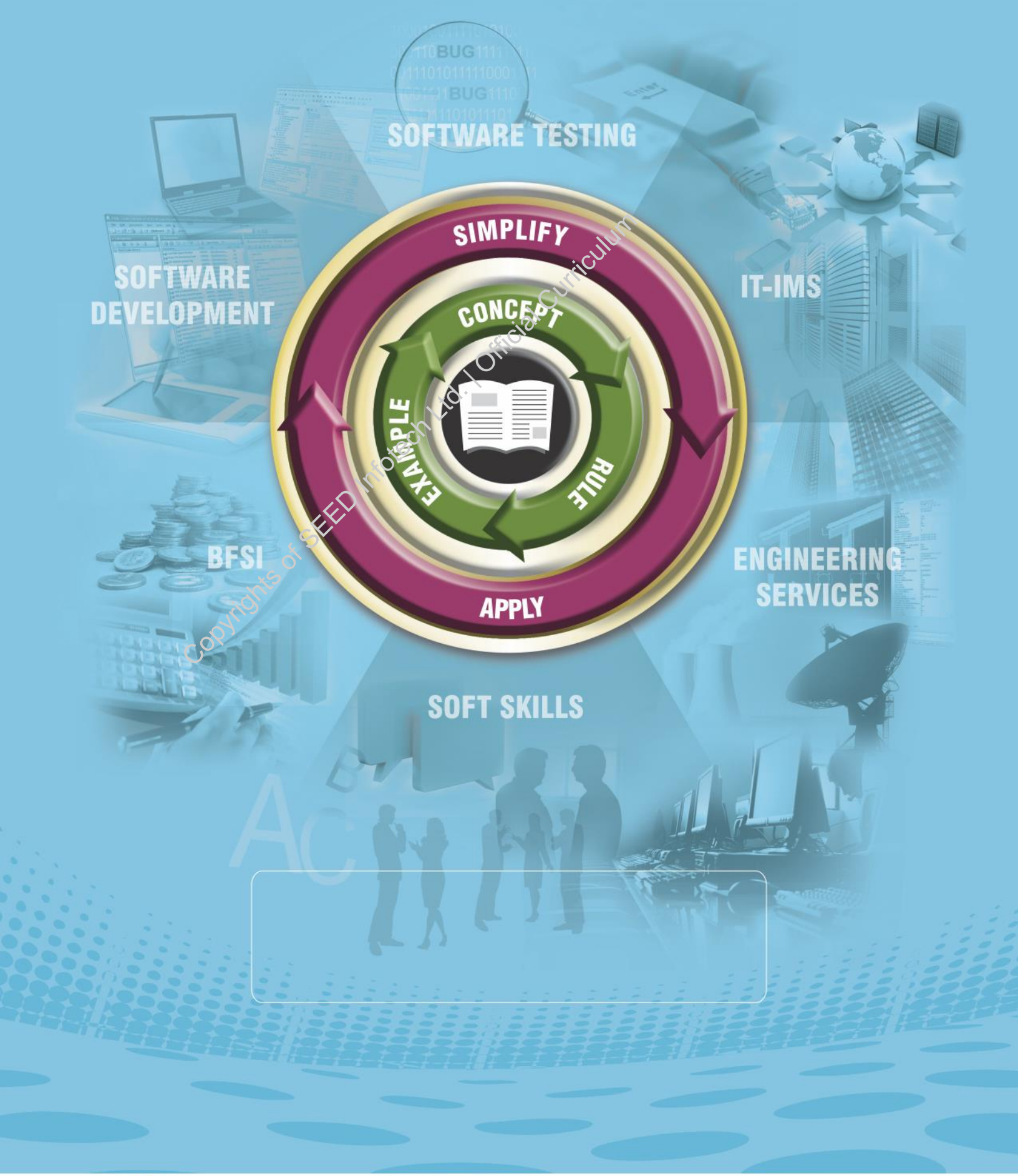


N·S·D·C
National Skill Development Corporation
Transforming the skill landscape
Training Partner

seed
Official Curriculum

Basic ‘C’ Programming

Student Book and Lab Manual

Module Code : [M-ILT-Obj-00001-Cpro-B-En]

Author : Ms Anagha Walvekar

This edition has been printed and published in house by Avani Publications.

Print Edition : August 2015

Issue No./ Date : 01 / Nov. 14, 2011 Revision No. & Date : 02 / Aug.10, 2013

Avani Publications

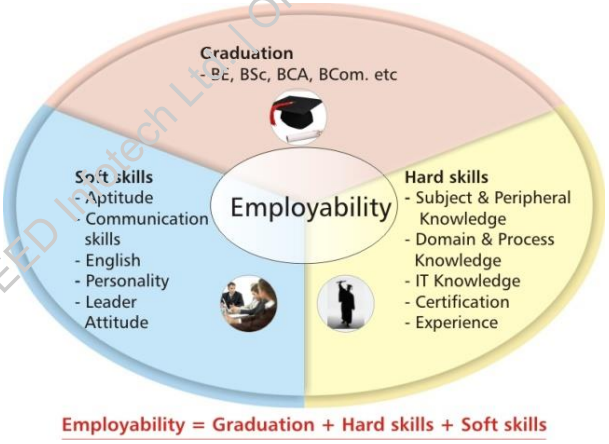
15A-1/2/4, Anandmayee Apartment, Off Karve Road, Pune 411004

Contents

Enhancing Employability

Employability depends on the knowledge, skills and abilities that individuals possess, the way they use these to solve problems and contribute to the growth of the employing organization and the society. Employability skills are those skills that are necessary for getting, keeping and doing well in a job.

Employability can be defined as an equation



Promise of this book is to help strengthen your "C programming skill" which can be classified under **Hard Skills** of employability equation.

Why learn 'C' programming?

'C' programming is a very popular programming language that beginners learn. Individuals seeking to make a career in software take the first step by learning the 'C' language. Following are the reasons:

a) Learning C programming builds problem solving ability through programs.
b) Higher level languages like C++, Java, C# have similar looking syntax as of C.
c) C is part of most of the college curriculums.
d) C programming is used for developing embedded systems, device drivers etc.
e) Most of the assessment tests for placement and entrance exams are based on 'C' language.

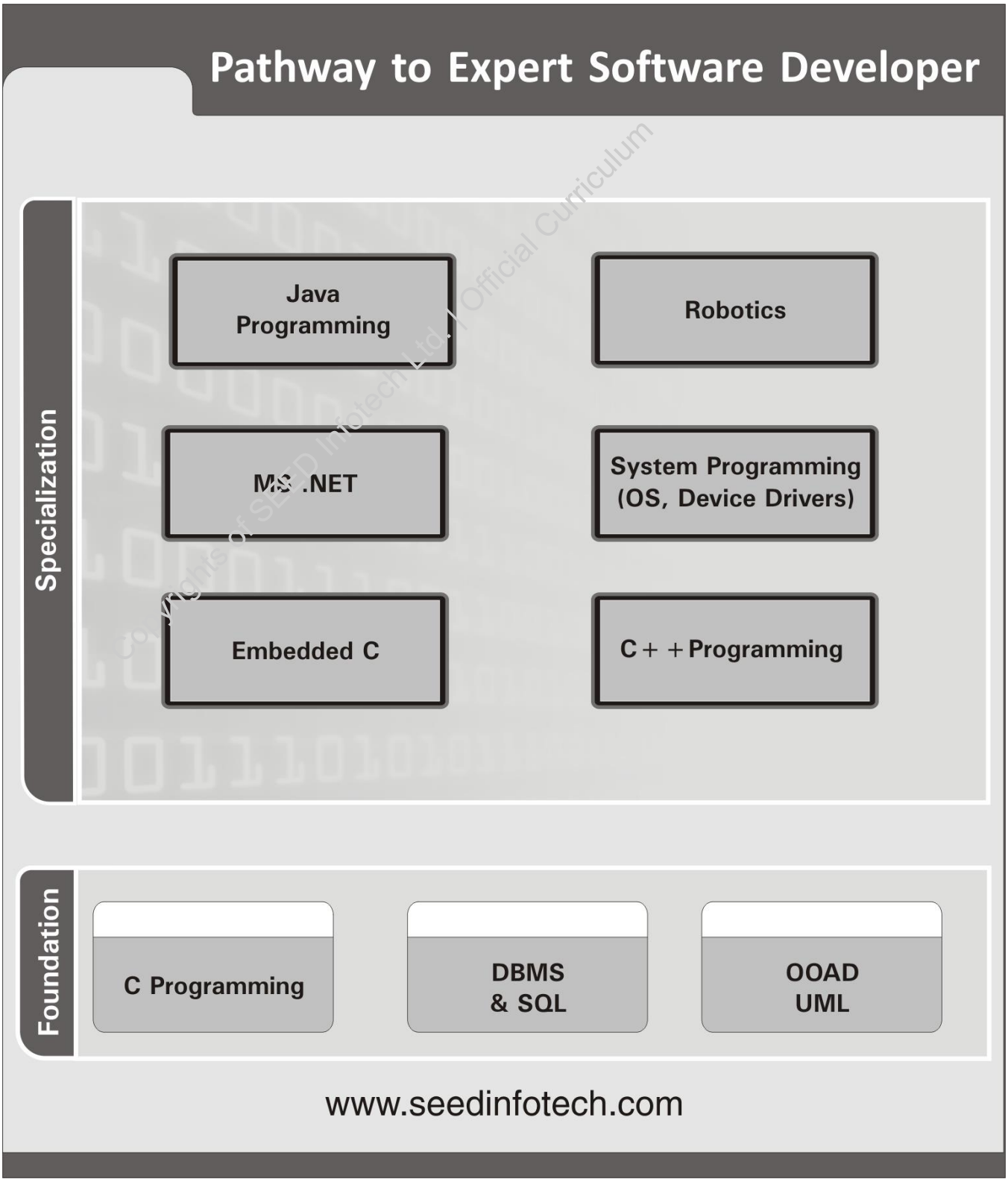
Role of this book

This book is a step by step guide to master C programming language skills and would aid and reinforce the learning in the classroom. To master any programming

language one needs hands-on practice along with clarity of concepts. This student book combined with lab manual is designed to serve the purpose.

The smart tips embedded in the form of Tech App, Interview Tip, Additional Reading, Best Practices, etc. would help increase your curiosity and also help you to become expert with knowledge of peripheral concepts.

We have strived hard to make technical contents of this book error free. However, some mistakes might have slipped our attention. We request the reader to send us any such errors that are sighted which will help us in improving this book further. Your issues or suggestions are welcome on email at ***product-issues@seedinfotech.com***

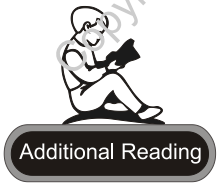


“Beyond Obvious” Icons

In the student book, we have included special icons (Beyond Obvious Icons) in the form of footnotes that are interspersed in the study material to give you the precise context of the concept you are learning. As you get accustomed to this way of learning, you will enjoy the fun of it which will make this learning highly productive!

This icon indicates a particular best practice which is followed while developing the applications, in design, in coding etc. Knowledge of best practices makes one a good developer or designer.

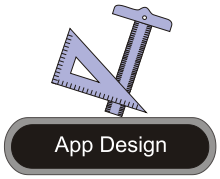
This icon indicates the points which are important from your technical interview. Before interview, you may visit these small tips as quick revision pointers.



This icon suggests that it is good for the student to go through this additional reading material to bring more clarity to the concepts.

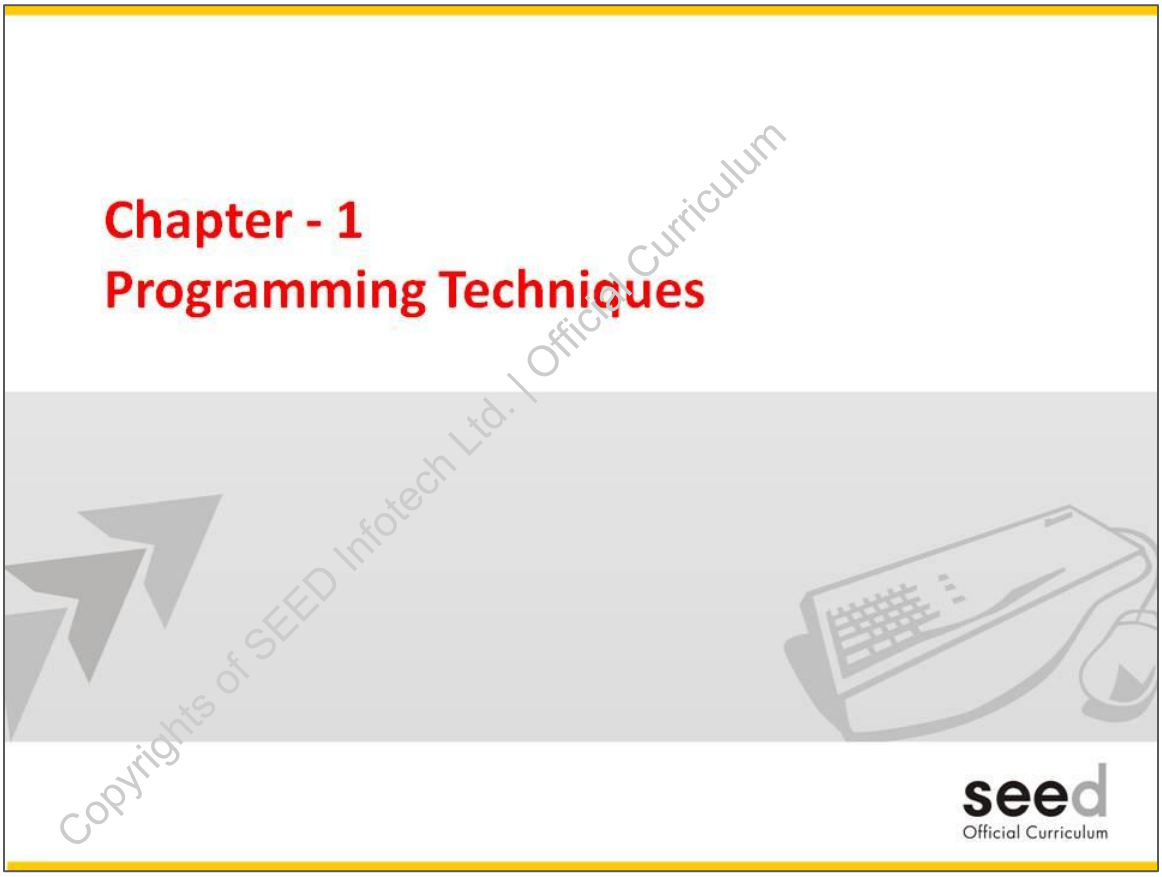
This icon indicates that this is a group discussion or group exercise. This is added for collaborative learning and problem solving. In the job environment one needs to take part in such discussions to arrive at the solution.

This icon indicates that more details are provided with respect to application of the technology/tool/concept which you are learning. This is application of technology learned to solve the real world problem.



This icon indicates an important note or tip from the application design perceptive.

This icon indicates quiz to be solved in the classroom. This is for reinforcement of what you have learnt by challenging you through the questions.

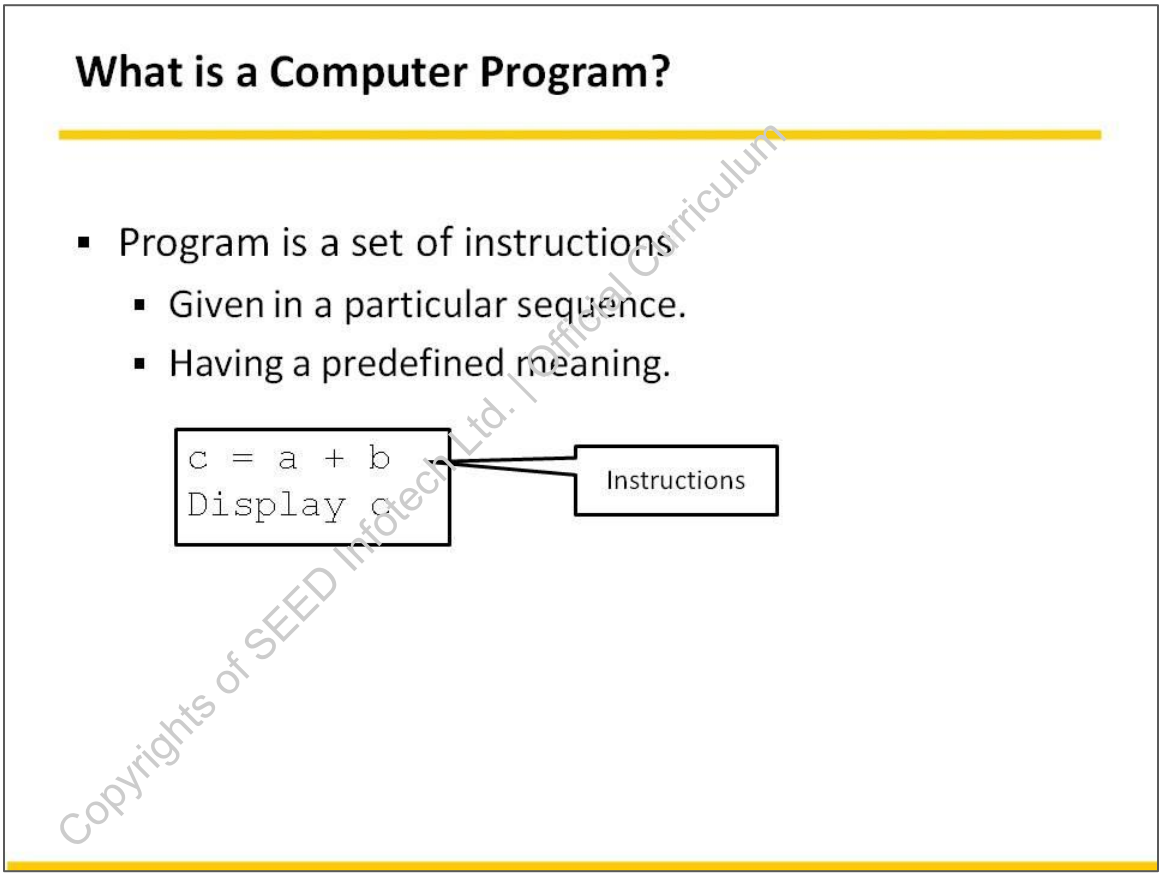


This chapter covers various topics such as a computer program, hardware, software, number systems, algorithms, flowcharts, interpreter and compiler and program development environment.

Objectives

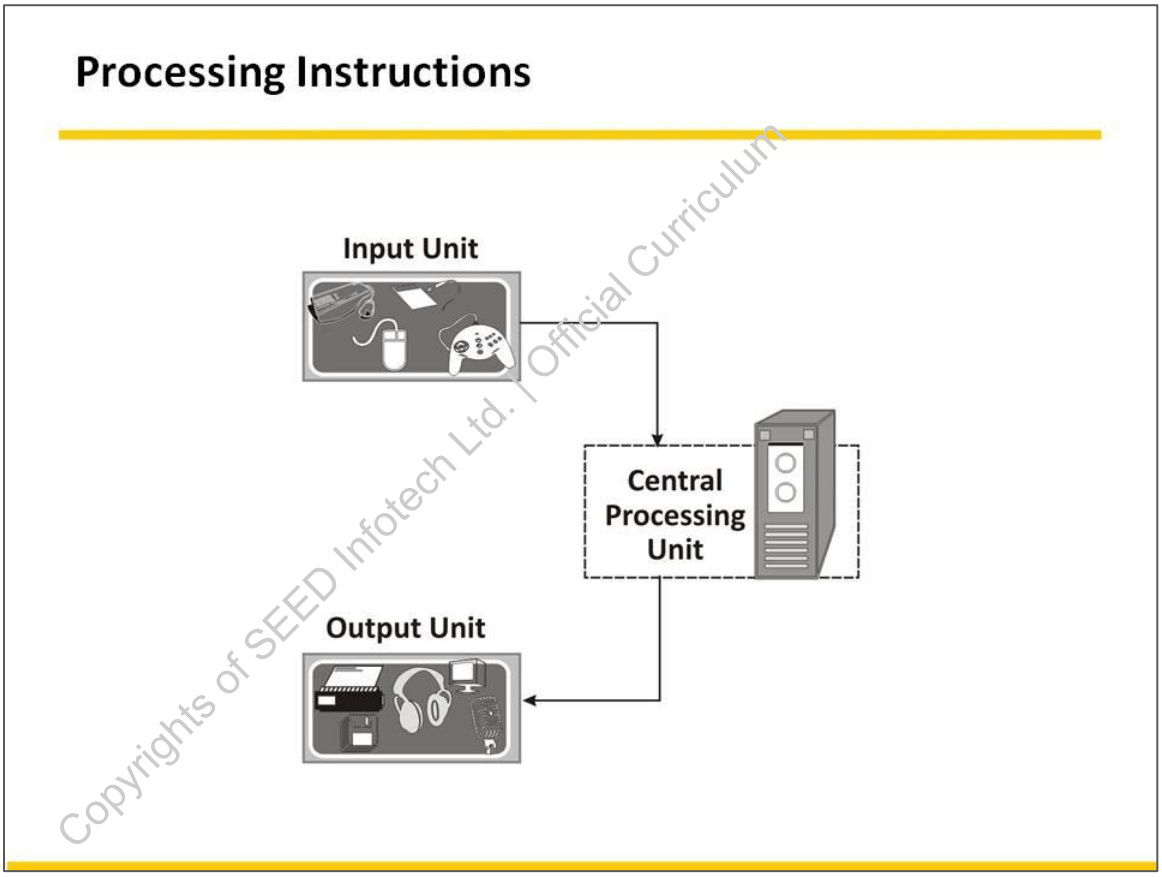
At the end of this chapter you will be able to:

- Define a computer program.
- List the components of a computer required to execute a program.
- List number systems.
- Distinguish between system software and application software.
- State different programming languages.
- Define an algorithm.
- Write algorithms for solution of given problems.
- Draw flowcharts for solution of given problems.
- Distinguish between an interpreter and a compiler.
- Describe steps involved in developing a C program.



A computer program is a set of instructions given to a computer to achieve a desired output. An instruction is a combination of some words which have a predefined meaning. Instructions are given in a particular sequence.

Suppose, c = a + b is to be performed. It requires some input and gives a desired output. In between there will be some calculations. So the some unit will be required that performs calculations. The values of a, b, and c will need to be stored; the storage unit will be required to do this. To co-ordinate between these units, a control unit is required. Thus, to execute a program, hardware is also required.



These instructions are written in the form of code. To write these instructions, a programming language is used. It is a language that is understood by the computer.

A computer has two inseparable parts -hardware and software. The instructions are the software and the physical components of a computer that are used in the process are the hardware.

The instructions in this case will be - to read two numbers, perform addition, and then display the result. Thus, some input will be required which will be processed and the end result will be the desired output.

Input and output are the inherent parts of interaction with a computer system. To say it in other words, the entire system can be shown to work as Input → Process → Output.

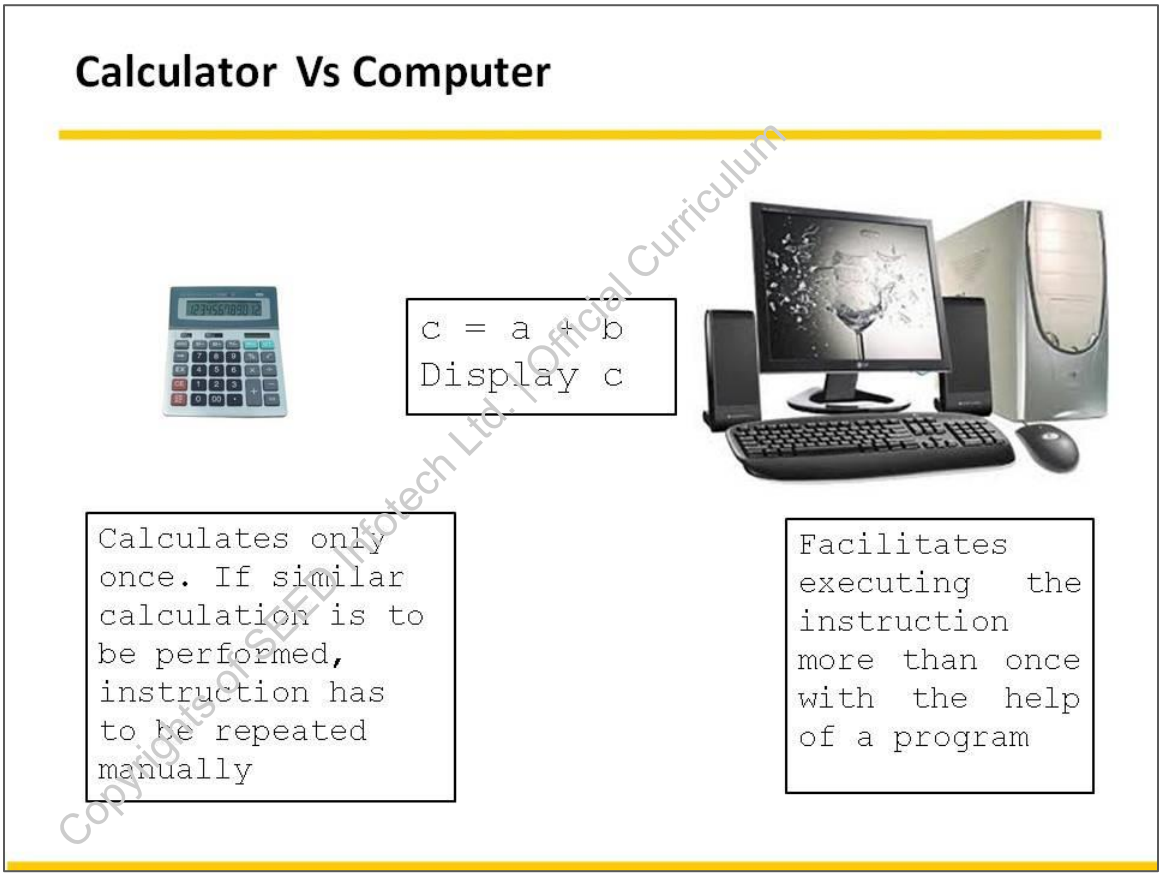
The components of a computer system, involved in this process are -

- Input devices (keyboard, mouse, scanner, joystick, camera, etc)

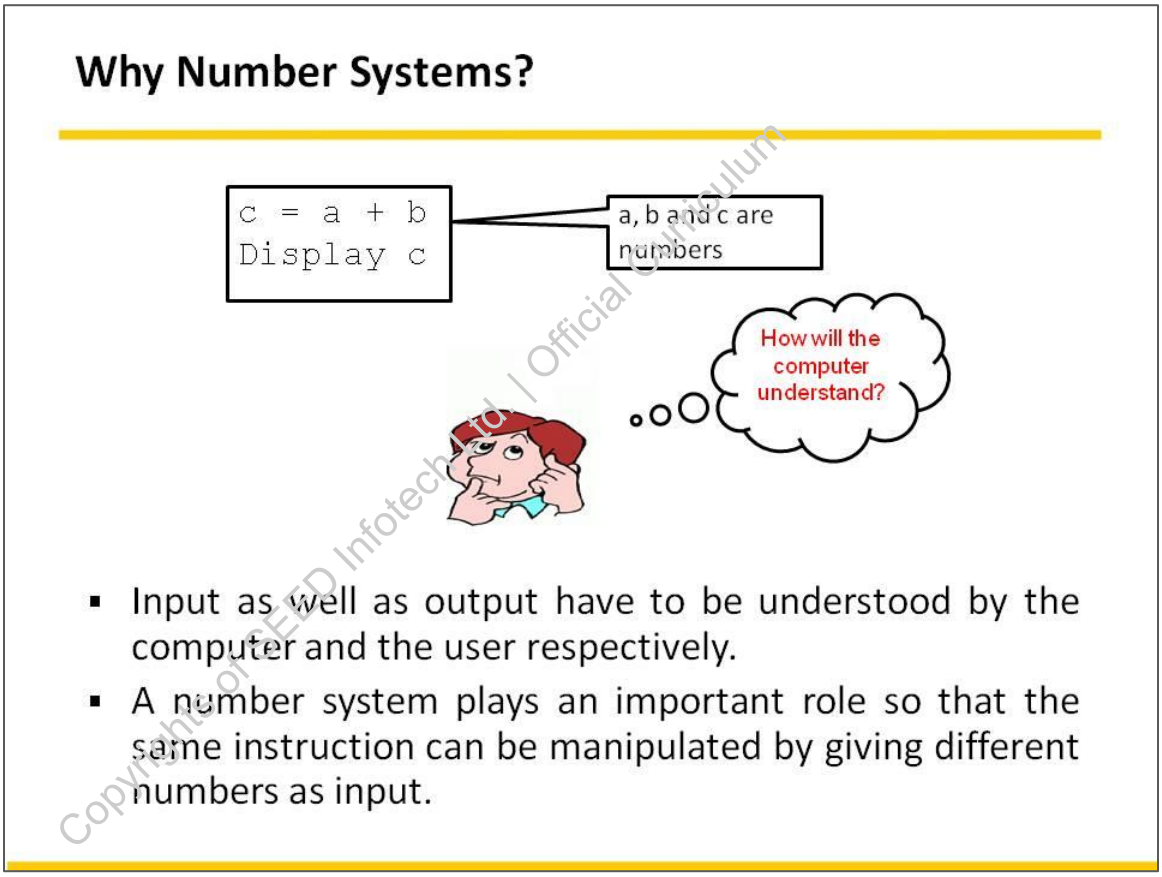
- Central Processing Unit (Control unit, ALU, Primary storage unit)
- Output devices (monitor, printer, speakers, headphones, etc.)

The data which is input by the user is temporarily stored in the Random Access Memory (primary storage unit) before processing it. The Central Processing Unit (CPU) processes the data. It is then either displayed to the user through the output device or stored on the secondary storage device.

RAM is a temporary storage device. So, to store data permanently, it is stored on the hard disk which is also called the secondary storage device.



The instruction `c = a + b` looks like a simple calculation that can be performed using a calculator. If addition is to be performed on ten more pairs of numbers, it will be a repetitive task. It will be boring for a human being; instead, if instructions to do the calculation are given to a machine, it will diligently perform the task as long as required.



Now, in the instruction `c = a + b`, the numbers are a part of the instruction. These numbers have to be represented in a form that can be understood by the computer and the output has to be understood by the user. A number system plays an important role; the same instruction can be manipulated by giving different numbers as input.

Number System

- Way of counting things.
- Computers use different types of number systems.
  - Decimal
  - Binary
  - Octal
  - Hexadecimal

Number System	Range of Values for Digits	Value Examples
Decimal	0-9	45
Binary	0-1	101101
Octal	0-7	55
Hexadecimal	0-F	2D

In the instruction `c = a + b`, the numbers are a part of the instruction. These numbers have to be represented in a form that can be understood by the computer and the output has to be understood by the user. A number system plays an important role so that the same instruction can be manipulated by giving different numbers as input.

A number system is a way of counting things. Humans use the base ten (10) system of decimal which is a fairly simple, yet complex system of counting. If counting is to be performed using a computer, then the decimal number system cannot be used. Number system with base two (2) is used to perform calculations using a computer.

Every number system has a base and the number of digits in the system is equal to the base. For example, decimal number system has base 10 and consists of 10 digits (0 to 9).

Number systems used by computers are –

Binary - (Base 2)

There are only two digits in binary, 0 and 1 (also called a bit). Because computers use a sequence of switches that can be on or off, base 2 works very well for them. Math in base 2 is very simple, but time consuming.

Octal - (Base 8)

It uses digits 0 to 7. There are eight bits in a byte which is used very often in the computer field. Math in octal is more complicated than decimal.

Hexadecimal - (Base 16)

It uses digits 0 to F. Since there are 16 values per place holder, six new digits were required needed to be created. These are A, B, C, D, E, and F. 'A' has a value of 10, 'B' is 11, and so on. This system is used in the computer field as a means of viewing lot of data much faster. Math in hexadecimal is not very simple compared to decimal.

Binary Number System

- A computer is an electronic device consisting of flip-flops or switches.
  - Can be represented as 1 (ON) and 0 (OFF)
- It consists of only 2 digits hence called as binary number system.
- Most modern computer systems use it.
- The weighted values for each position can be determined as follows:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}
128	64	32	16	8	4	2	1	0.5	0.25

There are different number systems that a computer uses. Since it is an digital machine it has switches (flip-flops) that have only two states - ON and OFF. Therefore, a number system consisting of only 2 digits - 1(ON) and 0 (OFF) was devised. This number system is called the binary number system.

The other number systems that are used by computers are octal and hexadecimal. These are used so that information occupies lesser space.

Most modern computers use the binary number system. The binary system works well for computers because the mechanical and electronic circuits recognize only two states of operation, such as on/off or open/close or true/false and can thus work with 2 digits – 1 and 0.

A **bi**nary digi**t** is called a **bit**. A bit is the smallest unit of information a computer can use. A group of 8 bits is called a **byte** and 1024 bytes is 1 **Kilobyte**. Likewise, 1024 kilobytes is 1 **Megabyte**.

Like in any number system, every position of a digit in a binary number has a weighted value. Using these weighted values, numbers can be converted from

binary to decimal. The least significant bit is the lowest bit in a series of numbers in binary; it is located at the far right of a binary number. For example, in the binary number: `10111000`, the least significant bit is the far right "`0`" and the most significant bit is the "`1`" in the beginning. It represents the sign of the number.

Converting a binary number to decimal can be done as follows:

Binary number	**Conversion to Decimal**
$(1001)_2$	$= 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ $= 8 + 1$ $= (9)_{10}$

$(10101011)_2$	$= 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$ $= 128 + 32 + 8 + 2 + 1$ $= (161)_{10}$

To convert a base-10 integer numeral to its base-2 (binary) equivalent, the number is divided by two, and the remainder is the least significant bit. The (integer) result is again divided by two; its remainder is the next most significant bit. This process repeats until the result of further division becomes zero.

For example, $(98)_{10}$, in binary, is:

Operation	**Remainder**
98 ÷ 2 = 49	0
49 ÷ 2 = 24	1
24 ÷ 2 = 12	0
12 ÷ 2 = 6	0

Read the remainders upwards

6 ÷ 2 = 3	0
3 ÷ 2 = 1	1
1 ÷ 2 = 0	1

Reading the sequence of remainders from the bottom up gives the binary numeral $(1100010)_2$.

Negative numbers can be represented using either 1's complement or 2's complement representations. Methods for their representation are:

- 1's complement – reverse all the bits

```
1's complement of 2 ( 0000 0010 ) is -2 ( 1111 1101 )
```

- 2's complement – reverse all the bits + 1

```
2's complement of 2
(2)2                :    0000 0010
Reverse all bits    :    1111 1101
Add 1 to it         :         1
2's complement      :    1111 1110
```


Like decimal numbers, arithmetic operations like addition, subtraction, multiplication and division can be performed on binary numbers also.

Binary Addition

Binary Addition works in the same way as decimal addition except that only 0s and 1s are used. For performing binary addition, following operations are considered.

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10
```

Example 1

```
 101   (5)2
+100   (4)2
```

1. To add these two numbers, first, consider the "ones" column and calculate 1 + 0, which (in binary) results in 1. There is no carry over, so leave the 0 in the "ones" column.
2. Moving on to the "tens" column, calculate (0 + 0), which gives 0. Nothing "carries" to the "hundreds" column, and leave 0 in the "tens" column.
3. Moving on to the "hundreds" column, calculate 1 + 1, which gives 10. Carry 1 to the "thousands" column, leaving the 0 in the "hundreds" column.

```
  1 0 1      (5)10
+ 1 0 0    +(4)10
-------    -----
1 0 0 1      (9)10
```


Example 2

```
  1 0 1 1    (11)10
+ 1 0 1 1  +(11)10
-----------  -----
1 0 1 1 0      (22)10
```

Note that in the "tens" column, 1 + (1 + 1), where the first 1 is "carried" from the "ones" column. In binary, $1 + 1 + 1 = 10 + 1 = 11$

Binary Subtraction

Binary subtraction can be performed by adding the 2's complement of the subtrahend to the minuend. If a final carry is generated, discard the carry and the answer is given by the remaining bits which is positive. If the final carry is zero, the answer is negative and is in 2's complement form.

Example 1

Subtract 5 from 7 using 2's complement method. It would be 7 + (-5). It is known that, number 5 is represented as 0101 in binary form. Its 2's complement is 1011.

```
          0 1 1 1
        + 1 0 1 1   (-5 in 2's complement format)
      -------------
        1 0 0 1 0
      -------------
```

Here the summation is obtained as 5 bits. So according to the rule discard the carry, and take the remaining value 0010 as result i.e $(2)_{10}$. Since the last carry was generated, the result is positive.

Example 2

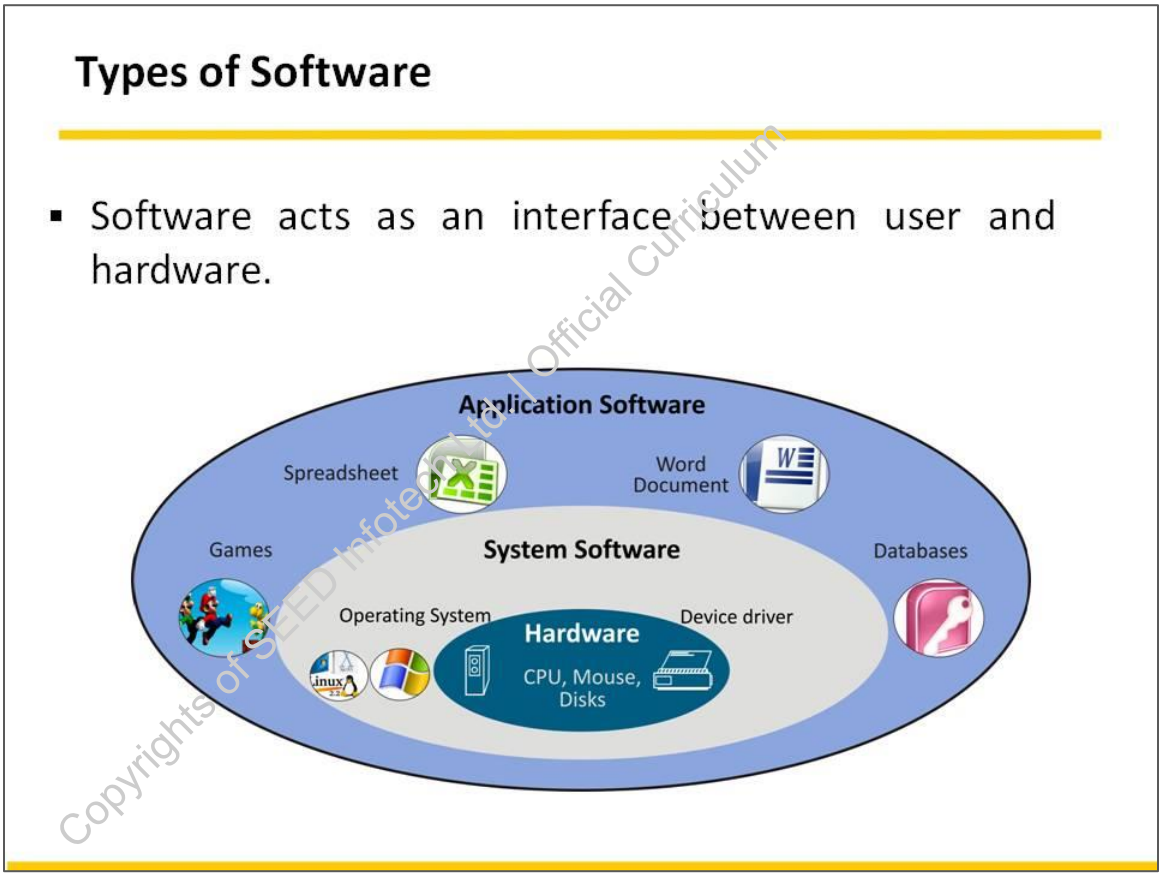
Subtract 7 from 5. It would be 5 + (-7). The number 7 is represented as 111 in binary form. Its 2's complement is 1001.

