Computer science 1101 - fundamentals of programming

Programming Fundamentals (University of the People)

Course Title: FUNDAMENTALS OF PROGRAMMING: (30 LECTURER HOURS:
COURSE OUTLINE

		Course content
		Program Planning Concepts: Algorithm; Advantages of Generalized Algorithms; How to Make Algorithms Generalized; Avoiding Infinite Loops in Algorithms
		Different ways of Representing an Algorithm – As a Program, As a Flowchart, As a Pseudo code; Need for Planning a Program before Coding; Program
		Flowcharts, Structure charts, Pseudo codes; Importance of use of Indentation in Programming; Structured Programming Concepts.
		Writing programs using Sequence Logic, Selection Logic and Iteration (or looping) Logic, functions.
		LabWork
		Programming Languages: Programming Language; Types of Programming Languages – Machine-level, Assembly-level and High-level Languages; advantages and Limitations;
		Linker, Interpreter, Intermediate Language Overview of High-level Languages – C, C++, Characteristics of a Good Programming Language
		LabWork
		Labwork
		Program Testing and Debugging: Definition of Testing & Debugging; Difference between Testing and Debugging; Types of Program Errors; Testing a Program; Debugging a Program for Syntax Errors; Debugging a Program for Logic Errors.
		Program Documentation:

		What is Documentation; Need for Documenting Programs and Software; Forms of Documentation – Comments, System Manual, User Manual; Documentation Standards and Notations.
		Revision

s.

Expected outcome:

1. Differentiate the features of 1st, 2nd, 3rd, 4th, 5th and 6th generation programming languages.
2. Explain the motivation for continuing evolution of programming languages with examples including move from procedural to object-oriented, to component-based, and to web services.
3. Design and use simple codes to demonstrate programming concepts.
4. Describe the basic features of a programming language (e.g. C, C++, HTML, Java)
5. Demonstrate the ability to code and run a computer program.

References:

1. Pradeep K. Sinha and Priti Sinha, "Computer Fundamentals: Fourth Edition", BPB Publications,
2007.

2. Behrouz A. Forouzan, Richard F. Gilberg, "COMPUTER SCIENCE – A Structured Programming approach using C", Indian Edition, Thomson 3rd edition

3. Carlo Ghezi, Mehdi Jazayeri, "Programming Language Concepts", John Wiley and Sons

Journals: www.literateprogramming.com/knuthweb.pdf----Literate programming

ejournalsofsciences.org--- Teaching computer programming in 21st century

A Brief History of C

C is a general-purpose language which has been closely associated with the UNIX operating system for which it was developed - since the system and most of the programs that run it are written in C.

Many of the important ideas of C stem from the language BCPL, developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language **B**, which was written by Ken Thompson in 1970 at Bell Labs, for the first UNIX system on a DEC PDP-7. **BCPL** and **B** are "type less" languages whereas C provides a variety of data types.

In 1972 Dennis Ritchie at Bell Labs writes C and in 1978 the publication of The C Programming Language by Kernighan & Ritchie caused a revolution in the computing world.

In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C", was completed late 1988.

Why use C?

C has been used successfully for every type of programming problem imaginable from operating systems to spreadsheets to expert systems - and efficient **compilers** are available for machines ranging in power from the Apple Macintosh to the **Cray** supercomputers. The largest measure of C's success seems to be based on purely practical considerations:

1. **the portability of the compiler;**
2. **the standard library concept;**
3. **a powerful and varied repertoire of operators;**
4. **an elegant syntax;**
5. **ready access to the hardware when needed;**
6. **and the ease with which applications can be optimized by hand-coding isolated procedures**

C is often called a "Middle Level" programming language. This is not a reflection on its lack of programming power but more a reflection on its capability to access the system's low level functions. Most high-level languages (e.g. **FORTRAN**) provides everything the programmer might want to do already built into the language. A low level language (e.g. **assembler**) provides nothing other than access to the machines basic instruction set. A middle level language, such as C, probably doesn't supply all the constructs found in high-languages - but it provides you with all the building blocks that you will need to produce the results you want!

Uses of C

C was initially used for system development work, in particular the programs that make-up the operating system. Why use C? Mainly because it produces code that runs nearly as fast as code written in assembly language. Some examples of the use of C might be:

1. **Operating Systems**
2. **Language Compilers**
3. **Assemblers**
4. **Text Editors**
5. **Print Spoolers**
6. **Network Drivers**
7. **Modern Programs**
8. **Data Bases**
9. **Language Interpreters**
10. **Utilities**

In recent years C has been used as a general-purpose language because of its popularity with programmers. It is not the world's easiest language to learn and you will certainly benefit if you are not learning C as your first programming language! Many well established programmers are switching to C for all sorts of reasons, but mainly because of the portability that writing standard C programs can offer.

C for Personal Computers

With regards to personal computers Microsoft C for IBM (or clones) PC's, and Borland's C are seen to be the two most commonly used systems. However, the latest version of Microsoft C is now considered to be the most powerful and efficient C compiler for personal computers.

Structure of C Programs

Objectives

Having completed this section you should know about:

1. **C's character set**
2. **C's keywords**
3. **The general structure of a C program**
4. **That all C statement must be ended with a semi colon as this one ;**
5. **That C is a free format language**
6. **All C programs us header files that contain standard library functions.**

C's Character Set

C does not use, nor requires the use of, every character found on a modern computer keyboard. The only characters required by the C Programming Language are as follows:

1. **A - Z**
2. **a -z**
3. **0 - 9**
4. **space . , : ; ' $ "**
5. **# % & ! _ {} [] () $$$$ &&&& |**
6. **+ - / * =**

The use of most of this set of characters will be discussed throughout the course.

The form of a C Program

All C programs will consist of at least one function, but it is usual (when your experience grows) to write a C program that comprises several functions. The only function that has to be present is the function called **main**. For more advanced programs the **main** function will act as a controlling function calling other functions in their turn to do the dirty work! The **main** function is the first function that is called when your program executes.

The layout of C Programs

The general form of a **C** program is as follows:

pre-processor directives

global declarations (if any)

main()

{ This is the area where declaration is made:

local variables to function main ;

statements associated with function main ;

}

Note the use of the bracket set () and {}. () are used in conjunction with function names whereas {} are used as to delimit the C statements that are associated with that function. Also note the semicolon (;) which is used to terminate C statements. C is a free format language and long statements can be continued, without truncation, onto the next line. The **semicolon** informs the C compiler that the end of the statement has been reached. Free format also means that you can add as many spaces as you like to improve the look of your programs.

A very common mistake made by everyone, who is new to the C programming language, is to miss off the semicolon. The C compiler will **concatenate** the various lines of the program together and then tries to understand them - which it will not be able to do. The error message produced by the compiler will relate to a line of you program which could be some distance from the initial mistake.

Pre-processor Directives

C is a small language but provides the programmer with all the tools to be able to write powerful programs. Some people don't like C because it is too primitive! Look again at the set of **keywords** that comprises the C language and see if you can find a command that allows you to print to the computer's screen the result of, say, a simple calculation. Don't look too hard because it doesn't exist.

It would be very tedious, for all of us, if every time we wanted to communicate with the computer we all had to write our own output functions. Fortunately, we do not have to. C uses libraries of standard functions which are included when we build our programs. For the novice C programmer one of the many questions always asked *is does a function already exist for what I want to do?* Only experience will help here but we do include a function listing as part of this course.

All programs you will write will need to communicate to the outside world - I don't think I can think of a program that doesn't need to tell someone an answer. So all our C programs will need at least one of C's standard libraries which deals with standard inputting and outputting of data. This library is called **stdin.h** and it is declared in our programs before the **main** function. The .h extension indicates that this is a header file.

I have already mentioned that C is a free format language and that you can layout your programs how you want to use as much white space as you like. The only exception are statements associated with the pre-processor.

All pre-processor directives begin with a # and the must start in the first column. The commonest directive to all C programs is:

#include <stdio.h>

Note the use of the angle brackets ($$$$ and &&&&) around the header's name. These indicate that the header file is to be looked for on the system disk which stores the rest of the C program application. Some text books will show the above statement as follows:

#include "stdio.h"

The double quotes indicate that the current working directory should be searched for the required header file. This will be true when you write your own header files but the standard header files should always have the angle brackets around them.

NOTE: just to keep you on your toes - pre-processor statements, such as **include**, however, and this is very important, DO NOT use semi-colons as delimiters! But also don't forget the # must be in the first column.

Your First Program

Objectives

Having read this section you should have an understanding of:

1. a pre-processor directive that must be present in all your C programs.
2. a simple C function used to write information to your screen.
3. how to add comments to your programs

Now that you've seen the compiler in action it's time for you to write your very own first C program. You can probably guess what it's going to be - the program that everyone writes just to check they understand the very, very, very basics of what is going on in a new language.

Yes - it's the ubiquitous **"Hello World"** program. All your first program is going to do is print the message "**Hello World**" on the screen.

The program is a short one, to say the least. Here it is:

```
#include <stdio.h>

main()

{ printf("Hello World\n"); }
```

The first line is the standard start for all C programs - **main ()**. After this comes the program's only instruction enclosed in curly brackets **{}**. The curly brackets mark the start and end of the list of instructions that make up the program - in this case just one instruction.

Notice the semicolon marking the end of the instruction. You might as well get into the habit of ending every C instruction with a semicolon - it will save you a lot of trouble! Also notice that the semicolon marks the end of an instruction - it isn't a separator as is the custom in other languages.

If you're puzzled about why the curly brackets are on separate lines I'd better tell you that it's just a layout convention to help you spot matching brackets. C is very unfussy about the way you lay it out. For example, you could enter the **Hello World** program as:

main (){printf("Hello World\n");} *However, this is unusual and poor way of program layout.*

The **printf** function does what its name suggest it does: it prints, on the screen, whatever you tell it to. The **"\n"** is a special symbols that forces a new line on the screen.

OK, that's enough explanation of our first program! Type it in and save it as **Hello.c**. Then use the compiler to compile it, then the linker to link it and finally run it. The output is as follows:

Hello World

Adding Comments to your Program

A **comment** is a note to yourself (or others) that you put into your source code. All comments are ignored by the compiler. They exist solely for your benefit. Comments are used primarily to document the meaning and purpose of your source code, so that you can remember later how it functions and how to use it. You can also use a comment to temporarily remove a line of code. Simply surround the line(s) with the comment symbols.

In **C**, the start of a comment is signaled by the **/*** character pair. A comment is ended by ***/**. For example, this is a syntactically correct C comment:

/* This is a comment. */

Comments can extend over several lines and can go anywhere except in the middle of any C keyword, function name or variable name. In C you can't have one comment within another comment. That is comments may not be nested. Let’s now look at our first program one last time but this time with comments:

main() /* main function heading */

{ printf("\n Hello, World! \n"); /* Display message on */ } /* the screen */

This program is not large enough to warrant comment statements but the principle is still the same.

Data Types

Objectives

Having read this section you should be able to:

1. declare (name) a local variable as being one of C's five data types
2. initialise local variables
3. perform simple arithmetic using local variables

Now we have to start looking into the details of the C language. How easy you find the rest of this section will depend on whether you have ever programmed before - no matter what the language was. There are a great many ideas common to programming in any language and C is no exception to this rule.

So if you haven't programmed before, you need to take the rest of this section slowly and keep going over it until it makes sense. If, on the other hand, you have programmed before you'll be wondering what all the fuss is about It's a lot like being able to ride a bike!

