DEVELOP COMPUTER PROGRAM NOTES

Definitions

Program
This is a complete set of step-by-step instructions that control and direct the computer hardware in carrying out a given task.   Tasks may vary from very simple e.g. computing surface area to complex ones like statistical analysis.

Programs are usually written to solve user problems on a computer.

Software
This term refers to all programs together with their associated

Programming Language
This is a set of symbols and the rules that govern their rules that are employed in the construction of a computer program.

Programmer
This is a person who is trained and/or specializes in the technique of creating, maintaining and modifying computer programs.

Programming Languages

Programming languages provide the basic building block for all software. They are the means by which people can tell the computer how to carry out a task.

A program can be written in a variety of programming languages. The languages can broadly be classified into two categories:

- Low-level language – which refers to the machine language and assembly language.
- High-Level languages: - which refers to languages such as COBOL, FORTRAN, BASIC

Low Level Languages

Machine Language (First Generation Language)
Digital computers represent and process data and instructions as binary numbers. This representation of instructions and data is called machine language. Program instructions were written as a series of binary digits (0's and 1's). When the program was entered into the computer execution was direct since machine language needs no translation. The binary combination allowed the program to have full access to and control the computers internal circuitry and memory addresses.

Advantages
- Program translation was fast because no translation was required.
- The program could directly address and control the internal circuitry meaning that these programs were more effective in hardware usage and control.

Disadvantages

- Writing programs was time consuming
- Tracing errors in a program was extremely difficult.
- Difficult to learn and use.
- Program modification was cumbersome.
- They were machine dependent i.e. a program created for one type of machine would not work on a different type of machine.
- To write an effective program the programmer had to have expert knowledge on the computer's internal workings.

Assembly Language (Second Generation)

This language was more user oriented than machine language. Instructions are represented using mnemonic code and symbolic addresses. Words like add, sum etc could be used in programs. An assembler translated these codes into machine language.

Advantages

- Much easier to learn compared to machine language.
- Coding took less time than coding using machine language.
- Error correction was less cumbersome.

Disadvantages

- Were also machine specific
- Execution took longer than machine language programs due to the translation process.

High Level Languages

These languages are user friendly and problem oriented compared to the low level languages. Programs written in high level languages are shorter than the machine language programs.

They have an extensive vocabulary of words and symbols therefore program instructions are written using familiar English-like statements and mathematical statements.

A single high-level language program is translated into multiple machine code instructions.

Advantages

- They are portable i.e. they can be used on more than one type of machine.
- Creating program and error correction takes less time.
- Are relatively easy to learn and use.
- The programmer does not have to be an expert on the internal workings of the machine.

Disadvantages

- Program execution takes more time due to the translation process.
- They do not address the internal circuitry of computers as effectively as the low level languages.
- A translated program will occupy more space.

High level languages can further be classified into :

- Procedural languages (Third Generation)
- Non-Procedural Languages (Fourth Generation Languages or 4GLs)

Procedural Languages

They require the programmer to specify step-by-step how the computer will accomplish a specific task. Program execution follows the exact sequence laid down by the programmer during coding. Examples include FORTRAN, PASCAL, BASIC,

Non-Procedural Languages

They allow the programmer to specify the desired result without having to specify the detailed procedure needed to achieve the result.

They are more user oriented and allow programmers to develop programs with fewer commands compared with 3rd generation languages. They are called non procedural because programmers can write programs that need only tell the computer what they want done, not all the procedures of doing it.

4GL consists of:

- Report Generators: also called report writers. This is a program for end users that is used to produce reports.
- Query Language: This is an easy to use language for retrieving data from a database management system.
- Application Generators: This is a program's tool that allows a person to give a detailed explanation of what data to be processed. The software then generates codes needed to create a program to perform the tasks.

Object Oriented Programming Languages

OOP is a design philosophy. It stands for Object Oriented Programming. **O**bject-**O**riented **P**rogramming (*OOP*) uses a different set of programming languages than old procedural programming languages (*C, Pascal*, etc.). Everything in *OOP* is grouped as self sustainable "*objects*". Hence, you gain re-usability by means of four main object-oriented programming concepts.

An object can be considered a "*thing*" that can perform a set of **related** activities. The set of activities that the object performs defines the object's behavior. For example, the hand can grip something or a *Student* (*object*) can give the name or address.

Definitions:

Syntax
These are the rules of a language that govern the ways in which words, symbols, expressions and statements may be formed and combined in that language.

Semantics
These are the rules of language that govern meaning of statements in any language.

Language Translators

These are programs that translate programs written in a language other than machine language into machine language programs for execution. Programs written in assembly or high level languages are called source programs or source code before they undergo translation. After the translation the machine language version of the same program is called object program or object code. Language translators can be classified into

- Assemblers
- Compilers
- Interpreters

Assembler

This is a program that translates a source program written in assembly language into its equivalent machine code (object code).

Compiler

During compilation both the high level source program and the compiler are loaded into the RAM. The compiler reads through the source program statement by statement, converting each correct statement into multiple machine code instructions. The process continues until the compiler has read through the entire source program or it encounters an error. An erroneous statement is not translated but is placed on the source program error listing, which will be displayed to the programmer at the end of the translation process. Where there are no errors the compiler will generate a machine code object program which is stored for subsequent execution.

Compilation can be divided into three stages including

Lexical Analysis which involves

- Checking for valid words like data names or operator symbols
- Checking for reserved words (keywords). These are words that have a special meaning for the compiler. These are then replaced with non-alphanumeric character codes known as tokens.

Syntax analysis which involves

- Checking the program statements for correct grammatical form.
- Breaking down the complex statements in the program into simpler equivalents and more manageable forms.

The first two stages of compilation are carried out by a part of the compiler known as the parser and can be referred to as parsing.

Code Generation involves

- Translation of each statement into its equivalent machine code. This is done using tables.
- Setting up linkages that allow the object program to communicate or work with various operating systems and hardware.

- Fetching of subroutines from the system library, to optimize the code and make it execute faster and use less storage space.

The optimized code is then used to generate the object program which is stored on media such as disk to await subsequent execution.

Interpreter

It is similar to the compiler in that it translates high level language programs into machine code for execution but no object code is generated. An interpreter translates the program one statement at a time and if it is error free it executes the statement. This continues till the last statement in the program has translated and executed. Thus an interpreter translates and executes a statement before it goes to the next statement. When it encounters an error it will immediately stop the execution of the program.

Program Development

Steps in Program Development

1. Design program objectives
It involves forming a clear idea in terms of the information you want to include in the program, computations needed and the output. At this stage, the programmer should think in general terms, not in terms of some specific computer language.

2. Design program
The programmer decides how the program will go about its implementation, what should the user interface be like, how should the program be organized, how to represent the data and what methods to use during processing. At this point, you should also be thinking generally although some of your decisions may be based on some general characteristics of the C language.

3. Write the Code
It is the design Implementation by writing the (C) code i.e. translating your program design into C language instructions. You will use C's text editor or any other editor such as notepad. Using another editor such as notepad requires you to save your program with the extension **.c**.

4. Compile the code
A compiler converts the source code into object code. Object code are instructions in machine language. Computers have different machine languages. C compilers also incorporate code for C libraries into the final program. The libraries contain standard routines. The end result is an executable file that the computer can understand.
The compiler also checks errors in your program and reports the error to you for correction. The object code can never be produced if the source code contains syntax errors.

5. Run the program
This is the actual execution of the final code, usually preceded by linking [#]. Once the executable code is complete and working, it can be invoked in future by typing its name in a 'run' command.

6. Test the program
This involves checking whether the system does what it is supposed to do. Programs may have bugs (errors). Debugging involves the finding and fixing of program mistakes.

7. Maintain and modify the program
Occasionally, changes become necessary to make to a given program. You may think of a better way to do something in a program, a clever feature or you may want to adapt the program to run in a different machine. These tasks are simplified greatly if you document the program clearly and follow good program design practices.

Software Life cycle model

A software life cycle model (also called process model) is a descriptive and diagrammatic representation of the software life cycle. A life cycle model represents all the activities required to make a software product. It also captures the order in which these activities are to be undertaken.
A life cycle model maps the different activities performed on a software product from its inception to retirement. Different lifecycle models may map the basic development activities to phases in different ways. The basic activities included in all life cycle models are similar though the activities may be carried out in different orders in different life cycle models. During any life cycle phase, more than one activity may also be carried out. For example, the design phase might consist of the structured analysis activity followed by the structured design activity.

Importance of a software life cycle model

The development team must identify a suitable life cycle model for the particular project and then follow it. Without using of a particular life cycle model the development of a software product would not be done in a systematic and disciplined manner. When a software product is being developed by a team there must be a clear understanding among team members about when and what to do. Otherwise it would lead to chaos and project failure. For example: A software development problem is divided into several parts and the parts are assigned to the team members. From then on, the team members are allowed the freedom to develop the parts assigned to them in whatever way they like. It is possible that one member might start writing the code for his part, another might decide to prepare the test documents first, and some other engineer might begin with the design phase of the parts assigned to him. This lack of consistency would lead to project failure.

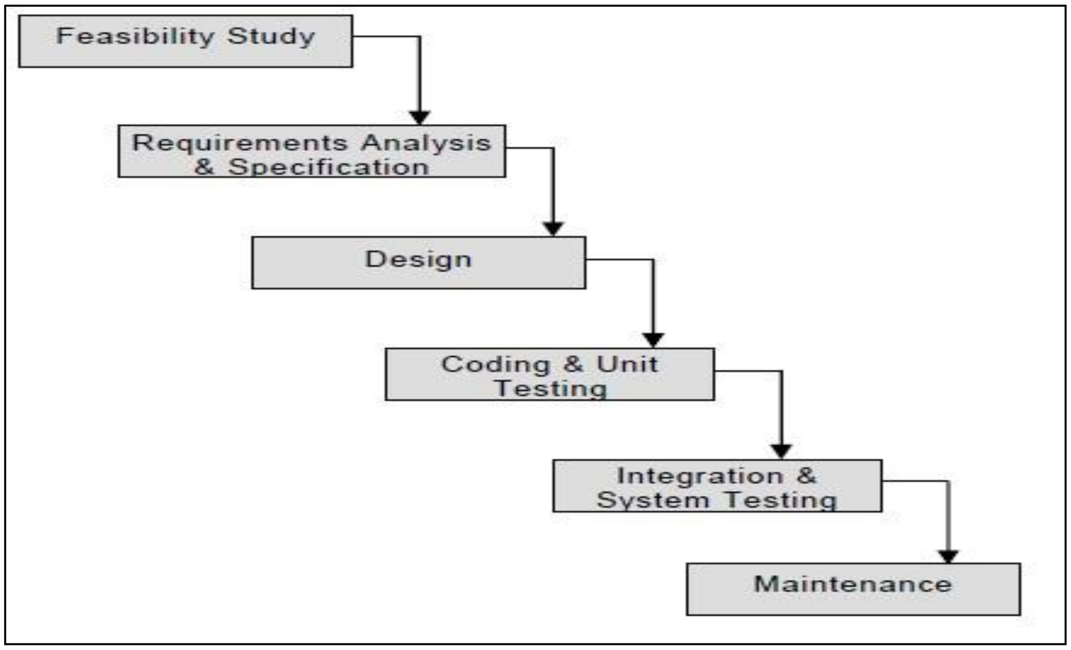
The classical waterfall model

[#] Combining all object code segments from different modules and libraries functions.

The waterfall model is a sequential design process, often used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of Feasibility Study, Requirements Analysis and Specification, Design, Coding and Unit Testing, Integration and System Testing and Maintenance.
The classical waterfall model is the most obvious way to develop software. This model can be considered to be a theoretical way of developing software, but all other life cycle models are essentially derived from the classical waterfall model.
In order to be able to appreciate other life cycle models it is necessary to learn the classical waterfall model.
Classical waterfall model divides the life cycle into the following phases:

- Feasibility Study
- Requirements Analysis and Specification
- Design
- Coding and Unit Testing
- Integration and System Testing
- Maintenance

Phases of water fall model


Feasibility Study
Requirements Analysis & Specification
Design
Coding & Unit Testing
Integration & System Testing
Maintenance

Feasibility study

The main aim of feasibility study is to determine whether it would be financially and technically feasible to develop the product. The project managers or team leaders try to understand what is required to be done by visiting the client side. They study different input data to the system and output data to be produced by the system. They study what kind of processing is needed to be done on these data and they look at the various constraints on the behavior of the system. When they have an overall understanding of the problem they investigate the different solutions that are possible.

Then they examine each of the solutions in terms of what kind of resources required, what would be the cost of development and what would be the development time for each solution.
Based on this analysis they select the best solution and determine whether the customer budget would meet the cost of the product and whether they have sufficient technical expertise in the area of development.

Requirement Analysis and Specification

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

- Requirements gathering and analysis
- Requirements specification

The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed. This is done so as to clearly understand the customer requirements so that incompleteness and inconsistencies are removed.
The requirements analysis activity is begins with collecting all relevant data regarding the product to be developed from the users of the product and from the customer .For example, to perform the requirements analysis of a business accounting software required by an organization, the analyst might interview all the accountants of the organization to ascertain their requirements. The data collected from such a group of users usually contains several contradictions and ambiguities, since each user has only a partial and incomplete view of the system. It is therefore necessary to identify all ambiguities and contradictions in the requirements and resolve them through further discussions with the customer. After all ambiguities, inconsistencies, and incompleteness have been resolved and the requirements properly understood, the requirements specification activity can begin.
During specification, the user requirements are systematically organized into a Software Requirements Specification (SRS) document. The customer requirements identified during the requirements gathering and analysis activity are organized into a SRS document. The important components of this document are functional requirements, the nonfunctional requirements, and the goals of implementation.

System design: -

The design phase is to transform the requirements specified from the SRS document into a design that is suitable for implementation in some programming language. During the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are available: the traditional design approach and the object-oriented design approach.

1. Traditional design approach

Traditional design consists of two different activities; first a structured analysis of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a structured design activity. During structured design, the results of structured analysis are transformed into the software design.

2. Object-oriented design approach

In this technique, various objects that occur in the problem domain and the solution domain are first identified, and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design.

Coding and unit testing:-

The coding and unit testing phase (sometimes referred to implementation phase because the design is implemented) of software development involves translating the software design into source code using a programming language. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been tested individually.
During this phase, each module is tested to determine that it works correctly. It involves testing each module in isolation as this is the most efficient way to debug the errors identified at this stage.

Integration and system testing: -

Integration of different modules is undertaken once they have been coded and unit tested. During the integration and system testing phase, the modules are integrated based on the system design. The different modules rarely integrated all at one instead integration is normally carried out incrementally over a number of steps.During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out. The goal of system testing is to ensure that the developed system conforms to its requirements laid out in the SRS document.
System testing usually consists of three different kinds of testing activities:

1. **Alpha** – testing: system testing performed by the development team.
2. **Beta**– testing: system testing performed by a friendly set of customers.
3. **Acceptance testing**: system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.

System testing is normally carried out in a planned manner according to the system test plan document. The system test plan identifies all testing related activities that must be performed, specifies the schedule of testing and allocates required resources. It also lists all the test cases and the expected outputs for each test case.

Maintenance: -

Maintenance of software products requires a lot more effort than to develop the product itself. Many studies carried out in the past confirm this and indicate that the relative effort of development of a typical software product to its maintenance effort is roughly in the 40:60 ratio. Maintenance involves performing any one or more of the following three kinds of activities:
Corrective Maintenance: correcting errors that were not discovered during the product development phase.
Perfective maintenance: Improving the implementation of the system, and enhancing the functionalities of the system according to the customer's requirements.
Adaptive maintenance : migrating the software to work in a new environment. For example, moving the software to work with a new operating system.
.

Advantages of waterfall model:

- Simple and easy to understand and use.

- Easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.

Disadvantages of waterfall model:

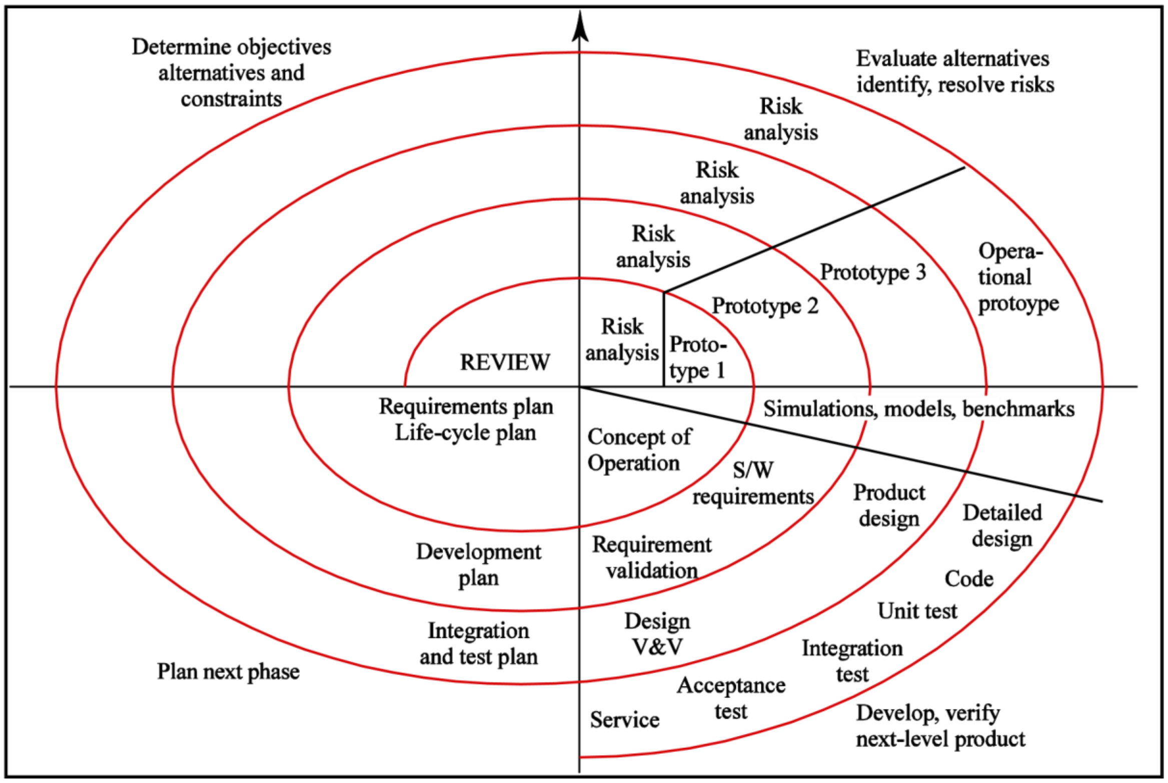
- Once an application is in the testing stage, it is very difficult to go back and change something that was not well-thought out in the concept stage.
- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing.

