting to text files first. Low level input/output functions have the disadvantage that they are less 'programmer friendly' than the high level ones, but they are likely to work faster.

Text Mode Versus Binary Mode

(i) New Lines

In text mode, a newline character is converted into the carriage return -linefeed combination before being written to the disk. Like wise, the carriage return-line feed combination on the disk is converted back in to a newline when the file is read by a c program. However if file is opened in binary mode, as opposed to text mode, these conversions will not take place.

(ii) End of File

The difference is that in text mode when end-of-file is detected a special character whose ascii value is 26, is inserted after the last character in the file to mark the end of file. If this character is detected at any point in the file, the read function will return the EOF signal to the program. As against this, there is no such special character present in the binary mode files to mark the end of file. The binary mode file keeps track of the end of file from the number of characters present in directory entry of the file.

Text Mode

The only function available for storing in a disk file is the fprintf( ) in text mode. Here numbers are stored as string of characters when written to the disk. These 1234, even though it occupies two bytes in memory, when transferred to the disk using fprintf( ), it would occupy four bytes, one byte per character. Similarly the floating point number 1234.56 would occupy 7 bytes on disk. These, numbers with more digits would require more disk space. In binary by using the functions (fread( ) and fwrite( )) numbers are stored in binary format. It means each number would occupy the same number of bytes on disk as it occupies in memory. Normally a programmer can get away with using the high level input/output functions, but there may be times when C's predilection for handling all high level input/output as text files, becomes a nuisance. A program can then use a set of low level I/O functions which are provided by the standard library. These are:

open()	close()	creat()	read()	write()
rename()	unlink()/remove()	lseek()		

These low level routines work on the operating system's end of the file portals. They should be regarded as being advanced features of the language because they are dangerous routines for bug ridden programs. The data which they deal with is untranslated: that is, no conversion from characters to floating point or integers or any type at all take place. Data are treated as a raw stream of bytes. Low level functions should not be used on any file at the same time as high level routines, since high level file handling functions often make calls to the low level functions. Working at the low level, programs can create, delete and rename files but they are restricted to the reading and writing of untranslated data: there are no functions such as fprintf() or fscanf() which make type conversions. As well as the functions listed above a local operating system will doubtless provide special function calls which enable a programmer to make the most of the facilities offered by the particular operating environment. These will be documented, either in a compiler manual, or in an operating system manual, depending upon the system concerned. (They might concern special graphics facilities or windowing systems or provide ways of writing special system dependent data to disk files, such as date/time stamps etc.)

FILE pointers

The <stdio.h> header contains a definition for a type FILE (usually via a typedef) which is capable of recording all the information needed to control a stream, including its file position indicator, a pointer to the associated buffer (if any), an error indicator that records whether a read/write error has occurred, and an end-of-file indicator that records whether the end of the file has been reached. It is considered bad manners to access the contents of FILE directly unless the programmer is writing an implementation of <stdio.h> and its contents. How, pray tell, is one going to know whether the file handle, for example, is spelt handle or _Handle? Access to the contents of FILE is better provided via the functions in <stdio.h>. It can be said that the FILE type is an early example of object-oriented programming.

File Positions

When data are read from a file, the operating system keeps track of the current position of a program within that file so that it only needs to make a standard library call to 'read the next part of the file' and the operating system obliges by reading some more and advancing its position within the file, until it reaches the end. Each single character which is read causes the position in a file to be advanced by one. Although the operating system does a great deal of hand holding regarding file positions, a program can control the way in which that position changes with functions such as ungetc() if need be. In most cases it is not necessary and it should be avoided, since complex movements within a file can cause complex movements of a disk drive mechanism which in turn can lead to wear on disks and the occurrence of errors. Most of the high level input/output functions which deal with files are easily recognizable in that they start with the letter 'f'. Some of these functions will appear strikingly familiar.

For instance:

fprintf() fscanf() fgets() fputs()

These are all generalized file handling versions of the standard input/output library. They work with generalized files, as opposed to the specific files stdin and stdout which printf() and scanf() use. The file versions differ only in that they need an extra piece of information: the file pointer to a particular portal. This is passed as an extra parameter to the functions. they process data in an identical way to their standard I/O counterparts.

Other filing functions will not look so familiar. For example:

fopen() freopen() fclose() getc() ungetc();
putc() fgetc() fputc() feof()

Before any work can be done with high level files, these functions need to be explained in some detail.

11.3 Operations on the Files

11.3.1 Opening

Before we can read (or write) information from (to) a file on a disk we must open the file. To open the file we have called the function **fopen( )**. It would open a file "PR1.C" in 'read' mode, which tells the C compiler that we would be reading the contents of the file. Note that "r" is a string and not a character; hence the double quotes and not single quotes. In fact **fopen( )** performs three important tasks when you open the file in "r" mode:

- Firstly it searches on the disk the file to be opened.
- Then it loads the file from the disk into a place in memory called buffer.
- It sets up a character pointer that points to the first character of the buffer.

A file is opened by a call to the library function fopen(): this is available automatically when the library file <stdio.h> is included. There are two stages to opening a file: firstly a file portal must be found so that a program can access information from a file at all. Secondly the file must be physically located on a disk or as a device or whatever. The fopen() function performs both of these services and, if, in fact, the file it attempts to open does not exist, that file is created anew. The syntax of the fopen() function is:

```
FILE *returnpointer;

returnpointer = fopen("filename","mode");
```

or

```
FILE returnpointer;

char *fname, *mode;

returnpointer = fopen(fname,mode);
```

The filename is a string which provides the name of the file to be opened. Filenames are system dependent so the details of this must be sought from the local operating system manual.

Trouble in Opening a File

There is a possibility that when we try to open a file using the function **fopen( )**, the file may not be opened. While opening the file in "r" mode, this may happen because the file being opened may not be present on the disk at all. And you obviously cannot read a file that doesn't exist. Similarly, while opening

the file for writing, **fopen()** may fail due to a number of reasons, like, disk space may be insufficient to open a new file, or the disk may be write protected or the disk is damaged and so on.

Crux of the matter is that it is important for any program that accesses disk files to check whether a file has been opened successfully before trying to read or write to the file. If the file opening fails due to any of the several reasons mentioned above, the **fopen( )** function returns a value NULL (defined in "stdio.h" as **#define NULL 0**).

11.3.2 Closing

When we have finished reading from the file, we need to close it. This is done using the function **fclose( )** through the statement,

```
fclose ( fp ) ;
```

The function returns zero if the file was successfully closed or EOF if any errors were detected. Once we close the file we can no longer read from it using **getc( )** unless we reopen the file. Note that to close the file we don't use the filename but the file pointer **fp**. On closing the file the buffer associated with the file is removed from memory.

When we close this file using **fclose( )** three operations would be performed:

- The characters in the buffer would be written to the file on the disk.
- At the end of file a character with ASCII value 26 would get written.
- The buffer would be eliminated from memory.

You can imagine a possibility when the buffer may become full before we close the file. In such a case the buffer's contents would be written to the disk the moment it becomes full. All this buffer management is done for us by the library functions.

11.3.3 Read from a File

Reading from file can be done in several ways :

- *getc()* - like *fgetc()*
- *fgetc() - int fgetc (FILE * stream);* - write a character to a file
- *fgets() - char * fgets (char * string , int num , FILE * stream);* - write a string to a file
- *fscanf() - int fscanf (FILE * stream , const char * format [, argument , ...]);* - works like *scanf()* except that it reads from a file instead of *STDIN*

Reading functions are classified in 3 categories :

- Character Input Functions : fgetc , fgets, getc, getchar, gets, ungetc
- Direct input function : fread function
- Formatted input functions : scanf family of functions

Once the file has been successfully opened for reading using **fopen( )**, as we have seen, the file's contents are brought into buffer (partly or wholly) and a pointer is set up that points to the first character in the buffer. This pointer is one of the elements of the structure to which **fp** is pointing.

To read the file's contents from memory there exists a function called **fgetc( )**. This has been used in our program as,

```
ch = fgetc ( fp ) ;
```

fgetc( ) reads the character from the current pointer position, advances the pointer position so that it now points to the next character, and returns the character that is read, which we collected in the variable **ch**. Note that once the file has been opened, we no longer refer to the file by its name, but through the file pointer **fp**.

While reading from the file, when **fgetc( )** encounters a special character (whose ASCII value is 26, signifies end of file. This character is inserted beyond the last character in the file, when it is created), instead of returning the character that it has read, it returns the macro EOF. The EOF macro has been defined in the file "stdio.h". In place of the function **fgetc( )** we could have as well used the macro **getc( )** with the same effect.

11.3.4 Write into a File

Writing to files can be done in various ways:

- *putc()* - like *fputc()*
- *fputc() - int fputc (int character, FILE * stream);* - write a character to a file
- *fputs() - int fputs (const char * string , FILE * stream);* - write a string to a file
- *fprintf()- int fprintf (FILE * stream , const char * format [, argument , ...]);* - works like *printf()* except that it writes to a file instead of
- *STDOUT.*

Similar to read function, this is also categorized into 3 classes :

- Character Output Functions : fputc, fputs, putc, putchar, puts
- Direct output function : fwrite function
- Formatted output functions : fprintf family of functions

The **fputc( )** function is similar to the **putch( )** function, in the sense that both output characters. However, **putch( )** function always writes to the VDU, whereas, **fputc( )** writes to the file. Which file? The file signified by **ft**. The writing process continues till all characters from the source file have been written to the target file, following which the **while** loop terminates.

Note that our sample file-copy program is capable of copying only text files. To copy files with extension .EXE or .COM, we need to open the files in binary mode, a topic that would be dealt with in sufficient detail in a later section.

11.4 Standard file functions

11.4.1 fopen

#include <stdio.h>

FILE *fopen(const char *filename, const char *mode);

FILE *freopen(const char *filename, const char *mode, FILE *stream);

The fopen and freopen functions open files.

The fopen function opens the file whose name is in the string pointed to by filename and associates a stream with it.

The argument mode points to a string beginning with any one of the following sequences:

r	open a text file for reading
w	truncate to zero length or create a text file for writing
a	append; open or create text file for writing at end-of-file
rb	open binary file for reading
wb	truncate to zero length or create a binary file for writing
ab	append; open or create binary file for writing at end-of-file
r+	open text file for update (reading and writing)
w+	truncate to zero length or create a text file for update
a+	append; open or create text file for update
r+b or rb+	open binary file for update (reading and writing)
w+b or wb+	truncate to zero length or create a binary file for update
a+b or ab+	append; open or create binary file for update

Opening a file with read mode ('r' as the first character in the mode argument) fails if the file does not exist or cannot be read. Opening a file with append mode ('a' as the first character in the mode argument) causes all subsequent writes to the file to be forced to the then-current end-of-file, regardless of intervening calls to the fseek function. In some implementations, opening a binary file with append mode ('b' as the second or third character in the above list of mode arguments) may initially position the file position indicator for the stream beyond the last data written, because of null character padding.

When a file is opened with update mode ('+' as the second or third character in the above list of mode argument values), both input and output may be performed on the associated stream. However, output may not be directly followed by input without an intervening call to the fflush function or to a file positioning function (fseek, fsetpos, or rewind), and input may not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file. Opening (or creating) a text file with update mode may instead open (or create) a binary stream in some implementations. When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators are cleared.

The fopen function returns a pointer to the object controlling the stream. If the open operation fails, fopen returns a null pointer.

The freopen function opens the file whose name is the string pointed to by filename and associates the stream pointed to by stream with it. The mode argument is used just as in the fopen function. The freopen function first attempts to close any file that is associated with the specified stream. Failure to close the file successfully is ignored. The error and end-of-file indicators for the stream are cleared. It returns a null pointer if the open operation fails, or the value stream if the open operation succeeds.

The operation mode is also a string, chosen from one of the following:

r	Open file for reading
w	Open file for writing
a	Open file for appending
rw	Open file for reading and writing (some systems)

This mode string specifies the way in which the file will be used. Finally, returnpointer is a pointer to a FILE structure which is the whole object of calling this function. If the file (which was named) opened successfully when fopen() was called, returnpointer is a pointer to the file portal. If the file could not be opened, this pointer is set to the value NULL. This should be tested for, because it would not make sense to attempt to write to a file which could not be opened or created, for whatever reason.

