

PROGRAMMING FUNDAMENTALS

WAN FAZLINI IDAYU BINTI W.FAKARI

KEJURUTERAAN ELEKTRIK

Wan Fazlini Idayu W. Fakari

PROGRAMMING FUNDAMENTALS / WAN FAZLINI IDAYU BINTI W.FAKARI.

Mode of access: Internet

eISBN 978-967-2240-28-0

1. Computer programming.
2. Programming languages (Electronic computers).
3. Government publications--Malaysia.
4. Electronic books.

I. Title.

005.13

First Published 2021

© Politeknik Kuala Terengganu

e-ISBN 978-967-2240-27-3

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic, including photocopying, recording or by any information storage or retrieval system, without prior written permission from the Director of Politeknik Kuala Terengganu

Author :

Wan Fazlini Idayu Binti W.Fakari

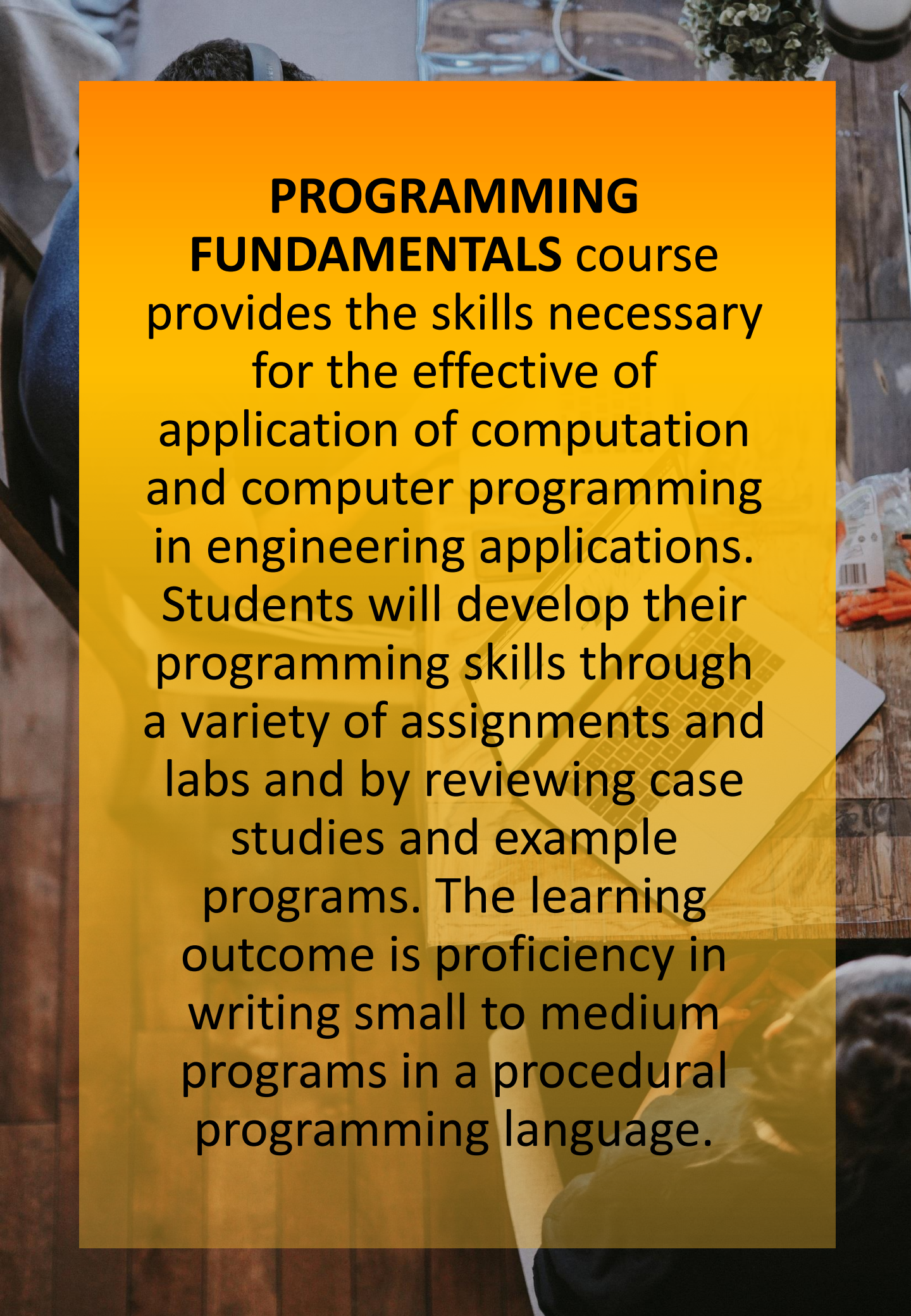
Published by :

Politeknik Kuala Terengganu,

Jalan Sultan Ismail,

20200 Kuala Terengganu, Terengganu.

09-6204100

A person is seen from behind, wearing a headset and working at a desk. On the desk, there is a laptop, a glass of water, and some papers. The background is a wooden wall. The text is overlaid on a yellow semi-transparent box.

**PROGRAMMING
FUNDAMENTALS** course
provides the skills necessary
for the effective of
application of computation
and computer programming
in engineering applications.
Students will develop their
programming skills through
a variety of assignments and
labs and by reviewing case
studies and example
programs. The learning
outcome is proficiency in
writing small to medium
programs in a procedural
programming language.




Table of contents

1 Introductory to Programming 1

- 1.1 Programming Language
- 1.2 Types of Programming
- 1.3 Structure Programming Methodology
- 1.4 Algorithm, Flow Chart and and pseudocode
- 1.5 Algorithm, flowchart and pseudocode
- 1.6 Algorithm, flowchart, pseudocode and analyze problem

2 Fundamentals of C Language 28

- 2.1 Variables, Constants and Data Types
- 2.2 Fundamentals of C Programming
- 2.3 Input, Proses & Output statements
- 2.4 Hardware & Software operation

3 Selection Statements 73

- 3.1 Selection statements
- 3.2 IF statements
- 3.3 IF-ELSE statement
- 3.4 Nested IF statement
- 3.5 SWITCH statements

4 Looping Statements 88

- 4.1 Looping statements
- 4.2 FOR statement
- 4.3 Nested FOR statement
- 4.4 WHILE, DO-WHILE loop statements

5 Function and Array 111

- 5.1 Function statement
- 5.2 Function prototype declaration
- 5.3 Returning function result
- 5.4 Function call function
- 5.5 Arrays statement.
- 5.6 Multidimensional arrays
- 5.7 I/O operation

CHAPTER 1



Introductory to Programming

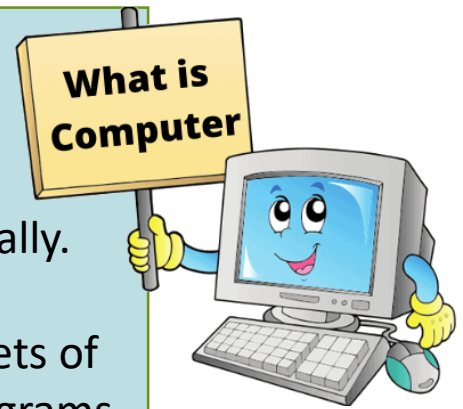


INTRODUCTION

A computer is a machine that can be programmed to carry out sequences of arithmetic or logical operations automatically.

Modern computers can perform generic sets of operations known as programs. These programs enable computers to perform a wide range of tasks.

Most of us have used computers in one way or another. For example, withdrawing money from the automated teller machine (ATM) is a form of interaction with computer



PROGAMMING

Computer programming is the process of designing and building an executable computer program to accomplish a specific computing result or to perform a specific task.

Programming involves tasks such as: analysis, generating algorithms, profiling algorithms' accuracy and resource consumption, and the implementation of algorithms in a chosen programming language (commonly referred to as coding).



The process of **writing, testing and maintaining** the source code of the computer program

What ?

Computer Programming

How to program ?

- Requires knowledge in the application domain
- Follow the steps in software development method

WHY computer Programming?

Programming is a problem-solving activity

To solve problems occurred in life with the assistance of computer

To ease daily process e.g.: transaction, payroll, accounting, registration, information exchange etc.

Know the programming language

A set of symbol, word, code or instructions which is understood by computer

What ?

Programming Language



Method of communication for which computers could understand and execute the instructions written in source code.

Function?

A programming language is therefore a practical way for us (humans) to give instructions to a computer.

Hello! What can I do to you?



Tell me $1 + 1$ is equal to what???



Background of C programming

C History



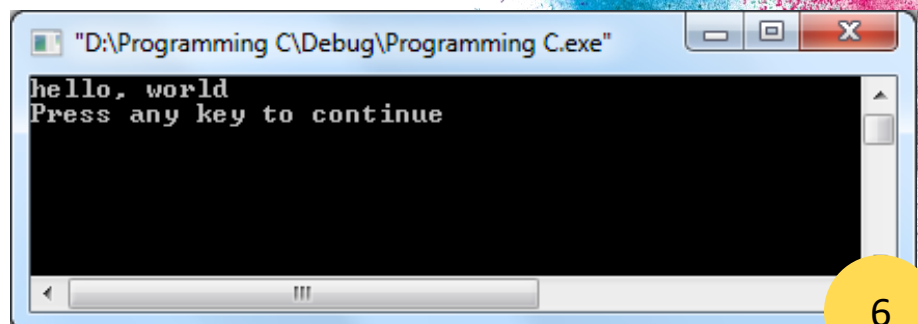
- Derived from the BCPL language by Martin Richards (1967).
- Ken Thompson developed a B language from BCPL language(1970).
- Evolved into the C language by Dennis Ritchie (1970) at Bell Telephone Laboratories Inc. (now the AT & T Bell Laboratories).
- C language was first used on a computer Digital Equipment Corporation PDP-11 to fully use in UNIX operating system.

Why use C?

- the portability of the compiler;
- the standard library concept;
- a powerful and varied repertoire of operators;
- an elegant syntax;
- ready access to the hardware when needed;
- and the ease with which applications can be optimised by hand-coding isolated procedures
- C is often called a "Middle Level" programming language. This is not a reflection on its lack of programming power but more a reflection on its capability to access the system's low level functions.

Sample of C program

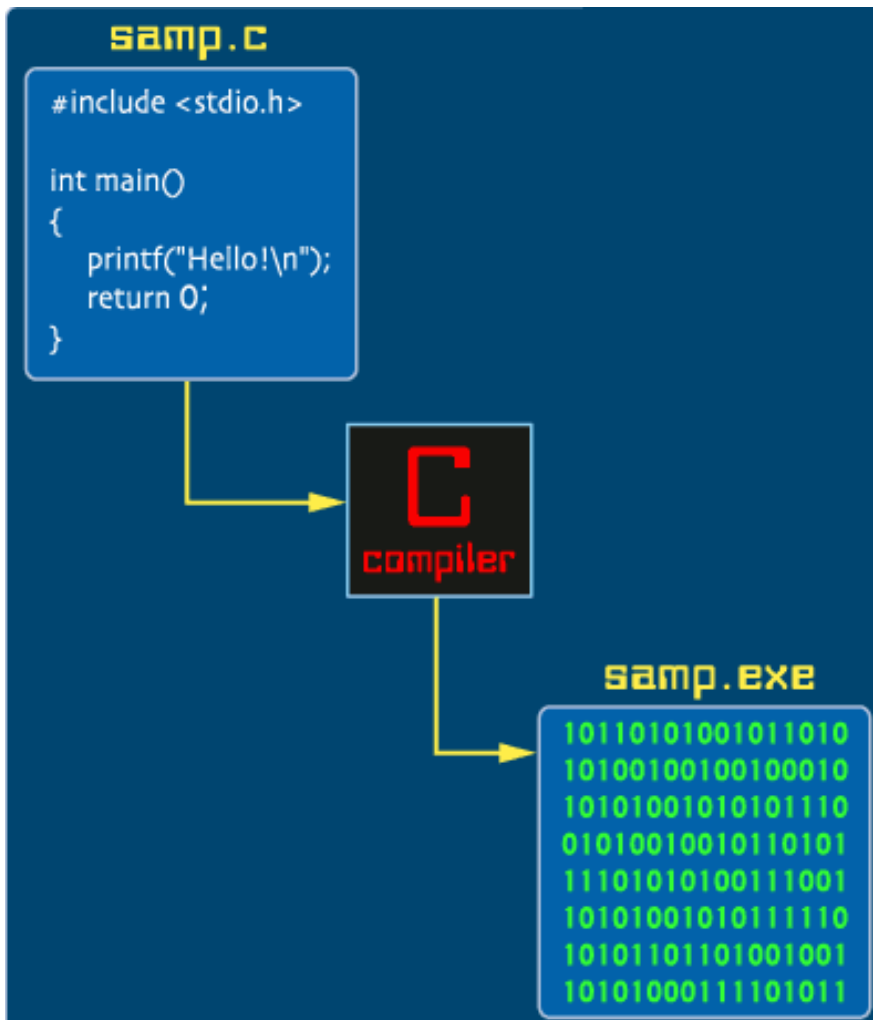
```
#include <stdio.h>
main()
{
    printf("hello, world\n");
    return 0;
}
```



```
"D:\Programming C\Debug\Programming C.exe"
hello, world
Press any key to continue
```

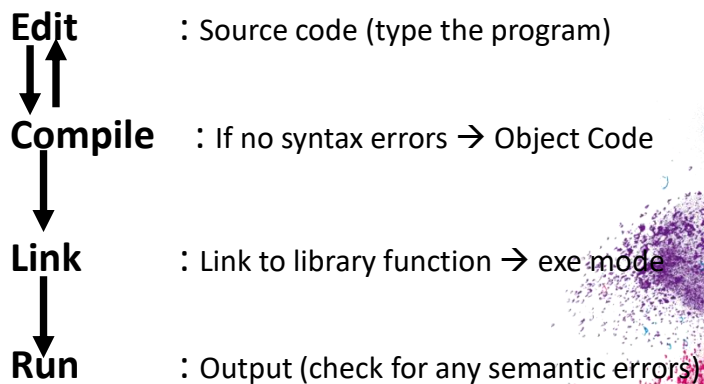
Examples of C programming

Sample of C program



Compile and execute programs.

- computer program run through the following steps:



Definition and types of programming

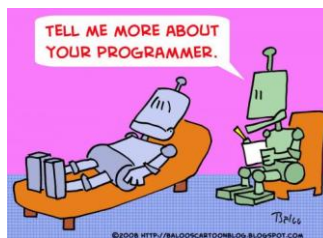
Programme

- A set of *step-by-step instructions* that tells a computer to perform a specific task and to produce the required results.
- written by the programmer
- Produced through programming



Programmer

- A Programmer is a person who *designs, writes and test* computer programs.
- Individual that composes instructions for computer systems to refer to when performing a given action.



Programming

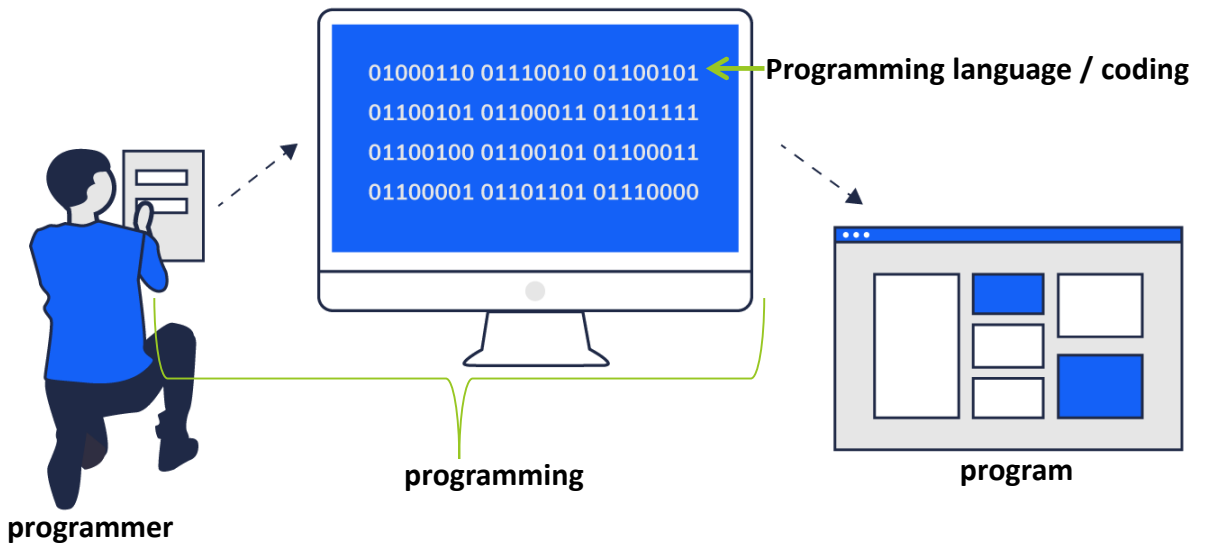
- Programming is a process of designing or creating a program.
- It is a communication technique to explain the instructions to the computer.
- Used to produce the program.



Programming Language

- a set of conventions in which instructions for the machine are written.
- A high-level language used to write computer programs, as COBOL or BASIC, or, sometimes, an assembly language.
- An artificial language used to write instructions that can be translated into machine language and then executed by a computer.

Understand the term of program, programming, programmer and programming language



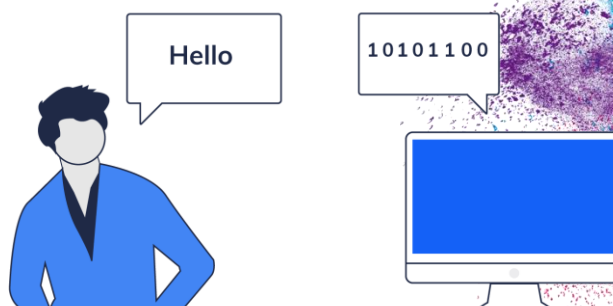
How does your computer understand your code?



- What most programmers write as “code” is a high level programming language. It is abstract by design. Abstraction in this context means that we are moving further away from machine code and programming languages are closer to spoken languages.
- But a computer can’t understand text based code. It needs to be compiled (translated) into machine code. Machine code is a set of instructions which can be understood by a computer’s central processing unit (CPU). Think of the CPU as the brain of a computer. Machine code is made up of ones and zeros. This is called binary.

What exactly is a programming language?

- Programming languages fall both within the spectrum of low-level languages, such as assembly, and high level programming languages, such as JavaScript.



C language

Table 1 - Software in C language for detection and storage of touches in the sensor file.

```
#include <stdio.h>
#include <io.h>
#include <dos.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

main()
{
char ch,*txt_File;
int bit;
int data;
FILE *rat;
struct time t;

clrscr();
gotoxy(1,1);
printf("\n...RECORDING...\n");
if(kbhit()) getch();

while (!kbhit())
{
outportb (0x378,0xFF);
On_Screen_ini();
data = inportb (0x378);
for (bit=0;bit<8;bit++)
{
if ((data&1)==0x00)
{
On_Screen(int s)
{
gotoxy(s*9+1,20);
printf("CAGE%d",s);
}
}
}
}
}

On_Screen(bit);
gettime(&t);

printf(txt_File,"rat%d.txt",bit);
rat = fopen(txt_File,"a");
fprintf(rat, "%2dt %02dt
%02dn",t.ti_hour,t.ti_min, t.ti_sec);
fclose(rat);
data=data>>1;
}
}

On_Screen_ini()
{
int i;
for (i=0;i<8;i++)
{
gotoxy(i*9+1,20);
printf("cage%d",i);
}
}

On_Screen(int s)
{
gotoxy(s*9+1,20);
printf("CAGE%d",s);
}
}
```

assembly Language Motorola I.C

```
MONITOR FOR 6802 1.4          9-14-80 TSC ASSEMBLER PAGE 2

C000          C000 8E 00 70  START  ORG  ROM+$0000 BEGIN MONITOR
C000          LDS  #STACK

*****
* FUNCTION: INITA - Initialize ACIA
* INPUT: none
* OUTPUT: none
* CALLS: none
* DESTROYS: acc A

0013          RESETA EQU  #00010011
0011          CTLREG EQU  #00010001

C003 86 13    INITA  LDA  A  #RESETA  RESET ACIA
C005 B7 80 04  STA  A  ACIA
C008 86 11    LDA  A  #CTLREG  SET 8 BITS AND 2 STOP
C00A B7 80 04  STA  A  ACIA

C00D 7E C0 F1          JMP  SIGNON  GO TO START OF MONITOR

*****
* FUNCTION: INCH - Input character
* INPUT: none
* OUTPUT: char in acc A
* DESTROYS: acc A
* CALLS: none
* DESCRIPTION: Gets 1 character from terminal

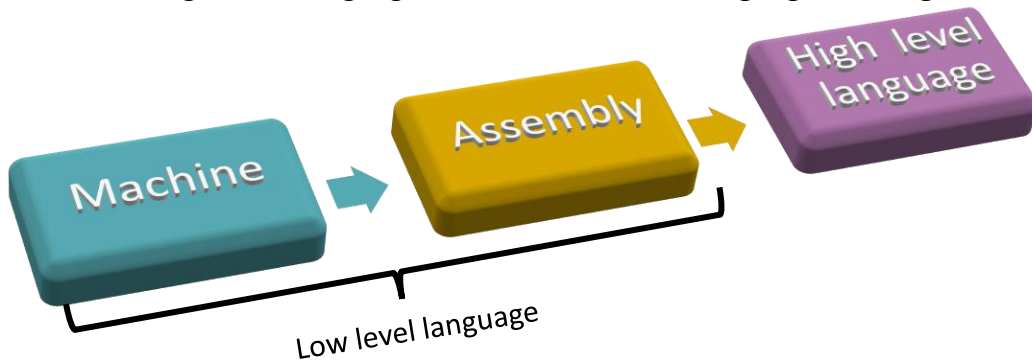
C010 86 80 04  INCH  LDA  A  ACIA  GET STATUS
C013 47        ASR  A  SHIFT RDRF FLAG INTO CARRY
C014 24 FA     BCC  INCH  RECEIVE NOT READY
C016 86 80 05  LDA  A  ACIA+1 GET CHAR
C019 84 7F     AND  A  #57F  MASK PARITY
C01B 7E C0 79  JMP  OTCCH  ECHO & RTS

*****
* FUNCTION: INHEX - INPUT HEX DIGIT
* INPUT: none
* OUTPUT: Digit in acc A
* CALLS: INCH
* DESTROYS: acc A
* Returns to monitor if not HEX input

C01E 8D F0    INHEX BSR  INCH  GET A CHAR
C020 81 30    CMP  A  #0  ZERO
C022 2B 11    BMI  HEXERR  NOT HEX
C024 81 39    CMP  A  #9  NINE
C026 2F 0A    BLE  HEXRTS  GOOD HEX
C028 81 41    CMP  A  #A  GOOD HEX
C02A 2B 09    BMI  HEXERR  NOT HEX
C02C 81 46    CMP  A  #F  GOOD HEX
C02E 2E 05    BGT  HEXERR
C030 80 07    SUB  A  #7  FIX A-F
C032 84 0F    HEXRTS AND  A  #50F  CONVERT ASCII TO DIGIT
C034 39      RTS

C035 7E C0 AF  HEXERR JMP  CTRL  RETURN TO CONTROL LOOP
```

- There are multiple types of programming language. Choosing a suitable one is important.
- There are two categories of language, which are **low-level languages** and **high-level languages**.

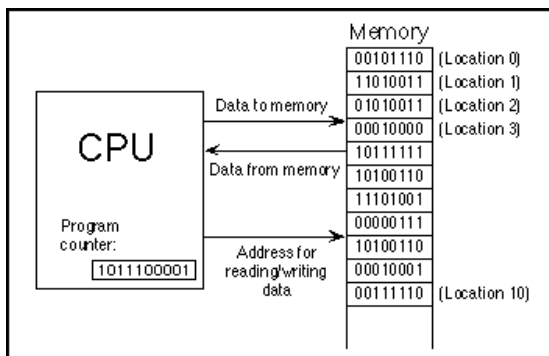


| Low Level Language | | High Level Language |
|---|---|---|
| Machine Language | Assembly Language | |
| Binary number codes understood by a specific CPU | Mnemonic codes that correspond to machine language instructions | Machine-independent programming language that combines algebraic expressions and English symbols. |
| Lowest-level programming language | Second-level of language. | Need translator (compiler) to convert high-level to low-level |
| Only understood by computers. Express in binary form. | Develop to replace "0" and "1" used in machine language | Machine independent language. Can be executed on any computer |
| Impossible for human to use | English-like abbreviations representing elementary computer operations (translate via assemblers) | English-like words. (eg. If, printf, scanf etc.) |
| Example: | Example: | Example: |
| 0101 1100 1101 0011 1001 1111 0011 1001 1111 1100 1101 1100 1101 0011 1001 1111 1001 1111 1001 1111 1100 0011 1001 1111 1011 1001 1111 1011 1001 1111 | STATUS equ 03h TRISA equ 85h PORTA equ 05h COUNT1 equ 08h COUNT2 equ 09h | #include <stdio.h> void main() { printf("\n"); printf("Hello World"); printf("\n"); } |

Programming Languages

#1: Machine Level Language

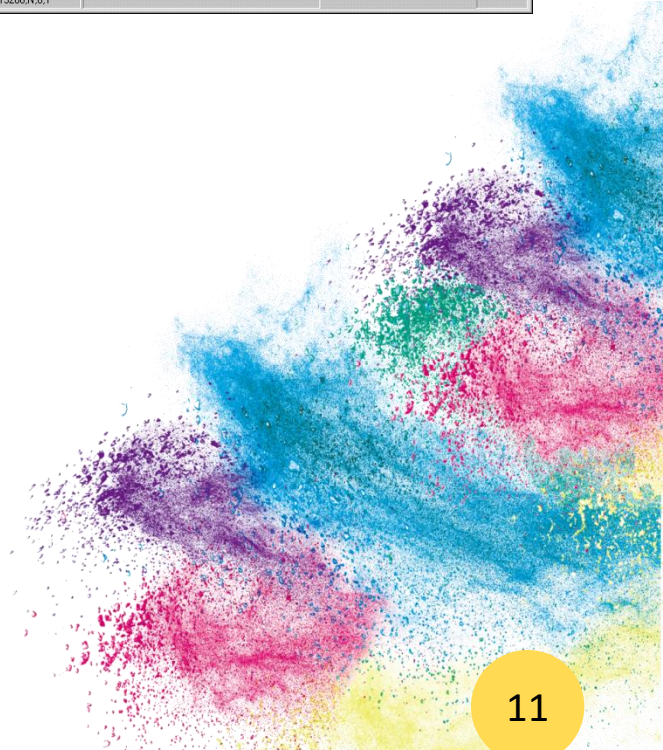
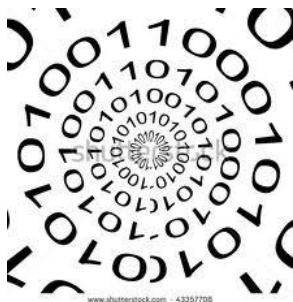
- Machine languages are the only Programming languages understood by computers. While easily understood by computers, machine languages are almost impossible for human to use because they consist entirely of numbers (binary bits i.e. 0 or 1).
- Advantages:
 - the programs are written in binary form,
 - there is no need of assemblers or **compilers** to convert the codes to machine readable form so the execution is fast.
 - This also leads to smaller file size.
- Disadvantages:
 - It is **extremely difficult** to learn , as program codes are to be written in binary form.
 - It is **machine dependent** so program written on one computer cannot be run on another computer of the same type.
 - As they are machine dependent proper knowledge of CPU.



A screenshot of a terminal emulator window titled 'BASCOM-8051 Terminal emulator'. The window displays the following text:

```
File Terminal
M052 Version 1.1
Grifo(r) ITALIAN TECHNOLOGY
Tel:+39 051 892 052 Fax:+39 051 893 661
http://www.grifo.com http://www.grifo.it

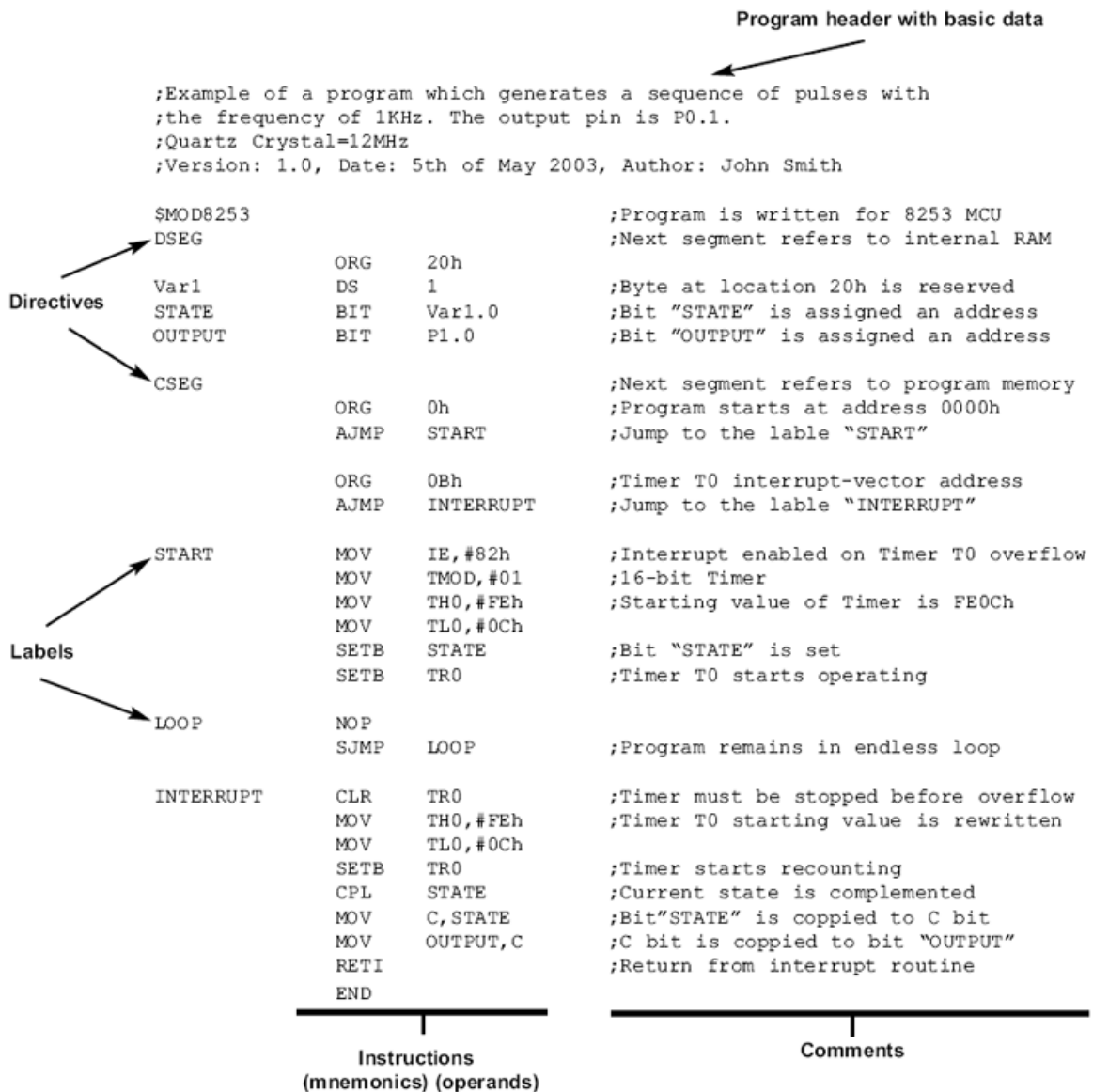
+R 00=FFFF SP=FF A=FF B=FF PSW=FF DPTR=FFFF
R0=FF R1=FF R2=FF R3=FF R4=FF R5=FF R6=FF R7=FF
+D 2050,2100
2050 01 02 04 08 10 20 40 80 55 AA 33 01 02 04 08 10 .....@.U.3.....
2060 20 40 80 55 AA 33 01 02 04 08 10 20 40 80 55 AA @.U.3.....@.U.
2070 33 01 02 04 08 10 20 40 80 55 AA 33 01 02 04 08 3.....@.U.3....
2080 10 20 40 80 55 AA 33 01 02 04 08 10 20 40 80 55 ..@.U.3.....@.U
2090 AA 33 01 02 04 08 10 20 40 80 55 AA 33 01 02 04 ..3.....@.U.3...
20A0 08 10 20 40 80 55 AA 33 01 02 04 08 10 20 40 80 ..@.U.3.....@.
20B0 55 AA 33 01 02 04 08 10 20 40 80 55 AA 33 01 02 U.3.....@.U.3...
20C0 04 08 10 20 40 80 55 AA 33 01 02 04 08 10 20 40 ...@.U.3.....@
20D0 80 55 AA 33 01 02 04 08 10 20 40 80 55 AA 33 01 ..U.3.....@.U.3...
20E0 02 04 08 10 20 40 80 55 AA 33 01 02 04 08 10 20 ...@.U.3.....
20F0 40 80 55 AA 33 01 02 04 08 10 20 40 80 55 AA 33 @.U.3.....@.U.3
2100 01 02 04 08 10 20 40 80 55 AA 33 01 02 04 08 10 .....@.U.3.....
```



Programming Languages

#2: Low Level or Assembly Language

- Assembly language is a type of programming language ,which is used to **program computers, microprocessors, microcontrollers**, and other (usually) integrated circuits.
- They implement a symbolic representation of the numeric machine codes and other constants needed to program a particular CPU architecture.
- An assembly language is thus specific to certain physical or virtual computer architecture. An assembly language programmer must understand the microprocessor's unique architecture (*such as its registers and instruction*).
- Program written in assembly language is converted to binary codes using special programs called assemblers.
- In assembly language a mnemonic is a code, usually from **1 to 5 letters**, that represent an operational code (op-code), followed by one or more numbers (the operands).
- Op-code or operation code is between one and three bytes in length and uniquely defines the function that is performed. It is the data that represents a microprocessor instruction.



Programming Languages

#2: Low Level or Assembly Language

- Advantages:

- Program written in assembly language are simpler than program written in machine codes as use of binary codes to represent operational codes is replaced by words (mnemonics).
- Program written for a family of microprocessors need not be rewritten i.e. machine dependence is somewhat reduced.
- Less knowledge of CPU architecture compared to machine level language is required.

- Disadvantages:

- Requirement of knowledge of CPU architecture is not completely eliminated.
- Assembly Languages are to be converted into binary codes using assemblers so final executable file size is large compared to machine level ones.



