[STRUCTURED PROGRAMMING 5](#_Toc452806530)

[DEFINITION OF TERMS 5](#_Toc452806531)

[INTRODUCTION TO STRUCTURED PROGRAMMING 7](#_Toc452806532)

[HISTORY OF PROGRAMMING LANGUAGES 7](#_Toc452806533)

[Low-Level Programming Languages 7](#_Toc452806534)

[High-Level Programming Languages 8](#_Toc452806535)

[*C*ompiled Languages VS Interpreted Languages 10](#_Toc452806536)

[Compiled Languages 10](#_Toc452806537)

[Interpreted Languages 10](#_Toc452806538)

[Comparison of Compiled and Interpreted Languages 10](#_Toc452806539)

[Criteria for Selecting a Programming Language 10](#_Toc452806540)

[*PROGRAMMING PARADIGMS* 12](#_Toc452806541)

[SOFTWARE CONSIDERATIONS 13](#_Toc452806542)

[ADVANTAGES of C LANGUAGE 13](#_Toc452806543)

[PROGRAM DESIGN AND DEVELOPMENT 15](#_Toc452806544)

[PROGRAM DEVELOPMENT CYCLE 15](#_Toc452806545)

[STRUCTURED PROGRAMMING DESIGN CONCEPTS 16](#_Toc452806546)

[PROGRAM DESIGN TOOLS 17](#_Toc452806547)

[3. PROGRAM STRUCTURE 23](#_Toc452806548)

[STRUCTURE OF A C PROGRAM 23](#_Toc452806549)

[SOURCE CODE FILES 26](#_Toc452806550)

[STEP BY STEP CREATION AND EXECUTION OF A C PROGRAM 27](#_Toc452806551)

[C DATA TYPES 27](#_Toc452806552)

[VARIABLES 29](#_Toc452806553)

[CONSTANTS 32](#_Toc452806554)

[TYPE CASTING 32](#_Toc452806555)

[C PROGRAMMING OPERATORS 33](#_Toc452806556)

[ARITHMETIC OPERATORS 33](#_Toc452806557)

[INCREMENT AND DECREMENT OPERATORS – Unary Operators 33](#_Toc452806558)

[ASSIGNMENT OPERATORS – Binary Operators 34](#_Toc452806559)

[RELATIONAL OPERATORS - Binary Operators 34](#_Toc452806560)

[LOGICAL OPERATORS - Binary Operators 35](#_Toc452806561)

[CONDITIONAL OPERATOR – Ternary Operators 36](#_Toc452806562)

[PRECEDENCE OF OPERATORS 36](#_Toc452806563)

[ASSOCIATIVITY OF OPERATORS 37](#_Toc452806564)

[CONTROL STRUCTURES 39](#_Toc452806565)

[IMPORTANCE OF CONTROL STRUCTURES 39](#_Toc452806566)

[TYPES OF CONTROL STRUCTURES 39](#_Toc452806567)

[SELECTION STRUCTURES 40](#_Toc452806568)

[(a) THE IF SELECTION STRUCTURE 40](#_Toc452806569)

[(b) THE IF/ELSE 41](#_Toc452806570)

[(c) THE IF...ELSE IF...ELSE STATEMENT 42](#_Toc452806571)

[(d) NESTED IF STATEMENTS 42](#_Toc452806572)

[(e) SWITCH STATEMENT 43](#_Toc452806573)

[(f) NESTED SWITCH STATEMENTS 45](#_Toc452806574)

[REPETITION/ITERATIVE/LOOP STRUCTURES 46](#_Toc452806575)

[(a) WHILE LOOP IN C 47](#_Toc452806576)

[(b) FOR LOOP IN C 47](#_Toc452806577)

[(c) DO...WHILE LOOP IN C – Post Test Loop 49](#_Toc452806578)

[(d) NESTED LOOPS IN C 50](#_Toc452806579)

[LOOP CONTROL STATEMENTS 52](#_Toc452806580)

[(a) BREAK STATEMENT IN C 52](#_Toc452806581)

[(b) CONTINUE STATEMENT IN C 53](#_Toc452806582)

[(c) THE COMMA OPERATOR 53](#_Toc452806583)

[(d) GOTO STATEMENT IN C 54](#_Toc452806584)

[(e) THE RETURN STATEMENT 55](#_Toc452806585)

[THE INFINITE LOOP 55](#_Toc452806586)

[SUBPROGRAMS IN C 57](#_Toc452806587)

[TYPES OF FUNCTIONS 57](#_Toc452806588)

[LIBRARY FUNCTION 57](#_Toc452806589)

[USER DEFINED FUNCTION 58](#_Toc452806590)

[DEFINING A FUNCTION 59](#_Toc452806591)

[FUNCTION DECLARATIONS 60](#_Toc452806592)

[CALLING A FUNCTION 60](#_Toc452806593)

[FUNCTION ARGUMENTS 62](#_Toc452806594)

[TYPES OF VARIABLES 62](#_Toc452806595)

[LOCAL VARIABLES 62](#_Toc452806596)

[GLOBAL VARIABLES 62](#_Toc452806597)

[DATA STRUCTURES 65](#_Toc452806598)

[ARRAYS 66](#_Toc452806599)

[DECLARING ARRAYS 66](#_Toc452806600)

[INITIALIZING ARRAYS 67](#_Toc452806601)

[ACCESSING ARRAY ELEMENTS 67](#_Toc452806602)

[SORT TECHNIQUES 68](#_Toc452806603)

[SEARCHING ARRAYS 74](#_Toc452806604)

[LINKED LISTS **Error! Bookmark not defined.**](#_Toc452806605)

[POINTERS 76](#_Toc452806606)

[C STRINGS 78](#_Toc452806607)

[Structures 84](#_Toc452806608)

[Defining a Structure 85](#_Toc452806609)

[Accessing Structure Members 86](#_Toc452806610)

[typedef 86](#_Toc452806611)

[QUEUES 88](#_Toc452806612)

[STACKS 88](#_Toc452806613)

[FILE HANDLING 89](#_Toc452806614)

[OPENING FILES 89](#_Toc452806615)

[CLOSING A FILE 90](#_Toc452806616)

[WRITING A FILE 90](#_Toc452806617)

[READING A FILE 91](#_Toc452806618)

[BINARY I/O FUNCTIONS 92](#_Toc452806619)

[ERROR HANDLING AND DEBUGGING 95](#_Toc452806620)

[Types of Errors 95](#_Toc452806621)

[Syntactic Errors 95](#_Toc452806622)

[Execution Errors 95](#_Toc452806623)

[Logical Errors 96](#_Toc452806624)

[DEBUGGING TECHNIQUES 96](#_Toc452806625)

[Error Isolation 96](#_Toc452806626)

[Tracing 96](#_Toc452806627)

[Watch Values 97](#_Toc452806628)

[Breakpoints 97](#_Toc452806629)

[Stepping 97](#_Toc452806630)

[SOFTWARE DOCUMENTATION 98](#_Toc452806631)

[INTERNAL DOCUMENTATION 98](#_Toc452806632)

[EXTERNAL DOCUMENTATION 99](#_Toc452806633)

[PROCESS DOCUMENTATION 99](#_Toc452806634)

[PRODUCT DOCUMENTATION 99](#_Toc452806635)

# STRUCTURED PROGRAMMING

## DEFINITION OF TERMS

1. **HEADER FILE** - A header file is a file with extension **.h** which contains C function declarations to be shared between several source files. A header file is used in a program by including it with the use of the preprocessing directive **#include**, which comes along with the compiler.

**#include<stdio**.h>

1. **EXPRESSION –** These are statements that return a value. Expressions combine variables and constants to create new values or logical conditions which are either true or false e.g.

x + y, x <= y etc

1. **KEYWORD -** A keyword is a reserved word in C. Reserved words may not be used as constants or variables or any other identifier names. Examples include auto, else, Long, switch, typedef ,break etc
2. **IDENTIFIER -** A C **identifier** is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore \_ followed by zero or more letters, underscores, and digits (0 to 9). C does not allow punctuation characters such as @, $, and % within identifiers.
3. **COMMENT -** These are non-executable program statements meant to enhance program readability and maintenance- they document the program.
4. **FUNCTION -** A function is a group of statements, enclosed within curly braces, which together perform a task.
5. **STATEMENT-** Statements are expressions, assignments, function calls, or control flow statements which make up C programs. Statements are terminated using a semicolon.
6. **SOURCE CODE –** Program instructions in their original form. C source code files have an extension **.c**
7. **OBJECT CODE –** Code produced by a compiler from source code and exists in machine readable language.
8. **EXECUTABLE FILE –** Refers to a file in a format that a computer can directly execute and is created by a compiler.
9. **STANDARD LIBRARY –** Refers to a collection of precompiled functions/routines that a program can use. The routines are stored in object format and contain descriptions of functions to perform I/O, string manipulations, mathematics etc.
10. **SIGNED INTEGER –** This is an integer that can hold either positive or negative numbers.
11. **COMPILER –** This is a program that translates source code into object code.
12. **PREPROCESSOR COMMAND -** The [C preprocessor](http://www.cprogramming.com/tutorial/cpreprocessor.html) modifies a source file before handing it over to the compiler for instance by including header files with [#include](http://www.cprogramming.com/reference/preprocessor/include.html) as I #include <stdio.h>
13. **LINKER/Binder/Link Editor –** This is a program that combines object modules to form an executable program. The linker combines the object code, the start up code and the code for library routines used in the program (all in machine language) into a single file- the executable file**.**
14. **OPERATOR -** A symbol that represents a specific action. For example, a plus sign (+) is an operator that represents addition. The basic mathematic [operators](http://www.webopedia.com/TERM/O/operator.html) are **+ addition, - subtraction,\* multiplication,/ division**
15. **OPERAND -** Operands are the [objects](http://www.webopedia.com/TERM/O/object.html) that are manipulated by operators in expressions. For example, in the expression 5 + x, x and 5 are operands and + is an operator. All expressions have at least one operand.
16. **EXPRESSION –** This is a statement that returns a value. For example, when you add two numbers together or test to see whether one value is equal to another.
17. **VARIABLE -** A variable is a memory location whose value can change during program execution. Variable declaration must have a type, which defines what values that variable can hold.
18. **Data type** – The data type of a variable etc determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.
19. **CONSTANT -** a constant is a value that never changes during program execution.
20. **WHITESPACE** *-* A line containing only whitespace, possibly with a comment, is known as a blank line, and a C compiler totally ignores it.

Whitespace is the term used in C to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins. Therefore, in the following statement:

*int age;*

There must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them. On the other hand, in the following statement

***CHAPTER 1***

# INTRODUCTION TO STRUCTURED PROGRAMMING

**Programming** means to convert problem solutions into instructions for the computer. It also refers to the process of developing and implementing various sets of instructions to enable a computer to do a certain task.

**Structured programming** (sometimes known as *modular programming*) is an approach to writing programs that are easier to test, debug, modify and maintain by enforcing a modular approach which breaks a large complex problem into sub-problems.

**A programming language** is a vocabulary and set of grammatical rules designed for instructing a [computer](http://www.webopedia.com/TERM/C/computer.html) to perform specific tasks.

## HISTORY OF PROGRAMMING LANGUAGES

### Low-Level Programming Languages

**First-Generation Programming Languages – Machine Language**

A first-generation of [programming language](http://en.wikipedia.org/wiki/Programming_language)s includes **machine-level** programming languages. These languages were introduced in the 1940s and had the following characteristics:

* Instructions were entered directly in binary format (1s and 0s) and therefore they were **tedious** and **error prone**. Programmers had to design their code by hand then transfer it to a computer using a punch card, punch tape or flicking switches.
* **I**[**nstructions**](http://en.wikipedia.org/wiki/Instruction_%28computer_science%29) **were executed** **directl**y by a [computer](http://en.wikipedia.org/wiki/Computer)'s [central processing unit](http://en.wikipedia.org/wiki/Central_processing_unit) (CPU) i.e. they were executed **very fast**.
* Memory management eg calculation of numerical addresses was done manually.
* Programs were very difficult to edit and debug.
* Used to code simple programs only.

**Second-Generation Programming Languages (**[**2GL**](http://en.wikipedia.org/wiki/Second-generation_programming_language)**) –Assembly Languages**

They were introduced to mitigate the error prone and excessively difficult nature of binary programming.

* Introduced in the 1950s
* Improved on first generation by providing human readable **source** **code** which must be assembled into machine code (binary instructions) before it can be executed by a CPU. By using codes resembling English, programming becomes much easier. The use of these [**mnemonic**](http://en.wikipedia.org/wiki/Mnemonic) **codes** such as **LDA** for **load** and **STA** for **store** means the code is easier to read and write.
* Specific to platform architecture i.e. 2GL source code is **not portable**across processors or processing environments.
* Designed to support logical structure and debugging.

To convert an assembly code program into object code to run on a computer requires an **Assembler** and each line of assembly can be replaced by the equivalent line of object (machine) code:

|  |  |  |
| --- | --- | --- |
| **Assembly Code** |  | **Machine Code** |
| LDA A  ADD #5  STA A  JMP #3 | -> Assembler -> | 000100110100 001000000101 001100110100 010000000011 |

Such languages are sometimes still used for **kernels**and **device drivers**, i.e. the core of the operating system and for specific machine parts. More often, such languages are used in areas of intense processing, like **graphics programming**, when the code needs to be **optimized for performance**.

Almost every CPU architecture has a companion assembly language.  Most commonly used are the assembly languages today like **Autocoder** for IBM mainframe systems, **Linoreum**, **MACRO -11**,etc.

### High-Level Programming Languages

**Third-Generation Languages (**[**3GL**](http://en.wikipedia.org/wiki/Third-generation_programming_language)**)**

Third generation languages are the primary languages used in general purpose programming today.  They each vary quite widely in terms of their particular abstractions and syntax.  However, they all share great enhancements in logical structure over assembly languages.

* Introduced in the 1950s
* Designed around ease of use for the programmer (Programmer friendly)
* Driven by desire for reduction in **bugs,**
* increases in **code reuse**
* Based on natural language
* Often designed with [structured programming](http://en.wikipedia.org/wiki/Structured_programming) in mind
* The languages are architecture independente.g. C, Java etc.

**Examples:**

Most Modern General Purpose Languages such as C, C++, C#, Java, Basic, COBOL, Lisp and ML.

**Fourth Generation Languages**

Fourth-generation programming languages are high-level languages **built around database systems.** They are generally used in commercial environments.

* Improves on 3GL and their development methods with **higher abstraction** and **statement power**, to reduce errors and increase development speed by reducing programming effort. They result in a reduction in the cost of software development.
* A 4GL is designed with a **specific purpose** in mind. For example languages to query databases **(**[**SQL**](http://en.wikipedia.org/wiki/SQL)**),** languages to make reports ([Oracle Reports](http://en.wikipedia.org/wiki/Oracle_Reports)) etc.
* 4GL are more oriented towards problem solving and systems engineering.   
    
  Examples: Progress 4GL, PL/SQL, Oracle Reports, Revolution language, SAS, SPSS, SQ

**Fifth Generation Languages**  
  
Improves on the previous generations by skipping algorithm writing and instead provide **constraints/conditions**.   
  
While 4GL are designed to build specific programs, 5GL are designed to make the computer solve a given problem without the programmer. The programmer only needs to worry about **what problems need to be solved and only inputs a set of logical constraints**, with no specified algorithm, and the **Artificial Intelligence (AI)-based compiler builds the program based on these constraints**  
  
Examples: Prolog, OPS5, Mercury

**A machine-language program that adds overtime pay to base pay and stores the result in gross pay.**

**+1300042774**

**+1400593419**

**+1200274027**

**Machine-language equivalent.**

**LOAD BASEPAY**

**ADD OVERPAY**

**STORE GROSSPAY**

**High-level language equivalent**

**grossPay = basePay + overTimePay**

**Low-Level Languages Versus** **High-Level Languages**

**Low-level languages** such as [**machine language**](http://www.webopedia.com/TERM/M/machine_language.html) and [**assembly language**](http://www.webopedia.com/TERM/A/assembly_language.html) are closer to the [hardware](http://www.webopedia.com/TERM/H/hardware.html) than are the [high-level programming languages](http://www.webopedia.com/TERM/H/high_level_language.html), which are closer to human [languages](http://www.webopedia.com/TERM/L/language.html).

Low-level languages are converted to machine code without using a compiler or interpreter, and the resulting code runs directly on the processor. A program written in a low-level language runs **very quickly**, and with a **very small memory footprint**; an equivalent program in a high-level language will be more heavyweight. Low-level languages are **simple**, but are considered **difficult to use**, due to the numerous technical details which must be remembered.