The first thing you need to know is that you can create *variables* to store *values* in. A variable is just a named area of storage that can hold a single value (numeric or character). C is very fussy about how you create variables and what you store in them. It demands that you declare the name of each variable that you are going to use and its *type*, or *class*, before you actually try to do anything with it.

In this section we are only going to be discussing *local* variables. These are variables that are used within the current program unit (or function) in a later section we will looking at *global* variables - variables that are available to all the program's functions.

There are **five** basic data types associated with variables:

1. **int** - integer: a whole number.
2. **float** - floating point value: ie a number with a fractional part.
3. **double** - a double-precision floating point value.
4. **char** - a single character.
5. **void** - valueless special purpose type which we will examine closely in later sections.

One of the confusing things about the C language is that the range of values and the amount of storage that each of these types takes is not defined. This is because in each case the 'natural' choice is made for each type of machine. You can call *variables* what you like, although it helps if you give them sensible names that give you a hint of what they're being used for - names like sum, total, average and so on. If you are translating a formula then use variable names that reflect the elements used in the formula. For example, 2(r (that should read as "2 pi r" but that depends upon how your browser has been set-up) would give local variables names of **pi** and **r**. Remember, C programmers tend to prefer short names!

The Important of Flowcharts and Pseudocode in Computer Programming

Flowcharts are diagram that indicate relationship between two or more phenomena. They are used to demonstrate associations, structures of organization or process and association between different phenomena (Algorithms & Pseudo-code, 2010). Flowcharts represent information in a simple and diagrammatical form and also present the sequences of how information is linked to one another to form a system. Pseudocodes on the other hand are high level description of algorithms in ways that is easier for humans to understand. Pseudocodes eliminate data that is irrelevant to human and retains only that which humans are able to comprehend.

Pseudocodes are used in computer programming to describe and document programming algorithm in ways that computer programmers are able to understand them (Algorithms & Pseudo-code, 2010). Pseudocodes are not meant for computer interpretation but for human interpretation and therefore carries only data that makes sense to humans. Pseudocodes assist a computer programmer to describe a programming language especially when using a program he or she is not familiar with. By interpreting the pseudocodes the computer programmer will be able to describe the programming language and come up with a structuring approach for the program.

Pseudocodes do not obey the syntax rules of any language, therefore making them universal. This means that any programmer can be able to interpret the pseudocodes regardless of the language he or she is using. Pseudocodes also eliminate bias and bulkiness brought about by talkative words (Algorithms & Pseudo-code, 2010).

Pseudocodes can be interpreted objectively by any professional or trained computer program. This is because the codes are usually universal making it easier to interpret the codes objectively as oppose to when using words which can be translated subjectively. In computer programming flowcharts are graphical alternative to pseudocodes (Moursund, 2006). Flowcharts are usually in the form of a diagram that represents an algorithm or process. They are usually useful in showing the steps involved in each program and how each steps transits into the next. Flowcharts give step by step solution a particular problem.

In the case of designing a program, the flowcharts will provide step by step details of how the program begins to operate to when the program produces its output. The flowcharts are also useful in providing process information about what happens at each step of the algorithm. The descriptions of these processes are contained in the boxes representation of the flow charts. In computer programming these boxes are usually of two types. The rectangular boxes which usually denote a process or a step and a diamond box which usually denote a decision. Flowcharts are also characterized by arrows. The arrows depict the direction of movements (flow) of a particular process.

Start and end symbols are represented as ovals or cycles in computer programming flowcharts. Flowcharts are also useful in designing, analyzing, managing and documentation programming process of function (Brent, 2009). Through flowcharts a computer programmer is able to plan how he or she is going to execute the designing process of a program. The flowcharts will provide the programmer with a systematic procedure to follow while carrying out the designing process. The flowcharts will also be useful in making visible to the programmer on what is expected at every level of program and what must be included to make his program design a functioning one.

At the same time flowcharts and pseudocodes as well provides a means for the computer programmer to document the process he or she has used to contract a program. It is an essential requirement that when computer program design a program he or she must document the design approach he has used. The flowcharts and pseudocodes are the most suitable media through which to document this information. Flowcharts are also very useful in assisting programming experts to analyze an already existing program (Brent, 2009).
Looking at a program based on the output it produces will tell little about how the program functions and how it was constructed. In order to understand these basic foundations of the program, the user must get a description from the programmer of how this program works. Description in words will be too cumbersome and sometime may not drive some points home. The flowcharts provide some sort of a blueprint that describes how the program was constructed and how it functions in order to produce results. Flowcharts help the program analyzer to visualize and have an overview of how the entire program functions in order to enhance his understanding of the program.

This is very useful when trying to analyze a particular program. Similarly, you may be in an organization and you are charged with the responsibility of managing a particular program that is essential to that organization. In order to conduct your duty effectively you must be in a position of understanding how the program functions. The flowcharts and pseudocodes will provide you with a "manual" through which you will be able to know how the program runs. This is will come in handy during trouble shooting when the program malfunctions.

Modern computers can carry out very complex tasks. Each task consists of well-defined procedures known as algorithms. Learn how computers use algorithms to perform the tasks we expect them to do.

What is an Algorithm?

Consider how you use a computer in a typical day. For example, you start working on a report, and once you have completed a paragraph, you perform a spell check. You open up a spreadsheet application to do some financial projections to see if you can afford a new car loan. You use a web browser to search online for a kind of car you want to buy.

You may not think about this very consciously, but all of these operations performed by your computer consist of algorithms. An **algorithm** is a well-defined procedure that allows a computer to solve a problem. Another way to describe an algorithm is a sequence of unambiguous instructions. The use of the term 'unambiguous' indicates that there is no room for subjective interpretation. Every time you ask your computer to carry out the same algorithm, it will do it in exactly the same manner with the exact same result.

Consider the earlier examples again. Spell checking uses algorithms. Financial calculations use algorithms. A search engine uses algorithms. In fact, it is difficult to think of a task performed by your computer that does not use algorithms.

How Do Algorithms Work?

A very simple example of an algorithm would be to find the largest number in an unsorted list of numbers. If you were given a list of five different numbers, you would have this figured out in no time, no computer needed. Now, how about five million different numbers? Clearly, you are going to need a computer to do this, and a computer needs an algorithm.

Below is what the algorithm could look like. Let's say the input consists of a list of numbers, and this list is called L. The number L1 would be the first number in the list, L2 the second number, etc. And we know the list is not sorted - otherwise, the answer would be really easy. So, the input to the algorithm is a list of numbers, and the output should be the largest number in the list.

The algorithm would look something like this:

```
Step 1: Let Largest = L1
```

This means you start by assuming that the first number is the largest number.

Step 2: For each item in the list:

This means you will go through the list of numbers one by one.

Step 3: If the itemLargest:

If you find a new largest number, move to step four. If not, go back to step two, which means you move on to the next number in the list.

Step 4: Then Largest = the item

This replaces the old largest number with the new largest number you just found. Once this is completed, return to step two until there are no more numbers left in the list.

Step 5: Return Largest

This produces the desired result.

You can also represent the above algorithm with a flowchart

Notice that the algorithm is described as a series of logical steps in a language that is easily understood. For a computer to actually use these instructions, they need to be written in a language that a computer can understand, known as a **programming language**.

Variables

A variable can be defined in many ways. At the basic level, a variable can be defined as a memory location declared to store any kind of data (which may change many times during program execution). It can also be defined as any quantity/entity which may vary during program execution. To identify a variable (the declared memory location), it can be assigned a name – known as variable name. ***The name given to a variable is known as an identifier***. An identifier can be a combination of alphabets, digits and underscore. There are certain set of rules which must be observed while naming a variable.

Rules for naming a variable:-

- A variable name (identifier) can be any combination of alphabets, digits and underscore.
- First character should be a letter (alphabet).
- Length of variable name can range from 1 to 8. (**Note:** Different compilers may allow different ranges, say up to 31. But it is a good practice to keep the variable name short.)
- A space in between is not allowed. **Ex:** A variable name cannot be declared as **var name**
- Underscore can be used to concatenate name combinations. **Ex**: **var_name** , **var_123** are valid identifiers.
- No commas or other special characters (other than underscore _) are allowed in a variable name.

- C is a case sensitive language – which means a variable name declared as **flag** is not same as **FLAG**. They both will be treated as different variables.
- There are certain reserved words in C language, known as keywords. Words similar to a keyword cannot be used as a variable name. Keywords such as **int**, **char**, **float** etc. are keywords that you can't declare a variable with names **int**, **char** or **float**.

There might be many reasons for the success of a language, but one obvious reason is the characteristics of the language. Several characteristics believed to be important for making a programming language good are:

- **Simplicity**: A good programming language must be simple and easy to learn and use. It should provide a programmer with a clear, simple and unified set of concepts, which can be easily grasped. The overall simplicity of a programming language strongly affects the readability of the programs written in that language, and programs, which are easier to read and understand, are also easier to maintain. It is also easy to develop and implement a compiler or an interpreter for a programming language, which is simple. However, the power needed for the language should not be sacrificed for simplicity.
- **Naturalness**: - A good language should be natural for the application area, for which it has been designed. That is, it should provide appropriate operators, data structures, control structures, and a natural syntax to facilitate the users to code their problem easily and efficiently.
- **Abstraction**: - Abstraction means the ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored. The degree of abstraction allowed by a programming language directly effects its writability. Object oriented language support high degree of abstraction. Hence, writing programs in object oriented language is much easier. Object oriented language also support re usability of program segments due to this features.
- **Efficiency: -** Programs written in a good programming language are efficiently translated into machine code, are efficiently executed, and acquire as little space in the memory as possible. That is a good programming language is supported with a good language translator which gives due consideration to space and time efficiency.
- **Structured**: - Structured means that the language should have necessary features to allow its users to write their programs based on the concepts of structured programming. This property of a moreover, it forces a programmer to look at a problem in a logical way, so that fewer errors are created while writing a program for the problem.
- **Compactness: -** In a good programming language, programmers should be able to express intended operations concisely. A verbose language is generally not liked by programmers, because they need to write too much.
- **Locality**: - A good programming language should be such that while writing a programmer concentrate almost solely on the part of the program around the statement currently being worked with.

Also read about character set in C

Variable Declaration

A variable must be declared first before we can use it in a program for manipulations.

A variable is declared with its storage class, data type and identifier. The format is shown below:-

Storage-class Data-Type Variable-name;

There are 4 storage classes namely Automatic, Static, External and Register storage classes. Each of them has its own meaning and usage. Storage class is used to attribute certain features to a variable like its scope (local or global), where to store the variable (in memory or in register), the life time of a variable etc. It is not necessary to specify a storage class while declaring a variable. By default all variable declarations (without any storage class specified) will be assigned to **"automatic storage class".**

So our variable declaration would be like:-

Data-Type Variable-name;

Examples:-

int a;

int count;

char name;

Two or more variables of the same data type can be declared in a single line, separating each variable names with a comma and ending the line with a semicolon.

Examples:-

int n, count, flag,i,j;

char name, address, nick_nm;

Initial values can be assigned to variables while declaring it.

