KABARAK UNIVERSITY
UNIT CODE: INTE 124
UNITE NAME: STRUCTURED PROGRAMMING

UNIT I

OVERVIEW OF PROGRAMMING
Introduction to Computer Based Problem Solving
Problem Definition
Influences on Languages
Computer Architecture

- The modern computers are based upon von Neumann architecture, in which both data and program are stored in memory. Program is fetched and executed, and modifying data in the process.
- Most of the programming languages based on von Neumann architecture are called imperative languages, in which machine state is kept in variables, and the operation modify the variables to perform desired computation.
- On the other hand, functional (or applicative) languages perform computation by applying a series of function on the given parameters.

Programming Methodology

Software development cost has become much higher than hardware costs. Software productivity becomes an important issue.

- Structured programming, top-down design, and stepwise refinement.
- Data abstraction
- Object-oriented methodology, inheritance, dynamic data type binding.

Programming methodologies

- 1950s and early 1960s: Simple applications; worry about machine efficiency
- Late 1960s: People efficiency became important; readability, better control structures
  - Structured programming
  - Top-down design and step-wise refinement
- Late 1970s: Process-oriented to data-oriented
  - data abstraction
- Middle 1980s: Object-oriented programming

Programming

To solve a computing problem, its solution must be specified in terms of sequence of computational steps such that they are effectively solved by a human agent or by a digital computer.

Programming Language

1) The specification of the sequence of computational steps in a particular programming language is termed as a program
2) The task of developing programs is called programming
3) The person engaged in programming activity is called programmer

Techniques of Problem Solving

Problem solving an art in that it requires enormous intuitive power & a science for it takes a pragmatic approach.

Here a rough outline of a general problem solving approach.

1) Write out the problem statement include information on what you are to solve & consider why you need to solve the problem
2) Make sure you are solving the real problem as opposed to the perceived problem. To check to see that you define & solve the real problem
3) Draw & label a sketch. Define & name all variables and /or symbols. Show numerical values of variables, if known.
4) Identify & Name
    a. relevant principles, theories & equations
    b. system & subsystems
    c. dependent & independent variables
    d. known & unknowns
    e. inputs & outputs
    f. necessary information
5) List assumptions and approximations involved in solving the problem. Question the assumptions and then state which ones are the most reasonable for your purposes.
6) Check to see if the problem is either under-specified, figure out how to find the missing information. If over-specified, identify the extra information that is not needed.
7) Relate problem to similar problem or experience
8) Use an algorithm
9) Evaluate and examine and evaluate the answer to see it makes sense.

Why should we create computer programs?

What is it that makes computer programming useful or pointless? What are the goals, common problems, and solutions related with computer programming?

Optional reading: History of programming—what problems and tasks motivated the invention of programmable machines?

A computer is a tool for solving problems with information. You can't write a program without knowing what problem you are trying to solve any more than you can swing a knife without knowing what it is that you want to cut.

Code with a purpose. This is how:

Basic goals of computer programming

When you are planning to create a computer program you should thoroughly consider these aspects of the program you intend to make:

1. Utility: Ensure your program effectively fulfills the need it is addressing.

2. Usability: Ensure that people can easily use your program without a major time investment.
3. Maintainability: Ensure that it is easy to understand, fix, and improve your program without a major time investment.

Common problems

These are very common mistakes programmers make, which you should avoid:

Utility

Your program does not do a job any better than an alternative program. At best, you're re-inventing the wheel, at worst, wasting time.

Your program does not work how it is intended to.

Usability

Your program is too complicated or too simple to be useful to most people. Only a small minority of people want to play twenty questions when they're using a program. Just because you like to know the details doesn't mean the average user of your program does.

Maintainability

Other people, or yourself at a later time can't easily understand the programming behind your program. This means your project won't grow and become all its capable of being.

Solutions

Utility and Usability

Survey the field to look for programs something like what you intend to make. Determine what you like and don't like about those programs. Figure out ways to improve those programs.

Find source code for similar programs to what you want to make if at all possible. Unfortunately, people often write a lot of inferior code and leave it floating around the Internet. Don't be one of those people. Only release useful, well-written programs.

Thoroughly plan what you want your program to do before you start working on it. One way to do this is to make a flowchart of what you want your program to do.

Aim towards the maximum functionality your program can achieve while maintaining minimal complexity. Think of the iPod, matchstick, doorknob and television. They are simple yet effective.

If possible, talk to people who might use your program to get an idea of what features the would want or expect.

Remember the 80–20 rule: 20% of the features in most programs are used 80% of the time. If you can successfully identify even a few of the seldom used features and leave them out, you can save yourself some time and frustration.

Maintainability

Make it easy for someone to look at your program's source code and understand what's going on. Use comments when you're doing somewhat complicated things in your program, or when in doubt.

Always choose readability of your code over memory performance. Though in the short run this may not always be the best choice, in the long run, the performance can be improved if the code is easy to understand.

Use simple, easy-to-understand names for variables. The convention is that a variable should be written variable_name or variableName.
Look at examples of source code, and especially source code intended as an example of proper programming form. Recognize the structure they're using to make things simple.

Make programming easy for you and everybody else

Computers and computer programs are intended to make our lives easier, not more complicated.

What is a program?
A program is a set of instructions that tell a computer how to do a task. When a computer follows the instructions in a program, we say it executes the program. You can think of it like a recipe that tells you how to make a peanut butter sandwich. In this model, you are the computer, making a sandwich is the task, and the recipe is the program that tells you how to execute the task.

Problem-solving and Program Design
Main stages you should go through when trying to solve a general problem.

Stage 1: Define the problem
Stage 2: Analyse the problem
Stage 3: Propose and evaluate possible solutions
Stage 4: Select and justify the optimal solutions
Stage 5: Implementation and review

Stage 1: Definition of the problem

In this stage, the problem is looked at carefully and if it is not phrased properly, it is modified to ensure that it is clearly stated.

Stage 2: Analyse the problem

In this stage, identify the inputs to be used, outputs required, values to be stored (if any), and the processing that needs to be done to get the correct outputs.

Stage 3: Propose and evaluate possible solutions

Here identify solutions to the problem and evaluate each option to see which is most appropriate and choose the best option.

Stage 4: Develop and represent an algorithm

In this stage, break down the problem into simple manageable steps called algorithm so that they can be handled easily.

Stage 5: Test and validate the algorithm

Here, check the algorithm written using some values to ensure that it produces the required results.

Stage 6: Implement the algorithm

In this stage, write the steps of algorithm using a programming language so that computer can operate on it.

What is an algorithm?

An algorithm is a finite number of accurate, unambiguous steps that solve a problem or task.

List four characteristics of a good algorithm.

The number of steps must be finite; the steps must be precise; the steps must be unambiguous; the steps must terminate.

What are the three main steps involved in creating an algorithm?

- Input step
- processing step
- output step

Flowchart

A **flowchart** is a common type of diagram that represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.

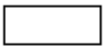
Flowchart symbols (aka, Flowchart Shapes, Business Process Map Symbols)

A typical flowchart may have the following different kinds of symbols:

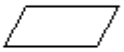
Flowcharts are usually drawn using some standard symbols; however,

Start or end of the program

Computational steps or processing function of a program



Input or output operation



Decision making and branching

Connector or joining of two parts of program

Flow line

process flow charts, whereas many of the flowchart symbols actually have their roots in the data processing diagrams and programming flow charts. So, not all the flowcharting shapes shown may be relevant to your needs.

ADVANTAGES OF USING FLOWCHARTS

The benefits of flowcharts are as follows:

1. **Communication**: Flowcharts are better way of communicating the logic of a system to all concerned.

2. **Effective analysis**: With the help of flowchart, problem can be analysed in more effective way.
3. **Proper documentation**: Program flowcharts serve as a good program documentation, which is needed for various purposes.
4. **Efficient Coding**: The flowcharts act as a guide or blueprint during the systems analysis and program development phase.
5. **Proper Debugging:** The flowchart helps in debugging process.
6. **Efficient Program Maintenance:** The maintenance of operating program becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on that part

LIMITATIONS OF USING FLOWCHARTS

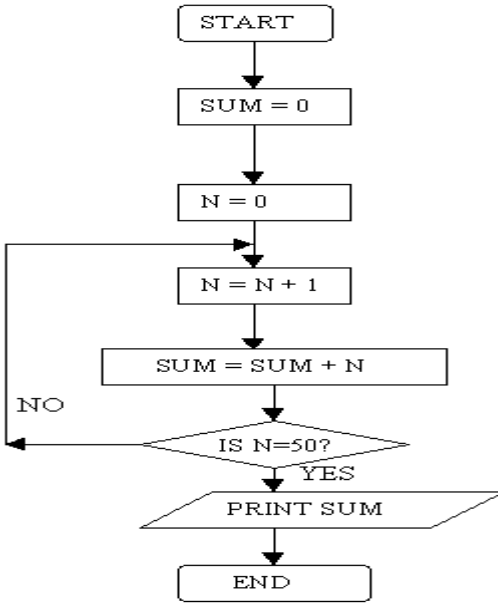
1. **Complex logic**: Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex and clumsy.
2. **Alterations and Modifications:** If alterations are required the flowchart may require re-drawing completely.
3. **Reproduction:** As the flowchart symbols cannot be typed, reproduction of flowchart becomes a problem.
4. The essentials of what is done can easily be lost in the technical details of how it is done.

EXAMPLES ON FLOWCHARTING

Example 1

Draw a flowchart to find the sum of first 50 natural numbers.

Answer: The required flowchart is given below:



How to write pseudo code

There are six basic computer operations

1. A computer can receive information
   Read (information from a file)
   Get (information from the keyboard)
2. A computer can put out information
   Write (information to a file)
   Display (information to the screen)
3. A computer can perform arithmetic
   Use actual mathematical symbols or the words for the symbols
   Add number to total
   Total = total + number
   +, -, *, /
   Calculate, Compute also used
4. A computer can assign a value to a piece of data
   3 cases
   to give data an initial value
   Initialize, Set
   To assign a value as a result of some processing
   '='
   * x = 5 + y
   to keep a piece of information for later use
   Save, Store
5. A computer can compare two piece of information and select one of two alternative actions
   IF condition THEN
   some action
   ELSE
   alternative action
   ENDIF
6. A computer can repeat a group of actions
   WHILE condition (is true)
   some action
   ENDWHILE

   FOR a number of times
   some action
   ENDFOR

Solving Problems

The problems programmers are asked to solve have been changing. Twenty years ago, programs were created to manage large amounts of raw data. The people writing the code and the people using the program were all computer professionals. Today, computers are in use by far more people, and most know very little about how computers and programs work. Computers are tools used by people who are more interested in solving their business problems than struggling with the computer.

Procedural, Structured, and Object-Oriented Programming

Until recently, programs were thought of as a series of procedures that acted upon data. A procedure, or function, is a set of specific instructions executed one after the other. The data was quite separate from the procedures, and the trick in programming was to keep track of which functions called which other functions, and what data was changed. To make sense of this potentially confusing situation, structured programming was created.