**High-level languages** are closer to human languages and further from [machine languages](http://www.webopedia.com/TERM/M/machine_language.html).

The main advantage of high-level languages over [low-level languages](http://www.webopedia.com/TERM/L/low_level_language.html) is that they are **easier to read**, **write, and maintain**. Ultimately, programs written in a high-level language must be translated into machine language by a [compiler](http://www.webopedia.com/TERM/C/compiler.html) or [interpreter](http://www.webopedia.com/TERM/I/interpreter.html).

The first high-level programming languages were designed in the 1950s. Now there are dozens of different languages, including [Ada](http://www.webopedia.com/TERM/A/Ada.html), Algol, [BASIC](http://www.webopedia.com/TERM/B/BASIC.html), [COBOL](http://www.webopedia.com/TERM/C/COBOL.html), C, [C++](http://www.webopedia.com/TERM/C/C_plus_plus.html), FORTRAN, [LISP](http://www.webopedia.com/TERM/L/LISP.html), Pascal, and [Prolog](http://www.webopedia.com/TERM/P/Prolog.html).

# *C*ompiled Languages VS Interpreted Languages

# Compiled Languages

A program written in a compiled language is first translated to machine language by another program called a compiler and outputs equivalent machine language to another file (the executable file). The compiler parses/interprets each statement in the program and outputs an equivalent sequence of machine or assembly instructions.

# Interpreted Languages

A program written in an interpreted language is never translated to machine language. Instead, the program is interpreted and executed by another program called an interpreter.

Like a compiler, an interpreter parses each statement in the source program, but instead of translating it to machine language, it simply executes the statement as soon as it determines its meaning.

# Comparison of Compiled and Interpreted Languages

Since a compiled program is translated in advance, compiled programs have the following advantages:

* They execute very fast, since they are executed natively (directly) by the CPU.

All of the expensive parsing of the source code is done before the program begins executing, and need only be done once. After that, the machine language produced by the compiler runs with no further need for the source code.

* There is no need to have a compiler on every computer that runs the executable. The compiler need only be present on the developer's computer, and the executables produced can be run on any computer with the same architecture and operating system.

Programs written in interpreted languages have the following disadvantages:

* They run far slower than the same program written in a compiled language. This is due to the fact that an interpreted program is being parsed *while it is executing.*
* The interpreter must be installed on every computer that runs the program.

The main **advantage** that is often cited for interpreted programs is that you don't have to wait for it to compile before you can test it. For this reason, interpreters are sometimes used during the development of a program, when a programmer wants to add small sections at a time and test them quickly. In addition, interpreters are often used in education because they allow students to program interactively.

# Criteria for Selecting a Programming Language

The following criteria can be used for language selection:

* Ease of learning the language
* Programming experience and Language Familiarity
* Whether it is compiled or interpreted.
* The general execution speed of the language
* Whether it is proprietary or open standard – Cost consideration.
* Whether it is portable across different hardware and operating systems or not.
* GUI requirements - If a GUI is required then it can take up a lot of time and code. For example, creating a GUI in C++ will be more complex than creating a GUI in Visual Basic.
* Time available for Project development - High level languages such as Visual Basic make programming both easier and faster.

# *PROGRAMMING PARADIGMS*

A programming [paradigm](http://en.wikipedia.org/wiki/Paradigm) is a fundamental style of [computer programming](http://en.wikipedia.org/wiki/Computer_programming), a way of building the structure and elements of computer programs. There are four main paradigms:

**a) Unstructured Programming**

In unstructured programs, the statements are executed in sequence (one after the other) as written. This type of programming uses the GoTo statement which allows control to be passed to any other section 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 execute it. This often makes it difficult to understand the logic of such a program.

**b) Structured Programming**

The approach was developed as a solution to the challenges posed by unstructured/procedural programming. During the 1960s, many large software-development efforts encountered severe difficulties. Software schedules were typically late, costs greatly exceeded budgets and the finished

products were unreliable.

People began to realize that software development was a far more complex activity than they had imagined. Research activity in the 1960s resulted in the evolution of structured programming—a disciplined approach to writing programs that are clearer than unstructured programs, easier to test and debug and easier to modify.

Structured programming frequently employs a **top-down design model**, in which developers **break the overall program** structure into **separate subsections**. A defined function or set of similar functions is coded in a separate module or sub-module, which means that **code can be loaded into** [**memory**](http://searchmobilecomputing.techtarget.com/definition/memory) **more efficiently** and that **modules can be reused in other programs**. After a module has been tested individually, it is then integrated with other modules into the overall program structure.

Program flow follows a simple hierarchical model that employs looping constructs such as "for," "repeat," and "while."

Use of the "GoTo" statement is discouraged.

Most programs will require thousands or millions of lines of code. (Windows 2000 – over 35 millions lines of code). The importance of splitting a problem into a series of self-contained modules then becomes obvious. A module should not exceed 100 lines, and preferably short enough to fit on a single page or screen.

Examples of structured programming languages include:

* C
* Pascal
* Fortran
* Cobol
* ALGOL
* Ada
* dBASE etc.

**c) Object-oriented programming (OOP)**

This is a [programming paradigm](http://en.wikipedia.org/wiki/Programming_paradigm) that represents concepts as "[objects](http://en.wikipedia.org/wiki/Object_%28computer_science%29)" that have [data fields](http://en.wikipedia.org/wiki/Field_%28computer_science%29) (attributes that describe the object) and associated procedures known as [methods](http://en.wikipedia.org/wiki/Method_%28computer_science%29). Objects, which are usually [instances](http://en.wikipedia.org/wiki/Instance_%28computer_science%29) of [classes](http://en.wikipedia.org/wiki/Class_%28computer_science%29), are used to interact with one another to design applications and computer programs. Rather than treat data and procedures separately, OOP packages them into “objects”.

In addition, programmers can create relationships between objects. For example, objects can inherit characteristics from other objects.

One of the principal advantages of object-oriented programming techniques over procedural programming techniques is that they enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from existing objects.

**d) Visual Programming**

A visual programming language uses a visual representation (such as graphics, drawings, animation or icons, partially or completely). A visual language manipulates visual information or supports visual interaction, or allows programming with visual expressions.

A VPL allows programming with visual expressions, spatial arrangements of text and graphic symbols, used either as elements of [syntax](http://en.wikipedia.org/wiki/Syntax) or [secondary notation](http://en.wikipedia.org/wiki/Secondary_notation). An example of visual programming languages is Microsoft Visual Basic which was derived from [BASIC](http://en.wikipedia.org/wiki/BASIC) and enables the [rapid application development (RAD)](http://en.wikipedia.org/wiki/Rapid_application_development) of [graphical user interface (GUI)](http://en.wikipedia.org/wiki/Graphical_user_interface) applications.

Programming in VB is a combination of visually arranging [components](http://en.wikipedia.org/wiki/GUI_widget) or [controls](http://en.wikipedia.org/wiki/GUI_widget) on a [form](http://en.wikipedia.org/wiki/Form_(programming)), specifying attributes and actions for those components, and writing additional lines of [code](http://en.wikipedia.org/wiki/Source_code) for more functionality.

**e) Internet Based Programming**

This is programming oriented to the development of internet applications using languages and tools such as PHP, ASP, Perl, JavaScript, HTML, Java etc.

## SOFTWARE CONSIDERATIONS

Before you can start programming in C, you will need a [text editor](http://en.wikipedia.org/wiki/Text_Editor) such as a plain text Notepad Editor though it does not offer code completion or debugging. Many programmers prefer and recommend using an [Integrated Development Environment](http://en.wikipedia.org/wiki/Integrated_development_environment) (IDE) instead of a text editor on which to code, compile and test their programs.

Memory requirements

Disk space required

## ADVANTAGES of C LANGUAGE

1. **C is a modular language:** we can split the C program into a number of modules instead of repeating the same logic statements (sequentially). It allows reusability of modules.
2. **General purpose programming language:** C can be used to implement any kind of applications such as math’s oriented, graphics, business oriented applications and system software.
3. **C is highly Portable:** we can compile or execute C program in any operating system (UNIX, dos, windows).
4. **Powerful and efficient programming language:** C is a very efficient and powerful programming language; it is best used for data structures and designing system software. It is efficient in that it is modular thus makes efficient use of memory and system resources.

***CHAPTER 2***

# PROGRAM DESIGN AND DEVELOPMENT

**Program/System design** is the process of defining the software structure (components/ modules, interfaces, and [data](http://en.wikipedia.org/wiki/Data)) to satisfy/realize specified [requirements](http://en.wikipedia.org/wiki/Requirement).

**Program development** refers to all the activities and processes involved between the conception of the desired software through to the final manifestation of the software, in a planned and structured process.

## PROGRAM DEVELOPMENT CYCLE

This refers to the stages that form the framework for planning and controlling the creation of an information system. Several approaches to program development have been devised and the System Development Life Cycle (SDLC) is one of the most popular. The SDLC is a methodology that aims at producing a high quality system that **meets or exceeds customer expectations**, **reaches completion within times** and **cost estimates**, **works efficiently** and is **inexpensive to maintain and cost-effective to enhance**.

1. **Problem statement** and **feasibility study:** The fundamental process of understanding why a system should be built and determining how the project team will go about building it. It involves
   1. Technical feasibility study: can the system be built
   2. Economic feasibility study: will the system provide business value, and what are the risks?
   3. Organizational feasibility study: if built, will the system be used, accepted by the users etc.
2. Systems **analysis/requirements definition:** this phase identifies the users of the system and what the system will do or is expected to do. It involves
   1. Analysis of the old system
   2. Requirements gathering. Various tools for collecting information are used. These include interviews, questionnaires, observation of the users of the old system etc.
   3. Development of the new system proposal document.
3. Systems **design:** describes how the system will operate, in terms of hardware, software, network infrastructure, user interface, forms and reports that will be used, the specific programs, databases and files that will be needed. Design phase steps include;
   1. Design strategy: method of development: in-house, outsourced, or purchased
   2. Architecture design – hardware , software, internet infrastructure, and user interface
   3. Database and file specification
   4. Program design: defines the program that needs to be done and exactly what each module will do.
4. System **Implementation:** The real code is written at this stage.
5. **Integration and testing:** Brings all the pieces of the project together into a special testing environment, then checks for errors and interoperability.
6. **Acceptance, installation, deployment:** The final stage of initial development, where the software is put into use(deployed) and runs actual business.
7. **Maintenance:** What happens during the rest of the software's life: changes, correction, additions, and moves to a different computing platforms etc. This step, perhaps most important of all, goes on seemingly forever.

The SDLC is a cycle i.e. iterative in that a new requirement might initiate the whole process again.

## STRUCTURED PROGRAMMING DESIGN CONCEPTS

1. **TOP-DOWN DESIGN**

A **top-down** approach (also known as **stepwise design** or [**deductive reasoning**](http://en.wikipedia.org/wiki/Deductive_reasoning), and in many cases used as a synonym of [analysis](http://en.wikipedia.org/wiki/Analysis) or [decomposition](http://en.wikipedia.org/wiki/Decomposition_%28disambiguation%29)) is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. Top- down approach starts with the big picture. It breaks down from there into smaller segments.

Top-down design(also called “Modular **programming** " and "**stepwise refinement**") therefore, is a software design technique that emphasizes separating the functionality of a program into independent modules such that each module is designed to execute only one aspect of the desired functionality.

**ADVANTAGES OF MODULAR PROGRAMMING**

* Allows a problem to be split in stages and for a team of programmers to work on the different modules thereby reducing program development time.
* Program modification and debugging is easier since changes can be isolated to specific modules.
* Modules can be tested and debugged independently.
* Since a module performs a specific and well defined task, it is possible to develop and place commonly used modules in a user library so that they can be accessed by different programs. This is also called code reuse. (E.g. Validation, Uppercase, Text color etc.)
* If a programmer cannot continue through the entire project, it is easier for another programmer to continue working on self-contained modules.

1. **BOTTOM-UP DESIGN**

A **bottom-up** approach (also known as [inductive reasoning](http://en.wikipedia.org/wiki/Inductive_reasoning), and in many cases used as a synonym of [synthesis](http://en.wiktionary.org/wiki/synthesis)) is the piecing together of systems to give rise to larger systems, thus making the original systems sub-systems of the emergent system. In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed.

**With this approach, there is more user and business awareness of the product**. Benefits are also realized in the early phases of development.

1. **DATA DRIVEN DESIGN**

Data driven design is a design model where the data itself controls the flow of the program and not the program logic. It is a model where you control the flow by offering different data sets to the program where the program logic is some generic form of flow or of state-changes.

1. **MONOLITHIC DESIGN**

The monolithic design philosophy is that the application is responsible not just for a particular task, but can perform every step needed to complete a particular function

A monolithic application describes a software application which is designed without modularity.

## PROGRAM DESIGN TOOLS

1. **Pseudocode**

Pseudocode is an informal [high-level](http://en.wikipedia.org/wiki/High-level_programming_language) description of the operating principle of a [computer program](http://en.wikipedia.org/wiki/Computer_program) or other [algorithm](http://en.wikipedia.org/wiki/Algorithm) and is intended for human reading rather than machine reading. pseudo-code represents instructions in independent of any programming language. Pseudocode is a way to describe the algorithm in order to transform the algorithm into real source code.

For example, the Pseudocode for comparing three numbers might be written:

**Begin**

**Get first number, A and second number, B**

**If A is greater than B  
A is the Biggest  
Otherwise  
B is the Biggest  
End**

Pseudocode cannot be [compiled](http://www.webopedia.com/TERM/C/compile.html) nor [executed](http://www.webopedia.com/TERM/E/execute.html), and there are no real formatting or syntax rules. It is simply one step - an important one - in producing the final [code](http://www.webopedia.com/TERM/C/code.html)

**Pseudocode to compute the area of a rectangle:**

**Begin**

**Get the length, *l*, and width, *w***

**Multiply *l* by *w* to get the area**

**Display the area**

**End**

1. **Algorithm**

This refers to an established, **computational procedure** for solving a problem in a finite number of steps. Algorithms can be expressed in any [language](http://www.webopedia.com/TERM/L/language.html) including [natural languages](http://www.webopedia.com/TERM/N/natural_language.html) such as English. Algorithm means a method/ logic for solving a given problem.

Pseudocode is successively refined to get the step by step detailed ***algorithm*** that is very close to a computer language.

**An algorithm to find the larger of two numbers entered by user.**

Step 1: Start

Step 2: Declare variables a and b.

Step 3: Input a and b.

Step 4: If a>b

Display a is the largest number.

Else

Display b is the largest number.

End if

Step 5: Stop

**Exercise 1**

A student designed a program to accept the age of an employee and then calculate the employee’s retirement year. If the user keys in a value below 0, the program should prompt the user for a valid input. However, if the value keyed in is greater than 65, the system should display the message that the employee should be retired already. Draw a flow chart to represent the logic above.   
Required:

* 1. Write out the pseudo-code and the algorithm for the program
  2. Write the program in both C and Pascal

**Algorithm**

Step 1: Start

Step 2: Declare variables retirement\_year,emp\_age,retirement\_age,current\_year

Step 3: Read variables emp\_age,retirement\_age,current\_year

Step 4: if emp\_age <= 0

Print “Key in a valid input”

Else if emp-age > 65

Print “Should be retired already”

Else

retirement\_year=(retirement\_age-employee\_age)+ current\_year

End if

Step 5: Display retirement\_year

Step 5: Stop

**Exercise 2**

A program was designed to determine a student’s mean score as the average of two marks: eng and maths. If the mean score is greater than 50, the student is given a pass and a fail if otherwise. Write the pseudocode and algorithm for the program.

1. **Flowchart**

A **flowchart** is a type of [diagram](http://en.wikipedia.org/wiki/Diagram) that **represents an** [**algorithm**](http://en.wikipedia.org/wiki/Algorithm) or [process](http://en.wikipedia.org/wiki/Process_%28science%29), showing the **steps** as boxes of various kinds, and their **order** by connecting them with arrows.

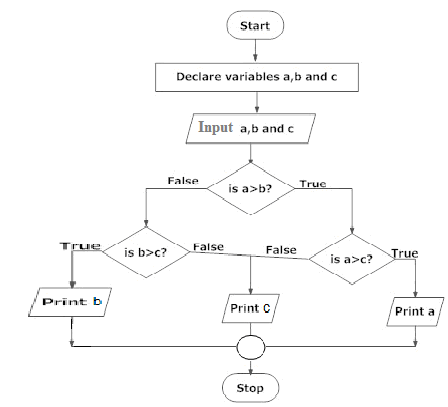
This diagrammatic [representation](http://en.wikipedia.org/wiki/Knowledge_representation_and_reasoning) **illustrates a solution to a given** [**problem**](http://en.wikipedia.org/wiki/Problem_solving). Flowcharts are used in **designing** and **documenting** complex processes or programs. Like other types of diagrams, they help to visualize what is going on and thereby help the viewer to understand a process, and perhaps also find flaws/errors, bottlenecks, and other less-obvious features within it.