```
          0 1 0 1
    +     1 0 0 1  (-7 in 2's complement format)
    ---------------
          1 1 1 0
    ---------------
```

No carry is generated and the summation is still 4 bits, so the answer is negative and its value is in 2's complement form.

2's complement of 1110 is 0010, hence the answer is $(-2)_{10}$. Again note, that since the last carry was not generated, the result is negative.

2's complement is used to represent negative numbers. Such a representation allows use of only Adder as a circuit to perform both -addition and subtraction.



Operating system is an interface between the hardware, the user and the application systems. It is required since the hardware is continuously changing but there is no need to modify or change the applications running on it to be compatible with the operating system.

Anything that can be stored electronically is software, whether it is computer instructions or data. Software is a general term for the various kinds of programs used to operate computers and related devices.

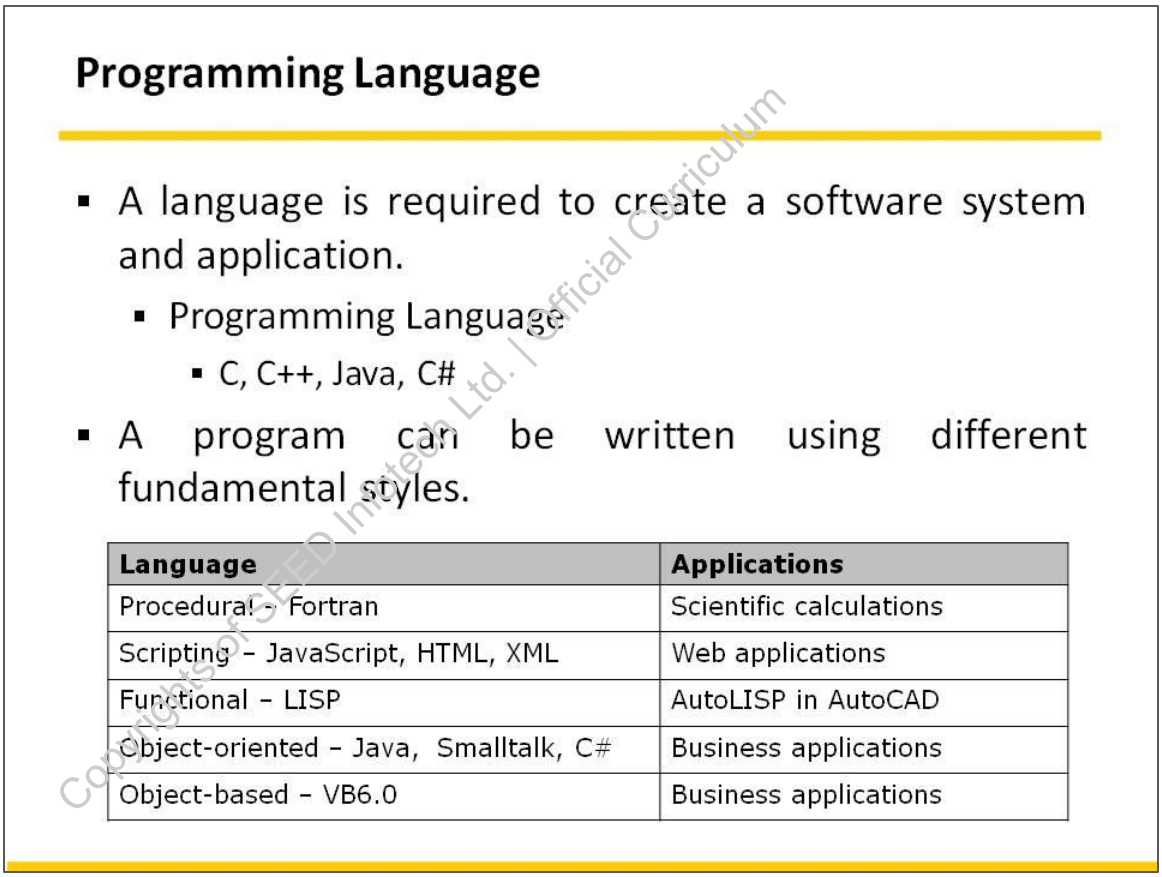
Software is often divided into two types:

1. System software consists of low-level programs that perform low level interaction with the system. It includes the operating systems, device drivers, and all the utilities that enable a computer to function.
    - **Operating system** is the software component of a computer system that is responsible for the management and coordination of activities and the sharing of the resources of the computer. E.g., Microsoft Windows, Linux, and Solaris, are some popular operating systems.

- **Device driver** is a small program specific to a device such as a printer, a soundcard or other computer peripheral that allows the device to be used by an operating system.
- **Utility** is a program that performs a very specific task, usually related to managing system resources. Operating systems contain a number of utilities for managing disk drives, printers, and other devices.

2. Application software includes programs that do real work for users. For example, word processors, spreadsheets, and database management systems.

   Typical examples of software applications are word processors, spread sheets and media players. Multiple applications bundled together as a package are sometimes referred to as an **application suite**. Microsoft Office, Open Office are typical examples.



Every **programming language** has a fundamental style of programming regarding how solutions to problems are to be formulated in a programming language. There are different types of programming paradigms -

Procedural

Procedural programming is based upon the concept of procedure calls. Procedures, also known as routines, subroutines, methods or functions, contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution by any other procedure or even by the procedure itself. 'C' language is procedural language.

Procedural programming is often a better choice than simple sequential or unstructured programming in many situations which involve moderate complexity or which require significant ease of maintainability.

Scripting

Programs which follow this paradigm are written in computer programming languages that are typically interpreted. They are called scripting language programs or scripts. Scripts are often distinguished from programs, because programs are converted permanently into binary executable files (i.e., zeros and ones) before they are run. Scripts remain in their original form and are interpreted command-by-command each time they are run. Scripts were created to shorten the traditional edit-compile-link-run process. VBScript, JavaScript etc. are scripting languages.

Scripts are widely used for creating graphical user interfaces or executing a series of commands that might otherwise have to be entered interactively through the keyboard.

Functional

Functional programming is a programming paradigm that emphasizes the application of functions, in contrast to the style that emphasizes changes in state. A state is a unique configuration of information in a program or machine.

Functional programming is in many respects a simpler and cleaner programming paradigm because it originates from a purely mathematical discipline: the theory of functions (Evaluate an expression and use the resulting value for something).

Object-based

In general, **object-based** indicates that something is based on the concept of object, it can be a theory, language, model, anything.

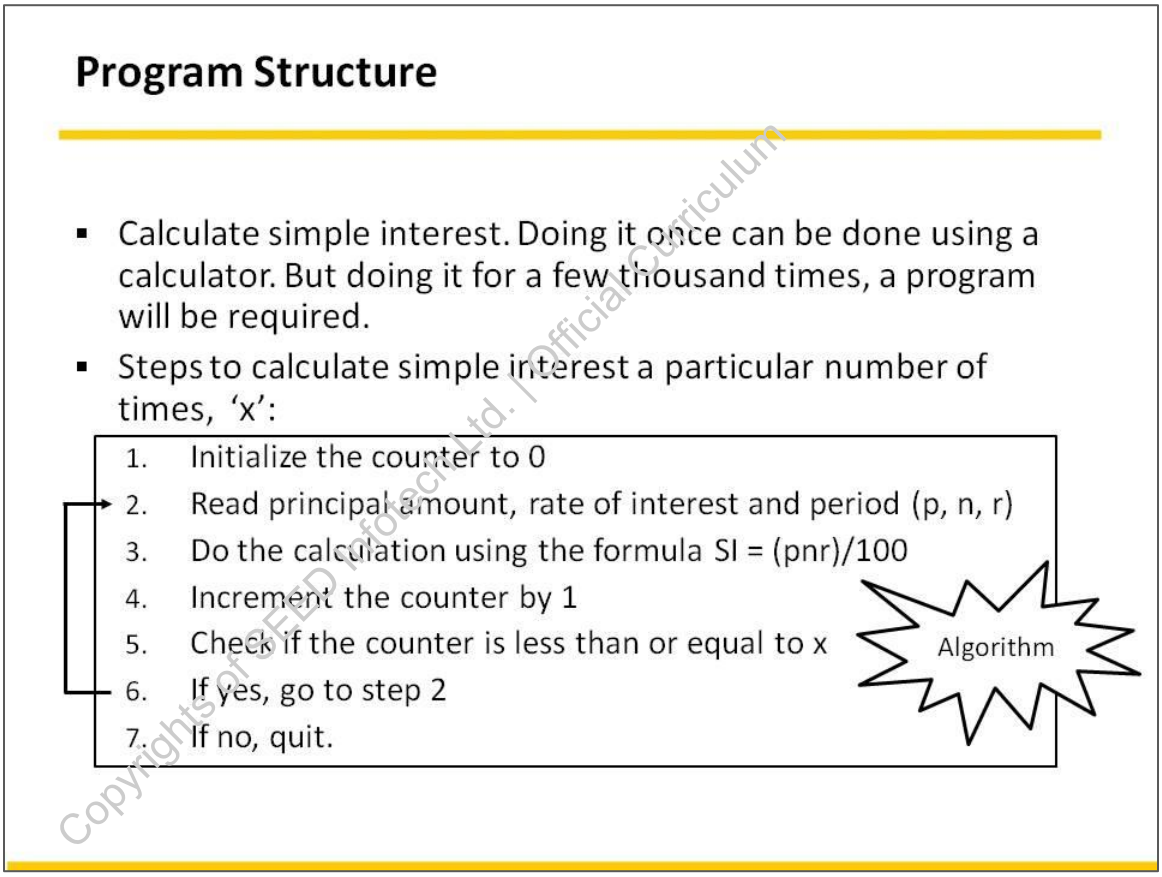
In technical sense, the term object based language may be used to describe any programming language that is based on the idea of encapsulating state and operations inside objects. Object-based languages do not support inheritance and are not considered to be true object-oriented languages. VB 6.0 is an example of object-based language.

Object-oriented

Object-oriented programming (OOP) is a programming paradigm that uses "objects" to design applications and computer programs. It can use several techniques from previously established paradigms, including inheritance, modularity, polymorphism, and encapsulation.

Object-oriented programming may be seen as a collection of co-operating objects, as opposed to a traditional view in which a program may be seen as a list of instructions to the computer. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent little machine with a distinct role or responsibility. C++, Java, C#, VB.NET are object-oriented languages.

Categorization of programming languages and their features is important.



A program is an organized list of instructions that, when executed, causes the computer to behave in a predetermined manner. A program is composed of data and executable statements.

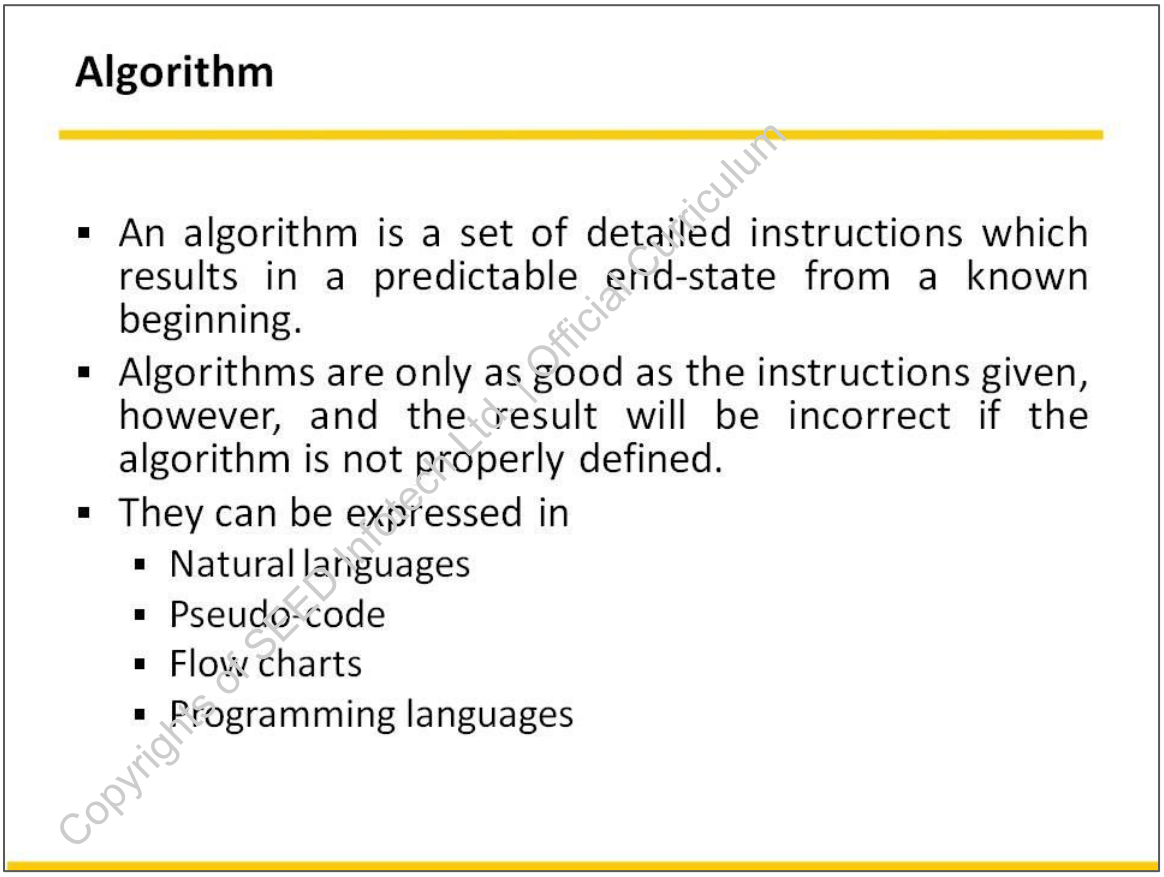
Data

In computing, data is information that has been translated into a form that is more convenient to move or process. It is the information that is converted into binary digital form. It may consist of numbers, words, or images, particularly as measurements or observations of a set of variables.

Executable statement

It is a statement that causes an action to be taken by the program; for example, to perform a calculation, test conditions, or alter normal sequential execution. Different types of executable statements are –

- Sequence - when one task is followed by another.
- Conditional - when instructions are carried out depending upon the fulfillment of a condition.
- Iteration - when an action or a group of actions have to be repeatedly executed.
- Flow control - when the flow of a program is diverted to another part of the program. For example, when a function call is given by a program, the function is executed before returning to the calling program or function.



A program can be written when the task to be performed is understood well. The steps to be performed to do the task need to be written down in plain simple language. This is called an algorithm. A program consists of various instructions like reading data, performing actions or processing the data, displaying data and so on.

An algorithm can be defined as a set of explicit and unambiguous (only one meaning) finite steps, which when carried out for a given set of initial conditions, produce the corresponding output, and terminate in a fixed amount of time.

An algorithm states how to solve a problem systematically to get a desired output. For example, algorithm to read user name from keyboard and say hello to the user can be defined as follows:

1. START
2. Read user name from keyboard
3. Store INPUT to NAME
4. Display hello to user
5. STOP

The following are the features of an algorithm -

- Finiteness: In the above example the algorithm terminates after a fixed number of steps i.e. at step 5.
- Definiteness: Action should be clearly specified. In the above example the username is read and greetings to the user are the output.
- Effectiveness: All the operations used in the algorithm are basic. For example, reading username from keyboard. It can be performed in a fixed duration.
- Input: In the above example input is the user name.
- Output: In above example output is the greetings to the user.

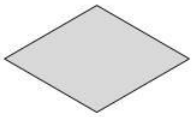
An algorithm can be expressed in

- Natural language - Languages spoken by humans can be used to write an algorithm.
- Pseudo-code - It is a compact and informal high-level description of a computer programming algorithm that uses the structural conventions of programming languages but omits detailed subroutines, variable declarations or language-specific syntax. It does not obey the syntax rules of any particular programming language.
- Flow chart - A flowchart is a diagrammatic representation of an algorithm or a stepwise process, showing the steps as boxes of various kinds, and their order by connecting these with arrows. Flowcharts are used in designing or documenting a process or program.
- Programming language - The term programming language usually refers to high-level languages, such as BASIC, C, C++, and so on. Each language has a unique set of keywords (words that it understands) and a special syntax for organizing program instructions.

Always make sure that understanding of the task at hand is represented as steps of algorithm. Once the algorithm is well designed, program can be created using any language.

It is a good practice to document logic as part of comments in the program. This improves understanding and makes further maintenance easy.

Flow Chart Symbols

Name	Symbol	Use
Oval		Indicates the beginning or end of a program
Flow Line		Indicates the direction of flow of logic in a program
Parallelogram		Indicates an input or output operation
Rectangle		Indicates a process
Diamond		Indicates a decision or branch in the logic of program.

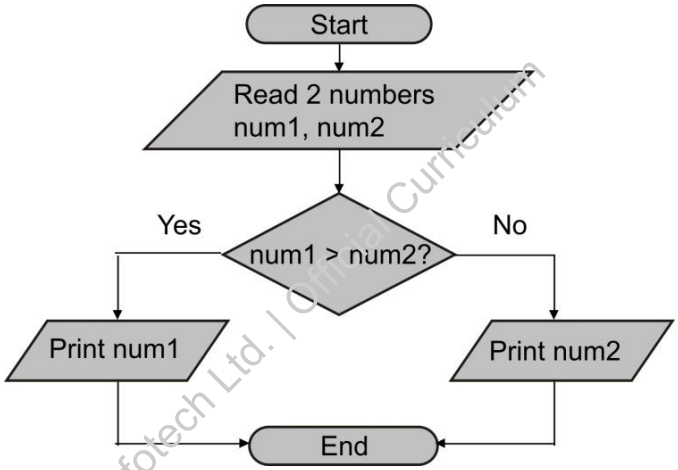
A flowchart is a graphical representation of an algorithm or a process. It shows the operations performed in the system and the sequence in which the operations are performed. It is often used in the design phase of programming to work out the logical flow of a program. Flowchart symbols are used to represent the operations and sequence of operations.

Problem 1: Find maximum of two numbers entered.

Algorithm

1. START
2. Accept two numbers as num1 and num2.
3. Check if num1 > num2. If this condition is true, print num1
4. Else print num2
5. STOP

Flowchart

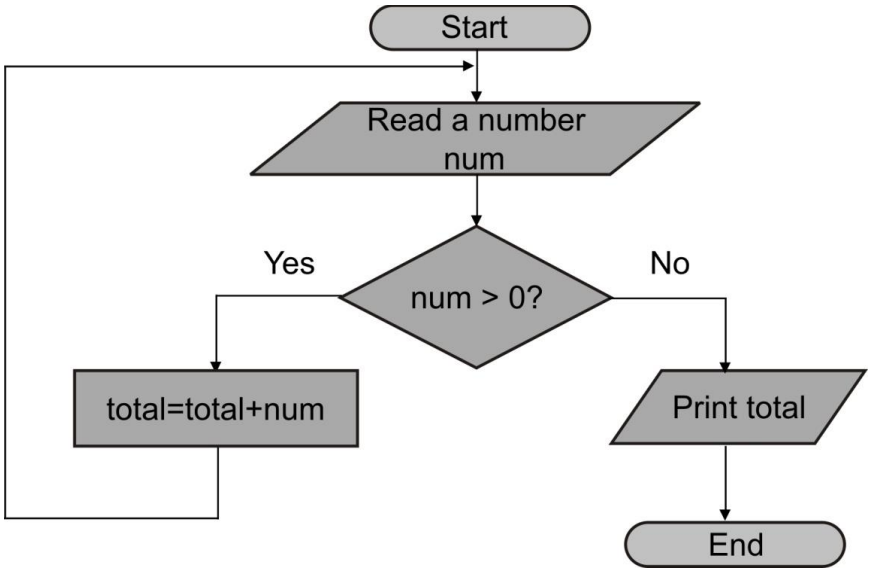


Problem 2: Calculate sum of numbers until entered number is not positive.

Algorithm

1. START
2. Accept a number, num.
3. Check if the num > 0.
4. If this is true, add to the total
5. Go to step number 2.
6. Else print the total.
7. STOP

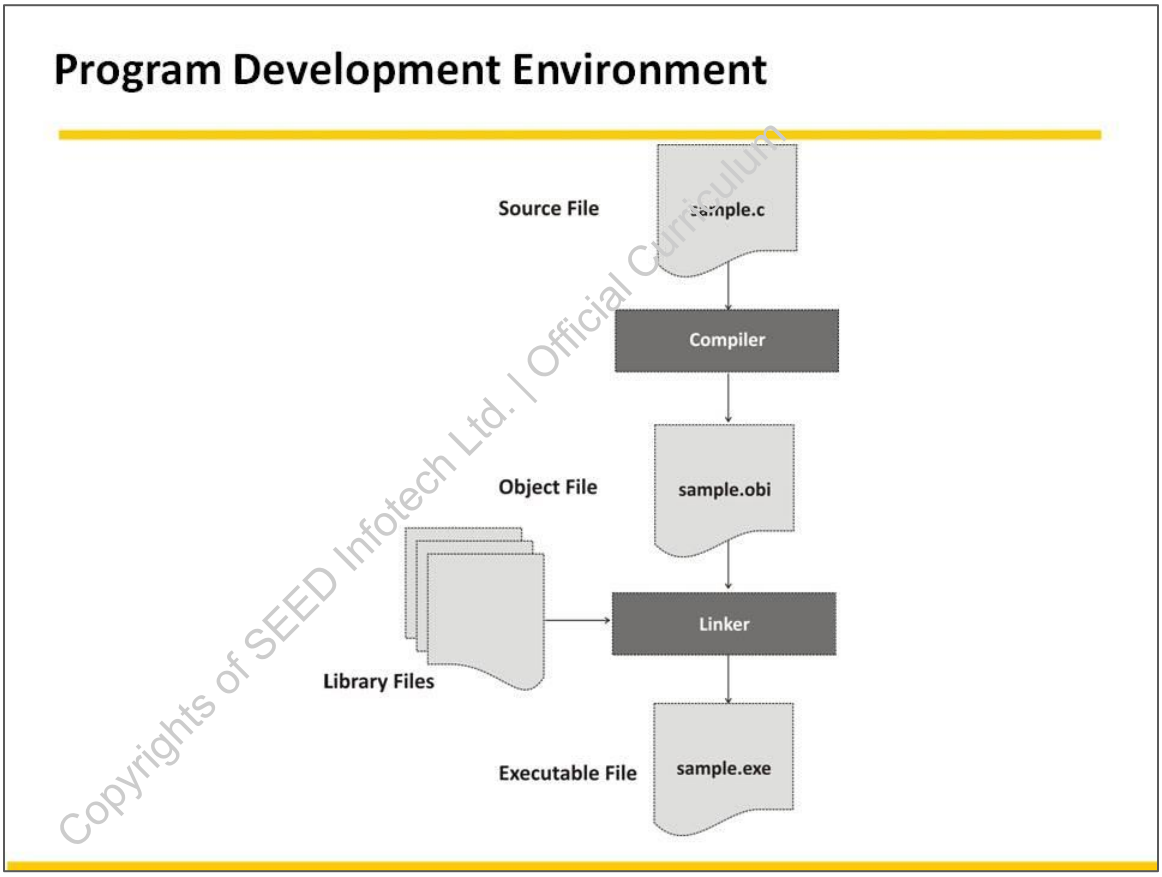
Flowchart



Create a flowchart to

1. Find the maximum of three numbers.
2. Find whether a triangle is a right angled triangle.

In application design using UML, activity diagram is similar to drawing a flowchart. It is a part of high-level design.



Whenever a program is written by a programmer in a programming language, it needs to be translated into a language that can be understood by a computer that is a language made up of zeroes and ones known as the binary language. This can be done in two ways - either by using an interpreter or a compiler.

Interpreter

It translates a program line by line and stops at the line where it finds an error. The programmer has to correct the error and again interpret it. For example, Visual Basic and Java both use an interpreter. It is also a program that executes instructions written in a programming language.

Compiler

It is a program that translates an entire high-level language program into a detailed set of machine language instructions that a computer requires. Errors, if any, are listed after compiling the program.

Some operations performed by a compiler are:

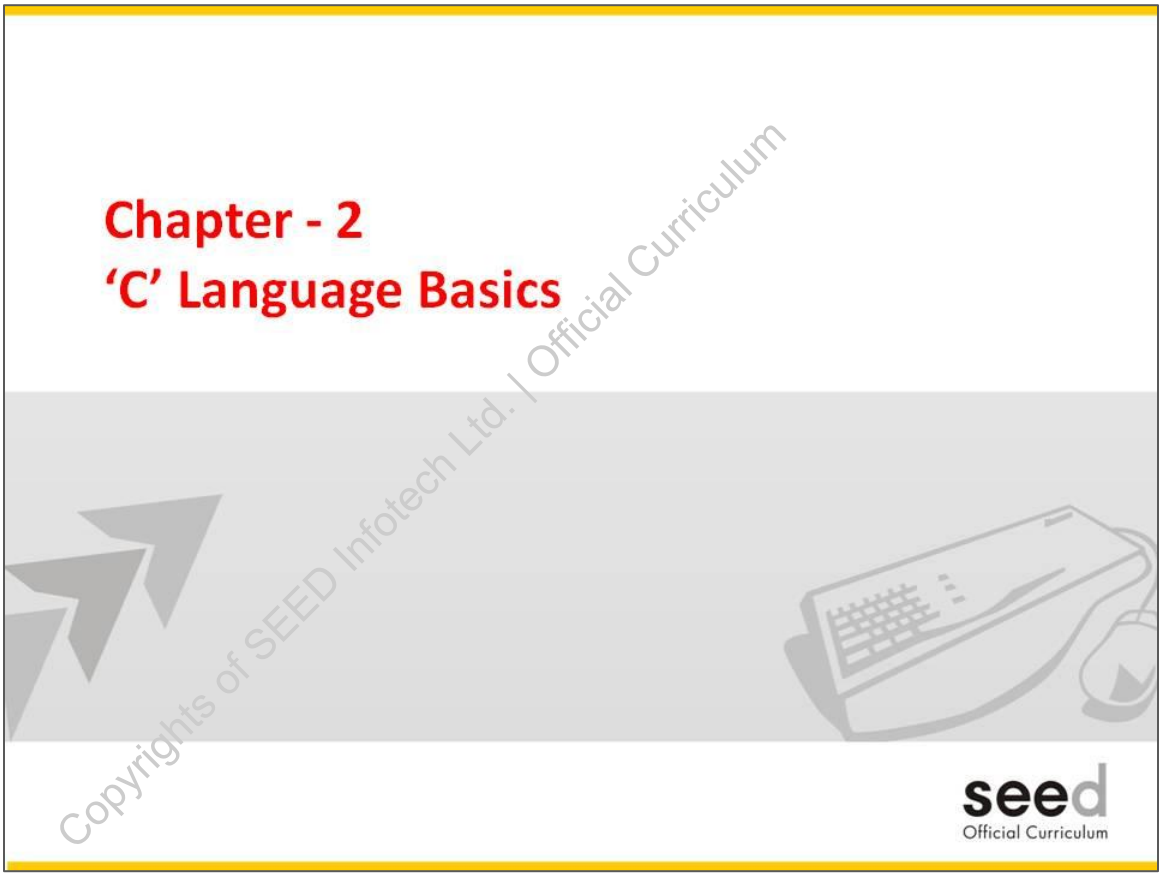
- Preprocessing: processes input data to produce output that is used as input to another program.
- Code generation: is the process by which a compiler's code generator converts some internal representation of source code into a form that is machine code, which can be readily executed by a machine.
- Code optimization: is the process of modifying a system to make some aspect of it to work more efficiently or use fewer resources.

The following are the steps to write, develop and execute a program in C:

1. Type the program in an editor like Notepad.
2. Save the program (`sample.c`)
3. Compile the program (`sample.obj`)
4. Link the program (`sample.exe`)

Why not directly link? Why do we have to compile the source code? This is because we can have more than one file which can be separately compiled and then linked together. This scenario can be seen when there is a team of developers developing an application. Each one of them can write code for the task assigned in a file. Each one of these files can then be compiled separately and linked together.

This will be clearer in chapter 5 - Functions.

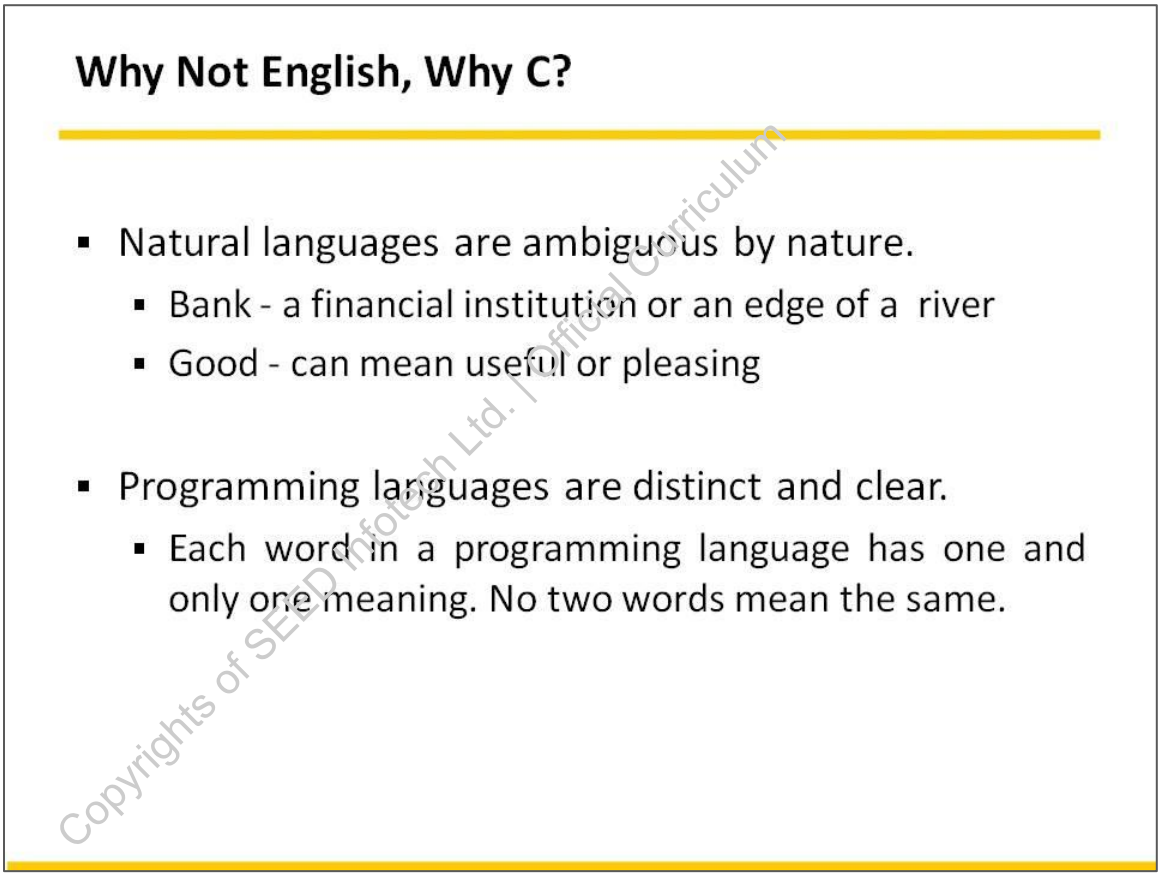


In this chapter covers the rules and the various words, operators, expressions, and library functions that can be used to write programs in C, which are its building blocks.

Objectives

At the end of this chapter you will be able to:

- State the need of programming language.
- Know the history of 'C' in brief.
- State the features of 'C' language.
- Use a variable.
- List basic data types in 'C'.
- Use expressions in a program.
- Use I/O library functions.
- State the need of type casting and use it.
- Construct 'C' programs using above mentioned features.



The languages that are used for communication between two people are ambiguous in nature. For example, suppose two people are having a dialogue. One says to the other, "I had been to the bank to deposit a cheque." The other person understands that it's a banking institution that is being referred and not the edge of a river.

In case of an instruction that has to be given to a machine, the instruction has to be very distinct and clear, because a machine cannot reason or think. Programming languages are therefore designed in such a way that an instruction given will have only one meaning because every statement in any programming language has one and only one meaning.

All natural languages are inherently ambiguous. One sentence can have more than one meaning hence it cannot be directly used for computer programming. All programming languages are designed by controlling ambiguity.

Programming Language - 'C' History

- 'C' was designed and developed by Brian Kernighan and Dennis Ritchie at the Bell Research Lab in 1972.
  - BCPL and B language was considered as a base.
- "K & R C" is the Kernighan and Ritchie's definitive description of the C language.
- ANSI C is the standard for 'C' defined by American National Standards Institute (ANSI).

Brian Kernighan

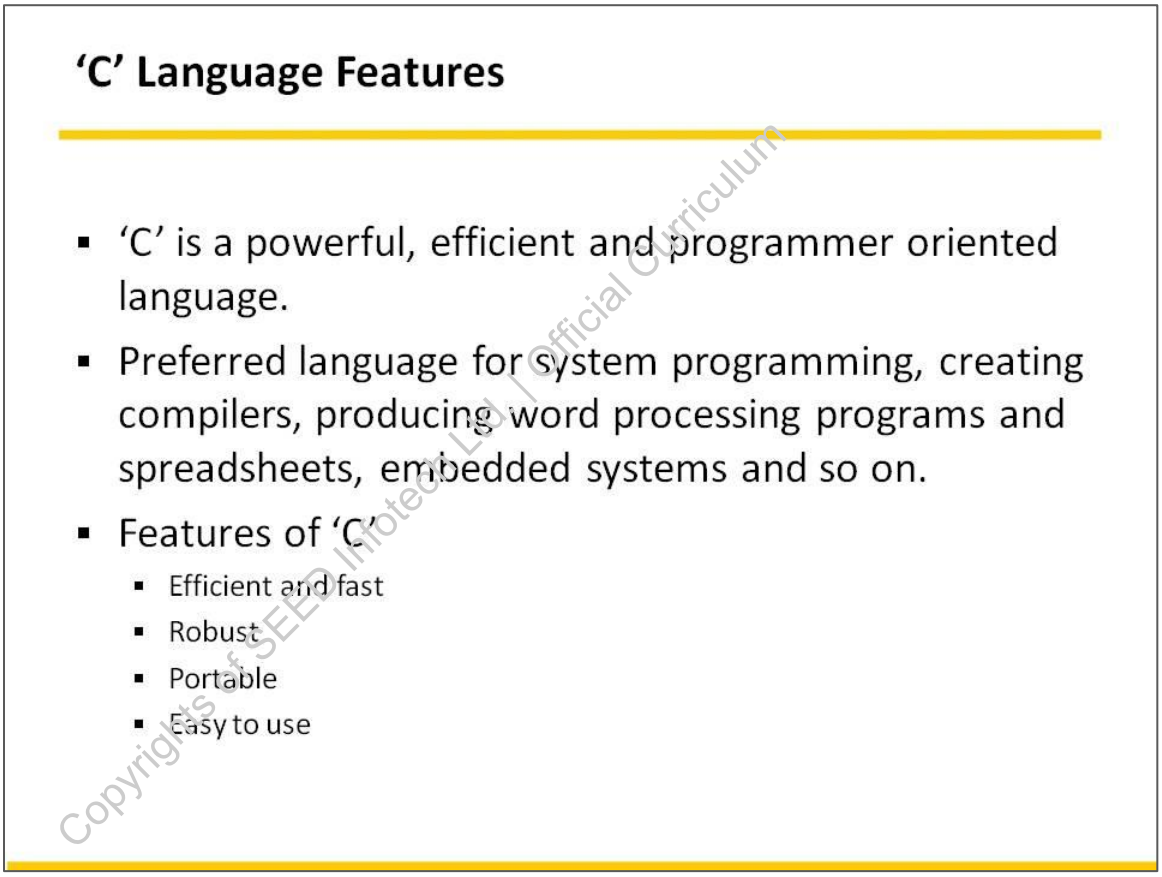
Dennis Ritchie

Martin Richards developed 'Basic Combined Programming Language' (BCPL) in 1967. It was primarily for writing system software. But BCPL was less powerful and too specific. In 1970, Ken Thompson created a language called 'B' using many features of BCPL at AT&T's Bell Labs. 'B' was used to create early versions of UNIX operating system at Bell Laboratories. But 'B' also was very specific. 'BCPL' and 'B' were type-less system programming languages.

'C' programming language was designed and developed by Brian Kernighan and Dennis Ritchie at 'The Bell Research Labs' in 1972. BCPL and B languages were used as a base. It was developed along with UNIX operating system and is strongly associated with UNIX.

Brian Kernighan and Ritchie published a definitive description of the language. The Kernighan and Ritchie description is commonly referred to as "K&R C".

In 1983, American National Standards Institute (ANSI) appointed a technical committee to define a standard for C. The committee approved a version of C in 1989, which is now known as ANSI C. It was then approved by the International Standards Organization (ISO) in 1990.



'C' is a powerful, efficient and programmer oriented language. It is compact and easy to learn, hence a popular choice of programmers. C is a portable language, which means that C programs written on one system can be run on other systems, but with some modifications whenever required.

Earlier, C was used in the minicomputer world of UNIX systems. Then, it was used on personal computers and mainframes. It is a preferred language for system programming, creating compilers, producing word processing programs and spreadsheets, embedded systems, developing computer games, and so on. The list is endless.

The following features make it a popular language:

Efficient

C programs are compact and are faster to execute. They can be written to give better programming efficiency like high level languages and at the same time achieve machine efficiency like low level languages.

Robust

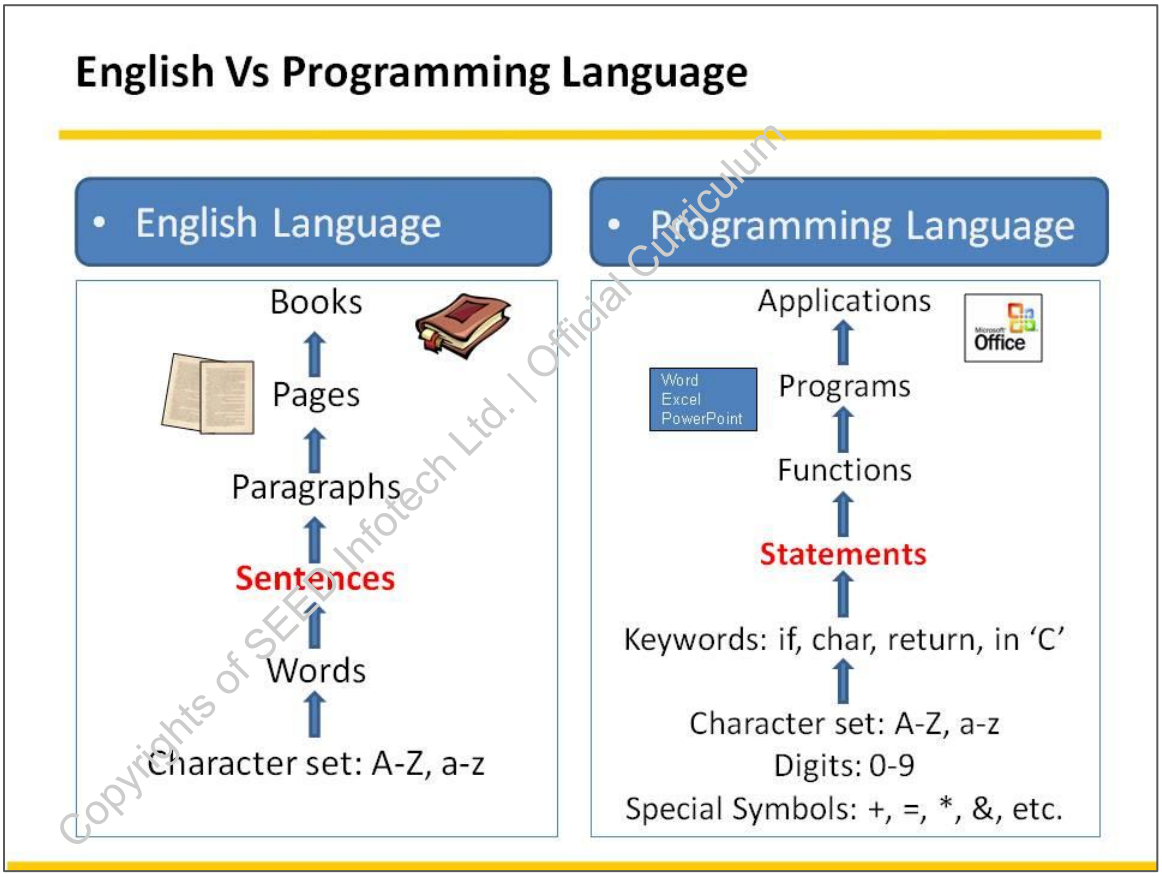
It can be used to design very complex systems with the help of its rich set of built-in functions. Powerful UNIX operating system is built using C language. Compilers of many languages like PASCAL are created using C language.

Portable

C programs written on one system can be run on other systems with very little or no modification. Only programs written for hardware devices or operating system are not portable.

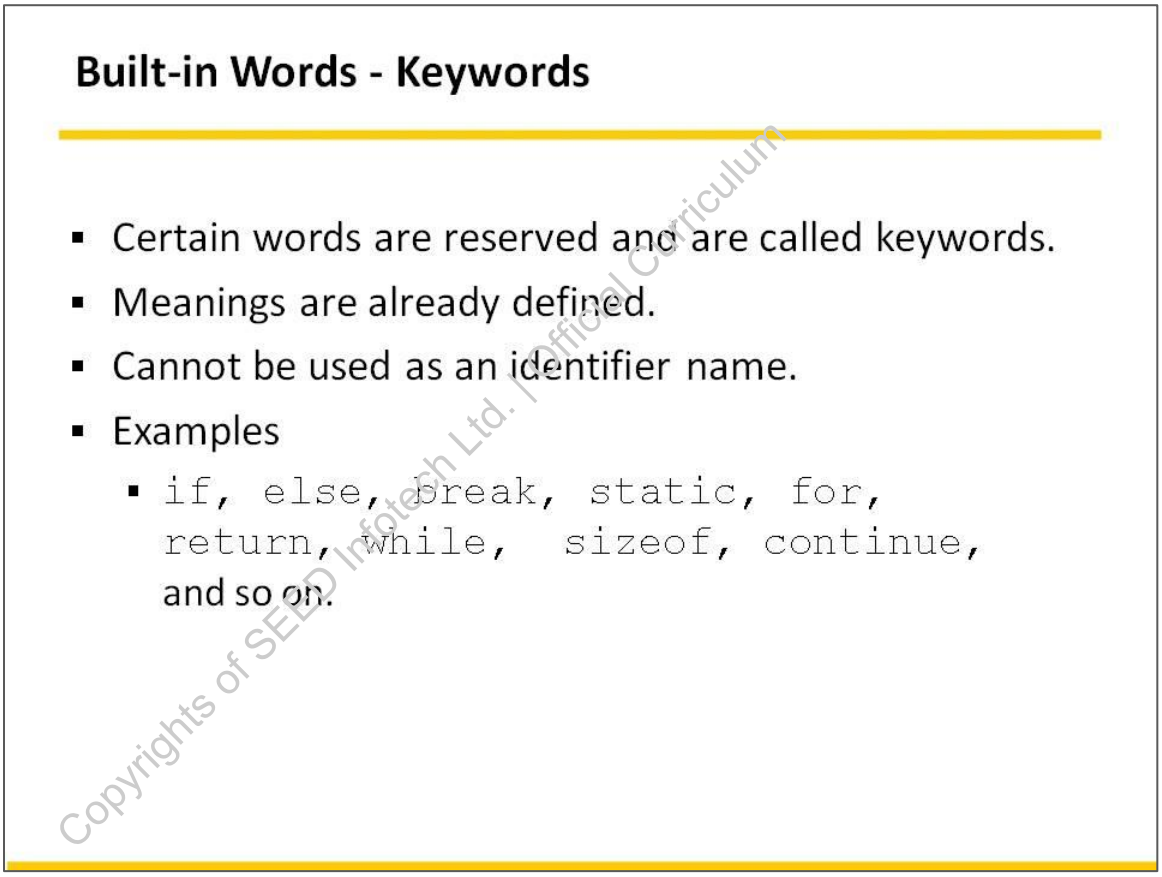
Easy to use

C has a rich set of operators, keywords and functions, which makes it a popular language amongst the programmers. It gives access to hardware and helps to manipulate individual bits in memory.



When a beginner learns a new language, its alphabets are learnt first. Then, simple and small words are constructed. This is followed by making small, simple sentences. But to be able to be fluent in the language, it is important to learn its grammar.

Applying the same analogy to learning a programming language, learning its basics is the first thing to do. Begin with its characters, special symbols, and its words, which have a special meaning - called keywords or reserved words. Knowing the rules of a programming language is necessary before beginning to write statements; then comes beginning to write small programs and later moving on to developing complex applications.



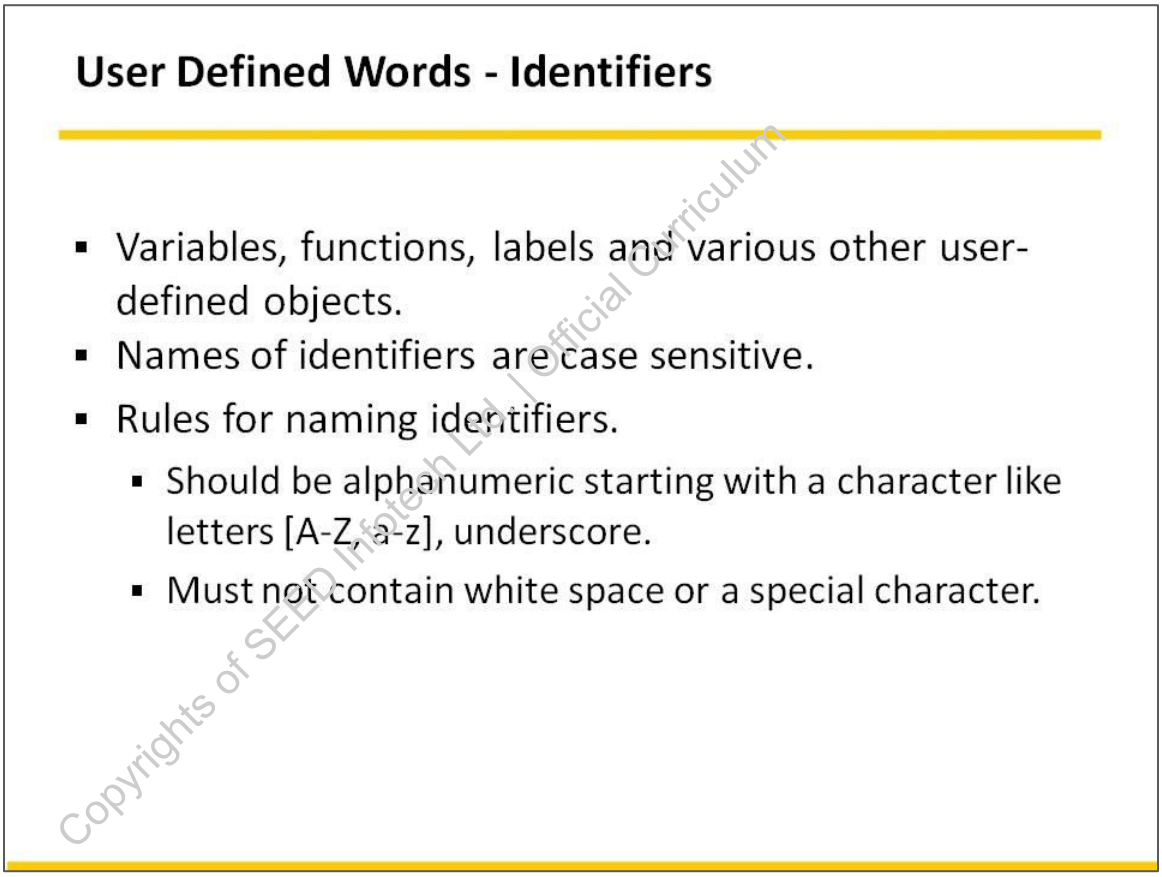
Keywords are the words which make up the language (analogous to the vocabulary of a language) and their meaning is pre-defined by the language. They are reserved words of the language.

- These words cannot be used for naming identifiers. If used as identifier names, the compiler generates an error.
- They are case sensitive and always written in small case letters.

The following table shows the list of ISO/ANSI C keywords.

ISO/ANSI C Keywords			
auto	enum	restrict	unsigned
break	extern	return	void
case	float	short	volatile
char	for	signed	while

ISO/ANSI C Keywords			
const	goto	sizeof	_Bool
continue	if	static	_Complex
default	inline	struct	_Imaginary
Do	int	switch	
double	long	typedef	
else	register	union	



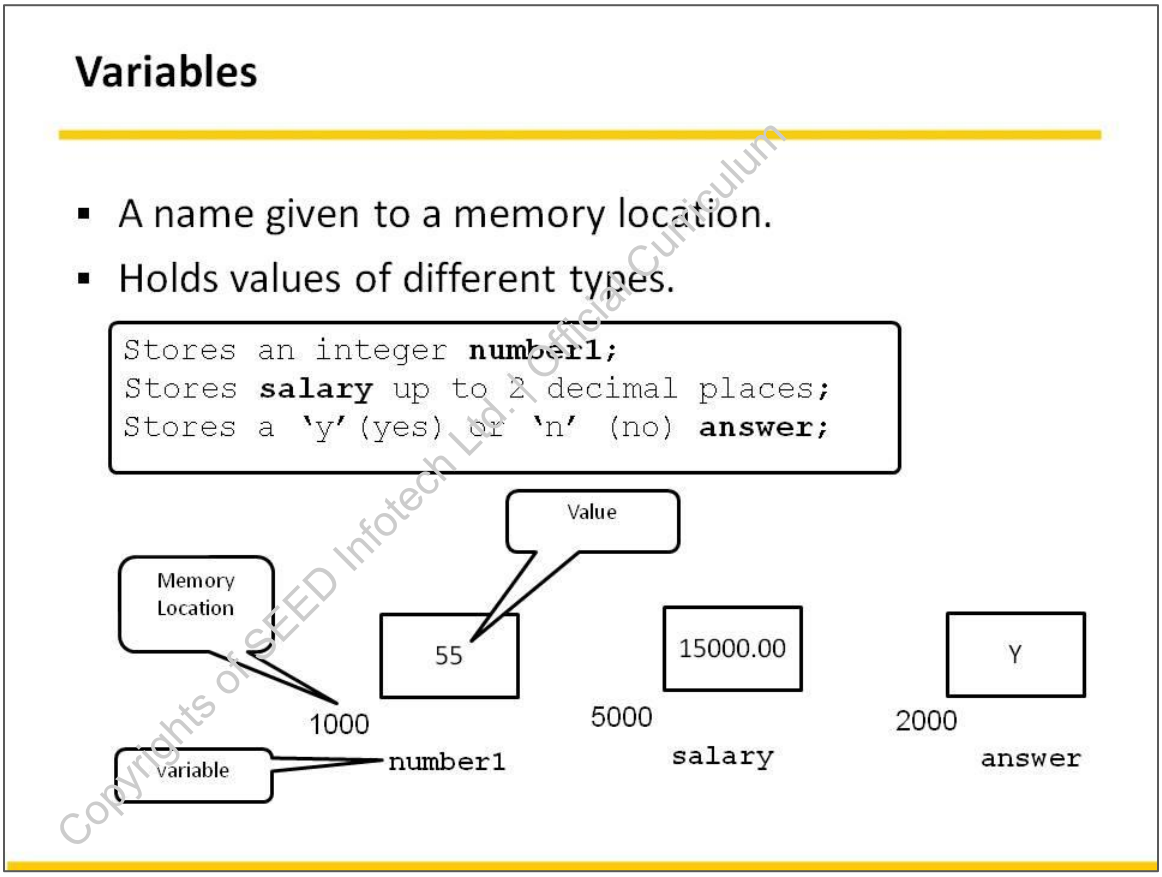
During program execution, there is a need to store information, some intermediate data and the final result. This information is stored in the computer's memory in different locations. To refer to these memory locations, identifiers are used. This is analogous to giving house numbers to be able to locate a particular house in a locality.

In C, the names of variables, functions, labels and various other user-defined objects are called identifiers. The language rules to be applied while naming identifiers are mentioned above.

C is said to be a case-sensitive language; meaning that the identifier "`rollNo`" is different from "`RollNo`".

Reserved identifiers like `printf()`, `scanf()`, etc. can be used as valid names for user defined identifiers. But C language defines them to serve some particular purpose like printing on screen, scanning the value from the user, as in this case. They might cause problems if they are used by the programmer for some other purpose. Hence they should not be used for user defined identifier names.

1. Give meaningful names to all identifiers. In real projects, naming convention is used across project for better readability.
2. Names with leading and trailing underscores are reserved for system purposes and should not be used for any user-created names.



To do the tasks assigned, a program needs to work with data (numbers, characters, decimal numbers, etc.). Some data may be subjected to change as the program executes. Such data is stored in variables.

A variable is a name given to a memory location where data is stored. The data can be accessed from the memory using the variable name. Depending on the type of data to be stored, variables are declared with the particular data type.

There are certain rules, which must be followed while naming a variable.

- A variable name can be any combination of alphabets, digits or underscores.
- The length of variable name can be up to 8 characters.
- The first character of the variable name must be an alphabet.
- No commas or blanks are allowed within a variable name and no special symbol other than underscore can be used.

'C' is a case sensitive language i.e. variable name 'NAME' and 'name' is different.

Every variable should be defined before it is used. Storage class is optional. If it is not mentioned, 'auto' storage class is considered. *Storage classes are covered later in the courseware.*

```
[storage-class] data-type variable-name;
```

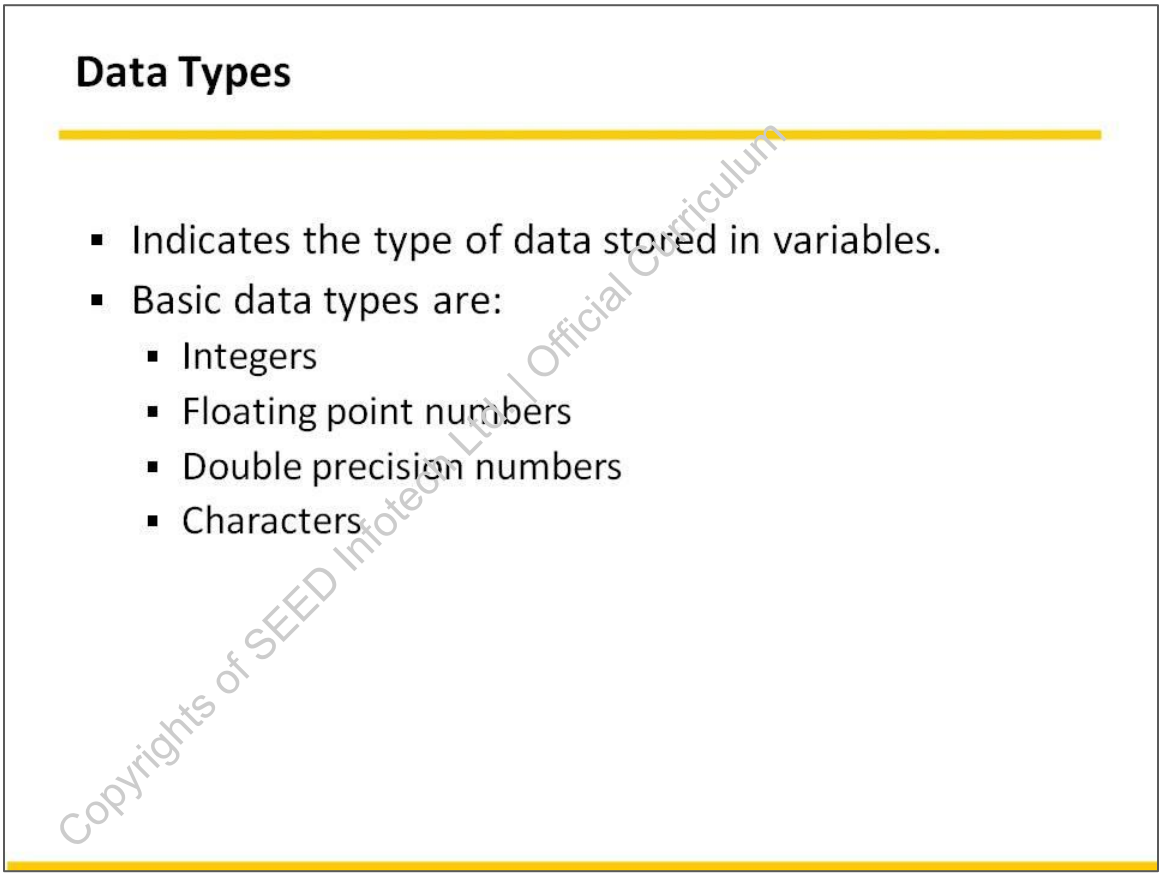
For example,

```
int number1;
float salary;
char answer;
```

Two or more variables of same data type can be declared in one statement.

```
int number1, number2;
```

Different variables are stored at different memory locations and hold different values. Each memory location has a number, which denotes its address.



Information or data to be stored in a computer consists of different types - numbers, characters, decimals, and so on. The computer needs a way to identify and use these different types of data.

Basic Data types

The built-in data types are called the basic data types.

- Roll number, department number, serial number, etc should be stored as integers. These numbers have no fractional part. To store such a kind of information, `int` data type is used.
- Temperature, salary, etc. are floating point numbers. These numbers are real numbers. `float` data type is used to store this type of information.
- To store numbers with more precision, `double` data type is used.
- `char` data type can be used to store a character or a set of characters (strings). Storing 'y' or 'n' (as in yes or no), name of the person, name of a city, phone number with characters like '-', and so on, `char` data type is used.

The accompanying table shows the different format specifiers to be used for different data types.

TYPE	'C' KEYWORD	FORMAT SPECIFIER
Character	`char`	`%c`
Integer	`int`	`%d`
Floating point	`float`	`%f`
Double	`double`	`%lf`

C provides modifiers like `short`, `long`, `signed`, and `unsigned` to b used with the basic data types. When only small numbers are required to be stored, `short int` is used. This requires less memory space than `int`. Similarly, to store numbers larger than `int`, `long int` is used.

```
short int id;
long int ac_no;
unsigned char c;
```

Using modifiers changes the range of numbers that can be stored normally by the basic data types, as can be seen from the accompanying table (On Turbo C complier).

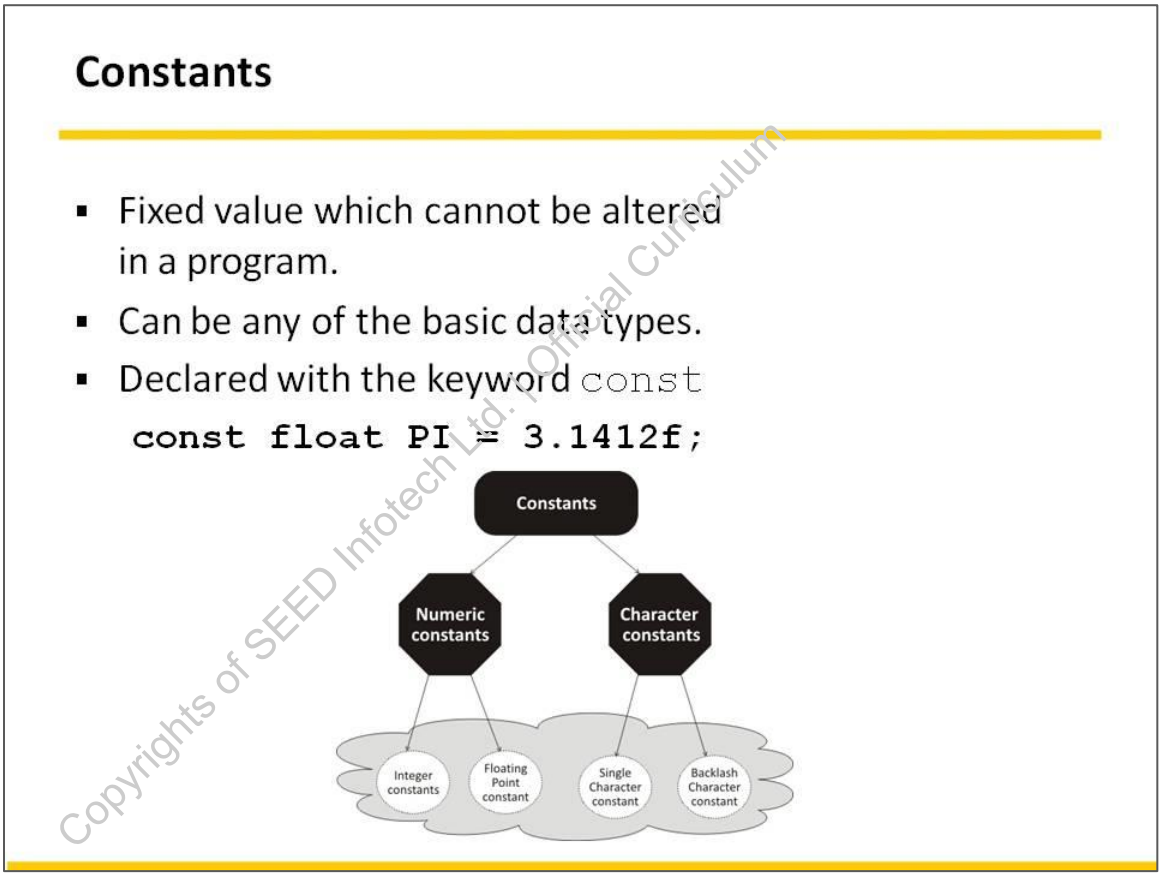
Data Type	Range	Bytes
signed char	-128 to + 127	1
unsigned char	0 to 255	1
short signed int	-32768 to +32767	2
short unsigned int	0 to 65535	2
long signed int	-2147483648 to 2147483647	4
unsigned int	0 to 4294967295	4

float	-3.4E38 to +3.4E38	4
double	-1.7E308 to 1.7E308	8
long double	-1.7E4932 to +1.7E4932	10

Read more about other data types at

http://msdn.microsoft.com/en-US/library/cc953fe1(v=VS.80).aspx

Data type is a very useful feature for developer. Compiler can catch and warn about errors such as adding number to character which is erroneous.



Data that should remain unchanged throughout the life of a program is defined as constant. It can be integer, character or floating point. The program cannot change it.

Integer constants

Integer constant is an integer quantity and has no decimal point. It can be either negative or positive. No commas or blanks are allowed in an integer constant.

- A decimal integer constant can consist of any combination of digits from 0 through 9.

```
const int deptCode = 101;
```

- An octal integer constant can consist of any combination of digits from 0 through 7. The first digit must be 0. For example, `0` or `074` are valid octal constants.
- A hexadecimal integer constant must begin with 0x or 0X, followed by any combination of digits from 0 through 9 and A through F or a to f. For example, 0x , 0x1, 0XAF7F

Certain suffixes can be given to constants, which identify them for the type.

Type	Suffix	Example
long	l or L	5000L long (decimal)
unsigned long	ul or UL	1234UL unsigned long (decimal)

Floating point constants

Floating point constant contains a decimal point or an exponent. It could be either negative or positive. No commas or blanks are allowed in a floating point constant.

```
const float PI = 3.1412f;
```

Type	Suffix
Float	f or F
Long double	l or L

Character constants

Single character constant is a character enclosed in single quotation marks. It can contain only one character.

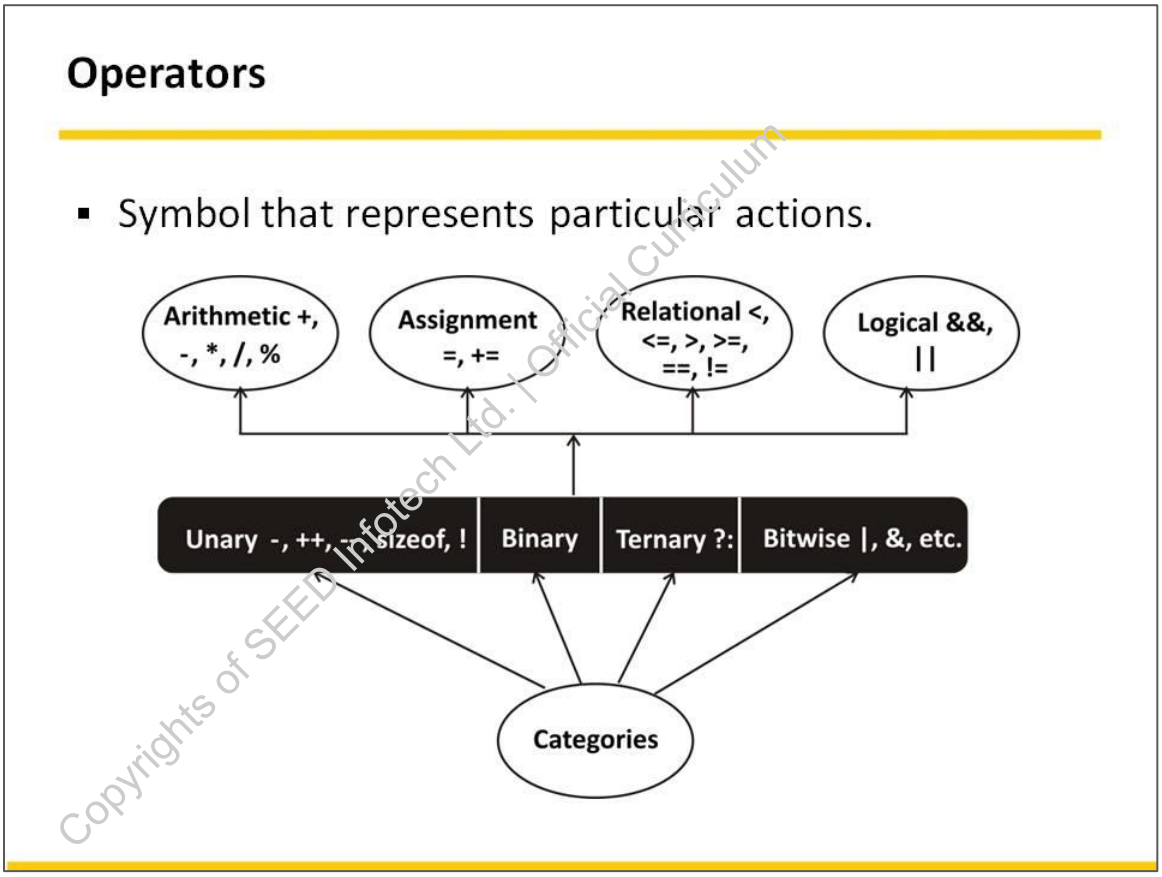
```
const char grade = 'A';
const char symbol = '$';
const char choice = '3';
```

Each character has a corresponding ASCII value (which is the numeric value of character in the machine's character set.) as given below.

Character	ASCII
'A'	65
'B'	66

Backslash character constants

Some characters cannot be entered into a string from the keyboard. For this reason, C includes special "backslash" characters. For example, `\n` represents the new line character.



Operators are symbols which are used to perform particular actions on different types of data. There are four types of operators - unary, binary, ternary and bitwise.

Unary operators

Unary operators require only one operand to perform an action. Minus operator, increment and decrement operators and `sizeof` operator are examples of unary operators. Consider the examples mentioned below.

Minus operator: Minus sign precedes a numerical constant, a variable or an expression. It is used for negation. It is different from arithmetic subtraction operator. For example,

```
-10, -5.54
```

Increment and decrement operators: These are used for incrementing and decrementing variables. Increment operator '++' adds 1 to its operand. While decrement operator '--' subtracts 1 from its operand. The operator may be post-

fixed or prefixed. Post-fixing and prefixing makes a difference when the operators are used in an expression.

