

Table of Contents

PART ONE . FUNDAMENTAL OF INFORMATION TECHNOLOGY

UNIT 1 BASIC CONCEPTS : INFORMATION, DATA, INFORMATION TECHNOLOGY

- 1.1. Information and Information Processing**Error! Bookmark not defined.**
 - 1.1.1. Data – Information – Knowledge **Error! Bookmark not defined.**
 - Information **Error! Bookmark not defined.**
 - Data **Error! Bookmark not defined.**
 - Knowledge **Error! Bookmark not defined.**
 - 1.1.2. Information Processing **Error! Bookmark not defined.**
- 1.2. Computers and Classification of Computers**Error! Bookmark not defined.**
 - 1.2.1. History of Computers **Error! Bookmark not defined.**
 - 1.2.2. Classification of Computers **Error! Bookmark not defined.**
- 1.3. What is IT? ICT?**Error! Bookmark not defined.**
 - 1.3.1. Computer Science **Error! Bookmark not defined.**
 - 1.3.2. Information Technology **Error! Bookmark not defined.**
 - 1.3.3. Information Technology and Communication **Error! Bookmark not defined.**

UNIT 2. INFORMATION REPRESENTATION IN COMPUTERS

- 2.1. Information Representation in Various Numeral Systems**Error! Bookmark not defined.**
 - 2.1.1. Base-b System **Error! Bookmark not defined.**
 - 2.1.2. Decimal System **Error! Bookmark not defined.**
 - 2.1.3. Binary System **Error! Bookmark not defined.**
 - 2.1.4. Octal System **Error! Bookmark not defined.**
 - 2.1.5. Hexa-decimal System **Error! Bookmark not defined.**
 - 2.1.6. Converting Between Base-b and Decimal Systems **Error! Bookmark not defined.**
 - 2.1.7. Logic Propositions **Error! Bookmark not defined.**
- 2.2. Information Coding in Computers - Units of Information**Error! Bookmark not defined.**
 - 2.2.1. Basic Principles **Error! Bookmark not defined.**
 - 2.2.2. Units of Information **Error! Bookmark not defined.**
- 2.3. Representation of Integers**Error! Bookmark not defined.**
 - 2.3.1. Unsigned Integers **Error! Bookmark not defined.**
 - 2.3.2. Signed Integers **Error! Bookmark not defined.**
- 2.4. Operations on Integers.....**Error! Bookmark not defined.**
 - 2.4.1. Addition and Subtraction **Error! Bookmark not defined.**
 - 2.4.2. Multiplication and Division **Error! Bookmark not defined.**
- 2.5. Logical operations on Binary Numbers.....**Error! Bookmark not defined.**

2.6. Symbol Representation.....	Error! Bookmark not defined.
2.6.1. Basic Principles	Error! Bookmark not defined.
2.6.2. ASCII Code Table	Error! Bookmark not defined.
2.6.3. Unicode Code Table	Error! Bookmark not defined.
2.7. Representation of Real Numbers	Error! Bookmark not defined.
2.7.1. Basic Principles	Error! Bookmark not defined.
2.7.2. IEEE 754/85 Standard	Error! Bookmark not defined.
UNIT 3. COMPUTER SYSTEMS	Error! Bookmark not defined.
3.1. Computer Architecture	Error! Bookmark not defined.
3.1.1. General Model of a Computer	Error! Bookmark not defined.
3.1.2. The Central Processing Unit (CPU)	Error! Bookmark not defined.
3.1.3. Memory	Error! Bookmark not defined.
3.1.4. Input-Output Devices	Error! Bookmark not defined.
3.1.5. Buses	Error! Bookmark not defined.
3.2. Computer Software	Error! Bookmark not defined.
3.2.1. Data and Algorithms	Error! Bookmark not defined.
3.2.2. Programs and Programming Languages	Error! Bookmark not defined.
3.2.3. Classification of Computer Software	Error! Bookmark not defined.
UNIT 4. COMPUTER NETWORKS	Error! Bookmark not defined.
4.1. History of Computer Networks	Error! Bookmark not defined.
4.2. Classification of Computer Networks	Error! Bookmark not defined.
4.3. Major Components of a Computer Network	Error! Bookmark not defined.
4.4. The Internet.....	Error! Bookmark not defined.
UNIT 5. PRINCIPLES OF OPERATING SYSTEMS	Error! Bookmark not defined.
5.1. Basic Concepts	Error! Bookmark not defined.
5.1.1. Functions of Operating Systems	Error! Bookmark not defined.
5.1.2. File Management	Error! Bookmark not defined.
5.1.3. File Management	Error! Bookmark not defined.
5.2. Some Typical Operating Systems.....	Error! Bookmark not defined.
5.2.1. MS-DOS	Error! Bookmark not defined.
5.2.2. Windows	Error! Bookmark not defined.
5.3. Commands of Operating Systems	Error! Bookmark not defined.
5.4. Windows.....	Error! Bookmark not defined.
5.4.1. Brief History of Windows	Error! Bookmark not defined.
5.4.2. Start and Exit From Windows XP	Error! Bookmark not defined.
5.4.3. Basic Terms and Operations	Error! Bookmark not defined.
5.4.4. The Control Panel	Error! Bookmark not defined.

5.4.5. Windows Explorer	Error! Bookmark not defined.
5.4.5.1. About Windows Explorer	Error! Bookmark not defined.
5.4.5.2. Managing Files and Folders	Error! Bookmark not defined.
5.4.6. Running Programs with Windows XP	Error! Bookmark not defined.
5.4.7. The Command Prompt	Error! Bookmark not defined.
5.4.8. The Recycle Bin	Error! Bookmark not defined.
PART 2. THE C PROGRAMMING LANGUAGE	Error! Bookmark not defined.
UNIT 1. AN OVERVIEW OF C	Error! Bo
1.1. History of the C Programming Language	Error! Bookmark not defined.
1.2. Basic Components of C Programs	Error! Bookmark not defined.
1.2.1. Symbols	Error! Bookmark not defined.
1.2.2. Keywords	Error! Bookmark not defined.
1.2.3. Identifiers	Error! Bookmark not defined.
1.2.4. Data Types	Error! Bookmark not defined.
1.2.5. Constants	Error! Bookmark not defined.
1.2.6. Variables	Error! Bookmark not defined.
1.2.7. Functions	Error! Bookmark not defined.
1.2.8. Expressions	Error! Bookmark not defined.
1.2.9. Statements	Error! Bookmark not defined.
1.2.10. Comments	Error! Bookmark not defined.
1.3. Program Structure in C	Error! Bookmark not defined.
1.4. C Compiler	Error! Bookmark not defined.
1.4.1. The Turbo C++ Compiler	Error! Bookmark not defined.
1.4.2. Install Turbo C++ 3.0	Error! Bookmark not defined.
1.4.3. Programming Environment of Turbo C++ 3.0	Error! Bookmark not defined.
1.5. Exercises	Error! Bookmark not defined.
UNIT 2. DATA TYPES AND EXPRESSIONS OF C (6 tiết)	Error! Bookmark not defined.
2.1. Standard Data Types	Error! Bookmark not defined.
2.1.1. Variable Declarations and Constant Definitions	Error! Bookmark not defined.
2.1.2. Functions printf, scanf	Error! Bookmark not defined.
2.1.3. Other Input and Output Functions	Error! Bookmark not defined.
2.2. Expressions	Error! Bookmark not defined.
2.2.1. Operators	Error! Bookmark not defined.
2.2.2. Arithmetic Operators	Error! Bookmark not defined.
2.2.3. Relational Operators	Error! Bookmark not defined.
2.2.4. Logical Operators	Error! Bookmark not defined.
2.2.5. Assignment Operators	Error! Bookmark not defined.
2.2.5. Precedence of Operators	Error! Bookmark not defined.

2.2.6. Special Operators of C	Error! Bookmark not defined.
UNIT 3. THE CONTROL FLOW	Error! Bo
3.1. Statements and Blocks	Error! Bookmark not defined.
3.2. If, If-else Statements	Error! Bookmark not defined.
3.3. Switch Statements	Error! Bookmark not defined.
3.4. Loops	Error! Bookmark not defined.
3.4.1. For Statement	Error! Bookmark not defined.
3.4.2. While Statement	Error! Bookmark not defined.
3.4.3. Do Statement	Error! Bookmark not defined.
3.5. Loop Flow Control	Error! Bookmark not defined.
3.5.1. Continue	Error! Bookmark not defined.
3.5.2. Break	Error! Bookmark not defined.
3.6. Exercises	Error! Bookmark not defined.
UNIT 4 ARRAYS AND POINTERS.	Error! Bo
4.1. Pointers and Addresses	Error! Bookmark not defined.
4.1.1. Pointers	Error! Bookmark not defined.
4.1.2. Operations on Pointers	Error! Bookmark not defined.
4.2. Arrays	Error! Bookmark not defined.
4.2.1. Basics of Arrays	Error! Bookmark not defined.
4.2.2. Declaration and Usage of Arrays	Error! Bookmark not defined.
4.2.3. Operations on Arrays	Error! Bookmark not defined.
Input Data to Arraysng	Error! Bookmark not defined.
Output Data from Arrays	Error! Bookmark not defined.
Maximal and Minimal Elements of an Array	Error! Bookmark not defined.
Searching on Arrays	Error! Bookmark not defined.
Sorting on Arrays	Error! Bookmark not defined.
4.2.4. Pointers vs. Arrays	Error! Bookmark not defined.
UNIT 5. STRINGS	Error! Bookmark not defined.
5.1. Basics of Strings	Error! Bookmark not defined.
5.2. Declaration and Usage of Strings	Error! Bookmark not defined.
5.3. Built in Functions for Characters and Strings	Error! Bookmark not defined.
5.3.1. Character Functions	Error! Bookmark not defined.
5.3.2. String Functions	Error! Bookmark not defined.
Input-Output with Strings	Error! Bookmark not defined.

Other String Functions	
Error! Bookmark not defined.	
Pointers vs Strings	
Error! Bookmark not defined.	
5.4. Exercises	Error! Bookmark not defined.
UNIT 6. FUNCTIONS.....	Error! Bo
6.1. Basics of C Functions	Error! Bookmark not defined.
6.2. Declaration and Usage of Functions in C.....	Error! Bookmark not defined.
6.2.1. Declaration	Error! Bookmark not defined.
Các thành phần của dòng đầu hàm	
Error! Bookmark not defined.	
6.2.2. Usage of Functions	Error! Bookmark not defined.
6.2.3. Classification of Variables : Global, Local, Static Variables	Error! Bookmark not defined.
Register, Static Statements	Error!
Bookmark not defined.	
6.2.4. Prototype of Functions.	Error! Bookmark not defined.
6.3. Exercise	Error! Bookmark not defined.
UNIT 7.STRUCTURES	Error! Bo
7.1. Basics of Structures	Error! Bookmark not defined.
7.2. Declarations and Usage of Structures	Error! Bookmark not defined.
7.3. Operations on Structures	Error! Bookmark not defined.
7.4. Arrays of Structures	Error! Bookmark not defined.
7.5. Exercises	Error! Bookmark not defined.
UNIT 8. FILES	Error! Bookmark no
8.1. Basics and Classification of Files	Error! Bookmark not defined.
8.2. Operations on Files	Error! Bookmark not defined.
8.2.1. Declaration	Error! Bookmark not defined.
8.2.2. Open File	Error! Bookmark not defined.
8.2.3. Access to Text Files	Error! Bookmark not defined.
8.2.4. Access to Binary Files	Error! Bookmark not defined.
8.2.5. Close File	Error! Bookmark not defined.
8.3. Exercises	Error! Bookmark not defined.
References.....	Error! Bookmark not defined.

Part I : Fundamentals of Information Technology

In this part, we introduce fundamentals of information technology. As the beginners of using computer, you need to know basic concepts : information, data, how to represent information in computers. In this part you will learn about the architecture of computer systems and have understanding of operating systems. Microsoft Windows will become a demonstration of an operating system.

This part includes 3 Units :

Unit 1 gives important definitions of information, data, knowledge, a description of information processing in computers, thence you will reach the concept of IT, TCT.

Unit 2 introduces methods of representing information in computers. You will know how to convert numbers (integer, rational, real) from decimal system to other numeral systems.

Unit 3 introduces architecture of typical computer systems and computer networks. This Unit also explain what an algorithm is, what a programming language is and you will know how to break a problem to steps of an algorithm.

Unit 1

Introduction

1.1.Information & Information Processing

Data – Information – Knowledge

The content of the human mind can be classified into four categories:

- Data: symbols
- Information: data that are processed to be useful; provides answers to "who", "what", "where", and "when" questions
- Knowledge: understanding of data and information; answers "how" questions
- Wisdom: evaluated understanding.

Data

Data consist of raw facts and figures - it does not have any meaning until it is processed and turned into something useful.

Data comes in many forms, the main ones being letters, numbers and symbols.

Data is a prerequisite to information. For example, the two data items below could represent some very important information:

DATA	INFORMATION
123424331911	Your winning Lottery ticket number
211192	Your Birthday

An organization sometimes has to decide on the nature and volume of data that is required for creating the necessary information.

Information

Information is data that has been processed in such a way as to be meaningful to the person who receives it.

$$\text{INFORMATION} = \text{DATA} + \text{CONTEXT} + \text{MEANING}$$

Example

Consider the number 19051890 .Is has no meaning or context. It's an instance of data.

If a context is given : it is a date (Vietnamese use French format ddmmyyyy). This allows us to register it as 19th May 1890. It still has no meaning and is therefore not information

Meaning : The Birth date of President Ho Chi Minh.

This gives us all the elements required for it to be called 'information'

Knowledge

By knowledge we mean human understanding of a subject matter that has been acquired through proper study and experience.

Knowledge is usually based on learning, thinking, and proper understanding of the problem area. It can be considered as the integration of human perceptive processes that helps them to draw meaningful conclusions.

Consider this scenario: *Person puts a finger into very hot water.*

Data gathered: Finger nerves sends pain data to the brain.

Processing: Brain considers the data and comes up with...

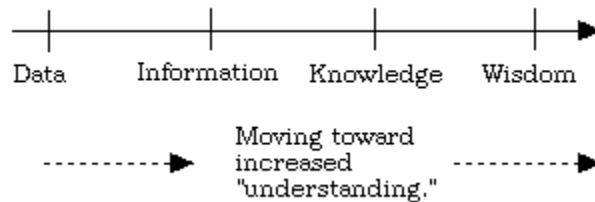
Information: Painful finger means it is not in a good place.

Action: Brain tells finger to remove itself from hot water.

Knowledge: Sticking finger in hot water is a bad idea.

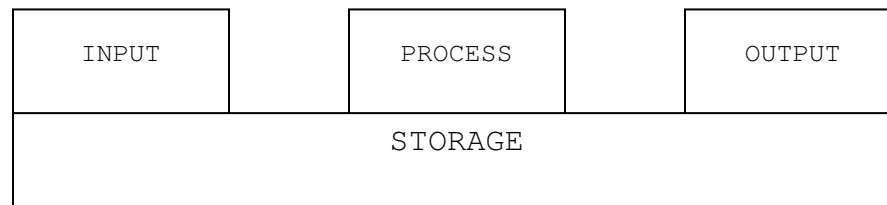
Knowledge is having an understanding of the "rules".

The terms Data, Information, Knowledge, and Wisdom are sometimes presented in a form that suggests a scale.



Information Processing

Information processing is the change (processing) of information in any manner detectable by an observer. Information processing may more specifically be defined in terms by Claude E. Shannon as the conversion of latent information into manifest. Input, process, output is a typical model for information processing. Each stage possibly requires data storage.



Now that computer systems have become so powerful, some have been designed to make use of information in a knowledgeable way. The following definition is of information processing

The electronic capture, collection, storage, manipulation, transmission, retrieval, and presentation of information in the form of data, text, voice, or image and includes telecommunications and office automation functions.

History and Classification of Computers

1.2.1. History of Computers

Webster's Dictionary defines "computer" as any programmable electronic device that can store, retrieve, and process data.

Blaise Pascal invents the first commercial calculator, a hand powered adding machine

In 1946, ENIAC, based on John Von Neuman model completes. The first commercially successful computer is IBM 701.

A generation refers to the state of improvement in the development of a product. This term is also used in the different advancements of computer technology. With each generation, the circuitry has gotten smaller and more advance than the previous generations before it. As a result of the miniaturization, speed, power and memory of computers has proportionally increased. New discoveries are constantly being developed that affect the the way we live, work and play. In terms of technological developments over time, computers have been broadly classed into five generations.

The First Generation - 1940-1956

The first computers used vacuum tubes for circuitry and magnetic drums for memory, and were often enormous, taking up entire rooms. They were very expensive to operate and in addition to using a great deal of electricity, generated a lot of heat, which was often the cause of malfunctions. First generation computers relied on machine language to perform operations, and they could only solve one problem at a time. Input was based on punched cards and paper tape, and output was displayed on printouts.

The computers UNIVAC , ENIAC of the US and BESEM of the former Soviet Union are examples of first-generation computing devices.

The Second Generation - 1956-1963

Transistors replaced vacuum tubes and ushered in the second generation of computers. Computers to become smaller, faster, cheaper, more energy-efficient and more reliable than their first-generation predecessors. Second-generation computers still relied on punched cards for input and printouts for output. High-level programming languages were being developed, such as early versions of COBOL and FORTRAN.

The first computers of this generation were developed for the atomic energy industry.

The computers IBM-1070 of the US and MINSK of the former Soviet Union belonged to the second generation.

The Third Generation - 1964-1971: Integrated Circuits

The development of the integrated circuit was the hallmark of the third generation of computers. Transistors were miniaturized and placed on silicon chips, called semiconductors, which drastically increased the speed and efficiency of computers. Users interacted with third generation computers through keyboards and monitors and interfaced with an operating system, which allowed the device to run many different applications at one time. Typical computers of the third generation are IBM 360 (United States) and EC (former Soviet Union).

The Fourth Generation - 1971-Present: Microprocessors

The microprocessor brought the fourth generation of computers, as thousands of integrated circuits were built onto a single silicon chip. What in the first generation filled an entire room could now fit in the palm of the hand. The Intel 4004 chip, developed in

1971, located all the components of the computer - from the central processing unit and memory to input/output controls - on a single chip.

In 1981 IBM introduced its first computer for the home user, and in 1984 Apple introduced the Macintosh. Microprocessors also moved out of the realm of desktop computers and into many areas of life as more and more everyday products began to use microprocessors.

As these small computers became more powerful, they could be linked together to form networks, which eventually led to the development of the Internet. Fourth generation computers also saw the development of GUI (Graphic User Interface), the mouse and handheld devices.

The Fifth Generation - Present and Beyond: Artificial Intelligence

Fifth generation computing devices, based on artificial intelligence, are still in development, though there are some applications, such as voice recognition, that are being used today. The use of parallel processing and superconductors is helping to make artificial intelligence a reality. Quantum computation and molecular and nanotechnology will radically change the face of computers in years to come. The goal of fifth-generation computing is to develop devices that respond to natural language input and are capable of learning and self-organization.

1.2.2. Classification of Computers

Computers are available in different shapes, sizes and weights, due to these different shapes and sizes they perform different sorts of jobs from one another.

- **Mainframe and Super Computers :**

The biggest in size, the most expensive in price than any other is classified and known as super computer. It can process trillions of instructions in seconds. Governments specially use this type of computer for their different calculations and heavy jobs. Different industries also use this huge computer for designing their products. This kind of computer is also helpful for forecasting weather reports worldwide.

Another giant in computers after the super computer is Mainframe, which can also process millions of instruction per second and capable of accessing billions of data. This computer is commonly used in big hospitals, air line reservations companies, and many other huge companies prefer mainframe because of its capability of retrieving data on a huge basis. This is normally too expensive and out of reach from a salary-based person who wants a computer for his home.

- **Minicomputers**

This computer offers less than mainframe in work and performance. These are the computers, which are mostly preferred by the small type of business personals, colleges, etc.

- **Microcomputers**

These computers are lesser in cost than the computers given above and also, small in size; They can store a big amount of data and having a memory to meet the assignments of students and other necessary tasks of business people. There are many types of microcomputers: desktop, workstation, laptop, PDA, etc.

Computer Science and Relevant Sciences

Computer Science

In 1957 the German computer scientist Karl Steinbuch coined the word *informatik* by publishing a paper called Informatik: Automatische Informationsverarbeitung (i.e. "Informatics: automatic information processing"). The French term *informatique* was coined in 1962 by Philippe Dreyfus together with various translations—*informatics* (English), *informatica* (Italian, Spanish, Portuguese), *informatika* (Russian) referring to the application of computers to store and process information.

The term was coined as a combination of "information" and "automation", to describe the science of automatic information processing.

Informatics is more oriented towards mathematics than *computer science*.

Computer Science is the study of computers, including both hardware and software design. Computer science is composed of many broad disciplines, for instance, artificial intelligence and software engineering.

Information Technology

Includes all matters concerned with the furtherance of computer science and technology and with the design, development, installation, and implementation of information systems and applications

Information and Communication Technology

ICT (information and communications technology - or technologies) is an umbrella term that includes any communication device or application, encompassing: radio, television, cellular phones, computer and network hardware and software, satellite systems and so on, as well as the various services and applications associated with them, such as videoconferencing and distance learning.

Unit 2.

Representation of Information in Computers

Computer must not only be able to carry out computations, they must be able to do them quickly and efficiently. There are several data representations, typically for integers, real numbers, characters, and logical values.

2.1. Numeral Systems

The system has ten as its base

Uses various symbols (called digits) for no more than ten distinct values (0, 1, 2, 3, 4, 5, 6, 7, 8 and 9) to represent any number

Decimal separator indicates the start of a fractional part,

Sign symbols + (positive) or – (negative) in front of the numerals to indicate sign

2.1.1. Decimal System

If i is an integer written in decimal form with digits d_j :

$$i = d_{n-1}d_{n-2} \dots d_1d_0$$

then i represents the sum:

$$\sum_{j=0}^{n-1} d_j * 10^j$$

where n is the total number of digits, and d_j is the j th digit from the rightmost position in the decimal number.

2.1.2. k – ary System

k digits is used in the representation of numbers.

Example

$1410_{10} = 226_6$ If i is an integer written in base - k with digits d_j :

$$i = d_{n-1}d_{n-2} \dots d_1d_0$$

then i represents the sum:

$$\sum_{j=0}^{n-1} d_j * k^j$$

where n is the total number of digits, and d_j is the j th digit from the rightmost position in the decimal number.

Addition in Base k

Use the standard rules to add in base k , making sure we ‘carry’ whenever we get a value of k or larger.

Example:

- 0001001012
- + 0011001002
- = 0100010012

- 01200021103
- + 00120021023
- = 02020102123

Converting from base- k to decimal

Write the number as a polynomial in k , where k is in its base-10 equivalent

In the polynomial, convert coefficient to base-10 notation

Compute the value of the polynomial, carrying out all calculations in base-10; this is the base-10 equivalent of the base- k number.

Example : To convert the number 1AC14

$$1 \cdot 142 + A \cdot 14 + C$$

$$1 \cdot 142 + 10 \cdot 14 + C$$

$$\bullet \quad 196 + 140 + 12 = 348$$

Remainder Method:

Let $value = (d_{n-1} d_{n-2} \dots d_2 d_1 d_0)_{10}$.

First divide $value$ by k , the remainder is the least significant digit b_0 .

Divide the result by k , the remainder is b_1 .

Continue this process until the result is less than k , giving the most significant digit, b_{n-1} .

• Example

• What is the base 2 (binary) representation of 4210?

• Base=	• 2	•
value	• Value \ 2	• Value mod 2
42	• 21	• 0
• 21	• 10	• 1
• 10	• 5	• 0
• 5	• 2	• 1
• 2	• 1	• 0
• 1	• 0	• 1
• Answer	• =	• 101010

• 2.1.3. Binary System

• All data, including programs, in a computer system is represented in terms of groups of binary digits

• A single bit can represent one of two values, 0 or 1.

• If we have several symbols to represent, we can make a one-to-one correspondence between the patterns and the symbols.

• Example : 0, 1, 2, 3 are mapped to the patterns 00, 01, 10, 11

• A group of k binary digits (bits) can be used to represent 2^k symbols

• Such a correspondence is called a code and the process of representing a symbol by the corresponding binary pattern is called coding or encoding.

• 2.1.4. Hexadecimal System

As a computer programmer, you might sometimes be required to work with numbers expressed in binary and hexadecimal notation. This appendix explains what these systems are and how they work. To help you understand, let's first review the common decimal number system.

The Decimal Number System

The decimal system is the base-10 system that you use every day. A number in this system--for example, 342--is expressed as powers of 10. The first digit (counting from the right) gives 10 to the 0 power, the second digit gives 10 to the 1 power, and

so on. Any number to the 0 power equals 1, and any number to the 1 power equals itself. Thus, continuing with the example of 342, you have:

$$3 \quad 3 * 10^2 = 3 * 100 = 300$$

$$4 \quad 4 * 10^1 = 4 * 10 = 40$$

$$2 \quad 2 * 10^0 = 2 * 1 = 2$$

$$\text{Sum} = 342$$

The base-10 system requires 10 different digits, 0 through 9. The following rules apply to base 10 and to any other base number system:

A number is represented as powers of the system's base.

The system of base n requires n different digits.

Now let's look at the other number systems.

The Binary System

The binary number system is base 2 and therefore requires only two digits, 0 and 1. The binary system is useful for computer programmers, because it can be used to represent the digital on/off method in which computer chips and memory work. Here's an example of a binary number and its representation in the decimal notation you're more familiar with, writing 1011 vertically:

$$1 \quad 1 * 2^3 = 1 * 8 = 8$$

$$0 \quad 0 * 2^2 = 0 * 4 = 0$$

$$1 \quad 1 * 2^1 = 1 * 2 = 2$$

$$1 \quad 1 * 2^0 = 1 * 1 = 1$$

$$\text{Sum} = 11 \text{ (decimal)}$$

Binary has one shortcoming: It's cumbersome for representing large numbers.

The Hexadecimal System

The hexadecimal system is base 16. Therefore, it requires 16 digits. The digits 0 through 9 are used, along with the letters A through F, which represent the decimal values 10 through 15. Here is an example of a hexadecimal number and its decimal equivalent:

$$2 \quad 2 * 16^2 = 2 * 256 = 512$$

$$D \quad 13 * 16^1 = 13 * 16 = 208$$

$$A \quad 10 * 16^0 = 10 * 1 = 10$$

$$\text{Sum} = 730 \text{ (decimal)}$$

The hexadecimal system (often called the hex system) is useful in computer work because it's based on powers of 2. Each digit in the hex system is equivalent to a four-

digit binary number, and each two-digit hex number is equivalent to an eight-digit binary number. Table C.1 shows some hex/decimal/binary equivalents.

Table C.1. Hexadecimal numbers and their decimal and binary equivalents.

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0 0	0000	
1 1	0001	
2 2	0010	
3 3	0011	
4 4	0100	
5 5	0101	
6 6	0110	
7 7	0111	
8 8	1000	
9 9	1001	
A 10	1010	
B 11	1011	
C 12	1100	
D 13	1101	
E 14	1110	
F 15	1111	
10 16	10000	
F0 240	11110000	
FF 255	11111111	

2.2. Coding Information in Computer - Units of Information

2.3. Representation of Integers

2.3.1. Unsigned Integers

Unsigned integers are represented by a fixed number of bits (typically 8, 16, 32, and/or 64)

Only a finite set of numbers that can be represented:

With 8 bits, 0...255 (0016...FF16) can be represented;

With 16 bits, 0...65535 (000016...FFFF16) can be represented

If an operation on bytes has a result outside this range, it will cause an 'overflow'

2.3.2. Signed Integers

The most significant bit is set to 0 and 1 for positive and negative numbers

Example

+4210 = 001010102

and so

-4210 = 110101102

All number whose leftmost bit is 1 is considered negative

If we have 4 bits in our representation then,

The most positive representable number using sign-and-magnitude is 0111

The most negative representable number using sign-and-magnitude is 1000

How to represent negative number?