A read only file is opened, for example, with some program code such as:

```
FILE *fp;
if ((fp = fopen ("filename","r")) == NULL)
{
printf ("File could not be opened\n");
error_handler();
}
```

A question which springs to mind is: what happens if the user has to type in the name of a file while the program is running? The solution to this problem is quite simple.

```
char *filename() /* return filename */
{ static char *filenm = "........................";
do
{
printf ("Enter filename :");
scanf ("%24s",filenm);
skipgarb();
}
while (strlen(filenm) == 0);
return (filenm);
}
```

This function makes file opening simple. The programmer would now write something like:

```
FILE *fp;
char *filename();
if ((fp = fopen (filename(),"r")) == NULL)
{
printf ("File could not be opened\n");
error_handler();
}
```

and then the user of the program would automatically be prompted for a filename. Once a file has been opened, it can be read from or written to using the other library functions (such as fprintf() and fscanf()) and then finally the file has to be closed again.

11.4.2 fclose

A file is closed by calling the function fclose(). fclose() has the syntax:

```
int returncode;
FILE *fp;
returncode = fclose (fp);
```

fp is a pointer to the file which is to be closed and returncode is an integer value which is 0 if the file was closed successfully. fclose() prompts the file manager to finish off its dealings with the named file and to close the portal which the operating system reserved for it. When closing a file, a program needs to do something like the following:

```
if (fclose(fp) != 0)
{
printf ("File did not exist.\n");
error_handler();
}
```

11.4.3 feof

This function returns a true or false result. It tests whether or not the end of a file has been reached and if it has it returns 'true' (which has any value except zero); otherwise the function returns 'false' (which has the value zero). The form of a statement using this function is:

```
FILE *fp;
int outcome;
outcome = feof(fp);
```

Most often feof() will be used inside loops or conditional statements. For example:

consider a loop which reads characters from an open file, pointed to by fp. A call to feof() is required in order to check for the end of the file.

```
while (!feof(fp))
{
ch = getc(fp);
}
```

Translated into pidgin English, this code reads: 'while NOT end of file, ch equals get character from file'. In better(?) English the loop continues to fetch characters as long as the end of the file has not been reached. Notice the logical NOT operator ! which stands before feof().

11.4.4 fseek, ftell

fseek : fseek function is used to move the file position to a desired location within the file. It takes the following form:

fseek(file ptr, offset, position)

File ptr is a pointer to the file concerned, offset is a number variable of type long and positi n is an nteger number. The offset specifics the number of positions(bytes) to the moved from the location specified by position.

The position can take one of the following three values

Values	Meaning
0	Beginning of file
1	Current position
2	End of file

offset may be positive meaning move forwards or negative meaning move backwards. The following examples illustrate the operation of the fseek function:

Statement	Meaning
fseek(fp,0L,0)	Go to beginning
fseek(fp, 0L, 1)	Stays at current position
fseek(fp, 0L, 2)	Go to end of the file, past the last character of the file
fseek(fp, m, 0)	Move to (m+1)th byte in the file
fseek(fp, m, 1)	Go forward by m bytes
fseek(fp, -m, 1)	Go backward by m bytes from the current position
fseek(fp, - m, 2)	Go backward by m bytes from the end

ftell : ftell takes a file pointer and returns a number of type long that corresponds to the current position. This function is useful in saving the current position of a file, which can be used later in the program. It takes the following form

```
n = ftell(fp);
```

n would give the relative offset(in bytes) of the current position. This means that n bytes have already been read (or written).ftell() tells a program its position within a file, opened by fopen() where as fseek() seeks a specified place within a file, opened by fopen(). Normally high level read/write functions perform as much management over positions inside files as the programmer wants, but in the event of their being insufficient, these two routines can be used. The form of the function calls is:

```
long int pos;
FILE *fp;
pos = ftell(fp);
```

fp is an open file, which is in some state of being read or written to. pos is a long integer value which describes the position in terms of the number of characters from the beginning of the file. Aligning a file portal with a particular place in a file is more sophisticated than simply taking note of the current position. The call to fseek() looks like this:

```
long int pos;
int mode,returncode;
FILE *fp;
returncode = fseek (fp,pos,mode);
```

The parameters have the following meanings. fp is a pointer to a file opened by fopen(). pos some way of describing the position required within a file. mode is an integer which specifies the way which pos is to be interpreted. Finally, returncode is an integer whose value is 0 if the operation wa successful and -1 if there was an error.

0 pos is an offset measured relative to the beginning of the file.

1 pos is an offset measured relative to the current position.

2 pos is an offset measured relative to the end of the file.

Some examples help to show how this works in practice:

```
long int pos = 50;
int mode = 0,returncode;
FILE *fp;
if (fseek (fp,pos,mode) != 0) /* find 50th character */
{
printf("Error!\n");
}
fseek(fp,0L,0); /* find beginning of file */
fseek(fp,2L,0); /* find the end of a file */
if (fseek (fp,10L,1) != 0) /* move 10 char's forward */
{
printf("Error!\n");
}
```

The L's indicate long constants.

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
long int ftell(FILE *stream);
```

The fseek function sets the file position indicator for the stream pointed to by stream. For a binary stream, the new position, measured in characters from the beginning of the file, is obtained by adding offset to the position specified by whence. Three macros in stdio.h called SEEK_SET, SEEK_CUR, and SEEK_END expand to unique values. If the position specified by whence is SEEK_SET, the specified position is the beginning of the file; if whence is SEEK_END, the specified position is the end of the file; and if whence is SEEK_CUR, the specified position is the current file position. A binary stream need not meaningfully support fseek calls with a whence value of SEEK_END.

For a text stream, either offset shall be zero, or offset shall be a value returned by an earlier call to the ftell function on the same stream and whence shall be SEEK_SET.

The fseek function returns nonzero only for a request that cannot be satisfied.

The ftell function obtains the current value of the file position indicator for the stream pointed to by stream.

For a binary stream, the value is the number of characters from the beginning of the file; for a text ream, its file position indicator contains unspecified information, usable by the fseek function for returning ne file position indicator for the stream to its position at the time of the ftell call; the difference between two such return values is not necessarily a meaningful measure of the number of characters written or read. If successful, the ftell function returns the current value of the file position indicator for the stream. On failure, the ftell function returns -1L and stores an implementation-defined positive value in errno.

11.4.5 rewind

rewind : rewind takes a file pointer and resets the position to the start of the file.

for e.g.

```
rewind(fp);
n = ftell(fp);
```

n would return 0. rewind() is a macro, based upon fseek(), which resets a file position to the beginning of the file. It is equivalent to

```
(void)fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the stream is also cleared.

e.g.

```
FILE *fp;
rewind(fp);
fseek(fp,0L,0); /* = rewind() */
```

11.4.6 fgetc, fputc, fread, fwrite

Single Character I/O

There are commonly four functions/macros which perform single character input/output to or from files. They are analogous to the functions/macros

getchar()   putchar()

for the standard I/O files and they are called:

getc()   ungetc()   putc()   fgetc()   fputc()

getc() and fgetc()

The difference between getc() and fgetc() will depend upon a particular system. It might be that getc() is implemented as a macro, whereas fgetc() is implemented as a function or vice versa. One of these alternatives may not be present at all in a library. Check the manual, to be sure! Both getc() and fgetc() fetch a single character from a file:

```
FILE *fp;
char ch;
/* open file */
ch = getc (fp);
ch = fgetc (fp);
```

These functions return a character from the specified file if they operated successfully, otherwise they return EOF to indicate the end of a file or some other error. Apart from this, these functions/macros are quite unremarkable.

ungetc()

ungetc() is a function which 'un-gets' a character from a file. That is, it reverses the effect of the last get operation. This is not like writing to a file, but it is like stepping back one position within the file. The purpose of this function is to leave the input in the correct place for other functions in a program when other functions go too far in a file. An example of this would be a program which looks for a word in a text file and processes that word in some way.

```
while (getc(fp) != ' ')
{
}
```

The program would skip over spaces until it found a character and then it would know that this was the start of a word. However, having used getc() to read the first character of that word, the position in the file would be the second character in the word! This means that, if another function wanted to read that word from the beginning, the position in the file would not be correct, because the first character would already have been read.

The solution is to use ungetc() to move the file position back a character:

```
int returncode;
returncode = ungetc(fp);
```

The returncode is EOF if the operation was unsuccessful.

putc() and fputc()

```
#include <stdio.h>
int putc(int c, FILE *stream);
```

The putc function is equivalent to fputc, except that if it is implemented as a macro, it may evaluate stream more than once, so the argument should never be an expressic 1 with side effects. The function returns the character written, unle write error occurs, in which case the error indicator for the stream is set and the function returns EOF.

These two functions write a single character to the output file, pointed to by fp. As with getc(), one of these may be a macro. The form of these statements is:

```
FILE *fp;
char ch;
int returncode;
returncode = fputc (ch,fp);
returncode = putc (ch,fp);
```

The returncode is the ascii code of the character sent, if the operation was successful, otherwise it is EOF.

The putchar function

```
#include <stdio.h>

int putchar(int c);
```

The putchar function is equivalent to putc with the second argument stdout. It returns the character written, unless a write error occurs, in which case the error indicator for stdout is set and the function returns EOF.

fgets() and fputs()

Just as gets() and puts() fetched and sent strings to standard input/output files stdin and stdout, so fgets() and fputs() send strings to generalized files. The fgets/fputs function reads/writes the string pointed to by s to the stream pointed to by stream. The terminating null character is not written. The function returns EOF if a read/write error occurs, otherwise it returns a nonnegative value.

The form of an fgets() statement is as follows:

```
char *strbuff,*returnval;

int n;

FILE *fp;

returnval = fgets (strbuff,n,fp);
```

strbuff is a pointer to an input buffer for the string; fp is a pointer to an open file. returnval is a pointer to a string: if there was an error in fgets() this pointer is set to the value NULL, otherwise it is set to the value of "strbuff". No more than (n-1) characters are read by fgets() so the programmer has to be sure to set n equal to the size of the string buffer. (One byte is reserved for the NULL terminator.) The form of an fputs() statement is as follows:

```
char *str;

int returnval;

FILE *fp;

returnval = fputs (str,fp);
```

Where str is the NULL terminated string which is to be sent to the file pointed to by fp. returnval is set to EOF if there was an error in writing to the file.

```
#include <stdio.h>

int fputc(int c, FILE *stream);
```

The fputc function writes the character specified by c (converted to an unsigned char) to the stream pointed to by stream at the position indicated by the associated file position indicator (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream is opened with append mode, the character is appended to the output stream. The function returns the character written, unless a write error occurs, in which case the error indicator for the stream is set and fputc returns EOF.

The puts function

```
#include <stdio.h>

int puts(const char *s);
```

The puts function writes the string pointed to by s to the stream pointed to by stdout, and appends a new-line character to the output. The terminating null character is not written. The function returns EOF if a write error occurs; otherwise, it returns a nonnegative value.

Direct input functions : fread , fwrite

fread :

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

The fread function reads, into the array pointed to by ptr, up to nmemb elements whose size is specified by size, from the stream pointed to by stream. The file position indicator for the stream (if defined) is advanced by the number of characters successfully read. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. If a partial element is read, its value is indeterminate. The fread function returns the number of elements successfully read, which may be less than nmemb if a read error or end-of-file is encountered. If size or nmemb is zero, fread returns zero and the contents of the array and the state of the stream remain unchanged.

fwrite :

```
#include <stdio.h>

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

The fwrite function writes, from the array pointed to by ptr, up to nmemb elements whose size is specified by size to the stream pointed to by stream. The file position indicator for the stream (if defined) is advanced by the number of characters successfully written. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. The function returns the number of elements successfully written, which will be less than nmemb only if a write error is encountered.

11.4.7 fscanf, fprintf

fscanf : The analogue of scanf() is fscanf() and, as with fprintf(), this function differs from its standard I/O counterpart only in one extra parameter: a file pointer. The form of an fscanf() statement is:

```
FILE *fp;

int n;

n = fscanf (fp,"string",pointers);
```

where n is the number of items matched in the control string and fp is a pointer to the file which is to be read from. For example, assuming that fp is a pointer to an open file:

```
int i = 10;

float x = -2.356;

char ch = 'x';

fscanf (fp, "%d %f %c", &i, &x, &ch);
```

The remarks which were made about scanf() also apply to this function: fscanf() is a 'dangerous' function in that it can easily get out of step with the input data unless the input is properly formatted.