The principle idea behind structured programming is as simple as the idea of divide and conquers. A computer program can be thought of as consisting of a set of tasks. Any task that is too complex to be described simply would be broken down into a set of smaller component tasks, until the tasks were sufficiently small and self-contained enough that they were easily understood.

As an example, computing the average salary of every employee of a company is a rather complex task. You can, however, break it down into these subtasks:

1. Find out what each person earns.
2. Count how many people you have.
3. Total all the salaries.
4. Divide the total by the number of people you have.

Totalling the salaries can be broken down into

1. Get each employee's record.
2. Access the salary.
3. Add the salary to the running total.
4. Get the next employee's record.

In turn, obtaining each employee's record can be broken down into

1. Open the file of employees.
2. Go to the correct record.
3. Read the data from disk.

Structured programming remains an enormously successful approach for dealing with complex problems. By the late 1980s, however, some of the deficiencies of structured programming had become all too clear.

First, it is natural to think of your data (employee records, for example) and what you can do with your data (sort, edit, and so on) as related ideas.

Second, programmers found themselves constantly reinventing new solutions to old problems. This is often called "reinventing the wheel," and is the opposite of reusability. The idea behind reusability is to build components that have known properties, and then to be able to plug them into your program as you need them. This is modeled after the hardware world--when an engineer needs a new transistor, she doesn't usually invent one, she goes to the big bin of transistors and finds one that works the way she needs it to, or perhaps modifies it. There was no similar option for a software engineer.

New Term: The way we are now using computers--with menus and buttons and windows--fosters a more interactive, event-driven approach to computer programming. *Event-driven* means that an event happens--the user presses a button or chooses from a menu--and the program must respond. Programs are becoming increasingly interactive, and it has became important to design for that kind of functionality.

Old-fashioned programs forced the user to proceed step-by-step through a series of screens. Modern event-driven programs present all the choices at once and respond to the user's actions. Object-oriented programming attempts to respond to these needs, providing techniques for managing enormous complexity, achieving reuse of software components, and coupling data with the tasks that manipulate that data.
The essence of object-oriented programming is to treat data and the procedures that act upon the data as a single "object"--a self-contained entity with an identity and certain characteristics of its own.

Linear Programming

Linear program is a method for straightforward programming in a sequential manner. This type of programming does not involve any decision making. General model of these linear programs is:

1. Read a data value
2. Computer an intermediate result
3. Use the intermediate result to computer the desired answer
4. Print the answer
5. Stop

Structured Programming

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

Structured programming frequently employs a top-down design model, in which developers map out the overall program structure into separate subsections. A defined function or set of similar functions is coded in a separate module or sub module, which means that code can be loaded into memory more efficiently and that modules can be reused in other programs. After a module has been tested individually, it is then integrated with other modules into the overall program structure.

Advantages of Structured Programming

1. **Easy to write:**
   Modular design increases the programmer's productivity by allowing them to look at the big picture first and focus on details later. Several Programmers can work on a single, large program, each working on a different module. Studies show structured programs take less time to write than standard programs. Procedures written for one program can be reused in other programs requiring the same task. A procedure that can be used in many programs is said to be reusable.
2. **Easy to debug:**
   Since each procedure is specialized to perform just one task, a procedure can be checked individually. Older unstructured programs consist of a sequence of instructions that are not grouped for specific tasks. The logic of such programs is cluttered with details and therefore difficult to follow.
3. **Easy to Understand:**
   The relationship between the procedures shows the modular design of the program. Meaningful procedure names and clear documentation identify the task performed by each module. Meaningful variable names help the programmer identify the purpose of each variable.
4. **Easy to Change:**
   Since a correctly written structured program is self-documenting, it can be easily understood by another programmer.

Structured Programming Constructs

It uses only three constructs -

- Sequence (statements, blocks)
- Selection (if, switch)
- Iteration (loops like while and for)

Sequence

- Any valid expression terminated by a semicolon is a statement.
- Statements may be grouped together by surrounding them with a pair of curly braces.
- Such a group is syntactically equivalent to one statement and can be inserted where ever
- One statement is legal.

Selection

The selection constructs allow us to follow different paths in different situations. We may also think of them as enabling us to express decisions.
The main selection construct is:

if (*expression*)
*statement*1
else
*statement*2

Statement1 is executed if and only if *expression* evaluates to some non-zero number. If *expression* evaluates to 0, *statement*1 is not executed. In that case, *statement*2 is executed.
If and else are independent constructs, in that if can occur without else (but not the reverse).Any else is paired with the most recent else-less if, unless curly braces enforce a different scheme. Note that only curly braces, not parentheses, must be used to enforce the pairing. Parentheses

Iteration

Looping is a way by which we can execute any some set of statements more than one times continuously .In C there are mainly three types of loops are used :

- while Loop
- do while Loop
- For Loop

The control structures are easy to use because of the following reasons:

1) They are easy to recognize
2) They are simple to deal with as they have just one entry and one exit point
3) They are free of the complications of any particular programming language

Modular Design of Programs

One of the key concepts in the application of programming is the design of a program as a set of units referred to as blocks or modules. A style that breaks large computer programs into smaller elements called modules. Each module performs a single task; often a task that needs to be performed multiple times during the running of a program. Each module also stands alone with defined input and output. Since modules are able to be reused they can be designed to be used for multiple programs. By debugging each module and only including it when it performs its defined task, larger programs are easier to debug because large sections of the code have already been evaluated for errors. That usually means errors will be in the logic that calls the various modules.

Languages like Modula-2 were designed for use with modular programming. Modular programming has generally evolved into object-oriented programming.

Programs can be logically separated into the following functional modules:

1) Initialization
2) Input
3) Input Data Validation
4) Processing
5) Output

6) Error Handling
7) Closing procedure

Basic attributes of modular programming:

1) Input
2) Output
3) Function
4) Mechanism
5) Internal data

Control Relationship between modules:

The structure charts show the interrelationships of modules by arranging them at different levels and connecting modules in those levels by arrows. An arrow between two modules means the program control is passed from one module to the other at execution time. The first module is said to call or invoke the lower level modules.There are three rules for controlling the relationship between modules.

1) There is only one module at the top of the structure. This is called the root or boss module.
2) The root passes control down the structure chart to the lower level modules. However, control is always returned to the invoking module and a finished module should always terminate at the root.
3) There can be more than one control relationship between two modules on the structure chart, thus, if module A invokes module B, then B cannot invoke module A.

Communication between modules:

1) **Data**: Shown by an arrow with empty circle at its tail.
2) **Control** : Shown by a filled-in circle at the end of the tail of arrow

Interpreter

An **interpreter** is a computer program that executes other programs. This is in contrast to a compiler which does not execute its input program (the source code) but translates it into executable machine code (also called object code) which is output to a file for later execution. It

may be possible to execute the same source code either directly by an interpreter or by compiling it and then executing the machine code produced.

It takes longer to run a program under an interpreter than to run the compiled code but it can take less time to interpret it than the total required to compile and run it. This is especially important when prototyping and testing code when an edit-interpret-debug cycle can often be much shorter than an edit-compile-run-debug cycle.

Interpreting code is slower than running the compiled code because the interpreter must analyses each statement in the program each time it is executed and then perform the desired action whereas the compiled code just performs the action. This run-time analysis is known as "interpretive overhead". Access to variables is also slower in an interpreter because the mapping of identifiers to storage locations must be done repeatedly at run-time rather than at compile time.

COMPILER

A program that translates *source code* into *object code*. The compiler derives its name from the way it works, looking at the entire piece of source code and collecting and reorganizing the instructions. Thus, a compiler differs from an *interpreter*, which analyzes and executes each line of source code in succession, without looking at the entire program. The advantage of interpreters is that they can execute a program immediately. Compilers require some time before an executable program emerges. However, programs produced by compilers run much faster than the same programs executed by an interpreter.

Every high-level programming language (except strictly interpretive languages) comes with a compiler. In effect, the compiler is the language, because it defines which instructions are acceptable.

Generations of Programming Languages

- Levels or generations of programming languages range from low to high
- Lower level languages are closer to the language the computer uses itself (machine code – 1s and 0s)
- Higher level languages are closer to human languages

UNIT - II

FUNDAMENTALS OF C PROGRAMMING

Origin of C

The C Programming Language was initially developed by Denis Ritchie using a Unix system in 1972. This was varied and modified until a standard was defined by Brian Kernighan and Dennis Ritchie in 1978 in "The C Programming Language".

By the early 80's many versions of C were available which were inconsistent with each other in many aspects. This led to a standard being defined by ANSI in 1983. It is this standard this set of notes primarily addresses.

Why use C?

Industry Presence: Over the last decade C has become one of the most widely used development languages in the software industry. Its importance is not entirely derived from its use as a primary development language but also because of its use as an interface language to some of the newer "visual" languages and of course because of its relationship with C++.
Middle Level: Being a Middle level language it combines elements of high level languages with the functionality of assembly language. C supports data types and operations on data types in much the same way as higher level languages as well as allowing direct manipulation of bits, bytes, words and addresses as is possible with low level languages.
Portability: With the availability of compilers for almost all operating systems and hardware platforms it is easy to write code on one system which can be easily ported to another as long as a few simple guidelines are followed.
Flexibility: Supporting its position as the mainstream development language C can be interfaced readily to other programming languages.
Malleable: C, unlike some other languages, offers little restriction to the programmer with regard to data types -- one type may be coerced to another type as the situation dictates. However this *feature* can lead to sloppy coding unless the programmer is fully aware of what rules are being bent and why.
Speed: The availability of various optimising compilers allow extremely efficient code to be generated automatically.

Characteristics of C

- C is a general purpose programming language.
- It is a structured programming language.
- Helps in development of system software.
- Has rich set of operators.
- Provides compact representation for expressions.
- Allows manipulation of internal processor registers.
- No rigid format as any number of statements can be typed in a single line.
- It is portable.
- Supports rich set of data types.
- Less number of reserved words.

Application of C

Because of its portability and efficiency, C is used to develop the system as well as application software. Eg
System software:

- Operating systems
- Interpreters

- Compilers
- Assemblers
- Editors
- Loaders
- Linkers

Application software:

- Database systems
- Graphics packages
- Spreadsheets
- CAD/CAM applications
- Word processors

Structure of C program

The basic structure of C program comprises of the following elements:

- Preprocessor statements
- Main()
- A pair of curl brace { }
- Declaration and statements
- User defined functions

Format:

```
Preprocessor statements
Global declaration;
main ()
{
        Declarations;
        Statements;
}
User defined functions
```

Preprocessor statements:

These statements start with # symbol. They are also called as preprocessor directives. They direct the C preprocessor to include header files and also symbolic constants into a C program.