When to use the waterfall model:

- Requirements are very well known, clear and fixed.
- Product definition is stable.
- Technology is understood.
- There are no ambiguous requirements
- Ample resources with required expertise are available freely
- The project is short.

Spiral model

Many projects are *risk driven*. The process model proposed by Professor Barry Boehm aims to control and manage project risks during the course of development and provide opportunities for understanding the problem domain throughout the development and for building the system incrementally. The spiral model begins with an initial plan for development, evaluate risks, perhaps do some prototyping to evaluate alternatives at the system level, and produces a *concept of operations* document which describes how the system should work at a high level. From the concept of operations, a set of requirements for the system is extracted and this should be as complete as possible. The spiral model then uses to iteration to complete the development.Each iteration typically involves risk analysis, prototyping to determine the feasibility and desirability of various alternatives and then design, coding and testing.
It allows for a complete re-evaluation of the project direction after each spiral, and the requirements can be re-evaluated. However we need to ensure that new insights, which lead to new requirements are consistent with prior requirements and with the what has been built thus far



The Spiral model of software development is shown above .The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed. Each loop of the spiral represents a phase of the software process. For example, the innermost loop might be concerned with feasibility study the next loop with requirements specification and so on. Each phase in this model is split into four sectors (or quadrants). The following activities are carried out during each phase of a spiral model.

1. First quadrant (Objective Setting)

- During the first quadrant, it is needed to identify the objectives of the phase.
- Examine the risks associated with these objectives.

2. Second Quadrant (Risk Assessment and Reduction)

- A detailed analysis is carried out for each identified project risk.
- Steps are taken to reduce the risks. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

3. Third Quadrant (Development and Validation)

- Develop and validate the next level of the product after resolving the identified risks.

4. Fourth Quadrant (Review and Planning)

- Review the results achieved so far with the customer and plan the next iteration around the spiral.
- Progressively more complete version of the software gets built with each iteration around the spiral.

Advantages of Spiral model:

- High amount of risk analysis hence, avoidance of Risk is enhanced.
- Good for large and mission-critical projects.
- Strong approval and documentation control.
- Additional Functionality can be added at a later date.
- ▪ Software is produced early in the software life cycle.

Disadvantages of Spiral model:

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- ▪ Doesn't work well for smaller projects.

When to use Spiral model:

- When costs and risk evaluation is important
- For medium to high-risk projects
- Long-term project commitment unwise because of potential changes to economic priorities
- Users are unsure of their needs
- Requirements are complex
- New product line
- Significant changes are expected (research and exploration)

Prototype

A prototype is a model implementation of the system. It usually exhibits limited functional capabilities, low reliability, and inefficient performance compared to the actual software. A prototype is usually built using several shortcuts. The shortcuts might involve using inefficient, inaccurate, or dummy functions. The shortcut implementation of a function, for example, may produce the desired results by using a table look-up instead of performing the actual computations.

There are several uses of a prototype. An important purpose is to illustrate the input data formats, messages, reports, and the interactive dialogues to the customer. This is a valuable mechanism for gaining better understanding of the customer's needs:

- how the screens might look
- how the user interface would behave
- how the system would produce outputs

This is something similar to what the architectural designers of a building do; they show a prototype of the building to their customer. The customer can evaluate whether he likes it or not and the changes that he would need in the actual product. A similar thing happens in the case of a software product and its prototyping model.

Another reason for developing a prototype is that it is impossible to get the perfect product in the first attempt. Many researchers and engineers advocate that if you want to develop a good product you must plan to throw away the first version. The experience gained in developing the prototype can be used to develop the final product.

A prototyping model can be used when technical solutions are unclear to the development team. A developed prototype can help engineers to critically examine the technical issues associated with the product development. Often, major design decisions depend on issues like the response time of a hardware controller, or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype may be the best or the only way to resolve the technical issues.

Following is the stepwise approach to design a software prototype:

- Basic Requirement Identification: This step involves understanding the very basics product requirements especially in terms of user interface. The more intricate details of the internal design and external aspects like performance and security can be ignored at this stage.
- Developing the initial Prototype: The initial Prototype is developed in this stage, where the very basic requirements are showcased and user interfaces are provided. These features may not exactly work in the same manner internally in the actual software developed and the workarounds are used to give the same look and feel to the customer in the prototype developed.
- Review of the Prototype: The prototype developed is then presented to the customer and the other important stakeholders in the project. The feedback is collected in an organized manner and used for further enhancements in the product under development.
- Revise and enhance the Prototype: The feedback and the review comments are discussed during this stage and some negotiations happen with the customer based on factors like , time and budget constraints and technical feasibility of actual implementation. The changes accepted are again incorporated in the new Prototype developed and the cycle repeats until customer expectations are met.

Prototypes can have horizontal or vertical dimensions. Horizontal prototype displays the user interface for the product and gives a broader view of the entire system, without concentrating on internal functions. A vertical prototype on the other side is a detailed elaboration of a specific function or a sub system in the product.

The purpose of both horizontal and vertical prototype is different. Horizontal prototypes are used to get more information on the user interface level and the business requirements. It can even be presented in the sales demos to get business in the market. Vertical prototypes are technical in nature and are used to get details of the exact functioning of the sub systems. For example, database requirements, interaction and data processing loads in a given sub system.

There are different types of software prototypes used in the industry. Following are the major software prototyping types used widely:

1. Throwaway/Rapid Prototyping: Throwaway prototyping is also called as rapid or close ended prototyping. This type of prototyping uses very little efforts with minimum requirement analysis to build a prototype. Once the actual requirements are understood, the prototype is discarded and the actual system is developed with a much clear understanding of user requirements.
2. Evolutionary Prototyping: Evolutionary prototyping also called as breadboard prototyping is based on building actual functional prototypes with minimal functionality in the beginning. The prototype developed forms the heart of the future prototypes on top of which the entire system is built. Using

evolutionary prototyping only well understood requirements are included in the prototype and the requirements are added as and when they are understood.

3. Incremental Prototyping: Incremental prototyping refers to building multiple functional prototypes of the various sub systems and then integrating all the available prototypes to form a complete system.
4. Extreme Prototyping: Extreme prototyping is used in the web development domain. It consists of three sequential phases. First, a basic prototype with all the existing pages is presented in the html format. Then the data processing is simulated using a prototype services layer. Finally the services are implemented and integrated to the final prototype. This process is called Extreme Prototyping used to draw attention to the second phase of the process, where a fully functional UI is developed with very little regard to the actual services.

A prototype of the actual product is preferred in situations such as:

- user requirements are not complete
- technical issues are not clear

Advantages of Prototype model:

- Users are actively involved in the development
- Since in this methodology a working model of the system is provided, the users get a better understanding of the system being developed.
- Errors can be detected much earlier.
- Quicker user feedback is available leading to better solutions.
- Missing functionality can be identified easily
- Confusing or difficult functions can be identified

Disadvantages of Prototype model:

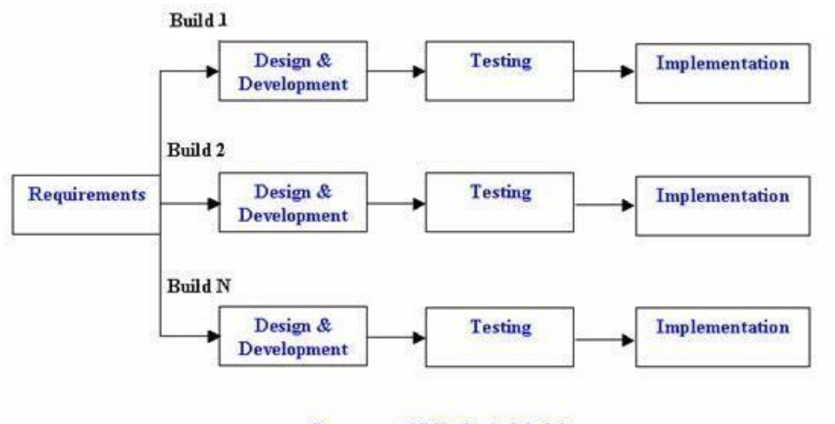
- Leads to implementing and then repairing way of building systems.
- Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans.
- Incomplete application may cause application not to be used as the full system was designed
- Incomplete or inadequate problem analysis.

When to use Prototype model:

- Prototype model should be used when the desired system needs to have a lot of interaction with the end users.
- Typically, online systems, web interfaces have a very high amount of interaction with end users, are best suited for Prototype model. It might take a while for a system to be built that allows ease of use and needs minimal training for the end user.
- Prototyping ensures that the end users constantly work with the system and provide a feedback which is incorporated in the prototype to result in a useable system. They are excellent for designing good human computer interface systems.

INCREMENTAL MODEL

The incremental model the whole requirement is divided into various builds. Multiple development cycles take place here, making the life cycle a "multi-waterfall" cycle. Cycles are divided up into smaller, more easily managed modules. Each module passes through the requirements, design, implementation and testing phases. A working version of software is produced during the first module, so you have working software early on during the software life cycle. Each subsequent release of the module adds function to the previous release. The process continues till the complete system is achieved.



Incremental Life Cycle Model

Advantages of Incremental model:

- Generates working software quickly and early during the software life cycle.
- More flexible – less costly to change scope and requirements.
- Easier to test and debug during a smaller iteration.
- Customer can respond to each built.
- Lowers initial delivery cost.
- Easier to manage risk because risky pieces are identified and handled during it'd iteration.

Disadvantages of Incremental model:

- Needs good planning and design.
- Needs a clear and complete definition of the whole system before it can be broken down and built incrementally.
- Total cost is higher than waterfall.

When to use the Incremental model:

- Requirements of the complete system are clearly defined and understood.
- Major requirements must be defined; however, some details can evolve with time.
- There is a need to get a product to the market early.
- A new technology is being used
- Resources with needed skill set are not available
- There are some high risk features and goals.

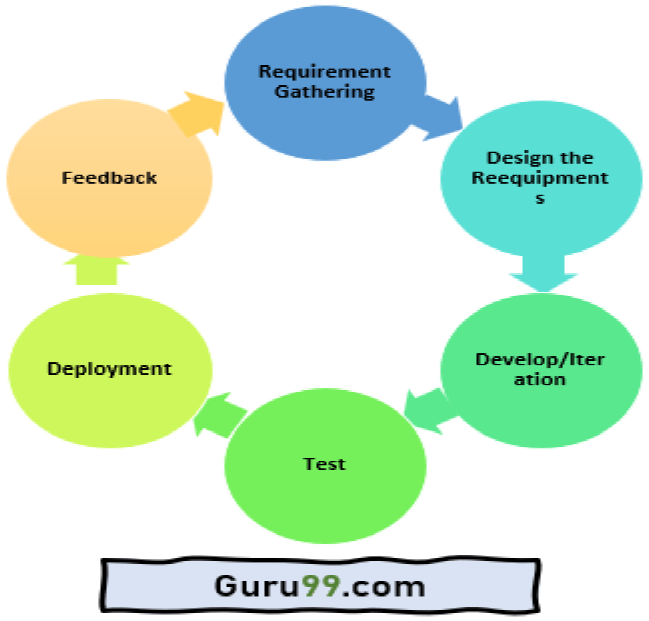
Agile Model

The Agile Model is an incremental and iterative process of software development. It defines each iteration's number, duration, and scope in advance. Every iteration is considered a short "frame" in the Agile process model, which mostly lasts from two to four weeks.

Agile Model divides tasks into time boxes to provide specific functionality for the release. Each build is incremental in terms of functionality, with the final build containing all the attributes. The division of the entire project into small parts helps minimize the project risk and the overall project delivery time.

Phases of Agile Model

Here are the different phases of Agile:



Here are the important stages involved in the Agile Model process in the SDLC life cycle:

- **Requirements Gathering:** In this Agile model phase, you must define the requirements. The business opportunities and the time and effort required for the project should also be discussed. By analyzing this information, you can determine a system's economic and technical feasibility.
- **Design the Requirements:** Following the feasibility study, you can work with stakeholders to define requirements. Using the UFD diagram or high-level UML diagram, you can determine how the new system will be incorporated into your existing software system.
- **Develop/Iteration:** The real work begins at this stage after the software development team defines and designs the requirements. Product, design, and development teams start working, and the product will undergo different stages of improvement using simple and minimal functionality.
- **Test:** This phase of the Agile Model involves the testing team. For example, the Quality Assurance team checks the system's performance and reports bugs during this phase.
- **Deployment:** In this phase, the initial product is released to the user.
- **Feedback:** After releasing the product, the last step of the Agile Model is feedback. In this phase, the team receives feedback about the product and works on correcting bugs based on the received feedback.

Compared to Waterfall, Agile cycles are short. There may be many such cycles in a project. The phases are repeated until the product is delivered.

Advantages

- Welcome changes in requirements, even late in the development phase.
- Daily co-operation between businesspeople and developers.
- Priority is customer collaboration over contract negotiation.
- It enables you to satisfy customers through early and frequent delivery.
- A strong emphasis is placed on face-to-face communication.
- Developing working software is the primary indicator of progress.
- Promote sustainable development pace.
- A continuous focus is placed on technical excellence and sound design.
- An improvement review is conducted regularly by the team.

Program Design Tools

Algorithms

An algorithm is a sequence of steps which results to a plan or strategy on how to go about solving a problem. There are different ways of presenting an algorithm. Some common ways include:-

- Pseudocode
- Flow charts

Pseudocode

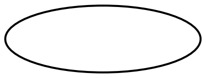
This is a case where an algorithm is expressed in English like statements (descriptions). For example the following pseudocode calculates the pay amount for five employees.

```
Start
    Initialize counter to 1
    Enter employee details
    Computer pay amount
    Print the pay amount
    Increment counter by one
    Check the value of the counter
    If counter < 6
        Loop to step 3
End
```

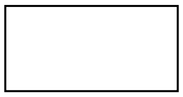
Flow Chart

This is a symbolic representation of an algorithm sequence. A flow chart uses predefined symbols to represent the various actions in an algorithm. The arrangement of the symbols indicates the flow of logic in the algorithm.

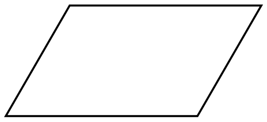
Flow chart symbols



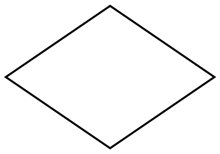
Start or end – They are used in a flow chart to mark the beginning and the end.



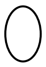
Process – used to represent operations on data e.g. computations. Details are written in a rectangular box.



Input/Output – Input/output operations are represented in a parallelogram.

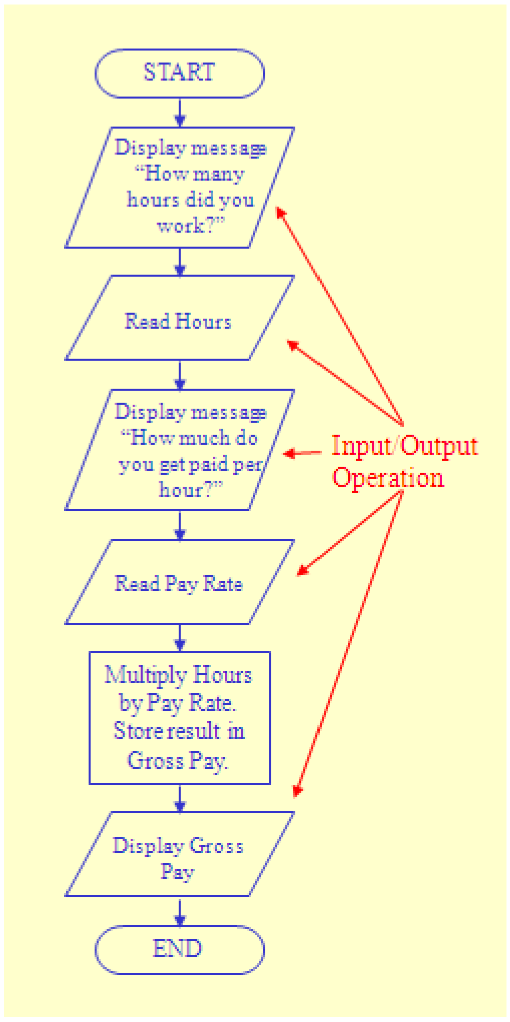


Decision – This symbol is used to indicate decision making and branching. The criterion is shown inside the symbol and the lines (paths) out show the results.



Connectors – A small circle containing a number or letter that is used to split a large flow chart into smaller parts.

Example


START
Display message "How many hours did you work?"
Read Hours
Display message "How much do you get paid per hour?"
Read Pay Rate
Multiply Hours by Pay Rate. Store result in Gross Pay.
Display Gross Pay
END
Input/Output Operation

Program Design

Structured Program Design

<u>**Structuring a program breaks it down into understandable chunks. Structured programming is an approach to writing programs that are easier to read, test, debug and modify. The approach assists in the development of large programs through stepwise refinement and modularity. Programs designed this way can be developed faster. When modules are used to develop large programs, several programmers can work on different modules, thereby reducing program development time.**</u>

In short, structured programming serves to increase programmer productivity, program reliability (readability and execution time), program testing, program debugging and serviceability.

Modular Programming

This is a technique that involves breaking down the entire problem into smaller, more manageable units.

Features

- Each module within the application carries out a singular task.
- Each module runs independently of the other modules.
- Since each module is independent, a breakdown in any module does not greatly affect the running of the application.
- Debugging is easier since errors can be traces to individual modules.

Advantages of modular programming

- Modules can be reused thus saving development time.
- Testing of individual modules in isolation makes tracing mistakes easier.
- An amendment to a single module does not affect the rest of the program.
- Ability to create libraries of often used routines which are reliable and can go into other programs.

Top-Down Approach

In this approach an outline program is designed first, showing the main tasks and components of the program, and the order in which they are to be executed. Each main component is then reduced to a number of smaller, simple and more manageable components and this process continues at each level until there is sufficient detail to allow the coding stage to proceed.

The process of reducing components into sequences of smaller components is often referred to a stepwise refinement and forms the basis of structured programming.

Bottom-up Approach

In this design approach, the application is developed starting at the bottom of the hierarchy i.e. the single task modules. As each category of programs is completed on the hierarchy, the controlling program for that category is created.

Program Coding – A readable program

- Data names – Meaningful names make a program code easier to read.
- Comments – Make the program more understandable.
- Indentation – Code should be laid out neatly, with proper use of indentation to reflect the logic structure of the code.
- Modularization- makes the program easy to understand.

Testing and Debugging

Testing is part of the procedures that ensure that corresponds with original specification and that it works in its intended environment. It is the process of running software to find errors (or bugs), though it is possible that some errors will get through the testing process without being found.

Types of Errors

There are three types of errors: Syntax, Semantic and Logic errors.