**SYMBOLS USED IN FLOWCHART**

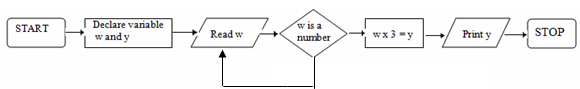
Different symbols are used for different states in flowcharts. The table below describes all the symbols that are used in making flowchart

| **Symbol** | **Purpose** | **Description** |
| --- | --- | --- |
| Flowline symbol in flowchart of programming | Flow line | Used to indicate the flow of logic by connecting symbols. |
| Terminal symbol in flowchart of programming | Terminal(Stop/Start) | Used to represent start and end of flowchart. |
| Input/Output symbol in flowchart of programming | Input/Output | Used for input and output operation. |
| Processing symbol in flowchart of programming | Processing | Used for arithmetic operations and data-manipulations. |
| Decision making symbol in flowchart of programming | Decision | Used to represent the operation in which there are two alternatives, true and false. |
| On-page connector symbol in flowchart of programming | On-page Connector | Used to join different flow lines |
| Off-page connector symbol in flowchart of programming | Off-page Connector | Used to connect flowchart portion on different pages. |
|  | Comment | Used to add comments or clarification |
| Predefined process symbol in flowchart of programming | Predefined Process/Function | Used to represent a group of statements performing one processing task. |

**Draw flowchart to find the largest among three different numbers entered by user.**



Or a flowchart to ask for a number from user and multiply with another number and print result as follows:



Examples of flowcharts include [Activity diagram](http://en.wikipedia.org/wiki/Activity_diagram), [Data flow diagram](http://en.wikipedia.org/wiki/Data_flow_diagram) and sequence diagrams etc.

**Exercise 1**

Draw a flow chart to represent the program described in exercise 2 above.

**STRUCTURE CHARTS (module chart, hierarchy chart)**

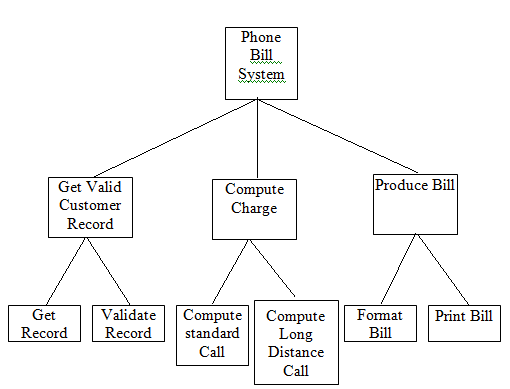
A structure chart is a [**top-down modular design**](http://en.wikipedia.org/wiki/Top-down_design) **tool**, constructed of squares representing the different modules in the [system](http://en.wikipedia.org/wiki/System), and lines that connect them. **A structure chart is a graphic depiction of the decomposition of a program.** It is a tool to aid in software *design* but is particularly helpful on large programs.

A structure chart illustrates the division of a program into sub-programss and shows the **hierarchical relationships among the parts/functions**. A classic "organization chart" for a company is an example of a structure chart.

The top of the chart is a box representing the entire problem, the bottom of the chart shows a number of boxes representing the less complicated sub-problems (e.g. Phone Bill System).

A structure chart is NOT a flowchart. It has nothing to do with the logical sequence of tasks. It does NOT show the order in which tasks are performed. It does NOT illustrate an algorithm.

Each block represents some function in the system, and thus should contain a verb phrase, e.g. "Print report."

****

**EXERCISE**

Draw a structure chart for an electricity billing system.

**Decision Tables**

Decision tables provide a handy and **compact way to** **represent complex business logic**. A decision table is an excellent tool to use in both testing and requirements management.

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, it is easy to see this.

In a decision table, business logic is divided into **conditions**, **actions** (decisions) and **rules** for representing the various components that form the business logic. Each column in the table corresponds to a rule in the business logic that describes the unique combination of circumstances that will result in the actions.

There is one row for each condition and each vertical column for each combination of values and resulting actions. **Conditions** are the factors to consider when making certain business decisions **Actions** are the possible actions to take when a condition is met or a certain business decision is made.

Each vertical column of a decision table is called a **rule** and each rule symbolizes the combinations of condition(s) and action(s) that form the business decision. If constructed properly, the decision table has a rule to cover every combination. Rules are made of **selectors** symbolized by Y (Yes), N (No) and – (for redundant or irrelevant rules).

**COMPLETING THE TABLE**

* If the number of conditions identified is n, then the number of possible rules is found by applying the formula; Number of **rules = 2n** where n is the number of conditions.
* Entries opposite the lowest condition should be completed first using Y an N alternately until all the vertical rules have been dealt with.
* Entries opposite the second lowest condition should be completed using Y an N in pairs until all the vertical rules have been dealt with.
* Entries for the next condition are then completed next using Ys and Ns in fours.
* This process continues using twice the number of Ys and Ns each time until all conditions are completed.
* Once entered, the rules are read vertically in each column and an X entered at the action that appropriately completes that rule. Actions which are mutually exclusive can be combined on a single line.

**EXAMPLE**

A student who passes the examinations and completes the coursework and project satisfactorily is awarded a pass. If the course work and the project are unsatisfactory, the student is asked to resubmit the unsatisfactory work, as long as the exams have been passed. A student who fails the examinations is deemed to have failed the whole course unless both the course work and the project are satisfactory, in which case the student is allowed to re-sit the examination.

**SOLUTION**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Before an overall pass** | | | | | | | | | |
|  |  | **Rules** | | | | | | | |
| **Conditions** | Exams Passed? | Y | Y | Y | Y | N | N | N | N |
| Completed Course work? | Y | Y | N | N | Y | Y | N | N |
|  | Completed Project? | Y | N | Y | N | Y | N | Y | N |
| **Actions** | Pass | X |  |  |  |  |  |  |  |
| Re-sit Exam |  |  |  |  | X |  |  |  |
| Resubmit Coursework |  |  | X | X |  |  |  |  |
| Resubmit Project |  | X |  | X |  |  |  |  |
| Failed |  |  |  |  |  | X | X | X |

**ASSIGNMENT 1**

Candidates are accepted for permanent employment if they pass an interview and their qualifications and referees are satisfactory. If they pass the interview and the qualifications or referees (but not both) are unsatisfactory, a job for a probationary period is offered. In all other circumstances, the candidate’s application is rejected.

**Assignment 2**

* Design the business logic (at least three conditions) that is applicable when a customer is applying for a bank loan.
* Draw a decision table to represent the business logic.

***CHAPTER 3***

# 3. PROGRAM STRUCTURE

## STRUCTURE OF A C PROGRAM

The [**C programming language**](http://en.wikipedia.org/wiki/C_(programming_language)) was designed by [Dennis Ritchie](http://en.wikipedia.org/wiki/Dennis_Ritchie) as a systems programming language for Unix.

A C program basically has the following structure:

* Preprocessor Commands/directive
* Functions
* Variable declarations
* Statements & Expressions
* Comments

Example:

#include <stdio.h>

int main()

{

/\* My first program\*/

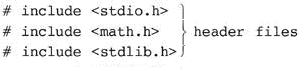
printf("Hello, World! \n");

return 0;

}

**Preprocessor Commands**

These commands tell the compiler to do **preprocessing before doing actual compilation**. Like *#include <stdio.h>* is a preprocessor command which tells a C compiler to include **stdio.h** file before going to actual compilation. The standard input and output header file (stdio.h) allows the program to interact with the screen, keyboard and file system of the computer.



NB/ Preprocessor directives are not actually part of the C language, but rather instructions from you to the compiler.

**Functions**

Theseare main building blocks of any C Program. Every C Program will have one or more functions and there is one mandatory function which is called *main()* function. When this function is prefixed with keyword *int*, it means this function returns an integer value when it exits. This integer value is retuned using *return* statement.

The C Programming language provides a set of built-in functions. In the above example printf() is a C built-in function which is used to print anything on the screen.

**A function is a group of statements that together perform a task**. A C program can be divide up into separate functions but logically the division usually is so each function performs a specific task. A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

The general form of a function definition in C programming language is as follows:

**return\_type function\_name( parameter list )**

**{**

**body of the function/Function definition**

**}**

**Variable Declarations**

In C, all variables must be declared before they are used. Thus, C is a **strongly typed** programming language. Variable declaration ensures that appropriate memory space is reserved for the variables. Variables are used to hold numbers, strings and complex data for manipulation e.g.

Int x;

Int num; int z;

**Statements & Expressions**

**Expressions** combine variables and constants to create new values e.g.

x + y;

**Statements** in C are **expressions, assignments, function calls, or control flow** statements which make up C programs.

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 an expression on the right.

z = x + y;

**Comments**

These are **non-executable program statements** meant to **enhance program readability** and allow **easier program maintenance**- they **document the program**. They are **ignored by the compiler**. Theseare used to give additional useful information inside a C Program. All the comments will be put inside /\*...\*/ or // for single line comments as given in the example above. A comment can span through multiple lines.

/\* Author: Mzee Moja \*/

or

/\*

\* Author: Mzee Moja

\* Purpose: To show a comment that spans multiple lines.

\* Language: C

\*/

or

Fruit = apples + oranges; // get the total fruit

**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**.

They include:

\a Alert/bell

\b Backspace

\n New line

\v Vertical tab

\t Horizontal tab

\\ Back slash

\” Double quote

\0 Null

**Note the following**

* C is a **case sensitive** programming language. It means in C *printf* and *Printf* will have different meanings.
* End of each C statement must be marked with a semicolon.
* Multiple statements can be on the same line.
* Any combination of spaces, tabs or newlines is called a **white space**. C is a **free-form** language as the C compiler chooses to ignore whitespaces. Whitespaces are allowed in any format to improve readability of the code. **Whitespace is the term used in C to describe blank lines, tabs, newline characters and comments.**Statements in C can be written from any position. Unlike in some languages like COBOL, where each field/section needs to be started at a prefefined position.
* Statements can continue over multiple lines.
* A C **identifier** is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore \_ followed by zero or more letters, underscores, and digits (0 to 9). C does not allow punctuation characters such as @, $, and % within identifiers.
* **A keyword is a reserved word** in C. Reserved words may not be used as constants or variables or any other identifier names

SAMPLE PROGRAM

//First program

#include<stdio.h>

main()

{

int num; // Declaration

num =1; // Assignment statement

printf(" My favorite number is %d because", num);

printf(" it is first.\n");

return 0;

}

The program will output (print on screen) the statement “**My favorite number is 1 because it is first**”.

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.

List of Keywords (Reserved Words)

The following list shows the **reserved words** in C. These reserved words may not be used as constants or variables or any other identifier names.

|  |  |  |  |
| --- | --- | --- | --- |
| auto | else | Long | switch |
| break | enum | register | typedef |
| case | extern | return | union |
| char | float | short | unsigned |
| const | for | signed | void |
| continue | goto | sizeof | volatile |
| default | if | static | while |
| do | int | struct | \_packed |
| double | | | |

## SOURCE CODE FILES

When you write a program in C language, **your instructions** form the source code/file. C files have an extension **.c**. The part of the name 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. You can compile individual modules and then combine modules later.

**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.

The startup code is a set of execution startup routines [linked](https://en.wikipedia.org/wiki/Linker_(computing)) into a [C](https://en.wikipedia.org/wiki/C_(programming_language)) program that performs any initialization work required before calling the program's [main function](https://en.wikipedia.org/wiki/Main_function).

NB/

* + An **interpreter** unlike a **compiler** is a computer program that directly executes, i.e. performs, instructions written in a programming language, without previously compiling them into a machine language program.
  + If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a **cross-compiler**.
  + A program that translates from a low level language to a higher level one is a **decompiler**.
  + A program that translates between high-level languages is usually called a **source-to-source compiler** or **transpiler**.

## STEP BY STEP CREATION AND EXECUTION OF A C PROGRAM

Step1: Creating and editing program

1. Write C program using a text editor
2. Save program using **.c** extension i.e. the Source Code

Step 2: Compiling

1. The source code is converted into machine equivalent instruction
2. Compiler checks for errors. If the source code is error free, the source code is converted to object code. If there are errors, the compiler will report them.

Step 3: Linking Libraries

1. Program is linked with included header files. This is done by the linker.
2. Other libraries are added to the program to generate the executable program.

**Library Functions**

There is a minimal set of library functions that should 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()** which is a general purpose output function. The quoted string between the parenthesis of the printf() function is called an argument.

**Printf(“This is a C program\n”)**

The \n at the end of the text is an **escape sequence** tells the program to print a new line as part of the output.

## C DATA TYPES

In the C programming language, data types refer to a system used for declaring variables or functions of different types. **A data type is, therefore, a data storage format that can contain a specific type or range of values**. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

The basic data types in C are as follows:

|  |  |
| --- | --- |
| **Type** | **Description** |
| **Char** | Character data and is used to hold a single character. A character can be a letter, number, space, punctuation mark, or symbol - 1 byte long |
| **Int** | A signed whole number in the range -32,768 to 32,767 - 2 bytes long |
| **Float** | A real number (that is, a number that can contain a fractional part) – 4 bytes |
| **Double** | A double-precision floating point value. Has more digits to the right of the decimal point than a float – 8 bytes |
| **Void** | Represents the absence of type. i.e. represents “no data” |

**USING C’S DATA TYPE MODIFIERS**

The five basic types (int, float, char,double and void) can be **modified** **to your specific need** using the following specifiers.

* **Signed**

Signed Data Modifier implies that the data type variable can store positive values as well as negative values.

The use of the modifier with integers is redundant because the default integer declaration assumes a signed number. The signed modifier is used with char to create a small signed integer. Specified as signed, a char can hold numbers in the range -128 to 127.

* **Unsigned**

If we need to change the data type so that it can only store positive values, “unsigned” data modifier is used.

This can be applied to char and int. When char is unsigned, it can hold positive numbers in the range 0 to 255.

* **Long**

Sometimes while coding a program, we need to increase the Storage Capacity of a variable so that it can store values higher than its maximum limit which is there as default.

This can be applied to both **int** and **double.** When applied to **int,** it doubles its **length**, in bits, of the base type that it modifies. For example, an integer is usually 16 bits long. Therefore a **long int** is 32 bits in length. When **long** is applied to a double, it roughly doubles the precision.

* **Short**

A “short” type modifier does just the opposite of “long”. If one is not expecting to see high range values in a program.

For example, if we need to store the “age” of a student in a variable, we will make use of this type qualifier as we are aware that this value is not going to be very high

The type modifier precedes the type name. For example this declares a **long integer.**

**short int age;**

**Integer Types**

Following table gives you details about standard integer types with its storage sizes and value ranges:

|  |  |  |
| --- | --- | --- |
| **Type** | **Storage size** | **Value range** |
| Char | 1 byte | -128 to 127 or 0 to 255 |
| unsigned char | 1 byte | 0 to 255 |
| signed char | 1 byte | -128 to 127 |
| Int (16 or 32 bit compiler) | 2 or 4 bytes | -32,768 to 32,767 or -2147483648 to 2147483647. |
| unsigned int | 2 or 4 bytes | 0 to 65,535 or 0 to 4,294,967,295 |
| Short int | 2 bytes | -32,768 to 32,767 |
| unsigned short | 2 bytes | 0 to 65,535 |
| Long int | 4 bytes | -2,147,483,648 to 2,147,483,647 |
| unsigned long | 4 bytes | 0 to 4,294,967,295 |

Floating-Point Types

Following table gives you details about standard floating-point types with storage sizes and value ranges and their precision:

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Storage size | Value range | Precision |
| float | 4 byte | 1.2E-38 to 3.4E+38 | 6 decimal places |
| double | 8 byte | 2.3E-308 to 1.7E+308 | 15 decimal places |
| long double | 10 byte | 3.4E-4932 to 1.1E+4932 | 19 decimal places |

The void Type

The void type specifies that no value is available. It is used in three kinds of situations:

|  |  |
| --- | --- |
|  | Types and Description |
| 1 | Function returns as void. There are various functions in C which do not return value or you can say they return void. A function with no return value has the return type as void. For example, void exit (int status); |
| 2 | Function arguments as void. There are various functions in C which do not accept any parameter. A function with no parameter can accept as a void. For example, int rand(void); |
| 3 | Pointers to void A pointer of type void \* represents the address of an object, but not its type. For example, a memory allocation function void \*malloc( size\_t size ); returns a pointer to void which can be casted to any data type. |

## 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 i.e. C is strongly typed.

**Variable Declaration**

**Declaring** a variable tells the compiler to reserve space in memory for that particular variable. A **variable definition** specifies a data type and the variable name and contains a list of one or more variables of that type .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;

int i, j, k;

char c, ch;

float f, salary;

Variables can be initialized 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\*/

**TYPES OF VARIABLES**

The Programming language C has two main variable types

* Local Variables
* Global Variables

**Local Variables**

A local variable is a variable that is declared inside a function.

