UNIT NAME: STRUCTURED PROGRAMMING.

**TOPIC 1 : INTRODUCTION TO STRUCTURED PROGRAMMING.**

**Programming**

Act of writing sets of rules and instructions that guides computer on how to perform a given task.

**Definitions**

**Structured programming** – a technique for organizing and coding computer programs in which a hierarchy of modules are used, each having a single entry and a single exit point, and in which control is passed downward through the structure without unconditional branches to higher levels of the structure.

**Computer hardware** - refers to the physical parts of a computer and related devices. Internal hardware devices include [motherboards](https://techterms.com/definition/motherboard), [hard drives](https://techterms.com/definition/harddrive), and [RAM](https://techterms.com/definition/ram). External hardware devices include [monitors](https://techterms.com/definition/monitor), keyboards, mice, printers, and scanners.

The internal hardware parts of a computer are often referred to as [components](https://techterms.com/definition/component), while external hardware devices are usually called [peripherals](https://techterms.com/definition/peripheral). Together, they all fall under the category of computer hardware. [Software](https://techterms.com/definition/software), on the other hand, consists of the programs and applications that run on computers. Because software runs on computer hardware, software programs often have [system requirements](https://techterms.com/definition/systemrequirements) that list the minimum hardware required for the software to run.

**Software** - refers to the set of electronic program instructions or data a computer processor reads in order to perform a task or operation. In contrast, the term 'hardware' refers to the physical components that you can see and touch, such as the computer hard drive, mouse, and keyboard.

Software can be categorized according to what it is designed to accomplish. There are two main types of software: systems software and application software.

**Systems Software**

Systems software includes the programs that are dedicated to managing the computer itself, such as the operating system, file management utilities, and disk operating system (or DOS). The operating system manages the computer hardware resources in addition to applications and data. Without systems software installed in our computers we would have to type the instructions for everything we wanted the computer to do.

**Applications Software**

Application software, or simply applications, are often called productivity programs or end-user programs because they enable the user to complete tasks, such as creating documents, spreadsheets, databases and publications, doing online research, sending email, designing graphics, running businesses.

**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.

**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.

**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.

**Programmer**

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

**Programming Languages**

A programming language is a tool for developing executable models for a class of problem domains.

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.

**Types of Structured Programming Language.**

**Pascal -** A high-level [programming language](http://www.webopedia.com/TERM/P/programming_language.html) developed by Niklaus Wirth in the late 1960s. The [language](http://www.webopedia.com/TERM/L/language.html) is named after Blaise Pascal, a seventeenth-century French mathematician who constructed one of the first mechanical adding machines. Pascal is best known for its affinity to structured [programming](http://www.webopedia.com/TERM/P/program.html) techniques. The nature of the language forces [programmers](http://www.webopedia.com/TERM/P/programmer.html) to design programs methodically and carefully. For this reason, it is a popular teaching language.

Despite its success in academia, Pascal has had only modest success in the business world. Part of the resistance to Pascal by professional programmers stems from its inflexibility and lack of tools for developing large [applications](http://www.webopedia.com/TERM/A/application.html). To address some of these criticisms, Wirth designed a new language called [Modula-2](http://www.webopedia.com/TERM/M/Modula_2.html). Modula-2 is similar to Pascal in many respects, but it contains additional [features](http://www.webopedia.com/TERM/F/feature.html).

**C** - C is a high-level and general-purpose programming language that is ideal for developing firmware or portable applications. Originally intended for writing system software, C was developed at Bell Labs by Dennis Ritchie for the Unix Operating System (OS) in the early 1970s.

Ranked among the most widely used languages, C has a compiler for most computer systems and has influenced many popular languages – notably C++

C is a structured, procedural programming language that has been widely used both for operating systems and applications and that has had a wide following in the academic community. Many versions of [UNIX](http://searchenterpriselinux.techtarget.com/definition/Unix)-based operating systems are written in C. C has been standardized as part of the Portable Operating System Interface ([POSIX](http://searchenterpriselinux.techtarget.com/definition/POSIX)).

**Fortran -** Fortran is a [programming language](https://en.wikipedia.org/wiki/programming_language) mainly used by the scientific community. Its name is a contraction of FORmula TRANslation, and its aim is to provide a way to tell computers to calculate complicated mathematical expressions, with more ease than assembly language. FORTRAN is one of the earliest programming languages. The original versions used [punched cards](https://en.wikipedia.org/wiki/punched_cards) to write programs with. FORTRAN's age is both a strength and a weakness. On one hand, FORTRAN has a huge number of [libraries](https://en.wikipedia.org/wiki/programming_library) of code available. However, Fortran also has many archaic features, especially in the earlier versions.

FORTRAN was designed for scientists and engineers, and has dominated this field. For the past 30 years FORTRAN has been used for such projects as the design of bridges and aeroplane structures, it is used for factory automation control, for storm drainage design, analysis of scientific data and so on. Throughout the life of this language, groups of users have written libraries of useful standard FORTRAN programs.