Syntax errors

They result from the incorrect use of the rules of programming. The compiler detects such errors as soon as you start compiling. A program that has syntax errors can produce no results. You should look for the error in the line suggested by the compiler. Syntax errors include;

- Missing semi colon at the end of a statement e.g. Area = Base * Length
- Use of an undeclared variable in an expression
- Illegal declaration e.g. int x, int y, int z;
- Use of a keyword in uppercase e.g. FLOAT, WHILE
- Misspelling keywords e.g. init instead of int

Note:

The compiler may suggest that other program line statements have errors when they may not. This will be the case when a syntax error in one line affects directly the execution of other statements, for example multiple use of an undeclared variable. Declaring the variable will remove all the errors in the other statements.

Logic Errors

These occur from the incorrect use of control structures, incorrect calculation, or omission of a procedure. Examples include: An indefinite loop in a program, generation of negative values instead of positive values. The compiler will not detect such errors since it has no way of knowing your intentions. The programmer must try to run the program so that he/she can compare the program's results with already known results.

Run Time Error

This is an error that occurs during the execution of a program. In contrast, *compile-time* errors occur while a program is being compiled. Runtime errors indicate bugs in the program or problems that the designers had anticipated but could do nothing about. For example, running out of memory will often cause a runtime error.

Semantic errors
They are caused by illegal expressions that the computer cannot make meaning of. Usually no results will come out of them and the programmer will find it difficult to debug such errors. Examples include a data overflow caused by an attempt to assign a value to a field or memory space smaller than the value requires division by zero, e.t.c.

C Language Basic Features

C is a general purpose programming language, unlike other languages such as PASCAL and FORTRAN developed for some specific uses. C is designed to work with both software and hardware. C has in fact been used to develop a variety of software such as:

- ✓ Operating systems: Unix and Windows.
- ✓ Application packages: WordPerfect and Dbase.

- **Source Code files**

When you write a program in C language, your instructions form the source code (or simply source file). C filenames have an extension .c. The part of the name before the period is called the base name and the part after the period is called the extension.

- **Object code, Executable code and Libraries**

An executable file is a file containing ready to run machine code. C accomplishes this in two steps.

- ✓ Compiling – The compiler converts the source code to produce the intermediate object code.
- ✓ The linker combines the intermediate code with other code to produce the executable file. C does this in a modular manner.

You can compile individual modules, and then combine the compiled modules later. Therefore, if you need to alter one module, you don't have to recompile the others.

Linking is the process where the object code, the start up code[*], and the code for library routines used in the program (all in machine language) are combined into a single file - the executable file.

Advantages of C over Other Languages

- **C Supports structured programming design features.**
  It allows programmers to break down their programs into functions. Further it supports the use of comments, making programs readable and easily maintainable.
- **Efficiency**
  - ✓ C is a concise language that allows you to say what you mean in a few words.
  - ✓ The final code tends to be more compact and runs quickly.
- **Portability**
  C programs written for one system can be run with little or no modification on other systems.
- **Power and flexibility**
  - ✓ C has been used to write operating systems such as Unix, Windows.
  - ✓ It has (and still is) been used to solve problems in areas such as physics and engineering.
- **Programmer orientation**
  - ✓ C is oriented towards the programmer's needs.
  - ✓ It gives access to the hardware. It lets you manipulate individual bits of memory.

[*] Code that acts as interface between the program and the operating system.

✓ It also has a rich selection of operators that allow you to expand programming capability.

C Programs' Components

Keywords

These are reserved words that have special meaning in a language. The compiler recognizes a keyword as part of the language's built – in syntax and therefore it cannot be used for any other purpose such as a variable or a function name. C keywords **must be used in lowercase** otherwise they will not be recognized.

Examples of keywords

auto	break	case	else	int	void
default	do	double	if	sizeof	long
float	for	goto	signed	unsigned**Error! Bookmark not defined.**	
register	return	short	union	continue	
struct	switch	typedef	const	extern	
volatile	while	char	enum	static	

A typical C program is made of the following components:

- Preprocessor directives
- Functions
- Declaration statements
- Comments
- Expressions
- Input and output statements

Sample Program

This program will print out the message: **This is a C program**.

```
#include<stdio.h>
main()
{
   printf("This is a C program \n");
    return 0;
}
```

Every C program contains a function called **main**. This is the start point of the program. **#include<stdio.h>** allows the program to interact with the screen, keyboard and file system of your computer. You will find it at the beginning of almost every C program.

main() declares the start of the function, while the two curly brackets show the start and finish of the function. Curly brackets in C are used to group statements together as in a function, or in the body of a loop. Such a grouping is known as a compound statement or a block.

printf("This is a C program \n"); prints the words on the screen. The text to be printed is enclosed in double quotes. The **\n** at the end of the text tells the program to print a new line as part of the output.

Most C programs are in lower case letters. You will usually find upper case letters used in preprocessor definitions (which will be discussed later) or inside quotes as parts of character strings.

C is case sensitive, that is, it recognises a lower case letter and it's upper case equivalent as being different.

Example: Basic C program features

```
/* Sample Program */
#include<stdio.h>
main()
{
        int num;                /* define a variable called num  */
        num = 1;                /* assignment  */
        printf(" This is a simple program ");
        printf("to display a message. \n");
        printf ("My favorite number is  %d  because ", num);
        printf(" it is first.\n ");
        return 0;
}
```

On running the above program, you get the following output.

```
        This is a simple program to display a message.
        My favorite number is 1 because it is first.
```

Functions

All C programs consist of one or more functions, each of which contains one or more **statements.** In C, a function is a named subroutine that can be called by other parts of the program. Functions are the building blocks of C.

A *statement* specifies an action to be performed by the program. In other words, statements are parts of your program that actually perform operations.

All C statements must end with a semicolon. C does not recognize the end of a line as a terminator. This means that there are no constraints on the position of statements within a line. Also you may place two or more statements on one line.

Although a C program may contain several functions, the only function that it must have is **main ( )**.

The **main( )** function is the point at which execution of your program begins. That is, when your program begins running, it starts executing the statements inside the **main( )** function, beginning with the first statement after the opening curly brace. Execution of your program terminates when the closing brace is reached.

Another important component of all C programs is *library functions*. The ANSI C standard specifies a minimal set of library functions to be supplied by all C compilers, which your program may use. This collection of functions is called the *C standard library*. The standard library contains functions to perform disk I/O (input / output), string manipulations, mathematics, and much more. When your program is compiled, the code for library functions is automatically added to your program.

One of the most common library functions is called **printf( )**. This is C's general purpose output function. Its simplest form is

printf("string – to – output");

The printf( ) outputs the characters that are contained between the beginning and ending double quotes.

For example, **printf(" This is a C program ");**

The double quotes are not displayed on the screen. In C, one or more characters enclosed between double quotes is called a *string*. The quoted string between printf( )'s parenthesis is called an *argument* to printf( ). In general, information passed to a function is called an argument. In C, calling a library function such as printf( ) is a statement; therefore it must end with a semicolon.

To call a function, you specify its name followed by a parenthesized list of arguments that you will be passing to it. If the function does not require any arguments, no arguments will be specified, and the parenthesized list will be empty. If there is more than one argument, the arguments must be separated by commas.

In the above program, line 7 causes the message enclosed in speech marks " " to be printed on the screen. Line 8 does the same thing.

The \n tells the computer to insert a new line after printing the message. \n is an example of an escape sequence.

Line 9 prints the value of the variable num (1) embedded in the phrase. The %d instructs the computer where and in what form to print the value. %d is a **type specifier** used to specify the output format for integer numbers.

Line 10 has the same effect as line 8.

Line11 indicates the value to be returned by the function **main( )** when it is executed. By default any function used in a C program returns an integer value (when it is called to execute). Therefore, line 3 could also be written **int main( ).** If the **int** keyword is omitted, still an integer is returned.

Then, why **return (0); ?** Since all functions are subordinate to **main( )**, the function does not return any value.

Note:

(i) Since the main function does not return any value, line 3 can alternatively be written as : **void main( )** – void means valueless. In this case, the statement **return 0;** is not necessary.

(ii) While omitting the keyword **int** to imply the return type of the **main( )** function does not disqualify the fact that an integer is returned (since **int** is default), you should explicitly write it in other functions, especially if another value other than zero is to be returned by the function.

Preprocessor directives and header files

A preprocessor directive performs various manipulations on your source file before it is actually compiled. Preprocessor directives are not actually part of the C language, but rather instructions from you to the compiler

The preprocessor directive #include is an instruction to read in the contents of another file and include it within your program. This is generally used to read in header files for library functions.

Header files contain details of functions and types used within the library. They must be included before the program can make use of the library functions.

Library header file names are enclosed in angle brackets, < >. These tell the preprocessor to look for the header file in the standard location for library definitions.

Comments

Comments are non – executable program statements meant to enhance program readability and allow easier program maintenance, i.e. they document the program. They can be used in the same line as the material they explain (see lines 4, 6, 7 in sample program).

A long comment can be put on its own line or even spread on more than one line. Comments are however optional in a program. The need to use too many comments can be avoided by good programming practices such as use of sensible variable names, indenting program statements, and good logic design. Everything between the opening /* and closing */ is ignored by the compiler.

Declaration statements

In C, all variables must be declared before they are used. Variable declarations ensure that appropriate memory space is reserved for the variables, depending on the data types of the variables. Line 6 is a declaration for an integer variable called num.

Assignment and Expression statements

An assignment statement uses the assignment operator "=" to give a variable on the operator's left side the value to the operator's right or the result of the expression on the right. The statement num =1; (Line 6) is an assignment statement.

Escape sequences

Escape sequences (also called back slash codes) are character combinations that begin with a backslash symbol (\) used to format output and represent difficult-to-type characters.

One of the most important escape sequences is \n, which is often referred to as the new line character. When the C compiler encounters \n, it translates it into a carriage return.

For example, this program:

```
#include<stdio.h>

        main()

        {

          printf("This is line one  \n");

          printf("This is line two \n");

          printf("This is line three");

          return 0;

        }
```

displays the following output on the screen.

```
This is line one
This is line two
This is line three
```

The program below sounds the bell.

```
#include<stdio.h>

main()

{

          printf("\a");

          return 0;

}
```

Remember that the escape sequences are character constants. Therefore to assign one to a character variable, you must enclose the escape sequence within single quotes, as shown in this fragment.

```
Char ch;
```

```
ch = '\t '            /*assign ch the tab character  */
```

Below are other escape sequences:

Escape sequence	Meaning
\a	alert/bell
\b	backspace
\n	new line
\v	vertical tab
\t	horizontal tab
\\	back slash
\'	Single quote (')
\"	Double quote ("")
\0	null

Revision Exercises

1. Outline the logical stages of C programs' development.
2. From the following program, suggest the syntax and logical errors that may have been made.
   The program is supposed to find the square and cube of 5, then output 5, its square and cube.

```
#include<stdio.h>
main()
{
  int , int n2, n3;
  n = 5;
  n2 = n *n
  n3 = n2 * n2;
  printf(" n = %d, n squared = %d, n cubed = %d \ n", n, n2, n3);
  return 0;
}
```

3. Give the meaning of the following, with examples
   (i) Preprocessor command
   (ii) Keyword
   (iii) Escape sequence
   (iv) Comment

(v) Linking
(vi) Executable file

4. Provide the meaning of the following keywords, giving examples of their use in C programs.
(i) void
(ii) return
(iii) extern
(iv) struct
(v) static

5. C is both ‘portable’ and ‘efficient’. Explain.

6. C is a ‘case sensitive’ language. Explain.

7. The use of comments in C programs is generally considered to be good programming practice. Why?

DATA HANDLING

Variables

A variable is a memory location whose value can change during program execution. In C, a variable must be declared before it can be used. Variables can be declared at the start of any block of code.

A declaration begins with the type, followed by the name of one or more variables. For example,

```
int high, low, results[20];
```

Declarations can be spread out, allowing space for an explanatory comment. Variables can also be initialised when they are declared. This is done by adding an equals sign and the required value after the declaration.

```
int high = 250;         /* Maximum Temperature */
int low = -40;          /* Minimum Temperature */
int results[20];   /* Series of temperature readings */
```

Variable Names

Every variable has a name and a value. The name identifies the variable and the value stores data. There is a limitation on what these names can be. Every variable name in C must start with a letter; the rest of the name can consist of letters, numbers and underscore characters.

C recognizes upper and lower case characters as being different (C is case- sensitive). Finally, you cannot use any of C's keywords like main, while, switch etc as variable names.

Examples of legal variable names

x	result	outfile	bestyet
x1	x2	out_file	best_yet
power	impetus	gamma	hi_score

It is conventional to avoid the use of capital letters in variable names. These are used for names of constants. Some old implementations of C only use the first 8 characters of a variable name. Most modern ones don't apply this limit though. The rules governing variable names also apply to the names of functions.

Basic Data Types

C supports five basic data types. The table below shows the five types, along with the C keywords that represent them. Don't be confused by *void*. This is a special purpose data type used to explicitly declare functions that return no value.

Type	**Meaning**	**Keyword**
Character	Character data	char
Integer	Signed whole number	int
Float	floating-point numbers	float
Double	double precision floating-point numbers	double
Void	Valueless	void

The 'int' specifier

It is a type specifier used to declare integer variables. For example, to declare count as an integer you would write:

int count;

Integer variables may hold signed whole numbers (numbers with no fractional part). Typically, an integer variable may hold values in the range –32,768 to 32,767 and are 2 bytes long.

The 'char' specifier

A variable of type char is 1 byte long and is mostly used to hold a single character. For example to declare **ch** to be a character type, you would write:

char ch;

The 'float' specifier

It is a type specifier used to declare floating-point variables. These are numbers that have a whole number part and a fractional or decimal part for example 234.936. To declare **f to** be of type float, you would write:

float f;

Floating point variables typically occupy 4 bytes.

The 'double' specifier

It is a type specifier used to declare double-precision floating point variables. These are variables that store float point numbers with a precision twice the size of a normal float value. To declare **d** to be of type double you would write:

double d;

Double-type variables typically occupy 8 bytes.

Using printf( ) To Output Values

You can use printf( ) to display values of characters, integers and floating - point values. To do so, however, requires that you know more about the **printf( )** function.

For example:

printf("This prints the number %d ", 99);

displays **This prints the number 99** on the screen. As you can see, this call to the printf( ) function contains two arguments. The first one is the quoted string and the other is the constant 99. Notice that the arguments are separated from each other by a comma.

In general, when there is more than one argument to a function, the arguments are separated from each other by commas. The first argument is a quoted string that may contain either normal characters or formal specifiers that begin with a percent (%) sign.

Normal characters are simply displayed as is on the screen in the order in which they are encountered in the string (reading left to right). A format specifier, on the other hand informs **printf( )** that a different type item is being displayed. In this case, the **%d**, means that an integer, is to be output in decimal format. The value to be displayed is to be found in the second argument. This value is then output at the position at which the format specifier is found on the string.

If you want to specify a character value, the format specifier is %c. To specify a floating-point value, use %f. The %f works for both **float** and **double.** Keep in mind that the values matched with the format specifier need not be constants, they may be variables too.

Code	**Format**
%c	Character
%d	Signed decimal integers
%i	Signed decimal integers
%e	Scientific notation (lowercase 'e')
%E	Scientific notation (lowercase 'E')
%f	Decimal floating point
%s	String of characters
%u	Unsigned decimal integers
%x	Unsigned hexadecimal (lowercase letters)
%X	Unsigned hexadecimal (Uppercase letters)

Examples

1. The program shown below illustrates the above concepts. First, it declares a variable called **num**. Second, it assigns this variable the value 100. Finally, it uses **printf( )** to display **the value is 100** on the screen. Examine it closely.

```
#include <stdio.h>
main()
{
   int num;
   num = 100;
   printf(" The value is %d ", num);
   return 0;
}
```

2. This program creates variables of types **char, float**, and **double** assigns each a value and outputs these values to the screen.

```
#include<stdio.h>
main()
{
      char ch;
      float f;
      double d;
      ch = 'X';
      f = 100.123;
      d  = 123.009;

      printf(" ch is %c ", ch);
      printf(" f  is %f ", f);
      printf(" d  is %f ", d);
      return 0;
}
```

Exercises

1. Enter, compile, and run the two programs above.
2. Write a program that declares one integer variable called **num**. Give this variable the 1000 and then, using one **printf ( )** statement, display the value on the screen like this:

1000 is the value of num

Inputting Numbers From The Keyboard Using scanf( )

There are several ways to input values through the keyboard. One of the easiest is to use another of C's standard library functions called **scanf( ).**

To use **scanf( )** to read an integer value from the keyboard, call it using the general form:
scanf("%d", &*int-var-name*);

Where *int-var-name* is the name of the integer variable you wish to receive the value. The first argument to **scanf( )** is a string that determines how the second argument will be treated. In this case the %d specifies that the second argument will be receiving an integer value entered in decimal format. The fragment below, for example, reads an integer entered from the keyboard.

```
int num;
scanf("%d", &num);
```

The **&** preceding the variable name means 'address of'. The values you enter are put into variables using the variables' location in memory. It allows the function to place a value into one of its arguments.

When you enter a number at the keyboard, you are simply typing a string of digits. The **scanf( )** function waits until you have pressed **<ENTER>** before it converts the string into the internal format used by the computer.

The table below shows format specifiers or codes used in the scanf() function and their meaning.

Code	**Meaning**
%c	Read a single character
%d	Read a decimal integer
%i	Read a decimal integer
%e	Read a floating point number
%f	Read a floating point number
%lf	Read a double
%s	Read a string
%u	Reads an unsigned integer

Examples

1. This program asks you to input an integer and a floating-point number and displays the value.

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

```
        float f;
        printf(" \nEnter an integer: ");
        scanf( "%d ", &num);
        printf("\n Enter a floating point number: ");
        scanf( "%f ", &f);
        printf( "%d  ", num);
        printf( "\n %f ", f);
        return 0;
      }
```

2. This program computes the area of a rectangle, given its dimensions. It first prompts the user for the length and width of the rectangle and then displays the area.

```
#include<stdio.h>
main()
{
          int len, width;
          printf("\n Enter length:  ");
          scanf ("%d ", &len);
          printf("\n Enter width :  " );
          scanf( " %d ", &width);
          printf("\n The area is %d ", len  * width);
        return 0;
}
```

Exercises

1. Enter, compile and run the example programs.
2. Write a program that inputs two floating-point numbers (use type **float)** and then displays their sum.
3. Write a program that computes the volume of a cube. Have the program prompt the user for each dimension.

Types of Variables

There are two places where variables are declared: inside a function or outside all functions.

Variables declared outside all functions are called **global variables** and they may be accessed by any function in your program. Global variables exist the entire time your program is executing.

Variables declared inside a function are called **local variables.** A local variable is known to and may be accessed by only the function in which it is declared. You need to be aware of two important points about local variables.