```
#include <stdio.h>

int fscanf(FILE *stream, const char *format, ...);

int scanf(const char *format, ...);

int sscanf(const char *s, const char *format, ...);
```

The fscanf function reads input from the stream pointed to by stream, under control of the string pointed to by format that specifies the admissible sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: one or more white-space characters; an ordinary multibyte character (neither % or a white-space character); or a conversion specification. Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

An optional assignment-suppressing character *.

An optional nonzero decimal integer that specifies the maximum field width.

An optional **h, l** (ell) or **L** indicating the size of the receiving object. The conversion specifiers **d, i,** and **n** shall be preceded by h if the corresponding argument is a pointer to short int rather than a pointer to int, or by l if it is a pointer to long int. Similarly, the conversion specifiers **o, u,** and **x** shall be preceded by **h** if the corresponding argument is a pointer to unsigned short int rather than unsigned int, or by l if it is a pointer to unsigned long int. Finally, the conversion specifiers e, f, and g shall be preceded by l if the corresponding argument is a pointer to double rather than a pointer to float, or by L if it is a pointer to long double. If an **h, l,** or **L** appears with any other format specifier, the behavior is undefined.

A character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The fscanf function executes each directive of the format in turn. If a directive fails, as detailed below, the fscanf function returns. Failures are described as input failures (due to the unavailability of input characters) or matching failures (due to inappropriate input). A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread) or until no more characters remain unread. A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If one of the characters differs from one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps: Input white-space characters (as specified by the isspace function) are skipped, unless the specification includes a [, c, or n specifier. (The white-space characters are not counted against the specified field width.) An input item is read from the stream, unless the specification includes an n specifier. An input item is defined as the longest matching sequences of input characters, unless that exceeds a specified field width, in which case it is the initial subsequence of that length in the sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.

Except in the case of a % specifier, the input item (or, in the case of a %n directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails; this condition is a matching failure. Unless assignment suppression was indicated by a *, the result of the conversion is placed in the object pointed to by the first argument following the format argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined. The following conversion specifiers are valid:

d : Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the strtol function with the value 10 for the base argument. The corresponding argument shall be a pointer to integer.

I : Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the strtol function with the value 0 for the base argument. The corresponding argument shall be a pointer to integer.

o : Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the strtoul function with the value 8 for the base argument. The corresponding argument shall be a pointer to unsigned integer.

u : Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the strtoul function with the value 10 for the base argument. The corresponding argument shall be a pointer to unsigned integer.

x : Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the strtoul function with the value 16 for the base argument. The corresponding argument shall be a pointer to unsigned integer.

e, f, g: Matches an optionally signed floating-point number, whose format is the same as expected for the subject string of the strtod function. The corresponding argument will be a pointer to floating.

s : Matches a sequence of non-white-space characters. (No special provisions are made for multibyte characters.) The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically.

[: Matches a nonempty sequence of characters (no special provisions are made for multibyte characters) from a set of expected characters (the *scanset*). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically. The conversion specifier includes all subsequent characters in the format string, up to and including the matching right bracket (]). The characters between the brackets (the *scanlist*) comprise the scanset, unless the character after the left bracket is a circumflex (^), in which case the scanset contains all the characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with [] or [^], the right-bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification; otherwise, the first right bracket character is the one that ends the specification. If a - character is in the scanlist and is not the first, nor the second where the first character is a ^, nor the last character, the behavior is implementation-defined.

c : Matches a sequence of characters (no special provisions are made for multibyte characters) of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence. No null character is added.

p : Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the fprintf function. The corresponding argument shall be a pointer to void. The interpretation of the input then is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the %p conversion is undefined.

n : No input is consumed. The corresponding argument shall be a pointer to integer into which is to be written the number of characters read from the input stream so far by this call to the fscanf function. Execution of a %n directive does not increment the assignment count returned at the completion of execution of the fscanf function.

% : Matches a single %; no conversion or assignment occurs. The complete conversion specification shall be

%%. : If a conversion specification is invalid, the behavior is undefined. The conversion specifiers E, G, and X are also valid and behave the same as, respectively, e, g, and x.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure. If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the %n directive. The fscanf function returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, the fscanf funciton returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure. The scanf function is equivalent to fscanf with the argument stdin interposed before the arguments to scanf. Its return value is similar to that of fscanf. The sscanf function is equivalent to fscanf, except that the argument s specifies a string from which the input is to be obtained, rather than from a stream. Reaching the end of the string is equivalent to encountering the end-of-file for the fscanf function. If copying takes place between objects that overlap, the behavior is undefined.

fprintf()

This is the highest level function which writes to files. Its name is meant to signify "fileprint- formatted" and it is almost identical to its stdout counterpart printf(). The form of the fprintf() statement is as follows:

```
fprintf (fp,"string",variables);
```

where fp is a file pointer, string is a control string which is to be formatted and the variables are those which are to be substituted into the blank fields of the format string.

For example, assume that there is an open file, pointed to by fp:

```
int i = 12;

float x = 2.356;

char ch = 's';

fprintf (fp, "%d %f %c", i, x, ch);
```

The conversion specifiers are identical to those for printf(). In fact fprintf() is related to printf() in a very simple way: the following two statements are identical.

```
printf ("Hello world %d", 1);

fprintf (stdout,"Hello world %d", 1);

#include <stdarg.h>

#include <stdio.h>

int fprintf(FILE *stream, const char *format, ...);

int printf(const char *format, ...);

int sprintf(char *s, const char *format, ...);

int vfprintf(FILE *stream, const char *format, va_list arg);

int vprintf(const char *format, va_list arg);

int vsprintf(char *s, const char *format, va_list arg);
```

Note: Some length specifiers and format specifiers are new in C99. These may not be available in older compilers and versions of the stdio library, which adhere to the C89/C90 standard. Wherever possible, the new ones will be marked with (C99).

The fprintf function writes output to the stream pointed to by stream under control of the string pointed to by format that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The fprintf function returns when the end of the format string is encountered. The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream. Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

Zero or more flags (in any order) that modify the meaning of the conversion specification. An optional minimum field width. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk * (described later) or a decimal integer. (Note that 0 is taken as a flag, not as the beginning of a field width.) An optional precision that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversions, the number of digits to appear after the decimal-point character for a, A, e, E, f, and F conversions, the maximum number of significant digits for the g and G conversions, or the maximum number of characters to be written from a string in s conversions. The precision takes the form of a period (.) followed either by an asterisk * (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined. Floating-point numbers are *rounded* to fit the precision; i.e. printf("%1.1f\n", 1.19); produces 1.2. An optional length modifier that specifies the size of the argument. A conversion specifier character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an int argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a - flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted. The flag characters and their meanings are:

- : The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)

+ : The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not specified. The results of all floating conversions of a negative zero, and of negative values that round to zero, include a minus sign.)

space : If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the space and + flags both appear, the space flag is ignored.

: The result is converted to an "alternative form". For o conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For x (or X) conversion, a nonzero result has 0x (or 0X) prefixed to it. For a, A, e, E, f, F, g, and G conversions, the result always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For g and G conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

0 : For d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the 0 and - flags both appear, the 0 flag is ignored. For d, i, o, u, x, and X conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behavior is undefined. The length modifiers and their meanings are:

hh : (C99) Specifies that a following d, i, o, u, x, or X conversion specifier applies to a signed char or unsigned char argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to signed char or unsigned char before printing); or that a following n conversion specifier applies to a pointer to a signed char argument.

h : Specifies that a following d, i, o, u, x, or X conversion specifier applies to a short int or unsigned short int argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to short int or unsigned short int before printing); or that a following n conversion specifier applies to a pointer to a short int argument.

l (ell) : Specifies that a following d, i, o, u, x, or X conversion specifier applies to a long int or unsigned long int argument; that a following n conversion specifier applies to a pointer to a long int argument; (C99) that a following c conversion specifier applies to a wint_t argument; (C99) that a following s conversion specifier applies to a pointer to a wchar_t argument; or has no effect on a following a, A, e, E, f, F, g, or G conversion specifier.

ll (ell-ell) : (C99) Specifies that a following d, i, o, u, x, or X conversion specifier applies to a long long int or unsigned long long int argument; or that a following n conversion specifier applies to a pointer to a long long int argument.

j : (C99) Specifies that a following d, i, o, u, x, or X conversion specifier applies to an intmax_t or uintmax_t argument; or that a following n conversion specifier applies to a pointer to an intmax_t argument.

z : (C99) Specifies that a following d, i, o, u, x, or X conversion specifier applies to a size_t or the corresponding signed integer type argument; or that a following n conversion specifier applies to a pointer to a signed integer type corresponding to size_t argument.

t : (C99) Specifies that a following d, i, o, u, x, or X conversion specifier applies to a ptrdiff_t or the corresponding unsigned integer type argument; or that a following n conversion specifier applies to a pointer to a ptrdiff_t argument.

L : Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a long double argument. If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

d, i : The int argument is converted to signed decimal in the style *[“]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

o, u, x, X : The unsigned int argument is converted to unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x or X) in the style *dddd*; the letters **abcdef** are used for x conversion and the letters **ABCDEF** for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

f, F : A double argument representing a (finite) floating-point number is converted to decimal notation in the style *[“]ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

(C99) A double argument representing an infinity is converted in one of the styles *[-]*inf or *[-]*infinity — which style is implementation-defined. A double argument representing a NaN is converted in one of the styles *[-]*nan or *[-]*nan(*n-char-sequence*) — which style, and the meaning of any *n-char-sequence*, is implementation-defined. The F conversion specifier produces INF, INFINITY, or NAN instead of inf, infinity, or nan, respectively. (When applied to infinite and NaN values, the -, +, and *space* flags have their usual meaning; the # and 0 flags have no effect.)

e, E : A double argument representing a (finite) floating-point number is converted in the style *[“]d.ddd*e±*dd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The E conversion specifier produces a number with E instead of e introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero. (C99) A double argument representing an infinity or NaN is converted in the style of an f or F conversion specifier.

g, G : A double argument representing a (finite) floating-point number is converted in style f or e (or in style F or E in the case of a G conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style e (or E) is used only if the exponent resulting from such a conversion is less than –4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result unless the # flag is specified; a decimal-point character appears only if it is followed by a digit. (C99) A double argument representing an infinity or NaN is converted in the style of an f or F conversion specifier.

a, A : (C99) A double argument representing a (finite) floating-point number is converted in the style *[“]*0x*h.hhhh*p±*d*, where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character (Binary implementations can choose the hexadecimal digit to the left of the decimal-point character so that subsequent digits align to nibble [4-bit] boundaries.) and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and FLT_RADIX is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and FLT_RADIX is not a power of 2, then the precision is sufficient to distinguish (The precision *p* is sufficient to distinguish values of the source type if $16p–1 > bn$ where *b* is FLT_RADIX and *n* is the number of base-*b* digits in the significand of the source type. A smaller *p* might suffice depending on the implementation's scheme for determining the digit to the left of the decimal-point character.) values of type double, except that trailing zeros may be omitted; if the precision is zero and the # flag is not specified, no decimal-point character appears. The letters **abcdef** are used for a conversion and the letters **ABCDEF** for A conversion. The A conversion specifier produces a number with X and P instead of x and p. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.

A double argument representing an infinity or NaN is converted in the style of an f or F conversion specifier.

c : If no l length modifier is present, the int argument is converted to an unsigned char, and the resulting character is written. (C99) If an l length modifier is present, the wint_t argument is converted as if by an ls conversion specification with no precision and an argument that points to the initial element of a two-element array of wchar_t, the first element containing the wint_t argument to the lc conversion specification and the second a null wide character.

s : If no l length modifier is present, the argument shall be a pointer to the initial element of an array of character type. (No special provisions are made for multibyte characters.) Characters from the array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character. (C99) If an l length modifier is present, the argument shall be a pointer to the initial element of an array of wchar_t type. Wide characters from the array are converted to multibyte characters (each as if by a call to the wcrtomb function, with the conversion state described by an mbstate_t object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array shall contain a null wide character. If a precision is specified, no more than that many characters (bytes) are written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the multibyte character sequence length given by the precision, the function would need to access a wide character one past the end of the array. In no case is a partial multibyte character written. (Redundant shift sequences may result if multibyte characters have a statedependent encoding.)

p The argument shall be a pointer to void. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner.

n The argument shall be a pointer to signed integer into which is written the number of characters written to the output stream so far by this call to fprintf. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

% A % character is written. No argument is converted. The complete conversion specification shall be %%. If a conversion specification is invalid, the behavior is undefined. If any argument is not the correct type for the corresponding coversion specification, the behavior is undefined. In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result. For a and A conversions, if FLT_RADIX is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision. It is recommended practice that if FLT_RADIX is not a power of 2, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction. It is recommended practice that for e, E, f, F, g, and G conversions, if the number of significant decimal digits is at most DECIMAL_DIG, then the result should be correctly rounded. (For binary-to-decimal conversion, the result format's values are the numbers representable with the given format specifier. The number of significant digits is determined by the format specifier, and in the case of fixed-point conversion by the source value as well.) If the number of significant decimal digits is more than DECIMAL_DIG but the source value is exactly representable with DECIMAL_DIG digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings *L* < *U*, both having DECIMAL_DIG significant digits; the value of the resultant decimal string *D* should satisfy *L d" D d" U*, with the extra stipulation that the error should have a correct sign for the current rounding direction.

The fprintf function returns the number of characters transmitted, or a negative value if an output or encoding error occurred. The printf function is equivalent to fprintf with the argument stdout interposed before the arguments to printf. It returns the number of characters transmitted, or a negative value if an output error occurred.

The sprintf function is equivalent to fprintf, except that the argument s specifies an array into which the generated input is to be written, rather than to a stream. A null character is written at the end of the characters written; it is not counted as part of the returned sum. If copying takes place between objects that overlap, the behavior is undefined. The function returns the number of characters written in the array, not counting the terminating null character.

The vfprintf function is equivalent to fprintf, with the variable argument list replaced by arg, which shall have been initialized by the va_start macro (and possibly subsequent va_arg calls). The vfprintf function does not invoke the va_end macro. The function returns the number of characters transmitted, or a negative value if an output error occurred.

The vprintf function is equivalent to printf, with the variable argument list replaced by arg, which shall have been initialized by the va_start macro (and possibly subsequent va_arg calls). The vprintf function does not invoke the va_end macro. The function returns the number of characters transmitted, or a negative value if an output error occurred.

The vsprintf function is equivalent to sprintf, with the variable argument list replaced by arg, which shall have been initialized by the va_start macro (and possibly subsequent va_arg calls). The vsprintf function does not invoke the va_end macro. If copying takes place between objects that overlap, the behavior is undefined. The function returns the number of characters written into the array, not counting the terminating null character.

11.4.8 Other file access functions

The fflush function