**COBOL** - It is [imperative](https://en.wikipedia.org/wiki/Imperative_programming), [procedural](https://en.wikipedia.org/wiki/Procedural_programming) and, since 2002, [object-oriented](https://en.wikipedia.org/wiki/Object-oriented_programming). COBOL is primarily used in business, finance, and administrative systems for companies and governments. COBOL is still widely used in legacy applications deployed on [mainframe computers](https://en.wikipedia.org/wiki/Mainframe_computer), such as large-scale [batch](https://en.wikipedia.org/wiki/Batch_processing) and [transaction processing](https://en.wikipedia.org/wiki/Transaction_processing) jobs. But due to its declining popularity and the retirement of experienced COBOL programmers, programs are being migrated to new platforms, rewritten in modern languages or replaced with software packages. Most programming in COBOL is now purely to maintain existing applications

**LOGO** - The Logo Programming Language, a dialect of Lisp, was designed as a tool for learning. Its features - interactivity, modularity, extensibility, flexibility of data types - follow from this goal. Although there are some versions of Logo that compile, it is generally implemented as an interpreted language. The interactivity of this approach provides the user with immediate feedback on individual instructions, thus aiding in the debugging and learning process. Error messages are descriptive. For example

**LISP** - Lisp was originally created as a practical [mathematical notation](https://en.wikipedia.org/wiki/Mathematical_notation) for [computer programs](https://en.wikipedia.org/wiki/Computer_programs), influenced by the notation of [Alonzo Church](https://en.wikipedia.org/wiki/Alonzo_Church)'s [lambda calculus](https://en.wikipedia.org/wiki/Lambda_calculus). It quickly became the favored programming language for [artificial intelligence](https://en.wikipedia.org/wiki/Artificial_intelligence) (AI) research. As one of the earliest programming languages, Lisp pioneered many ideas in [computer science](https://en.wikipedia.org/wiki/Computer_science), including [tree data structures](https://en.wikipedia.org/wiki/Tree_data_structure), [automatic storage management](https://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29), [dynamic typing](https://en.wikipedia.org/wiki/Dynamic_typing), [conditionals](https://en.wikipedia.org/wiki/Conditional_%28computer_programming%29), [higher-order functions](https://en.wikipedia.org/wiki/Higher-order_function), [recursion](https://en.wikipedia.org/wiki/Recursion_%28computer_science%29), the [self-hosting](https://en.wikipedia.org/wiki/Self-hosting) [compiler](https://en.wikipedia.org/wiki/Compiler),[[6]](https://en.wikipedia.org/wiki/Lisp_%28programming_language%29#cite_note-Graham-6) [closures](https://en.wikipedia.org/wiki/Closure_%28computer_programming%29), and the [read–eval–print loop](https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop)

**History of Programming Languages**

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/1GLS)**

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 conversion 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 (non portable) 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/2GLS)**

This language was more user oriented than machine language. Instructions are represented using mnemonic code and symbolic addresses. Words like add, sum,SUB 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/non-portable-Cannot be transferred to other machine
* Execution took longer than machine language programs due to the translation process.

NB

-Translation is done by using assembler.

**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/ 3GLS)
* Non-Procedural Languages (Fourth Generation Languages or 4GLs)

**Procedural languages (Third Generation).**

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 (Fourth Generation Languages or 4GLs).**

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.

**Fifth-generation programming language**

A fifth generation (programming) language (5GL) is a grouping of programming languages build on the premise that a problem can be solved, and an application built to solve it, by providing constraints to the program (constraint-based programming), rather than specifying algorithmically how the problem is to be solved (imperative programming).   
  
In essence, the programming language is used to denote the properties, or logic, of a solution, rather than how it is reached. Most constraint-based and logic programming languages are 5GLs. A common misconception about 5GLs pertains to the practice of some 4GL vendors to denote their products as 5GLs, when in essence the products are evolved and enhanced 4GL tools.

**PROGRAMMING PARADIGMS.**

Programming paradigms are a way to classify [programming languages](https://en.wikipedia.org/wiki/Programming_language) according to the style of [computer programming](https://en.wikipedia.org/wiki/Computer_programming). Features of various programming languages determine which programming paradigms they belong to; as a result, some languages fall into only one paradigm, while others fall into multiple paradigms. Some paradigms are concerned mainly with implications for the [execution model](https://en.wikipedia.org/wiki/Execution_model) of the language, such as allowing [side effects](https://en.wikipedia.org/wiki/Side_effect_%28computer_science%29), or whether the sequence of operations is defined by the execution model. Other paradigms are concerned mainly with the way that code is organized, such as grouping code into units along with the state that is modified by the code. Yet others are concerned mainly with the style of syntax and grammar. A programming paradigm is an approach to solving programming problems.

**1) Un structured programming.**

An unstructured program is a procedural program – the statements are executed in sequence as written. But this type of programming uses the goto statement. A goto statement allows control to be passed to any other place in the program. When a goto statement is executed, the sequence continues from the target of the goto. Thus to understand how a program works, you have to pretend to execute it. This means that it is often difficult to understand the logic of such a program. Some program compilers cross-index where a goto connects to, making it practical to rapidly navigate the source code. However, it was a common practice in some programming languages to use a variable in association with where the goto goes, making automated indexing impractical. There are similar problems in some structured programming languages, such as how foreign language views are implemented, to permit many people to view the same computer data, in their human language.

**2)** [**Structured**](http://c2.com/cgi/wiki?ImperativeProgramming) **programming**

Structured programming (sometimes known as modular programming) is a subset of procedural programming that enforces a logical structure on the program being written to make it more efficient and easier to understand and modify. Certain languages such as [Ada](http://whatis.techtarget.com/definition/Ada), [Pascal](http://searchenterpriselinux.techtarget.com/definition/Pascal), and dBASE are designed with features that encourage or enforce a logical program structure.