```
int number1, number2 = 4;
number1 = number2++;            ----- I
number1 = ++number2;            ----- II
```

In the statement I, value of `number2` is first assigned to `number1` and then incremented by 1. Hence, `number1` = 4 and `number2` = 5;

In the statement II, value of number2 is first incremented by 1 and then assigned to number1. Hence, `number1 = 5` and `number2 = 5`.

The post and pre decrement operators work in a similar way, except that the value of the variable is decreased by 1.

`sizeof` operator - it returns the size of its operand in bytes.

```
int i, j;
j = sizeof(i);          //i=2. Since integer occupies 2
bytes
j = sizeof(char)        //j=1. Since, character
occupies 1 byte.
```

`NOT(!)` operator is used to negate the existing value.

Binary operators

The operators which require two operands to perform an action are called binary operators. They are as follows:

Arithmetic Operators

These operators perform arithmetical operations of addition (+), subtraction (−), multiplication (*) and division (/). The division operator performs integer division; that is `5/2` will evaluate to 2, not to 2.5. But if the evaluated answer is being stored in a float type variable, then the value stored in it will be 2.0, which is a float value.

The modulus operator (%) returns the remainder after performing the division operation. It cannot be used with float data type. Thus, 5%2 will be evaluated to 1. For example,

```
c = a + b;
c = a % d;
```


Assignment Operator

It is the value assigning operator. After evaluating the expression on the RHS, it assigns the value to the variable on the LHS. For example,

```
a = b * c;
```

The assignment operator can be used in another way too. Consider the expression,

```
j = j + 2;
```

The variable on the left is repeated immediately on the right. Therefore, the same expression can be written as

```
j += 2;
```

The operator '+=' is called the assignment operator. Most binary operators have corresponding assignment operators. General form of assignment expression being

```
expr1 op = expr2 ;
```

Instead of

```
expr1 = (expr1) op (expr2);
```

where expr1 and expr2 are expressions and op can be one of the following operators:

```
+       -       *       /       %       <<      >>      &       ^
|
```

Relational Operators

Relational Operators like <, >, <=, >=, ==, !=, perform comparisons between two operands. All the relational expressions evaluate to a 0 if the expression is false and to a non-zero value if the expression evaluates as `true`. For example, two variables can be compared as mentioned in the following expression.

```
c = a > b;          //return 1 if a is greater than b.
0 otherwise
c = a >= b;    //returns 1 if a is greater than or
equal to b.
               // 0 otherwise
c = a == b;
c = a != b;
c = a < b;
```


Logical Operators

AND (`&&`) and OR (`||`) operators are used to perform logical comparisons. Example to find the largest of the three numbers is mentioned below.

```
(a > b) && (a > c)
```

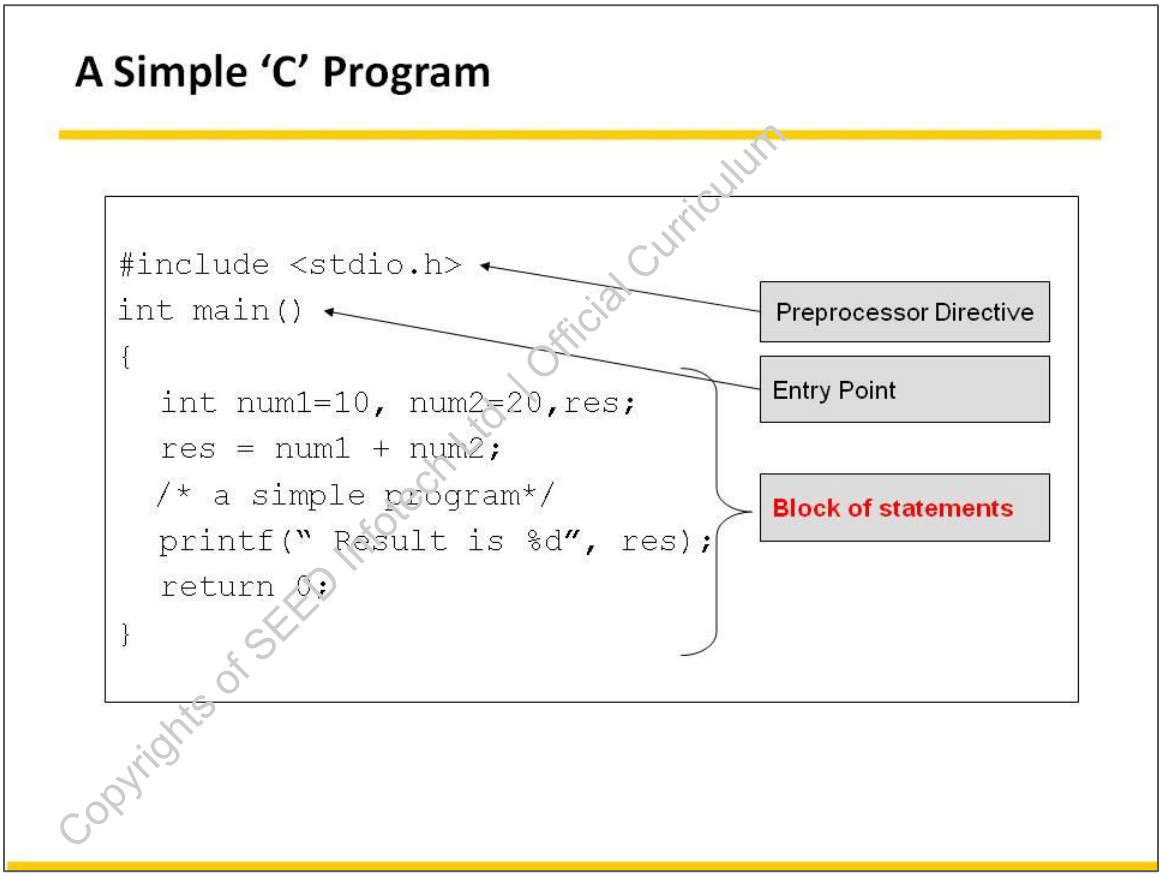
Ternary Operator

It requires three operands to perform an action. The ternary operator can be used to find the larger of two numbers as shown below.

```
a > b ? a : b
```

Bitwise Operators

There are three types of bitwise operators – one's complement operator `(~)`, logical bitwise operators `(&, |, ^)` and the shift operators `(>>, <<)`. They perform operations on individual bits and thus can be used in hardware manipulations.



- `#include <stdio.h>`

  This statement includes the information found in `stdio.h` file before compilation.

- `int main(void)`

  It is the entry point of a C program. It is always the first function called.

- { … }

  These opening and closing brace bracket mark the beginning and the end of the statements that make up the function or a block of statements.

- `int num1 = 10;`

  The above statement defines a variable called `num1` that will be used in the program and will be of `int` (integer) data type and is initialized to 10. All variables must be defined before they are used in the program.

- `/* a simple program */`

The symbols `/*` and `*/` enclose comments, which are remarks that help clarify a program. They are intended only for the reader and are ignored by the compiler.

- `printf("Result is %d", res);`

  This statement displays "**Result is 30"**, leaving the cursor on the same line. The `%d` instructs the computer where and in what form to print the value of `num1`.

- `return 0;`

  This is the `return` statement which returns the control to the calling function.

- `;`

  Semicolon (;) marks the end of every statement in C.

Format Specifiers

- Define the output format for display of each item
  - Decided by what type of data is to be read or displayed in the formatted I/O functions
- For Example :

%c	Character
%s	String of characters
%d	Signed decimal integer
%f	Floating point numbers

Internally all data is represented in a format understood by the computer, i.e., 1s and 0s. The user inputs data in a format understood by the user, which are numbers and characters. But for the C compiler to understand the data type, the format of the data has to be specified.

Different data types have different a format specifier. When the compiler comes across a particular format specifier, it knows which data type to expect and performs the actions according to the instructions given by the program.

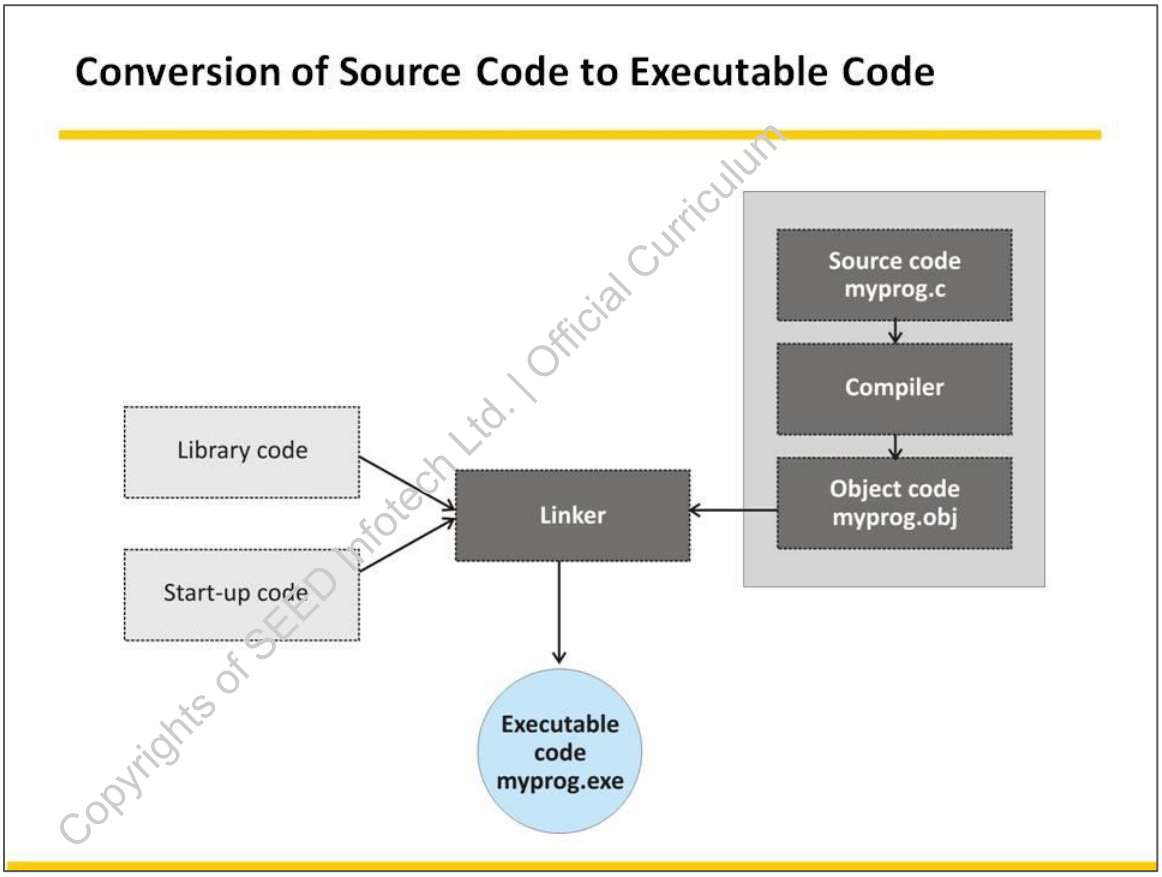
The following code snippet demonstrates how the format specifier is used with formatted I/O functions:

```
char choice;
char name[15];
int rollNo;
float percent;
printf("\nEnter your choice (Y/N)");
```

```
scanf("%c", &choice);
printf("\nEnter the name, roll number and percent marks: ");
scanf("%d %s %f", &rollNo, name, &percent);
```

Note: Strings will be covered later in detail.

Conversion Character	Meaning
c	Data item is a single character.
d	Data item is a decimal integer.
e	Data item is a floating-point value.
f	Data item is a floating-point value.
g	Data item is a floating-point value.
h	Data item is a short integer.
i	Data item is a decimal, hexadecimal or octal integer.
o	Data item is an octal integer.
s	Data item is a string followed by a whitespace character (the null character \0 will automatically be added at the end.)
u	Data item is an unsigned decimal integer.
x	Data item is a hexadecimal integer.
[…]	Data item is a string, which may include white space characters.



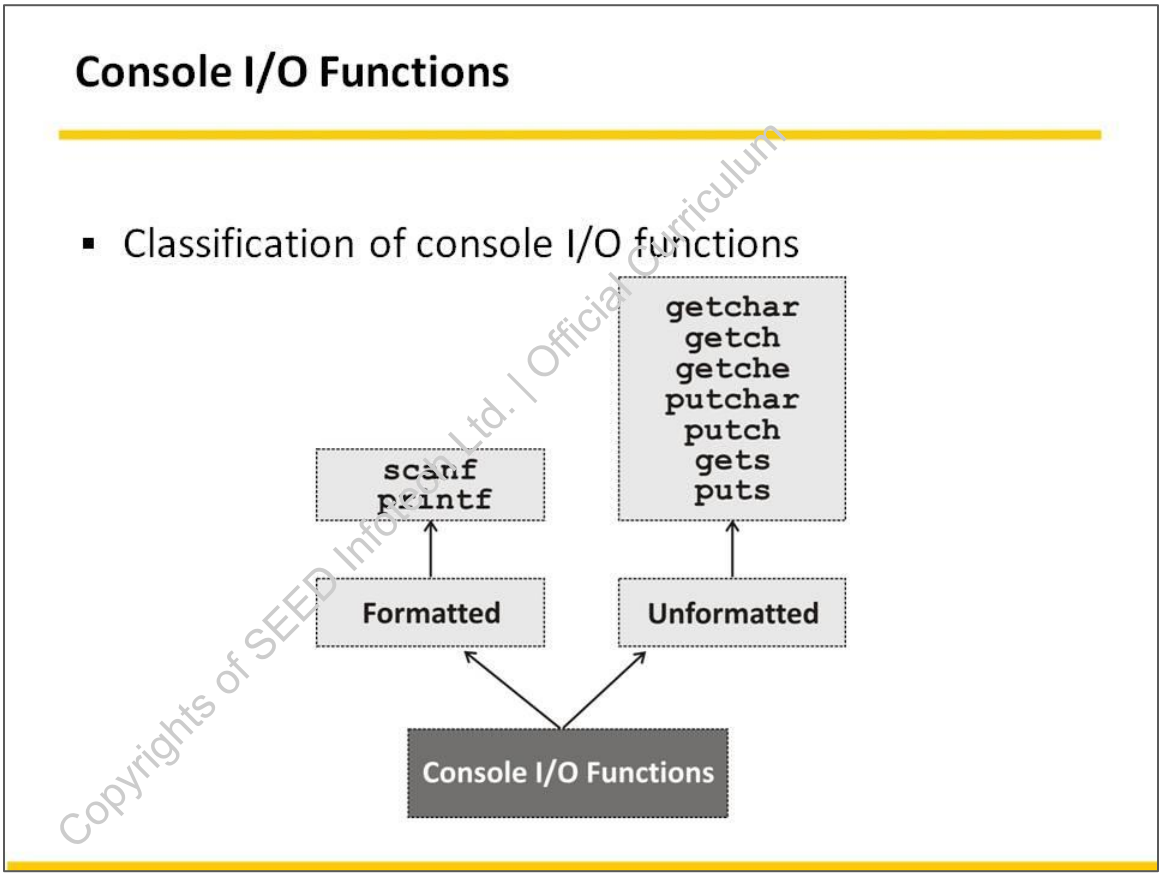
Following are the steps in the conversion of a C source code to executable code.

1. A source code file myprog.c is first created as per the problem statement.
2. This file has to be now compiled. The compiler translates the source code to machine understandable code and generates an object code file: myprog.obj. The object code file contains the source code converted to machine language. It is not in “ready to run” state.
3. To bring the program to read-to-run state, it has to be linked.
4. There are two elements which are not present in the object code to make it executable. First is the start-up code which is the code that acts as an interface between the program and the operating system. Another missing element is the code for library routines. The role of the linker is to bring together the object code, the standard startup code and the library code and put them together in an executable file. The executable file is in a “ready-to-run” state.

To summarize, an object file and an executable file both consist of machine language instructions. However, the object file contains the machine language translation only for the source code, but the executable file also has machine code for the library code used and for the startup code.

On some systems, programs to compile and link are separate. On other systems, the compiler starts the linker automatically, so only the compile command needs to be given.

Understand the process of compilation and execution of C program thoroughly. You need to know what happens in each step.



Large number of library functions that carry out commonly used calculations and operations accompany 'C'. These library functions are not a part of the language but are provided with the compiler.

Few library functions are given below.

Function	Purpose
`abs(j)`	Returns absolute value of j.
`sin(d)`	Returns sine of d.
`getchar()`	Accepts a character from standard i/p device.

In order to use these functions they have to be first declared. This information is stored in special files called the header files. Each header file contains information about a group of related library functions. The required information can be

obtained by accessing these header files in the program. This can be done using the preprocessor directive `#include<file name>`.

Note: Functions and preprocessor directives will be studied in detail in later chapters.

Input and output is an important part of any program execution. Many times, user interaction is required during program execution; input needs to be taken from the user and output is also required to be shown. This is achieved through some built-in (ready-to-use) functions. They are called console input - output (I/O) functions because they make use of the screen or the console.

There are two types of console input-output functions - formatted and unformatted I/O functions.

Formatted I/O functions

There are two formatted functions `printf()` and `scanf()`.

`printf()`

This function is used to display output on the screen. It returns the number of bytes that are output.

```
printf("control string", arg1, arg2, …);
```

`arg1, arg2`, and so on, are the items to be printed. They can be variables or constants, array names or even expressions that are evaluated first before the value is printed. `Control-string` is a character string describing how the items are to be printed. The control string should contain a conversion specifier for each item to be printed. For example,

```
int rollno = 101;
float average = 75.6f;
char grade = 'A';
printf("Roll No: %d, Average = %f, Grade %c", rollno, average, grade);
```

`scanf()`

This function is used to accept data from an input device like a keyboard and is stored in the respective variables. It returns the number of fields accepted,

converted and stored successfully. The `scanf()` function requires an enter key to be pressed after supplying the input, in order to accept the input.

```
scanf("control string", arg1, arg2, …);
```

Like `printf()`, `scanf()` uses a control string followed by a list of arguments. The control string should contain a conversion specifier for each item to be accepted. The main difference is in the argument list. The `printf()` function uses variable names, constants, and expressions. The `scanf()` function uses address of variables. For example,

```
int rollno;
float average;
char grade;
scanf("%d%f%c", &rollNo, &average, &grade);
```

The conversion specifiers for `scanf()` are the same as those used for `printf()`, with […] being an additional format specifier for `scanf`. A string with blank spaces cannot be accepted by using the conversion specifier `%s`. This can be achieved by using […]. It can be used in the following way.

```
scanf("%[ABC' 'D]",line);
```

Then `scanf()` reads characters from input as long as they match characters specified in the square brackets and terminates on a mismatch. The order of entering characters is not important.

```
char str[15];
scanf("%[abcd]",str);      //accepts only the
characters mentioned
scanf("%[^r'\n ']",str);  //accepts all characters,
even a white space // except 'r' and the newline
character
```

If the characters in square brackets are preceded with a circumflex (^), it has the opposite effect. `scanf()` reads characters from input as long as no character from the square bracket is encountered.

Unformatted I/O functions

There are many console I/O functions for unformatted input and output - some for accepting and displaying characters and some for accepting and displaying strings.

Character reading and writing functions can read and display only one character at a time, whereas string input and output functions can read and display more than one character.

Following are the character input and output functions:

`getchar()` – gets one character at a time from the standard input device and returns the character typed. It waits for the enter key to be pressed.

`getche()` - gets a character from the keyboard and echoes it to the screen. If a mistake is made while typing, the backspace key cannot be used to correct it because as soon as the character is typed, it is gobbled up by the program.

`getch()` - gets a character directly from the keyboard. It does not echo the character to the screen and returns the character typed.


```
char ch;
ch = getchar();
ch = getche();
ch = getch();
```

`putchar()` - it is the counterpart of `getchar()` and displays a single character on the screen.

`putch()` - it is the counterpart of `getch()` and displays a single character on the screen.

```
char ch;
putchar(ch);
putch(ch);
```

All the above are character I/O functions available for the input and output of strings by reading or writing one character at a time.

`gets()` and `puts()` are functions for the input and output of strings directly. Strings are a sequence characters which are terminated with the `'\0'` character.

This character is called the `NULL` character. These functions facilitate the transfer of strings between the computer and the standard I/O devices.

`gets()` - is used to accept a string which may include white space characters. The enter key has to be pressed to indicate the end of the string.

`puts()` - is used to display a string. Each call to the `puts()` function displays the output on a newline. The newline character is appended by `puts()`.

```
char line[80];
gets(line);
puts(line);
```

Input Data

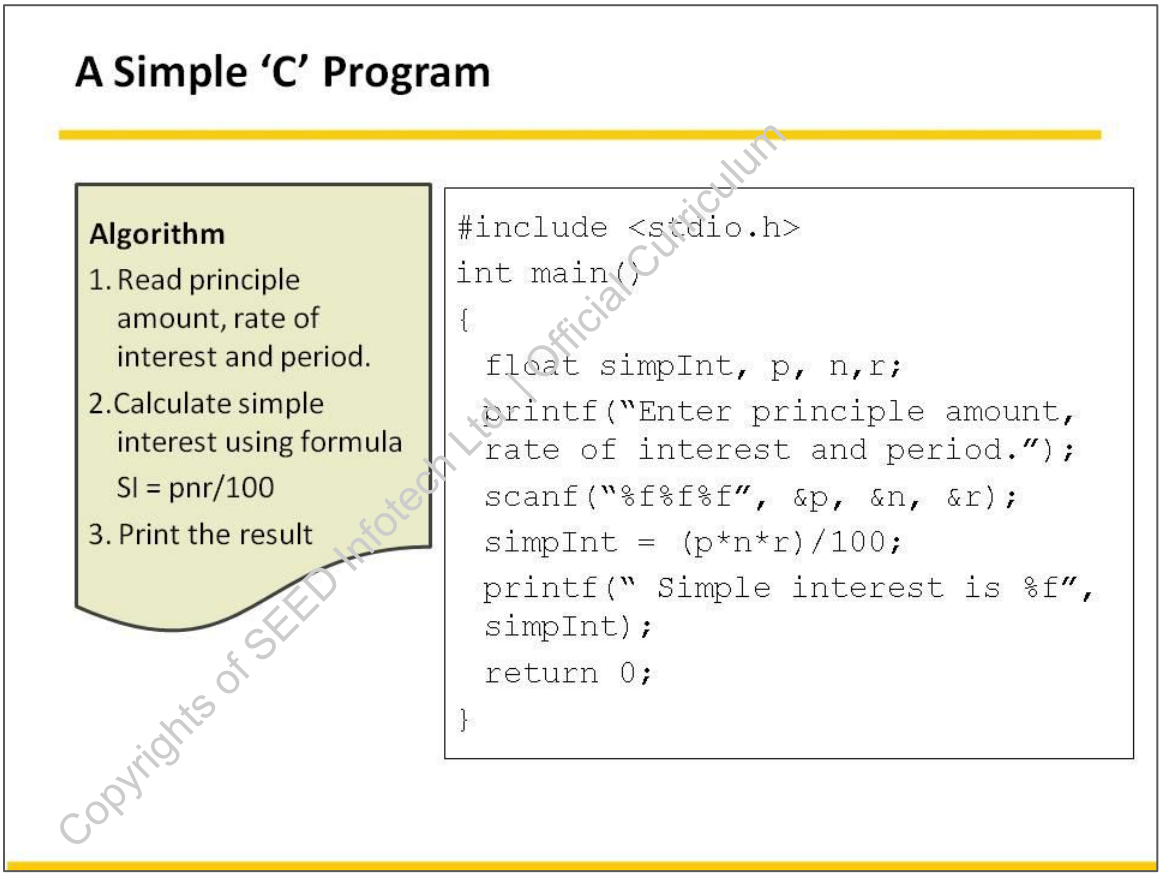
- The field specification for reading an input number is: %wd
  - w - [Optional] is an integer number which specifies the field width of the number to be read
  - d - Data type character, which indicates that the number to be read is of integer type

    For example,

    ```
    scanf ("%2d %4d",&num1, &num2);
    ```
- Field width specification can be used to validate input data.

When taking input from the user, the number of characters to be read can be restricted. This can be done by specifying the field width 'w', as in `%2d`. This will restrict the number of digits to be read. That is 99 will be the highest number that can be input by the user. `%4d` will restrict the input digits to 4. So, 9999 will be the highest number that can be read by the program.

Thus, input can be validated by controlling the width or the number of digits typed by the user.



A simple program in C can be easily understood from all the earlier topics that have been explained.

In the program above, four float variables have been defined. The values of principal amount, period and rate of interest are accepted from the user and the simple interest is calculated using the formula for simple interest. The interest is then displayed.

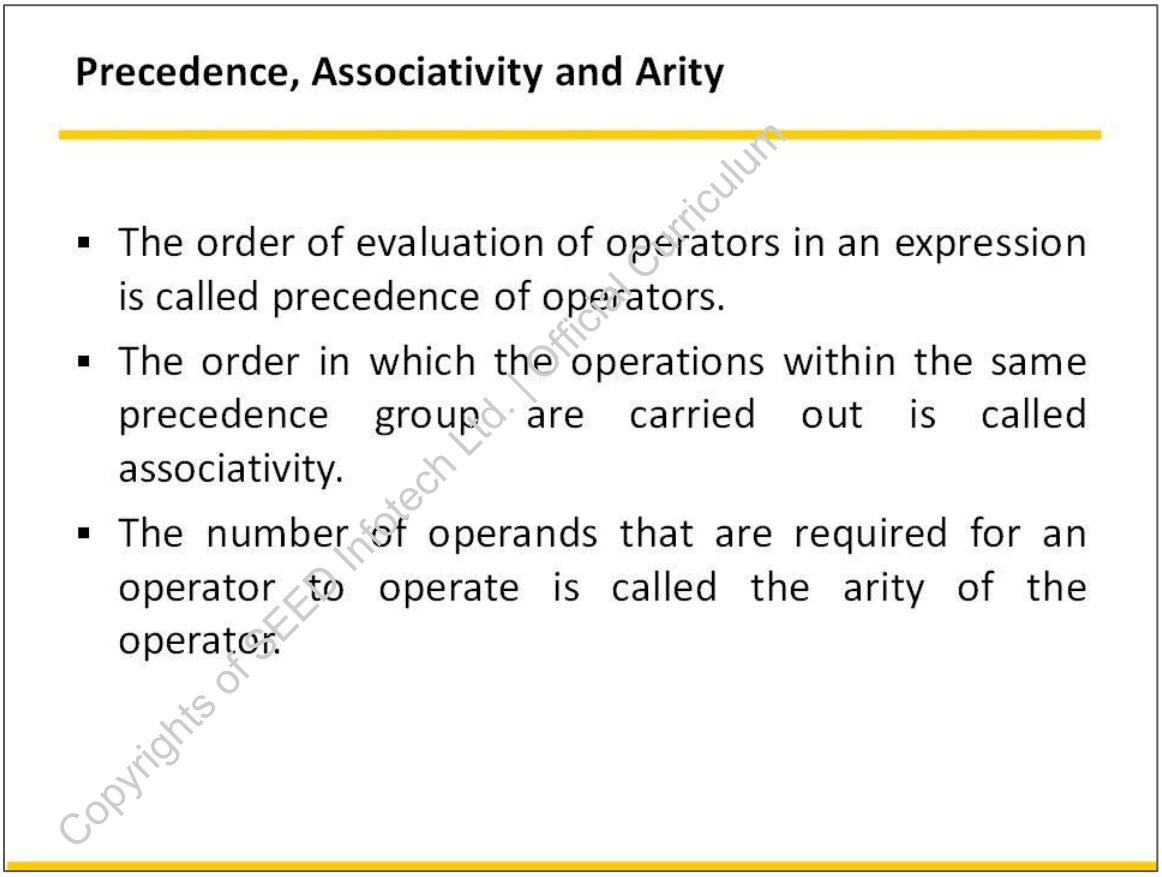
Escape Sequences

Character combination consisting of a backslash (\) followed by a letter or by a combination of digits is called an "escape sequence." The backward slash is called the escape character because it makes the character following it escape from its original meaning and gives a special meaning to it. To represent a newline character, single quotation mark, or certain other characters in a character constant, escape sequences must be used. An escape sequence is regarded as a single character and is therefore valid as a character constant.

Escape sequences are typically used to specify actions such as carriage returns and tab positions on terminals and printers. They are also used to provide literal representations of nonprinting characters and characters that usually have special meanings, such as the double quotation mark ("). The following table lists the ANSI escape sequences and what they represent.

Escape Sequence	Meaning	Purpose
\a	Alert (ANSI C)	Alerts user by sounding speaker inside computer.
\b	Backspace	Moves the cursor one position to the left of the current position.
\f	Form feed	Advances the computer stationary attached to the printer to the top of the next page.
\n	New line	Takes cursor to the beginning of the next line.
\r	Carriage return	Takes cursor to the beginning of a line in which it is placed.
\t	Horizontal tab	Moves cursor to next tab stop. Tab stop: An 80-column screen has 10 tab stops i.e. the screen is divided into 10 zones of 8 columns each. Printing a tab takes the cursor to the beginning of the next printing zone.
\\	Backslash(\)	Prints a black-slash.

\'	Single quote (')	Prints a single quote.
\"	Double quote (")	Prints double quotes.
\?	Question mark (?)	Prints a question mark.
\Ooo	Octal value	o represents an octal digit.
\xhh	Hexadecimal value	h represent a hexadecimal digit.



Different types of operators are used in an expression. These operators have different priorities for evaluation. Some have the highest priority while some have the lowest. The table above shows the precedence and the associativity of some of the operators.

Precedence

The order of evaluation of operators in an expression is called precedence of operators. The natural order of precedence can be altered by using parentheses. Sometimes, when more than one operator has the same precedence level, all of them are said to belong to the same precedence group. For example, the multiplication and division operators belong to the same precedence group whereas the addition and subtraction form another precedence group.

Associativity

The order in which the operations within the same precedence group are carried out is called associativity. Most of the operators have associativity from left to right except all the assignment operators and the `++` (prefix), `--` (prefix), `-`, `+`,

`~`, `!`, `sizeof`, `*` (dereference), `&` (address), `(type)`, (all unary) and the ternary operator.

Arity

The number of operands that are required for an operator to operate is called the arity of the operator.

Example

The following two expressions are evaluated step by step, to demonstrate the precedence and associativity of operators. Assume, `a = 10, b = 12, c = 3, d = 2` are all integer values.

```
a – b / c * d    ---------- I
a – 12 / 3 * d
a – 4 * 2
10 – 8
2
```


In expression I, multiplication and division operators belong to the same precedence group; but the associativity is from left to right. So, `b/c` would be evaluated first.

```
a – b / (c * d)     ----------- II
a – b / 6
a – 12 / 6
a – 2
8
```

In expression II, use of parentheses has forced `c*d` to be evaluated before `b/c`, thus over riding the natural order of precedence.

To make code more readable, use vertical and horizontal whitespaces generously. Indentation and spacing should reflect block structure of code.

Type Casting

- Converting the value of an expression of one data type into another data type.
- Types
  - Implicit and explicit type casting

```
...
int x = 5, y = 2;
float a;
a = x / y;              //implicit
a = x / 2.0f;           //implicit
a = (float)x/y;         //explicit
...
```


The compiler causes internal type conversion automatically when it has to handle operands of different types. There is also provision for programmers to explicitly convert value of an expression into another data type. To do so, the expression must be preceded by the name of the desired data type enclosed in parentheses.

```
(data-type) expression;
```

This is called **type casting**, done with a unary operator cast. For example,

```
{  . . .
  int x = 5, y = 2;
  float a;
  a = x/y;
  printf("\na = %.2f", a);          // will print 2.00
  a = (float)x/y;
  printf("\na = %.2f", a);          // will print 2.50
```

```
  printf("\nx = %d", x);       // will print 5
}
```

The expression `x/y` will cause integer division (i.e. decimal part will be truncated) and the result will be `2`. It will then be converted to float (2.00) since it is to be assigned to `a`, which is a float. This is called implicit type casting.

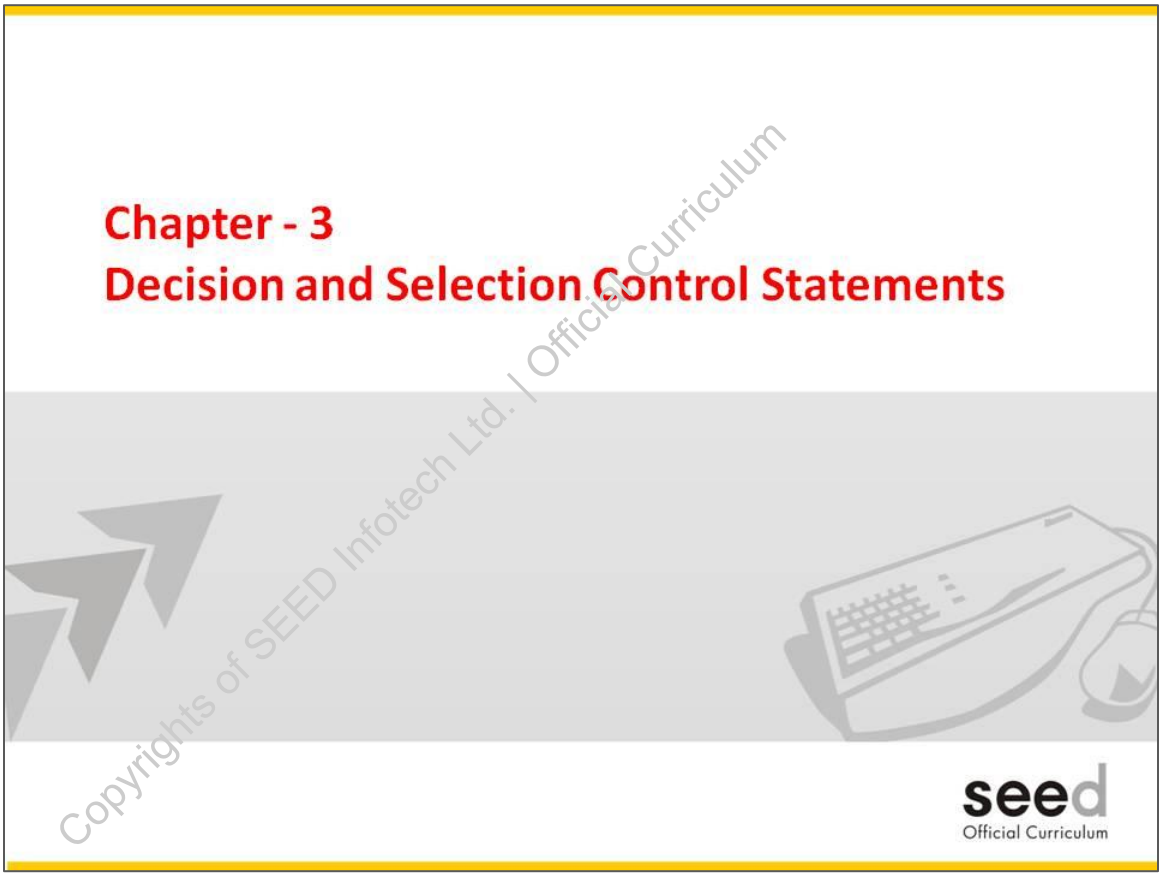
But if the result should not be truncated and some other factors do not allow us to define `x` as `float` then type casting can be used explicitly as demonstrated in the above code snippet.

`(float)x`, converts `x` into a floating point value before performing the division. Therefore, the result is floating point. This conversion temporarily changes `x` into the required type. The value of `x` is not changed permanently as seen from the `printf()` statement that follows.

Another example can be given as follows – function '`sqrt`', which takes double as argument will give absurd results if some other type of data is given.

```
int x;
double y;
y = sqrt(double(x));       //x is converted to type
double
```

Use explicit typecast wherever possible.



C provides various types of statements that control the flow of execution of a program. The '`if`' and '`if-else`' statements, the ternary operator to make decisions, and the `switch-case` statement to make selective decisions, all are covered in this chapter.

Objectives

At the end of this chapter you will be able to:

- List different types of statements in C.
- Use decision making statements like `if`, `if-else` and nested `if-else`.
- Use ternary operator to make decisions.
- Use switch statement to make selective decisions.

Control Flow

- A C program executes one command after another, from top to bottom.
- Control statements enable
  - altering the flow of the program.
  - repeating commands according to the different conditions in the program.
- One of the basic flow control statements is the `if` statement.
- The `if` statement enables us to skip commands or execute commands depending on a condition.

Normally, a program is executed from the first statement to the last. But the control statements allow the user to alter the flow of the program and to repeat some statements as per requirement.

The basic flow control statement used in C programs is the '`if`' statement. This statement can be used when certain action or a group of actions need to be taken depending on a condition.

'C' Control Statements

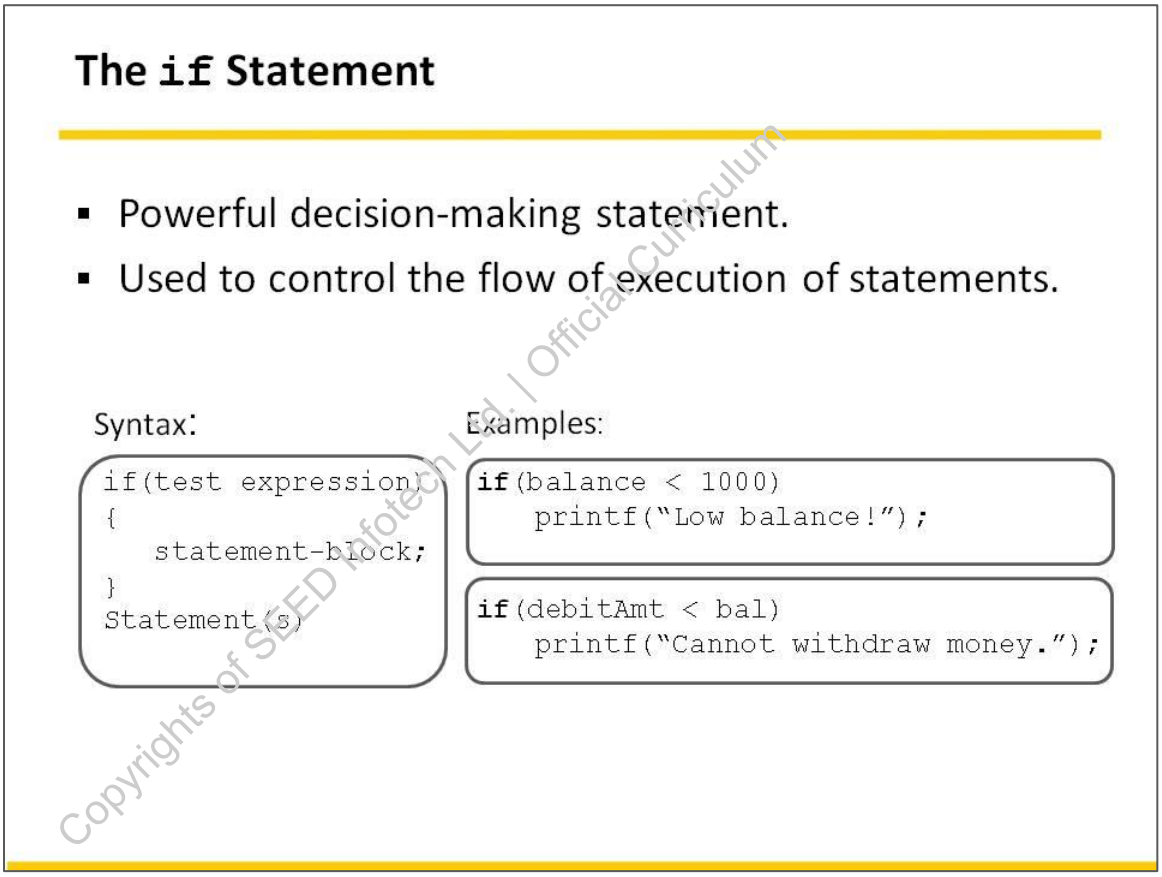
- Control the flow of program execution.
- Consist of tokens, expressions, and other statements.
- Control-flow statements are as follows:

Statement Type	Keyword
Decision and Selection Control	`if-else, switch-case,` conditional operator
Looping	`while, do-while, for`
Branching	`break, continue, return, goto`

The control statements in C, control the flow of program execution. They also alter the normal flow of program execution.

Control statements consist of tokens, expressions and other statements. There are three different types of control statements – decision making, looping and branching.

- The '`if`', '`if-else`' and '`switch-case`' are the keywords used in the decision making statements.
- Depending on whether a condition is true or not, the looping statement repeatedly executes a statement or a block of statements. The `while`, `do-while`, and `for` are the keywords used in executing the looping type of control flow statements.
- The '`break`', '`continue`', '`goto`' and '`label:`' and '`return`' are the keywords used for the branching type of statements.



This is a powerful decision making statement. The above code snippet demonstrates the use of `if` statement in its simplest form.

The 'if' keyword is followed by a test expression; it can be any logical expression. It is followed by a statement or a block of statements to be executed if the test expression evaluates to true.

1. In the first example, the 'if' statement checks whether the balance is less than 1000. If the condition evaluates to true, then "**Low balance!**" is printed.
2. In the second example, the 'if' statement checks whether the amount to be debited from an account is less than balance. If the condition evaluates to true, then the message "**Cannot withdraw money.**" is printed.

Both the above examples, execute a single statement if the condition evaluates to true. In C, a condition evaluating to zero indicates false value and any non-zero value is true.

The if-else Statement

- Extension of the simple `if` statement.

Syntax :

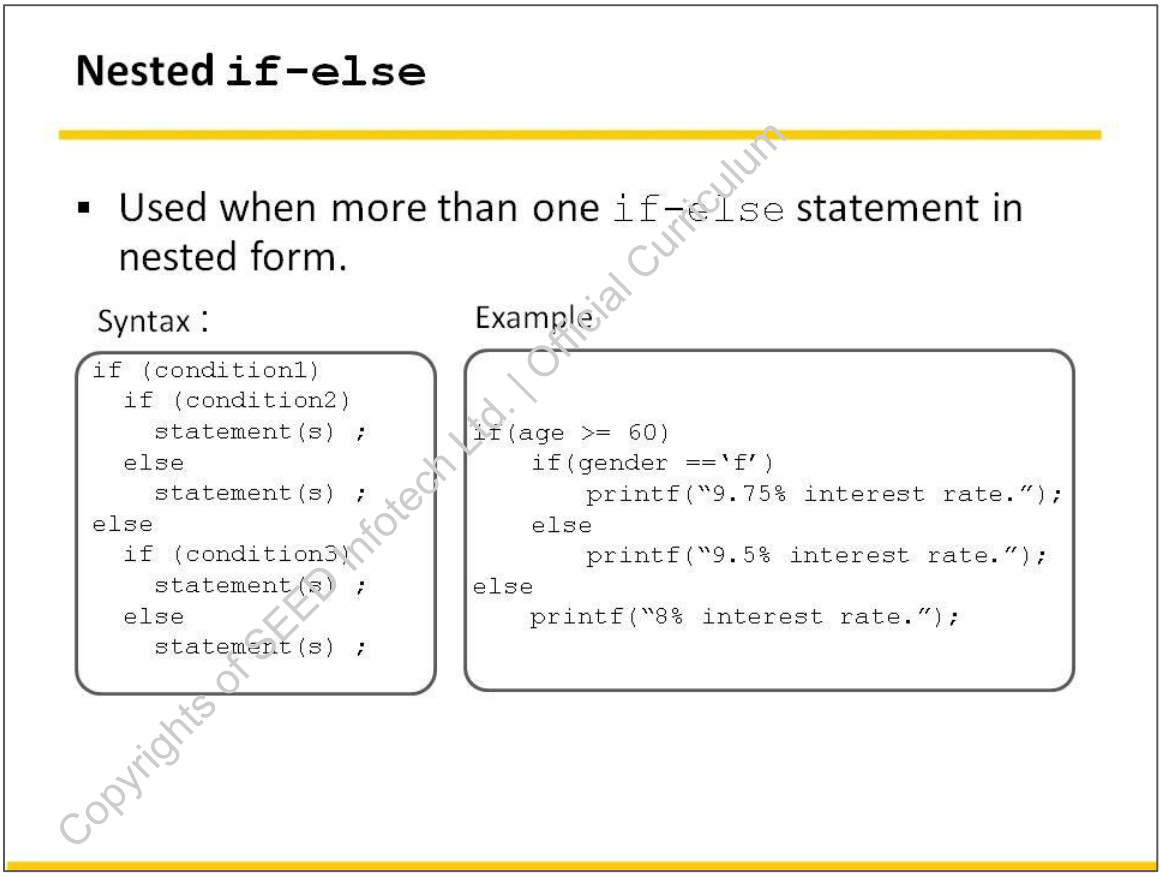
```
if (test-expression)
{
  statement(s) block
}
else
{
  statement(s) block
}
statement(s);
```

Example

```
if(balance < 1000)
{
  printf("Low balance!");
}
else
{
  printf("Balance is OK.");
}
```


The 'if-else' statement provides a way to take actions in both cases – whether the test expression evaluates to true or false. If the condition in the expression of the 'if' statement evaluates to false, then the statement(s) in the else block are executed.

The example demonstrates how the if-else statement can be used to check whether the balance is low or not. If the condition in the expression (balance < 1000) evaluates to true, it means that the balance is low. So the message "**Balance is low**." is printed. If the expression evaluates to false, that is, the balance is greater than 1000, then the balance is as required and the message "**Balance is OK.**" is printed.



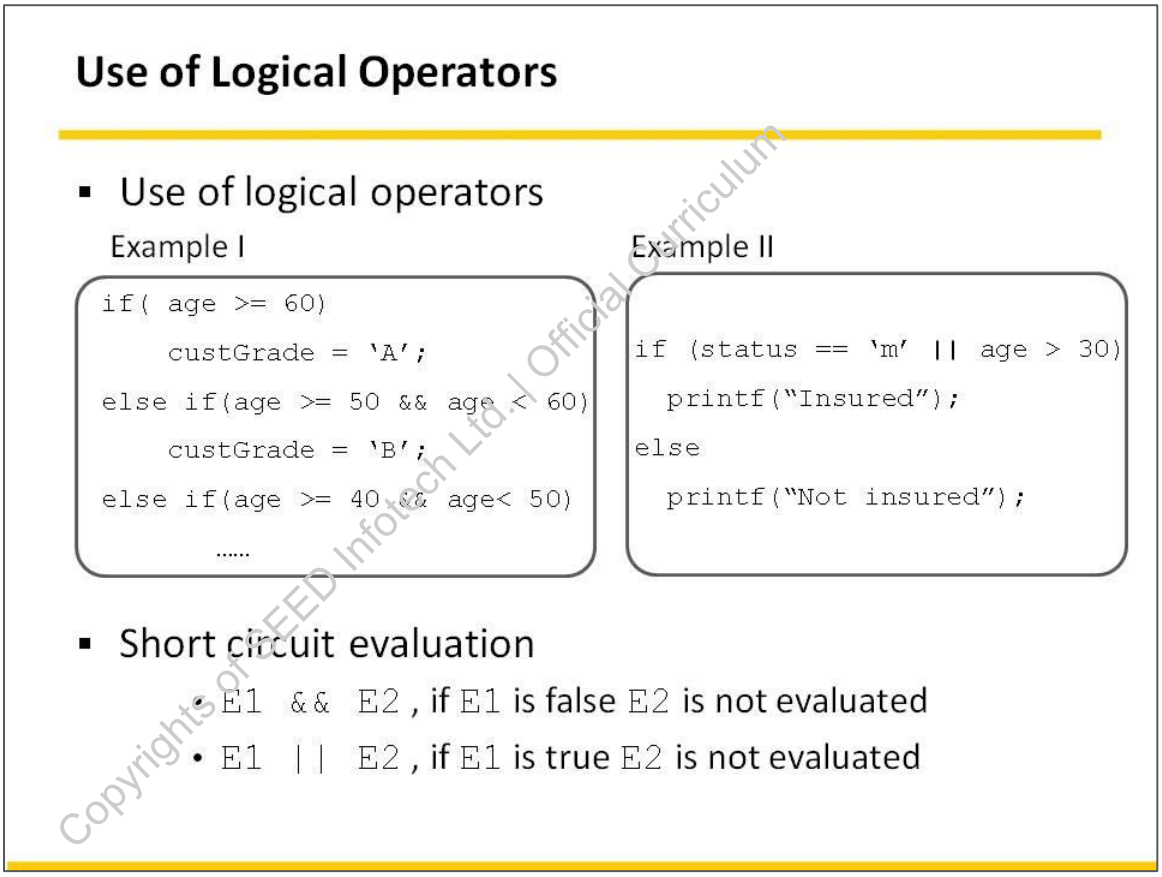
Sometimes, it may be required to check another condition after the first one has evaluated as true. In such situations, the nested `if-else` statement is used. One `if-else` statement can occur inside another `if-else` statement. The inner `if-else` statement is said to be nested in the outer `if-else` statement. There can be more than one nested `if-else` statements and the nesting can go up to any level.

One of the many forms of the nested `if-else` statements has been shown above. Here,

- If `condition1` is true, then `condition2` is checked. If this condition is true, then the statement(s) following the ‘`if`’ statement are executed. If the `condition2` evaluates to false, then the statements in the `else` block are executed.
- If `condition1` is not true, then the statements in the outer `else` block are executed. In the outer `else` block, there is another `if-else` statement. Here, if the `condition3` is true, then the statements following

`if` block are executed. If this condition is not true, then, the statement(s) following the `else` block are executed.

Solve plenty of code snippets to predict the output of the code.



The logical operators - `&&` (AND), `||` (OR) and `!` (NOT) can be used in the `if-else` statement to combine more than one relational expression.

1. Example I checks whether the marks secured by a student fall within a certain range by using the `&&` operator along with the relational operators (>= and <).
2. Example II shows how the || operator is implemented to check if either the status or the age satisfy the conditions given, using the relational operators (== and >).

While using the logical operators, compiler uses short circuit evaluation to optimize the code. Consider E1 and E2 to be two expressions in `if` statement separated by logical operators.

- For the condition, `if (E1 && E2)`, `E2` is not evaluated if the expression `E1` evaluates as false.
- For the condition, `if (E1 || E2)`, `E2` is not evaluated if the expression `E1` evaluates as true.

Classroom Quiz

1. Understand short circuit evaluation by solving many code snippets.
2. Find the output of the following statements:

i.
```
int x = 0, y = 5, z;
if(x && y)
     printf("z = %d", z);
else printf("\nNo output");
```

ii.
```
int x = 10, y = 0, z;
if(x || y)
       printf("z = %d", z);
else printf("\nNo output.");
```

Conditional Operator

- Also known as Ternary Operator (? :)

Syntax

```
Expression1 ? Expression2 : Expression3;
```

- Example : To find maximum of 2 numbers

```
max  = (num1 > num2) ? num1 : num2 ;
```

- This is equivalent to

```
if(num1 > num2)
    max = num1;
else
    max = num2;
```


The conditional or `ternary' or (`? :`) operator functions like `if-else` statement.

Syntax

```
expression1 ? expression2 : expression3
```

First, `expression1` is evaluated. If it is true then `expression2` is evaluated, else `expression3` is evaluated. For example, if in the following expression, `num1` is greater than `num2`, then `max` is assigned the value of `num1` else `max` is assigned the value of `num2`.