(i) The local variables in one function have no relationship to the local variables in another function. That is, if a variable called **count** is declared in one function, another variable called **count** may also be declared in a second function – the two variables are completely separate from and unrelated to one another.

(ii) Local variables are created when a function is called, and they are destroyed when the function is exited. Therefore local variables do not maintain their values between function calls.

Constants

A constant is a value that does not change during program execution. In other words, constants are fixed values that may not be altered by the program.

Integer constants are specified as numbers without fractional components. For example –10, 1000 are integer constants.

Floating - point constants require the use of the decimal point followed by the number's fractional component. For example, 11.123 is a floating point constant. C allows you to use scientific notation for floating point numbers. Constants using scientific notation must follow this general form:

number* E *sign exponent

The number is optional. Although the general form is shown with spaces between the component parts for clarity, there may be no spaces between parts in an actual number . For example, the following defines the value 1234.56 using scientific notation.

```
123.456E1
```

Character constants are usually just the character enclosed in single quotes; 'a', 'b', 'c'.

```
ch = ‘z’;
```

Note:
There is nothing in C that prevents you from assigning a character variable a value using a numeric constant. For example the ASCII Code for ‘A ‘ is 65. Therefore, these two assignments are equivalent.

```
char ch;
ch = “A’;
ch = 65;
```

Types of Constants

Constants can be used in C expressions in two ways:

- **Directly**
  Here the constant value is inserted in the expression, as it should typically be.
  For example:

  ```
  Area = 3.14 * Radius * Radius;
  ```

The value **3.14** is used directly to represent the value of **PI** which never requires changes in the computation of the area of a circle

- **Using a Symbolic Constant**
  This involves the use of another C preprocessor, #define.

For example, **#define SIZE 10**
A symbolic constant is an identifier that is replaced with replacement text by the C preprocessor before the program is compiled. For example, all occurrences of the symbolic constant **SIZE** are replaced with the replacement text 10.

This process is generally referred to as *macro substitution.* The general form of the #define statement is;

#define ***macro-name string***

Notice that this line does not end in a semi colon. Each time the ***macro - name*** is encountered in the program, the associated ***string*** is substituted for it. For example, consider the following program.

Example: Area of a circle

```
#include <stdio.h>
#define PI  3.14
main()
{
  float radius, area;
  printf("Enter the radius of the circle \n");
  scanf("%f", &radius);
  area = PI * radius * radius;  /* PI is a symbolic constant */
  printf("Area is %.2f cm  squared ",area);
  return 0;
}
```

Note:
At the time of the substitution, the text such as 3.14 is simply a string of characters composed of 3, ., 1 and 4. The preprocessor does not convert a number into any sort of internal format. This is left to the compiler.

Revision Exercises

1. Discuss four fundamental data types supported by C, stating how each type is stored in memory.
2. Distinguish between a variable and a constant.
3. Suggest, with examples two ways in which constant values can be used in C expression statements.
4. Give the meaning of the following declarations;
   (i) **char name[20];**
   (ii) **int num_emp;**
   (iii) **double tax, basicpay;**
   (iv) **char response;**

5. What is the difference between a local and a global variable?
6. Write a program that computes the number of seconds in a year.
   The mass of a single molecule of water is about 3.0×10^{-23} grams. A quart of water is about 950 grams. Write a program that requests an amount of water in quarts and displays the number of water molecules in that amount.

OPERATORS

Operators And Operands

An **operator** is a component of any expression that joins individual constants, variables, array elements and function references.

An **operand** is a data item that is acted upon by an operator. Some operators act upon two operands (binary operators) while others act upon only one operand (unary operators).

An operand can be a constant value, a variable name or a symbolic constant.

Note*: An expression is a combination of operators and operands.*

Examples
(i) x + y ; x, y are operands, + is an addition operator.
(ii) 3 * 5; 3, 5 are constant operands, * is a multiplication operator.
(iii) x % 2.5; x, 5 are operands, % is a modulus (remainder) operator.
(iv) sizeof (int); sizeof is an operator (unary), int is an operand.

Arithmetic Operators

There are five arithmetic operators in C.

Operator	**Purpose**
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder after integer division

Note:

(i) There exists no exponential operators in C.
(ii) The operands acted upon by arithmetic operators must represent numeric values, that is operands may be integers, floating point quantities or characters (since character constants represent integer values).
(iii) The % (remainder operator) requires that both operands be integers. Thus;
- 5 % 3
- int x = 8;
- int y = 6 ; x % y are valid while;
- 8.5 % 2.0 and
- float p = 6.3, int w = 7 ; 5 %p , p % w are invalid.

(iv) Division of one integer quantity by another is known as an integer division. If the quotient (result of division) has a decimal part, it is truncated.

(v) Dividing a floating point number with another floating point number, or a floating point number with an integer results to a floating point quotient .

Exercise

Suppose a = 10, b = 3, v1 = 12.5, v2 = 2.0, c1 =’P’, c2 = ‘T’. Compute the result of the following expressions.

a + b	v1 * v2
a - b	v1 / v2
a * b	c1
a / b	c1 + c2 +5
a % b	c1 + c2 +’9’

Note:

(i) c1 and c2 are character constants
(ii) ASCII codes for 9 is 57, P = 80,T = 84.

If one or both operands represent negative values, then the addition, subtraction, multiplication, and division operators will result in values whose signs are determined by their usual rules of algebra. Thus if a b, and c are 11, -3 and –11 respectively, then

a + b = 8
a – b = 14
a * b = -33
a / b = -3
a % b = -2
c % b = -2
c / b = 3

Examples of floating point arithmetic operators

r1 = -0.66, r2 = 4.50 (operands with different signs)
r1 + r2 = 3.84
r1 - r2 = -5.16
r1 * r2 = -2.97
r1 / r2 = -0.1466667

Note:

(i) If both operands are floating point types whose precision differ (e.g. a float and a double) the lower precision operand will be converted to the precision of the other operand, and the result will be expressed in this higher precision. (Thus if an expression has a float and a double operand, the result will be a double).
(ii) If one operand is a floating-point type (e.g. float, double or long double) and the other is a character or integer (including short or long integer), the character or integer will be converted to the floating point type and the result will be expressed as such.

(iii) If neither operand is a floating-point type but one is long integer, the other will be converted to long integer and the result is expressed as such. (Thus between an int and a long int, the long int will be taken).

(iv) If neither operand is a floating type or long int, then both operands will be converted to int (if necessary) and the result will be int (compare short int and long int)

From the above, evaluate the following expressions given:
i =7, f = 5.5, c = ’w’. State the type of the result.

(i) i + f
(ii) i + c
(iii) i + c-‘w’
(iv) (i + c) - (2 * f / 5)

(‘w” has ASCII decimal value of 119)

***Note*:** Whichever type of result an expression evaluates to, a programmer can convert the result to a different data type if desired. The general syntax of doing this is:

(data type) expression.
The data type must be enclosed in parenthesis (). For example the expression (i + f) above evaluates to 12.5. To convert this to an integer, it will be necessary to write
(int) (i + f).

Operator Precedence

The order of executing the various operations makes a significant difference in the result. C assigns each operator a precedence level. The rules are;

(i) Multiplication and division have a higher precedence than addition and subtraction, so they are performed first.

(ii) If operators of equal precedence; (*, /), (+, -) share an operand, they are executed in the order in which they occur in the statement. For most operators, the order (associativity) is from left to right with the exception of the assignment (=) operator.

Consider the statement;
butter = 25.0 + 60.0 * n / SCALE;
Where n = 6.0 and SCALE = 2.0.

The order of operations is as follows;

First: 60.0 * n = 360.0
(Since * and / are first before + but * and / share the operand n with * first)

Second: 360.0 / SCALE = 180
(Division follows)

Third: 25.0 + 180 = 205.0 (Result)
(+ comes last)

Note that it is possible for the programmer to set his or her own order of evaluation by putting, say, parenthesis. Whatever is enclosed in parenthesis is evaluated first.

What is the result of the above expression written as:
(25.0 + 60.0 * n) / SCALE.

Example: Use of operators and their precedence

```
/* Program to demonstrate use of operators and their precedence */
include<stdio.h >
main()
{
        int score,top;
        score = 30;
        top = score - (2*5) + 6 * (4+3) + (2+3);
        printf ("top = %d \ n" , top);
        return 0;
}
```

Try changing the order of evaluation by shifting the parenthesis and note the change in the top score.

Exercise

The roots of a quadratic equation $ax^2 + bx + c = 0$ can be evaluated as:

$$x1 = (-b + \sqrt{(b^2 - 4ac)})/2a$$
$$x2 = (-b + \sqrt{(b^2 - 4ac)})/2a$$

where a, b ,c are double type variables and $b^2 = b * b$, 4ac = 4 * a * c, 2a = 2 * a.

Write a program that calculates the two roots x_1 x_2 with double precision, and displays the roots on the screen.

Example: Converting seconds to minutes and seconds using the % operator

```
#include<stdio.h >
#define SEC_PER_MIN 60
main()
{
  int sec, min, sec_left;
  printf(" Converting  seconds to minute and seconds \n ") ;
  printf( "Enter number of seconds you wish to convert \n ") ;
  scanf("% d" , &sec ) ;             /* Read in number of seconds */
  min = sec / SEC_PER_MIN ;    / * Truncate number of seconds */
  sec_left = sec % SEC_PER_MIN ;
  printf("% d seconds is % d minutes,% seconds\n " ,sec,min,sec_left);
  return 0;
}
```

The sizeof operator

sizeof returns the size in bytes, of its operand. The operand can be a data type e.g. *sizeof (int)*, or a specific data object e.g. *sizeof n*.

If it is a name type such as int, float etc. The operand should be enclosed in parenthesis.

Example : Demonstrating 'sizeof' operator

```
#include <stdio.h>
main()
{
        int n;
        printf("n has % d bytes; all ints have % d bytes \n",
                                                        sizeof n, sizeof(int)) ;
          return 0;
}
```

The Assignment Operator

The Assignment operator (=) is a value assigning operator. There are several other assignment operators in C. All of them assign the value of an expression to an identifier.

Assignment expressions that make use of the assignment operator (=) take the form;

identifier = expression;

where *identifier* generally represents a variable, constant or a larger expression.

Examples of assignment;

```
a = 3 ;
x = y ;
pi = 3.14;
sum = a + b ;
```

area_circle = pi * radius * radius;

Note

(i) You cannot assign a variable to a constant such as 3 = a ;
(ii) The assignment operator = and equality operator (= =) are distinctively different. The = operator assigns a value to an identifier. The equality operator (= =) tests whether two expressions have the same value.
(iii) Multiple assignments are possible e.g. a =b = 5 ; assigns the integer value 5 to both a and b.
(iv) Assignment can be combined with +, -, /, *, and %

The Conditional Operator

Conditional tests can be carried out with the conditional operator (**?**). A conditional expression takes the form:

expression1 ? expression2 : expression3 and implies;

evaluate **expression1**. If **expression1** evaluates to **true** (value is 1 or non zero) then evaluate **expression 2**, otherwise (i.e. if expression 1 is false or zero) , evaluate **expression3**.

Consider the statement **(i < 0) ? 0 :100**

Assuming **i** is an integer, the expression (i < 0) is evaluated and if it is true, then the result of the entire conditional expression is zero (0), otherwise, the result will be 100.

Unary Operators

These are operators that act on a single operand to produce a value. The operators may precede the operand or are after an operand.

Examples

(i) Unary minus e.g. - 700 or –x
(ii) Incrementation operator e.g. c++
(iii) Decrementation operator e.g. f - -
(iv) sizeof operator e.g. sizeof(float)

Relational Operators

There are four relational operators in C.

<	Less than
<=	Less than or equal to
>	Greater than

- > = Greater than or equal to

Closely associated with the above are two equality operators;

- = = Equal to
- ! = Not equal to

The above six operators form **logical expressions.**

A logical expression represents conditions that are either true (represented by integer 1) or false (represented by 0).

Example

Consider a, b, c to be integers with values 1, 2,3 respectively. Note their results with relational operators below.

Expression	**Result**
a < b	1 (true)
(a+ b) > = c	1 (true)
(b + c) > (a+5)	0 (false)
c : = 3	0 (false)
b = = 2	1 (true)

Logical operators

&&	Logical AND
||	Logical OR
!	NOT

The two operators act upon operands that are themselves logical expressions to produce more complex conditions that are either true or false.

Example

Suppose i is an integer whose value is 7, f is a floating point variable whose value is 5.5 and C is a character that represents the character ‘w’, then;

(i > = = 6) && (C = = ‘w’) is 1 (true)
(C’ > = 6) || (C = 119) is 1 (true)
(f < 11) && (i > 100) is 0 (false)
(C! = ‘ p’) || ((i + f) < = 10) is 1 (true)

Revision Exercises

1. Describe with examples, four relational operators.
2. What is 'operator precedence'? Give the relative precedence of arithmetic operators.
3. Suppose a, b, c are integer variables that have been assigned the values a =8, b = 3 and c = - 5, x, y, z are floating point variables with values x =8.8, y = 3.5, z = -5.2.

Further suppose that c1, c2, c3 are character-type variables assigned the values E, 5 and ? respectively.

Determine the value of each of the following expressions:

(i) a / b
(ii) 2 * b + 3 * (a – c)
(iii) (a * c) % b
(iv) (x / y) + z
(v) x % y
(vi) 2 * x / (3 * y)
(vii) c1 / c3
(viii) (c1 / c2) * c3

CONTROL STRUCTURES

Introduction

Control structures represent the forms by which statements in a program are executed.

Three structures control program execution:

- Sequence
- Selection or decision structure
- Iteration or looping structure

Sequence Control Structure

This is the simplest control structure. It dictates the order of execution from one statement to the next within the program. In the absence of repetition or branching, the program instructions will be executed in the order in which they were coded.

Basically, program statements are executed in the sequence in which they appear in the program.

Decision/Selection Control Structure

The structure is used for decision making within a program. It allows alternative actions to be taken according to conditions that exist at particular stages within a program.

The structure uses a test condition statement, which upon evaluation by the computer gives rise to certain conditions which may evaluate to a Boolean true or false. Based on the outcome of the test condition, the computer may execute one or more statements.

In reality, a logical test using logical and relational operators may require to be used in order to determine which actions to take (subsequent statements to be executed) depending on the outcome of the test. This is **selection**. For example:

```
if (score >= 50)
     printf("Pass");
else
     printf("Fail");
```

In addition, a group of statements in a program may have to be executed repeatedly until some condition is satisfied. This is known as **looping**. For example, the following code prints digits from 1 to 5.

```
for(digit = 1;  digit < = 5; digit++)
     printf("\n %d", digit)
```

Selection Structure

The if statement

The *if* statement provides a junction at which the program has to select which path to follow. The general form is :

```
if(expression)
     statement;
```

If *expression* is true (i.e. non zero) , the *statement* is executed, otherwise it is skipped. Normally the expression is a relational expression that compares the magnitude of two quantities (For example x > y or c = = 6)

Examples

```
(i) if (x<y)
    printf("x is less that y");

(ii) if (salary >500)
            Tax_amount = salary  *  1.5;

(iii) if(balance<1000 || status ='R')
            print ("Balance  =  %f", balance);
```

The statement in the if structure can be a single statement or a block (compound statement).
If the statement is a block (of statements), it must be marked off by braces.

```
if(expression)
      {
            block of statements;
      }
```

Example

```
if(salary>5000)

{

   tax_amt = salary *1.5;

   printf("Tax charged is %f", tax_amt);

}
```

if - else statement

The if else statement lets the programmer choose between two statements as opposed to the simple if statement which gives you the choice of executing a statement (possibly compound) or skipping it.

The general form is:

```
        if (expression)
            statement;1
      else
              statement2;
```

If expression is true, statement1 is executed. If expression is false, the single statement following the else (statement2) is executed. The statements can be simple or compound.

Note: Indentation is not required but it is a standard style of programming.

Example:

```
if(x >=0)
{
  printf("let us increment x:\n");
  x++;
}
 else
  printf("x  < 0 \n");
```

Multiple Choice: else if (Nested IF)

This is a control structure that is used when more than two choices have to be made. It involves having IF structure inside another IF structure either in the true or false part.

The general form is:

```
 if (expression 1)
       statement 1;
else if (expression 2)
          statement 2;
else if (expression 3)
      statement 3;
-------------
```

```
                    else
                    statement n;
```

(Braces still apply for block statements) In this structure expression 2 is only tested if condition one is false. Explain how execution of the statements occurs.

Examples:
1. A program to call students marks, grade and output the marks and grade.
2. A program to determine the job group of an employee given some scales.
3. A program to prompt for three values a, b and c and determine which is the greatest value.

Example

```
if(sale_amount>=10000)
  Disc= sal_amt* 0.10;                                  /*ten percent/
else if (sal_amt >= 5000  && sal_amt < 1000 )
   printf (“The discount is %f “,sal_amt*0.07 ); /*seven percent */
else  if (sal_amt = 3000  &&  sal_amt < 5000)
{

          Disc = sal_amt * 0.05;             /* five percent  */
          printf ( “ The discount is %f “ , Disc ) ;

}
else
    printf ( “ The discount is 0”) ;
```

Example : Determining grade category

```
#include<stdio.h >
#include<string.h >
main()
{
  int marks;
  char grade [15];
  printf (" Enter the students marks  \n");
  scanf( "%d ",&marks ) ;
  if ( marks > =75  &&  marks <=100)
  {
    strcpy(grade, "Distinction");              /* Copy the string to the grade */
    printf("The grade is %s" , grade);
  }
  else if( marks > = 60 &&  marks < 75 )
  {
          strcpy(grade, "Credit");
    printf("The grade is %  s" , grade );
  }
  else if(marks>=50  &&  marks<60)
  {
   strcpy(grade, "Pass");
   printf("The grade is %  s" , grade );
  }
  else if (marks>=0 && marks<50)
  {
          strcpy(grade, "Fail");
    printf ("The grade is %  s" , grade)  ;
   }
   else
    printf("The mark is impossible!" );
   return 0;
}
```

The ‘switch’ and ‘break’ statements

The *switch - break* statements can be used in place of the *if - else* statements when there are several choices to be made. The switch structure is used to test if a variable equals some constant values then executes the equivalent statement.