Two's Complement

Representation for signed binary numbers

Leading bit is a sign bit

Binary number with leading 0 is positive

Binary number with leading 1 is negative

Magnitude of positive numbers is just the binary representation

Magnitude of negative numbers is found by

Complement the bits

Replace all the 1's with 0's, and all the 0's with 1's

Add one to the complemented number

The carry in the most significant bit position is thrown away when performing arithmetic

Performing two's complement on the decimal 42 to get -42

Using a eight-bit representation

42 = 00101010 Convert to binary

11010101 Complement the bits

11010101 Add 1 to the complement

+ 00000001

11010110 Result is -42 in two's complement

Two's Complement Arithmetic

Computing 50 - 42 using a two's complement representation with eight-bit numbers

50 - 42 = 50 + (-42) = 8

00110010 Two's complement of 50

11010110 Two's complement of -42

00110010 Add 50 and -42

+ 11010110

100001000

00001000 Is the eight-bit result

Throw away the high-order carry as we are using a eight bit representation

2.4. Operations on Integers

2.4.1. Addition and Subtraction

Binary Addition

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 0, and carry 1 to the next more significant bit

For example, $00011010 + 00001100 = 00100110$ 1 1 carries

0 0 0 1 1 0 1 0	=	26(base 10)	
+ 0 0 0 0 1 1 0 0	=	12(base 10)	
0 0 1 0 0 1 1 0	=	38(base 10)	

$00010011 + 00111110 = 01010001$ 1 1 1 1 1 carries

0 0 0 1 0 0 1 1	=	19(base 10)	
+ 0 0 1 1 1 1 1 0	=	62(base 10)	
0 1 0 1 0 0 0 1	=	81(base 10)	

Binary Subtraction

0 - 0 = 0

0 - 1 = 1, and borrow 1 from the next more significant bit

1 - 0 = 1

1 - 1 = 0

2.4.2. Multiplication and Division

Binary Multiplication

$0 \times 0 = 0$

$0 \times 1 = 0$

$1 \times 0 = 0$

$1 \times 1 = 1$, and no carry or borrow bits

For example, $00101001 \times 00000110 = 11110110$

0 0 1 0 1 0 0 1	=	41(base 10)	
× <u>0 0 0 0 0 1 1 0</u>	=	6(base 10)	
0 0 0 0 0 0 0 0			
0 1 0 1 0 0 1			
<u>0 1 0 1 0 0 1</u>			
0 0 1 1 1 1 0 1 1 0	=	246(base 10)	

$00010111 \times 00000011 = 01000101$

0 0 0 1 0 1 1 1	=	23(base 10)	
× <u>0 0 0 0 0 0 1 1</u>	=	3(base 10)	
1 1 1 1 1	carries		
0 0 1 0 1 1 1			
0 0 1 0 1 1 1			
0 0 1 0 0 0 1 0 1	=	69(base 10)	

Binary division is the repeated process of subtraction, just as in decimal division.

2.5. Logical operations on Binary Numbers

NOT operation

- The NOT operation, or complement, is a unary operation which performs logical negation on each bit, forming the ones' complement of the given binary value. Digits which were 0 become 1, and vice versa.

- For example:

- NOT 0111

- = 1000

AND operation

- An AND operation takes two binary representations of equal length and performs the logical AND operation on each pair of corresponding bits. In each pair, the result is 1 if the first bit is 1 AND the second bit is 1. Otherwise, the result is 0.

- For example:

- 0101

- AND 0011

- = 0001

OR operation

- An OR operation takes two bit patterns of equal length, and produces another one of the same length by matching up corresponding bits (the first of each; the second of each; and so on) and performing the logical OR operation on each pair of corresponding bits.

- For example:

- 0101

- OR 0011

- = 0111

XOR Operation

- An exclusive or operation takes two bit patterns of equal length and performs the logical XOR operation on each pair of corresponding bits.

- For example:

- 0101

- XOR 0011

- = 0110

Logical Operation

AND

- **The logical AND operation compares 2 bits and if they are both "1", then the result is "1", otherwise, the result is "0".**

- 1

- 0 0 0

- 1 0 1

OR

- **The logical OR operation compares 2 bits and if either or both bits are "1", then the result is "1", otherwise, the result is "0".**

- 1

- 0 0 1

- 1 1 1

- **XOR**
- The logical XOR (Exclusive OR) operation compares 2 bits and if exactly one of them is "1" (i.e., if they are different values), then the result is "1"; otherwise (if the bits are the same), the result is "0".
- 1
- 0 0 1
- 1 1 0
- **NOT**
- The logical NOT operation simply changes the value of a single bit. If it is a "1", the result is "0"; if it is a "0", the result is "1". Note that this operation is different in that instead of comparing two bits, it is acting on a single bit.
- 1
- 0
- 2.6. Symbol Representation
-
- 2.6.1. Basic Principles
- It's important to handle character data
- Character data isn't just alphabetic characters, but also numeric characters, punctuation, spaces, etc
- They need to be represented in binary
- There aren't mathematical properties for character data, so assigning binary codes for characters is somewhat arbitrary
-
- 2.6.2. ASCII Code Table
- ASCII -- American Standard Code for Information Interchange - permitted machines from different manufacturers to exchange data.
- The ASCII standard was developed in 1963
- ASCII consists of 128 binary values (0 to 127), each associated with a character or command
- Now the non-printing characters are rarely used for their original purpose

Non-Printing Characters					Printing				
Name	Ctrl char	Dec	Hex	Char		Dec	Hex	Char	Dec
null	ctrl-@	0	00	NUL		32	20	Space	64
start of heading	ctrl-A	1	01	SOH		33	21	!	65
start of text	ctrl-B	2	02	STX		34	22	"	66
end of text	ctrl-C	3	03	ETX		35	23	#	67
end of transmit	ctrl-D	4	04	EOT		36	24	\$	68
enquiry	ctrl-E	5	05	ENQ		37	25	%	69
acknowledge	ctrl-F	6	06	ACK		38	26	&	70
bell	ctrl-G	7	07	BEL		39	27	'	71
backspace	ctrl-H	8	08	BS		40	28	(72
horizontal tab	ctrl-I	9	09	HT		41	29)	73
line feed	ctrl-J	10	0A	LF		42	2A	*	74
vertical tab	ctrl-K	11	0B	VT		43	2B	+	75
form feed	ctrl-L	12	0C	FF		44	2C	,	76
carriage feed	ctrl-M	13	0D	CR		45	2D	-	77
shift out	ctrl-N	14	0E	SO		46	2E	.	78
shift in	ctrl-O	15	0F	SI		47	2F	/	79
data line escape	ctrl-P	16	10	DLE		48	30	0	80
device control 1	ctrl-Q	17	11	DC1		49	31	1	81
device control 2	ctrl-R	18	12	DC2		50	32	2	82
device control 3	ctrl-S	19	13	DC3		51	33	3	83
device control 4	ctrl-T	20	14	DC4		52	34	4	84
negative acknowledge	ctrl-U	21	15	NAK		53	35	5	85
synchronous idel	ctrl-V	22	16	SYN		54	36	6	86
end of transmit block	ctrl-W	23	17	ETB		55	37	7	87
cancel	ctrl-X	24	18	CAN		56	38	8	88
end of medium	ctrl-Y	25	19	EM		57	39	9	89
substitute	ctrl-Z	26	1A	SUB		58	3A	:	90
escape	ctrl-[27	1B	ESC		59	3B	;	91
file separator	ctrl-\	28	1C	FS		60	3C	<	92
group separator	ctrl-]	29	1D	GS		61	3D	=	93
record separator	ctrl-^	30	1E	RS		62	3E	>	94
unit separator	ctrl-`	31	1F	US		63	3F	?	95

-
- ASCII coding table
-

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	À
129	81	ù	161	A1	í	193	C1	Á
130	82	é	162	A2	ó	194	C2	Â
131	83	â	163	A3	ú	195	C3	Ã
132	84	ä	164	A4	ñ	196	C4	Ä
133	85	à	165	A5	Ñ	197	C5	Å
134	86	ä	166	A6	ª	198	C6	Æ
135	87	ç	167	A7	º	199	C7	Ç
136	88	ê	168	A8	¿	200	C8	È
137	89	ë	169	A9	¸	201	C9	É
138	8A	è	170	AA	¸	202	CA	Ê
139	8B	ï	171	AB	½	203	CB	Ë
140	8C	î	172	AC	¾	204	CC	Ì
141	8D	ì	173	AD	¸	205	CD	Í
142	8E	Ä	174	AE	«	206	CE	Î
143	8F	Å	175	AF	»	207	CF	Ï
144	90	É	176	B0	░	208	DO	Ð
145	91	æ	177	B1	▒	209	D1	Ñ
146	92	Æ	178	B2	▓	210	D2	Ò
147	93	ó	179	B3		211	D3	Ó
148	94	ô	180	B4	¸	212	D4	Ô
149	95	ò	181	B5	¸	213	D5	Õ
150	96	û	182	B6	¸	214	D6	Ö
151	97	ù	183	B7	¸	215	D7	Û
152	98	ý	184	B8	¸	216	D8	Ü
153	99	Ö	185	B9	¸	217	D9	Ý
154	9A	Û	186	BA	¸	218	DA	Þ
155	9B	◊	187	BB	¸	219	DB	ß
156	9C	£	188	BC	¸	220	DC	
157	9D	¥	189	BD	¸	221	DD	
158	9E	₹	190	BE	¸	222	DE	
159	9F	ƒ	191	BF	¸	223	DF	

- The extended ASCII characters
- Problems of ASCII
- String datatypes allocated one byte per character.
- Logographic languages such as Chinese, Japanese, and Korean need far more than 256 characters for reasonable representation.
- Vietnamese need 61 characters for representation.
- Where can we find number for our characters?
- ⇒ 2bytes per character?
- 2.6.3. Unicode Code Table
- Before Unicode was invented, there were hundreds of different encoding systems
- No single encoding could contain enough characters

- These encoding systems conflict with one another : two encodings can use the same number for two different characters, or use different numbers for the same character.
- Unicode provides a unique number for every character
- The Unicode Standard has been adopted by such industry leaders as HP, IBM, Microsoft, Oracle, Sun, and many others.
- It is supported in many operating systems, all modern browsers, and many other products.
-
- Advantages of using Unicode
- Offers significant cost savings over the use of legacy character sets.
- Enables a single software product or a single website to be targeted across multiple platforms, languages and countries without re-engineering.
- Allows data to be transported through many different systems without corruption.
- 2.7. Representation of Real Numbers
- 2.7.1. Basic Principles

Converting Decimal Fractions to Binary

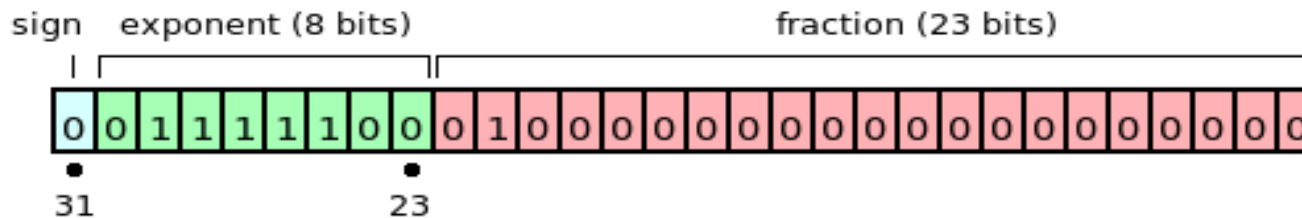
Step 1: Begin with the decimal fraction and multiply by 2. The whole number part of the result is the first binary digit to the right of the point.

Step 2: Next we disregard the whole number part of the previous result and multiply by 2 once again. The whole number part of this new result is the second binary digit to the right of the point. We will continue this process until we get a zero as our decimal part or until we recognize an infinite repeating pattern.

- Example
- Binary representation of the decimal fraction 0.1
 - Because $.1 \times 2 = 0.2$, the first binary digit to the right of the point is a 0
 - Because $.2 \times 2 = 0.4$, the second binary digit to the right of the point is also a 0.
 - Because $.4 \times 2 = 0.8$, the third binary digit to the right of the point is also a 0.
 - Because $.8 \times 2 = 1.6$, the fourth binary digit to the right of the point is a 1.
 - Because $.6 \times 2 = 1.2$, the fifth binary digit to the right of the point is a 1.
 - The next step to be performed (multiply 2. x 2) is exactly the same action we had in step 2. We are then bound to repeat steps 2-5, then return to Step 2 again indefinitely
- $.1$ (decimal) = .00011001100110011 . . .
- The repeating pattern is 0011
- Floating Point Representation

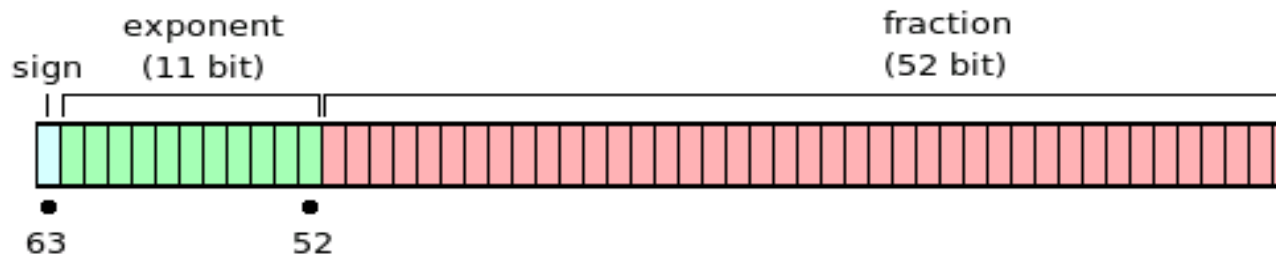
- No human system of numeration can give a unique representation to real numbers
- One approach : a real number x can be approximated by any number in the range from $x - \epsilon$ to $x + \epsilon$ (fixed-point representation)
- Fixed-point representations are unsatisfactory for most applications involving real numbers
- Scientific notation: a number is expressed as the product of a mantissa and some power of ten.
- A system of numeration for real numbers will typically store the same three data - a sign, a mantissa, and an exponent -- into an allocated region of storage
- The analogues of scientific notation in computer are described as floating-point representations.
- In the decimal system, the decimal point indicates the start of negative powers of 10.
- $12.34 = 1 \cdot 10^1 + 2 \cdot 10^0 + 3 \cdot 10^{-1} + 4 \cdot 10^{-2}$
- If we are using a system in base k (ie the radix is k), the 'radix point' serves the same function:
- $$\begin{aligned} 101.1012 &= 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} \\ &= 410 + 110 + 0.510 + 0.12510 \\ &= 5.62510 \end{aligned}$$
- A *floating point representation* allows a large range of numbers to be represented in a relatively small number of digits by separating the digits used for *precision* from the digits used for *range*.
- The following have 4 decimal digits of precision, but together cover a large range of values
- $1234 = 1.234 \times 10^3$
- $0.000000001234 = 1.234 \times 10^{-9}$
- $12,340,000,000 = 1.234 \times 10^9$
- One number can be represented in different ways.
- The following are all the same value
- 3.14159×10^0 - normalized
- 0.314159×10^1 ,
- 31.4159×10^{-1}
- 3141.59×10^{-3}
- To avoid multiple representations of the same number floating point numbers are usually *normalized* so that there is only one nonzero digit to the left of the 'radix' point, called the leading digit.
- Representation of normalized floating point numbers
- A normalized (non-zero) floating-point number will be represented using
 - $(-1)^s d_0.d_1d_2\dots d_{p-1} \times k^e$,
- where
- s is the sign,
- $d_0.d_1d_2\dots d_{p-1}$ - termed the *significand* - has p significant digits
- each digit satisfies $0 \leq d_i < k$
- e , $e_{\min} \leq e \leq e_{\max}$, is the exponent

- k is the base (or radix)
- Example:
- If $k = 10$ (base 10) and $p = 3$, the number 0.1 is represented as 0.100
- If $k = 2$ (base 2) and $p = 24$, the decimal number 0.1 cannot be represented exactly but is approximately $1.10011001100110011001101 \times 2^{-4}$.
- Formally, $(-1)^s d_0.d_1d_2\dots d_{p-1} \times k^e$ represents the value $(-1)^s (d_0 + d_1k^{-1} + d_2k^{-2} + \dots + d_{p-1}k^{-(p-1)}) k^e$
- Precision & Range allowed by a Representation
- The **range** of numbers : the number of digits in the exponent (i.e. by e_{\max}) and the base k to which it is raised
- The **precision** : the number of digits p in the significand and its base k
- 2.7.2. IEEE 754/85 Standard
- There are many ways to represent floating point numbers.
- In order to improve portability most computers use the IEEE 754 floating point standard.
-
- There are two primary formats:
 - 32 bit single precision
 - 64 bit double precision.
- Single precision consists of:
 - A single sign bit, 0 for positive and 1 for negative;
 - An 8 bit base-2 ($k=2$) excess-127 exponent, with $e_{\min} = -126$ (stored as $12710 - 12610 = 110 = 000000012$) and $e_{\max} = 127$ (stored as $12710 + 12710 = 25410 = 111111102$).
 - a 23 bit base-2 ($k=2$) significand, with a hidden bit giving a precision of 24 bits (i.e. $1.d_1d_2\dots d_{23}$)



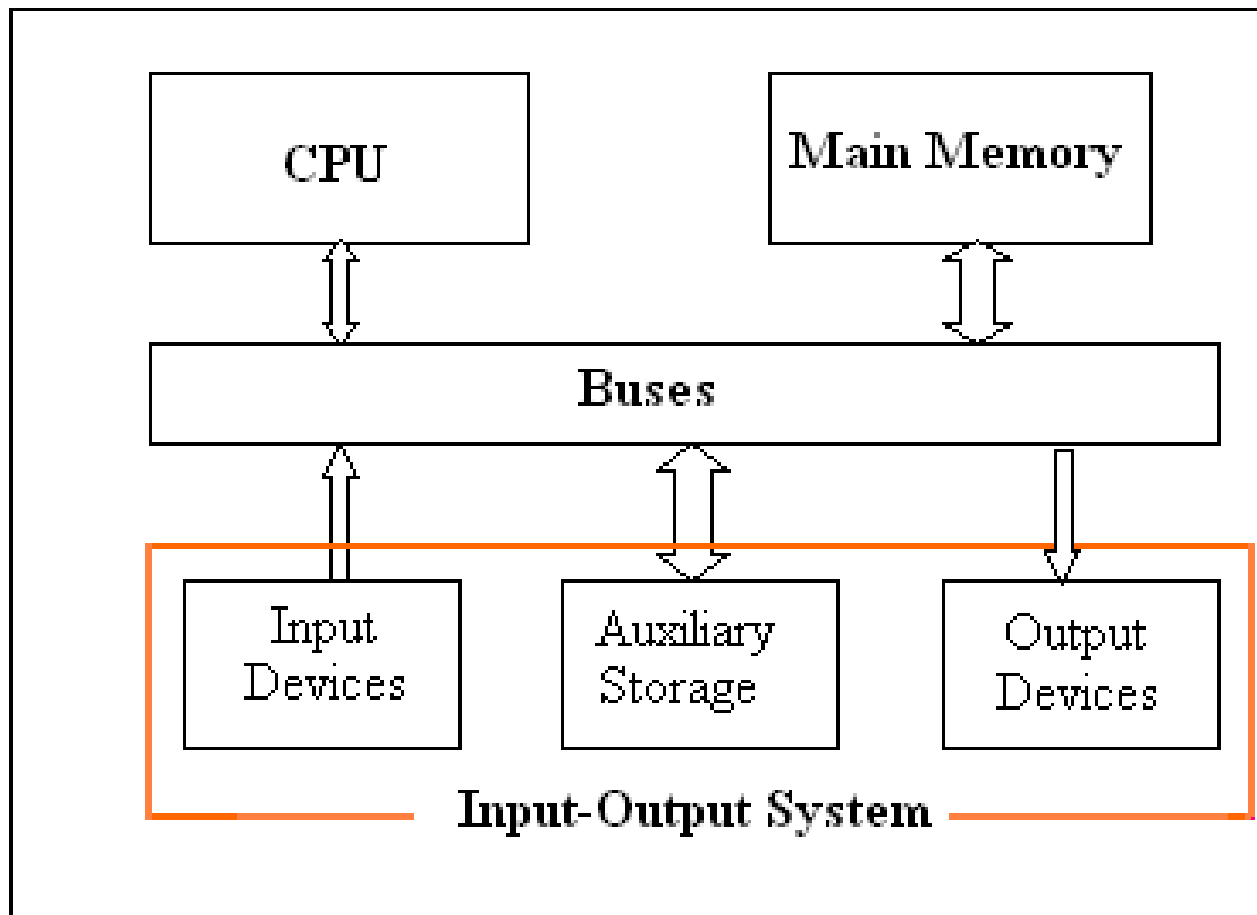
- Notes
- Single precision has 24 bits precision, equivalent to about 7.2 decimal digits.
- The largest representable non-infinite number is almost $2 \times 2^{127} \approx 3.402823 \times 10^{38}$
- The smallest representable non-zero normalized number is $1 \times 2^{-126} \approx 1.17549 \times 10^{-38}$
- *Denormalized numbers* (eg 0.01×2^{-126}) can be represented.
- There are two zeros, ± 0 .
- There are two *infinities*, $\pm \infty$.

- A *NaN* (not a number) is used for results from undefined operations
- IEEE 754 double precision floating point standard
- Requires a 64 bit word
- The first bit is the sign bit
- The next eleven bits are the exponent bits
- The final 52 bits are the fraction
- Range of double numbers : $[\pm 2.225 \times 10^{-308} \div \pm 1.7977 \times 10^{308}]$



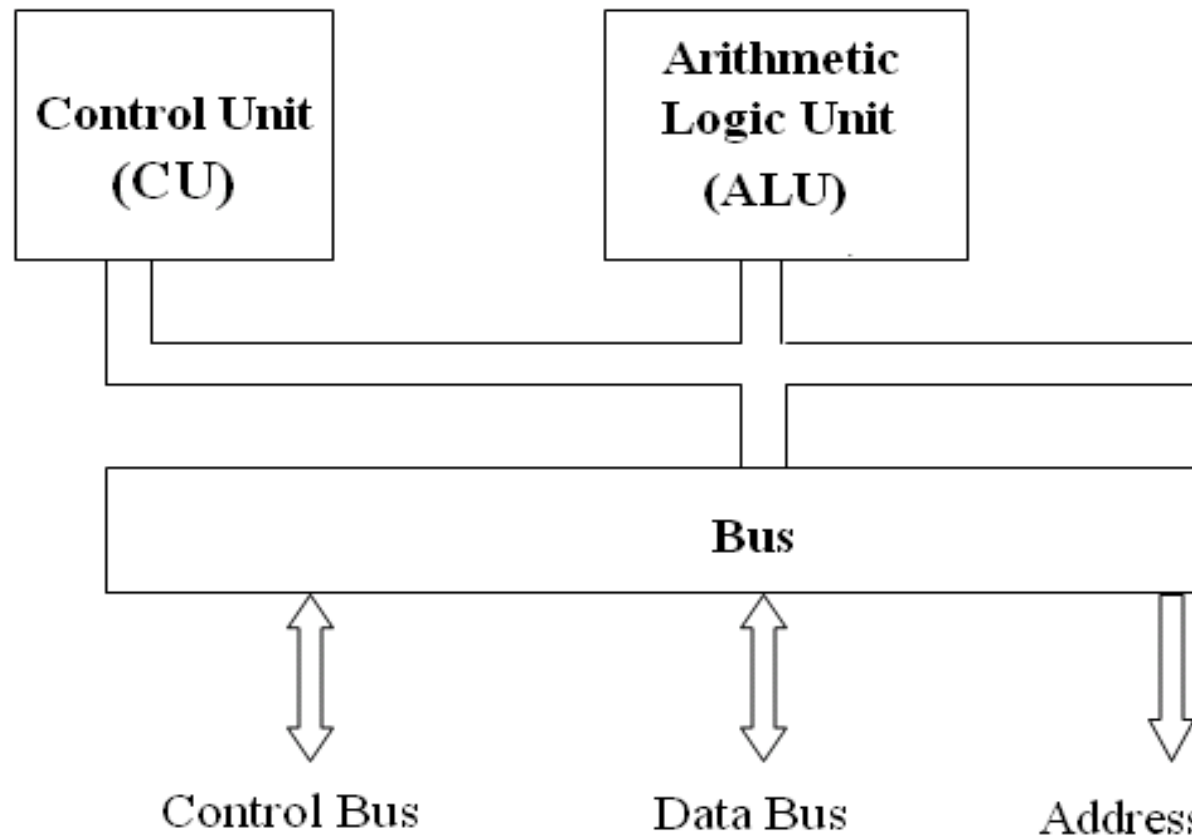
-
- Unit 3. Computer Systems
- 3.1. Computer Architecture
- Computer Architecture
- Just as buildings, each computer has a “visible” structure referred to as its architecture. The architecture of a building can be examined at various level of detail :the number of stories and the size of rooms, the detail of door and window placement . . .We can look at a computer’s architecture at similar level of detail.
- At the highest level, a computer can be seen as a collection of component connected by a device called a BUS. The bus is a medium for communication amongst the components, with each component having an address on the bus (the BUS ADDRESS). If one component must transfer a packet of data to another device, it will create a message (containing the data) and put it on the bus along with the bus address of the target device; each device “listen” to the bus and removes those message with the appropriate bus address.
- The components connected by the bus are those
-
- 3.1.1. General Model of a Computer
- The functions of a computer
 - Data Processing
 - Data Storage
 - Data Interchange
 - Control

- The activities of the central processor are cyclical. The processor fetches an instruction, performs the operations required, fetches the next instruction, and so on
- The CPU requires a free running oscillator clock which furnishes the reference for all processor actions
- The combined fetch and execution of a single instruction is referred to as an Instruction Cycle.
- When the entire instruction is present in the CPU, the program counter is incremented (in preparation for the next instruction fetch) and the instruction is decoded
- The instruction may call for a memory read or write, an input or output and/or an internal CPU operation, such as a register to register transfer or an add registers operation.

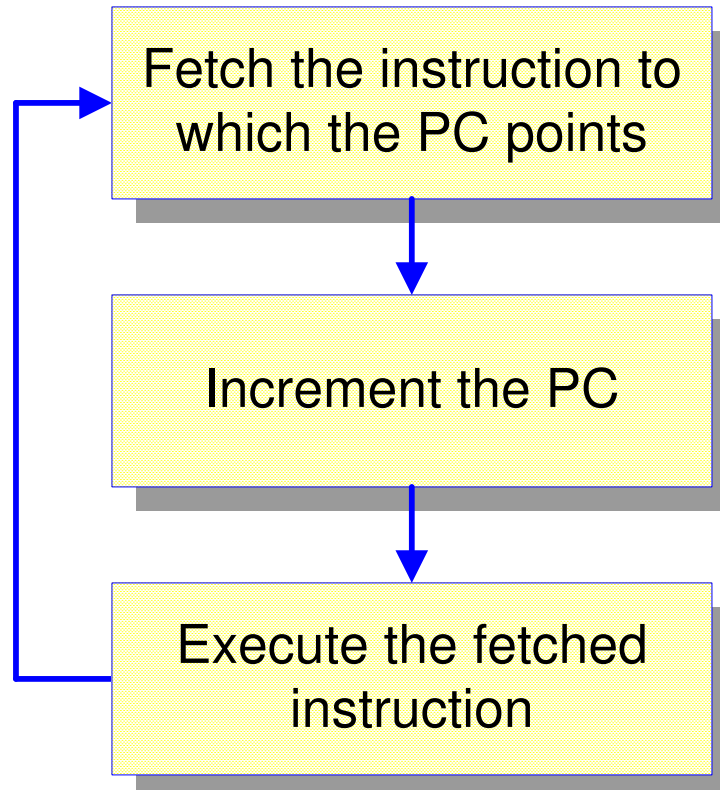


- *Fig. Computer Organization*
-
- Architecture of Computer Systems
-
- Computer Operations
-
- 3.1.2. The Central Processing Unit (CPU)

- *Brains* of the computer
- Arithmetic calculations are performed using the Arithmetic/Logical Unit or ALU
- Control unit decodes and executes instructions
- Arithmetic operations are performed using binary number system
- CPU is contained on one (or a small number of) integrated circuits called microprocessors

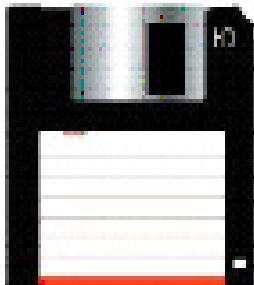


-
- Basic Model of the Central Processing Unit (CPU)
- Arithmetic Logic Units (ALU)
- The ALU, as its name implies, is that portion of the CPU hardware which performs the arithmetic and logical operations on the binary data .
- The ALU contains an Adder which is capable of combining the contents of two registers in accordance with the logic of binary arithmetic
- Control Unit
- The fetch/execute cycle is the steps the CPU takes to execute an instruction
- Performing the action specified by an instruction is known as *executing the instruction*
- The program counter (PC) holds the memory address of the next instruction



- Registers
- Registers are temporary storage units within the CPU.
- Some registers, such as the program counter and instruction register, have dedicated uses.
- Other registers, such as the accumulator, are for more general purpose use.
- Clock
- A circuit in a processor that generates a regular sequence of electronic pulses used to synchronize operations of the processor's components.
- The time between pulses is the cycle time and the number of pulses per second is the clock rate (or frequency).
- The execution times of instructions on a computer are usually measured by a number of clock cycles rather than seconds.
- The higher clock rate, the quicker speed of instruction processing
- The clock rate for a Pentium 4 processor is about 2.0, 2.2 GHz or higher
- 3.1.3. Memory
- Memory refer to computer components, devices and recording media that retain digital data used for computing for some interval of time.
- Computer memory includes internal and external memory
- Internal memory
- Accessible by a processor without the use of the computer input-output channels.