Examples:-

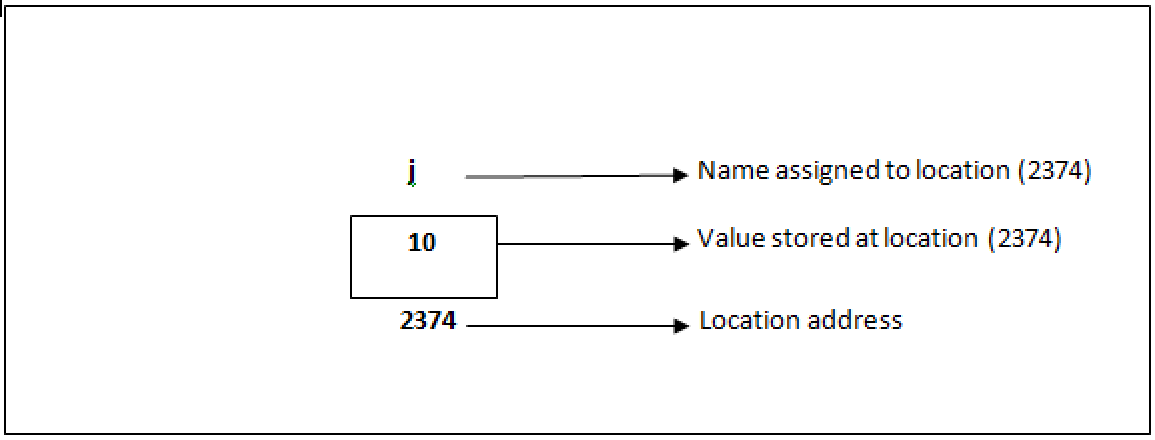
int a, num, count=0,flag=10,mark=100;

Examine what happens when we declare a variable.

int j=10;

Here we have declared a variable of data **type integer** with **name as j** and initial **value as 10.** This declaration tells the C compiler to:-

- Reserve space in memory to hold the integer value.
- Assign the name 'j' to that reserved memory space.
- Store the value 10 in this memory location.


j
Name assigned to location (2374)
10
Value stored at location (2374)
2374
Location address

Keywords

Keywords are reserved words in C which has a predefined meaning. The compiler already knows the meaning of these words and it will do a particular operation according to the meaning of the keyword.

Example: - **int** – is a keyword used to identify an integer quantity. When we declare a variable as **int a;** the compiler assumes that variable **a** is used to store an integer quantity.

There are **32 keywords** in C language. All of them are listed in the table below.

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

Take a note of the following:

- A keyword name cannot be used as a variable name.
- Keywords must be written in lower case.

C TOKENS:

- C tokens are the basic building blocks in **C** language which are constructed together to write a C program.
- Each and every smallest individual units in a C program are known as C tokens.

C tokens are of six types. They are,

1. Keywords—->eg: int, while
2. Identifiers—->eg: total
3. Constants—->eg: 10, 20
4. Strings———>eg: “total”, “hello”
5. Special symbols—->eg: ( ), {}
6. Operators———→eg: +, /,-,*

C TOKENS EXAMPLE PROGRAM:

A programming **token** is the basic component of source code. Characters are categorized as one of five classes of **tokens** that describe their functions (**constants, identifiers, operators, reserved words, and separators**) in accordance with the rules of the programming language.

```
int main()
{
 int x, y, sum;
 x = 10, y = 20;
 total = x + y;
 printf ("sum = %d \n", sum);
}
```

where,

- main – identifier
- {,}, (,) – delimiter
- int – keyword
- x, y, sum – identifier
- main, {, }, (,), int, x, y, sum – tokens

There are also symbolic constants that are used in C:

Symbolic constant in c Language

A symbolic constant is name that substitute for a sequence of character that cannot be changed. The character may represent a numeric constant, a character constant, or a string. When the program is compiled, each occurrence of a symbolic constant is replaced by its corresponding character sequence. They are usually defined at the beginning of the program. The symbolic constants may then appear later in the program in place of the numeric constants, character constants, etc., that the symbolic constants represent.

A symbolic constant is name that substitute for a sequence of character that cannot be changed. The character may represent a numeric constant, a character constant, or a string. When the program is compiled, each occurrence of a symbolic constant is replaced by its corresponding character sequence.

They are defined at the beginning of any program.

Example:-

a C program consists of the following symbolic constant definitions.

#define symbolic_name value

For example:

```
#define PI 3.141593

#define TRUE 1

#define FALSE 0
```

#define PI 3.141593 defines a symbolic constant PI whose value is 3.141593.

A sample program is mentioned below:

C PROGRAM:

```
#include<stdio.h>

#define TRUE 1

#define PI 3.141593

void main()

{ float a,  b,  c, float d=PI;

if(TRUE)

{

a=100;
b=a*10;

c=b-a;

}

printf("\na=%f\nb=%f\nc=%f\nPI=%f",a,b,c,d);
```

Control structures and statements in C

Control structures form the basic entities of a "**structured programming language**". You should know languages like C/C++ or Java are all structured programming languages. ***Control structures*** are used to alter the flow of execution of the program**.** Why do we need to alter the program flow? The reason is "***decision making***"! In life, we may be given a set of option like doing "Electronics" or "Computer science". We do make a decision by analyzing certain conditions (like our personal interest, scope of job opportunities etc). With the decision we make, we alter the flow of our life's direction. This is exactly what happens in a C/C++ program. We use control structures to make decisions and alter the direction of program flow in one or the other path(s) available.

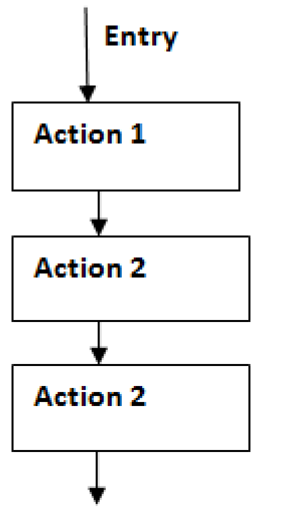
There are **three** types of control structures available in C and C++, that is, in Modular programming.

1) Sequence structure (straight line paths)

2) Selection structure (one or many branches)

3) Loop structure (repetition of a set of activities)

All the **3** control structures and its flow of execution is represented in the flow charts given below.



Sequence Structure

Control statements in C/C++ to implement control structures

We have to keep in mind one important fact: - all program processes can be implemented with these 3 control structures only. That's why I said "***control structures are the basic entities of a structured programming language***". To implement these "control structures" in a C/C++ program, the language provides 'control statements'. So to implement a particular control structure in a programming language, we need to learn how to use the relevant control statements in that particular language.

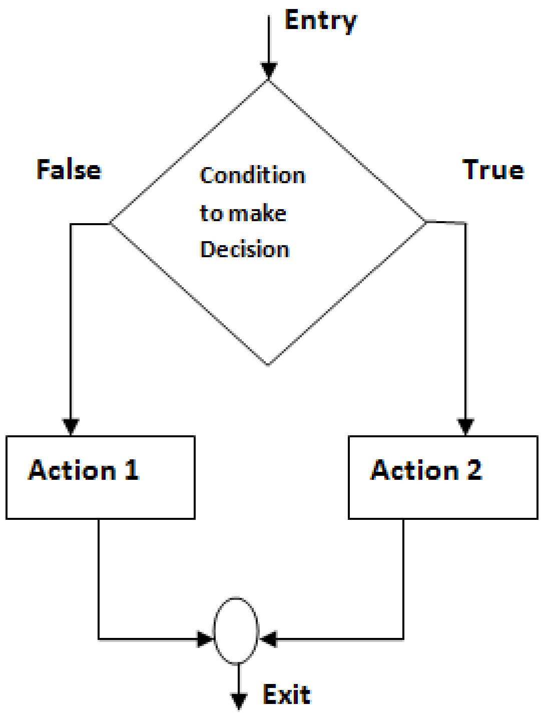
The control statements are:-

- **Switch**
- **If**
- **If Else**
- **While**
- **Do While**
- **For**

As shown in the flow charts:-

- Selection structures are implemented using **If**, **If Else** and **Switch** statements.
- Looping structures are implemented using **While**, **Do While** and **For** statements.

Selection structures



Selection Structure

Implemented using:- If and If..else control statements

switch is used for multi branching

Selection structures are used to perform '**decision making** 'and then branch the program flow based on the outcome of decision making. Selection structures are implemented in C/C++ with If, If Else and Switch statements. *if and if else* statements are 2 way branching statements whereas Switch is a multi-branching statement.

The simple If statement

The syntax format of a simple if statement is as shown below.

```
if (expression) // This expression is evaluated. If expression is TRUE statements inside the braces will,be,executed
{
statement1;
statement2;
}
statement 1;// Program control is transferred directly to this line, if the expression is FALSE
statement 2;
```

The expression given inside the brackets after **if** is evaluated first. If the expression is true, then statements inside the curly braces that follow if(expression) will be executed. If the expression is false, the statements inside curly braces will not be executed and program control goes directly to statements after curly braces.

Example program to demo "If" statement

A simple example program to demo the use of If, If-Else and Switch is shown here. An integer If the integer entered by user is 1 – output on screen "UNITED STATES". If the integer is 2 – output "SPAIN", If the integer is 3 output "INDIA". If the user enters some other value – output "WRONG ENTRY".

Note:- The same problem is used to develop example programs for **"if else"** and **"switch"** statements. Follow the curly braces/brackets very keenly.

```
#include
void main()
{
int num;
printf("Hello user, Enter a number");
scanf("%d",&num); // Collects the number from user
if(num==1)
{
printf("UNITED STATES");
}
if(num==2)
{
printf("SPAIN");
}
if(num==3)
{
printf("INDIA");
}
}
```

The If Else statement.

Syntax format for If Else statement is shown below.

```
if(expression 1)// Expression 1 is evaluated. If TRUE, statements inside the curly braces are executed.
{ //If FALSE program control is transferred to immediate else if statement.

statement1;
statement2;
}
else if(expression 2)// If expression 1 is FALSE, expression 2 is evaluated.
{
statement1;
statement2;
}
else if(expression 3) // If expression 2 is FALSE, expression 3 is evaluated
{
statement1;
statement2;
}
else // If all expressions (1, 2 and 3) are FALSE, the statements that follow this else (inside curly braces) is executed.

{
statement1;
statement2;
}
other statements;
```

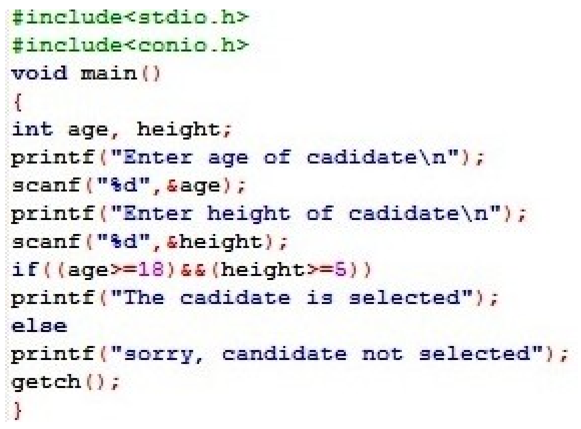
The execution begins by evaluation expression 1. If it is **TRUE**, then statements inside the immediate curly braces is evaluated. If it is **FALSE**, program control is transferred directly to immediate else if statement. Here expression 2 is evaluated for TRUE or FALSE. The process continues. If all expressions inside the different if and else if statements are FALSE, then the last **else** statement (without any expression) is executed along with the statements 1 and 2 inside the curly braces of last **else** statement.