```
#include <stdio.h>

int fflush(FILE *stream);
```

If stream points to an output stream or an update stream in which the most recent operation was not input, the fflush function causes any unwritten data for that stream to be deferred to the host environment to be written to the file; otherwise, the behavior is undefined.

If stream is a null pointer, the fflush function performs this flushing action on all streams for which the behavior is defined above.

The fflush functions returns EOF if a write error occurs, otherwise zero.

The reason for having a fflush function is because streams in C can have buffered input/output; that is, functions that write to a file actually write to a buffer inside the FILE structure. If the buffer is filled to capacity, the write functions will call fflush to actually "write" the data that is in the buffer to the file. Because fflush is only called every once in a while, calls to the operating system to do a raw write are minimized.

The setbuf function

```
#include <stdio.h>

void setbuf(FILE *stream, char *buf);
```

Except that it returns no value, the setbuf function is equivalent to the setvbuf function invoked with the values _IOFBF for mode and BUFSIZ for size, or (if buf is a null pointer) with the value _IONBF for mode.

The setvbuf function

```
#include <stdio.h>

int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

The setvbuf function may be used only after the stream pointed to by stream has been associated with an open file and before any other operation is performed on the stream. The argument mode determines how the stream will be buffered, as follows: _IOFBF causes input/output to be fully buffered; _IOLBF causes input/output to be line buffered; _IONBF causes input/output to be unbuffered. If buf is not a null pointer, the array it points to may be used instead of a buffer associated by the setvbuf function. (The buffer must have a lifetime at least as great as the open stream, so the stream should be closed before a buffer that has automatic storage duration is deallocated upon block exit.) The argument size specifies the size of the array. The contents of the array at any time are indeterminate.

The setvbuf function returns zero on success, or nonzero if an invalid value is given for mode or if the request cannot be honored.

Functions that Modify the File Position Indicator

The stdio.h library has five functions that affect the file position indicator besides those that do reading or writing: fgetpos, fseek, fsetpos, ftell, and rewind.

The fseek and ftell functions are older than fgetpos and fsetpos.

The fgetpos and fsetpos functions

```
#include <stdio.h>

int fgetpos(FILE *stream, fpos_t *pos);

int fsetpos(FILE *stream, const fpos_t *pos);
```

The fgetpos function stores the current value of the file position indicator for the stream pointed to by stream in the object pointed to by pos. The value stored contains unspecified information usable by the fsetpos function for repositioning the stream to its position at the time of the call to the fgetpos function. If successful, the fgetpos function returns zero; on failure, the fgetpos function returns nonzero and stores an implementation-defined positive value in errno. The fsetpos function sets the file position indicator for the stream pointed to by stream according to the value of the object pointed to by pos, which shall be a value obtained from an earlier call to the fgetpos function on the same stream.

A successful call to the fsetpos function clears the end-of-file indicator for the stream and undoes any effects of the ungetc function on the same stream. After an fsetpos call, the next operation on an update stream may be either input or output. If successful, the fsetpos function returns zero; on failure, the fsetpos function returns nonzero and stores an implementation-defined positive value in errno.

The standard library provides an error function/macro which returns a true/false result according to whether or not the last filing function call returned an error condition. This is called ferror(). To check for an error in an open file, pointed to by fp:

```
FILE *fp;

if(ferror(fp))

{

error_handler();

}
```

This function/macro does not shed any light upon the cause of errors, only whether errors have occurred at all. A detailed diagnosis of what went wrong is only generally possible by means of a deeper level call to the disk operating system (DOS).

11.4.9 Error Handling Functions

The clearerr function

```
#include <stdio.h>

void clearerr(FILE *stream);
```

The clearerr function clears the end-of-file and error indicators for the stream pointed to by stream.

The feof function

```
#include <stdio.h>

int feof(FILE *stream);
```

The feof function tests the end-of-file indicator for the stream pointed to by stream and returns nonzero if and only if the end-of-file indicator is set for stream, otherwise it returns zero.

The ferror function

```
#include <stdio.h>

int ferror(FILE *stream);
```

The ferror function tests the error indicator for the stream pointed to by stream and returns nonzero if and only if the error indicator is set for stream, otherwise it returns zero.

The perror function

```
#include <stdio.h>

void perror(const char *s);
```

The perror function maps the error number in the integer expression errno to an error message. It writes a sequence of characters to the standard error stream thus: first, if s is not a null pointer and the character pointed to by s is not the null character, the string pointed to by s followed by a colon (:) and a space; then an appropriate error message string followed by a new-line character. The contents of the error message are the same as those returned by the strerror function with the argument errno, which are implementation-defined.

11.5 File handling through programs

We frequently use files for storing information which can be processed by our programs. In order to store information permanently and retrieve it we need to use files. Files are not only used for data. Our programs are also stored in files. The editor which you use to enter your program and save it, simply manipulates files for you. For eg. the Unix commands cat, cp, cmp are all programs which process your files.

In order to use files we have to learn about ***File I/O*** i.e. how to write information to a file and how to read information from a file. We will see that file I/O is almost identical to the terminal I/O that we generally use. The primary difference between manipulating files and doing terminal I/O is that we must specify in our programs which files we wish to use.

As you know, you can have many files on your disk. If you wish to use a file in your programs, then you must specify which file or files you wish to use. Specifying the file you wish to use is referred to as

opening the file. When you open a file you must also specify what you wish to do with it i.e. **Read** from the file, **Write** to the file, or both. Because you may use a number of different files in your program, you must specify when reading or writing which file you wish to use. This is accomplished by using a variable called a **file pointer.** Every file you open has its own file pointer variable. When you wish to write to a file you specify the file by using its file pointer variable. You declare these file pointer variables as follows:

FILE *fopen(), *fp1, *fp2, *fp3;

The variables fp1, fp2, fp3 are file pointers. You may use any name you wish. You should note that a file pointer is simply a variable like an integer or character. It does **not** *point* to a file or the data in a file. It is simply used to indicate which file your I/O operation refers to.

The file <stdio.h> contains declarations for the Standard I/O library and should always be **included** at the very beginning of C programs using files. Constants such as FILE, EOF and NULL are defined in <stdio.h>.

The function **fopen** is one of the Standard Library functions and returns a file pointer which you use to refer to the file you have opened e.g.

fp = fopen("prog.c", "r") ;

The above statement **opens** a file called prog.c for **reading** and associates the file pointer fp with the file. When we wish to access this file for I/O, we use the file pointer variable fp to refer to it. You can have up to about 20 files open in your program - you need one file pointer for each file you intend to use (Check out the manual of the language version).

File I/O

The Standard I/O Library provides similar routines for file I/O to those used for standard I/O. The routine getc(fp) is similar to getchar() and putc(c,fp) is similar to putchar(c). Thus the statement

c = getc(fp);

reads the next character from the file referenced by fp and the statement

RULE: Always check when opening files, that fopen succeeds in opening the files appropriately. If you attempt to read from an non-existent file, your program will crash!!

putc(c,fp);

writes the character c into file referenced by fp.

Files of records

Most database files are binary files which can logically be divided into fixed length records. Each record will consist of data that conforms to a previously defined structure. In C, this structure is a *struct* data type. You should recall that struct data types were defined and discussed in C defines a number of features, additional to the ones already defined for text files, to assist programmers to manipulate binary files, and in particular files of records. These include the ability to:

- read and write to the one file without closing the file after a read and reopening it before a write
- read or write a block of bytes (which generally correspond to a logical record) without readin of writing the block one character at a time
- position the file pointer to any byte within the file, thus giving the file direct access capabilities.

Processing files of records

The key functions required to process a file of records are:

fopen: This function is almost identical to the function used to open text files. If the file is opened correctly then the function will return a non-zero value. If an error occurs the function will return a NULL value. The nature of the error can be determined by the ferror function.

fclose: This function is common to both text and binary files. The function closes the specified file at flushes the output buffer to disk.

fseek: fseek will position the file pointer to a particular byte within the file. The file pointer is a parameter maintained by the operating system and derternines where the next read will comes from, or to where the next write will go.

fread: The fread function will read a specified number of bytes from the current file position and store them in a data variable. It is the programmer's responsibility to ensure that the data type of the variable matches the data being read, and that the number of characters that are read will fit into the allocated data space.

fwrite: The fwrite function will write a specified number of bytes transferred from the specified data variable to the disk starting at the current file position. It is the programmer's responsibility to ensure that the file position is located correctly before the block is written to the file.

ferror: This function will return the status of the last disk operation. A value of zero indicates that no error has occurred. A nonzero value indicates that the last disk operation resulted in an error. Consult the manual for an explanation of the error numbers returned by this function.

The following examples show how these functions can be used in a C program. Consider the program [BINARY-W.C]. This program creates a file of four records.

Each record contains four fields.

```
#include "stdio.h"
#include "string.h"
struct record {
char last_name[20];
char first_name[15];
int age;
float salary;
};
typedef struct record person;
FILE *people;
void main()
{
person employee;
people = fopen("PEOPLE.DAT", "wb");
strcpy(employee.last_name, "CAMPBELL");
strcpy(employee.first_name, "MALCOLM");
employee.age = 40;
employee.salary = 35123.0;
```

```
fwrite(&employee, sizeof(employee), 1, people);
strcpy(employee.last_name, "GOLDSMITH");
strcpy(employee.first_name, "SALLY");
employee.age = 35;
employee.salary = 50456.0;
fwrite(&employee, sizeof(employee), 1, people);
strcpy(employee.last_name, "GOODMAN");
strcpy(employee.first_name, "ALBERT");
employee.age = 42;
employee.salary = 97853.0;
fwrite(&employee, sizeof(employee), 1, people);
strcpy(employee.last_name, "ROGERS");
strcpy(employee.first_name, "ANNE");
employee.age = 50;
employee.salary = 100254.0;
fwrite(&employee, sizeof(employee), 1, people);
fclose(people);
}
```

11.6 Summary

All files, irrespective of the data they contain or the methods used to process them, have certain important properties. They have a name. They must be opened and closed. They can be written to, or read from, or appended to. Conceptually, until a file is opened nothing can be done to it. When it is opened, we may have access to it at its beginning or end. To prevent accidental misuse, we must tell the system which of the three activities (reading, writing, or appending) we will be performing on it. When we are finished using the file, we must close it. If the file is not closed the operating system cannot finish updating its own housekeeping records and data in the file may be lost.