- Usually includes several types of storage, such as main storage, cache memory, and special registers, all of which can be directly accessed by the processor.
- Cache memory : A buffer, smaller and faster than main storage, used to hold a copy of instructions and data in main storage that are likely to be needed next by the processor and that have been obtained automatically from main storage.
- Main memory (Main Storage) : addressable storage from which instructions and other data may be loaded directly into registers for subsequent execution or processing.
-
- ROM Read-Only Memory, a class of storage media used in computers and other electronic devices. This tells the computer how to load the operating system.
- RAM Random Access Memory, computer memory that can be read from and written to in arbitrary sequence
- Storage capacity: the total amount of stored information that a storage device or medium can hold. It is expressed as a quantity of bits or bytes
- External Memory
- Holds information too large for storage in main memory.
- Information on external memory can only be accessed by the CPU if it is first transferred to main memory.
- External memory is slow and virtually unlimited in capacity.
- It retains information when the computer is switched off; used to keep a permanent copy of programs and data.
-



Floppy disk

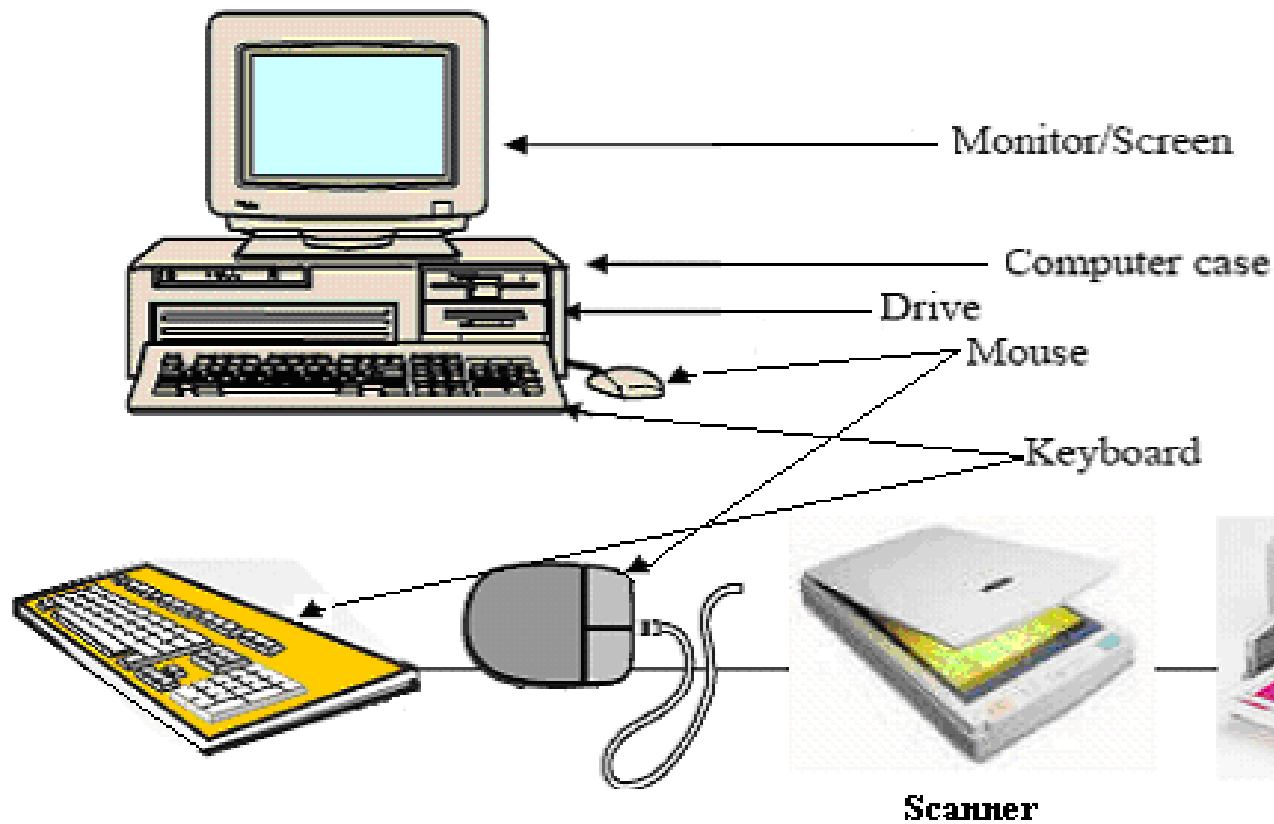


Compact disk



Compact Flash Card

- 3.1.4. Input-Output Devices



-
-
- Accessories that allow computer to perform specific tasks
- Receive information for processing
- Return the results of processing
- Store information
-
- Common input and output devices
- Speakers Mouse Scanner
- Printer Joystick CD-ROM
- Keyboard Microphone DVD
-
- Some devices are capable of both input and output
- Floppy drive Hard drive Magnetic tape units
- Monitor
- Display device that operates like a television
- Also known as CRT (cathode ray tube)
- Controlled by an output device called a *graphics card*
- Displayable area

- Measured in dots per inch, dots are often referred to as pixels (short for picture element)
- Standard resolution is 640 by 480
- Many cards support resolution of 1280 by 1024 or better
- Number of colors supported varies from 16 to billions
-
-
-
- 3.1.5. Buses
- Bus is a subsystem that transfers data or power between computer components inside a computer or between computers.
- Bus can logically connect several peripherals over the same set of wires.
- Each bus defines its set of connectors to physically plug devices, cards or cables together.
-

3.2. Computer Software

The software is divided to System Software and Application Software with each having several sub levels.

System software is the low –level software required to manage computer resources and support the production or execution of application program.

Application software is software program that perform a specific function directly for the end user.

System Software includes

- Operating Systems software
- Network Software : network management software, server software, security and encryption software, etc.
- Database management software
- Development tools and programming language software: software testing tools and testing software, program development tools, programming languages software
- Etc.

Application Software includes

- General business productivity applications : software program that perform a specific function directly for the end user, examples include : office applications, word processors, spreadsheet, project management system ,etc.
- Home use applications : software used in the home for entertainment, reference or educational purposes, examples include games, home education etc.
- Cross-industry application software : software that is designed to perform and/or manage a specific business function or process that is not unique to a

particular industry, examples include professional accounting software, human resources management, Geographic Information Systems (GIS) software, etc.

- Vertical market application software : software that perform a wide range of business functions for a specific industry such as manufacturing, retail, healthcare , engineering, restaurant, etc.
- Utilities software : a small program that performs a very specific task. Examples include : compression programs, antivirus, search engines, font, file viewers, voice recognition software, etc.

3.2.1. Data and Algorithms

3.2.2. Programs and Programming Languages

3.2.3. Classification of Computer Software : System Software and Utilities

Simple model of computation



- In order to compute, the algorithm need to transfer data in the process part from the input part – reading
- In order to reveal the results of the computation, the algorithm must transfer them from the process part to the output part - writing

- N. Wirth :
- Data structure + Algorithm = Program
- Definition:
- Algorithm is a finite list of well-defined instructions for accomplishing some task that, given an initial state, will terminate in a defined end-state

A step-by-step method for accomplishing some task

Shampoo Algorithm

Step 1: Wet hair

Step 2: Lather

Step 3: Rinse

Step 4: Repeat

Is this enough information?

Which step will be repeated? How many times do we need to repeat it?

Shampoo Algorithm (Revision #1):

Step 1: Wet hair

Step 2: Lather

Step 3: Rinse

Step 4: Repeat Steps 1-4

How many times do we repeat step 1? Infinitely? Or at most 2 times?

Keep a count of the number of times to repeat the steps

Repeat the algorithm at most 2 times

Shampoo Algorithm (Revision #2)

Step 1: Set count = 0
Step 2: Wet hair
Step 3: Lather
Step 4: Rinse
Step 5: Increment count (increase its value by 1)
Step 6: If count < 2 repeat steps 2-6
Step 7: EXIT
Will execute steps 2-6 **two** times

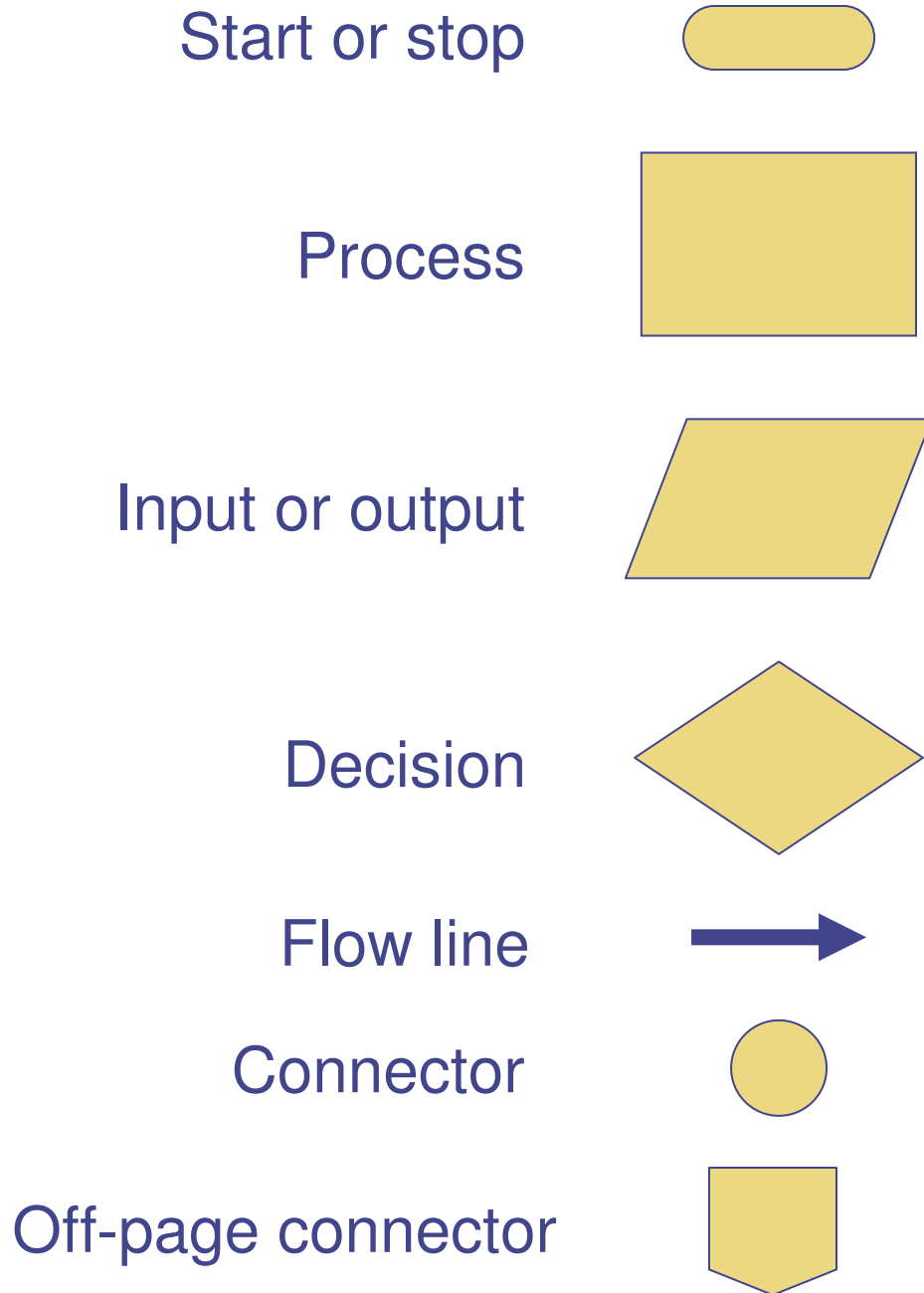
Algorithm Building Blocks

All problems can be solved by employing any one of the following building blocks or their combinations

Sequences

Conditionals

Loops



This review was essential because we we will be using these building blocks quite often today.

OK. Now on with the three building blocks of algorithms. First ..

Sequences

A sequence of instructions that are executed in the precise order they are written in:

3.3. Computer Networks

3.4. Principles of Operating Systems

Unit 1. Introduction to C

1.1. History of the C Programming Language

C is a general-purpose programming language with features economy of expression, modern flow control and data structures, and a rich set of operators. C was developed at Bell Laboratories in 1972 by Dennis Ritchie. Many of its principles and ideas were taken from the earlier language B and B's earlier ancestors BCPL and CPL. CPL (Combined Programming Language) was developed with the purpose of creating a language that was capable of both high level, machine independent programming and would still allow the programmer to control the behavior of individual bits of information. The one major drawback of CPL was that it was too large for use in many applications. In 1967, BCPL (Basic CPL) was created as a scaled down version of CPL while still retaining its basic features. In 1970, Ken Thompson, while working at Bell Labs, took this process further by developing the B language. B was a scaled down version of BCPL written specifically for use in systems programming. Finally in 1972, a co-worker of Ken Thompson, Dennis Ritchie, returned some of the generality found in BCPL to the B language in the process of developing the language we now know as C.

There are some of C's characteristics that define the language and also have lead to its popularity as a programming language. Naturally we will be studying many of these aspects throughout the course.

- Small size
- Extensive use of function calls
- Loose typing -- unlike PASCAL
- Structured language
- Low level (BitWise) programming readily available
- Pointer implementation - extensive use of pointers for memory, array, structures and functions.

C has now become a widely used professional language for various reasons.

- It has high-level constructs.
- It can handle low-level activities.
- It produces efficient programs.
- It can be compiled on a variety of computers.

Its main drawback is that it has poor error detection which can make it off putting to the beginner. However diligence in this matter can pay off handsomely since having learned the rules of C we can break them. Not many languages allow this. This if done properly and carefully leads to the power of C programming.

C's power and flexibility soon became apparent. Because of this, the Unix operating system which was originally written in assembly language, was almost immediately re-written in C (only the assembly language code needed to "bootstrap" the C code was kept). During the rest of the 1970's, C spread throughout many colleges and universities because of it's close ties to Unix and the availability of C compilers. Soon, many different organizations began using their own versions of C causing compatibility problems. In response to this in 1983, the American National Standards Institute (ANSI) formed a committee to establish a standard definition of C which became known as ANSI Standard C. Today C is in widespread use with a rich standard library of functions.

1.2. Basic Components of C Programs

1.2.1. Symbols

A C program consists of the following characters:

26 capital letter of English alphabet : A, B, C, D, X, Y, Z

26 small letter of English alphabet : a, b, c, d, x, y, z

10 digits : 0, 1, . . . 9

Math operators : + - * / = < >

Other symbols : |, \, #, %, ~,

1.2.2. Key Words

A keyword is an identifier which indicate a specific command. Keywords are also considered reserved words. You shouldn't use them for any other purpose in a C program.

The most important keywords of Turbo C are

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

1.2.3. Identifiers

Identifiers or names refer to a variety of things : functions; tag of structures, union and enumerations; member of structures or unions; enumeration constants; typedef names and objects. There are some restrictions on the names .

Names are made up of letters and digit; The first character must be a letter. The underscore “_” count as a letter; sometime it is useful for improving the readability of long variable names. For example, name *unit_price* is easier to understand than *unitprice*. However, don’t begin variable names with underscore, since library routines often use such names.

Upper and lower case are distinct, so x and X are different names. Traditional C practise use lower case for variable names, and all upper case for symbolic constant.

Only the first 31 characters are significant.

Keywords are reserved: you can’t use them as variable names.

Example : The following names are valid

i, x, b55, max_val

and the following names are invalid

12w

income tax

char

the first character is a digit

use invalid character “ ”

char is a keyword

It is wise to choose variable names that are related to the purpose of the variable, for example, *count_of_girls*, *MAXWORD*.

1.2.4. Data Types

Data is valuable resources of computers. Data may comprise numbers, text, images . . . They belong to different data types.

In programming languages, a data type is a set of values and the operations on those values.

For example, "int" type is the set of 32-bit integers within the range -2,147,483,648 to 2,147,483,647 together with the operations described in the following table.

Operations	Symbol
Opposite	-
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulus	%
Equal to	==
Greater than	>
Less than	<
...	

A data type can also be thought of as a constraint placed upon the interpretation of data in a type system in computer programming.

Common types of data in programming languages include primitive types (such as integers, floating point numbers or characters), tuples, records, algebraic data types, abstract data types, reference types, classes and function types. A data type describes representation, interpretation and structure of values manipulated by algorithms or objects stored in computer memory or other storage device. The type system uses data type information to check correctness of computer programs that access or manipulate the data.

1.2.5. Constants

In general, a constant is a specific quantity that does not or cannot change or vary. A constant's value is fixed at compile-time and cannot change during program execution. C supports three types of constants : numeric, character, string.

Numeric constants of C are usually just the written version of numbers. For example 1, 0, 56.78, 12.3e-4. We can specify our constant in octal or hexadecimal, or force them to be treated as long integers.

- Octal constants are written with a leading zero : -0.15
- Hexadecimal constants are written with a leading 0x : 0x1ae
- Long constants are written with a trailing L : 890L or 890l

Character constants are usually just the character enclosed in single quotes; 'a', 'b', 'c'. Some characters can't be represented in this way, so we use a 2 character sequence (escape sequence).

'\n'	newline
'\t'	horizontal tab
'\v'	vertical tab
'\b'	backspace
'\r'	carriage return
'\\'	backslash
'\''	single quote
'\"'	double quotes
'\0'	null (used automatically to terminate character strings)

Character constants participate in numeric operations just as any other integers (they are represented by their order in the ASCII character set), although they are most often used in comparison with other characters.

Character constants are rarely used, since string constants are more convenient. A string constant is a sequence of characters surrounded by double quotes e.g. "Brian and Dennis".

A character is a different type to a single character string. This is important.

It is helpful to assign a descriptive name to a value that does not change later in the program. That is the value associated with the name is constant rather than variable, and thus such a name is referred to as symbolic constant or simply a constant.

1.2.6. Variables

Variables are the names that refer to sections of memory into which data can be stored.

Let's imagine that memory is a series of different size boxes. The box size is memory storage area required in bytes. In order to use a box to store data, the box must be given a name, this process is known as *declaration*. It helps if you give a box a meaningful name that relates to the type of information and it is easier to find the data. The boxes must be of the correct size for the data type you are going to put into it. An integer number such as 2 requires a smaller box than a floating point number as 123e12.

Data is placed into a box by assigning the data to the box. By using the name of the box you can retrieve the box contents, some kind of data.

Variable named by an identifier. The conventions of identifiers were shown in 1.2.3.

Names should be meaningful or descriptive, for example, `studentAge` or `student_age` is more meaningful than `age`, and much more meaningful than a single letter such as `a`.

1.2.7. Operators

Programming languages have a set of operators that perform arithmetical operations, and others such as Boolean operations on truth values, and string operators manipulating strings of text. Computers are mathematical devices, but compilers and interpreters require a full syntactic theory of all operation in order to parse formulae involving any combination correctly.

1.2.8. Expressions

An expression in a programming language is a combination of values, functions, tokens, atoms and procedures, interpreted according to the particular rules of precedence and association for a particular programming language, which computes and returns another value.

C language expressions are arranged in the following groups based on the operators they contain and how you use them:

- Primary expression
- Unary expression
- Binary expression
- Conditional expression
- Assignment expression

- Comma expression
- lvalue
- Constant

Expressions are used as

- Right hands of assignment statements
- Actual parameters of functions
- Conditions of if statements
- Indexes of while statements
- Operands of other expressions

1.2.9. Functions

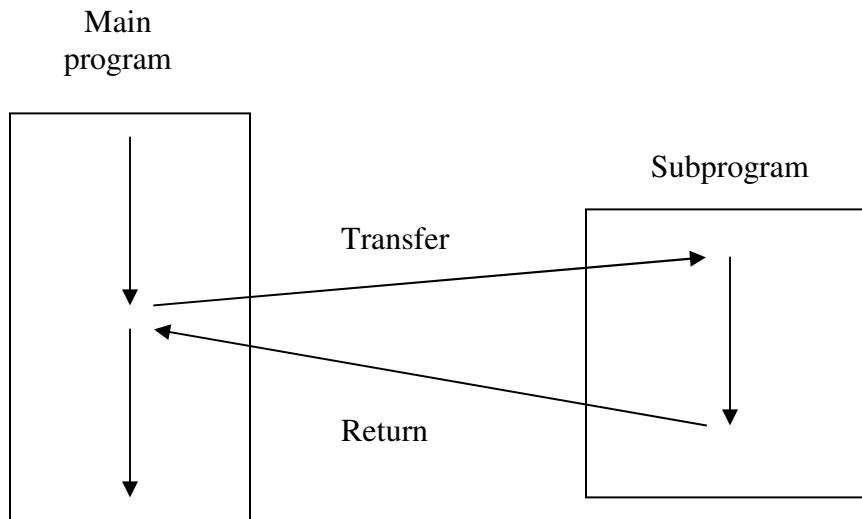
A subprogram (also known as a procedure or subroutine) is nothing more than a collection of instructions forming a program unit written independently of the main program yet associated with it through a transfer/return process. Control is passed to the subprogram at the time its services are required, and then control is returned to the main program after the subprogram has finished.

The syntax used to represent the request of subprogram varies among the different language. The techniques used to describe a subprogram also varies from language to language. Many systems allow such program units to be written in languages other than that of the main program.

In most procedural programming languages, a subprogram is implemented as though it were completely separate entity with its own data and algorithm so that an item of data in either the main program or the subprogram is not automatically accessible from within the other. With this arrangement, any transfer of data between the two program parts must be specified by the programmer. This is usually done by listing the items called *parameters* to be transferred in the same syntactic structure used to request the subprogram's execution.

In its most general form, the transfer of data through parameters takes place in two directions. When execution of the subprogram is requested, the parameters are effectively transferred to the subprogram, the subprogram is executed, the (possibly modified) parameters are transferred back to the main program., and the main program continues. In other cases, the transfers can take place in only one direction: either to the subprogram before it is executed or to the main program after the main program. Languages that provide more than one of these transfer techniques also provide a mean by which the programmer can specify which option is desired.

The names used for the parameters within the subprogram can be thought of as merely standing in for the actual data values that are supplied when the subprogram is requested. As a result, you often hear them called formal parameters, whereas the data values supplied from the main program are referred to actual parameters.



C only accept one kind of subprogram, function. A function is a sub program in which input values are transferred through a parameter list. However, information is returned from a function to the main program in the form of the “value of the function”. That is the value returned by a function is associated with the name of the function in a manner similar to the association between a value and a variable name. The difference is that the value associated with a function name is computed (according to the function’s definition) each time it is required, whereas when a variable ‘s value is required, it is merely retrieve from memory.

C also provide a rich collection of built-in functions. There are more than twenty functions declared in <math.h>. Here are some of the more frequently used.

Name	Description	Math Symbols	Example
sqrt(x)	square root	\sqrt{x}	sqrt(16.0) is 4.0
pow(x,y)	compute a value taken to an exponent, x^y	x^y	pow(2,3) is 8
exp(x)	exponential function, computes e^x	e^x	exp(1.0) is 2.718282
log(x)	natural logarithm	$\ln x$	log(2.718282) is 1.0

$\log_{10}(x)$	base-10 logarithm	$\log x$	$\log_{10}(100)$ is 2
$\sin(x)$	sine	$\sin x$	$\sin(0.0)$ is 0.0
$\cos(x)$	cosine	$\cos x$	$\cos(0.0)$ is 1.0
$\tan(x)$	tangent	$\tan x$	$\tan(0.0)$ is 0.0
$\text{ceil}(x)$	smallest integer not less than parameter	$\lceil x \rceil$	$\text{ceil}(2.5)$ is 3 $\text{ceil}(-2.5)$ is -2
$\text{floor}(x)$	largest integer not greater than parameter	$\lfloor x \rfloor$	$\text{floor}(2.5)$ is 2 $\text{floor}(-2.5)$ is -3

Library

The library is not part of the C language proper, but an environment that support C will provide the function declarations and type and macro definitions of this library. The functions, types and macro of the library are declared in headers.

C header files have extensions .h. Header files should not contain any source code. They are used purely to store function prototypes, common #define constants, and any other information you wish to export from the C file that the header file belongs to.

A header can be accessed by

```
#include <header>
```

Here are some headers of Turbo C library

stdio.h Provides functions for performing input and output.

stdlib.h Defines several general operation functions and macros.

conio.h Declares several useful library functions for performing "console input and output" from a program.

math.h Defines several mathematic functions.

string.h Provides many functions useful for manipulating strings (character arrays).

io.h Defines the file handle and low-level input and output functions

graphics.h Includes graphics functions

1.2.10. Statements

A statement specifies one or more action to be perform during the execution of a program.

C requires a semicolon at the end of every statement.

1.2.11. Comments

Comments are marked by symbol “/*” and “*/”. C also use // to mark the start of a comment and the end of a line to indicate the end of a comment.

Example :

The Hello program written using the first commenting style of C

```
/* A simple program to demonstrate
```

```
   C style comments
```

The following line is essential
in the C version of the hello HUT program

```
*/  
#include <stdio.h>  
main()  
{  
    printf /* just print */ ("Hello HUT\n");  
  
}
```

The Hello program written using the second commenting style of C

```
// A simple program to demonstrate  
// C style comments  
//  
// The following line is essential  
// in the C version of the hello HUT program  
#include <stdio.h>  
main()  
{  
    printf("Hello HUT\n"); //print the string and then go to a new line  
  
}
```

By the first way, a program may have a multi-line comments and comments in the middle of a line of code. However, you shouldn't mix the two style in the same program.

Unit 2

Data types and Expressions

2.1. Standard Data Types

The *type* of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted. Standard data types in C are listed in the following table:

Variable Type	Keyword	Range	Storage in Bytes
Character	char	-127 to 127	1
Unsigned character	unsigned char	0 to 255	1
(Signed) integer	int	-32,768 to 32,767	2
Unsigned integer	unsigned int	0 to 65,535	2

Short integer	short	-32,768 to 32,767	2
Unsigned short integer	unsigned short	0 to 65,535	2
Long integer	long	-2,147,483,648 to 2,147,483,647	4
Unsigned long	integer unsigned long	0 to 4,294,967,295	4
Single precision floating point	float	1.2E-38 to 3.4E38, approx. range precision = 7 digits.	4
Double precision floating point	double	2.2E-308 to 1.8E308, approx. range precision = 19 digits.	8

Table 2.1. Table of Variable types

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

2.1.1. Declaration and Usage of Variables and Constants

Variables

A variable is an object of a specified type whose value can be changed. In programming languages, a variable is allocated a storage location that can contain data that can be modified during program execution. Each variable has a name that uniquely identifies it within its level of scope.

In C, a variable must be declared before use, although certain declarations can be made implicitly by content. Variables can be declared at the start of any block of code, but most are found at the start of each function. Most local variables are created when the function is called, and are destroyed on return from that function.

A declaration begins with the type, followed by the name of one or more variables. Syntax of declare statement is described as:

data type list of variables;

A list of variables includes one or many variable names separated by commas.

Example:

Single declarations

```
int age;           //integer variable
float amountOfMoney; //float variable
char initial;     // character variable
```

Multiple declarations:

```
int age, houseNumber, quantity;
float distance, rateOfDiscount;
char firstInitial, secondInitial;
```

Variables can also be initialized when they are declared, this is done by adding an equals sign and the required value after the declaration.

Example:

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

Constants

A constant is an object whose value cannot be changed. There are two methods to define a constant in C:

- By #define statement. Syntax of that statement is:

#define *constant_name* *value*

Example

```
#define MAX_SALARY_LEVEL 15 //An integer constant
#define DEP_NAME "Computer Science"
// A string constant
```

- By using const keyword

const *data_type* *variable_name* = *value*;

Example

```
const double e = 2.71828182845905;
```

2.1.2. Functions printf, scanf

Usually i/o, input and output, form the most important part of any program. To do anything useful your program needs to be able to accept input data and report back your results. In C, the standard library (stdio.h) provides routines for input and output. The standard library has functions for i/o that handle input, output, and character and string manipulation. In this part, all the input functions described read from standard input and all the output functions described write to standard output. Standard input is usually means input using the keyboard. Standard output is usually means output onto the monitor.