A technique for organizing and [coding](https://www.its.bldrdoc.gov/fs-1037/dir-007/_1049.htm) [computer](https://www.its.bldrdoc.gov/fs-1037/dir-008/_1184.htm) programs in which a hierarchy of modules is used, each having a single entry and a single exit point, and in which control is passed downward through the structure without unconditional branches to higher levels of the structure. Three types of control flow are used: sequential, test, and iteration.

Differences between unstructured and structured

|  |  |
| --- | --- |
| Unstructured / procedural | structured |
| 1. Use goto statement | 1. Don’t use goto statement |
| 1. The code is considered as one block | 2.Divide the code into codes called modules |
| 1. Based program are not easy to read | 1. Based program are easy to read |
| 1. Code is not efficient and not easy to understand | 4. Code is efficient and easy to understand |
| 1. Difficult to maintain code (modify code) | 5. Easy to maintain code |

**3)** [**Object**](http://c2.com/cgi/wiki?LogicProgramming) **Oriented Programming**

Object Oriented Programming (OOP) is a paradigm in which real-world objects are each viewed as seperate entities having their own state which is modified only by built in procedures, called methods. Because objects operate independently, they are encapsulated into modules which contain both local environments and methods. Communication with an object is done by message passing.

Object-oriented programming (OOP) is a programming language model organized around [objects](http://searchsoa.techtarget.com/definition/object) rather than "actions" and data rather than logic. Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data.

**4) Visual Programming.**

In [computing](https://en.wikipedia.org/wiki/Computing), a visual programming language (VPL) is any [programming language](https://en.wikipedia.org/wiki/Programming_language) that lets users create [programs](https://en.wikipedia.org/wiki/Computer_program) by manipulating program elements graphically rather than by specifying them textually. A VPL allows programming with visual expressions, spatial arrangements of text and graphic symbols, used either as elements of [syntax](https://en.wikipedia.org/wiki/Syntax) or [secondary notation](https://en.wikipedia.org/wiki/Secondary_notation). For example, many VPLs (known as dataflow or diagrammatic programming) are based on the idea of "boxes and arrows", where boxes or other screen objects are treated as entities, connected by arrows, lines or arcs which represent relations.

**5) Internet Based programming.**

Web programming refers to the writing, markup and coding involved in Web development, which includes Web content, Web client and server scripting and network security. The most common languages used for Web programming are XML, HTML, JavaScript, Perl 5 and PHP. Web programming is different from just programming, which requires interdisciplinary knowledge on the application area, client and server scripting, and database technology.

Web programming can be briefly categorized into client and server coding. The client side needs programming related to accessing data from users and providing information. It also needs to ensure there are enough plug ins to enrich user experience in a graphic user interface, including security measures.

* To improve user experience and related functionalities on the client side, JavaScript is usually used. It is an excellent client-side platform for designing and implementing Web applications.
* HTML5 and CSS3 supports most of the client-side functionality provided by other application frameworks.

**Monolithic programming**

Describe as a single tiered application in which the user interface and data caccess code are combined into a single program.

* Style that consist of sequential codes.
* Does not support subroutine concepts.
* Program flow is achieved by jumps

Disadvantages

1. Programs are large and complex
2. Debugging and testing is difficult
3. Program maintenance is difficult

**Modular programming**

Program is subdivided into small sections(subprograms/ subroutines) called modules

Advantages

1. Easy to code and understand
2. Debugging & maintenance becomes easy
3. Saves programmers time
4. Code can be reused

* **Hardware and Software Considerations for Structured Programming.**

- Structured programs written for one system can be run with little or no modification on other systems.

- There are several Pascal compilers and interpreters available for general use. Among these are:

 Turbo Pascal: provides an IDE and compiler for running Pascal programs on CP/M, CP/M-86, DOS, Windows and Macintosh.

 Delphi: provides compilers for running Object Pascal and generates native code for 32- and 64-bit Windows operating systems, as well as 32-bit Mac OS X and iOS. Embarcadero is planning to build support for the Linux and Android operating system.

 Free Pascal: it is a free compiler for running Pascal and Object Pascal programs. Free Pascal compiler is a 32- and 64-bit Turbo Pascal and Delphi compatible Pascal compiler for Linux, Windows, OS/2, FreeBSD, Mac OS X, DOS and several other platforms.

 Turbo51: it is a free Pascal compiler for the 8051 family of microcontrollers, with Turbo Pascal 7 syntax.

 Oxygene: it is an Object Pascal compiler for the .NET and Mono platforms.

 GNU Pascal (GPC): it is a Pascal compiler composed of a front end to GNU Compiler Collection.

- The source code written in source file is the human readable source for your program. It needs to be "compiled", to turn into machine language so that your CPU can actually execute the program as per instructions given.

This C programming language compiler will be used to compile your source code into final executable program. Most frequently used and free available compiler is GNU C/C++ compiler.

**Hardware requirements.**

### Architecture

All computer [operating systems](https://en.wikipedia.org/wiki/Operating_system) are designed for a particular [computer architecture](https://en.wikipedia.org/wiki/Computer_architecture). Most software applications are limited to particular operating systems running on particular architectures. Although architecture-independent operating systems and applications exist, most need to be recompiled to run on a new architecture.