This is another example of *else if* statement:

```
#include <stdio.h>

int main(void){

int num;

printf("Enter your mark ");

scanf("%d",&num);

printf(" You entered %d", num); // printing outputs

        if(num >= 80)

        { printf(" You got A grade"); // printing outputs

                }

        else if ( num >=60)

               { printf(" You got B grade"); }  // Note the space between else & if

        else if ( num >=40)

              { printf(" You got C grade");
```

```
                }
        else if ( num < 40)
                { printf(" You Failed in this exam");   }
return 0;
}
```

You can also use logical operators as shown below:



```
#include<stdio.h>
#include<conio.h>
void main()
{
int age, height;
printf("Enter age of cadidate\n");
scanf("%d",&age);
printf("Enter height of cadidate\n");
scanf("%d",&height);
if((age>=18)&&(height>=5))
printf("The cadidate is selected");
else
printf("sorry, candidate not selected");
getch();
}
```

Switch statement:

Switch is a multi-branching control statement. **Syntax for switch statement is shown below.**

switch (expression)// Expression is evaluated. The outcome of the expression should be an integer or a character constant
{
case value1: // case is the keyword used to match the integer/character constant from, expression.
//value1, value2 ... are different possible values that can come in expression
statement1;
statement2;
break; // break is a keyword used to break the program control from switch block.
casevalue2:
statement1;
statement2;
break;
default: // default is a keyword used to execute a set of statements inside switch, if no case values match the expression value.
statement1;
statement2;
break;
}

Execution of switch statement begins by evaluating the expression inside the switch keyword brackets. The expression should be an integer (1, 2, 100, 57 etc.) or a character constant like **'a'**,

'b' etc. This expression's value is then matched with each case values. There can be any number of case values inside a switch statements block. If first case value is not matched with the expression value, program control moves to next case value and so on. **When a case value matches with expression value, the statements that belong to a particular case value are executed.**

Notice that last set of lines that begins with **default**. The word **default** is a **keyword in C/C++**. When used inside switch block, it is intended to execute a set of statements, if no case values matches with expression value. So if no case values are matched with expression value, the set of statements that follow **default:** will get executed.

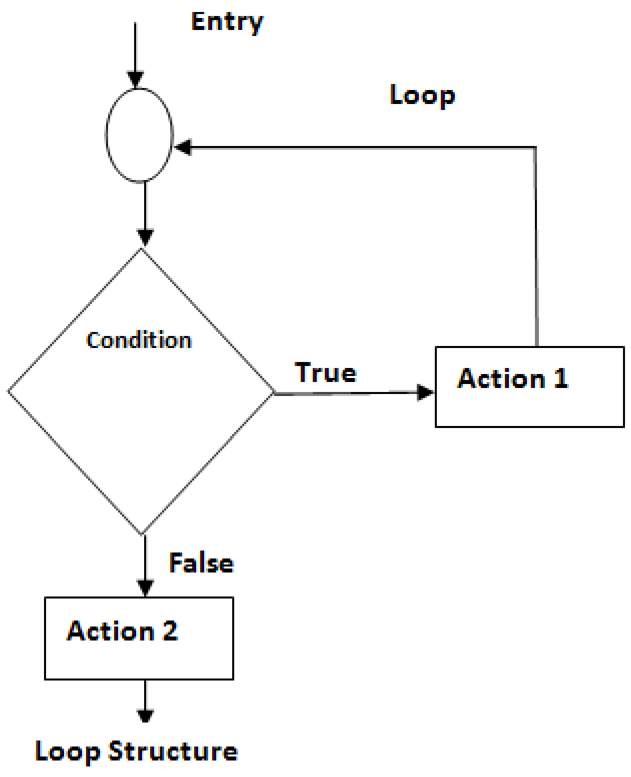
Note: Notice the **break** statement used at the end of each case values set of statements. The word break is a **keyword in C/C++** used to break from a block of curly braces. The switch block has two curly braces { }. The keyword break causes program control to exit from switch block.

Example program to demonstrate the working of "switch"

```
#include<stdio.h>
void main()
{
int num;
printf("Hello user, Enter a number");
scanf("%d",&num);  // Collects the number from user
switch(num)
{
case 1:
printf("UNITED STATES");
break;
case 2:
printf("SPAIN");
break;
case 3:
printf("INDIA");
default:
printf("WRONG ENTRY");
}
}
```

Note: - Switch statement is used for multiple branching. The same can be implemented using **nested "If Else"** statements. But use of nested if else statements make program writing tedious and complex. Switch makes it much easier. Compare this program with above one. Below is a loop where iteration takes place, that is, the repetition of the loop.

Loop structures



Implemented using:- While , Do While and For control statements

Loop Structure

A loop structure is used to execute a certain set of actions for a predefined number of times or until a particular condition is satisfied. There are 3 control statements available in C/C++ to implement loop structures. **While, Do while and For statements.**

The while statement

Syntax for while loop is shown below:

while (**condition**) // This condition is tested for TRUE or FALSE. Statements inside curly braces are executed as long as condition is TRUE
{
statement 1;
statement 2;
statement 3;
}

The condition is checked for TRUE first. If it is TRUE then all statements inside curly braces are executed. Then program control comes back to check the condition has changed or to check if it is still TRUE. The statements inside braces are executed repeatedly, as long as the condition is TRUE. When the condition turns FALSE, program control exits from while loop.

Note:- while is an entry controlled loop. Statement inside braces are allowed to execute only if condition inside while is TRUE.

Example program to demonstrate the working of “while loop”

An example program to collect a number from user and then print all numbers from zero to that particular collected number is shown below. That is, if user enters 10 as input, then numbers from 0 to 10 will be printed on screen.

Note: - The same problem is used to develop programs for do while and for loops

```
#include
void main()
{
int num;
int count=0;           // count is initialized as zero to start printing from zero.
printf("Hello user, Enter a number");
scanf("%d",&num);      // Collects the number from user
while(count<=num)   // Checks the condition - if value of count has reached value of num or not.
{ printf("%d",count); count=count+1; // value of count is incremented by 1 to print next
number. } }
```

Syntax for do while loop is shown below:

```
do
{
statement 1;
statement 2;
statement 3;
}
while(condition);
```

Unlike while, do while is an exit controlled loop. Here the set of statements inside braces are executed first. The condition inside while is checked only after finishing the first time execution of statements inside braces. If the condition is TRUE, then statements are executed again. This process continues as long as condition is TRUE. Program control exits the loop once the condition turns FALSE.

Example program to demonstrate the working of "do while"

```
#include<stdio.h>
void main()
{
int num;
int count=0; // count is initialized as zero to start printing from zero.
printf("Hello user, Enter a number");
scanf("%d",&num); // Collects the number from user
do
{
printf("%d", count); // Here value of count is printed for one time intially and then only
condition is checked.
count=count+1; // value of count is incremented by 1 to print next number.
}    while(count<=num); }
```

This is how you can display numbers from 10 to 1:

```
#include <stdio.h>

int main ()

{     int n = 10;

   do

   {   printf(" %d", n);

       n--;

   } while (n > 0);

}
```

The for loop statement

Syntax of *for loop* statement is shown below:

```
for(initialization statements; test condition; iteration statements)
{
statement 1;
statement 2;
statement 3;
}
```

The ***for*** statement is an entry controlled loop. The difference between while and for is in the number of repetitions. The ***for*** *loop* is used when an action is to be executed for a predefined number of times. The while loop is used when the number of repetitions is not predefined.

The program control enters the ***for*** *loop*. At first it execute the statements given as initialization statements. Then the condition statement is evaluated. If conditions are TRUE, then the block of statements inside curly braces is executed. After executing curly brace statements fully, the control moves to the "iteration" statements. After executing iteration statements, control comes back to condition statements. Condition statements are evaluated again for TRUE or FALSE. If TRUE the curly brace statements are executed. This process continues until the condition turns FALSE.

Note 1:- The statements given as "***initialization statements***" are executed only once, at the beginning of a ***for*** loop.
Note 2: There are 3 statements given to a ***for*** loop as shown. One for ***initialization*** purpose, other for ***condition testing*** and last one ***for iterating the loop***. Each of these 3 statements are separated by semicolons.

Example program to demo working of "for loop"

```
#include<stdio.h>
void main()
{
int num,count;
printf("Hello user, Enter a number");
scanf("%d",&num);   // Collects the number from user
for(count=0;count<=num;count++)  // count is initialized to zero inside for statement. The
condition is checked and statements are executed.

{ printf("%d",count);     // Values from 0 are printed. } }
```

If you wish to display 10 numbers, this is how you would do it:

```
#include <stdio.h>

void main()

{        int i;

          printf("The first 10 natural numbers are:\n");

          for (i=1;i<=10;i++)

          {     printf("%d ",i);

          }

}
```

The output would be:

1 2 3 4 5 6 7 8 9 10

To do the same using a while loop:

```
#include <stdio.h>

int main ()

{    int i = 1;

   while (i <=10)

   {       printf(" %d", i);

      i++;

   }
```

The output would be:

```
1 2 3 4 5 6 7 8 9 10
```

To display the numbers and their sum and average, it is done this way:

```
#include <stdio.h>

void main()

{       int i, avg, sum = 0;

           printf("The first 10 natural numbers are:\n");

           for (i=1; i<=10; i++)

           {  printf (" %d", i);

           }

           for(i=1; i<=10;  i++)

           {

           sum=sum + i; }

           avg=sum/10;
```

```
        printf ("\nThe sum of these numbers is = %d, and average is = %d", sum, avg);

}
```

Structured programming (modular 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.

Why C is called a structured programming language?

C is called a structured programming language because to solve a large problem, C programming language divides the problem into smaller modules called functions or procedures each of which handles a particular responsibility. The program which solves the entire problem is a collection of such functions. There are advantages and disadvantages of C, such as:

Advantages

- C as a structured programming is simple and easy to understand and implement.
- It is well suited for small size implementation. However this is not restricted. A good design can extend it to large size implementation.
- Programmers do not require to know complex design concepts to start a new program.

Disadvantages

- Data and methods cannot be bind together in a module.
- Polymorphism and inheritance are not available.
- Complex design and full object oriented design cannot be implemented.
- Programmers generally prefer object oriented programming language over structured programming language when implementing a complex gaming applications or front end business applications.

However, C is the mother of all languages and few reasons to consider learning C is that it makes your fundamentals very strong.

Apart from that, C offers a very flexible memory management. Memory is allocated statically, automatically, or dynamically in C programming with the help of malloc and calloc functions. This gives a choice and control to programmers as to when and where to allocate and deallocate memory.

C was designed to implement the Unix OS (Which also forms a base for Linux) and C sits close to the OS which makes it an efficient language because of its efficient system level resource management. Another excellent usage of C is that C programming language is not limited to but used widely in operating systems, language compilers, network drivers, language interpreters, and system utilities areas of development.

Agreed that there are many better programming languages out there now with the advent of technologies - however C offers some very good features / advantages like:

1. C is ubiquitous. Whatever the platform, C is probably available.
2. C is portable. Write a piece of clean C, and it compiles with minimal modifications on other platforms - sometimes it even works out-of-the-box.
3. Simple! C is really simple to learn and practically does not require any dependencies. A simple PC with the compiler and you are good to go to make programs.
4. As mentioned earlier since C is widely used for writing drivers, compilers etc, C is versatile!
5. FREE
6. Compatible and Powerful.