11.7 Glossary

C99 is a modern dialect of the C programming language. It extends the previous version (C89) to make better use of available computer hardware and to better employ the latest advances in compiler technology. (The C standard is often refered to as CXX, where XX is the year in which it was adopted. So, C99, is just the nickname for the latest version of the standard.)

file An object that can be written to, or read from, or both. A file has certain attributes, including access permissions and type. File types include regular file, character special file, block special file, FIFO special file and directory. Other types of files may be supported by the implementation.

file access permissions The standard file access control mechanism uses the file permission bits, as described below. These bits are set at the time of file creation by functions such as ***open()***, ***creat()***, ***mkdir()*** and ***mkfifo()*** and are changed by ***chmod()***. These bits are read by ***stat()*** or ***fstat()***.

filename A name consisting of 1 to {NAME_MAX} bytes used to name a file. The characters composing the name may be selected from the set of all character values excluding the slash character and the null byte. A filename is sometimes referred to as a *pathname component*. Filenames should be constricted from the portable filename character set because the use of other characters can be confusing or ambiguous in certain contexts. (For instance, the use of a colon (:) in a pathname could cause ambiguity if that pathname were included in a *PATH* definition.)

open file A file that is currently associated with a file descriptor.

standard error An output stream usually intended to be used for diagnostic messages.

standard input An input stream usually intended to be used for primary data input.

standard output An output stream usually intended to be used for primary data output.

standard utilities The utilities described in the XCU specification.

stream Appearing in lower case, a stream is a file access object that allows access to an ordered sequence of characters, as described by the ISO C standard. Such objects can be created by the *fdopen()*, *fopen()* or *popen()* functions, and are associated with a file descriptor. A stream provides the additional services of user-selectable buffering and formatted input and output.

11.8 Further Readings

1. Programming in 'C' – Balaguruswamy, Tata McGraw-Hill Publications
2. The C programming language – Dennis Ritchie, Prentice Hall, Delhi
3. Let us C – Yashwant Kanetkar, BPB Publication

11.9 Unit end questions

1. What are the following?
   a. File name
   b. File pointer
   c. File handle
2. What is the difference between high and low level filing?
3. Write a statement which opens a high level file for reading.
4. Write a statement which opens a low level file for writing.
5. Write a program which checks for illegal characters in text files. Valid characters are ASCII codes 10,13,and 32..126. Anything else is illegal for programs.
6. What statement performs formatted writing to text files?
7. Print out all the header files on your system so that you can see what is defined where.
8. Describe the different ways in which data files can be categorized in C.
9. What is the purpose of library function feof ? . How the feof function be utilized within a program that updates an unformatted data file

Unit 12 : Additional features of C

Structure of the Unit

12.0 Objective

12.1 Introduction

12.2 Preprocessor directive & their use

12.3 Macro Substitution directives

12.4 File inclusion directives

12.5 Enumeration

12.6 Storage classes

12.7 Command line arguments

12.8 Multifile programs using own header files

12.9 Summary

12.10 Glossary

12.11 Further Readings

12.12 Unit end questions

12.0 Objective

Students who complete this unit should be able to understand the following tasks:

- Understand the types of Preprocessor
- Understand the fundamental of following;
  - o Enumeration
  - o Storage classes
  - o Command line arguments
  - o Multifile programs using own header files

12.1 Introduction

This chapter focuses on the additional features of C Language : Enumeration, Storage classes, Command line arguments, Multifile programs using own header files.

12.2 Preprocessor directive & their use

The C preprocessor implements the macro language used to transform C, C++, and Objective-C programs before they are compiled. The C preprocessor, often known as *cpp*, is a *macro processor* that is used automatically by the C compiler to transform your program before compilation. It is called a macro processor because it allows you to define *macros*, which are brief abbreviations for longer constructs.

The C preprocessor is intended to be used only with C, C++, and Objective-C source code. In the past, it has been abused as a general text processor. For example, apostrophes will be interpreted as the beginning of character constants, and cause errors.

Wherever possible, you should use a preprocessor geared to the language you are writing in. Modern versions of the GNU assembler have macro facilities. Most high level programming languages have their own conditional compilation and inclusion mechanism. If all else fails, try a true general text processor, such as GNU M4.

12.3 Macro Substitution directives

A more advanced use of macros is also permitted by the preprocessor. This involves macros which accept parameters and hand back values. This works by defining a macro with some dummy parameter, say x. For example: a macro which is usually defined in one of the standard libraries is abs() which means the absolute or unsigned value of a number. It is defined below:

```
#define ABS(x) ((x) < 0) ? -(x) : (x)
```

The result of this is to give the positive (or unsigned) part of any number or variable. This would be no problem for a function which could accept parameters, and it is, in fact, no problem for macros. Macros can also be made to take parameters. Consider the ABS() example. If a programmer were to write ABS(4) then the preprocessor would substitute 4 for x. If a program read ABS(i) then the preprocessor would substitute i for x and so on.

(There is no reason why macros can't take more than one parameter too. The programmer just includes two dummy parameters with different names. Notice that this definition uses a curious operator which belongs to C:

<test> ? *<true result>* : *<false result>*

This is like a compact way of writing an if..then..else statement, ideal for macros. But it is also slightly different: it is an expression which returns a value, where as an if..then..else is a statement with no value. Firstly the test is made. If the test is true then the first statement is carried out, otherwise the second is carried out. As a memory aid, it could be read as:

if *<test>* then *<true result>* else *<false result>*

(Do not be confused by the above statement which is meant to show what a programmer might think. It is not a valid C statement.) C can usually produce much more efficient code for this construction than for a corresponding if-else statement.

Macros versus Functions

In the above example a macro was used to calculate the area of the circle. As we know, even a function can be written to calculate the area of the circle. Though macro calls are 'like' function calls, they are not really the same things. Then what is the difference between the two?

In a macro call the preprocessor replaces the macro template with its macro expansion, in a stupid, unthinking, literal way. As against this, in a function call the control is passed to a function along with certain arguments, some calculations are performed in the function and a useful value is returned back from the function.

This brings us to a question: when is it best to use macros with arguments and when is it better to use a function? Usually macros make the program run faster but increase the program size, whereas functions make the program smaller and compact.

If we use a macro hundred times in a program, the macro expansion goes into our source code at hundred different places, thus increasing the program size. On the other hand, if a function is used, then even if it is called from hundred different places in the program, it would take the same amount of space in the program.

But passing arguments to a function and getting back the returned value does take time and would therefore slow down the program. This gets avoided with macros since they have already been expanded and placed in the source code before compilation.

Moral of the story is—if the macro is simple and sweet like in our examples, it makes nice shorthand and avoids the overheads associated with function calls. On the other hand, if we have a fairly large macro and it is used fairly often, perhaps we ought to replace it with a function.

When and when not to use macros with parameters

It is tempting to forget about the distinction between macros and functions, thinking that it can be ignored. To some extent this is true for absolute beginners, but it is not a good idea to hold on to. It should always be remembered that macros are substituted whole at every place where they are used in a program: this is potentially a very large amount of repetition of code. The advantage of a macro, however, is speed. No time is taken up in passing control over to a new function, because control never leaves the home function when a macro is used: it just makes the function a bit longer. There is a limitation with macros though. Function calls cannot be used as their parameters, such as:

ABS(function())

has no meaning. Only variables or number constants will be substituted. Macros are also severely restricted in complexity by the limitations of the preprocessor. It is simply not viable to copy complicated sequences of code all over programs.

Choosing between functions and macros is a matter of personal judgement. No simple rules can be given. In the end (as with all programming choices) it is experience which counts towards the final ends. Functions are easier to debug than macros, since they allow us to single step through the code(with TRACE utility). Errors in macros are very hard to find, and can be very confusing.

Example Listing

```
/*******************************************************/
/* MACRO DEMONSTRATION */
/*******************************************************/
#include <stdio.h>
#define STRING1 "A macro definition\n"
#define STRING2 "must be all on one line!!\n"
#define EXPRESSION 1 + 2 + 3 + 4
#define EXPR2 EXPRESSION + 10
#define ABS(x) ((x) < 0) ? -(x) : (x)
#define MAX(a,b) (a < b) ? (b) : (a)
#define BIGGEST(a,b,c) (MAX(a,b) < c) ? (c) : (MAX(a,b))
*******************************************************/
main () /* No #definitions inside functions! */
{
printf(STRING1);
printf(STRING2);
printf("%d\n",EXPRESSION);
printf("%d\n",EXPR2);
```

```
printf("%d\n",ABS(-5));
printf("Biggest of 1 2 and 3 is %d",BIGGEST(1,2,3));
}
```

WARNING: Preprocessor macros, although tempting, can produce quite unexpected results if not done correctly. Always keep in mind that macros are textual substitutions done to your source code before anything is compiled. The compiler does not know anything about the macros and never gets to see them. This can produce obscure errors, amongst other negative effects. Prefer to use language features, if there are equivalent (In example use const int or enum instead of #defined constants). That said, there are cases, where macros are very useful (see the debug macro below for an example).

The #define directive is used to define values or macros that are used by the preprocessor to manipulate the program source code before it is compiled. Because preprocessor definitions are substituted before the compiler acts on the source code, any errors that are introduced by #define are difficult to trace. By convention, values defined using #define are named in uppercase. Although doing so is not a requirement, it is considered very bad practice to do otherwise. This allows the values to be easily identified when reading the source code.

Today, #define is primarily used to handle compiler and platform differences. E.g, a define might hold a constant which is the appropriate error code for a system call. The use of #define should thus be limited unless absolutely necessary; typedef statements and constant variables can often perform the same functions more safely.

Another feature of the #define command is that it can take arguments, making it rather useful as a pseudo function creator. Consider the following code:

```
#define ABSOLUTE_VALUE( x ) ( ((x) < 0) ? -(x) : (x) )
...
int x = -1;
while( ABSOLUTE_VALUE( x ) ) {
...
}
```

It's generally a good idea to use extra parentheses when using complex macros. Notice that in the above example, the variable "x" is always within its own set of parentheses. This way, it will be evaluated in whole, before being compared to 0 or multiplied by -1. Also, the entire macro is surrounded by parentheses, to prevent it from being contaminated by other code. If you're not careful, you run the risk of having the compiler misinterpret your code. Because of side-effects it is considered a very bad idea to use macro functions as described above.

```
int x = -10;
int y = ABSOLUTE_VALUE( x++ );
```

If ABSOLUTE_VALUE() was a real function 'x' would now have the value of '-9', but because it was an argument in a macro it was expanded 3 times (in this case) and thus has a value of -7.

Example:

To illustrate the dangers of macros, consider this naive macro

```
#define MAX(a,b) a>b?a:b
```

and the code

```
i = MAX(2,3)+5;
j = MAX(3,2)+5;
```

Take a look at this and consider what the the value after execution might be. The statements are turned into

```
int i = 2>3?2:3+5;
int j = 3>2?3:2+5;
```

Thus, after execution i=8 and j=3 instead of the expected result of i=j=8! This is why you were cautioned to use an extra set of parenthesis above, but even with these, the road is fraught with dangers. The alert reader might quickly realize that if a,b contains expressions, the definition must parenthesize every use of a,b in the macro definition, like this:

```
#define MAX(a,b) ((a)>(b)?(a):(b))
```

This works, provided a,b have no side effects. Indeed,

```
i = 2;
j = 3;
k = MAX(i++, j++);
```

would result in k=4, i=3 and j=5. This would be highly surprising to anyone expecting MAX() to behave like a function. So what is the correct solution? The solution is not to use macro at all. A global, inline function, like this

```
inline max(int a, int b) {
return a>b?a:b
};
```

has none of the pitfalls above, but will not work with all types.

NOTE: Actually the explicit inline declaration is not really necessary, since your compiler can inline functions for you). The compiler is usually better than the programmer at predicting which functions would be worth inlining. Also function calls are not really expensive (they used to be).

The compiler is actually free to ignore the inline keyword. It is only a hint.

(#, ##) Concatenation

It is often useful to merge two tokens into one while expanding macros. This is called *token pasting* or *token concatenation*. The '##' preprocessing operator performs token pasting. When a macro is expanded, the two tokens on either side of each '##' operator are combined into a single token, which then replaces the '##' and the two original tokens in the macro expansion. Usually both will be identifiers, or one will be an identifier and the other a preprocessing number. When pasted, they make a longer identifier. This isn't the only valid case. It is also possible to concatenate two numbers (or a number and a name, such as 1.5 and e3) into a number. Also, multi-character operators such as += can be formed by token pasting.