* Local variables scope is confined within the block or function where it is defined. Local variables must always be defined at the top of a block.
* When execution of the block/module starts the variable is available, and when the block ends the variable 'dies'.

**Global Variables**

Global variable is defined at the top of the program file and it can be visible and modified by any function that may reference it. Global variables are declared outside **all** functions.

**Sample Program.**

#include <stdio.h>

int area; //global variable

int main ()

{

int a, b; //local variable

/\* actual initialization \*/

a = 10;

b = 20;

printf("\t Side a is %d cm and side b is %d cm long\n",a,b);

area = a\*b;

printf("\t The area of your rectangle is : %d \n", area);

return 0;

}

**Variable Names**

Every variable has a name and a value. The name identifies the variable and the value stores data. Every variable name in C must start with a letter; the rest of the name can consist of letters, numbers and underscore characters. C is case sensitive i.e. it recognizes upper and lower case characters as being different. You cannot use any of C’s keywords like main, while, switch etc as variable names,

Examples of legal variable names:

X result outfile x1 out\_file etc

It is conventional in C not to use capital letters in variable names. These are used for names of constants.

**Declaration vs Definition**

A declaration provides basic attributes of a symbol: its type and its name. A definition provides all of the details of that symbol--if it's a function, what it does; if it's a class, what fields and methods it has; if it's a variable, where that variable is stored. Often, the compiler only needs to have a declaration for something in order to compile a file into an object file, expecting that the linker can find the definition from another file. If no source file ever defines a symbol, but it is declared, you will get errors at link time complaining about undefined symbols. In the following short code, the definition of variable x means that the storage for the variable is that it is a global variable.

int x;

int main()

{

x = 3;

}

**Inputting Numbers From The Keyboard Using Scanf()**

Variables can also be initialized during program execution (run time). The **scanf()** function is used to read values from the keyboard. For example, to read an integer value use the following general form:

scanf(“%d”, &var\_name)

As in

scanf(“%d”, &num)

The %d is a **format specifier** which tells the compiler that the second argument will be receiving an integer value.

The & preceding the variable name means “address of”. The function allows the function to place a value into one of its arguments.

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

|  |  |
| --- | --- |
| %c | Read a single character |
| %d | Read an integer |
| %f | Read a floating point number |
| %lf | Read a double |
| %s | Read a string |
| %u | Read a an unsigned integer |

When used in a printf() function, a type specifier informs the function that a different type item is being displayed.

**SAMPLE PROGRAM USING SCANF()**

#include <stdio.h>

int area; //global variable

int main ()

{

int a, b; //local variables

/\* actual initialization \*/

printf("Enter the value of side a: ");

scanf("%d", &a);

printf("Enter the value of side b: ");

scanf("%d", &b);

printf("\n");

printf("\t You have entered %d for side a and %d for side b\n", a, b);

area = a\*b;

printf("\t The area of your rectangle is : %d \n", area);

return 0;

}

## CONSTANTS

C allows you to declare constants. When you declare a constant it is a bit like a variable declaration except the value cannot be changed during program execution.

The **const** keyword is used to declare a constant, as shown below:

int const A = 1;

const int A = 5;

const char name= ‘c’;

These fixed values are also called **literals**.

Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are also enumeration constants as well.

The constants are treated just like regular variables except that their values cannot be modified after their definition.

## TYPE CASTING

Type casting is a way to convert a variable from one data type to another. For example, if you want to store a long value into a simple integer then you can type cast long to int. You can convert values from one type to another explicitly using the **cast operator** as follows:

(type\_name) expression

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating-point operation:

#include <stdio.h>

main()

{

int sum = 17, count = 5;

double mean;

mean = (double) sum / count;

printf("Value of mean is %lf \n", mean );

}

When the above code is compiled and executed, it produces the following result:

Value of mean: 3.400000

It should be noted here that the cast operator has precedence over division, so the value of **sum** is first converted to type **double** and finally it gets divided by count yielding a double value.

**Type conversions can be** implicit which is performed by the compiler automatically, or it can be specified **explicitly** through the use of the **cast operator**. It is considered good programming practice to use the cast operator whenever type conversions are necessary.

## C PROGRAMMING OPERATORS

Operator is the symbol which operates on a value or a variable (operand). For example: + is an operator to perform addition.

C programming language has a wide range of operators to perform various operations. For better understanding of operators, these operators can be classified as:

**OPERATORS IN C PROGRAMMING**

1. [Arithmetic Operators](http://www.programiz.com/c-programming/c-operators#arithmetic)
2. [Increment and Decrement Operators](http://www.programiz.com/c-programming/c-operators#increment)
3. [Assignment Operators](http://www.programiz.com/c-programming/c-operators#assignment)
4. [Relational Operators](http://www.programiz.com/c-programming/c-operators#relational)
5. [Logical Operators](http://www.programiz.com/c-programming/c-operators#logical)
6. [Conditional Operators](http://www.programiz.com/c-programming/c-operators#conditional)
7. [Bitwise Operators](http://www.programiz.com/c-programming/c-operators#bitwise)
8. [Special Operators](http://www.programiz.com/c-programming/c-operators#other)

### ARITHMETIC OPERATORS

Assume variable A holds 10 and variable B holds 20 then

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Adds two operands | A + B will give 30 |
| - | Subtracts second operand from the first | A - B will give -10 |
| \* | Multiplies both operands | A \* B will give 200 |
| / | Divides numerator by de-numerator | B / A will give 2 |
| % | Modulus Operator - remainder of after an integer division | B % A will give 0 |

**Note:** % operator can only be used with integers.

### INCREMENT AND DECREMENT OPERATORS – Unary Operators

In C, **++** and **--** are called increment and decrement operators respectively. Both of these operators are **unary operators**, i.e, used on single operand. ++ adds 1 to operand and -- subtracts 1 to operand respectively. For example:

Let a=5

a++; //a becomes 6

a--; //a becomes 5

++a; //a becomes 6

--a; //a becomes 5

[**Difference between ++ and -- operator as postfix and prefix**](http://www.programiz.com/article/increment-decrement-operator-difference-prefix-postfix)

When i++ is used as prefix(like: ++var), ++var will increment the value of var and then return it but, if ++ is used as postfix(like: var++), operator will return the value of operand first and then increment it. This can be demonstrated by an example:

#include <stdio.h>

int main(){

int c=2;

printf("%d\n",c++); /\*this statement displays 2 then, only c incremented by 1 to 3.\*/

printf("%d",++c); /\*this statement increments 1 to c then, only c is displayed.\*/

return 0;

}

**Output**

20

4

### ASSIGNMENT OPERATORS – Binary Operators

The most common assignment operator is =. This operator assigns the value in the right side to the left side. For example:

var=5 //5 is assigned to var

a=c; //value of c is assigned to a

5=c; // Error! 5 is a constant.

| **Operator** | **Example** | **Same as** |
| --- | --- | --- |
| = | a=b | a=b |
| += | a+=b | a=a+b |
| -= | a-=b | a=a-b |
| \*= | a\*=b | a=a\*b |
| /= | a/=b | a=a/b |
| %= | a%=b | a=a%b |

NB/ += means Add and Assign etc.

### RELATIONAL OPERATORS - Binary Operators

Relational operators check relationship between two operands. If the relation is true, it returns value 1 and if the relation is false, it returns value 0. For example:

a>b

Here, > is a relational operator. If a is greater than b, a>b returns 1 if not then, it returns 0.

Relational operators are used in decision making and loops in C programming.

| **Operator** | **Meaning of Operator** | **Example** |
| --- | --- | --- |
| = = | Equal to | 5= =3 returns false (0) |
| > | Greater than | 5>3 returns true (1) |
| < | Less than | 5<3 returns false (0) |
| != | Not equal to | 5!=3 returns true(1) |
| >= | Greater than or equal to | 5>=3 returns true (1) |
| <= | Less than or equal to | 5<=3 return false (0) |

### LOGICAL OPERATORS - Binary Operators

**Logical operators are used to combine expressions containing relational operators.** In C, there are 3 logical operators:

| **Operator** | **Meaning of Operator** | **Example** |
| --- | --- | --- |
| && | Logical AND | If c=5 and d=2 then,((c= =5) && (d>5)) returns false. |
| || | Logical OR | If c=5 and d=2 then, ((c= =5) || (d>5)) returns true. |
| ! | Logical NOT | If c=5 then, !(c= =5) returns false. |

The following table shows the result of operator && evaluating the expression a&&b:

|  |  |  |
| --- | --- | --- |
| && OPERATOR (and) | | |
| a | b | a && b |
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |

The operator || corresponds to the Boolean logical operation OR, which yields true if either of its operands is true, thus being false only when both operands are false. Here are the possible results of a || b:

|  |  |  |
| --- | --- | --- |
| || OPERATOR (or) | | |
| a | b | a || b |
| true | true | true |
| true | false | true |
| false | true | true |
| false | false | false |

**Explanation**

For expression, ((c==5) && (d>5)) to be true, both c==5 and d>5 should be true but, (d>5) is false in the given example. So, the expression is false. For expression ((c==5) || (d>5)) to be true, either the expression should be true.

Since, (c==5) is true. So, the expression is true. Since, expression (c==5) is true, !(c==5) is false.

### 

### CONDITIONAL OPERATOR – Ternary Operators

Conditional operator takes three operands and consists of two symbols **?** and **:** . Conditional operators are used for decision making in C. For example:

d=(c>0)?10:-10;

If c is greater than 0, value of c will be 10 but, if c is less than 0, value of c will be -10.

**BITWISE OPERATORS**

Bitwise operators work on bits and performs bit-by-bit operation.



## PRECEDENCE OF OPERATORS

If more than one operator is involved in an expression then, C language has a **predefined rule of priority of operators**. This rule of priority of operators is called **operator precedence**.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. **Within an expression, higher precedence operators will be evaluated first.**

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [ ] ++ - - | Left to right |
|  |  |  |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | = = != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %=>>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

## ASSOCIATIVITY OF OPERATORS

Associativity indicates in which order two operators of same precedence (priority) executes. Let us suppose an expression:

a= =b!=c

Here, operators == and != have the same precedence. The associativity of both == and != is **left to right**, i.e., the expression in left is executed first and execution take pale towards right. Thus, a==b!=c equivalent to :

(a= =b)!=c

Operators may be **left-associative** (meaning the operations are grouped from the left), **right-associative (**meaning the operations are grouped from the right)

|  |  |  |
| --- | --- | --- |
| **Operator Name** | **Associativity** | **Operators** |
| Multiplicative | left to right | \*  /  % |
| Additive | left to right | +  - |
| Bitwise Shift | left to right | <<  >> |
| Relational | left to right | <  >  <=  >= |
| Equality | left to right | ==  != |
| Bitwise AND | left to right | & |
| Bitwise Exclusive OR | left to right | ^ |
| Bitwise Inclusive OR | left to right | | |
| Logical AND | left to right | && |
| Logical OR | left to right | || |
| Conditional | right to left | ? : |
| Assignment | right to left | =  +=  -=  \*=   /=  <<=  >>=  %=   &=  ^=  |= |
| Comma | left to right | , |

**MORE SAMPLE PROGRAMS**

1. Program to return the square and square root of a number.

#include <stdio.h>

#include <math.h>

int main ()

{

int num, sqr; double squarert;

printf("Key in a number to get its square root: ");

scanf("%d",&num);

squarert=sqrt(num);

sqr=num\*num;

printf("Square root of \"%d\" is \"%lf\"\n",num,squarert);

printf("The square of the number is %d\n",sqr);

return(0);

}

1. Program to return a number raised to the power of another number.

#include<stdio.h>

#include<math.h>

void main()

{

int num, power;

double result;

printf("Key a number : ");

scanf("%d",&num);

printf("Key the power : ");

scanf("%d",&power);

result=pow(num,power);

printf("%d raised to power %d is %lf",num,power,result);

}

***Chapter 4***

# CONTROL STRUCTURES

**Definition**

**Control structures** represent the forms by which statements in a program are executed. **Flow of control** refers to the order in which the individual statements, instructions or function calls of a program are executed or evaluated.

## IMPORTANCE OF CONTROL STRUCTURES

Generally, a program should execute the statements one by one until the defined end. This type of a program structure is called **sequential structure**. The functionality of this type of program is limited since it flows in a single direction. However, all high-level programming languages enable the programmer to change the flow of program execution. This is done by the use of control structures whose main benefits are to enable **decision making** and **repetition** as well asgiving the power to do far more **complex processing** and **provide** **flexibility with logic.** The sophisticated logic is necessary for a program to solve complex problems.

The kinds of control flow statements supported by different languages vary, but can be categorized by their effect:

* continuation at a different statement i.e. unconditional jump e.g. GoTo statements
* executing a set of statements only if some condition is met i.e. choice
* executing a set of statements zero or more times, until some condition is met i.e. loop
* executing a set of distant statements, after which the flow of control usually returns e.g. subroutines/functions

## TYPES OF CONTROL STRUCTURES

There are three types in C:

1. **Sequence structures**

Program statements are executed in the sequence in which they appear in the program.

1. **Selection structures/Decision Structures**

Statement block is executed only if some condition is met. These structures include **if**, **if**/**else**, and **switch.** Selection structures are extensively used in programming because they allow the program to decide an action based upon user's input or other processes for instance in password checking.

1. **Repetition/Iterative/looping structures**

This is where a group of statements in a program may have to be executed repeatedly until some condition is satisfied.These include **while**, **do**/**while** and **for**

## SELECTION STRUCTURES

## THE IF SELECTION STRUCTURE

* + Used to choose among alternative courses of action i.e. the if statement provides a junction at which the program has to select which path to follow. The **if selection** performs an action only if the condition is **true,**

General form

*If (expression)*

*statement*

**Pseudocode**:

*If student’s marks is greater than or equal to 600  
Print “Passed”*

As in

if (marks>=600)

printf(“Passed”);

If condition is **true**

* + Print statement executed and program goes on to next statement
  + If **false**, print statement is ignored and the program goes onto the next statement

**NB/ Indenting makes programs easier to read**

true

false

grade >= 60

print “Passed”

Flow chart for the **if** selection structure

NB/ The statement in the if structure can be a single statement or a block (Compound statement).

If it’s a block of statements, it must be marked off by braces.

*if (expression)*

*{*

*Block of statements*

*}*

As in

If (salary>5000)

{

VAT = salary \* 0.16;

printf(“Tax charged is %f”, VAT);

}

## THE IF/ELSE

While **if** only performs an action if the condition is **true, if**/**else s**pecifies an action to be performed both when the condition is **true** and when it is **false. E.g.**

**Pseudocode:**

*If student’s grade is greater than or equal to 60  
Print “Passed”*

*else  
Print “Failed”*

true

false

print “Failed”

print “Passed”

grade >= 60

Flow chart for the **if**/**else** selection structure

**Example**

if (x >=100)

{

printf(“Let us increment x:\n”);

x++;

}

else

printf(“x < 0 \n);

## THE IF...ELSE IF...ELSE STATEMENT

* + Test for multiple cases/conditions.
  + Once a condition is met, the other statements are skipped
  + Deep indentation usually not used in practice

**Pseudocode for an if..else if..else structure**

*If student’s grade is greater than or equal to 90  
 Print “A”*

*Else If student’s grade is greater than or equal to 80  
 Print “B”  
 else If student’s grade is greater than or equal to 70   
 Print “C”  
 else If student’s grade is greater than or equal to 60   
 Print “D”  
 else  
 Print “F”*

**Example**

#include <stdio.h>

main()

{

int marks;

printf("Please enter your MARKS:");

scanf("%d", &marks);

if (marks>=90 && marks <=100)

printf("Your grade is A\n");

else if (marks>=80 && marks <=89)

printf("Your grade is B\n");

else if (marks>=70 && marks <=79)

printf("Your grade is C\n");

else if (marks>=60 && marks <=69)

printf("Your grade is D\n");

else if (marks >100)

printf("Marks out of range\n");

else

printf("Your grade is F\n");

}

## NESTED IF STATEMENTS

One **if** or **else if** statement can be used inside another **if** or **else if** statement(s).

**Syntax**

The syntax for a **nested if** statement is as follows:

if (boolean\_expression 1)

{

/\* Executes when the boolean expression 1 is true \*/

if(boolean\_expression 2)

{

/\* Executes when the boolean expression 2 is true \*/

}

}

You can nest else **if...else** in the similar way as you have **nested if** statement.