Benefits of C

- As a middle level language, C combines the features of both high level and low level languages. It can be used for low-level programming, such as scripting for drivers and kernels and it also supports functions of high level programming languages, such as scripting for software applications etc.
- C is a structured programming language which allows a complex program to be broken into simpler programs called functions. It also allows free movement of data across these functions.
- C language is case-sensitive.
- C is highly portable and is used for scripting system applications which form a major part of Windows, UNIX and Linux operating system.
- C is a general purpose programming language and can efficiently work on enterprise applications, games, graphics, and applications requiring calculations.
- C language has a rich library which provides a number of built-in functions. It also offers dynamic memory allocation.
- **Real-World Applications of C**

1. Operating Systems:

Scripting of UNIX operating system was the primary purpose behind creation of C. Additionally, as programs scripted in C get executed with speeds equivalent to assembly language, C language has been an integral part of the development of multiple operating systems. Unix-Kernel, Microsoft Windows utilities and operating system applications, and a large segment of the Android operating system have all been scripted in C.

2. Development of New Languages:

Efficiency of code execution and simplicity have resulted in C directly or indirectly influencing development of many languages including C++ which is C with classes, C#, D, Java, Limbo, JavaScript, Perl, UNIX's C Shell, PHP and Python, and Verilog. These languages use C in variable capacity: for instance, in Python, C is used for building standard libraries, while others like C++, Perl and PHP have syntax and control structures based upon C.

3. Computation Platforms:

C implements algorithms and data structures swiftly, facilitating faster computations in programs. This has enabled the use of C in applications requiring higher degrees of calculations like MATLAB and Mathematica.

4. Embedded Systems:

Various features of C including direct access to machine level hardware APIs, presence of C compilers, deterministic resource use and dynamic memory allocation make C language an optimum choice for scripting applications and drivers of embedded systems.

5. Graphics and Games:

C language has been used in the development of a variety of graphics and gaming applications, such as chess, bouncing ball, archery etc.

Let us review C one more time

Every full **C** program begins inside a function called "main". A function is simply a collection of commands that do "something". The main function is always called when the program first executes. From main, we can call other functions, whether they be written by us or by others or use built-in language features. To access the standard functions that comes with your compiler, you need to include a header with the #include directive. What this does is effectively take everything in the header and paste it into your program. Let's look at a working program:

```
#include <stdio.h>

int main()

{   printf( "I am alive!  Beware.\n”);

   getchar();

   return 0; }
```

Let's look at the elements of the program. The #include is a "preprocessor" directive that tells the compiler to put code from the header called stdio.h into our program before actually creating the executable. By including header files, you can gain access to many different functions--both the printf and getchar functions are included in stdio.h.

The next important line is int main (). This line tells the compiler that there is a function named main, and that the function returns an integer, hence int. The "curly braces" ({and}) signal the beginning and end of functions and other code blocks. If you have programmed in Pascal, you will know them as BEGIN and END. Even if you haven't programmed in Pascal, this is a good way to think about their meaning.

The printf( ) function is the standard C way of displaying output on the screen. The quotes tell the compiler that you want to output the literal string as-is (almost). The '\n' sequence is actually treated as a single character that stands for a newline (we'll talk about this later in more detail); for the time being, just remember that there are a few sequences that, when they appear in a string literal, are actually not displayed literally by printf() function and that '\n' is one of them. The actual effect of '\n' is to move the cursor on your screen to the next line. Notice the semicolon: it tells the compiler that you're at the end of a command, such as a function call. You will see that the semicolon is used to end many lines in C.

The next command is getchar(). This is another function call: it reads in a single character and waits for the user to hit enter before reading the character. This line is included because many compiler environments will open a new console window, run the program, and then close the window before you can see the output. This command keeps that window from closing because the program is not done yet because it waits for you to hit enter. Including that line gives you time to see the program run.

Finally, at the end of the program, we return a value from main to the operating system by using the return statement. This return value is important as it can be used to tell the Operating System whether our program succeeded or not. A return value of 0 means success.

The final brace closes off the function. You should try compiling this program and running it. You can cut and paste the code into a file, save it as a .c file, and then compile it. If you are using a command-line compiler, such as Borland C++ 5.5, you should read the compiler instructions for information on how to compile. Otherwise compiling and running should be as simple as clicking a button with your mouse (perhaps the "build" or "run" button).

You might start playing around with the printf() function and get used to writing simple C programs.

Explaining your Code

Comments are critical for all but the most trivial programs and this tutorial will often use them to explain sections of code. When you tell the compiler a section of text is a comment, it will ignore it when running the code, allowing you to use any text you want to describe the real code. To create a comment in C, you surround the text with /* and then */ to block off everything between as a comment. Certain compiler environments or text editors will change the color of a commented area to make it easier to spot, but some will not. Be certain not to accidentally comment out code (that is, to tell the compiler part of your code is a comment) you need for the program.

When you are learning to program, it is also useful to comment out sections of code in order to see how the output is affected.

Using Variables

So far you should be able to write a simple program to display information typed in by you, the programmer and to describe your program with comments. That's great, but what about interacting with your user? Fortunately, it is also possible for your program to accept input.

But first, before you try to receive input, you must have a place to store that input. In programming, input and data are stored in variables. There are several different types of variables; when you tell the compiler you are declaring a variable, you must include the data type along with the name of the variable. Several basic types include char, int, and float. Each type can store different types of data.

A variable of type char stores a single character, variables of type int store integers (numbers without decimal places), and variables of type float store numbers with decimal places. Each of these variable types - char, int, and float - is each a keyword that you use when you declare a variable. Some variables also use more of the computer's memory to store their values.

It may seem strange to have multiple variable types when it seems like some variable types are redundant. But using the right variable size can be important for making your program efficient because some variables require more memory than others. For now, suffice it to say that the different variable types will almost all be used!

Before you can use a variable, you must tell the compiler about it by declaring it and telling the compiler about what its "type" is. To declare a variable you use the syntax <variable type> <name of variable>;. (The brackets here indicate that your replace the expression with text described within the brackets.) For instance, a basic variable declaration might look like this:

int myVariable;

Note once again the use of a semicolon at the end of the line. Even though we're not calling a function, a semicolon is still required at the end of the "expression". This code would create a variable called myVariable; now we are free to use myVariable later in the program.

It is permissible to declare multiple variables of the same type on the same line; each one should be separated by a comma. For example, int a, b, c, d, e;

If you attempt to use an undefined variable, your program will not run, and you will receive an error message informing you that you have made a mistake.

Here are some variable declaration examples:

int x;

```
int a, b, c, d;

char letter;

float the_float;
```

While you can have multiple variables of the same type, you cannot have multiple variables with the same name. Moreover, you cannot have variables and functions with the same name.

A final restriction on variables is that variable declarations must come before other types of statements in the given "code block" (a code block is just a segment of code surrounded by { and }). So in C you must declare all of your variables before you do anything else:

Wrong code

```
#include <stdio.h>

int main()

{

   /* wrong!  The variable declaration must appear first */

   printf( "Declare x next" );

   int x;

   return 0;

}
```

Let us fix the above problem with the following code:

```
 #include <stdio.h>

int main()

{    int x;

   printf( "Declare x first" );

   return 0;

}
```

Reading the input:

Using variables in C for input or output can be a bit of a hassle at first, but bear with it and it will make sense. We'll be using the scanf function to read in a value and then printf to read it back out. Let's look at the program and then pick apart exactly what's going on. You can even compile this and run it if it helps you follow along.

```
#include <stdio.h>

int main()

  {     int number;

    printf( "Please enter a number: " );

    scanf( "%d", &number );

    printf( "You entered %d", number );

    getchar();

    return 0;

  }
```

So what does all of this mean? We've seen the #include and main function before; main must appear in every program you intend to run, and the #include gives us access to printf (as well as scanf). (As you might have guessed, the io in stdio.h stands for "input/output"; std just stands for "standard.") The keyword int declares ***number*** to be an integer.

This is where things start to get interesting: the scanf function works by taking a string and some variables modified with &. The string tells scanf what variables to look for: notice that we have a string containing only "%d" -- this tells the scanf function to read in an integer. The second argument of scanf is the variable, sort of. We'll learn more about what is going on later, but the gist of it is that scanf needs to know where the variable is stored in order to change its value. Using & in front of a variable allows you to get its location and give that to scanf instead of the value of the variable. Think of it like giving someone directions to the soda aisle and letting them go get a coca-cola instead of fetching the coke for that person. The & gives the scanf function directions to the variable.

When the program runs, each call to scanf checks its own input string to see what kinds of input to expect, and then stores the value input into the variable.

The second printf statement also contains the same '%d'--both scanf and printf use the same format for indicating values embedded in strings. In this case, printf takes the first argument after the string, the variable ***number***, and treats it as though it were of the type specified by the "format specifier". In this case, printf treats ***number*** as an integer based on the format specifier.

So what does it mean to treat a number as an integer? If the user attempts to type in a decimal number, it will be truncated (that is, the decimal component of the number will be ignored) when stored in the variable. Try typing in a sequence of characters or a decimal number when you run the example program; the response will vary from input to input, but in no case is it particularly pretty.

Of course, no matter what type you use, variables are uninteresting without the ability to modify them. Several operators used with variables include the following: *, -, +, /, =, ==, >, <. The * multiplies, the / divides, the - subtracts, and the + adds. It is of course important to realize that to modify the value of a variable inside the program it is rather important to use the equal sign. In some languages, the equal sign compares the value of the left and right values, but in C == is used for that task. The equal sign is still extremely useful. It sets the value of the variable on the left side of the equals sign equal to the value on the right side of the equals sign. The operators that perform mathematical functions should be used on the right side of an equal sign in order to assign the result to a variable on the left side.

Here are a few examples:

a = 4 * 6; /* (Note use of comments and of semicolon) a is 24 */

a = a + 5; /* a equals the original value of a with five added to it */

a == 5   /* Does NOT assign five to a. Rather, it checks to see if a equals 5.*/

The other form of equal, ==, is not a way to assign a value to a variable. Rather, it checks to see if the variables are equal. It is extremely useful in many areas of C; for example, you will often use == in such constructions as conditional statements and loops. You can probably guess how < and > function. They are greater than and less than operators.

For example:

a < 5 /* Checks to see if a is less than five */

a > 5 /* Checks to see if a is greater than five */

a == 5 /* Checks to see if a equals five, for good measure */

If statement in C

The ability to control the flow of your program, letting it make decisions on what code to execute, is valuable to the programmer. The *if* statement allows you to control if a program enters a section of code or not based on whether a given condition is true or false. One of the important functions of the *if* statement is that it allows the program to select an action based upon the user's input. For example, by using an *if* statement to check a user-entered password, your program can decide whether a user is allowed access to the program.

Without a conditional statement such as the *if* statement, programs would run almost the exact same way every time, always following the same sequence of function calls. If statements allow the flow of the program to be changed, this leads to more interesting code.

Before discussing the actual structure of the *if* statement, let us examine the meaning of TRUE and FALSE in computer terminology. A true statement is one that evaluates to a nonzero number. A false statement evaluates to zero. When you perform comparison with the relational operators, the operator will return 1 if the comparison is true, or 0 if the comparison is false. For example, the check 0 == 2 evaluates to 0. The check 2 == 2 evaluates to a 1. If this confuses you, try to use a printf statement to output the result of those various comparisons (for example printf ("%d", 2 == 1);)

When programming, the aim of the program will often require the checking of one value stored by a variable against another value to determine whether one is larger, smaller, or equal to the other.