However, two tokens that don't together form a valid token cannot be pasted together. For example, you cannot concatenate x with + in either order. If you try, the preprocessor issues a warning and emits the two tokens. Whether it puts white space between the tokens is undefined. It is common to find unnecessary uses of '##' in complex macros. If you get this warning, it is likely that you can simply remove the '##'.

Both the tokens combined by '##' could come from the macro body, but you could just as well write them as one token in the first place. Token pasting is most useful when one or both of the tokens comes from a macro argument. If either of the tokens next to an '##' is a parameter name, it is replaced by its actual argument before '##' executes. As with stringification, the actual argument is not macro-expanded first. If the argument is empty, that '##' has no effect.

Keep in mind that the C preprocessor converts comments to whitespace before macros are even considered. Therefore, you cannot create a comment by concatenating '/' and '*'. You can put as much whitespace between '##' and its operands as you like, including comments, and you can put comments in arguments that will be concatenated. However, it is an error if '##' appears at either end of a macro body.

Consider a C program that interprets named commands. There probably needs to be a table of commands, perhaps an array of structures declared as follows:

```
struct command
{
 char *name;
 void (*function) (void);
};
struct command commands[] =
{
 { "quit", quit_command },
 { "help", help_command },
  ...
};
```

It would be cleaner not to have to give each command name twice, once in the string constant and once in the function name. A macro which takes the name of a command as an argument can make this unnecessary. The string constant can be created with stringification, and the function name by concatenating the argument with '_command'. Here is how it is done:

```
#define COMMAND(NAME)  { #NAME, NAME ## _command }
struct command commands[] =
{
 COMMAND (quit),
 COMMAND (help),
 ...
};
```

It is possible to concatenate a macro argument with a constant prefix or suffix to obtain a valid identifier as in

```
#define make_function( name ) int my_ ## name (int foo) {}
make_function( bar )
```

which will define a function called my_bar(). But it isn't possible to integrate a macro argument into a constant string using the concatenation operator. In order to obtain such an effect, one can use the ANSI C property that two or more consecutive string constants are considered equivalent to a single string constant when encountered.

Using this property, one can write

```
#define eat( what ) puts( "I'm eating " #what " today." )
eat( fruit )
```

which the macro-processor will turn into

```
puts("I'm eating " "fruit" " today.")
```

which in turn will be interpreted by the C parser as a single string constant. The following trick can be used to turn a numeric constants into string literals

```
#define num2str(x) str(x)
#define str(x) #x
#define CONST 23
puts(num2str(CONST));
```

This is a bit tricky, since it is expanded in 2 steps. First num2str(CONST) is replaced with str(23), which in turn is replaced with "23". This can be useful in the following example:

```
#ifdef DEBUG
#define debug(msg) fputs(__FILE__ ":" num2str(__LINE__) " - " msg,
    stderr)
#else
#define debug(msg)
#endif
```

This will give you a nice debug message including the file and the line where the message was issued. If DEBUG is not defined however the debugging message will completely vanish from your code. Be careful not to use this sort of construct with anything that has side effects, since this can lead to bugs, that appear and disappear depending on the compilation parameters.

More about Macros :

Macros aren't type-checked and so they do not evaluate arguments. Also, they do not obey scope properly, but simply take the string passed to them and replace each occurrence of the macro argument in the text of the macro with the actual string for that parameter (the code is literally into the location it was called from).

An example on how to use a macro:

```
#include <stdio.h>
#define SLICES 8
#define ADD(x) ( (x) / SLICES )
int main()
{
int a = 0, b = 10, c = 6;
a = ADD(b + c);
printf("%d\n", a);
return 0;
}
```

The result of "a" should be "2" (b + c = 16 -> passed to ADD -> 16 / SLICES -> result is "2")

NOTE: It is usually bad practice to define macros in headers. A macro should be defined only when it is not possible to achieve the same result with a function or some other mechanism. Some compilers

are able to optimize code to where calls to small functions are replaced with inline code, negating any possible speed advantage. Using typedefs, enums, and inline is often a better option.

#pragma

The **pragma** (pragmatic information) directive is part of the standard, but the meaning of any pragma depends on the software implementation of the standard that is used. Pragmas are used within the source program.

```
#pragma token(s)
```

You should check the software implementation of the C standard you intend on using for a list of the supported tokens. For instance one of the most implemented preprocessor directives, #pragma once when placed at the beginning of a header file, indicates that the file where it resides will be skipped if included several times by the preprocessor.

12.4 File inclusion directives

As we have read in Unit 10, the preprocessor directive "#include file name" can be used to include any file in to your program if the functions or macro definitions are present in an external file they can be included in your file

In the directive the filename is the name of the file containing the required definitions or functions alternatively the this directive can take the form

```
#include< filename >
```

Without double quotation marks. In this format the file will be searched in only standard directories.

If using angle brackets like the example above, the preprocessor is instructed to search for the include file along the development environment path for the standard includes.

#include "other.h" If you use quotation marks (" "), the preprocessor is expected to search in some additional, usually user-defined, locations for the header file, and to fall back to the standard include paths only if it is not found in those additional locations. It is common for this form to include searching in the same directory as the file containing the #include directive.

The '#include' directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the '#include' directive. For example, if you have a header file header.h as follows,

```
char *test (void);
```

and a main program called program.c that uses the header file, like this,

```
int x;
#include "header.h"
int
main (void)
{
 puts (test ());
}
```

the compiler will see the same token stream as it would if program.c read

```
int x;
char *test (void);
int
main (void)
{
 puts (test ());
}
```

Included files are not limited to declarations and macro definitions; those are merely the typical uses. Any fragment of a C program can be included from another file. The include file could even contain the beginning of a statement that is concluded in the containing file, or the end of a statement that was started in the including file. However, an included file must consist of complete tokens. Comments and string literals which have not been closed by the end of an included file are invalid. For error recovery, they are considered to end at the end of the file.

To avoid confusion, it is best if header files contain only complete syntactic units—function declarations or definitions, type declarations, etc.

The line following the '#include' directive is always treated as a separate line by the C preprocessor, even if the included file lacks a final newline.

12.5 Enumeration

An *enumeration* is a data type consisting of a set of named values that represent integral constants, known as *enumeration constants*. An enumeration also referred to as an *enumerated type* because you must list (enumerate) each of the values in creating a name for each of them. In addition to providing a way of defining and grouping sets of integral constants, enumerations are useful for variables that have a small number of possible values.

It is used to:

- name a finite set.
- declare elements of that set (enumerators).
- Used as programmer-specified constants.

Eg. enum color {red, blue, green, yellow};/*color is the tag name*\

The enumerated type in C, specified with the enum keyword, and often just called an "enum," is a type designed to represent values across a series of named constants. Each of the enumerated constants has type int. Each enum type itself is compatible with char or a signed or unsigned integer type, but each implementation defines its own rules for choosing a type.

Enumeration type definition

An enumeration type definition contains the enum keyword followed by an optional identifier (the enumeration tag) and a brace-enclosed list of enumerators. A comma separates each enumerator in the enumerator list.

An enumerated type is declared with the enum specifier, an optional name for the enum, a list of one or more constants contained within curly braces and separated by commas, and an optional list of variable names. Subsequent references to a specific enumerated type use the enum keyword and the name of the enum. By default, the first constant in an enumeration is assigned value zero, and each subsequent value is incremented by one over the previous constant. Specific values may also be assigned to constants

in the declaration, and any subsequent constants without specific values will be given incremented values from that point onward.

For example, consider the following declaration:

```
enum colors { RED, GREEN, BLUE = 5, YELLOW } paint_color;
```

Which declares the enum colors type; the int constants RED (whose value is zero), GREEN (whose value is one greater than RED, one), BLUE (whose value is the given value, five), and YELLOW (whose value is one greater than BLUE, six); and the enum colors variable paint_color. The constants may be used outside of the context of the enum, and values other than the constants may be assigned to paint_color, or any other variable of type enum colors.

The *tag_identifier* gives a name to the enumeration type. If you do not provide a tag name, you must put all variable definitions that refer to the enumeration type within the declaration of the type, as described in Enumeration type and variable definitions in a single statement. Similarly, you cannot use a type qualifier with an enumeration definition; type qualifiers placed in front of the enum keyword can only apply to variables that are declared within the type definition.

Enumeration members

The list of enumeration members, or *enumerators*, provides the data type with a set of values.

In C, an *enumeration constant* is of type int. If a constant expression is used as an initializer, the value of the expression cannot exceed the range of int (that is, INT_MIN to INT_MAX as defined in the header limits.h).

In C++, each enumeration constant has a value that can be promoted to a signed or unsigned integer value and a distinct type that does not have to be integral. You can use an enumeration constant anywhere an integer constant is allowed, or anywhere a value of the enumeration type is allowed.

The value of a constant is determined in the following way:

1. An equal sign (=) and a constant expression after the enumeration constant gives an explicit value to the constant. The identifier represents the value of the constant expression.

2. If no explicit value is assigned, the leftmost constant in the list receives the value zero (0).

3. Identifiers with no explicitly assigned values receive the integer value that is one greater than the value represented by the previous identifier.

The following data type declarations list oats, wheat, barley, corn, and rice as enumeration constants. The number under each constant shows the integer value.

```
enum grain { oats, wheat, barley, corn, rice };
   /*  0    1     2      3     4   */

enum grain { oats=1, wheat, barley, corn, rice };
   /*  1    2     3      4     5   */

enum grain { oats, wheat=10, barley, corn=20, rice };
   /*  0    10    11     20    21  */
```

It is possible to associate the same integer with two different enumeration constants. For example, the following definition is valid. The identifiers suspend and hold have the same integer value.

```
enum status { run, clear=5, suspend, resume, hold=6 };
```

/* 0 5 6 7 6 */

Each enumeration constant must be unique within the scope in which the enumeration is defined. In the following example, the second declarations of average and poor cause compiler errors:

```
func()
 {
  enum score { poor, average, good };
  enum rating { below, average, above };
  int poor;
 }
```

You can declare an enumeration type separately from the definition of variables of that type, as described in Enumeration type definition and Enumeration variable declarations; or you can define an enumeration data type and all variables that have that type in one statement, as described in Enumeration type and variable definitions in a single statement.

Some compilers warn if an object with enumerated type is assigned a value that is not one of its constants. However, such an object can be assigned any values in the range of their compatible type, and enum constants can be used anywhere an integer is expected. For this reason, enum values are often used in place of the preprocessor #define directives to create a series of named constants.

Enumeration type and variable definitions in a single statement

You can define a type and a variable in one statement by using a declarator and an optional initializer after the type definition. To specify a storage class specifier for the variable, you must put the storage class specifier at the beginning of the declaration. For example:

```
register enum score { poor=1, average, good } rating = good;
```

Either of these examples is equivalent to the following two declarations:

```
enum score { poor=1, average, good };
register enum score rating = good;
```

Both examples define the enumeration data type score and the variable rating. rating has the storage class specifier register, the data type enum score, and the initial value good.

Combining a data type definition with the definitions of all variables having that data type lets you leave the data type unnamed. For example:

```
enum { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
          Saturday } weekday;
```

defines the variable weekday, which can be assigned any of the specified enumeration constants. However, you can not declare any additional enumeration variables using this set of enumeration constants.

Storage duration specifiers

Every object has a storage class, which may be automatic, static, or allocated. Variables declared within a block by default have automatic storage, as do those explicitly declared with the auto or register storage class specifiers. The auto and register specifiers may only be used within functions and function argument declarations; as such, the auto specifier is always redundant. Objects declared outside of all blocks and those explicitly declared with the static storage class specifier have static storage duration.

Objects with automatic storage are local to the block in which they were declared and are discarded when the block is exited. Additionally, objects declared with the register storage class may be given higher priority by the compiler for access to registers; although they may not actually be stored in registers, objects with this storage class may not be used with the address-of (&) unary operator. Objects with static storage persist upon exit from the block in which they were declared. In this way, the same object can be accessed by a function across multiple calls. Objects with allocated storage duration are created and destroyed explicitly with malloc, free, and related functions.

The extern storage class specifier indicates that the storage for an object has been defined elsewhere. When used inside a block, it indicates that the storage has been defined by a declaration outside of that block. When used outside of all blocks, it indicates that the storage has been defined outside of the file. The extern storage class specifier is redundant when used on a function declaration. It indicates that the declared function has been defined outside of the file.