```
max = (num1 > num2) ? num1 : num2 ;
```

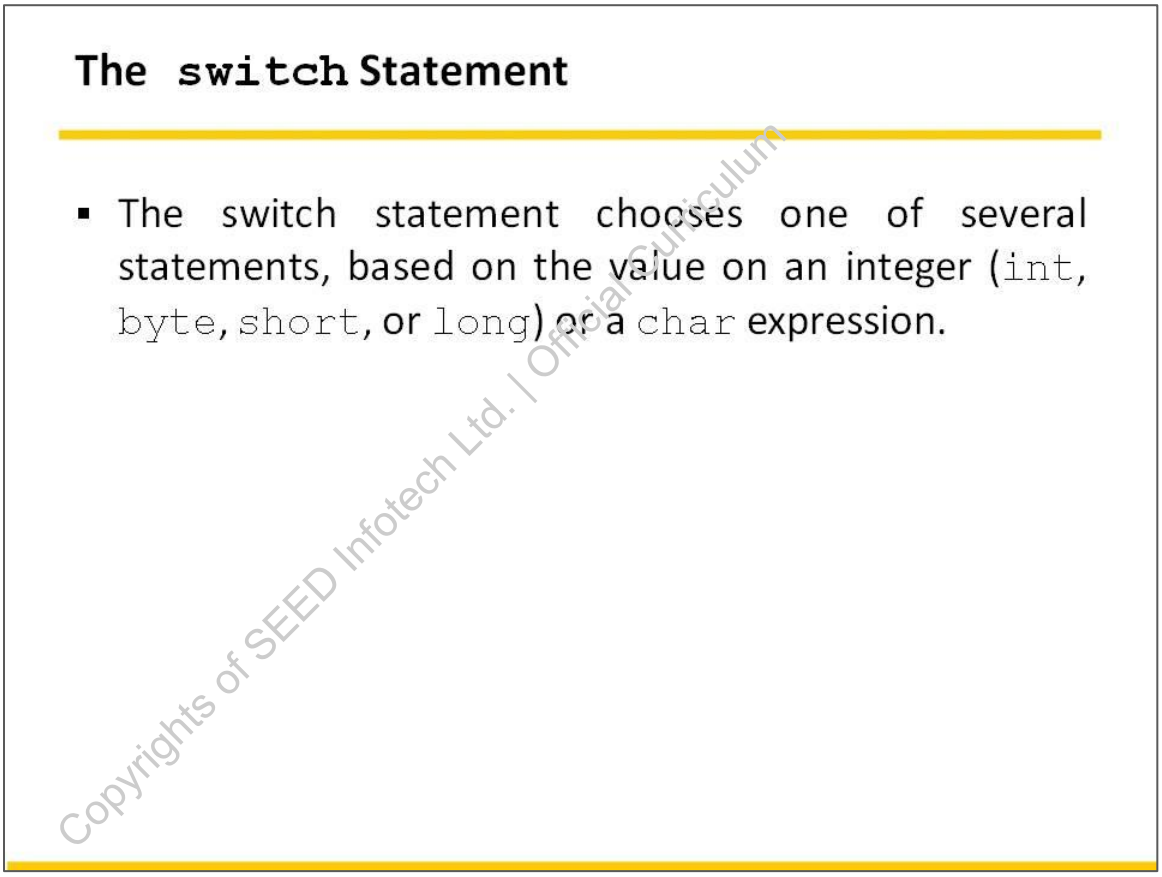
- Apart from this arithmetic statement, conditional operator can be used with `printf( )`. For example:

```
(j==1 ? printf ("Anyone") : printf ("Only one"));
```

- Conditional operator can be nested. For example:

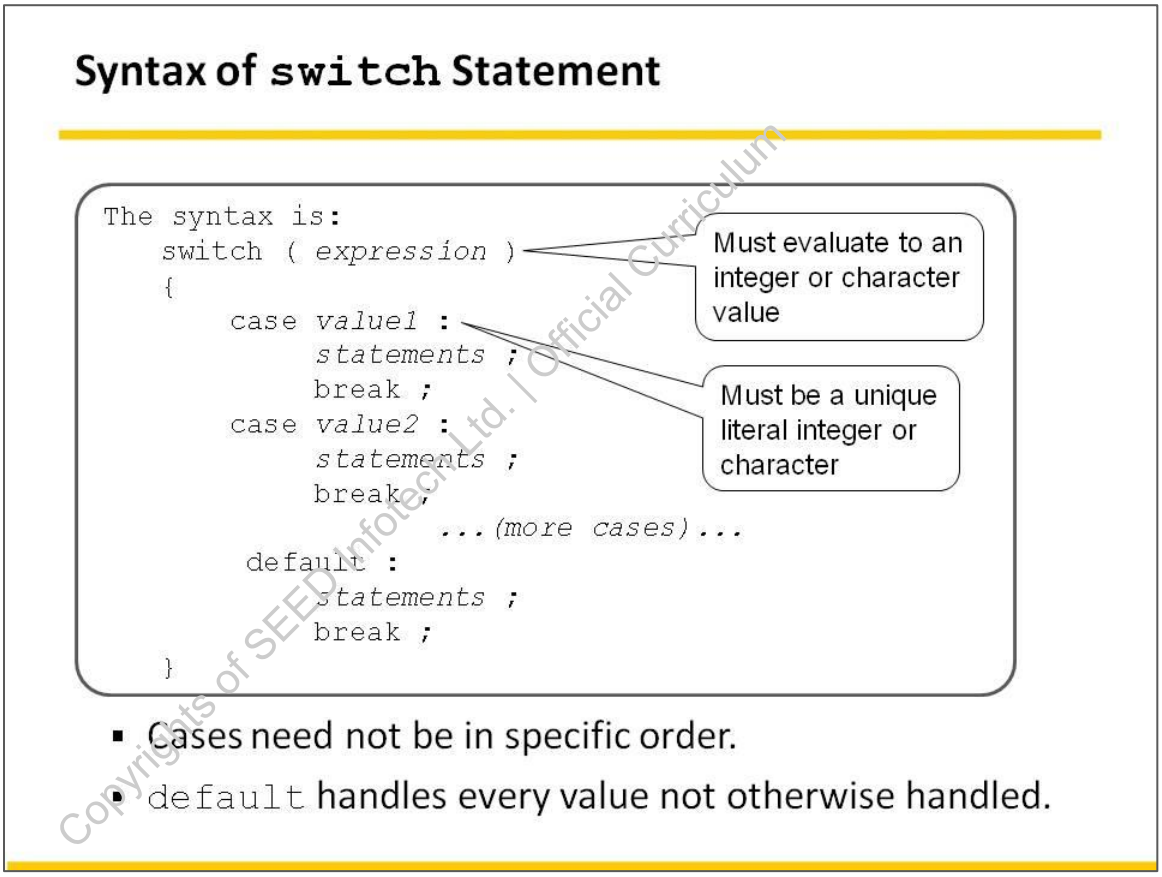
```
int big, x ,y ,z;
big = ( x > y ? (x > z ? x : z) : ( y > z ? y : z) );
```

The limitation of conditional operator is that only one statement can appear after '?' or after ':'.



The `switch-case` statement is used for selective decision. In the `switch` statement, the keyword `switch` is followed by an *expression* which must yield an integer or a character value. The statements to be executed are listed below the `switch`.

- The statement consists of one or more `case` statements followed by a colon (:), followed by a group of statements.
- Each value in the `case` statement must be a literal integer or character. Notice that colon (:) is used after each value and the statements are terminated by semicolons. These values must be unique; that is, in a switch block there cannot be duplicate cases.
- The last statement in every `case` should be `break` to prevent fall through.
- The `default` case handles every value not otherwise handled.



In the `switch-case` statement, the `break` statement is included as the last statement in the matching `case` statement block. If the `break` statement is not given, then the `switch` executes the `case` where a match is found and all the subsequent `cases` and the `default` as well. This is called the fall-through.

The following code snippet demonstrates the fall through in the absence of the `break` statement.

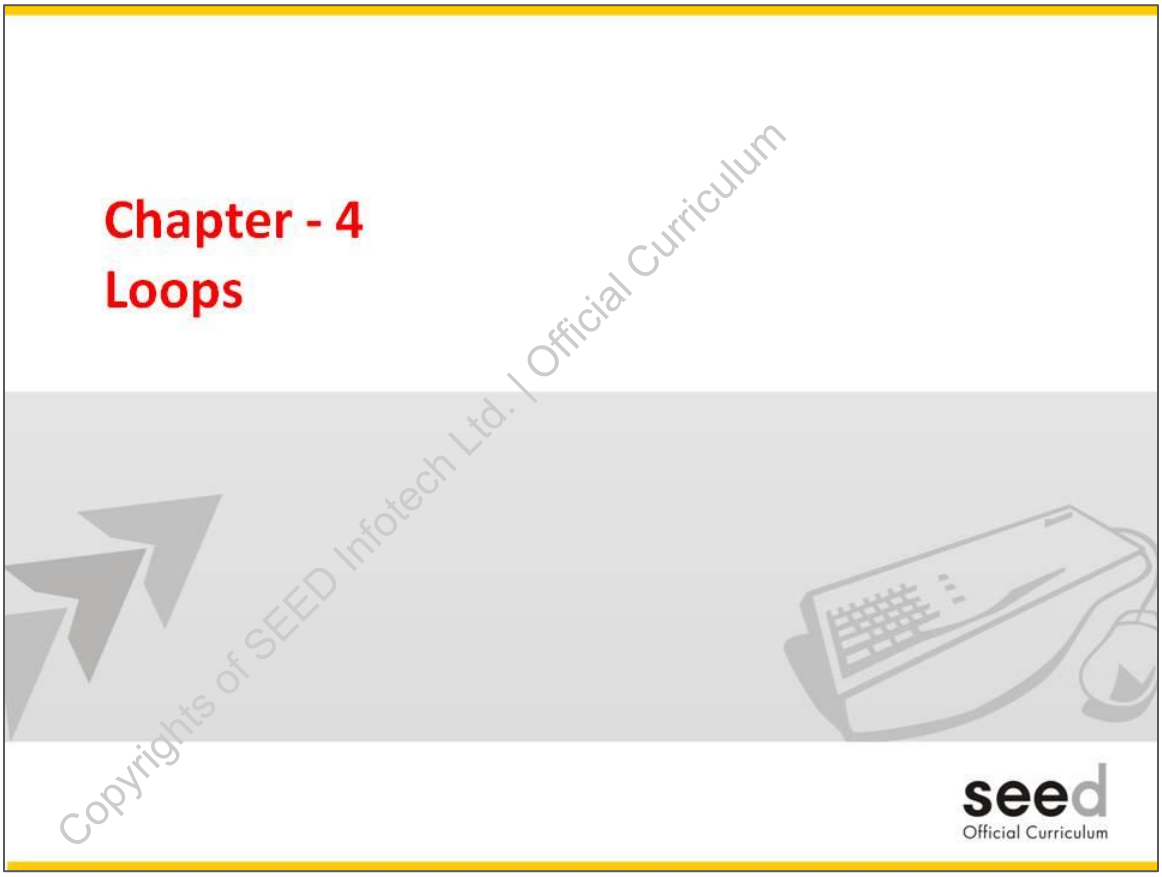
```
int i = 2;
switch (i)
{
  case 1:
      printf("\nIn case 1");
  case 2:
      printf("\nIn case 2");
    case 3:
```

```
      printf("\nIn case 3");
    default:
      printf("\nIn default.");
}
```

The output of the above snippet will be

```
In case 2
In case 3
In default
```

Drawback of switch-case is that it can be used only with an integer expression; it cannot be used on strings.



This chapter covers the working of different loops like the `for`, `while` and `do-while` as well as some keywords like `break`, `continue`, and the `exit()` function.

Objectives

At the end of this chapter you will be able to:

- Identify the need of iterations.
- List different types of loops in 'C'.
- Use '`for`' loop to implement iteration.
- Use '`while`' loop to implement iteration.
- Use '`do-while`' loop to implement iteration.
- Describe the execution of nested loops.
- Alter code execution using branching statements like `continue` and `break`, `exit()` function.
- State the working of comma operator.

Loops

- Loops are used when a task needs to be repeated a number of times.
  - For example
    - Printing numbers from 1 to 100
    - Printing even numbers from 1 to 50
    - Printing details of all bank customers
    - Sequentially searching a record in a file
- Loops can be categorized as
  - Pre - tested loop
    - `while` loop , `for` loop
  - Post - tested loop
    - `do-while` loop

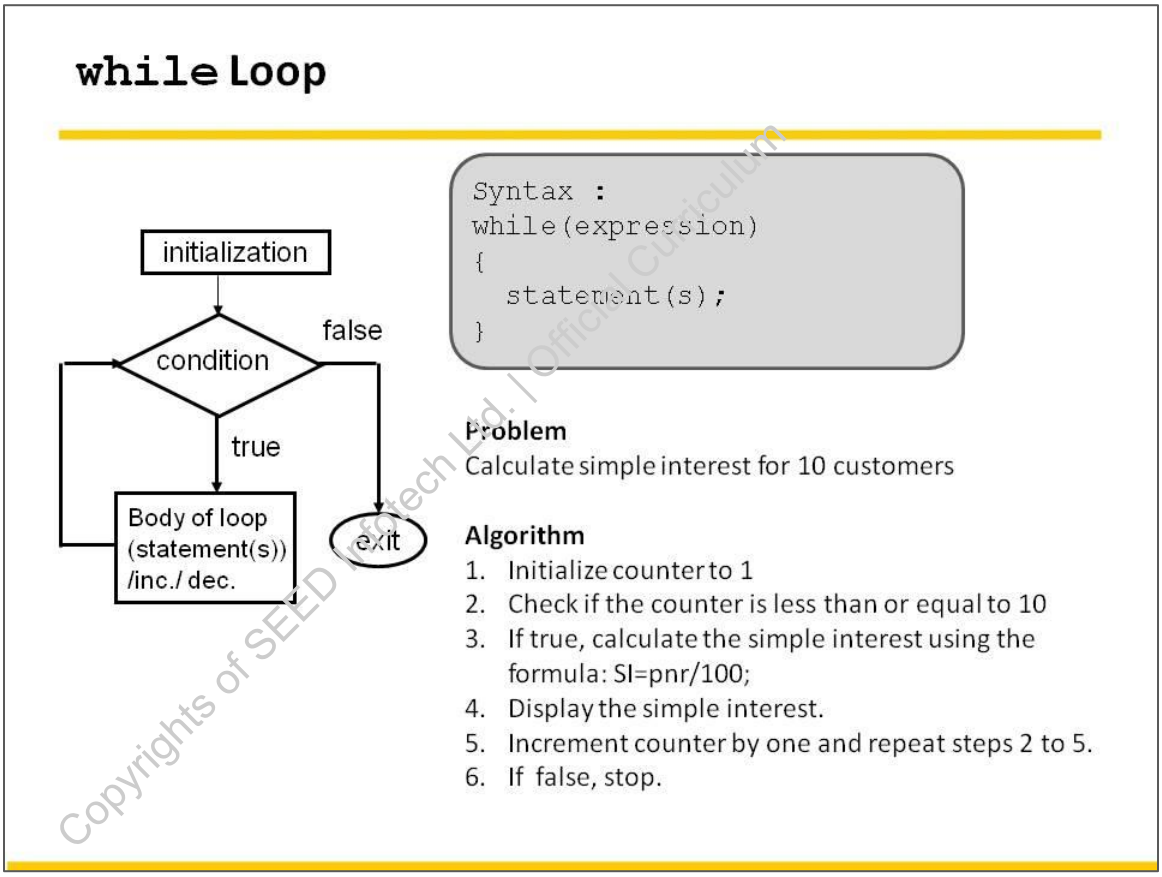
Using only sequential execution and conditional execution sets a limit to the range of problems that can be solved. Even in real life it is needed to perform actions again and again. It may also be needed to repeat an action for a specified number of times or until a certain condition is reached. For example,

- Reciting a poem again and again to learn it by-heart.
- If an item is purchased on instalments with period of one year; paying the instalment each month (total 12 times).

Loops are used to perform a certain action or task which needs to be repeated for specific number of times or until a certain condition is true. For example, printing the numbers from 1 to 100. The task is printing a number and this has to be repeated 100 times. Loops are the best solution to implement such type of tasks.

There are different types of loops or iterative control structures - `while` loop, `for` loop, and `do-while` loop. These loops can be further categorized as pre-tested or post-tested.

- A pre-tested loop is a loop where condition is tested before the loop starts executing. It is also called as entry control loop. For example, while and for loops.
- A post-tested loop is a loop where the loop is executed at least once before the loop is executed. It is also called as exit control loop. For example, 'do-while' loop



It is a pre - tested loop. Its general form is:

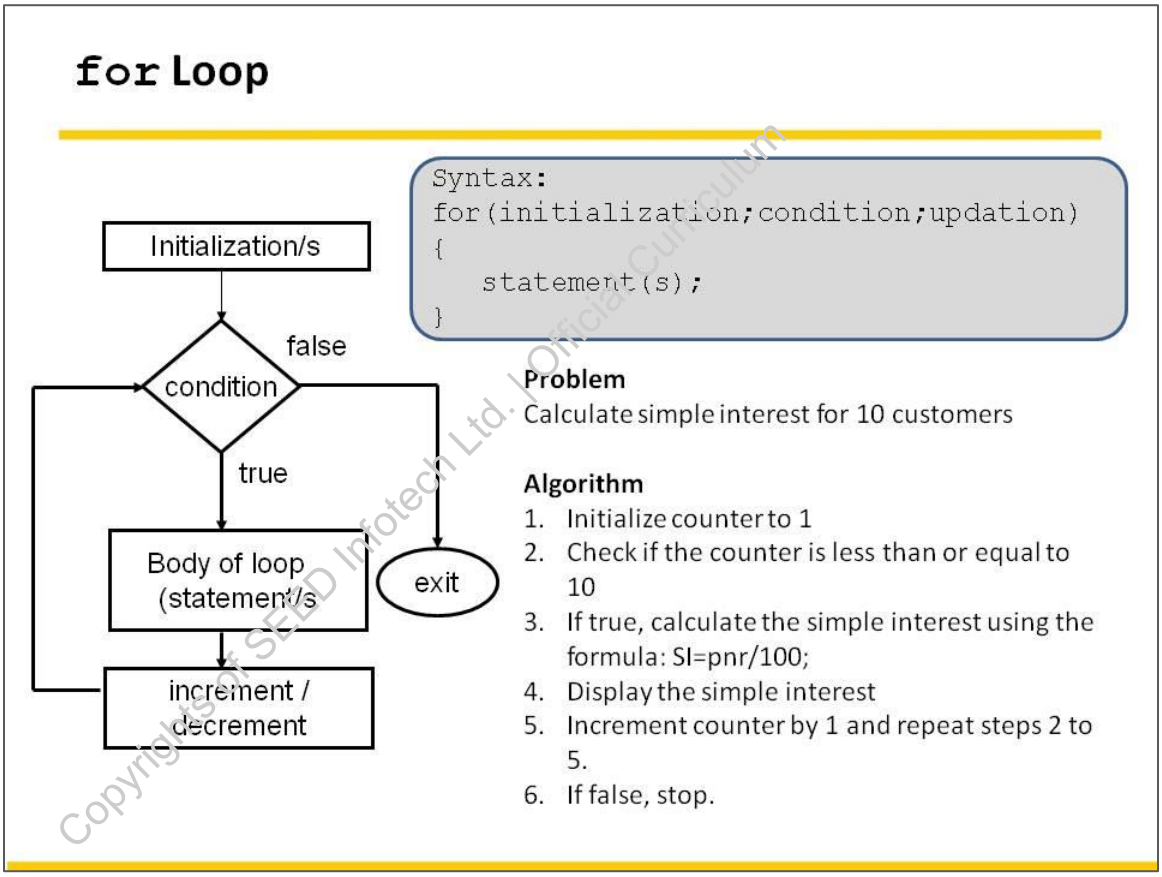
```
while (expression)
{
  statement(s);
}
```

The expression following `while` is usually a logical expression, which results in true (non-zero) or false (0). The statements following `while` form the body of the `while` loop. There can either be a single statement or compound statements.

The statements are executed as long as the logical expression is true. The loop terminates when the condition becomes false. There should be a statement in the body of the `while` loop that will eventually cause the expression to become false. If there is no terminating condition, then it becomes an infinite loop causing stack overflow.

Suppose, simple interest is to be calculated for 10 customers; first the task needs to be identified. The task is calculating simple interest. This task needs to be repeated 10 times (or for whatever number of times required). The code snippet mentioned above demonstrates this. An important point to note is that the loop counter variable is incremented within the body of the loop to ensure that the conditional expression in `while` statement will evaluate to false at some point. The following code snippet demonstrates this.

```
/*code to accept information from user*/
  . . .
  cnt = 1;              //initialization
  while( cnt <= 10 )  // condition
  {
  simpInt = (p*n*r)/100;
printf("%f\t", simpInt);
     cnt++;       // increment the counter
  }
  . . .
```



Like `while` loop, `for` is also a pre-tested loop. The general form of the `for` loop is

```
for(Expression1; Expression2; Expression3)
{
  statement;
}
```

It contains three expressions.

1. `Expression1` is used to initialize some parameter/s, called the loop counter/s that controls the looping action. It is an assignment expression.
2. `Expression2` is the condition that must result to true to continue execution of `for` loop. It is a logical expression.
3. `Expression3` is used to increment or decrement value of the counter. It is a unary or assignment expression.

This loop is relatively simple to use when the number of times it should iterate is known.

```
/*Calculate simple interest for 10 customers*/
. . .
for(cnt=1;cnt<=10; cnt++) //intialization, condition
and updation
{
  simpInt = (p*n*r)/100;
  printf("%f\t", simpInt);
}
. . .
```


In the code snippet mentioned above, simple interest is calculated for 10 customers. The body of the loop encloses the task of calculating the simple interest. The condition checks the upper limit of the range specified, i.e. 10. The other two parts of the loop initialize the loop counter variable '`cnt`' and increment it so that the logical expression evaluates to false after specified number of times.. The sum is printed outside the loop.

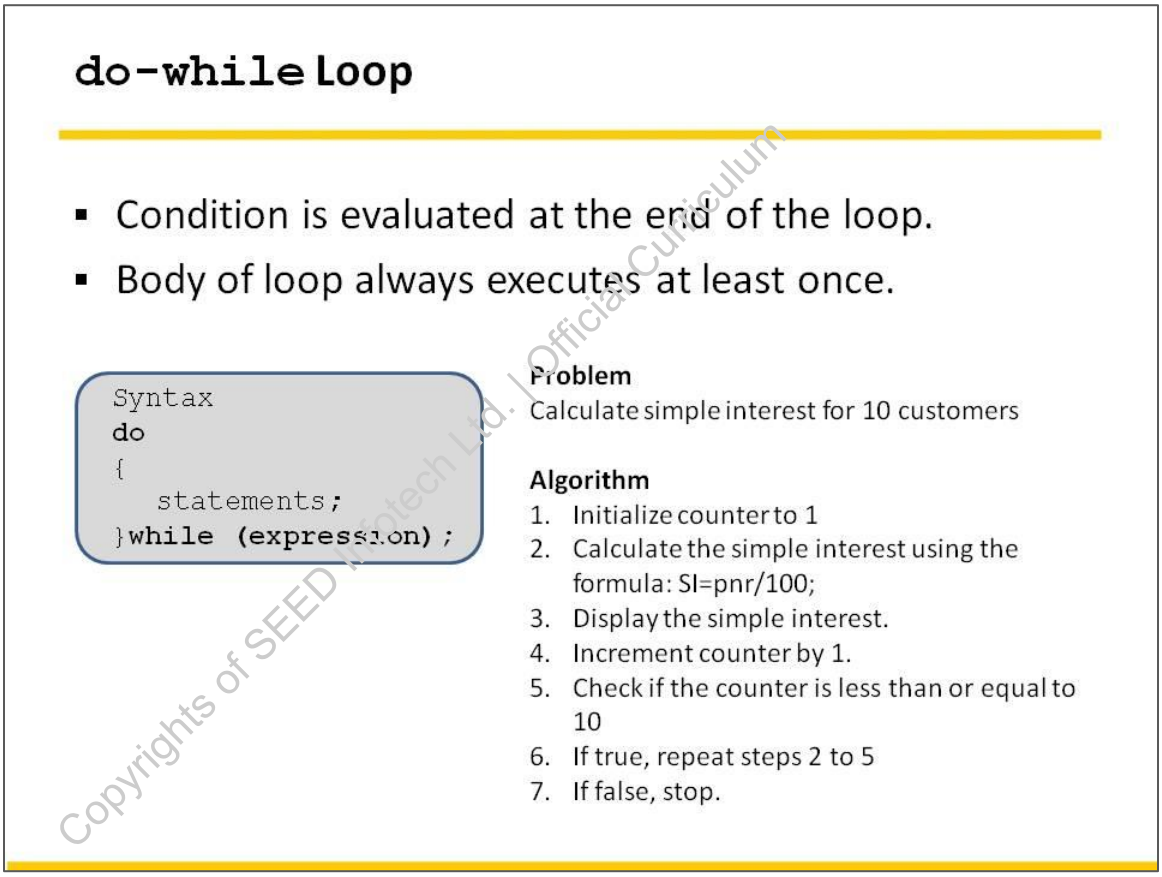
- Expression1 is `(cnt = 0)` (Initialization)
- Expression2 is `(cnt <= 10)` (Condition)
- Expression3 is `(cnt++)` (Updation – in this case increment)

The loop counter `cnt` is initialized to 1. The condition expression (`cnt<=10`) checks whether the counter is less than or equal to 10. If it is true, then simple interest is calculated for 1 customer at a time. The counter `cnt` is then incremented by 1 using `cnt++`. The other alternatives are

`cnt = cnt + 1` and `cnt += 1`.

The condition `cnt <= 10` is checked again. If it is true, the simple interest for next customer is calculated. This repeats until condition `cnt<=10`, results in false. Here `cnt` is the index. Index can be incremented or decremented by any value (1 is not compulsory). If it is decremented, initialization of index should be done accordingly.

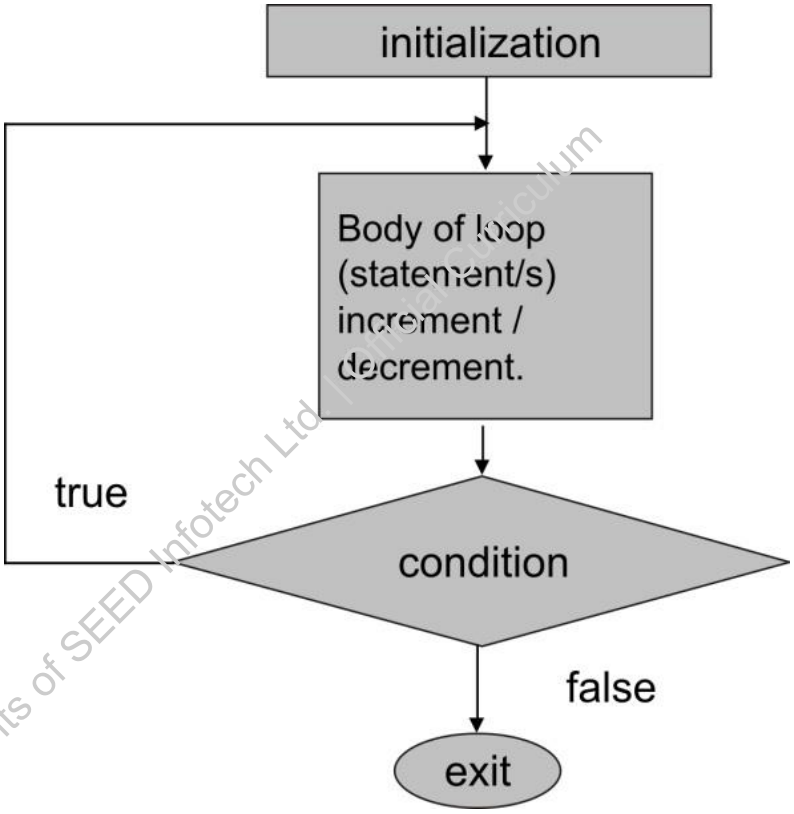
The same program can be written using the `while` loop as well.



`do-while` is a form of `while` loop but unlike `while` loop, it is a post-tested loop. The general form is

```
do
{
  statements;
} while (expression);
```

`do-while` loop is terminated by a semicolon (**;**). This loop is used in places where statements are to be executed at least once, irrespective of the result of the conditional expression.



The following code snippet demonstrates the use of `do- while` loop. This code will calculate the simple interest at least once. It will execute subsequently only if the logical condition evaluates to true.

```
. . .
cnt = 1;  //intialization
do
{
  simpInt = (p*n*r)/100;
  printf("%f\t", simpInt);
  cnt++;  //increment
}while(cnt<=10);  //condition
. . .
```

fflush

The reason for using the function `fflush` is that when `scanf` is used, data accepted from user, along with the 'enter' key goes into the keyboard buffer. `scanf` assigns data to the given variable but keeps the 'enter' key in the buffer. Now when the user is prompted for Y/N, this 'enter' key is provided as a response since `getche` requires only one character.

To avoid this, `fflush` is used. It flushes out all the remains in the input buffer and accepts a fresh input from the user. The parameter passed to `fflush` is the buffer to be flushed out. Here `stdin` is the buffer related with standard input device – the keyboard.

Think of some examples where a particular loop will be best suited.

Nested Loop

- A loop can be a part of another loop.
  - Inner loop is said to be nested in the outer loop.
- Inner loop executes completely for each value of outer loop.
- Nesting of `while` and `for` loops is similar to decision control statements (i.e. `if-else` statements)

Loops can be nested inside one another in the same way as the decision control statements are nested. The inner loop is said to be nested inside the outer one. Different types of loops can be nested within each other as per requirement. Looping can go up to any level.

```
. . .
int i,j;
for(i= 2; i <= 7; i++)          // outer loop
 {
   for(j=1;j <= 10;j++)         // inner loop
    {
       printf("%d * %d = %d", i, j , i * j);
    }
  printf("\n");
```

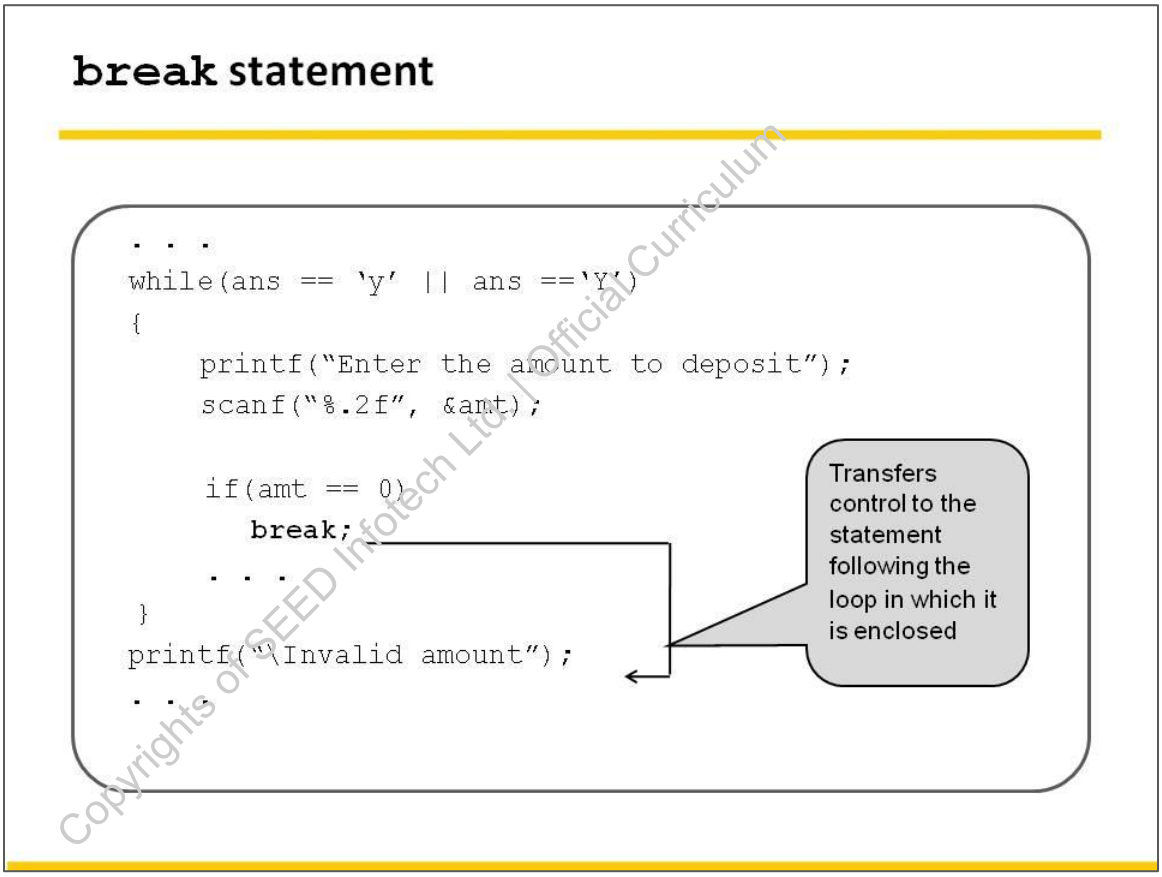
```
}

. . .
```

Consider an example to print the multiplication tables from 2 to 7. First, the task involved is identified and then how many times it is to be repeated. The task is printing multiplication table of a number. The lower limit is 2 and upper limit 7. First, the inner loop iterates 10 times to print the multiplication table for the value `i = 2`. Then the value of `'i'` is incremented. Again the inner loop iterates 10 times to print the multiplication table of 3. This continues till the value of `i` becomes `7`. When the value of `i` becomes `8`, the outer loop terminates.

This indicates that inner loop executes completely for each value of the outer loop.

It is necessary that one loop should be completely embedded within another loop. There should be no loop overlapping.

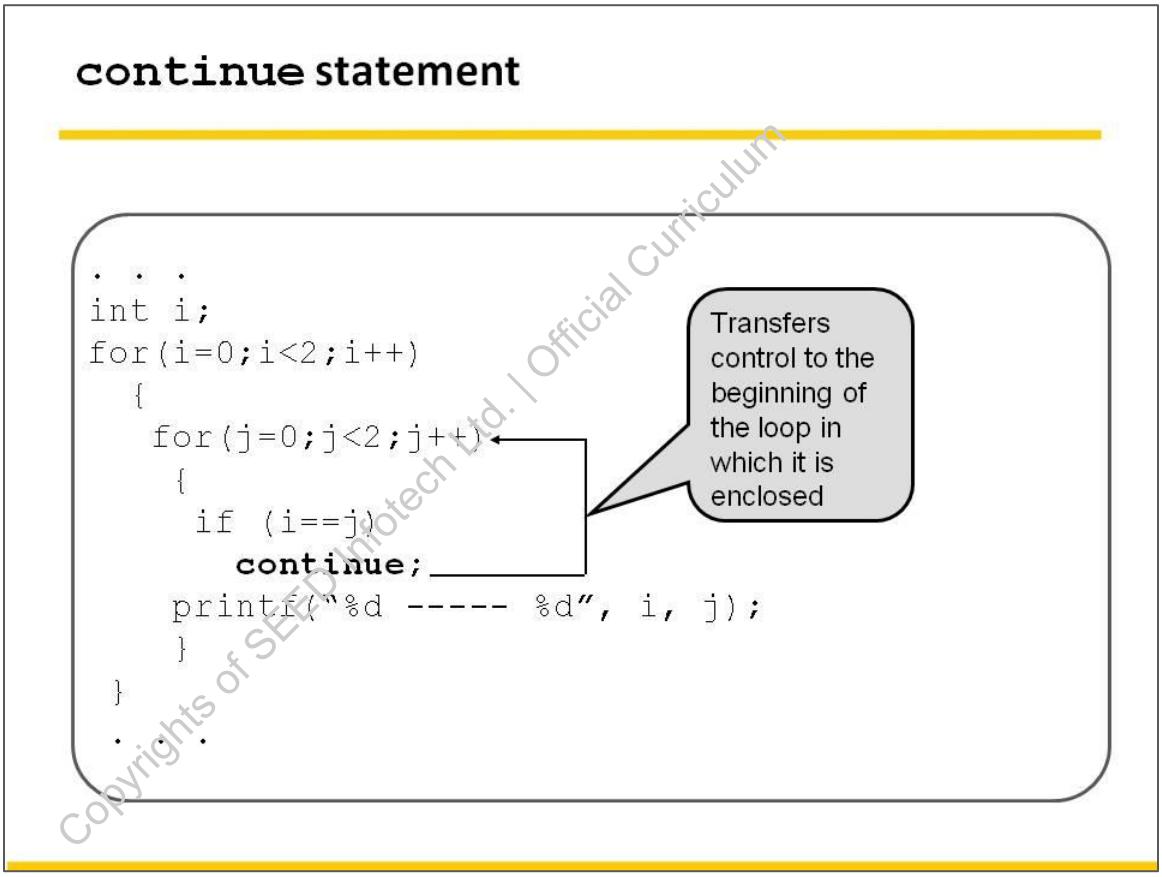
Write a program to print multiplication tables from 2 to 10.



C has some jump statements, which shift the program flow to another part of the program. Normally, after the body of a loop has been entered, all the statements in the body are executed before doing the next loop test. The `break` and `continue` statements are used to skip part of a loop or even terminate it before the predetermined condition, depending on tests made in the body of the loop.

It can also be used to handle errors or some exceptional condition. The keyword `break` is used for this purpose. If the `break` statement is inside nested loops, it brings the control out of the loop containing it. The statements following the loop are executed.

In the code snippet mentioned above, the loop is terminated when value of `amt` is entered as zero. Otherwise its value is further processed in the loop.



This statement can also be used only in the three loop forms - `while`, `for` and `do-while`. When encountered, it causes the rest of the iteration to be skipped and the next iteration starts.

If this statement is inside nested loop, it affects only the loop containing it.

The code snippet above demonstrates the implementation of `continue`. Its output is

```
0-----1
1-----0
```

- It is evident from the output that, for the value `i=0` and `j=0`, the condition in the inner loop becomes true. Further statements in the inner loop are not executed and the value of `j` is incremented.
- For dissimilar values of `i` and `j`, the `printf()` statement is executed.

`exit()` function

This function is used to terminate the program. Before terminating, the exit function closes all open files and writes all buffered output that is waiting to be sent to the output device.

Status of the program is sent with exit. Typically, a value of 0 indicates normal exit and non-zero indicates abnormal exit. For example,

```
Example I
if( den == 0)
  exit(1);                    //abnormal exit
Example II
printf("Do you want to continue? ");
scanf("%c", &choice);
if(choice == 'n')
  exit(0);                  //normal exit
```


`goto` statement

Control can be transferred to some other part of the program by using this statement. The general form is:

```
goto label;
```

where, `label` is an identifier used to label the target statement to which control will be transferred. Control can be transferred to any statement within the same function where `goto` is written. The target statement will look like this:

```
label: statement;
```

The label must be followed by a colon and must precede the target statement. In a program no two labels must be same. `goto` statements can be used to exit from deeply nested loops. That is, if it is required to exit from two loops (from inner loop which is nested in another loop) at a time, `goto` can be used, since `break` can exit from only one loop at a time. Or `if` and `break` statements can be used in both the loops.

Using `goto label`

```
 for(…) {
   for(…) {
    ..
    if(disaster)
       goto error;
   }
}
error :  printf("\nClean up the mess");
```

In 'C', `goto` statements should be avoided as far as possible. 'C' being a structured language, programs should be written in an orderly manner instead of jumping to places at random.

Using two breaks

```
for(…)
{
   for(…)
   {
    ..
    if(calamity)
       break; //breaks from the inner loop
   }
   if(disaster)
   break;     //breaks from the outer loop
}
```

Comma Operator

- Comma operator, with the lowest precedence is most often used in the `for` statement.
- It expands the flexibility of the `for` loop, by allowing more than one initialization (`expr1, expr2`) or more than one update (`expr3 and expr4`). For example,

```
. . .
for( expr1, expr2; condition; expr3, expr4)
. . .);
```

- Generally, if we have: `expr1, expr2` the following happens: `expr1` is evaluated (if it contains an assignment, it is executed). Then the value of `expr1` is discarded, and `expr2` is evaluated. This will be the value of the whole expression.

The comma operator is the lowest precedence operator and can be used in `for` statement.

1. It can be used when two or more related indices are to be initialized at the same time and/or if more than one variable is to be incremented or decremented.

```
for(expression1, expression2 ; condition;
expression3)
```

   Here, `expression1` and `expression2` are two expressions separated by comma operator. They will initialize two separate variables.

2. Here, `expression2` and `expression3` are separated by comma operator

```
for(expression1; condition; expression2, expression3)
```

   They are used to alter two different variables used simultaneously within the loop.

3. Only one expression is allowed in the test expression; multiple conditions can be combined using any of the logical operators.

```
for(j = 0, k = 3; j < k ; j = j + 2, k++)
{
   . . .
}
```

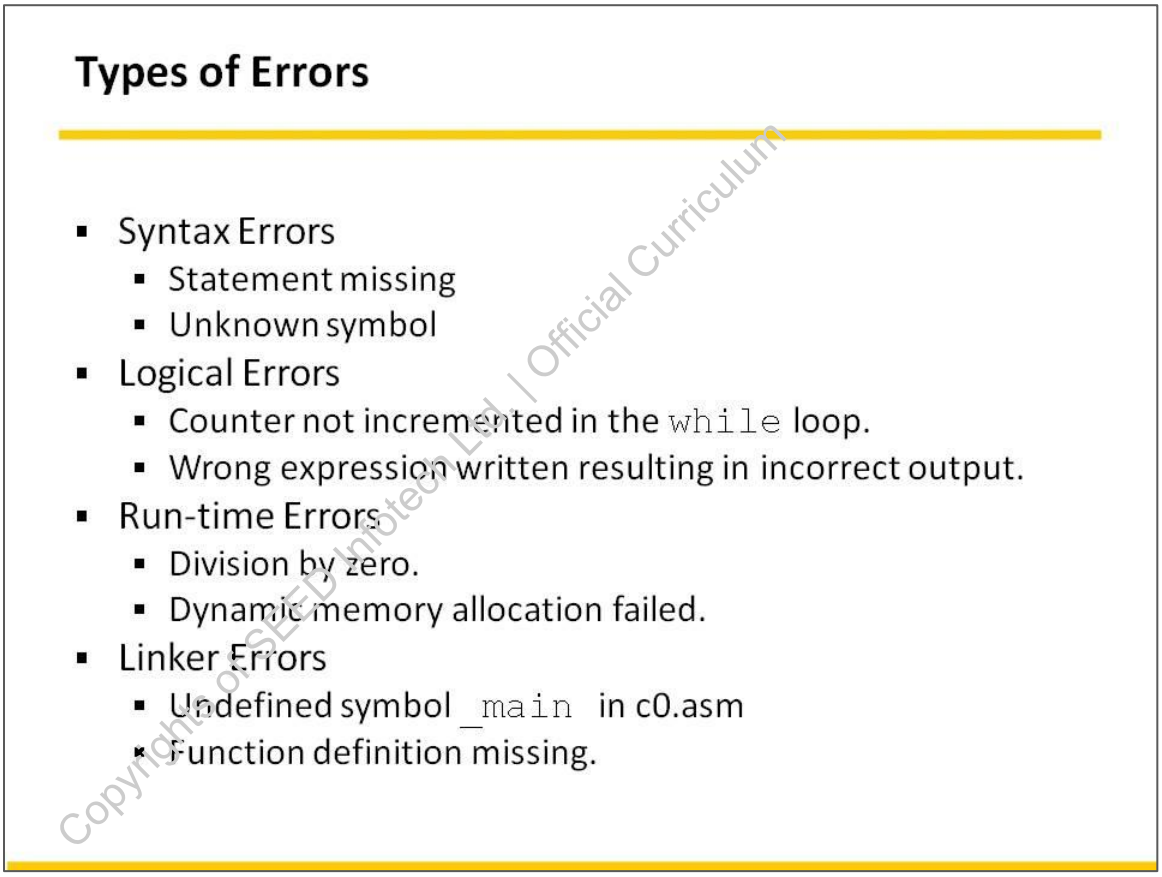

Expression(s) that can be omitted in the `for` loop

- 1st and 3rd expressions may be omitted, if other statements are provided for initializing the index and/or altering the index.
- If the second expression is omitted, it will be assumed to have value of 1 and loop will continue indefinitely, unless terminated by `break` or `return` statement.
- Any of the three expressions can be omitted, but the semicolons must remain.