Eg:

```
#include <stdio.h>      //for the standard input/output functions.
Void main ()            //compiler will start from here
{                       //start of a program
        printf ("hi");  //display some values on the screen
        scanf ("%d %d);//enter some values through keyboard
}                       //end of the program
```

Global declarations:

Are functions or variables whose existence is known in the main () function and other user defined functions are called global variables, and their declarations global declarations.

main ( ):

It is the main function of every C program where all execution starts. No C program can be executed without the main ( ) function. It should be written in lower case letters and should not be terminated by a semicolon and it should be only one.

Curl brace { }:
The left curl brace indicate the beginning of the main function and right brace indicates the end of the main function.

printf():
This is the actual print statement which is used in our c program frequently. We have header file stdio.h! But what it does? How it is defined?

return 0:
Then the return 0 statement. Seems like we are trying to give something back, and it gives the result as an integer. Maybe if we modified our main function definition: int main (), now we are saying that our main function will be returning an integer! So, you should always explicitly declare the return type on the function.

Declaration:
It is a part of C program where all the variables, arrays, functions etc…., used in C program are declared and may be initialized with their basic data types.

Statements:
These are instructions to the computer to do specific operations. They may be input-output statements, arithmetic statements, control statements, and other statements. They also include comments (within /*…..*/). What about the documentation? We should probably document some of our code so that other people can understand what we are doing. Comments in the C89 standard are noted by this: /* */. The comment always begins with /* and ends with */.

User-defined functions
These are sub-programs/functions. They contain a set of statements to perform specific task. They may be written before or after the main ( ) function.

Executing a C program
Execution is the process of running the program, to execute a program in C, the following steps are followed:

1) **Creating the program**
   Is the process of entering and editing the program in standard C editor and save with .c extension.
2) **Compiling the program**
   Is the process of converting the high level language program into machine language.
3) **Linking the with system library**
   C language is a collection of predefined functions; these functions are already written in some standard C header files.

Before execution they link the system library; this is done automatically at the execution time.

4) **Executing the program**
   It is the process of running and testing the program.

Program Structure

A C program basically has the following form:

- Pre-processor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

The following program is written in the C programming language.

```
#include <stdio.h>

int main()
{
/* My first program */
printf("Hello, TechPreparation! \n");

return 0;
}
```

Preprocessor Commands:

These commands tells the compiler to do preprocessing before doing actual compilation. Like #include <stdio.h> is a preprocessor command which tells a C compiler to include stdio.h file before going to actual compilation.

Functions:

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

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

Variables:

Variables are used to hold numbers, strings and complex data for manipulation.

Statements & Expressions :
Expressions combine variables and constants to create new values. Statements are expressions, assignments, function calls, or control flow statements which make up C programs.

Comments:
Comments are used to give additional useful information inside a C Program. All the comments will be put inside /*...*/ as given in the example above. A comment can span through multiple lines.

Note the followings

- C is a case sensitive programming language. It means in C printf and Printf will have different meanings.
- C has a free-form line structure. End of each C statement must be marked with a semicolon.
- Multiple statements can be one the same line.
- White Spaces (i.e. tab space and space bar) are ignored.
- Statements can continue over multiple lines.

C-Language keywords

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			

Data Types

A C language programmer has to tell the system before-hand, the type of numbers or characters he is using in his program. These are data types. There are many data types in C language. A C programmer has to use appropriate data type as per his requirement.
C language data types can be broadly classified as

- Primary data type
- Derived data type
- User defined data type

Primary data type

All C Compilers accept the following fundamental data types

1.	Integer	int

2.	Character	char
3.	Floating Point	float
4.	Double precision floating point	double
5.	Void	void

The size and range of each data type is given in the table below

DATA TYPE	RANGE OF VALUES
char	-128 to 127
Int	-32768 to +32767
float	3.4 e-38 to 3.4 e+38
double	1.7 e-308 to 1.7 e+308

Integer Type:

Integers are whole numbers with a machine dependent range of values. A good programming language has to support the programmer by giving a control on a range of numbers and storage space. C has 3 classes of integer storage namely short int, int and long int. All of these data types have signed and unsigned forms. A short int requires half the space than normal integer values. Unsigned numbers are always positive and consume all the bits for the magnitude of the number. The long and unsigned integers are used to declare a longer range of values.

Floating Point Types:

Floating point number represents a real number with 6 digits precision. Floating point numbers are denoted by the keyword float. When the accuracy of the floating point number is insufficient, we can use the double to define the number. The double is same as float but with longer precision. To extend the precision further we can use long double which consumes 80 bits of memory space.

Void Type:

Using void data type, we can specify the type of a function. It is a good practice to avoid functions that does not return any values to the calling function.

Character Type:

A single character can be defined as a character type of data. Characters are usually stored in 8 bits of internal storage. The qualifier signed or unsigned can be explicitly applied to char. While unsigned characters have values between 0 and 255, signed characters have values from –128 to 127.

Size and Range of Data Types on 16 bit machine.

TYPE	SIZE (Bits)	Range
Char or Signed Char	8	-128 to 127
Unsigned Char	8	0 to 255
Int or Signed int	16	-32768 to 32767
Unsigned int	16	0 to 65535
Short int or Signed short int	8	-128 to 127
Unsigned short int	8	0 to 255
Long int or signed long int	32	-2147483648 to 2147483647
Unsigned long int	32	0 to 4294967295
Float	32	3.4 e-38 to 3.4 e+38
Double	64	1.7e-308 to 1.7e+308
Long Double	80	3.4 e-4932 to 3.4 e+4932

Variable:

Variable is a name of memory location where we can store any data. It can store only single data (Latest data) at a time. In C, a variable must be declared before it can be used. Variables can be declared at the start of any block of code, but most are found at the start of each function.

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

DataType Name_of_Variable_Name;

int a,b,c;

Variable Names

Every variable has a name and a value. The name identifies the variable, the value stores data. There is a limitation on what these names can be. Every variable name in C must start with a letter; the rest of the name can consist of letters, numbers and underscore characters. C recognizes upper and lower case characters as being different. Finally, you cannot use any of C's keywords like main, while, switch etc as variable names.

Examples of legal variable names include:

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

It is conventional to avoid the use of capital letters in variable names. These are used for names of constants. Some old implementations of C only use the first 8 characters of a variable name.

Local Variables

Local variables are declared within the body of a function, and can only be used within that function only.

Syntax:

```
Void main( ){

int a,b,c;

}

Void fun1()

{

int x,y,z;

}
```

Here a,b,c are the local variable of void main() function and it can't be used within fun1() Function. And x, y and z are local variable of fun1().

Global Variable

A global variable declaration looks normal, but is located outside any of the program's functions. This is usually done at the beginning of the program file, but after preprocessor directives. The variable is not declared again in the body of the functions which access it.

Syntax:

```
#include<stdio.h>
int a,b,c;
void main()
{
}
Void fun1()
{
}
```

Here a,b,c are global variable .and these variable can be accessed (used) within a whole program.

Constants

C constant is usually just the written version of a number. For example 1, 0, 5.73, 12.5e9. We can specify our constants in octal or hexadecimal, or force them to be treated as long integers.

- Octal constants are written with a leading zero - 015.
- Hexadecimal constants are written with a leading 0x - 0x1ae.
- Long constants are written with a trailing L - 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.

'\n'	newline
'\t'	tab
'\\ '	backslash
'\''	single quote
'\0'	null (used automatically to terminate character strings).

In addition, a required bit pattern can be specified using its octal equivalent.

'\044' produces bit pattern 00100100.

Character constants are rarely used, since string constants are more convenient. A string constant is surrounded by double quotes e.g. "Brian and Dennis". The string is actually stored as an array of characters. The null character '\0' is automatically placed at the end of such a string to act as a string terminator.

Constant is a special types of variable which cannot be changed at the time of execution. Syntax:

const int a=20;

Operators Introduction

An operator is a symbol which helps the user to command the computer to do a certain mathematical or logical manipulations. Operators are used in C language program to operate on data and variables. C has a rich set of operators which can be classified as

1. Arithmetic operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Increments and Decrement Operators
6. Conditional Operators
7. Bitwise Operators
8. Special Operators

1. Arithmetic Operators

All the basic arithmetic operations can be carried out in C. All the operators have almost the same meaning as in other languages. Both unary and binary operations are available in C language. Unary operations operate on a single operand, therefore the number 5 when operated by unary – will have the value –5.

Arithmetic Operators

Operator	Meaning
+	Addition or Unary Plus
–	Subtraction or Unary Minus
*	Multiplication
/	Division
%	Modulus Operator

Examples of arithmetic operators are

x + y
x - y
-x + y
a * b + c
-a * b
etc.,

here a, b, c, x, y are known as operands. The modulus operator is a special operator in C

language which evaluates the remainder of the operands after division.

Example

```
#include //include header file stdio.h
void main() //tell the compiler the start of the program
{
int numb1, num2, sum, sub, mul, div, mod; //declaration of variables
scanf ("%d %d", &num1, &num2); //inputs the operands

sum = num1+num2; //addition of numbers and storing in sum.
printf("\n Thus sum is = %d", sum); //display the output

sub = num1-num2; //subtraction of numbers and storing in sub.
printf("\n Thus difference is = %d", sub); //display the output

mul = num1*num2; //multiplication of numbers and storing in mul.
printf("\n Thus product is = %d", mul); //display the output

div = num1/num2; //division of numbers and storing in div.
printf("\n Thu division is = %d", div); //display the output

mod = num1%num2; //modulus of numbers and storing in mod.
printf("\n Thu modulus is = %d", mod); //display the output
}
```

Integer Arithmetic

When an arithmetic operation is performed on two whole numbers or integers than such an operation is called as integer arithmetic. It always gives an integer as the result. Let x = 27 and y = 5 be 2 integer numbers. Then the integer operation leads to the following results.

x + y = 32
x – y = 22
x * y = 115
x % y = 2
x / y = 5

In integer division the fractional part is truncated.

Floating point arithmetic

When an arithmetic operation is preformed on two real numbers or fraction numbers such an operation is called floating point arithmetic. The floating point results can be truncated according to the properties requirement. The remainder operator is not applicable for floating point arithmetic operands.

Let x = 14.0 and y = 4.0 then

x + y = 18.0
x – y = 10.0
x * y = 56.0
x / y = 3.50

Mixed mode arithmetic

When one of the operand is real and other is an integer and if the arithmetic operation is carried out on these 2 operands then it is called as mixed mode arithmetic. If any one operand is of real type then the result will always be real thus 15/10.0 = 1.5

2. Relational Operators

Often it is required to compare the relationship between operands and bring out a decision and program accordingly. This is when the relational operator comes into picture. C supports the following relational operators.

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to

It is required to compare the marks of 2 students, salary of 2 persons; we can compare those using relational operators.

A simple relational expression contains only one relational operator and takes the following form.

exp1 relational operator exp2

Where exp1 and exp2 are expressions, which may be simple constants, variables or combination of them. Given below is a list of examples of relational expressions and evaluated values.

6.5 <= 25 TRUE
-65 > 0 FALSE
10 < 7 + 5 TRUE

Relational expressions are used in decision making statements of C language such as if, while and for statements to decide the course of action of a running program.