### Processing power

The power of the [central processing unit](https://en.wikipedia.org/wiki/Central_processing_unit) (CPU) is a fundamental system requirement for any software. Most software running on [x86 architecture](https://en.wikipedia.org/wiki/X86_architecture) define processing power as the [model](https://en.wikipedia.org/wiki/List_of_microprocessors) and the [clock speed](https://en.wikipedia.org/wiki/Clock_rate) of the CPU. Many other features of a CPU that influence its speed and power, like [bus speed](https://en.wikipedia.org/wiki/Front_side_bus), [cache](https://en.wikipedia.org/wiki/CPU_cache), and [MIPS](https://en.wikipedia.org/wiki/Instructions_per_second) are often ignored. This definition of power is often erroneous, as [AMD](https://en.wikipedia.org/wiki/Advanced_Micro_Devices) [Athlon](https://en.wikipedia.org/wiki/Athlon) and [Intel](https://en.wikipedia.org/wiki/Intel) [Pentium](https://en.wikipedia.org/wiki/Pentium_%28brand%29) CPUs at similar clock speed often have different throughput speeds. Intel [Pentium](https://en.wikipedia.org/wiki/Pentium_%28brand%29) CPUs have enjoyed a considerable degree of popularity, and are often mentioned in this category[[citation needed](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed" \o "Wikipedia:Citation needed)].

### Memory

All software, when run, resides in the [random access memory](https://en.wikipedia.org/wiki/Random_access_memory) (RAM) of a computer. Memory requirements are defined after considering demands of the application, operating system, supporting software and files, and other running processes. Optimal performance of other unrelated software running on a multi-tasking computer system is also considered when defining this requirement.

### Secondary storage

Hard-disk requirements vary, depending on the size of software installation, temporary files created and maintained while installing or running the software, and possible use of [swap space](https://en.wikipedia.org/wiki/Virtual_memory) (if RAM is insufficient).

### Display adapter

Software requiring a better than average [computer graphics](https://en.wikipedia.org/wiki/Computer_graphics) display, like [graphics editors](https://en.wikipedia.org/wiki/Graphics_software) and high-end [games](https://en.wikipedia.org/wiki/Video_game), often define high-end [display adapters](https://en.wikipedia.org/wiki/Graphics_processing_unit) in the system requirements.

### Peripherals

Some software applications need to make extensive and/or special use of some [peripherals](https://en.wikipedia.org/wiki/Peripheral), demanding the higher performance or functionality of such peripherals. Such peripherals include [CD-ROM drives](https://en.wikipedia.org/wiki/CD-ROM), [keyboards](https://en.wikipedia.org/wiki/Computer_keyboard), [pointing devices](https://en.wikipedia.org/wiki/Pointing_device), [network devices](https://en.wikipedia.org/wiki/Computer_networking_device), etc.

## Software requirements

[Software requirements](https://en.wikipedia.org/wiki/Software_requirements) deal with defining software resource requirements and prerequisites that need to be installed on a computer to provide optimal functioning of an application. These requirements or prerequisites are generally not included in the software installation package and need to be installed separately before the software is installed.

### Platform

A [computing platform](https://en.wikipedia.org/wiki/Computing_platform) describes some sort of [framework](https://en.wikipedia.org/wiki/Software_framework), either in [hardware](https://en.wikipedia.org/wiki/Computer_hardware) or [software](https://en.wikipedia.org/wiki/Software), which allows software to run. Typical platforms include a computer's [architecture](https://en.wikipedia.org/wiki/Computer_architecture), [operating system](https://en.wikipedia.org/wiki/Operating_system), or [programming languages](https://en.wikipedia.org/wiki/Programming_language) and their [runtime](https://en.wikipedia.org/wiki/Run-time_system) libraries.

[Operating system](https://en.wikipedia.org/wiki/Operating_system) is one of the requirements mentioned when defining system requirements (software). Software may not be compatible with different versions of same line of operating systems, although some measure of backward compatibility is often maintained. For example, most software designed for [Microsoft Windows XP](https://en.wikipedia.org/wiki/Microsoft_Windows_XP) does not run on [Microsoft Windows 98](https://en.wikipedia.org/wiki/Microsoft_Windows_98), although the converse is not always true. Similarly, software designed using newer features of [Linux](https://en.wikipedia.org/wiki/Linux) [Kernel](https://en.wikipedia.org/wiki/Linux_Kernel) v2.6 generally does not run or compile properly (or at all) on Linux distributions using Kernel v2.2 or v2.4.

### APIs and drivers

Software making extensive use of special hardware devices, like high-end [display adapters](https://en.wikipedia.org/wiki/Graphics_processing_unit), needs special [API](https://en.wikipedia.org/wiki/Application_programming_interface) or newer device drivers. A good example is [DirectX](https://en.wikipedia.org/wiki/DirectX), which is a collection of APIs for handling tasks related to multimedia, especially game programming, on [Microsoft](https://en.wikipedia.org/wiki/Microsoft) platforms.

**TOPIC 2 : PROGRAM DEVELOPMENT AND DESIGN.**

**Program Development**

Software development is the process of [computer programming](https://en.wikipedia.org/wiki/Computer_programming), [documenting](https://en.wikipedia.org/wiki/Software_documentation), [testing](https://en.wikipedia.org/wiki/Software_testing), and [bug fixing](https://en.wikipedia.org/wiki/Software_bugs) involved in creating and maintaining [applications](https://en.wikipedia.org/wiki/Application_software) and [frameworks](https://en.wikipedia.org/wiki/Software_framework) resulting in a [software product](https://en.wikipedia.org/wiki/Software_product). Software development is a process of writing and [maintaining](https://en.wikipedia.org/wiki/Software_maintenance) the [source code](https://en.wikipedia.org/wiki/Source_code), but in a broader sense, it includes all that is involved between the conception of the desired software through to the final manifestation of the software, sometimes in a planned and [structured](https://en.wikipedia.org/wiki/Software_development_process) process. Therefore, software development may include research, new development, prototyping, modification, reuse, re-engineering, maintenance, or any other activities that result in software products.