```
j = 0;
for(   ;j   <= 10; j++)
{

}
```

```
for(j = 0; j<= 10; )
{
   j++;
}
```

```
for( ; ; )
{

/*infinite

   loop */
}
```



Writing program code and errors go hand in hand. They can occur as a result of incorrect syntax, incorrect logic or can occur at run time.

There are different types of errors:

Syntax Errors

Syntax errors occur when the syntax of the language is not followed correctly. They are detected at the time of compilation. For example, in C, it is necessary to terminate every statement with a semicolon. If the programmer forgets to add it at the end of the statement, compiler reports it as an error. Using a variable that is not declared is another example where the compiler throws an error.

This type of errors can be corrected by the programmer by following the rules and syntax of the language.

Logical Errors

Logical errors occur when there is something wrong with the logic used by the programmer. For example, when using the `while` loop, the programmer has to

give a terminating condition. But if the counter which leads to the terminating condition is not incremented / decremented, the loop will execute infinitely.

This type of errors are not detected by the compiler but can be corrected by the programmer using debug options or giving dry run to the program.

Run-time Errors

Run-time errors occur at run-time. For example, when there is division to be performed and the denominator entered by the user is zero, it will lead to a run-time error.

This type of errors can also be controlled by predicting the error and giving appropriate condition in the code.

Linker Errors

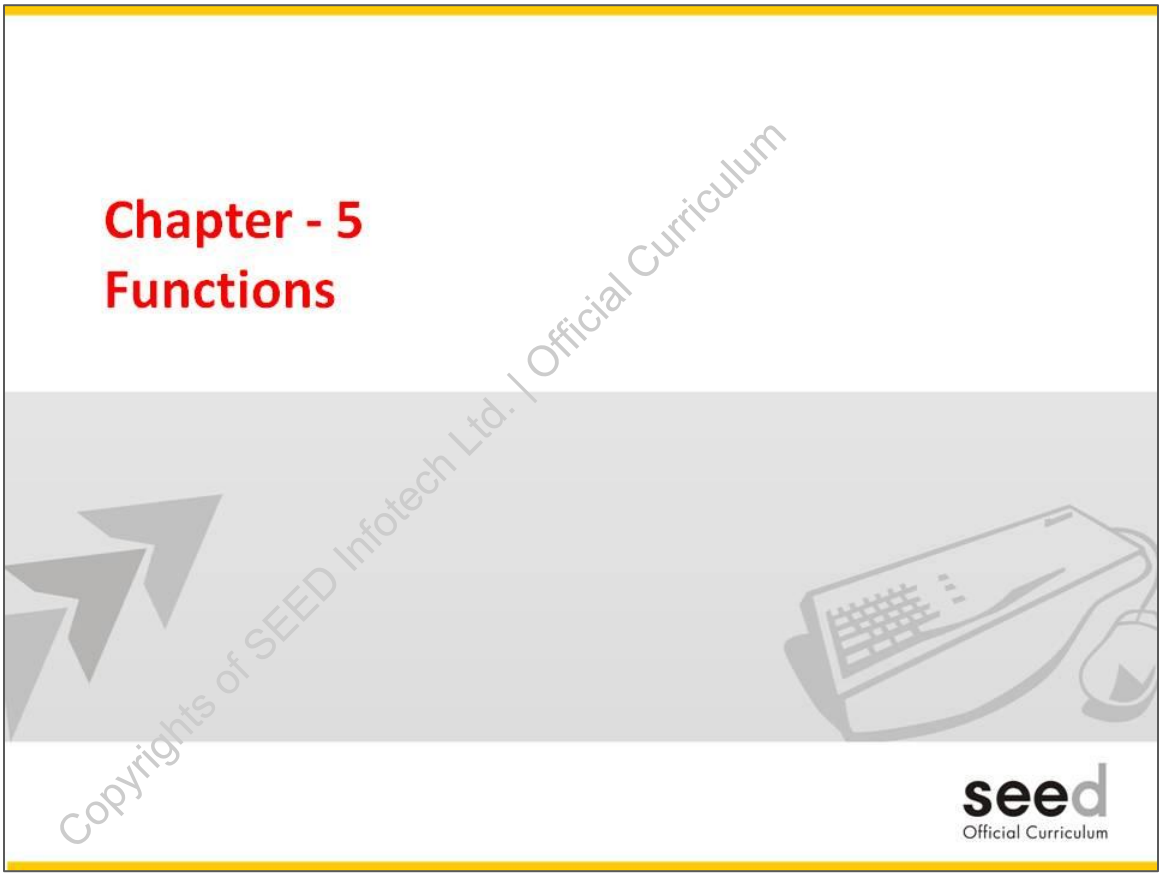
Linker errors are detected when the program is being linked. For example, giving a call to function which has not been defined or linking a file in which the `main()` function is missing.

These errors can be corrected by the programmer.

1. List different errors that you have come across during your lab sessions.
2. What will be the output of:

```
int y=1;
while(y<5)
{
    printf(" %d", y);
}
```

Always move loop invariant outside the loop.

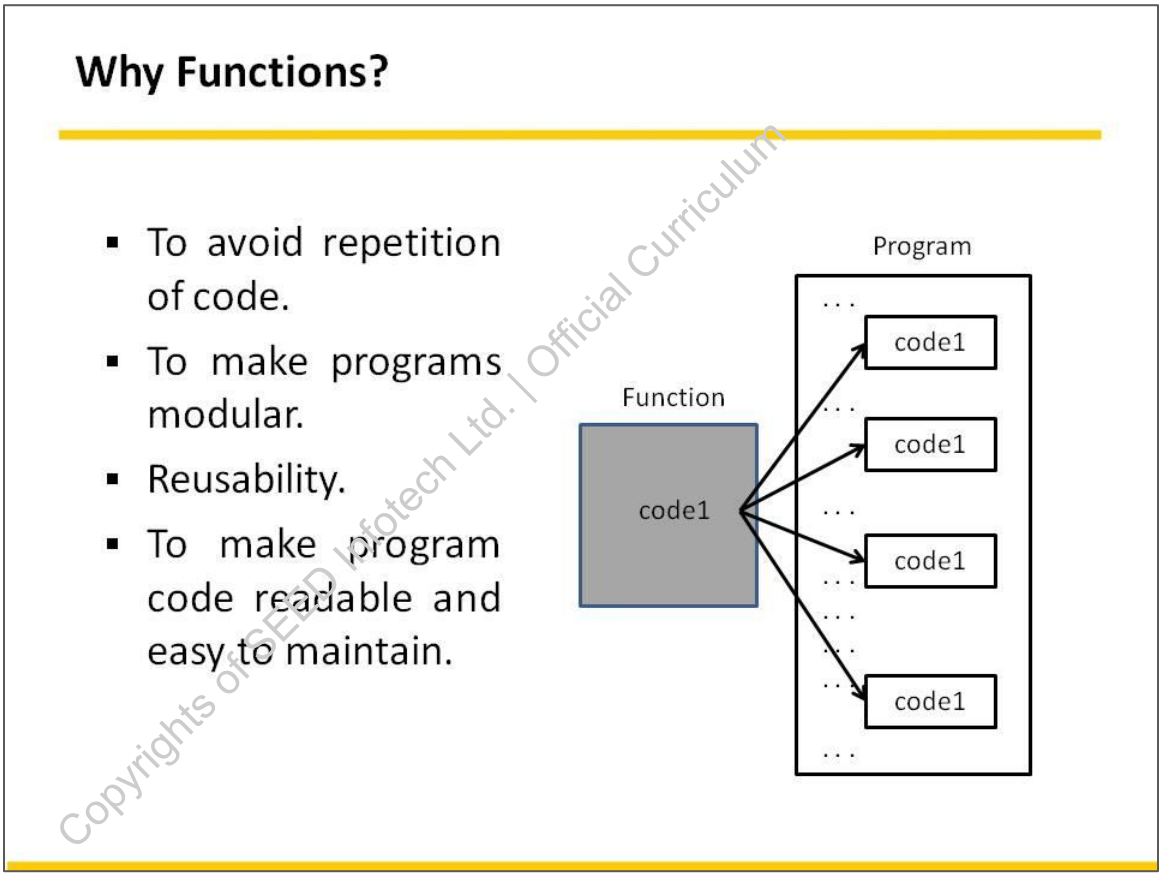


In this chapter, various aspects of a function have been covered: what a function is, how it is declared, how to define functions and call them. The need and use of pointers, and recursive functions is also discussed.

Objectives

At the end of this chapter you will be able to:

- Identify the need of function.
- State the rules of function declaration.
- List the types of functions.
- Pass arguments to a function – by address and by value.
- Define a pointer variable.
- Use a pointer to access the original value of a variable from a function.
- Use return statement
- Construct a program to define a recursive function.



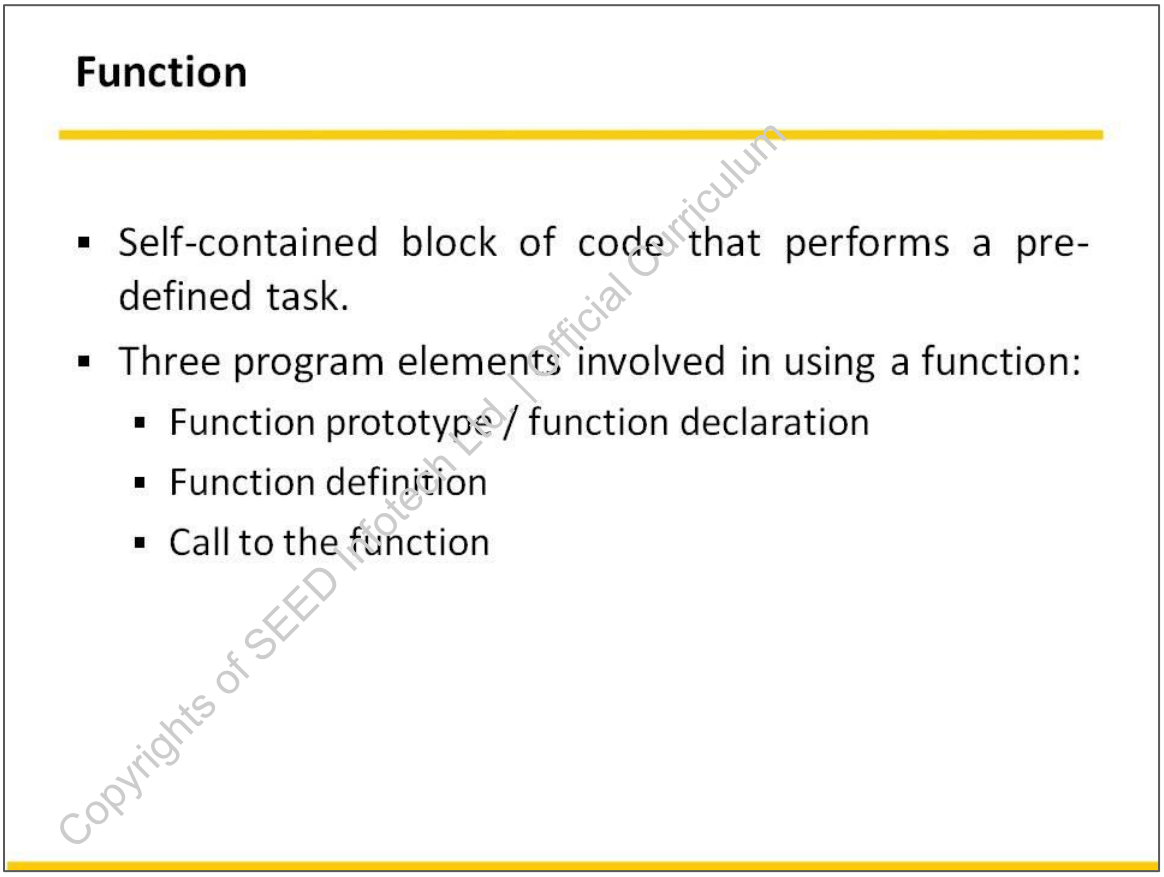
There are several tasks or operations that are required by a program to be performed more than once. So, the block of code that performs the task has to be repeated in the program as many number of times as required. This makes the program very lengthy.

The program can be shortened by writing this block of code only once and using is everywhere it is required and as many number of times as required. This avoids repetition of code, makes it reusable and at the same time the program becomes modular. The program becomes short and its readability is increased. In the future, if the requirements change, it is easy to make changes only in that block of code where it is required. The rest of the code can be left as it is. This makes it easy to maintain the program.

For example, to perform console input and output functionality, standard library functions like `printf()`, `scanf()`, `gets()`, `puts()`, etc., are used. The programmer need not worry about the programming logic required to implement

any of the built-in functions. Instead he/she can concentrate on the program's overall design.

Similarly, in a program there may be certain tasks that are frequently required but are not provided by the built-in functions such as accepting values from the user, displaying them, sorting the data, finding average, generating a report, reading from the file, and so on. These tasks can be written in the form of user-defined functions.



A function is a self-contained block of code that carries out some specific, well-defined task. Every 'C' program consists of one or more functions. One of these functions is `main()`. Execution of the program begins with `main()`. Other functions are called from `main`. These functions can also call other functions. A function acts like a black-box, which receives a pre-defined form of input and generates the desired value as an output.

There are three program elements involved in using a function.

1. Function prototype / function declaration, which indicates that a function with specified name and number of arguments and a particular return value is used in the program.
2. The function definition or function implementation that contains the programming logic or actual code to perform the specific task.
3. Call to the function to perform the task.

The syntax for writing function definition is as follows:

```
return-type functionName (arguments declaration)
{
  declaration and statements
}
```

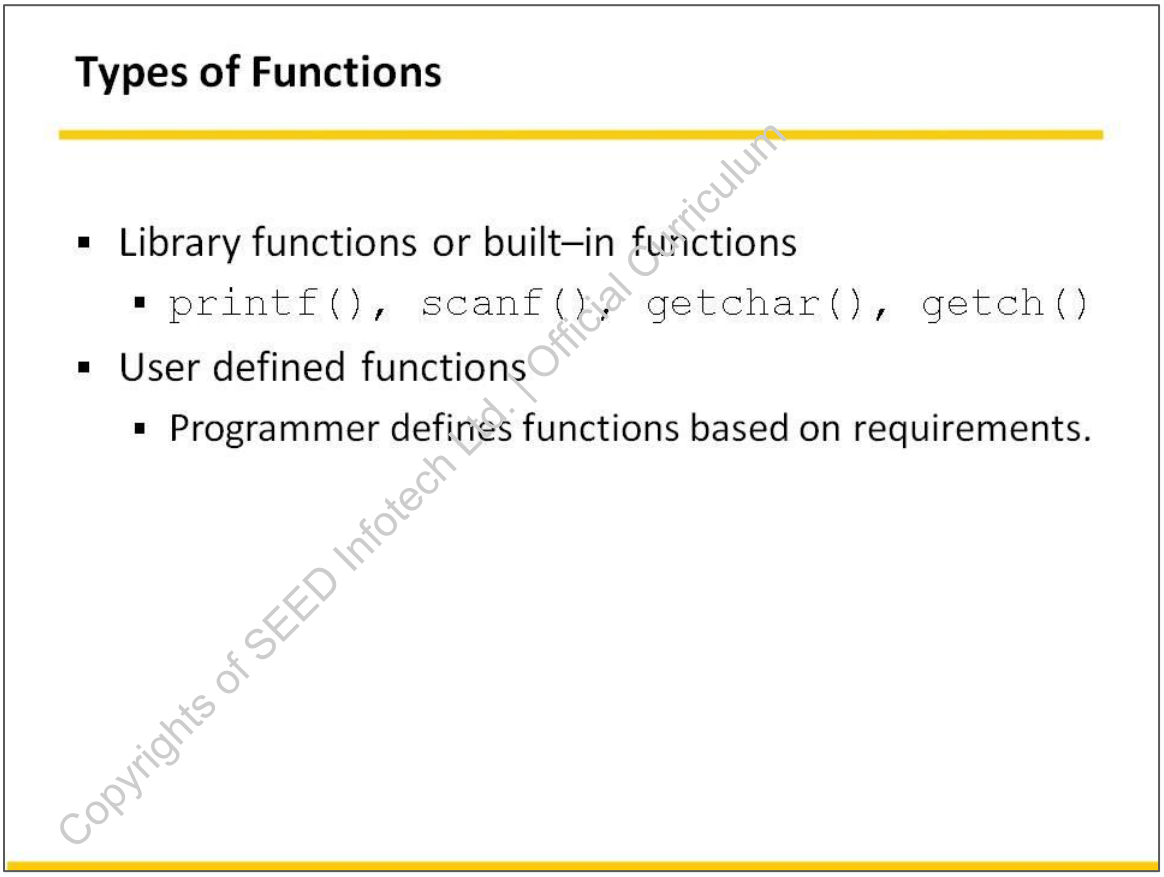
Function definition is the actual code of the function within brace brackets. It is also called as the body or implementation of the function. The linker links the function call with the definition of the function. It gives an error if the definition is not present.

Program flow with a function is demonstrated below:

```
void printDetails();  //prototype

int main()
{
. . .
//calling function
printDetails();   //function call
. . .
}

void printDetails()   //definition
{
  //called function
  . . .
}
```

List some built-in functions that you have already used. Open stdio.h and see its contents.

Explain the prototype of `printf()`.



There are two types of functions – library functions also known as built-in functions and user-defined functions.

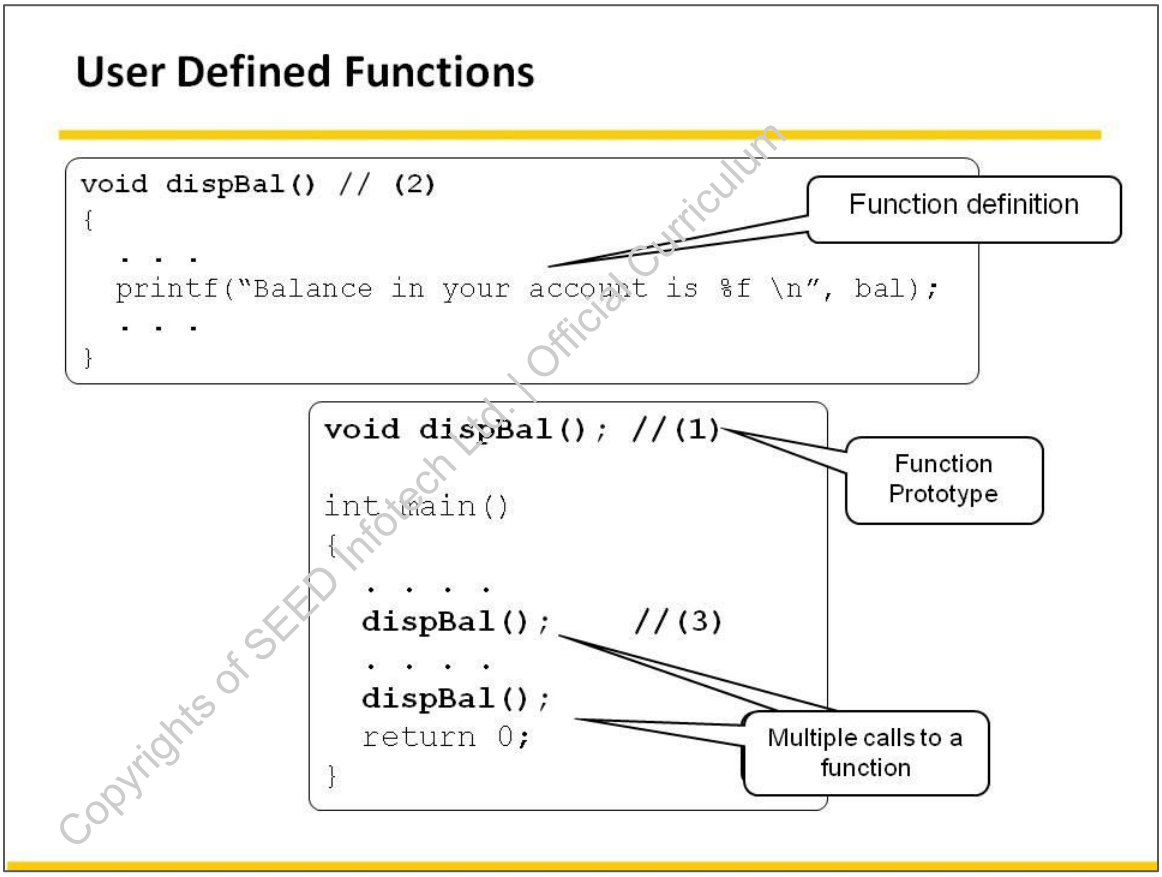
Library Functions

Library functions are standardized collection of header files and library routines used to implement common operations, such as input/output and string handling, and so on. They need to be simply called from the calling program. The header file containing the prototype of the called function must be included in the calling program.

For example, to use `printf()` or `scanf()` function, the `stdio.h` file should be included in the program.

User Defined Functions

User defined functions are written by the programmer as per the requirement of the program. They can later become a part of the 'C' function library.



When a function calls another function, control passes to the called function and the calling function waits for the control to return to it.

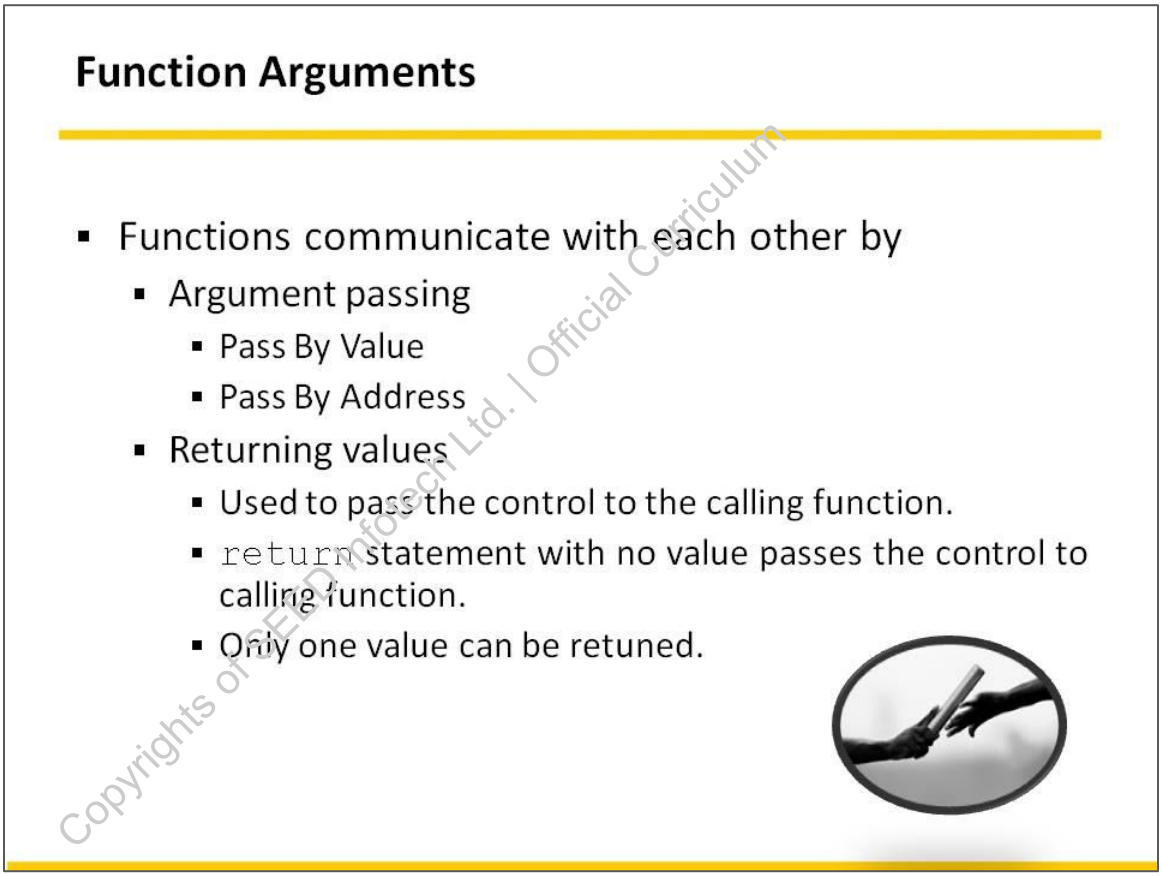
1. ANSI 'C' allows including a function prototype which gives the return type of the function and the type of each argument before defining the function. The prototype indicates the number and type of arguments by using a comma-separated list of the types.

   Following are some examples of prototypes:

```
void dispBal();
// or
int findSquare(int);
```

2. Implementation of a function is written as a block of code with arguments if required.
3. In the example shown on the slide, code to print a dashed line, forms the implementation of the function.

4. Call to the function can be given any number of times either from `main()` or from any other function as well. Function call is given by mentioning the name of the function or by passing arguments in some cases.



Functions are useful mainly because they can communicate with each other. According to K & R C, *parameter* is used for a variable named in the parenthesized list in the function definition, and *argument* for the value used in a call of the function. The terms *formal* argument and *actual* argument are sometimes used for the same distinction. They accept values from functions calling them and return values to the calling function.

Communication between functions is possible by

1. Argument passing from calling function to the called function.
2. Returning values from the called function to the calling function.

Argument Passing

Communication between the calling and called function is possible through arguments. There are two terms in this context. They are - actual arguments and formal arguments.

- Actual argument is the value or an expression that is passed during function call.

- Formal argument is the variable that is declared in the parentheses of function definition to accept the value passed as actual argument from the calling function. These variables are local variables and private to the function. They are assigned values every time the function is called.

```
int main(){                //demonstrates actual and
formal arguments
    . . .
  int num, result=0;
  result = findSquare(num);     /* num – actual
argument */
    . . .
}
int findSquare (int a) {        /* a – formal argument
*/
 int res;
  res = a * a;
  return(res);
}
```


In C, arguments can be passed to a function in two ways -

- Passing arguments by value
- Passing arguments by address (pointer)

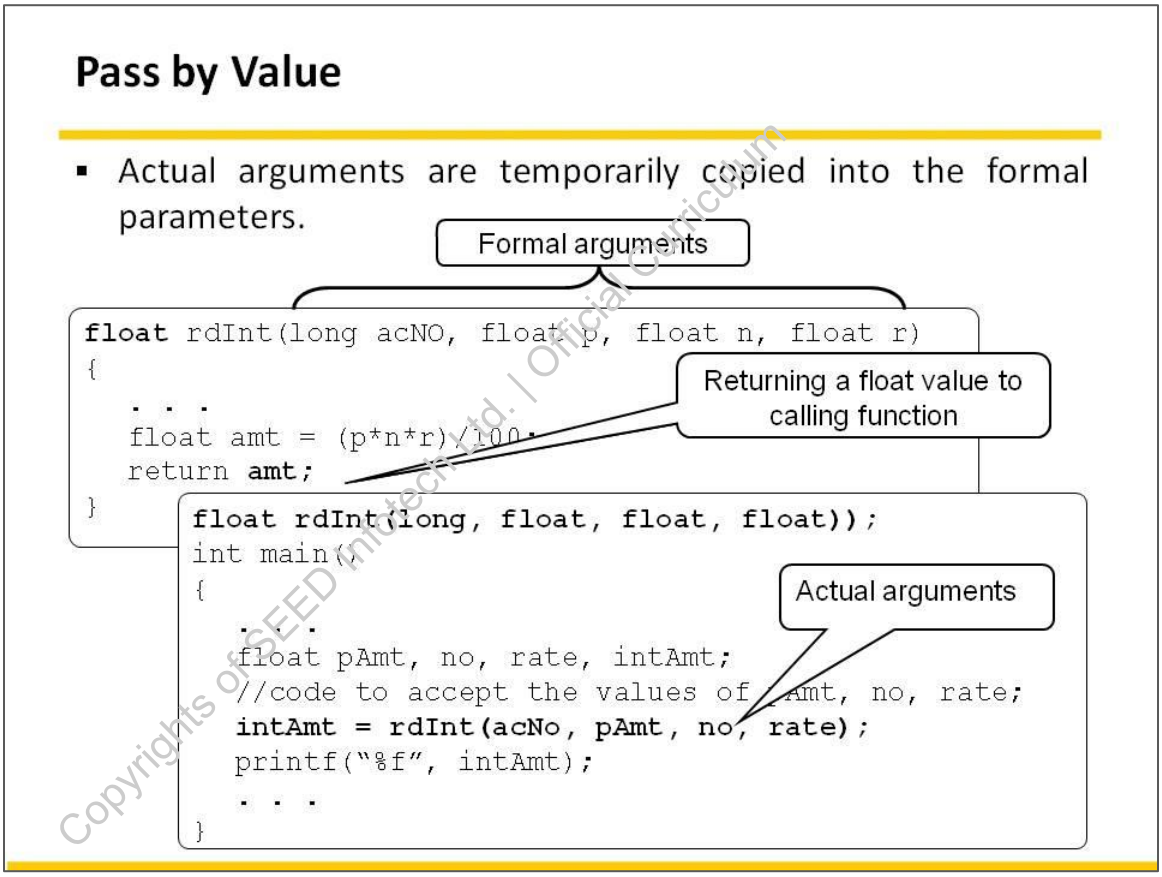
Returning Value

To send the information back to the calling function, the value is returned using `return` keyword.

- Only one value can be returned to the calling function.
- A calling function can ignore the value returned by a function. For example, the standard built-in function `printf()` returns an integer value which is usually ignored by the calling function.
- A function can contain a `return` statement with no value actually being returned. This is used to pass the control back to the calling function.

```
void printPyramid(int num)  // num indicates number
of rows
{
  . . .
  int num;
  if(num < 1)
    return;

  // print the pyramid of numbers
   . . .
}
```


- It is not necessary that a function should return a value. In such a case it is declared using `void` keyword.
- A function can return only one value but can have multiple `return` statements.

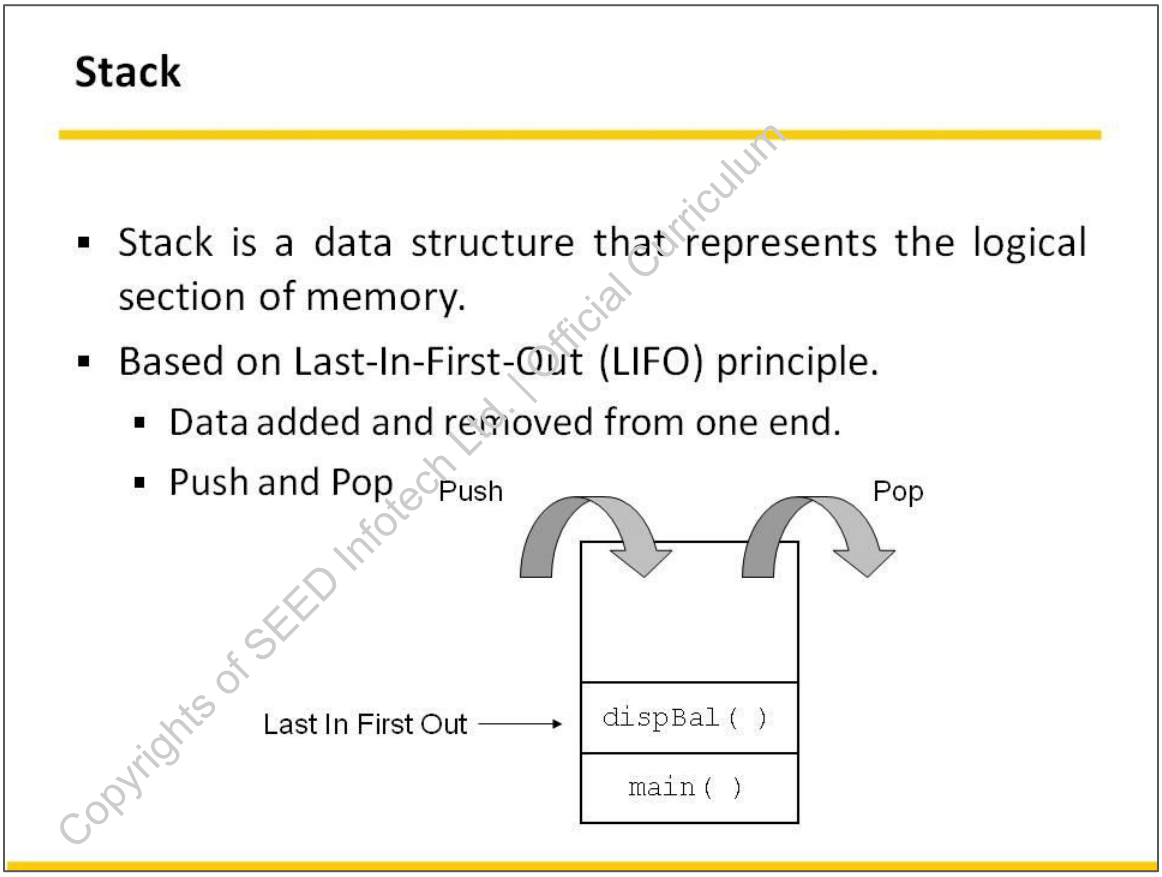


One of the ways to pass arguments to a function is "Pass by Value". When the arguments are passed by value, a temporary copy of the actual arguments is made into the formal arguments. The called function works on this copy of variables. The changes are made to the copy of variables. Hence the original values of these variables remain unchanged in the calling function. In this case, the actual arguments are read-only arguments.

In the above code snippet, the function `rdInt()`, calculates simple interest and returns it to the calling function. `acNo, pAmt, no, and rate` are actual arguments which are passed to the function. The arguments `acNo, p, n, and r` are formal arguments. The values of `acNo, pAmt, no and rate` are copied into the formal arguments.

Cdecl is a calling convention. In cdecl, actual arguments are pushed from left to right on the stack while the formal arguments are always pushed from right to left.

Read about calling conventions from MSDN.



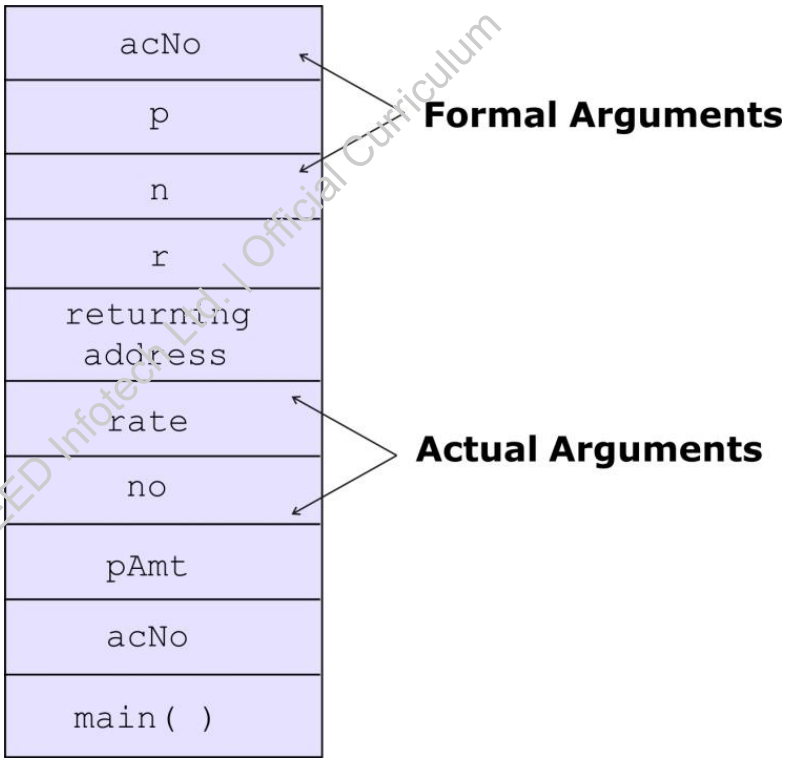
The term “Stack” is commonly used while referring to a collection of books or plates placed on top of the other. Consider an example of a stack of plates. A plate can be placed and removed from the top position only. Actually stack is a data structure which is based on the Last-In-First-Out (LIFO) axiom.

Every time a function is called, the data is pushed onto the stack. This collection of data for one function call is called a "stack frame". The stack frame contains the following data:

- The actual arguments
- The address (on the stack) of the calling program’s stack frame
- The return address – where to go when the function returns
- The automatic (local) variables

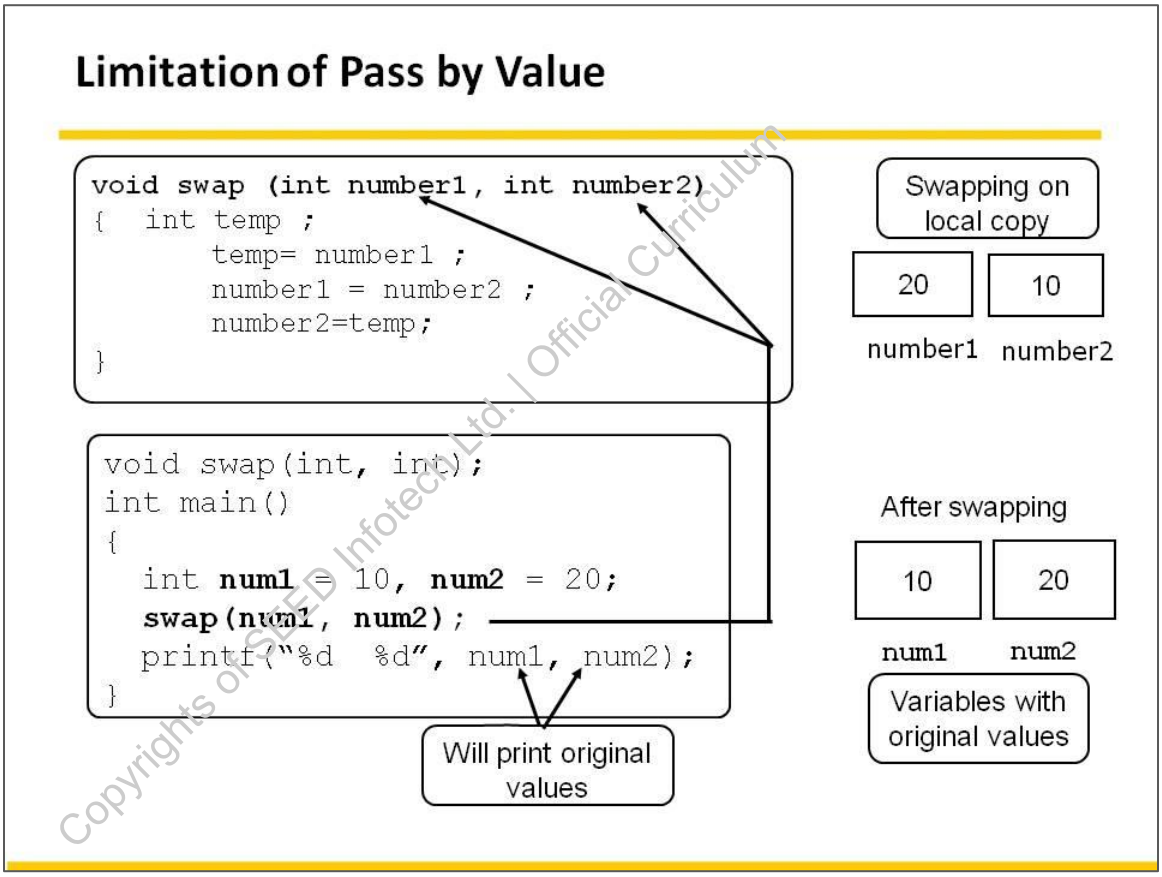
There are two key characteristics of frame allocations. First, when a local variable is defined, enough space is allocated on the stack frame to hold the entire variable, even if it is a large array or data structure. Second, frame variables are automatically deleted when they go out of scope.

The act of putting data on the stack is known as push and the act of removing data from the stack is known as pop.



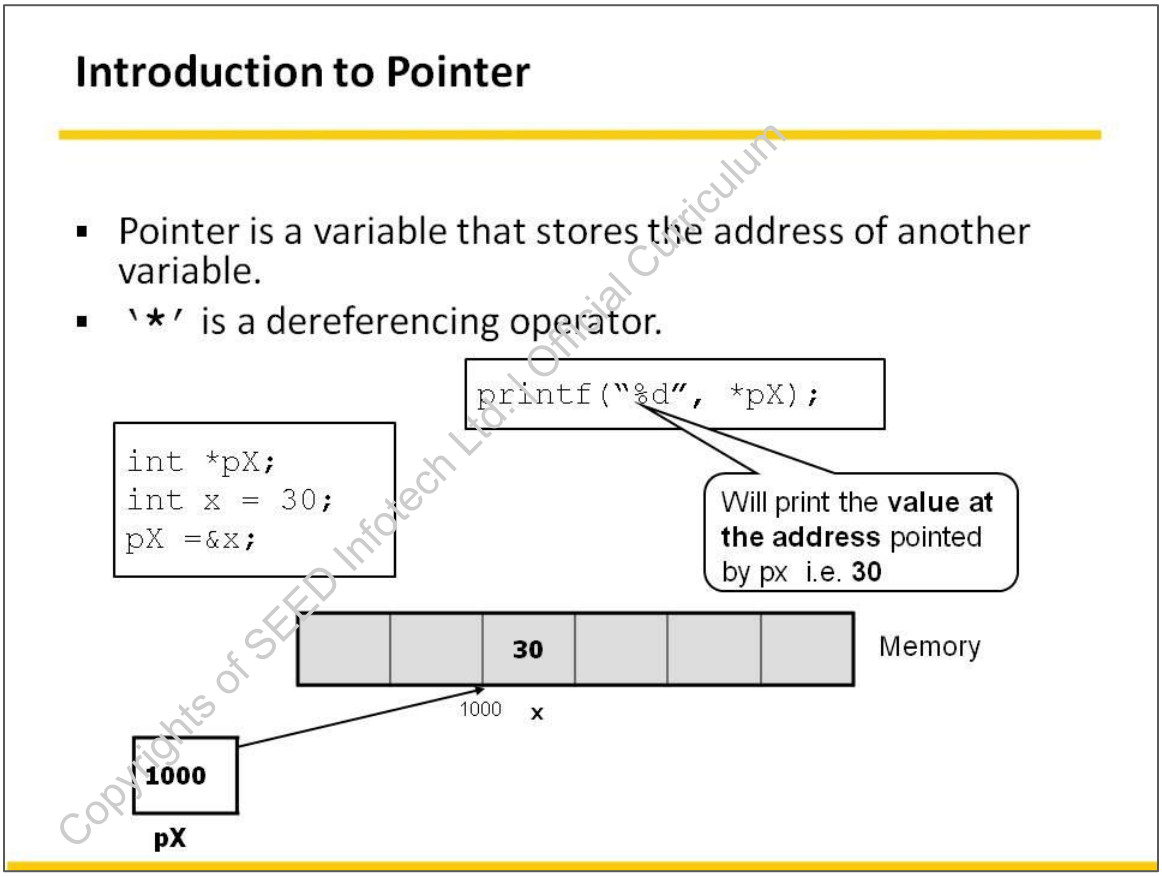
In the example above, when the `rdInt()` function is called, the variables (`acNo, p, n, and r`) declared within the function are automatically created on the stack. These variables are automatically destroyed (popped) when the control returns to the calling function.

When the function execution ends, the program control returns to `main()` by popping the return address from the stack.



In case of 'pass by value', the actual arguments remain unchanged. Changes made by the called function are made to the local copy of the arguments. This can become a limitation of the 'pass by value' mechanism. In the example shown above, the swapping of values takes place on the local copy and hence is not reflected in `main()`.

This can be overcome by passing the arguments by address. But this requires an introduction to the concept of pointers.



When a variable is defined, a memory location is allocated to it. This location has a number associated with it. This number is called its **address**. The symbol '&' (ampersand) is used to represent the address. A 'pointer' is a variable containing the address of another variable.

Symbol '*' is used to represent a pointer. It is called the dereferencing or indirection operator. For example,

```
int x = 30;
int *pX;
pX = &x;
printf("%d", *pX);
```

- Here, pX is a pointer variable and stores the address of an integer variable x (an arbitrary value as 1000)
- So pX will have value 1000. (x and *pX are both equal to 30)
- ***pX** will give the value at the address pointed to by pX, that is 30.

Similarly, pointers can be declared to a `float`, `char`, `double`, arrays and structures too.

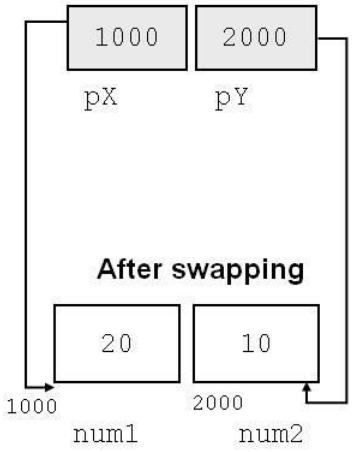
```
char *pCh;          //pointer to a character variable
float *pF;          //pointer to a float variable
double *pD;         //pointer to a double variable
```

Pointers are widely used in the implementation of data structures. You should implement at least linked list using pointers.

Pass by Address (Pointer)

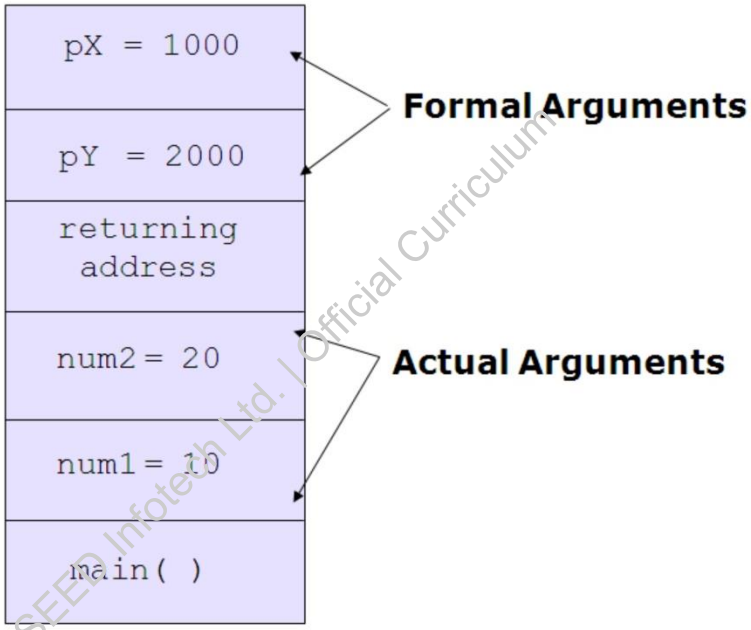
```
void swap(int *pX, int *pY)
{    int temp ;
     temp = *pX ;
     *pX = *pY ;
     *pY = temp;
}
```

```
void swap(int *, int *);
int main()
{
  int num1 = 10,num2 = 20;
  swap(&num1,&num2);
  printf("%d  %d", num1,num2);
}
```



The `swap()` function can now be implemented using 'Pass by Address' mechanism or by passing the pointer. In this case, addresses of `num1` and `num2` are passed by value to the `swap()` function. Pointer variables `pX` and `pY` are formal arguments. They point to the addresses of `num1` and `num2` respectively.

When the address of the variable is passed, the pointer points to the actual arguments. Thus the values of actual arguments i.e. `num1` and `num2` are modified.



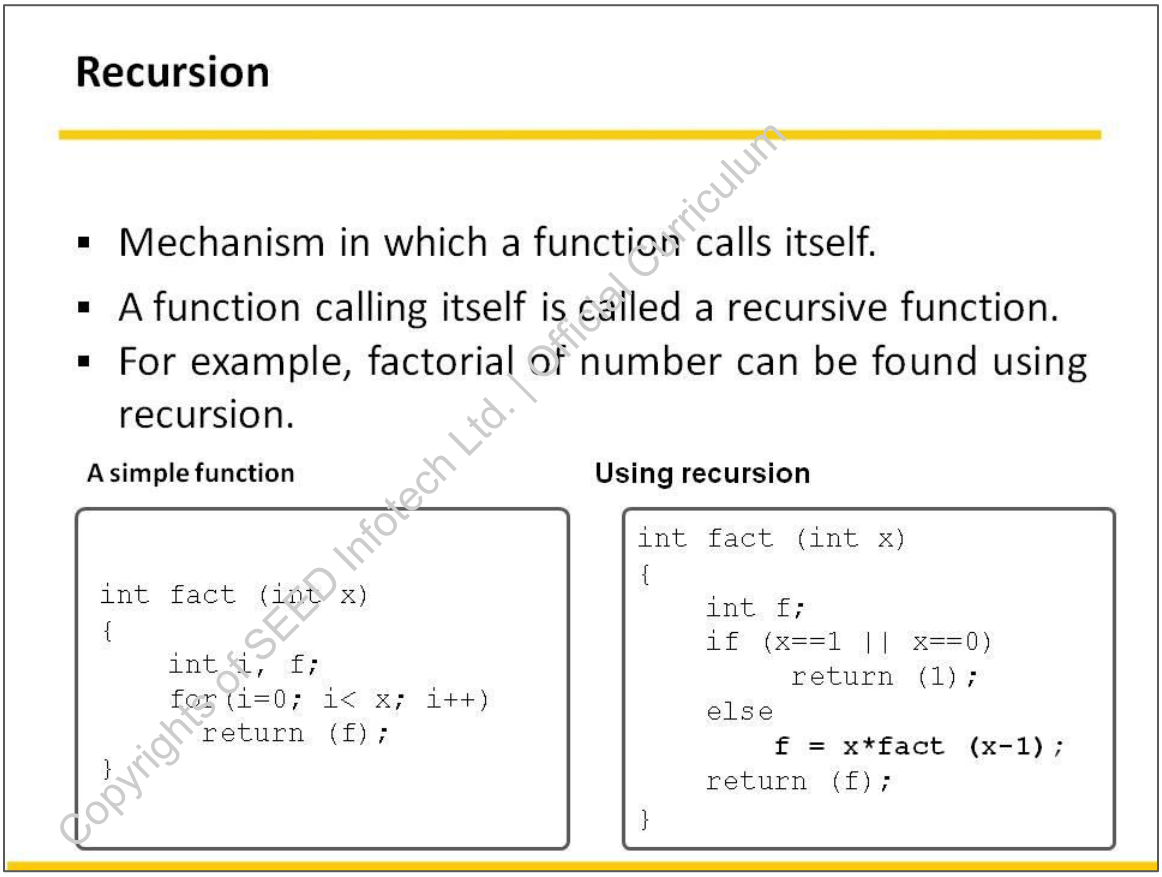
With this mechanism, more than one value can be implicitly returned to the calling function.

Write one single function to convert rupees into dollars and pounds. Print the converted amounts in `main()`.

(Assume 1 $ = Rs.40 and 1£ = Rs. 70)

Differentiate between ‘pass by value’ and ‘pass by address’.

Read about functions with variable number of arguments.



Recursion is a programming technique in which a function calls itself. A function calling itself is called a recursive function. Recursion is used often where loops are used to implement iteration. Simple algorithms are easier to implement using loops. But writing complex programming logic for problems like Towers of Hanoi, traversal mechanism in data structures like Trees, etc., recursion proves to be a better solution.

Though recursive solutions tend to be more elegant in such cases, they have performance issues. Every time the function calls itself, the control is transferred back and forth from calling to called functions. This causes many overheads of pushing and popping address of the function from stack.

It is interesting to note that `main()` can be called recursively, though this is rarely done.

Writing a recursive function

There is always an end condition for recursion, otherwise it may cause stack overflow. A simple example is mentioned on the slide that is implemented using a recursive function. Though this example works efficiently using loops, recursion can be easily understood with it.

The statement `f = x * fact(x-1);` is a recursive call to the function `fact()`. The function `fact()` calls itself but the value of arguments passed each time is different.

Execution of recursive function

On executing the program, if the input given is 1 or 0, then

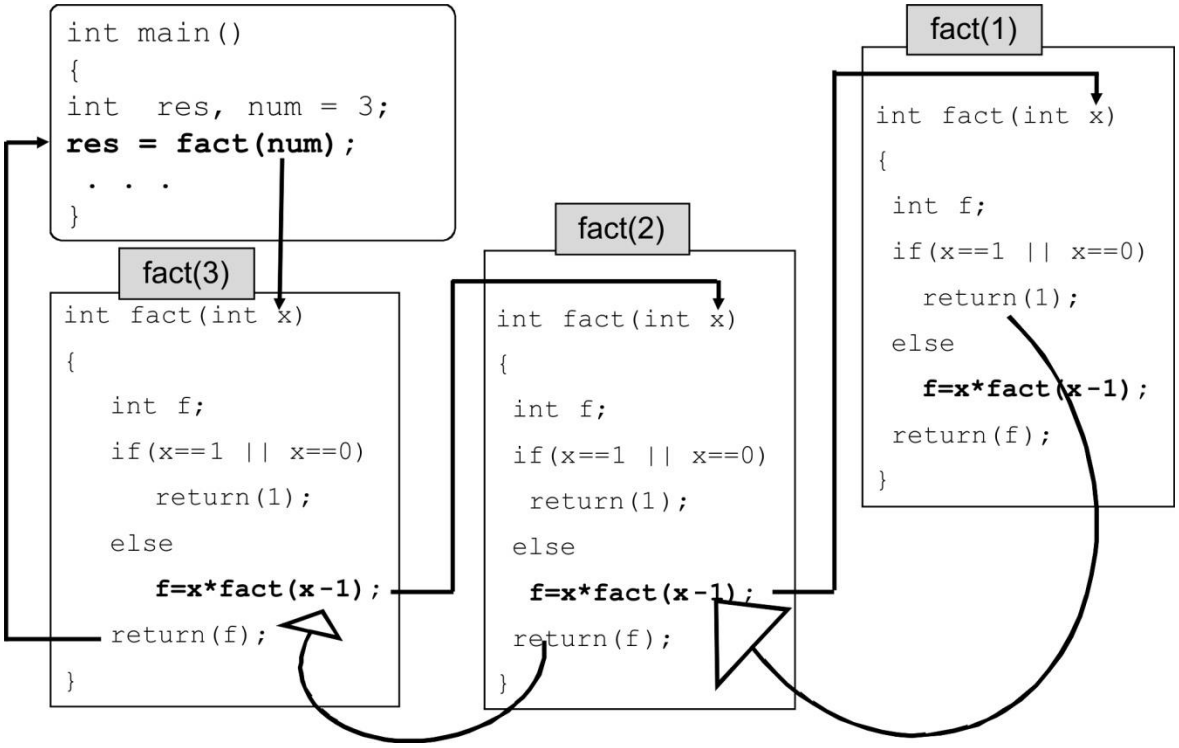
- The value of `num = 1` is copied to x.
- The condition (`x == 1`) or (x == 0) is true hence '1' is returned to the `main()` and printed there.

				`fact(1)`						
		`fact(2)`		`fact(2)`		`fact(2)`				
`fact(3)`		`fact(3)`		`fact(3)`		`fact(3)`		`fact(3)`		
`main()`		`main()`		`main()`		`main()`		`main()`		`main()`
Fig 1		Fig 2		Fig 3		Fig 4		Fig 5		Fig 6

When the return statement is encountered in a function, control had to return to the calling function. The returning address is stored on the stack. The above figure shows the stack with recursive function calls when the input value of 3 is passed to the function.

- When function `main()` starts executing, it is put on the stack. `fact()` is called from main with a value of 3. `fact(3)` is added to stack. Fig 1.
- In `fact(3)` value of x is equal to 3. (x == 1) is false hence `f=3*fact(3-1)` is executed i.e. `fact(2)` is called from `fact(3)`. `fact(2)` gets added to stack. Fig 2.

- In `fact(2)` value of x is equal to 2. Again, `(x==1)` is false hence `f = 2*fact(2-1)` is executed i.e. `fact(1)` is called from `fact(2)`.`fact(1)` gets added to stack. Fig 3.
- In `fact(1)` value of x is equal to 1. But here `(x == 1)` is true hence `fact(1)` returns value of 1 and is removed from stack. Fig 4.
- Now the topmost function on stack is `fact(2)`. The value is returned to function `fact(2)` at the place it left i.e. at the statements `f=2*fact(1)`. Here `fact(1)` is replaced by 1.Therefore f is assigned the value 2. Next statement `return(f)` returns value of 2. `fact(2)` is removed from stack. Fig 5.
- Now topmost function on stack is `fact(3)`. The control returns to `fact(3)` at the place it left i.e. at `f = 3*fact(2)`. That is, `f=3*2=6`. Therefore, `return(f)` returns value of 6. `fact(3)` is removed from stack. Fig 6.
- Finally value 6 is returned to function `main()`.



The condition that leads to a recursive function returning without making another recursive call is referred to as the base case. This is the condition that terminates the recursive calls. It is essential that every recursive function should have a base case to prevent infinite recursion and the consequent stack overflow.

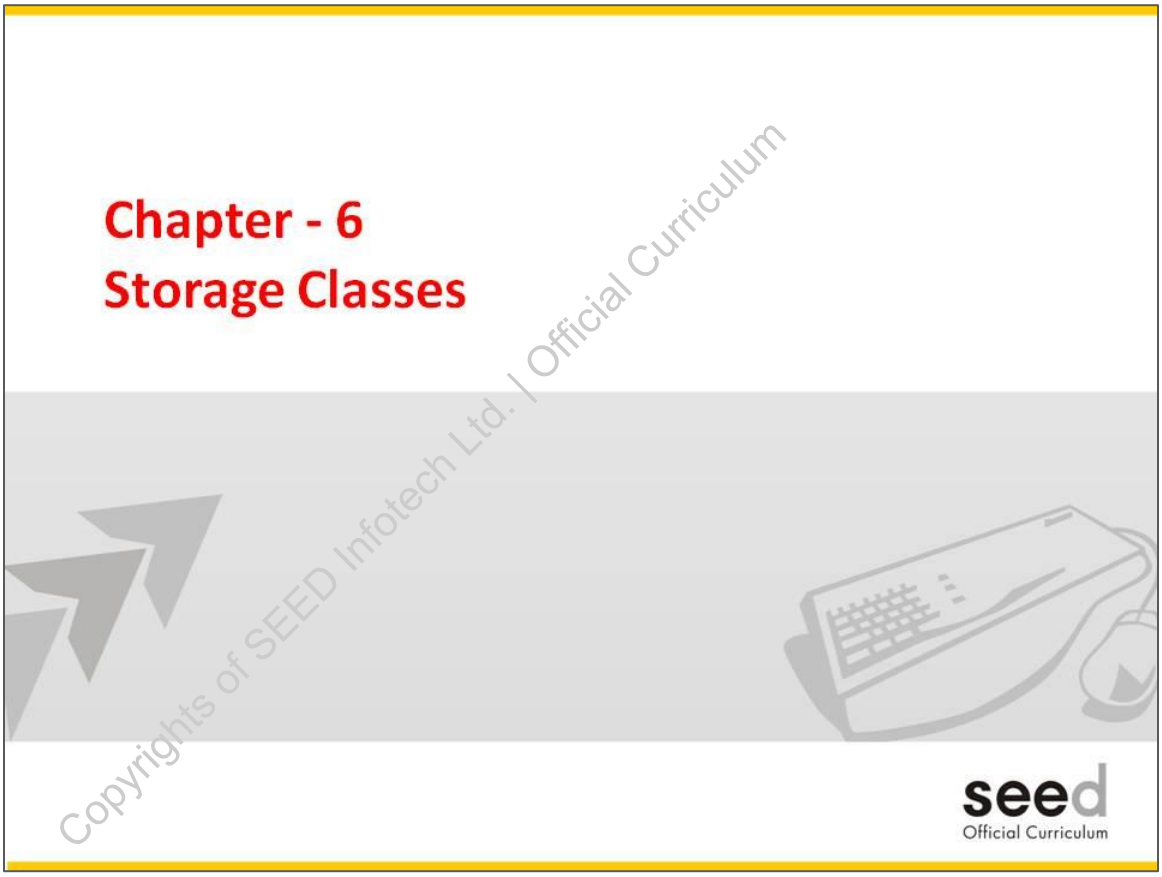
Overheads in Recursion

Calling a function involves certain overheads. Control is transferred from the location of the call to the beginning of the function. In addition, the arguments to the function and the address to which the function should return are pushed onto the stack so that the function can access the argument values and know where to return.

As a result of the overhead, the `while` loop approach executes more quickly than the recursive approach. If there are a large number of function calls due to the recursive function, it might be desirable to avoid recursion.

Recursion has another drawback. Memory is used to store all the intermediate arguments and return values on the stack. This might cause problems if there is a large amount of data, leading to stack overflow.

Recursion is used in the implementation of complex algorithms like Tree traversals in data structures. Use recursion only when necessary; otherwise it kills brevity and slows performance.

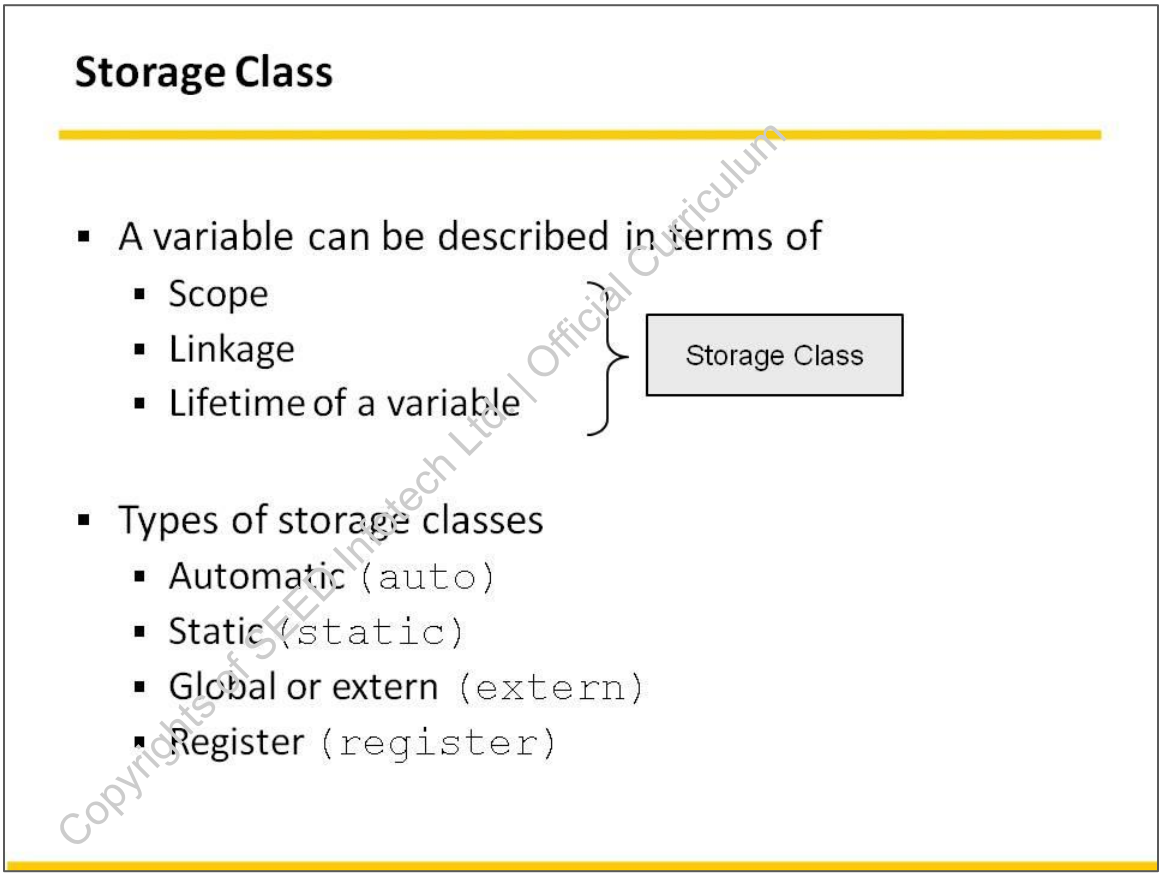


This chapter covers types of storage classes like automatic, static, global and register and how to use variables of these storage classes.

Objectives

At the end of this chapter you will be able to:

- Define a storage class.
- List the different types of storage classes.
- Declare and use
  - Automatic variable
  - static variable
  - Global variable and
  - Register variable



A data type of a variable indicates the type of data that will be stored in the variable and the amount of memory to be allocated to the variable. A variable is described in terms of its scope, linkage, and lifetime. Storage class is a combination of all the three.

Scope

Scope, also known as visibility of the variable, describes the part of a program that can access a variable. A variable can have block scope, function prototype scope or file scope depending on the storage class of that variable.

Linkage

A variable may or may not have a linkage depending on the scope where it is declared.

- Variables with block scope or function prototype scope have no linkage.
- A variable with file scope can have either internal or external linkage. A variable with external linkage can be used anywhere in a multi-file program. A variable with internal linkage can be used anywhere within a single file.

Storage duration (Lifetime)

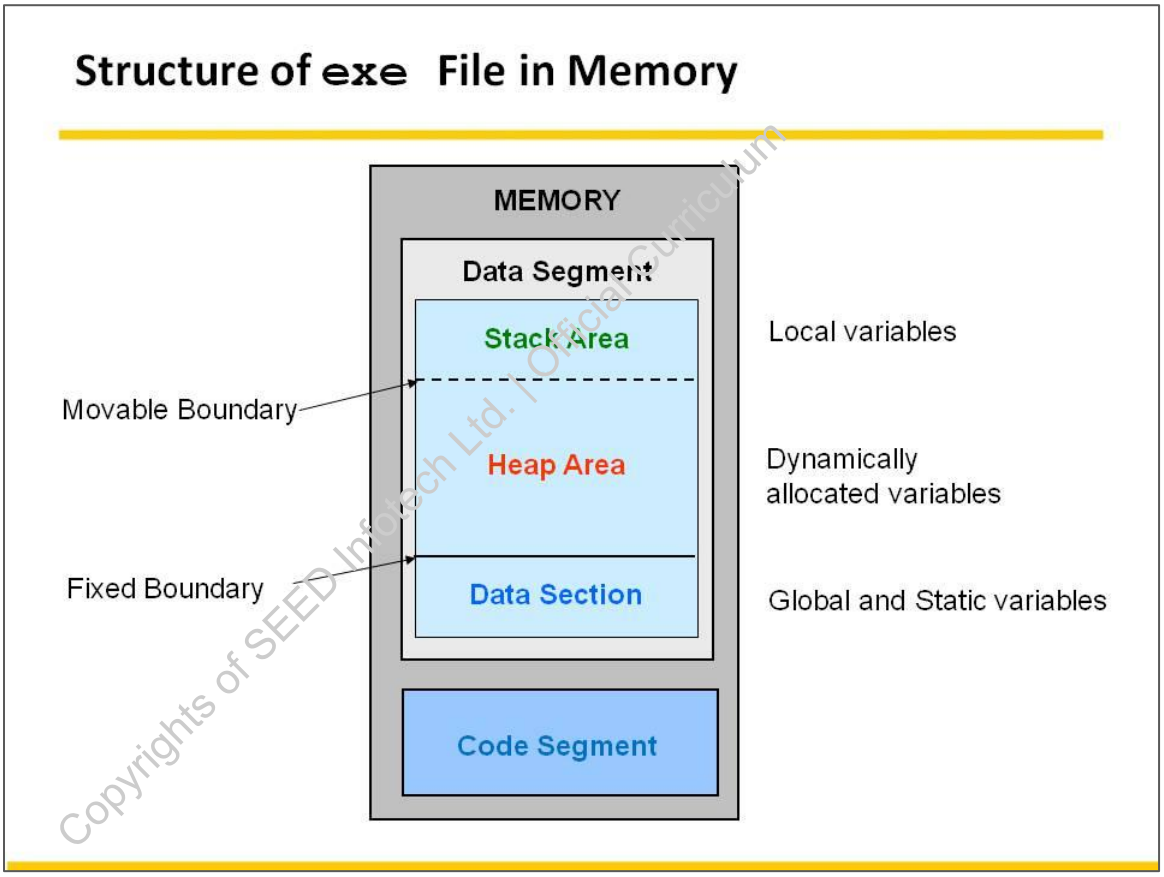
The storage duration or lifetime of the variable is the duration up to which the variable exists or the length of time that it retains the value that has been assigned to it.

Initial Value

The initial value of the variables declared with each of these storage classes is different.

There are four types of storage classes:

1. Automatic
2. Static
3. Global
4. Register



Before going ahead with storage classes, it is essential to understand the structure of an `.exe` file in the memory. This would help in understanding the part of memory where the variables are stored.

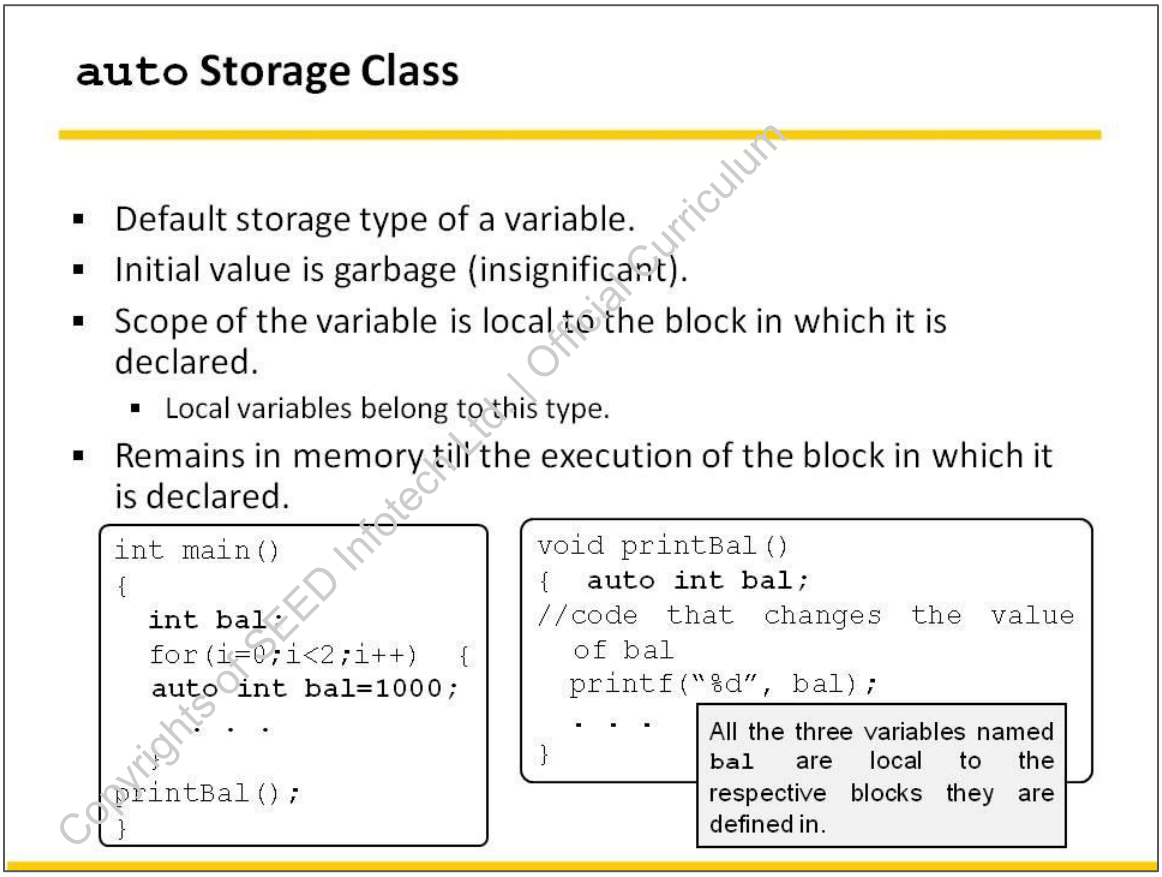
When `.exe` is loaded into memory, it is organized into two areas - code segment and data segment.

The code segment is where the compiled code of the program resides.

The data segment contains the data part of the program.

- The stack section is where memory is allocated for local (automatic) variables within functions. Memory allocated for variables in this area contains garbage values.
- The heap section provides storage for variables that are dynamically allocated memory.
- The data section is where the global and static variables are allocated memory.

You should know what is created on stack and what on heap.



Variables having automatic storage class are called automatic or local variables. It is the default storage class of variables declared without any specific storage class. These variables are stored in the stack section of the memory.

Variables of this storage class are declared with **auto** keyword.