- Input by using the scanf() function
- Output by using the printf() function

To use printf and scanf functions, it is required to declare the header <stdio.h>

The printf() function

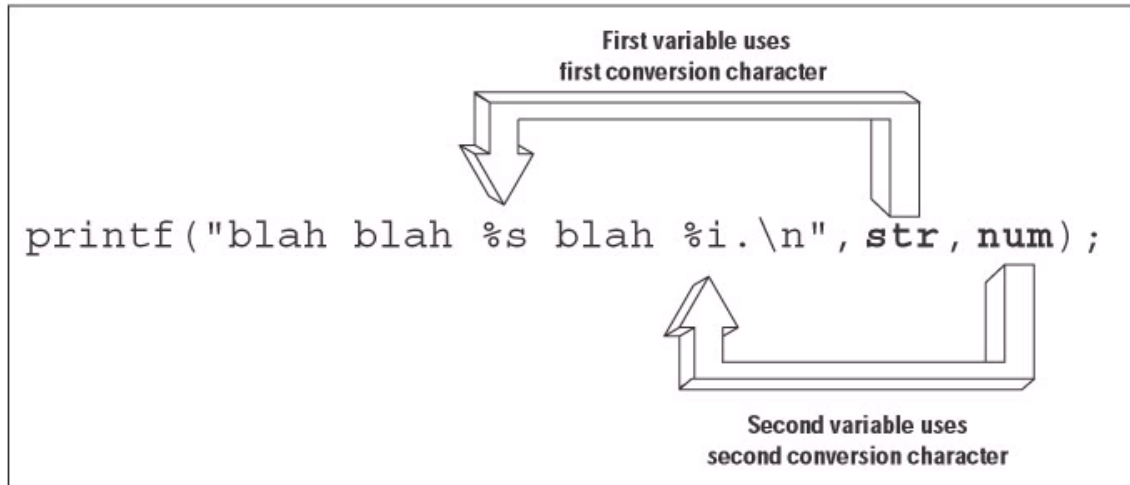
The standard library function printf is used for formatted output. It makes the user input a string and an optional list of variables or strings to output. The variables and strings are output according to the specifications in the printf() function. Here is the general syntax of printf.

printf("[string]"[,list of arguments]);

The **list of arguments** allow expressions, separated by commas.

The **string** is all-important because it specifies the type of each variable in the list and how you want it printed. The string is usually called the *control string* or the *format string*. The way that this works is that printf scans the string from left to right and prints on the screen any characters it encounters - except when it reaches a % character. The %

character is a signal that what follows it is a specification for how the next variable in the list of variables should be printed. `printf` uses this information to convert and format the value that was passed to the function by the variable and then moves on to process the rest of the control string and anymore variables it might specify.



For example:

```
printf("Hello World");
```

only has a control string and, as this contains no `%` characters it results in Hello World being displayed and doesn't need to display any variable values. The specifier `%d` means convert the next value to a signed decimal integer and so:

```
printf("Total = %d",total);
```

will print Total = and then the value passed by `>total` as a decimal integer.

The `%d` isn't just a format specifier, it is a conversion specifier. It indicates the data type of the variable to be printed and how that data type should be converted to the characters that appear on the screen. That is `%d` says that the next value to be printed is a signed integer value (i.e. a value that would be stored in a standard `int` variable) and this should be converted into a sequence of characters (i.e. digits) representing the value in decimal. If by some accident the variable that you are trying to display happens to be a float or a double then you will still see a value displayed - but it will not correspond to the actual value of the float or double.

The reason for this is twofold.

The first difference is that an `int` uses two bytes to store its value, while a float uses four and a double uses eight. If you try to display a float or a double using `%d` then only the first two bytes of the value are actually used.

with the wrong type of variable then you will see some strange things on the screen and the error often propagates to other items in the printf list.

You can also add an 'l' in front of a specifier to mean a long form of the variable type and h to indicate a short form (long and short will be covered later in this course). For example, %ld means a long integer variable (usually four bytes) and %hd means short int. Notice that there is no distinction between a four-byte float and an eight-byte double. The reason is that a float is automatically converted to a double precision value when passed to printf

The % Format Specifiers

The % specifiers that you can use are:

	Usual variable type	Display
%c	char	single character
%d (%i)	int	signed integer
%e (%E)	float or double	exponential format
%f	float or double	signed decimal
%g (%G)	float or double	use %f or %e as required
%o	int	unsigned octal value
%s	array of char	sequence of characters
%u	int	unsigned decimal
%x (%X)	int	unsigned hex value

Formatting Your Output

The type conversion specifier only does what you ask of it - it convert a given bit pattern into a sequence of characters that a human can read. If you want to format the characters then you need to know a little more about the printf function's control string.

Each specifier can be preceded by a modifier which determines how the value will be printed. The most general modifier is of the form:

flag width.precision

The flag can be any of

flag	meaning
-	left justify
+	always display sign
space	display space if there is no sign
0	pad with leading zeros
#	use alternate form of specifier

The width specifies the number of characters used in total to display the value and precision indicates the number of characters used after the decimal point.

For example,

`%10.3f` will display the float using ten characters with three digits after the decimal point. Notice that the ten characters includes the decimal point, and a - sign if there is one. If the value needs more space than the width specifies then the additional space is used - width specifies the smallest space that will be used to display the value. (This is quiet reassuring, you won't be the first programmer whose program takes hours to run but the output results can't be viewed because the wrong format width has been specified!)

`%-10d` will display an int left justified in a ten character space.

The specifier `%+5d` will display an int using the next five character locations and will add a + or - sign to the value.

The only complexity is the use of the # modifier. What this does depends on which type of format it is used with:

`%#o` adds a leading 0 to the octal value

`%#x` adds a leading 0x to the hex value

`%#f` or

`%#e` ensures decimal point is printed

`%#g` displays trailing zeros

Strings will be discussed later but for now remember: if you print a string using the `%s` specifier then all of the characters stored in the array up to the first null will be printed. If you use a width specifier then the string will be right justified within the space. If you include a precision specifier then only that number of characters will be printed.

For example:

```
printf("%s,Hello")
```

will print Hello,

```
printf("%25s ,Hello")
```

will print 25 characters with Hello right justified and

```
printf("%25.3s,Hello")
```

will print Hello right justified in a group of 25 spaces.

Also notice that it is fine to pass a constant value to `printf` as in `printf("%s,Hello")`.

Finally there are the control codes:

\b	backspace
\f	formfeed
\n	new line
\r	carriage return
\t	horizontal tab
\'	single quote
\0	null

If you include any of these in the control string then the corresponding ASCII control code is sent to the screen, or output device, which should produce the effect listed. In most cases you only need to remember \n for new line.

The scanf() function

The scanf function works in much the same way as the printf. That is it has the general form:

scanf(“control string”,variable,variable,...)

In this case the control string specifies how strings of characters, usually typed on the keyboard, should be converted into values and stored in the listed variables. However there are a number of important differences as well as similarities between scanf and printf.

The most obvious is that scanf has to change the values stored in the parts of computers memory that is associated with parameters (variables).

To understand this fully you will have to wait until we have covered functions in more detail. But, just for now, bare with us when we say to do this the scanf function has to have the addresses of the variables rather than just their values. This means that simple variables have to be passed with a preceding >&.

The second difference is that the control string has some extra items to cope with the problems of reading data in. However, all of the conversion specifiers listed in connection with printf can be used with scanf.

The rule is that scanf processes the control string from left to right and each time it reaches a specifier it tries to interpret what has been typed as a value. If you input multiple values then these are assumed to be separated by white space - i.e. spaces, newline or tabs. This means you can type:

3 4 5
or

3
4
5

and it doesn't matter how many spaces are included between items. For example:

```
scanf("%d %d",&i,&j);
```

will read in two integer values into i and j. The integer values can be typed on the same line or on different lines as long as there is at least one white space character between them.

The only exception to this rule is the %c specifier which always reads in the next character typed no matter what it is. You can also use a width modifier in scanf. In this case its effect is to limit the number of characters accepted to the width.

For example:

```
scanf("%10d",&i)
```

would use at most the first ten digits typed as the new value for i.

There is one main problem with scanf function which can make it unreliable in certain cases. The reason being is that scanf tends to ignore white spaces, i.e. the space character. If you require your input to contain spaces this can cause a problem.

As the previous section has said already, scanf will skip over white space such as blanks, tabs and new lines in the input stream. The exception is when trying to read single characters with the conversion specifiers %c. In this case, white space is read in. So, it is more difficult to use scanf for single characters. An alternate technique, using getchar, will be described later.

2.1.3. Other Input and Output Functions

getchar

getchar reads a single character from standard input. An example is:

```
int getchar();
```

It requires the user to press enter after entering

putchar

putchar writes a single character to standard output. An example is:

```
int putchar(int value);
```

Here is a sample program using both both `getchar` and `putchar`

```
#include <stdio.h>

int main(void)
{
    int c;

    while ((c = getchar()) != EOF)
    {
        putchar(c);
    }

    return 0;
}

gets
```

`gets` reads a line of input into a character array. The syntax is:

```
gets(name of string)
```

puts

`puts` writes a line of output to standard output. The syntax is:

```
int puts(name of string);
```

It terminates the line with a new line, '\n'. It will return EOF if an error occurred. It will return a positive number on success.

Here is a long example of strings and their definitions. The example will discuss the use of strings and how they should be declared.

```
#include <stdio.h>

int main()
{
    /* character arrays are sized properly for data */

    char name[60];
    char address[120];
```

```
char city[60];
char state[20];
char zip[15];
```

```
/* For responses that can include white space such as blanks
use gets. For single word responses scanf is easier */
```

```
printf("Enter your name: ");
```

```
if ((pt = gets(name)) == NULL)
{
    printf("Error on read\n");
    return(1);
}
```

```
printf("Enter your address: ");
```

```
if ((pt = gets(address)) == NULL)
{
    printf("Error on read\n");
    return(1);
}
```

```
printf("Enter your city: ");
scanf("%s",&city);
```

```
printf("Enter your state: ");
scanf("%s",&state);
```

```
printf("Enter your zip: ");
scanf("%s",&zip);
```

```
/* Output user information */
printf("%s\n",name);
printf("%s\n",address);
printf("%s, %s %s\n",city,state,zip);
```

```
return 0;
```

2.2. Expressions

2.2.1. Operators

C contains the following operator groups.

- Arithmetic

- Assignment
- Logical/relational
- Bitwise
- Odds and ends!

2.2.2. Arithmetic Operators

The arithmetic operators are +, -, /, * and the modulus operator %. Integer division truncates any fractional part. The expression $x\%y$ produces the remainder when x is divided by y , and thus is zero when y divide x exactly. The % operator cannot be applied to a float or double.

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

2.2.3. Assignment Operators

These all perform an arithmetic operation on the lvalue and assign the result to the lvalue. So what does this mean in English? Here is an example:

counter = counter + 1;

can be reduced to

counter += 1;

Here is the full set.

=	
*=	Multiply
/=	Divide.
%=	Modulus.
+=	add.
-=	Subtract.
<<=	left shift.
>>=	Right shift.
&=	Bitwise AND.
^=	bitwise exclusive OR (XOR).
=	bitwise inclusive OR.

if expr_1 and expr_2 are expressions then

$\text{expr}_1 \text{ op } \text{expr}_2$

is equivalent to

$\text{expr}_1 = \text{expr}_1 \text{ op } \text{expr}_2$

2.2.4. Logical and Relational Operators

== Equal to

!= Not equal to

>

<

>=

<=

&& Logical AND

|| Logical OR

! Logical NOT

2.2.5. Bitwise Operators

& AND (Binary operator)
| inclusive OR
^ exclusive OR
<< shift left. (C ++ use of <<)
>> shift right. (C ++ use of >>)
~ one's complement

2.2.6. Increment and Decrement Operators

Incrementing, decrementing and doing calculations on a variable is a very common programming task and C has quicker ways of writing the code. The code is rather cryptic in appearance.

The increment operator ++ adds 1 to its operand while the decrement operator -- subtract 1. We have frequently used ++ to increment variables, as in

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

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

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

sets x to 5 but

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

sets x to 6. In both cases, n becomes 6.

Note: The increment and decrement operator can only be applied to variables; an expression like (i + j)++ is illegal.

2.2.7. Type Conversions

When an operator has operands of different types, they are converted to a common type according to a small number of rules. In general, the only automatic conversion are those that convert a narrower operand into a wider one without losing information, such as converting an integer into floating point.

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

If either operand is long double, convert the other to long double.

Otherwise, if either operand is double, convert the other to double.

Otherwise if either operand is float, convert the other to float.

Otherwise convert char and short to int.

Then if either operand is long, convert the other to long.

A char is just a small integer, so chars may be freely used in arithmetic expressions. For example,

2.2.8. Precedence of Operators

Operators listed by type.

All operators on the same line have the same precedence. The first line has the highest precedence.

()	[]	->	.	++	-- (postfix)				
!	~	++	-- (prefix)	+	-	*	&	sizeof	
*	/	%							
+	-								
<<	>>								
<	<=	>=	>						
==	!=								
&									
^									
&&									
?:									
=	+=	-=	*=	/=	%=	&=	^=	=	<<= >>=

2.3. Control Flow

The control flow of a language specifies the order in which operations are performed. Each program includes many statements. Statements are processed one after another in sequence, except where such control statements result in jumps.

2.3.1. Statements and Blocks

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

```
x=0;
i++;
printf(. . .);
```

In the C language, the semicolon is a statement terminator.

A block also called a *compound statement*, or compound statement, lets you group any number of data definitions, declarations, and statements into one statement. All definitions, declarations, and statements enclosed within a single set of braces are treated as a single statement. You can use a block wherever a single statement is allowed.

In blocks, declarations and definitions can appear anywhere, mixed in with other code.

Note that there is no semicolon after the right brace that ends a block.

Example

```
{ int i = 0;    /* Declarations */
```

```
static long a;
extern long max;

++a;      /* Statements */
if( a >= max)
{ ... } /* A nested block */
...
}
```

An *expression statement* is an expression followed by a semicolon. The syntax is:

```
[expression] ;
```

Example

```
y = x;    // Assignment
```

The expression—an assignment or function call, for example—is evaluated for its side effects. The type and value of the expression are discarded.

A statement consisting only of a semicolon is called an *empty statement*, and does not perform any operation. For example:

```
for ( i = 0; str[i] != '\0'; ++i )
;           // Empty statement
```

2.3.2. If, If else

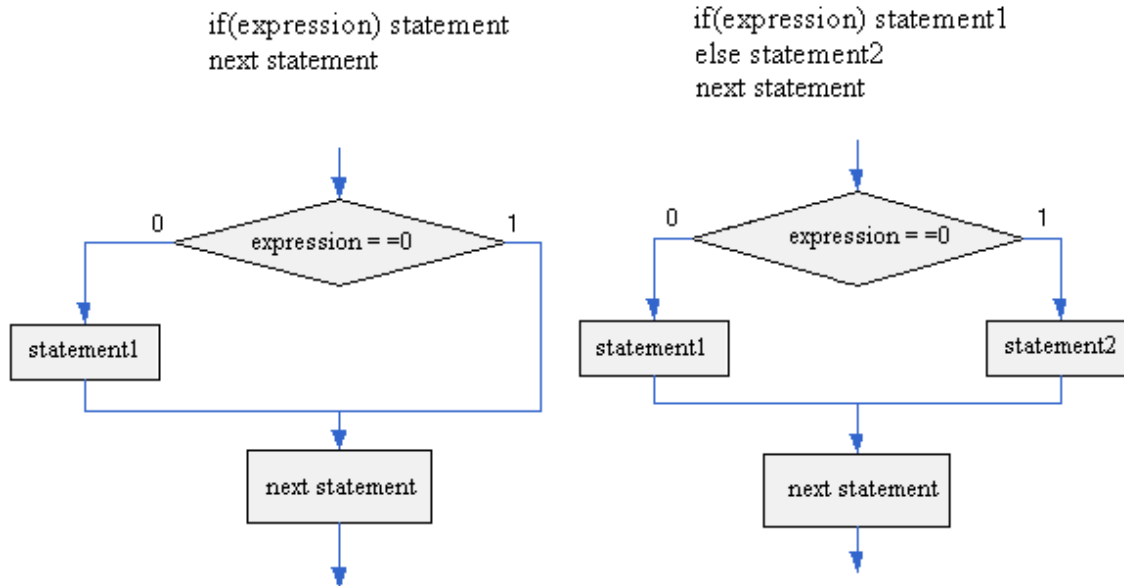
The if statement has two forms:

```
if(expression) statement
```

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

In the first form, if (and only if) the expression is non-zero, the statement is executed. If the expression is zero, the statement is ignored. Remember that the statement can be compound; that is the way to put several statements under the control of a single if.

The second form is like the first except that if the statement shown as statement1 is selected then statement2 will not be, and vice versa.



Either form is considered to be a single statement in the syntax of C, so the following is completely legal.

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

The first if (expression) is followed by a properly formed, complete if statement. Since that is legally a statement, the first if can be considered to read

```
if(expression) statement
```

and is therefore itself properly formed. The argument can be extended as far as you like, but it's a bad habit to get into. It is better style to make the statement compound even if it isn't necessary. That makes it a lot easier to add extra statements if they are needed and generally improves readability.

The form involving else works the same way, so we can also write this.

```
if(expression)
    if(expression)
        statement
    else
        statement
```

this is now ambiguous. It is not clear, except as indicated by the indentation, which of the ifs is responsible for the else. If we follow the rules that the previous example suggests, then the second if is followed by a statement, and is therefore itself a statement, so the else belongs to the first if.

That is not the way that C views it. The rule is that an else belongs to the first if above that hasn't already got an else. In the example we're discussing, the else goes with the second if.

To prevent any unwanted association between an else and an if just above it, the if can be hidden away by using a compound statement, here it is.

```
if(expression){
    if(expression)
        statement
}else
    statement
```

Putting in all the compound statement brackets, it becomes this:

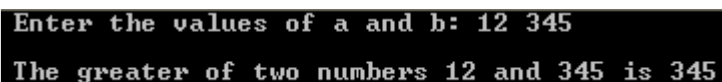
```
if(expression){
    if(expression){
        statement
    }
}else{
    statement
}
```

If you happen not to like the placing of the brackets, it is up to you to put them where you think they look better; just be consistent about it. You probably need to know that this a subject on which feelings run deep.

Example

```
#include <conio.h>
#include <stdio.h>
void main()
{
    // variable declaration
    float a, b;
    float max;
    printf(" Enter the values of a and b: ");
    scanf("%f %f",&a,&b);
    if(a<b) //Assign the greater of x and y to the variable max
        max = b;
    else
        max = a;
    printf("\n The greater of two numbers %.0f and %.0f is %.0f ",a,b,max);
    getch();
}
```

Here is the result of the program

A screenshot of a terminal window showing the output of the C program. The first line is "Enter the values of a and b: 12 345" and the second line is "The greater of two numbers 12 and 345 is 345".

```
Enter the values of a and b: 12 345
The greater of two numbers 12 and 345 is 345
```

2.3.3. Switch

It is used to select one of a number of alternative actions depending on the value of an expression, and nearly always makes use of another of the lesser statements: the break. It looks like this.

```
switch (expression){  
case const1:  statements  
case const2:  statements  
...  
default:     statements  
}
```

The expression is evaluated and its value is compared with all of the const1 etc. expressions, which must all evaluate to different constant values (strictly they are integral constant expressions). If any of them has the same value as the expression then the statement following the case label is selected for execution. If the default is present, it will be selected when there is no matching value found. If there is no default and no matching value, the entire switch statement will do nothing and execution will continue at the following statement.

One curious feature is that the cases are not exclusive, as this example shows.

```
#include <stdio.h>  
#include <stdlib.h>  
  
main(){  
    int i;  
    for(i = 0; i <= 10; i++){  
        switch(i){  
            case 1:  
            case 2:  
                printf("1 or 2\n");  
            case 7:  
                printf("7\n");  
            default:  
                printf("default\n");  
        }  
    }  
    exit(EXIT_SUCCESS);  
}
```

Example 3.5

The loop cycles with i having values 0–10. A value of 1 or 2 will cause the printing of the message 1 or 2 by selecting the first of the printf statements. What you might not expect is the way that the remaining messages would also appear! It's because the switch only selects one entry point to the body of the statement; after starting at a given point all of the following statements are also executed. The case and default labels simply allow you

to indicate which of the statements is to be selected. When *i* has the value of 7, only the last two messages will be printed. Any value other than 1, 2, or 7 will find only the last message.

The labels can occur in any order, but no two values may be the same and you are allowed either one or no default (which doesn't have to be the last label). Several labels can be put in front of one statement and several statements can be put after one label.

The expression controlling the switch can be of any of the integral types. Old C used to insist on only *int* here, and some compilers would forcibly truncate longer types, giving rise on rare occasions to some very obscure bugs.

3.2.5.1. The major restriction

The biggest problem with the switch statement is that it doesn't allow you to select mutually exclusive courses of action; once the body of the statement has been entered any subsequent statements within the body will all be executed. What is needed is the *break* statement. Here is the previous example, but amended to make sure that the messages printed come out in a more sensible order. The *break* statements cause execution to leave the switch statement immediately and prevent any further statements in the body of the switch from being executed.

```
#include <stdio.h>
#include <stdlib.h>
main(){
    int i;
    for(i = 0; i <= 10; i++){
        switch(i){
            case 1:
            case 2:
                printf("1 or 2\n");
                break;
            case 7:
                printf("7\n");
                break;
            default:
                printf("default\n");
        }
    }
    exit(EXIT_SUCCESS);
}
```

Example 3.6

The *break* has further uses. Its own section follows soon.

2.3.4. Loops : While and Do While, For

The while statement

The while is simple:

```
while(expression)
    statement
```

The statement is only executed if the expression is non-zero. After every execution of the statement, the expression is evaluated again and the process repeats if it is non-zero. What could be plainer than that? The only point to watch out for is that the statement may never be executed, and that if nothing in the statement affects the value of the expression then the while will either do nothing or loop for ever, depending on the initial value of the expression.

Example

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

    /* initialise */
    i = 0;
    /* check */
    while(i <= 10){
        printf("%d\n", i);
        /* update */
        i++;
    }
    exit(EXIT_SUCCESS);
}
```

The do statement

It is occasionally desirable to guarantee at least one execution of the statement following the while, so an alternative form exists known as the do statement. It looks like this:

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

and you should pay close attention to that semicolon—it is not optional! The effect is that the statement part is executed before the controlling expression is evaluated, so this guarantees at least one trip around the loop. It was an unfortunate decision to use the keyword while for both purposes, but it doesn't seem to cause too many problems in practice.

The for statement

A very common feature in programs is loops that are controlled by variables used as a counter. The counter doesn't always have to count consecutive values, but the usual arrangement is for it to be initialized outside the loop, checked every time around the loop to see when to finish and updated each time around the loop. There are three important places, then, where the loop control is concentrated: initialize, check and update. This example shows them.

As you will have noticed, the initialization and check parts of the loop are close together and their location is obvious because of the presence of the while keyword. What is harder to spot is the place where the update occurs, especially if the value of the controlling variable is used within the loop. In that case, which is by far the most common, the update has to be at the very end of the loop: far away from the initialize and check. Readability suffers because it is hard to work out how the loop is going to perform unless you read the whole body of the loop carefully. What is needed is some way of bringing the initialize, check and update parts into one place so that they can be read quickly and conveniently. That is exactly what the for statement is designed to do. Here it is.

for (initialize; check; update) statement

The initialize part is an expression; nearly always an assignment expression which is used to initialize the control variable. After the initialization, the check expression is evaluated: if it is non-zero, the statement is executed, followed by evaluation of the update expression which generally increments the control variable, then the sequence restarts at the check. The loop terminates as soon as the check evaluates to zero.

There are two important things to realize about that last description: one, that each of the three parts of the for statement between the parentheses are just expressions; two, that the description has carefully explained what they are intended to be used for without proscribing alternative uses—that was done deliberately. You can use the expressions to do whatever you like, but at the expense of readability if they aren't used for their intended purpose.

Here is a program that does the same thing twice, the first time using a while loop, the second time with a for. The use of the increment operator is exactly the sort of use that you will see in everyday practice.

Example

```
#include <stdio.h>
#include <stdlib.h>
void main(){
    int i;
    /* the same done using ``for" */
```

```
    for(i = 0; i <= 10; i++){  
        printf("%d\n", i);  
    }  
    exit(EXIT_SUCCESS);  
}
```

There isn't any difference between the two, except that in this case the for loop is more convenient and maintainable than the while statement. You should always use the for when it's appropriate; when a loop is being controlled by some sort of counter. The while is more at home when an indeterminate number of cycles of the loop are part of the problem. As always, it needs a degree of judgement on behalf of the author of the program; an understanding of form, style, elegance and the poetry of a well written program. There is no evidence that the software business suffers from a surfeit of those qualities, so feel free to exercise them if you are able.

Any of the initialize, check and update expressions in the for statement can be omitted, although the semicolons must stay. This can happen if the counter is already initialized, or gets updated in the body of the loop. If the check expression is omitted, it is assumed to result in a 'true' value and the loop never terminates. A common way of writing never-ending loops is either
for(;;)

or
while(1)

and both can be seen in existing programs.

2.3.5. Break and Continue

The control of flow statements that we've just seen are quite adequate to write programs of any degree of complexity. They lie at the core of C and even a quick reading of everyday C programs will illustrate their importance, both in the provision of essential functionality and in the structure that they emphasize. The remaining statements are used to give programmers finer control or to make it easier to deal with exceptional conditions. Only the switch statement is enough of a heavyweight to need no justification for its use; yes, it can be replaced with lots of ifs, but it adds a lot of readability. The others, break, continue and goto, should be treated like the spices in a delicate sauce. Used carefully they can turn something commonplace into a treat, but a heavy hand will drown the flavour of everything else.

This is not an essential part of C. You could do without it, but the language would have become significantly less expressive and pleasant to use.

The break statement

This is a simple statement. It only makes sense if it occurs in the body of a switch, do, while or for statement. When it is executed the control of flow jumps to the statement

immediately following the body of the statement containing the break. Its use is widespread in switch statements, where it is more or less essential to get the control that most people want.

The use of the break within loops is of dubious legitimacy. It has its moments, but is really only justifiable when exceptional circumstances have happened and the loop has to be abandoned. It would be nice if more than one loop could be abandoned with a single break but that isn't how it works. Here is an example.

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

    for(i = 0; i < 10000; i++){
        if(getchar() == 's')
            break;
        printf("%d\n", i);
    }
    exit(EXIT_SUCCESS);
}
```

Example 3.7

It reads a single character from the program's input before printing the next in a sequence of numbers. If an 's' is typed, the break causes an exit from the loop.

If you want to exit from more than one level of loop, the break is the wrong thing to use. The goto is the only easy way, but since it can't be mentioned in polite company, we'll leave it till last.

The continue statement

This statement has only a limited number of uses. The rules for its use are the same as for break, with the exception that it doesn't apply to switch statements. Executing a continue starts the next iteration of the smallest enclosing do, while or for statement immediately. The use of continue is largely restricted to the top of loops, where a decision has to be made whether or not to execute the rest of the body of the loop. In this example it ensures that division by zero (which gives undefined behaviour) doesn't happen.

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

    for(i = -10; i < 10; i++){
        if(i == 0)
            continue;
        printf("%f\n", 15.0/i);
    }
}
```



```
    /*  
    * Lots of other statements .....  
    */  
}  
exit(EXIT_SUCCESS);  
}
```

Example 3.7

You could take a puritanical stance and argue that, instead of a conditional `continue`, the body of the loop should be made conditional instead—but you wouldn't have many supporters. Most C programmers would rather have the `continue` than the extra level of indentation, particularly if the body of the loop is large.

Of course the `continue` can be used in other parts of a loop, too, where it may occasionally help to simplify the logic of the code and improve readability. It deserves to be used sparingly.

Do remember that `continue` has no special meaning to a `switch` statement, where `break` does have. Inside a `switch`, `continue` is only valid if there is a loop that encloses the `switch`, in which case the next iteration of the loop will be started.

There is an important difference between loops written with `while` and `for`. In a `while`, a `continue` will go immediately to the test of the controlling expression. The same thing in a `for` will do two things: first the update expression is evaluated, then the controlling expression is evaluated.

2.3.6. *Goto and Labels*

Everybody knows that the `goto` statement is a 'bad thing'. Used without care it is a great way of making programs hard to follow and of obscuring any structure in their flow. Dijkstra wrote a famous paper in 1968 called 'Goto Statement Considered Harmful', which everybody refers to and almost nobody has read.

What's especially annoying is that there are times when it is the most appropriate thing to use in the circumstances! In C, it is used to escape from multiple nested loops, or to go to an error handling exit at the end of a function. You will need a label when you use a `goto`; this example shows both.

```
goto L1;  
/* whatever you like here */  
L1: /* anything else */
```

A label is an identifier followed by a colon. Labels have their own 'name space' so they can't clash with the names of variables or functions. The name space only exists for the

function containing the label, so label names can be re-used in different functions. The label can be used before it is declared, too, simply by mentioning it in a goto statement.

Labels must be part of a full statement, even if it's an empty one. This usually only matters when you're trying to put a label at the end of a compound statement—like this.