**Program Design**

Software design is the process by which an [agent](https://en.wikipedia.org/wiki/Agency_%28philosophy%29) creates a specification of a [software artifact](https://en.wikipedia.org/wiki/Artifact_%28software_development%29), intended to accomplish [goals](https://en.wikipedia.org/wiki/Goal), using a set of primitive components and subject to [constraints](https://en.wikipedia.org/wiki/Constraint_%28mathematics%29). Software design may refer to either all the activity involved in conceptualizing, framing, implementing, commissioning, and ultimately modifying complex systems or the activity following [requirements](https://en.wikipedia.org/wiki/Software_requirements) specification and before [programming](https://en.wikipedia.org/wiki/Computer_programming), as a stylized software engineering process.

Software design usually involves problem solving and planning a [software](https://en.wikipedia.org/wiki/Software) solution. This includes both a low-level component and [algorithm design](https://en.wikipedia.org/wiki/Algorithm_design) and a high-level, [architecture](https://en.wikipedia.org/wiki/Software_architecture) design.

**Program Design Concepts**

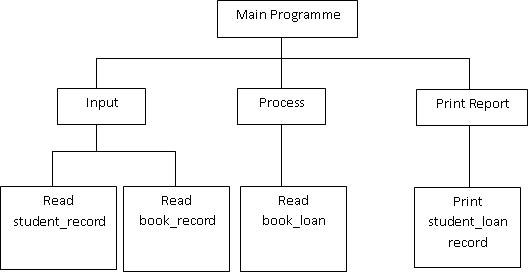
Structuring a program breaks it down into understandable chunks. Structured programming is an approach to writing programs that are easier to read, test, debug and modify. The approach assists in the development of large programs through stepwise refinement and modularity. Programs designed this way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

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

**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. It's a long-standing principle of programming style that the functional elements of a program should not be too large. If some component of a program grows beyond the stage where it's readily comprehensible, it becomes a mass of complexity which conceals errors as easily as a big city conceals fugitives. Such software will be hard to read, hard to test, and hard to debug.

**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.

**Monolithic programming.**

In [software engineering](https://en.wikipedia.org/wiki/Software_engineering), a monolithic application describes a single-tiered [software application](https://en.wikipedia.org/wiki/Application_software) in which the [user interface](https://en.wikipedia.org/wiki/User_interface) and data access code are combined into a single program from a single [platform](https://en.wikipedia.org/wiki/Platform_(computing)). A monolithic application is self-contained, and independent from other computing applications.

Indicates the program which contains a single function for the large program i.e. monolithic programming will not divide the program and it is a single thread of execution. When the program size increases it becomes difficult to maintain. This design approach is highly discouraged due to the following reasons.

* Difficult to check errors on large programs.
* Code can be specific to a particular problem i.e. it cannot be re-used.

**1.3: Properties of a Good Program**

A good program should:

* Be **reliable**, i.e. work according to specifications (or serve the intended purpose). It should for example have correct logic, should do all the processing specified in the **user specification**, etc.
* Be **efficient** (should minimize the use of computer resources, e.g. memory). This implies that a program should be as small as possible. It should also use simplified logic. For example when we come to Chapter Three (Control Structures), you will observe that you can simplify the logic of decision making or repetition structures in a program. This implies that the CPU does less work.
* Be **usable**, i.e. easy to use. This is usually ensured through use simplified logic. For example, when observe that you can simplify the logic of decision making or repetition structures in a program. This implies that the CPU does less work.
* Be **usable**, i.e. easy to use. This is usually ensured through use of consistent user interface, use of graphical user interface, use of menus, toolbars, tool tip text, user help menu, e.t.c.
* Be easy to **read and understand**, and be **maintainable** (easy to modify). These are usually ensured through use of functions, code documentation, code indenting, line spacing, etc. These will be explained more in Chapter two.
* Be easy to find and correct errors.
* Be able to handle possible errors i.e. report an error but continue functioning rather than aborting. For example, if the user inputs a character instead of a value, the program should not abort, but should report this data entry error and continue executing.
* Be ready on time.

**Control Flow Structure**

Algorithms

A **programming algorithm** is a **computer** procedure that is a lot like a recipe (called a procedure) and tells your **computer** precisely what steps to take to solve a problem or reach a goal. The ingredients are called inputs, while the results are called the outputs.

The characteristics of a good algorithm are:

* Precision – the steps are precisely stated(defined).
* Uniqueness – results of each step are uniquely defined and only depend on the input and the result of the precedingsteps.
* Finiteness – the algorithm stops after a finite number ofinstructions are executed.
* Input – the algorithm receives input.
* Output – the algorithm produces output.
* Generality – the algorithm applies to a set of inputs.

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

**The Program’s code**

(i) **Pseudo code**

Begin

Pi=3.14

Input Radius

Area=Pi\*Radius\*Radius

Circumference=2\*Pi\*Radius

Output Area, Circumference

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.