3. Logical Operators
C has the following logical operators; they compare or evaluate logical and relational expressions.

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

Logical AND (&&)
This operator is used to evaluate 2 conditions or expressions with relational operators simultaneously. If both the expressions to the left and to the right of the logical operator is true then the whole compound expression is true.

Example

a > b && x = = 10

The expression to the left is a > b and that on the right is x == 10 the whole expression is true only if both expressions are true i.e., if a is greater than b and x is equal to 10.

Logical OR (||)
The logical OR is used to combine 2 expressions or the condition evaluates to true if any one of the 2 expressions is true.

Example

a < m || a < n

The expression evaluates to true if any one of them is true or if both of them are true. It evaluates to true if a is less than either m or n and when a is less than both m and n.

Logical NOT (!)
The logical not operator takes single expression and evaluates to true if the expression is false and evaluates to false if the expression is true. In other words it just reverses the value of the expression.

For example

! (x >= y) the NOT expression evaluates to true only if the value of x is neither greater than or equal to y

4. Assignment Operators
The Assignment Operator evaluates an expression on the right of the expression and substitutes it to the value or variable on the left of the expression.

Example

x = a + b

Here the value of a + b is evaluated and substituted to the variable x.

In addition, C has a set of shorthand assignment operators of the form.

var oper = exp;

Here var is a variable, exp is an expression and oper is a C binary arithmetic operator. The operator oper = is known as shorthand assignment operator
Example

x + = 1 is same as x = x + 1

The commonly used shorthand assignment operators are as follows

Shorthand assignment operators.

Statement with simple assignment operator	Statement with shorthand operator
a = a + 1	a += 1
a = a – 1	a -= 1
a = a * (n+1)	a *= (n+1)
a = a / (n+1)	a /= (n+1)
a = a % b	a %= b

Example for using shorthand assignment operator

5. Increment and Decrement Operators

The increment and decrement operators are one of the unary operators which are very useful in C language. They are extensively used in for and while loops. The syntax of the operators is given below

1. ++ variable name
2. Variable name++
3. – –variable name
4. Variable name– –

The increment operator ++ adds the value 1 to the current value of operand and the decrement operator – – subtracts the value 1 from the current value of operand. ++variable name and variable name++ mean the same thing when they form statements independently, they behave differently when they are used in expression on the right hand side of an assignment statement.

Consider the following.

m = 5;
y = ++m; (prefix)

In this case the value of y and m would be 6

Suppose if we rewrite the above statement as

m = 5;
y = m++; (post fix)

Then the value of y will be 5 and that of m will be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on the left. On the other hand, a postfix operator first assigns the value to the variable on the left and then increments the operand.

6. Conditional or Ternary Operator

The conditional operator consists of 2 symbols the question mark (?) and the colon (:)
The syntax for a ternary operator is as follows.

exp1 ? exp2 : exp3

The ternary operator works as follows

exp1 is evaluated first. If the expression is true then exp2 is evaluated & its value becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression. Note that only one of the expression is evaluated.

For example

```
a = 10;
b = 15;
x = (a > b) ? a : b
```

Here x will be assigned to the value of b. The condition follows that the expression is false therefore b is assigned to x.

```
/* Example : to find the maximum value using conditional operator)
#include
void main() //start of the program
{
int i,j,larger; //declaration of variables
printf (“Input 2 integers : ”); //ask the user to input 2 numbers
scanf(“%d %d”,&i, &j); //take the number from standard input and store it
larger = i > j ? i : j; //evaluation using ternary operator
printf(“The largest of two numbers is %d \n”, larger); // print the largest number
} // end of the program
```

Output

```
Input 2 integers: 34 45
The largest of two numbers is 45
```

7. <u>Bitwise Operators</u>

C has a distinction of supporting special operators known as bitwise operators for manipulation data at bit level. A bitwise operator operates on each bit of data. Those operators are used for testing, complementing or shifting bits to the right on left. Bitwise operators may not be applied to a float or double.

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive
<<	Shift left
>>	Shift right

8. Special Operators

C supports some special operators of interest such as comma operator, size of operator, pointer operators (& and *) and member selection operators (. and ->). The size of and the comma operators are discussed here. The remaining operators are discussed in forth coming chapters.

The Comma Operator

The comma operator can be used to link related expressions together. A comma-linked list of expressions are evaluated left to right and value of right most expression is the value of the combined expression.

For example the statement

value = (x = 10, y = 5, x + y);

First assigns 10 to x and 5 to y and finally assigns 15 to value. Since comma has the lowest precedence in operators the parenthesis is necessary. Some examples of comma operator are

In for loops:
for (n=1, m=10, n <=m; n++,m++)
In while loops
While (c=getchar(), c != '10')
Exchanging values.

t = x, x = y, y = t;

Specifier Meaning

%c – Print a character
%d – Print a Integer
%i – Print a Integer
%e – Print float value in exponential form.
%f – Print float value
%g – Print using %e or %f whichever is smaller
%o – Print actual value
%s – Print a string
%x – Print a hexadecimal integer (Unsigned) using lower case a – F
%X – Print a hexadecimal integer (Unsigned) using upper case A – F
%a – Print a unsigned integer.
%p – Print a pointer value
%hx – hex short
%lo – octal long
%ld – long unsigned integer.

Input and Output

Input and output are covered in some detail. C allows quite precise control of these. This section discusses input and output from keyboard and screen.

The same mechanisms can be used to read or write data from and to files. It is also possible to treat character strings in a similar way, constructing or analyzing them and storing results in variables. These variants of the basic input and output commands are discussed in the next section

- The Standard Input Output File
- Character Input / Output
  - getchar
  - putchar
- Formatted Input / Output
  - printf
  - scanf
- Whole Lines of Input and Output
  - gets
  - puts

printf

This offers more structured output than putchar. Its arguments are, in order; a control string, which controls what gets printed, followed by a list of values to be substituted for entries in the control string

Example: int a,b;

printf(" a = %d,b=%d",a,b);.

Control String Entry	What Gets Printed
%d	A Decimal Integer
%f	A Floating Point Value
%c	A Character
%s	A Character String

It is also possible to insert numbers into the control string to control field widths for values to be displayed. For example %6d would print a decimal value in a field 6 spaces wide, %8.2f would print a real value in a field 8 spaces wide with room to show 2 decimal places. Display is left justified by default, but can be right justified by putting a - before the format information, for example %-6d, a decimal integer right justified in a 6 space field

scanf

scanf allows formatted reading of data from the keyboard. Like printf it has a control string, followed by the list of items to be read. However scanf wants to know the address of the items to be read, since it is a function which will change that value. Therefore the names of variables are preceded by the & sign. Character strings are an exception to this. Since a string is already a character pointer, we give the names of string variables unmodified by a leading &.

Control string entries which match values to be read are preceeded by the percentage sign in a similar way to their printf equivalents.

Example: int a,b;

scan f("%d%d",& a,& b);

getchar

getchar returns the next character of keyboard input as an int. If there is an error then EOF (end of file) is returned instead. It is therefore usual to compare this value against EOF before using it. If the return value is stored in a char, it will never be equal to EOF, so error conditions will not be handled correctly.

As an example, here is a program to count the number of characters read until an EOF is encountered. EOF can be generated by typing Control - d.

```
#include <stdio.h>

main()
{   int ch, i = 0;

    while((ch = getchar()) != EOF)
       i ++;

    printf("%d\n", i);
}
```

putchar

putchar puts its character argument on the standard output (usually the screen).

The following example program converts any typed input into capital letters. To do this it applies the function to upper from the character conversion library c type .h to each character in turn.

```
#include <ctype.h>  /* For definition of toupper */
#include <stdio.h>  /* For definition of getchar, putchar, EOF */

main()
{   char ch;

    while((ch = getchar()) != EOF)
        putchar (toupper(ch));
}
```

gets

gets reads a whole line of input into a string until a new line or EOF is encountered. It is critical to ensure that the string is large enough to hold any expected input lines.

When all input is finished, NULL as defined in studio is returned.

```
#include <stdio.h>

main()
{   char ch[20];

    gets(x);
    puts(x);
}
```

puts

puts writes a string to the output, and follows it with a new line character.

Example: Program which uses gets and puts to double space typed input.

```
#include <stdio.h>

main()
{       char line[256]; /* Define string sufficiently large to
                           store a line of input */

        while(gets(line) != NULL)   /* Read line        */
        {       puts(line);         /* Print line       */
                printf("\n");       /* Print blank line */
        }
}
```

Note that putchar, printf and puts can be freely used together

Expression Statements

Most of the statements in a C program are *expression statements*. An expression statement is simply an expression followed by a semicolon. The lines

```
i = 0;
i = i + 1;
```

and

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

are all expression statements. (In some languages, such as Pascal, the semicolon separates statements, such that the last statement is not followed by a semicolon. In C, however, the semicolon is a statement terminator; all simple statements are followed by semicolons. The semicolon is also used for a few other things in C; we've already seen that it terminates declarations, too.

Branching and looping
CONTROL FLOW STATEMENT
IF- Statement:
It is the basic form where the if statement evaluate a test condition and direct program execution depending on the result of that evaluation.
Syntax:

```
If (Expression)
{
Statement 1;
Statement 2;
}
```

Where a statement may consist of a single statement, a compound statement or nothing as an empty statement. The Expression also referred so as test condition must be enclosed in parenthesis, which cause the expression to be evaluated first, if it evaluate to true (a non zero value), then the statement associated with it will be executed otherwise ignored and the control will pass to the next statement.
Example:

```
if (a>b)
{
printf (“a is larger than b”);
}
```

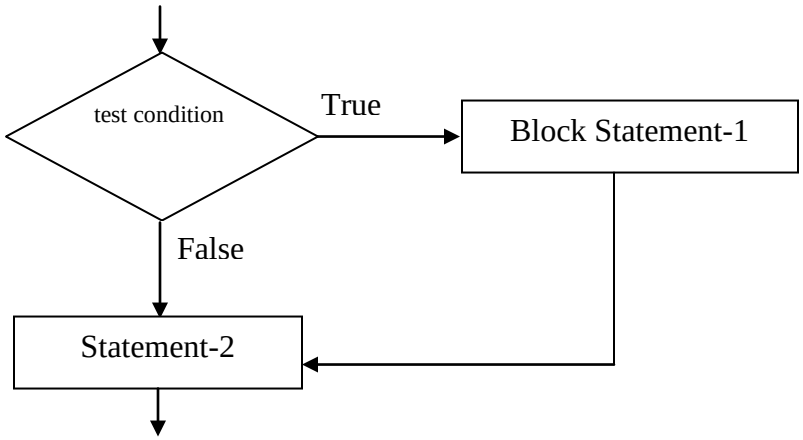
It is most powerful and widely used statement used for decision making. The syntax of the ‘if’ statement is:

```
if(test condition) {
        Block Statement-1
}
```

Statement-2

.....