There are a number of operators that allow these checks.

Here are the relational operators, as they are known, along with examples:

>	greater than	5 > 4 is TRUE
<	less than	4 < 5 is TRUE
>=	greater than or equal	4 >= 4 is TRUE
<=	less than or equal	3 <= 4 is TRUE
==	equal to	5 == 5 is TRUE
!=	not equal to	5 != 4 is TRUE

It is highly probable that you have seen these before, probably with slightly different symbols. They should not present any hindrance to understanding. Now that you understand TRUE and FALSE well as the comparison operators, let us look at the actual structure of if statements.

Basic *If* Syntax

The structure of an *if* statement is as follows:

```
if ( statement is TRUE )
   Execute this line of code
```

Here is a simple example that shows the syntax:

```
if ( 5 < 10 )
   printf( "Five is now less than ten, that's a big surprise" );
```

Here, we're just evaluating the statement, "*is five less than ten*", to see if it is true or not; with any luck, it is! If you want, you can write your own full program including stdio.h and put this in the main function and run it to test.

To have more than one statement execute after an if statement that evaluates to true, use braces, like we did with the body of the main function. Anything inside braces is called a compound statement, or a block. When using if statements, the code that depends on the if statement is called the "body" of the if statement.

For example:

```
if ( TRUE ) {     /* between the braces is the body of the if statement */

    Execute all statements inside the body

}
```

I recommend always putting braces following if statements. If you do this, you never have to remember to put them in when you want more than one statement to be executed, and you make the body of the if statement more visually clear.

else

Sometimes when the condition in an ***if*** statement evaluates to false, it would be nice to execute some code instead of the code executed when the statement evaluates to true. The "else" statement effectively says that whatever code after it (whether a single line or code between brackets) is executed if the if statement is FALSE.

It can look like this:

```
if ( TRUE ) {     /* Execute these statements if TRUE */

}

else {     /* Execute these statements if FALSE */

}
```

else if

Another use of else is when there are multiple conditional statements that may all evaluate to true, yet you want only one if statement's body to execute. You can use an "else if" statement following an if statement and its body; that way, if the first statement is true, the "else if" will be ignored, but if the if statement is false, it will then check the condition for the else if statement. If the if statement was true the else statement will not be checked. It is possible to use numerous else if statements to ensure that only one block of code is executed.
Let's look at a simple program for you to try out on your own.

```
#include <stdio.h>

int main()                        /* Most important part of the program!  */

{     int age;                     /* Need a variable... */
```

```
     printf( "Please enter your age" );  /* Asks for age */
    scanf( "%d", &age );                /* The input is put in age */
    if ( age < 100 )
  {                 /* If the age is less than 100 */
       printf ("You are still a young person!\n" ); /* Just to show you it works... */
    }
    else if ( age == 100 ) {            /* I use else just to show an example */
       printf( "You are old\n" );
    }
    else {
       printf( "You are really old\n" );     /* Executed if no other statement is */
    }
  return 0;
}
```

Continuation of Boolean operators.

Boolean operators allow you to create more complex conditional statements. For example, if you wish to check if a variable is both greater than five and less than ten, you could use the Boolean AND to ensure both var > 5 and var < 10 are true. In the following discussion of Boolean operators, I will capitalize the Boolean operators in order to distinguish them from normal English. The actual C operators of equivalent function will be described further along into the tutorial - the C symbols are not: OR, AND, NOT, although they are of equivalent function.

When using if statements, you will often wish to check multiple different conditions. You must understand the Boolean operators OR, NOT, and AND. The Boolean operators function in a similar way to the comparison operators: each returns 0 if evaluates to FALSE or 1 if it evaluates to TRUE.
NOT: The NOT operator accepts one input. If that input is TRUE, it returns FALSE, and if that input is FALSE, it returns TRUE. For example, NOT (1) evaluates to 0, and NOT (0) evaluates to 1. NOT (any number but zero) evaluates to 0. In C NOT is written as !. NOT is evaluated prior to both AND and OR.

AND: This is another important command. AND returns TRUE if both inputs are TRUE (if 'this' AND 'that' are true). (1) AND (0) would evaluate to zero because one of the inputs is false (both must be TRUE for it to evaluate to TRUE). (1) AND (1) evaluates to 1. (Any number but 0) AND (0) evaluates to 0. The AND operator is written && in C. Do not be confused by thinking it checks equality between numbers: it does not. Keep in mind that the AND operator is evaluated before the OR operator.

OR: Very useful is the OR statement! If either (or both) of the two values it checks are TRUE then it returns TRUE. For example, (1) OR (0) evaluates to 1. (0) OR (0) evaluates to 0. The OR is written as || in C. Those are the pipe characters. On your keyboard, they may look like a stretched colon. On my computer the pipe shares its key with \. Keep in mind that OR will be evaluated after AND.

It is possible to combine several Boolean operators in a single statement; often you will find doing so to be of great value when creating complex expressions for if statements. What is! (1 && 0)? Of course, it would be TRUE. It is true is because 1 && 0 evaluates to 0 and!0 evaluates to TRUE (i.e., 1).

Try some of these - they're not too hard:

A. !(1 || 0)   ANSWER: 0

B. !(1 || 1 && 0)   ANSWER: 0 (AND is evaluated before OR)

LOOPS

Loops are used to repeat a block of code. Being able to have your program repeatedly execute a block of code is one of the most basic but useful tasks in programming -- many programs or websites that produce extremely complex output (such as a message board) are really only executing a single task many times. (They may be executing a small number of tasks, but in principle, to produce a list of messages only requires repeating the operation of reading in some data and displaying it.) Now, think about what this means: a loop lets you write a very simple statement to produce a significantly greater result simply by repetition.

You should understand the concept of C's true and false, because it will be necessary when working with loops (the conditions are the same as with if statements). There are three types of loops: **for, while**, and **do..while**. Each of them has their specific uses. They are all outlined below.

for- for loops are the most useful type. The syntax for a for loop is

for (*variable initialization; condition; variable update*) {

  Code to execute while the condition is true

}

The variable initialization allows you to either declare a variable and give it a value or give a value to an already existing variable. Second, the condition tells the program that while the conditional expression is true the loop should continue to repeat itself. The variable update section is the easiest way for a for loop to handle changing of the variable. It is possible to do things like x++, x = x + 10, or even x = random (5), and if you really wanted to, you could call other functions that do nothing to the variable but still have a useful effect on the code. Notice that a semicolon separates each of these sections and that is important. Also note that every single one of the sections may be empty, though the semicolons still have to be there. If the condition is empty, it is evaluated as true and the loop will repeat until something else stops it.

Example:

```
#include <stdio.h>

int main()

{    int x;

   /* The loop goes while x < 10, and x increases by one every loop*/

   for ( x = 0; x < 10; x++ ) {

       /* Keep in mind that the loop condition checks

          the conditional statement before it loops again.

          consequently, when x equals 10 the loop breaks.

          x is updated before the condition is checked. */

       printf( "%d\n", x );

   }

}
```

This program is a very simple example of a *for loop*. X is set to zero, while x is less than 10 it calls printf to display the value of the variable x, and it adds 1 to x until the condition is met. Keep in mind also that the variable is incremented after the code in the loop is run for the first time.

WHILE - WHILE loops are very simple. The basic structure is
while (condition) { Code to execute while the condition is true } The true represents a boolean expression which could be x == 1 or while (x != 7) (x does not equal 7). It can be any combination of boolean statements that are legal. Even, (while x ==5 || v == 7) which says execute the code while x equals five or while v equals 7. Notice that a while loop is like a stripped-down version of a *for loop*-- it has no initialization or update section. However, an empty condition is not legal for a while loop as it is with a *for loop*.

Example:

```
#include <stdio.h>

int main()
```

```
{   int x = 0;  /* Don't forget to declare variables */
     while ( x < 10 ) { /* While x is less than 10 */
    printf( "%d\n", x );
    x++;            /* Update x so the condition can be met eventually */
  }
}
```

Here is another example, with the output shown in descending order, and also going downwords.

```
//example program to illustrate while looping
#include <stdio.h>
int main ()
{
    int n = 10;
    while (n > 0)
    {
        printf("%d", n);
        printf("\n");
        n--;
    }
}
```

The output of the program is:

```
 10
  9
  8
  7
  6
  5
  4
  3
  2
  1
```

This was another simple example, but it is longer than the above FOR loop. The easiest way to think of the loop is that when it reaches the brace at the end it jumps back up to the beginning of the loop, which checks the condition again and decides whether to repeat the block another time, or stop and move to the next statement after the block.

DO..WHILE - DO..WHILE loops are useful for things that want to loop at least once. The structure is:

```
do {

} while ( condition );
```

Notice that the condition is tested at the end of the block instead of the beginning, so the block will be executed at least once. If the condition is true, we jump back to the beginning of the block and execute it again. A do..while loop is almost the same as a while loop except that the loop body is guaranteed to execute at least once. A while loop says "Loop while the condition is true, and execute this block of code", a *do..while loop* says "Execute this block of code, and then continue to loop while the condition is true".

Example:

```
#include <stdio.h>

int main()

{

  int x;

  x = 0;

  do {

    /* "Hello, world!" is printed at least one time

      even though the condition is false */

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

  }

 while ( x != 0 );

}
```

Another example of a do-while loop

```
//example program to illustrate do-while looping
#include <stdio.h>

int main ()
{
    int n = 10;
    do
    {
        printf(" %d", n);
        n--;
    }while (n > 0);
}
```

The output would be: 10 9 8 7 6 5 4 3 2 1

Keep in mind that you must include a trailing semi-colon after the while in the above example. A common error is to forget that a *do..while loop* must be terminated with a semicolon (the other loops should not be terminated with a semicolon, adding to the confusion). Notice that this loop will execute once, because it automatically executes before checking the condition.

Break and Continue

Two keywords that are very important to looping are *break* and *continue*. The break command will exit the most immediately surrounding loop regardless of what the conditions of the loop are. *Break* is useful if we want to exit a loop under special circumstances

Continue is another keyword that controls the flow of loops. If you are executing a loop and hit a continue statement, the loop will stop its current iteration, update itself (in the case of for loops) and begin to execute again from the top. In other words, the control is back to the beginning of the loop. Essentially, the continue statement is saying "this iteration of the loop is done, let's continue with the loop without executing whatever code comes after me."

Basic commands in C programming to write basic C Program:

Below are few commands and syntax used in C programming to write a simple C program. Let’s see all the sections of a simple C program line by line.

S.no	Command	Explanation
1	#include <stdio.h>	This is a preprocessor command that includes standard input output header file(stdio.h) from the C library before compiling a C program
2	int main()	This is the main function from where execution of any C program begins.
3	{	This indicates the beginning of the main function.
4	/* some comments */	whatever is given inside the command “/* */” in any C program, won’t be considered for compilation and execution.
5	printf(“Hello World! “);	printf command prints the output onto the screen.
6	getch();	This command waits for any character input from keyboard.
7	return 0;	This command terminates C program (main function) and returns 0.
8	}	This indicates the end of the main function.