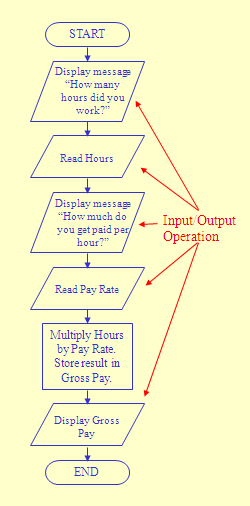
Process – used to represent operations on data e.g. computations. Details are written in a rectangular box.

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



**Two types of connectors as used in flow chart.**

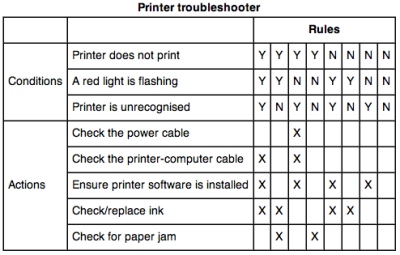
On-Page Connectors. A small circle containing a number or letter that is used to split

large flow chart into smaller parts.

Off-Page Connectors. They show a jump from one part of the flowchart to another.

**Decision Tables.**

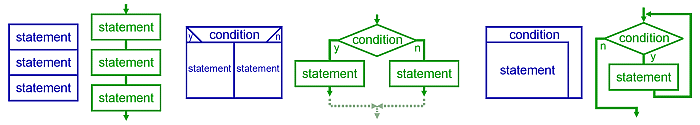
A decision table is an excellent tool to use in both testing and requirements management. Essentially it is a structured exercise to formulate requirements when dealing with complex business rules. Decision tables are used to model complicated logic. They can make it easy to see that all possible combinations of conditions have been considered and when conditions are missed. Decision tables, like [flowcharts](https://en.wikipedia.org/wiki/Flowchart), [if-then-else](https://en.wikipedia.org/wiki/Conditional_%28programming%29), and [switch-case](https://en.wikipedia.org/wiki/Switch_statement) statements, associate conditions with actions to perform, but in many cases do so in a more elegant way.



**TOPIC 3 : PROGRAM STRUCTURE.**

**Program Structure** - The overall form of a program, with particular emphasis on the individual components of the program and the interrelationships between these components. Programs are frequently referred to as either well structured or poorly structured. With a well-structured program the division into components follows some recognized principle such as [information hiding](http://www.encyclopedia.com/computing/dictionaries-thesauruses-pictures-and-press-releases/information-hiding), and the interfaces between components are explicit and simple. By contrast, with a poorly structured program the division into components is largely arbitrary (or even nonexistent), and interfaces are implicit and complex.

**Format of A Structured Programming Language**

****

* "Sequence"; ordered statements or subroutines executed in sequence.
* "Selection"; one or a number of statements is executed depending on the state of the program. This is usually expressed with [keywords](https://en.wikipedia.org/wiki/Keyword_%28computer_programming%29) such as [if..then..else..end if](https://en.wikipedia.org/wiki/Conditional_%28programming%29).
* "Iteration"; a statement or block is executed until the program reaches a certain state, or operations have been applied to every element of a collection. This is usually expressed with keywords such as [while](https://en.wikipedia.org/wiki/While_loop), [repeat](https://en.wikipedia.org/wiki/Do_while_loop), [for](https://en.wikipedia.org/wiki/For_loop) or [do..Until](https://en.wikipedia.org/wiki/Do_while_loop). Often it is recommended that each loop should only have one entry point (and in the original structural programming, also only one exit point, and a few languages enforce this).
* "Recursion"; a statement is executed by repeatedly calling itself until termination conditions are met. While similar in practice to iterative loops, recursive loops may be more computationally efficient, and are implemented differently as a cascading stack.

**INTRODUCTION TO C LANGUAGE**

**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 immediate object code
* The linker combines the intermediate with the code to produce the executable file. C does this in a modular manner.

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

Reasons why C programming language is widely used.

 Easy to learn

 Structured language

 It produces efficient programs.

 It can handle low-level activities.

 It can be compiled on a variety of computer platforms

Terminologies

Keyword

1. **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.

Examples

Auto, break, case, else, int, void, default, do, double, if, sizeof, long Float,for, goto, signed,unsigned, register, return, short, union, continue, struct, switch, typedef, const, extern, volatile, while, char, enum, static.

**ii. 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).

**iii. Text Editor**

This will be used to type your program.

**iv. The C Compiler**

it is used to compile your source code into final executable program.

**v. Identifiers**

A C identifier is a name used to identify a variable, function, or any other user-defined item.

**Rules of naming identifier**

* 1. No used of keyword
  2. No white spaces allowed
  3. Must not exceed 255 characters
  4. No use of special characters e.g. @,&,.

**Components of C programming language:**

* 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;

}

**#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.
* Upper case letters are usually used in preprocessor definitions (**later**) or inside quotes of character strings.

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

**Example: Basic C program features**

**Example 1**

#include <stdio.h>

int main()

{

/\* my first program in C \*/

printf("Hello, World! \n");

return 0; }

**Example 2**

**#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;

}

The output will be;

**This is a simple program to display a message.**

**My favorite number is 1 because it is first.**

C Data Types

It refers to an extensive system used for declaring variables or functions of different types.

Examples

## Basic types

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | Meaning | **Keyword** | Format Specifier |
| Character | Character data | Char | %c |
| Integer | Signed whole number | Int | %d |
| Float | floating-point numbers | Float | %f |
| Double | double precision floating-point numbers | Double | %lf |

# **C - Variables**

It is a name given to a storage area that our programs can manipulate.