12.6 Storage classes

From C compiler's point of view, a variable name identifies some physical location within the computer where the string of bits representing the variable's value is stored. There are basically two kinds of locations in a computer where such a value may be kept— Memory and CPU registers. It is the variable's storage class that determines in which of these two locations the value is stored.

Moreover, a variable's storage class tells us:

(a) Where the variable would be stored?

(b) What will be the initial value of the variable, if initial value is not specifically assigned.(i.e. the default initial value)?

(c) What is the scope of the variable; i.e. in which functions the value of the variable would be available?

(d) What is the life of the variable; i.e. how long would the variable exist?

There are four storage classes in C:

(a) Automatic storage class

(b) Register storage class

(c) Static storage class

(d) External storage class

The general form for declaring a storage class is:

```
storage_class declarator;
```

For example:

```
extern int value;
autolong p=5;
auto int q;
static intx;
```

Let us examine these storage classes one by one.

Automatic Storage Class

The features of a variable defined to have an automatic storage class are as under:

Storage Memory.

Default initial value	An unpredictable value, which is often called a garbage value.
Scope	Local to the block in which the variable is defined.
Life	Till the control remains within the block in which the variable is defined.

Following program shows how an automatic storage class variable is declared, and the fact that if the variable is not initialized it contains a garbage value.

```
main( )
{
auto int i, j ;
printf ( "\n%d %d", i, j ) ;
}
```

The output of the above program could be...

```
1211 221
```

where, 1211 and 221 are garbage values of **i** and **j**. When you run this program you may get different values, since garbage values are unpredictable. So always make it a point that you initialize the automatic variables properly, otherwise you are likely to get unexpected results. Note that the keyword for this storage class is **auto**, and not automatic.

Scope and life of an automatic variable is illustrated in the following program.

```
main( )
{
auto int i = 1 ;
{
   {
     {
         printf ( "\n%d ", i ) ;
     }
    printf ( "%d ", i ) ;
   }
  printf ( "%d", i ) ;
}
}
```

The output of the above program is:

```
1 1 1
```

This is because, all **printf( )** statements occur within the outermost block (a block is all statements enclosed within a pair of braces) in which **i** has been defined. It means the scope of **i** is local to the block in which it is defined. The moment the control comes out of the block in which the variable is defined, the variable and its value is irretrievably lost. To catch my point, go through the following program.

```
main( )
```

```
auto int i = 1 ;
{
auto int i = 2 ;
{
auto int i = 3 ;
printf ( "\n%d ", i ) ;
}
printf ( "%d ", i ) ;
}
printf ( "%d", i ) ;
}
```

The output of the above program would be:

```
3 2 1
```

Note that the Compiler treats the three **i**'s as totally different variables, since they are defined in different blocks. Once the control comes out of the innermost block the variable **i** with value 3 is lost, and hence the **i** in the second **printf( )** refers to **i** with value 2. Similarly, when the control comes out of the next innermost block, the third **printf( )** refers to the **i** with value 1.

Understand the concept of life and scope of an automatic storage class variable thoroughly before proceeding with the next storage class.

Register Storage Class

The features of a variable defined to be of **register** storage class are as under:

Storage	CPU registers
Default initial value	Garbage value.
Scope	Local to the block in which the variable is defined.
Life	Till the control remains within the block in which the variable is defined.

A value stored in a CPU register can always be accessed faster than the one that is stored in memory. Therefore, if a variable is used at many places in a program it is better to declare its storage class as **register**. A good example of frequently used variables is loop counters. We can name their storage class as **register**.

```
main( )
{
register iant i ;
for ( i = 1 ; i <= 10 ; i++ )
printf ( "\n%d", i ) ;
}
```

Here, even though we have declared the storage class of **i** as **register**, we cannot say for sure that the value of **i** would be stored in a CPU register. Why? Because the number of CPU registers are limited;

and they may be busy doing some other task. What happens in such an event... the variable works as if its storage class is **auto**.

Not every type of variable can be stored in a CPU register.

For example, if the microprocessor has 16-bit registers then they cannot hold a **float** value or a **double** value, which require 4 and 8 bytes respectively. However, if you use the **register** storage class for a **float** or a **double** variable you won't get any error messages. All that would happen is the compiler would treat the variables to be of **auto** storage class.

Static Storage Class

The features of a variable defined to have a **static** storage class are as under:

Storage	Memory.
Default initial value	Zero
Scope	Local to the block in which the variable is defined
Life	Value of the variable persists between different function calls.

Compare the two programs and their output given in Figure 6.3 to understand the difference between the **automatic** and **static** storage classes.

Figure 6.3

The programs above consist of two functions **main( )** and **increment( )**. The function **increment ( )** gets called from **main( )** thrice. Each time it increments the value of **i** and prints it. The only difference in the two programs is that one uses an **auto** storage class for variable **i**, whereas the other uses **static** storage class.

```
main( )                              main( )
{                                    {
increment( ) ;                       increment( ) ;
increment( ) ;                       increment( ) ;
increment( ) ;                       increment( ) ;
}                                    }
increment( ) { static int i = 1 ;    increment( )
printf ( "%d\n", i ) ;               {
i = i + 1 ;                          auto int i = 1 ;
}                                    printf ( "%d\n", i ) ;
                                     i = i + 1 ;
                                     }
```

The output of the above programs would be:

```
1                                    1
2                                    1
3                                    1
```

Like **auto** variables, **static** variables are also local to the block in which they are declared. The difference between them is that **static** variables don't disappear when the function is no longer active. Their values persist. If the control comes back to the same function again the **static** variables have the same values they had last time around.

In the above example, when variable **i** is **auto**, each time **increment( )** is called it is re-initialized to one. When the function terminates, **i** vanishes and its new value of 2 is lost. The result: no matter how many times we call **increment( ), i** is initialized to 1 every time.

On the other hand, if **i** is **static**, it is initialized to 1 only once. It is never initialized again. During the first call to **increment( ), i** is incremented to 2. Because **i** is static, this value persists. The next time **increment( )** is called, **i** is not re-initialized to 1; on the contrary its old value 2 is still available. This current value of **i** (i.e. 2) gets printed and then **i = i + 1** adds 1 to **i** to get a value of 3. When **increment( )** is called the third time, the current value of **i** (i.e. 3) gets printed and once again **i** is incremented. In short, if the storage class is **static** then the statement **static int i = 1** is executed only once, irrespective of how many times the same function is called.

Consider one more program.

```
main( )
{
int *j ;
int * fun( ) ;
j = fun( ) ;
printf ( "\n%d", *j ) ;
}
int *fun( )
{ int k = 35 ;
return ( &k ) ;
}
```

Here we are returning an address of **k** from **fun( )** and collecting it in **j**. Thus **j** becomes pointer to **k**. Then using this pointer we are printing the value of **k**. This correctly prints out 35. Now try calling any function (even **printf( )**) immediately after the call to **fun( )**. This time **printf( )** prints a garbage value. Why does this happen? In the first case, when the control returned from **fun( )** though **k** went dead it was still left on the stack. We then accessed this value using its address that was collected in **j**. But when we precede the call to **printf( )** by a call to any other function, the stack is now changed, hence we get the garbage value. If we want to get the correct value each time then we must declare **k as static**. By doing this when the control returns from **fun( ), k** would not die.

All this having been said, a word of advice—avoid using **static** variables unless you really need them. Because their values are kept in memory when the variables are not active, which means they take up space in memory that could otherwise be used by other variables.

External Storage Class

The features of a variable whose storage class has been defined as external are as follows:

Storage	Memory
Default initial value	Zero
Scope	Global.
Life	As long as the program's execution doesn't come to an end.

External variables differ from those we have already discussed in that their scope is global, not local. External variables are declared outside all functions, yet are available to all functions that care to use them. Here is an example to illustrate this fact.

```
int i ;
main()
{
printf ( "\ni = %d", i ) ;
increment() ;
increment() ;
decrement() ;
decrement() ;
}
increment()
{
i = i + 1 ;
printf ( "\non incrementing i = %d", i ) ;
}
decrement()
{
i = i - 1 ;
printf ( "\non decrementing i = %d", i ) ;
}
```

The output would be:

```
    i = 0
on incrementing i = 1
on incrementing i = 2
on decrementing i = 1
on decrementing i = 0
```

As is obvious from the above output, the value of **i** is available to the functions **increment( )** and **decrement( )** since **i** has been declared outside all functions.

Look at the following program.

```
int x = 21 ;
main()
{
extern int y ;
printf ( "\n%d %d", x, y ) ;
}
int  y = 31 ;
```

Here, **x** and **y** both are global variables. Since both of them have been defined outside all the functions both enjoy external storage class. Note the difference between the following:

```
extern int y ;
int y = 31 ;
```

Here the first statement is a declaration, whereas the second is the definition. When we declare a variable no space is reserved for it, whereas, when we define it space gets reserved for it in memory. We had to declare **y** since it is being used in **printf( )** before it's definition is encountered. There was no need to declare **x** since its definition is done before its usage. Also remember that a variable can be declared several times but can be defined only once.

Another small issue—what will be the output of the following program?

```
int x = 10 ;
main( )
{
int x = 20 ;
printf ( "\n%d", x );
display( ) ;
}
display( )
{
printf ( "\n%d", x ) ;
}
```

Here **x** is defined at two places, once outside **main( )** and once inside it. When the control reaches the **printf( )** in **main( )** which **x** gets printed? Whenever such a conflict arises, it's the local variable that gets preference over the global variable. Hence the **printf( )** outputs 20. When **display( )** is called and control reaches the **printf( )** there is no such conflict. Hence this time the value of the global **x**, i.e. 10 gets printed.

One last thing—a **static** variable can also be declared outside **all** the functions. For all practical purposes it will be treated as an **extern** variable. However, the scope of this variable is limited to the same file in which it is declared. This means that the variable would not be available to any function that is defined in a file other than the file in which the variable is defined.

Which to Use When

Dennis Ritchie has made available to the C programmer a number of storage classes with varying features, believing that the programmer is in a best position to decide which one of these storage classes is to be used when. We can make a few ground rules for usage of different storage classes in different programming situations with a view to:

(a) economise the memory space consumed by the variables.

(b) improve the speed of execution of the program.

The rules are as under:

- Use **static** storage class only if you want the value of a variable to persist between different function calls.
- Use **register** storage class for only those variables that are being used very often in a program. Reason is, there are very few CPU registers at our disposal and many of them might be busy doing something else. Make careful utilization of the scarce resources. A typical application of **register** storage class is loop counters, which get used a number of times in a program.

- Use **extern** storage class for only those variables that are being used by almost all the functions in the program. This would avoid unnecessary passing of these variables as arguments when making a function call. Declaring all the variables as **extern** would amount to a lot of wastage of memory space because these variables would remain active throughout the life of the program.

- If you don't have any of the express needs mentioned above, then use the **auto** storage class. In fact most of the times we end up using the **auto** variables, because often it so happens that once we have used the variables in a function we don't mind loosing them.

Let try some experiment. First of all, let create a simple class. Create a header file namedobject.h, save this file, do not run or compile this program.

12.7 Command line arguments

Command Line Parameters: Arguments to main()

Accessing the command line arguments is a very useful facility. It enables you to provide commands with arguments that the command can use e.g. the command

```
% cat prog.c
```

takes the argument "prog.c" and opens a file with that name, which it then displays. A technique to write a function which takes variable number of arguments, is called command line arguments. The command line arguments include the command name itself so that in the above example, "cat" and "prog.c" are the command line arguments. The first argument i.e. "cat" is argument number zero, the next argument, "prog.c", is argument number one and so on.

To access these arguments from within a C program, you pass parameters to the function main(). The use of arguments to main is a key feature of many C programs.

The declaration of main looks like this:

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

This declaration states that

1. main returns an integer value (used to determine if the program terminates successfully).
2. argc is the number of command line arguments including the command itself i.e argc must be at least 1.
3. argv is an array of the command line arguments.

Two special identifiers, argc and argv are used to pass to main( ) the number of command line arguments and pointers to each argument we have to set up main( ) as follows.

main(int argc, char*argv[ ])

argc will then provide the number of command line arguments including the command itself so argC its never less than 1.

The argv is an array of pointer to char or equivalently an array of strings. Each of argv[0], argv[1],... up to argv[argc-1] is a pointer to command line argument, namely a NULL terminated string. The pointer argv[argc] is set to NULL to mark the end of the array.