```
label_at_end: ; /* empty statement */  
}
```

The goto works in an obvious way, jumping to the labelled statements. Because the name of the label is only visible inside its own function, you can't jump from one function to another one.

It's hard to give rigid rules about the use of gotos but, as with the do, continue and the break (except in switch statements), over-use should be avoided. Think carefully every time you feel like using one, and convince yourself that the structure of the program demands it. More than one goto every 3–5 functions is a symptom that should be viewed with deep suspicion.

Unit 3. Pointers and Arrays

From the beginning, we only show how to access or change directly the values of variables through their names. However, the C language provides the developers an effective method to access variables indirectly which is pointer.

A pointer is a variable that contains the address of a variable. Pointers are much used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways. Pointers and arrays are closely related; this Unit also explores this relationship and shows how to exploit it.

Pointers and Addresses

Let us begin with a simplified picture of how memory is organized. A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups. One common situation is that any byte can be a char, a pair of one-byte cells can be treated as a short integer, and four adjacent bytes form a long.

Any variable in a program is stored at a specific area in memory. If you declare a variable, the compiler will allocate this variable to some consecutive memory cells to hold the value of the variable. The address of the variable is the address of the first memory cell.

One variable always has two properties:

- The address of the variable
- The value of the variable.

Consider the following example:

```
int i, j;  
i = 3;  
j = i;
```

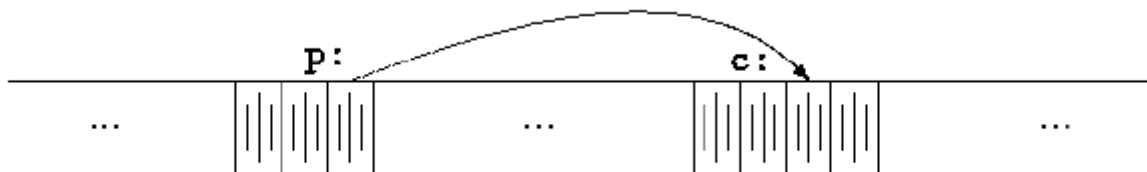
Type of these two variables is integer so they are stored in 2-byte memory area. Suppose that the compiler allocates *i* at the FFEC address in memory and *j* in FFEE, we have:

Variable	Address	Value
i	FFEC	3
j	FFEE	3

Two different variables have different addresses. The *i* = *j* assignment affects only on the value of variables, that means the content of the memory area for *j* will be copied to the content of the memory area for *i*.

Pointers

A pointer is a group of cells (often two or four) that can hold an address. So if *c* is a char and *p* is a pointer that points to it, we could represent the situation this way:



The unary operator `&` gives the address of an object, so the statement

```
p = &c;
```

assigns the address of *c* to the variable *p*, and *p* is said to “point to” *c*. The `&` operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants, or register variables.

Pointer declaration

If you declare a variable, its name is a direct reference to its value. If you have a pointer to a variable or any other object in memory, you have an indirect reference to its value. A pointer variable stores the address of another object or a function. We describe pointers to arrays and functions a little further on. To start out, the declaration of a pointer to an object that is not an array has the following syntax:

type * [type-qualifier-list] name [= initializer];

In declarations, the asterisk (*) means “pointer to”. The identifier name is declared as an object with the type `type *`, or pointer to type. The unary operator `*` is the *indirection* or *dereferencing* operator; when applied to a pointer, it accesses the object the pointer points to.

Here is a simple example:

```
int *iPtr;    // Declare iPtr as a pointer to int.
```

The type `int` is the type of object that the pointer `iPtr` can point to. To make a pointer refer to a certain object, assign it the address of the object. For example, if `iVar` is an `int` variable, then the following assignment makes `iPtr` point to the variable `iVar`:

```
iPtr = &iVar;    // Let iPtr point to the variable iVar.
```

In a pointer declaration, the asterisk (*) is part of an individual declarator. We can thus define and initialize the variables `iVar` and `iPtr` in one declaration, as follows:

```
int iVar = 77, *iPtr = &iVar; // Define an int variable and a pointer to it.
```

The second of these two declarations initializes the pointer `iPtr` with the address of the variable `iVar`, so that `iPtr` points to `iVar`. Figure 0.1 illustrates one possible arrangement of the variables `iVar` and `iPtr` in memory. The addresses shown are purely fictitious examples. As Figure 0.1 shows, the value stored in the pointer `iPtr` is the address of the object `iVar`.

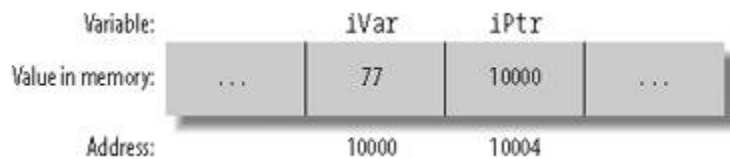


Figure 0.1. A pointer and another object in memory

It is often useful to output addresses for verification and debugging purposes. The `printf()` functions provide a format specifier for pointers: `%p`. The following statement prints the address and contents of the variable `iPtr`:

```
printf("Value of iPtr (i.e. the address of iVar): %p\n"  
      "Address of iPtr: %p\n", iPtr, &iPtr);
```

The size of a pointer in memory given by the expression `sizeof(iPtr)`, for example is the same regardless of the type of object addressed. In other words, a `char` pointer takes up just as much space in memory as a pointer to a large structure. On 32-bit computers, pointers are usually four bytes long.

NULL Pointers

There are times when it's necessary to have a pointer that doesn't point to anything. A *null pointer* is what results when you convert a null pointer constant to a pointer type. A *null pointer constant* is an integer constant expression with the value 0, or such an expression cast as the type `void *`. The macro `NULL`, defined in `stdlib.h`, `stdio.h` and other header files as a null pointer constant, has a value that's guaranteed to be different from any valid pointer. `NULL` is a literal zero, possibly cast to `void*` or `char*`.

You can't use an integer when a pointer is required. The exception is that a literal zero value can be used as the null pointer. (It doesn't have to be a literal zero, but that's the only useful case. Any expression that can be evaluated at compile time, and that is zero, will do. It's not good enough to have an integer variable that might be zero at runtime.)

A null pointer is always unequal to any valid pointer to an object or function. For this reason, functions that return a pointer type usually use a null pointer to indicate a failure

condition. One example is the standard function `fopen()`, which returns a null pointer if it fails to open a file in the specified mode:

```
#include <stdio.h>
/* ... */
FILE *fp = fopen( "demo.txt", "r" );
if ( fp == NULL )
{
    // Error: unable to open the file demo.txt for reading.
}
```

Null pointers are implicitly converted to other pointer types as necessary for assignment operations, or for comparisons using `==` or `!=`. Hence no cast operator is necessary in the previous example.

void Pointers

A pointer to `void`, or *void pointer* for short, is a pointer with the type `void *`. As there are no objects with the type `void`, the type `void *` is used as the all-purpose pointer type. In other words, a void pointer can represent the address of any object but not its type. To access an object in memory, you must always convert a void pointer into an appropriate object pointer.

To declare a function that can be called with different types of pointer arguments, you can declare the appropriate parameters as pointers to `void`. When you call such a function, the compiler implicitly converts an object pointer argument into a void pointer. A common example is the standard function `memset()`, which is declared in the header file *string.h* with the following prototype:

```
void *memset( void *s, int c, size_t n );
```

The function `memset()` assigns the value of `c` to each of the `n` bytes of memory in the block beginning at the address `s`. For example, the following function call assigns the value 0 to each byte in the structure variable `record`:

```
struct Data { /* ... */ } record;
memset( &record, 0, sizeof(record) );
```

The argument `&record` has the type `struct Data *`. In the function call, the argument is converted to the parameter's type, `void *`.

The compiler likewise converts void pointers into object pointers where necessary. For example, in the following statement, the `malloc()` function returns a void pointer whose value is the address of the allocated memory block. The assignment operation converts the void pointer into a pointer to `int`:

```
int *iPtr = malloc( 1000 * sizeof(int) );
```

Initializing Pointers

Pointer variables with automatic storage duration start with an undefined value, unless their declaration contains an explicit initializer. All variables defined within any block, without the storage class specifier `static`, have automatic storage duration. All other pointers defined without an initializer have the initial value of a null pointer.

You can initialize a pointer with the following kinds of initializers:

- A null pointer constant.
- A pointer to the same type, or to a less qualified version of the same type.
- A void pointer, if the pointer being initialized is not a function pointer. Here again, the pointer being initialized can be a pointer to a more qualified type.

Pointers that do not have automatic storage duration must be initialized with a constant expression, such as the result of an address operation or the name of an array or function. When you initialize a pointer, no implicit type conversion takes place except in the cases just listed. However, you can explicitly convert a pointer value to another pointer type. For example, to read any object byte by byte, you can convert its address into a char pointer to the first byte of the object:

```
double x = 1.5;
char *cPtr = &x;      // Error: type mismatch; no implicit conversion.
char *cPtr = (char *)&x; // OK: cPtr points to the first byte of x.
```

& and * operator

Suppose that x and y are integers and ip is a pointer to int. This artificial sequence shows how to declare a pointer and how to use & and *:

```
int x = 1, y = 2, z[10];
int *ip; /* ip is a pointer to int */
ip = &x; /* ip now points to x */
y = *ip; /* y is now 1 */
*ip = 0; /* x is now 0 */
ip = &z[0]; /* ip now points to z[0] */
```

The declaration of x, y, and z are what we've seen all along. The declaration of the pointer ip,

```
int *ip;
```

is intended as a mnemonic; it says that the expression *ip is an int. The syntax of the declaration for a variable mimics the syntax of expressions in which the variable might appear. This reasoning applies to function declarations as well. For example,

```
double *dp, atof(char *);
```

says that in an expression *dp and atof(s) have values of double, and that the argument of atof is a pointer to char.

If ip points to the integer x, then *ip can occur in any context where x could, so

```
*ip = *ip + 10;
```

increments *ip by 10.

The unary operators * and & bind more tightly than arithmetic operators, so the assignment

```
y = *ip + 1
```

takes whatever ip points at, adds 1, and assigns the result to y, while

```
*ip += 1
```

increments what ip points to, as do

```
++*ip
```

and

```
(*ip)++
```

The parentheses are necessary in this last example; without them, the expression would increment ip instead of what it points to, because unary operators like * and ++ associate right to left.

Finally, since pointers are variables, they can be used without dereferencing. For example, if iq is another pointer to int,

```
iq = ip
```

copies the contents of ip into iq, thus making iq point to whatever ip pointed to.

Operators with Pointers

This section describes the operations that can be performed using pointers. The most important of these operations is accessing the object or function that the pointer refers to. You can also compare pointers, and use them to iterate through a memory block.

Using Pointers to Read and Modify Objects

The indirection operator * yields the location in memory whose address is stored in a pointer. If ptr is a pointer, then *ptr designates the object (or function) that ptr points to. Using the indirection operator is sometimes called *dereferencing* a pointer. The type of the pointer determines the type of object that is assumed to be at that location in memory. For example, when you access a given location using an int pointer, you read or write an object of type int.

Unlike the multiplication operator *, the indirection operator * is a unary operator; that is, it has only one operand. In Listing 0.1, ptr points to the variable x. Hence the expression *ptr is equivalent to the variable x itself.

Listing 0.1. Dereferencing a pointer

```
double x, y, *ptr;    // Two double variables and a pointer to double.
ptr = &x;              // Let ptr point to x.
*ptr = 7.8;           // Assign the value 7.8 to the variable x.
*ptr *= 2.5;          // Multiply x by 2.5.
y = *ptr + 0.5;       // Assign y the result of the addition x + 0.5.
```

Do not confuse the asterisk (*) in a pointer declaration with the indirection operator. The syntax of the declaration can be seen as an illustration of how to use the pointer. An example:

```
double *ptr;
```

As declared here, ptr has the type double * (read: "pointer to double"). Hence the expression *ptr would have the type double.

Of course, the indirection operator * must be used with only a pointer that contains a valid address. This usage requires careful programming! Without the assignment ptr = &x in Listing 0.1, all of the statements containing *ptr would be senseless dereferencing an undefined pointer value and might well cause the program to crash.

A pointer variable is itself an object in memory, which means that a pointer can point to it. To declare a pointer to a pointer, you must use two asterisks, as in the following example:

```
char c = 'A', *cPtr = &c, **cPtrPtr = &cPtr;
```

The expression *cPtrPtr now yields the char pointer cPtr, and the value of **cPtrPtr is the char variable c. The diagram in Figure 0.2 illustrates these references.

Figure 0.2. A pointer to a pointer



Pointers to pointers are not restricted to the two-stage indirection illustrated here. You can define pointers with as many levels of indirection as you need. However, you cannot assign a pointer to a pointer its value by mere repetitive application of the address operator:

```
char c = 'A', **cPtrPtr = &(&c);    // Wrong!
```

The second initialization in this example is illegal: the expression (&c) cannot be the operand of &, because it is not an lvalue. In other words, there is no pointer to char in this example for cPtrPtr to point to.

If you pass a pointer to a function by reference so that the function can modify its value, then the function's parameter is a pointer to a pointer. The following simple example is a function that dynamically creates a new record and stores its address in a pointer variable:

```
#include <stdlib.h>
// The record type:
typedef struct { long key; /* ... */ } Record;

_Bool newRecord( Record **ppRecord )
{
    *ppRecord = malloc( sizeof(Record) );
    if ( *ppRecord != NULL )
    {
```



```
/* ... Initialize the new record's members ... */
return 1;
}
else
    return 0;
}
```

The following statement is one possible way to call the `newRecord()` function:

```
Record *pRecord = NULL;
if ( newRecord( &pRecord) )
{
    /* ... pRecord now points to a new Record object ... */
}
```

The expression `*pRecord` yields the new record, and `(*pRecord).key` is the member `key` in that record. The parentheses in the expression `(*pRecord).key` are necessary, because the dot operator `(.)` has higher precedence than the indirection operator `(*)`.

Instead of this combination of operators and parentheses, you can also use the arrow operator `->` to access structure or union members. If `p` is a pointer to a structure or union with a member `m`, then the expression `p->m` is equivalent to `(*p).m`. Thus the following statement assigns a value to the member `key` in the structure that `pRecord` points to:

```
pRecord->key = 123456L;
```

Modifying and Comparing Pointers

Besides using assignments to make a pointer refer to a given object or function, you can also modify an object pointer using arithmetic operations. When you perform *pointer arithmetic*, the compiler automatically adapts the operation to the size of the objects referred to by the pointer type.

You can perform the following operations on pointers to objects:

- Adding an integer to, or subtracting an integer from, a pointer.
- Subtracting one pointer from another.
- Comparing two pointers.

When you subtract one pointer from another, the two pointers must have the same basic type, although you can disregard any type. Furthermore, you may compare any pointer with a null pointer constant using the equality operators (`==` and `!=`), and you may compare any object pointer with a pointer to `void`.

The three pointer operations described here are generally useful only for pointers that refer to the elements of an array. To illustrate the effects of these operations, consider two pointers `p1` and `p2`, which point to elements of an array `a`:

- If `p1` points to the array element `a[i]`, and `n` is an integer, then the expression `p2 = p1 + n` makes `p2` point to the array element `a[i+n]` (assuming that `i+n` is an index within the array `a`).
- The subtraction `p2 - p1` yields the number of array elements between the two pointers, with the type `ptrdiff_t`. The type `ptrdiff_t` is defined in the header file

stddef.h, usually as `int`. After the assignment `p2 = p1 + n`, the expression `p2 - p1` yields the value of `n`.

- The comparison `p1 < p2` yields `TRUE` if the element referenced by `p2` has a greater index than the element referenced by `p1`. Otherwise, the comparison yields `false`.

Because the name of an array is implicitly converted into a pointer to the first array element wherever necessary, you can also substitute pointer arithmetic for array subscript notation:

- The expression `a + i` is a pointer to `a[i]`, and the value of `*(a+i)` is the element `a[i]`.
- The expression `p1 - a` yields the index `i` of the element referenced by `p1`.

In Listing 0.2, the function `selection_sortf()` sorts an array of float elements using the selection-sort algorithm. The helper function `swapf()` remains unchanged.

Listing 0.2. Pointer version of the `selection_sortf()` function

```
// The swapf( ) function exchanges the values of two float variables.
// Arguments: Two pointers to float.

inline void swapf( float *p1, float *p2 );
{
    float tmp = *p1; *p1 = *p2; *p2 = tmp; // Swap *p1 and *p2.
}
// The function selection_sortf( ) uses the selection-sort
// algorithm to sort an array of float elements.
// Arguments: An array of float, and its length.

void selection_sortf( float a[ ], int n ) // Sort an array a of n float elements.
{
    if ( n <= 1 ) return; // Nothing to sort.

    register float *last = a + n-1, // A pointer to the last element.
        *p, // A pointer to a selected element.
        *minPtr; // A pointer to the current minimum.

    for ( ; a < last; ++a ) // Walk the pointer a through the array.
    {
        minPtr = a; // Find the smallest element
        for ( p = a+1; p <= last; ++p ) // between a and the end of the array.
            if ( *p < *minPtr )
                minPtr = p;
        swapf( a, minPtr ); // Swap the smallest element
    } // with the element at a.
}
```

The pointer version of such a function is generally more efficient than the index version, since accessing the elements of the array `a` using an index `i`, as in the expression `a[i]` or `*(a+i)`, involves adding the address `a` to the value `i*sizeof(element_type)` to obtain the address of the corresponding array element. The pointer version requires less arithmetic, because the pointer itself is incremented instead of the index, and points to the required array element directly.

Arrays

In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand.

An array contains objects of a given type, stored consecutively in a continuous memory block. The individual objects are called the *elements* of an array. The elements' type can be any object type. No other types are permissible: array elements may not have a function type or an incomplete type.

An array is also an object itself, and its type is derived from its elements' type. More specifically, an array's type is determined by the type and number of elements in the array. If an array's elements have type *T*, then the array is called an "array of *T*." If the elements have type `int`, for example, then the array's type is "array of `int`." The type is an incomplete type, however, unless it also specifies the number of elements. If an array of `int` has 16 elements, then it has a complete object type, which is "array of 16 `int` elements."

Declarations of Arrays

The definition of an array determines its name, the type of its elements, and the number of elements in the array. An array definition without any explicit initialization has the following syntax:

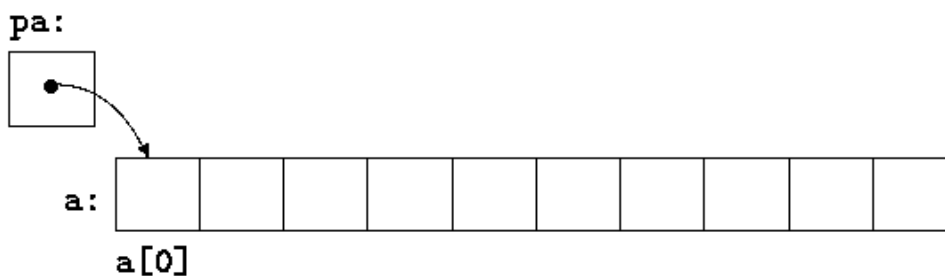
```
type name[ number_of_elements ];
```

The number of elements, between square brackets (`[]`), must be an integer expression whose value is greater than zero.

For example, the declaration

```
int a[10];
```

defines an array of size 10, that is, a block of 10 consecutive objects named `a[0]`, `a[1]`, ..., `a[9]`.



Another example:

```
char buffer[4*512];
```

defines an array with the name `buffer`, which consists of 2,048 elements of type `char`.

You can determine the size of the memory block that an array occupies using the `sizeof` operator. The array's size in memory is always equal to the size of one element times the number of elements in the array. Thus, for the array `buffer` in our example, the expression `sizeof(buffer)` yields the value of `2048 * sizeof(char)`; in other words, the array `buffer` occupies 2,048 bytes of memory, because `sizeof(char)` always equals one.

In an array definition, you can specify the number of elements as a constant expression, or, under certain conditions, as an expression involving variables. The resulting array is accordingly called a *fixed-length* or a *variable-length* array.

Fixed-Length Arrays

Most array definitions specify the number of array elements as a constant expression. An array so defined has a fixed length. Thus the array `buffer` defined in the previous example is a fixed-length array.

Fixed-length arrays can have any storage class: you can define them outside all functions or within a block, and with or without the storage class specifier `static`. The only restriction is that no function parameter can be an array. An array argument passed to a function is always converted into a pointer to the first array element.

The four array definitions in the following example are all valid:

```
int a[10];           // a has external linkage.
static int b[10];    // b has static storage duration and file scope.
```

```
void func( )
{
    static int c[10]; // c has static storage duration and block scope.
    int d[10];        // d has automatic storage duration.
    /* ... */
}
```

Variable-Length Arrays

C99 also allows you to define an array using a nonconstant expression for the number of elements, if the array has automatic storage duration; in other words, if the definition occurs within a block and does not have the specifier `static`. Such an array is then called a *variable-length array*.

Furthermore, the name of a variable-length array must be an *ordinary identifier*. Thus members of structures or unions cannot be variable-length arrays. In the following examples, only the definition of the array `vla` is a permissible definition:

```
void func( int n )
{
    int vla[2*n];        // OK: storage duration is automatic.
    static int e[n];      // Illegal: a variable length array cannot have
                        //      static storage duration.
    struct S { int f[n]; }; // Illegal: f is not an ordinary identifier.
    /* ... */
}
```

Like any other automatic variable, a variable-length array is created anew each time the program flow enters the block containing its definition. As a result, the array can have a different length at each such instantiation. Once created, however, even a variable-length array cannot change its length during its storage duration.

Storage for automatic objects is allocated on the stack, and is released when the program flow leaves the block. For this reason, variable-length array definitions are useful only for small, temporary arrays. To create larger arrays dynamically, you should generally allocate storage space explicitly using the standard functions `malloc()` and `calloc()`. The storage duration of such arrays then ends with the end of the program, or when you release the allocated memory by calling the function `free()`.

Accessing Array Elements

The subscript operator `[]` provides an easy way to address the individual elements of an array by index. If `myArray` is the name of an array and `i` is an integer, then the expression `myArray[i]` designates the array element with the index `i`. Array elements are indexed beginning with 0. Thus, if `len` is the number of elements in an array, the last element of the array has the index `len-1`.

The following code fragment defines the array `myArray` and assigns a value to each element.

```
#define A_SIZE 4
long myarray[A_SIZE];
for (int i = 0; i < A_SIZE; ++i)
    myarray[i] = 2 * i;
```

The diagram in Figure 0.3 illustrates the result of this assignment loop.



Figure 0.3. Values assigned to elements by index

An array index can be any integer expression desired. The subscript operator `[]` does not bring any range checking with it; C gives priority to execution speed in this regard. It is up to you the programmer to ensure that an index does not exceed the range of permissible values. The following incorrect example assigns a value to a memory location outside the array:

```
long myarray[4];
myArray[4] = 8;    // Error: subscript must not exceed 3.
```

Such "off-by-one" errors can easily cause a program to crash, and are not always as easy to recognize as in this simple example.

Another way to address array elements, as an alternative to the subscript operator, is to use pointer arithmetic. After all, the name of an array is implicitly converted into a pointer to the first array element in all expressions except `sizeof` operations. For example, the expression `myArray+i` yields a pointer to the element with the index `i`, and the expression `*(myArray+i)` is equivalent to `myArray[i]`.

The following loop statement uses a pointer instead of an index to step through the array `myArray`, and doubles the value of each element:

```
for (long *p = myArray; *p < myArray + A_SIZE; ++p)
    *p *= 2;
```

Initializing Arrays

If you do not explicitly initialize an array variable, the usual rules apply: if the array has automatic storage duration, then its elements have undefined values. Otherwise, all elements are initialized by default to the value 0. If the elements are pointers, they are initialized to `NULL`.

Writing Initialization Lists

To initialize an array explicitly when you define it, you must use an *initialization list*: this is a comma-separated list of *initializers*, or initial values for the individual array elements, enclosed in braces. An example:

```
int a[4] = { 1, 2, 4, 8 };
```

This definition gives the elements of the array `a` the following initial values:

```
a[0] = 1, a[1] = 2, a[2] = 4, a[3] = 8
```

When you initialize an array, observe the following rules:

- You cannot include an initialization in the definition of a variable-length array.
- If the array has static storage duration, then the array initializers must be constant expressions. If the array has automatic storage duration, then you can use variables in its initializers.
- You may omit the length of the array in its definition if you supply an initialization list. The array's length is then determined by the index of the last array element for which the list contains an initializer. For example, the definition of the array `a` in the previous example is equivalent to this:

```
int a[ ] = { 1, 2, 4, 8 }; // An array with four elements.
```

- If the definition of an array contains both a length specification and an initialization list, then the length is that specified by the expression between the square brackets. Any elements for which there is no initializer in the list are initialized to zero (or `NULL`, for pointers). If the list contains more initializers than the array has elements, the superfluous initializers are simply ignored.
- A superfluous comma after the last initializer is also ignored.

As a result of these rules, all of the following definitions are equivalent:

```
int a[4] = { 1, 2 };
int a[ ] = { 1, 2, 0, 0 };
int a[ ] = { 1, 2, 0, 0, };
int a[4] = { 1, 2, 0, 0, 5 };
```

In the final definition, the initializer 5 is ignored. Most compilers generate a warning when such a mismatch occurs.

Array initializers must have the same type as the array elements. If the array elements' type is a union, structure, or array type, then each initializer is generally another initialization list. An example:

```
typedef struct { unsigned long pin;
    char name[64];
    /* ... */
} Person;
Person team[6] = { { 1000, "Mary"}, { 2000, "Harry"} };
```

The other four elements of the array `team` are initialized to 0, or in this case, to `{ 0, "" }`. You can also initialize arrays of `char` or `wchar_t` with string literals.

Initializing Specific Elements

C99 has introduced *element designators* to allow you to associate initializers with specific elements. To specify a certain element to initialize, place its index in square brackets. In other words, the general form of an element designator for array elements is:

[constant_expression]

The index must be an integer constant expression. In the following example, the element designator is `[A_SIZE/2]`:

```
#define A_SIZE 20
int a[A_SIZE] = { 1, 2, [A_SIZE/2] = 1, 2 };
```

This array definition initializes the elements `a[0]` and `a[10]` with the value 1, and the elements `a[1]` and `a[11]` with the value 2. All other elements of the array will be given the initial value 0. As this example illustrates, initializers without an element designator are associated with the element following the last one initialized.

If you define an array without specifying its length, the index in an element designator can have any non-negative integer value. As a result, the following definition creates an array of 1,001 elements:

```
int a[ ] = { [1000] = -1 };
```

All of the array's elements have the initial value 0, except the last element, which is initialized to the value -1.

Multidimensional Arrays

A *multidimensional* array in C is merely an array whose elements are themselves arrays. The elements of an n -dimensional array are $(n-1)$ -dimensional arrays. For example, each element of a two-dimensional array is a one-dimensional array. The elements of a one-dimensional array, of course, do not have an array type.

A multidimensional array declaration has a pair of brackets for each dimension:

```
char screen[10][40][80];    // A three-dimensional array.
```

The array `screen` consists of the 10 elements `screen[0]` to `screen[9]`. Each of these elements is a two-dimensional array, consisting in turn of 40 one-dimensional arrays of 80 characters each. All in all, the array `screen` contains 32,000 elements with the type `char`.

To access a `char` element in the three-dimensional array `screen`, you must specify three indices. For example, the following statement writes the character `Z` in the last `char` element of the array:

```
screen[9][39][79] = 'Z';
```

Matrices

Two-dimensional arrays are also called *matrices*. Because they are so frequently used, they merit a closer look. It is often helpful to think of the elements of a matrix as being arranged in rows and columns. Thus the matrix `mat` in the following definition has three rows and five columns:

```
float mat[3][5];
```

The three elements `mat[0]`, `mat[1]`, and `mat[2]` are the rows of the matrix `mat`. Each of these rows is an array of five `float` elements. Thus the matrix contains a total of $3 \times 5 = 15$ `float` elements, as the following diagram illustrates:

	0	1	2	3	4
mat[0]	0.0	0.1	0.2	0.3	0.4
mat[1]	1.0	1.1	1.2	1.3	1.4
mat[2]	2.0	2.1	2.2	2.3	2.4

The values specified in the diagram can be assigned to the individual elements by a nested loop statement. The first index specifies a row, and the second index addresses a column in the row:

```
for ( int row = 0; row < 3; ++row )  
    for ( int col = 0; col < 5; ++col )  
        mat[row][col] = row + (float)col/10;
```

In memory, the three rows are stored consecutively, since they are the elements of the array `mat`. As a result, the `float` values in this matrix are all arranged consecutively in memory in ascending order.

Declaring Multidimensional Arrays

In an array declaration that is not a definition, the array type can be incomplete; you can declare an array without specifying its length. Such a declaration is a reference to an array that you must define with a specified length elsewhere in the program. However, you must always declare the complete type of an array's elements. For a multidimensional array declaration, only the first dimension can have an unspecified length. All other dimensions must have a magnitude. In declaring a two-dimensional matrix, for example, you must always specify the number of columns.

If the array `mat` in the previous example has external linkage, for example that is, if its definition is placed outside all functions then it can be used in another source file after the following declaration:

```
extern float mat[ ][5];    // External declaration.
```

The external object so declared has an incomplete two-dimensional array type.

Initializing Multidimensional Arrays

You can initialize multidimensional arrays using an initialization list according to the rules described in "Initializing Arrays" earlier in this Unit. There are some peculiarities, however: you do not have to show all the braces for each dimension, and you may use multidimensional element designators.

To illustrate the possibilities, we will consider the array defined and initialized as follows:

```
int a3d[2][2][3] = { { { 1, 0, 0 }, { 4, 0, 0 } },  
                    { { 7, 8, 0 }, { 0, 0, 0 } } };
```

This initialization list includes three levels of list-enclosing braces, and initializes the elements of the two-dimensional arrays `a3d[0]` and `a3d[1]` with the following values:

	0	1	2
<code>a3d[0][0]</code>	1	0	0
<code>a3d[0][1]</code>	4	0	0
	0	1	2
<code>a3d[1][0]</code>	7	8	0
<code>a3d[1][1]</code>	0	0	0

Because all elements that are not associated with an initializer are initialized by default to 0, the following definition has the same effect:

```
int a3d[ ][2][3] = { { { 1 }, { 4 } }, { { 7, 8 } } };
```

This initialization list likewise shows three levels of braces. You do not need to specify that the first dimension has the size 2, as the outermost initialization list contains two initializers.

You can also omit some of the braces. If a given pair of braces contains more initializers than the number of elements in the corresponding array dimension, then the excess initializers are associated with the next array element in the storage sequence. Hence these two definitions are equivalent:

```
int a3d[2][2][3] = { { 1, 0, 0, 4 }, { 7, 8 } };  
int a3d[2][2][3] = { 1, 0, 0, 4, 0, 0, 7, 8 };
```

Finally, you can achieve the same initialization pattern using element designators as follows:

```
int a3d[2][2][3] = {1, [0][1][0]=4, [1][0][0]=7, 8};
```

Again, this definition is equivalent to the following:

```
int a3d[2][2][3] = {{1}, [0][1]={4}, [1][0]={7, 8}};
```

Using element designators is a good idea if only a few elements need to be initialized to a value other than 0.

Pointers to Arrays and Arrays of Pointers

Pointers occur in many C programs as references to arrays, and also as elements of arrays. A pointer to an array type is called an *array pointer* for short, and an array whose elements are pointers is called a *pointer array*.

Array Pointers

For the sake of example, the following description deals with an array of `int`. The same principles apply for any other array type, including multidimensional arrays.

To declare a pointer to an array type, you must use parentheses, as the following example illustrates:

```
int (* arrPtr)[10] = NULL; // A pointer to an array of
                          // ten elements with type int.
```

Without the parentheses, the declaration `int * arrPtr[10];` would define `arrPtr` as an array of 10 pointers to `int`. Arrays of pointers are described in the next section.

In the example, the pointer to an array of 10 `int` elements is initialized with `NULL`. However, if we assign it the address of an appropriate array, then the expression `*arrPtr` yields the array, and `(*arrPtr)[i]` yields the array element with the index `i`. According to the rules for the subscript operator, the expression `(*arrPtr)[i]` is equivalent to `*((*arrPtr)+i)`. Hence `**arrPtr` yields the first element of the array, with the index 0.

In order to demonstrate a few operations with the array pointer `arrPtr`, the following example uses it to address some elements of a two-dimensional array that is, some rows of a matrix:

```
int matrix[3][10];    // Array of three rows, each with 10 columns.
                      // The array name is a pointer to the first
                      // element; i.e., the first row.
arrPtr = matrix;      // Let arrPtr point to the first row of
                      // the matrix.
(*arrPtr)[0] = 5;     // Assign the value 5 to the first element of the
                      // first row.
                      //
arrPtr[2][9] = 6;     // Assign the value 6 to the last element of the
                      // last row.
                      //
++arrPtr;             // Advance the pointer to the next row.
(*arrPtr)[0] = 7;     // Assign the value 7 to the first element of the
                      // second row.
```

After the initial assignment, `arrPtr` points to the first row of the matrix, just as the array name `matrix` does. At this point you can use `arrPtr` in the same way as `matrix` to access the elements. For example, the assignment `(*arrPtr)[0] = 5` is equivalent to `arrPtr[0][0] = 5` or `matrix[0][0] = 5`.

However, unlike the array name `matrix`, the pointer name `arrPtr` does not represent a constant address, as the operation `++arrPtr` shows. The increment operation increases the address stored in an array pointer by the size of one array in this case, one row of the matrix, or ten times the number of bytes in an `int` element.

If you want to pass a multidimensional array to a function, you must declare the corresponding function parameter as a pointer to an array type.

One more word of caution: if `a` is an array of ten `int` elements, then you cannot make the pointer from the previous example, `arrPtr`, point to the array `a` by this assignment:

```
arrPtr = a; // Error: mismatched pointer types.
```

The reason is that an array name, such as `a`, is implicitly converted into a pointer to the array's first element, not a pointer to the whole array. The pointer to `int` is not implicitly converted into a pointer to an array of `int`. The assignment in the example requires an explicit type conversion, specifying the target type `int (*)[10]` in the cast operator:

```
arrPtr = (int (*)[10])a; // OK
```

You can derive this notation for the array pointer type from the declaration of `arrPtr` by removing the identifier (see "[Type Names](#)" in [Unit 11](#)). However, for more readable and more flexible code, it is a good idea to define a simpler name for the type using `typedef`:

```
typedef int ARRAY_t[10]; // A type name for "array of ten int elements".
ARRAY_t a,              // An array of this type,
*arrPtr;                // and a pointer to this array type.
arrPtr = (ARRAY_t *)a;   // Let arrPtr point to a.
```

Pointer Arrays

Pointer arrays that is, arrays whose elements have a pointer type are often a handy alternative to two-dimensional arrays. Usually the pointers in such an array point to dynamically allocated memory blocks.

For example, if you need to process strings, you could store them in a two-dimensional array whose row size is large enough to hold the longest string that can occur:

```
#define ARRAY_LEN 100
#define STRLEN_MAX 256
char myStrings[ARRAY_LEN][STRLEN_MAX] =
{ // Several corollaries of Murphy's Law:
  "If anything can go wrong, it will.",
  "Nothing is foolproof, because fools are so ingenious.",
  "Every solution breeds new problems."
};
```

However, this technique wastes memory, as only a small fraction of the 25,600 bytes devoted to the array is actually used. For one thing, a short string leaves most of a row

empty; for another, memory is reserved for whole rows that may never be used. A simple solution in such cases is to use an array of pointers that reference the objects in this case, the strings and to allocate memory only for the pointer array and for objects that actually exist. Unused array elements are null pointers.

```
#define ARRAY_LEN 100
char *myStrPtr[ARRAY_LEN] = // Array of pointers to char
{ // Several corollaries of Murphy's Law:
    "If anything can go wrong, it will.",
    "Nothing is foolproof, because fools are so ingenious.",
    "Every solution breeds new problems."
};
```



Figure 0.4. Pointer array

The diagram in Figure 0.4 illustrates how the objects are stored in memory.

The pointers not yet used can be made to point to other strings at runtime. The necessary storage can be reserved dynamically in the usual way. The memory can also be released when it is no longer needed.

The program in Listing 0.3 is a simple version of the filter utility *sort*. It reads text from the standard input stream, sorts the lines alphanumerically, and prints them to standard output. This routine does not move any strings: it merely sorts an array of pointers.

Listing 0.3. A simple program to sort lines of text

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

char *getline(void);          // Reads a line of text
int str_compare(const void *, const void *);

#define NLINES_MAX 1000      // Maximum number of text lines.
char *linePtr[NLINES_MAX];   // Array of pointers to char.

int main( )
{
    // Read lines:
    int n = 0;                // Number of lines read.
    for ( ; n < NLINES_MAX && (linePtr[n] = getline( )) != NULL; ++n )
```

```

;

if ( !feof(stdin) )      // Handle errors.
{
    if ( n == NLINES_MAX )
        fputs( "sorttext: too many lines.\n", stderr );
    else
        fputs( "sorttext: error reading from stdin.\n", stderr );
}
else                      // Sort and print.
{
    qsort( linePtr, n, sizeof(char*), str_compare );    // Sort.
    for ( char **p = linePtr; p < linePtr+n; ++p )    // Print.
        puts(*p);
}
return 0;
}

// Reads a line of text from stdin; drops the terminating newline character.
// Return value: A pointer to the string read, or
//              NULL at end-of-file, or if an error occurred.
#define LEN_MAX 512                // Maximum length of a line.

char *getline( )
{
    char buffer[LEN_MAX], *linePtr = NULL;
    if ( fgets( buffer, LEN_MAX, stdin ) != NULL )
    {
        size_t len = strlen( buffer );

        if ( buffer[len-1] == '\n' )        // Trim the newline character.
            buffer[len-1] = '\0';
        else
            ++len;

        if ( (linePtr = malloc( len )) != NULL ) // Get enough memory for the line.
            strcpy( linePtr, buffer );        // Copy the line to the allocated block.
    }
    return linePtr;
}

// Comparison function for use by qsort( ).
// Arguments: Pointers to two elements in the array being sorted:
//             here, two pointers to pointers to char (char **).
int str_compare( const void *p1, const void *p2 )
{

```

```
    return strcmp( *(char **)p1, *(char **)p2 );
}
```

The maximum number of lines that the program in Listing 0.3 can sort is limited by the constant `NLINES_MAX`. However, we could remove this limitation by creating the array of pointers to text lines dynamically as well.

Strings

A *string* is a continuous sequence of characters terminated by `'\0'`, the null character. The length of a string is considered to be the number of characters excluding the terminating null character. There is no string type in C, and consequently there are no operators that accept strings as operands.

Instead, strings are stored in arrays whose elements have the type `char` or `wchar_t`. Strings of wide characters that is, characters of the type `wchar_t` are also called *wide strings*. The C standard library provides numerous functions to perform basic operations on strings, such as comparing, copying, and concatenating them.

Declarations and Uses of Strings

You can initialize arrays of `char` or `wchar_t` using string literals. For example, the following two array definitions are equivalent:

```
char str1[30] = "Let's go";    // String length: 8; array length: 30.
```

```
char str1[30] = { 'L', 'e', 't', '\'', 's', '\'', 'g', 'o', '\0' };
```

An array holding a string must always be at least one element longer than the string length to accommodate the terminating null character. Thus the array `str1` can store strings up to a maximum length of 29. It would be a mistake to define the array with length 8 rather than 30, because then it wouldn't contain the terminating null character.

If you define a character array without an explicit length and initialize it with a string literal, the array created is one element longer than the string length. An example:

```
char str2[ ] = " to London!";  // String length: 11 (note leading space);
                               // array length: 12.
```

The following statement uses the standard function `strcat()` to append the string in `str2` to the string in `str1`. The array `str1` must be large enough to hold all the characters in the concatenated string.

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

```
char str1[30] = "Let's go";
char str2[ ] = " to London!";
```

```
/* ... */
```

```
strcat( str1, str2 );  
puts( str1 );
```

The output printed by the `puts()` call is the new content of the array `str1`:

Let's go to London!

The names `str1` and `str2` are pointers to the first character of the string stored in each array. Such a pointer is called a *pointer to a string*, or a *string pointer* for short. String manipulation functions such as `strcat()` and `puts()` receive the beginning addresses of strings as their arguments. Such functions generally process a string character by character until they reach the terminator, `'\0'`. The function in Listing 0.1 is one possible implementation of the standard function `strcat()`. It uses pointers to step through the strings referenced by its arguments.

Listing 0.1. Function `strcat()`

```
// The function strcat( ) appends a copy of the second string  
// to the end of the first string.  
// Arguments:  Pointers to the two strings.  
// Return value: A pointer to the first string, now concatenated with the second.
```

```
char *strcat( char * restrict s1, const char * restrict s2 )  
{  
    char *rtnPtr = s1;  
    while ( *s1 != '\0' )          // Find the end of string s1.  
        ++s1;  
    while (( *s1++ = *s2++ ) != '\0' ) // The first character from s2 replaces  
        ;                          // the terminator of s1.  
    return rtnPtr;  
}
```

The char array beginning at the address `s1` must be at least as long as the sum of both the two strings' lengths, plus one for the terminating null character. To test for this condition before calling `strcat()`, you might use the standard function `strlen()`, which returns the length of the string referenced by its argument:

```
if ( sizeof(str1) >= ( strlen( str1 ) + strlen( str2 ) + 1 )  
    strcat( str1, str2 );
```

Accordingly, the initialization of a `wchar_t` array looks like this:

```
#include <stddef.h>          // Definition of the type wchar_t.  
/* ... */  
wchar_t dinner[ ] = L"chop suey"; // String length: 10;  
                                // array length: 11;  
                                // array size: 11 * sizeof(wchar_t)
```

Character and String Processing

Character Processing Functions

The standard library provides a number of functions to classify characters and to perform conversions on them. The header *ctype.h* declares such functions for byte characters, with character codes from 0 to 255. The header *wctype.h* declares similar functions for wide characters, which have the type `wchar_t`. These functions are commonly implemented as macros.

The results of these functions, except for `isdigit()` and `isxdigit()`, depends on the current locale setting for the locale category `LC_CTYPE`. You can query or change the locale using the `setlocale()` function.

Character Classification Functions

The functions listed in Table 0.1 test whether a character belongs to a certain category. Their return value is nonzero, or true, if the argument is a character code in the given category.

Table 0.1. Character classification functions

Category	Functions in <i>ctype.h</i>	Functions in <i>wctype.h</i>
Letters	<code>isalpha()</code>	<code>iswalpha()</code>
Lowercase letters	<code>islower()</code>	<code>iswlower()</code>
Uppercase letters	<code>isupper()</code>	<code>iswupper()</code>
Decimal digits	<code>isdigit()</code>	<code>iswdigit()</code>
Hexadecimal digits	<code>isxdigit()</code>	<code>iswxdigit()</code>
Letters and decimal digits	<code>isalnum()</code>	<code>iswalnum()</code>
Printable characters (including whitespace)	<code>isprint()</code>	<code>iswprint()</code>
Printable, non-whitespace characters	<code>isgraph()</code>	<code>iswgraph()</code>
Whitespace characters	<code>isspace()</code>	<code>iswspace()</code>
Whitespace characters that separate words in a line of text	<code>isblank()</code>	<code>iswblank()</code>
Punctuation marks	<code>ispunct()</code>	<code>iswpunct()</code>
Control characters	<code>isctrl()</code>	<code>iswctrl()</code>

The functions `isgraph()` and `iswgraph()` behave differently if the execution character set contains other byte-coded, printable, whitespace characters (that is, whitespace characters which are not control characters) in addition to the space character (' '). In that case, `iswgraph()` returns false for all such printable whitespace characters, while `isgraph()` returns false only for the space character (' ').

The header *wctype.h* also declares the two additional functions listed in Table 0.2 to test wide characters. These are called the *extensible* classification functions, which you can use to test whether a wide-character value belongs to an implementation-defined category designated by a string.

Table 0.2. Extensible character classification functions

Purpose	Function
Map a string argument that designates a character class to a scalar value that can be used as the second argument to <code>iswctype()</code> .	<code>wctype()</code>
Test whether a wide character belongs to the class designated by the second argument.	<code>iswctype()</code>

The two functions in Table 0.2 can be used to perform at least the same tests as the functions listed in Table 0.1. The strings that designate the character classes recognized by `wctype()` are formed from the name of the corresponding test functions, minus the prefix `isw`. For example, the string "alpha", like the function name `iswalph()`, designates the category "letters." Thus for a wide character value *wc*, the following tests are equivalent:

```
iswalph( wc )
iswctype( wc, wctype("alpha") )
```

Implementations may also define other such strings to designate locale-specific character classes.

Case Mapping Functions

The functions listed in Table 0.3 yield the uppercase letter that corresponds to a given lowercase letter, and vice versa. All other argument values are returned unchanged.

Table 0.3. Character conversion functions

Conversion	Functions in <i>ctype.h</i>	Functions in <i>wctype.h</i>
Upper- to lowercase	<code>tolower()</code>	<code>towlower()</code>
Lower- to uppercase	<code>toupper()</code>	<code>towupper()</code>

Here again, as in the previous section, the header *wctype.h* declares two additional *extensible functions* to convert wide characters. These are described in Table 0.4. Each kind of character conversion supported by the given implementation is designated by a string.

Table 0.4. Extensible character conversion functions

Purpose	Function
Map a string argument that designates a character conversion to a scalar value that can be used as the second argument to <code>towctrans()</code> .	<code>wctrans()</code>

Purpose	Function
Perform the conversion designated by the second argument on a given wide character.	towctrans()

The two functions in Table 0.4 can be used to perform at least the same conversions as the functions listed in Table 0.3. The strings that designate those conversions are "tolower" and "toupper". Thus for a wide character `wc`, the following two calls have the same result:

```
towupper(wc);  
towctrans(wc, wctrans("toupper"));
```

Implementations may also define other strings to designate locale-specific character conversions.

String Processing Functions

A *string* is a continuous sequence of characters terminated by `'\0'`, the string terminator character. The length of a string is considered to be the number of characters before the string terminator. Strings are stored in arrays whose elements have the type `char` or `wchar_t`. Strings of wide characters that is, characters of the type `wchar_t` are also called *wide strings*.

C does not have a basic type for strings, and hence has no operators to concatenate, compare, or assign values to strings. Instead, the standard library provides numerous functions, listed in

Table 0.5, to perform these and other operations with strings. The header *string.h* declares the functions for conventional strings of `char`. The names of these functions begin with `str`. The header *wchar.h* declares the corresponding functions for strings of wide characters, with names beginning with `wcs`.

Like any other array, a string that occurs in an expression is implicitly converted into a pointer to its first element. Thus when you pass a string as an argument to a function, the function receives only a pointer to the first character, and can determine the length of the string only by the position of the string terminator character.

Table 0.5. String-processing functions

Purpose	Functions in <code>string.h</code>	Functions in <code>wchar.h</code>
Find the length of a string.	<code>strlen()</code>	<code>wcslen()</code>
Copy a string.	<code>strcpy()</code> , <code>strncpy()</code>	<code>wscpy()</code> , <code>wcsncpy()</code>
Concatenate strings.	<code>strcat()</code> , <code>strncat()</code>	<code>wscat()</code> , <code>wcsncat()</code>
Compare strings.	<code>strcmp()</code> , <code>strncmp()</code> , <code>strcoll()</code>	<code>wscmp()</code> , <code>wcsncmp()</code> , <code>wscoll()</code>
Transform a string so that a comparison of two	<code>strxfrm()</code>	<code>wcsxfrm()</code>

Purpose	Functions in string.h	Functions in wchar.h
transformed strings using strcmp() yields the same result as a comparison of the original strings using the locale-sensitive function strcoll().		
In a string, find:		
<ul style="list-style-type: none"> The first or last occurrence of a given character 	strchr(), strrchr()	wcschr(), wcsrchr()
<ul style="list-style-type: none"> The first occurrence of another string 	strstr()	wcsstr()
<ul style="list-style-type: none"> The first occurrence of any of a given set of characters 	strcspn(), strpbrk()	wcscspn(), wcpbrk()
<ul style="list-style-type: none"> The first character that is not a member of a given set 	strspn()	wcsspn()
Parse a string into tokens	strtok()	wcstok()

Functions

Functions break large computing tasks into smaller ones, and enable people to build on what others have done instead of starting over from scratch. Appropriate functions hide details of operation from parts of the program that don't need to know about them, thus clarifying the whole, and easing the pain of making changes.

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

Function declaration and definition is the area where the ANSI standard has made the most changes to C. It is now possible to declare the type of arguments when a function is declared. The syntax of function declaration also changes, so that declarations and definitions match. This makes it possible for a compiler to detect many more errors than it could before. Furthermore, when arguments are properly declared, appropriate type coercions are performed automatically.

Every function is defined exactly once. A program can declare and call a function as many times as necessary.

The standard clarifies the rules on the scope of names; in particular, it requires that there be only one definition of each external object. Initialization is more general: automatic arrays and structures may now be initialized

Scope of Variables

One of the C language's strengths is its flexibility in defining data storage. There are two aspects that can be controlled in C: scope and lifetime. Scope refers to the places in the code from which the variable can be accessed. Lifetime refers to the points in time at which the variable can be accessed.

Three scopes are available to the programmer:

- **extern:** This is the default for variables declared outside any function. The scope of variables with **extern** scope is all the code in the entire program.
- **static:** The scope of a variable declared **static** outside any function is the rest of the code in that source file. The scope of a variable declared **static** inside a function is the rest of the local block.
- **auto:** This is the default for variables declared inside a function. The scope of an **auto** variable is the rest of the local block.

Three lifetimes are available to the programmer. They do not have predefined keywords for names as scopes do. The first is the lifetime of **extern** and **static** variables, whose lifetime is from before **main()** is called until the program exits. The second is the lifetime of function arguments and automatics, which is from the time the function is called until

it returns. The third lifetime is that of dynamically allocated data. It starts when the program calls `malloc()` or `calloc()` to allocate space for the data and ends when the program calls `free()` or when it exits, whichever comes first.

Local block

A local block is any portion of a C program that is enclosed by the left brace (`{`) and the right brace (`}`). A C function contains left and right braces, and therefore anything between the two braces is contained in a local block. An if statement or a switch statement can also contain braces, so the portion of code between these two braces would be considered a local block. Additionally, you might want to create your own local block without the aid of a C function or keyword construct. This is perfectly legal. Variables can be declared within local blocks, but they must be declared only at the *beginning* of a local block. Variables declared in this manner are visible only within the local block. Duplicate variable names declared within a local block take precedence over variables with the same name declared outside the local block. Here is an example of a program that uses local blocks:

```
#include <stdio.h>
void main(void);
void main()
{
    /* Begin local block for function main() */
    int test_var = 10;
    printf("Test variable before the if statement: %d\n", test_var);
    if (test_var > 5)
    {
        /* Begin local block for "if" statement */
        int test_var = 5;
        printf("Test variable within the if statement: %d\n", test_var);
        {
            /* Begin independent local block (not tied to any function or keyword) */
            int test_var = 0;
            printf("Test variable within the independent local block: %d\n", test_var);
        }
        /* End independent local block */
    }
    /* End local block for "if" statement */
    printf("Test variable after the if statement: %d\n", test_var);
}
/* End local block for function main() */
```

This example program produces the following output:

```
Test variable before the if statement: 10
Test variable within the if statement: 5
Test variable within the independent local block: 0
Test variable after the if statement: 10
```

Notice that as each `test_var` was defined, it took precedence over the previously defined `test_var`. Also notice that when the `if` statement local block had ended, the program had reentered the scope of the original `test_var`, and its value was 10.

Function Definitions

The definition of a function consists of a function head (or the *declarator*) and a function block. The function head specifies the name of the function, the type of its return value, and the types and names of its parameters, if any. The statements in the function block specify what the function does. The general form of a function definition is as follows:

```
return-type function-name(argument declarations) //function head
//Function block
{
    declarations and statements
}
```

In the function head, *name* is the function's name, while *type* consists of at least one type specifier, which defines the type of the function's return value. The return type may be `void` or any object type, except array types. Furthermore, *type* may include the function specifier `inline`, and/or one of the storage class specifiers `extern` and `static`.

A function cannot return a function or an array. However, you can define a function that returns a pointer to a function or a pointer to an array.

The *parameter declarations* are contained in a comma-separated list of declarations of the function's parameters. If the function has no parameters, this list is either empty or contains merely the word `void`.

The type of a function specifies not only its return type, but also the types of all its parameters. Listing 0.1 is a simple function to calculate the volume of a cylinder.

Listing 0.1. Function `cylinderVolume()`

```
// The cylinderVolume( ) function calculates the volume of a cylinder.
// Arguments: Radius of the base circle; height of the cylinder.
// Return value: Volume of the cylinder.
```

```
extern double cylinderVolume( double r, double h )
{
    const double pi = 3.1415926536; // Pi is constant
    return pi * r * r * h;
}
```

This function has the name `cylinderVolume`, and has two parameters, `r` and `h`, both with type `double`. It returns a value with the type `double`.

return statement

The `return` statement ends execution of the current function, and jumps back to where the function was called:

```
return [expression];
```

expression is evaluated and the result is given to the caller as the value of the function call. This return value is converted to the function's return type, if necessary.

A function can contain any number of return statements:

```
// Return the smaller of two integer arguments.
```

```
int min( int a, int b )
{
    if ( a < b ) return a;
    else      return b;
}
```

The contents of this function block can also be expressed by the following single statement:

```
return ( a < b ? a : b );
```

The parentheses do not affect the behavior of the return statement. However, complex return expressions are often enclosed in parentheses for the sake of readability.

A return statement with no *expression* can only be used in a function of type void. In fact, such functions do not need to have a return statement at all. If no return statement is encountered in a function, the program flow returns to the caller when the end of the function block is reached.

Functions and Storage Class Specifiers

The function in Listing 0.1 is declared with the storage class specifier `extern`. This is not strictly necessary, since `extern` is the default storage class for functions. An ordinary function definition that does not contain a `static` or `inline` specifier can be placed in any source file of a program. Such a function is available in all of the program's source files, because its name is an external identifier. You merely have to declare the function before its first use in a given translation unit. Furthermore, you can arrange functions in any order you wish within a source file. The only restriction is that you cannot define one function within another. C does not allow you to define "local functions" in this way.

You can hide a function from other source files. If you declare a function as `static`, its name identifies it only within the source file containing the function definition. Because the name of a static function is not an external identifier, you cannot use it in other source files. If you try to call such a function by its name in another source file, the linker will issue an error message, or the function call might refer to a different function with the same name elsewhere in the program.

The function `printArray()` in Listing 0.2 might well be defined using `static` because it is a special-purpose helper function, providing formatted output of an array of float variables.

Listing 0.2. Function `printArray()`

```
// The static function printArray() prints the elements of an array
// of float to standard output, using printf() to format them.
```

// Arguments: An array of float, and its length.
// Return value: None.

```
static void printArray( const float array[ ], int n )
{
    for ( int i=0; i < n; ++i )
    {
        printf( "%12.2f", array[i] );    // Field width: 12; decimal places: 2
        if ( i % 5 == 4 ) putchar( '\n' );// New line after every 5 numbers
    }
    if ( n % 5 != 0 ) putchar( '\n' );// New line at the end of the output
}
```

If your program contains a call to the `printArray()` function before its definition, you must first declare it using the `static` keyword:

```
static void printArray(const float [ ], int);
```

```
int main( )
{
    float farray[123];
    /* ... */
    printArray( farray, 123 );
    /* ... */
}
```

Parameters of Functions

The parameters of a function are ordinary local variables. The program creates them, and initializes them with the values of the corresponding arguments, when a function call occurs. Their scope is the function block. A function can change the value of a parameter without affecting the value of the argument in the context of the function call. In Listing 0.3, the `factorial()` function, which computes the factorial of a whole number, modifies its parameter `n` in the process.

Listing 0.3. Function `factorial()`

// `factorial()` calculates $n!$, the factorial of a non-negative number `n`.
// For $n > 0$, $n!$ is the product of all integers from 1 to `n` inclusive.
// $0!$ equals 1.
// Argument: A whole number, with type unsigned int.
// Return value: The factorial of the argument, with type long double.

```
long double factorial(register unsigned int n)
{
    long double f = 1;
    while ( n > 1 )
        f *= n--;
```



```
    return f;
}
```

Although the factorial of an integer is always an integer, the function uses the type `long double` in order to accommodate very large results. As Listing 0.3 illustrates, you can use the storage class specifier `register` in declaring function parameters. The `register` specifier is a request to the compiler to make a variable as quickly accessible as possible. No other storage class specifiers are permitted on function parameters.

Arrays as Function Parameters

If you need to pass an array as an argument to a function, you would generally declare the corresponding parameter in the following form:

```
type name[ ]
```

Because array names are automatically converted to pointers when you use them as function arguments, this statement is equivalent to the declaration:

```
type *name
```

When you use the array notation in declaring function parameters, any constant expression between the brackets (`[]`) is ignored. In the function block, the parameter name is a pointer variable, and can be modified. Thus the function `addArray()` in Listing 0.4 modifies its first two parameters as it adds pairs of elements in two arrays.

Listing 0.4. Function `addArray()`

```
// addArray( ) adds each element of the second array to the
// corresponding element of the first (i.e., "array1 += array2", so to speak).
// Arguments:  Two arrays of float and their common length.
// Return value: None.
```

```
void addArray( register float a1[ ], register const float a2[ ], int len )
{
    register float *end = a1 + len;
    for ( ; a1 < end; ++a1, ++a2 )
        *a1 += *a2;
}
```

An equivalent definition of the `addArray()` function, using a different notation for the array parameters, would be:

```
void addArray( register float *a1, register const float *a2, int len )
{ /* Function body as earlier. */ }
```

An advantage of declaring the parameters with brackets (`[]`) is that human readers immediately recognize that the function treats the arguments as pointers to an array, and not just to an individual `float` variable. But the array-style notation also has two peculiarities in parameter declarations :

- In a parameter declaration and only there C99 allows you to place any of the type qualifiers `const`, `volatile`, and `restrict` inside the square brackets. This ability allows you to declare the parameter as a qualified pointer type.
- Furthermore, in C99 you can also place the storage class specifier `static`, together with a integer constant expression, inside the square brackets. This approach indicates that the number of elements in the array at the time of the function call must be at least equal to the value of the constant expression.

Here is an example that combines both of these possibilities:

```
int func( long array[const static 5] )
{ /* ... */ }
```

In the function defined here, the parameter `array` is a constant pointer to `long`, and so cannot be modified. It points to the first of at least five array elements.