Example

#include <stdio.h>

int main ()

{

/\* local variable definition \*/

int a = 100;

int b = 200;

/\* check the boolean condition \*/

if( a = = 100 )

{

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

if( b = = 200 )

{

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

printf("Value of a is 100 and b is 200\n" );

}

}

return 0;

}

When the above code is compiled and executed, it produces the following result:

Value of a is 100 and b is 200

Exact value of a is : 100

Exact value of b is : 200

## SWITCH STATEMENT

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a **case**, and the variable being switched on is checked for each switch case.

**Syntax**

The syntax for a switch statement in C programming language is as follows:

switch(expression)

{

case constant-expression:

statement(s);

break;

case constant-expression :

statement(s);

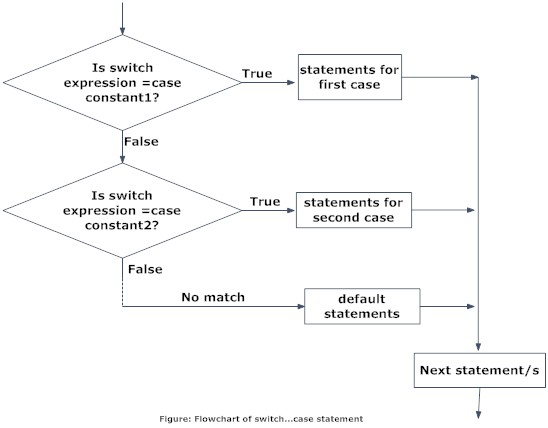
break;

/\* you can have any number of case statements \*/

default :

statement(s);

}



**The following rules apply to a switch statement:**

1. You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
2. The constant-expression for a case must be the same data type as the variable in the switch
3. When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
4. When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
5. Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
6. A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

**Sample Switch Statement**

#include<stdio.h>

Int main()

{

char grade;

printf("Enter your grade:");

scanf("%c", &grade);

switch (grade)

{

case 'A':

printf("Excellent!\n");

break;

case 'B':

printf("Very Good!\n");

break;

case 'C':

printf("Good!\n");

break;

case 'D':

printf("Work harder!\n");

break;

default:

printf("Fail!\n");

}

return 0;

}

## NESTED SWITCH STATEMENTS

It is possible to have a **switch** as part of the statement sequence of an **outer switch**. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

Syntax

The syntax for a **nested switch** statement is as follows:

switch(ch1) {

case 'A':

printf("This A is part of outer switch" );

switch(ch2) {

case 'A':

printf("This A is part of inner switch" );

break;

case 'B':

}

break;

case 'B':

}

Example

#include <stdio.h>

int main ()

{

/\* local variable definition \*/

int a = 100;

int b = 200;

switch(a) {

case 100:

printf("This is part of outer switch\n", a );

switch(b) {

case 200:

printf("This is part of inner switch\n", a );

printf(“A is equals to %d and B is equals to %d”, a, b);

}

}

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

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

return 0;

}

When the above code is compiled and executed, it produces the following result:

This is part of outer switch

This is part of inner switch

A is equals to 100 and B is equals to 200

Exact value of a is : 100

Exact value of b is : 200

## REPETITION/ITERATIVE/LOOP STRUCTURES

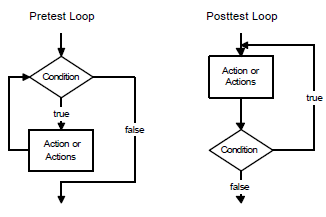
A loop statement allows the execution of a statement or a group of statements multiple times until a condition either tests true or false. There are two types of loops: **Pre-test** and **post-test** loops.

In a pretest loop, a logical condition is checked before each repetition to determine if the loop should terminate. These loops include:

– *while loop*

*– for loop*

Post-test loops check a logical condition after each repetition for termination. The do-while loop is a *post*-*test loop*.



## WHILE LOOP IN C

A **while** loop statement repeatedly executes a target statement **as long as** a given condition is **true**.

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

while(condition)

{

statement(s);

update expression

}

The statement(s) may be a single statement or a block of statements. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

Example

#include <stdio.h>

int main ()

{

/\* local variable definition \*/

int a = 10; //loop index

/\* while loop execution \*/

while( a < 20 )

{

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

a++;

}

return 0;

}

When the above code is compiled and executed, it produces the following result:

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

## FOR LOOP IN C

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

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

for ( initial expression; test expression/logical condition; update expression )

{

statement(s);

}

Here is the flow of control in a for loop:

1. This step initializes any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
2. Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
3. After the body of the for loop executes, the flow of control jumps back up to the update expression. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself. After the condition becomes false, the for loop terminates.

Flow Diagram



Example

#include <stdio.h>

int main ()

{

int a;//loop index

/\* for loop execution \*/

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

{

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

}

return 0;

}

When the above code is compiled and executed, it produces the following result:

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

## DO...WHILE LOOP IN C – Post Test Loop

Unlike for and while loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming language checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except that a **do...while** loop is guaranteed to execute at least one time. The structure, therefore loops until a condition tests false i.e. loop until.

Syntax

do

{

statement(s);

update expression;

}while( condition );

If the condition is **true**, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes **false**.



**Example**

#include <stdio.h>

int main ()

{

/\* local variable definition \*/

int a = 10;

/\* do loop execution \*/

do

{

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

a = a + 1;

}while( a < 20 );

return

When the above code is compiled and executed, it produces the following result:

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

## NESTED LOOPS IN C

C programming language allows the use of one loop inside another loop. The following section shows a few examples to illustrate the concept.

Syntax

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 );

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example, a **for loop** can be inside a while loop or vice versa.

**Example**

#include <stdio.h>

int main()

{

int n, c, k;

printf("Enter number of rows:");

scanf("%d",&n);

for ( c = 1 ; c <= n ; c++ )

{

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

{

printf("%d",k);

}

printf("\n");

}

return 0;

}

Result:

If the user interred 5 as the number of rows, the output would be:

1

1 2

1 2 3

1 2 3 4

1 2 3 4 5

**TERMINATING LOOPS**

• Counter-controlled loops - a loop controlled by a counter variable, generally where the number of times the loop will execute is known ahead of time especially in *for loops.*

• Event-controlled loops - loops where termination depends on an event rather than executing a fixed number of times for example when a zero value is keyed in or search through data until an item is found. Used mostly in while loops and do-while loops.

**Using a Sentinel**

• The value -999 is sometimes referred to as a sentinel value. The value serves as the “guardian” for the

termination of the loop. It is a good idea to make the sentinel a constant:

#define STOPNUMBER -999

while (number != STOPNUMBER) ...

## LOOP CONTROL STATEMENTS

## BREAK STATEMENT IN C

The **break** statement has the following two uses:

* 1. When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
  2. It can be used to terminate a case in the **switch** statement.
  3. If you are using **nested loops** (i.e., one loop inside another loop), the **break** statement will stop the execution of the innermost loop and start executing the next line of code after the block.

#include <stdio.h>

int main ()

{

/\* local variable definition \*/

int a = 10;

/\* do loop execution \*/

do

{

if( a = = 15)

{

/\* skip the iteration \*/

break;

}

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

a++;

}while( a < 20 );

return 0;

}

## CONTINUE STATEMENT IN C

The **continue** statement works somewhat like the break statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between.

For the **for** **loop**, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do**...**while** loops, **continue** statement causes the program control to pass to the conditional tests.

Example

//program to demonstrate the working of continue statement in C programming

#include<stdio.h>

main()

{

int j;

for (j=0; j<=8; j++)

{

if (j==4)

{

continue;

}

printf("%d ", j);

}

}

## THE COMMA OPERATOR

We now introduce the comma operator (,) which is used primarily in conjunction with the **f o r** statement.

This operator permits two different expressions to appear in situations where only one expression would

ordinarily be used.

For example, it is possible to write **f o r** ( ***expression 7a, expression 7b; expression 2; expression 3) statement*** where ***expression la*** and ***expression 7b*** are the two expressions, separated by the comma operator, where only one expression ***(expression I)*** would normally appear. These two expressions would typically initialize two separate indices that would be used simultaneously within the **f o r** loop.

Similarly, a **f o r** statement might make use of the comma operator in the following manner.

**f o r** ( ***expression* 7; *expression 2; expression 3a, expression 36) statement***

Here ***expression 3a*** and ***expression 3b,*** separated by the comma operator, appear in place of the usual

single expression. In this application the two separate expressions would typically be used to alter (e.g.,

increment or decrement) two different indices that are used simultaneously within the loop. For example, one index might count forward while the other counts backward.

*#include <stdio.h>*

*int main ()*

*{*

*/\* for loop execution \*/*

*int a,b,product=0,product2;*

*for(a = 10,b=10; a <=20;a++,b--)*

*{*

*product = 10 \* a; product2 = 10 \* b;*

*printf("10 \* %d = %d ", a,product);*

*printf("10 \* %d = %d\n", b,product2);*

*}*

*return 0;*

*}*

## GOTO STATEMENT IN C

A **goto** statement provides an unconditional jump from the **goto** to a labeled statement in the same function.

**NOTE:** Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a **goto** can be rewritten so that it doesn't need the **goto**.

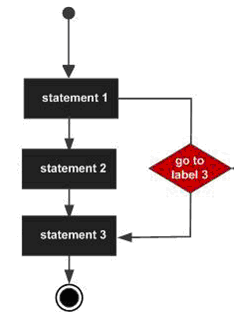
Syntax

The syntax for a **goto** statement in C is as follows:

goto label;

..

.



Example

#include <stdio.h>

int main ()

{

/\* for loop execution \*/

int a,userinput,sum=0;

for(a = 0; a < 5;a++)

{

printf("Enter a number: ");

scanf("%d",&userinput);

if (userinput <1)

goto jump;

sum+=userinput;

}

jump:

printf("The sum of the values is %d\n", sum);

return 0;

}

## THE RETURN STATEMENT

The last of the loop control branching statements is the return statement. The return statement exits from the current function, and control flow returns to where the function was invoked. The return statement has two forms: one that returns a value, and one that doesn't. To return a value, simply put the value (or an expression that calculates the value) after the return keyword.

return count;

The data type of the returned value must match the type of the method's declared return value. When a function is declared void, use the form of return that doesn't return a value.

return;

## THE INFINITE LOOP

A loop becomes **infinite** loop if a condition never becomes **false**. The **for loop** is traditionally used for this purpose. Since **none** of the three expressions that form the **for loop** are required, you can make an endless loop by leaving the conditional expression empty.

#include <stdio.h>

int main ()

{

for( ; ; )

{

printf("This loop will run forever.\n");

}

return 0;

}

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the **for(;;)** construct to signify an infinite loop.

**NOTE**: You can terminate an infinite loop by pressing **Ctrl + C** keys.

***CHAPTER 5***

# SUBPROGRAMS IN C

A sub-program is a series of C statements that perform a specific task in a program. A subprogram can be called within another procedure. Every C program has at least one function, which is **main().** A C program can be divided up into separate functions.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, function strcat() to concatenate two strings, function memcpy() to copy one memory location to another location and many more functions.

A function is known with various names like a method or a sub-routine or a procedure, etc. However, a **function** returns a value while a **procedure** doesn’t: it just executes commands.

A Subprogram is:

* a part of a program that performs one or more related tasks
* has its own name
* written as an independent part of the program

Benefits of using sub-procedures in programming are:

* Sub-programs help to break programs into several but logical sections. The smaller programs/routines make **programming**, **debugging** and subsequent **maintenance** easier.
* They also help in **coding repeated operations** such as frequently used calculations, text etc thus making programming less repetitive and faster.
* Procedures used in one program can act as building blocks for other programs with slight modifications i.e. **code re-use**. Examples include the library functions such as printf() and scanf().
* Programmers working on large projects can divide the workload by making different functions.

## TYPES OF FUNCTIONS

Basically, there are two types of functions in C on basis of whether it is defined by user or not.

* Library functions
* User defined functions

### LIBRARY FUNCTION

Library functions are the in-built functions in C programming system. For example:

**printf()**

- **printf()** is used for displaying output in C.

**scanf()**

- **scanf()** is used for taking input in C.

### USER DEFINED FUNCTION

C allows programmers to define their own function according to their requirements known as user defined functions.

**How user-defined function works in C Programming?**

#include <stdio.h>

void function\_name(){

................

................

}

int main(){

...........

...........

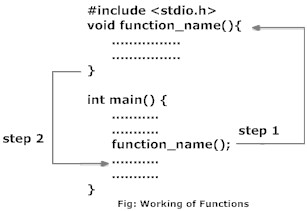
function\_name();

...........

...........

}

As mentioned earlier, every C program begins from main() and program starts executing the codes inside main() function. When the control of program reaches to function\_name() inside main() function. The control of program jumps to void function\_name() and executes the codes inside it. When, all the codes inside that user-defined function are executed, control of the program jumps to the statement just after function\_name() from where it is called. Analyze the figure below for understanding the concept of function in C programming.



Remember, the function name is an identifier and should be unique.

## DEFINING A FUNCTION

The general form of a function definition in C programming language is as follows:

**return\_type function\_name ( parameter list )**

**{**

**body of the function**

**}**

A **function definition** in C programming language consists of a function header and a function body. Here are all the parts of a function:

1. **Return Type:** A function may return a value. The return\_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return\_type is the keyword **void**.
2. **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the **function signature**.
3. **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the **formal parameter**. This value is referred to as **actual parameter** or **argument**. The **parameter list** refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
4. **Function Body:** The function body contains a collection of statements that define what the function does.

**Example**

Following is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum between the two:

/\* function returning the max between two numbers \*/

int max(int num1, int num2)

{

/\* local variable declaration \*/

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

## FUNCTION DECLARATIONS

A **function declaration** tells the compiler about a function name and how to call the function. The actual **body of the function** can be defined separately.

A function declaration has the following parts:

*return\_type function\_name( parameter list );*

For the above defined function max(), following is the function declaration:

*int max(int num1, int num2);*

Parameter names are not important in function declaration; only their type is required, so the following is also valid declaration:

*int max(int, int);*

Function declaration is required when you define a function in one source file and you call that function in another file. In such case you should declare the function at the top of the file calling the function.

## CALLING A FUNCTION

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, program control is transferred to the called function. A called function performs defined task, and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program. Therefore, the calling program is suspended during execution of the called subprogram.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

#include <stdio.h>

/\* function declaration \*/

int max(int num1, int num2);

int main ()

{

/\* local variable definition \*/

int a = 100;

int b = 200;

int ret;

/\* calling a function to get max value \*/

ret = max(a, b);

printf( "Max value is : %d\n", ret );

return 0;

}

/\* function returning the max between two numbers \*/

int max(int num1, int num2)

{

/\* local variable declaration \*/

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

**Ways of paasing values during function calling**

While calling a function, there are two ways that arguments can be passed to a function:

|  |  |
| --- | --- |
| **Pass Type** | **Description** |
| [**Pass by value**](http://www.tutorialspoint.com/cprogramming/c_function_call_by_value.htm) | This method copies the **actual value of an** argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. /any changes made to the value **does not** affect te original value |
| [**Pass by reference**](http://www.tutorialspoint.com/cprogramming/c_function_call_by_reference.htm) | This method copies the **address of an argument** into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.  /any changes made to the value affect te original value |

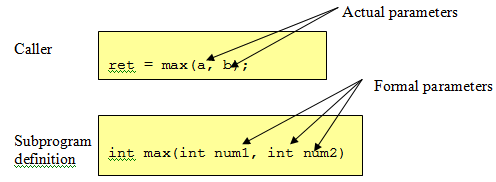
By default, C uses **pass by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

## FUNCTION ARGUMENTS

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

A formal parameter is a dummy variable listed in the subprogram header and used in the subprogram. An actual parameter represents a value used in the subprogram call statement.



When max() is called, we pass it the arguments which the function uses as the values of ret. This process is called **parameter passing**.

\*\*\*\*\*

*Parameters* refers to the list of variables in a method declaration. *Arguments* are the actual values that are passed when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.

## TYPES OF VARIABLES

The Programming language C has two main variable types

* Local Variables
* Global Variables

### LOCAL VARIABLES

**A local variable is a variable that is declared inside a function.**

* Local variables scope is confined within the block or function where it is defined. Local variables must always be defined at the top of a block.
* When execution of the block starts the variable is available, and when the block ends the variable 'dies'.

### GLOBAL VARIABLES

Global variable is defined at the top of the program file and it can be visible and modified by any function that may reference it. Global variables are declared outside **all** functions.