```
auto int variableName;
```

Initial Value

If an automatic variable is not initialized, it contains garbage value.

Scope

Automatic variables are always declared within a function. Their scope is confined to the function or block in which they are declared. Function prototype scope runs from the point the variable is defined to the end of the prototype declaration.

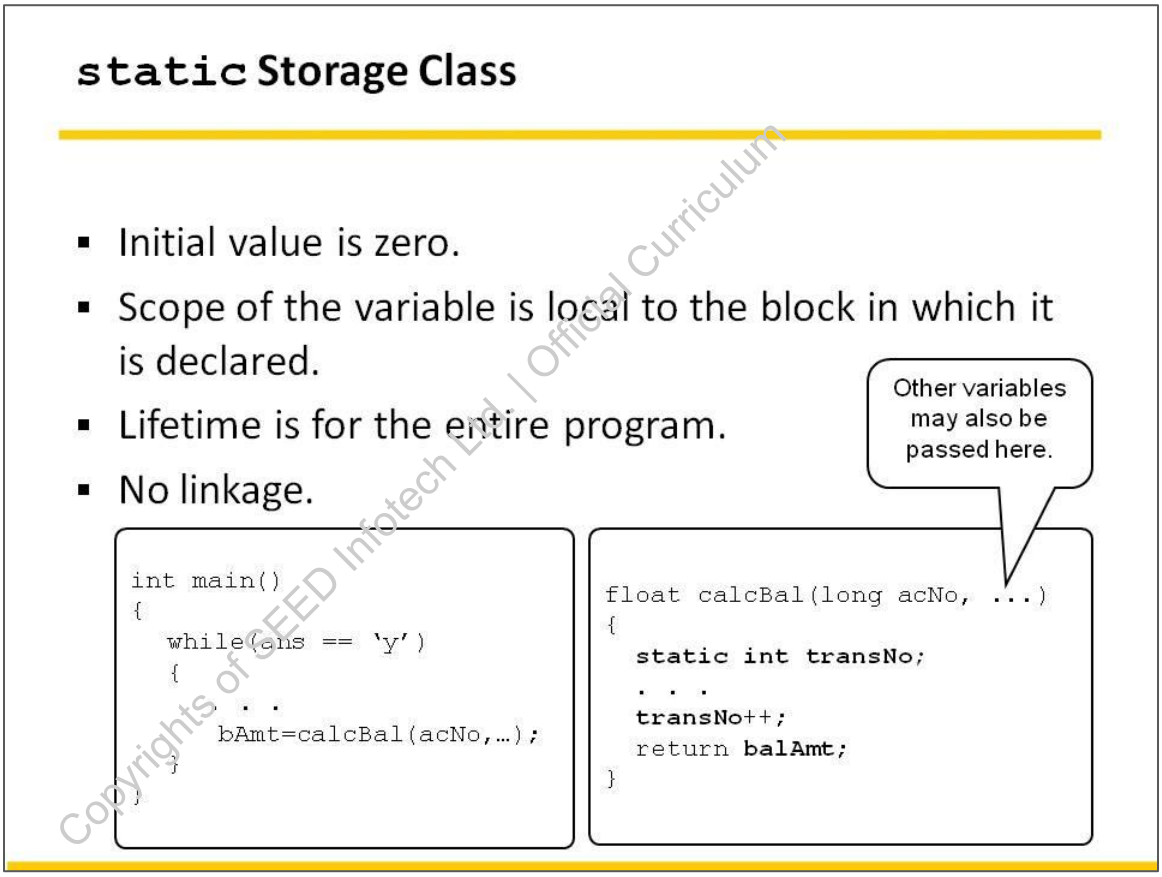
Lifetime

The value of automatic variables is lost after the control goes out of the block in which they are defined and are re-initialized when the block or function is re-entered.

Linkage

Automatic variables have no linkage.

In the example mentioned above, variable `bal` is declared at different places in a program – in `main()` function and `for` loop. The scope is different and hence the variables would hold different values. The variable `sum` is local to `printBal()` and would be initialized to zero every time the function is called.



Variables declared with keyword `static` are called static variables. Static variables are stored in data section of memory.

```
static int variableName;
```

Initial value

Static variables are initialized to zero by default. Once initialized, they retain the value which they are assigned during the execution of the block.

Scope

Like automatic variables, the scope of static variables is local to the block in which they are declared.

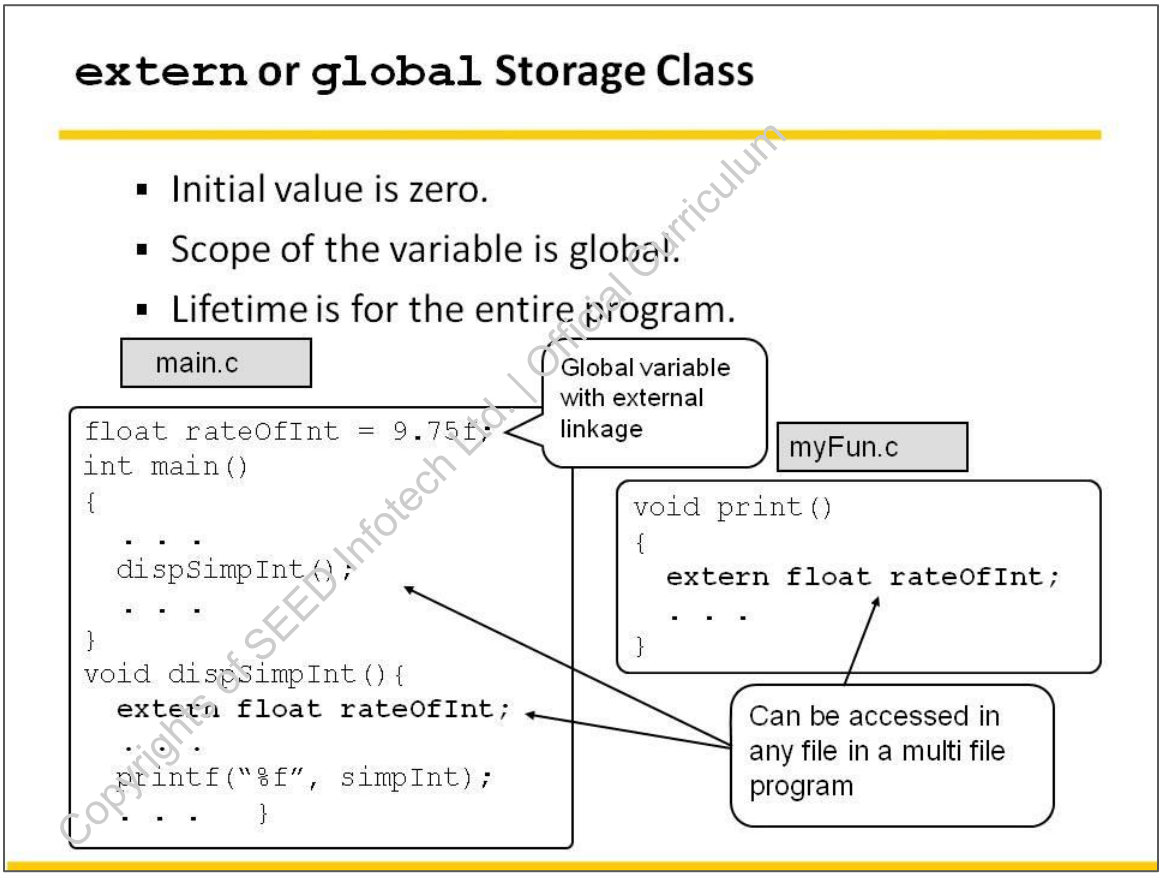
Lifetime

Since these variables reside in the data section of the memory; their lifetime is till the program executes.

Linkage

Static variables have no linkage if declared in function scope.

In the example given above, the variable `transNo` is declared as a static variable. It is initialized to zero though not explicitly mentioned. The changed value of `transNo` would be retained between consecutive calls to `calcBal()`.



External variables are defined outside any function. They are global to the entire program. When referred in any function they are declared as `extern`. Like static variables, they are stored in data section of the memory.

```
extern int variableName;
```

Initial value

Extern variables are initialized to zero by default. Once initialized, they retain the changed value during the execution of the block.

Scope

The scope of `extern` variables extends from the point at which they are defined to the end of the program. Scope of `extern` variables is said to be global. They can be accessed from any function within the program.

Lifetime

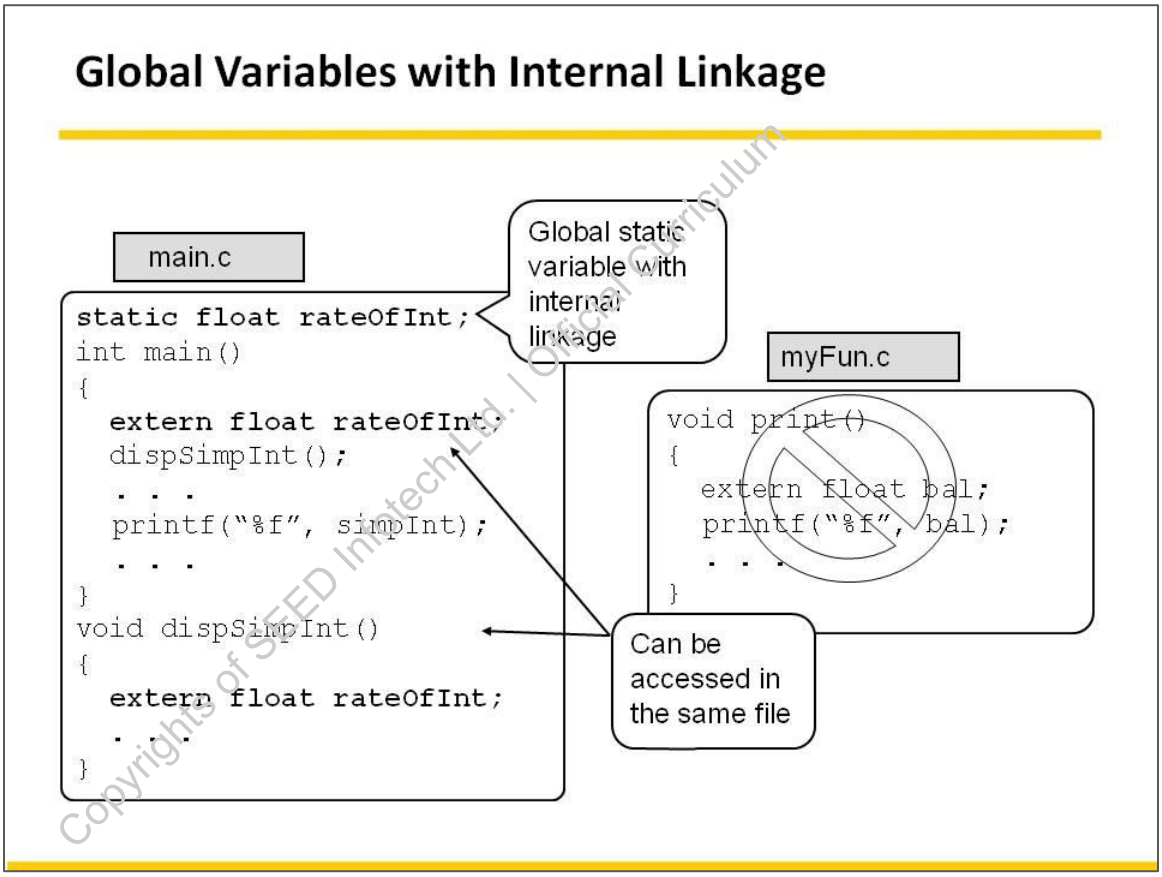
The lifetime of `extern` variable is until the end of program execution.

Linkage

Global variables have external linkage.

In the program example above, variable `rateOfInt` is declared as an `extern` (global) variable. It is declared before all the function definitions in the file `main.c`. It is a global variable with external linkage. It is visible to all functions in all files within a multi-file program.

Since the global variable is stored in the data section, the value of `rateOfInt` variable will persist throughout the program. Initially, its value is 9.75. This value will be accessed in `dispBal()`.The next time it is called, `rateOfInt` is still the same. Thus, the value of `rateOfInt` will be retained between different function calls. It would be also accessible to `print()which is in` another file.



Consider the same example used earlier, to understand global variables with internal linkage. The variable `rateOfInt` can be accessed by functions only in `main.c` file, as it is defined with `static` keyword. Such variables have only file scope. Scope, lifetime and initial value are same as that of extern variables with external linkage.

register Storage Class

- Storage is within CPU registers.
  - If registers are unavailable, treated as auto variables.
- Initial value is garbage.
- Scope of the variable is local to the block in which it is declared.
- Remains in memory till the execution of the block in which it is declared.

```
int main()
{
  register int i;
  for (i=1;i<=10;i++)
     printf ("\n%d", i);
     . . .
}
```


Normally, when operations are carried out, information is transferred from the memory to the registers. The results are then transferred back from registers to the memory. This takes some time. Variables like loop counters, are required a number of times. For faster execution of loops, the loop counter variables can be stored in CPU registers itself. These variables are declared with `register` keyword.

```
register int variableName;
```

Initial value, scope, lifetime and linkage specifications are the same as that of automatic variables.

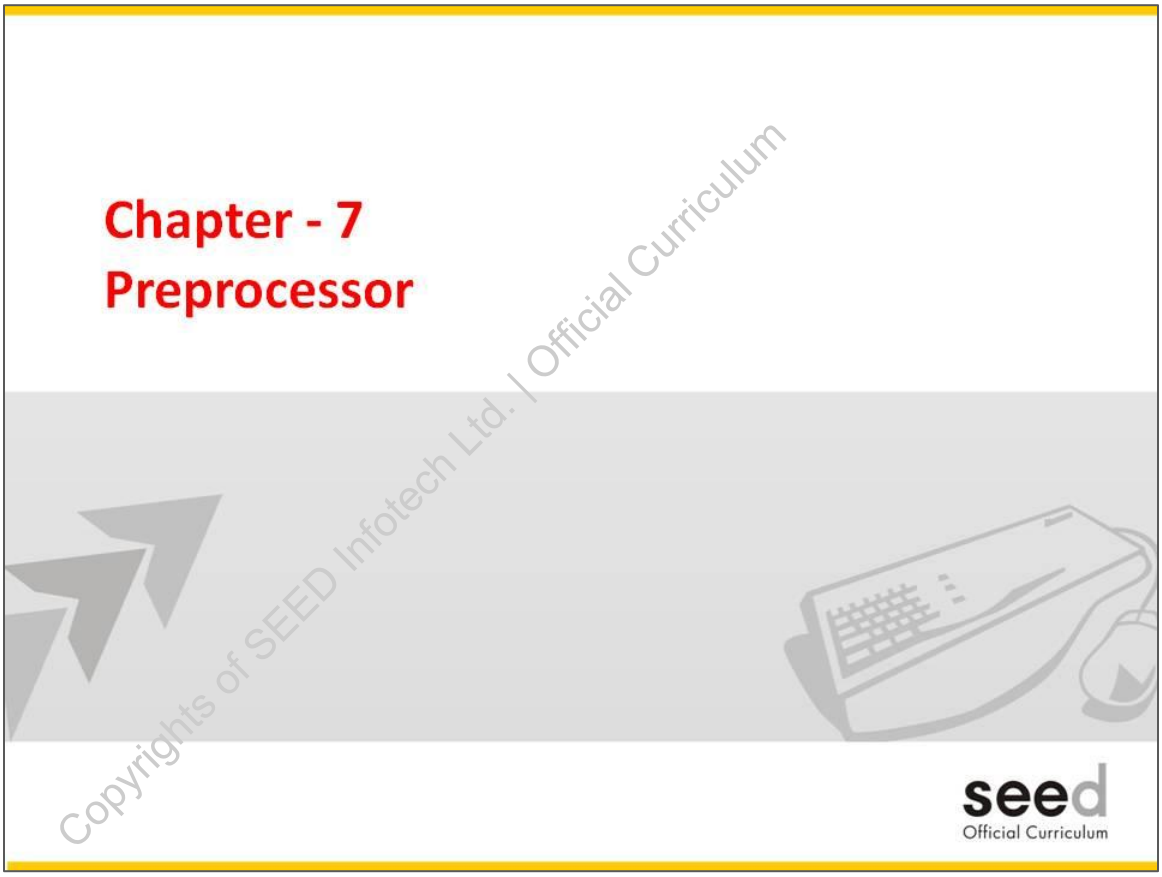
Important thing to note is that declaring a variable to be a `register` does not necessarily mean that CPU registers will always be available, as they are limited. If a register is unavailable, then the variable is treated as an automatic variable. The `register` storage class cannot be used for all types of variables because of the size of registers. They are mainly used for integer variables.

The following is the summary of all storage classes:

	Auto	Register	Static	Global
Keyword	auto	register	static	extern
Scope	Local	Local	Within a Function	Across files
Life	Within a block	Within a block	Throughout the program	Throughout the program
Initial Value	Garbage	Garbage	Zero	Zero
Memory	On Stack	Registers	Data Section	Data Section

Read more about storage classes from C Primer – by Stephen Prata

When you think of referring a variable, give it most limited scope. Local variables are safest. Use global variables if and only if there is no other option. Giving more scope or lifetime than needed can create more issues.

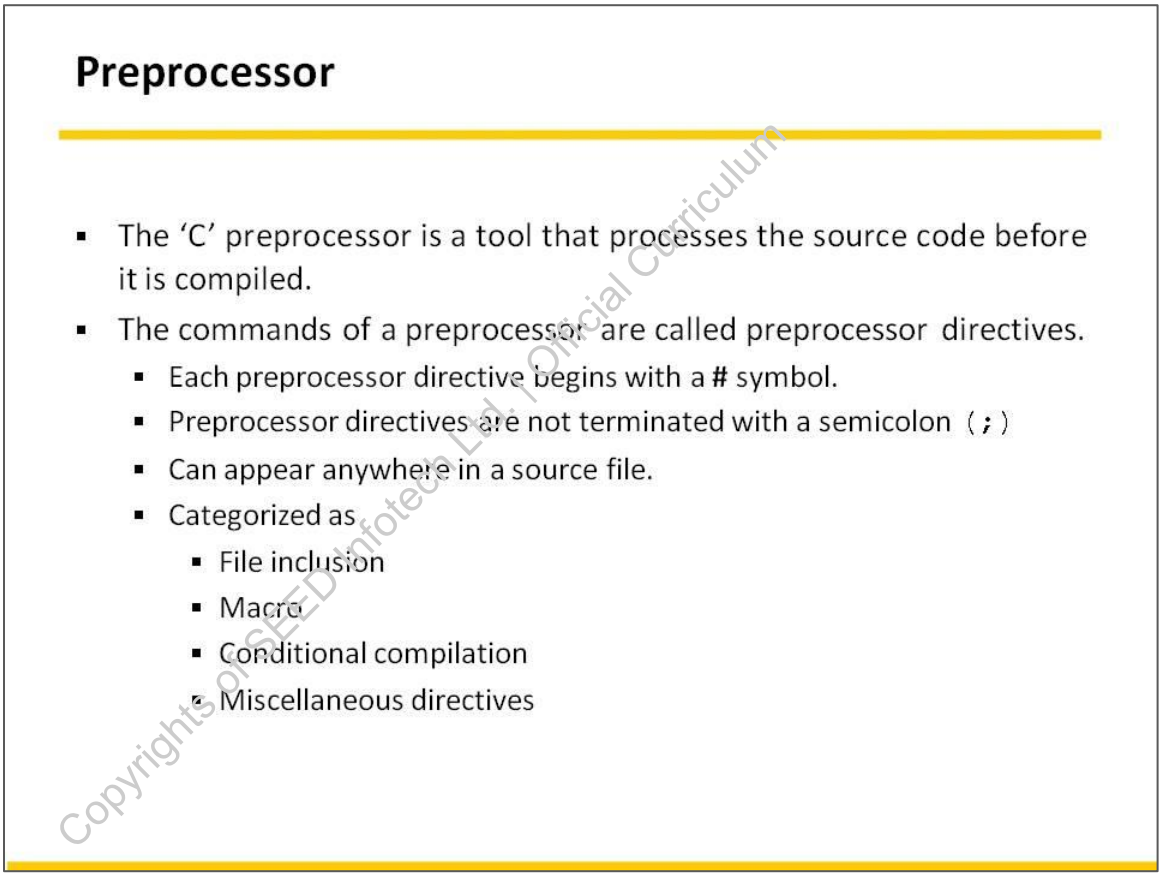


This chapter covers the different types of preprocessor directives like macros, file inclusion, conditional compilation and other directives.

Objectives

At the end of this chapter you will be able to:

- Define a preprocessor.
- List different types of preprocessor directives.
- Define a simple macro and use it in a program.
- Define a function macro and use it in a program.
- State differences between a macro and a function.
- Use file inclusion preprocessor directive.
- Identify the need of conditional compilation.
- List other macros.



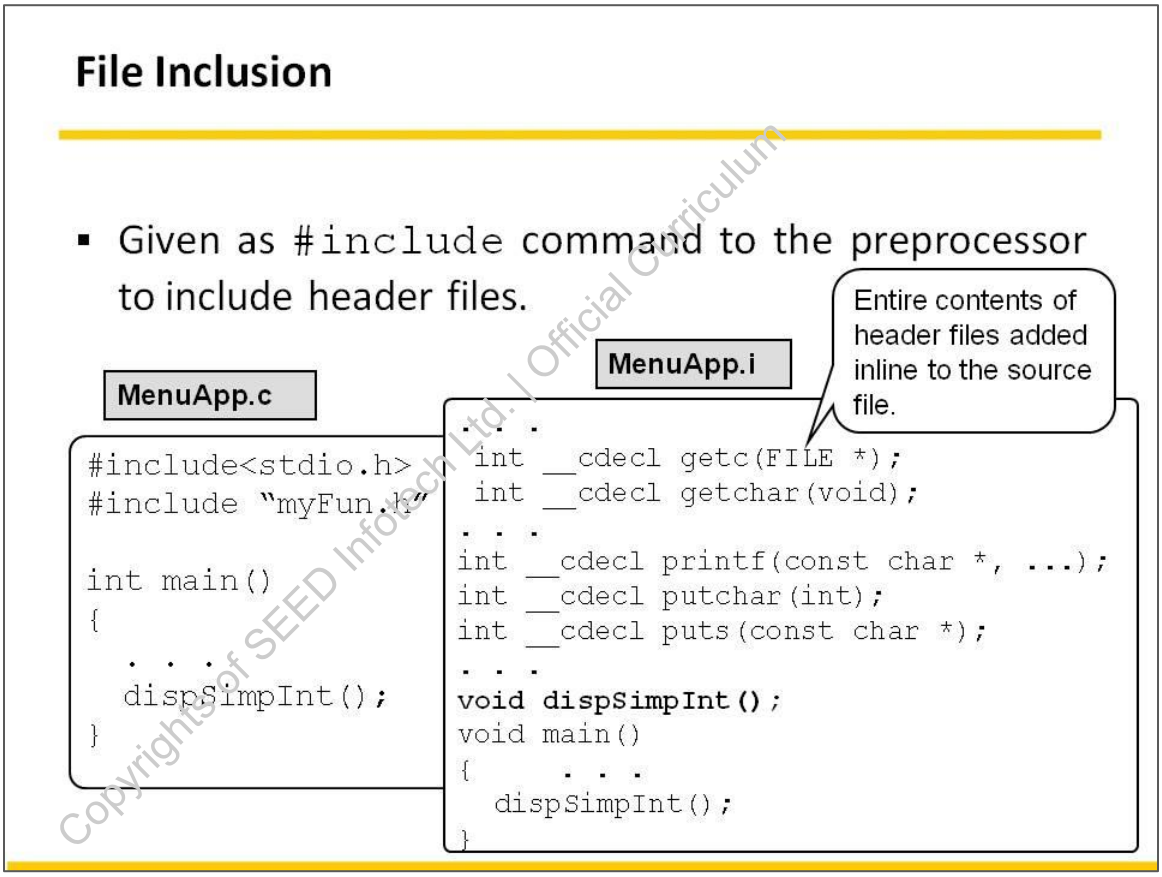
The C preprocessor is a tool that processes the source code before it is compiled, hence the name preprocessor. It replaces the symbolic abbreviations in a program with the directions it represents. A 'C' preprocessor is a text substitution tool.

The commands of a preprocessor are called preprocessor directives. A directive can appear anywhere in the source file. It starts with a `'#'` symbol.

- The preprocessor requires its expressions to be one logical line.
- Preprocessor directive can be placed anywhere in the 'C' program. But it is generally placed at the beginning of the program before the `main()` function.
- Preprocessor directives are not terminated with a semicolon.

Preprocessor directives can be categorized into four types.

1. File Inclusion
2. Macro
3. Conditional compilation
4. Miscellaneous directives.



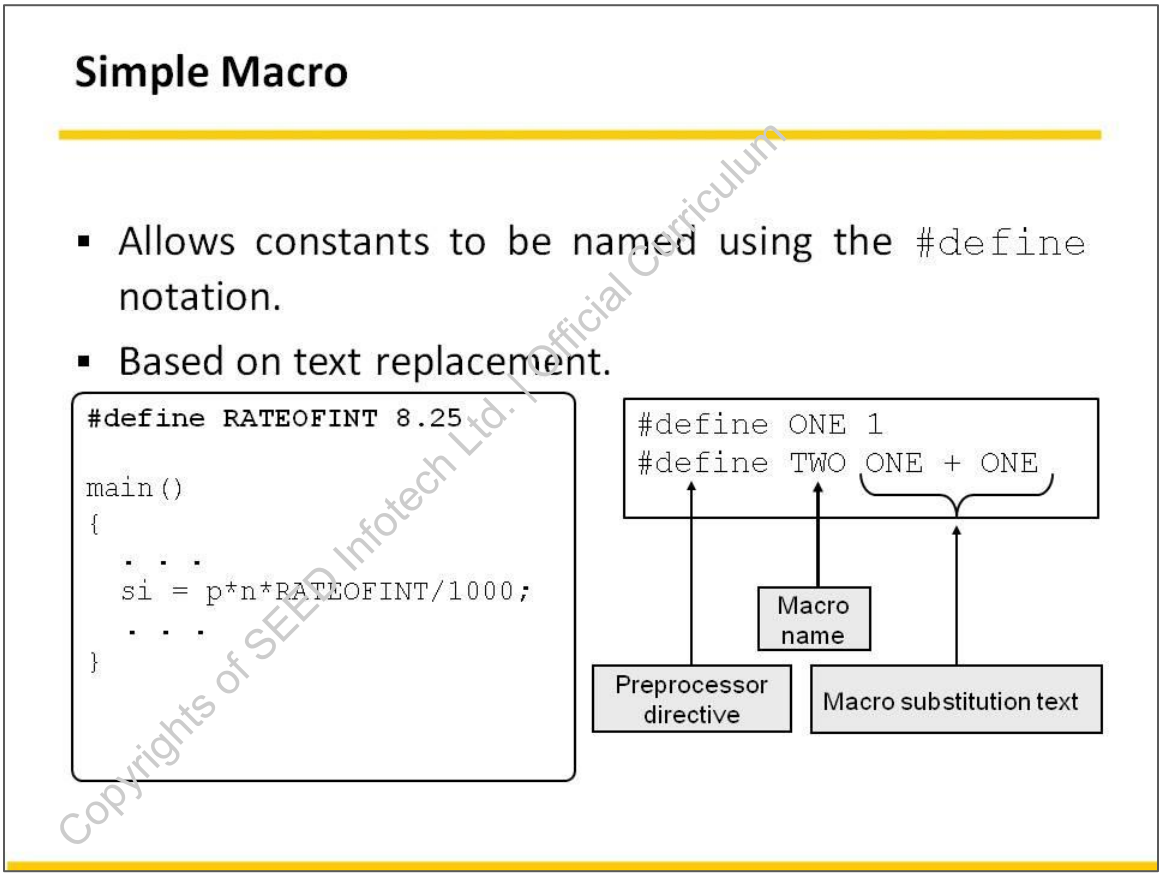
The `#include` directive has been used right from the first program that was written in C but now its use can be understood well. It is an instruction to include the entire contents of another file at that point. This is generally used to read header files. The entire contents of the included file are added to the source code file. This is a temporary file with a '`.i`' extension. The compiler requires information like function declarations, structure declarations, macro definitions, etc., before it starts compiling the code.

Library header file names are enclosed in angle brackets < >. These instruct the preprocessor to search for the header file in the standard path or system directories.

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

In multi-file programs, user defined function declarations or macro definitions are kept in header files and included in the program file using `#include " "`. Local header file names are usually enclosed by double quotes. In this case, the preprocessor searches for the specified file in the current working directory or in the specified path.

```
#include "myFun.h" //searches in the current working directory
#include "/user/Common/Employee.h" //searches in specified path
```



Macros are preprocessor directives that are defined using #define directive. The combinations backslash/newline is deleted before preprocessing begins, so the directive can be spread over several physical lines. These lines should constitute a single logical line. There can be two types of macros

1. Simple macro
2. Macro with arguments (function macro)

Simple Macro

Macro with no arguments is called a simple macro. Whenever a macro name is encountered in the program, simple text replacement takes place at that point in the program.

Syntax

```
#define MACRONAME macroSubstitutionText
```

- The rules for giving a macro name are the same as for a variable name. Generally macro name is written in uppercase letters.
- `#define` defines an identifier (the macro name) and a string (the macro substitution text) which is substituted for the identifier everywhere the identifier is encountered in the source file.

```
#define RATE_OF_INT 8.25      //simple macro
definition
si = p*n*RATE_OF_INT/100;     //in main() function
```

- The `#define` preprocessor allows constants to be named using the `#define` notation

```
#define UPPER 20  // simple macro defined
int main()
{
  int i;
  for(i=0;i<UPPER;i++)
  printf("%d", i*2);
}
```


The program is first passed to the preprocessor. The preprocessor checks the presence of macros. It replaces all occurrences of `UPPER` in the program with 20. Substitutions are done only for separate whole names, not names within double quotes or if name is part of another name. i.e. replacement will not take place for "`UPPER`" or `UPPERCASE`.

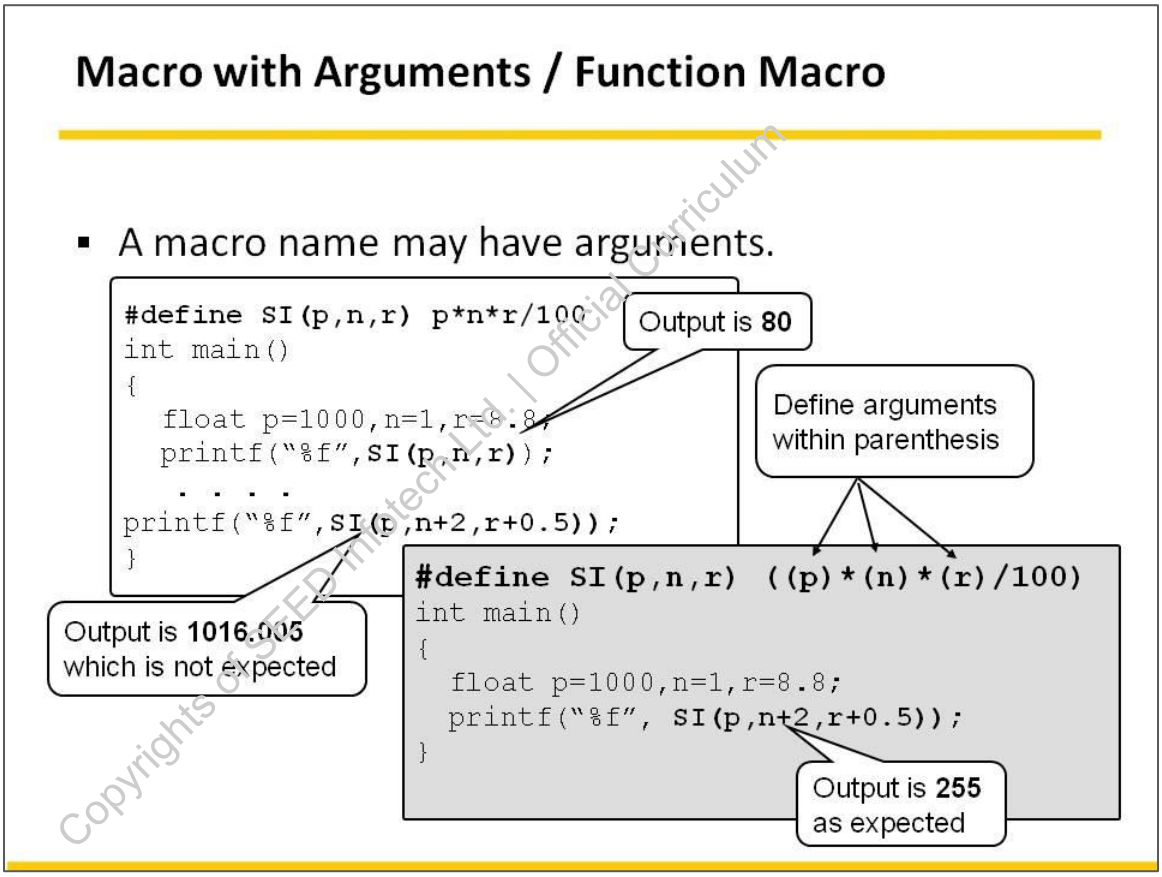
- Once a macro name has been defined, it may be used as part of the definition of other macro names.

```
#define ONE 1
#define TWO ONE + ONE
```

- If the string is longer than one line, it may be continued by placing a backslash at the end of the first line.

```
#define MAX(x,y,z)     if(x > y && x > z)\
  printf("\n First number is maximum.");\
                  else if(y > z)\
                     printf("\n Second number is
maximum.");\
                  else\
                     printf("\nThird number is
maximum.");
```

Constant names should be in all CAPS, with multiple words separated by an underscore. For example, RATE_OF_INT



Macro can also be used with arguments. Such a macro is called macro with arguments or simply function macro. It can be used with different types of arguments. Every time the macro name is encountered, the arguments associated with it are replaced by the actual arguments found in the program. No blank should be left between the macro template and its argument. The entire macro expansion should be enclosed in parentheses. For example,

```
#define AREA(l, b) ((l) * (b))
int main()
{
  int len = 2, breadth = 3;
  printf("Area is %d", AREA(len+2,breadth+3));
}
```

The output of this program is 24 as expected.

Consider the following macro definition. If this definition is used, then according to the rules of precedence and associativity for the same macro call above, the output of the program would be 1016.005

```
#define SI(p,n,r) p*n*r/100
```

Another example to demonstrate the use of different data types used with macro.

```
#define MAX(num1,num2) (num1 > num2 ? num1 : num2)
. . .
int res, no1=20, no2=24;
float result,f1,f2;
res = MAX(no1, no2);  // macro used with integer type
arguments
result = MAX(f1, f2); // macro used with float type
arguments
```

Function Vs Macro

	Function	Macro
Memory required	Less as only one copy exists.	More. Since inline code is produced.
Time required	More. Since the control shifts to called function.	Less. Due to inline expansion.
Data type	Considered by compiler for function invocation.	Not considered. Since text replacement takes place before compilation.

- Use macro for small code.
- Use function to implement a complex logic for a given task.

The differences between function and macro are as follows:

- Memory required for function is less since only one copy exists which is used by all calling blocks. In case of macro, text substitution takes place at the point where macro name is encountered. This increases the size of the code.
- Time required for execution is more for a function. With every function call, the control shifts to the called function and back to the calling function. Macros execute faster due to inline expansion.
- Compiler checks the data type of formal parameters for every function call. It gives an error for data type mismatch. Preprocessor does not consider the data type of the macro arguments.

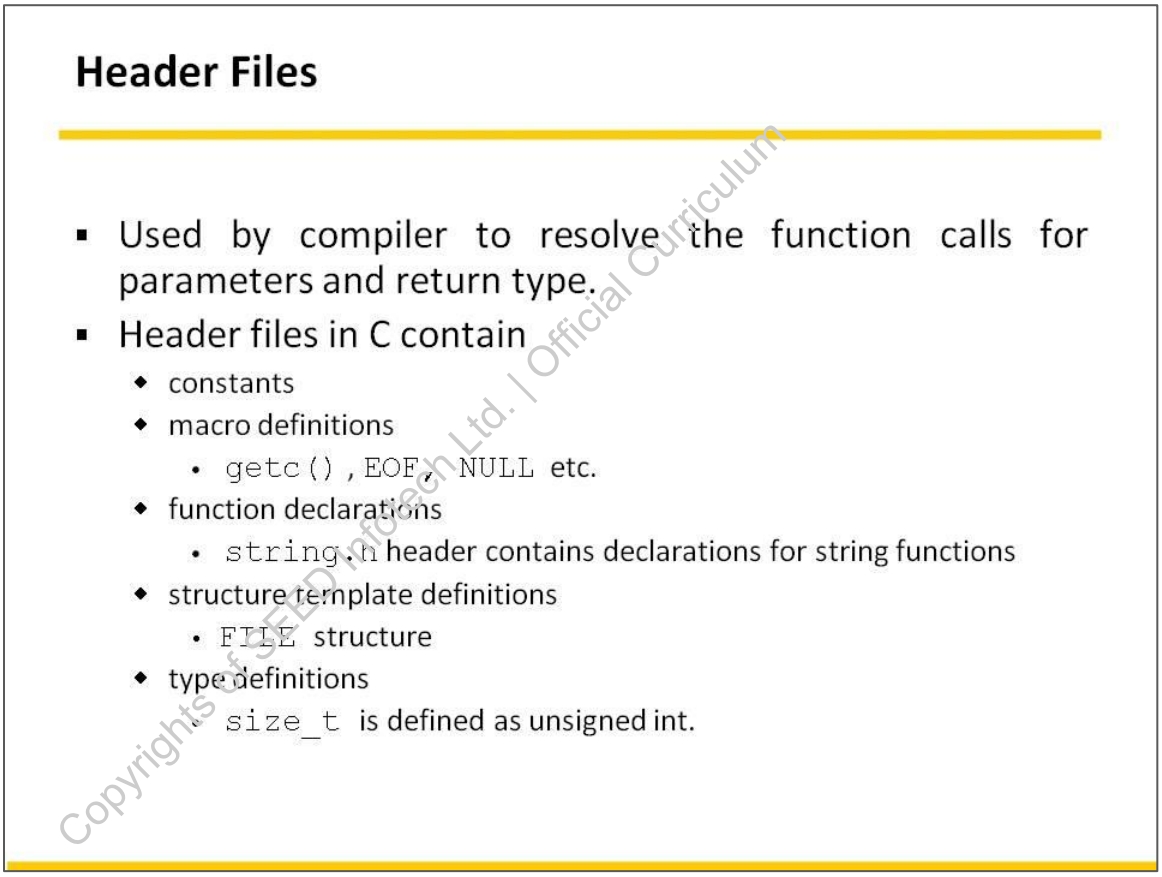
There are no rules for using a function or a macro. But there are some considerations that have to be made for the selection of a macro.

1. Macros are a good choice where the programmer has to operate on real or integer operands or a mixture of the two.

2. When the code for replacement is small, macros should be used for faster execution.
3. Macros are used to increase the readability by giving relevant and suitable names.
4. Another important use of macro is that if a programmer decides to change the value of `RATEOFINT` macro from 8.25 to 8.5, then the change would be required to be made in only one place.

Moreover, functions are used to implement a complex logic for a particular task. They play a vital role in modular approach of programming.

Prepare for comparison between a function and a macro.



Header files have `.h` extension and are used by the compiler to resolve function calls wherein it checks for the data type of formal parameters and the return type. Header files are present in `C:\TC\INCLUDE` directory.

Header files in C can contain the following features:

Constants

Constants can be defined in header files. For example, `stdio.h` file contains constants like `EOF`, `NULL`, etc.

Macro definitions

Macro definitions (simple or function macro) can be defined as a part of header files. For example, `getchar()` is defined in `stdio.h` file as `getc(stdin)`. `getc()` is defined as a complex macro.

Function declarations

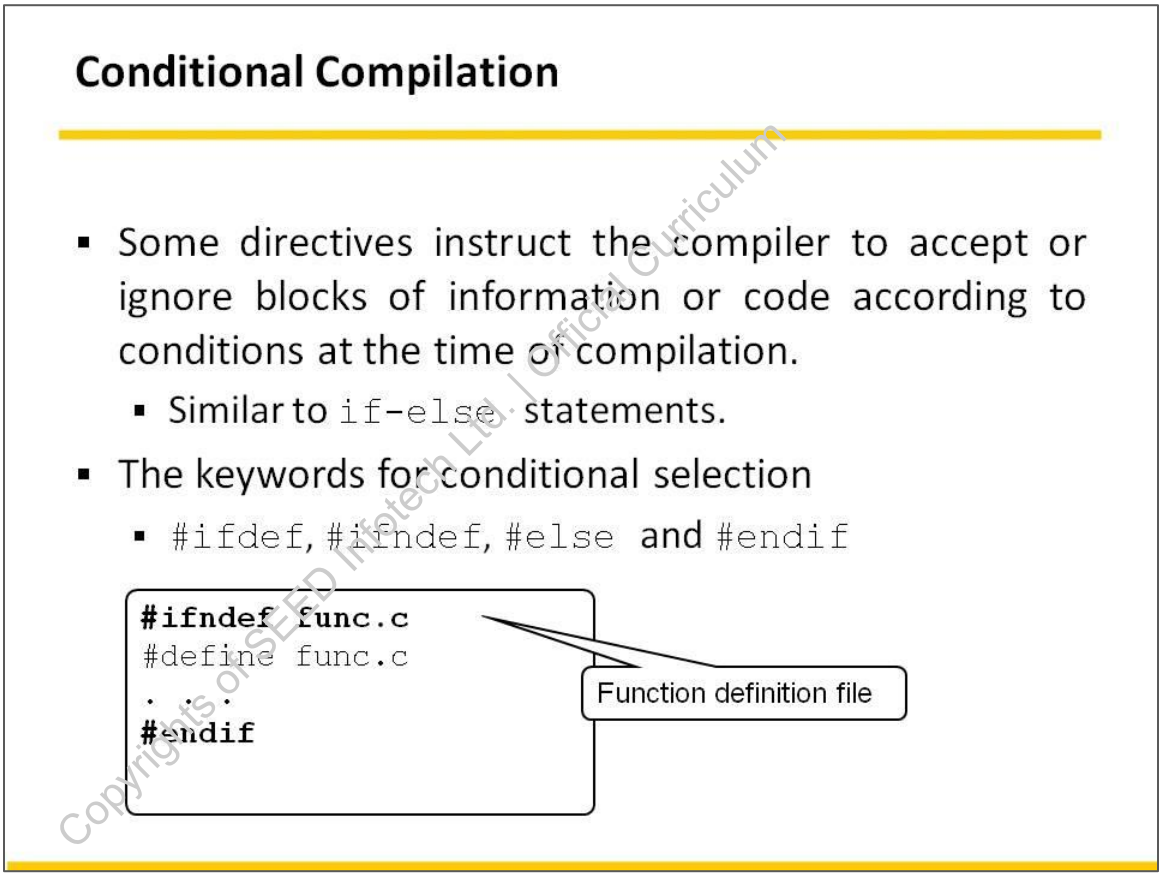
User defined function declarations are written in header files of an application. For example, `stdio.h` file contains function declarations for built-in I/O functions. Declaration of `exit()` function is found in `process.h` and `stdlib.h` files.

Structure template definitions

Structures can be defined in header files and used by other files in an application just by including the file containing the structure definition. `FILE` structure is defined in `stdio.h` file. This structure stores important information about a file and hence is used in FILE I/O operations.

Type definitions

Some `typedef` declarations can also be stored in header files. For example, `size_t` is defined as `unsigned int`.



Conditional compilation instructs the compiler to accept or ignore blocks of information or code according to conditions at the time of compilation. The following are conditional compilation directives: `#if`, `#elif`, `#else`, `#endif`, `#ifdef`, `#ifndef`, `#ifdef-#endif`

Lines of source code that are sometimes desired in the program and some other times not, are surrounded by `#ifdef-#endif` directive pairs like the one shown below.