In Listing 0.5, the `maximum()` function's third parameter is a two-dimensional array of variable dimensions.

Listing 0.5. Function `maximum()`

```
// The function maximum( ) obtains the greatest value in a
// two-dimensional matrix of double values.
// Arguments:  The number of rows, the number of columns, and the matrix.
// Return value: The value of the greatest element.
```

```
double maximum( int nrows, int ncols, double matrix[nrows][ncols] )
{
    double max = matrix[0][0];
    for ( int r = 0; r < nrows; ++r )
        for ( int c = 0; c < ncols; ++c )
            if ( max < matrix[r][c] )
                max = matrix[r][c];
    return max;
}
```

The parameter `matrix` is a pointer to an array with `ncols` elements.

Pointers as Function Parameters

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. For instance, a sorting routine might exchange two out-of-order arguments with a function called `swap`. It is not enough to write

```
swap(a, b);
```

where the `swap` function is defined as

```
void swap(int x, int y) /* WRONG */
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

```
}
```

Because of call by value, `swap` can't affect the arguments `a` and `b` in the routine that called it. The function above swaps copies of `a` and `b`.

The way to obtain the desired effect is for the calling program to pass pointers to the values to be changed:

```
swap(&a, &b);
```

Since the operator `&` produces the address of a variable, `&a` is a pointer to `a`. In `swap` itself, the parameters are declared as pointers, and the operands are accessed indirectly through them.

```
void swap(int *px, int *py) /* interchange *px and *py */
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```

Pictorially in Figure 0.1:

in caller:

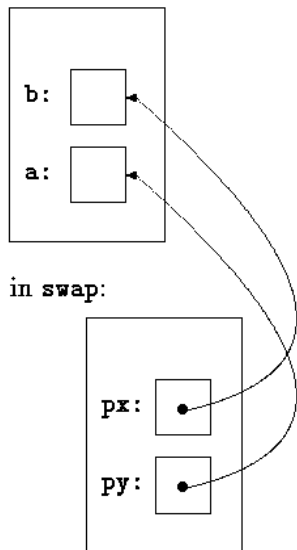


Figure 0.1. swap function with pointer parameters

Function Declarations

By declaring a function before using it, you inform the compiler of its type: in other words, a declaration describes a function's interface. A declaration must indicate at least the type of the function's return value, as the following example illustrates:

```
int rename( );
```

This line declares `rename()` as a function that returns a value with type `int`. Because function names are external identifiers by default, that declaration is equivalent to this one:

```
extern int rename( );
```

As it stands, this declaration does not include any information about the number and the types of the function's parameters. As a result, the compiler cannot test whether a given call to this function is correct. If you call the function with arguments that are different in number or type from the parameters in its definition, the result will be a critical runtime error. To prevent such errors, you should always declare a function's parameters as well. In other words, your declaration should be a function prototype. The prototype of the standard library function `rename()`, for example, which changes the name of a file, is as follows:

```
int rename( const char *oldname, const char *newname );
```

This function takes two arguments with type pointer to `const char`. In other words, the function uses the pointers only to read `char` objects. The arguments may thus be string literals.

The identifiers of the parameters in a prototype declaration are optional. If you include the names, their scope ends with the prototype itself. Because they have no meaning to the compiler, they are practically no more than comments telling programmers what each parameter's purpose is. In the prototype declaration of `rename()`, for example, the parameter names `oldname` and `newname` indicate that the old filename goes first and the new filename second in your `rename()` function calls. To the compiler, the prototype declaration would have exactly the same meaning without the parameter names:

```
int rename( const char *, const char * );
```

The prototypes of the standard library functions are contained in the standard header files. If you want to call the `rename()` function in your program, you can declare it by including the file `stdio.h` in your source code. Usually you will place the prototypes of functions you define yourself in a header file as well, so that you can use them in any source file simply by adding the appropriate include directive.

Declaring Optional Parameters

C allows you to define functions so that you can call them with a variable number of arguments. The best-known example of such a function is `printf()`, which has the following prototype:

```
int printf( const char *format, ... );
```

As this example shows, the list of parameters types ends with an *ellipsis* (...) after the last comma. The ellipsis represents optional arguments. The first argument in a `printf` function call must be a pointer to `char`. This argument may be followed by others. The prototype contains no information about what number or types of optional arguments the function expects.

Declaring Variable-Length Array Parameters

When you declare a function parameter as a variable-length array elsewhere than in the head of the function definition, you can use the asterisk character (*) to represent the array length specification. If you specify the array length using a nonconstant integer expression, the compiler will treat it the same as an asterisk. For example, all of the following declarations are permissible prototypes for the `maximum()` function defined in Listing 0.5:

```
double maximum( int nrows, int ncols, double matrix[nrows][ncols] );
double maximum( int nrows, int ncols, double matrix[ ][ncols] );
double maximum( int nrows, int ncols, double matrix[*][*] );
double maximum( int nrows, int ncols, double matrix[ ][*] );
```

How Functions Are Executed

The instruction to execute a functionthe function callconsists of the function's name and the operator (). For example, the following statement calls the function `maximum()` to compute the maximum of the matrix `mat`, which has `r` rows and `c` columns:

```
maximum( r, c, mat );
```

The program first allocates storage space for the parameters, then copies the argument values to the corresponding locations. Then the program jumps to the beginning of the function, and execution of the function begins with first variable definition or statement in the function block.

If the program reaches a `return` statement or the closing brace `}` of the function block, execution of the function ends, and the program jumps back to the calling function. If the program "falls off the end" of the function by reaching the closing brace, the value returned to the caller is undefined. For this reason, you must use a `return` statement to stop any function that does not have the type `void`. The value of the `return` expression is returned to the calling function.

Recursive Functions

A *recursive* function is one that calls itself, whether directly or indirectly. Indirect recursion means that a function calls another function (which may call a third function, and so on), which in turn calls the first function. Because a function cannot continue calling itself endlessly, recursive functions must always have an exit condition.

In Listing 0.6, the recursive function `binarySearch()` implements the binary search algorithm to find a specified element in a sorted array. First the function compares the search criterion with the middle element in the array. If they are the same, the function returns a pointer to the element found. If not, the function searches in whichever half of the array could contain the specified element by calling itself recursively. If the length of the array that remains to be searched reaches zero, then the specified element is not present, and the recursion is aborted.

Listing 0.6. Function `binarySearch()`

```
// The binarySearch() function searches a sorted array.
// Arguments:  The value of the element to find;
```

Do Van Uy - Nguyen Thi Thu Huong – Nguyen Khanh Phuong – Nguyen Thi Thu 101
Trang

Faculty of Information Technology – Hanoi University of Technology

```
//          the array of long to search; the array length.  
// Return value: A pointer to the element found,  
//          or NULL if the element is not present in the array.
```

```
long *binarySearch( long val, long array[ ], int n )  
{  
    int m = n/2;  
    if ( n <= 0 )      return NULL;  
    if ( val == array[m] ) return array + m;  
    if ( val < array[m] ) return binarySearch( val, array, m );  
    else              return binarySearch( val, array+m+1, n-m-1 );  
}
```

For an array of n elements, the binary search algorithm performs at most $1+\log_2(n)$ comparisons. With a million elements, the maximum number of comparisons performed is 20, which means at most 20 recursions of the `binarySearch()` function.

Recursive functions depend on the fact that a function's automatic variables are created anew on each recursive call. These variables, and the caller's address for the return jump, are stored on the stack with each recursion of the function that begins. It is up to the programmer to make sure that there is enough space available on the stack. The `binarySearch()` function as defined in Listing 0.6 does not place excessive demands on the stack size, though.

Recursive functions are a logical way to implement algorithms that are by nature recursive, such as the binary search technique, or navigation in tree structures. However, even when recursive functions offer an elegant and compact solution to a problem, simple solutions using loops are often possible as well. For example, you could rewrite the binary search in Listing 0.6 with a loop statement instead of a recursive function call. In such cases, the iterative solution is generally faster in execution than the recursive function.

Unit 5. Structures

- 5.1. Introduction
- 5.2. Declarations and Uses of Structures
- 5.3. Arrays of Structures
- 5.4. Operations on Structures
- 5.5. Enumerated Types

Upon successful completion of this Unit students will be able to:

- Understand the proper organisation of data into data types and structures;
- Develop structure data types;
- Initialise data structures;
- Declare and use variables of structure types and enumerated types, including arrays;
- Pass structures to functions by value and by reference;
- Develop enumerated types.

5.1. Introduction

A structure type can contain a number of dissimilar data objects within it. It is unlike a simple variable (which contains only one data object), and it is unlike an array (which, although it contains more than one data item, only contains items of a single data type). A structure is a collection of related data, but that data can be of different types. A name, for example, might be a char array and an age might be an int. A structure representing a person, say, could contain both a name and an age, each represented in the appropriate format.

5.2. Declarations and Uses of Structures

Until now, all the data that we have dealt with has been either of a *basic type* such as **char**, **int** and **double**..., or an *array* of those types. However, there are many situations in real life where a data item needs to be made up from other more basic types. We could do this with an array if the constituent types were all the same, but often they are different. For example, suppose we want to record the details of each student in a class. The detail of each student might be as follow:

- A unique student number, which could be represented as a string (an array of **char**).
- The student's name, which could be represented as a string (an array of **char**).
- Final mark for the Introduction to computer science course, which is a floating point value (a **float**).

Creating Structures as New Data Types

The definition of a structure type begins with the keyword **struct**, and contains a list of declarations of the structure's members, in braces:

```
struct structTag
{
    <list of members>;
};
```

Note: Definition ends with semicolon (;)

Example: The three components above can be placed in a structure declared like this:

```
struct Student
{
    char StudentID[10];
    char name[30];
    float markCS ;
};
```

The keyword **struct** introduces a structure declaration, which is a list of declarations enclosed in braces. An optional name called a structure tag may follow the word **struct** (as with **Student** here). The tag names this kind of structure, and can be used subsequently as a shorthand for the part of the declaration in braces.

The variables named in a structure are called members. A structure member or tag and an ordinary (i.e., non-member) variable can have the same name without conflict, since they

can always be distinguished by context. Furthermore, the same member names may occur in different structures, although as a matter of style one would normally use the same names only for closely related objects.

Creating variable of a struct type:

Structure types are not a variable declaration, just definition of a new type, so they cannot store anything until we declare variable of this type.

Here is how we would create:

type_name_of_struct name_of_variable;

Example:

Creating three variables *a*, *b*, *c* of the Student type:

Student *a*, *b*, *c*;

Creating an array of the Student type:

Student *studentCS*[50];

The members of a structure may have any desired complete type, including previously defined structure types. They must not be variable-length arrays, or pointers to such arrays. For instance, now we want to record more information of students, for example their date of birth, which comprises the day, month and year. So first, let's start with the date, because that is a new type that we may be able to use in a variety of situations. We can declare a new type for a **Date** thus:

```
struct Date
{
    int day;
    int month;
    int year;
};
```

We can now use this **Date** type, together with other types, as members of a **Student** type which we can declare as follows:

```
struct Student
{
    char studentID[10];
    char name[30];
    float markCS ;
    Date dateOfBirth;
};
```

Or:

```
struct Student
{
    char studentID[10];
    char name[30];
    float markCS;
    struct Date {
        int day;
        int month;
        int year;
    };
```



```
    } dateOfBirth;  
};
```

We can also declare structured variables when we define the structure itself:

```
struct Student  
{  
    char studentID[10];  
    char name[30];  
    float markCS ;  
    Date dateOfBirth;  
} a, b, c;
```

Interestingly, C permits we to declare untagged structures that enable us to declare structure variables without defining a name for their structures.

For example, the following structure definition declares the variables *a*, *b*, *c* but omits the name of the structure:

```
struct  
{  
    char studentID[10];  
    char name[30];  
    float markCS ;  
    Date dateOfBirth;  
} a, b, c;
```

A structure type cannot contain itself as a member, as its definition is not complete until the closing brace (}). However, structure types can and often do contain pointers to their own type. Such *self-referential structures* are used in implementing linked lists and binary trees, for example. The following example defines a type for the members of a singly linked list:

```
struct List  
{ struct Student stu;    // This record's data.  
  struct List *pNext;    // A pointer to the next student.  
};
```

Referencing Structure Members with the Dot Operator

Whenever we need to refer to the members of a structure, we normally use the dot operator.

For example, if we wanted to access the number member of **newStudent** we could do so as follows:

newStudent.studentID

We can then access the member as if it were a normal variable. For instance, we can write to this member as follows.

```
newStudent.studentID= "C0681008";
```

newStudent			
studentID	C0681008		
-			
-			
-			

We can also read from the member in a similar fashion.

```
printf("Student identification: %s", newStudent.studentID);
```

When a structure is nested in another structure, the dot operator can *cascade* to find the member of the nested structure.

Each **Student** structure contains a **dateOfBirth** structure. To write to the **day** member of this nested structure we can use the following statement.

```
newStudent.dateOfBirth.day = 1;
```

newStudent			
studentID	C0681008		
-			
-			
dateofBirth	1		
	Day	Month	Year

The following code outputs the contents of an **Student** structure.

```
printf("Student Details\n");
printf("Identification: %s\n", newStudent.studentID);
printf("Name: %s\n", newStudent.name);
printf("Mark: %.2f\n", newStudent.markCS);
printf("Date of Birth: %i/%i/%i\n",
    newStudent.dateOfBirth.day,
    newStudent.dateOfBirth.month,
    newStudent.dateOfBirth.year
);
```

Suppose we wish to input the details of this employee from a user. We could do so as follows.

```
Student newStudent;
printf("Enter student identification: ");
scanf("%s", &newStudent.studentID);
printf("Enter student name: ");
fflush(stdin);gets(newStudent.name);
printf("Enter mark for Introduction to computer science course: ");
scanf("%f", &newStudent.markCS);
printf("Enter birth date (dd/mm/yyyy): ");
scanf("%i/%i/%i",
    &newStudent.dateOfBirth.day,
    &newStudent.dateOfBirth.month,
    &newStudent.dateOfBirth.year
);
```

Initialising Structure Variables

When we declare a new variable of a basic data type we can initialise its value at declaration. We can also initialise structure variables at declaration as shown below.

```
Student newStudent = {  
    "C0681008",  
    "Cao Anh Huy",  
    8.50,  
    {1, 2, 1985}  
};
```

Notice how we include the initialisation values in curly brackets, just like when we initialise an array. Furthermore, we include the values for any nested structure type (in this case the **dateOfBirth** member is a nested structure), in a further set of curly brackets.

Copying Structure Variables

One of the most convenient features of structures is that we can copy them in a single assignment operation. This is unlike an array, which must be copied item-by-item. The name of a structure variable when it appears on its own represents the entire structure. If a structure contains an array as a member, that array *is* copied if the entire structure is copied.

```
Student newStudent1, newStudent2;  
// Get the values for newStudent2  
...  
// Copy newStudent2's value to newStudent1  
newStudent1 = newStudent2;
```

Comparing Values of Structures

We cannot compare structures in a single operation. If we wish to compare the values of two structure variables, we need to compare each of their members.

5.3. Arrays of Structures

Just as we can have an array of basic data types, we can also have an array of structures. Suppose that we created an array of **Student** structures as follows. We could then copy **newStudent** into each position in the array.

```
Student students[100];  
Student newStudent;  
int i;  
for (i=0; i<100; i++)  
{  
    // Get the values for newStudent  
    ...  
    // Copy into the next position in the array  
    students[i] = newStudent;  
}
```

5.4. Operations on Structures

Passing Structures to and from Functions

Structures can be passed to functions just like any other data type. Functions can also return structures, just as they can return any basic type. Structures can be also be passed to functions by reference.

Just like passing variable of a basic data type, when we pass a structure as an argument to a function, a copy is made of the entire structure. Structures are *passed by value*. We can easily take the code that output a student and put it into a function as follows.

```
void outputStudent(Student stu)
{
    printf("Student Details\n");
    printf("Identification: %s\n", stu.studentID);
    printf("Name: %s\n", stu.name);
    printf("Mark: %.2f\n", stu.markCS);
    printf("Date of Birth: %i/%i/%i\n",
           stu.dateOfBirth.day,
           stu.dateOfBirth.month,
           stu.dateOfBirth.year
    );
}
```

If we had an array of 100 students and wanted to output them this would be straightforward:

```
Student students[100];
int i;
...
for (i=0; i<100; i++) {
    outputStudent(students[i]);
}
```

We could similarly place the code to input a student into a function, but now we have a problem. The function can return a structure of type **Student** as follows.

```
Student inputStudent()
{
    Student tempStudent;
    printf("Enter Student identification: ");
    scanf("%s", &tempStudent.studentID);
    printf("Enter Student name: ");
    fflush(stdin); gets(tempStudent.name);
    printf("Enter final mark: ");
    scanf("%f", &tempStudent.markCS);
    printf("Enter birth date (dd/mm/yyyy):");
    scanf("%i/%i/%i",
           &tempStudent.dateOfBirth.day,
           &tempStudent.dateOfBirth.month,
           &tempStudent.dateOfBirth.year
    );
}
```

```
    );  
    return tempStudent;  
}
```

In the example above we are filling the structure variable **tempStudent** with values. At the end of the function, the value of **tempStudent** is returned as the return value of the function. The code to input 100 students can now be modified to use this function:

```
Student students[100];  
int i;  
for (i=0; i<100; i++) {  
    students[i] = inputStudent();  
}
```

The Arrow Operator

In order to dereference a pointer we would normally use the dereferencing operator (*) and if our pointer was to a structure, we could subsequently use the dot '.' operator to refer to a member of the structure. Suppose we have declared a pointer which could be used to point to a structure of type employee as follows.

```
Student stuVariable;  
Student *stuPtr;  
stuPtr = &stuVariable;
```

To refer to the student identification we could say:

```
(*stuPtr).studentID
```

Note that the brackets are necessary because the dereference operator has lower precedence than the dot operator. This form of syntax is a little cumbersome, so another operator is provided to us as a convenient shorthand:

```
stuPtr->studentID
```

This method of accessing the number member through a pointer is completely equivalent to the previous form. The '->' operator is called the *indirect member selection* operator or just the *arrow* operator and it is almost always used in preference to the previous form.

Passing Structures by Reference

Passing structures to a function using *pass-by-value* can be simple and successful for simple structures so long as we do not wish to do so repeatedly. But when structures can contain a large amount of data (therefore occupying a large chunk of memory) then creating a new copy to pass to a function can create a burden on the memory of the computer. If we were going to do this repeatedly (say several thousand times within the running of a computer) then there would also be a cost in time to copy the structure for each function call.

In the example at the beginning of this section we created and filled a structure variable called **tempStudent**. When the function ended it returned the value of **tempStudent**. The same inefficiency exists with the return value from the function, where the **Student** structure must be copied to a local variable at the function call.

Whether such inefficiencies are of any significance or not depends on the circumstances and on the size of the structure. Each **Student** structure probably occupies about 50 bytes, so this is a reasonably significant amount of memory to be copying each time the output function is called or each time the input function returns, especially if this is happening frequently.

A better solution would be to pass the **Student** structure *by reference*, which means we will pass a pointer to the structure.

We can now revise the input function by passing an **Student** structure by reference using a pointer. Because the function is no longer returning an **Student** structure, we can also enhance the function to return a Boolean status indicating whether an **Student** structure was successfully read or not. We can enhance our function to do some better error checking. Below is the revised version.

```
bool inputStudent(Student *stuPtr)
{
    printf("Enter Student identification: ");
    if (scanf("%s", &stuPtr->studentID) != 1) return false;
    printf("Enter Student name: ");
    fflush(stdin); gets(stuPtr->name);
    printf("Enter mark: ");
    if (scanf("%f", &stuPtr->markCS) != 1) return false;
    printf("Enter birth date: ");
    if (scanf("%i/%i/%i", &stuPtr->dateOfBirth.day,
        &stuPtr->dateOfBirth.month, &stuPtr->dateOfBirth.year) != 3)
        return false;
    return true;
}
```

The code to input 100 students can now be revised as follows.

```
Student students[100];
int i;
for (i=0; i<100; i++)
{
    while (!inputStudent(&students[i]))
    {
        printf("Invalid student details - try again!\n");
        fflush(stdin);
    }
}
```

As a final example, consider a function to give a student a mark rise. The function takes two parameters. The first is an **Student** structure passed by reference, (a pointer to an **Student** structure) and the second is the increase of mark.

```
void markRise(Student *stuPtr, float increase)
{
    stuPtr->markCS += increase;
}
```

```
}
```

What use is such a function? Having input many students into an array, we might then wish to give certain students a mark rise. For each student we can easily call this function, passing a pointer to the appropriate **Student** structure.

5.5. Enumerated Types

Another way of creating a new type is by creating an enumerated type. With an enumerated type we build a new type from scratch by stating which values are in the type. The syntax for an enumerated type is as follows.

```
enum TypeIdentifier { list... };
```

Here is an example of a definition of an enumerated type that can be used to refer to the days of the week.

```
enum DayOfWeek {sun, mon, tue, wed, thu, fri, sat};
```

Just like when we define a structure type, defining an enumerated type does not give us any space to store information. We use the type like a template to create variables of that type.

For instance we can create a variable of type **DayOfWeek** as follows.

```
DayOfWeek nameOfDay;
```

With variables of enumerated types we can do almost anything we could do with a variable of a basic data type. For instance we can assign a value as follows.

```
nameOfDay = tue;
```

Note that **tue** is a literal value of type **DayOfWeek** and we do not need to place quotes around it.

The values in **DayOfWeek** are ordered and each has an equivalent **int** value; **sun==0**, **mon==1**, and so on. The value of **sun** is less than the value of **wed** because of the order they were presented in the list of values when defining the type. We can compare two values of enumerated types as follows.

```
DayOfWeek day1, day2;
// Get day values
...
if(day1 < day2) {
...
}
```

Here is another example that uses enumerated types.

```
#include <stdio.h>
#include <conio.h>
enum TrafficLight {red, orange, green};
int main()
{
    TrafficLight light;
```

```
printf("Please enter a Light Value: (0)Red (1)Orange (2)Green:\n");
scanf("%i", &light);
switch(light)
{
case red:
    printf("Stop!\n");
    break;
case orange:
    printf("Slow Down\n");
    break;
case green:
    printf("Go\n");
}
getch();
}
```

Quiz

1. What's wrong with the following structure declaration?

```
struct automobile
{
    int year;
    char model[8];
    int engine_power;
    float weight;
}
```

2. How many structure variables are defined in the following statement?

```
struct x {int y; char z} u, v, w;
```

3. Given a structure declaration

```
struct automobile
{
    int year;
    char model[8]
};
```

and two car models, Taurus and Accord, which are made in 1997, initialize an array of two elements, car, that is defined with the automobile structure data type.

4. In the following structure declarations, which one is a forward-referencing structure, and which one is a self-referencing structure? (Assume that the structures, employment and education, have not been declared yet.)

```
struct member
{
    char name[32];
```



```
    struct employment *emp;
    struct education *edu;
};

struct list
{
    int x, y;
    float z;
    struct list *ptr_list;
};
```

Unit 6. Files

- 6.1. Basics and Classification of Files
- 6.2. Operations on Files
 - 6.2.1. Declarations
 - 6.2.2. Open Files
 - 6.2.3. Access Text Files
 - 6.2.4. Access Binary Files
 - 6.2.5. Close Files

Upon successful completion of this module students will be able to:

- Appreciate the importance of storing data in a file,
- Explain what text files and binary files are and how to access them,
- Demonstrate the use of file pointers in opening and closing files, and
- Demonstrate the ability to create a text/binary files, write data to the file and read data from the file.

6.1. Basics and Classification of Files

When reading input from the keyboard and writing output to the monitor you have been using a special case of file I/O (input/output). You already know how to read and write text data, as you have been doing it every time you use **scanf()** and **printf()**. All you need to do now is learn how to direct I/O to file other than from your keyboard or to your monitor.

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

a machine decipherable storage media where programs and data are stored for machine usage.

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

- **Text files** are any files that contain only ASCII characters. Examples include C source code files, HTML files, and any file that can be viewed using a simple text editor.
- **Binary files** are any files that created by writing on it from a C-program, not by an editor (as with text files). Binary files are very similar to arrays of records, except the records are in a disk file rather than in an array in memory. Because the records in a binary file are on disk, you can create very large collections of them (limited only by your available disk space). They are also permanent and always available. The only disadvantage is the slowness that comes from disk access time.

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

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

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

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

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

For example, a database file will be created and processed as a binary file. A record update operation will involve locating the appropriate record, reading the record into memory, modifying it in some way, and finally writing the record back to disk at its

appropriate location in the file. These kinds of operations are common to many binary files, but are rarely found in applications that process text files.

For all file operations you should always follow the 5-step plan as outlined below.

1. Declare file pointer.
2. Attach the file pointer to the file (open file).
3. Check file opened correctly.
4. Read or Write the data from or to the file.
5. Close the file.

6.2. Operations on Files

6.2.1. Declarations

In C, we usually create variables of type **FILE *** to point to a file located on the computer.

```
FILE *file_pointer_name;
```

Example:

```
FILE * f1, * f2;
```

6.2.2. Open Files

First things first: we have to open a file to be able to do anything else with it. For this, we use **fopen** function, like all the I/O functions, is made available by the **stdio.h** library. The **fopen()** function prototype is as follows.

```
FILE *fopen(char *filename, char *mode);
```

In the above prototype, there are two arguments:

- **filename** is a string containing the name of the file to be opened. So if your file sits in the same directory as your C source file, you can simply enter the filename in here - this is probably the one you'll use most.
- **mode** determines how the file may be accessed.

Mode	Meaning
"r"	Open a file for read only, starts at beginning of file (default mode).
"w"	Write-only, truncates existing file to zero length or create a new file for writing.
"a"	Write-only, starts at end of file if file exists, otherwise creates a new file for writing.
"r+"	Open a file for read-write, starts at beginning of file. If the file is not exist, it will cause an error.
"w+"	Read-write, truncates existing file to zero length or creates a new file for

reading and writing.

“a+” Read-write, starts at end of file if file exists, otherwise creates a new file for reading and writing.

So there are 12 different values that could be used: "rt", "wt", "at", "r+t", "w+t", "a+t" and "rb", "wb", "ab", "r+b", "w+b", "a+b".

Characte	Type
r	
“t”	Text File
“b”	Binary File

Note: When work with the text file, you also can use only "r", "w", "a", "r", "w", "a", instead of "rt", "wt", "at", "r+t", "w+t", "a+t" respectively.

Example:

```
FILE *f1, *f2, *f3, *f4;
```

To open text file *c:\abc.txt* for ready only:

```
f1 = fopen("c:\\abc.txt", "r");
```

To open text file *c:\list.dat* for write only:

```
f2 = fopen("c:\\list.dat", "w");
```

To open text file *c:\abc.txt* for read-write:

```
f3 = fopen("c:\\abc.txt", "r+");
```

To open binary file *c:\liststudent.dat* for write only:

```
f4 = fopen("c:\\liststudent.dat", "wb");
```

The **file pointer** will be used with all other functions that operate on the file and it must never be altered or the object it points to.

File checking

```
if (file_pointer_name == NULL)
{
    printf("Error opening file.");
    <Action for error >
}
else
{
    <Action for success>
}
```

Before using an input/output file it is worth checking that the file has been correctly opened first. A call to **fopen()** may result in an error due to a number of reasons including:

- A file opened for reading does not exist;
- A file opened for reading is read protected;

- A file is being opened for writing in a folder or directory where you do not have write access.

If the operation is successful, **fopen()** returns an address which can be used as a stream. If a file is not successfully opened, the value **NULL** is returned. An error opening a file can occur if the file was to be opened for reading and did not exist, or a file opened for writing could not be created due to lack of disk space. It is important to always check that the file has opened correctly before proceeding in the program.

Example:

```
FILE *fp;  
if ((fp = fopen("myfile", "r")) == NULL){  
    printf("Error opening file\n");  
    exit(1);  
}
```

Once a file has been opened, depending upon its mode, you may read and/or write bytes to or from it.

6.2.3. Access Text Files

Write data to text files

When writing data to text files, C provides three functions: **fprintf()**, **fputs()**, **fputc()**. The **fprintf()** function prototype is as follows.

```
int fprintf(FILE *fp, char *format, ...);
```

This function writes to the file specified by file pointer *fp* a sequence of data formatted as the *format* argument specifies. After the *format* parameter, the function expects at least as many additional arguments as specified in *format*. Depending on the *format* string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each %-tag specified in the *format* parameter, if any. There should be the same number of these arguments as the number of %-tags that expect a value.

Return value: On success, the total number of characters written is returned. On failure, a negative number is returned.

Example:

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

```
int main ()  
{  
    FILE * fp;  
    int n;  
    char name [50];  
  
    fp = fopen ("myfile.txt","w");  
    for (n=0 ; n<3 ; n++)
```

```
{
    puts ("Please, enter a name: ");
    gets (name);
    fprintf (fp, "Name %d [%-10.10s]\n",n,name);
}
fclose (fp);
return 0;
}
```

This example prompts 3 times the user for a name and then writes them to myfile.txt each one in a line with a fixed length (a total of 19 characters + newline). Two format tags are used: %d : signed decimal integer, %-10.10s : left aligned (-), minimum of ten characters (10), maximum of ten characters (.10), String (s).