#3: High Level Language

- High level programming languages are those programming language which use normal everyday word to represent the executable operational codes.
- These types of programming language follow strictly followed rule for writing these instruction. This rule is known as **syntax**.
- High level languages are easy to learn as they avoid the need to understand the complex CPU architecture and also because the commands are in plain **understandable English form**.
- These programs written in plain English form following syntax are converted in machine understandable form using either **compilers** or **interpreters**.
- Advantages.
 - They are easy to understand and user friendly.
 - It reduces the complexity of programming as need of knowledge of CPU architecture is eliminated.
 - High level programs are very easy to maintain than machine and lower level languages. In machine and lower level languages, instructions are difficult and very hard to locate, correct and modify but in high level language, it is very easy to understand and modify when desired.
 - Each high-level language provides a large number of built-in functions or procedures that can be used to perform specific tasks during designing of new programs. In this way, a large amount of time of programmer is saved.
 - Program written in high-level language is machine independent. It means that a program written on one type of computer can be executed on another type of computer.

Programming Languages

#3: High Level Language

- Advantages.
 - They are easy to understand and user friendly.
 - It reduces the complexity of programming as need of knowledge of CPU architecture is eliminated.
 - High level programs are very easy to maintain than machine and lower level languages. In machine and lower level languages, instructions are difficult and very hard to locate, correct and modify but in high level language, it is very easy to understand and modify when desired.
 - Each high-level language provides a large number of built-in functions or procedures that can be used to perform specific tasks during designing of new programs. In this way, a large amount of time of programmer is saved.
 - Program written in high-level language is machine independent. It means that a program written on one type of computer can be executed on another type of computer.
- Disadvantages.
 - The additional process of compilation needs more machine time than the straight assembly process.
 - There is no control of hardware part while writing high level programs.
 - The programs have to be compiled every time a change is made.

Compiler

- Compiled languages are converted directly into machine code that the processor can execute. As a result, they tend to be faster and more efficient to execute than interpreted languages.

Interpreter

- Interpreters run through a program line by line and execute each command.
- Interpreted languages were once significantly slower than compiled languages.

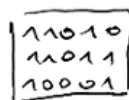
Source code:

hello.c



COMPILER

Machine code:



Program (also called binary, executable ...)

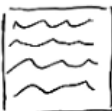
run the program

result



Source code:

hello.py



INTERPRETER

result



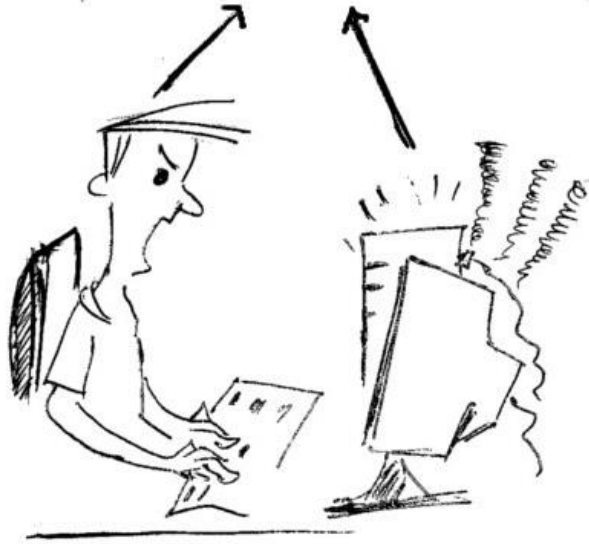
Computer only understands machine language.

So, computer need translator call:
 -Assembler
 -Compiler or interpreter.

Assembler:
 assembly → machine

compiler or interpreter
 high level → machine

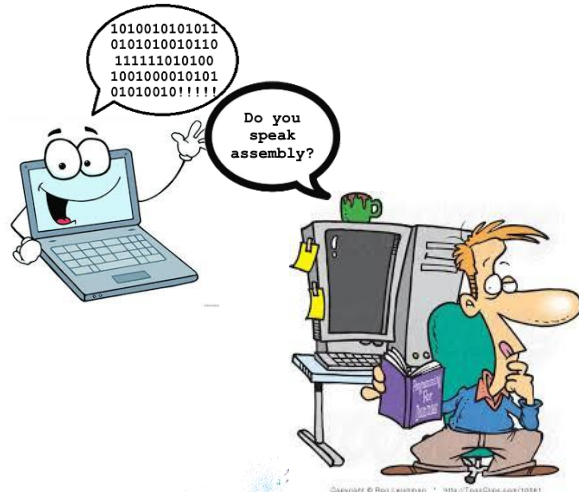
Can you even do one thing RIGHT you MORON!!
 How many more segfaults and core dumps will you give me
 Even I have a life!! you are killing me!!



```

;CLEAR SCREEN USING BIOS
CLR: MOV AX,0600H ;SCROLL SCREEN
      MOV BH,30 ;COLOUR
      MOV CX,0000 ;FROM
      MOV DX,184FH ;TO 24,79
      INT 10H ;CALL BIOS;
;INPUTTING OF A STRING
KEY: MOV AH,0AH ;INPUT REQUEST
      LEA DX,BUFFER ;POINT TO BUFFER WHERE STRING STORED
      INT 21H ;CALL DOS
      RET ;RETURN FROM SUBROUTINE TO MAIN PROGRAM;
; DISPLAY STRING TO SCREEN
SCR: MOV AH,09 ;DISPLAY REQUEST
      LEA DX,STRING ;POINT TO STRING
      INT 21H ;CALL DOS
      RET ;RETURN FROM THIS SUBROUTINE;
    
```

Assembly code



Assembler

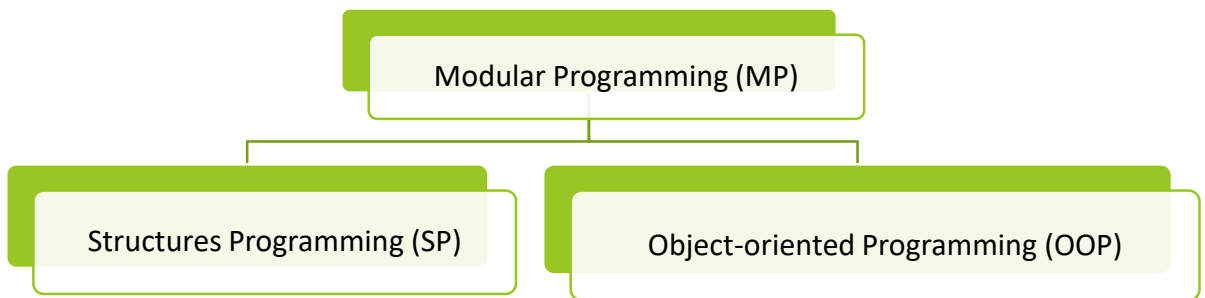
```

00010100101101010101010101010100010
111011010101010101010101110010100010110
00101001010100101110101110101101010
1001010010110101010101010101010110110
0110100100110010111010111010100010
0001000101011101010100010101011010
10101001010010101011101011010110101
000101001011010101010101010100010
    
```

Object code
 (machine language)

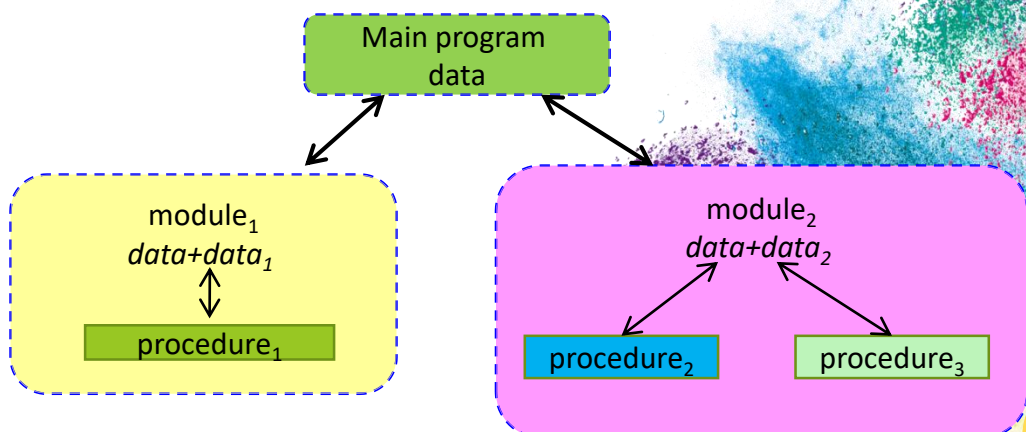
Types Of Programming And Structure Programming Methodology

- Various software design techniques have been introduced to improve the structure of programs for better understanding and efficiency. These techniques are as follows:
 - Modular programming (MP)
 - Structured programming (SP)
 - Object-oriented programming (OOP)
- The goal of MP, SP and OOP are similar, which is to facilitate the construction of large software programs and systems by decomposition into smaller pieces. These pieces are called subdivisions, modules, units, functions, procedure subroutines or objects.
- Modular programming (MP) refers to high-level decomposition of the entire program into modules. The modules are differentiated by an independent set of tasks or functions such as input/output, mathematical process or domain-specific processes. Example, a function to calculate average marks, mode mark, standard deviation or minimal mark.
- MP can use Structured programming (SP) approach or Object-oriented programming (OOP) approach. In SP approach, the modules are functions. In OOP approach, the modules are objects.



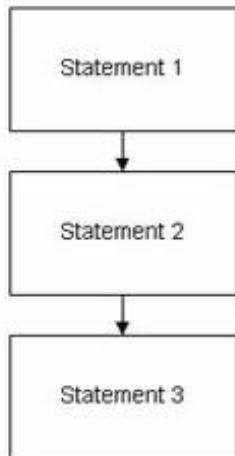
Modular programming

- Modular programming is a software design **technique** that increases the extent to which software is composed of separate, **interchangeable components** called modules by **breaking down** program functions into modules, each of which accomplishes one function and contains everything necessary to accomplish this.
- Conceptually, modules represent a separation, and improve maintainability by enforcing logical boundaries between components.



Structured programming

- Structured programming is a programming paradigm/**technique** aimed on improving the clarity, quality, and development time of a computer program **by making extensive use of subroutines, block structures and for and while loops.**
- Top-down approach.
- The most popular structured programming languages include **C,C++, Ada,** and **Pascal.**
- Split the task into modular/specific box to make program much more easier to develop

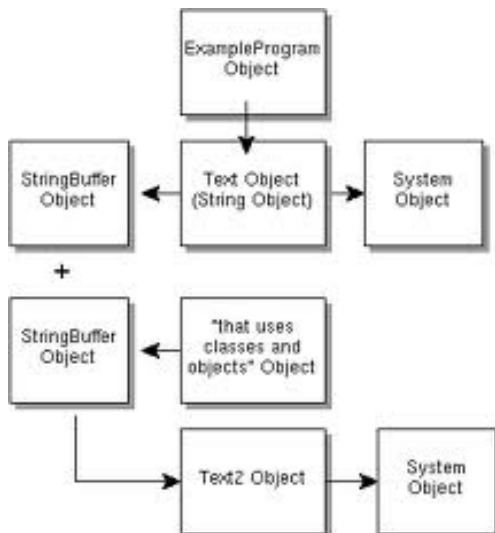


program fragments, the first is structured, while the second

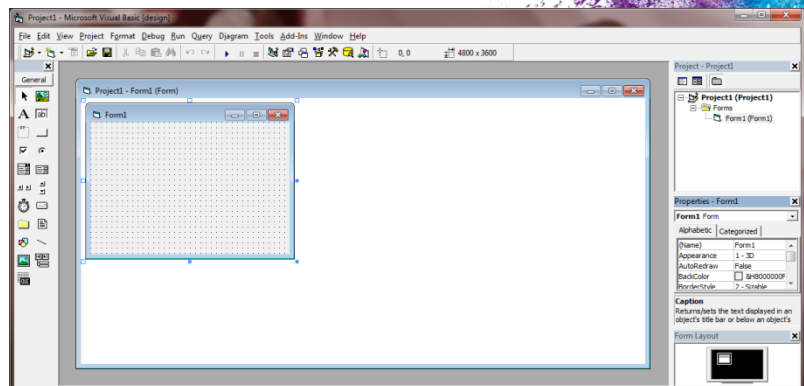
| | |
|---|---|
| <pre> Structured: IF x<=y THEN BEGIN z := y-x; q :=SQRT(z); END ELSE BEGIN z := x-y; q := -SQRT(z) END; WRITELN(z,q); </pre> | <pre> Unstructured: IF x>y THEN GOTO 2; z := y-x; q := SQRT(z); GOTO 1; 2: z:= x-y; q:=-SQRT(z); 1: writeln(z,q); </pre> |
|---|---|

Object-Oriented programming

- Programming techniques may include features such as **data abstraction, encapsulation, messaging, modularity, polymorphism, and inheritance.** Many modern programming languages now support OOP, at least as an option.
- The most popular object-oriented programming languages include **Java, Visual Basic, C#, C++,** and **Python.**



Everything in *OOP* is grouped as self sustainable "objects"



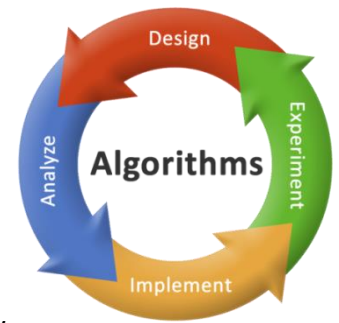
Comparison between Structured Programming and Object-Oriented Programming

| Structured Programming | Object Oriented Programming |
|--|---|
| Structured Programming is designed which focuses on process/ logical structure and then data required for that process . | Object Oriented Programming is designed which focuses on data . |
| Structured programming follows top-down approach . | Object oriented programming follows bottom-up approach . |
| Structured Programming is also known as Modular Programming and a subset of procedural programming language . | Object Oriented Programming supports inheritance, encapsulation, abstraction, polymorphism, etc. |
| In Structured Programming, Programs are divided into small self contained functions . | In Object Oriented Programming, Programs are divided into small entities called objects . |
| Structured Programming is less secure as there is no way of data hiding . | Object Oriented Programming is more secure as having data hiding feature . |
| Structured Programming can solve moderately complex programs. | Object Oriented Programming can solve any complex programs. |
| Structured Programming provides less reusability, more function dependency. | Object Oriented Programming provides more reusability, less function dependency. |
| Less abstraction and less flexibility. | More abstraction and more flexibility. |

Algorithm, Flow Chart and and pseudocode

Algorithm In Programming

- **Definition:** an algorithm is a step-by-step procedure to solve a given problem.
- An algorithm gives a solution to a particular problem as a well defined set of steps.
- A recipe in a cookbook is a good example of an algorithm. When a computer is used for solving a particular problem, the steps to the solution should be communicated to the computer.
- An algorithm is executed in a computer by combining lot of elementary operations such as additions and subtractions to perform more complex mathematical operations.



Define the algorithm in programming

- Let's say that you have a friend arriving at the airport, and your friend needs to get from the airport to your house. Here are four different algorithms that you might give your friend for getting to your home:
 - The taxi algorithm:
 - Go to the taxi stand.
 - Get in a taxi.
 - Give the driver my address.
 - The call-me algorithm:
 - When your plane arrives, call my cell phone.
 - Meet me outside.
 - I'll take u to my home!!
 - The rent-a-car algorithm:
 - Take the shuttle to the rental car place.
 - Rent a car.
 - Follow the directions to get to my house.
 - The bus algorithm:
 - Outside baggage claim, catch bus number 70.
 - Transfer to bus 14 on Main Street.
 - Get off on Elm street.
 - Walk two blocks north to my house.
- All four of these algorithms accomplish exactly the same goal,
- BUT each algorithm does it in completely different way.
- In computer programming, there are often many different ways - algorithms -- to accomplish any given task.
- Each algorithm has advantages and disadvantages in different situations.

Algorithm to add two numbers in C:

- 1) Start
- 2) Accept Number one
- 3) Accept Number two
- 4) Add both the numbers
- 5) Print the result.
- 6) End

Program to add two numbers in C:

```
/*Program to add two numbers in C.  
Programmer: Harsh Shah, Date: 29/6/13*/  
  
#include<stdio.h>  
#include<conio.h>  
  
void main()  
{  
int one, two, add; //declaring Variables  
printf("Enter first number - ");  
scanf("%d",&one); //Accepting Number one  
printf("Enter second number - ");  
scanf("%d",&two); //Accepting Number two  
add = one + two; //Adding both of them  
printf("The addition of numbers %d and %d is %d",one,two,add); //Printing the Result  
getch();  
}
```

Write an algorithm to add two numbers entered by user.

```
Step 1: Start  
Step 2: Declare variables num1, num2 and sum.  
Step 3: Read values num1 and num2.  
Step 4: Add num1 and num2 and assign the result to sum.  
        sum=num1+num2  
Step 5: Display sum  
Step 6: Stop
```

Write an algorithm to find the largest among three different numbers entered by user.

```
Step 1: Start  
Step 2: Declare variables a,b and c.  
Step 3: Read variables a,b and c.  
Step 4: If a>b  
        If a>c  
            Display a is the largest number.  
        Else  
            Display c is the largest number.  
    Else  
        If b>c  
            Display b is the largest number.  
        Else  
            Display c is the greatest number.  
Step 5: Stop
```


There are two algorithm representations:

- (a) Pseudocode – An English-like list of instructions.
- (b) Flow chart – Graphical notation for easy reading.

Pseudocode In Programming

- Pseudocode is one of the methods that could be used to represent an algorithm. It is not written in a specific syntax that is used by a programming language and therefore cannot be executed in a computer.
- There are lots of formats used for writing pseudocode and most of them borrow some of the structures from popular programming languages such as C, Lisp, FORTRAN, etc.

Pseudocode in C programming Adding 2 numbers


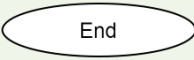


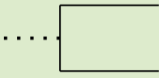



1. Initialize total to zero
2. Initialize number1
3. Initialize number2
4. Input number1
5. Input number2
6. Add number1 and number2 = total
7. Display the result.

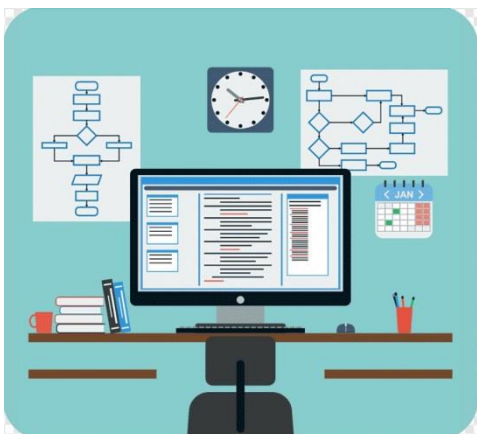
-
- 1 *Set total to zero*
 - 2 *Set grade counter to one*
 - 3
 - 4 *While grade counter is less than or equal to ten*
 - 5 *Input the next grade*
 - 6 *Add the grade into the total*
 - 7 *Add one to the grade counter*
 - 8
 - 9 *Set the class average to the total divided by ten*
 - 10 *Print the class average*
-

Fig. 3.5 | Pseudocode algorithm that uses counter-controlled repetition to solve the class average problem.

Flow chart In Programming

- Graphic representation of algorithm which is consist of geometric symbols.
- Symbols are connected by line of arrows called flow lines which indicate the direction of flow of processes or activities.
- The shape of the symbol indicates the type of operation that is to occur.
- Flowchart should flow from the top of the page to the bottom. The symbols used in flowchart are standardized.

| Symbol | Name | Description |
|---|----------------|--|
|  | Terminal | Marks the beginning of a program |
|  | Terminal | Marks the ending of a program |
|  | Input / Output | To enter data or to display data |
|  | Process | A set of instructions to transform input into output |
|  | Annotation | To put comments or additional information |
|  | Connector | Entry point or exit point to another part of the flow chart |
|  | Decision | Condition determining which of two separate paths to follow |
|  | Flow Line | Connector between flow chart nodes indicating sequence of the steps. The arrow head indicates the sequence direction |



The Advantages And Disadvantages Of Flow chart

| Advantages | Disadvantages |
|---|---|
| <p>Clear: It graphically shows the logic of an algorithm. It is easier to visualize a flow chart than to read code. One can analyse a flow chart like reading a map. It is easy to analyse bad relationships between components or to detect a logical path that is not complete.</p> | <p>Clumsy: One has to be familiar with the notations, and drawing notations require some effort to produce.</p> |
| <p>Standard notation: The notations are standard, and are therefore easy to recognize by a wider audience. Flow charts include standard notations for selection structures and looping structures.</p> | <p>Complex drawing: A flow chart may take up a big drawing space. It's important to be aware of the space utilized for drawing the notations. Otherwise, the drawing may become clumsy and complex.</p> |
| <p>Logical accuracy: Flow charts provide a positive constraint to the programmer who drafts the algorithm, ensuring an algorithm is defined using only a specific set of notations. This way, it is not possible to accidentally include a design that cannot be implemented as code. Furthermore, using notation forces one to analyse the solution instead of stopping at a basic or abstract level, since it is not possible to represent an abstract step in notation.</p> | <p>Challenging to translate: Flow charts may not be as convenient as pseudocode when being translated to corresponding programming code. This is because a flow chart is in drawing form while a pseudocode's form is closer to actual programming code.</p> |

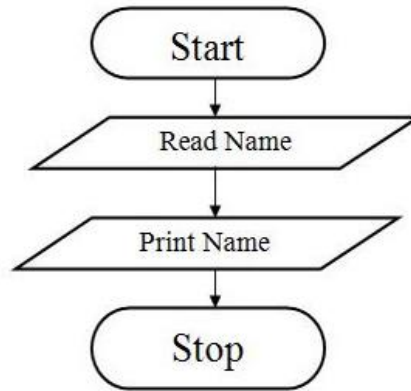
Construct flowchart for the given problem.

Algorithm and flow chart to read the name and print the name.

Algorithm

- Step 1 : Start
- Step 2 : Read input name
- Step 3 : Print name
- Step 4 : Stop

Flow Chart

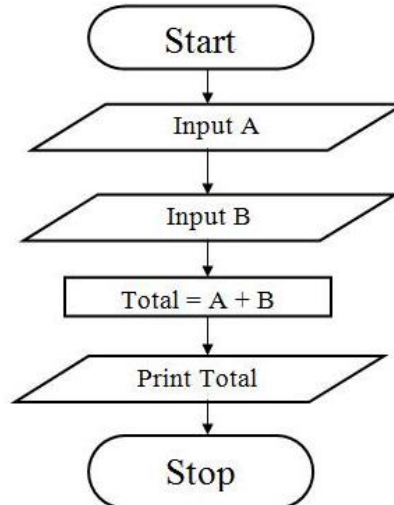


Algorithm and flow chart to add two numbers.

Algorithm

- Step 1 : Start
- Step 2 : Input first number A
- Step 3 : Input second number B
- Step 4 : Add the two numbers and store it in total
- Step 5 : Print Total
- Step 6 : Stop

Flow Chart

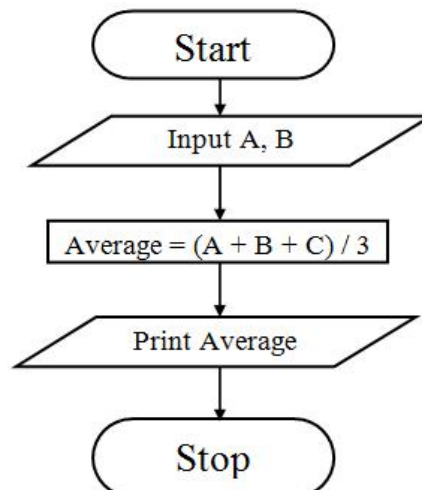


Algorithm and flow chart to find the average of three numbers.

Algorithm

- Step1 : Start
- Step 2 : Enter Three Numbers A, Band C
- Step 3 : Compute Average = $(A+B+C)/3$
- Step 4 : Print Average
- Step 5 : Stop

Flow Chart



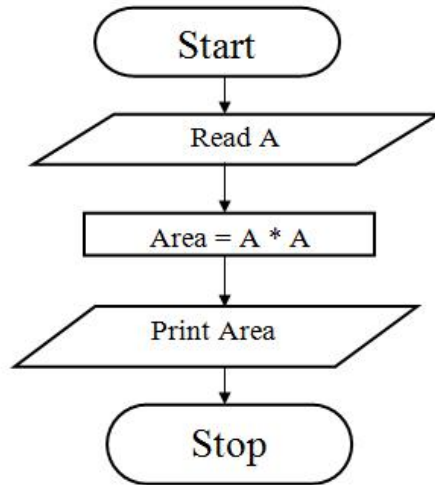
Construct flowchart for the given problem.

Algorithm and a flow chart to calculate area of square.

Algorithm

- Step 1 : Start
- Step 2 : Read value for a (side)
- Step 3 : [Compute] $Area = A * A$
- Step 4 : Output Area
- Step 5 : Stop

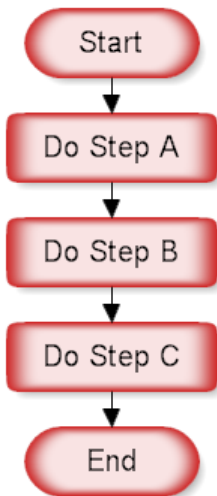
Flow Chart



Apply flowchart for the following

a. Sequence structure.

In a computer program or an algorithm, sequence involves simple steps which are to be executed one after the other. The steps are executed in the same order in which they are written.



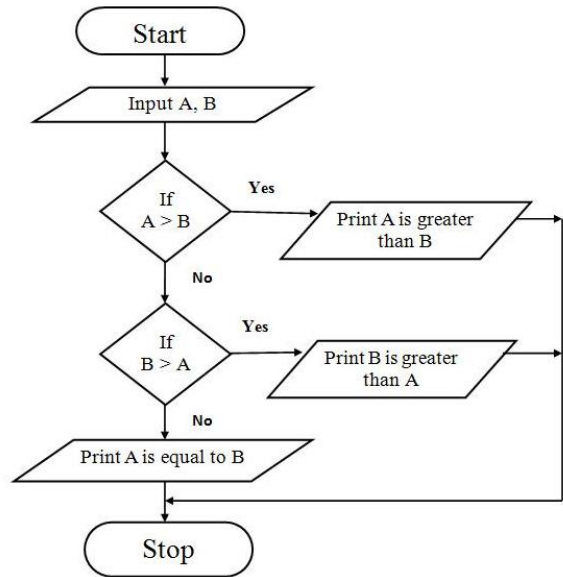
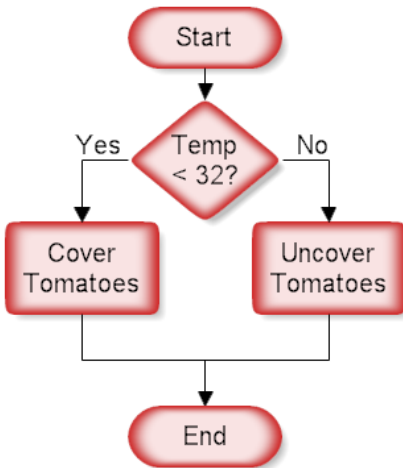
An Example Using Sequence

Problem: Write a set of instructions that describe how to make a pot of tea.

| Pseudocode | Flowchart |
|--|-----------|
| BEGIN fill a kettle with water boil the water in the kettle put the tea leaves in the pot pour boiling water in the pot END | |

b. Selection Structure.

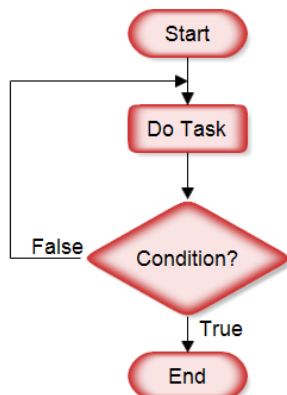
Selection is used in a computer program or algorithm to determine which particular step or set of steps is to be executed.



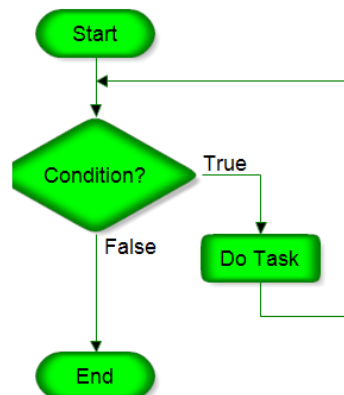
c. Looping Structure.

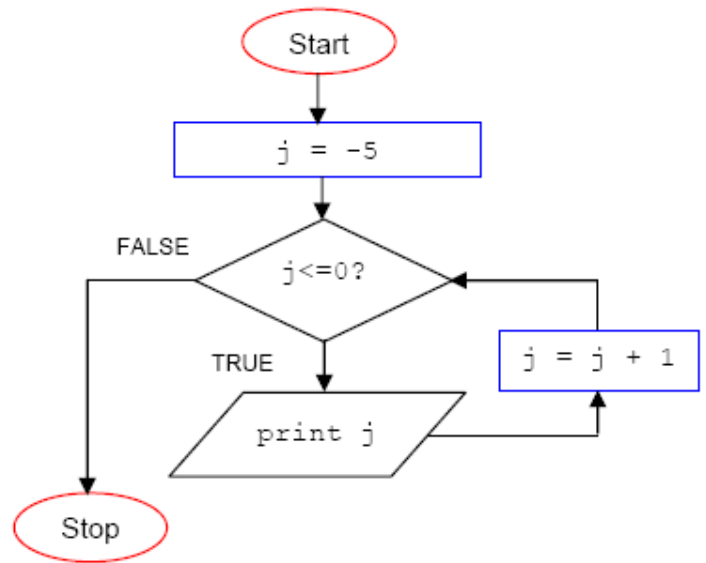
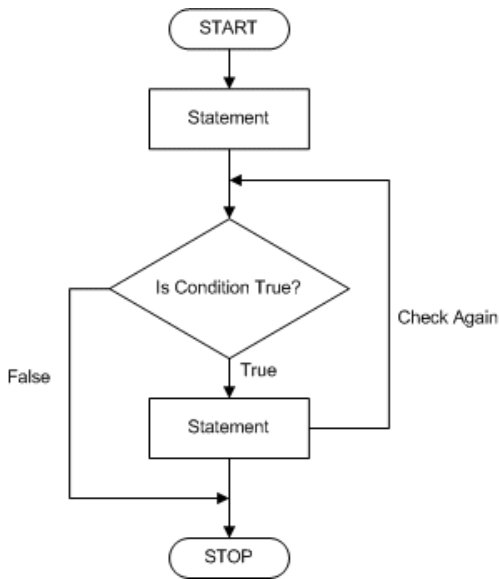
- Repetition allows for a portion of an algorithm or computer program to be done any number of times dependent on some condition being met.
- An occurrence of repetition is usually known as a loop.
- An essential feature of repetition is that each loop has a termination condition to stop the repetition, or the obvious outcome is that the loop never completes execution (an infinite loop).

Do While Loop



Repeat Until Loop

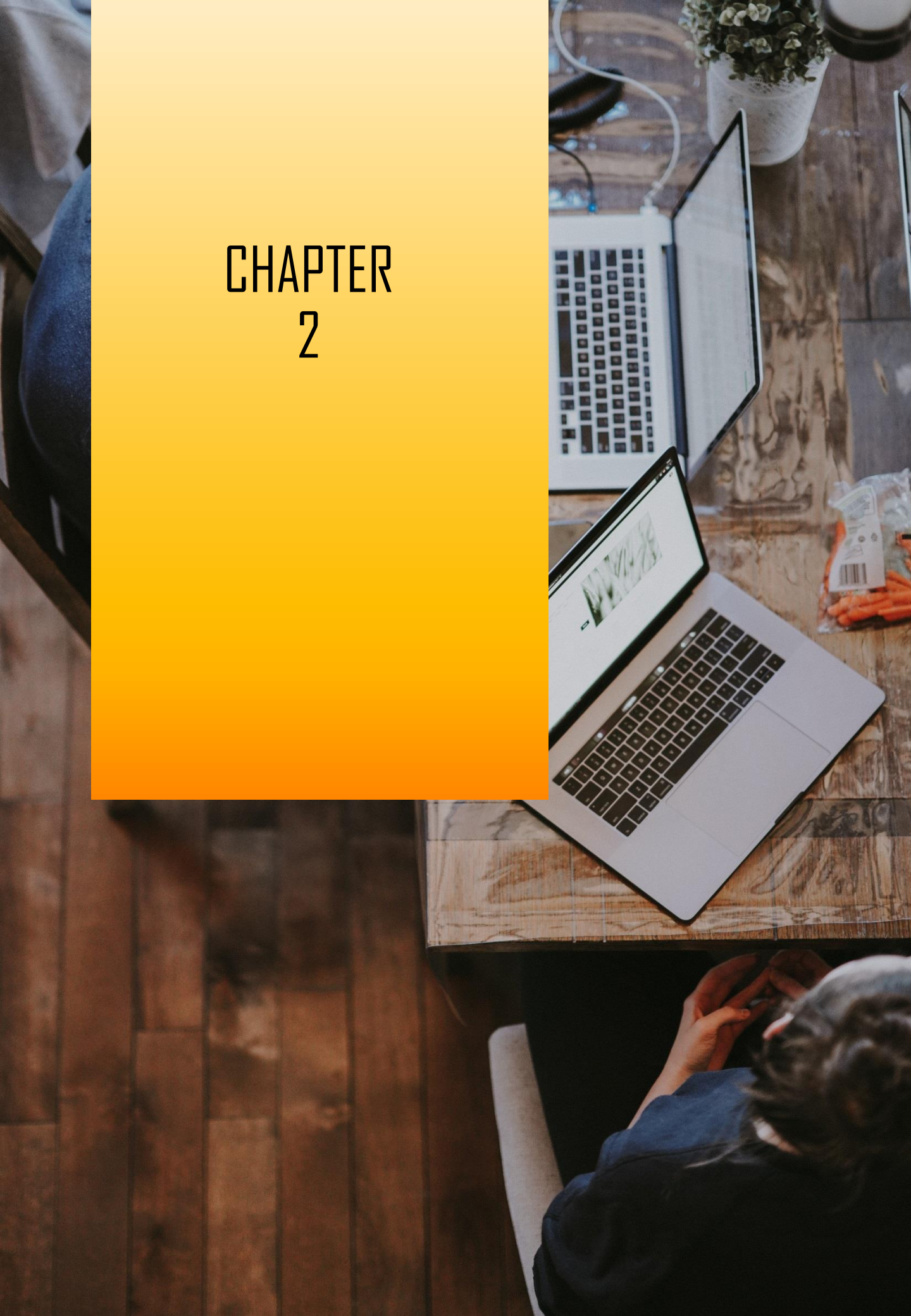




Questions

1. Define 'programming language'.
2. Can programming be used to solve all types of problems? Why and why not?
3. Describe how Computational Thinking can be used to solve programming problems.
4. Identify THREE (3) advantages offered by high-level languages (HLLs) over low-level languages (LLS).
5. Machine language is fast but not programmer-friendly. Discuss.
6. (a) Describe how 'structured programming' is different from 'spaghetti coding'. (b) Between Fortran and C, which programming language is more structured? Why?
7. Why is machine-independent language usually preferred over machine independent language?
8. Before a program can be executed, it has to be compiled from its source code. Describe the difference between executable file and source code.