First your computer will check the test condition. If the condition is true then computer put non-zero value and execute the 'Block Statement-1', after executing block statement it will execute 'Statement-2'. If condition is false then computer put zero value and will not execute 'Block Statement-1' but directly execute 'Statement-2'.



The example of the 'if' statement is:

```
if(marks < 40) {
        printf("You are fail\n");
}
printf("Total Marks = %d\n",marks);
```

Here first the 'if' statement is executed. Your computer will check whether marks is less than 40 or not and if it is that then it will print message "You are fail" and then print the total marks otherwise it directly print the total marks.

IF-ELSE Statement:

An if statement may also optionally contain a second statement, the ``else clause," which is to be executed if the condition is not met. Here is an example:

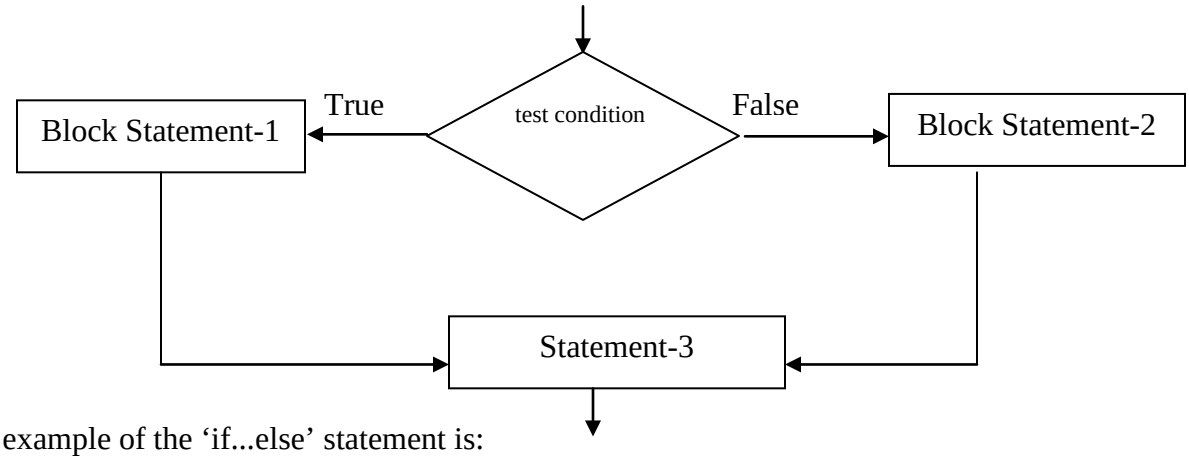
```
if(n > 0)
        average = sum / n;
else    {
        printf("can't compute average\n");
```

```
        average = 0;
        }
```

It is extension of the 'if' statement. It's format is:

```
if(test condition) {
        Block Statement-1
}
else {
Block Statement-2
}
Statement-3
.....................................................
```

First your computer will check the test condition. If the condition is true then computer execute the 'Block Statement-1'. If condition is false then computer will execute 'Block Statement-2'. After executing either 'Block Statement-1' or 'Block Statement-2' it will execute the 'Statement-3'.



The example of the 'if...else' statement is:

```
if(marks < 40) {
        printf("You are fail\n");
}
else {
        printf("You are pass\n");
}
printf("Total Marks = %d\n",marks);
```

Here first the 'if' statement is executed. Your computer will check whether marks is less than 40 or not and if it is that then it will print message "You are fail" otherwise it will print the message "You are pass". After print pass or fail it will print total marks.

***NESTED-IF- Statement*:**

It's also possible to nest one `if` statement inside another. (For that matter, it's in general possible to nest any kind of statement or control flow construct within another.) For example, here is a little piece of code which decides roughly which quadrant of the compass you're walking into, based on an `x` value which is positive if you're walking east, and a `y` value which is positive if you're walking north:

```
if(x > 0)
        {
        if(y > 0)
                printf("Northeast.\n");
        else    printf("Southeast.\n");
        }
else    {
        if(y > 0)
                printf("Northwest.\n");
        else    printf("Southwest.\n");
        }
```

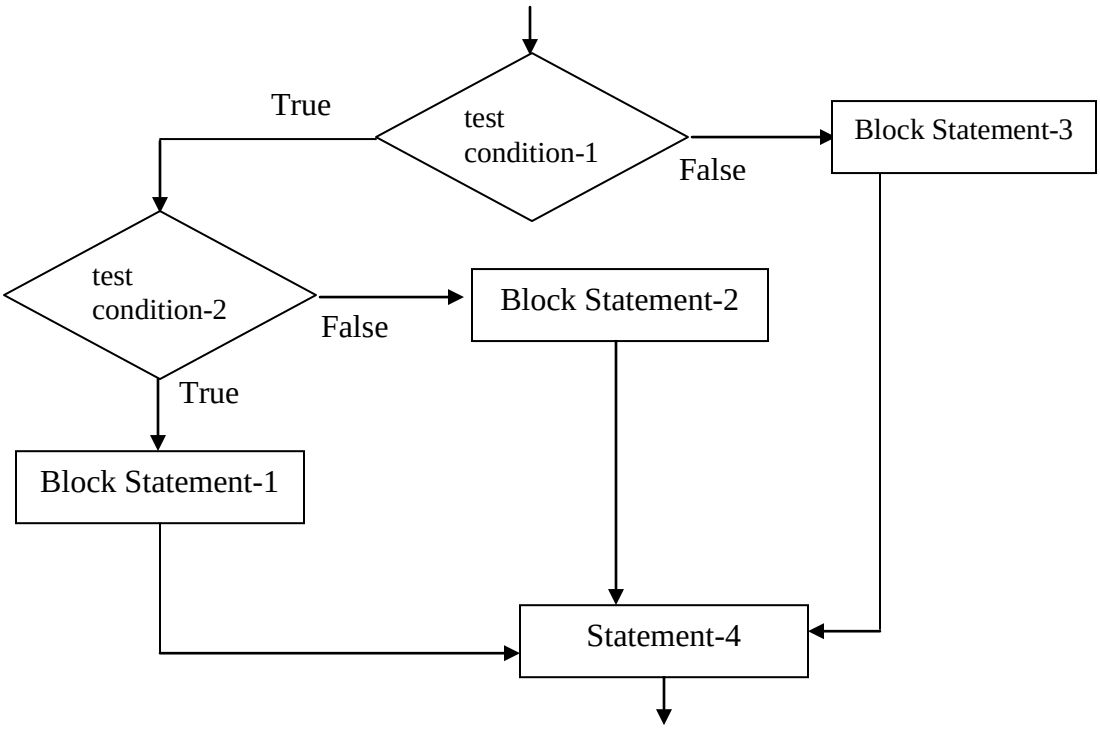
When we are using 'if...else' statement with in one 'if...else' statement then it will make the nested 'if...else' statement. The format of nested 'if...else' statement is:

```
        if(test condition-1) {
                if(test condition-2) {
                        Block Statement-1
    }
    else {
                        Block Statement-2
    }
}
else {
    Block Statement-3
}
Statement-4
```

.....

Here first your computer will check the 'test condition-1'. If the condition is true then computer check another 'test condition-2'. If 'test condition-2' is true then it will execute 'Block Statement-1' otherwise execute the 'Block Statement-2'. If 'test-condition1' is false then it will execute 'Block Statement-3'.

After completing the nested 'if...else' statement execution computer will execute 'Statemetn-4'.



The example of the nested 'if...else' is:

```
if(no1>=no2) {
        if(no1>=no3) {
```

```
                printf("Number1 is Maximum\n");
        }
        else {
                printf("Number3 is Maximum\n");
        }
    }
    else  {
        if(no2>=no3) {
                printf("Number2 is Maximum\n");
        }
        else {
                printf("Number3 is Maximum\n");
        }
    }
```

Here first the 'if' statement is to check whether no1>=no2. If condition is true then it will check whether no1>=no3. If second condition is true then it will print message "Number1 is Maximum" otherwise print the message "Number3 is Maximum".

If condition no1>=no2 is false then computer will check the condition no2>=no3 and it true then it will print message "Number2 is Maximum" otherwise print the message "Number3 is Maximum."

```
/* Illustates nested if else and multiple arguments to the scanf function.  */
    #include <stdio.h>

    main()
    {
        int    invalid_operator = 0;
        char   operator;
        float  number1, number2, result;
        printf("Enter two numbers and an operator in the format\n");
        printf(" number1 operator number2\n");
        scanf("%f %c %f", &number1, &operator, &number2);

        if(operator == '*')
            result = number1 * number2;
```

```
        else if(operator == '/')
                result = number1 / number2;
        else if(operator == '+')
                result = number1 + number2;
        else if(operator == '-')
                result = number1 - number2;
        else
                invalid _ operator = 1;

        if( invalid _ operator != 1 )
        printf("%f %c %f is %f\n", number1, operator, number2, result );
        else
                printf("Invalid operator.\n");
}
```

Sample Program Output

```
Enter two numbers and an operator in the format
number1 operator number2
23.2 + 12
23.2 + 12 is 35.2
```

Switch Case

This is another form of the multi way decision. It is well structured, but can only be used in certain cases where;