The structure of a switch is as follows:

```
switch (integer expression)
{
 case constant 1:
        statement; optional
  case constant 2:
          statement; optional
            ………..

  default: (optional)
        statement; (optional)
}
```

Note:

(i) The switch labels (case labels) must be type int (including char) constants or constant expression.
(ii) You cannot use a variable for an expression for a label expression.
(iii) The expressions in the parenthesis should be one with an integer value. (again including type char)
Examples:

1. A program using switch statement that prompts for entry of a number and states the number entered.

Example: Demonstrating the ‘switch’ structure

```
#include<stdio.h>
main()
{
  int choice;
  printf(“Enter a number of your choice  ”);
  scanf(“ %d”, &choice);
  if (choice >=1 && choice <=9)              /* Range of choice values */
  switch (choice)
  {                                /* Begin of switch* /
        case 1: printf("\n You typed  1"); break;
        case 2:                          /* label 2* /
       printf("\n You typed 2");
       break;
        case 3:                              /* label 3* /
       printf("\n You typed 3");
       break;
         case 4:                     /* label 4* /
            printf( " \n You typed 4");
                    break;
         default:
           printf("There is no match in your choice");
 }                                  /* End of switch*/
 else
       printf("Your choice is out of range");
       return (0);
}                                  /* End of main*/
```

Explanation

The expression in the parenthesis following the switch is evaluated. In the example above, it has whatever value we entered as our choice.

Then the program scans a list of labels (case 1, case 2,…. case 4) until it finds one that matches the one that is in parenthesis following the switch statement.

If there is no match, the program moves to the line labeled default, otherwise the program proceeds to the statements following the switch.

The break statement causes the program to break out of the switch and skip to the next statement after the switch. Without the break statement, every statement from the matched label to the end of the switch will be processed.

For example if we remove all the break statements from the program and then run the program using the number 3 we will have the following exchange.

```
Enter a number of your choice 3
You typed 3
You typed 4
There is no match in your choice
```

Example: Demonstrating the ‘switch’ structure

```
#include <stdio.h>
main()
{
        char ch;
        printf(“Give me a letter of the alphabet \n”);
        printf(“An animal beginning  with letter”);
        printf (“is displayed \n “);
        scanf(“%c”, &ch);
        if (ch>=’a’ && ch<=’z’)            /*lowercase letters only */

        switch (ch)
        {                                        /*begin of switch*/
        case  `a`:
        printf(“Alligator , Australian aquatic animal \n”):
                    break;
        case  ‘b’:
                    printf(“Barbirusa, a wild pig of Malaysia \n”);
                    break;
        case ‘c’:
                    printf(“Coati, baboon like animal \n”);
                    break;
        case ‘d’:
                    printf(“Desman, aquatic mole-like creature \n”);
                    break;

default:
     printf(" That is a stumper! \n")
        }
     else
     printf("I only recognize lowercase letters.\n");
        return 0;
}          /* End of main  */
```

The 'continue' statement

Like the break statement the continue statement is a jump that interrupts the flow of a program. It is used in loops to cause the rest of an iteration to be skipped and the next iteration to be started.

If a break is used in a loop it quits the entire loop.

The 'goto' statement

It takes the form *goto labelname;*

Example

```
    goto part2;

    part2: printf("programming in c"\n";)
```

In principle you never need to use *goto* in a C statement. The *if* construct can be used in its place as shown below.

Alternative 1   **Alternative 2**

```
if (a>14)            if (a>14)
  goto a;                sheds=3;
 sheds=2;              else
 goto b;                 sheds=2;
a: sheds=3;             k=2*sheds;
b: k=2 * sheds;
```

Looping/Repetition Control Structure

In many programming problems a sequence of statements or in some cases the entire program may need to be executed repeatedly a definite or indefinite number of times. The repetition or iteration control structure is used to control this. In a finite loop the number of iterations is determined and set by the programmer. In an infinite loop the number of repetitions is dictated by a user or other factors.

C supports three loop versions:

- *while* loop
- *do while* loop
- *for* loop.

The 'while' loop

The while statement is used to carry out looping instructions where a group of instructions executed repeatedly until some conditions are satisfied. This is a pretest loop in that the test condition is placed before the statement block that is to be repeatedly executed. The computer evaluates the test condition statement and as long as it returns the Boolean value of true the statement block is executed then control returns to the test condition statement for re-evaluation. Repetition will terminate when the test condition statement returns false.

General form:

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

The statement will be executed as long as the expression is true, the statement can be a single or compound

```
/* counter.c */
/* Displays the digits 1 through 9 */
main()
{
  int digit=0;                  /* Initialisation */
        while (digit<=9)
        {
          printf("%d \n", digit);
        digit++;
        }
        return 0;
}
```

Example: Calculating the average of n numbers using a ‘while’ loop

Algorithm:

(i) Initialise an integer count variable to 1. It will be used as a loop counter.
(ii) Assign a value of 0 to the floating-point sum.
(iii) Read in the variable for n (number of values)
(iv) Carry out the following repeatedly (as long as the count is less or equal to n).
(v) Read in a number, say x.
(vi) Add the value of x to current value of sum.
(vii) Increase the value of count by 1.
(viii) Calculate the average: Divide the value of sum by n.
(ix) Write out the calculated value of average.

Solution

```
/* To add numbers and compute the average */
#include<stdio.h>
main()
{
        int n, count = 1;
        float x, average, sum=0.0;
        /* initialise and read in a value of n */
        printf(“How many numbers? “);
        scanf(“%d”, &n);

        /*Read in the number */
        while (count<=n)
        {
          printf(“x = “);
          scanf(“%f”, &x);
          sum+=x;
          count++;
        }
        /* Calculate the average and display the answer */
        average = sum/n;
        printf("\n The average is %f \n", average);
        return 0;
}
```

(Note that using the while loop, the loop test is carried out at the beginning of each loop pass).

The 'do .. While' loop (Post-Test)

In this structure the test condition is placed after the block of code that is to be repeatedly executed. The computer first executes the block of code then evaluates the test condition statement.

General form:

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

The statement (simple or compound) will be executed repeatedly as long as the value of the *expression* is true. (i.e. non zero).

Notice that since the test comes at the end, the loop body (statement) must be executed at least once.

Rewriting the program that counts from 0 to 9, using the *do while* loop:

```
/* counter1.c */
/* Displays the digits 1 through 9 */
main()
{
   int digit=0;          /* Initialisation */
        do
        {
           printf("%d \n", digit);
           digit++;
        } while (digit<=9);
        return 0;
}
```

Exercise: Rewrite the program that computes the average of n numbers using the do while loop**.**

The 'for' loop

This is the most commonly used looping statement in C.

General form:

```
  for (expression1;expression2;expression3)
        statement;
```

where:

expression1 is used to initialize some parameter (called an index). The index controls the loop action. It is usually an assignment operator.

expression2 is a test expression, usually comparing the initialised index in *expression1* to some maximum or minimum value.

expression3 is used to alter the value of the parameter index initially assigned by *expression* and is usually a unary expression or assignment operator);

Example

```
for (int k=0; k<=5; k++)
        printf(k = %d \n", k);
```

Output

```
0
1
2
3
4
5
```

Example: Counting 0 to 9 using a 'for' loop

```
/* Displays the digits 1 through 9 */
#include<stdio.h>
main()
{
        int digit;
```

```
            for(digit=0;digit<=9; digit++)
            printf("%d \n" , digit);
            return 0;
}
```

Example: Averaging a set of numbers using a 'for' loop

```
/* average.c */
/* To add numbers and compute the average */
#include<stdio.h>
main()
{
 int n, count;
 float x, average, sum=0.0;.

            /* initialise and read in a value of n */
            printf("How many numbers? ");
            scanf("%d", &n);
            /*Read in the number */
            for(count=1;count<=n;count++)
            {
               printf("x = ");
               scanf("%f", &x);
               sum+=x;
            }
            /* Calculate the average and display the answer */
            average = sum/n;
            printf("\n The average is %f \n", average);
            return 0;
}
```

Example: Table of cubes

```
/ Using a loop to make a table of cubes */
#include<stdio.h>
main()
{
        int number;
        printf("n       n cubed ");
        for(num=1; num<=6;num++)
        printf("%5d  %5d \n", num, num*num*num);
        return 0;
}
```

Also note the following points about the **for** structure.

- You can count down using the decrement operator
- You can count by any number you wish; two's threes, etc.
- You can test some condition other than the number of operators.
- A quantity can increase geometrically instead of arithmetically.

Nesting statements

It is possible to embed (place one inside another) control structures, particularly the if and for statements.

Nested 'if' statement

It is used whenever choosing a particular selection leads to an additional choice

Example

```
if (number>6)
        if (number<12)
          printf("You are very close to the target!");
else
          printf("Sorry, you lose!");
```

Nested 'for' statement

Suppose we want to calculate the average of several consecutive lists of numbers, if we know in advance how many lists are to be averaged.

Example: Nested 'for' statements

```
/* Calculate the averages of several different lists of number */
#include<stdio.h>
main()
{
        int n, count, loops, loopcount;
        float x, average, sum;
        /*Read in the number of loops */
        printf("How many lists? ");
        scanf("%d", &loops);
        /*Outer loop processes each list of numbers */
        for (loopcount=1; loopcount<=loops; loopcount++)
        {
          /* initialise sum and read in a value of n */
          sum=0.0;
        printf("List number %d \n How many numbers ? ",loopcount);
        scanf("%d", &n);
        /*Read in the numbers */
        for(count=1;count<=n; count++)
        {
          printf("x = ");
          scanf("%f", &x);
          sum+=x;
        }           /* End of inner loop */
        /* Calculate the average and display the answer */
        average = sum/n;
        printf("\n The average is %f \n", average);
}       /*End of outer loop */
```

```
        return 0;
}
```

NESTED LOOPS

C supports nesting of loops. **Nesting of loops** is the feature in C that allows the looping of statements inside another loop.

Any number of loops can be defined inside another loop, i.e., there is no restriction for defining any number of loops. The nesting level can be defined at n times. You can define any type of loop inside another loop; for example, you can define '**while**' loop inside a '**for**' loop.

The syntax for a **nested for loop** statement in C is as follows −

```
for ( init; condition; increment )
{
       for ( init; condition; increment )
       {
      statement(s);
       }
  statement(s);
}
```

The syntax for a **nested while loop** statement in C programming language is as follows −

```
while(condition)
{

       while(condition)
       {
        statement(s);
       }
  statement(s);
}
```

The syntax for a **nested do...while loop** statement in C programming language is as follows −

```
do {
   statement(s);

   do {
      statement(s);
   }while( condition );
}while( condition );
```

Examples

Example 1

```
#include <stdio.h>
int main()
{
 int n;// variable declaration
   printf("Enter the value of n :");
   scanf("%d", &n);
   // Displaying the n tables.
   for(int i=1;i<=n;i++)  // outer loop
   {
       for(int j=1;j<=10;j++)  // inner loop
       {
           printf("%d\t",(i*j)); // printing the value.
       }
       printf("\n");
   }
Return 0;
}
```

Explanation of the above code

- First, the 'i' variable is initialized to 1 and then program control passes to the i<=n.
- The program control checks whether the condition 'i<=n' is true or not.
- If the condition is true, then the program control passes to the inner loop.
- The inner loop will get executed until the condition is true.
- After the execution of the inner loop, the control moves back to the update of the outer loop, i.e., i++.
- After incrementing the value of the loop counter, the condition is checked again, i.e., i<=n.
- If the condition is true, then the inner loop will be executed again.
- This process will continue until the condition of the outer loop is true.

Example 2

This code prints prime numbers between 2-100 using nested for loop

```
#include <stdio.h>

int main () {

   /* local variable definition */
   int i, j;

   for(i = 2; i<100; i++) {

      for(j = 2; j <= (i/j); j++)
      if(!(i%j)) break; // if factor found, not prime
      if(j > (i/j)) printf("%d is prime\n", i);
   }

   return 0;
}
```

Example 3

```
#include <stdio.h>
int main()
{
  int rows;  // variable declaration
  int columns; // variable declaration
  int k=1; // variable initialization
  printf("Enter the number of rows :");  // input the number of rows.
  scanf("%d",&rows);
  printf("\nEnter the number of columns :"); // input the number of columns.
  scanf("%d",&columns);
    int a[rows][columns]; //2d array declaration
    int i=1;
  while(i<=rows) // outer loop
  {
     int j=1;
    while(j<=columns)  // inner loop
     {
         printf("%d\t",k);  // printing the value of k.
         k++;   // increment counter
         j++;
     }
     i++;
     printf("\n");
  }
}
```

Explanation of the above code.

- We have created the 2d array, i.e., int a[rows][columns].
- The program initializes the 'i' variable by 1.
- Now, control moves to the while loop, and this loop checks whether the condition is true, then the program control moves to the inner loop.
- After the execution of the inner loop, the control moves to the update of the outer loop, i.e., i++.
- After incrementing the value of 'i', the condition (i<=rows) is checked.
- If the condition is true, the control then again moves to the inner loop.
- This process continues until the condition of the outer loop is true.

Example 4 – Using nested loops to print a pattern

```
#include<stdio.h>

int main()

{

   printf("\n\n\t\tStudytonight - Best place to learn\n\n\n");

   printf("\n\nNested loops are usually used to print a pattern in c. \n\n");

   printf("\n\nThey are also used to print out the matrix using a 2 dimensional
array. \n\n");

   int i,j,k;

   printf("\n\nOutput of the nested loop is :\n\n");
```

```
    for(i = 0; i < 5; i++)
    {
       printf("\t\t\t\t");
       for(j = 0; j < 5; j++)
       printf("* ");


       printf("\n");
    }
    printf("\n\n\t\t\tCoding is Fun !\n\n\n");
    return 0;
}
```

Revision Exercises

1. A retail shop offers discounts to its customers according to the following rules:

   Purchase Amount >= Ksh. 10,000 - Give 10% discount on the amount.
   Ksh. 5, 000 <= Purchase Amount < Ksh. 10,000 - Give 5% discount on the amount.
   Ksh. 3, 000 <= Purchase Amount < Ksh. 5,000 - Give 3% discount on the amount.
   0 > Purchase Amount < Ksh. 3,000 - Pay full amount.

2. Write a program that asks for the customer's purchase amount, then uses *if* statements to recommend the appropriate payable amount. The program should cater for negative purchase amounts and display the payable amount in each case.
3. In what circumstance is the *continue* statement used in a C program?
4. Using a nested if statement, write a program that prompts the user for a number and then reports if the number is positive, zero or negative.
5. Write a *while* loop that will calculate th*e* sum of every fourth integer, beginning with the integer 3 (that is calculate the sum 3 + 7 +11 + 15 + ...) for all integers that are less than 30.

FUNCTIONS

Introduction

A function is a self-contained program segment that carries out some specific well - defined task. Every C program consists of one or more functions. One of these functions must be called main. Execution of the program will always begin by carrying out the instructions in main. Additional functions will be subordinate to main, and perhaps to one another.

If a program contains multiple functions, their definitions may appear in any order, though they must be independent of one another. That is, one function definition cannot be embedded within another. A function will carry out its intended action whenever it is accessed (whenever the function is called) from some other portion of the program. The same function can be accessed from several different places within a program. Once the function has carried out its intended action, control will be returned to the point from which the function was accessed.

Generally the function will process information that is passed to it from the calling portion of the program and return a single value. Information is passed to the function via special identifiers called *arguments* (also called parameters), and returned via the return statement. Some functions however, accept information but do not return anything.

Why Use Functions?

The use of programmer-defined functions allows a large program to be broken down to a number of smaller, self-contained components each of which has some unique identifiable purpose. Thus a C program can be modularized through the intelligent use of functions.

There are several advantages to this modular approach to program development; for example many programs require that a particular group of instructions be accessed repeatedly from several different places in the program. The repeated instructions can be placed within a single function which can then be accessed whenever it is needed. Moreover a different set of data can be transferred to the function each time it is accessed. Thus the use of a function ***eliminates the need for redundant programming of the same instructions.***

Equally important is the ***logical clarity*** resulting from the decomposition of a program into several concise functions where each function represents some well-defined part of the overall problem. Such programs are easier to write and easier to debug and their logical structure is more apparent than programs which lack this type of structure. This is especially true of lengthy, complicated programs. Most C programs are therefore modularized in this manner, even though they may not involve repeated execution of some task. In fact the decomposition of a program into individual program modules is generally considered to be good programming practice.

This use of functions also enables a programmer to build a ***customized library of frequently used routines*** or of routines containing system-dependent features. Each routine can be programmed as a separate function and stored within a special library file. If a program requires a particular routine, the corresponding library function can be accessed and attached to the program during the compilation process. Hence a single function can be utilized by many different programs.

Defining a Function

A function definition has two principle components; the first line (including the argument declaration), and the body of a function.

-The first line of a function definition contains the value returned by the function, followed by the function name, and (optionally) a set of arguments separated by commas and enclosed in parenthesis. Each argument is preceded by its associated type declaration. Any empty pair of parenthesis must follow the function name if the function definition does not include any arguments.

In general terms, the first line can be written as:
data-type ***functionname*** *(type 1 argument 1 , type 2 argument 2 ,……, type n argument n)*

Where data-type represents the data type of the item that is returned by the function.
functionname represents the function name , and type 1 , type 2 ,…….., type n represents the data types of the arguments, *argument 1, argument 2 ,……..argument n .*

The data types are assumed to be of type *int* if they are not shown explicitly. However, the omission of the data type is considered poor programming practice, even if the data items are integers.

The arguments are called **formal arguments** because they represent the names of data items that are transferred into the function from the calling portion of the program. They are also known as parameters or formal parameters. (The corresponding arguments in the function reference are called actual arguments since they define the data items that are actually transferred). The names of the formal arguments need not be the same as the names of the **actual arguments** in the calling portion of the program. Each formal argument must be of the same data type, however, as the data item it receives from the calling portion of the program.

-The remainder/second part of the function definition is a compound statement that defines the action to be taken by the function. This compound statement is sometimes referred to as the body of the function. Like any other compound statement, this statement can contain expressions statements, other compound statements, control statements and so on. It should include one or more return statements in order to return a value to the calling portion of the program. A function can access other functions. In fact it can access itself (a process known as *recursion).*

-Information is returned from the function to the calling portion of the program via a **return** statement. The return statement also causes the program logic to return to the point from which the function was accessed.
In general terms the return statement is written as:

```
Return (expression);
```

Only one expression can be included in the return statement. Thus, a ***function can return only one value to the calling portion via return.***

Accessing a Function

-A function can be accessed by specifying its name followed by a list of arguments enclosed in parenthesis and separated by commas. If the function call does not require any arguments an empty pair of parenthesis must follow the name of the function. The function call may be part of a simple expression (such as an assignment statement), or it may be one of the operands within an expression.

-The arguments appearing in the function are referred to as ***actual arguments*** in contrast to the formal arguments that appear in the first line of the function definition. (They are also known as actual parameters or arguments).

In a normal function call, there will be one actual argument for each formal argument. Each actual argument must be of the same data type as its corresponding formal argument. Remember that it is the value of each actual argument that is transferred into the function and assigned into the corresponding formal argument.

-There may be several different calls to the same function from various places within a program. The actual arguments may differ from one function call to another. Within each function call however the actual arguments must correspond to the formal arguments in the functions definition; i.e. the number of actual arguments must be the same as the number of formal arguments and each actual argument must be of the same data type as its corresponding formal argument.