CHAPTER 2



Fundamentals of C Language



- **Variables**

- A variable **is just a named area of storage** that **can hold a single value** (numeric or character).
- they **represent** some **unknown**.
- Programming language C has two main variable types
 - **Local Variables**
 - **Global Variables**

Example: Local vs. Global

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

```
void print_number(void);
```

```
int p;
```

p is declared outside of all functions. So, it is a global variable.

```
void main (void)
```

```
{
```

```
int q = 5;
```

```
printf("q=%d", q);
```

```
p=10;
```

```
print_number();
```

```
}
```

q is declared inside the function main. So, it is a local variable to the function.

```
void print_number(void)
```

```
{
```

```
printf("%d", p);
```

```
q = q + 5;
```

```
}
```

p can be used anywhere

Error! q can only be used in the function main, because it is a local variable

```

1 #include <stdio.h>
2 #include <conio.h>
3
4 void main ()
5 {
6     int A;
7     int B;
8     int C;
9     int D;
10    float Eagle;
11    float Ferry;
12    char Game;
13    char House;

```

a variable must be declared before it can be used !!

This is Variable!!



• **Constants:**

- the values that **never change**,
- Constants can be very useful in C programming whenever you have any value that is repeated in your program.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define PI 3.1415926
5
6 void main ()
7 {

```

```

#include <stdio.h>

#define PIE 3.14

float ComputeVolume(float r, float h, float ans, float r2);

int main() {
    float radius, height, ans, r2;
    printf("Radius: ");
    scanf("%g", &radius);
    printf("Height: ");
    scanf("%g", &height);
    ans = ComputeVolume(radius, height, ans, r2);
    printf("Volume: %g\n", ans);
    return 0;
}

float ComputeVolume(float r, float h, float ans, float r2)
{
    r2 = r*r;
    ans = (1/3)*PIE*r2*h;
    return ans;
}

```



- **Variables and Constants**

- Name or identifier can be declared with constant,

This is Variable.!!
and it can be declared
with constant...

```
4 void main()  
5 {  
6     int A = 21;  
7     int B = 1;  
8     int C = 3;  
9     int D = 810;  
10    float Eagle = 65.4;  
11    float Ferry = 45.124;  
12    char Game = 'g';  
13    char House = 'p';
```

- **Rules for Variables & Constants**

- may be given representations containing multiple characters. But there are **rules** for these representations..
 - May only consist of letters, digits, and underscores
 - May be as long as you like, but only the first 31 characters are significant
 - May not begin with a number
 - May not be a C reserved word (keyword)
 - May only consist combination of letters, digits, and underscores
 - May be as long as you like, but only the first 31 characters are significant

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3  
4 #define PI 3.1415926  
5  
6 void main()  
7 {  
8     int umur;  
9     int umur_saya;  
10    int umur_remaja_1;  
11    int umur_remaja_2;  
12    int umur_remaja_3;  
13    int umur_masa_saya_remaja;  
14
```

just
another
example



- Rules for Variables & Constants
 - May not begin with a number
 - May not be a **C reserved word** (keyword)

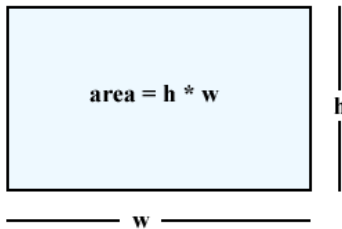
```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define PI 3.1415926
5
6 void main()
7 {
8     int 3_hari;
9     int 455_baris;
10    int int;
11

```



- Rules for Variables & Constants
 - To **declare** a variable means to create a memory space for the variable depending on the data type used and associate the memory location with the variable name.
 - The **shortest** variable name is a letter of the alphabet.
 - Variables are typically in lowercase. (All of C is lowercase for the most part.) They can contain letters and numbers.
 - AGAIN!!..You **should not begin** a variable name with a number. They can contain numbers, but you begin it with a letter.

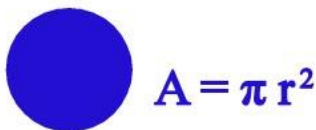


```

int h;
int w;
int area;

```

Example:
 h= 2 meter
 W = 4 meter
 area = ? output



If r is integer??

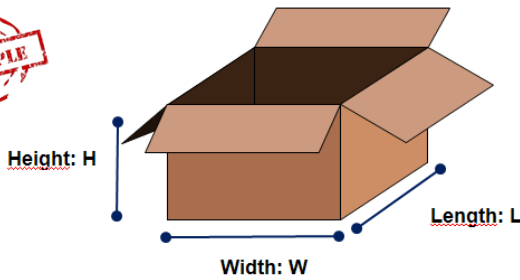
```
int r;
```

Example r = 8 meter

If all value is floating point??

```
float r;
float A;
```

Example
r = 8.24 meter
 A = π x 8.24 x 8.24



If all value is floating point??

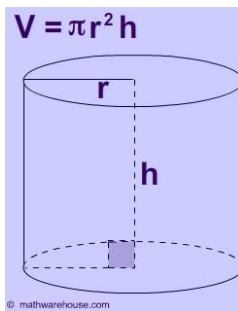
float height;
float width;
float length;

If all value is integer??

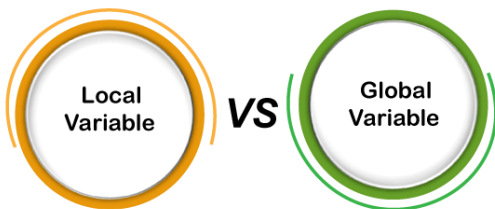
int height;
int width;
int length;

Questions

Write down the variable name with data type



Global variables and Local variables



- Variables are classified into Global variables and Local variables based on their scope.
- The main difference between Global and local variables is that global variables can be accessed globally in the entire program, whereas local variables can be accessed only within the function or block in which they are defined.
- The scope of variables can be defined with their declaration, and variables are declared mainly in two ways:
 - **Global Variable:** Outside of all the functions
 - **Local Variable:** Within a function block

Global variables Example:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int A;
5
6 void main()
7 {
8     int B;
9     int C;
10    int D;
```

Local variables Example:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int A;
5
6 void main()
7 {
8     int B;
9     int C;
10    int D;
```

Advantages and Disadvantages of Global and Local Variable

| | Global variable | Local variable |
|---------------|---|--|
| Advantages | <ul style="list-style-type: none">• Global variables can be accessed by all the functions present in the program.• Only a single declaration is required.• Very useful if all the functions are accessing the same data. | <ul style="list-style-type: none">• The value of a global variable can be changed accidentally as it can be used by any function in the program.• If we use a large number of global variables, then there is a high chance of error generation in the program. |
| Disadvantages | <ul style="list-style-type: none">• The same name of a local variable can be used in different functions as it is only recognized by the function in which it is declared.• Local variables use memory only for the limited time when the function is executed; after that same memory location can be reused. | <ul style="list-style-type: none">• The scope of the local variable is limited to its function only and cannot be used by other functions.• Data sharing by the local variable is not allowed. |

Comparison Chart Between Global Variable and Local Variable

| Global Variable | Local Variable |
|--|---|
| Global variables are declared outside all the function blocks. | Local Variables are declared within a function block. |
| The scope remains throughout the program. | The scope is limited and remains within the function only in which they are declared. |
| Any change in global variable affects the whole program, wherever it is being used. | Any change in the local variable does not affect other functions of the program. |
| A global variable exists in the program for the entire time the program is executed. | A local variable is created when the function is executed, and once the execution is finished, the variable is destroyed. |
| It can be accessed throughout the program by all the functions present in the program. | It can only be accessed by the function statements in which it is declared and not by the other functions. |
| If the global variable is not initialized, it takes zero by default. | If the local variable is not initialized, it takes the garbage value by default. |
| Global variables are stored in the data segment of memory. | Local variables are stored in a stack in memory. |
| We cannot declare many variables with the same name. | We can declare various variables with the same name but in other functions. |

Keywords in C Programming Language :

- Keywords are those words whose meaning is **already defined by Compiler**
- **Cannot** be used as **Variable Name**
- There are **32** Keywords in C
- C Keywords are also called as Reserved words

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

- Use keywords in programmes.

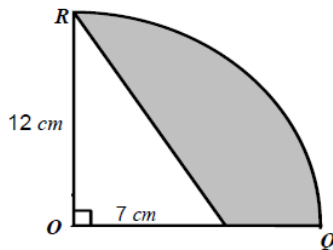
```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void main()
5 {
6     int Temperature;
7     float Height;
8     double Volume;
9     char Name;
10    unsigned Distance;
11
12

```

Exercise: determine the variables & constants

Diagram shows sector of circle ORQ with centre O .



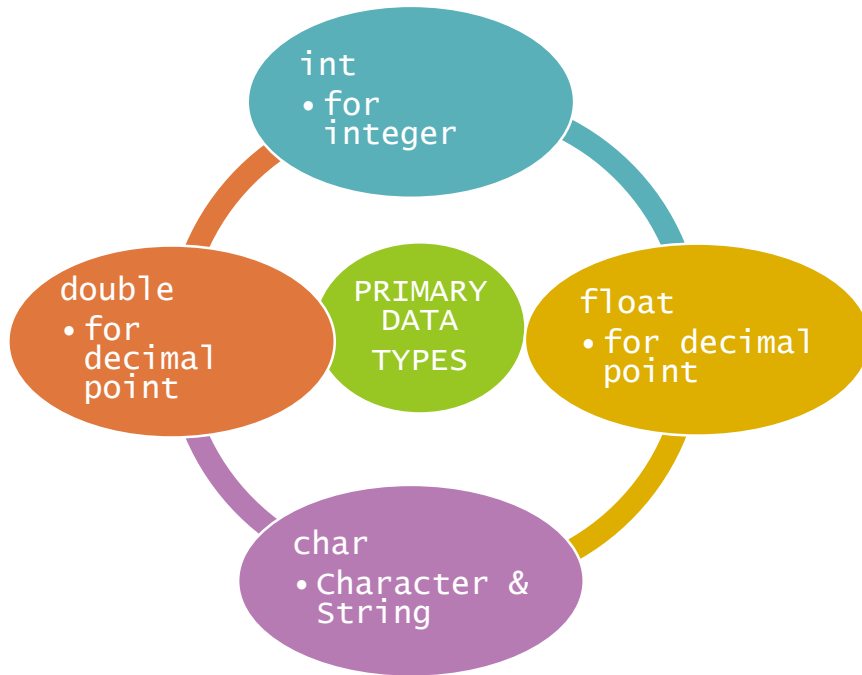
By using $\pi = \frac{22}{7}$, calculate

- the perimeter for the whole diagram in cm,
- area of the shaded region in cm^2 .

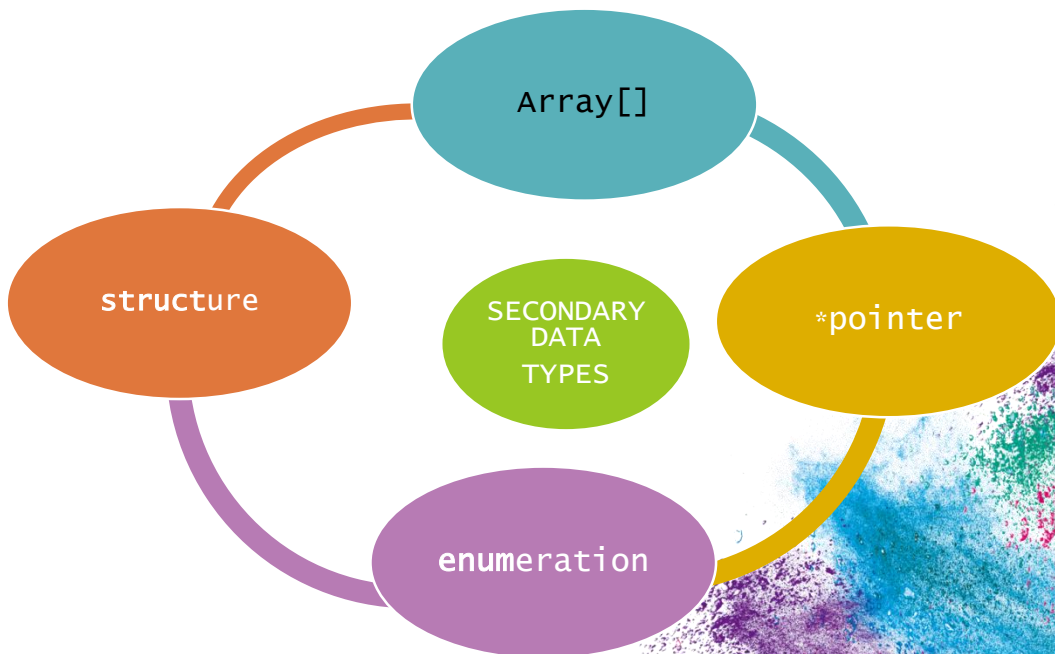
The basic data types in C.

- A program usually contains **different types of data types** (integer, float, character etc.) and **need to store the values** being used in the program.
- C language is rich of data types. A C programmer has **to employ proper data type as per his/her requirements**. C language provides various data types for holding different kinds of values.
- C language provides various data types for holding different kinds of values.
- There are several integer data types, a character data type, floating point data types for holding real numbers and more.
- C has a concept of 'data types' which are used to define a variable before its use. The definition of a variable will assign storage for the variable and define the type of data that will be held in the location.
- C has different data types for different types of data and can be broadly classified as:
 - Primary Data Types
 - Secondary Data Types

Primary Data Types

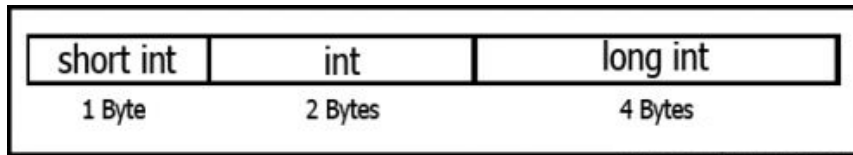


Secondary Data Types

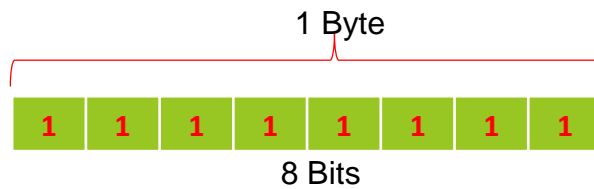


Integer types

- C provides several standard **integer types**, from small magnitude to large magnitude numbers: char, short int, int, long int, long long int.
- Each type can be signed or unsigned. Signed types can represent positive and negative numbers while unsigned can represent zero and positive numbers.
- C provides several standard integer types, from small magnitude to large magnitude numbers: short int; int; long int; long long int;

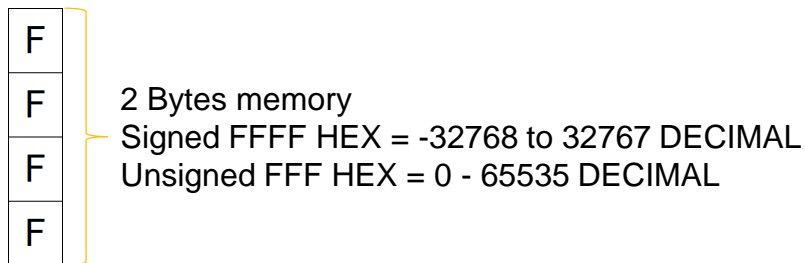


1 Byte = 8 Bits

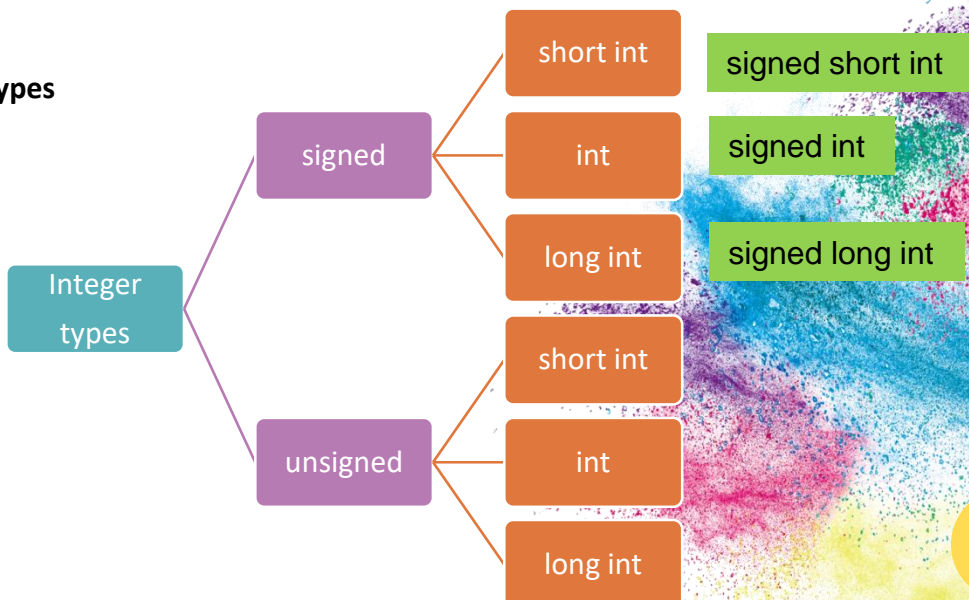


Numeric: Integer types

- Generally an integer (int) occupies 2 bytes memory space and its value range limited to -32768 to +32767 (that is, -2¹⁵ to +2¹⁵-1).
- A signed integer use one bit for storing sign and rest 15 bits for number.



Integer types



Integer types

| Data Type | Range | Bytes | Format |
|--------------------|----------------------------|-------|--------|
| signed char | -128 to + 127 | 1 | %c |
| unsigned char | 0 to 255 | 1 | %c |
| short signed int | -32768 to +32767 | 2 | %d |
| short unsigned int | 0 to 65535 | 2 | %u |
| signed int | -32768 to +32767 | 2 | %d |
| unsigned int | 0 to 65535 | 2 | %u |
| long signed int | -2147483648 to +2147483647 | 4 | %ld |
| long unsigned int | 0 to 4294967295 | 4 | %lu |
| float | -3.4e38 to +3.4e38 | 4 | %f |
| double | -1.7e308 to +1.7e308 | 8 | %lf |
| long double | -1.7e4932 to +1.7e4932 | 10 | %Lf |

Note: The sizes and ranges of int, short and long are compiler dependent. Sizes in this figure are for 16-bit compiler.

Syntax:

int <variable name>;

int num1;

short int num2;

long int num3;

Example: 5, 6, 100, 2500.

Numeric: Floating point types

- The float data type is used to store fractional numbers (real numbers) with **6 digits of precision**. Floating point numbers are denoted by the keyword float. (eg. 0.000001)
- When the accuracy of the floating point number is insufficient, we can use the **double to define the number**. The double is same as float but with longer precision and takes double space (**8 bytes**) than float.
- To extend the precision further we can **use long double** which occupies 10 bytes of memory space

| | | |
|---------|---------|-------------|
| float | double | long double |
| 4 Bytes | 8 Bytes | 10 Bytes |

Syntax:

```
float <variable name>; like  
float num1;  
double num2;  
long double num3;
```

Example: 9.125, 3.1254.

Floating Point Data Type Memory Allocation

Character:

- Character type variable **can hold a single character**. As there are signed and unsigned int (either short or long), in the same way there are signed and unsigned chars;
- Both occupy **1 byte each**, but having different ranges. Unsigned characters have values **between 0 and 255**, signed characters have values from -128 to 127.

Syntax:

```
char <variable name>; like  
char ch = 'a';
```

Example: a, b, g, S, j.

String:

- **A string in C is an array** of char values terminated by a special null character value '\0'. For example, here is a statically declared string that is initialized to "bye":

```
char str[4]; // need space for chars in str, plus for terminating '\0' char  
str[0] = 'b';  
str[1] = 'y';  
str[2] = 'e';  
str[3] = '\0';  
printf("%s\n", str); // prints bye to stdout
```

Data type and conversion specification

| Data type | printf conversion specification | scanf conversion specification |
|-------------------|---------------------------------|--------------------------------|
| long double | %Lf | %Lf |
| double | %f | %lf |
| float | %f | %f |
| unsigned long int | %lu | %lu |
| long int | %ld | %ld |
| Unsigned int | %u | %u |
| int | %d | %d |
| Unsigned short | %hu | %hu |
| short | %hd | %hd |
| char | %c | %c |

Exercise:

Determine what types of data to used if the given number is?

Question:

Number is

234

23.122

2

'a'

"Dad"

2314.1121231

Input-Process-Output (IPO) analysis

- Understand the input, process and output before start to code. Begin the coding with Input-Process-Output (IPO) analysis.
- Use IPO analysis to understand the problem statement clearly. This is done by breaking a problem statement into the following components:
 - a) **INPUT:** Find out what the inputs are. Inputs are data inserted into the program before it begins processing. Identifying input is not as easy as it sounds, so please pay attention to the problem statement.
 - b) **PROCESS:** This describes how to process the INPUT into a desired OUTPUT. Processing includes using mathematical formula, word processing steps, or computer logic to transform the value of the input.
 - c) **OUTPUT:** Output is the expected result after processing. This is usually the displayed output on the computer screen or data saved into a text file.
- It may be handy to create an IPO chart when analysing the problem statement. An IPO chart is a three-column chart with Input, Process and Output as the column headers. See the example above:

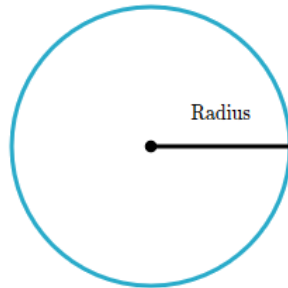
Example:

| INPUT | PROCESS | OUTPUT |
|--|--|---|
| <ul style="list-style-type: none">• Enter the first number• Enter the second number• Choose the "addition" operation | <ul style="list-style-type: none">• Store the first number in memory• Store the second number in memory• Perform the operation on the two numbers stored in memory | <ul style="list-style-type: none">• Result of operation |

- To produce an IPO chart, one should first analyse the problem statement or case. An IPO analysis can be done systematically by asking the following questions:
 - **Step 1:** Ask yourself, "What is the expected output?"
 - **Step 2:** Ask yourself, "What is the process required to get the output?" This can be any formula or conversion steps given by the problem. If no formula or steps are given, determine the appropriate formula or steps from your general knowledge or from research.
 - **Step 3:** Ask yourself, "What are the inputs the program needs from the user?"

Questions

- Given the following problem statement, conduct an IPO analysis to produce the IPO chart:
- Problem statement: “ A shape program reads the radius and computes the circumference and area of a circle.



IPO analysis:

- The program ‘display circumference and area’, indicating the output of the program.
- The formula to compute the circumference and area are not given. They are standard formula ($A = \pi r^2$).
- The formula requires inputs from users. The phrase ‘read radius’ indicates that the radius is the input.

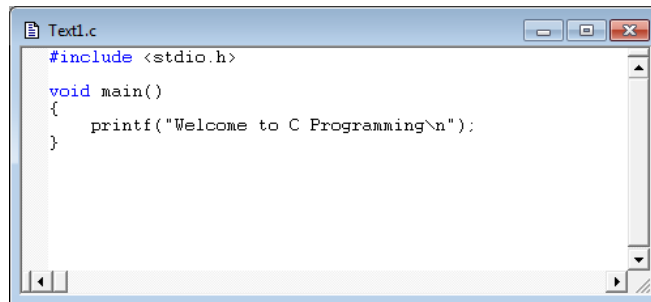
IPO chart:

| Input | Process | Output |
|--|--|--|
| <ul style="list-style-type: none">• Radius | <ol style="list-style-type: none">1. Get radius2. Circumference = $2 \times 3.142 \times$ radius3. Area = $3.142 \times$ radius \times radius4. Display circumference and area | <ul style="list-style-type: none">• Circumference• Area |

Six (6) phase of C development environment:

1. Editor - **Software packages** for the C/C++ integrated program development environments such as Microsoft Visual Studio have editors that are integrated into the programming environment.

C program file names should end with **the .c extension**.



```
Text1.c
#include <stdio.h>

void main()
{
    printf("Welcome to C Programming\n");
}
```

2. Preprocessor

- In a C system, a preprocessor program **executes automatically before** the compiler's translation phase begins.
- commands called preprocessor directives, which indicate that **certain manipulations** are to be performed on the program before compilation.

3. Compiler

- the compiler translates the C program into machine-language code. (**High language → Machine language**)

4. Linker

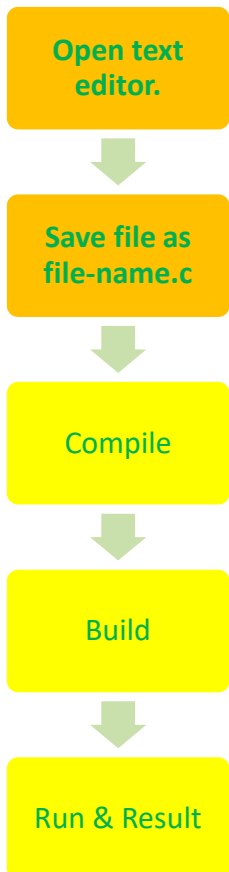
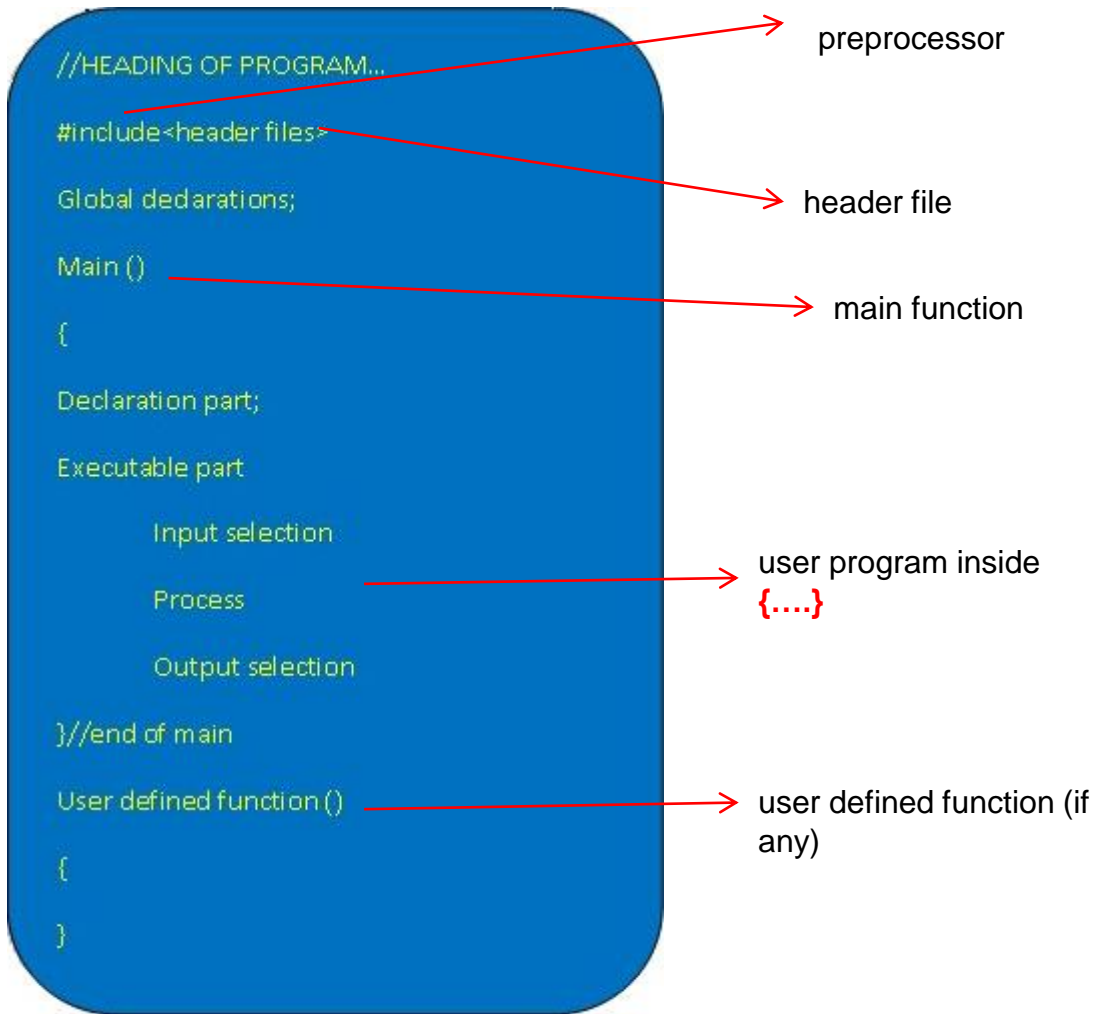
- contain **references** to functions defined elsewhere, such as in the standard libraries (**e.g <stdio.h>**) or in the private libraries of groups of programmers working on a particular project.

5. Loading

- Before a program can be executed, the program must **first be placed in memory**. This is done by the loader, which takes the executable 'image' from disk and transfers it to memory. Additional components from shared libraries that support the program are also loaded (if).

6. CPU

- Finally, the computer, under the control of its CPU, executes the program one instruction at a time. (***.exe**)



```

1  /* Fig. 2.1: fig02_01.c
2     A first program in C */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main()
7  {
8     printf( "welcome to C!\n" );
9
10 return 0; /* indicate that program ended
           successfully */
11 } // end function main.

```

Welcome to C!

Try this....and observe..

```
1 /* Fig. 2.3: fig02_03.c
2    Printing on one line with two printf statements */
3 #include <stdio.h>
4
5 /* function main begins program execution */
6 int main( void )
7 {
8     printf( "Welcome " );
9     printf( "to C!\n" );
10
11     return 0; /* indicate that program ended successfully */
12 }
```

```
Welcome to C!
```

Try this..

```
1 /* Fig. 2.4: fig02_04.c
2    Printing multiple lines with a single printf */
3 #include <stdio.h>
4
5 /* function main begins program execution */
6 int main( void )
7 {
8     printf( "Welcome\n\tto\n\tC!\n" );
9
10     return 0; /* indicate that program ended successfully */
11 }
```

```
Welcome
to
C!
```

put **\n**

- Notice that the characters `\n` were not printed on the screen. The backslash (`\`) is called an escape character..

try to change `\n` → `\a`.. observe what happen



| Escape sequence | Description |
|-----------------|---|
| <code>\n</code> | Newline. Position the cursor at the beginning of the next line. |
| <code>\t</code> | Horizontal tab. Move the cursor to the next tab stop. |
| <code>\a</code> | Alert. Sound the system bell. |
| <code>\\</code> | Backslash. Insert a backslash character in a string. |
| <code>\"</code> | Double quote. Insert a double-quote character in a string. |

How to view number..try edit, compile and run!

```

1  /*This program is show
2  how to..
3  a) initialize the number,
4  b) print the number
5  */
6
7  #include <stdio.h>
8  #include <stdlib.h> //<-this one no need to write..
9
10 int main()
11 {
12     int number_1;
13     int number_2,number_3,number_4;
14
15     number_1 = 12;
16     number_2 = 23;
17     number_3 = 2;
18     number_4 = 18;
19
20     printf("the number is %d\n",number_3);
21
22     system("pause");//<-this one no need to write..
23     return 0;
24 }
25 |

```

int is refer to the integer

try to change this..

number_3 → %d

Value change → %d change

How to view number..try edit, compile and run!

```
1 /*This program is show
2 how to..
3 a) initialize the number,
4 b) print the number
5 */
6
7 #include <stdio.h>
8 #include <stdlib.h> //<-this one no need to write..
9
10 int main()
11 {
12     int number_1;
13     int number_2,number_3,number_4;
14
15     number_1 = 12;
16     number_2 = 23;
17     number_3 = 2;
18     number_4 = 18;
19
20     printf("the number is %d\n",number_3);
21     printf("the number is %d\n",number_1);
22     printf("the number is %d\n",number_2);
23
24     system("pause");//<-this one no need to write..
25     return 0;
26 }
27
```

%d just for integer!!

try to change this..



number_3 → %d

Value change → %d change

The Structure Of C Programs

A C program basically consists of the following parts :

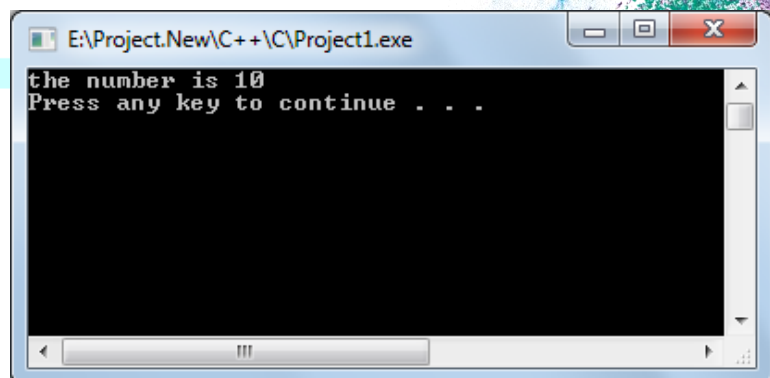
- Preprocessor directives
 - Functions
 - Variables
 - Statements & Expressions
 - Comments
- **Preprocessor directives.**
 - Text **must be start** with **#include**<header.h>
 - **#define** is one of the preprocessor directives.

```
1  /* Fig. 2.1: fig02_01.c
2     A first program in C */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main()
7  {
8     printf( "welcome to C!\n" );
9
10     return 0; /* indicate that program ended successfully */
11
12 } // end function main.
```

Welcome to C!