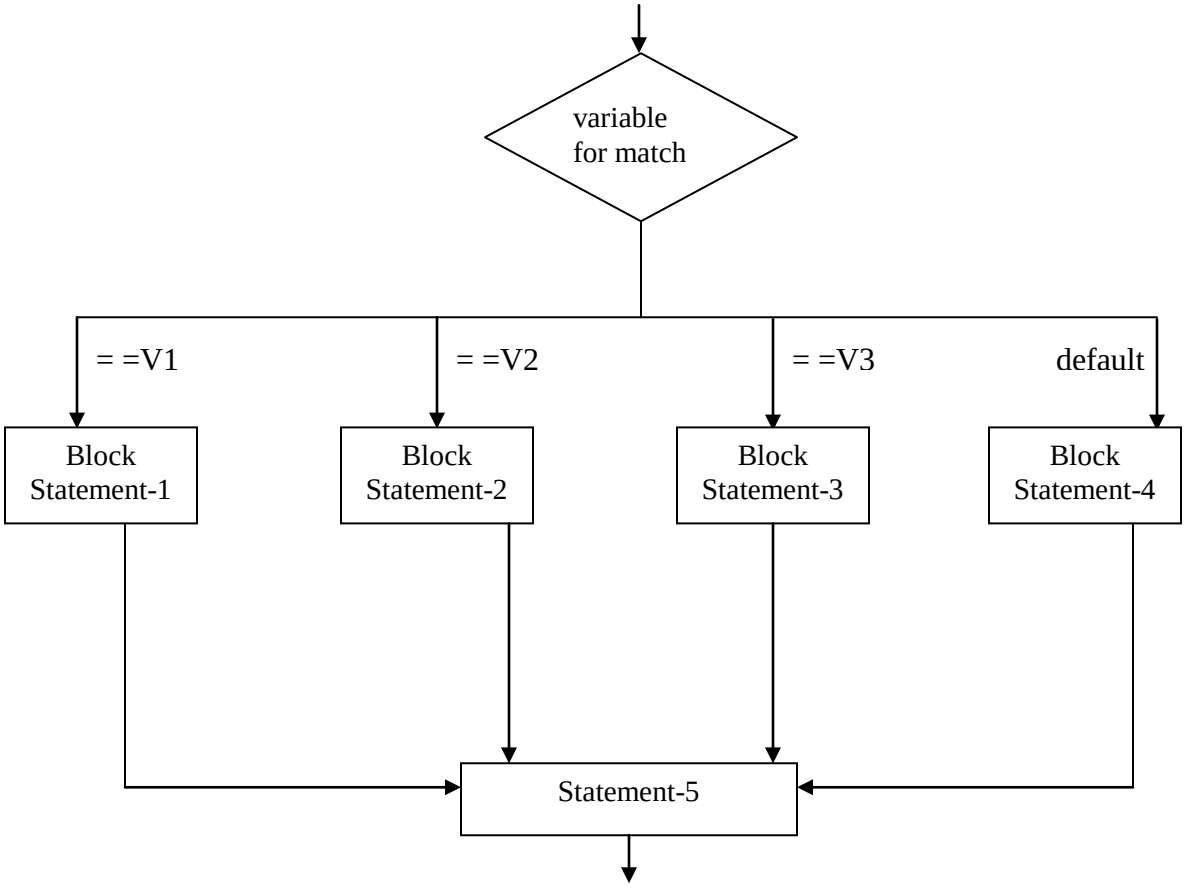
- Only one variable is tested, all branches must depend on the value of that variable. The variable must be an integral type. (int, long, short or char).
- Each possible value of the variable can control a single branch. A final, catch all, default branch may optionally be used to trap all unspecified cases.
- When there is multiple condition in the 'if' statement it is become difficult to read and understand program. In this condition we can use 'switch' statement, which allows checking the value of variable against the multiple case value. The format of the this statement is:

```
                switch(variablename) {
case    v1      :       Block Statement-1
                        break;
case    v2      :       Block Statement-2
                        break;
case    v3      :       Block Statement-3
```

```
                    break;
default :       Block Statement-4
                    break;
}
Statement-5
```

Here if the value of variable= = v1 then 'Block Statement-1' will be executed. If variable= = v2 then 'Block Statement-2' will be executed. In similar way if at last no match found for the variable then default statement is executed.

Here we have to put the 'break' statement to complete the execution of 'switch' statement after matching one case.



Consider the following example of the ‘switch’ statement to accept the month in number and print into word.

```
        switch(month) {
case    1       :       printf(“January”);
                        break;
case    2       :       printf(“February”);
                        break;
case    5       :       printf(“May”);
                        break;
case    10      :       printf(“October”);
                        break;
default :       printf(“You have entered wrong value”);
                        break;
}
```

In above program if you enter month equal to 1 then print January, month equal to 2 then February, month equal to 5 then May, month equal to 10 then October. If no any match is found for month then print, “You have entered wrong value”.

Similar to nested ‘if...else’ you can use nested ‘switch’ statement. You can use only **15 level of nested switch**. Also you can take only **257 cases into switch** statement.

Hopefully an example will clarify things. This is a function which converts an integer into a vague description. It is useful where we are only concerned in measuring a quantity when it is quite small.

```
estimate(number)
int number;
/* Estimate a number as none, one, two, several, many */
{       switch(number) {
        case 0 :
                printf("None\n");
                break;
```

```
    case 1 :
        printf("One\n");
        break;
    case 2 :
        printf("Two\n");
        break;
    case 3 :
    case 4 :
    case 5 :
        printf("Several\n");
        break;
    default :
        printf("Many\n");
        break;
    }
}
```

Each interesting case is listed with a corresponding action. The break statement prevents any further statements from being executed by leaving the switch. Since case 3 and case 4 have no following break, they continue on allowing the same action for several values of number.

Both if and switch constructs allow the programmer to make a selection from a number of possible actions.

- **Conditional Operator OR ?: Operator: -**
  'C' provides the facility of the operator to check the condition by using the conditional operator having three operands. The format of the condition:

  (Condition)? Statement1 : Statement2

  If condition is true then statement1 will be executed otherwise statement2 will be executed. Here you can also represent the condition for assignment operation:

  variable = (Condition)? Value1: Value2

  Here if condition is true then variable has assigned the value1 otherwise it has assigned the value2.

  Consider the following example to understand the conditional operator:

```
commission = (sales > 50000) ? 2000 : 0;
```

In above example if your sales is greater than 50000 then your commission value is 2000 otherwise it is zero. Above condition you can represent using 'if' statement as under:

```
if (sales > 50000) {
        commission=2000;
}
else {
        commission=0;
}
```

The advantage of the conditional operator is that you can represent the condition in shorter form. But some times it is difficult to understand when you are using multiple conditional operators.

Loops

*"When sequences of statements are executed repeatedly up to some condition then it is known as **Program Loop**."*

A loop consists mainly two parts:

1. Body of loop:
   This part contains the sequences of statements, which are to be executed repeatedly.

2. Control statement:
   This part contains certain conditions, which indicates how many times we have to execute the body of loop. The control statement should be write carefully otherwise it may possible your loop is executed infinite times which is known as **Infinite Loop.**

Out loop contains one or more variable, which control the execution of loop that is known as the **Loop Counter**.

Loop is executed basically in following steps:

1. Setting and initialization of Loop Counter.
2. Test the condition.
3. Executing the body of loop.
4. Incrementing or decrementing the Loop Counter value.

In above steps, 2nd and 3rd steps may be executed interchangeably.

Looping is a way by which we can execute any some set of statements more than one times continuously .In c there are mainly three types of loops are use :

There are mainly two kinds of loops:

1. **Entry-controlled loop:**
   In entry-controlled loop we first check the condition and then the body of loop is executed. If at first time condition is false then the body of loop and not executed any time. 'While' loop and 'for' loop is example of entry-controlled loop.
2. **Exit-controlled loop:**
   In exit-controlled loop we first execute the body of the loop and then check the condition and if condition is true then next again executed the body of the loop otherwise not. In exit-controlled loop it is sure the body of loop is executed once. 'do...while' loop is example of the exit-controlled loop.

- while Loop
- do while Loop
- For Loop

While Loop

Loops generally consist of two parts: one or more *control expressions* which (not surprisingly) control the execution of the loop, and the *body*, which is the statement or set of statements which is executed over and over.

'while' loop is entry-controlled loop, in which first condition is evaluated and if condition is true the body of loop is executed. After executing the body of loop the test condition is again evaluated and if it is true then again body of loop is executed. This process is going to continue up to your test condition is true. If the test condition is false on initial stage then body of loop is never executed. The format of 'while' loop is:

```
while(test condition) {
        body of loop
}
```

Body of loop has one or more statements. Here the curly bracket is not necessary if body of loop contains only one statement.

'While' Loop execution flow chart

Consider following example to print 10 numbers start from 1.

```
main( ) {
        int i;
```

```
        i=1;                        // Initialization of Loop Counter

        while(i <= 10) {            // Test condition
            printf("I = %d\n",i);   // Body of loop
            i++;                    // Increment of Loop counter
        }
}
```

The general syntax of a while loop is

```
    Initialization

while( expression )
{
        Statement1
        Statement2
        Statement3

}
```

The most basic *loop* in C is the while loop. A while loop has one control expression, and executes as long as that expression is true. This example repeatedly doubles the number 2 (2, 4, 8, 16, ...) and prints the resulting numbers as long as they are less than 1000:

```
int x = 2;
while(x < 1000)
        {
        printf("%d\n", x);
        x = x * 2;
        }
```

(Once again, we've used braces {} to enclose the group of statements which are to be executed together as the body of the loop.)

For Loop

Our second loop, which we've seen at least one example of already, is the for loop. The general syntax of a while loop is

```
for( Initialization;expression;Increments/decrements )
{
        Statement1
        Statement2
```

Statement3

}

The first one we saw was:

```
for (i = 0; i < 10; i = i + 1)
        printf ("i is %d\n", i);
```

(Here we see that the for loop has three control expressions. As always, the *statement* can be a brace-enclosed block.)

Do while Loop

This is very similar to the while loop except that the test occurs at the end of the loop body. This guarantees that the loop is executed at least once before continuing. Such a setup is frequently used where data is to be read. The test then verifies the data, and loops back to read again if it was unacceptable.

'do...while' loop is exit-controlled loop, in which first condition body of loop is executed first and then the test condition is evaluated and if the test condition is true then again body of loop is executed. This process is repeated continuously till test condition is true. Here the body of loop must be executed at least once. The format of 'do...while' loop is:

```
do {
        body of loop
}
while (test condition);
```

```
do
{
  printf("Enter 1 for yes, 0 for no :");
     scanf("%d", &input_value);
} while (input_value != 1 && input_value != 0)
```

The break Statement

We have already met break in the discussion of the switch statement. It is used to exit from a loop or a switch, control passing to the first statement beyond the loop or a switch.

With loops, break can be used to force an early exit from the loop, or to implement a loop with a test to exit in the middle of the loop body. A break within a loop should always be protected within an if statement which provides the test to control the exit condition.

The continue Statement

This is similar to break but is encountered less frequently. It only works within loops where its effect is to force an immediate jump to the loop control statement.

- In a while loop, jump to the test statement.
- In a do while loop, jump to the test statement.
- In a for loop, jump to the test, and perform the iteration.

Like a break, continue should be protected by an if statement. You are unlikely to use it very often.

Take the following example:

```
int i;
for (i=0;i<10;i++)
{
if (i==5)
continue;
printf("%d",i);
if (i==8)
break;
}
```

This code will print 1 to 8 except 5.

Continue means, whatever code that follows the continue statement WITHIN the loop code block will not be exectued and the program will go to the next iteration, in this case, when the program reaches i=5 it checks the condition in the if statement and executes 'continue', everything after continue, which are the printf statement, the next if statement, will not be executed.

Break statement will just stop execution of the look and go to the next statement after the loop if any. In this case when i=8 the program will jump out of the loop. Meaning, it won't continue till i=9, 10.

Comment:

- The compiler is "line oriented", and parses your program in a line-by-line fashion.
- There are two kinds of comments: single-line and multi-line comments.
- The single-line comment is indicated by "//"

This means everything after the first occurrence of "//", UP TO THE END OF CURRENT LINE, is ignored.

- The multi-line comment is indicated by the pair "/*" and "*/".

  This means that everything between these two sequences will be ignored. This may ignore any number of lines.