```
int main()
{
  . . .
  #ifdef CHECK
     printf("debug: value before swap ---- x = %d, y
= %d\n", x, y);
    ... // code here
```

```
  #endif
  . . .
}
```


The `#ifdef` directive specifies that if `CHECK` exists as a defined macro, i.e. it is defined by means of a `#define` directive, then, the statements between the `#ifdef` directive and the `#endif` directive are passed to the compiler. If `CHECK` does not exist as a macro, then these statements are not passed on to the compiler. Thus to ''turn on'' debugging statements, define the macro.

```
#define CHECK
```

Remember to remove the macro definition after debugging. The advantage is that debug statements do not have to be physically tracked down and removed. Also, if a program needs modification, the debug statements are in place and can simply be reactivated.

#if-#else

`#else` works much like the 'C' keyword `else`. `#elif` means "else if" and establishes an `if-else-if` compilation chain. Amongst other things, `#if` provides an alternative method of "commenting out" code.

The general form of `#if` is:

```
#if constant_expression
  . . . statement sequence
#endif
```

If a program uses a variety of printers, a header file should be included in the program to support the use of a specific printer. Assume that the specific printer used in an installation is defined by a macro `DEVICE`. Conditional compilation directives can be then written to include the appropriate header file.

```
#if DEVICE == IBM
    #include ibmprn.h
```

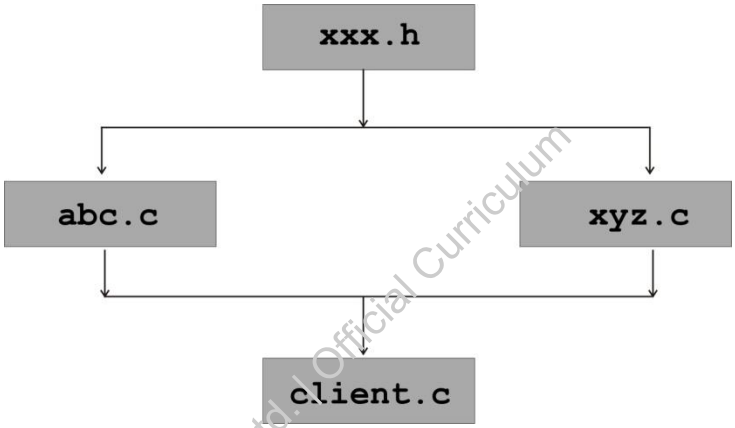
```
#elif DEVICE == HP
    #include hpprn.h
#else
    #include comprn.h
#endif
```


#ifndef-#endif

This directive prevents multiple inclusions of a particular file, thereby preventing conflicts. These conflicts occur when the preprocessor includes the same header file more than once, and the compiler detects that there are duplicate definitions of certain constants, macros or functions. To avoid this, the following code can be used in the header file itself.

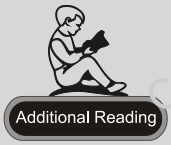
```
#ifndef _HEADER_H
#include "myHeader.h"
#define _HEADER_H
  void dispInt();
  float dispBal(long, . .);
 . . .
#endif
```

The `#ifndef` directive specifies that if `_HEADER_H` exists then the statements between the `#ifndef` directive and the `#endif` directive are passed to the compiler. If `_HEADER_H` does not exist, then these statements are not passed on to the compiler.



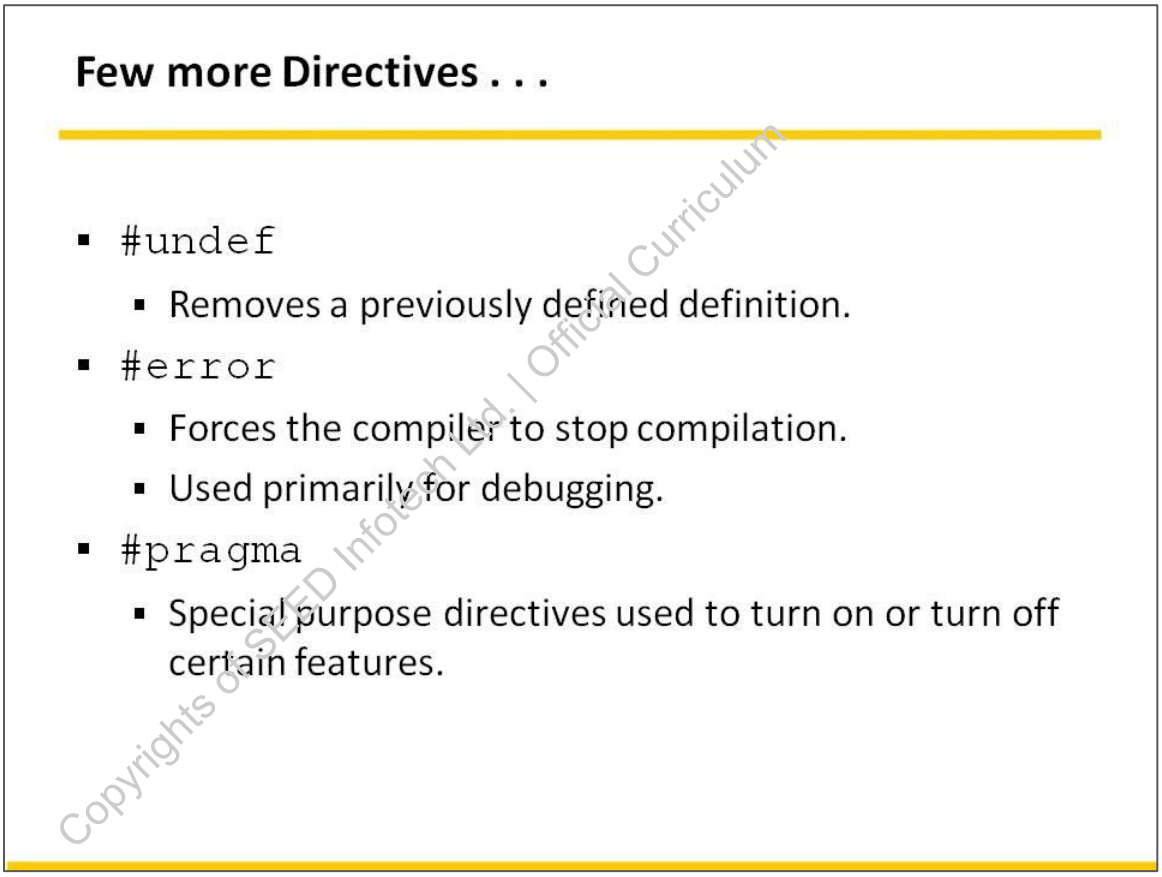
The declarations should be enclosed within the conditional compilation statement i.e. `#ifndef`, as shown above to avoid re-declaration errors.

All the functions are defined a `.h` file and the `main()` in other `.c` file.



Read about conditional compilation from C Primer – by Stephen Prata.

It is a good coding practice to use the conditional compilation statements in the code. User defined functions should be declared in a separate header file.



There are few more directives, which do not fit into the above categories. Hence they are described here separately.

#undef

The `#undef` directive "undefines" a given `#defined` macro.

For Example

```
#define SIZE 25
main()
{
   . . .
   #undef SIZE
   . . .
}
```

#error

`#error` forces the compiler to stop compilation. It is used primarily for debugging. The general form is:

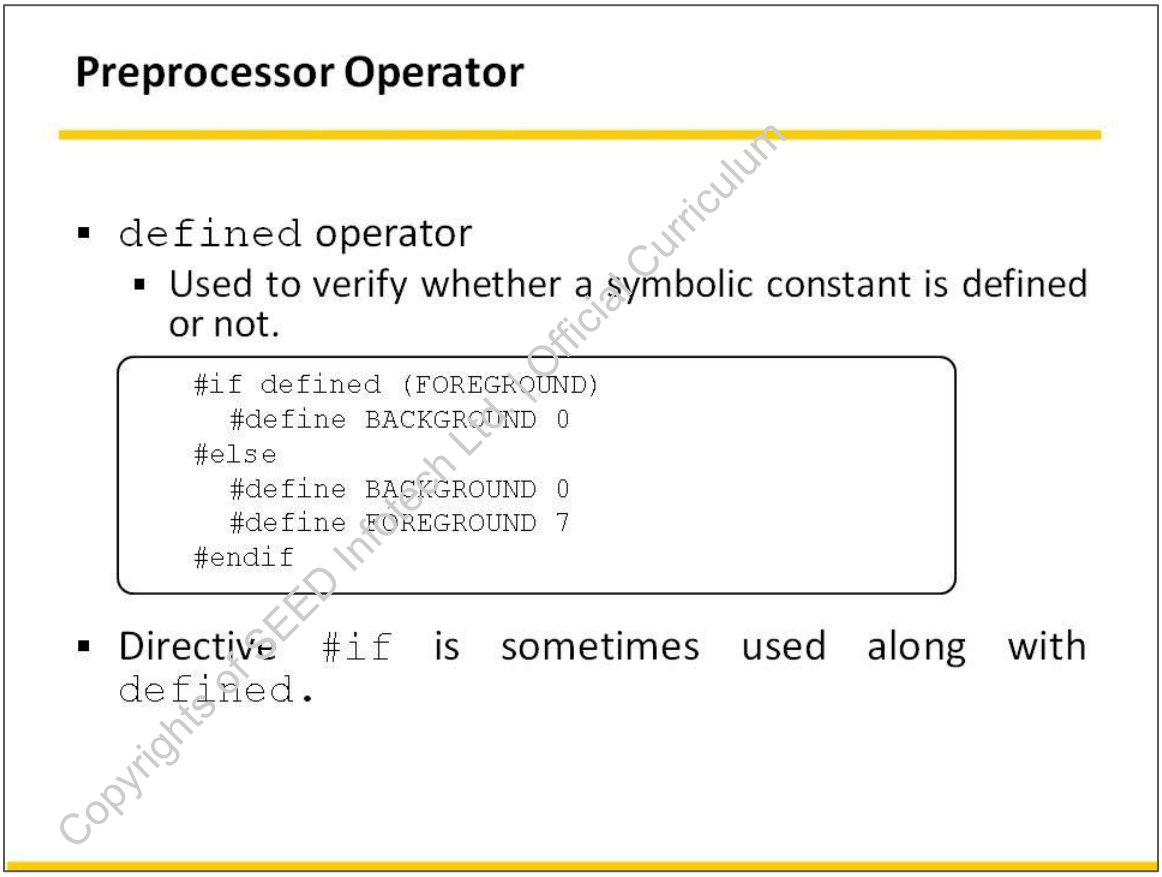
```
#error error_message
```

When the directive is encountered, the error message is displayed, possibly along with other information, depending on the compiler.

#pragma

These are the special purpose directives that are used to turn on or turn off certain features. The `#pragma` directive is an implementation-defined directive which allows various instructions to be given to the compiler.

Turbo 'C' Compiler has got a `pragma` which allows writing assembly languages statements in 'C' program.



Preprocessor provides an operator called as `defined` operator. It is used to verify if a symbolic constant is defined or not. `#if` is sometimes used along with `defined` operator.

`defined`(macro name)

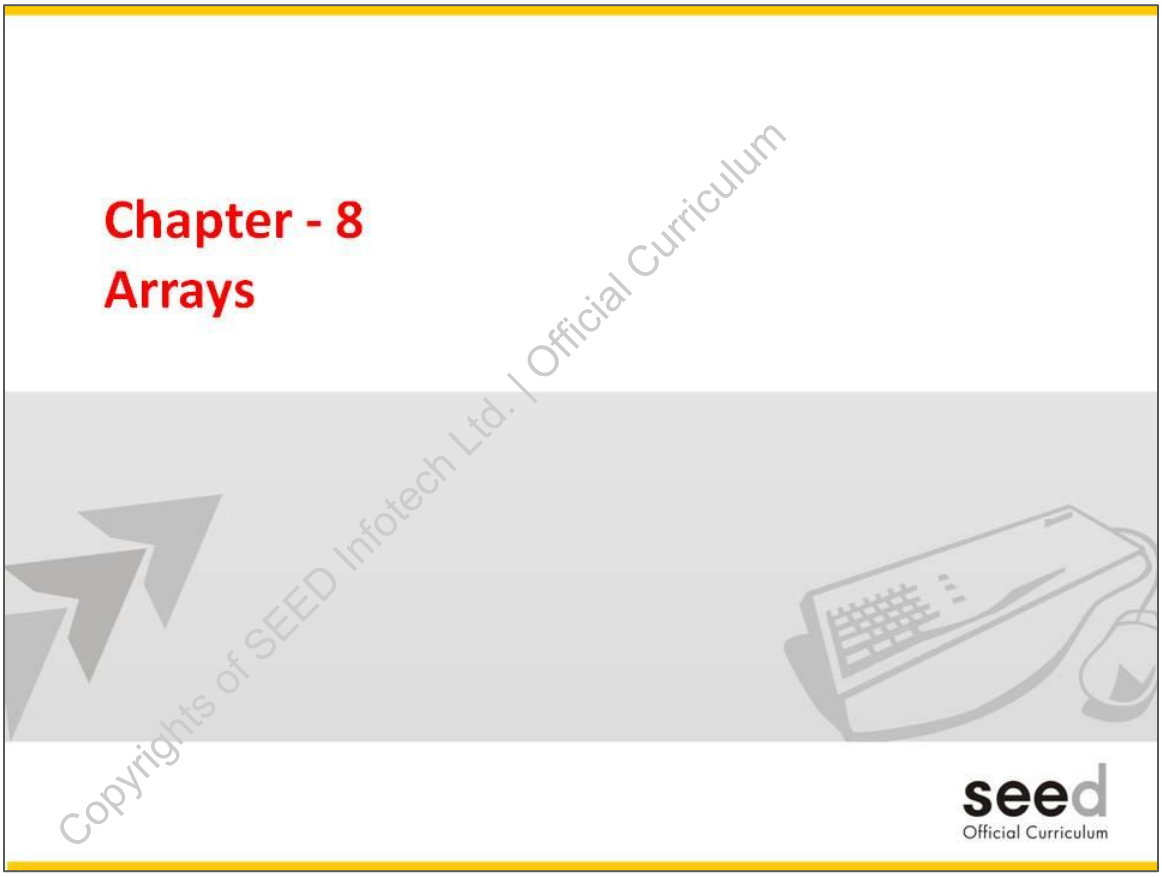
returns 1 if macro is defined.

returns 0 if macro is not defined.

For Example,

```
#if defined (FOREGROUND)
  #define BACKGROUND 0
#else
  #define BACKGROUND 0
  #define FOREGROUND 7
#endif
```

In tests on C, it is common to ask questions on preprocessor operators. Master these.

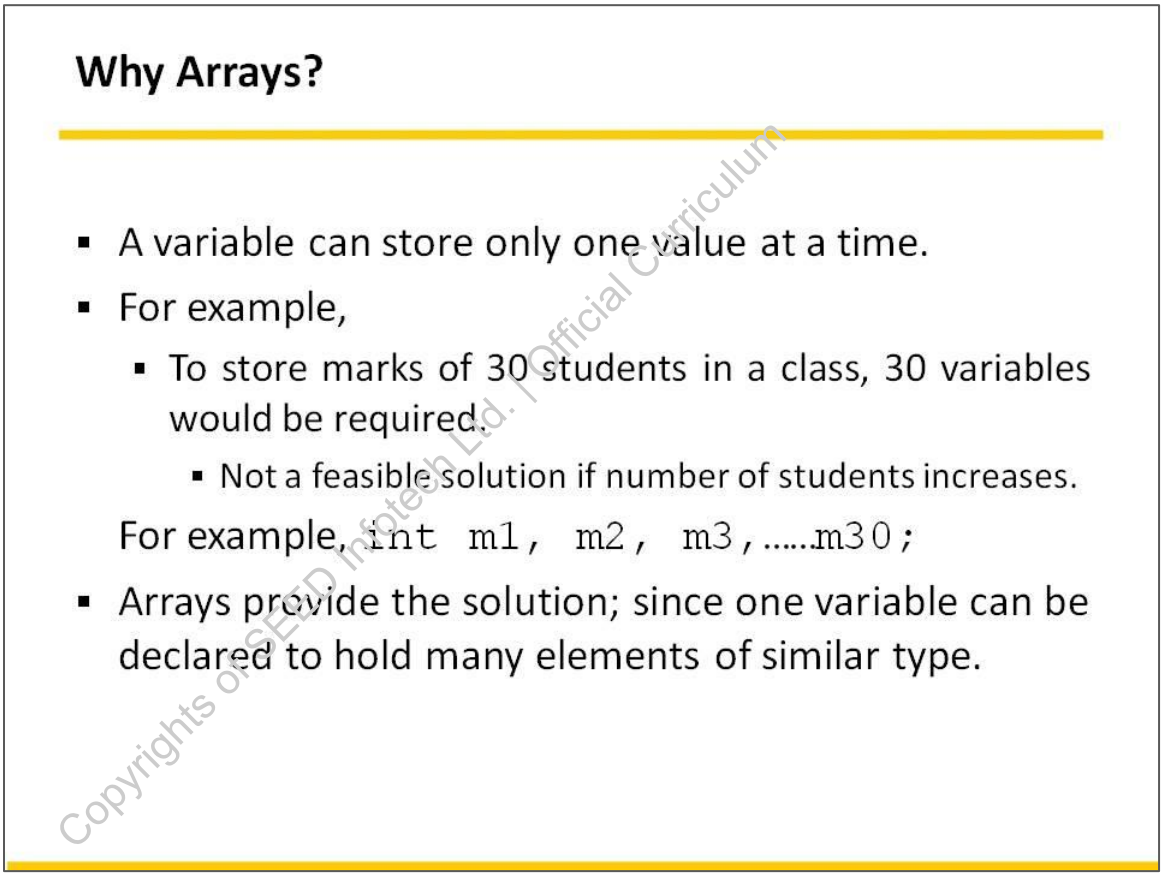


This chapter covers one dimensional array. It covers array declaration and initialization, and operations performed on an array. It also focuses on pointer arithmetic and different methods of passing an array to a function.

Objectives

At the end of this chapter you will be able to:

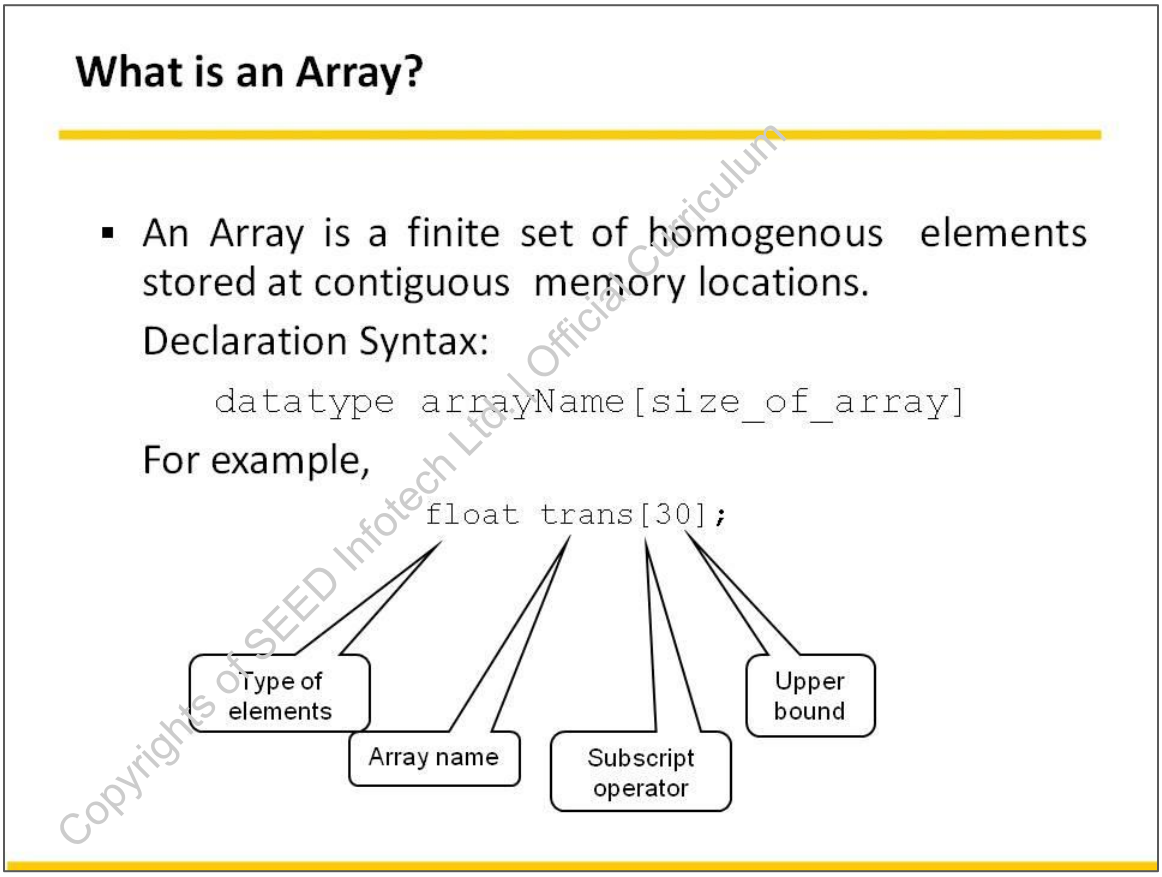
- Use array as a derived data type.
- Declare and initialize an array.
- Perform array I/O operations.
- Identify the need of bounds checking in arrays.
- Pass an array to function.
- State rules for pointer arithmetic.
- Perform array I/O operations using pointers.



Using variables is convenient when handling small amount of data. For example, finding average marks of a student in 5 subjects, or printing the largest of three numbers, and so on. But while handling a large volume of data like printing average marks of 30 of students, declaring 30 individual variables is not feasible. Array is an easy and simple solution to this problem. One array variable can be declared to hold many elements of similar data type.

Arrays can be used to create a list of

- Temperatures recorded every hour in a day, or a month, or a year.
- Names of employees in an organization.
- Names of products sold by a store.
- Test score of a class of students.
- Customer id and their telephone numbers and so on.



An array can be defined as a finite set of homogeneous elements stored at contiguous memory locations. Like other variables, an array needs to be declared before it is used, so the compiler would know the type of the array, and the number of elements that can be stored in it. Declaration syntax of an array is:

```
storage-class data-type array [expression];
```

- By default, storage class is automatic, if not included in the declaration.
- An array can store elements of any data type, but all elements in an array must be of the same type.
- Naming rules for an array are the same as that for any variable.
- The maximum number of elements that can be stored in the array needs to be specified in `[]` operator. It is also called the upper bound or the size of an array. It must be a positive integer.

Examples

```
int marks[30];     //marks is an array of 30 integer
elements
```

```
float temp[SIZE]; //float type elements, SIZE is a # defined constant
char name[80];    //name is an array of 80 character elements
```

Array definition and declaration

An array is defined like any other variable, except that it has a size-specifier which gives the size, i.e., the maximum number of elements that can be stored in the array.

The square bracket identifies it as an array. In the example above, an array has been defined as `float trans[30]`. The name of the array is `trans` and it is an array of 30 float values. Memory allocated to it will be 30 * 4 = 120 bytes.

If an array is defined in one file and used in a second file, it will be required to be declared in the second file as `extern`. In such a case of array declaration, size of the array is optional. If specified, it must match with the array definition. Initial values cannot be present in array declaration.


```
//File 1
main()
{
//array definition
int arr[10];
. . .
}
```

```
main()
{
// array declaration
extern arr[];
  . . .
}
```

Array Initialization

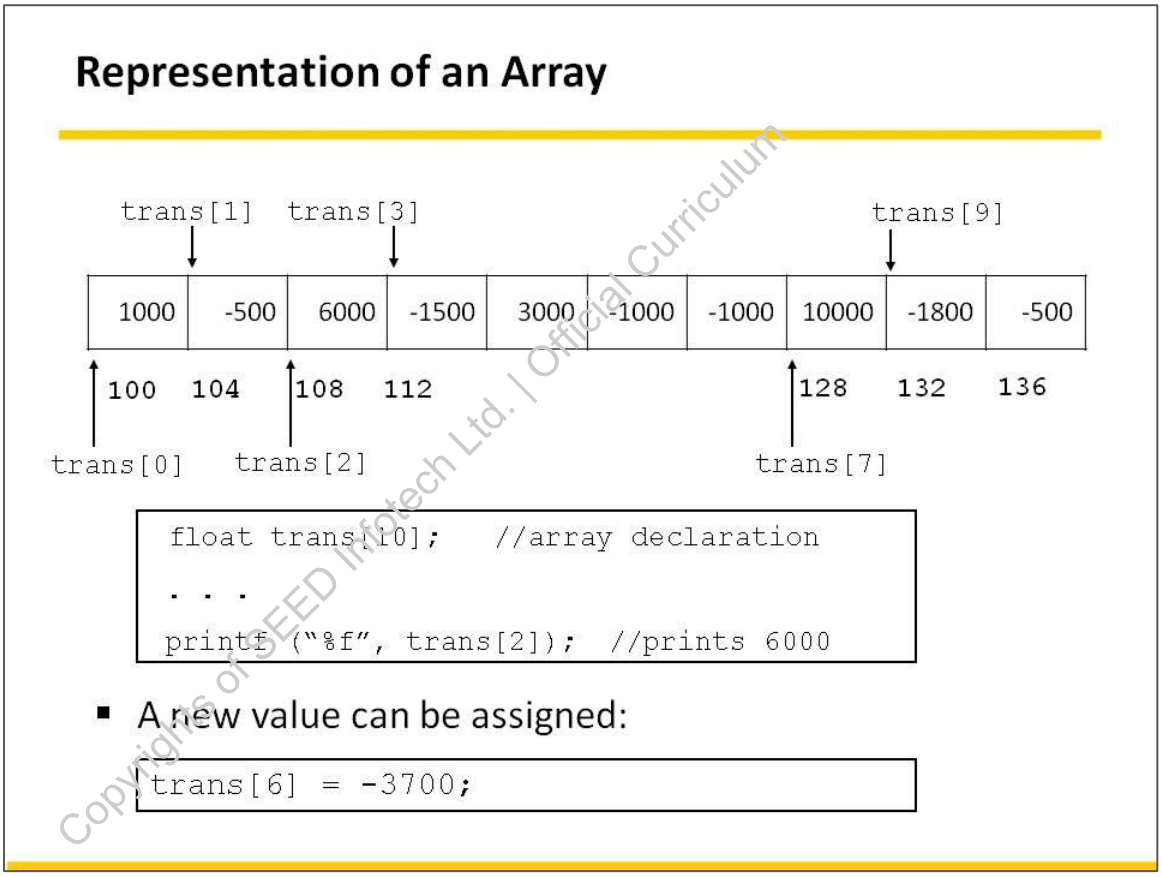
If an array is not initialized, it contains values depending on the storage class of the array. An array can be initialized as follows.

```
int months[12] =
{31,28,31,30,31,30,31,31,30,31,30,31};
float temperature[ ] = { 28.5f , 30.2f , 27f , 31f ,
32f };
```

Omitting the size of the array is allowed if the array is initialized. In such cases, the compiler allocates enough space for all initialized elements.

The values in the list are separated by commas. The initial values must appear in the order in which they will be assigned to the individual array elements. If partial initialization is done as the one done below, all the remaining elements are initialized to zero.

```
float total[5] = {0.0f,15.75f,3.45f};
```



The pictorial representation on the slide shows the memory representation of an array named `trans`. It is declared as:

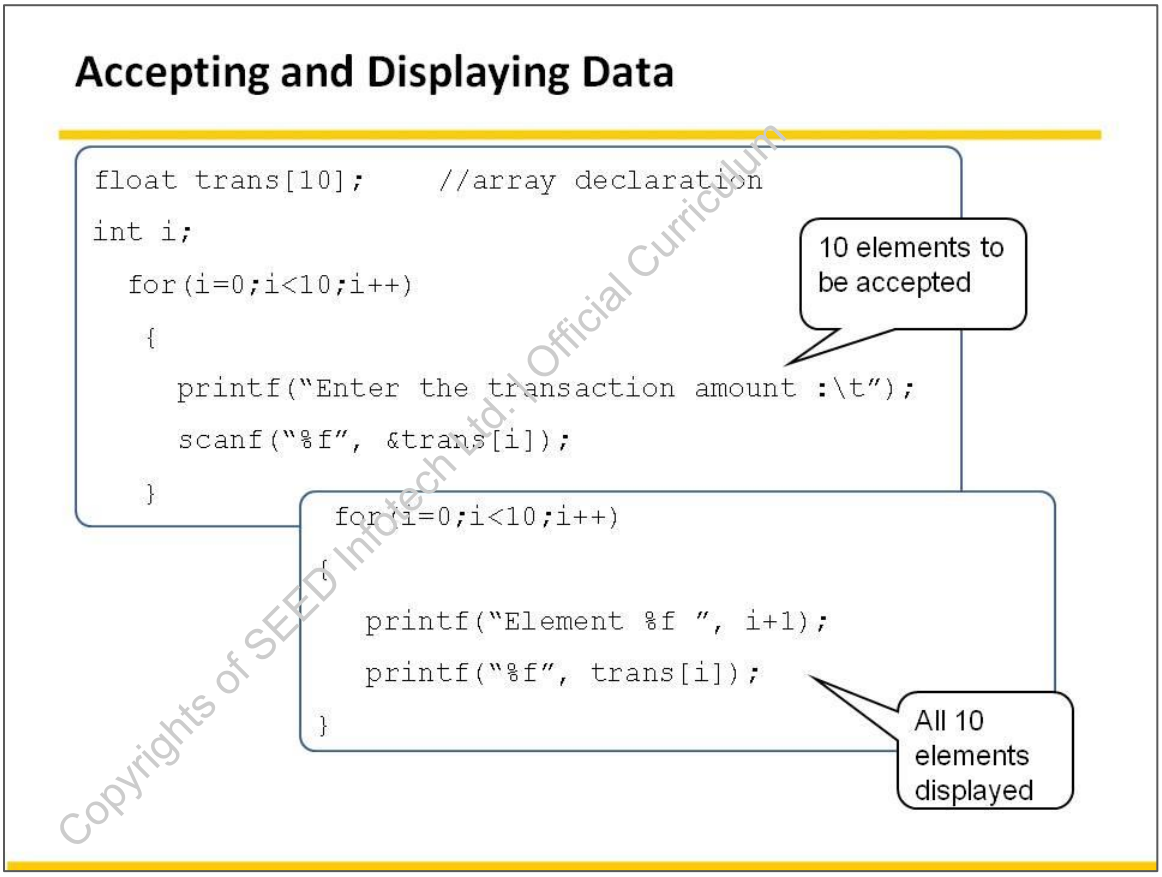
```
float trans[10];
```

- The elements are stored at contiguous memory locations.
- `trans[0]` is the first element of the array and `trans[9]` is the last element.
- The index of an array starts at 0. Therefore, the index of the last element of an array will be 1 less than its size. Every individual element of an array can be accessed using the index. This allows manipulating the array elements in different ways – altering values, printing, reading, and so on.
- An array position can be assigned a new value like the one below.

```
trans[7] = -3700;
```

- When an array is declared, enough space is allocated to hold all its elements. For example, the above statement, would allocate, 30 * 4 = 120 bytes to store all the float elements.
- The name of an array is the address of the array. It is also called the base address. Thus, `trans` is the base address of the array or the address of the first element of the array. The address of the first element can also be represented as `(trans+0)`. Since the elements are stored in contiguous memory locations, the next element will have address `(trans+1)`, and the one after that will have address `(trans+2)` and so on.
- Since the address of an array element can be represented in this manner, it can be de-referenced to get the value stored at that location. The de-referencing operator is used for this purpose. For example,

Address		**Value**	
Trans	100	* trans	1000
(trans + 0)	100	*(trans + 0)	1000
(trans + 1)	104	*(trans + 1)	-500
(trans + 2)	108	*(trans + 2)	6000



An array can be accepted from the user at runtime and displayed. This can be done using `for` loop as shown on the slide.

The same code can be written as follows:

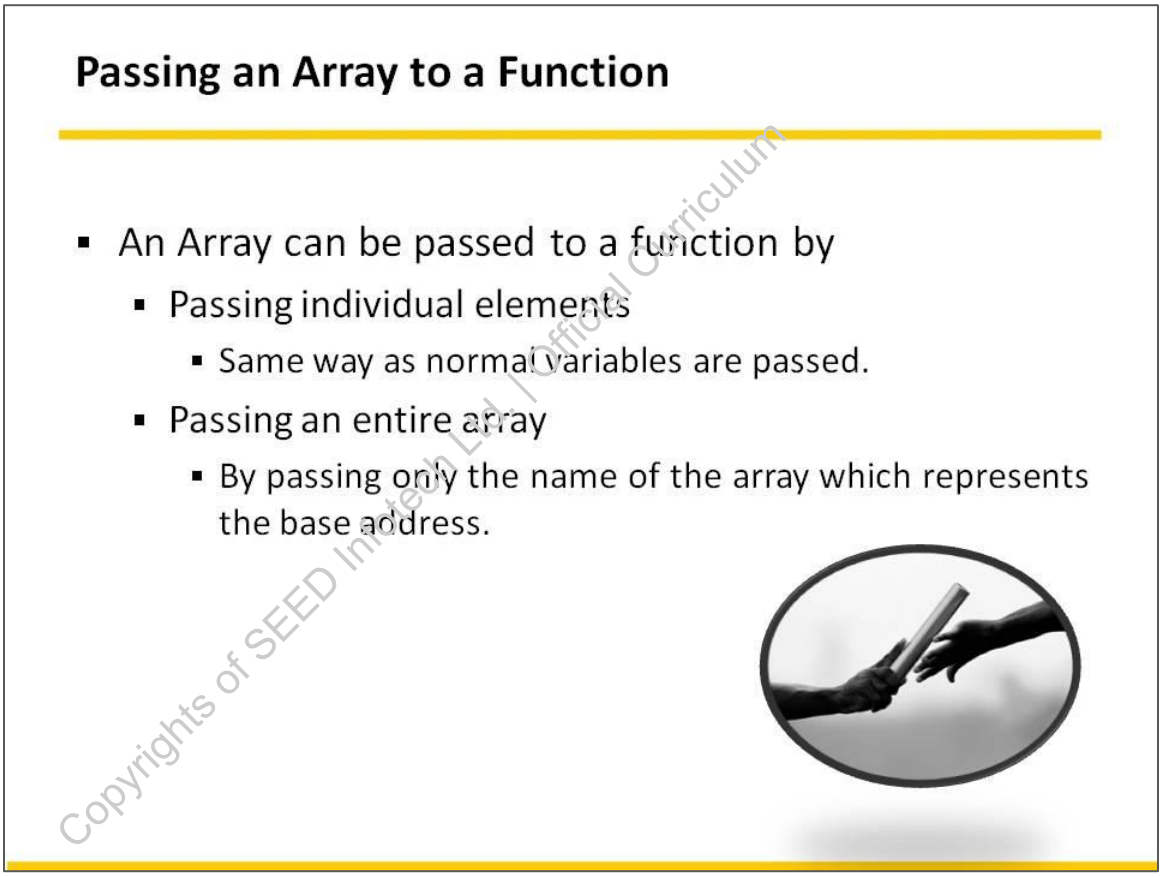
Code to accept an array

```
float trans[10];  //array declaration
int i;
for(i=0;i<10;i++)
{
    printf("Enter the transaction amount:\t");
    scanf("%f", (trans + i));
    //can be alternatively written as
    //scanf("%f", &trans[i]);
}
```

Code to display an array

```
for(i=0;i<10;i++)
{
   printf("Element %f ", i+1);
  printf("%f", *(trans + i));
}
```

Write code to stay within the arrays bounds whenever dealing with arrays.



Like normal variables, array variables can also be passed to a function. There are two ways in which this can be done –

1. Passing individual elements
2. Passing an entire array

Passing Individual Elements

The following example demonstrates how to pass individual array elements. This method is similar to passing normal variables to a function.

```
/* To
```

Passing an Entire Array

Passing an entire array to a function is done by passing the name of the array which is its base address. An array can be passed in two ways:

- Using the subscript (`[]`) operator
- Using a pointer

Using the Subscript ([]) Operator

```
float miniStmt(float [], long);
int main()
{
  float trans[10],bal;
  /*accept the transaction amounts to generate a mini
statement*/
  bal = calcBal(trans, acNo);
  . . .
}
float miniStmt(float temp[], long acNo)
{
  float bal;
  /* code to calculate the balance amount in the
account and return it*/
  . . .
  return bal;
}
```

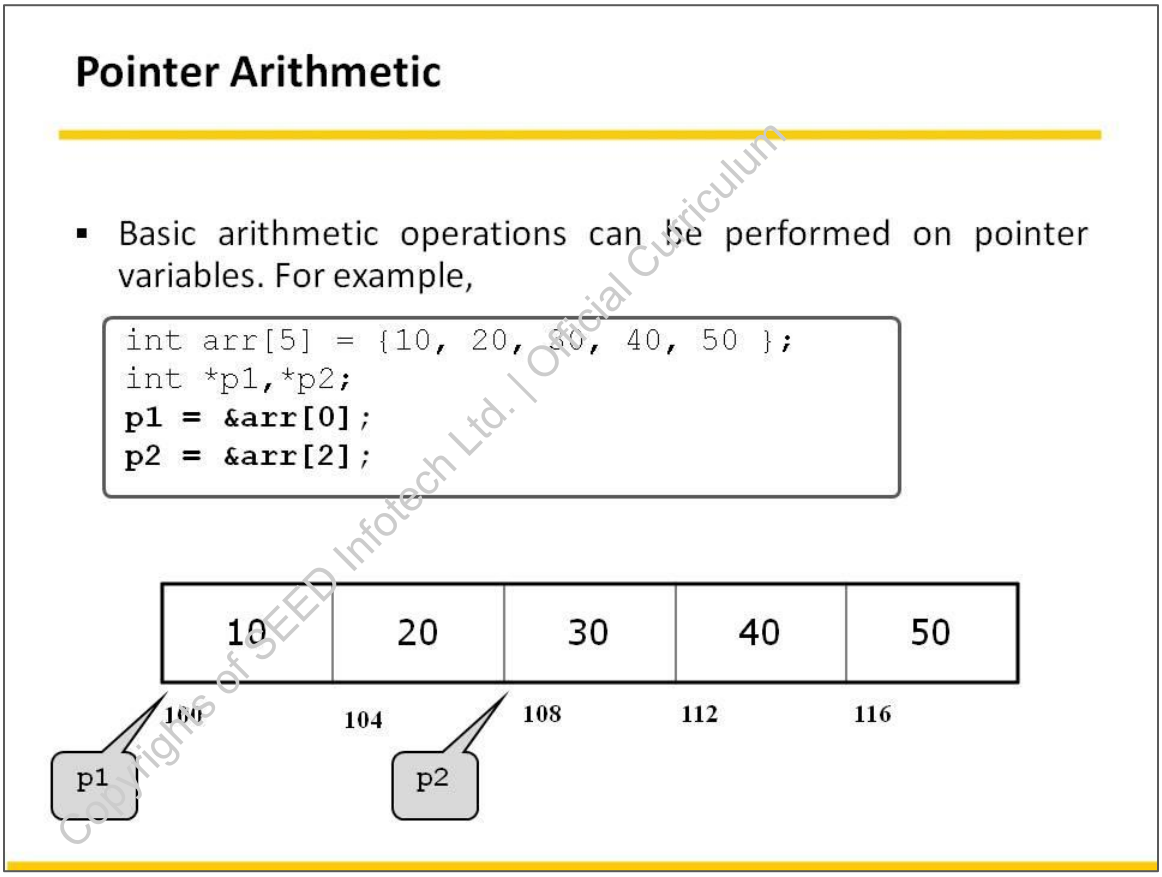

The above code snippet demonstrates the use of subscript [] operator in passing the entire array to the function. The subscript operator has been used in the function prototype and function definition. The function calculates the balance amount and returns it to the calling function.

Using a Pointer

```
float miniStmt( float *,long);
int main()
{
  float trans [10],bal;
  /* accept the transaction amounts to generate a
mini statement*/
```

```
  bal = calcBal(trans, acNo);
  . . .
}
float miniStmt(float *temp, long acNo)
{
  float bal;
  for(i=0; i < num; i++)
   /*  code to calculate the balance amount in the
account and return it*/
  . . .
  return bal;
}
```


An entire array can also be passed using pointers. The method is the same as using the subscript operator. The only difference is in the way the array is represented in the function prototype and definition. In this method, the formal parameter is a pointer '`float *`'.



C is consistent and regular in its approach to address arithmetic; its integration of pointers, arrays and address arithmetic is one of the strengths of the language.

Like arithmetic operations are performed on ordinary variables, they can also be performed on pointers. But, to perform these operations, the pointers must point to elements of the same array. It is to be kept in mind that a pointer is a variable but an array name is not a variable, hence the assignment, increment, decrement operations can be done on pointers but not on array names.

Consider the example mentioned on the slide. `p1` and `p2` are two pointers pointing to first and third elements of type integer respectively.

- A pointer variable can be assigned the value of another pointer variable provided both pointers point to objects of the same data type. After the assignment, both pointers point to the same element.

```
p2 = p1;
```

- An integer value can be added to or subtracted from a pointer variable, but the resulting expression must be interpreted very carefully.

```
p1 = p1 + 2;  // moves the pointer two locations
forward
p2 = p2 – 2;  //moves the pointer two locations
backward
```

- Pointers can be incremented as well as decremented using pre and post unary operators.

```
p1++; //will point to one location ahead of the
current location
p1--; //will point to one location behind the current
location
```

- One pointer can be subtracted from the other. The result is the number of elements between corresponding array elements.

```
int howFar = p2 – p1; // gives an integer value
depending on how
                          //many locations they are
apart
```

- Comparison of two pointer variables is possible provided both point to the elements of the same array. Pointers can also be compared with NULL.

```
if(p1 == p2)  //whether they are pointing to same
location
    printf("\nThe two pointers point to same
location. ");
else
   printf("\nThe two pointers do not point to same
location. ");
```

The following operations are not allowed on pointers.

1. Addition of two pointers
2. Multiplication of a pointer with a number
3. Dividing a pointer with a number

Practice writing programs using pointers to manipulate arrays.

More about Arrays

- Size of array can be a #defined constant

```
#define SIZE 40
 . . .
int arr[SIZE]
```

- Compiler does not perform the bound checking of an array.
- const can be used in the declaration to make the array read–only.

```
const int powers[5] = { 1, 4, 9, 16, 20 };
```

- If an array is initialized partially, the remaining elements are set to zero.
- The sizeof operator gives the size in bytes, of the object or type provided to it in parentheses.

Following are some additional points to be noted about arrays:

- The size of an array is a constant value and must be given at compile time. It can also be defined using a macro.

```
#define SIZE 40
. . .
int arr[SIZE];
```

- The compiler does not perform bound checking and gives no warning if the upper bound is exceeded. Bound checking has to be done explicitly.

1. For compile time initialization – the error displayed is “too many initializers”, if more values than the size of array are assigned.

```
int arr[5] = {1,2,3,4,5,6,7};      //too many
initializers
```

2. For run time initialization – C does not warn the user when an array subscript exceeds the size of the array. Data entered with too large a subscript will simply be placed in memory outside the array. May be, even above the existing data or above the program itself. This will lead to unpredictable results.