- **#define** is one of the preprocessor directives.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define Max 10
5
6 int main()
7 {
8     printf("the number is %d\n",Max);
9     system("pause");
10    return 0;
11 }
12
13
```



- The C preprocessor (cpp) is the preprocessor for the C and C++ computer programming languages.
- The preprocessor handles directives for
 - source file inclusion (**#include**)
 - macro definitions (**#define**), and
 - conditional inclusion (**#if**).
- **Header file** is a file that **allows programmers to separate** certain elements of a program's source code into reusable files.
- These are collectively known as the standard libraries and include:
 - string.h : for string handling
 - **stdlib.h : for some miscellaneous functions**
 - **stdio.h : standardized input and output**
 - math.h : mathematical functions

```

E:\...C\Project1\Text2.c
#include <stdio.h>
#include <stdlib.h>

void main()
{
    printf("Test\n");
    system("pause");
    system("color fc");
    system("pause");
}
  
```

Compile & Run

```

"E:\Project.New\C\Project1\Debug\Text2...
Test
Press any key to continue . . . _
  
```

```

"E:\Project.New\C\Project1\Debug\Text2...
Test
Press any key to continue . . .
Press any key to continue . . .
  
```

Color attributes are specified by TWO hex digits -- the first corresponds to the background; the second the foreground. Each digit can be any of the following values:

- | | |
|------------|------------------|
| 0 = Black | 8 = Gray |
| 1 = Blue | 9 = Light Blue |
| 2 = Green | A = Light Green |
| 3 = Aqua | B = Light Aqua |
| 4 = Red | C = Light Red |
| 5 = Purple | D = Light Purple |
| 6 = Yellow | E = Light Yellow |
| 7 = White | F = Bright White |

under header **stdlib.h**

- These are collectively known as the standard libraries and include:
 - `ctype.h` : for character handling
 - `conio.h` : library functions for performing "console input and output" from a program
- Tells computer to load contents of a certain file;
 - `<stdio.h>` allows standard input/output operations

stdio.h (standard input output header) example.

| | |
|---------------------|---|
| <code>printf</code> | prints formatted byte/wchar_t output to stdout, |
| <code>scanf</code> | reads a byte string from stdin |
| <code>puts</code> | writes a byte string to stdout |
| <code>gets</code> | reads a byte string from stdin |

- `conio.h` is a C header file used in **old MS-DOS** compilers to create text user interfaces.;
 - `<conio.h>` allows console input/output operations

conio.h (console input output header) example.

| | |
|----------------------|--|
| <code>getch</code> | Reads a character directly from the console without buffer |
| <code>putch</code> | Writes a character directly to the console |
| <code>cscanf</code> | Reads formatted values directly from the console |
| <code>cprintf</code> | Formats values and writes them directly to the console. |

- Defines numeric conversion functions, pseudo-random numbers generation functions, memory allocation, process control functions;
 - `<stdlib.h>` allows standard input/output operations

stdlib.h (standard library header) example.

| | |
|---------------------|------------------------------------|
| <code>system</code> | Execute system command (function) |
| <code>rand</code> | Generate random number (function) |
| <code>abort</code> | Abort current process (function) |


```

1  /* Fig. 2.1: fig02_01.c
2     A first program in C */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main()
7  {
8     printf( "welcome to C!\n" );
9
10     return 0; /* indicate that program ended successfully */
11
12 } // end function main.

```

Welcome to C!

- Block {...}
 - Text surrounded by braces {...}.
 - See line no.7 and no.12 above.
- Brace {}
 - A left brace, {, begins the body of every function (line 7). A corresponding right brace ends each function (line 11).

```

1  /* Fig. 2.1: fig02_01.c
2     A first program in C */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main()
7  {
8     printf( "welcome to C!\n" );
9
10     return 0; /* indicate that program ended successfully */
11
12 } // end function main.

```

Welcome to C!

- return 0; statement
 - in line no.10, there are the statement return 0; which is indicate that the function is successful normally terminated.
 - Why return 0, bcoz main function should return integer data type.
 - void main() does no need to return 0.

```

1  /* Fig. 2.1: fig02_01.c
2     A first program in C */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main()
7  {
8     printf( "welcome to C!\n" );
9
10     return 0; /* indicate that program ended successfully */
11
12 } // end function main.

```

Welcome to C!

- Comments
 - Text surrounded by `/* and */` is **ignored** by computer, also by `//`. See line no.1 & 2
 - Used to describe program

```

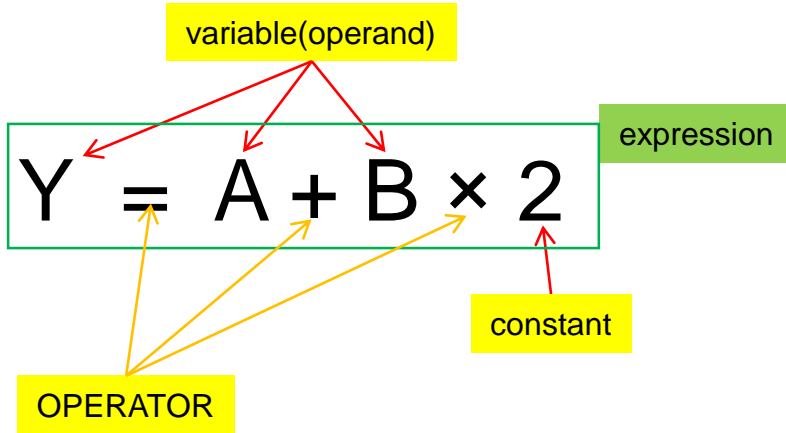
1  /* Fig. 2.1: fig02_01.c
2     A first program in C */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main()
7  {
8     printf( "welcome to C!\n" );
9
10     return 0; /* indicate that program ended successfully */
11
12 } // end function main.

```

Welcome to C!

Understand Operators and Expressions

- Operators are symbols which take one or more operands or expressions and perform arithmetic or logical computations.
- Types of operators available in C are as follows:
 - Arithmetic
 - Assignment
 - relational
 - Logical
 - Boolean operator/Bitwise



- Arithmetic Operator
 - All the **basic arithmetic** operations can be carried out in C.
 - Both **unary** and **binary** operations are available in C language.
 - *Unary operations operate on a **single operand**, therefore the number 5 when operated by unary `-` will have the value `-5`.

| C operation | Arithmetic operator | Algebraic expression | C expression |
|----------------|---------------------|--------------------------------------|--------------------|
| Addition | + | $f + 7$ | <code>f + 7</code> |
| Subtraction | - | $p - c$ | <code>p - c</code> |
| Multiplication | * | bm | <code>b * m</code> |
| Division | / | x/y or $\frac{x}{y}$ or $x \div y$ | <code>x / y</code> |
| Remainder | % | $r \text{ mod } s$ | <code>r % s</code> |

Questions

Write the C expression from the algebraic expression

| algebraic expression | c expression |
|---|------------------------------|
| $Y = mX^2 + d \div m \times k$ | $Y = m * X * X + d / m * k;$ |
| $s = s^3 + d - w^2$ | |
| $v = \frac{1}{2}V + a \times t$ | |
| $E = mc^2$ | |
| $Z = kD - bV^2 + m$ | |
| $o = 0.23m - 0.11k^2$ | |
| $\text{Area} = \frac{1}{2}(P \times L)$ | |
| Formula | C Expression |

- Arithmetic Operator
 - Both unary and binary operations are available in C.

+a Positive a

```
int a;  
a=10;  
m=a;
```

-b Negative b

```
int b;  
b=10;  
m=-b;
```

- Arithmetic Operator

- Both unary and binary operations are available in C

prefix ++a
postfix a++ a+1

```

int a;
a=10;
m=a++;

```

m=11 is the answer

prefix --a
postfix a-- a-1

```

int a;
a=10;
m=a--;

```

m=9 is the answer

Arithmetic Operational

| Operator name | | Syntax |
|--|--------|--------|
| Basic assignment | | a = b |
| Addition | | a + b |
| Subtraction | | a - b |
| Unary plus (integer promotion) | | +a |
| Unary minus (additive inverse) | | -a |
| Multiplication | | a * b |
| Division | | a / b |
| Modulo (integer remainder) ^[note 1] | | a % b |
| Increment | Prefix | ++a |
| | Suffix | a++ |
| Decrement | Prefix | --a |
| | Suffix | a-- |

- Relational Operator
 - C supports the following relational operators.
 - Often it is **required** to **compare** the relationship between operands and bring out a decision and program accordingly.
 - Example, we might make a decision in a program, for example, to determine if a **person's grade** on an exam is greater than or equal to 60 and if it is to print the message **"Congratulations! You passed."**

Questions

Write the C expression from the algebraic expression

| algebraic expression | c expression |
|-----------------------------------|--------------|
| X is greater than 46 | X>46 |
| y is lower than 423 | |
| 221 is lower than m | |
| A is greater than or equal to 257 | |
| A is greater than or equal to s | |
| W is not equal to K | |

Relational Operator

| Algebraic equality or relational operator | C equality or relational operator | Example of C condition | Meaning of C condition |
|---|-----------------------------------|------------------------|---------------------------------|
| <i>Equality operators</i> | | | |
| = | == | x == y | x is equal to y |
| ≠ | != | x != y | x is not equal to y |
| <i>Relational operators</i> | | | |
| > | > | x > y | x is greater than y |
| < | < | x < y | x is less than y |
| ≥ | >= | x >= y | x is greater than or equal to y |
| ≤ | <= | x <= y | x is less than or equal to y |

- Logical Operator
 - So far we have studied only simple conditions, such as `counter <= 10`, `total > 1000`, and `number != Value`.
 - C provides logical operators that may be used to form more complex conditions by combining simple conditions.
 - The logical operators are
 - **&& (logical AND)**,
 - **|| (logical OR)** and
 - **! (logical NOT also called logical negation)**.
- Logical Operator (&& - and)
 - **Variable1&&Variable2**

| expression 1 | expression 2 | expression 1 && expression 2 |
|--------------|--------------|------------------------------|
| 0 | 0 | 0 |
| 0 | nonzero | 0 |
| nonzero | 0 | 0 |
| nonzero | nonzero | 1 |

Questions

Write the C expression from the algebraic expression and result

| algebraic expression | c expression | result |
|----------------------|--------------|-----------|
| 0 and with 1 | 0&&1 | 0 (false) |
| 0 and with 0 | | |
| 1 and with 1 | | |
| 0 and with 124 | | |
| 12 and with 111 | | |

- Logical Operator (|| - or)
 - Variable1 || Variable2**

| expression1 | expression2 | expression1 expression2 |
|-------------|-------------|----------------------------|
| 0 | 0 | 0 |
| 0 | nonzero | 1 |
| nonzero | 0 | 1 |
| nonzero | nonzero | 1 |

- Logical Operator(Bitwise) (! - not)
 - !(Variable1)**

| expression | !expression |
|------------|-------------|
| 0 | 1 |
| nonzero | 0 |

Questions

Write the C expression from the algebraic expression and result

| algebraic expression | c expression | result |
|----------------------|--------------|----------|
| 0 or with 1 | 0 1 | 1 (true) |
| 0 or with 0 | | |
| 1 or with 1 | | |
| 0 or with 124 | | |
| 12 or with 111 | | |

| algebraic expression | c expression | result |
|----------------------|--------------|----------|
| not 0 | !0 | 1 (true) |
| not 1 | | |
| not (1 or 1) | | |
| not (0 &&124) | | |
| (not 12) or 111 | | |

Logical Operational

| Operator name | Syntax |
|------------------------|--------|
| Logical negation (NOT) | !a |
| Logical AND | a && b |
| Logical OR | a b |

- Boolean Operator(Bitwise)

- The bitwise operators perform bitwise-AND (&), bitwise-exclusive-OR (^), and bitwise-inclusive-OR (|) operations.

- & - The bitwise-AND operator compares each bit of its first operand to the corresponding bit of its second operand. If both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| & | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |

- Boolean Operator(Bitwise)

- The bitwise operators perform bitwise-AND (&), bitwise-exclusive-OR (^), and bitwise-inclusive-OR (|) operations.

- ^ - The bitwise-exclusive-OR operator compares each bit of its first operand to the corresponding bit of its second operand. If one bit is 0 and the other bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| ^ | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |

- Boolean Operator(Bitwise)

- The bitwise operators perform bitwise-AND (&), bitwise-exclusive-OR (^), and bitwise-inclusive-OR (|) operations.
- |- The bitwise-inclusive-OR operator compares each bit of its first operand to the corresponding bit of its second operand. If either bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

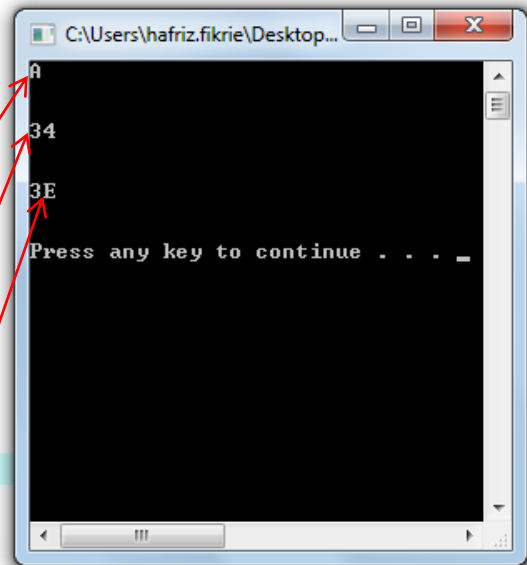
| | | | | | | | | |
|--|---|---|---|---|---|---|---|---|
| | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Example

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     short int digit1;
7     short int digit2;
8     short int answer;
9
10    digit1=0x3E; //00001110
11    digit2=0x0A; //00001010
12
13    answer=digit1&digit2;
14    printf("%X\n\n\n", answer);
15
16    answer=digit1^digit2;
17    printf("%X\n\n\n", answer);
18
19    answer=digit1|digit2;
20    printf("%X\n\n\n", answer);
21
22    system("pause");
23    return 0;
24 }
25

```



- Exercise** . Bitwise ~ & | ^

| C Expression | result |
|------------------------------------|--------|
| 4 & 3 | |
| 4A & 2B (assume in hex) | |
| 2 4 | |
| 2 & 6 3 | |
| (8^3) & 6 | |
| (9 ^ 3) (3 & 6) | |
| 2 & 3 & 5 10 (assume in decimal) | |
| 4 + 4 & 4 | |
| 5 - 2 & 5 | |

Boolean(Bitwise) Operational

| Operator name | Syntax |
|---|--------|
| Bitwise NOT | ~a |
| Bitwise AND | a & b |
| Bitwise OR | a b |
| Bitwise XOR | a ^ b |
| Bitwise left shift ^[note 2] | a << b |
| Bitwise right shift ^{[note 2][note 3]} | a >> b |

Left shift

| BITWISE OPERATORS | | |
|----------------------------|--------------------|--------------|
| << Shift Left | | |
| <u>SYNTAX</u> | <u>BINARY FORM</u> | <u>VALUE</u> |
| <code>x = 7;</code> | 00000111 | 7 |
| <code>x=x<<1;</code> | 00001110 | 14 |
| <code>x=x<<3;</code> | 01110000 | 112 |
| <code>x=x<<2;</code> | 11000000 | 192 |

- Assignment

- often just called the "assignment operator", is a special case of assignment operator where the source (right-hand side) and destination (left-hand side) are of the same class type.

eg: `a=a+b;` mean $a(\text{new}) = a(\text{old}) + b;$

`int a,b;`

`a=3; b=5; →`

`a=a+b; →`

`a=a+b;`

`d=d-b;`

`e=e×b;`

`m=m÷k;`

`a+=b;`

`d-=b;`

`e*=b`

`m/=k`

Compound(assignment) Operational

| Operator name | syntax | meaning |
|--------------------------------|---------|----------|
| Addition assignment | $a+=b$ | $a=a+b$ |
| Subtraction assignment | $a-=b$ | $a=a-b$ |
| Multiplication assignment | $a*=b$ | $a=a*b$ |
| Division assignment | $a/=b$ | $a=a/b$ |
| Modulo assignment | $a%=b$ | $a=a\%b$ |
| Bitwise AND assignment | $a\&=b$ | $a=a\&b$ |
| Bitwise OR assignment | $a =b$ | $a=a b$ |
| Bitwise XOR assignment | $a^=b$ | $a=a^b$ |
| Bitwise left shift assignment | $a<<=b$ | $a=a<<b$ |
| Bitwise right shift assignment | $a>>=b$ | $a=a>>b$ |

Questions

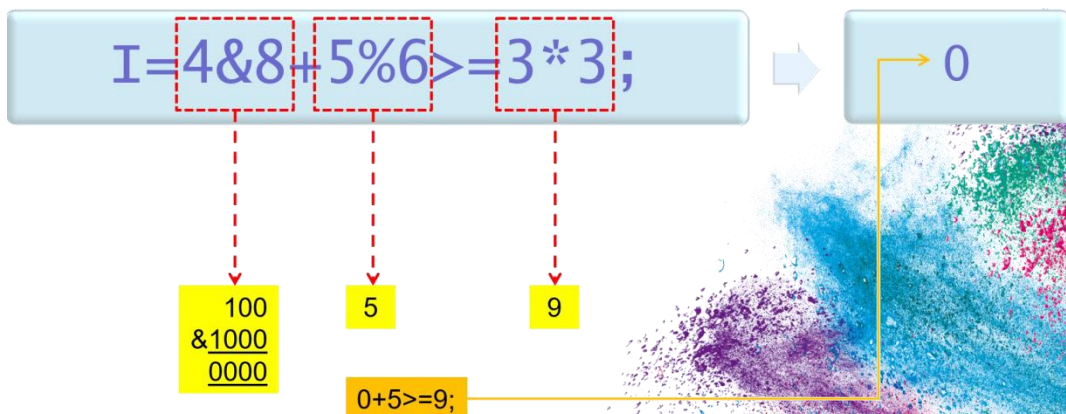
Write the answers for each expression

| Question | Answer |
|----------------|--------|
| $i=3*5+5\%3;$ | |
| $i=3+5*2\%3;$ | |
| $i=4+5\%3*3;$ | |
| $i=6\%2*5\%3;$ | |
| $i=4>2;$ | |
| $i=4>2+5;$ | |
| $i=4>=4*0\%2;$ | |
| $i=4!=3;$ | |

Hierarchy Of Operator

| Operator | Associativity | Type |
|-----------------------------------|---------------|----------------|
| () [] . -> | left to right | highest |
| + - ++ -- ! & * ~ sizeof (type) | right to left | unary |
| * / % | left to right | multiplicative |
| + - | left to right | additive |
| << >> | left to right | shifting |
| < <= > >= | left to right | relational |
| == != | left to right | equality |
| & | left to right | bitwise AND |
| ^ | left to right | bitwise OR |
| | left to right | bitwise OR |
| && | left to right | logical AND |
| | left to right | logical OR |
| ?: | right to left | conditional |
| = += -= *= /= &= = ^= <<= >>= %= | right to left | assignment |
| , | left to right | comma |

Example



Questions

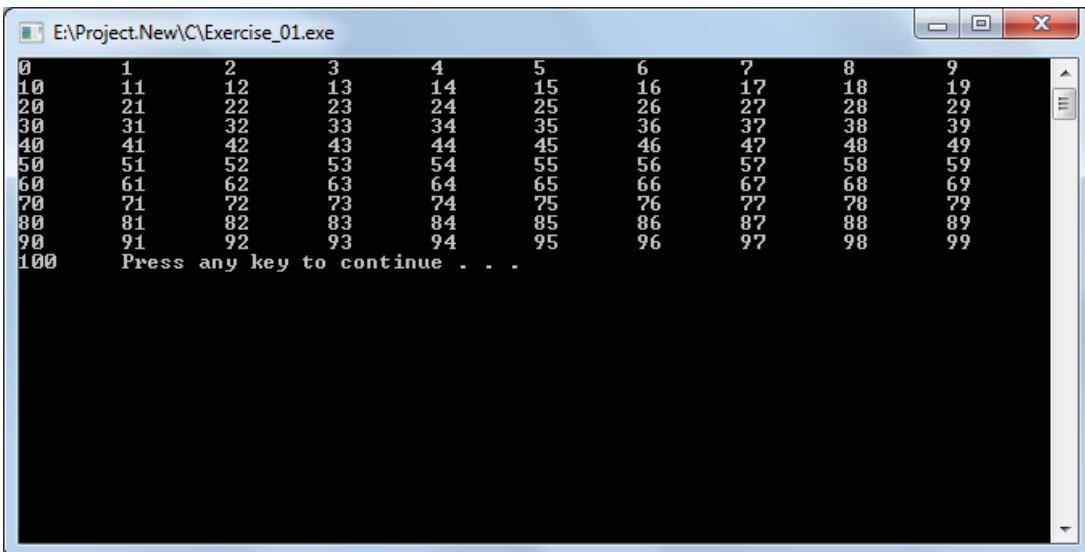
Write the answers for each expression

| Question | Answer |
|---------------------------|--------|
| $I=3\%4+6-8*(2+22) 3;$ | |
| $l=13\&(5+9*2\%4-l4);$ | |
| $I=4+3*2-5\%6<=19\%10;$ | |
| $i=6\%2*5\%3;$ | |

Programming Example

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void main()
5 {
6     int j=0;    //assign j=0
7
8     while(j<=100)
9         {printf("%d\t",j);
10            j=j+1;
11        }
12
13     system("pause");
14
15 }
16
```

Prompt number
0 to 100



Programming Example

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void main()
5 {
6     int j=0;    //assign j=0
7
8     while(j<=100)
9         {printf("%d\t",j);
10        j=j+1;
11        }
12
13     system("pause");
14
15 }
16
```

Exercise

Modify the programming above

Modify the program to print as follow

```
0           2           4           6.....
.....200
```

Modify the program to print as follow

```
0           0.1           0.2           0.3.....
.....20
```

Modify the program to print as follow

```
0
2
4
6
:
:
:
200
```


Programming Example

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void main()
5 {
6     int j=0;    //assign j=0
7
8     while(j<=100)
9         {printf("%d\t",j);
10          j=j+1;
11         }
12
13     system("pause");
14
15 }
16
```

Exercise

Modify the programming above

Modify the program to print as follow

| No. | x2 | x3 |
|-----|----|----|
| 0 | 0 | 0 |
| 1 | 2 | 3 |
| 2 | 4 | 6 |
| 3 | 6 | 9 |
| 4 | 8 | 12 |
| 5 | 10 | 15 |
| : | : | : |
| : | : | : |
| 20 | 40 | 60 |

extra notes

```
printf( "%d" , (x == w || x == y || x == z) );
```

- In this example, if x is equal to either w, y, or z, the second argument to the printf function evaluates to true and the value 1 is printed. Otherwise, it evaluates to false and the value 0 is printed. As soon as one of the conditions evaluates to true, evaluation ceases.

extra notes

- The following examples illustrate the logical operators:

```
int w, x, y, z;

if ( x < y && y < z )
    printf( "x is less than z\n" );
```

- In this example, the printf function is called to print a message if x is less than y and y is less than z. If x is greater than y, the second operand (y < z) is not evaluated and nothing is printed.

Exercises

Observe what the output for each programming

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int a; //variable a untuk simpan satu nombor.
7
8     printf("Enter one number:\n");
9     scanf("%d",&a); //must have '&' before a
10
11
12     printf("The entered numbers is = %d\n",a);
13
14     system("pause");
15     return 0;
16 }
17
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int a; //variable a untuk simpan satu nombor.
7
8     printf("Enter one number:\n");
9     scanf("%d",&a); //must have '&' before a
10
11
12     printf("The entered numbers is = %d\n",a);
13
14     system("pause");
15     return 0;
16 }
17
```

Try make a program to key-in and view number:

a) 23 & 24

b) 33, 155 & 112

hint.: just add more variable

Eg: int a;

int b;

int c;

Exercises

Observe what the output for each programming

Program to add 2 integers

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int a, b, c;
7
8     printf("Enter first numbers to add:");
9     scanf("%d", &a);
10    printf("\n");
11
12    printf("Enter second numbers to add:");
13    scanf("%d", &b);
14    printf("\n");
15
16    c = a + b;
17
18    printf("Sum of entered numbers = %d\n", c);
19
20    system("pause");
21    return 0;
22 }
23 |
```

have 3 variable,
a for first number..
b for second number..

..and.....

c for the total..

Observe what the output..

Try make a program to key-in and add this number:

a) 23 & 24

b) 33, 155 & 112

hint.: just add more variable

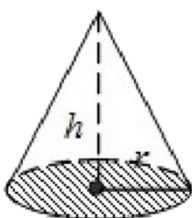
Eg: int a;

int b;

int c;

Create C Program..

Kon - Cone



$$\begin{aligned}\text{Volume} &= \frac{1}{3} \times \text{base area} \times h \\ &= \frac{1}{3} \pi r^2 h\end{aligned}$$

Exercises

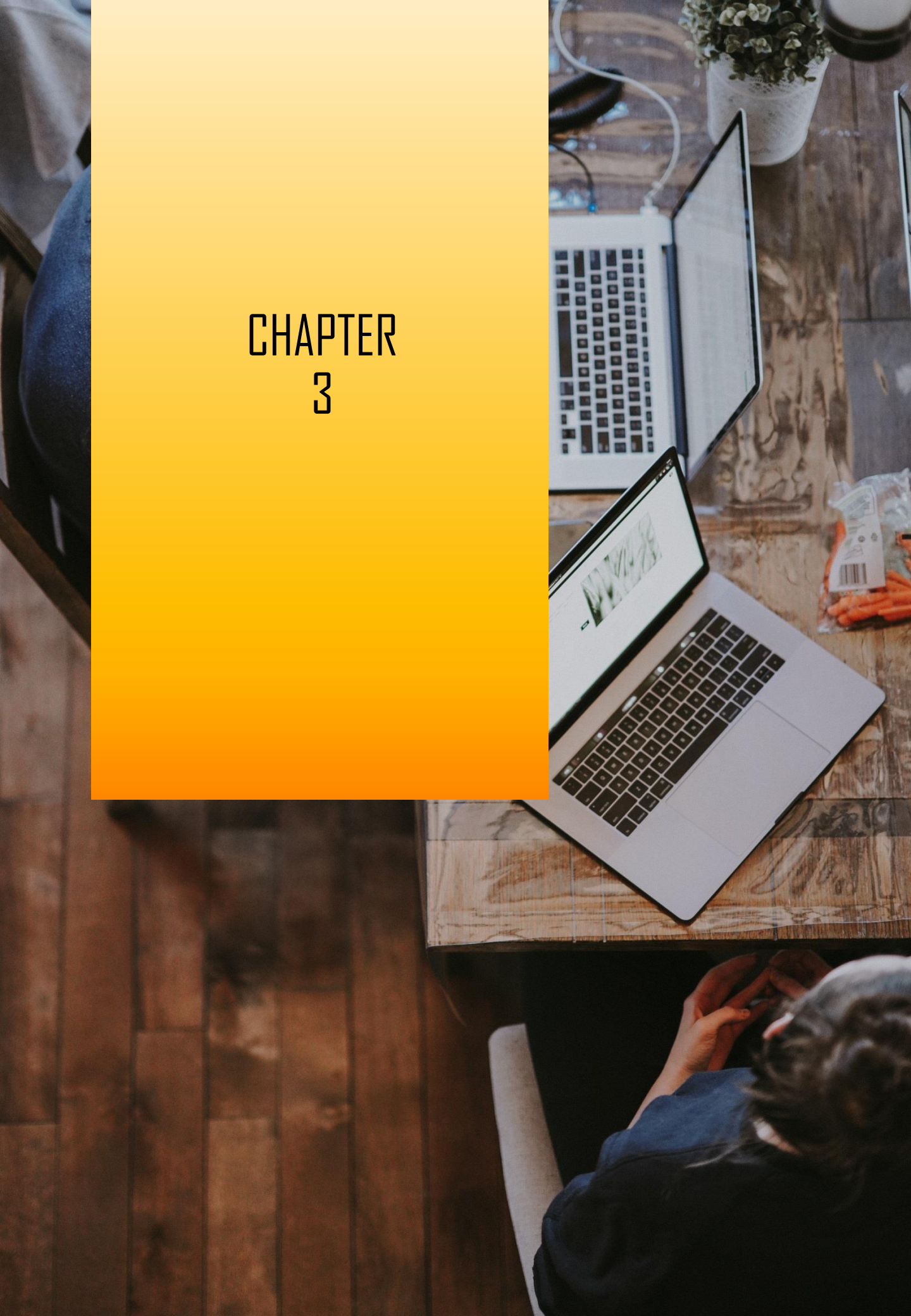
Create C Program to display 1 to 100

| Numbers 1 to 100 Chart - nicholasacademy.com | | | | | | | | | |
|--|----|----|----|----|----|----|----|----|-----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 |
| 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 |
| 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 |
| 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 |

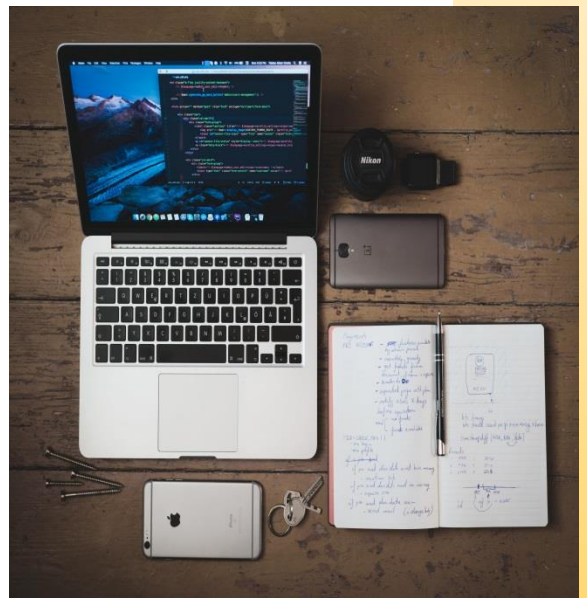
Exercises

- Implement mathematical calculations in simple C program
- Implement mathematical calculations using the function in the main function

CHAPTER 3



Selection Statements



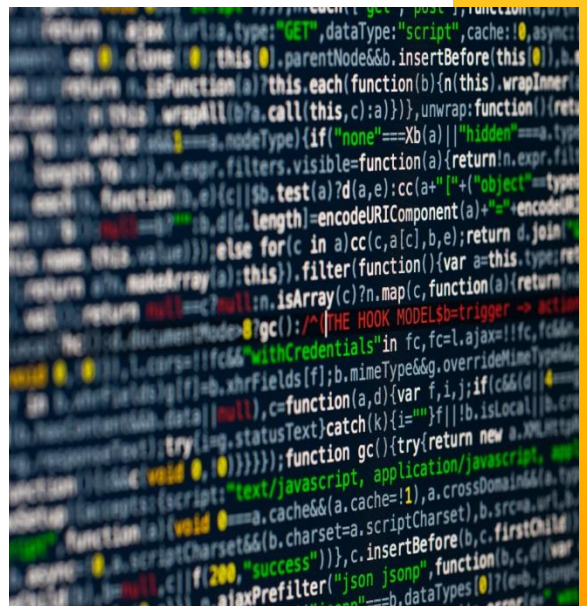
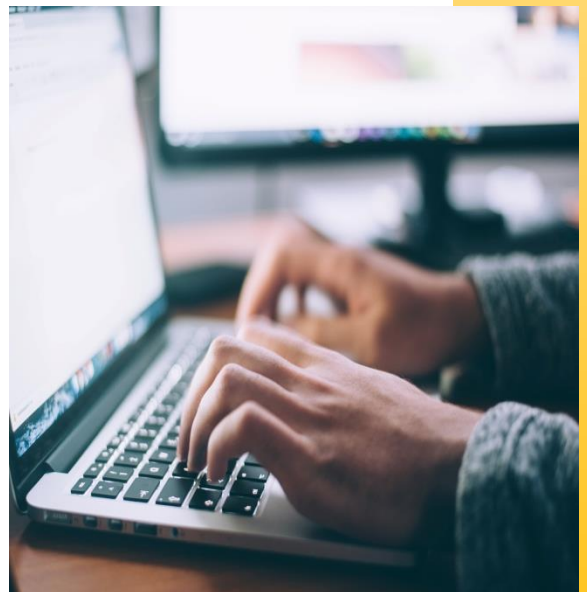
- IF statements
 - IF statements
 - Nested IF statements



IF-ELSE statement



SWITCH statements



Control Structure

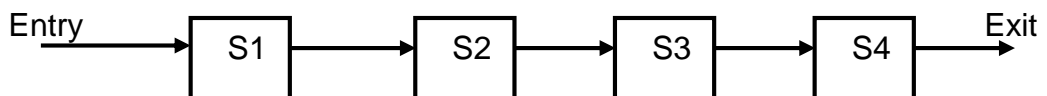
- C provides several programs for control statement and lets to execute the instructions in a non-sequential tasks (skipping a block of instructions or execute a block of instructions repetitively).
- 4 basic control structures:-
 - sequence structure
 - selection structure
 - repetition structure (iteration/ looping)
 - Jumps statements

A. Sequence Structure

- the simplest of all the structures.
- The program instructions are executed one by one, starting from the first instruction and ending in the last instruction as in the program segment.

Example :

```
x = 5; (S1)
y = 10; (S2)
Total = x * y; (S3)
printf(" Total =%d", Total); (S4)
```



B. Selection Structure

- The selection structure allows to be executed non-sequentially.
- It allows the comparison of two expressions, and based on the comparison, to select a certain course of action.
- There are three types of selection statements:
 - **if** statement
 - either performs (selects) an action if a condition is true or skips the action if the condition is false.
 - **if-else** statement
 - performs an action if a condition is true and performs a different action if the condition is false.
 - **switch** statement.
 - performs one of many different actions depending on the value of an expression

Selection Structure (if)

This is used to decide whether to do something at a special point, or to decide between two courses of action.

if selection statement : "For example, suppose the passing grade on an exam is 60."

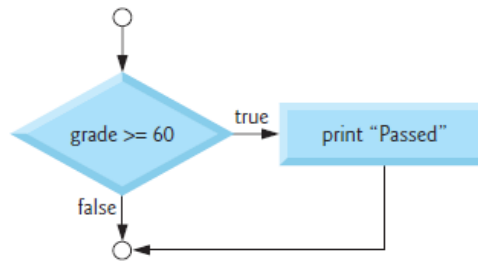
- Pseudo-code:

```
If student's grade is greater than or equal to 60  
Print "Passed"
```

- C syntax:

```
if ( grade >= 60 ) {  
    printf( "Passed\n" );  
} /* end if */
```

- Flowchart:



An example of if statement positive and negative number:

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3  
4 int main()  
5 {  
6     int a;  
7  
8     printf("\n Enter a number:");  
9     scanf("%d", &a);  
10  
11     if(a>=0)  
12     {  
13         printf( "\n The number %d is positive.\n",a);  
14     }  
15  
16     if(a<0)  
17     {  
18         printf("\n The number %d is negative.\n ",a);  
19     }  
20  
21     system("pause");  
22     return 0;  
23 }
```


Another example of if statement: compare 2 numbers

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int a;
7     int b;
8
9     printf("\nEnter 1st number: ");
10    scanf("%d", &a);
11
12    printf("\nEnter 2nd number: ");
13    scanf("%d", &b);
14
15    if(a>b)
16    {
17        printf( "\nThe number a = %d is more than b = %d\n",a,b);
18    }
19
20    if(a<b)
21    {
22        printf( "\nThe number a = %d is less than b = %d\n",a,b);
23    }
24
25    system("pause");
26    return 0;
27 }
```

Another example of if statement: check ur gender

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     char gender;
7
8     printf("\nChoose:\n"
9         " m: if you are male\n"
10        " f: if you are female\n"
11        " x: if you are not sure\n");
12
13    printf("\nEnter: ");
14    scanf("%c", &gender);
15
16    if(gender=='m')
17    { printf( "\nCongrats..You are the Man!!\n");
18    }
19
20    if(gender=='f')
21    { printf( "\nWell, you are a Woman\n");
22    }
23
24    if(gender=='x')
25    { printf( "\nOMG..you should check back your gender!!\n");
26    }
27    system("pause");
28    return 0;
29 }
```

Selection Structure (if...else)

- [If-else selection](#) statement : "For example, suppose the passing grade on an exam is 60."