**Sample Program.**

#include <stdio.h>

int area; //global variable

int main ()

{

int a, b; //local variable

/\* actual initialization \*/

a = 10;

b = 20;

printf ("\t Side a is %d cm and side b is %d cm long\n", a, b);

area = a\*b;

printf ("\t The area of your rectangle is : %d \n", area);

return 0;

}

**EXERCISES**

1. **Write a C program to add two integers. Define a function add to add integers and display sum in main() function.**

//main function

#include<stdio.h>

int add(int a, int b);

int main()

{

int a, b, sum;

printf ("Enter the first integer:");

scanf ("%d", &a);

printf("Enter the second integer:");

scanf("%d", &b);

sum = add(a,b);

printf("The sum of the two numbers is %d\n", sum);

}

//start of second function

int add(int a,int b)

{

int sum;

sum = a+b;

return result;

}

1. Write a C program– *max()*- to determine the greater of two integers. Call the function from main() and supply it with two integers and then display the greater of the two.

#include <stdio.h>

/\* function declaration \*/

int max(int, int);

int main ()

{

/\* local variable definition \*/

int a = 100;

int b = 200;

int ret;

/\* calling a function to get max value \*/

ret = max(a, b);

printf( "Max value is : %d\n", ret );

return 0;

}

/\* function returning the max between two numbers \*/

int max(int num1, int num2)

{

/\* local variable declaration \*/

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

***CHAPTER 6***

# DATA STRUCTURES

These refer to **groups of data elements that are organized in a single unit so that they can be used more efficiently** as compared to the simple variables such as integers and strings. Ordinary variables store one value at a time while a data structure will store more than one value at a time in a single variable name.

Data structures are important for **grouping sets of similar data together** **and passing them as one**. For example, if you have a method that prints a set of data but you don't know when writing the procedure how large that set is going to be, you could use an array to pass the data to that method and loop through it.

The data structures available in C include **arrays, Linked Lists, Stacks, Queues, Trees, Graphs, Tables, Sets, Pointers** and **References.**

In Pascal, the data structures include **arrays, records, files** and **sets.**

Data structures can be **classified** using various criteria.

##### Linear

In linear data structures, elements are arranged in linear fashion. A linear data structure traverses the data elements sequentially. The elements in the structure are adjacent to one another other and every element has exactly two neighbour elements to which it is connected. Arrays, linked lists, stacks and queues are examples of linear data structures.

##### Non-Linear

The data values in this structure are not arranged in order but every data item is attached to several other data items in a way that is specific for reflecting relationships. Tree, graph, table and sets are examples of non-linear data structures.

##### Homogenous

In this type of data structures, values of the same types of data are stored, as in an array.

##### Non-homogenous

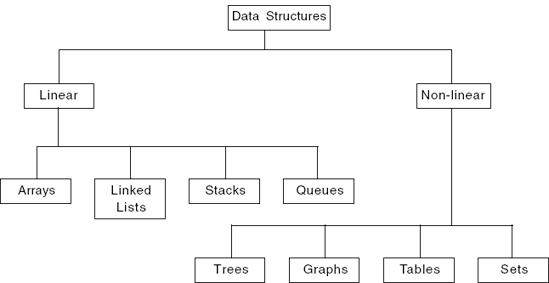
In this type of data structures, data values of different types are grouped together, as in structures and classes.

##### Dynamic

In dynamic data structures such as references and pointers, size and memory locations can be changed during program execution. These data structures can grow and shrink during execution.

##### Static

With a static data structure, the size of the structure is fixed. Static data structures such as arrays are very good for storing a well-defined number of data items.

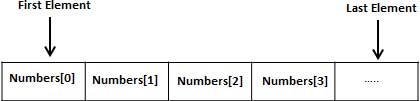


## ARRAYS

An array is a named list of elements, all with the same data type. It is also defined as a consecutive group of memory locations all of which have the same name and the same data type. Arrays store a fixed-size sequential collection of elements of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as *numbers* and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an **index**.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



### DECLARING ARRAYS

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

*type arrayName [ arraySize ];*

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement:

*double balance[10];*

Now *balance* is a variable array which is sufficient to hold up to 10 double numbers.

### INITIALIZING ARRAYS

You can initialize an array in C either one by one or using a single statement as follows:

*double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};*

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

*double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};*

You will create exactly the same array as you did in the previous example.

*balance[4] = 50.0;*

The above statement assigns element number 5 in the array a value of 50.0. Array with 4th index will be 5th ie. last element because all arrays have 0 as the index of their first element which is also called **base index**. Following is the pictorial representation of the same array we discussed above:



### ACCESSING ARRAY ELEMENTS

An element is accessed by **indexing** the array name. This is done by placing the **index** of the element within square brackets after the name of the array. For example:

*double salary = balance[9];*

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

*#include <stdio.h>*

*int main ()*

*{*

*int n[ 9 ]; /\* n is an array of 10 integers \*/*

*int i,j;*

*/\* initialize elements of array n to 0 \*/*

*for ( i = 0; i < 9; i++ )*

*{*

*n[ i ] = i + 100; /\* set element at location i to i + 100 \*/*

*}*

*/\* output each array element's value \*/*

*for (j = 0; j < 10; j++ )*

*{*

*printf("Element[%d] = %d\n", j, n[j] );*

*}*

*return 0;*

*}*

When the above code is compiled and executed, it produces the following result:

*Element[0] = 100*

*Element[1] = 101*

*Element[2] = 102*

*Element[3] = 103*

*Element[4] = 104*

*Element[5] = 105*

*Element[6] = 106*

*Element[7] = 107*

*Element[8] = 108*

*Element[9] = 109*

### SORT TECHNIQUES

1. Bubble sort
2. Selection sort
3. Merge sort
4. Quick sort
5. Exchange sort
6. Shell sort
7. Insertion sort

**Bubble Sort**

In the **bubble sort,**as elements are sorted they gradually "bubble" (or rise) to their proper location in the array, like bubbles rising in a glass of soda.  The bubble sort repeatedly compares **adjacent elements** of an array.  The first and second elements are compared and swapped if out of order.  Then the second and third elements are compared and swapped if out of order.  This sorting process continues until the last two elements of the array are compared and swapped if out of order. 

|  |
| --- |
| BubbleChart |

When this first pass through the array is complete, the bubble sort returns to elements one and two and starts the process all over again. 

The table below follows an array of numbers before, during, and after a bubble sort for*descending* order.  A "pass" is defined as one full trip through the array comparing and if necessary, swapping, **adjacent**elements.  Several passes have to be made through the array before it is finally sorted

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Array at beginning:** | 84 | 69 | 76 | 86 | 94 | 91 |
| **After Pass #1:** | 84 | 76 | 86 | 94 | 91 | 69 |
| **After Pass #2:** | 84 | 86 | 94 | 91 | 76 | 69 |
| **After Pass #3:** | 86 | 94 | 91 | 84 | 76 | 69 |
| **After Pass #4:** | 94 | 91 | 86 | 84 | 76 | 69 |
| **After Pass #5 (done):** | 94 | 91 | 86 | 84 | 76 | 69 |

The bubble sort is an **easy algorithm to program**, but it is **slower than many other sorts**.  With a bubble sort, it is always necessary to make one final "pass" through the array to check to see that no swaps are made to ensure that the process is finished.  In actuality, the process is finished before this last pass is made.

**// Bubble Sort Function for Descending Order**  
*#include<stdio.h>*

*main()*

*{*

*int control , control2, marks, temp;*

*int allmarks[5]={9,7,12,5,8};*

*for (control = 0; control < 4; control++) {*

*for (control2 = 0; control2 < 4; control2++) {*

*if (allmarks[control2] > allmarks[control2+1])*

*{*

*temp = allmarks[control2];*

*allmarks[control2]= allmarks[control2+1];*

*allmarks[control2+1] = temp;*

*}*

*}*

*}*

*printf("\nThe sorted list of marks is:\n");*

*for (control=0; control<=4;control++)*

*{*

*printf("%d\n", allmarks[control]);*

*}*

*}*

**Selection Sort**

The **selection sort** is a combination of searching and sorting.  
  
**During each pass, the unsorted element with the smallest (or largest) value is moved to its proper position in the array.**   
The number of times the sort passes through the array is one less than the number of items in the array.  In the selection sort, the inner loop finds the next smallest (or largest) value and the outer loop places that value into its proper location.

Let's look at our same table of elements using a selection sort for descending order.  Remember, a "pass" is defined as one full trip through the array comparing and if necessary, swappingelements.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | |
| **Array at beginning:** | 84 | 69 | 76 | 86 | 94 | 91 |
| **After Pass #1:** | 84 | 91 | 76 | 86 | 94 | 69 |
| **After Pass #2:** | 84 | 91 | 94 | 86 | 76 | 69 |
| **After Pass #3:** | 86 | 91 | 94 | 84 | 76 | 69 |
| **After Pass #4:** | 94 | 91 | 86 | 84 | 76 | 69 |
| **After Pass #5 (done):** | 94 | 91 | 86 | 84 | 76 | 69 |

While being an easy sort to program, the selection sort is **one of the least efficient**.  **The algorithm offers no way to end the sort early, even if it begins with an already sorted list.**

**// Selection Sort Function for Descending Order**  
void main()  
{  
      int i, j, first, temp;  
      int num[5]  
      for (i= 4; i > 0; i--)  
     {  
           first = 0;                **// initialize to subscript of first element**  
           for (j=1; j<=i; j++)   **// locate smallest between positions 1 and i.**  
          {  
                 if (num[j] < num[first])  
                 first = j;  
          }  
         temp = num[first];   **// Swap smallest found with element in position i.**  
         num[first] = num[i];  
         num[i] = temp;  
     }  
     return;  
}

**Shell Sort**

The **shell sort**is named after its inventor D. L. Shell.  Instead of comparing adjacent elements, like the bubble sort, the shell sort repeatedly compares elements that are a certain distance away from each other (*d* represents this distance).  The value of *d* starts out as half the input size and is halved after each pass through the array.  The elements are compared and swapped when needed.  The equation  *d*= (N + 1) / 2  is used.  Notice that only integer values are used for *d* since integer division is occurring.

Let's look at our same list of values for descending order with the shell sort.  Remember, a "pass" is defined as one full trip through the array comparing and if necessary, swappingelements.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Array at beginning:** | | 84 | | 69 | | 76 | | 86 | | 94 | | 91 | *d* | | |
| **After Pass #1:** | | 86 | | 94 | | 91 | | 84 | | 69 | | 76 | 3 | | |
| **After Pass #2:** | | 91 | | 94 | | 86 | | 84 | | 69 | | 76 | 2 | | |
| **After Pass #3:** | | 94 | | 91 | | 86 | | 84 | | 76 | | 69 | 1 | | |
| **After Pass #4 (done):** | | 94 | | 91 | | 86 | | 84 | | 76 | | 69 | | | 1 |

**First Pass:***d* = (6 + 1) / 2 = 3.  Compare 1st and 4th , 2nd and 5th, and 3rd and 6th items since they are  3 positions away from each other))  
**Second Pass:**value for *d* is halved  *d* = (3 + 1) / 2 = 2.  Compare items two places away such as 1st and 3rd ……  
**Third Pass:**value for *d* is halved  *d* = (2 + 1) / 2 = 1.  Compare items one place away such as 1st and 2nd …..  
**Last Pass:**sort continues until *d*= 1 **and** the pass occurs without any swaps.

This sorting process, with its comparison model, **is an efficient sorting algorithm**.

**//Shell Sort Function for Descending Order**void main()  
{  
     int I,d , temp, length[5];  
     while( (d > 1))     **// boolean flag (true when not equal to 0)**  
     {  
          d = (d+1) / 2;  
          for (i = 0; i < (5 - d); i++)  
        {  
               if (num[i + d] > num[i])  
              {  
                      temp = num[i + d];     **// swap positions i+d and i**  
                      num[i + d] = num[i];  
                      num[i] = temp;  
                      flag = 1;                  **// tells swap has occurred**  
              }  
         }  
     }  
     return;  
}

**Quick Sort**

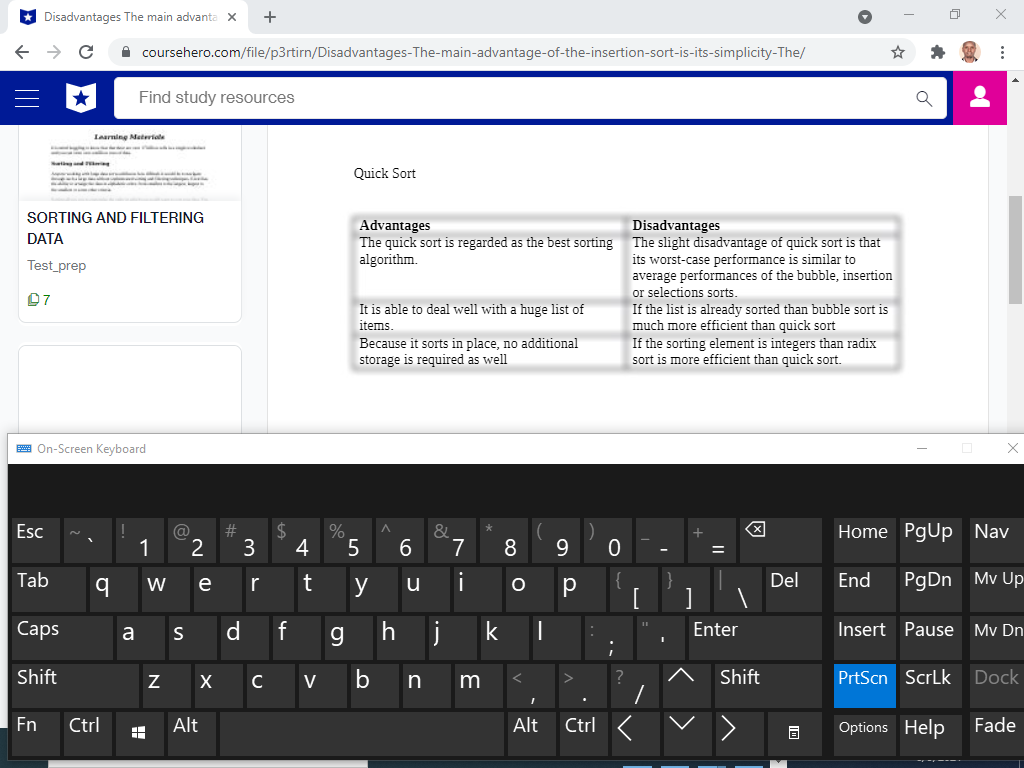
The **quicksort** is considered to be very efficient,with its "divide and conquer" algorithm.  This sort starts by dividing the original array into two sections (partitions) based upon the value of the first element in the array.  Since our example sorts into descending order, the first section will contain all the elements greater than the first element.  The second section will contain elements less than (or equal to) the first element. It is possible for the first element to end up in either section. This sort uses recursion - the process of "calling itself".

Let's examine our same example

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Array at beginning:** | 84 | 69 | 76 | 86 | 94 | 91 |
| |  |  |  | | --- | --- | --- | | |  | | --- | |  | | **= 1st partition** | | 86 | 94 | 91 | 84 | 69 | 76 |
| |  |  |  | | --- | --- | --- | | |  | | --- | |  | | **= 2nd partition** | | 94 | 91 | 86 | 84 | 69 | 76 |
|  | 94 | 91 | 86 | 84 | 69 | 76 |
|  | 94 | 91 | 86 | 84 | 69 | 76 |
| **Done:** | 94 | 91 | 86 | 84 | 76 | 69 |

**//Quick Sort Functions for Descending Order  
// (2 Functions)**  
void main()  
{  
      **// top = subscript of beginning of array  
      // bottom = subscript of end of array**  
     int middle;  
     if (top < bottom)  
    {  
          middle = partition(num, top, bottom);  
          quicksort(num, top, middle);  **// sort first section**  
          quicksort(num, middle+1, bottom);   **// sort second section**  
     }  
     return;  
}

**//Function to determine the partitions  
// partitions the array and returns the middle subscript**  
int main()  
{  
     int x = array[top];  
     int i = top - 1;  
     int j = bottom + 1;  
     int temp;  
     do  
     {  
           do        
           {  
                  j - -;  
           }while (x >array[j]);  
  
          do    
         {  
                 i++;  
          } while (x <array[i]);  
  
          if (i < j)  
         {   
                 temp = array[i];     
                 array[i] = array[j];  
                 array[j] = temp;  
         }  
     }while (i < j);       
     return j;           **// returns middle subscript**}



**Merge Sort**

The **merge sort** combines two **sorted** arrays into one larger sorted array.     
As the diagram below shows, Array A and Array B merge to form Array C.

Arrays to be merged **MUST be SORTED FIRST!!**

Be sure to declare Array C  in main( ) and establish its size.