Example 1: Factorial of an integer n

The factorial of a positive integer quantity *n* is defined as n! = 1 * 2 * 3 *…….* (n - 1) * n.
Thus, 2! = 1 * 2 = 2; 3! = 1 * 2 * 3 = 6; 4! = 1 * 2 * 3 * 4 = 24; and so on.

The function shown below calculates the factorial of a given positive integer n. The factorial is returned as a long integer quantity, since factorials grow in magnitude very rapidly as n increases.

```
long int factorial (int n) /*Calculate the factorial of n */
{
   int i;
   long int prod = 1;
   if (n >1 );
      for(i =2; i <=n; i++)
       prod * = i;
       return(prod);
 }
```

Notice the ***long int*** specification that is included in the first line of the function definition. The local variable ***prod*** is declared to be a long integer within the function. It is assigned an initial value of 1 though its value is recalculated within a for loop. The final value of ***prod*** which is returned by the function represents the desired value of n factorial (n!).

If the data type specified in the first line is inconsistent with the expression appearing in the return statement, the compiler will attempt to convert the quantity represented by the expression to the data type specified in the first line. This could result in a compilation error or it may involve a partial loss in data (due to truncation). Inconsistency of this type should be avoided at all costs.

Example 2: Factorial of an integer n (return type not specified)

The following definition is identical to that in **Example 1** except that the first line does not include a type specification for the value that is returned by the function.

```
factorial (int n)    /* calculate the factorial of n */
{
            int i;
            long int prod = 1;
    if (n > 1)
    for i =2; i < = n; i++)
      return (prod);
}
```

The function expects to return an ordinary integer quantity since there is no explicit type declaration in the first line of the function definition. However the quantity being returned is declared as a long integer within the function. This inconsistency can result in an error (some compilers will generate a diagnostic error and then stop without completing the compilation). The problem can be avoided however by adding a long int type declaration to the first line of the function definition as **Example 1** shows.

The keyword **void** can be used as a type specifier when defining a function that does not return anything or when the function definition does not include any arguments. The presence of this keyword is not mandatory but it is good programming practice to make use of this feature.

Consider a function that accepts two integer quantities, determines the larger of the two and displays it (the larger one). This function does not return anything to the calling portion. Therefore the function can be written as;

```
void maximum (int x, int y)
{
    int z;
    z = (x >= y)? x : y;
    printf("\n \n maximum value  = %d " , z);
}
```

The keyword **void** added to the first line indicates that the function does not return anything.

FUNCTION EXAMPLES:

Void function

This is a function that does not return a value to the function. Their definition starts with the word void:

```
e.g void OutputA()
       {
       Printf("This is output A\n");
}
```

```
#include <stdio.h>
//first function
Void OutputA()
{
Printf("This is output A\n");
}

//Second function
Void OutputB()
{
Printf("This is output B\n");
}
```

//The main function
Void main()
{
//Execute the above functions
OutputA();
OutputB();
}

-Non-void function

This is a function that returns a value to the function call. They can be defined using various data types like int.
e.g sqrt (x)

-User defined function

-Inbuilt function

Are functions that have been written and exist as library codes. They can be directly interpreted by the compiler. The inbuilt functions are normally embodied in certain header files e.g math.h
Examples include sqrt(x), sin(x), cos(x) and pow(x) these to write programs Use

Example 4: Determining the maximum of two integers (Complete program)

The following program determines the largest of three integers quantities. The program makes use of a function that determines the larger of two integer quantities. The overall strategy is to determine the larger of the first two quantities and then compare the value with the third quantity. The largest quantity is then displayed by the main part of the program.

```
/*Determine the largest of the three integer quantities*/
#include <stdio.h>
int maximum (int x, int y) /*Determine the larger of two quantities*/
{
   int z;
   z = (x > = y )? x : y;
   return(z);
}
main()
{
   int  a , b , c ,d;
   /*read the integer quantities*/
    printf("\n a = ");
    scanf("%d", &a);
    printf("\n b = " );
    scanf("%d", &b);
    printf("\n c =  ");
    scanf("%d", &c);
 /* Calculate and display the maximum value */
         d = maximum (a, b);
         printf ("\n \n  maximum = % d ", maximum (c ,d));
         return 0;
}
```

The function **maximum** is accessed from two different places in **main**. In the first call to maximum, the actual arguments are the variables **a** and **b** whereas in the second call, the arguments are c and d. (d is a temporary variable representing the maximum value of *a* and *b*).

Note the two statements in main that access maximum, i.e.

```
    d = maximum (a, b);
          printf(" \n \n maximum = %d ", maximum (c, d));
```

A single statement can replace these two statements, for example:

```
        printf (" \n\n maximum = %d " maximum(c, maximum (a, b)));
```

In this statement, we see that one of the calls to maximum is an argument for the other call. Thus the calls are embedded one within the other and the intermediary variable d is not required. Such embedded functions calls are permissible though their logic may be unclear. Hence they should generally be avoided by beginning programmers.

Function Prototypes

In the previous function examples, the programmer -defined function has always preceded main. Thus when the programs are compiled, the programmer-defined function will have been defined before the first function access. However many programmers prefer a top down approach in which main appears ahead of the programmer-defined function definition. In such a situation, the function access (within main) will precede the function definition. This can be confusing to the compiler unless the compiler is first alerted to the fact that the function being accessed will be defined later in the program. A function prototype is used for this purpose

Function prototypes are usually written at the beginning of a program ahead of any programmer-defined function (including main) .
The general form of a function prototype is;

data_type ***function_name*** *(type 1 argument 1, type 2 argument 2, ., type n argument n);*

Where *data_type* represents the type of the item that is returned by the function, *function_name* represents the name of the function, type 1, type 2, … …., type n represent the types of the arguments 1 to n.

Note that a function prototype resembles the first line of a function definition (although a definition prototype ends with a semicolon).
Function prototypes are not a must but are desirable however because they *further facilitate error checking between the calls to a function and the corresponding function definition.*

The names of the argument within the function prototype need not be declared else where in the program since these are “dummy” argument names that are recognised only within the prototype. In fact, the argument names can be omitted (though it is not a good idea to do so). However the arguments data types are essential.

In practice, the argument names usually included are often the same as the names of the actual arguments appearing in one of the function calls. The data types of the actual arguments must conform to the data types of the arguments within the prototype.

Function prototypes are not mandatory in C. They are desirable however because they *further facilitate error checking between the calls to a function and the corresponding function definition.*

Example 6: Factorial of an integer n

Here is a complete program to calculate the factorial of a positive integer quantity. The program utilises the function factorial defined in example 1 and 2. Note that the function definition precedes **main**.

1. /*Calculate the factorial of an integer quantity*/

#include <stdio.h>

```
long int factorial (int n);
main()
{
   int n;
   /* read in the integer quantity  */
   printf (“\n  n = “);
   scanf  (“%d “, &n);
   /* Calculate and display the factorial*/
  printf (“\n n =%\d”, factorial (n));
  return 0;
}
/*Calculate the factorial of n*/
long int factorial (int n)
{
  int  i;
  long int prod=1;
  if (n >1)
  for( i=2;  i<=n; i ++)
      prod *= i;
            return (prod);
}
```

2. *//a program to calc the square of a number using a fn prototype*

```
#include <stdio.h>
int square(int n);
main()
{
   int n, s;
   /* read in the integer quantity  */
   printf (“Enter n value\n “);
   scanf  (“%d “, &n);
   /* calculate and display the square*/
```

```
  S=square (n);
printf ("\n the square of n = %d", s);
  return 0;
}
/*Calculate the square of n*/
 int square (int n)
{
int  sqr;
sqr=n*n;
return (sqr);
}
```

The programmer-defined function makes use of an integer argument (n) and two local variables (an ordinary integer and a long integer). Since the function returns a long integer, the type declaration long int appears in the first line of the function definition.

Parameter Passing in C Functions

Parameter passing **is the mechanism used to pass** parameters **to a procedure (subroutine) or function.**
The most common methods are to pass the value of the actual parameter (*call by value***), or to pass the address of the memory location where the actual parameter is stored (***call by reference***). The passing by reference allows the procedure to change the value of the parameter, whereas the former method guarantees that the procedure will not change the value of the parameter.**

There are various types of parameter passing techniques available in programming languages later displayed below. These are based on how actually parameter is passed to called function (Function which is called) from calling function (Function which calls the other function also called Caller function). Given the program below:

```
float square ( float ) ;
main( )
{
```

```
float a, b ;
printf ( "\nEnter any number " ) ;
scanf ( "%f", &a ) ;
b = square ( a ) ;
printf ( "\nSquare of %f is %f", a, b ) ;
}
float square ( float x )
{
float y ;
y = x * x ;
return ( y ) ;
}
```

In above program main() calls the square() so main() is caller function and square() is called function.

1. **Call by value Parameter passing technique**

When we call the function, we pass parameter to called function and it is by default passed by the value of variable in calling function. The passing by value guarantees that the procedure will not change the value of the parameter. Generally we use call by value mechanism if we don't need the changes made in calling function to reflect in called function. E.g.

```
/* Call by value Parameter passing technique */
int calsum (int  x, int y , int z ) ;
main( )
{
int  a, b, c, sum ;
printf ( "\nEnter any three numbers " ) ;
scanf ( "%d %d %d", &a, &b, &c ) ;
sum = calsum ( a, b, c ) ;
printf ( "\nSum = %d", sum ) ;
}
calsum (int x, int y, int z )
{
int d ;
d = x + y + z ;
return ( d ) ;
}
```

If we give input 20,30,40, then output will be sum=90.The value of variable a,b,c in main are 20,30,40 on calling function calsum(a,b,c) as we are passing their values to variable x,y,z respectively. So value of variable x,y,z in function calsum() after is also 20,30,40 respectively.

```
/* Call by value Parameter passing technique */
Void exchange ( int x, int y );
main( )
{
int a = 10, b = 20 ;
printf ( "\na = %d b = %d", a, b ) ;
exchange( a, b ) ;
printf ( "\na = %d b = %d", a, b ) ;
}
exchange ( int x, int y )
{
int  t ;
t = x ;
x = y ;
y = t ;
printf ( "\nx = %d y = %d", x, y ) ;
}
```

Before calling exchange() value of a,b are 10,20 respectively. After calling exchange() x,y also receives 10,20 respectively . In exchange() we are interchanging value of x,y, So new value of x,y are 20 ,10 respectively. After successful execution of function exchange() control transfer to main function but in main function value of a,b remain unchanged i.e. 10,20 respectively.

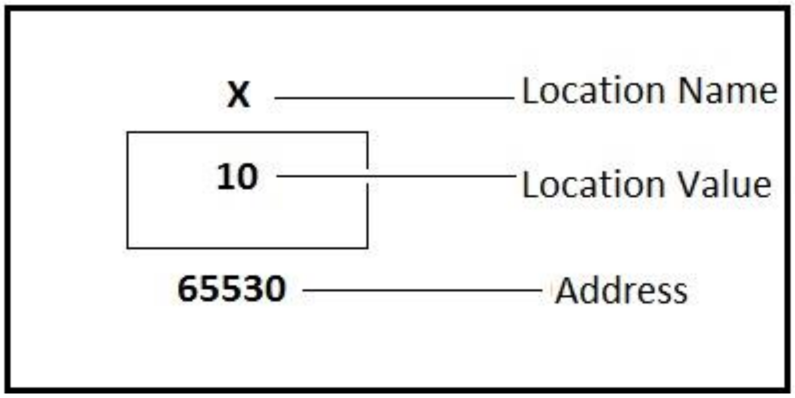
So output of above program is

```
a=10 b=20
x=20 y=10
a=10 b=20
```

2. **Call by reference Parameter passing technique**

This type of parameter passing technique is also called as call by address because we are passing the address of actual variable as an argument to called function instead of value of actual argument. Each variable has some fixed address which remains constant throughout execution of program in memory. Using this address of variable, it is also possible to access and modify the value of variable by using pointer. But if we required the changes to reflect in called function we use call by reference mechanism.

Consider declaration of variable..
int x= 10; this tells the compiler:
1) to allocate 2 byte (depending on type of compiler and type of variable) memory location on stack frame of that particular function in which it is declared.
2) to hold value of type integer and also associate this address with name x and
3) this memory location is initialized with value 3.



Memory Map

Selection of address for particular variable is compiler dependent and is always whole number. We can access value 10 by using variable name x in this case and by address 65530 also.
'&' is a special operator in c used as 'address of'.
Eg The expression &x in above case return address of variable x which happens to be 65530. As address is always positive, we use operator %u for unsigned integer.
'*' is special operator in c used as 'value at address' also called indirection operator.
E.g. *(&x) which means value at address (&x) i.e. value at address (65530) which happens to be value of variable x=30.

```
/* Call by Reference Parameter passing technique */


#include <stdio.h>

Void main( )
{
```

```
int  x = 10;
printf ( "\nAddress of x = %u", &x ) ;
printf ( "\nValue of x = %d", x ) ;
printf ( "\nValue of x = %d", *( &x ) ) ;
}
```

Output of program is

```
Address of x = 65530
Value of x = 10
Value of x = 10
```

In call by reference we pass the address of actual parameter in calling function and is copied in formal argument in called function .As we are passing address of actual parameter, changes made to formal argument would reflect in calling function.

```
/* Call by Reference Parameter passing technique */


#include<stdio.h>

Void exchange (a,b);

void main( )
{
int  a = 10, b = 20 ;
printf ( "\na = %d b = %d", a, b ) ;
exchange( &a, &b ) ;
printf ( "\na = %d b = %d", a, b ) ;
}

void exchange( int *x, int *y )
{
int  t ;
t = *x ;
*x = *y ;
*y = t ;
}
```

The output of above program is

```
a = 10 b = 20
a = 20 b = 10
```

Consider declaration of variable
int *x= 10; which means x is declared as pointer variables, i.e. variable is capable of holding address which is always positive whole number.
int * x does not mean that x is going to hold integer value rather it simply means it is going to hold address (memory location) of integer type of variable.
We are passing address of variables a, b to function exchange () so while receiving we are using two pointer to integer.

The Scope of Variables

Local variables:
Variable whose existence is known only to the main program or functions are called local variables. Local variables are declared with in the main program or a function.

Global variables:
Variables whose existence is known to the both main() as well as other functions are called global variables. Global variables are declared outside the main() and other functions.
The following Program illustrates the concept of both local as well as global variables.

Statement of C Program: This program does not accept anything from the keyboard. It initialises the variables a to 10 outside the main() function and value().

```
#include<stdio.h>
int a = 10;                     /* global declaration */
main()
{
int b;
printf("  = %d\n" , i);
b = value(i)
printf(" j = %d\n");
}                                          /* End of main() */
            /* Function to compute value */
int value(i)
int i;
{
int c;
c = i + 10;
return(c);
}                           /* End of Function */
```

Explanation:
The statement int **i = 10;** that appears before the main() is a global declaration. The value of **i** is accessed by the main program as well as the function value(). The variable **c** is local to the function value(). It has no existence in the function value(). The variable **b** is also local to the main() but has no scope in the function value().

Recursion

Recursion is the process by which a function calls itself repeatedly until a special condition is satisfied.

To use recursion, two conditions must be satisfied:
(i) The problem must be written in a recursive form.
(ii) There must be a stopping case (terminating condition).

Example

The factorial of any possible integer can be expressed as;
*n ! =n * (n-1) * (n-2) *………* 1.*
*e.g. 5 ! = 5 * 4 * 3 * 2 * 1.*

However we can rewrite the expression as; 5! = 5 * 4!

Or generally,

n! = n * (n –1)! (Recursive statement)

This is to say that in the factorial example, the calculation of n is expressed in form of a previous result (condition (i) is satisfied).

Secondly 1! = 1 by definition, therefore condition (ii) is satisfied.

Example: factorial in recursive form

```
#include<stdio.h>
long int factorial (int n);    /*factorial function  prototype*/
main()
{
  int n;
  /*Read in the integer quantity*/
  printf (“n = ” );
  scanf (“%d ”, &n);
```

```
 /*Calculate and display the factorial*/
  printf ("n! =%d \n",  factorial (n));
          return 0;
}
/* Function definition */
long int factorial (int n)
{
   if (n <=1)            /*terminating condition*/
        return (1);
   else
      return (n * factorial (n-1));
}
```

The functional factorial calls itself recursively with an actual argument that decreases in magnitude for each successive call. The recursive call terminates when the value of the actual argument becomes equal to 1.

Revision Exercises

Explain the meaning of each of the following function prototypes

- (i) **int f(int a);**
- (ii) **void f(long a, short b, unsigned c);**
- (iii) **char f(void);**

2. Each of the following is the first line of a function definition. Explain the meaning of each.
   - (i) **float f(float a, float b)**
   - (ii) **long f(long a)**
3. Write appropriate function prototypes for each of the following skeletal outlines shown below.

(a)
```
main()
{
        int a, b, c;
        ……
        c =function1(a,b);
        ……
        }
        int fucntion1(int x, int y)
        {
                int z;
                ……
                z = ……
                return(z);
        }
(b)     main()
    {
        int a;
        float b;
        long int c;
        ……
        c = funct1(a,b);
        ……
        }
        int func1(int x, float y)
        {
                long int z;
                ……
                ……
                z = ……
                return (z);
        }
```

4. Describe the output generated by the followed program.

```
#include<stdio.h>
int func(int count);
main()
{
    int a,count;
    for (count=1; count< = 10; count + +)
    {
        a = func(count);
         printf("%d",a);
    }
    return 0;
}
int func(int x)
{
        int y;
        y = x * x;
        return(y);
    }
```

5. (a) What is a recursive function?.
   (b) State two conditions that must be satisfied in order to solve a problem using recursion.

ARRAYS

Introduction

It is often necessary to store data items that have common characteristics in a form that supports convenient access and processing e.g. a set of marks for a known number of students, a list of prices, stock quantities, etc. Arrays provide this facility.

What Is An Array?

An array is a data structure that consists of a homogeneous ordered set of elements or a series of data objects of the same type stored sequentially. That is to say that an array has the following characteristics;

- Items share a name
- Items can be of any simple data type e.g. char, float, int, double.
- Individual elements are accessed using an integer index whose value ranges from 0 to the value of the array size.

Illustration examples:

a. age

An array of 10 student ages (stored as integers)

22	19	20	21	21	22	23	10	19	20

b. letters

An array of 5 characters in an employee's name

O	K	O	T	H

Declaring Arrays

An array definition comprises;

(i) Storage class (optional)
(ii) Data type
(iii) Array name
(iv) Arraysize expression (usually a positive integer or symbolic constant). This is enclosed in square brackets.

***Syntax*:**

Array_type array_name[maximum size];

Examples:

(i) int c [100];

int is the data type of the array (type of elements held), **c** is the array name
100 is the maximum number of elements (array size)

This declaration tells the compiler to create an array called c for storing 100 numbers of type integer.