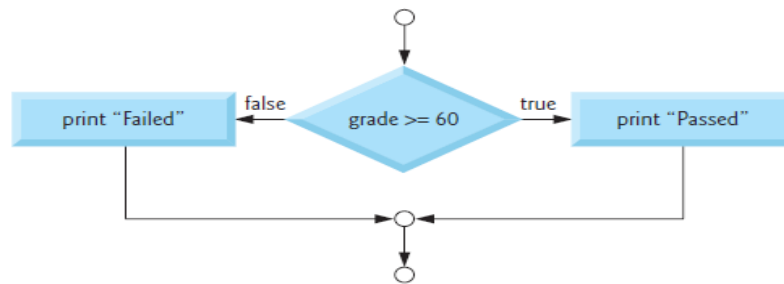
- Pseudo-code:

```
If student's grade is greater than or equal to 60  
Print "Passed"  
else  
Print "Failed"
```

- C syntax:

```
if ( grade >= 60 ) {  
    printf( "Passed\n" );  
} /* end if */  
else {  
    printf( "Failed\n" );  
} /* end else */
```

- Flowchart:



An example of if...else statement positive and negative number

```
1 #include<stdio.h>  
2 #include <stdlib.h>  
3  
4 int main()  
5 {  
6     int a;  
7  
8     printf("\n Enter a number:");  
9     scanf("%d", &a);  
10  
11     if(a>=0)  
12     {  
13         printf( "\n The number %d is positive.",a);  
14     }  
15  
16     else  
17     {  
18         printf("\n The number %d is negative.",a);  
19     }  
20  
21     return 0;  
22  
23 }
```

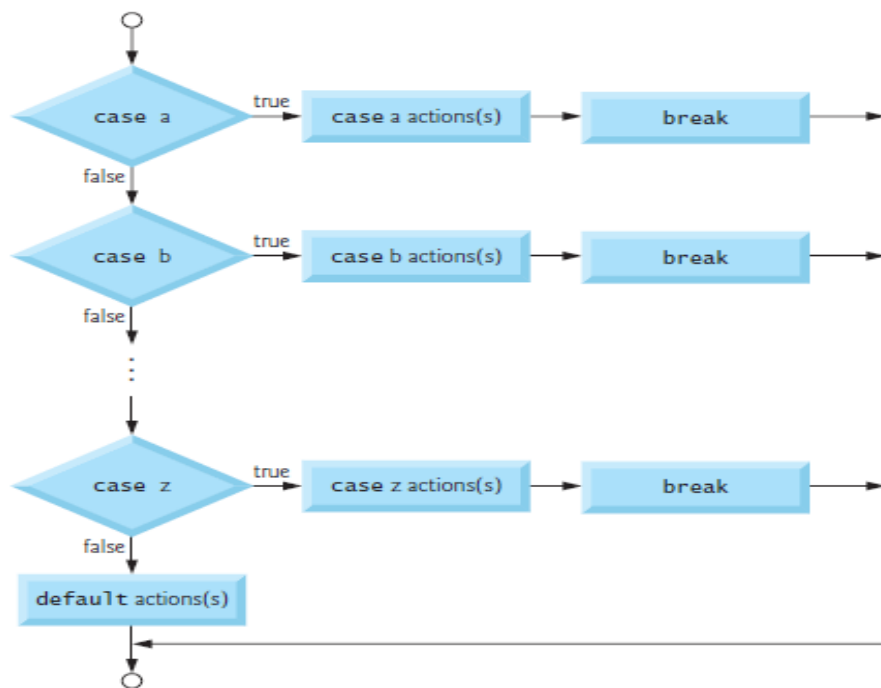
Selection Structure (switch...case)

- [switch selection](#) statement :

- C syntax:

```
switch (condition)
{
    case 1 : statement1; break;
    case 2 : statement2; break;
    default: statement3;
}
```

- Flowchart:



Selection Structure (switch...case)

An example of switch: [to switch a color]

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int color;
7     printf("Please choose a color:\n"
8           " 1. Red\n"
9           " 2. Green\n"
10          " 3: Blue\n");
11     scanf("%d", &color);
12
13     switch (color)
14     {
15         case 1:
16             printf("You chose RED color\n");
17             system("color 40");
18             break;
19         case 2:
20             printf("you chose GREEN color\n");
21             system("color 20");
22             break;
23         case 3:
24             printf("you chose BLUE color\n");
25             system("color 10");
26             break;
27         default:
28             printf("you did not choose any color\n");
29     }
30
31     system("pause");
32     return 0;
33 }
```

Inform the user

store number in "color"

Choose between
number 1,2,3 only!!

Exercise

Create a program if you;

enter number 1, it will display Sunday

enter number 2, it will display Monday

enter number 3, it will display Tuesday

enter number 4, it will display Wednesday

enter number 5, it will display Thursday

enter number 6, it will display Friday

enter number 7, it will display Saturday

Selection Structure (switch...case)

An example of switch: to check a grade!

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     char Grade;
7     printf("Please enter your grade:\n");
8     scanf("%c", &Grade);
9
10    switch( Grade )
11    {
12        case 'A' :
13            printf( "Excellent\n" );
14            break;
15        case 'B' :
16            printf( "Good\n" );
17            break;
18        case 'C' :
19            printf( "OK\n" );
20            break;
21        case 'D' :
22            printf( "Mmmmm....\n" );
23            break;
24        case 'F' :
25            printf( "You must do better than this\n" );
26            break;
27        default :
28            printf( "What is your grade anyway?\n" );
29            break;
```

Inform the user

store character in "grade"

Choose between character A,B,C,D,F only!!

- C programming is case sensitive, so the letter 'A' and 'a' for case is different. So, to choice both, even lower case of upper case, just modify the coding by adding both cases like example:

```
switch (Grade)
{
    case 'A':
    case 'a':
        printf(" Excellent\n");
        break;
    ...
}
```

Nested if...else statements

- Nested if...else statements test for multiple cases by placing if...else statements inside if...else statements.
- For example, the following pseudocode statement will print A for exam grades greater than or equal to 90, B for grades greater than or equal to 80, C for grades greater than or equal to 70, D for grades greater than or equal to 60, and F for all other grades.

```
If student's grade is greater than or equal to 90  
  Print "A"  
else  
  If student's grade is greater than or equal to 80  
    Print "B"  
  else  
    If student's grade is greater than or equal to 70  
      Print "C"  
    else  
      If student's grade is greater than or equal to 60  
        Print "D"  
      else  
        Print "F"
```

This pseudocode may be written in C as

```
if ( grade >= 90 )  
    printf( "A\n" );  
else  
    if ( grade >= 80 )  
        printf("B\n");  
    else  
        if ( grade >= 70 )  
            printf("C\n");  
        else  
            if ( grade >= 60 )  
                printf( "D\n" );  
            else  
                printf( "F\n" );
```

- C syntax can be simply as above:

```
if ( expression 1 )  
    program statement 1  
else  
    if ( expression 2 )  
        program statement 2  
    else  
        program statement 3
```

```
if ( expression 1 )  
    program statement 1  
else if ( expression 2 )  
    program statement 2  
else  
    program statement 3
```

Example:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void main()
5  {
6      int Mark;
7
8
9      printf("PROGRAM START!\n");
10     printf("Your Marks is :");
11     scanf("%d", &Mark);
12
13     if( (Mark>=80) && (Mark<=100) )
14     {printf("Grade A\n");}
15
16     else if( (Mark>=70) && (Mark<80) )
17     {printf("Grade B\n");}
18
19     else if( (Mark>=60) && (Mark<70) )
20     {printf("Grade C\n");}
21
22     else if( (Mark>=40) && (Mark<60) )
23     {printf("Grade D\n");}
24
25     else if( (Mark>=0) && (Mark<40) )
26     {printf("Grade D\n");}
27
28     else
29     {printf("Out of Range\n");}
30
31
32     system("pause");
33 }
```

- C syntax:

if (condition)

else if (condition)

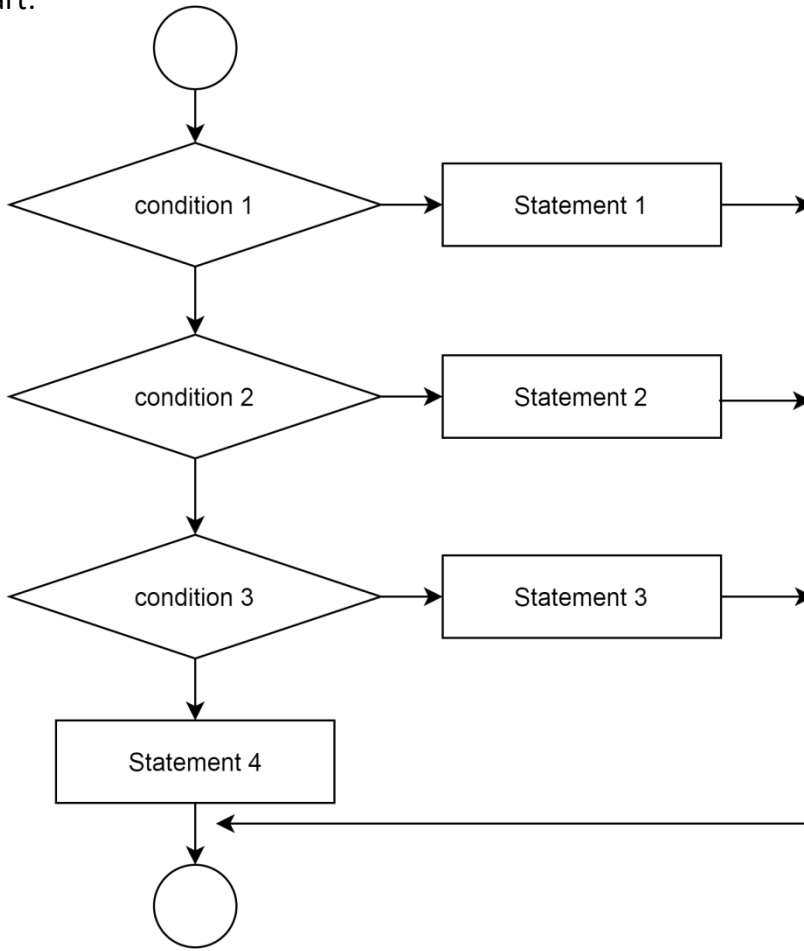
else if (condition)

else if (condition)

else

Depend on how
Many condition
You want to use

- Flowchart:



Syntax:

```
if(condition1)
{
    statement1
}
```

```
else if(condition2)
{
    statement2
}
```

```
else if(condition3)
{
    statement3
}
```

```
else
    statement4
```


Comparison if..else & switch..case

| BASIS FOR COMPARISON | IF-ELSE | SWITCH |
|------------------------------|--|--|
| Basic | Which statement will be executed depend upon the output of the expression inside if statement. | Which statement will be executed is decided by user. |
| Expression | if-else statement uses multiple statement for multiple choices. | switch statement uses single expression for multiple choices. |
| Testing | if-else statement test for equality as well as for logical expression. | switch statement test only for equality. |
| Evaluation | if statement evaluates integer, character, pointer or floating-point type or boolean type. | switch statement evaluates only character or integer value. |
| Sequence of Execution | Either if statement will be executed or else statement is executed. | switch statement execute one case after another till a break statement is appeared or the end of switch statement is reached. |
| Default Execution | If the condition inside if statements is false, then by default the else statement is executed if created. | If the condition inside switch statements does not match with any of cases, for that instance the default statements is executed if created. |
| Editing | It is difficult to edit the if-else statement, if the nested if-else statement is used. | It is easy to edit switch cases as, they are recognized easily. |

Comparison if-statement and Nested if-else-statement

if

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void main()
5 {
6     int Mark;
7
8
9     printf("PROGRAM START!\n");
10    printf("Your Marks is :");
11    scanf("%d", &Mark);
12
13    if( (Mark>=80) && (Mark<=100) )
14    {printf("Grade A\n");}
15
16    if( (Mark>=70) && (Mark<80) )
17    {printf("Grade B\n");}
18
19    if( (Mark>=60) && (Mark<70) )
20    {printf("Grade C\n");}
21
22    if( (Mark>=40) && (Mark<60) )
23    {printf("Grade D\n");}
24
25    if( (Mark>=0) && (Mark<40) )
26    {printf("Grade D\n");}
27
28    if( (Mark<0) || (Mark>100) )
29    {printf("Out of Range\n");}
30
31
32    system("pause");
33 }
```

Nested if else

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void main()
5 {
6     int Mark;
7
8
9     printf("PROGRAM START!\n");
10    printf("Your Marks is :");
11    scanf("%d", &Mark);
12
13    if( (Mark>=80) && (Mark<=100) )
14    {printf("Grade A\n");}
15
16    else if( (Mark>=70) && (Mark<80) )
17    {printf("Grade B\n");}
18
19    else if( (Mark>=60) && (Mark<70) )
20    {printf("Grade C\n");}
21
22    else if( (Mark>=40) && (Mark<60) )
23    {printf("Grade D\n");}
24
25    else if( (Mark>=0) && (Mark<40) )
26    {printf("Grade D\n");}
27
28    else
29    {printf("Out of Range\n");}
30
31
32    system("pause");
33 }
```

Comparison if-statement and Nested if-else-statement

Nested if else

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void main()
5 {
6     float value1, value2;
7     char operator;
8
9     printf ("Type in your expression.\n");
10    scanf ("%f %c %f", &value1, &operator, &value2);
11
12    if ( operator == '+' )
13        printf (".2f\n", value1 + value2);
14
15    else if ( operator == '-' )
16        printf (".2f\n", value1 - value2);
17
18    else if ( operator == '*' )
19        printf (".2f\n", value1 * value2);
20
21    else if ( operator == '/' )
22        printf (".2f\n", value1 / value2);
23
24    system("pause");
25 }
```

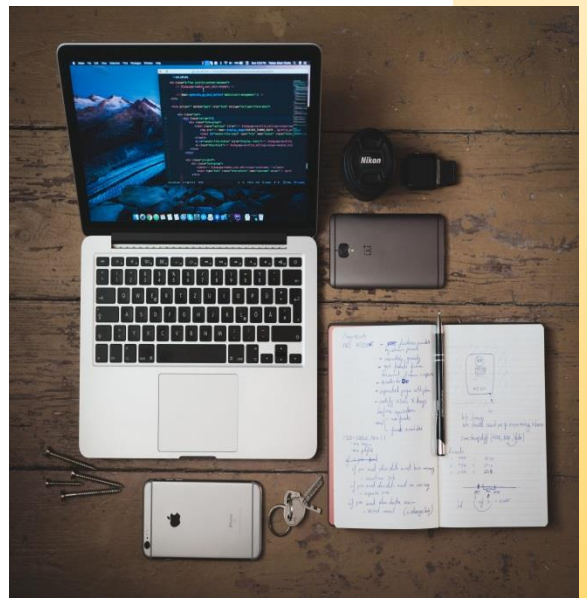
switch case

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void main()
5 {
6     float value1, value2;
7     char operator;
8
9     printf ("Type in your expression. \nexample: operand operator operand. \n1 + 1\n");
10    scanf ("%f %c %f", &value1, &operator, &value2);
11
12    switch(operator)
13    {
14        case '+':
15            printf (".2f\n", value1 + value2);
16            break;
17
18        case '-':
19            printf (".2f\n", value1 - value2);
20            break;
21
22        case '*':
23            printf (".2f\n", value1 * value2);
24            break;
25
26        case '/':
27            printf (".2f\n", value1 / value2);
28            break;
29
30        default:
31            printf ("Please enter the right value of operand and operator");
32            break;
33    }
34
35
36
37    system("pause");
38 }
```

CHAPTER 4



Looping Statements



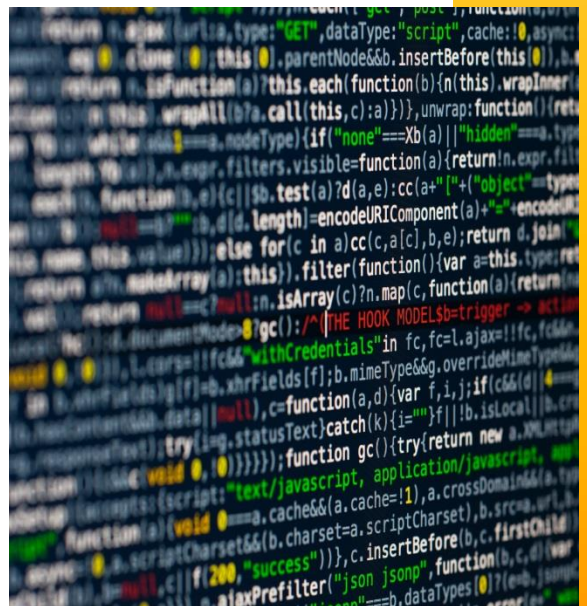
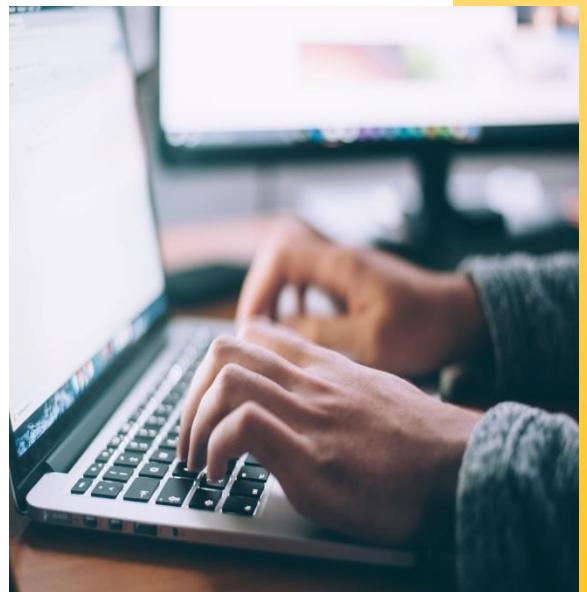
- FOR statements
- FOR statements
 - Nested FOR statements



WHILE statement

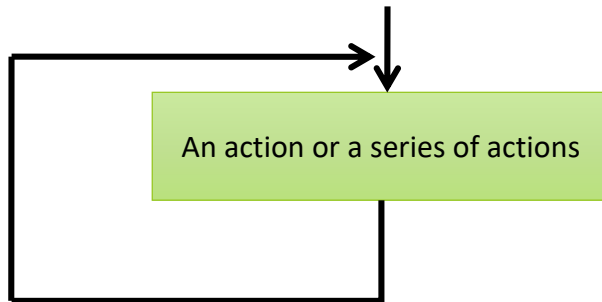


DO-WHILE statements



Definition of looping or repetition

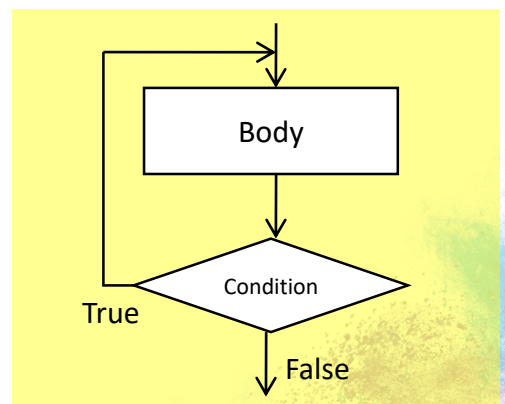
- Loops provide a way to repeat commands and control how many times they are repeated.
- C provides three types of looping structures in the form of statements.
 - **for** statement
 - **while** statement
 - **do...while** statement
- The main idea of a loop is to repeat an action or a series of actions.



The concept of a loop

- But, when to stop looping?
- In the following flowchart, the action is executed over and over again. It never stop – This is called infinite loop
- Solution – put a condition to tell the loop either continue looping or stop.

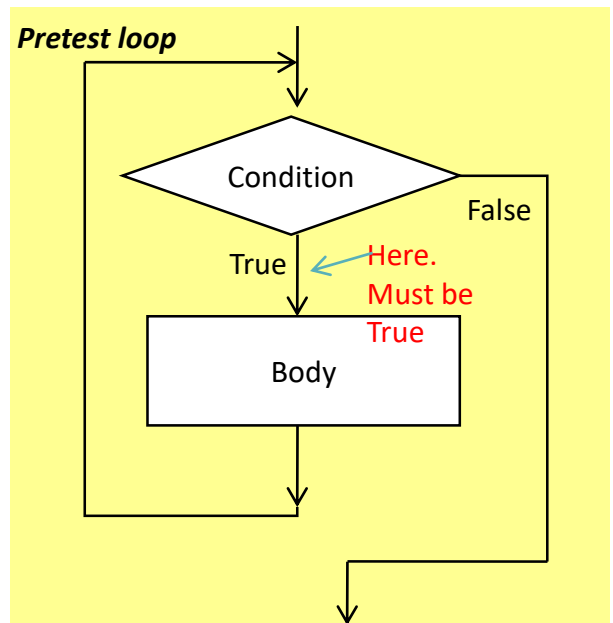
- A loop has two parts – **body** and **condition**
- **Body** – a statement or a block of statements that will be repeated.
- **Condition** – is used to control the iteration – either to continue or stop iterating.



- Two forms of loop:
 - **Pretest loop**
 - **post-test loop**

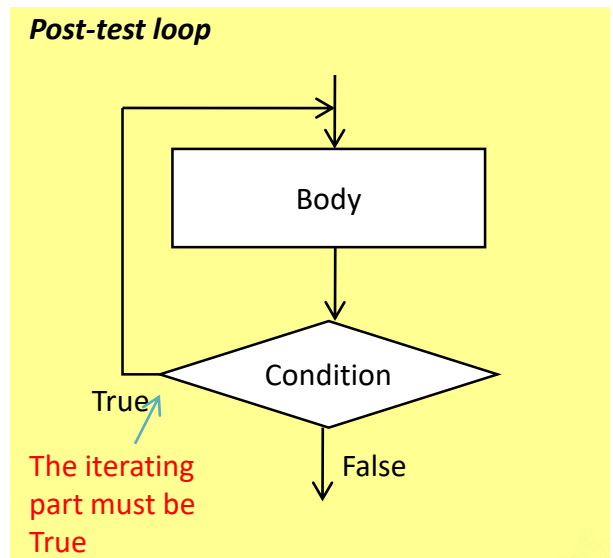
- Pretest loop

- the condition is tested first, before we start executing the body.
- The body is executed if the condition is true.
- After executing the body, the loop repeats



- Post-test loop

- the condition is tested later, after executing the body.
- If the condition is true, the loop repeats, otherwise it terminates.
- The body is always executed at least once.



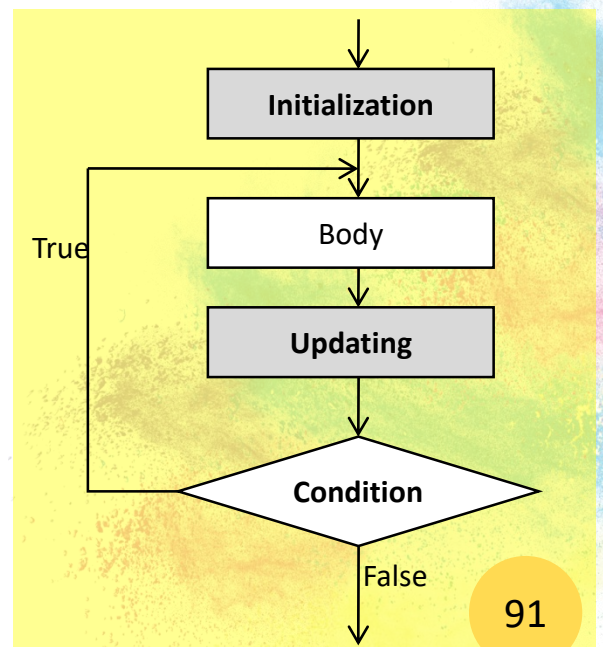
Parts of a loop

- Initialization

- is used to prepare a loop before it can start – usually, here we initialize the condition.
- The initialization must be written outside of the loop – before the first execution of the body.

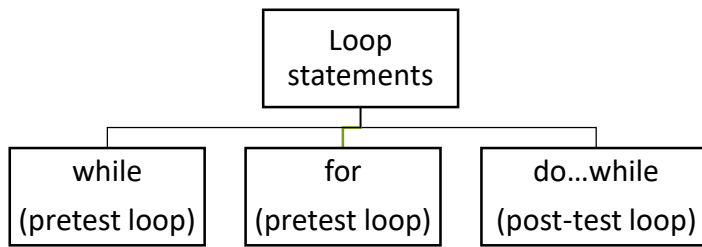
- Updating

- is used to update the condition.
- If the condition is not updated, it always true => the loop always repeats- an infinite loop.
- The updating part is written inside the loop – it is actually a part of the body.



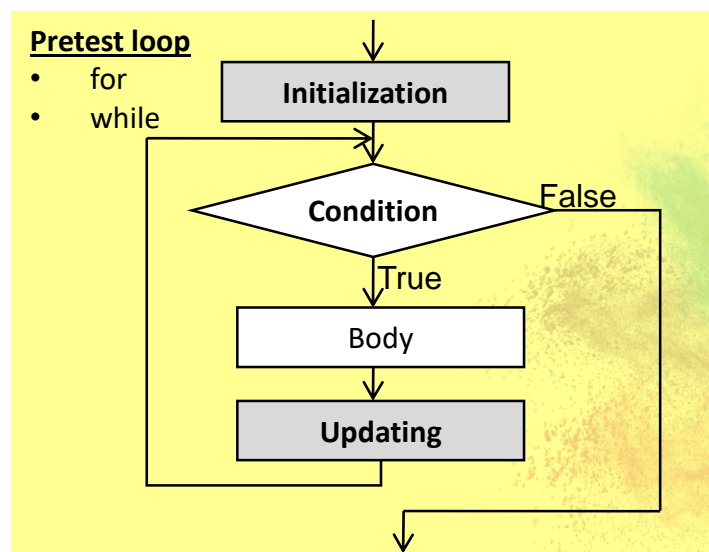
Loop statements

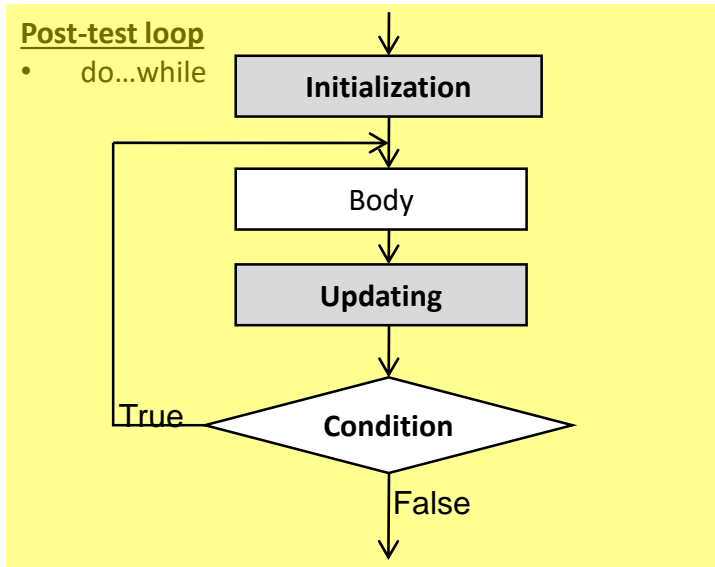
- C provides three loop statements:



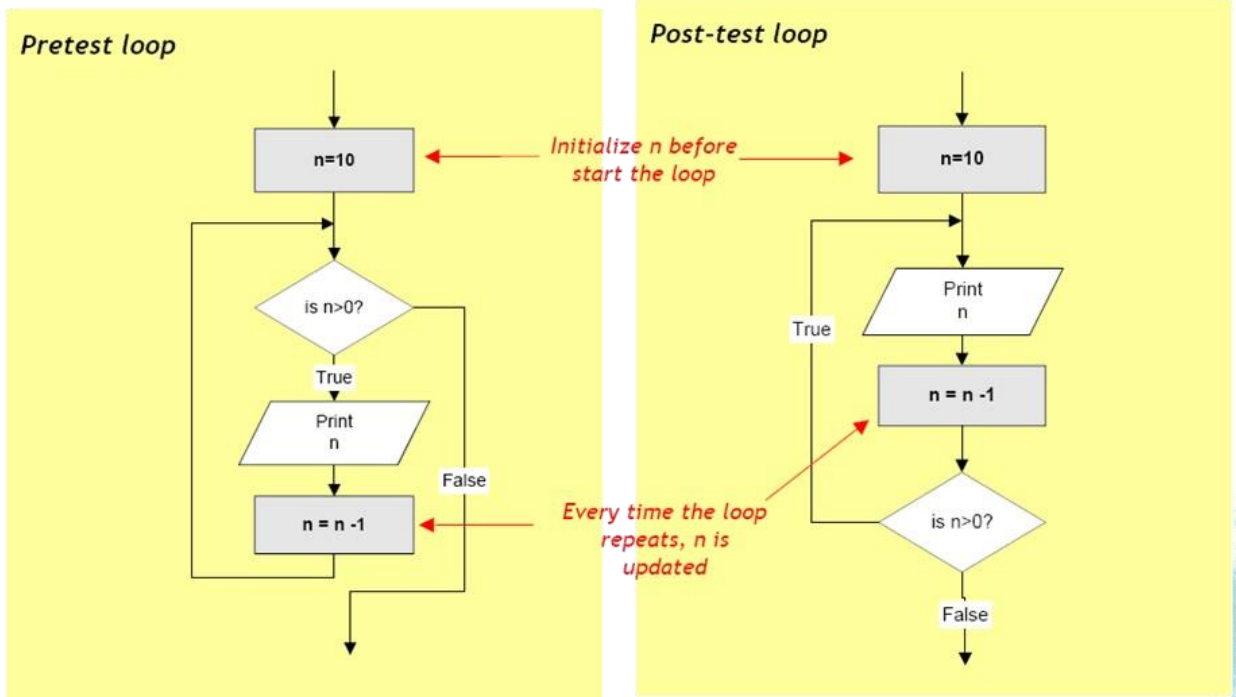
Definition

- **while** loop statement
 - A while statement is like a repeating if statement.
 - Like an if statement, if the test condition is true: the statements get executed.
 - The difference is that after the statements have been executed, the test condition is checked again.
 - If it is still true the statements get executed again.
 - This cycle repeats until the test condition evaluates to false.
- **do...while** loop statement
 - do ... while is just like a while loop except that the test condition is checked at the end of the loop rather than the start.
 - This has the effect that the content of the loop are always executed at least once.
- **for** loop statement
 - for loop is **very flexible** based on the combination of the **three expression** is used.
 - The counter can be not only counted up but also counted down. You can count by twos, threes and so on. You can count by not only number but also character.
- Beside the body and condition, a loop may have two other parts – **Initialization** and **Updating**





- **Example: These flowcharts print numbers 10 down to 1**



Comparison

- **while :**
 - while tests a condition at the beginning of the loop
 - condition must first be true for the loop to run even once
- **do while :**
 - do/while tests a condition at the end of the loop
 - loop will run at least once
- **for :**
 - for facilitates initializing and incrementing the variable that controls the loop
 - Especially helpful for:
 - Looping for a known number of times

for looping

The C for statement lets you specify the initialization, test, and update operations of a structured loop in a single statement. The for statement is created as follows:

```
for (init_exp; cond_exp; update_exp)
{
    loop_body_statement;
}
```

where:

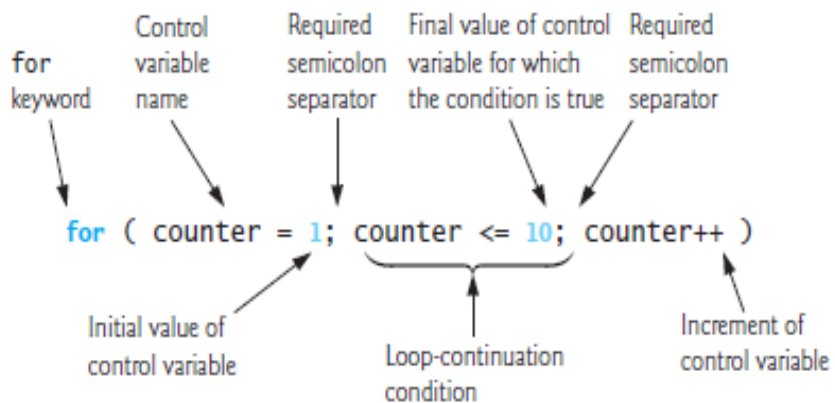
init_exp: is an expression that is evaluated before the loop is entered.

cond_exp: is an expression that is evaluated before each pass through the loop.

update_exp: is an expression that is evaluated at the end of each pass through the loop, after the loop body has been executed, and just before looping back to evaluate cond_exp again.

The C for statement lets you specify the initialization, test, and update operations of a structured loop in a single statement. The for statement is created as follows:

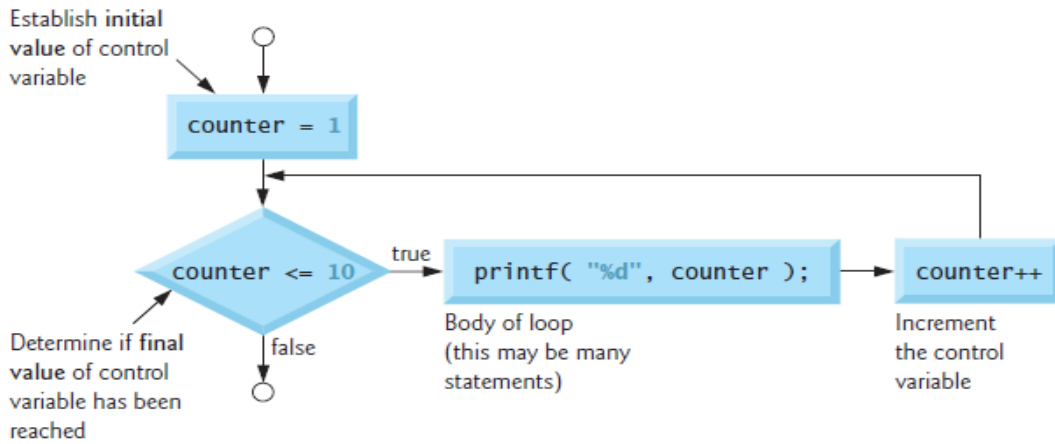
C syntax



for looping

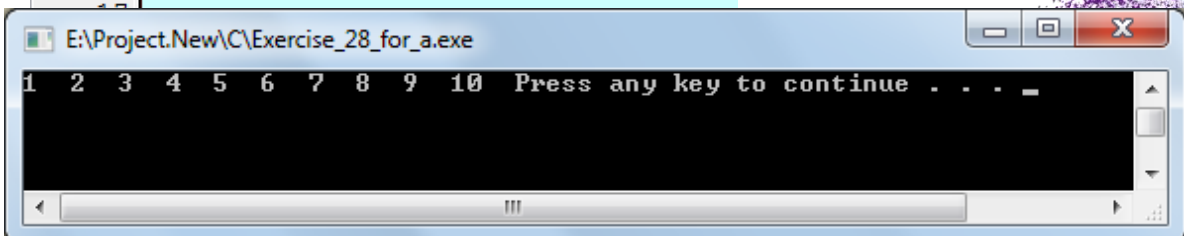
- The C for statement lets you specify the initialization, test, and update operations of a structured loop in a single statement. The for statement is created as follows:

flowchat



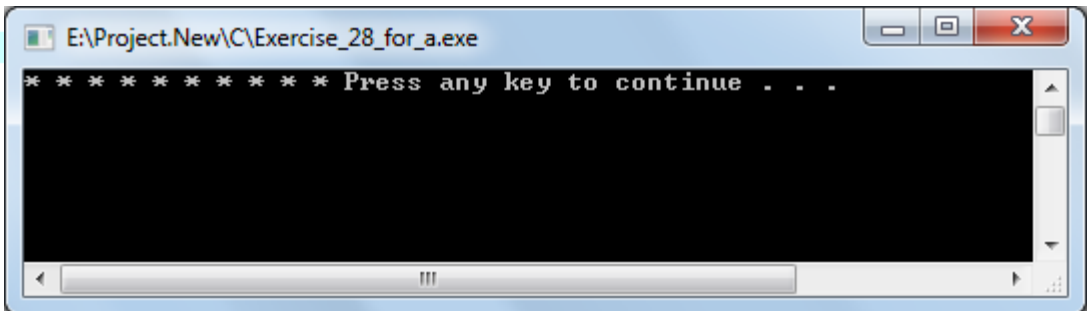
Example 1 - Display number 1 to 10

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int count;
7
8     for(count=1;count<=10;count++)
9     {
10        printf("%d ",count);
11
12    }
13
14    system("pause");
15
16 }
```



Example 2 - Display asterisk

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int count;
7
8     for(count=1;count<=10;count++)
9     {
10        printf("* ");
11
12    }
13
14    system("pause");
15
16 }
17
```



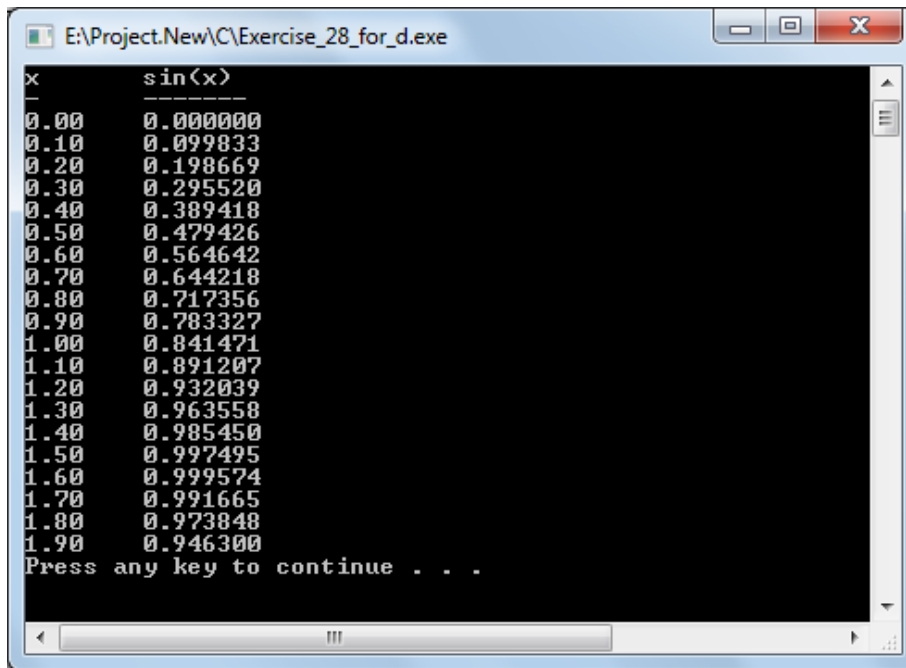
Example 3 - Display asterisk (based on user REQUEST!)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int count;
7     int number;
8
9     printf("How much star u want??\n");
10    scanf("%d",&number);
11
12
13    for(count=1;count<=number;count++)
14    {
15        printf("* ");
16
17    }
18
19    system("pause");
20    return 0;
21 }
22
```



Example 4 - Display sin(x)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main()
6 {
7     float count;
8
9     printf("x\tsin(x)\n");
10    printf("-\t-----\n");
11
12    for(count=0;count<=2.0;count=count+0.1) //simplified count+=0.1
13    {
14        printf("%0.2f\t%f\n",count,sin(count));
15    }
16
17    system("pause");
18    return 0;
19 }
20
21
```



```
E:\Project.New\C\Exercise_28_for_d.exe
x      sin(x)
-----
0.00   0.000000
0.10   0.099833
0.20   0.198669
0.30   0.295520
0.40   0.389418
0.50   0.479426
0.60   0.564642
0.70   0.644218
0.80   0.717356
0.90   0.783327
1.00   0.841471
1.10   0.891207
1.20   0.932039
1.30   0.963558
1.40   0.985450
1.50   0.997495
1.60   0.999574
1.70   0.991665
1.80   0.973848
1.90   0.946300
Press any key to continue . . .
```

while looping

- The while loop can be used if you don't know how many times a loop must run (sometimes) until the condition is met. The statement of while is:

```
while( condition )  
{ Code to execute while the condition is true }
```

- [while loop/repetition](#) statement : "For example,

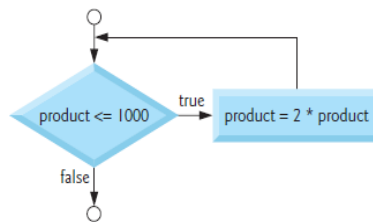
- Pseudo-code:

```
while product is below than 100  
  repeat product with three (3)
```

- C syntax:

```
product = 3;  
while ( product <= 100 ) {  
  product = 3 * product;  
} /* end while */
```

- Flowchart:



Example 1 – Display 20 to 0. (Decrement numbers)

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3 #include <math.h>  
4  
5 int main()  
6 {  
7     int count;  
8  
9     count = 20;  
10  
11     while( count >= 0 )  
12     { printf("%d ", count);  
13       count = count - 1;  
14     }  
15  
16     system("pause");  
17     return 0;  
18 }  
19
```

```
E:\Project.New\C\Exercise_29_while_a.exe  
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0  
any key to continue . . .
```

[Example 2](#) – We add 0+1+2+3+4+5+6+.....+100

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main()
6 {
7     int count;
8     int sum;
9
10    sum = 0;
11    count = 0;
12
13    printf("No.\tSum\n");
14    while( count<=100 )
15    {
16        sum = count + sum;
17        printf("%d\t%d\n",count ,sum) ;
18        count=count+1;
19    }
20
21    system("pause");
22    return 0;
23 }
```

```
E:\Project.New\C\Exercise_29_while_b.exe
No. Sum
0 0
1 1
2 3
3 6
4 10
5 15
6 21
7 28
8 36
9 45
10 55
11 66
12 78
13 91
14 105
15 120
16 136
17 153
18 171
19 190
20 210
21 231
22 253
23 276
24 300
25 325
26 351
27 378
28 406
29 435
30 465
31 496
32 528
33 561
34 595
35 630
36 666
37 703
38 741
39 780
40 820
41 861
42 903
43 946
44 990
45 1035
46 1081
47 1128
48 1176
49 1225
50 1275
51 1326
52 1378
53 1431
54 1485
55 1540
```

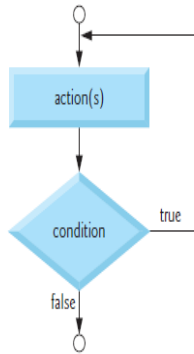
do..while looping

- [do..while loop/repetition](#) statement :

- C syntax:

```
do {  
    statement  
} while ( condition );
```

- Flowchart:



Example 1 - count -10 to 10

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3  
4 int main()  
5 {  
6     int count=-10;  
7  
8     do  
9     {  
10        printf("%d\n",count);  
11        count++;  
12  
13    }while(count<=10);  
14  
15    system("pause");  
16    return 0;  
17 }  
18
```

The screenshot shows a Windows command prompt window titled 'E:\Project.New\C\Exercise_30_do_while_a.exe'. The window displays the output of the program, which is a list of integers from -10 to 10, each on a new line. Below the list, the text 'Press any key to continue . . .' is visible. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

- **Difference between while and do while loop**

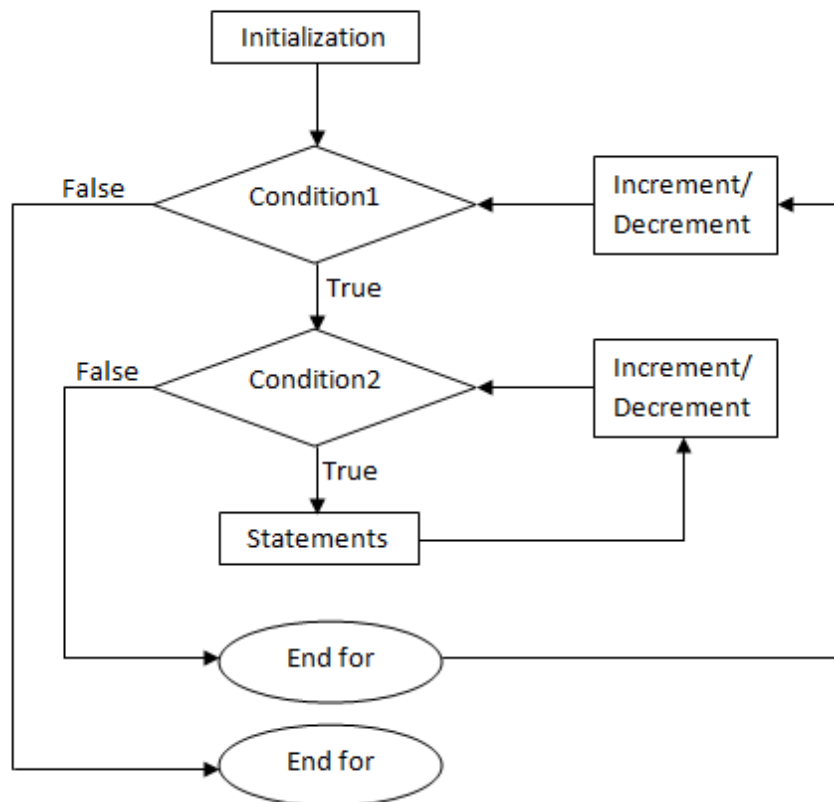
- The do while statement is similar to the while statement **except** that its termination condition is at the end of the body of the loop only. Thus, you want to use a do statement, if you want to perform the body of the loop at least once, regardless of the condition.

nested for

- A for loop inside another for loop is called nested for loop.
- Syntax of Nested for loop:

```
for (initialization; condition; increment/decrement)
{
    statement(s);
    for (initialization; condition; increment/decrement)
    {
        statement(s);
        ... ..
    }
    ... ..
}
```

Flowchart of Nested for loop



nested for

- **Example 1:** C program to print all the composite numbers from 2 to a certain number entered by user.

```
#include<stdio.h>
#include<math.h>
int main()
{
    int i,j,n;
    printf("Enter a number:");
    scanf("%d",&n);
    for(i=2;i<=n;i++)
    {
        for(j=2;j<=(int)pow(i,0.5);j++)
        {
            if(i%j==0)
            {
                printf("%d is composite\n",i);
                break;
            }
        }
    }
    return 0;
}
```

Output

```
Enter a number:15
4 is composite
6 is composite
8 is composite
9 is composite
10 is composite
12 is composite
14 is composite
15 is composite
```

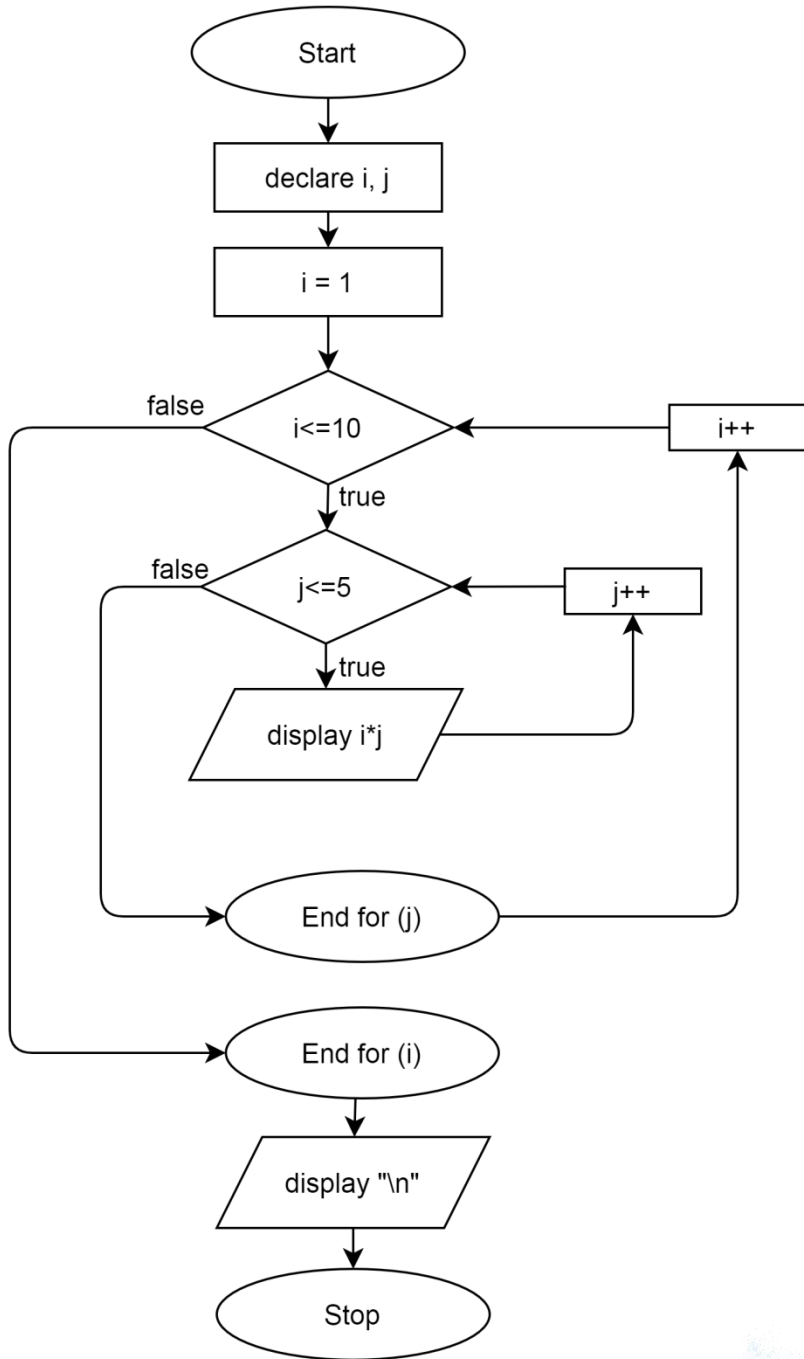
- **Example 2:** C program to print multiplication table from 1 to 5

```
1  /**
2   * C program to print multiplication table from 1 to 5
3   */
4  #include <stdio.h>
5
6  int main()
7  {
8      /* Loop counter variable declaration */
9      int i, j;
10
11     /* Outer loop */
12     for(i=1; i<=10; i++)
13     {
14         /* Inner loop */
15         for(j=1; j<=5; j++)
16         {
17             printf("%d\t", (i*j));
18         }
19
20         /* Print a new line */
21         printf("\n");
22     }
23
24     return 0;
25 }
```

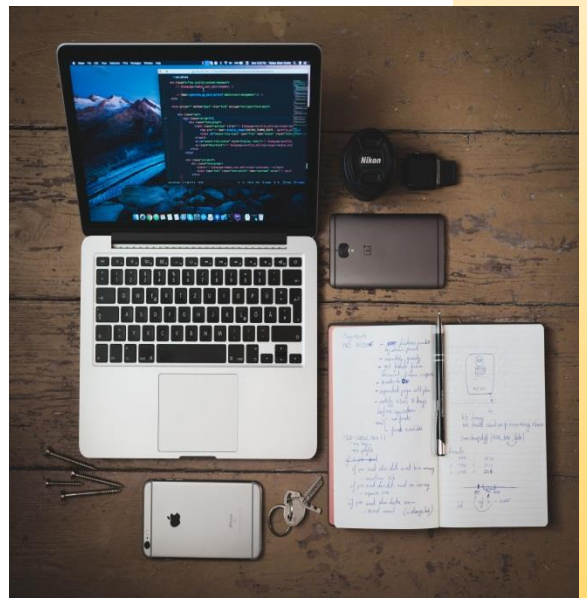
Output:

| | | | | |
|----|----|----|----|----|
| 1 | 2 | 3 | 4 | 5 |
| 2 | 4 | 6 | 8 | 10 |
| 3 | 6 | 9 | 12 | 15 |
| 4 | 8 | 12 | 16 | 20 |
| 5 | 10 | 15 | 20 | 25 |
| 6 | 12 | 18 | 24 | 30 |
| 7 | 14 | 21 | 28 | 35 |
| 8 | 16 | 24 | 32 | 40 |
| 9 | 18 | 27 | 36 | 45 |
| 10 | 20 | 30 | 40 | 50 |

• Flowchart for Example 2



Jump Statements



BREAK statements



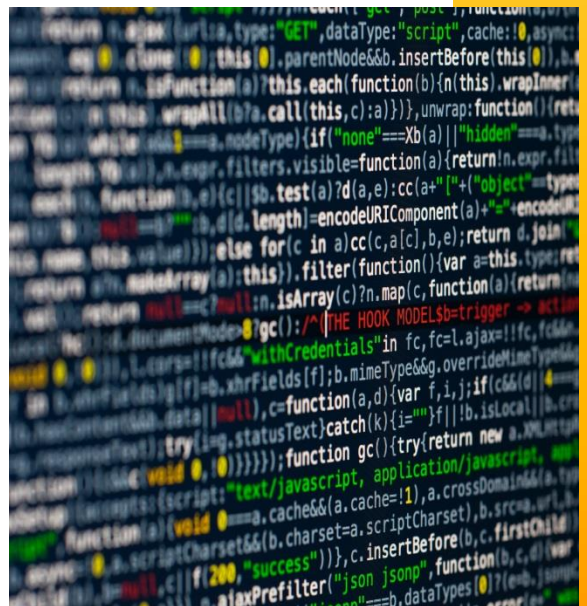
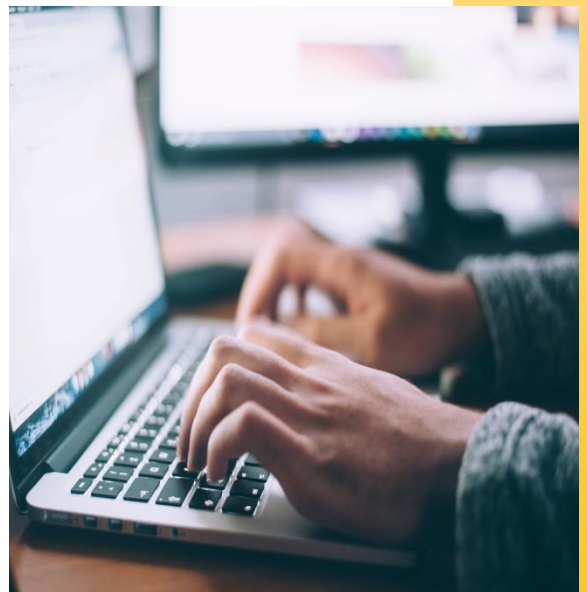
CONTINUE statement



RETURN statements

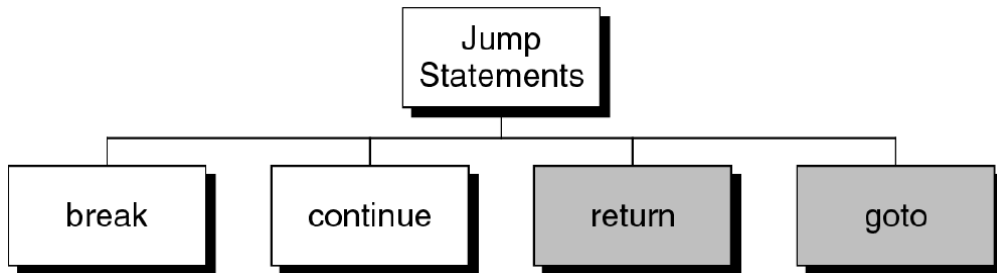


GOTO statements



Jump statements

- In addition to the sequence, repetition and selection , C also provides jump statements.
- The statements allow program control to be transferred from one part of the program to another program unconditionally.
- There are four jump statements:



break statement

Break statement – The break statement in C programming language has following two usage:

- When the break statement is encountered **inside a loop**, the loop is immediately terminated and program control resumes at the next statement following the loop.
- It can be **used to terminate a case in the switch statement** (covered in the previous chapter).
- It causes a loop to terminate

Example:

```
for (n=10; n>0; n=n-1)
{
  if (n<8) break;
  printf("%d ", n);
}
```

Output:

10 9 8

- break statement:
 - It performs a **one-way transfer** of control to another line of code;
 - The set of *identifier* names following a **goto** has its own name space so the names do not interfere with other identifiers. Labels cannot be redeclared.

```

1 //program to show BREAK;
2
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 /* function main begins program execution */
7 int main(void)
8 {
9     int counter; /* initialize counter */
10
11     for(counter = 1; counter <=20; counter++){
12         if(counter==8)
13             break;
14
15         printf("%d ",counter);
16     }
17
18     printf("\n\nBreak is at number %d",counter);
19
20     printf("\n\n" );
21     system("pause");
22     return 0;
23 }
24

```

continue statement

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

- The **continue** statement, when executed in a **while**, **for** or **do...while** statement, skips the remaining statements in the body of that control statement and performs the next iteration of the loop.

```

1 //program to show CONTINUE
2
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 /* function main begins program execution */
7 int main(void)
8 {
9     int counter; /* initialize counter */
10
11     for(counter = 1; counter <=20; counter++){
12         if(counter==8)
13             continue;
14
15         printf("%d ",counter);
16     }
17
18     printf("\n\nNumber is skip at integer 8",counter);
19
20     printf("\n\n" );
21     system("pause");
22     return 0;
23 }

```

continue statement

```
for (n=10; n>0; n=n-1)
{
    if (n%2==1) continue;
    printf("%d ",n);
}
```

Output:

10 8 6 4 2

Continue examples using for statement and while statement :

Example:

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

    for (n=10; n>0; n=n-1)
    {
        if (n%2==1) continue;
        printf("%d ",n);
    }
    system ("pause");
}
```

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

    n = 10;
    while (n>0)
    {
        printf("%d ",n);
        if (n%2==1) continue;
        n = n -1;
    }
    system ("pause");
}
```

Example:

```
n = 10;
while (n>0)
{
    printf("%d ",n);
    if (n%2==1) continue;
    n = n -1;
}
```

Output:

10 9 9 9 9 9

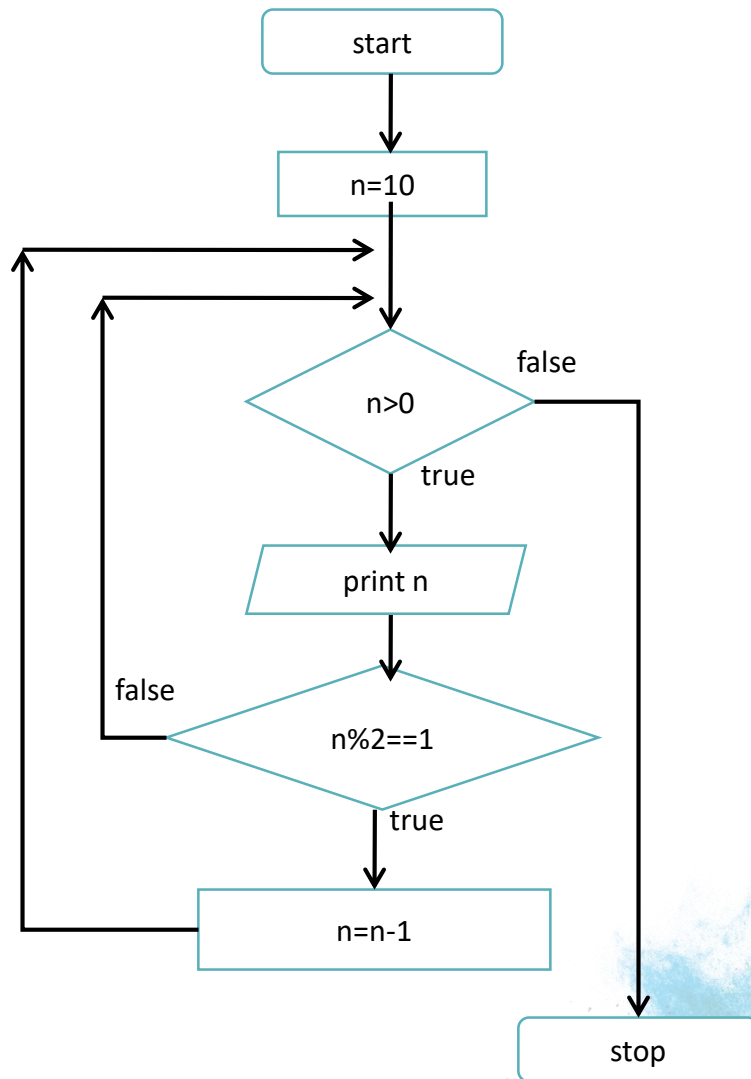
The loop then prints number 9 over and over again. It never stops.

continue statement

Example:

```
n = 10;
while (n>0)
{
printf("%d ",n);
if (n%2==1) continue;
n = n -1;
}
```

Transfer to the loop condition



goto statement

`goto` - The `goto` statement is used to alter the normal sequence of program execution by transferring control to some other part of the program unconditionally. In its general form, the `goto` statement is written as `goto label`;

where the label is an identifier that is used to label the target statement to which the control is transferred.

- `goto` statement:

- It performs a **one-way transfer** of control to another line of code;
- The set of *identifier* names following a **goto** has its own name space so the names do not interfere with other identifiers.
- **Labels cannot be redeclared.**

```
1 //program to do switch statement
2
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main(int numeric)
7 {
8
9     value:
10    printf("Enter the integer value between 1 to 5:\n\n");
11    printf("1 : For Blue Foreground\n");
12    printf("2 : For GREEN Foreground\n");
13    printf("3 : For RED Foreground\n");
14    printf("4 : For PURPLE Foreground\n");
15    printf("5 : For YELLOW Foreground\n");
16    printf("6 : To CLEAR SCREEN\n");
17    printf("7 : For EXIT\n");
18
19    printf("\nYour value is:");
20    scanf("%d",&numeric);printf("\n");
21
22    switch(numeric){
23        case 1: printf("\nBLUE\n\n");
24                system("color 10");
25                break;
26        case 2: printf("\nGREEN\n\n");
27                system("color 20");
28                break;
29        case 3: printf("\nRED\n\n");
30                system("color 40");
31                break;
32        case 4: printf("\nPURPLE\n\n");
33                system("color 50");
34                break;
35        case 5: printf("\nYELLOW\n\n");
36                system("color 60");
37                break;
38        case 6: system("cls");
39                break;
40        case 7: goto exit;
41                break;
42
43        default:printf("\nWrong Number!..\nplease enter the integer value between 1 to 5\n\n");
44    }
45
46    goto value;
47
48
49    exit:
50    system("pause");
51    return 0;
52 }
```

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void main()
5 {
6     int pilih;
7
8     value:
9     printf("\n1.merah\n");
10    printf("2.biru\n");
11    printf("3.kuning\n");
12    printf("4.exit\n");
13    printf("sila pilih no brp:");
14    scanf("%d",&pilih);
15
16    switch(pilih)
17    {
18        case 1:
19            printf("awak pilih merah\n");|
20            break;
21        case 2:
22            printf("awak pilih biru\n");
23            break;
24        case 3:
25            printf("awak pilih kuning\n");
26            break;
27        case 4:
28            goto exit;
29            break;
30        default:
31            printf("\nsalah pilih. pilih balik ");
32    }
33
34    goto value;
35    exit:
36        system("pause");
37
38 }

```

CHAPTER 5



Function

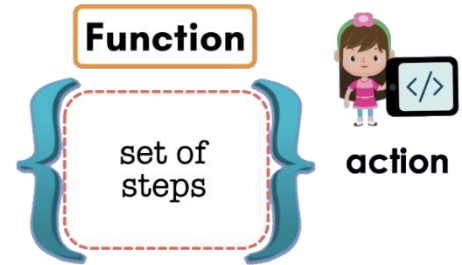


Functions

- A function is a group of statements that together perform a task.
- Every C program has at least one function, which is **main()**, and most programs can define additional functions.
- You can divide up your code into separate functions, where each function performs a specific task.
- A function **declaration** tells the compiler about a function's name, return type, and parameters.
- A function **definition** provides the actual body of the function.

Types of function

- There are two type of function:-
 - a. Predefined function
 - b. User-defined function



a) Predefined function

- Predefined functions are functions that have been written and we can use them in our C statements.
- These functions are also called as 'library functions'. These functions are provided by system. These functions are stored in library files. Example:-
 - scanf()
 - printf()
 - strcpy
 - strlwr
 - strcmp
 - strlen
 - strcat

b) User-defined function

- The functions which are created by user for program are known as 'User defined functions'.
- Advantages :
 - It is easy to use.
 - Debugging is more suitable for programs.
 - It reduces the size of a program.
 - It is easy to understand the actual logic of a program.
 - Highly suited in case of large programs.
 - By using functions in a program, it is possible to construct modular and structured programs.

User-defined function

Syntax:

```
void main()
{
    // Function prototype
    <return_type><function_name>([<argu_list>]);