1. C Basic Program:

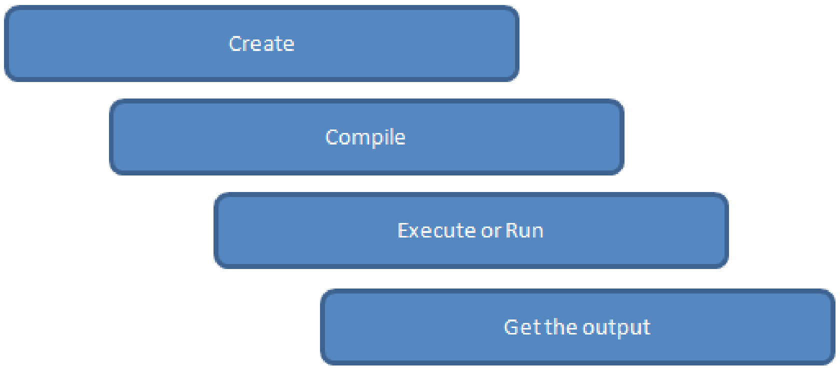
```
#include <stdio.h>
int main()
{
/* Our first simple C basic program */
printf("Hello!");

return 0;
}
                                    .
```

Output:

```
Hello!
```

2. Steps to write C programs and get the output:

Below are the steps to be followed for any C program to create and get the output. This is common to all C program and there is no exception whether it's a very small C program or very large C program.



Basic structure of C program:

Structure of C program is defined by set of rules called protocol, to be followed by programmer while writing C program. All C programs are having sections/parts which are mentioned below.

1. Documentation section
2. Link Section
3. Definition Section
4. Global declaration section
5. Function prototype declaration section
6. Main function
7. User defined function definition section

Example C program to compare all the sections:

You can compare all the sections of a C program with the below C program.

```
*/#include <stdio.h> /* Link section */
int total = 0; /* Global declaration and definition section */
int sum (int, int); /* Function declaration section */
int main () /* Main function */
{
printf (“This is a C basic program \n”);
total = sum (4, 6);
printf (“Sum of two numbers : %d \n”, total);
return 0;
}
int sum (int a, int b) /* User defined function */
{ /* definition section */
return a + b;
}
```

Output:

```
This is a C basic program
Sum of two numbers : 10
```

Description for each section of a C program:

- Let us see about each section of a C basic program in detail below.
- Please note that a C program may not have all below mentioned sections except main function and link sections.
- Also, a C program structure mayn’t be in below mentioned order.

S.No	Sections	Description
1	Documentation section	We can give comments about the program, creation or modified date, author name etc in this section. The characters or words or anything which are given between “/*” and “*/”, won’t be considered by C compiler for compilation process. These will be ignored by C compiler during compilation. Example : /* comment line1 comment line2 comment 3 */
2	Link Section	Header files that are required to execute a C program are included in this section

3	Definition Section	In this section, variables are defined and values are set to these variables.
4	Global declaration section	Global variables are defined in this section. When a variable is to be used throughout the program, can be defined in this section.
5	Function prototype declaration section	Function prototype gives many information about a function like return type, parameter names used inside the function.
6	Main function	Every C program is started from main function and this function contains two major sections called declaration section and executable section.
7	User defined function section	User can define their own functions in this section which perform particular task as per the user requirement.

C – printf and scanf

- printf() and scanf() functions are inbuilt library functions in C which are available in C library by default. These functions are declared and related macros are defined in “stdio.h” which is a header file.
- We have to include “stdio.h” file as shown in below C program to make use of these printf() and scanf() library functions.

1. C printf() function:

- printf() function is used to print the “character, string, float, integer, octal and hexadecimal values” onto the output screen.
- We use printf() function with %d format specifier to display the value of an integer variable.
- Similarly %c is used to display character, %f for float variable, %s for string variable, %lf for double and %x for hexadecimal variable.
- To generate a newline,we use “\n” in C printf() statement.

Note:

- C language is case sensitive. For example, printf() and scanf() are different from Printf() and Scanf(). All characters in printf() and scanf() functions must be in lower case.

2. C scanf() function:

- scanf() function is used to read character, string, numeric data from keyboard
- Consider below example program where user enters a character. This value is assigned to the variable “ch” and then displayed.

- Then, user enters a string and this value is assigned to the variable “str” and then displayed.

Example program for printf() and scanf() functions in C:

```
#include <stdio.h>
int main()
{
char ch;
char str[100];
printf(“Enter any character \n”);
scanf(“%c”, &ch);
printf(“Entered character is %c \n”, ch);
printf(“Enter any string ( upto 100 character ) \n”);
scanf(“%s”, &str);
printf(“Entered string is %s \n”, str);
}
```

Output :

```
Enter any character
a
Entered character is a
Enter any string ( upto 100 character )
hai
Entered string is hai
```

- The format specifier %d is used in scanf() statement. So that, the value entered is received as an integer and %s for string.
- Ampersand is used before variable name “ch” in scanf() statement as &ch.
- It is just like in a pointer which is used to point to the variable.

1. Example program for local variable in C:

- The scope of local variables will be within the function only.
- These variables are declared within the function and can’t be accessed outside the function.
- In the below example, m and n variables are having scope within the main function only. These are not visible to another function.

C – Decision Control statement.

- In decision control statements (C if else and nested if), group of statements are executed when condition is true. If condition is false, then else part statements are executed.

- There are 3 types of decision making control statements in C language. They are,

1. if statements
2. if else statements
3. nested if statements

"If", "else" and "nested if" decision control statements in C:

- Syntax for each C decision control statements are given in below table with description.

• Decision control statements	Syntax	Description

if	if (condition)
	{ Statements; }
		In these type of statements, if condition is true, then respective block of code is executed.

if…else	if (condition)
	{ Statement1; Statement2; }
	else
	{ Statement3; Statement4; }
		In these type of statements, group of statements are executed when condition is true. If condition is false, then else part statements are executed.

nested if	if (condition1)
	{ Statement1; }
	else if(condition2)
	{ Statement2; }
	else Statement 3;
		If condition 1 is false, then condition 2 is checked and statements are executed if it is true. If condition 2 also gets failure, then else part is executed.

Example program for if statement in C:

In "if" control statement, respective block of code is executed when condition is true.

```
int main()

{  int m=40,n=40;

  if (m == n)

  {  printf("m and n are equal");

  }

}
```

Output:

```
m and n are equal
```

Example program for if else statement in C:

In C if else control statement, group of statements are executed when condition is true. If condition is false, then else part statements are executed.

```
#include <stdio.h>

int main()

{  int m=40,n=20;

  if (m == n)

  {  printf("m and n are equal");

  }

  else   {

  printf("m and n are not equal");

  }

 }
```

Output:

```
m and n are not equal
```

Example program for nested if statement in C:

- In “nested if” control statement, if condition 1 is false, then condition 2 is checked and statements are executed if it is true.
- If condition 2 also fails, then else part is executed.

```
#include <stdio.h>

int main()

{  int m=40,n=20;

  if (m>n)

{printf("m is greater than n");

  }

  else if(m<n)

{   printf("m is less than n");
```

```
 }
 else
{  printf("m is equal to n");
 }
}
```

Output:

```
m is greater than n
```

C – Loop control statements

Loop control statements in C are used to perform looping operations until the given condition is true. Control comes out of the loop statements once condition becomes false.

Types of loop control statements in C:

There are 3 types of loop control statements in C language. They are,

1. for
2. while
3. do-while

- Syntax for each C loop control statements are given in below table with description.

S.no	Loop Name	Syntax	Description

1	for	for (exp1; exp2; expr3)
		{ statements; }
			Where,
			exp1 – variable initialization
			(Example: i=0, j=2, k=3)
			exp2 – condition checking
			(Example: i>5, j<3, k=3)
			exp3 – increment/decrement
			(Example: ++i, j–, ++k)

2	while	while (condition)
		{ statements; }
			where,
			condition might be a>5, i<10

3	do while	do { statements; }
		while (condition);
			where,
			condition might be a>5, i<10

Example program (for loop) in C:

In for loop control statement, loop is executed until condition becomes false.

```
#include <stdio.h>
 int main()
{
  int i;
   for(i=0;i<10;i++)
  {     printf("%d ",i);
  }
 }
```

Output:

```
0 1 2 3 4 5 6 7 8 9
```

Once again here is yet another example of a for loop:

What would be the output of this program when executed?

```
//example program to illustrate for looping
#include <stdio.h>

int main ()
{
    int n;
    for(n = 10; n>0 ; n--){
        printf("%d",n);
        printf("\n");
    }
}
```

Take a look at this one:

```
//example program to illustrate more than one statement using the comma
// for looping
#include <stdio.h>

int main ()
{
    int a, b;
    for (a = 1, b = 4; a < b; a++, b--)
    {
        printf("a = %d \n", a);
        printf("b = %d \n", b);
    }
}
```

The output of the program is:

```
a = 1
b = 4
a = 2
b = 3
```

Here is a nested for loop:

```
// nested loops
#include <stdio.h>

int main ()
{
    int i, j;
    for (i = 0; i < 8; i++)
    {
        for (j = i; j < 8; j++)
            printf(".");
        printf("\n");
    }
}
```

Here, two ***for*** loops are nested. The number times inner loop iterates depends on the value of `i` in outer loop.

The output of the program is:

```
........
.......
......
.....
....
...
..
.
```

Or it could be values as:

```
// nested loops
#include <stdio.h>

int main ()
{
    int i, j;
    for (i = 0; i < 8; i++)
    {
        for (j = i; j < 8; j++)
            printf("%d", j);
        printf("\n");
    }
}
```

The output would be:

```
01234567
1234567
234567
34567
4567
567
67
7
```

Example program (while loop) in C:

In while loop control statement, loop is executed until condition becomes false.

```
#include <stdio.h>

 int main()

{   int i=3;

   while(i<10)

  { printf("%d\n",i);

i++;  }    }
```

Output:

```
3 4 5 6 7 8 9
```

Example program (do while loop) in C:

In do..while loop control statement, while loop is executed irrespective of the condition for first time. Then 2nd time onwards, loop is executed until condition becomes false.

```
#include <stdio.h>

 int main()

{
   int i=1;
     do
   {
    printf("Value of i is %d\n",i);
   i++;
   }while(i<=4 && i>=2);
  }
```

Output:

```
Value of i is 1
Value of i is 2
Value of i is 3
Value of i is 4
```

Difference between while & do while loops in C:

S.no	**while**	**do while**
1	Loop is executed only when condition is true. No semi colon in the condition	Loop is executed for first time irrespective of the condition. After executing while loop for first time, then condition is checked. Semi colon is allowed in the condition

C – Array

C Array is a collection of variables belongings to the same data type. You can store group of data of same data type in an array.

- Array might be belonging to any of the data types
- Array size must be a constant value.
- Always, Contiguous (adjacent) memory locations are used to store array elements in memory.
- It is a best practice to initialize an array to zero or null while declaring, if we don't assign any values to array.