Suppose these is a program Vkcpy in which main contains argument argc, and argv means skeleton of program is as follows

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

```
{______
  ______
  ______
  ______
  ______
}
```

Suppose we now execute the program by typing VK COPY Hellow.C Hello.CBK Here argc = 3(command plus 2 arguments)

argv[0] points to "C:\VKCPY\0"

argv[1] points to "HELLO.C\0"

argv[2] points to "HELLO.CBK\0"

argv[3] is NULL

Example Program to demonstrate command line arguments : 'print_args' echoes its arguments to the standard output – is a form of the Unix echo command.

```
/* print_args.c: Echo command line arguments */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    int i = 0 ;
    int num_args ;
    num_args = argc ;
    while( num_args > 0)
    {
        printf("%s\n", argv[i]);
        i++ ;
        num_args--;
    }
}
```

If the name of this program is print_args, an example of its execution is as follows:

```
% print_args hello goodbye solong
print_args
hello
goodbye
```

solong

%

Exercise: Rewrite print_args so that it operates like the Unix echo command. **Hint**: You only need to change the printf statement.

12.8 Multifile programs using own header files

Writing programs in multiple file

Writing *header files* is just part of writing C code in multiple files. While you will not *need* to write C programs in multiple files (indeed most of the programs that you write during this course will be easily small enough that a single file is the best way to go) it is a good technique to know – your project will probably be large enough to be spread across multiple files. Individual .c files are often called *modules*. Why write C programs where the source code is spread across more than one *module*? There are a number of reasons:

1) Code which is related can be kept together making routines easier to find.
2) Two programmers can work on the same program by editing different files.
3) On a very large program we can save time by only recompiling part of the program.
4) If, for some reason, we need to rewrite part of a program, the bits being rewritten may be in one file.

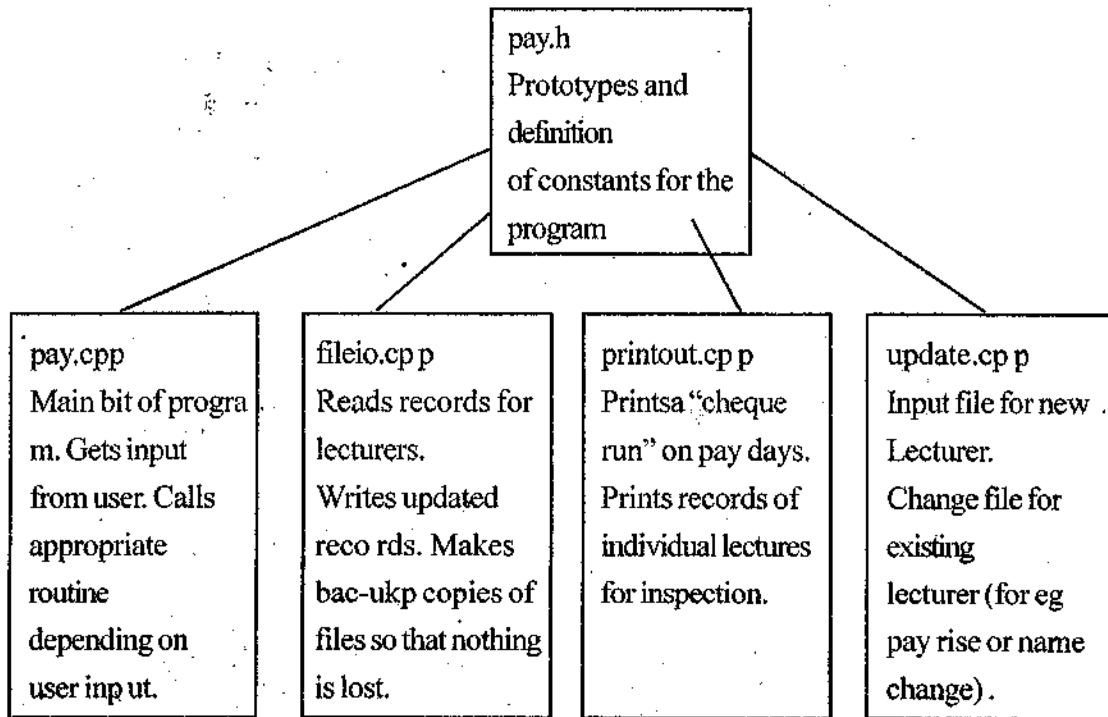
The *how to* of writing programs in multiple *modules* is fairly straightforward. Remember that you should never have more than one main (indeed you should never have two functions with the same name). If you use a struct or a typedef a #define or an enum in a *module* then that module should include the header file containing the appropriate bit of code for the struct, typedef, #define or enum. If you are following the rules we set up above, then you're including all these things in header files anyway.

IMPORTANT RULE: Any *module* that calls a function should have access to the *prototype* for that function. (That is, in fact, what you've been doing all the time you've been #include ing headers – making sure that the *prototype* for printf, or whatever, was available). The best way to do that is to put the prototype into a header file and #include the header in any *module* which wishes to use the function.

The problem with talking about this topic is that any program which is large enough to require multiple files is probably too large to consider in detail. Instead let's consider an abstract example of how we might split up functions between files. Our imaginary program might be a program to deal with the pay packets of university lecturers. The program would need (amongst other things) the following features:

1) Input info for a new lecturer.
2) Delete a lecturer who leaves.
3) Promote a lecturer who gets an individual pay-rise.
4) Read and write this information to disk.
5) Inspect an individual lecturer's record.
6) Print off a months wage packets for all lecturers.

We might therefore decide to split the functions up between files like so:



Note that all the .cpp files would include the file pay.h. This is quite normal, small C programs in only two or three files will typically only have one or two header files. Larger programs in more files may need more header files. One common question is "how many functions should be in each module?" Really, it is up to the programmer. Too many is a pain because each module is long, it's hard to remember where in the file you put the code and each module takes a long time to compile. It also may be harder for another programmer looking at your code to know which functions are the most important if they're all together. Too few functions in a module is a pain because you have a large number of *modules* to maintain. Generally, somewhere between 1 and 100 functions is the right number but it is up to the preference of the individual programmer.

The extern statement

The extern statement is used in multiple file programming. It is used to say "the global variable used in this file is initialised in another file". Put extern in front of a global variable when it is declared in another file. In the following example we declare three variables (two arrays and one char) in testprog1.c and use them in testprog2.c

```
testprog.h
/* function prototypes */
void testfunc(void);
```

```
testprog1.cpp
#include "testprog.h"
/* Define some global
variables */
float farray[1000];
int iarray[500];
char c;
int main()
{
c= '\n';
farray[0]= 1.2;
iarray[0]= 3;
testfunc();
/* Rest of code*/
return 0;
}
```

```
testprog2.cpp
#include "testprog.h"
/* Globals from testprog1.c must
be defined as extern here*/
extern float farray[1000];
extern int iarray[500];
extern char c;
void testfunc (void)
{
printf ("farray[0] is %f\n",
farray[0]);
printf ("c is %c\n",c);
printf ("iarray[0] is %d\n",
iarray[0]);
/* Rest of code*/
}
```

Every file (sometimes called a module) in the program which uses the global variables, except for the one where they are declared, must have them declared as extern to say to the compiler "don't worry about this variable, you will find out about it from another file." If we had forgotten to declare the variables as extern in testprog2.cpp this would be equivalent to having two global variables with the same name in the program (confusing and probably will stop your program working). If we had not declared them at all in testprog2.cpp then the compiler would flag an error because it wouldn't know what farray, iarray and c were.

The extern statement is used to tell the compiler that a global variable will be declared in another file in the program.

12.9 Summary

The # and ## operators are used with the #define macro. Using # causes the first argument after the # to be returned as a string in quotes. For example, the command

```
#define as_string( s ) # s
```

will make the compiler turn this command

```
puts( as_string( Hello World! ) ) ;
```

into

```
puts( "Hello World!" );
```

Using ## concatenates what's before the ## with what's after it. For example, the command

```
#define concatenate( x, y ) x ## y
...
Int xy = 10;
...
```

will make the compiler turn

```
printf( "%d", concatenate( x, y ));
```

into

```
printf( "%d", xy);
```

which will, of course, display 10 to standard output.

Storage Classes

- Storage class specifiers tell compiler the duration and visibility of the variables or objects declared, as well as, where the variables or objects should be stored.
- In C++ program we have multiple files. In these files we may have normal variables, array, functions, structures, unions, classes etc. So, variables and objects declared must have the visibility, the lifetime and the storage, when values assigned.
- In C / C++ there are 4 different storage classes available:automatic,external,static and register.

Storage class	Keyword
Automatic	auto
External	extern
Static	static
Register	register

Automatic Variable - auto

- Local variables are variables declared within a function or blocks (after the opening brace, { of the block). Local variables are automatic by default. This means that they come toexistence when the function in which it is declared is invoked anddisa ppears when the function ends.
- Automatic variables are declared by using the keywordauto. But since the variables declared in functions are automatic by default, this keyword may be dropped in the declaration as you found in many source codes.
- The same variable names may be declared and used in different functions, but they are only known in the functions in which they are declared. This means, there is no confusion even if the same variables names are declared and used in different functions.

External Variable - extern

- External variables are variables that are recognized globally, rather than locally. In other words, once declared, the variable can be used in any line of codes throughout the rest of the program.
- A variable defined outside a function is external. An external variable can also be declared within the function that uses it by using the keywordextern hence it can be accessed by other code in other files.

Static Variable - static

- In a single file program, static variables are defined within individual functions that they are local to the function in which they are defined. Static variables are local variables that retain their values

throughout the lifetime of the program. In other words, their same (or the latest) values are stillavailable when the function is re-invoked later.

- Their values can be utilized within the function in the same manner as other variables, but they cannot be accessed from outside of their defined function.
- Thestatic has internal linkage (that is not visible from outside) except for the static members of a class that have external linkage. The example using the static variable has been presented in Module 13 and some other part of the program examples in this Tutorial.

Register Variable - register

- The above three classes of variables are normally stored in computer memory. Register variables however are stored in the processor registers, where they can be accessed and manipulated faster. Register variables, like automatic variables, are local to the function in which they are declared.
- Defining certain variables to be register variables does not, however, guarantee that they will actually be treated as register variables.
- Registers will be assigned to these variables by compiler so long as they are available. If a register declaration cannot be fulfilled, the variables will be treated as automatic variables. So, it is not a mandatory for the compiler to fulfill the register variables.
- Usually, only register variables are assigned the register storage class. If all things equal, a program that makes use of register variables is likely to run faster than an identical program that uses just automatic variables.

Global Variables are defined outside of all function bodies ({...}) and are available to all parts of the program. The lifetime or availability of a global variable last until the program ends. As explained before, if extern keyword is used when declaring the global variable, the data is available to this file by telling it the data is exist somewhere in another files.

Local Variables are local to a function, including the main() function. They are automatic variables, exist when the scope is entered and disappear when the scope closes.

12.10 Glossary

Directive A directive is an instruction by the programmer for a compiler. Most programming languages have directives. In C, **#include** is used to include header files. You can use the #define **directive** to give a meaningful name to a constant in your program

Enumeration Short for enumeration, an *enum* variable type can be found in C (Ansi, not the original K&R), C++ and C#. The idea is that instead of using an *int* to represent a set of values, a *type* with a restricted set of values in used instead.

Global These variables can be accessed (ie known) by any function comprising the program. They are implemented by associating memory locations with variable names. They do not get recreated if the function is recalled.

Local These variables only exist inside the specific function that creates them. They are unknown to other functions and to the main program. As such, they are normally implemented using a stack. Local variables cease to exist once the function that created them is completed. They are recreated each time a function is executed or called.

Macro In C and C++, a Macro is a piece of text that is expanded by the preprocessor part of the compiler. This is used in to expand text before compiling.

Preprocessor The preprocessor is the part of the compiler in C and C++ that reads the source code files and expands text wherever it finds a # in column one.

12.11 Further Readings

- Programming in 'C' – Balaguruswamy, Tata McGraw-Hill Publications
- The C programming language Dennis Ritchie, Prentice Hall, Delhi
- Let us C – Yashwant Kanetkar, BPB Publication

12.12 Unit end questions

- Define a macro called "birthday" which describes the day of the month upon which your birthday falls.
- Write a command to the preprocessor to include to maths library math.h.
- A macro is always a number. True or false?
- A macro is always a constant. True or false?