    // Function Call
    <function_name>([<arguments>]);
}
// Function definition
<return_type><function_name>([<argu_list>]);
{
    <function_body>;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
void add(); //function declare

void add()
{
    int a,b,c;
    system("cls");
    printf("\nEnter Any 2 number: ");
    scanf("%d %d", &a,&b);
    c=a+b;
    printf("\nAddition is : %d\n\n", c);
    system("pause");
}

void main()
{
    add();
    add();
    add();
    getch();
}
```

```
Enter Any 2 Numbers : 23 6
Addition is : 29_
```

Three steps in using functions

1. Declare the function:

- Known as function declaration or function prototyping.
- Write a **function prototype** that specifies:
 - the **name** of the function
 - the **type** of its return value
 - its list of **arguments** and their types

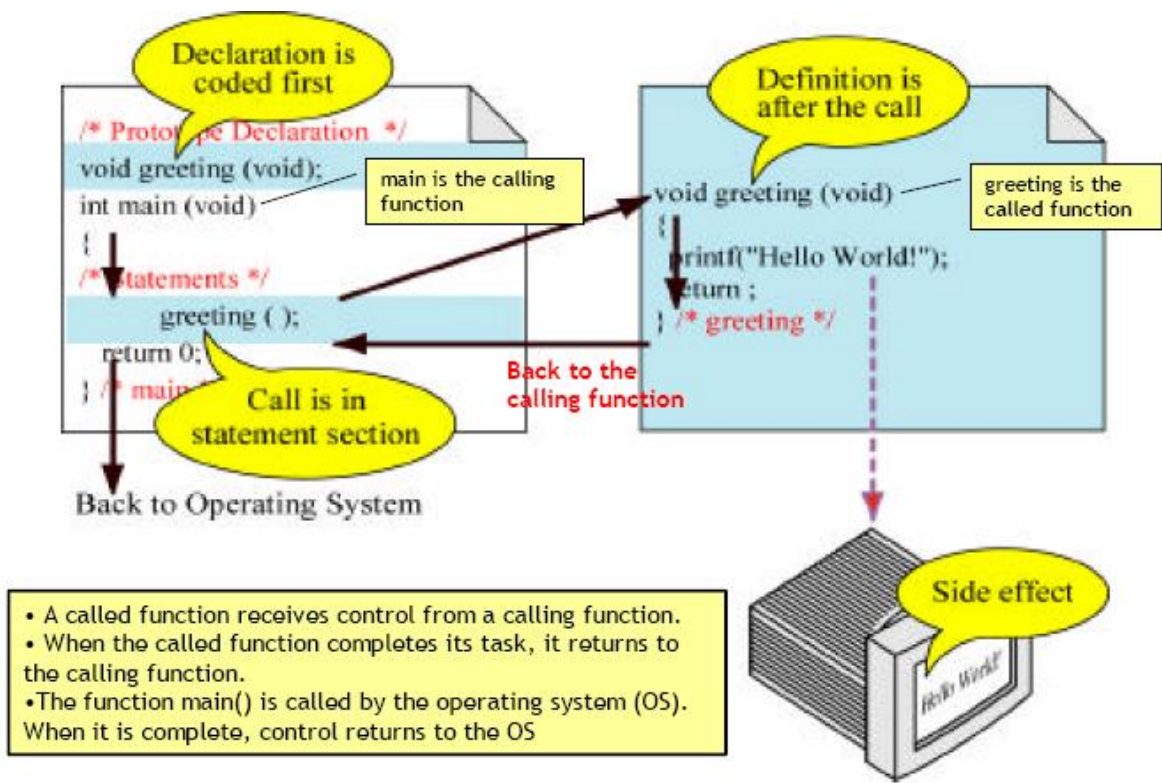
2. Define the function:

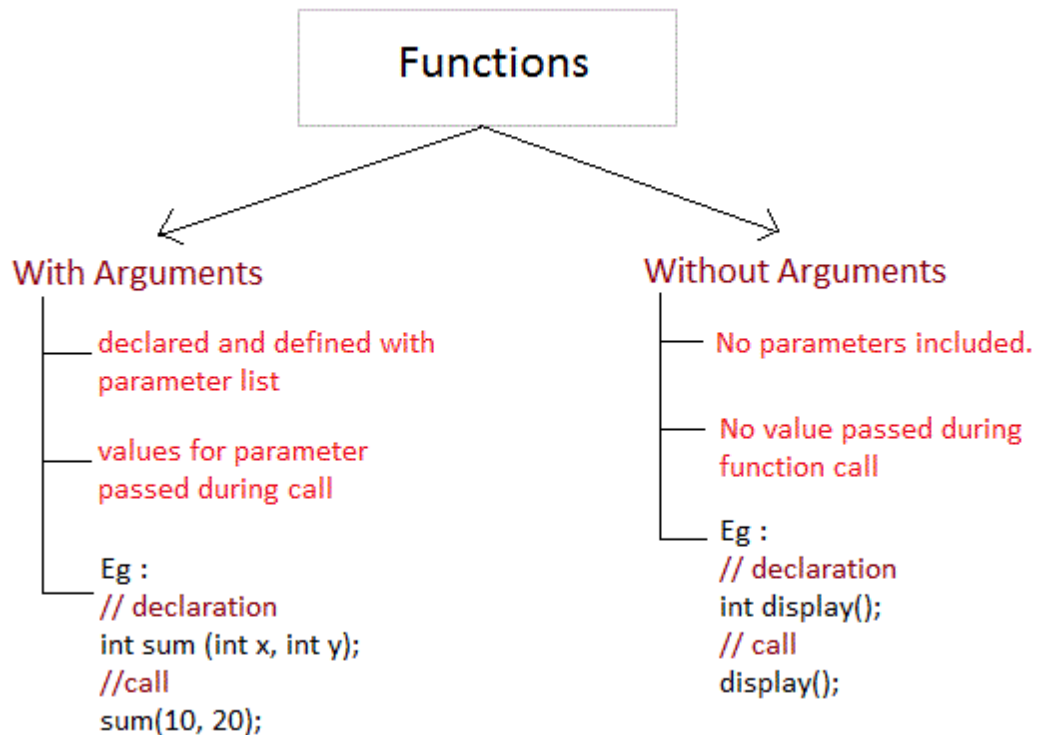
- Known as function definition or function implementation.
- Write the block of statements (body) of the function to define processes should be done by the function.

3. Call the function:

- Known as function call or function invocation.
- Call the name of the function in order to execute it.

Figure : Declaring, calling and defining functions

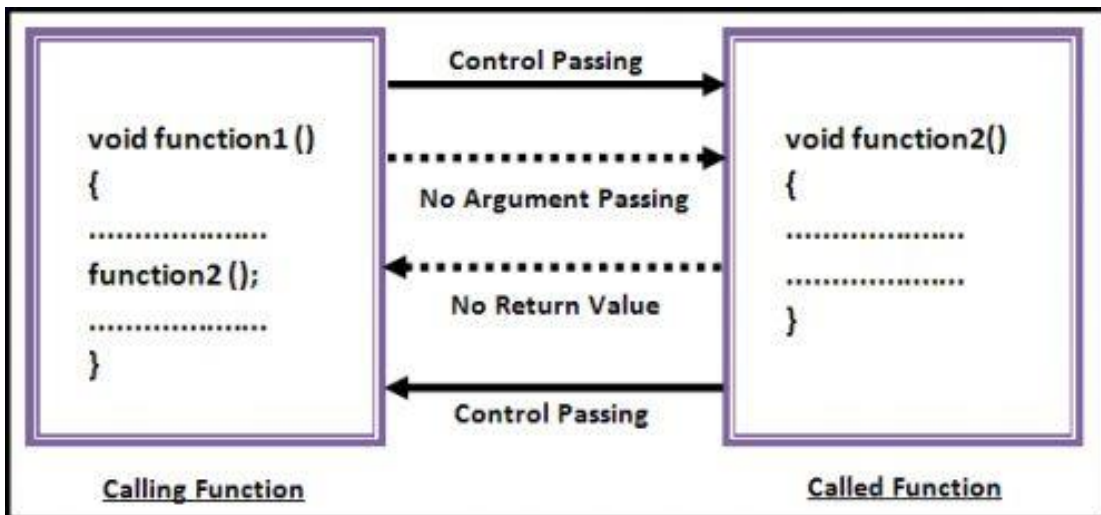




FUNCTIONS

- there are five types of functions and they are:
 1. Functions with no arguments and no return values.
 2. Functions with arguments and no return values.
 3. Functions with arguments and return values.
 4. Functions that return multiple values.
 5. Functions with no arguments and return values.

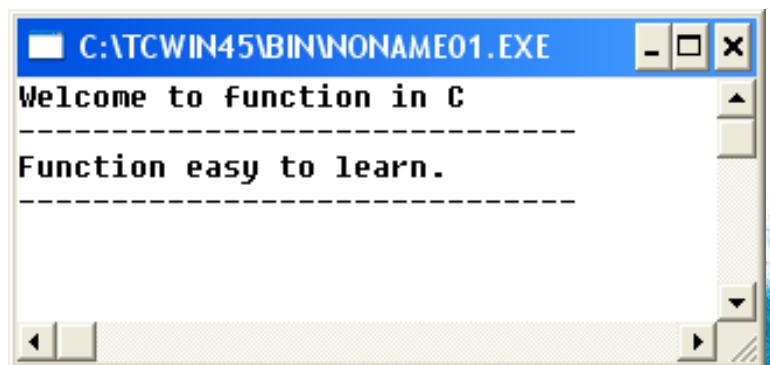
1. Functions with no arguments and no return value.



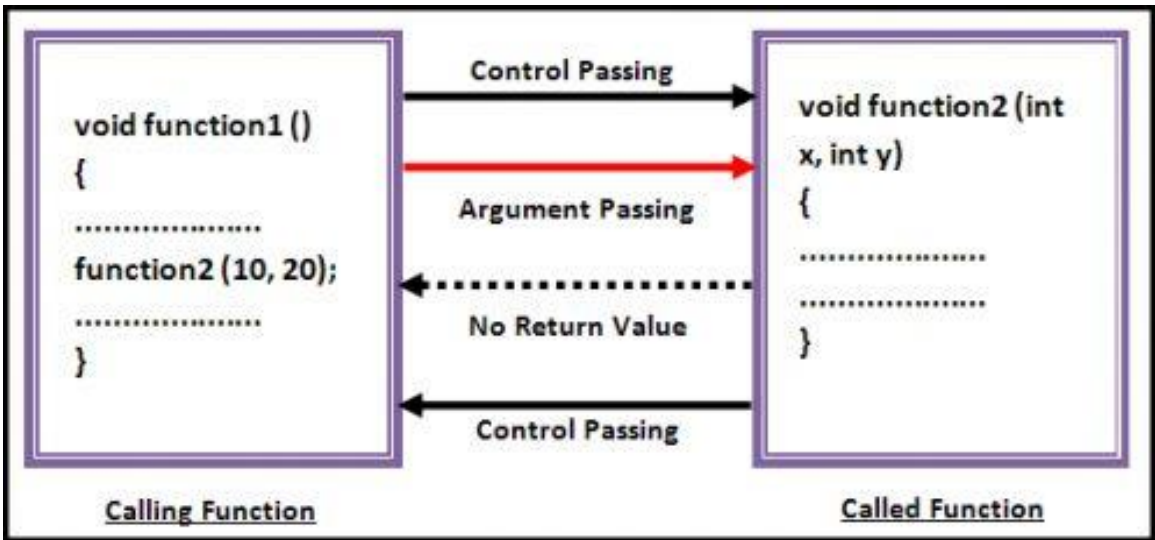
```
#include<stdio.h>
#include<conio.h>

void printline()
{
    int i;
    printf("\n");
    for(i=0;i<30;i++)
    {
        printf("-");
    }
    printf("\n");
}

void main()
{
    clrscr();
    printf("Welcome to function in C");
    printline();
    printf("Function easy to learn.");
    printline();
    getch();
}
```



2. Functions with arguments and no return value.



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

```
#include<conio.h>
```

```
void add(int x, int y)
```

```
{
```

```
int result;
```

```
result = x+y;
```

```
printf("Sum of %d and %d is %d.\n\n",x,y,result);
```

```
}
```

```
void main()
```

```
{
```

```
clrscr();
```

```
add(30,15);
```

```
add(63,49);
```

```
add(952,321);
```

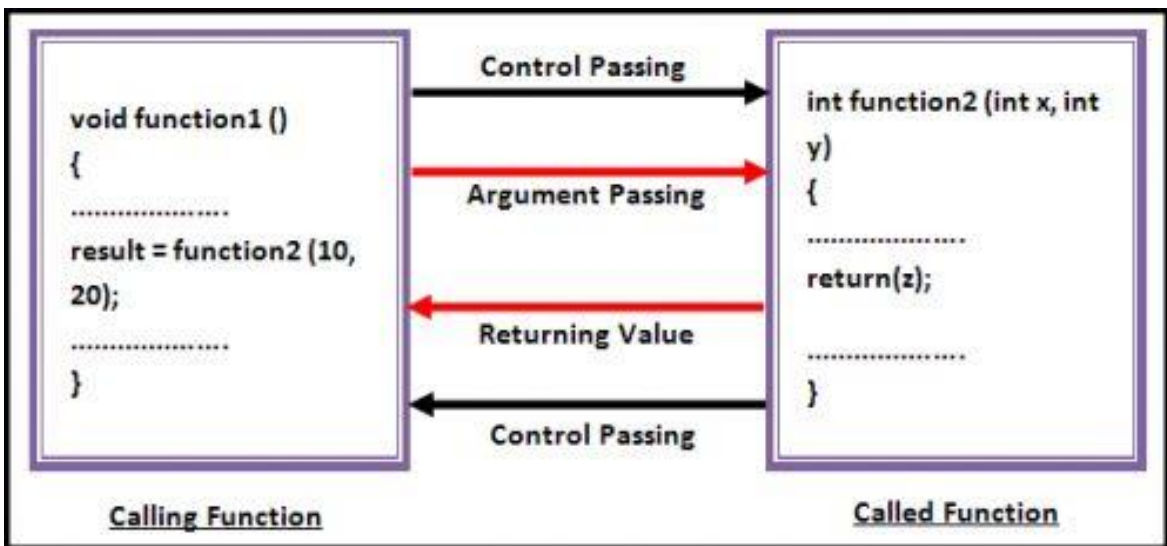
```
getch();
```

```
}
```

The screenshot shows a command prompt window titled 'C:\TCWIN45\BIN\NONAME01.EXE'. The output of the program is displayed as follows:

```
Sum of 30 and 15 is 45.  
Sum of 63 and 49 is 112.  
Sum of 952 and 321 is 1273.
```

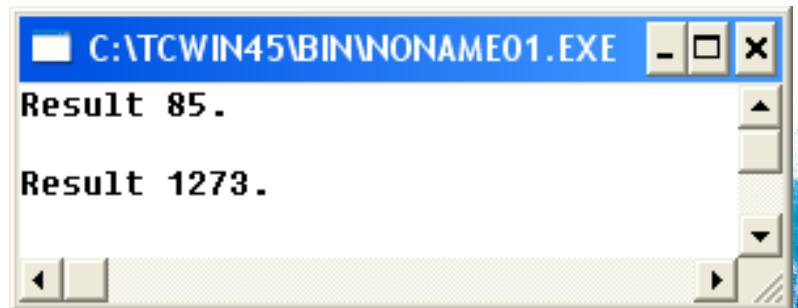
3. Functions with arguments and return value.



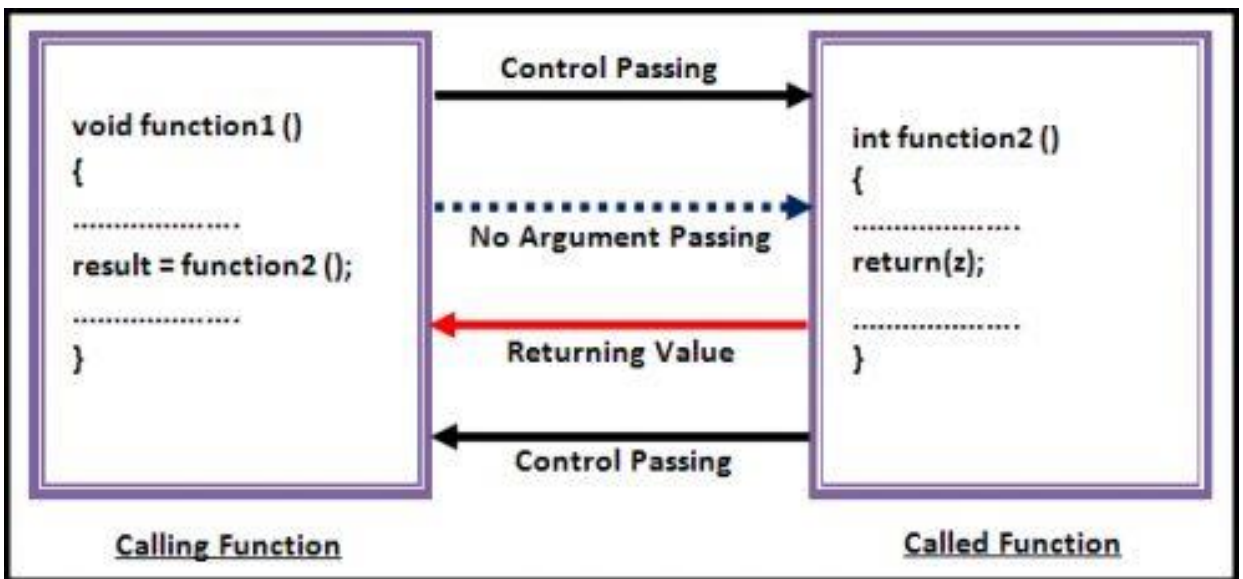
```
#include<stdio.h>
#include<conio.h>

int add(int x, int y)
{
    int result;
    result = x+y;
    return(result);
}

void main()
{
    int z;
    clrscr();
    z = add(952,321);
    printf("Result %d.\n\n",add(30,55));
    printf("Result %d.\n\n",z);
    getch();
}
```

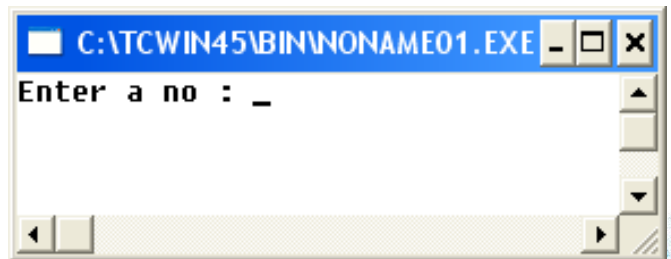


4. Functions with no arguments but returns value.

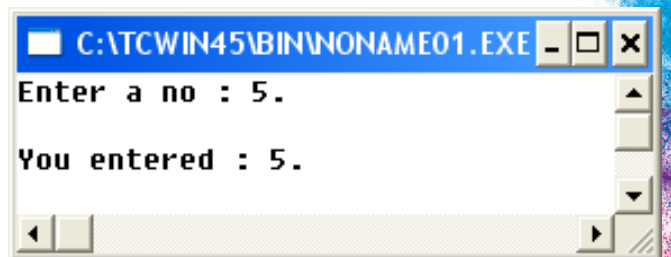


```
#include<stdio.h>
#include<conio.h>
```

```
int send()
{
    int no1;
    printf("Enter a no : ");
    scanf("%d",&no1);
    return(no1);
}
```



```
void main()
{
    int z;
    clrscr();
    z = send();
    printf("\nYou entered : %d.", z);
    getch();
}
```

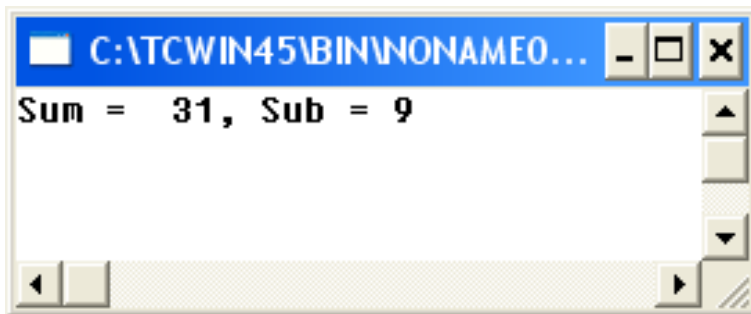


5. Functions that return multiple values.

```
#include<stdio.h>
#include<conio.h>

void calc(int x, int y, int *add, int *sub)
{
    *add = x+y;
    *sub = x-y;
}

void main()
{
    int a=20, b=11, p,q;
    clrscr();
    calc(a,b,&p,&q);
    printf("Sum = %d, Sub = %d",p,q);
    getch();
}
```



Declaring a function

```
return_type function_name ( formal_parameter_list );
```

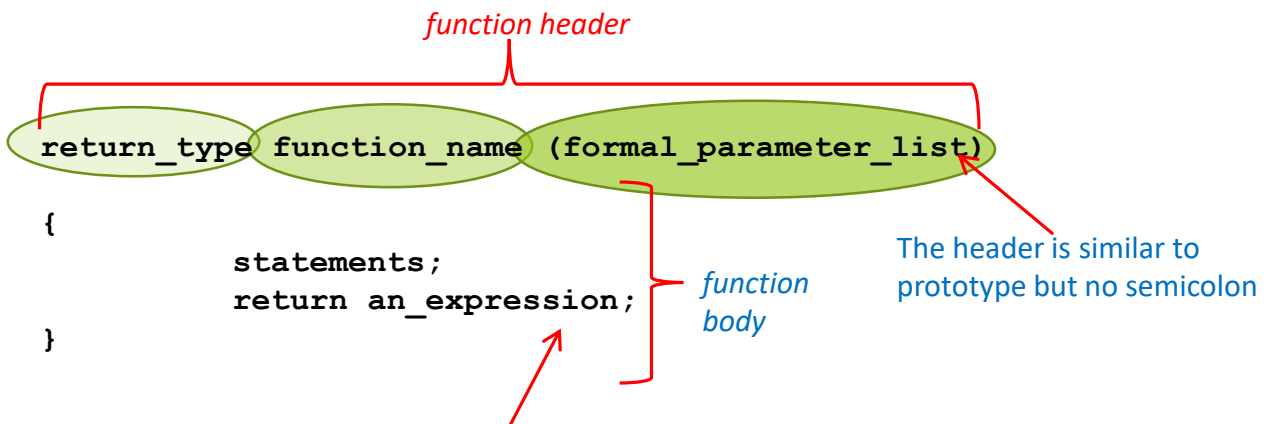
- The syntax of a function declaration (formally called *prototype*) contains:
 - The type of the return value of the function
 - if the function does not return anything, the type is **void**
 - if *return_type* is not written the Compiler will assume it as **int**
 - The name of the function
 - same rules as for variable naming
 - A list of formal parameter made up of its names and its types. They are enclosed in parentheses
 - The prototype must be terminated by a semicolon
- Function prototypes are usually written between the preprocessor directives and main().

Examples of function prototypes

- `float avrg(int num1, int num2, int num3);`
 - Function `avrg` takes three integers as parameters and returns a floatingpoint value.
- `void mix(double num1, int num2);`
 - This function receives a double and an integer value as parameters. But, it does not return any value.
- `void greeting(void);`
 - This function does not receive any parameter and also does not return any value.
- `calculate();`
 - The return type and the formal parameters are not written.
 - This function does not receive any parameter. It returns an integer value.

Defining a function

- The syntax of a function definition is:



If the *return_type* is not **void**, the function must have a **return statement**.

But, if the *return_type* is **void**, the **return** statement is optional or just put **return;** (**without an_expression**)

Calling a function

- The name of a function is called in order to execute the function.
- A called function receives control from a calling function.
- When the called function completes its task, it returns to the calling function.
- The called function may or may not returns a value to the calling function

Functions that return a value can be used in an expression or as a statement.

Example:

if given function definition as below:

```
float avrg(int a, int b, int c)
{
    return (a+b+c)/3.0;
}
```

All function calls below are valid

```
result = avrg(1,2,3) + avrg(4,5,6); // function calls are
                                     // used in an expression
avrg(1,2,3); // function call is used as a statement
printf("The average is %.2f", avrg(1,2,3) );
```

void function cannot be used in an expression because it does not return any value. It can only be used as a statement.

Example:

if given function definition as below:

```
void greeting(void)
{
printf("Hello");
return;
}
```

Function call below would be an error

```
result = greeting(); // Error! greeting() is a void
function
```

- Formal parameters are variables that are declared in the header of the function definition
- Actual parameters are the expressions in the calling statement
- When making a function call, the formal and actual parameters must match exactly in type, order and number.
- The parentheses is compulsory, even when no parameters present. This is the way, how the compiler knows an identifier either it is a function or a variable.

• Example:

```
greeting; // Error. greeting is a function.
//So, it must have the ()
// eventhough no parameter present
```

Figure : void function with parameters

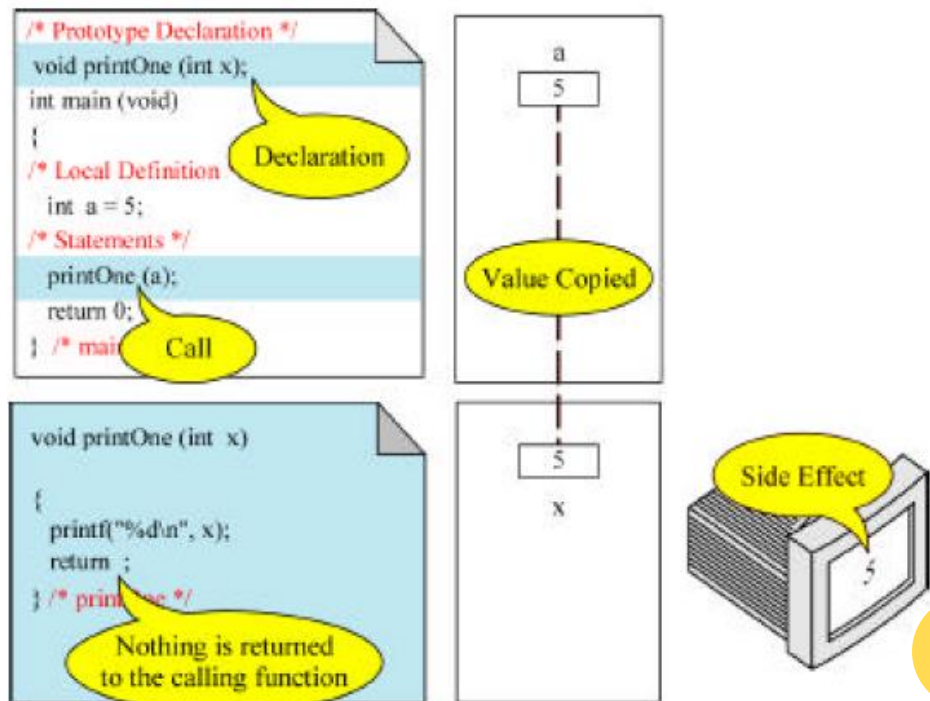
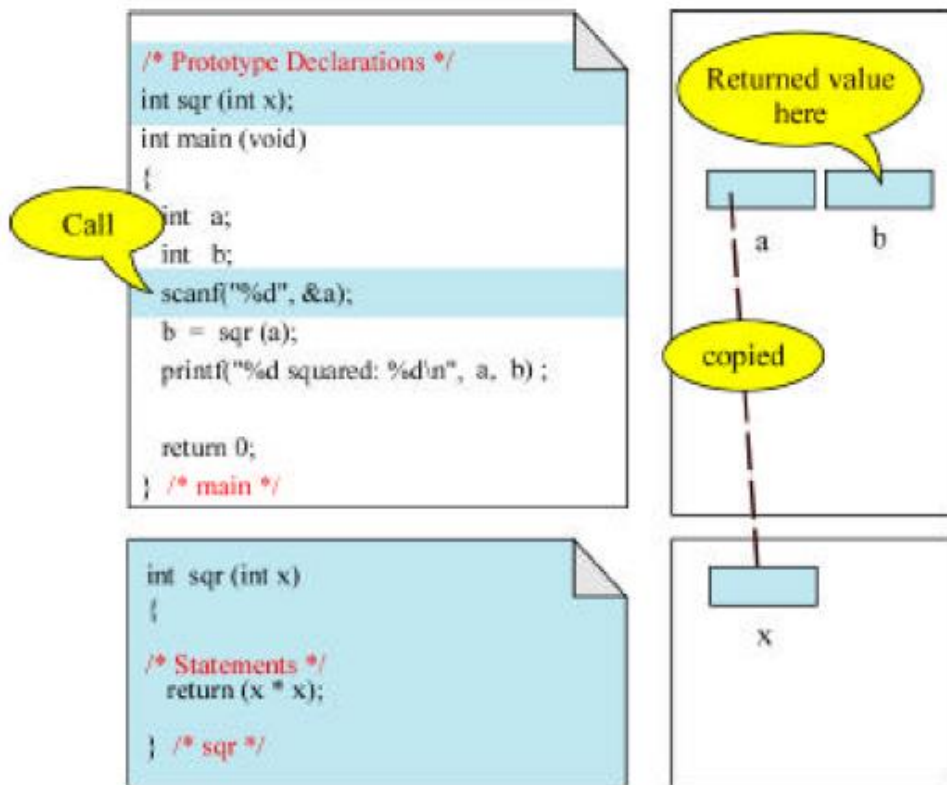


Figure : Function that returns a value



Function that calls itself is known as recursive function

Example:

```
int factorial(int n)
{
    if (n>1) return n * factorial(n-1);
    return 1;
}
```

This function calculates the factorial of n,

$$n! = n \times (n-1) \times (n-2) \times \dots \times 2 \times 1$$

At the first statement of the function definition, the function calls itself.

return statement

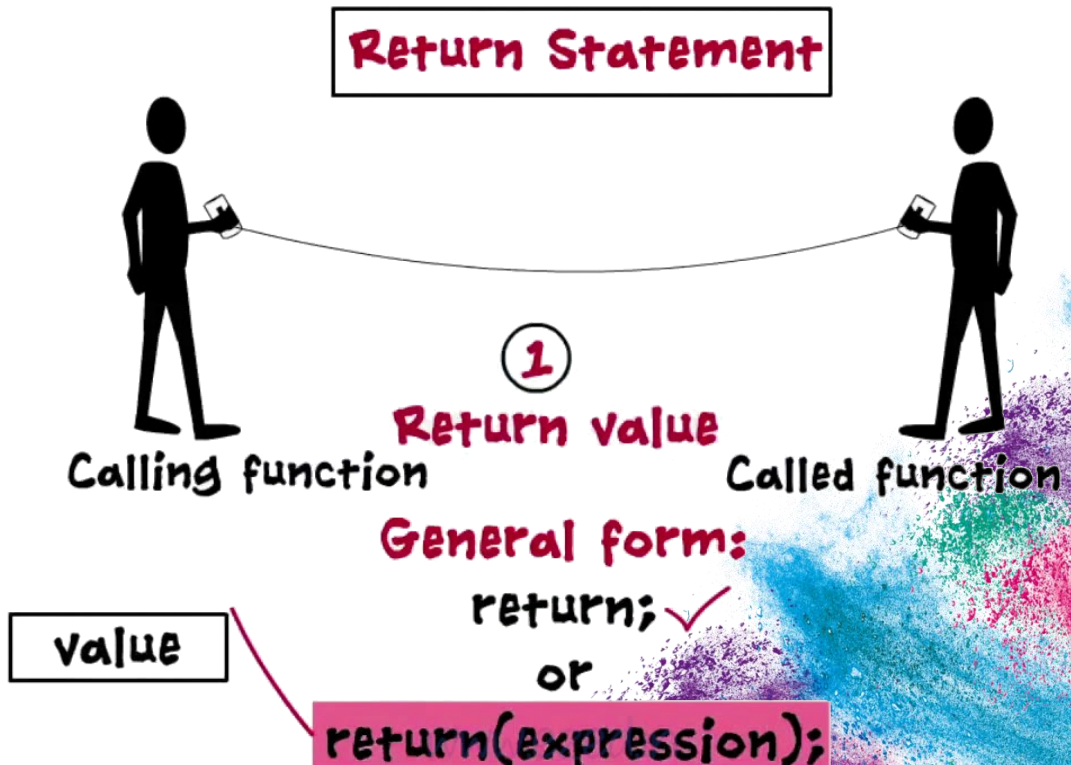
- A function returns a computed value back to the calling function via a return statement.
- A function with a non-void return type must always have a return statement.
- Code after a return statement is never executed.

The following function always returns 10.

```
int square (int n)
{
return 10;
n = n * n;
return n;
}
```

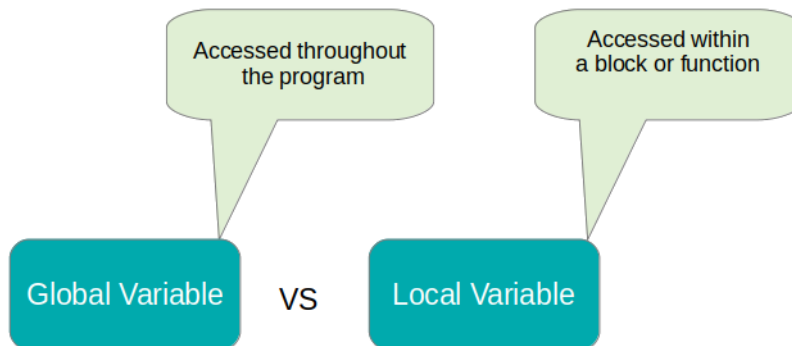
This line causes the control back to the calling function and ignores the rest of lines.

These two lines are ignored and never executed



Local & global variables

- Local variable is a variable declared inside a function.
 - This variable can only be used in the function.
- Global variable is a variable declared outside of any functions.
 - This variable can be used anywhere in the program



Example: Local vs. Global

```
#include<stdio.h>
void print_number(void);
```

```
int p;
```

p is declared outside of all functions. So, it is a global variable.

```
void main (void)
```

```
{
  int q = 5;
  printf("q=%d", q);
  p=10;
  print_number();
}
```

q is declared inside the function main. So, it is a local variable to the function.

```
void print_number(void)
```

```
{
  printf("%d", p);
  q = q + 5;
}
```

p can be used anywhere

```
void print_number(void)
```

```
{
  printf("%d", p);
  q = q + 5;
}
```

Error! q can only be used in the function main, because it is a local variable

Example: Local vs. Global

```
#include<stdio.h>
double compute_average (int num1, int num2);
void main (void)
{
double average;
int age1 = 18, age2 = 23;
average = compute_average (age1, age2);
return 0;
}
double average (int num1, int num2)
{
double average;
average = (num1 + num2) / 2.0;
return average;
}
```

Same variable names?!?
--it's OK; they're local to
their functions. Compiler treat
them as different variables.



Scope

- **Scope** determines the **area** of the program in which an identifier is visible (*ie. the identifier can only be used in that area*)
- Remember, identifier can be a variable, constant, function, etc.
- Examples:
 - Scope of a local variable : only in the function body where it was declared.
 - Scope of a global variable : everywhere in the program.
- Scope that enclosed in { } is called a **block**.
- Inner block can use identifiers that were declared outside of it.
 - eg. Any function can use any global variables.
- But outer block cannot use identifiers that were declared in inner block.
- Any block cannot use any identifier that was declared in other block.
 - eg. You cannot use any local variable from a function in another function.