Example for C Arrays:

- int a[10];   // integer array
- char b[10];   // character array   i.e. string

Types of C arrays:

There are 2 types of C arrays. They are,

1. One dimensional array
2. Multi-dimensional array
    1. Two dimensional array
    2. Three dimensional array, four dimensional array etc…

1. One dimensional array in C:

- Syntax : data-type arr_name[array_size];

Array declaration	**Array initialization**	**Accessing array**
Syntax: data _type arr_name [arr_size];	data_type arr_name [arr_size]=(value1, value2, value3,….);	arr_name[index];
int age [5];	int age[5]={0, 1, 2, 3, 4};	age[0]; /*0 is accessed*/age[1]; /*1 is accessed*/ age[2]; /*2 is accessed*/

char str[10];	char str[10]={'H','a','i'}; (or)char str[0] = 'H';char str[1] = 'a';
	
	char str[2] = 'i;
		str[0]; /*H is accessed*/str[1]; /*a is accessed*/str[2]; /* i is accessed*/

Example program for one dimensional array in C:

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

int i;

int arr[5] = {10,20,30,40,50};

// declaring and Initializing array in C

//To initialize all array elements to 0, use int arr[5]={0};
/* Above array can be initialized as below also

arr[0] = 10;

arr[1] = 20;

arr[2] = 30;

arr[3] = 40;

arr[4] = 50; */

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

{

// Accessing each variable

printf("value of arr[%d] is %d \n", i, arr[i]);
}

}
```

Output:

```
value of arr[0] is 10
value of arr[1] is 20
value of arr[2] is 30
value of arr[3] is 40
value of arr[4] is 50
```

2. Two dimensional array in C:

- Two dimensional array is nothing but array of array.
- syntax : data_type array_name[num_of_rows][num_of_column]

S.no	Array declaration	Array initialization	Accessing array
1	Syntax: data_type arr_name [num_of_rows][num_of_column];	data_type arr_name[2][2] = {{0,0},{0,1},{1,0},{1,1}};	arr_name[index];

2	Example:int arr[2][2];	int arr[2][2] = {1,2, 3, 4};	arr [0] [0] = 1; arr [0]]1] = 2;arr [1][0] = 3;
			
			arr [1] [1] = 4;

Example program for two dimensional array in C:

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

 // declaring and Initializing array

 int arr[2][2] = {10,20,30,40};

 /* Above array can be initialized as below also

 arr[0][0] = 10; // Initializing array

 arr[0][1] = 20;

 arr[1][0] = 30;

 arr[1][1] = 40; */

 for (i=0;i<2;i++)
  {
 for (j=0;j<2;j++)

 {
  // Accessing variables

 printf("value of arr[%d] [%d] : %d\n",i,j,arr[i][j]);

 }

 }

 }
```

Output:

```
value of arr[0] [0] is 10
value of arr[0] [1] is 20
value of arr[1] [0] is 30
value of arr[1] [1] is 40
```

C – Function

C functions are basic building blocks in a program. All C programs are written using functions to improve re-usability, understandability and to keep track on them. You can learn below concepts of C functions in this section in detail.

1. What is C function?
2. Uses of C functions
3. C function declaration, function call and definition with example program
4. How to call C functions in a program?
    1. Call by value
    2. Call by reference
5. C function arguments and return values
    1. C function with arguments and with return value
    2. C function with arguments and without return value
    3. C function without arguments and without return value
    4. C function without arguments and with return value
6. Types of C functions
    1. Library functions in C
    2. User defined functions in C
        1. Creating/Adding user defined function in C library
7. Command line arguments in C
8. Variable length arguments in C

1. What is C function?

A large C program is divided into basic building blocks called C function. C function contains set of instructions enclosed by "{ }" which performs specific operation in a C program. Actually, Collection of these functions creates a C program.

2. Uses of C functions:

- C functions are used to avoid rewriting same logic/code again and again in a program.
- There is no limit in calling C functions to make use of same functionality wherever required.
- We can call functions any number of times in a program and from any place in a program.
- A large C program can easily be tracked when it is divided into functions.
- The core concept of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality and to improve understandability of very large C programs.

3. C function declaration, function call and function definition:

There are 3 aspects in each C function. They are,

- **Function declaration or prototype** – This informs compiler about the function name, function parameters and return value's data type.
- **Function call** – This calls the actual function

- **Function definition** – This contains all the statements to be executed.

S.no	C function aspects	syntax
1	function definition	return_type function_name (arguments list) { Body of function; }
2	function call	function_name (arguments list);
3	function declaration	return_type function_name (argument list);

Simple example program for C function:

- As you know, functions should be declared and defined before calling in a C program.
- In the below program, function "square" is called from main function.
- The value of "m" is passed as argument to the function "square". This value is multiplied by itself in this function and multiplied value "p" is returned to main function from function "square".

```
#include<stdio.h>

// function prototype, also called function declaration

float square ( float x );

// main function, this is where the program starts from.

int main( )

{       float m, n ;

        printf ( "\nEnter some number for finding square \n");

        scanf ( "%f", &m ) ;

          // function call

        n = square ( m ) ;

        printf ( "\nSquare of the given number %f is %f",m,n );

 }

 float square ( float x )   // function definition

{         float p ;

        p = x * x ;
        return ( p ) ;
 }
```

Output:

```
Enter some number for finding square

2

Square of the given number 2.000000 is 4.000000
```

4. How to call C functions in a program?

There are two ways that a C function can be called from a program. They are,

1. Call by value
2. Call by reference

1. Call by value:

- In call by value method, the value of the variable is passed to the function as parameter.
- The value of the actual parameter cannot be modified by formal parameter.
- Different Memory is allocated for both actual and formal parameters. Because, value of actual parameter is copied to formal parameter.

Note:

- Actual parameter – This is the argument which is used in function call.
- Formal parameter – This is the argument which is used in function definition

Example program for C function (using call by value):

- In this program, the values of the variables “m” and “n” are passed to the function “swap”.
- These values are copied to formal parameters “a” and “b” in swap function and used.

```
#include<stdio.h>

// function prototype, also called function declaration

void swap(int a, int b);

 int main()

{    int m = 22, n = 44;
```

```
    // calling swap function by value
    printf(" values before swap  m = %d \nand n = %d", m, n);
    swap(m, n);
 }
 void swap(int a, int b)
 {
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
    printf(" \nvalues after swap m = %d\n and n = %d", a, b);
 }
```

Output:

```
values before swap m = 22
and n = 44
values after swap m = 44
and n = 22
```

2. Call by reference:

- In call by reference method, the address of the variable is passed to the function as parameter.
- The value of the **actual parameter** can be modified by **formal parameter**.
- Same memory is used for both actual and formal parameters since only address is used by both parameters.

Example program for C function (*using call by reference*):

- In this program, the address of the variables “m” and “n” are passed to the function “swap”.
- These values are not copied to formal parameters “a” and “b” in swap function.
- Because, they are just holding the address of those variables.
- This address is used to access and change the values of the variables.

```
#include<stdio.h>

// function prototype, also called function declaration

void swap(int *a,  int *b);

 int main()

{

     int m = 22, n = 44;

    //  calling swap function by reference

    printf("values before swap m = %d \n and n = %d",m,n);

    swap(&m, &n);

 }

 void swap(int *a, int *b)

{

    int tmp;

    tmp = *a;

    *a = *b;

    *b = tmp;

    printf("\n values after swap a = %d \nand b = %d", *a, *b);

}
```

Output:

```
values before swap m = 22

and n = 44
values after swap a = 44
and b = 22
```

Another set of notes

Reasons for Studying Concepts of Programming Languages

- *Increased ability to express ideas*.
    - It is believed that the depth at which we think is influenced by the expressive power of the language in which we communicate our thoughts. It is difficult for people to conceptualize structures they can't describe, verbally or in writing.
    - Language in which they develop S/W places limits on the kinds of control structures, data structures, and abstractions they can use.
    - Awareness of a wider variety of P/L features can reduce such limitations in S/W development.
    - Can language constructs be simulated in other languages that do not support those constructs directly?

- *Improved background for choosing appropriate languages*
    - Many programmers, when given a choice of languages for a new project, continue to use the language with which they are most familiar, even if it is poorly suited to new projects.
    - If these programmers were familiar with other languages available, they would be in a better position to make informed language choices.

- *Greater ability to learn new languages*
    - Programming languages are still in a state of continuous evolution, which means continuous learning is essential.
    - Programmers who understand the concept of OO programming will have easier time learning Java.
    - Once a thorough understanding of the fundamental concepts of languages is acquired, it becomes easier to see how concepts are incorporated into the design of the language being learned.

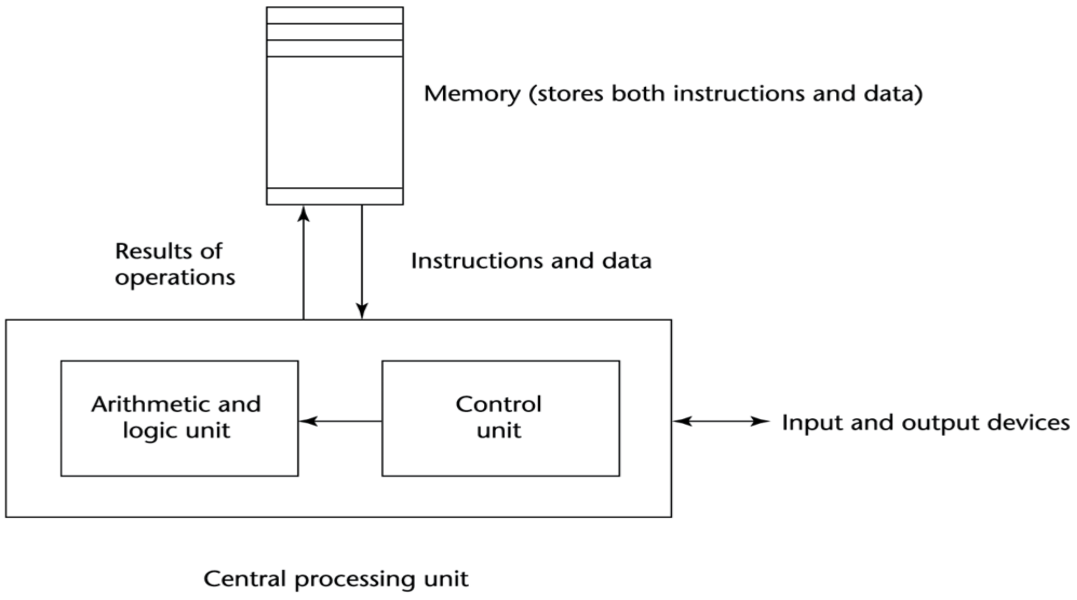
- *Understand significance of implementation*
    - Understanding of implementation issues leads to an understanding of why languages are designed the way they are.
    - This in turn leads to the ability to use a language more intelligently, as it was designed to be used.

- *Ability to design new languages*
    - The more languages you gain knowledge of, the better understanding of programming languages concepts you understand.

- *Overall advancement of computing*
    - In some cases, a language became widely used, at least in part, b/c those in positions to choose languages were not sufficiently familiar with P/L concepts.
    - Many believe that ALGOL 60 was a better language than Fortran; however, Fortran was most widely used. It is attributed to the fact that the programmers and managers didn't understand the conceptual design of ALGOL 60.
    - Do you think IBM has something to do with it?

Influences on Language Design

- **Computer architecture**: Von Neumann
- We use imperative languages, at least in part, because we use von Neumann machines
  - Data and programs stored in same memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - Results of operations in the CPU must be moved back to memory
  - Basis for imperative languages
    - Variables model memory cells
    - Assignment statements model piping
    - Iteration is efficient



ASSIGNMENTS:

a. **Write a C program to produce the following pattern as output:**

```
........
.......
......
.....
....
...
..
.
```

b. **Using a while loop, write a program to display even numbers between 1 and 10:**

c. **By using PI as a symbolic constant, write a program to compute the area of a circle whose radius the user enters through the keyboard.**

d. **What are the differences between actual parameter formal parameter?**