**Example: Ascending Order**  
Array A: {7. 12}  
Array B: {5,  7, 8}  
Array C: {5, 7, 7, 8, 12} after merge

*Here is how it works:*The first element of array A is compared with the first element of array B.  If the first element of array A is smaller than the first element of array B, the element from array A is moved to the new array C.  The subscript of array A is now increased since the first element is now set and we move on.

If the element from array B should be smaller, it is moved to the new array C.  The subscript of array B is increased.  This process of comparing the elements in the two arrays continues until either array A or array B is empty.  When one array is empty, any elements remaining in the other (non-empty) array are "pushed" into the end of array C and the merge is complete.

**//Function to merge two pre-sorted arrays**  
void main()  
{  
     int indexA = 0;     **// initialize variables for the subscripts**  
     int indexB = 0;  
     int indexC = 0;

Int arrayC[5];  
  
     while((indexA < 5) && (indexB < 5)  
     {

          if (arrayA[indexA] < arrayB[indexB])  
          {  
                 arrayC[indexC] = arrayA[indexA];  
                 indexA++;    **//increase the subscript**  
          }  
         else  
         {  
                 arrayC[indexC] = arrayB[indexB];  
                 indexB++;      **//increase the subscript**  
         }  
        indexC++;      **//move to the next position in the new array**  
     }  
     **// Move remaining elements to end of new array when one merging array is empty**  
     while (indexA < 5)  
     {  
           arrayC[indexC] = arrayA[indexA];  
           indexA++;  
           indexC++;  
     }  
     while (indexB < 5)  
     {  
           arrayC[indexC] = arrayB[indexB];  
           indexB++;  
           indexC++;  
     }  
     return;  
}

### SEARCHING ARRAYS

When working with arrays, it is often necessary to perform a search or "lookup" to determine whether an array contains a value that matches a certain key value The process of locating a particular element value in an array is called searching. There are two types of search mechanisms: **serial**/**linear search** and **binary search**

**a) Serial Search**

The technique used here is called a **serial search**, because the integer elements of the array are compared one by one to the user input being looked for (userValue) until either a match is found or all elements of the array are examined without finding a match.

In the code below, if a match is found, the text “There is a match” is printed on the form and the execution of the procedure is terminated (Exit Sub). If no match is found, the program exits the loop and prints the text “No match found”.

#include <stdio.h>

int main()

{

int array[5]={10,7,8,2,5}, searchvalue, c;

printf("\tEnter the number to search: ");

scanf("%d", &searchvalue);

for (c = 0; c < 5; c++)

{

if (array[c] == searchvalue) // if required element found

{

printf("\n%d is present at location %d.\n", searchvalue, c++);

break;

}

}

if (c == 5) // if looped more than 5 times ie 6 times

printf("\n%d is not present in the array.\n", searchvalue);

return 0;

}

**Binary Search**Binary search uses the concept of splitting your searchable array in two, discarding the half that does not have the element for which you are looking.

You place your items in an array and sort them. Then you simply get the middle element and test if it is <, >, or = to the element for which you are searching. If it is less than, you discard the greater half, get the middle index of the remaining elements and do it again. Binary search divides your problem in half every time you execute your loop.

#include <stdio.h>

int main()

{

int c, first, last, middle, n, search, array[10];

printf("Enter number of elements**\n**");

scanf("%d",&n);

printf("Enter %d integers**\n**", n);

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

scanf("%d",&array[c]);

printf("Enter value to find**\n**");

scanf("%d",&search);

first = 0;

last = n - 1;

middle = (first+last)/2;

while( first <= last )

{

if ( array[middle] < search )

first = middle + 1;

else if ( array[middle] == search )

{

printf("%d found at location %d.**\n**", search, middle+1);

**break**;

}

else

last = middle - 1;

middle = (first + last)/2;

}

if ( first > last )

printf("Not found! %d is not present in the list.**\n**", search);

return 0;

}

## POINTERS

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is:

type \*var-name;

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk \* you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

int \*ip; /\* pointer to an integer \*/

double \*dp; /\* pointer to a double \*/

float \*fp; /\* pointer to a float \*/

char \*ch /\* pointer to a character \*/

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

**Dereferencing**

**HOW TO USE POINTERS?**

Pointers are used for:

* Strings
* Dynamic memory allocation
* Sending function arguments by reference
* Pointing to functions
* Building special data strcutures such as Trees

There are few important operations, which we will do with the help of pointers very frequently. **(a)** we define a pointer variable **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator **\*** that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

#include <stdio.h>

int main ()

{

int var = 20; /\* actual variable declaration \*/

int \*ip; /\* pointer variable declaration \*/

ip = &var; /\* store address of var in pointer variable\*/

printf("Address of var variable: %x\n", &var );

/\* address stored in pointer variable \*/

printf("Address stored in ip variable: %x\n", ip );

/\* access the value using the pointer \*/

printf("Value of \*ip variable: %d\n", \*ip );

return 0;

}

When the above code is compiled and executed, it produces result something as follows:

Address of var variable: bffd8b3c

Address stored in ip variable: bffd8b3c

Value of \*ip variable: 20

**NULL Pointers in C**

It is always a good practice to assign a NULL value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

A NULL pointer is usually used to mark the end of a linked list. i.e. Since pointers in a list points to the next member of the list, if the pointer is NULL, then it is the last node in the list.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

*#include <stdio.h>*

*int main ()*

*{*

*int \*ptr = NULL;*

*printf("The value of ptr is : %x\n", ptr );*

*return 0;*

*}*

When the above code is compiled and executed, it produces the following result:

*The value of ptr is 0*

On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer you can use an if statement as follows:

if(ptr) /\* succeeds if p is not null \*/

if(!ptr) /\* succeeds if p is null \*/

## C STRINGS

In C, one or more characters enclosed between double quotes is called a string. C does not have built-in string data type. Instead, C supports strings using one-dimensional arrays. A string is defined as a *null terminated* array i.e. \0. This means that you must define the array that is going to hold a string to be one byte larger than the largest string it is going to hold, in order 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 **string.h**  header file. 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. E.g.

#include<stdio.h>

Main ( )

{

Char name [80];

Printf (” Enter a string: \n”);

gets(name);

//Output the name

Printf(name);

}

The following declaration and initialization create a string consisting of the word **"Hello"**. To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word **"Hello"**.

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

**Initialization of strings**

In C, string can be initialized in a different number of ways.

char c[]="abcd";

OR,

char c[5]="abcd";

OR,

char c[]={'a','b','c','d','\0'};

OR;

char c[5]={'a','b','c','d','\0'};

Initialization of strings in C programming

Strings can also be initialized using pointers

char \*c="abcd";

char c[20];

scanf("%s",c);

String variable c can only take a word. It is beacause when white space is encountered, the scanf() function terminates.

**Write a C program to illustrate how to read string from terminal.**

#include <stdio.h>

int main(){

char name[20];

printf("Enter name: ");

scanf("%s",name);

printf("Your name is %s.",name);

return 0;

}

**Output**

Enter name: Dennis Ritchie

Your name is Dennis.

Here, program will ignore Ritchie because, scanf() function takes only string before the white space.

C supports a wide range of functions that manipulate null-terminated strings:

|  |  |
| --- | --- |
| **S.N.** | **Function & Purpose** |
| 1 | **strcpy(s1, s2);**  Copies string s2 into string s1. |
| 2 | **strcat(s1, s2);**  Concatenates string s2 onto the end of string s1. |
| 3 | **strlen(s1);**  Returns the length of string s1. |
| 4 | **strcmp(s1, s2);**  Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2. |
| 5 | **strchr(s1, ch);**  Returns a pointer to the first occurrence of character ch in string s1. |
| 6 | **strstr(s1, s2);**  Returns a pointer to the first occurrence of string s2 in string s1. |

The C library function **int strcmp(const char \*str1, const char \*str2)** compares the string pointed to by **str1** to the string pointed to by **str2**.

Following is the declaration for strcmp() function.

strcmp(str1, str2)

**PARAMETERS**

* **str1** -- This is the first string to be compared.
* **str2** -- This is the second string to be compared.

**RETURN VALUE**

This function returned values are as follows:

* if Return value < 0 then it indicates str1 is less than str2
* if Return value > 0 then it indicates str2 is less than str1
* if Return value = 0 then it indicates str1 is equal to str2

**Example**

The following example shows the usage of strncmp() function.

#include <stdio.h>

#include <string.h>

int main ()

{

char str1[15];

char str2[15];

int ret;

strcpy(str1, "abcdef");

strcpy(str2, "ABCDEF");

ret = strcmp(str1, str2);

if(ret > 0)

{

printf("str1 is less than str2");

}

else if(ret < 0)

{

printf("str2 is less than str1");

}

else

{

printf("str1 is equal to str2");

}

return(0);

}

## Dynamic allocation

Dynamic allocation of memory is a very important subject in C - it allows building complex data structures such as linked lists. Allocating memory dynamically helps us to store data without initially knowing the size of the data in the time we wrote the program.

To allocate a chunk of memory dynamically, we have to have a pointer ready - which will store the location of the newly allocated memory. We can access memory that was allocated to us using that same pointer, and we can use that pointer to free the memory we got, once we finish using it.

To dynamically allocate a person structure, the person is defined like this:

typedef struct {

char \* name;

char age;

} person;

Execute Code

To allocate a new person in the *myperson* argument, we use the following syntax:

person \* myperson = malloc(sizeof(person));

Execute Code

This tells the compiler that we want to dynamically allocate just enough to hold a person struct in memory, and then return a pointer to the newly allocated data.

Note that sizeof is not an actual function, because the compiler interprets it and translates it to the actual memory size of the pointer struct.

To access the person's members, we can use the -> notation:

myperson->name = "John";

myperson->age = 27;

Execute Code

After we are done using the dynamically allocated struct, we can release it using free:

free(myperson);

Execute Code

Note that the free does not delete the myperson variable itself, it simply releases the data that it points to. The myperson variable will still point to somewhere in the memory - but after calling myperson we are not allowed to access that area anymore. We must not use that pointer again until we allocate new data using it.

## Linked lists

Linked lists are the best and simplest example of a dynamic data structure that uses pointers for its implementation. However, understanding pointers is crucial to understanding how linked lists work, so if you've skipped the pointers tutorial, you should go back and redo it. You must also be familiar with dynamic memory allocation and structures.

Essentially, linked lists function as an array that can grow and shrink as needed, from any point in the array.

Linked lists have a few advantages over arrays:

1. Items can be added or removed from the middle of the list
2. There is no need to define an initial size

However, linked lists also have a few disadvantages:

1. There is no "random" access - it is impossible to reach the nth item in the array without first iterating over all items up until that item. This means we have to start from the beginning of the list and count how many times we advance in the list until we get to the desired item.
2. Dynamic memory allocation and pointers are required, which complicates the code and increases the risk of memory leaks and segment faults.
3. Linked lists have a much larger overhead over arrays, since linked list items are dynamically allocated (which is less efficient in memory usage) and each item in the list also must store an additional pointer.

**A linked list, therefore, is a set of dynamically allocated nodes, arranged in such a way that each node contains one value and one pointer**. The pointer always points to the next member of the list. If the pointer is NULL, then it is the last node in the list.

A linked list is held using a local pointer variable which points to the first item of the list. If that pointer is also NULL, then the list is considered to be empty.

Defining a linked list node:

typedef struct node {

int val;

struct node \* next;

} node\_t;

Notice that we are defining the struct in a recursive manner, which is possible in C. Let's name our node type node\_t.

Now we can use the nodes. Let's create a local variable which points to the first item of the list (called head).

node\_t \* head = NULL;

head = malloc(sizeof(node\_t));

if (head == NULL) {

return 1;

}

head->val = 1;

head->next = NULL;

We've just created the first variable in the list. We must set the value, and the next item to be empty, if we want to finish populating the list. Notice that we should always check if malloc returned a NULL value or not.

To add a variable to the end of the list, we can just continue advancing to the next pointer:

node\_t \* head = NULL;

head = malloc(sizeof(node\_t));

head->val = 1;

head->next = malloc(sizeof(node\_t));

head->next->val = 2;

head->next->next = NULL;

This can go on and on, but what we should actually do is advance to the last item of the list, until the next variable will be NULL.

**Iterating over a list**

To print out all the items of a list. Use a current pointer that will keep track of the node we are currently printing. After printing the value of the node, we set the current pointer to the next node, and print again, until we've reached the end of the list (the next node is NULL).

while (current != NULL) {

printf("%d\n", current->val);

current = current->next;

}

}

**More Examples**

1. [C Program to Find the Length of a String](http://www.programiz.com/c-programming/examples/string-length)

#include <stdio.h>

#include <string.h>

int main()

{

char s[1000],i;

printf("Enter a string: ");

gets(s);

i = strlen(s);

printf("Length of the string is: %d\n",i);

return 0;

}**Output**

Enter a string: Programiz

Length of string: 9

1. Code to Concatenate Two Strings Manually

*#include<stdio.h>*

*#include<string.h>*

*main()*

*{*

*char fname[10],mname[10],newone[20];*

*int len,len2; char \*mypt;char \*secstr;*

*printf("Key in your first name: ");*

*scanf("%s",fname);*

*printf("Key in your middle name: ");*

*scanf("%s",mname);*

*len=strlen(fname);*

*strcat(fname,mname);*

*len2=strlen(fname);*

*strcpy(newone,fname);*

*mypt = strchr(fname,'N');*

*secstr= strstr(newone,mname);*

*printf("Your full name is %s\n",fname);*

*puts(fname);*

*printf("The new array is %s\n",newone);*

*printf("The length of the first name before concatenation is %d\n",len);*

*printf("The length of the first name after concatenation is %d\n",len2);*

*printf("The first instance of fname is %x\n",mypt);*

*printf("The first instance of mname in newone is %x\n",secstr);*

## Structures

C **arrays** allow you to define types of variables that can hold several data items of the same kind. On the other hand, a **structure** is a **User Defined Data Type (UDT)** available in C programming, which allows you to combine data items of different kinds.

**Structures are used to group together different types of variables under the same name**. For example you could create a structure “Books”: which is made up of three strings (that is used to hold the author’s name, Subject and title) and an integer (that is used to hold the book’s ID).

**Structures are used to represent records of data**. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

* Title
* Author
* Subject
* Book ISBN No

## Defining a Structure

To define a structure, you must use the **struct** statement. **The struct statement defines a new data type,** with more than one member for your program.

With the declaration of the structure you have created a new type, called Books. Before you can use the type telephone you have to create a variable of the type BOOKS. The format of the struct statement is this:

struct structure\_name

{

member definition;

member definition;

...

member definition;

};

## Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define **variables of structure type**. Following is the example to explain usage of structure:

#include <stdio.h>

#include <string.h>

struct Books

{

char title[50];

char author[50];

char subject[100];

int book\_id;

};

int main( )

{

struct Books Book1; /\* Declare Book1 of type Book \*/

/\* book 1 specification \*/

strcpy( Book1.title, "C Programming");

strcpy( Book1.author, "Nuha Ali");

strcpy( Book1.subject, "C Programming Tutorial");

Book1.book\_id = 6495407;

/\* print Book1 info \*/

printf( "Book 1 title : %s\n", Book1.title);

printf( "Book 1 author : %s\n", Book1.author);

printf( "Book 1 subject : %s\n", Book1.subject);

printf( "Book 1 book\_id : %d\n", Book1.book\_id);

return 0;

}

When the above code is compiled and executed, it produces the following result:

Book 1 title : C Programming

Book 1 author : Nuha Ali

Book 1 subject : C Programming Tutorial

Book 1 book\_id : 6495407

## typedef

The C programming language provides a keyword called typedef, which you can **use to give a user defined type (UDT) a new name**.

For example you can use typedef with structure to **define a new data type** and then **use that data type to define structure variables directly** as follows:

#include<stdio.h>

#include<string.h>

typedef struct Books

{

int isbnNo;

char \*title;

char author[50];

}BOOK;

main()

{

BOOK mybook;

mybook.isbnNo=22001;

mybook.title="Wizard of the Crow";

strcpy(mybook.author,"Ngugi wa Thiong'o");

printf("The book's title is %s: \n",mybook.title);

printf("The book's Author is %s: \n",mybook.author);

printf("The book's ISBN No is %d: \n",mybook.isbnNo);

}

typedef vs #define

The #define is a C-directive which is also used to define the aliases for various data types similar to typedef but with a different:

The typedef is limited to giving symbolic names to types only where as #define can be used to define an alias for values as well, like you can define 1 as ONE etc.

The typedef interpretation is performed by the compiler where as #define statements are processed by the pre-processor.