```
int arr[5];
for(i=0;i<7;i++)  //gives no warning about exceeding
the bound
  scanf("%d", &arr[i]);
```

- To make an array read-only, it can be declared as a constant. For example,

```
const int squares[5] = { 1, 4, 9, 16, 20 };
```

- If an array is initialized partially, the remaining elements are set to zero.

The **sizeof** operator is used to find the size in bytes of the data type provided to it in parentheses. For example,

```
int rollNo;
int arrNum[10];
int sizeOfInt = sizeof(rollNo);   // will give 4
bytes
int sizeOfArray = sizeof(arrNum); //will give 40
bytes
```

Basic ‘C’ Programming

Lab Manual and Appendix

Contents

Introduction to Lab Manual

C lab manual aims at giving complete understanding of the core concepts of the topic and applying them in the exercises.

The lab manual consists of a set of lab exercises defined chapter wise. Each exercise has a definite objective defined. These objectives map with the terminal objectives defined at the beginning of each chapter. The problem statement is defined with clear instructions. Advanced lab exercises are given for extra practice.

Appendix given at the end of lab manual contains stepwise instructions to use Visual Studio IDE.

Configuration

Visual Studio 6.0 is a sophisticated IDE as compared to IDEs like Turbo C or Borland C. It has a user friendly interface as compared to the earlier IDEs. It has been therefore decided to use Visual Studio 6.0 IDE in the lab. The lab exercises should be coded using C language.

Structure of Lab Manual

Lab manual consists of different sections. The explanation of each structure is given below.

Objective

It states what you will achieve after completing a particular application. At the end of each lab exercise you should keep a track whether the objectives of that session are achieved.

Problem Statement

Problem statement for each lab exercise is given. It defines clear instructions to achieve the defined objective.

How to use the Lab Manual?

Whenever a programmer has to transform a problem statement into a program which a computer can execute, you should split the activity in the following manner:

- Read the problem statement carefully. Hint is given for some problem statements to help you to solve the problem.
- Preparation
  - Write the algorithm or steps to be followed to solve the problem. You can also draw flowcharts if required.
  - The program is made modular by writing functions. So pen down expected function prototypes and arguments on paper.
  - Also decide how one module (function) would communicate with other module (function).
- Give a dry run to the algorithm written.
- Write the code.
- Execute the code.

Coding Practices

The maintenance of code is easy if the code is written using coding practices. You should follow following coding practices while solving the lab exercises in this lab manual:

- Use meaningful names for variables, functions, file.
- Use uniform notation throughout your code. For example, camel case notation as used in the courseware.
- Code should be properly indented.
- Code should be commented. While documenting your code, write the purpose of your piece of code. What task is assigned to a method, what arguments are passed to it, what it returns should be clearly stated. Write a clear comment if you have added any statements for testing purpose.

Writing right kind of well-documented software is an art along with your technical skills. Enough necessary documentation should be done. This makes the code easily maintainable.

Chapter 1 - Programming Techniques

Lab Exercise 1

Objective

- To get familiar with Visual Studio IDE.

Problem Statement

Get familiar with Visual Studio IDE demo

1. Components of Integrated Development Environment.
2. Use of Visual Studio for editing, compiling, debugging features.
3. Getting familiar with menus and relevant sub menus
4. **File**: New, Open, Close, Open Workspace.
5. **Edit**: Undo, Redo, Cut, Copy, Paste, Delete, Breakpoint.
3. **View**: Workspace, Output, Debug Windows.
4. **Project**: Set Active Project, Settings.
5. **Build**: Compile, Build, Rebuild All, Start Debug
6. **Debug**: Step Into, Step Over, Step Out, Run to Cursor
7. **Compile**: Compile, Link, Information, Remove messages
8. **Tools**: Options →Format
9. **Help**: Using MSDN help
6. Demonstration of a simple C program that prints “Hello World” on screen by the lab faculty. Conversion of `.c` to `.exe` should be demonstrated.

Lab Exercise 2

Objective

- To write algorithms for the given problem statements.

Problem Statements

1. Find the area of a circle whose radius is accepted from the user.
2. Find the maximum of two numbers and display it.
3. Find whether a given number is odd or even and accordingly display the message “Odd” or “Even”.
4. Find whether the number accepted from the user is positive or negative.

5. Accept the age from the user and display appropriate message for issuing of license:
6. Less than 18 years – Display “No license”.
7. Above 18 years – Display “Issue license”.

- Print multiples of 5 that are less than 100.
- Find the sum of even numbers from 1 to 20.
- Find the factorial of a number.
- To find whether the number accepted from the user is a prime number or not.
- Consider the number is 3425. Find the sum of the digits of the number and display the sum.

Chapter 2 - C Language Basics

Lab Exercise 3

Objectives

- Use escape sequences
- Use formatted input/output functions

Problem Statement

Write a program to print "`WELCOME TO SEED`" using escape sequences: `\n, \t, \r, \"`. Observe the differences in the output.

Note: Do not use all the escape sequences only at the beginning or at the end of a sentence. Their real purpose will be clear only if you try using them elsewhere in the statement.

Lab Exercise 4

Objective

- Use expressions and type casting

Problem Statement

Write a program to accept marks of 5 subjects from the user and calculate their average. Use implicit and explicit type conversion.

Lab Exercise 5

Objectives

- Validating input
- Use formatted output

Problem Statement

Write a program to calculate the sum of digits of a user-entered number. Ensure that your program scans not more than 4 digits. Also print the output in following manner.

For example, if the number is `1234`, output should be `1000+200+30+4`.

Advanced Exercises

1. Write a program to add two numbers and store the result in a third variable. Print the result.
2. Write a program to swap two variables using a third variable and without using third variable.
3. Write a program to calculate Net Salary of an employee. Accept Basic Salary (BS) from the user.

- HRA is 20% of BS.
- DA is 40% of BS.
- PF is 10 % of Gross Salary.
- Gross Salary is BS + HRA + DA.

Note: Net Salary = Gross Salary – PF

4. Accept a character from the user. It may be an alphabet, digit or any other character. Print its ASCII value.
5. Print the following table for given data types. Student is expected to print the correct information for size, range and format specifier columns.

DATA TYPES	SIZE	RANGE	FORMAT SPECIFIER
char	← Print relevant information in these columns →		
int			
float			
double			
long int			
long double			

Chapter 3 - Selection Statements

Lab Exercise 6

Objectives

- Use decision control statements
- Use logical, relational and conditional operators

Problem Statement

Write a program to find leap year by using `if-else` and logical operators (&& and ||).

(A Leap year is divisible by 4 and is not divisible by 100 but could be divisible by 400)

Problem Statement

Write a program to find leap year by using conditional or the ternary operator (? :).

(A Leap year is divisible by 4 and is not divisible by 100 but could be divisible by 400)

Lab Exercise 7

Objective

- Use nested `if-else` construct

Problem Statement

1. Write a program to accept the basic salary and total sales amount for a salesperson and calculate commission according to the sales amount. Display the net salary and the commission earned. The range is as follows –

Sales Amount in Rs.	Commission (%) on Sales
5,000 to 7,500	3

7,501 to 10,500	8
10,501 to 15,000	11
15,000 and above	15

Lab Exercise 8

Objectives

- Use nested `switch-case` construct
- Use `break` statement

Problem Statement

Using a switch-case construct, write a menu driven program to perform basic calculations (addition, subtraction, multiplication and division) on two numbers accepted from the user.

Advanced Exercises

1. Write a program to find maximum of 3 numbers using conditional operator.
2. Enter date in dd/mm/yy format. Write a program to print total number of days in a month and the month in words. For example, if date is entered as 23/07/2009 then display the message as - July has 31 days. Your program should take care of leap years.
3. Accept a number from user and find its absolute value. Absolute value is positive value. Use conditional operator.
4. Write a program to convert user entered character to its opposite case. Program should also flash an error message if the character entered by the user is not an alphabet.
5. Write a program to accept an `empID` and `deptNo` as numeric data and `desigCode` as character data. Display the entered data with proper messages.

Department No.	Department Name	Designation Code	Designation
10	Purchase	'M'	Manager
20	Sales	'A'	Analyst
30	Production	'W'	Worker
40	Marketing	'S'	Sales-person
50	Accounts	'C'	Clerk

e.g. If `empID = 101`, `desigCode = 'M'` and `deptCode = 10` then the message displayed should be like this:– "Employee with `empID 101 works in "Purchase" department as a "Manager"`.

Chapter 4 - LOOPS

Lab Exercise 9

Objective

- Use a loop

Problem Statement 1

Write a program to display ASCII characters in the range (0-255). The display should pause after displaying every 10 characters.

Problem Statement 2

Write a program to print whether a user entered number is an Armstrong number. Armstrong number is one which the sum of the cube of all its digits is same as the number. For example, 153=(1*1*1)+(5*5*5)+(3*3*3)

Lab Exercise 10

Objective

- Use a loop and `break`.

Problem Statement

Write a program to display whether a user entered number is prime number or not.

Lab Exercise 11

Objective

- Use `for` loop

Problem Statement

Modify Lab Exercise 2 to display all Armstrong numbers in the range 0 to 1000. Use `for` loop.

Lab Exercise 12

Objective

- Use `while` loop

Problem Statement

Modify assignment 4 in Lab 4 to display first 25 prime numbers (2, 3, 5, …97) .

Lab Exercise 13

Objective

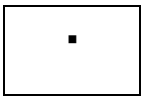
- Use nested `for` loop

Problem Statement 1

Write a program to find Pythagorean triplets in the range 0 to 100. For example, $3^2 + 4^2 = 5^2$. None of the numbers should repeat.

Problem Statement 2

Draw a rectangle for user-defined dimensions using for loop. Make use of extended ASCII characters. It should look like this:



Use the following characters to draw the rectangle:

-
- ▪ ┌ 218
- ▪ ┘ 217
- ▪ ┐ 191
- ▪ └ 192
- ▪ ─ 196
- ▪ │ 179

Lab Exercise 14

Objective

- Use nested `do-while` loop

Problem Statement

Modify Lab Exercise 3 in Chapter 3 to display the menu until the user desires to continue.

Advanced Exercises

1. Write a program to fill entire screen with smiling faces. (ASCII value = 1). Press any key to terminate. Hint: use `kbhit()`.
2. Write a program to generate all possible combinations of 1, 2, 3 using `for` loop.
3. Write a program for a matchstick game between the computer and a user. Your program should ensure that the computer always wins. Rules for the game are as follows:-
   a. There are 21 matchsticks.
   b. The computer asks the player to pick 1, 2, 3, or 4 matchsticks.
   c. The player is given the chance to pick the sticks first then the computer picks the sticks.
   d. Whoever is forced to pick up the last matchstick loses the game.
4. Write a program to generate the following output:

		1	1		
	2	1	1	2	
3	2	1	1	2	3

Chapter 5 - Functions

Note: The programs should be implemented as multi-file programs. Function declarations should be in a header file. Implementation of functions should be in a source file (.c). Client code (main()) should be in a separate source file(.c). Include the necessary files wherever required.

Lab Exercise 15

Objectives

- Implement multi-file C programs.
- Declare, call and define a function.
- Pass the parameters by value.

Program Statement 1

Write a program to generate the following table:

Data type	Size in bytes
Integer	2
Character	1
Float	4

The border of the above table should be generated using a function `charLine()`. This function accepts a character and the length of the line to be printed, entered by the user in `main()`. Use the character to print the line.

Program Statement 2

Write a function `fibonacci()` to generate the terms of the Fibonacci series. The number of terms should be entered by the user in `main()`. The series should be printed in the function. The terms of the Fibonacci series are 0 1 1 2 3 5 8 13 21 . . .

Lab Exercise 16

Objective

- Use `return` statement

Program Statement

Accept a number from the user and pass it to a function called `check()`. This function returns the status of the number as either positive or negative. As long as the number is negative the user should be asked to re-enter the number. Otherwise the number should be passed as a parameter to `prime()` function. This function accepts a number as a parameter and checks whether the number is prime or not. This message should be printed in `main()`.

Lab Exercise 17

Objective

- Pass parameters by address

Program Statement 1

Write a menu driven program, which allows the user to select either a circle or a rectangle.

a. Write a single function `areaCircum()` that calculates the area and circumference of a circle. The values of area and circumference should be printed in `main()`.
b. Write a single function `areaPeri()` that calculates the area and perimeter of a rectangle. The values of area and perimeter should be printed in `main()`.

Program Statement 2

Write a function `int power(int base, int index, int *result)`, which calculates the power of a user entered positive base and index passed to it from `main()`. Trap the 'overflow' error if any, in this function, and return the error status. Display the result or error status in `main()`. If the value of result exceeds the range of its data type then we say that overflow has occurred. In this case, consider overflow if the result is negative.

Program Statement 3

Write a program to accept date, month, year from the user in a function called `getDate()`, and print that date in `main()` in dd/mm/yy format.

Lab Exercise 18

Objectives

- Write recursive function.

Problem Statement 1

Implement function `fibonacci()` to make it recursive. Call the function from `main()`.

Problem Statement 2

Write a recursive function, `digSum()`, to find the sum of digits of an integer number.

Advanced Exercises

1. Accept a character in `main()` and pass it to a function called `convert()`. If this character is an alphabet, convert it to opposite case and print it in `main()`. If this character is not an alphabet, then print an error message in `main()`.
2. Write a function equivalent to the library function `sqrt()`, to find square root of a user entered positive number. Test the function in `main()`.
3. Accept a positive number in `main()` and call a function that calculates the sum of the digits of that number. The function should return the value to `main()`.

4. Find the sine value of an angle by calculating the following series up to first 5 terms

   $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + x^9/9! \ldots.$

   Where, x is in radians.

   $\text{Radian} = 180^0/PI$

   $PI = 3.14$

   Make use of the following functions:

   `int factorial(int);` and `int power(int, int);`

5. Write a program which accepts two integers in `main()`. Pass these as arguments to function `max()`. In `max()`, find the maximum of these two numbers and using a return statement, return the address of the maximum number and print it in `main()`.
6. A circle C can be completely specified by the co-ordinates of its center (x, y) and the radius r. Given two circles C1 and C2, test the following in a function:
   a. Does C1 intersect C2?
   b. Does C1 touch C2, if so whether internally or externally?
   c. Does Circle C1 completely embed circle C2?
7. Write a recursive function to calculate factorial of a user entered number using pointers.

Chapter 6 - Storage Classes

Lab Exercise 19

Objective

- Implement storage classes.

Problem Statement

Write a function `printLine()` that prints a dashed line. This function is called from main to create a table. Print the number of times the function is called from `main()`.

Hint: Use a `static` variable.

Chapter 7 - Preprocessor

Lab Exercise 20

Objective

- Use preprocessor directives

Problem Statement 1

Write a program to implement a macro to find the maximum of 3 numbers.

Problem Statement 2

Write macro definitions with arguments for calculation of area and perimeter of a triangle, a square and a circle. Store these macro definitions in a file called "`areaPeri.h`". Include this file in your program and call the macro definitions for calculating area and perimeter for a square, triangle and a circle.

Problem Statement 3

Write a program to test whether a user entered character is an alphabet or not by using macro.

Lab Exercise 21

Objective

- Know the difference between macro-substitution and functions.

Problem Statement

Write macro definitions with arguments for calculation of area and perimeter of a triangle, a square and a circle. Store these macro definitions in a file called "`areaPeri.h`". Include this file in your program and call the macro definitions for calculating area and perimeter for a square, triangle and a circle.

Advanced Exercises

1. Write a program using macros to –
   a. Find the square root of a number
   b. Print whether a given number is odd or even.

Chapter 8 - Arrays

Lab Exercise 22

Objectives

- To declare and initialize arrays.

Problem Statement

Write a program to calculate and display the average marks of 5 subjects obtained by a student.

Lab Exercise 23

Objective

- Use pointers to pass the entire array to a function

Problem Statement

Modify Lab Exercise 1 by writing separate functions to accept and display their average.

Lab Exercise 24

Objective

- Pass entire array to a function

Problem Statement

Accept five integers in an array and use two separate functions to:

a. Find the maximum and minimum of the integers. Do not sort the array.
b. Multiply each element of the array by 5 and store it in another array. Then display the new array.

Advanced Exercises

1. Write a program that defines a float array and accepts elements from the user. Interchange these elements at all consecutive even and odd positions

and display the original and the modified array. 1st with 2nd and 3rd with 4th and so on.

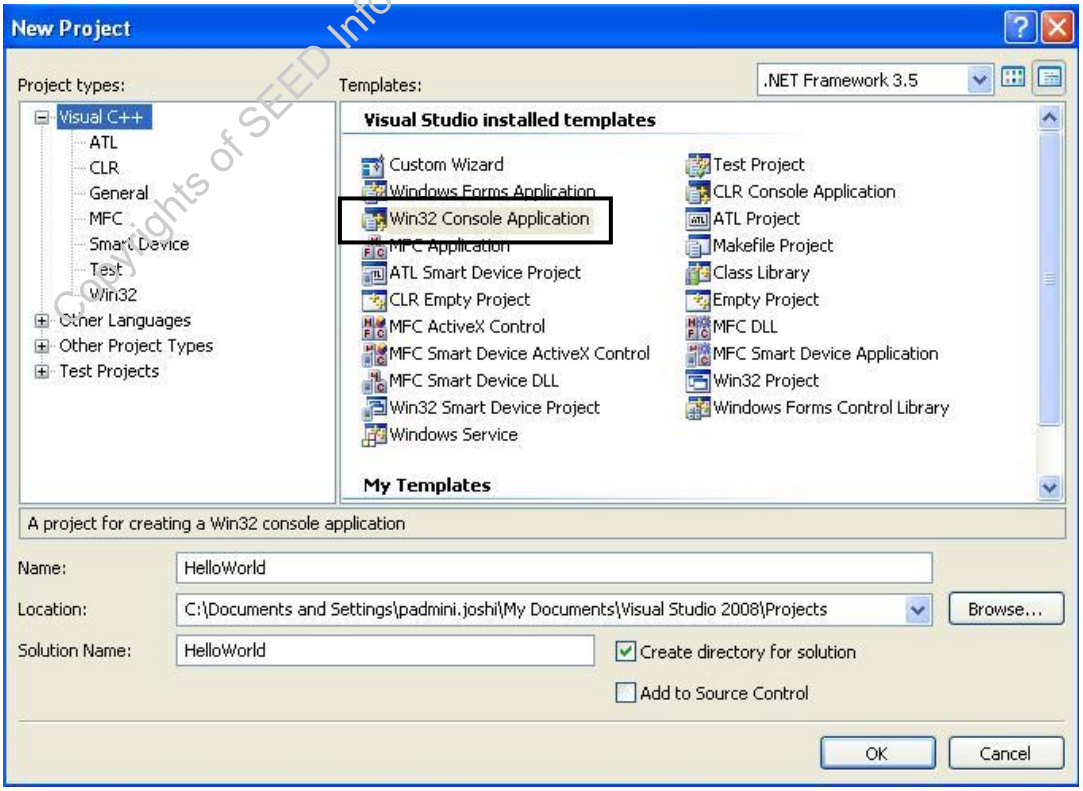
2. Write a program to evaluate the expression $z = x^2 + y^2$, where x and y are two arrays. Each array holds 10 user entered elements. Store this result in another array, z and display it.
   For example, if x[0]=3 and y[0]=4 then, display the result z[0]=25 That is 9 + 16.
3. Write a menu driven program for
   a. Deleting an element from an array. (Position of the element should be considered).
   b. Inserting element into an array. (Position of the element should be considered).

Appendix

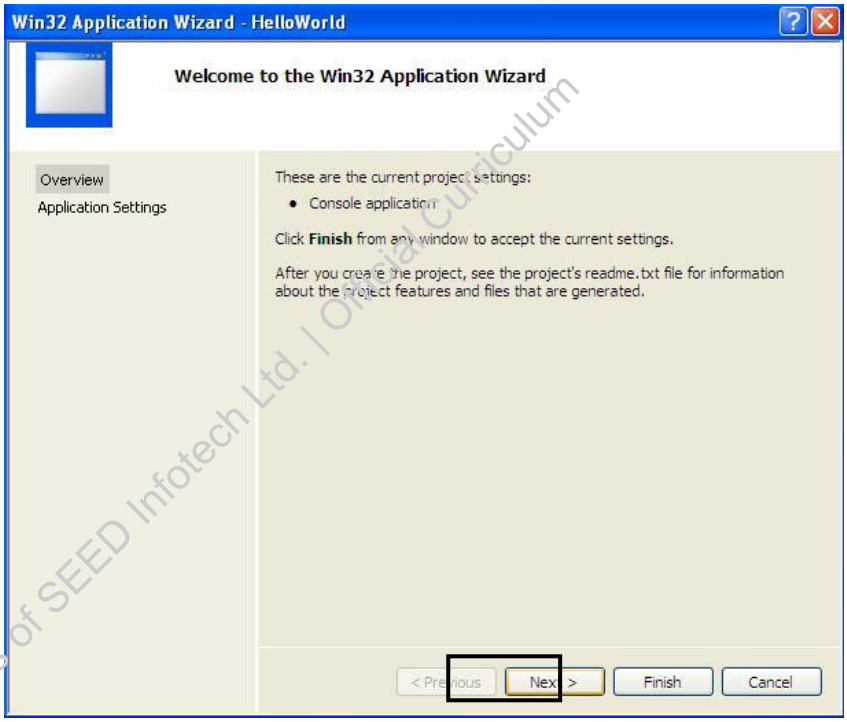
Introduction to Visual Studio IDE

Starting Visual Studio

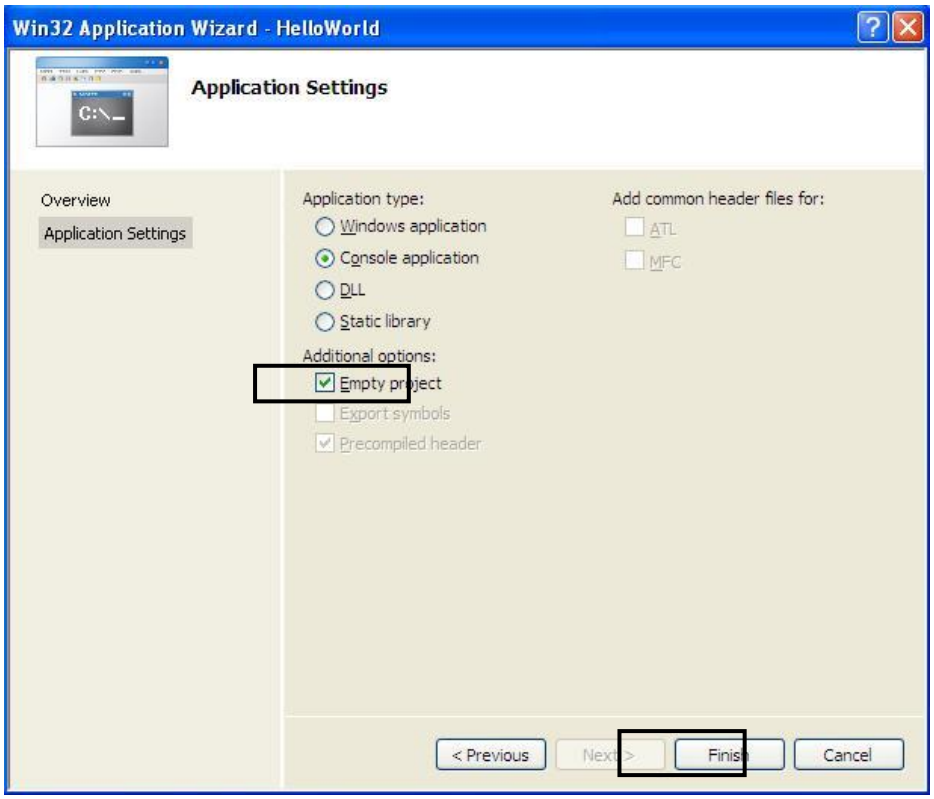
1. Start → All Programs → Microsoft Visual Studio 2008 → Microsoft Visual Studio 2008
2. Click on File →New →Project



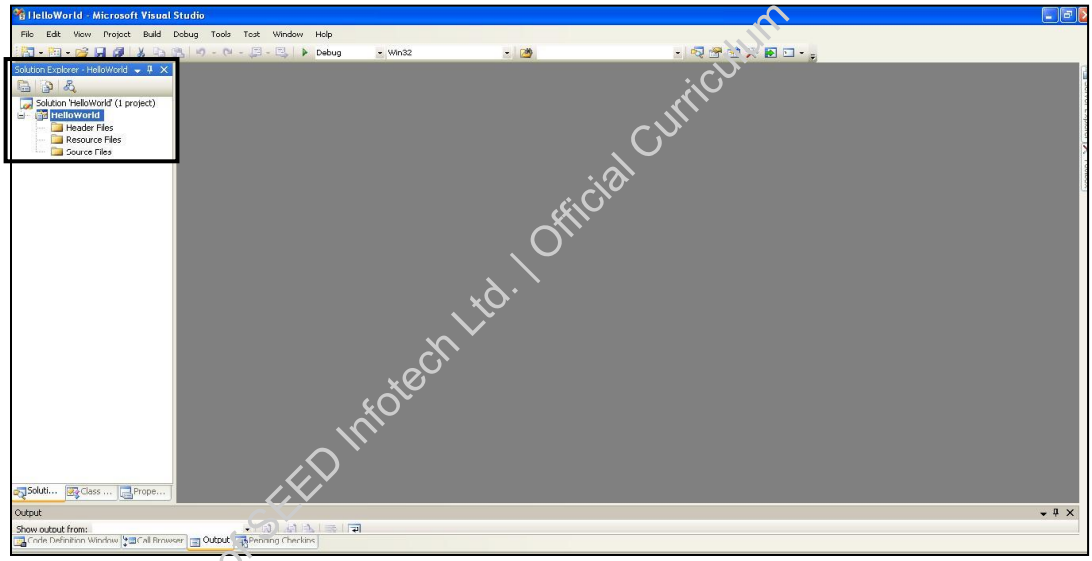
3. Select Visual C++ as Project Type; in the templates section select Win32 Console Application as your template.
4. Give a Suitable name to your project; if required, browse to the specific location to store the project.
5. Click on OK button.
6. As a result of step 5, a new Win32 Application wizard window will appear.
7. Click the Next button on the first screen.



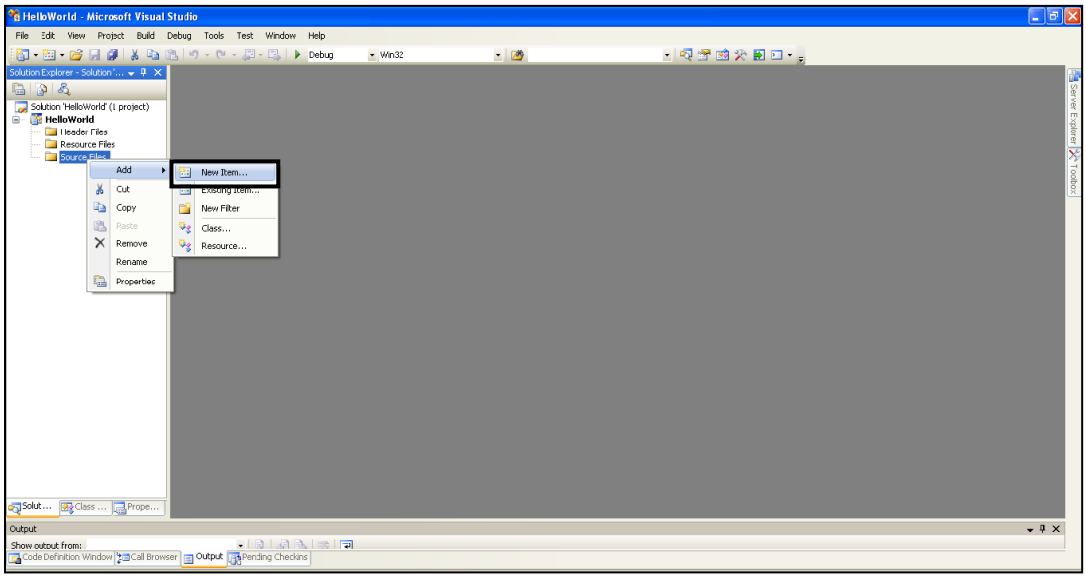
8. On the second screen check the Empty project additional option.
9. Then click on the Finish button.



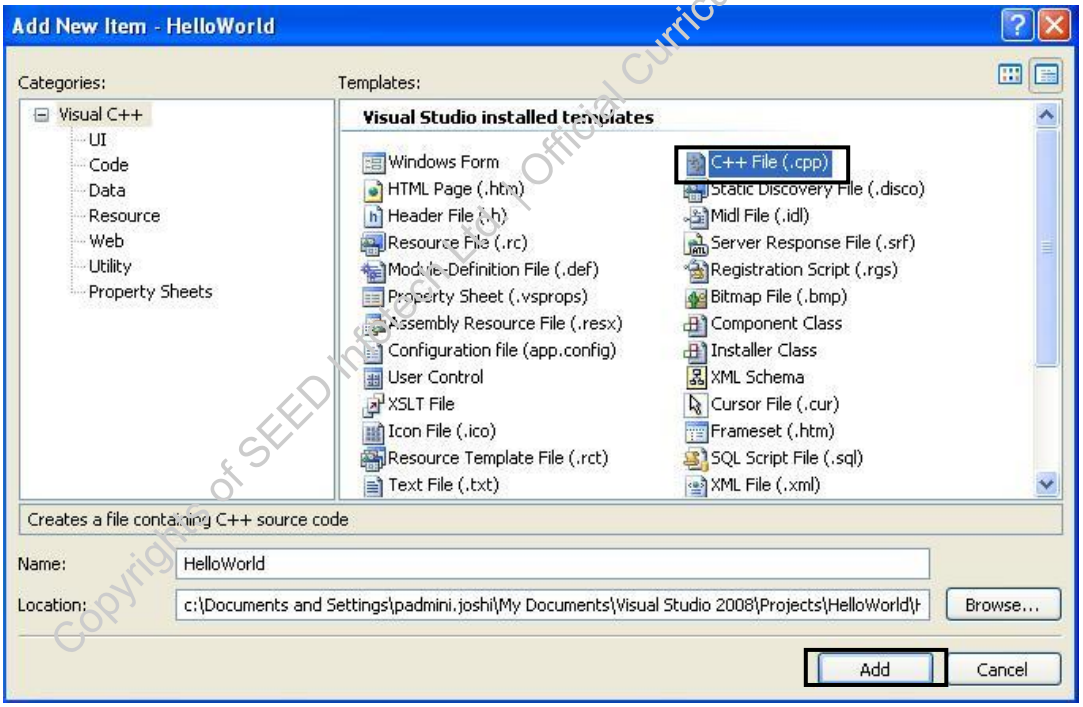
10. As a result of step 9, project window will appear.



11. Now in the solution explorer panel, right click on the Source Files folder, hover on Add menu and then click on New item entry in the submenu.

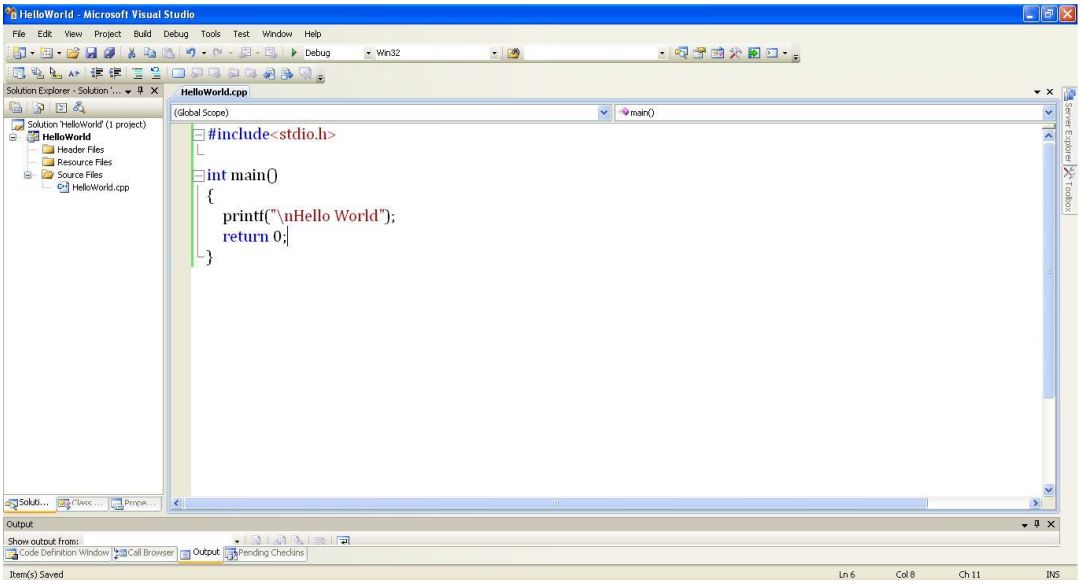


12. Add New Item window will appear. Now, select Visual C++ as category, C++ File (.cpp) as template and give a suitable name to your program. You may browse and go to the desired storage location if you want.

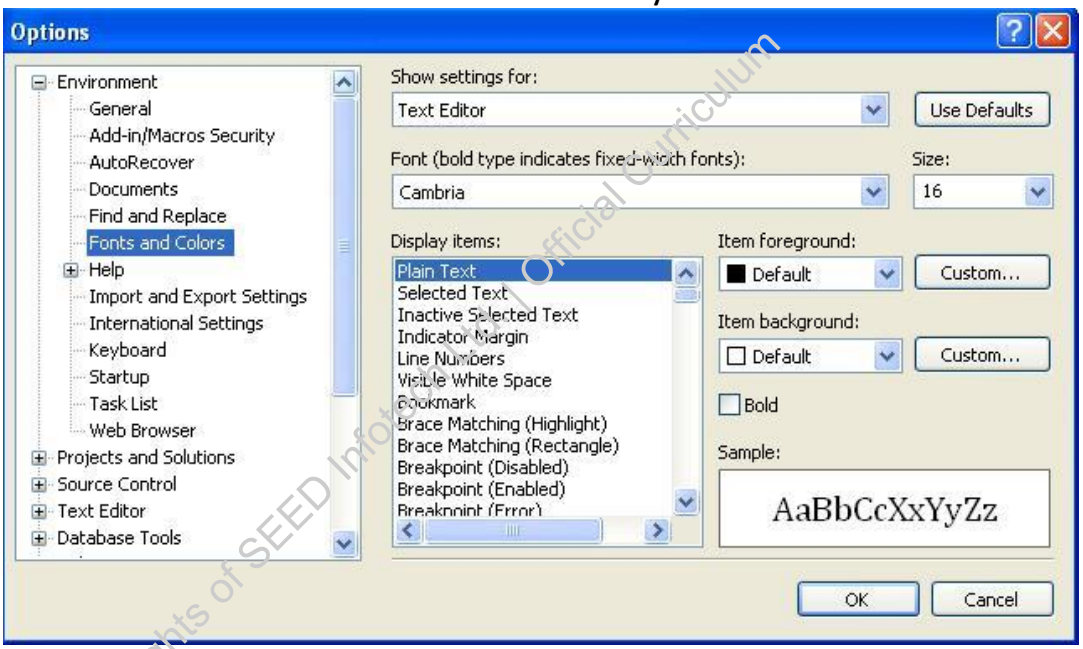


13. Click on the Add button.

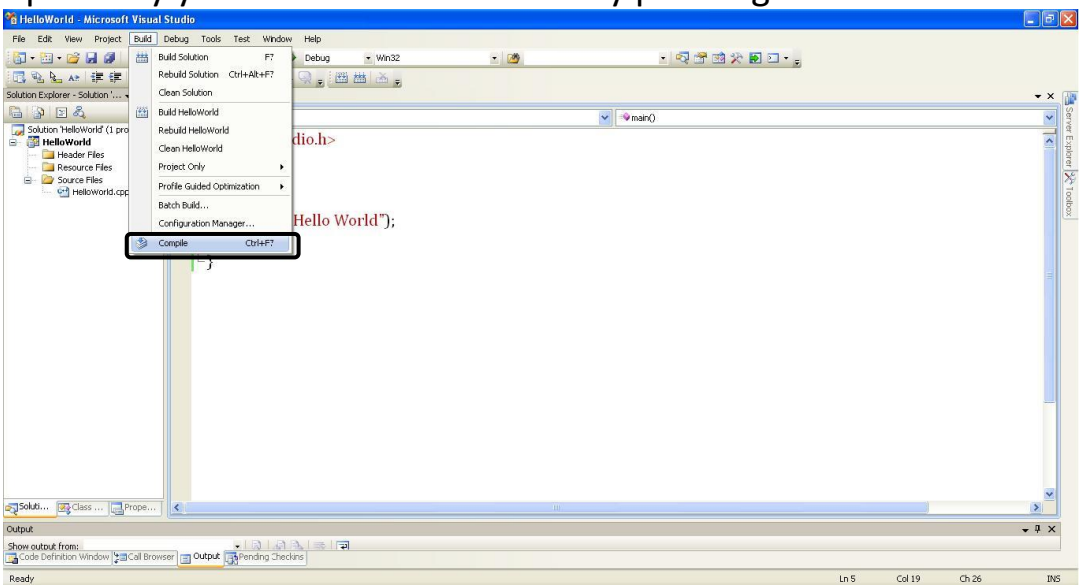
14. As a result of step 13, code editor window will take the center stage. Type your code in the editor window space.



15. If you want to change the font size, go to Tools →> Options →Fonts and Colors. And select font and color of your choice.

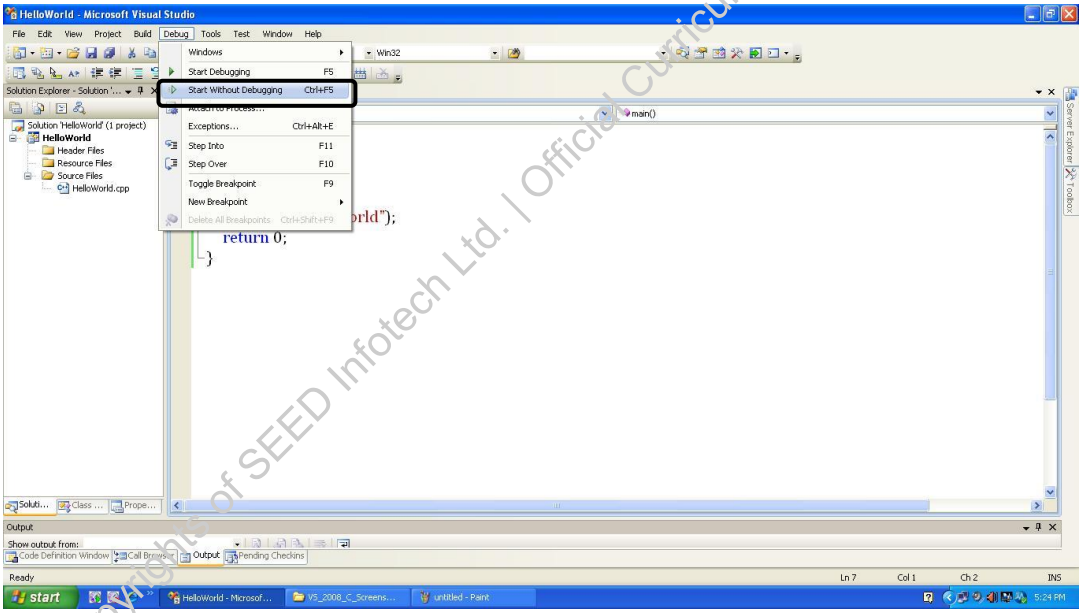


16. The program is ready. You have to compile, build and execute it.
17. To compile the program, click on Build menu option, and then click on compile menu entry.
Optionally you can also achieve this by pressing Ctrl + F7.



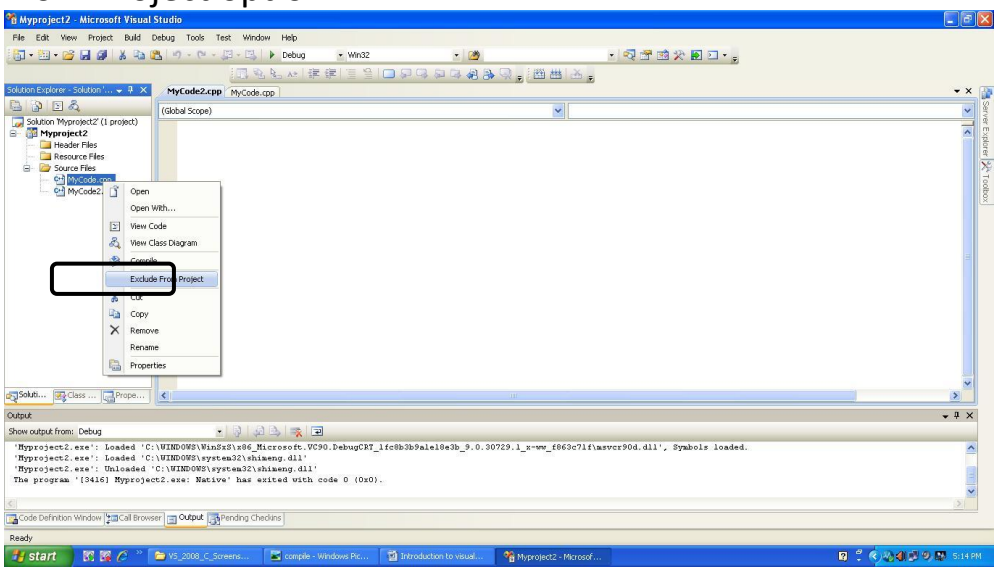
18. After successful compilation you need to build the solution. Go to build menu again and click on Build Solution menu option. Optionally you can also achieve this by pressing F7.

19. In order to execute your program, go to Debug menu option and click on start without debugging menu option. Optionally you can also achieve this by pressing Ctrl+ F5.



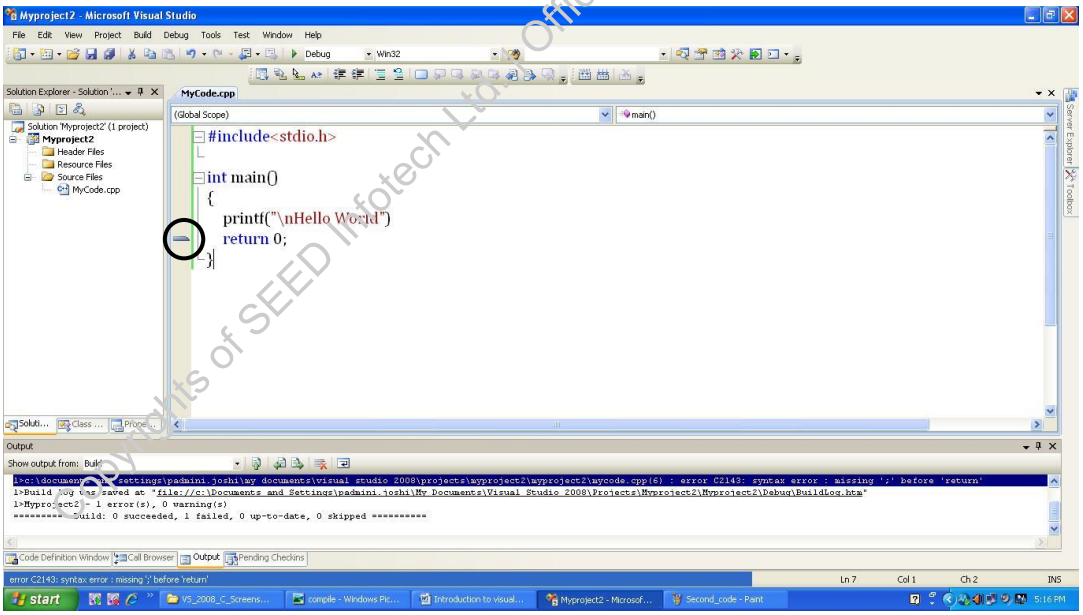
20. If you want you can add another source code (.cpp) file to your project; but as only one main() function can act as the starting point of execution. Make sure to exclude the earlier source code from the project.

21. To do so, right click on the source file to be excluded, and select Exclude from Project option.

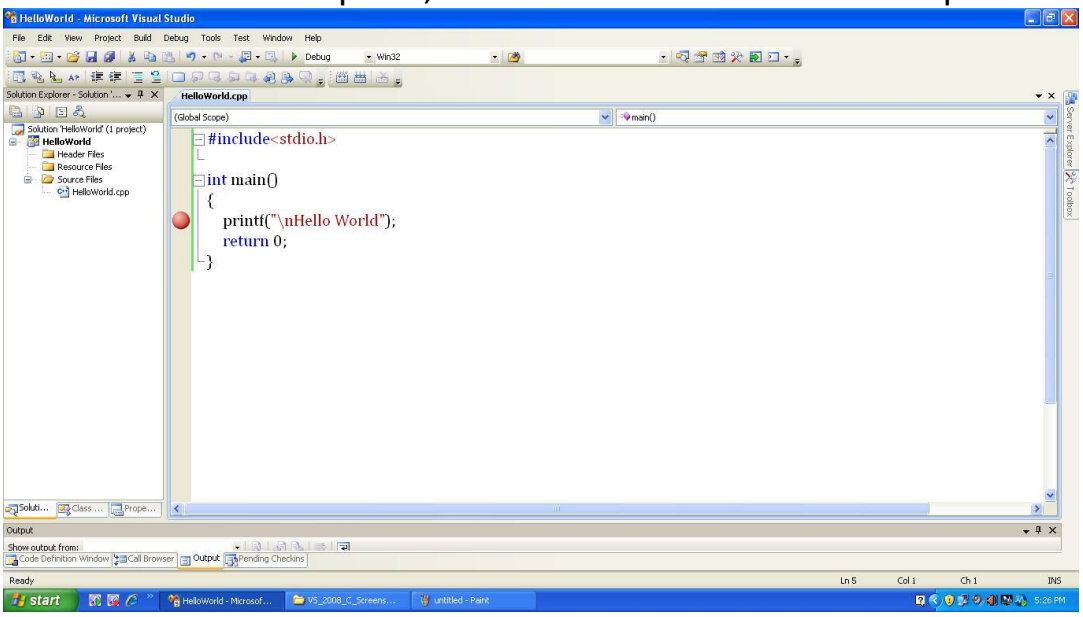


Debugging

When there are compiler errors, the line in the code, in which they occur, can be found out by pressing the F4 key. The error will be pointed by a blue arrow in the editor and in the message window the error message is highlighted. Explore the debug menu further to find the different debugging options like Step into, Step over etc.



To insert breakpoint, take the cursor to the required line and press F9 key. To remove the breakpoint, take the cursor to the line and press F9 again.

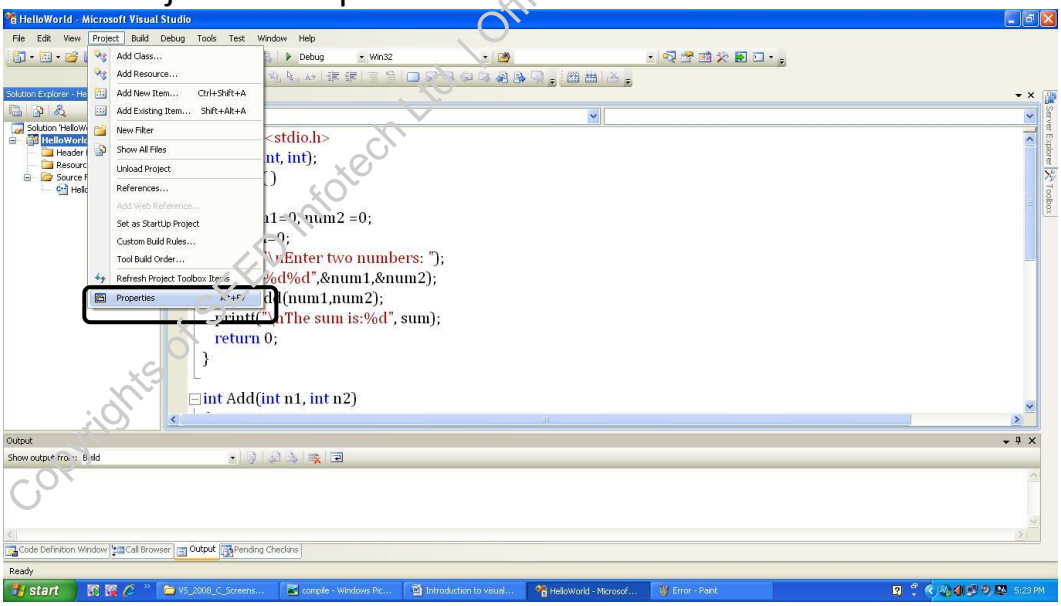


Using Command Line Arguments

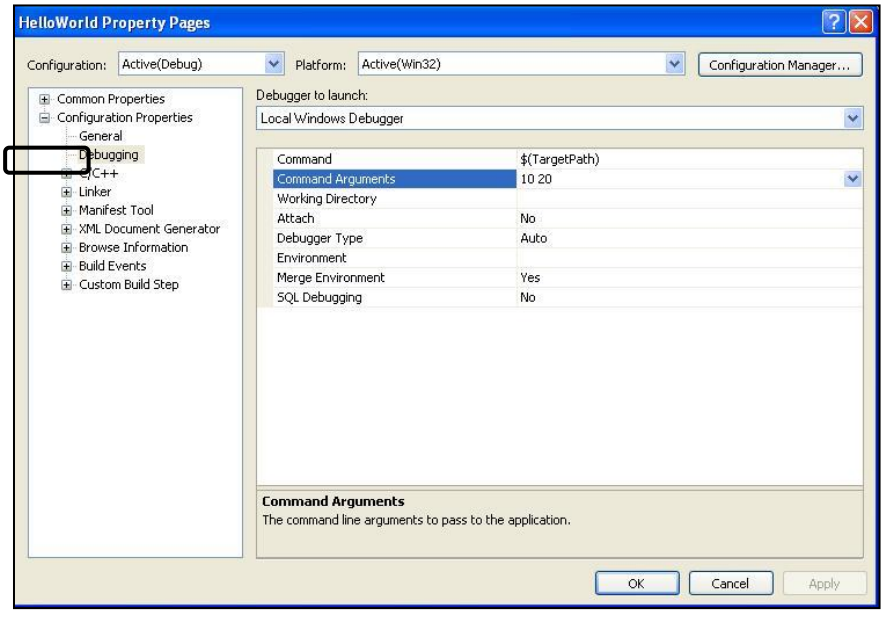
There is another way to execute a program that needs some input – by command prompt and passing the input using the command prompt.

To do so in Visual Studio:

1. Select Project → Properties

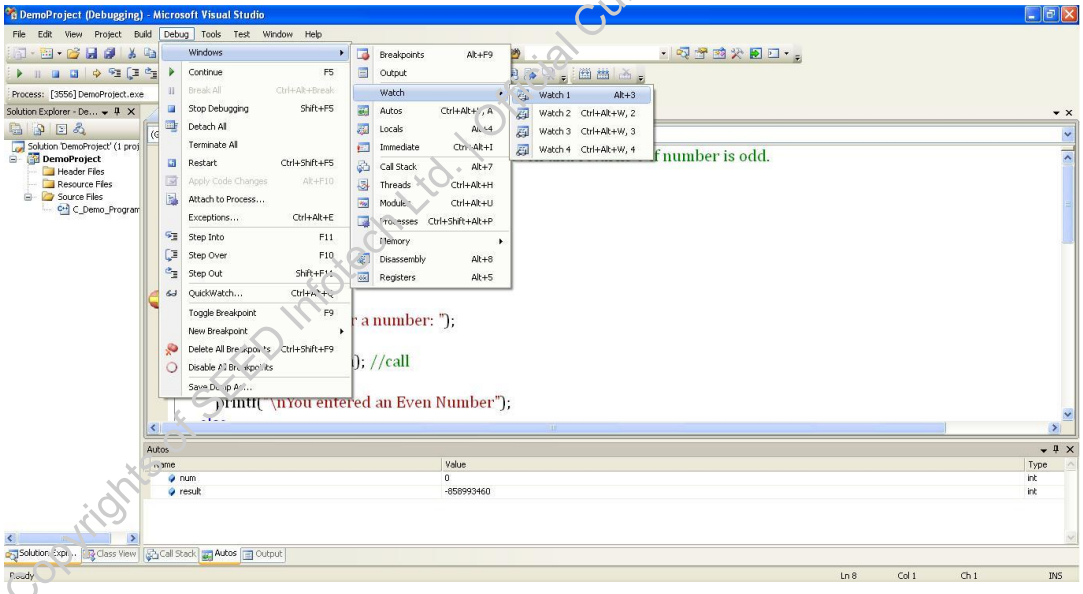


2. Then, select the Debugging tab. For further reference see the snapshot below. Here we have given 10 and 20 as command line arguments.

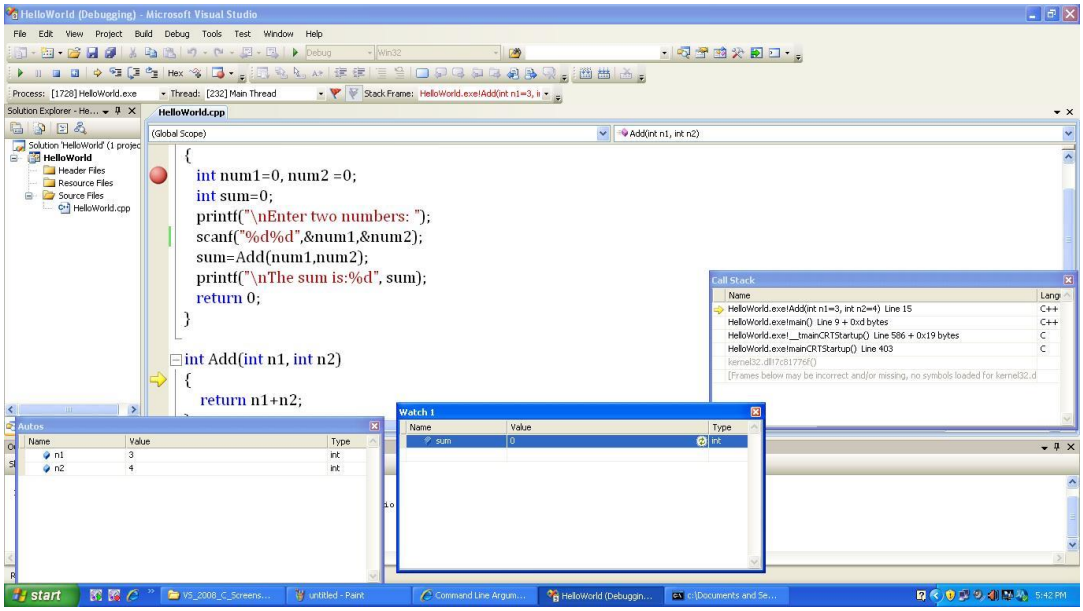


Using the Watch Window

When you start debugging, in the window below you can type the names of variables whose values you want to watch during the program execution. This can be seen in the snap shot below:



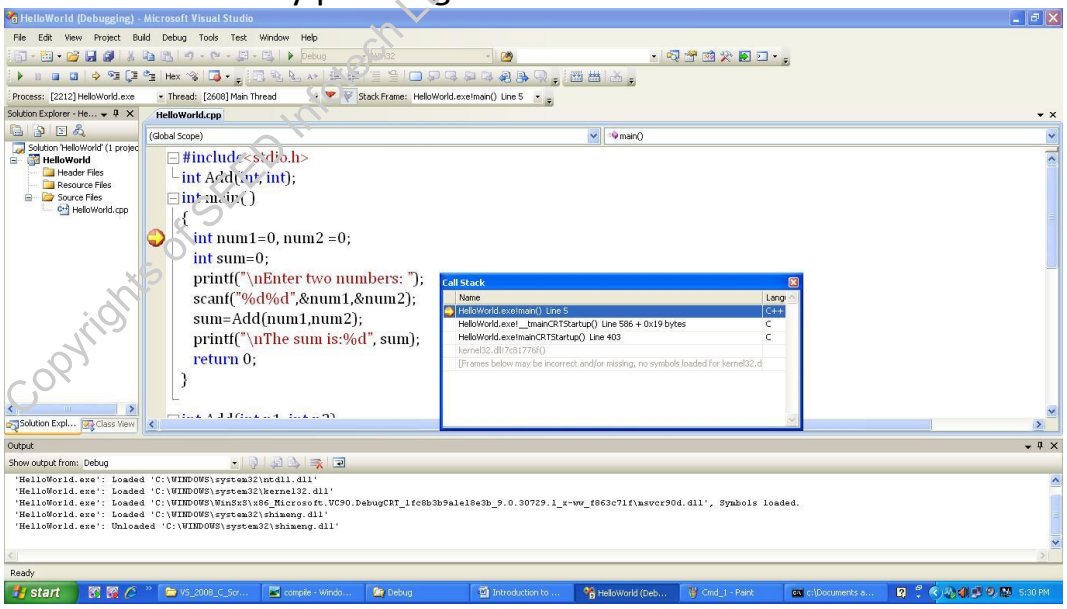
To open the watch window, first start debugging. Then go to Debug menu option, and then hover on windows option and in the subsequent submenu, hover on watch menu item. In the last sub menu you can select any of the watch windows (from watch1, watch2 etc).



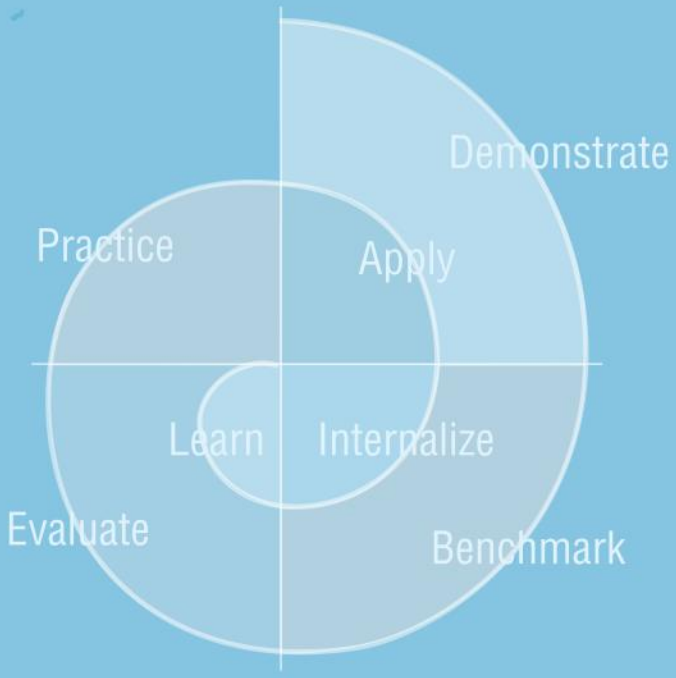
In the Autos window all the variables in the function under execution are listed whereas in the watch window are the variables whose values you want to watch.

Using the Call Stack

The call stack (for various function calls) can be seen in the run mode. To do so, debug the code by pressing F5 and then go to debug menu, hover on windows and select Call Stack menu entry in sub menu. Optionally you can achieve this by pressing Alt+7.



Here the various function calls can be seen on the stack. When a function is called, it is placed on stack.



Avani publications

N·S·D·C National Skill Development Corporation
Transforming the skill landscape
Training Partner

seed
Official Curriculum