Here is a variant of our first program:

```
/* This is a variant of my first program.
* It is not much, I admit.
*/
int main() {
printf("Hello World!\n");    // that is all?
return(0);
}
```

ARRAY AND STRING

Arrays are widely used data type in 'C' language. It is a collection of elements of similar data type. These similar elements could be of all integers, all floats or all characters. An array of character is called as string whereas and array of integer or float is simply called as an array. So array may be defined as a group of elements that share a common name and that are defined by position or index. The elements of an array are store in sequential order in memory.
There are mainly two types of Arrays are used:

- One dimensional Array
- Multidimensional Array

One dimensional Array

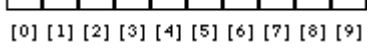
So far, we've been declaring simple variables: the declaration

```
int i;
```

declares a single variable, named i, of type int. It is also possible to declare an *array* of several elements. The declaration

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

declares an array, named a, consisting of ten elements, each of type int. Simply speaking, an array is a variable that can hold more than one value. You specify which of the several values you're referring to at any given time by using a numeric *subscript*. (Arrays in programming are similar to vectors or matrices in mathematics.) We can represent the array a above with a picture like this:



In C, arrays are *zero-based*: the ten elements of a 10-element array are numbered from 0 to 9. The subscript which specifies a single element of an array is simply an integer expression in square brackets. The first element of the array is a[0], the second element is a[1], etc. You can use these ``array subscript expressions" anywhere you can use the name of a simple variable, for example:

```
a[0] = 10;
a[1] = 20;
a[2] = a[0] + a[1];
```

Notice that the subscripted array references (i.e. expressions such as a[0] and a[1]) can appear on either side of the assignment operator. it is possible to initialize some or all elements of an array when the array is defined. The syntax looks like this:

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

The list of values, enclosed in braces {}, separated by commas, provides the initial values for successive elements of the array.

The subscript does not have to be a constant like 0 or 1; it can be any integral expression. For example, it's common to loop over all elements of an array:

```
int i;

for(i = 0; i < 10; i = i + 1)
        a[i] = 0;
```

This loop sets all ten elements of the array a to 0.

Arrays are a real convenience for many problems, but there is not a lot that C will do with them for you automatically. In particular, you can neither set all elements of an array at once nor assign one array to another; both of the assignments

```
a = 0;                  /* WRONG */
```

and

```
int b[10];
b = a;                  /* WRONG */
```

are illegal.

To set all of the elements of an array to some value, you must do so one by one, as in the loop example above. To copy the contents of one array to another, you must again do so one by one:

```
int b[10];

for(i = 0; i < 10; i = i + 1)
        b[i] = a[i];
```

Remember that for an array declared

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

there is no element a[10]; the topmost element is a[9]. This is one reason that zero-based loops are also common in C. Note that the for loop

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

does just what you want in this case: it starts at 0, the number 10 suggests (correctly) that it goes through 10 iterations, but the less-than comparison means that the last trip through the loop has i set to 9. (The comparison i <= 9 would also work, but it would be less clear and therefore poorer style.)

Multidimensional Array

The declaration of an array of arrays looks like this:

```
int a2[5][7];
```

You have to read complicated declarations like these ``inside out.'' What this one says is that a2 is an array of 5 something's, and that each of the something's is an array of 7 ints. More briefly, ``a2 is an array of 5 arrays of 7 ints,'' or, ``a2 is an array of array of int.'' In the declaration of a2, the brackets closest to the identifier a2 tell you what a2 first and foremost is. That's how you know it's an array of 5 arrays of size 7, not the other way around. You can think of a2 as having 5 ``rows'' and 7 ``columns,'' although this interpretation is not mandatory. (You could also treat the ``first'' or inner subscript as ``x'' and the second as ``y.'' Unless you're doing something fancy, all you have to worry about is that the subscripts when you access the array match those that you used when you declared it, as in the examples below.)

To illustrate the use of multidimensional arrays, we might fill in the elements of the above array a2 using this piece of code:

```
int i, j;
for(i = 0; i < 5; i = i + 1)
        {
        for(j = 0; j < 7; j = j + 1)

 a2[i][j] = 10 * i + j;
        }
```

This pair of nested loops sets a[1][2] to 12, a[4][1] to 41, etc. Since the first dimension of a2 is 5, the first subscripting index variable, i, runs from 0 to 4. Similarly, the second subscript varies from 0 to 6.

We could print a2 out (in a two-dimensional way, suggesting its structure) with a similar pair of nested loops:

```
for (i = 0; i < 5; i = i + 1)
        {
        for (j = 0; j < 7; j = j + 1)
                printf ("%d\t", a2[i][j]);
        printf ("\n");
        }
```

(The character \t in the printf string is the tab character.)

Just to see more clearly what's going on, we could make the ``row" and ``column" subscripts explicit by printing them, too:

```
for(j = 0; j < 7; j = j + 1)
        printf("\t%d:", j);
printf ("\n");

for(i = 0; i < 5; i = i + 1)
        {
        printf("%d:", i);
        for(j = 0; j < 7; j = j + 1)
                printf("\t%d", a2[i][j]);
        printf("\n");
        }
```

This last fragment would print

```
      0:    1:    2:    3:    4:    5:    6:
0:    0     1     2     3     4     5     6
1:    10    11    12    13    14    15    16
2:    20    21    22    23    24    25    26
3:    30    31    32    33    34    35    36
4:    40    41    42    43    44    45    46
```

STRING

String are the combination of number of characters these are used to store any word in any variable of constant. A string is an array of character. It is internally represented in system by using ASCII value. Every single character can have its own ASCII value in the system. A character string is stored in one array of character type.

e.g. “Ram” contains ASCII value per location, when we are using strings and then these strings are always terminated by character ‘\0’. We use conversion specifies %s to set any string we can have any string as follows:-

char nm [25].

When we store any value in nm variable then it can hold only 24 character because at the end of the string one character is consumed automatically by ‘\0’.

#include<string.h>

There are some common inbuilt functions to manipulation on string in string.h file. these are as follows:

1. ***strlen*** - *string length*

2. ***strcp****y* - *string copy*

3. ***strcm****p* - *string compare*

4. ***strup****s* - *string upper*

5. ***strlw****r* - *string lower*

6. ***strca****t* - *string concatenate*

FUNCTIONS

Function

A function is a ``black box" that we've locked part of our program into. The idea behind a function is that it *compartmentalizes* part of the program and in particular, that the code within the function has some useful properties:

1. It performs some well-defined task, which will be useful to other parts of the program.
2. It might be useful to other programs as well; that is, we might be able to reuse it (and without having to rewrite it).
3. The rest of the program doesn't have to know the details of how the function is implemented. This can make the rest of the program easier to think about.
4. The function performs its task *well*. It may be written to do a little more than is required by the first program that calls it, with the anticipation that the calling program (or some other program) may later need the extra functionality or improved performance. (It's important that a finished function do its job well, otherwise there might be a reluctance to call it, and it therefore might not achieve the goal of reusability.)
5. By placing the code to perform the useful task into a function, and simply calling the function in the other parts of the program where the task must be performed, the rest of the program becomes clearer: rather than having some large, complicated, difficult-to-understand piece of code repeated wherever the task is being performed, we have a single simple function call, and the name of the function reminds us which task is being performed.
6. Since the rest of the program doesn't have to know the details of how the function is implemented, the rest of the program doesn't care if the function is reimplemented later, in some different way (as long as it continues to perform its same task, of course!). This

means that one part of the program can be rewritten, to improve performance or add a new feature (or simply to fix a bug), without having to rewrite the rest of the program.

Functions are probably the most important weapon in our battle against software complexity. You'll want to learn when it's appropriate to break processing out into functions (and also when it's not), and *how* to set up function interfaces to best achieve the qualities mentioned above: reusability, information hiding, clarity, and maintainability.

So what defines a function? It has a *name* that you call it by, and a list of zero or more *arguments* or *parameters* that you hand to it for it to act on or to direct its work; it has a *body* containing the actual instructions (statements) for carrying out the task the function is supposed to perform; and it may give you back a *return value*, of a particular type.

Here is a very simple function, which accepts one argument, multiplies it by 2, and hands that value back:

```
int multbytwo(int x)
{
        int retval;
        retval = x * 2;
        return retval;
}
```

On the first line we see the return type of the function (int), the name of the function (multbytwo), and a list of the function's arguments, enclosed in parentheses. Each argument has both a name and a type; multbytwo accepts one argument, of type int, named x. The name x is arbitrary, and is used only within the definition of multbytwo. The caller of this function only needs to know that a single argument of type int is expected; the caller does not need to know what name the function will use internally to refer to that argument. (In particular, the caller does not have to pass the value of a variable named x.)

Next we see, surrounded by the familiar braces, the body of the function itself. This function consists of one declaration (of a local variable retval) and two statements. The first statement is a conventional expression statement, which computes and assigns a value to retval, and the second statement is a return statement, which causes the function to return to its caller, and also specifies the value which the function returns to its caller.

The return statement can return the value of any expression, so we don't really need the local retval variable; the function could be collapsed to-

```
int multbytwo(int x)
{
        return x * 2;
}
```

How do we call a function? We've been doing so informally since day one, but now we have a chance to call one that we've written, in full detail. Here is a tiny skeletal program to call multby2:

```
#include <stdio.h>

extern int multbytwo(int);

int main()
{
        int i, j;
        i = 3;
        j = multbytwo(i);
        printf("%d\n", j);
        return 0;
}
```

This looks much like our other test programs, with the exception of the new line

```
extern int multbytwo(int);
```

This is an *external function prototype declaration*. It is an external declaration, in that it declares something which is defined somewhere else. (We've already seen the defining instance of the function multbytwo, but maybe the compiler hasn't seen it yet.) The function prototype declaration contains the three pieces of information about the function that a caller needs to know: the function's name, return type, and argument type(s). Since we don't care what name the multbytwo function will use to refer to its first argument, we don't need to mention it. (On the other hand, if a function takes several arguments, giving them names in the prototype may make it easier to remember which is which, so names may optionally be used in function prototype declarations.) Finally, to remind us that this is an external declaration and not a defining instance, the prototype is preceded by the keyword extern.

The presence of the function prototype declaration lets the compiler know that we intend to call this function, multbytwo. The information in the prototype lets the compiler generate the correct code for calling the function, and also enables the compiler to check up on our code (by making sure, for example, that we pass the correct number of arguments to each function we call).

Down in the body of main, the action of the function call should be obvious: the line

```
j = multbytwo(i);
```

calls multbytwo, passing it the value of i as its argument. When multbytwo returns, the return value is assigned to the variable j. (Notice that the value of main's local variable i will become the value of multbytwo's parameter x; this is absolutely not a problem, and is a normal sort of affair.)

This example is written out in ``longhand,'' to make each step equivalent. The variable i isn't really needed, since we could just as well call

```
        j = multbytwo(3);
```
And the variable j isn't really needed, either, since we could just as well call
```
        printf("%d\n", multbytwo(3));
```

Here, the call to multbytwo is a sub expression which serves as the second argument to printf. The value returned by multbytwo is passed immediately to printf. (Here, as in general, we see the flexibility and generality of expressions in C. An argument passed to a function may be an arbitrarily complex sub expression, and a function call is itself an expression which may be embedded as a sub expression within arbitrarily complicated surrounding expressions.)