Following is a simplest usage of #define:

#include <stdio.h>

#define TRUE 1

#define FALSE 0

int main( )

{

printf( "Value of TRUE : %d\n", TRUE);

printf( "Value of FALSE : %d\n", FALSE);

return 0;

}

When the above code is compiled and executed, it produces the following result:

Value of TRUE : 1

Value of FALSE : 0

## QUEUES

Queue is a specialized data storage structure (Abstract data type). Unlike arrays, access of elements in a Queue is restricted. It has two main operations enqueue and dequeue. Insertion in a queue is done using enqueue function and removal from a queue is done using dequeue function. An item can be inserted at the end (‘rear’) of the queue and removed from the front (‘front’) of the queue. It is therefore, also called First-In-First-Out (FIFO) list. Queue has five properties - capacity stands for the maximum number of elements Queue can hold, size stands for the current size of the Queue, elements is the array of elements, front is the index of first element (the index at which we remove the element) and rear is the index of last element (the index at which we insert the element).

**Primitive operations**

1. enqueue (q, x): inserts item **x** at the rear of the queue **q**
2. x = dequeue (q): removes the front element from **q** and returns its value.
3. isEmpty(q) : true if the queue is empty, otherwise false.

**Example**

enqueue(q, ‘A’);

enqueue(q, ‘B’);

enqueue(q, ‘C’);

x = dequeue(q);

enqueue(q, ‘D’);

enqueue(q, ‘E’);



x= dequeue (q) -> x= ‘A’



## STACKS

A stack is a data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack; the only element that can be removed is the element that was at the top of the stack. Consequently, a stack is said to have "first in last out" behavior (or "last in, first out"). The first item added to a stack will be the last item removed from a stack.

***CHAPTER 7***

# FILE HANDLING

This chapter explains how C programmers can create, open and close text or binary files for their data storage. A file represents a sequence of bytes. It does not matter if it is a text file or binary file.

## OPENING FILES

You can use the **fopen( )** function to create a new file or to open an existing file, this call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream. Following is the prototype of this function call:

FILE \*fopen( filename, mode );

Here, **filename** is a string literal, which you will use to name your file and access **mode** can have one of the following values:

|  |  |
| --- | --- |
| **Mode** | **Description** |
| r | Opens an existing text file for reading purpose. |
| w | Opens a text file for writing, if it does not exist then a new file is created. Here your program will start writing content from the beginning of the file. |
| a | Opens a text file for writing in appending mode, if it does not exist then a new file is created. Here your program will start appending content in the existing file content. |
| r+ | Opens a text file for both reading and writing. |
| w+ | Opens a text file for reading and writing both. It first truncate the file to zero length if it exists otherwise create the file if it does not exist. |
| a+ | Opens a text file for reading and writing both. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended. |

**Modes of Binary Files**

If you are going to handle binary files then you will use below mentioned access modes instead of the above mentioned:

"rb", "wb", "ab", "ab+", "a+b", "wb+", "w+b"

**Example**

#include<stdio.h>

Int main ()

{

char name[20];

FILE \*myfile;

printf("Key in your name: ");

gets(name);

myfile = fopen("C:/tmp/test.txt","w+");

fprintf(myfile,"Hello ");

fputs(name,myfile);

fputs(" and welcome\n",myfile);

fclose(myfile);

return 0;

}

## CLOSING A FILE

To close a file, use the **fclose( )** function. The prototype of this function is:

int fclose( FILE \*fp );

The **fclose( )** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually, **flushes any data still pending in the buffer to the file**, **closes the file**, and **releases any memory used for the file**. The EOF is a constant defined in the header file **stdio.h**.

There are various functions provided by C standard library to read and write a file character by character or in the form of a fixed length string.

## WRITING A FILE

Following is the simplest function to write individual characters to a stream:

int fputc( int c, FILE \*fp );

where,

**fputc()**function- writes the character value of the argument c to the output stream referenced by fp.

It returns the written character on success otherwise **EOF** if there is an error.

You can use the following functions to write a null-terminated string to a stream:

int fputs( const char \*s, FILE \*fp );

The function **fputs()** writes the string **s** to the output stream referenced by fp.

It returns a non-negative value on success, otherwise **EOF** is returned in case of any error. You can use **int fprintf(FILE \*fp,const char \*format, ...)** function as well to write a string into a file. Try the following example:

#include <stdio.h>

main()

{

FILE \*fp;

fp = fopen("/tmp/test.txt", "w+");

fprintf(fp, "This is testing for fprintf...\n");

fputs("This is testing for fputs...\n", fp);

fclose(fp);

}

When the above code is compiled and executed, it creates a new file **test.txt** in /tmp directory and writes two lines using two different functions. Let us read this file in next section.

## READING A FILE

Following is the simplest function to read a single character from a file:

int fgetc( FILE \*fp );

where,

The **fgetc()** function reads a character from the input file referenced by fp.

The return value is the character read, or in case of any error it returns **EOF**. The following functions allow you to read a string from a stream:

The functions **fgets()** reads up to n - 1 characters from the input stream referenced by fp. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.

char \*fgets( char \*str, int num, FILE \* stream );

**PARAMETERS OF THE fgets() FUNCTION**

**Str –** This is the pointer to an array of chars where the string is to be stored

**num –** The maximum number of characters to be read (including the final null-character). Usually, the length of the array to be used is given here.

**Stream –** This is the pointer to a file object where the characters are to be read from.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including new line character. You can also use **int fscanf(FILE \*fp, const char \*format, ...)** function to read strings from a file but it stops reading after the first space character it encounters.

#include <stdio.h>

main()

{

FILE \*myfile;

char buffer[50];

myfile=fopen("C:/tmp/cfiles.txt","r");

fgets(buffer,50,myfile);

printf("%s\n",buffer);

fclose(myfile);

}

When the above code is compiled and executed, it reads the file created in previous section and produces the following result:

1 : This

2: is testing for fprintf...

3: This is testing for fputs...

Let's see a little more detail about what happened here. First **fscanf()** method read just **This** because after that it encountered a space, second call is for **fgets()** which read the remaining line till it encountered end of line. Finally last call **fgets()** read second line completely.

## BINARY I/O FUNCTIONS

There are following two functions, which can be used for binary input and output:

size\_t fread(void \*ptr, size\_t size\_of\_elements,

size\_t number\_of\_elements, FILE \*a\_file);

size\_t fwrite(const void \*ptr, size\_t size\_of\_elements,

size\_t number\_of\_elements, FILE \*a\_file);

Both of these functions should be used to read or write blocks of memories - usually arrays or structures.

**MORE EXAMPLES**

1. **Writing array elements (from user) into a text file.**

#include<stdio.h>

main()

{

int i, marks[5];

FILE \*myfile;

printf("Key in five integers:\n");

for(i=1; i<=5;i++)

{

scanf("%d",&marks[i]);

}

myfile = fopen("C:/tmp/array.txt","w+");

fputs("The following are the array elements",myfile);

for(i=1;i<=5;i++)

{

fprintf(myfile,"%d,",marks[i]);

}

fclose(myfile);

}

1. **Writing a pattern of values into a text file.**

#include<stdio.h>

main()

{

int k,l;

FILE \*myfile;

myfile=fopen("C:/tmp/test2.txt", "w");

for(k=1;k<=5;k++)

{

for(l=5;l>=k;l--)

{

fprintf(myfile,"%d ",l);

}

fprintf(myfile,"\n");

}

fclose(myfile);

}

1. **Writing a customized greeting into a text file**

#include<stdio.h>

main()

{

char name[20];

FILE \*myfile;

printf("Key in your name: ");

gets(name);

myfile=fopen("C:/tmp/test.txt","w+");

fprintf(myfile,"Hello ");

fputs(name,myfile);

fputs(" and welcome\n",myfile);

fclose(myfile);

}

***Chapter 7***

# ERROR HANDLING AND DEBUGGING

Programming errors often remain undetected until an attempt is made to compile or execute the program.

## Types of Errors

1. Syntax/ compiler Errors
2. Execution/ run time Errors
3. Logical Errors.

### Syntax Errors

-errors resulting from violation of grammatical rules.

Examples

1. Omission of semi colon the end of program statement
2. Naming a vartiable using keyword

Syntax errors are **generated when the programmer from the strict rules of the programming language**. These errors will **prevent the program from being compiled or executed successfully.**

Some particularly common errors of this type are:

1. Improperly declared variables
2. A reference to an undeclared variable
3. Incorrect punctuation, etc.

Most C compilers will **generate diagnostic messages** when syntactic errors have been detected during the compilation process. These diagnostic messages are not always straightforward in their meaning and they may not correctly identify where the error occurred (though they may attempt to do so). Nevertheless, they are helpful in identifying the nature and the approximate location of the errors.

If a program includes several different syntactic errors, they may not all be detected on the first pass

through the compiler. Thus, it may be necessary to correct some syntactic errors before others can be found.

This process could repeat itself through several cycles before all of the syntactic errors have been identified and corrected.

### Run time/ Execution Errors

**These errors occur if code attempts to execute an instruction that is impossible to carry out**. Execution errors **occur during program execution**, **after a successful compilation**.

Example of some common execution errors are:

1. A numerical overflow of underflow (exceeding the largest or smallest permissible number that can be stored in a variable)
2. Division by zero
3. Attempting to compute the logarithm or the square root of a negative number, etc.

Diagnostic messages will often be generated in situations of this type, making it easy to identify and correct the errors. These diagnostics are sometimes called execution messages or run-time messages, to distinguish them from the compilation messages described earlier.

### Logical Errors

Here the program executes correctly, carrying out the programmer’s wishes, but the programmer has supplied the computer with instructions that are logically incorrect. Logical errors can be very difficult to detect, since the output resulting from a logically incorrect program may appear to be error-free. Moreover, logical errors are often hard to locate even when they are known to exist (as, for example, when the computed results are obviously incorrect).

Fortunately, methods are available for finding the location of execution errors and logical errors within a program. Such methods are generally referred to as debugging techniques. Some of the more commonly used **debugging techniques** are described below.

## DEBUGGING TECHNIQUES

Syntactic errors are relatively easy to find and correct, even if the resulting error messages are unclear. Execution errors, on the other hand, can be much more troublesome. When an execution error occurs, we must first determine its location (where it occurs) within the program. Once the location of the execution error has been identified, the source of the error (why it occurs) must be determined.

### Error Isolation

Error isolation is useful for locating an error resulting in a diagnostic message. If the general location of the error is not known, it can frequently be found by temporarily deleting a portion of the program and then rerunning the program to see if the error disappears. The temporary deletion is accomplished by surrounding the instructions with comment markers (/ \* and \* /), causing the enclosed instructions to become comments.

If the error message then disappears, the deleted portion of the program contains the source of the error.

A closely related technique is that of inserting several unique **printf** statements, such as

printf ("Debugging - l i n e 1\ n " ) ;

printf ("Debugging - l i n e 2 \ n " ) ;

etc.at various places within the program. When the program is executed, the debug messages will indicate the approximate location of the error. Thus, the source of the error will lie somewhere between the last printf statement whose message did appear, and the first printf statement whose message did not appear.

### Tracing

Tracing involves the use of printf statements to display the values assigned to certain key variables, or to display the values that are calculated internally at various locations within the program. This information serves several purposes. For example, it verifies that the values actually assigned to certain variables really are (or are not) the values that should be assigned to those values. It is not uncommon to find that the actual assigned values are different than those expected. In addition, this information allows you to monitor the progress of the computation as the program executes. In many situations, you will be able to identify a particular place where things begin to go wrong because the values generated will be obviously incorrect.

### Watch Values

A watch value is the value of a variable or an expression which is displayed continuously as the program executes. Thus, you can see the changes in a watch value as they occur, in response to the program logic. By monitoring a few carefully selected watch values, you can often determine where the program begins to generate incorrect or unexpected values.

In Turbo C++, watch values can be defined by selecting Add Watch from the Debug menu (see Fig. 5.4

earlier in this chapter), and then specifying one or more variables or expressions in the resulting dialog box.

The watch values will then be displayed within a separate window as the program executes.

### Breakpoints

A breakpoint is a temporary stopping point within a program. Each breakpoint is associated with a particular instruction within the program. When the program is executed, the program execution will temporarily stop at the breakpoint, before the instruction is executed. The execution may then be resumed, until the next breakpoint is encountered. Breakpoints are often used in conjunction with watch values, by observing the current watch value at each breakpoint as the program executes.

To set a breakpoint in Turbo C++,select Add Breakpoint from the Debug menu (see Fig. 5.4), and then

provide the requested information in the resulting dialog box. Or, select a particular line within the program and designate it a breakpoint by pressing function key F5. The breakpoint may later be disabled by again pressing F5. (Function key F5 is called a "toggle" in this context, since it turns the breakpoint on or off by successively pressing the key.)

### Stepping

Stepping refers to the execution of one instruction at a time, typically by pressing a fbnction key to execute each instruction. In Turbo C++, for example, stepping can be carried out by pressing either function key F7 or F8. (F8 steps over subordinate functions, whereas F7 steps through the functions.) By stepping through an entire program, you can determine which instructions produce erroneous results or generate error messages.

Stepping is often used with watch values, allowing you to trace the entire history of a program as it

executes. Thus, you can observe changes to watch values **as** they happen. This allows you to determine

which instructions generate erroneous results.

***Chapter 8***

# SOFTWARE DOCUMENTATION

**Software documentation** is written text that accompanies [computer software](http://en.wikipedia.org/wiki/Computer_software). It both explains how the software operates or how to use it and may mean different things to people in different roles.

**Importance/Needs of software documentation**

1. Provide for communication among team members i.e. programmers/ developers, users, management
2. They should provide information for management to help them plan, budget and schedule the software development process.
3. It acts as an information repository to be used by maintenance engineers
4. Describe to users how to operate and administer the system
5. In all software projects some amount of documentation should be created prior to any code being written for example Design docs, etc.
6. Documentation should continue after the code has been completed for example User’s manuals, etc.
7. It describe istallation process

The two main types of documentation created are Internal and External.

## INTERNAL DOCUMENTATION

It is also known as In-Line Program Documentation and refers to the notes on how and why various parts of code operate that are included within the source code as comments. It is often combined with meaningful variable names with the intention of providing potential future programmers a means of understanding the workings of the code.

The internal documentation should include the following:

1. ‘block comment’ which should be placed at the head of every method (also known as the function or subprogram). This will include the method name; the purpose of the method; the method’s pre– and post–conditions; the method’s return value (if any); and a list of all parameters, including direction of information transfer (into this method, out from the method back to the calling method, or both), and their purposes.
2. Meaningful identifier names. Traditionally, simple loop variables may have single letter variable names, but all others should be meaningful. Never use nonstandard abbreviations. If the programming language has a naming convention for variables, methods, classes, etc., then those conventions should be used.
3. Each variable and constant must have a brief comment immediately after its declaration that explains its purpose. This applies to all variables, as well as to fields of structure declarations.
4. Complex sections of the program that need some more explanations should have comments just before or embedded in those program sections.

## EXTERNAL DOCUMENTATION

This is documentation that is kept external to the program and provides a comprehensive description of the design, development, and structure of a program. There are two types of external documentation:

***Process*** and ***Product*** documentation.

### PROCESS DOCUMENTATION

1. Used to record and track the development process
   * + Planning documentation
     + Cost, Schedule, Funding tracking
     + Schedules
     + Standards e.t.c.
2. This documentation is created to allow for successful management of a software product
3. Has a relatively short lifespan
4. Only important to internal development process
5. Except in cases where the customer requires a view into this data
6. Some items, such as papers that describe design decisions should be extracted and moved into the product documentation category when they become implemented

### PRODUCT DOCUMENTATION

Describes the delivered product

Must evolve with the development of the software product

There are two main categories of process documentation:

1. **Technical/System Documentation**

This describes how the system works, but not how to operate it

Examples:

* + Requirements Spec
  + Architectural Design
  + Detailed Design
  + Commented Source Code
  + Including output such as JavaDoc
  + Test Plans
  + Including test cases
  + V&V plan and results
  + List of Known Bugs

1. **User Documentation**

The user documentation describes each feature of the program, and assists the user in realizing these features. A good user document can also go so far as to provide thorough troubleshooting assistance.

User Documentation has two main types

* + End User
  + System Administrator

In some cases these are the same people. The target audience must be well understood.

There are five important areas that should be documented for a formal release of a software application. These do not necessarily each have to have their own document, but the topics should be covered thoroughly. These include:

* + - Functional Description of the Software
    - Installation Instructions
    - Introductory Manual
    - Reference Manual
    - System Administrator’s Guide

**Document Quality**

Providing thorough and professional documentation is important for any size product development team

**Document Structure**

All documents for a given product should have a similar structure

The authors “best practices” are:

* + Put a cover page on all documents
  + Divide documents into chapters with sections and subsections
  + Add an index if there is lots of reference information
  + Add a glossary to define ambiguous terms