## **Variable Definition in C**

Syntax

Datatype Variable list;

Examples

int i, j, k;

char c, ch;

float f, salary;

double d;

Types of Variables

1. Local variable

Declared and used within a function

1. Global variable

Declared outside a main function. It is used anywhere in the program.

OPERATORS

An operator is a symbol that tells the compiler to perform a certain mathematical or logical manipulation. Operators are used in programs to manipulate data and variables.

* Arithmetic operators
* Relational operators
* Logical operators
* Bitwise operators
* Assignment operators
* Conditional operators
* Special operators

### **Arithmetic operators**

They perform arithmetic’s operations.

Examples

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + | adds two operands |
| - | subtract second operands from first |
| \* | multiply two operand |
| / | divide numerator by denominator |
| % | remainder of division |
| ++ | Increment operator - increases integer value by one |
| -- | Decrement operator - decreases integer value by one |

### **Relational operators**

Use to compare values.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| == | Check if two operand are equal |
| != | Check if two operand are not equal. |
| > | Check if operand on the left is greater than operand on the right |
| < | Check operand on the left is smaller than right operand |
| >= | check left operand is greater than or equal to right operand |
| <= | Check if operand on left is smaller than or equal to right operand |

### **Logical operators**

Perform logical operations.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Logical AND | (a && b) is false |
| || | Logical OR | (a || b) is true |
| ! | Logical NOT | (!a) is false |

### **Bitwise operators**

Bitwise operators perform manipulations of data at **bit level**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| & | Bitwise AND |
| | | Bitwise OR |
| ^ | Bitwise exclusive OR |
| << | left shift |
| >> | right shift |

### **Assignment Operators**

Use to assign values to variables.

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**

1. x + y ; x, y are operands, + is an addition operator.
2. 3 \* 5; 3, 5 are constant operands, \* is a multiplication operator.
3. x % 2.5; x, 5 are operands, % is a modulus (remainder) operator.
4. 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:

* + 1. There exists no exponential operators in C.
    2. 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).
    3. 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.
  + 1. 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.
    2. 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:

1. c1 and c2 are character constants
2. 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:

1. 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).
2. 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.
3. 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).
4. 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.

1. i + f
2. i + c
3. i + c-‘w’
4. ( 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).

**Conditional operator/ ternary operator**

* Use in decision making
* Return the statement depending upon a given expression results.

Program examples

#include<<stdio.h>

Int man()

{

Int age;

Printf(“Enter the value of age”);

Scantf(“%d”,&age);

(age>=18) ? printf(“You can vote”):printf(“You can’t vote”);

Return 0;

}

* 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;

1. Multiplication and division have a higher precedence than addition and subtraction, so they are performed first.
2. 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:

1. + 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);

system(“pause”);

return 0;

}

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

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);

system(“pause”);

return 0;

}

The size of 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)) ;

system(“pause”);

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**

1. You cannot assign a variable to a constant such as 3 = a ;
2. 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.
3. Multiple assignments are possible e.g. a =b = 5 ; assigns the integer value 5 to both a and b.
4. 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**

1. Unary minus e.g. - 700 or –x
2. Incrementation operator e.g. c++
3. Decrementation operator e.g. f - -
4. 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:

* + 1. a / b (v) x % y
    2. 2 \* b + 3 \* (a – c) (vi) 2 \* x / (3 \* y)
    3. (a \* c) % b (vii) c1 / c3
    4. (x / y) + z (viii) (c1 / c2) \* c3

**Escape sequences**

Escape sequences (also called back slash codes) are character combinations that begin

with a back 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 charater. When the C compiler encounters **\n,** it translates it into a carriage

return.

**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

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**

**Types of Errors**

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

**Syntax errors**

* They result from the incorrect use of rules 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 keyboard in uppercase e.g. FLOAT, WHILE
* Misspelling keywords e.g. init instead of int

**Logic Errors**

* These occur from the incorrect use of control structures, incorrect calculation, or omission of 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 intensions. The programmer must dry run the program so that he/she compare the program’s results with already know results.

**Sematic 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 required, division by zero, etc.

**Decision Making in C/ CONTROL STRUCTURES**

It controls the flow of program execution.

* + 1. **if statement**

An if statement consists of a boolean expression followed by one or more statements.

**Syntax**

The syntax of an if statement in C programming language is:

if(boolean\_expression)

{

/\* statement(s) will execute if the boolean expression is true \*/

}



Examples

(i) if (x<y)

printf(“x is less that y”);

**Example 1**

#include <stdio.h>

int main ()

{

/\* local variable definition \*/

int a = 10;

/\* check the boolean condition using if statement \*/

if( a < 20 )

{

/\* if condition is true then print the following \*/

printf("a is less than 20\n" );

}

printf("value of a is : %d\n", a);

system(“pause”);

return 0;

}

* **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.

Syntax

The syntax of an **if...else** statement in C programming language is:

if(boolean\_expression)

{

/\* statement(s) will execute if the boolean expression is true \*/

}

else

{

/\* statement(s) will execute if the boolean expression is false \*/

}

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.

##### **Flow diagram**



**Example**

#include <stdio.h>

main()

{

int i;

printf("Enter a value \n");

scanf("%d",&i);

if (i%2 == 0)

printf("%d - is Even \n", i);

else

printf("%d - is Odd \n", i);

system("pause");

return 0;

}

* **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.