We should say a little more about the mechanism by which an argument is passed down from a caller into a function. Formally, C is *call by value*, which means that a function receives *copies* of the values of its arguments. We can illustrate this with an example. Suppose, in our implementation of multbytwo, we had gotten rid of the unnecessary retval variable like this:

```
int multbytwo(int x)
{
        x = x * 2;
        return x;
}
```

Recursive Functions

A recursive function is one which calls itself. This is another complicated idea which you are unlikely to meet frequently. We shall provide some examples to illustrate recursive functions.

Recursive functions are useful in evaluating certain types of mathematical function. You may also encounter certain dynamic data structures such as linked lists or binary trees. Recursion is a very useful way of creating and accessing these structures.

Here is a recursive version of the Fibonacci function. We saw a non recursive version of this earlier.

```
int fib(int num)
/* Fibonacci value of a number */
{       switch(num) {
        case 0:
                return(0);
                break;
        case 1:
                return(1);
                break;
        default:  /* Including recursive calls */
                return(fib(num - 1) + fib(num - 2));
```

```
            break;
        }
}
```

We met another function earlier called power. Here is an alternative recursive version.

```
double power(double val, unsigned pow)
{
    if(pow == 0)  /* pow(x, 0) returns 1 */
        return(1.0);
    else
        return(power(val, pow - 1) * val);
}
```

Notice that each of these definitions incorporate a test. Where an input value gives a trivial result, it is returned directly; otherwise the function calls itself, passing a changed version of the input values. Care must be taken to define functions which will not call themselves indefinitely, otherwise your program will never finish.

The definition of fib is interesting, because it calls itself twice when recursion is used. Consider the effect on program performance of such a function calculating the Fibonacci function of a moderate size number.

Input Value	Number of times `fib` is called
0	1
1	1
2	3
3	5
4	9
5	15
6	25
7	41
8	67
9	109
10	177

If such a function is to be called many times, it is likely to have an adverse effect on program performance.

Don't be frightened by the apparent complexity of recursion. Recursive functions are sometimes the simplest answer to a calculation. However there is always an alternative non-recursive solution available too. This will normally involve the use of a loop, and may lack the elegance of the recursive solution.

Pointer

a pointer is a variable that points to or references a memory location in which data is stored. In the computer, each memory cell has an address that can be used to access that location so a pointer variable points to a memory location we can access and change the contents of this memory location via the pointer.

Pointer declaration:

A pointer is a variable that contains the memory location of another variable in which data is stored. Using pointer, you start by specifying the type of data stored in the location. The asterisk helps to tell the compiler that you are creating a pointer variable. Finally you have to give the name of the variable. The syntax is as shown below.

type * variable name

The following example illustrate the declaration of pointer variable :

```
Int *ptr;
float *string;
```

Address operator:

Once we declare a pointer variable then we must point it to something we can do this by assigning to the pointer the address of the variable you want to point as in the following example:

ptr=#

The above code tells that the address where num is stores into the variable ptr. The variable ptr has the value 21260,if num is stored in memory 21260 address then

The following program illustrate the pointer declaration :

```
/* A program to illustrate pointer declaration*/
main()
{
int *ptr;
int sum;
sum=45;
ptr=&ptr;
printf (”\n Sum is %d\n”, sum);
printf (”\n The sum pointer is %d”, ptr);
}
```

Pointer expressions & pointer arithmetic:

In expressions, like other variables pointer variables can be used. For example if p1 and p2 are properly initialized and declared pointers, then the following statements are valid.

```
y=*p1**p2;
sum=sum+*p1;
z= 5* – *p2/p1;
*p2= *p2 + 10;
```

C allows us to subtract integers to or add integers from pointers as well as to subtract one pointer from the other. We can also use short hand operators with pointers p1+=; sum+=*p2; etc., By using relational operators, we can also compare pointers like the expressions such as p1 >p2 , p1==p2 and p1!=p2 are allowed.

The following program illustrate the pointer expression and pointer arithmetic :

```
/*Program to illustrate the pointer expression and pointer arithmetic*/
#include< stdio.h >
main()
{ int ptr1,ptr2;
int a,b,x,y,z;
a=30;b=6;
ptr1=&a;
ptr2=&b;
x=*ptr1+ *ptr2 6;
y=6*- *ptr1/ *ptr2 +30;
printf(”\nAddress of a +%u”,ptr1);
printf(”\nAddress of b %u”,ptr2);
printf(”\na=%d, b=%d”,a,b);
printf(”\nx=%d,y=%d”,x,y);
ptr1=ptr1 + 70;
ptr2= ptr2;
printf(”\na=%d, b=%d,”a,b);
}
```

Pointers and function:

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

1. **Call by reference**
2. **Call by value.**

Call by value:

We have seen that there will be a link established between the formal and actual parameters when a function is invoked. As soon as temporary storage is created where the value of actual parameters is stored. The formal parameters picks up its value from storage area the mechanism of data transfer between formal and actual parameters allows the actual parameters mechanism of data transfer is referred as call by value. The corresponding formal parameter always represents a local variable in the called function. The current value of the corresponding actual parameter becomes the initial value of formal parameter. In the body of the actual parameter, the value of formal parameter may be changed. In the body of the subprogram, the value of formal parameter may be changed by assignment or input statements. This will not change the value of the actual parameters.

```
/* Include< stdio.h >
void main()
{
int x,y;
x=20;
y=30;
printf(”\n Value of a and b before function call =%d %d”,a,b);
fncn(x,y);
```

```
printf(”\n Value of a and b after function call =%d %d”,a,b);
}
fncn(p,q)
int p,q;
{
p=p+p;
q=q+q;
}
```

Call by Reference:

The address should be pointers, when we pass address to a function the parameters receiving. By using pointers, the process of calling a function to pass the address of the variable is known as call by reference. The function which is called by reference can change the value of the variable used in the call.

```
/* example of call by reference*?
/* Include< stdio.h >
void main()
{
int x,y;
x=20;
y=30;
printf(”\n Value of a and b before function call =%d %d”,a,b);
fncn(&x,&y); printf(”\n Value of a and b after function call =%d %d”,a,b);
}
fncn(p,q)
int p,q;
{
*p=*p+*p;
*q=*q+*q;
}
```

Pointer to arrays:

an array is actually very much similar like pointer. We can declare as int *a is an address, because a[0] the arrays first element as a[0] and *a is also an address the form of declaration is also equivalent. The difference is pointer can appear on the left of the assignment operator and it is a is a variable that is lvalue. The array name cannot appear as the left side of assignment operator and is constant.

```
/* A program to display the contents of array using pointer*/
main()
{
int a[100];
int i,j,n;
printf(”\nEnter the elements of the array\n”);
scanf(%d,&n);
printf(”Enter the array elements”);
for(I=0;I< n;I++)
```

```
scanf(%d,&a[I]);
printf(”Array element are”);
for(ptr=a,ptr< (a+n);ptr++)
printf(”Value of a[%d]=%d stored at address %u”,j+=,*ptr,ptr);
}
```

Pointers and structures :

We know the name of an array stands for address of its zeros element the same concept applies for names of arrays of structures. Suppose item is an array variable of the struct type. Consider the following declaration:

```
struct products
{
char name[30];
int manufac;
float net;
item[2],*ptr;
```

STRUCTURES

What is a Structure?

- Structure is a method of packing the data of different types.
- When we require using a collection of different data items of different data types in that situation we can use a structure.
- A structure is used as a method of handling a group of related data items of different data types.

A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used to select it from the structure. A structure is a convenient way of grouping several pieces of related information together.

A structure can be defined as a new named type, thus extending the number of available types. It can use other structures, arrays or pointers as some of its members, though this can get complicated unless you are careful.

Defining a Structure

A structure type is usually defined near to the start of a file using a typedef statement. typedef defines and names a new type, allowing its use throughout the program. typedefs usually occur just after the #define and #include statements in a file.

Here is an example structure definition.

```
typedef struct {
    char name[64];
```

```
        char course[128];
        int age;
        int year;
  } student;
```

This defines a new type student variables of type student can be declared as follows.

```
  student st_rec;
```

Notice how similar this is to declaring an int or float.

The variable name is st_rec, it has members called name, course, age and year.

Accessing Members of a Structure

Each member of a structure can be used just like a normal variable, but its name will be a bit longer. To return to the examples above, member name of structure st_rec will behave just like a normal array of char, however we refer to it by the name .

```
  st_rec.name
```

Here the dot is an operator which selects a member from a structure.

Where we have a pointer to a structure we could dereference the pointer and then use dot as a member selector. This method is a little clumsy to type. Since selecting a member from a structure pointer happens frequently, it has its own operator -> which acts as follows. Assume that st_ptr is a pointer to a structure of type student We would refer to the name member as.

```
  st_ptr -> name
```

```
/* Example program for using a structure*/
#include< stdio.h >
void main()
{
int id_no;
char name[20];
char address[20];
char combination[3];
int age;
}newstudent;
printf(”Enter the student information”);
printf(”Now Enter the student id_no”);
scanf(“%d”,&newstudent.id_no);
printf(“Enter the name of the student”);
scanf(“%s”,&new student.name);
printf(“Enter the address of the student”);
```

```
scanf("%s",&new student.address);printf("Enter the cmbination of the student");
scanf("%d",&new student.combination);printf(Enter the age of the student");
scanf("%d",&new student.age);
printf("Student information\n");
printf("student id_number=%d\n",newstudent.id_no);
printf("student name=%s\n",newstudent.name);
printf("student Address=%s\n",newstudent.address);
printf("students combination=%s\n",newstudent.combination);
printf("Age of student=%d\n",newstudent.age);
}
```

Arrays of structure:

It is possible to define a array of structures for example if we are maintaining information of all the students in the college and if 100 students are studying in the college. We need to use an array than single variables. We can define an array of structures as shown in the following example:

```
structure information
{
int id_no;
char name[20];
char address[20];
char combination[3];
int age;
}
student[100];
```

An array of structures can be assigned initial values just as any other array can. Remember that each element is a structure that must be assigned corresponding initial values as illustrated below.

```
#include< stdio.h >
{
struct info
{
int id_no;
char name[20];
char address[20];
char combination[3];
int age;
}
struct info std[100];
int I,n;
printf("Enter the number of students");
```

```
scanf(“%d”,&n);
printf(“ Enter Id_no,name address combination age\m”);
for(I=0;I < n;I++)
scanf(“%d%s%s%s%d”,&std[I].id_no,std[I].name,std[I].address,std[I].combination,&std[I].age);
printf(“\n Student information”);
for (I=0;I< n;I++)
printf(“%d%s%s%s%d\n”,
”,std[I].id_no,std[I].name,std[I].address,std[I].combination,std[I].age);
}
```

Structure within a structure:

A structure may be defined as a member of another structure. In such structures the declaration of the embedded structure must appear before the declarations of other structures.

```
struct date
{
int day;
int month;
int year;
};
struct student
{
int id_no;
char name[20];
char address[20];
char combination[3];
int age;
structure date def;
structure date doa;
}oldstudent, newstudent;
```

The structure student contains another structure date as its one of its members.