The positions of values in an array are normally indicated using an index. The indices normally begin from 0 onwards e.g c[0], c[1]………

`(iii)` **Static char message[20];** A 20 character-type array called **message**. The individual array values persist within function calls (static).

`(iv)` **float debts [20];**
This statement declares debts as an array of 20 elements. The first element is called debts[0], the second debts [1], - - - - , debts[19] .

`(v)` Because the array is declared as type float, each element can be assigned a float value such as **debts[5] = 32.54**;

Other examples;
`(vi)` **int emp_no[15]; /*An array to hold 15 integer employee numbers */**
`(vii)` **char alpha [26]; /*an array to hold 26 characters */**

Array Dimensions and types

An array's dimension is the number of indices required to manipulate the elements of the array.

(i) A **one-dimensional** array requires a single index e.g. int numbers [10];
Resembles a single list of values and therefore requires a single index to vary between 0 to (array size -1).

(ii) **Multi dimensional arrays**
They are defined the same way as a one-dimensional array except that a separate pair of square brackets is required for each subscript. Thus a two-dimensional array will require two pairs of brackets, a three dimensional array will require three pairs of square brackets, etc.

Two – dimensional array

An m by n two-dimensional array can be thought of as a table of values having m rows and n columns. The number of elements can be known by the product of m (rows) and n(columns).

Examples of two-dimensional array declarations

float table[50][50];
char page[24][80];
Static double records[100][60][255];

Example

Two-dimensional array representing sales (‘000 tonnes for a product in four months for five years).

	Yr1	Yr2	Yr3	Yr4	Yr5
Month 1	23	21	27	23	22
Month 2	24	20	19	18	20
Month 3	26	23	26	29	24
Month 4	27	25	24	23	25

Arrays, like simple variables can be automatic, external or static.

Automatic array

An **automatic array** is one defined inside a function including formal arguments. C allows us to initialise automatic array variables as follows.

```
main()
{

        int marks[5] = [30, 40, 50, 90, 60];
        ---------
        ---------
}
```

Because the array is defined inside main, its an automatic array. The first element marks[0] is assigned the value of 30 , marks[1] as 40 and so on.

External array

An **external array** is one defined outside a function.
They

(i) are known to all functions following them in a file e.g. from above, both main ( ) and feed ( ) can use and modify the array SOWS.

(ii) Persist (retain values) as long as the program runs. Because they are not defined in any particular function, they don’t expire when a particular function terminates.

Have a look at the following example.

```
int SOWS [5] =  {12, 100, 8, 9 ,6};
main()
{

   ----------
   ----------

}
```

```
int feed(int n)
{

   ---------
   ---------

}
```

Static array

A **static array** is local to the function in which it is declared but like an external array, it retains its values between function calls and is inititialised to zero by default.

Example

```
int account(int n, int m)
{
        static int k[2] = {343, 332};
        ---------
        ---------
}
```

Initializing Arrays

Like other types of variables, you can give the elements of arrays initial values. This is accomplished by specifying a list of values the array elements will have. The general form of array initialisation for a one-dimensional array is shown below.

type array_name[size] = { value list };

The value list is a comma separated list of constants that are type compatible with the base type of the array. The first constant will be placed in the first position of the array, the second constant in the second position and so on. Note that a semi colon follows the }.

In the following example, a five – element integer array is initialised with the squares of the number 1 though 5.

```
int i[5] = {1, 4, 9, 16, 25};
```

This means that **i[0]** will have the value 1 and **i[4]** will have the value 25.

You can initialise character arrays in two ways. First, if the array is not holding a
null -terminated string, you simply specify each character using a comma separated list. For example, this initialises **a** with the letters 'A', 'B', and 'C'.

```
char a[3] = { 'A', 'B', 'C'};
```

If the character array is going to hold a string, you can initialise the array using a quoted string, as shown here.

```
char name[6] =  "Peter";
```

Notice that no curly braces surround the string. They are not used in this form of initialisation. Because strings in C must end with a null, you must make sure that the array you declare is long enough to include the null. This is why name is 6 characters long, even though "Peter" is only 5 characters. When a string constant is used, the compiler automatically supplies the null terminator.

Initializing multidimentional arrays

Multidimensional arrays are initialised the same way as one-dimensional ones.

For example, here the array **sqr** is initialised with the values 1 though 9, using row order.

```
int  sqr [3][3] = {
                        1,  2,  3,
                        4,  5,  6,
                        7,  8,  9
                  };
```

This initialisation causes **sqr[0][0]** to have the value 1, **sqr[0][1]** to contain 2, **sqr[0][2]** to contain 3, and so forth.

If you are initialising a one-dimensional array, you need not specify the size of the array, simply put nothing inside the square brackets. If you don't specify the size, the compiler simply counts the number of initialisation constants and uses that that value as the size of the array.

For example **int p[] = {1,2,4,8,16,32,64,128};**   causes the compiler to create an initialised array eight elements long.

Arrays that don't have their dimensions explicitly specified are called *unsized arrays*. An unsized array is useful because it is easier for you to change the size of the initialisation list without having to count it and then change the array dimension dimension. This helps avoid counting errors on long lists, which is especially important when initialising strings.

Here an unsized array is used to hold a prompting message.

```
char prompt[ ] = "Enter your name: ";
```

If at a later date, you wanted to change the prompt to "Enter your last name: " , you would not have to count the characters and then change the array size.

For multi dimensional arrays, you must specify all but the left dimension to allow C to index the array properly. In this way you may build tables of varying lengths with the compiler allocating enough storage for them automatically.

For example, the declaration of **sqr** as an unsized array is shown here.

```
        int sqr[][3] = {
                                    1,  2,  3,
                                    4,  5,  6,
                                    7,  8,  9
                                    };
```

The advantage to this declaration over the sized version is that tables may be lengthened or shortened without changing the array dimensions.

EXAMPLES OF IMPLEMENTATION IN PROGRAMS

1. A PROGRAM TO DISPLAY VALUES FROM AN ARRAY
2. A PROGRAM TO DO SIMPLE CALCULATIONS USING SELECTED ARRAY VALUES
3. APROGRAM TO SUM ALL THE VALUES OF AN ARRAY
4. INPUT VALUES TO AN ARRAY USING A LOOP AND DO CALCULATIONS

Example: Array that prints the number of days per month

```
#include<stdio.h>
#define MONTHS 12
int days [MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
main()
{
        int index;
        extern int days[ ];
        for (index=0; index <MONTHS;  index + + )
        printf( "Month  %d  has  %d  days. \n ", index+1,  days [index]);
        return 0;
}
```

Here is the output

```
                Month 1 has 31 days.
                Month 2 has 28 days.
                …
                ….
                Month 12 has 31 days.
```

Explanation

- By defining **days [ ]** outside the function, we make it external. We initialise it with a list enclosed in braces, commas are used to separate the members of the list.

- Inside the function the optional declaration extern int days [ ]; uses the keyword extern to remind us that days array is defined elsewhere in the program as an external array. Because it is defined elsewhere we need not give its size here. (ommitting it has no effect on how the program works)

Note:
The number of items in the list should match the size of the array.

Processing an array

Single operations which involve entire arrays are not permitted in C. Operations such as assignment, comparison operators, sorting etc must be carried out on an element-by-element basis. This is usually accomplished within a loop where each pass through the loop is used to process one array element. The number of passes through the loops will therefore be equal to the number of array elements to be processed.

Example: Calculating the average of n numbers

```
#include<stdio.h>
main()
{
        int n, count;
        float avg, d, sum =0.0;
        float list[100];
          /* Read in a value of n */
          printf(" \n How many numbers will be averaged ? ");
        scanf(" %d ", &n);
        printf(" \n");
          /* Read in the numbers */
        for (count = 0; count < n; count++)
        {
          printf(" i  = %d x = ", count+1);
          scanf(" %f ", &list[count]);
          sum+=list[count];
        }

        /* Calculate the average */
        avg = sum/n;
        printf("\n  The  average is %5.2f  \n\n ", avg);


        /* Calculate deviations from the average */
        for (count =0; count  < n; count ++)
        {
           d = list[count] – avg;
           printf(" I = %d   x = %5.2f , d = %5.2f ", count  + 1, list[count], d);
        }
        return 0;
}   /* End of program */
```

Output

```
How many numbers will be averaged ?    3
i = 1              x = 3
i = 2              x = - 4
i = 3              x = 7

The average is 2

i = 1              x = 3              d = 1
i = 2              x = - 4            d = - 6
i = 3              x =  7             d =  5
```

Exercise

Assuming that the number of values in the list is already known to be 3, and that the list values are 5, 6, 8, rewrite the above program without having to request input from the keyboard.

Example: Bubble sort

Arrays are especially useful when you want to sort information. For example, this program lets the user enter up to 100 numbers and then sorts them. The sorting algorithm is the **bubble sort**. The general concept of the bubble sort is the repeated comparisons and, if necessary exchanges of adjacent elements. This is a little, like bubbles in a tank of water with each bubble, in turn, seeking its own level.

The following code implements the bubble sort algorithm.

```
#include<stdio.h>
main()
{
        int item[100];
        int a, b, t;
        int count;
        / * Read in the numbers */
        printf(“ How many numbers?  “);
        scanf(“ %d “, &count);
        for (a = 0; a < count; a ++)
        scanf(“ %d”, &item[a]);
        /* Now sort them using a bubble sort */
        for(a  = 1; a <  count; + + a)
          for(b  = count –1; b > =a; - - b)
          {
                      /* Compare adjacent items */
                     if (item[b –1] > item[b])
                     /* exchange the elements */
            {
                              t = item[b – 1];
                              item[b –1] = item[b];
                              item[b] = t;
             }
          }
        /* Display sorted list  */
        for(t = 0; t < count; t++)
          printf(“ %d “, item[t]);
        return 0;
}
```

Example: Two – Dimensional array of scores

```
#include<stdio.h>
#define STUDENT 5     /* Set maximum number of students */
#define CATS 4        /* Set maximum number of cats */
main()
{
        /* Declare and initialize required variables and array */
        int rows, cols, SCORES[STUDENT][CATS];
        float cats_sum , stud_average, total_sum=0.0, average;
        printf("Entering the marks ..............\n\n");
        /* Read in  scores into the array */
          for(rows=0;rows<STUDENT; rows++) /* Outer student loop */
        {
   printf("\n Student % d\n", rows+1);
        cats_sum=0.0;    /* Initializes sum of a student's marks */
        for(cols=0;cols<CATS;cols++)  /* Inner loop for cats */
        {
          printf("CAT  %d\n",cols+1);
          scanf(" %d", &SCORES[rows][cols]);
          cats_sum + =SCORES[rows][cols];  /* Adjust sum of marks */
        }
stud_average=cats_sum/CATS; /*Calculate the average of each student */
printf("\n Total marks for student %d is %3.2f ",rows+1, cats_sum);
printf("\n Average score for the student is %3.2f ",stud_average);
total_sum+=cats_sum;           /* Adjust the class total marks */
}
average=total_sum/(STUDENT*CATS);   /* Compute the class average */
printf("\n Total sum of marks for the class is %3.2f\n ", total_sum);
printf("\n The class average is %3.2f\n ",average);
/*Printing the array elements */
for(rows=0;rows<STUDENT; rows++)
for(cols=0;cols<CATS;cols++)
{
        printf("\n Student %d, Cat %d ",rows+1, cols+1);
        printf("\n\t %d \n", SCORES[rows][cols]);
}
          return 0;
}
```

Strings

In C, one or more characters enclosed between double quotes is called a *string.* C has no built-in string data type. Instead, C supports strings using one dimensional character arrays. A string is defined as a *null terminated character array.* In C, a null is 0. This fact means that you must define the array is going to hold a string to be one byte larger then the largest string it is going to hold, to make room for the null.

To read a string from the keyboard you must use another of C's standard library functions, **gets( )**, which requires the **STDIO.H** header file. To use **gets( )**, call it using the name of a character array without any index. The **gets( )** function reads characters until you press **<ENTER>**. The carriage return is not stored, but it is replaced by a null, which terminates the string. For example, this program reads and writes a string entered at the keyboard.

```
#include<stdio.h>
main()
{
          char str[80];
          int i;
          printf( " Enter a string (less than 80 characters): \n");
          gets(str);
          for( i = 0 ; str[i]; i++)
          printf(" %c", str[i]);
          return 0;
}
```

The **gets( )** function performs no bounds checking, so it is possible for the user to enter more characters that **gets( )** is called with can hold. Therefore be sure to call it with an array large enough to hold the expected input.

In the previous program, the string that was entered by the user was output to the screen a character at a time. There is however a much easier way to display a string, using printf( ). Here is the previous program rewritten..

```
#include<stdio.h>

main()

{

        char str[80];

        printf( “ Enter a string (less than 80 characters): \n”);

        gets(str);

        printf(str);

        return 0;

}
```

If you wanted to output a new line, you could output **str** like this:

```
printf( “%s \n”, str);
```

This method uses the %s format specifier followed by the new line character and uses the array as a second argument to be matched by the %s specifier.

The C standard library supplies many string-related functions. The four most important are **strcpy( )**, **strcat( ), strcmp( )** and **strlen( )**. These functions require the header file STRING.H.

The strcpy( ) function has this general form.

strcpy(*to, from*);

It copies the contents of *from* to *to*. The contents of *from* are unchanged. For example, this fragment copies the string “hello’ into **str** and displays it on the screen.

```
        char str[80];

        strcpy(str, “hello”);

        printf(“%s”, str);
```

The **strcpy( )** function performs no bounds checking, so you just make sure that the array on the receiving end is large enough to hold what is being copied, including the null terminator.

The **strcat( )** function adds the contents of one string to another. This is called *concatenation*. Its general form is

```
strcat( to, from);
```

It adds the contents of *from* to *to*. It performs no bounds checking, so you must make sure *to* is large enough to hold its current contents plus what it will be receiving. This fragment displays **hello there.**

```
char str[80];
strcpy(str, “hello”);
strcat(str, “there”);
printf(str);
```

The **strcmp( )** function compares two strings. It takes this general form.

```
strcmp(s1, s2);
```

It returns 0 if the strings are the same. It returns less than 0 if *s1* is less than *s2* and greater than 0 if *s1* is greater than *s2.* The strings are compared lexicographically; that is in dictionary order. Therefore, a string is less than another when it would appear before the other in a dictionary. A string is greater than another when it would appear after the other. The comparison is not based upon the length of the string. Also, the comparison is case-sensitive, lowercase characters being greater than uppercase. This fragment prints 0, because the strings are the same.

```
printf( “ %d “, strcmp(“ one”, “one”));
```

The **strlen( )** function returns the length , in characters, of a string. Its general form is

```
strlen(str);
```

The **strlen( )** function does not count the null terminator.

Example: Demonstrating string functions

```
#include<string.h>
#include<stdio.h>
main()
{
  char str1[80], str2[80];
  int i;
  printf(“ Enter the first string: “);
  gets(str1);
  printf(“ Enter the second string: “);
  gets(str2);
```

```
        /* See how long the strings are */
        printf( " %s is %d characters long \n ", str1, strlen(str1));
        printf( " %s is %d characters long \n ", str2, strlen(str2));
        /* Compare the strings */
        i = strcmp(str1, str2);
        if ( ! i)
                printf("The strings are equal. \n");
        else if (i < 0)
                printf("%s is less than %s  \n", str1,str2);
        else
                printf("%s  is greater than %s  \n", str1,str2);
        /* Concatenate str2 to end of str1 if there is enough room */
        if (strlen(str1) + strlen(str2) < 80)
        {
                strcat(str1, str2);
                printf( "%s \n", str1);
        }
        /* copy str2 to str1 */
        strcpy(str1, str2);
        printf( "%s  %s \n", str1, str2);
        return 0;
}
```

Note:
You can use scanf( ) to read a string using the %s specifier, but you probably won't need to. Why? This is because when scanf( ) inputs a string, it stops reading that string when the first white space character is encountered. A white space character is a space, a tab, or a new line. This means that you cannot use scanf() to read input like the following string.

This is one string

Because there is a space after the word ***This***, **scanf( )** will stop inputting the string at that point. That is why gets( ) is generally used to input strings.

Revision Exercises

1. What is an array structure?
2. Give and explain the syntax of a two-dimensional array declaration.
3. In the course *Structured Programming using C,* the following percentage marks were recorded for six students in four continuous assessment tests.

	CAT 1	CAT 2	CAT 3	CAT 4
NANCY	90	34	76	45
JAMES	55	56	70	67
MARY	45	78	70	89
ALEX	89	65	56	90
MOSES	67	56	72	76
CAROL	70	90	68	56

If you were to implement the above table in a C program:

(a) Write a statement that would create the above table and initialize it with the given scores.

(b) Suppose the name of the above table was SCORES.

(i) What is the value of SCORES[2][3]?

(ii) What is the result of: (SCORES[3][3] % 11) *3?

(iii) Write a complete program that initializes the above values in the table, computes and displays the total mark and average scored by *each student*.

4. Show how to initialise an integer array called **items** with the values 1 through 10.
5. (i) Write a program that defines a 3 by 3 by 3 three dimensional array, and load it with the numbers 1 to 27.

   (ii) Have the program in (i) display the sum of the elements.

POINTERS

What Is A Pointer?

A pointer is a variable that holds the memory address of another variable. For example, if a variable called **p** contains the address of another variable called **q**, then p is said to point to q.

Therefore if q were at location 100 in memory, then p would have the value 100.

Pointer Declaration

To declare a pointer variable, use this general form:

*type *var_name;*

Here, ***type*** is the base type of the pointer. The base type specifies the type of the object that the pointer can point to. Notice that an asterisk precedes the variable name. This tells the computer that a pointer variable is being created. For example, the following statement creates a pointer to an integer.

```
int *p;
```

Pointer Operators

C contains two special pointer operators: ***** and **&**. The **&** operator returns the address of the variable it precedes. The * operator returns the value stored at the address that it precedes. The * pointer operator has no relationship to the multiplication operator, which uses the same symbol). For example, examine this short program.

```
#include <stdio.h>
main()
{
        int *p, q;
        q = 100;                /* assign q 100 */
        p = &q;                 /* assign p the address of q*/
        printf("%d", *p);/*  display q's value using pointer*/
        return 0;
}
```

This program prints **100** on the screen. Lets see why.

First, the line **int *p, q;** defines two variables: **p**, which is declared as an integer pointer, and **q**, which is an integer. Next, **q** is assigned the value **100**.

In the next line, **p** is assigned the address of **q**. You can verbalize the **&** operator as "address of." Therefore, this line can be read as: assign **p** the address of **q**. Finally, the value is displayed using the ***** operator applied to p. The * operator can be verbalized as "at address".