#include <stdio.h>

int main ()

{

/\* local variable definition \*/

int a = 100;

/\* check the boolean condition \*/   
 if( a < 20 )

{ /\* if condition is true then print the following \*/

printf("a is less than 20\n" );

}

else

{ /\* if condition is false then print the following \*/

printf("a is not less than 20\n" );

}

printf("value of a is : %d\n", a);

return 0;

}

**Example 1:**

/\* Program to find roots of a quadratic equation when coefficients are entered by user. \*/

/\* Library function sqrt() computes the square root. \*/

#include <stdio.h>

#include <math.h> /\* This is needed to use sqrt() function.\*/

int main()

{

float a, b, c, determinant, r1,r2, real, imag;

printf("Enter coefficients a, b and c:\n");

scanf("%f%f%f",&a,&b,&c);

determinant=b\*b-4\*a\*c;

if (determinant>0)

{

r1= (-b+sqrt(determinant))/(2\*a);

r2= (-b-sqrt(determinant))/(2\*a);

printf("Roots are: %.2f and %.2f",r1 , r2);

}

else if (determinant==0)

{

r1 = r2 = -b/(2\*a);

printf("Roots are: %.2f and %.2f", r1, r2);

}

else

{

real= -b/(2\*a);

imag = sqrt(-determinant)/(2\*a);

/\* printf("Roots are: %.2f+%.2fi and %.2f-%.2fi", real, imag, real, imag);\*/

printf("Roots are: %.2f and %.2f \n", real, imag, real, imag);

}

system("pause");

return 0;

}

**Example 2**:

#include<stdio.h >

int main()

{

int marks;

printf (" Enter the students marks \n");

scanf( "%d",&marks );

if ( marks >=75 && marks <=100)

{

printf("The grade is A");

}

else if( marks >= 60 && marks < 75 )

{

printf("The grade is B");

}

else if(marks>=50 && marks<60)

{

printf("The grade is C");

}

else if(marks>=40 && marks<50)

{

printf("The grade is D");

}

else if (marks>=0 && marks<40)

{

printf ("The grade is E");

}

else

{

printf("The mark is impossible!" );

}

system ("pause");

return 0;

}

1. **Looping/Repetition Control Structure**

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.

The syntax of a **while** loop in C programming language is:

while(condition)

{

statement(s);

}



Example1

#include <stdio.h>

int main ()

{

/\* local variable definition \*/

int a = 10;

/\* while loop execution \*/

while( a < 20 )

{ printf("value of a: %d\n", a);

a++;

}

return 0;

}

For loop

for ( init; condition; increment )

{

statement(s);

}

#include <stdio.h>

int main ()

{

/\* for loop execution \*/

for( int a = 10; a < 20; a = a + 1 )

{

printf("value of a: %d\n", a);

}

return 0;

}

Example 3

/\* Displays the digits 1 through 9 \*/

#include<stdio.h>

main()

{

int digit;

for(digit=0;digit<=9; digit++)

printf(“%d \n” , digit);

return 0;

}

do...while loop

it checks its condition at the bottom of the loop.

Syntax

The syntax of a do...while loop in C programming language is:

do { statement(s);

} while( condition );



Example 1:

/\* Displays the digits 1 through 9 \*/

main()

{

int digit=0; /\* Initialisation \*/

do

{

printf(“%d \n”, digit);

digit++;

} while (digit<=9);

system(“pause”);

return 0;

}

Example 2:

#include <stdio.h>

main()

{

int value =0;

do

{ if (value %2 != 0)

{

printf("Value is %d\n", value);

}

value++;

}while(value<=30);

system("pause");

return 0;

}

FUNCTIONS

A function is combined of a block of code that can be called or used anywhere in the program by calling the name. A function is a block of statements that performs a specific task.

Body of a function starts with { and ends with } . This is similar to the main function.

### Types of functions

1) **Predefined standard library functions** – such as puts(), gets(), printf(), scanf() etc – These are the functions which already have a definition in header files (.h files like stdio.h), so we just call them whenever there is a need to use them.

2) **User Defined functions –**The functions that we create in a program are known as user defined functions.

### Syntax of a function

return\_type function\_name (argument list)

{

Set of statements – Block of code

}

Components of a Function

* 1. **return\_type:**

Return type can be of any data type such as int, double, char, void, short etc. it’s data type of the value return by a function during function calling.

* 1. **function\_name:**

Name of the function given by programmer. Must follow rules of naming identifier.

* 1. **argument list/ parameters:**

Argument list contains variables names along with their data types. These arguments are kind of inputs for the function.

Types

1. Actual parameters
2. Formal parameters
   1. **Block of code:**Set of C statements, which will be executed whenever a call will be made to the function.

Example

#include<stdio.h>

/\*Function prototypes\*/

myfunc();

main()

{

myfunc();

}

/\*Function Defination\*/

myfunc()

{

printf("Hello, this is a test\n");

}

Reasons for using Functions:

* Writing functions avoids rewriting the same code over and over.
* Using functions it becomes easier to write programs and keep track of what they are doing.
* To improve the readability of code.
* Improves the reusability of the code, same function can be used in any program rather than writing the same code from scratch.
* Debugging of the code would be easier if you use functions, as errors are easy to be traced.
* Reduces the size of the code, duplicate set of statements are replaced by function calls.

Functions with Parameters

#include<stdio.h>

int min(int a,int b);

main()

{

int m;

m=min(3,6);

printf("Minimum is %d",m);

return 0;

}

int min(int a,int b)

{

if(a<b)

return a;

else

return b;

}