Assuming that we have entered John, Jean-Francois and Yoko as the 3 names, myfile.txt would contain:

myfile.txt
Name 1 [John]
Name 2 [Jean-Franc]
Name 3 [Yoko]

The **fputc()** function prototype is as follows.

int fputc(int character, FILE *fp);

The **fputc()** function writes a character to the file associated with *fp*. The character is written at the current position of the *fp* as indicated by the internal position indicator, which is then advanced one character. Return value: If there are no errors, the same character that has been written is returned. If an error occurs, **EOF** is returned and the error indicator is set.

Example: Write the program that creates a file called alphabet.txt and writes ABCDEFGHIJKLMNOPQRSTUVWXYZ to it.

```
#include <stdio.h>
int main ()
{
    FILE * fp;
    char c;

    fp = fopen ("alphabet.txt","w");
    if (fp!=NULL)
    {
        for (c = 'A' ; c <= 'Z' ; c++)
        {
            fputc ((int) c , fp);
        }
        fclose (fp);
    }
    return 0;
}
```

The **fputs()** function prototype is as follows.

```
int fputs(char *str, FILE *fp);
```

The **fputs()** function writes the string pointed to by *str* to the file associated with *fp*.

Return value: On success, a non-negative value is returned. On error, the function returns [EOF](#). The null that terminates *str* is not written and it does not automatically append a carriage return/linefeed sequence.

Example: Write the program allows to append a line to a file called myfile.txt each time it is run.

```
#include <stdio.h>  
int main ()  
{  
    FILE * fp;  
    char name [50];  
  
    puts ("Please, enter a name: ");  
    gets (name);  
    fp = fopen ("myfile.txt", "a");  
    fputs (name, fp);  
    fclose (fp);  
    return 0;  
}
```

Read data from text files

When reading data from text files, C provides three functions: **fscanf()**, **fgetc()**, **fgets()**. The **fscanf()** function prototype is as follows.

```
int fscanf(FILE *fp, char *format, ...);
```

This function reads data from the file specified by file pointer *fp* and stores them according to the parameter *format* into the locations pointed by the additional arguments. The additional arguments should point to already allocated objects of the type specified by their corresponding format tag within the *format* string.

Return value: On success, the function returns the number of items successfully read. This count can match the expected number of readings or be less -even zero- in the case of a matching failure. In the case of an input failure before any data could be successfully read, [EOF](#) is returned.

Example: Read an integer number and a character from file associated with a file pointer *fp* and stores them to two variables *a* and *c*.

```
fscanf(fp, "%d %c", &a, &c);
```

Example:

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

```
char str [80];
float f;
FILE * fp;

fp = fopen ("myfile.txt","w+");
fprintf (fp, "%f %s", 3.1416, "PI");
rewind (fp);
fscanf (fp, "%f", &f);
fscanf (fp, "%s", str);
fclose (fp);
printf ("I have read: %f and %s \n",f,str);
return 0;
}
```

This sample code creates a file called myfile.txt and writes a float number and a string to it. Then, the stream is rewinded and both values are read with fscanf. It finally produces an output similar to:

I have read: 3.141600 and PI

feof() function

```
int feof(FILE *fp);
```

This function check if End-of-File indicator associated with *fp* is set

Return value: A non-zero value is returned in the case that the End-of-File indicator associated with the *fp* is set. Otherwise, a zero value is returned.

Example: Create a text file called **fscanf.txt** in Notepad with this content:

```
0      1      2      3      4
5      6      7      8      9
10     11     12     13
```

Remember how **scanf** stops reading input when it encounters a space, line break or tab character? **fscanf** is just the same. So if all goes to plan, this example should open the file, read all the numbers and print them out:

```
#include <stdio.h>
int main() {
    FILE *fp;
    int numbers[30];
    /* make sure it is large enough to hold all the data! */
    int i,j;

    fp = fopen("fscanf.txt", "r");

    if(fp==NULL) {
```



```
printf("Error: can't open file.\n");
return 1;
}
else {
printf("File opened successfully.\n");

i = 0 ;

while(!feof(fp)) {
    /* loop through and store the numbers into the array */
    fscanf(fp, "%d", &numbers[i]);
    i++;
}

printf("Number of numbers read: %d\n\n", i);
printf("The numbers are:\n");

for(j=0 ; j<i ; j++) { /* now print them out one by one */
    printf("%d\n", numbers[j]);
}

fclose(fp);
return 0;
}
}
```

***fflush()* function**

Same as `scanf()`, before using `fscanf()` to read the character or string from the file, we need use `fflush()`. The **`fflush()`** function prototype is as follows.

`int fflush(FILE *fp)`

If the given file that specified by *fp* was open for writing and the last I/O operation was an output operation, any unwritten data in the output buffer is written to the file. If the file was open for reading, the behavior depends on the specific implementation. In some implementations this causes the input buffer to be cleared. If the argument is a null pointer, all open files are flushed. The files remains open after this call. When a file is closed, either because of a call to [fclose](#) or because the program terminates, all the buffers associated with it are automatically flushed.

Return Value: A zero value indicates success. If an error occurs, [EOF](#) is returned and the error indicator is set (see [feof](#)).

The ***fgetc()*** function prototype is as follows.

`int fgetc(FILE *fp);`

This function returns the character currently pointed by the internal file position indicator of the specified *fp*. The internal file position indicator is then advanced by one character to point to the next character.

Return value: The character read is returned as an `int` value. If the EOF is reached or a reading error happens, the function returns `EOF` and the corresponding error or eof indicator is set. You can use either `ferror` or `feof` to determine whether an error happened or the EOF was reached.

Example: Write the program reads an existing file called `myfile.txt` character by character and uses the `n` variable to count how many dollar characters (\$) does the file contain.

```
#include <stdio.h>
int main ()
{
    FILE * fp;
    int c;
    int n = 0;
    fp=fopen ("myfile.txt","r");
    if (fp==NULL) printf("Error opening file");
    else
    {
        do {
            c = fgetc (fp);
            if (c == '$') n++;
        } while (c != EOF);
        fclose (fp);
        printf ("File contains %d$.\n",n);
    }
    return 0;
}
```

Example: Write the program opens the file called `myfile.txt`, and counts the number of characters that it contains by reading all of them one by one. Finally the total amount of bytes is printed out.

```
#include <stdio.h>
int main ()
{
    FILE * fp;
    long n = 0;
    fp = fopen ("myfile.txt","rb");
    if (fp==NULL) printf ("Error opening file");
    else
    {
        while (!feof(fp)) {
            fgetc (fp);
            n++;
        }
        fclose (fp);
        printf ("Total number of bytes: %d\n",n);
    }
}
```

```
    return 0;
}
```

Example: Opens a file called input.txt which has some random text (less than 200 characters), stores each character in an array, then spits them back out into another file called "output.txt" in reverse order:

```
#include <stdio.h>
int main() {
    char c;      /* declare a char variable */
    char name[200]; /* Initialise array of total
                    200 for characters */
    FILE *f_input, *f_output; /* declare FILE pointers */
    int counter = 0; /* Initialise variable for counter to zero */

    f_input = fopen("input.txt", "r");
    /* open a text file for reading */

    if(f_input==NULL) {
        printf("Error: can't open file.\n");
        return 1;
    }
    else {
        while(1) { /* loop continuously */
            c = fgetc(f_input); /* fetch the next character */
            if(c==EOF) {
                /* if end of file reached, break out of loop */
                break;
            }
            else if (counter<200) { /* else put character into array */
                name[counter] = c;
                counter++; /* increment the counter */
            }
            else {
                break;
            }
        }

        fclose(f_input); /* close input file */

        f_output = fopen("output.txt", "w");
        /* create a text file for writing */

        if(f_output==NULL) {
            printf("Error: can't create file.\n");
            return 1;
        }
        else {
```

```
    counter--; /* we went one too step far */
    while(counter >= 0) { /* loop while counter's above zero */
        fputc(name[counter], f_output);
        /* write character into output file */
        counter--; /* decrease counter */
    }

    fclose(f_output); /* close output file */
    printf("All done!\n");
    return 0;
}
}
```

Reading one character at a time can be a little inefficient, so we can use **fgets** to read one line at a time. The **fgets()** function prototype is as follows.

char *fgets(char *str, int num, FILE *fp);

The **fgets()** function reads characters from the file associated with *fp* into a string pointed to by *str* until *num*-1 characters have been read, a newline character is encountered, or the end of the file is reached. The string is null-terminated and the newline character is retained.

Return value: the function returns *str* if successful and a **null pointer** if an error occurs.

You can't use an **!=EOF** check here, as we're not reading one character at a time (but you can use **feof**).

Example: Create a file called **myfile.txt** in Notepad, include 3 lines and put tabs in the last line.

```
111 222 333
444 555 666
777    888    999
```

```
#include <stdio.h>
int main()
{
    char c[10]; /* declare a char array */
    FILE *file; /* declare a FILE pointer */

    file = fopen("myfile.txt", "r");
    /* open a text file for reading */

    if(file==NULL)
    {
        printf("Error: can't open file.\n");
        /* fclose(file); DON'T PASS A NULL POINTER TO fclose !! */
    }
}
```

```
    return 1;
}
else
{
    printf("File opened successfully. Contents:\n\n");

    while(fgets(c, 10, file)!=NULL) {
        /* keep looping until NULL pointer... */
        printf("String: %s", c);
        /* print the file one line at a time */
    }

    printf("\n\nNow closing file...\n");
    fclose(file);
    return 0;
}
}
```

Output:

```
File opened successfully. Contents:

String: 111 222 3String: 33
String: 444 555 6String: 66
String: 777 888 9String: 99

Now closing file...
```

The main area of focus is the while loop - notice how I performed the check for the return of a NULL pointer. Remember that passing in `char *` variable, `c` as the first argument assigns the line read into `c`, which is printed off by `printf`. We specified a maximum number of characters to be 10 - we knew the number of characters per line in our text file is more than this, but we wanted to show that **`fgets`** reads 10 characters at a time in this case.

Notice how **`fgets`** returns when the newline character is reached - this would explain why **444** and **777** follow the word "String". Also, the tab character, `\t`, is treated as one character.

Other function:

`fseek()` function

```
int fseek (FILE *fp, long int offset, int origin);
```

In the above prototype, there are two arguments:

- **fp**: Pointer to a [FILE](#) object that identifies the stream.
- **offset**: Number of bytes to offset from *origin*.
 - If offset ≥ 0 : set the position indicator toward to the end of file,
 - If offset < 0 : set the position indicator toward to the beginning of file.
- **origin**: Position from where *offset* is added. It is specified by one of the following constants defined in `<stdio.h>`:

Constant	Value	Meaning
SEEK_SET	0	Beginning of file
SEEK_CUR	1	Current position of the file pointer
SEEK_END	2	End of file

This function sets the position indicator associated with the *fp* to a new position defined by adding *offset* to a reference position specified by *origin*. The End-of-File internal indicator of the file is cleared after a call to this function.

Return Value: If successful, the function returns a zero value. Otherwise, it returns nonzero value.

Example:

```
#include <stdio.h>
int main ()
{
    FILE * fp;
    fp = fopen ( "myfile.txt" , "w" );
    fputs ( "This is an apple." , fp );
    fseek ( fp , -8 , SEEK_END );
    fputs ( " sam" , fp );
    fclose ( fp );
    return 0;
}
```

After this code is successfully executed, the file `myfile.txt` contains:

This is a sample.

Example:

```
#include <stdio.h>
int main ()
{
    FILE * fp;
    fp = fopen ( "myfile.txt" , "w" );
    fputs ( "This is an apple." , fp );
    fseek ( fp , 9 , SEEK_SET );
    fputs ( " sam" , fp );
    fclose ( fp );
}
```

```
    return 0;
}
```

After this code is successfully executed, the file myfile.txt contains:

This is a sample.

rewind() function

```
void rewind (FILE *fp);
```

This function sets the current position indicator associated with *fp* to the beginning of the file. A call to *rewind* is equivalent to:

```
fseek (fp, 0, SEEK_SET);
```

except that, unlike **fseek**, *rewind* clears the error indicator.

On streams open for update (read+write), a call to *rewind* allows to switch between reading and writing.

Example:

```
#include <stdio.h>
#include <conio.h>
int main ()
{
    char str [80];
    int n;
    FILE * fp;
    fp = fopen ("myfile.txt","w+");
    for ( n='A' ; n<='Z' ; n++)
        fputc ( n, fp);
    rewind (fp);
    n=0;
    while (!feof(fp))
    {
        str[n]= fgetc(fp);
        n++;
    }
    fclose (fp);
    printf ("I have read: %s\n",str);
    getch();
    return 0;
}
```

A file called myfile.txt is created for reading and writing and filled with the alphabet. The file is then rewinded, read and its content is stored in a buffer, that then is written to the standard output:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

Example:

```
#include <stdio.h>
int main()
{
    FILE *file;
```

```
char sentence[50];
int i;

file = fopen("sentence.txt", "w+");
/* we create a file for reading and writing */

if(file==NULL) {
    printf("Error: can't create file.\n");
    return 1;
}
else {
    printf("File created successfully.\n");

    printf("Enter a sentence less than 50 characters: ");
    gets(sentence);

    for(i=0 ; sentence[i] ; i++) {
        fputc(sentence[i], file);
    }

    rewind(file); /* reset the file pointer's position */

    printf("Contents of the file: \n\n");

    while(!feof(file)) {
        printf("%c", fgetc(file));
    }

    printf("\n");
    fclose(file);
    return 0;
}
}
```

Output depends on what you entered. First of all, we stored the inputted sentence in a char array, since we're writing to a file one character at a time it'd be useful to detect for the null character. Recall that the null character, `\0`, returns 0, so putting `sentence[i]` in the condition part of the for loop iterates until the null character is met.

Then we call `rewind`, which takes the file pointer to the beginning of the file, so we can read from it. In the while loop we print the contents a character at a time, until we reach the end of the file - determined by using the `feof` function.

Note that it is essential to have the include file `stdio.h` referenced at the top of your program in order to use any of these functions: `fscanf()`, `fgets()`, `fgetc()`, `fflush()`, `fprintf()`, `fputs()`, `fputc()`, `feof()`, `fseek()` và `rewind()`.

EOF and errors

When a function returns **EOF** (or, occasionally, 0 or **NULL**, as in the case of **fread** and **fgets** respectively), we commonly say that we have reached “end of file” but it turns out that it's also possible that there's been some kind of I/O error. When you want to distinguish between end-of-file and error, you can do so with the **feof** and **ferror** functions. **feof(fp)** returns nonzero (that is, “true”) if end-of-file has been reached on the file pointer **fp**, and **ferror(fp)** returns nonzero if there has been an error.

Notice **feof** returns nonzero if end-of-file *has been reached*. It does *not* tell you that the next attempt to read from the stream will reach end-of-file, but rather that the previous attempt (by some other function) already *did*. (If you know Pascal, you may notice that the end-of-file detection situation in C is therefore quite different from Pascal.) Therefore, you would never write a loop like

```
while(!feof(fp))
    fgets(line, max, fp);
```

Instead, check the return value of the input function directly:

```
while(fgets(line, max, fp) != NULL)
```

With a very few possible exceptions, you don't use **feof** to *detect* end-of-file; you use **feof** or **ferror** to distinguish between end-of-file and error. (You can also use **ferror** to diagnose error conditions on output files.)

Since the end-of-file and error conditions tend to persist on a stream, it's sometimes necessary to clear (reset) them, which you can do with **clearerr(FILE *fp)**.

What should your program do if it detects an I/O error? Certainly, it cannot continue as usual; usually, it will print an error message. The simplest error messages are of the form

```
fp = fopen(filename, "r");
if(fp == NULL)
{
    fprintf(stderr, "can't open file\n");
    return;
}
```

or

```
while(fgets(line, max, fp) != NULL)
{
    ... process input ...
}
```

```
if(ferror(fp))
    fprintf(stderr, "error reading input\n");
```

or

```
fprintf(fp, "%d %d %d\n", a, b, c);
if(ferror(fp))
    fprintf(stderr, "output write error\n");
```

Error messages are much more useful, however, if they include a bit more information, such as the name of the file for which the operation is failing, and if possible *why* it is failing. For example, here is a more polite way to report that a file could not be opened:

```
#include <stdio.h> /* for fopen */
#include <errno.h> /* for errno */
#include <string.h> /* for strerror */
```

```
fp = fopen(filename, "r");
if(fp == NULL)
{
    fprintf(stderr, "can't open %s for reading: %s\n",
              filename, strerror(errno));
    return;
}
```

errno is a global variable, declared in `<errno.h>`, which may contain a numeric code indicating the reason for a recent system-related error such as inability to open a file. The **strerror** function takes an *errno* code and returns a human-readable string such as “No such file” or “Permission denied”.

An even more useful error message, especially for a “toolkit” program intended to be used in conjunction with other programs, would include in the message text the name of the program reporting the error.

6.2.4. Access Binary Files

Write data to binary files

```
size_t fwrite(void *buf, size_t sz, size_t n, FILE *fp)
```

This function writes to file associated with *fp*, *num* number of objects, each object size bytes long, from the buffer pointed to by *buffer*.

Return value: It returns the number of objects written. This value will be less than *num* only if an output error as occurred.

The **void pointer** is a pointer that can point to any type of data without the use of a TYPE cast (known as a generic pointer). The type *size_t* is a variable that is able to hold a value equal to the size of the largest object supported by the compiler.

As a simple example, this program write an integer value to a file called MYFILE using its internal, binary representation.

```
#include <stdio.h> /* header file */
#include <stdlib.h>
void main(void)
{
    FILE *fp; /* file pointer */
    int i;

    /* open file for output */
    if ((fp = fopen("myfile", "w"))==NULL){
        printf("Cannot open file \n");
        exit(1);
    }
    i=100;

    if (fwrite(&i, 2, 1, fp) !=1){
        printf("Write error occurred");
        exit(1);
    }
    fclose(fp);
}
```

}

Read data from binary files

size_t fread(void *buf, size_t sz, size_t n, FILE *fp)

fread reads up to *n* objects, each of size *sz*, from the file specified by *fp*, and copies them to the buffer pointed to by *buf*. It reads them as a stream of bytes, without doing any particular formatting or other interpretation. (However, the default underlying stdio machinery may still translate newline characters unless the stream is open in binary or "b" mode).

Return value: returns the number of items read. It returns 0 (not EOF) at end-of-file.

Example:

```
#include <stdio.h>
int main() {
    FILE *file;
    char c[30]; /* make sure it is large enough to hold all the data! */
    char *d;
    int n;

    file = fopen("numbers.txt", "r");

    if(file==NULL) {
        printf("Error: can't open file.\n");
        return 1;
    }
    else {
        printf("File opened successfully.\n");


        n = fread(c, 1, 10, file); /* passing a char array,
                                   reading 10 characters */
        c[n] = '\0'; /* a char array is only a
                       string if it has the
                       null character at the end */
        printf("%s\n", c); /* print out the string */
        printf("Characters read: %d\n\n", n);

        fclose(file); /* to read the file from the beginning, */
                       /* we need to close and reopen the file */
        file = fopen("numbers.txt", "r");

        n = fread(d, 1, 10, file);
        /* passing a char pointer this time - 10 is irrelevant */
        printf("%s\n", d);
        printf("Characters read: %d\n\n", n);
    }
}
```

```
    fclose(file);  
    return 0;  
}  
}
```

Output:



```
File opened successfully.  
111  
222  
33  
Characters read: 10  
  
111  
222  
333  
  
444  
5ive  
Characters read: 10
```

The above code: passing a char pointer reads in the entire text file, as demonstrated. Note that the number `fread` returns in the char pointer case is clearly incorrect. This is because the char pointer (`d` in the example) must be initialized to point to something first.

An important line is: `c[n] = '\0'`; Previously, we put 10 instead of n (n is the number of characters read). The problem with this was if the text file contained less than 10 characters, the program would put the null character at a point past the end of the file.

There are several things you could try with this program:

- After reading the memory allocation section, try allocating memory for `d` using `malloc()` and freeing it later with `free()`.
- Read 25 characters instead of 10: `n = fread(c, 1, 25, file);`
- Not bother adding a null character by removing: `c[n] = '\0'`;
- Not bother closing and reopening the file by removing the `fclose` and `fopen` after printing the char array.

Binary files have two features that distinguish them from text files: You can jump instantly to any record in the file, which provides random access as in an array; and you can change the contents of a record anywhere in the file at any time. Binary files also usually have faster read and write times than text files, because a binary image of the record is stored directly from memory to disk (or vice versa). In a text file, everything has to be converted back and forth to text, and this takes time.

Pascal supports the file-of-records concept very cleanly. You declare a variable such as **`var f:file of rec;`** and then open the file. At that point, you can read a record, write a record, or seek to any record in the file. This file structure supports the concept of a *file pointer*. When the file is opened, the pointer points to record 0 (the first record in the file). Any read operation reads the currently pointed-to record and moves the pointer down one record. Any write operation writes to the currently pointed-to record and moves the pointer down one record. Seek moves the pointer to the requested record.

In C, the concepts are exactly the same but less concise. Keep in mind that C thinks of everything in the disk file as blocks of bytes read from disk into memory or read from memory onto disk. C uses a file pointer, but it can point to any byte location in the file.

The following program illustrates these concepts:

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

```
/* random record description - could be anything */

struct rec
{
    int x,y,z;
};

/* writes and then reads 10 arbitrary records from the file "junk". */
void main()
{
    int i,j;
    FILE *f;
    struct rec r;

    /* create the file of 10 records */
    f=fopen("junk","w");
    for (i=1;i<=10; i++)
    {
        r.x=i;
        fwrite(&r,sizeof(struct rec),1,f);
    }
    fclose(f);

    /* read the 10 records */
    f=fopen("junk","r");
    for (i=1;i<=10; i++)
    {
        fread(&r,sizeof(struct rec),1,f);
        printf("%d\n",r.x);
    }
    fclose(f);

    printf("\n");

    /* use fseek to read the 10 records in reverse order */
    f=fopen("junk","r");
    for (i=9; i>=0; i--)
    {
        fseek(f,sizeof(struct rec)*i,SEEK_SET);
        fread(&r,sizeof(struct rec),1,f);
        printf("%d\n",r.x);
    }
    fclose(f);

    printf("\n");
```

```
/* use fseek to read every other record */
f=fopen("junk","r");
fseek(f,0,SEEK_SET);
for (i=0;i<5; i++)
{
    fread(&r,sizeof(struct rec),1,f);
    printf("%d\n",r.x);
    fseek(f,sizeof(struct rec),SEEK_CUR);
}
fclose(f);

printf("\n");

/* use fseek to read 4th record, change it, and write it back */
f=fopen("junk","r+");
fseek(f,sizeof(struct rec)*3,SEEK_SET);
fread(&r,sizeof(struct rec),1,f);
r.x=100;
fseek(f,sizeof(struct rec)*3,SEEK_SET);
fwrite(&r,sizeof(struct rec),1,f);
fclose(f);

printf("\n");

/* read the 10 records to insure 4th record was changed */
f=fopen("junk","r");
for (i=1;i<=10; i++)
{
    fread(&r,sizeof(struct rec),1,f);
    /* read 12 bytes (the size of rec) from the file f (from the current location
    of the file pointer) into memory address &r. */
    printf("%d\n",r.x);
}
fclose(f);
}

#include <stdio.h>

int main() {
    FILE *sourceFile;
    FILE *destinationFile;
    char *buffer;
    int n;

    sourceFile = fopen("file.c", "r");
    destinationFile = fopen("file2.c", "w");
```

```
if(sourceFile==NULL) {
    printf("Error: can't access file.c.\n");
    return 1;
}
else if(destinationFile==NULL) {
    printf("Error: can't create file for writing.\n");
    return 1;
}
else {
    n = fread(buffer, 1, 1000, sourceFile); /* grab all the text */
    fwrite(buffer, 1, n, destinationFile); /* put it in file2.c */
    fclose(sourceFile);
    fclose(destinationFile);

    destinationFile = fopen("file2.c", "r");
    n = fread(buffer, 1, 1000, destinationFile); /* read file2.c */
    printf("%s", buffer); /* display it all */

    fclose(destinationFile);
    return 0;
}
}
```

Besides reading and writing “blocks” of characters, you can use **fread** and **fwrite** to do “binary” I/O. For example, if you have an array of **int** values:

```
int array[N];
```

you could write them all out at once by calling

```
fwrite(array, sizeof(int), N, fp);
```

This would write them all out in a byte-for-byte way, i.e. as a block copy of bytes from memory to the output stream, i.e. *not* as strings of digits as `printf %d` would. Since some of the bytes within the array of **int** might have the same value as the `\n` character, you would want to make sure that you had opened the stream in binary or “wb” mode when calling **fopen**.

Later, you could try to read the integers in by calling

```
fread(array, sizeof(int), N, fp);
```

Similarly, if you had a variable of some structure type:

```
struct somestruct x;
```

you could write it out all at once by calling

```
fwrite(&x, sizeof(struct somestruct), 1, fp);
```

and read it in by calling

```
fread(&x, sizeof(struct somestruct), 1, fp);
```

6.2.5. Close Files

int fclose(FILE* fp);

This function closes the file associated with the *fp* and disassociates it. All internal buffers associated with the file are flushed: the content of any unwritten buffer is written and the content of any unread buffer is discarded. Even if the call fails, the *fp* passed as parameter will no longer be associated with the file.

Return value: If the file is successfully closed, a zero value is returned.

Example:

```
#include <stdio.h>
int main ()
{
    FILE * fp;
    fp = fopen ("myfile.txt","wt");
    fprintf (fp, "fclose example");
    fclose (fp);
    return 0;
}
```

Exercise

Exercise 1: Writes the program which calculates all the prime numbers between 3 and 100 and writes the output to a file called *primes.txt* rather than to the screen.

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

int main(void)
{
    FILE *outfile;    // this is the pointer to the FILE object
    int N;            // the integer being considered
    int D;            // needed for the integer division
    int N_is_prime;    /* = 1 (default) when N is prime and = 0 when N
                        is not prime */

    // opens a file called primes.txt for writing to
    outfile=fopen("primes.txt","w");
    for (N = 3; N <=100; N+=2)
    // This is the loop that considers all integers between 3 and 100
    {
        N_is_prime = 1; // assume N is prime (IMPORTANT)
        for (D = 3; D <= N-1; D+=2)
        {
            // if the remainder is 0 then N is prime:
            if ( N%D == 0 ) N_is_prime = 0;
        }
    }
}
```



```
        // if N is prime don't do any more integer divisions:
        if (N_is_prime == 0) break;
    }
    if (N_is_prime == 1)
        fprintf(outfile, " The integer %d is prime\n", N);
    }
    fclose(outfile);
    system("pause");
    return 0;
}
```

Exercise 2. Assumes that the *vectors.dat* exists and consists of 100 lines each with 3 real numbers representing the x, y and z components of a vector.

Using `fopen()`, `fscanf()` and `fclose()` to read data from the file, and write the function:

```
double dotprod (double x, double y, double z);
```

which for each vector the *dot product* is calculated and written to the screen.

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

```
double dotprod (double x, double y, double z);
int main(void)
{
    FILE *infile;
    int i;
    double xvec, yvec, zvec;

    infile=fopen("vectors.dat","r");
    for (i = 0; i <100; i++) // Loop 100 times
    {
        fscanf(infile, "%lf%lf%lf", &xvec, &yvec, &zvec );
        printf(" Dot product is %lf\n", dotprod(xvec, yvec, zvec) );
    }
    fclose(infile);
    system("pause");
    return 0;
}
```

```
double dotprod (double x, double y, double z)
{
    double dot = sqrt(x*x + y*y + z*z);
    return dot;
}
```

Exercise 3. In the exercise 2 we knew exactly how many lines there were in the input file *vectors.dat* and so we were able to loop over exactly that number. What if we **don't** know how many lines there are and we just want to loop until we have exhausted the data in the file? This is easy to do since **fscanf()** is a function

which returns a value which is equal to the number of *successful conversions* it has made. If this value is zero or negative then there is no more data. Therefore it is possible to both read in data and check if there are still valid data with a **while** loop which looks like:

```
while(fscanf(...) > 0)
{
}
```

The program above could thus be rewritten to work for a file *vectors.dat* which has *any* number of lines in the following way:

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

double dotprod (double x, double y, double z);
int main(void)
{
    FILE *infile;
    double xvec, yvec, zvec;

    infile=fopen("vectors.dat","r");
    while(fscanf(infile, "%lf%lf%lf", &xvec, &yvec, &zvec ) > 0)
        printf(" Dot product is %lf \n", dotprod(xvec, yvec, zvec) );
    fclose(infile);
    system("pause");
    return 0;
}

double dotprod (double x, double y, double z)
{
    double dot = sqrt(x*x + y*y + z*z);
    return dot;
}
```