Therefore the printf( ) statement can be read as "print the value at address q," which is 100.

When a variable value is referenced through a pointer, the process is called indirection. It is possible to use the ***** operator on the left side of an assignment statement in order to assign a variable a new value using a pointer to it. For example, this program assigns a value **q** indirectly using the pointer **p.**

```
#include <stdio.h>
main()
{
  int *p, q;
  p = &q;                /* get q's address */
  *p = 199;              /* assign q a value using a pointer */
  printf("q's value is %d", q);
  return 0;
}
```

Note: The type of the variable and type of pointer must match.

<u>Operations Performed Using Pointers</u>

(a) **Assignment**

One can assign an address to a pointer by:

(i) Using an array name or
(ii) Using the address operator

From the previous example, p1 is assigned the address of the beginning of the array which is cell 234.

Dereferencing (value – finding)

The * operator gives the value pointed to.

From the previous example, p1 = 100 which is the value stored in location 234.

(c) **Take a pointer address**

Pointer variables have an address and a value. The & operator tells us where the pointer itself is stored.

From the previous example, p1 is stored in address 3606 whose value is 234.

Example: Demonstrating pointers

Program that uses a for loop that counts from 0 to 9. It puts in the numbers using a pointer.

```
#include <stdio.h>
main()
{
   int i,*p;
   p = &i;
   for ( i =0;  i <10; i++)
   printf (“ %d ”, *p);
   return 0;
}
```

Example : Further demonstration of pointers

```
#include<stdio.h>
main()
{
        int u1, u2;
        int v = 3;
        int *pv;
        u1 = 2 * ( v +  5 );
        pv  = &v;
        u2 = 2 * (*pv + 5 );
        printf(“ \n u1 = %d   u2  = %d ”,  u1, u2);
        return 0;
}
```

Output

```
u1 = 16,  u2 = 16.
```

Explain why.

Note

- Never use a pointer of one type to point to an object of a different type.

  For example:

  ```
  int q;
  float  *fp;
  fp = &q;      /* pointer fp assigned the address of an integer */
  fp = 100.23;              / address used for assignment */
  ```

- Do not use a pointer before it has been assigned the address of a variable. May cause program to crash.

  For example:

  ```
  main()
  {
      int  *p;
      *p =10;*/Incorrect since p is not pointing to anything */
        …
  }
  ```

The above is meaningless and dangerous.

STRUCTURES

Introduction

Suppose you want to write a program that keeps tracks of students [Name, Marks] i.e. a variety of information to be stored about each student. This requires;

- An array of strings (for the Names).
- Marks in an array of integers.

Keeping track of many related arrays can be problematic, for example in operations such as sorting all of the arrays using a certain format.

A data form or type containing both the strings and integers and somehow keep the information separate is required. Such a data type is called a ***structure***.

What Is A Structure?

A structure is an aggregate data type that is composed of two or more related elements.

Difference between Structures and Arrays

Unlike arrays, each element of a structure can have its own type, which may differ from the types of any other elements.

Setting Up A Structure

A template is the master plan describing how a structure is put together.

In our example, we can have the following template;

```
struct student
{
        char name[SIZE];
        int marks;
};
```

The above describes a structure made up of a character array **name**, and int variable **marks**.

Explanation

- The keyword *struct* announces to the computer that what follows is a structure data type template.

- The *tag* follows: which is a shorthand label that can be used later to refer to this structure. In the above example, the tag name is student.
- A list of structure members enclosed in a pair of braces. Each member is described by its own declaration. For example; the name portion is a character array of SIZE elements.

Note: Members of a structure can be any data types including other structures.

Example: Bank Account

Account number (integer)
Account type (character)
Account holder name [30]
Account balance (float)

```
 struct Bank_ account
 {
        int acc_number;
        char acc_type;
        char holder_name [30];
       float balance;
  };
```

Defining A Structure Variable

The structure template doesn't tell the computer how to represent the data. Having set up the template we can declare a structure variable as follows;

```
struct student  mystudent;
```

The computer creates a variable mystudent following the structure template. It allocates space for a character array of SIZE elements and for an integer variable.

One can declare as many variables of type **student** as possible. For example,
struct student mystudent, student1, x, y, z; *etc.*

Note*:* One can combine the process of defining the structure template and the process of defining a structure variable as follows;

```
struct student
{
          char name;
          int marks;
  }mystudent;
```

A general form of a structure can be given as follows;

```
struct tagname{
        type1 element1;
        type2 element2;
                .
                .
        typeN elementN;
        }variable list;
```

Initializing A Structure

A structure can be initialized like any other variable - external, static or automatic. This will depend on where the structure is defined.

Example

```
static struct student mystudent =
{
   "Fred Otieno",25;
};
```

Each member is given its own line of initialization and a comma separator, one member initialization from the next.

Accessing Structure Members

Individual structure members are accessed using a period (.) - structure member operator.

For example **mystudent.marks** which means the marks portion of struct mystudent.

Note: You can use mystudent.marks exactly the way you use other integer variables.
For example, you can use **gets(mystudent.name);**

or

scanf("%d", &mystudent.marks);

***Note**:* Although **mystudent** is a structure, **mystudent.marks** is an int type and is like any other integer variable. Therefore;

The use of *scanf("%d".........)* requires the address of an integer and that is what *&mystudent.marks does.*

If **stud1** is another student structure declared as follows;

struct student stud1;, then it is possible to read the name and marks into the variable using the statements;

```
gets(stud1.name)
scanf("%d", &stud1.marks);
```

Example: A student structure program

```
#include<stdio.h>
#define SIZE 40
struct student
{
        char name[SIZE];
        int marks;
};

main()
{
   struct student mystudent; /* declare mystudent as a student type */
  printf("Please enter the name of the student \n");
  scanf("%s", mystudent.name);
  printf("Enter the marks obtained \n");
  scanf("%d", &mystudent.marks);
  printf("%s:    got  %d ", mystudent.name, mystudent.marks);
  return 0;
}
```

Use of Structures

Where are structures useful?

The immediate application that comes to mind is database management. For example, to maintain data about employees in an organization, books in a library, items in a store, financial transactions in a company, etc.

Their use however, stretches beyond database management. They can be used for a variety of applications like:

- Checking the memory size of the computer.
- Hiding a file from a directory
- Displaying the directory of a disk.
- Interacting with the mouse
- Formatting a floppy
- Drawing any graphics shape on the screen.
- Changing the size of the cursor.

To program the above applications, you need thorough knowledge of internal details of the operating system

Unions

A union is a single memory location that stores two or more variables. Members within a union all share the same storage area, whereas each member within a structure is assigned its own unique storage area.

Thus, unions are used to conserve memory. They are useful for applications involving multiple members, where values need not be assigned to all members at a time.

The similarity between structures and unions is that both contain members whose individual data types may differ from one another.

Format of a union

```
union tag
{
        member 1;
        member 2;
        …..
        …..
        member n;
};
        Or

union tag
{
                member 1;
                member 2;
                …..
                …..
                member n;
}variable list;
```

Consider that a C program contains the following union declaration:

```
union id{
                    char color[12];
                    int size;
          }shirt, blouse;
```

Explanation

(i) There are two union variables **shirt** and **blouse**. Each variable can represent either a 2 character string (colour) or a integer quantity (size) at any one time. The 12-character string will require more storage area within the computer's memory than the integer quantity.

Therefore a block of memory large enough for the 12-character string will be allocated to each union variable.

(ii) A union may be a member of a structure and a structure may be a member of a union and may be freely mixed with arrays.

Also consider the following example.

```
union id{
                    char color[12];
                    int size;
          };

struct clothes
{
          char manufact[20];
          float cost;
          union id descr;
}shirt,blouse;
```

Explanation

shirt and **blouse** are structure variables of type **clothes**
Each variable will contain the following members;

- A string manufact
- A floating point quantity cost
- A union descr. The union may represent either a string (color) or an integer quantity (size).

An alternative declaration of the variables **shirt** and **blouse** is:

```
struct clothes
{
        char manufact[20];
        float cost;
        union
        {
                char color[12];
                int size;
        }descr;
};shirt, blouse;
```

Revision Exercises

1. How is a structure different from:
   (a) a union?
   (b) an array?
2. Give a suitable structure declaration for a bank account that should hold the details; Account number (int), Account type (character), account holder name (string), account balance (float)
3. Show how to create a structure called s_type that contains these five elements:

```
char ch;
float d;
int i;
char str[80];
double balance;
```

   Also define one variable called s_var using this structure.

5. Set up a suitable structure for an invoice that should hold the following details:

Element	**Type**
Invoice number	integer
Customer number	integer
Invoice date	structure (with three integer elements; day, month, year)
Customer address	string (20 characters)
Item	structure [with product code (integer), unit price (float) quantity (float) , amount (double)]
Invoice Total	double

6. How is a structure different from a union?

8. (a) Declare a structure **card** which stores the following library information.

Student Number (int)
Student Name (25 characters)
Course Code (int)
Telephone (12 characters)
Address (14 Characters)

FILE HANDLING (IN C LANGUAGE :)

What is a File?

Abstractly, a file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind. The collection of bytes may be interpreted, for example, as characters, words, lines, paragraphs and pages from a textual document; fields and records belonging to a database; or pixels from a graphicalv image. The meaning attached to a particular file is determined entirely by the data structures and operations used by a program to process the file. It is conceivable (and it sometimes happens) that a graphics file will be read and displayed by a program designed to process textual data. The result is that no meaningful output occurs (probably) and this is to be expected. A file is simply a machine decipherable storage media where programs and data are stored for machine usage.

Essentially there are two kinds of files that programmers deal with text files and binary files.

ASCII Text files

A text file can be a stream of characters that a computer can process sequentially. It is not only processed sequentially but only in forward direction. For this reason a text file is usually opened for only one kind of operation (reading, writing, or appending) at any given time.

Similarly, since text files only process characters, they can only read or write data one character at a time. (In C Programming Language, Functions are provided that deal with lines of text, but these still essentially process data one character at a time.) A text stream in C is a special kind of file. Depending on the requirements of the operating system, newline characters may be converted to or from carriage-return/linefeed combinations depending on whether data is being written to, or read from, the file. Other character conversions may also occur to satisfy the storage requirements of the operating system. These translations occur transparently and they occur because the programmer has signalled the intention to process a text file.

Binary files

A binary file is no different to a text file. It is a collection of bytes. In C Programming Language a byte and a character are equivalent. Hence a binary file is also referred to as a character stream, but there are two essential differences.

1. No special processing of the data occurs and each byte of data is transferred to or from the disk unprocessed.
2. C Programming Language places no constructs on the file, and it may be read from, or written to, in any manner chosen by the programmer.

Binary files can be either processed sequentially or, depending on the needs of the application, they can be processed using random access techniques. In C Programming Language, processing a file using random access techniques involves moving the current file position to an appropriate place in the file before reading or writing data. This indicates a second characteristic of binary files.
They a generally processed using read and write operations simultaneously.

For example, a database file will be created and processed as a binary file. A record update operation will involve locating the appropriate record, reading the record into memory, modifying it in some way, and finally writing the record back to disk at its appropriate location in the file. These kinds of operations are common to many binary files, but are rarely found in applications that process text files.

Creating a file and output some data

In order to create files we have to learn about File I/O i.e. how to write data into a file and how to read data from a file. We will start this section with an example of writing data to a file. We begin as before with the include statement for stdio.h, and then define some variables for use in the example including a rather strange looking new type.

```
/* Program to create a file and write some data the file */
#include <stdio.h>
main( )
{
     FILE *fp;
     char stuff[25];
     int index;
     fp = fopen("TENLINES.TXT","w"); /* open for writing */
     strcpy(stuff,"This is an example line.");
     for (index = 1; index <= 10; index++)
     fprintf(fp,"%s Line number %d\n", stuff, index);
     fclose(fp); /* close the file before ending program */
}
```

The type FILE is used for a file variable and is defined in the stdio.h file. It is used to define a file pointer for use in file operations. Before we can write to a file, we must open it. What this really means is that we must tell the system that we want to write to a file and what the file name is. We do this with the fopen() function illustrated in the first line of the program. The file pointer, fp in our case, points to the file and two arguments are required in the parentheses, the file name first, followed by the file type.

The file name is any valid DOS file name, and can be expressed in upper or lower case letters, or even mixed if you so desire. It is enclosed in double quotes. For this example we have chosen the name TENLINES.TXT. This file should not exist on your disk at this time. If you have a file with this name, you should change its name or move it because when we execute this program, its contents will be erased. If you donâ€™t have a file by this name, that is good because we will create one and put some data into it. You are permitted to include a directory with the file name. The directory must, of course, be a valid directory otherwise an error will occur. Also, because of the way C handles literal strings, the directory separation character â€˜\â€™ must be written twice. For example, if the file is to be stored in the \PROJECTS sub directory then the file name should be entered as “\\PROJECTS\\TENLINES.TXT”. The second parameter is the file attribute and can be any of three letters, r, w, or a, and must be lower case.

Reading (r)

When an r is used, the file is opened for reading, a w is used to indicate a file to be used for writing, and an indicates that you desire to append additional data to the data already in an existing file. Most C compilers have other file attributes available; check your Reference Manual for details. Using the r indicates that the file is assumed to be a text file. Opening a file for reading requires that the file already exist. If it does not exist, the file pointer will be set to NULL and can be checked by the program.

Here is a small program that reads a file and display its contents on screen.

```
/* Program to display the contents of a file on screen */
#include <stdio.h>
void main()
{
   FILE *fopen(), *fp;
   int c;
   fp = fopen("prog.c","r");
   c = getc(fp) ;
   while (c!= EOF)
   {
               putchar(c);
               c = getc(fp);
   }
   fclose(fp);
}
```

Writing (w)

When a file is opened for writing, it will be created if it does not already exist and it will be reset if it does, resulting in the deletion of any data already there. Using the w indicates that the file is assumed to be a text file.

Here is the program to create a file and write some data into the file.

```
#include <stdio.h>
int main()
{
 FILE *fp;
 file = fopen("file.txt","w");
 /*Create a file and add text*/
 fprintf(fp,"%s","This is just an example :)"); /*writes data to the file*/
 fclose(fp); /*done!*/
 return 0;
}
```

Appending (a)

When a file is opened for appending, it will be created if it does not already exist and it will be initially empty. If it does exist, the data input point will be positioned at the end of the present data so that any new data will be added to any data that already exists in the file. Using the a indicates that the file is assumed to be a text file.

Here is a program that will add text to a file which already exists and there is some text in the file.

```
#include <stdio.h>
int main()
{
    FILE *fp
    file = fopen("file.txt","a");
    fprintf(fp,"%s","This is just an example :)"); /*append some text*/
    fclose(fp);
    return 0;
}
```

Outputting to the file

The job of actually outputting to the file is nearly identical to the outputting we have already done to the standard output device. The only real differences are the new function names and the addition of the file pointer as one of the function arguments. In the example program, fprintf replaces our familiar printf function name, and the file pointer defined earlier is the first argument within the parentheses. The remainder of the statement looks like, and in fact is identical to, the printf statement.

Closing a file

To close a file you simply use the function fclose with the file pointer in the parentheses. Actually, in this simple program, it is not necessary to close the file because the system will close all open files before returning to DOS, but it is good programming practice for you to close all files in spite of the fact that they will be closed automatically, because that would act as a reminder to you of what files are open at the end of each program.

You can open a file for writing, close it, and reopen it for reading, then close it, and open it again for appending, etc. Each time you open it, you could use the same file pointer, or you could use a different one. The file pointer is simply a tool that you use to point to a file and you decide what file it will point to. Compile and run this program. When you run it, you will not get any output to the monitor because it doesnâ€™t generate any. After running it, look at your directory for a file named TENLINES.TXT and type it; that is where your output will be. Compare the output with that specified in the program; they should agree! Do not erase the file named TENLINES.TXT yet; we will use it in
some of the other examples in this section.

Reading from a text file

Now for our first program that reads from a file. This program begins with the familiar include, some data definitions, and the file opening statement which should require no explanation except for the fact that an r is used here because we want to read it.

```
#include <stdio.h>
   main( )
   {
     FILE *fp;
     char c;
     funny = fopen("TENLINES.TXT", "r");
     if (fp == NULL)
               printf("File doesn't exist\n");
     else {
      do {
       c = getc(fp); /* get one character from the file
       */
         putchar(c); /* display it on the monitor
       */
       } while (c != EOF); /* repeat until EOF (end of file)
     */
}
    fclose(fp);
   }
```

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. At this point, we have the potential for one of the most common and most perplexing problems of programming in C. The variable returned from the getc function is a character, so we can use a char variable for this purpose. There is a problem that could develop here if we happened to use an unsigned char however, because C usually returns a minus one for an EOF – which an unsigned char type variable is not capable of containing. An unsigned char type variable can only have the values of zero to 255, so it will return a 255 for a minus one in C. This is a very frustrating problem to try to find. The program can never find the EOF and will therefore never terminate the loop. This is easy to prevent: always have a char or int type variable for use in returning an EOF. There is another problem with this program but we will worry about it when we get to the next program and solve it with the one following that.

After you compile and run this program and are satisfied with the results, it would be a good exercise to change the name of TENLINES.TXT and run the program again to see that the NULL test actually works as stated. Be sure to change the name back because we are still not finished with TENLINES.TXT.

Others programs:

A PROGRAM THAT CONVERTS STRINGS FROM LOWERCASE ANS UPPERCASE

1.
```
#include <stdio.h>
#include<string.h>
int main()
{
  char str[20];
  int i;
  printf("Enter any string->");
  scanf("%s",str);
  printf("The string is->%s",str);
  for(i=0;i<=strlen(str);i++){
            if(str[i]>=97&&str[i]<=122)
            str[i]=str[i]-32;
  }
  printf("\nThe string in lowercase is->%s", str);
  return 0;
}
```

2.
```
#include <stdio.h>

int main(void) {
    putchar(lower('A'));

}

lower(a)
int a;
{
    if ((a >= 65) && (a >= 90))
        a = a + 32;
    return a;
}
```

A PROGRAM THAT CONVERTS STRINGS FROM UPPERCASE TO LOWERCASE

```
#include<stdio.h>
#include<string.h>
int main()
{
  char str[20];
  int i;
  printf("Enter any string->");
  scanf("%s", str);
  printf("The string is->%s", str);

  for(i=0;i<=strlen(str);i++)
{
      if(str[i]>=65&&str[i]<=90)
       str[i]=str[i]+32;
  }

  printf("\nThe string in lower case is->%s",str);
  return 0;
}
```

A C PROGRAM TO FIND THE LENGTH OF STRING

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

int main()
{
   char a[100];
   int length;

   printf("Enter a string to calculate it's length\n");
   gets(a);

   length = strlen(a);

   printf("Length of entered string is = %d\n",length);

   return 0;
}
```

BINARY TREES (PPT)