```

/* This is a sample to demonstrate scope. The techniques
   used in this program should never be used in practice.
*/
#include <stdio.h>
int fun (int a, int b);

int main ( void )
{
  int a;
  int b;
  float c;
  ...
  { /* Beginning of nested block */
    float u = y / 2;
    float y;
    float z;
    ...
    z = a * b;
    ...
  } /* End of nested block */
  ...
} /* End of Main */

int fun (int i,
        int j)
{
  int a;
  int y;
  ...
} /* fun */

```

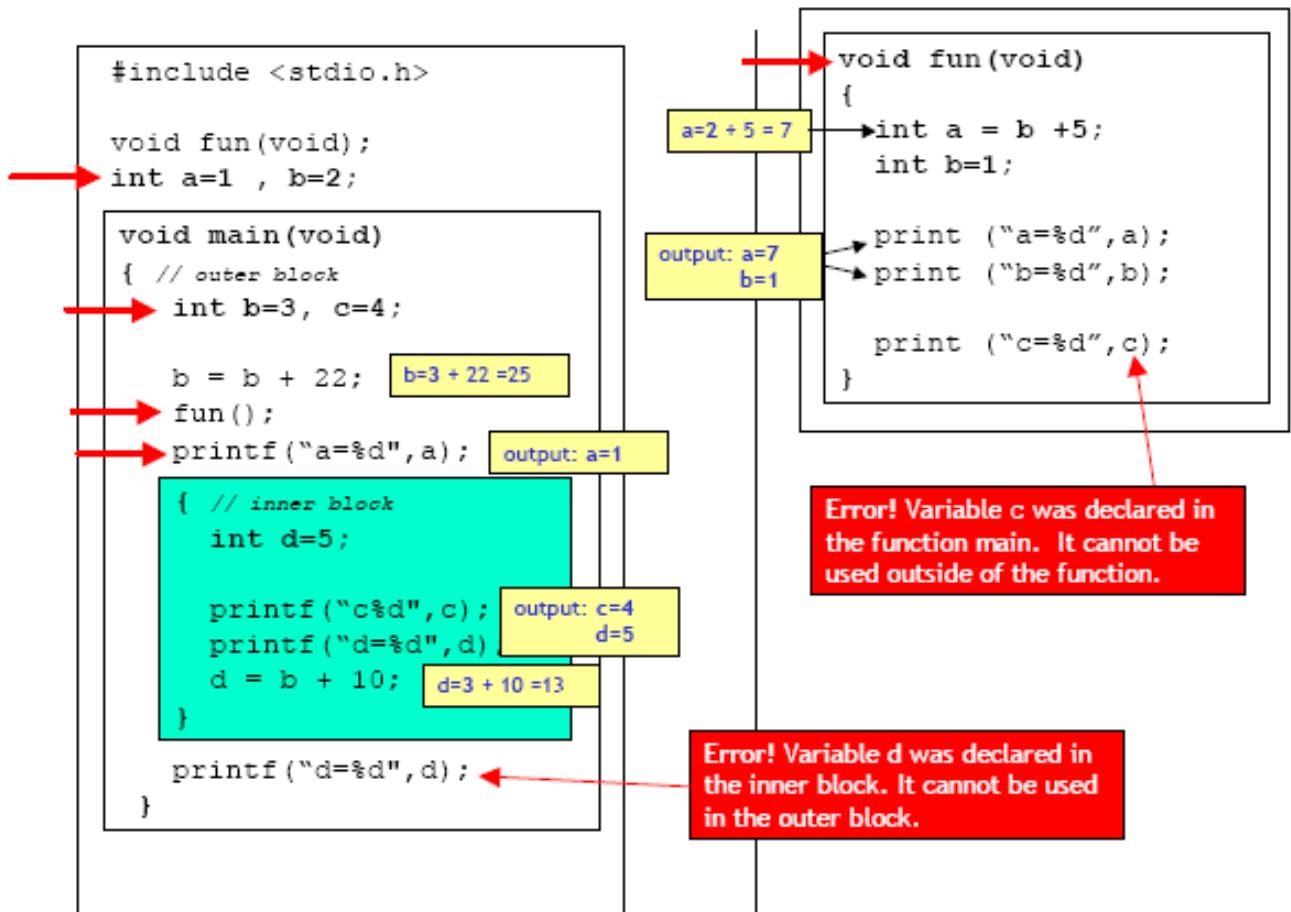
Global area

main's area

Nested block area

fun's area

Example: Scope of inner and outer block and function



Parameter Passing

- To call a function, we write its name and give it some information which are called parameters.
- Giving information to a *function call is called* parameter passing.
- You have learnt these:
 - formal parameters – parameters that are used in *function definition*
 - actual parameters – parameters that are used in *function call*
- In order to pass parameters, the actual and formal parameters must match exactly in type, order and number.
 - Eg. If you have defined a function with its formal parameter as an “output parameter”, you must use the ampersand (&) for its actual parameter when you call the function. Otherwise you will get a *syntax error* “Type mismatch”.
- Two types of passing:
 - Pass by value
 - Pass by reference

Pass by Value

- When a data is passed by value, a copy of the data is created and placed in a local variable in the called function.
- Pass by value does not cause side effect.
 - After the function call completed, *the original data remain* unchanged.

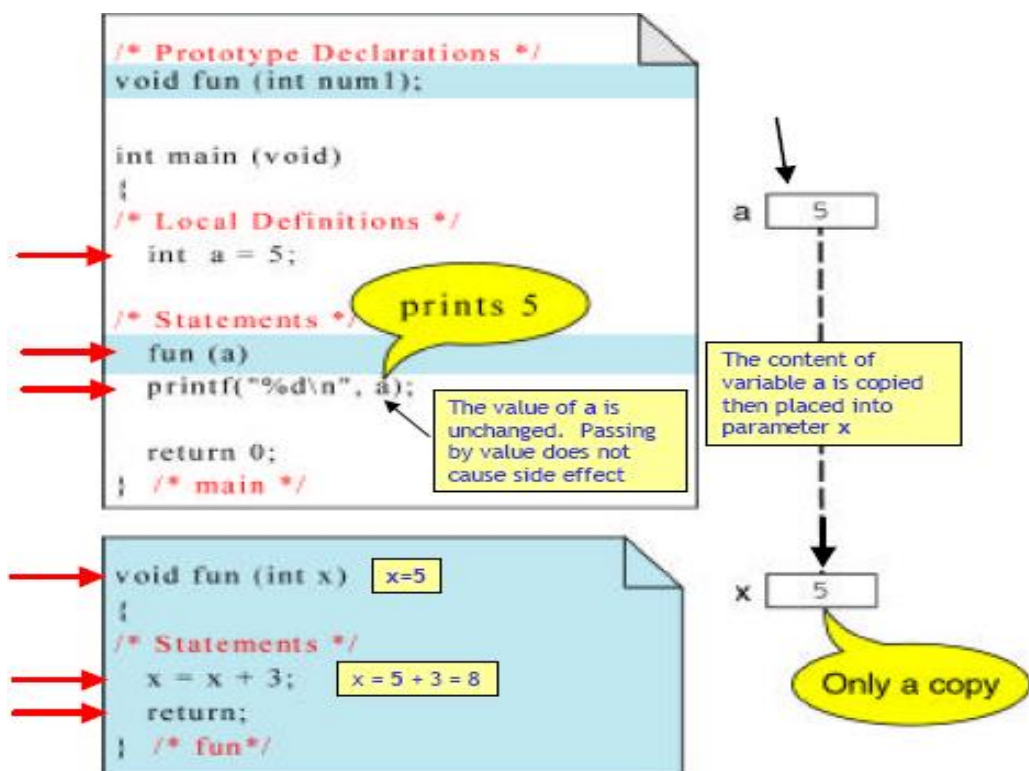
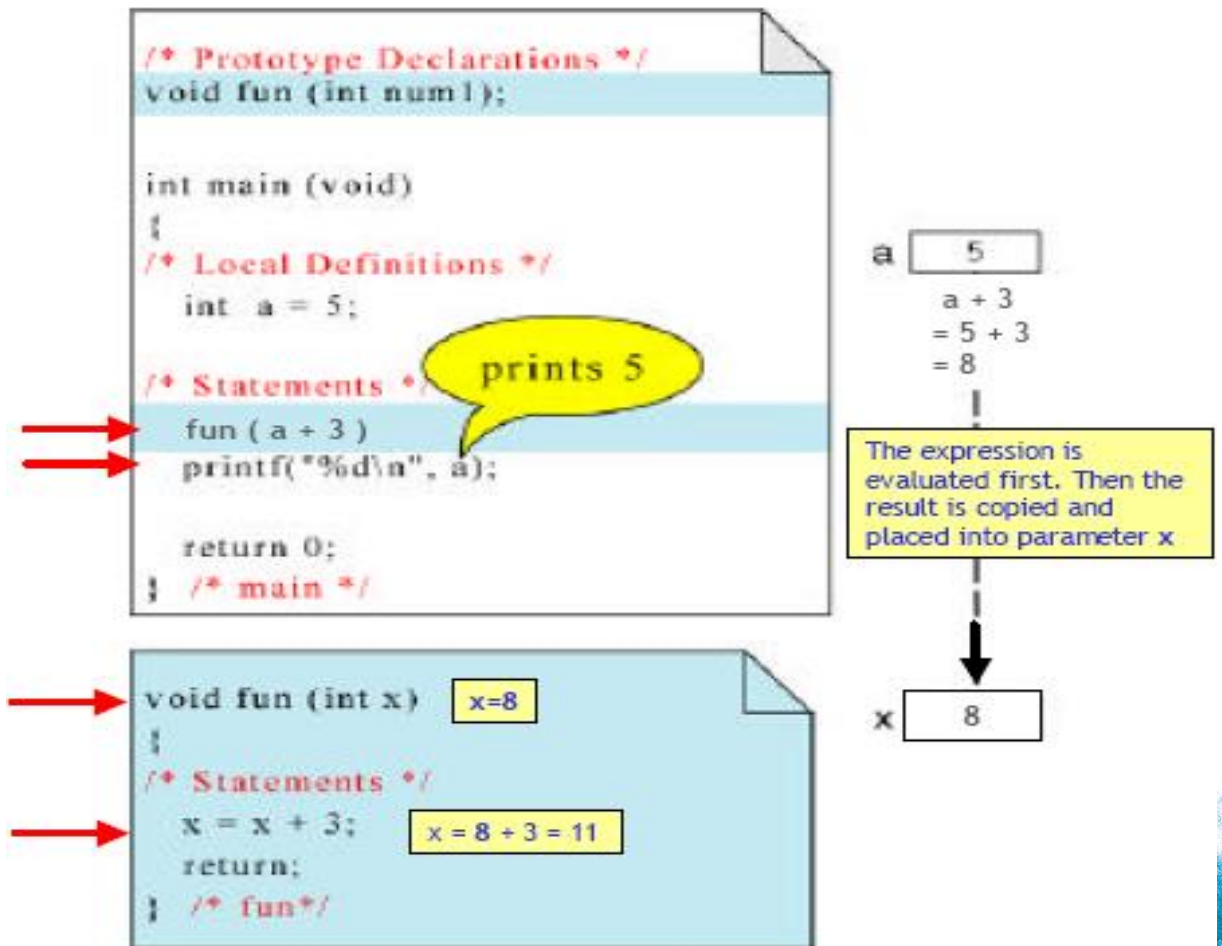


Figure : Pass by value

Pass by Value

- You have been introduced with the term “input parameter”. This type of parameter is passed using “Pass by Value”.
- When passing an expression, the expression is evaluated first, then the result is passed to the called function.

Passing expression by value



Examples (Pass by Value)

Syntax:

```
// Declaration
void <function_name>(<data_type><var_nm>)

// Calls
<function_name>(<var_nm>);

// Definition
void <function_name>(<data_type><var_nm>)
{
    <function_body>;
    - - - - -;
}
```

Examples (Pass by Value)

Program :

```
/* Program to demonstrate function call by passing value.
Creation Date : 24 Nov 2010 12:08:26 AM
Author : www.technoexam.com [Technowell, Sangli] */

#include <stdio.h>
#include <conio.h>

void printno(int a)
{
    printf("\n Number is : %d", a);
}

void main()
{
    int no;
    void printno(int);
    clrscr();
    printf("\n Enter Number : ");
    scanf("%d", &no);
    printno(no);
    getch();
}
```



Pass by Reference

- Passing by reference is a passing technique that passes the **address of a variable** instead of its value.
 - That's why it is also called Pass by Address
- Pass by reference causes side effect to the actual parameters.
 - When the called function changes a value, it actually changes the original variable in the calling function.
- Only variables can be passed using this technique.
- The formal parameter must be a pointer.

Example:

```
void fun(int *x) // x is a pointer variable
{
    // function body
}
```

- The actual parameter must be an address of a variable.

Example:

```
int n;
fun(&n); // &n means "address of variable n"
```



Pass by Reference

```
/* function definition to swap the values */
void swap(int *x, int *y) {

    int temp;
    temp = *x;    /* save the value at address x */
    *x = *y;     /* put y into x */
    *y = temp;   /* put temp into y */

    return;
}
```



Pass by Reference

```
#include <stdio.h>

int main () {

    /* local variable definition */
    int a = 100;
    int b = 200;

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

    /* calling a function to swap the values */
    swap(&a, &b);

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

    return 0;
}

void swap(int *x, int *y) {

    int temp;

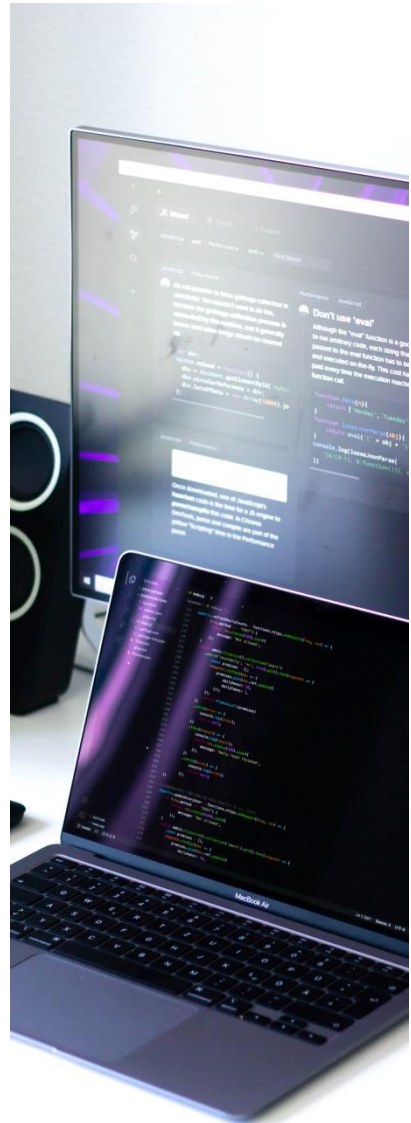
    temp = *x; /* save the value of x */
    *x = *y;   /* put y into x */
    *y = temp; /* put temp into y */

    return;
}
```

Arrays



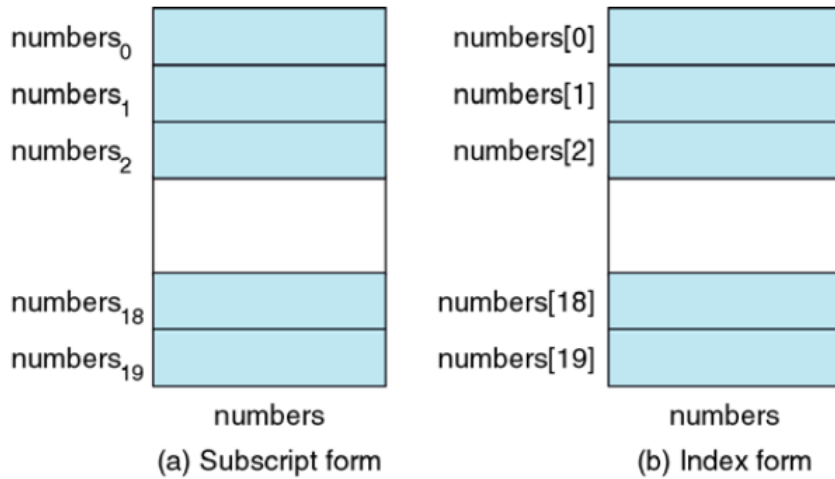
```
components |> filter(is.na(components)) |>
  mutate(
    components = components[is.na(components) == FALSE]
  )
#> # A tibble: 1 x 1
#>   components
#>   <list>
#> 1 <[1] "Car" >
```



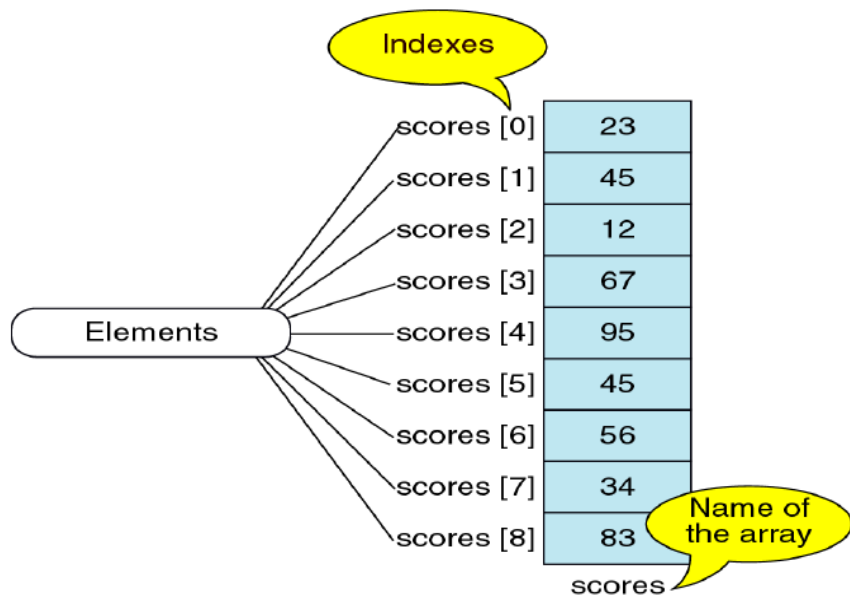
Concept of an array

- An ordinary variable can only contain a single value.
- An array is a **variable that contains a collection of values of the same type**. These values are stored in a sequential location.

Example:



Terms



- A single value in an array is called an **element**.
- An **index** (or a subscript) is a reference of an element
 - It is an integer number
 - Index **0** refers to the first element

Using arrays

- Two things to do when using arrays:
 - Declaration and definition of arrays
 - Accessing elements in arrays
 - for putting values
 - for getting values

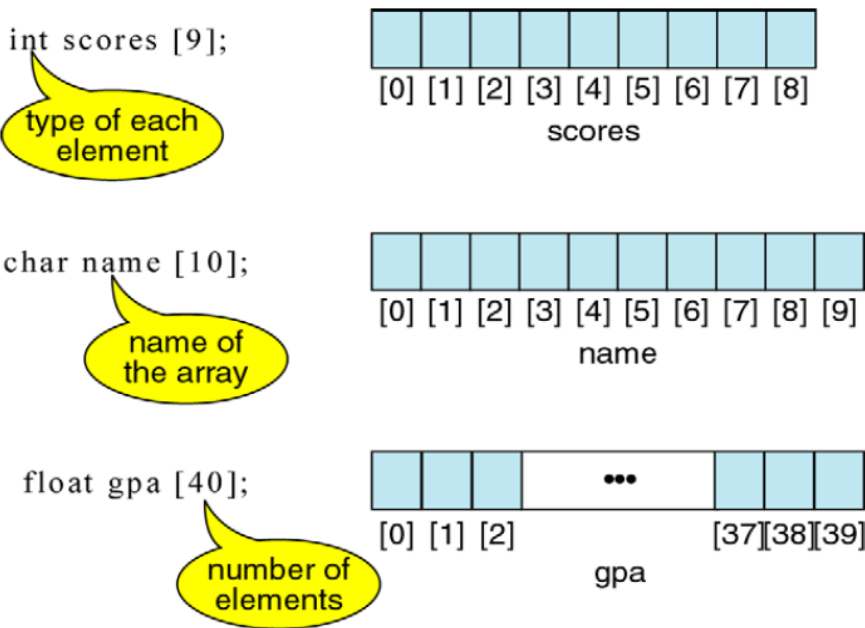
Declaring and defining Arrays

- Since **an array is a variable**, it must be declared and defined before it can be used.
- Declaration and definition tell the compiler:
 - the **name** of the array
 - the **data type** of each element
 - the **number of elements** in the array

Syntax:

```
data_type variable_name[n]; // n = number of elements
```

Examples:



Declaring and defining Arrays

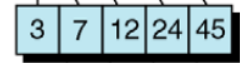
Like ordinary variables, arrays may also be initialized:

```
int numbers [5] = {3,7,12,24,45};
```



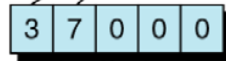
(a) Basic initialization

```
int numbers [ ] = {3,7,12,24,45};
```



(b) Initialization without size

```
int numbers [5] = {3,7};
```



The rest are filled with 0s

(c) Partial initialization

```
int lotsOfNumbers [1000] = n {0};
```



All filled with 0s

(d) Initialization to all zeros

Accessing elements in arrays

- We use an index to access an element from an array.
- The index must be in a valid range
 - The following example would be an error – array A has only 2 elements, but we try to access the third element which is not exist.

```
int A[2];
```

```
A[2] = 100; // this line would an error
```

- We access an element for two purposes:
 - assigning new value into it
 - getting its current value

Assigning values into elements

Examples:

1. *Assigning a new value into the 2nd element of array A.*

```
int A[] = {1,3,5,7};  
A[1] = 100;
```

2. *Incrementing the value of 3rd element of array B.*

```
int B[] = {11,23,35,47};  
B[2]++;
```

3. *Assigning each element of array C with a value that is twice its index*

```
int C[9];  
int i;  
for (i=0; i<9; i++)  
C[i] = i*2;
```

4. *Assigning each element of array D with a value that is read from the keyboard*

```
int D[5];  
int i;  
  
for (i=0; i<5; i++)  
scanf("%d", &D[i]);
```

5. *The following example would be an error – elements of an array must be assigned individually.*

```
int E[4];  
E = {10,20,30,40}; // this would be an error  
  
// solutions: - assign them individually.  
E[0]=10;  
E[1]=20;  
E[2]=30;  
E[3]=40;
```

Getting values from elements

Examples:

1. *Assigning variable n with the value of first element of array A.*

```
int A[] = {1,3,5,7};  
int n;  
n = A[0];
```

2. *Printing the second element of array B*

```
int B[] = {10,30,50,70};  
printf("%d", B[1]);
```

3. *Assigning the first element of array C with the value of the second element,*

```
int C[] = {11,23,35,47};  
C[0] = C[1];
```

4. *Printing all elements of array D*

```
int D[]={1,4,3,6,7,8,9,0,2};  
int i;  
  
for (i=0; i<9; i++)  
    printf("%d\n", D[i] );
```

- Passing an element of an array to a function can be in two forms:

- Pass by value - pass its content:

eg. `printf("%d", A[2]);`

- Pass by reference - pass its address.

eg. `scanf("%d", &A[2]);`

Passing the whole array to a function can only be done by using pass by reference.

- It actually passes the address of the first element.

Example:

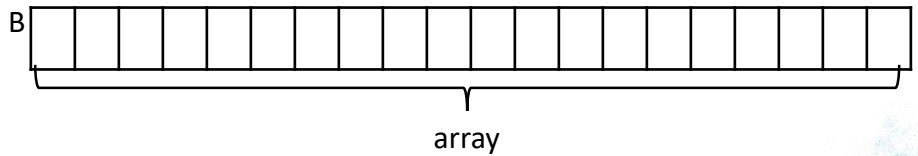
```
void increase(int x[3])
{
    x[0] += 1;
    x[1] += 2;
    x[2] += 3;
}

void main(void)
{
    int A[3]={10,20,30};
    increase(A); // or, increase(&A[0]);
}
```

Summary

Array declaration:

`int B [20];` [] means array
20 means 20 element/ 20 box



`char GRED [10];`

← Array of char only

Reference



access the initial array

```
int B[5];  
B[5]={26,3,6,107,20};
```

```
int B[5]={26,3,6,107,20};
```

| | | | | |
|-----|-----|-----|-----|-----|
| 26 | 3 | 6 | 107 | 20 |
| [0] | [1] | [2] | [3] | [4] |

- C allows a character array to be represented by a character string rather than a list of characters, with the null terminating character automatically added to the end. For example, to store the string "Merkkijono", we would write:

```
char string[] = "Merkkijono";
```

OR

```
char string[] = {'M', 'e', 'r', 'k', 'k', 'i', 'j',  
                'o', 'n', 'o', '\0'};
```

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|
| M | e | r | k | k | i | j | o | n | o | \0 |
|---|---|---|---|---|---|---|---|---|---|----|

String "Merkkijono" stored in memory

To read/print array using looping:

- Read:

```
for (i=0; i<5; i++)  
{  
    scanf("%d ", &A[i]);  
}
```

- Print:

```
for (i=0; i<5; i++)  
{  
    printf("%d ", A[i]);  
}
```



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

```
void increase(int x[3])
{
    x[0] += 1;
    x[1] += 2;
    x[2] += 3;
}
```

Another Function:
void increase(int x[3])

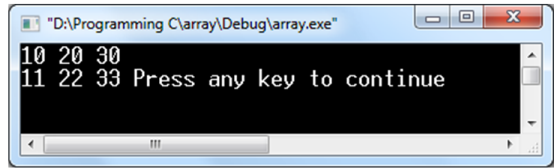
output:

```
void main(void)
{
    int A[3]={10,20,30};
    int i;

    for (i=0; i<3; i++)
    {
        printf("%d ", A[i]);
    }

    printf("\n");
    increase(A); // or, increase(&A[0]);

    for (i=0; i<3; i++)
    {
        printf("%d ", A[i]);
    }
}
```



Array is use looping. If not:
printf("%d ", A[0]);
printf("%d ", A[1]);
printf("%d ", A[2]);



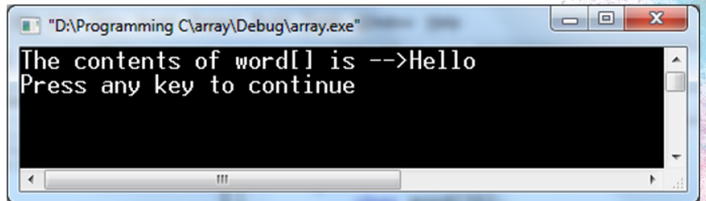
```
#include <stdio.h>
```

```
void main()
{
    char word[20];

    word[0] = 'H';
    word[1] = 'e';
    word[2] = 'l';
    word[3] = 'l';
    word[4] = 'o';
    word[5] = 0;

    printf("The contents of word[] is -->%s\n", word );
}
```

output:





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

```
void main()  
{
```

```
char day1[7] = "Sunday";  
char day2[7] = "Monday";  
char day3[8] = "Tuesday";  
char day4[10] = "Wednesday";  
char day5[] = "Thursday";  
char day6[] = "Friday";  
char day7[] = "Saturday";
```

```
printf("Day 1 is-->%s\n", day1 );  
printf("Day 2 is-->%s\n", day2 );  
printf("Day 3 is-->%s\n", day3 );  
printf("Day 4 is-->%s\n", day4 );  
printf("Day 5 is-->%s\n", day5 );  
printf("Day 6 is-->%s\n", day6 );  
printf("Day 7 is-->%s\n", day7 );
```

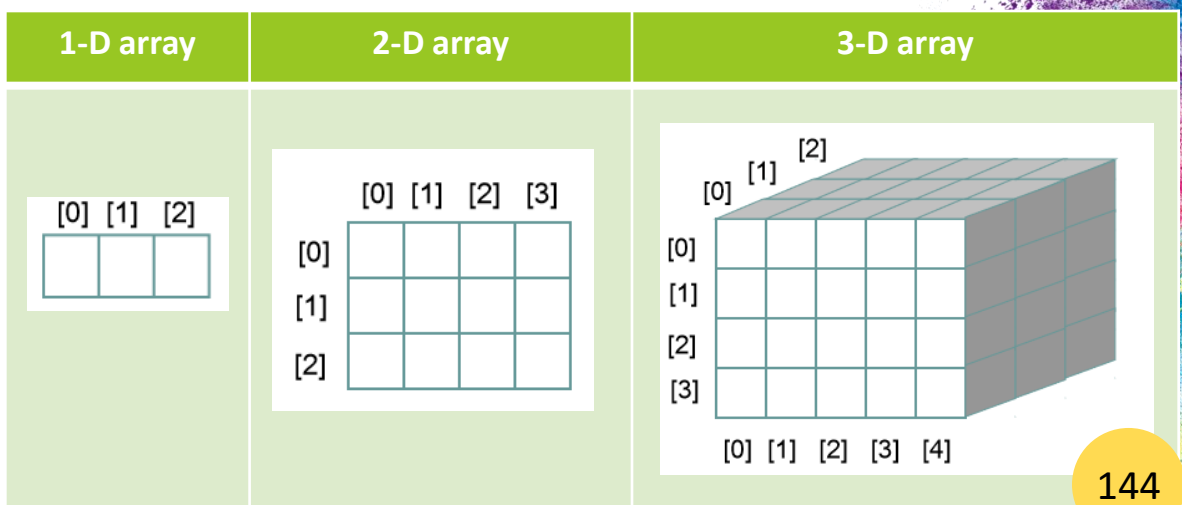
```
}
```

output:

```
"D:\Programming C\array\Debug\array.exe"  
Day 1 is-->Sunday  
Day 2 is-->Monday  
Day 3 is-->Tuesday  
Day 4 is-->Wednesday  
Day 5 is-->Thursday  
Day 6 is-->Friday  
Day 7 is-->Saturday  
Press any key to continue
```

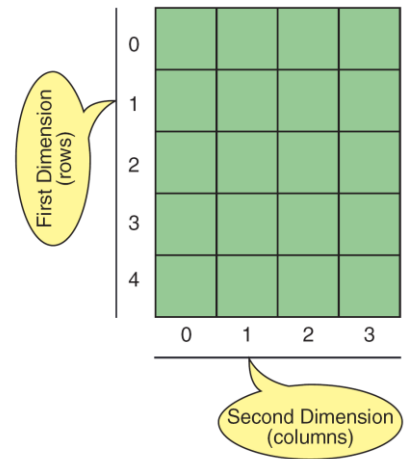
Multi-Dimensional Arrays

- 1-D array can also be extended to Multi-dimensional Array
- Multi-Dimensional array allows us to handle all these using a single identifier
- Multi-Dimensional Arrays:-
 - Two Dimensional ARRAY
 - Three Dimensional ARRAY



Two-Dimensional Arrays

- The two-dimensional array can be defined as an array of arrays.
- The 2D array is organized as matrices which can be represented as the collection of rows and columns.
- The two dimensional (2D) array in C programming is also known as matrix.
- A matrix can be represented as a table of rows and columns.



$$\mathbf{B} = \begin{bmatrix} 51, 52, 53 \\ 54, 55, 56 \end{bmatrix}$$

← Row 1
← Row 2

↑ ↑ ↑
Col 1 Col 2 Col 3

Algebraic notation

Array type Array name Array dimension = 2

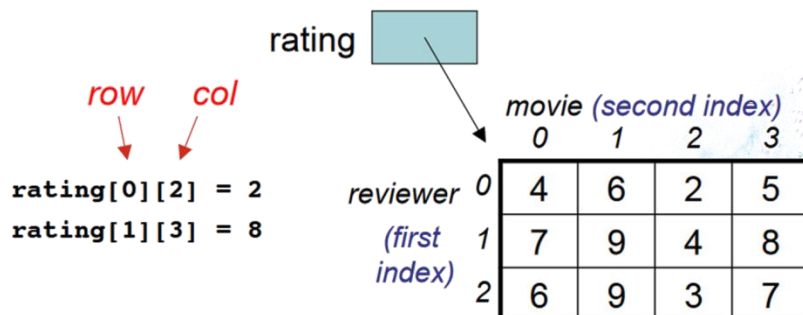
`Int b[2][3] = {(51, 52, 53), (54, 55, 56)};`

Two rows First row second row

Three columns

C notation

- Two-dimensional (2D) arrays are indexed by two subscripts, one for the row and one for the column.
- Example:



INITIALIZATION (2D array)

- Initialized directly in the declaration statement
 - `int b[2][3] = {51, 52, 53, 54, 55, 56};`

| | |
|---------------------------|---------------------------|
| <code>b[0][0] = 51</code> | <code>b[1][0] = 54</code> |
| <code>b[0][1] = 52</code> | <code>b[1][1] = 55</code> |
| <code>b[0][2] = 53</code> | <code>b[1][2] = 55</code> |

- Use braces to separate rows in 2-D arrays.
 - `int c[4][3] = {{1, 2, 3},
 {4, 5, 6},
 {7, 8, 9},
 {10, 11, 12}};`
 - `int c[][3] = {{1, 2, 3},
 {4, 5, 6},
 {7, 8, 9},
 {10, 11, 12}};`
- Implicitly declares the number of rows to be 4.
- Data may be input into two-dimensional arrays using nested *for* loops interactively or with data files.
- A nested *for* loop is used to input elements in a two dimensional array.
- In this way by increasing the index value of the array the elements can be entered in a 2d array.

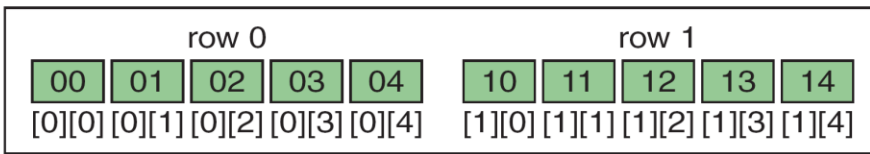
PROGRAM:

Two-dimensional Array

```
1  /* ===== fillArray =====
2  This function fills array such that each array element
3  contains a number that, when viewed as a two-digit
4  integer, the first digit is the row number and the
5  second digit is the column number.
6  Pre table is array in memory
7  numRows is number of rows in array
8  Post array has been initialized
9  */
10 void fillArray (int table[][MAX_COLS], int numRows)
11 {
12 // Statements
13   for (int row = 0; row < numRows; row++)
14     {
15       table [row][0] = row * 10;
16       for (int col = 1; col < MAX_COLS; col++)
17         table [row][col] = table [row][col - 1] + 1;
18     } // for
19   return;
20 } // fillArray
```

| | | | | |
|----|----|----|----|----|
| 00 | 01 | 02 | 03 | 04 |
| 10 | 11 | 12 | 13 | 14 |

User's View



Memory View

FIGURE: Memory Layout

PROGRAM: Convert Table to One-dimensional Array

```

1  /* This program changes a two-dimensional array to the
2     corresponding one-dimensional array.
3     Written by:
4     Date:
5  */
6  #include <stdio.h>
7  #define ROWS 2
8  #define COLS 5
9
10 int main (void)
11 {
12     // Local Declarations
13     int table [ROWS] [COLS] =
14         {
15             {00, 01, 02, 03, 04},
16             {10, 11, 12, 13, 14}
17         }; // table
18     int line [ROWS * COLS];
19
20     // Statements
21     for (int row = 0; row < ROWS; row++)
22         for (int column = 0; column < COLS; column++)
23             line[row * COLS + column] = table[row][column];
24
25     for (int row = 0; row < ROWS * COLS; row++)
26         printf(" %02d ", line[row]);
27
28     return 0;
29 } // main

```

Results:

00 01 02 03 04 10 11 12 13 14

```

#define MAX_ROWS 5
#define MAX_COLS 4
// Function Declarations
void print_square (int []);
int main (void)
{
    int table [MAX_ROWS][MAX_COLS] =
        {
            { 0, 1, 2, 3 },
            { 10, 11, 12, 13 },
            { 20, 21, 22, 23 },
            { 30, 31, 32, 33 },
            { 40, 41, 42, 43 }
        }; /* table */

    ...
    for (int row = 0; row < MAX_ROWS; row++)
        print_square (table [row]);

    ...
    return 0;
} // main

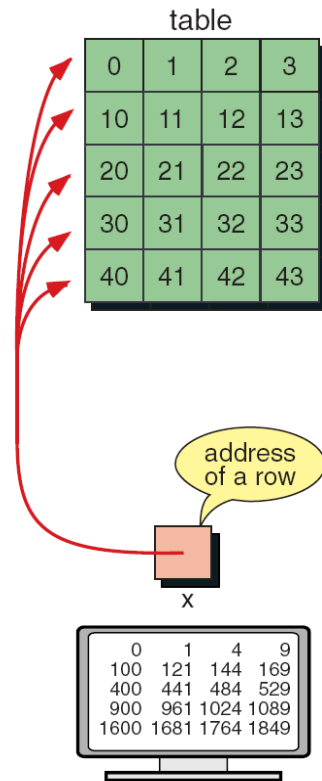
```

```

void print_square (int x[])
{
    for (int col = 0; col < MAX_COLS; col++)
        printf("%6d", x[col] * x[col]);
    printf ("\n");
    return;
} // print_square

```

FIGURE: Passing a Row



```

#define MAX_ROWS 5
#define MAX_COLS 4
// Function Declarations
void print_square (int []);
int main (void)
{
    int table [MAX_ROWS][MAX_COLS] =
        {
            { 0, 1, 2, 3 },
            { 10, 11, 12, 13 },
            { 20, 21, 22, 23 },
            { 30, 31, 32, 33 },
            { 40, 41, 42, 43 }
        }; /* table */

    ...
    for (int row = 0; row < MAX_ROWS; row++)
        print_square (table [row]);

    ...
    return 0;
} // main

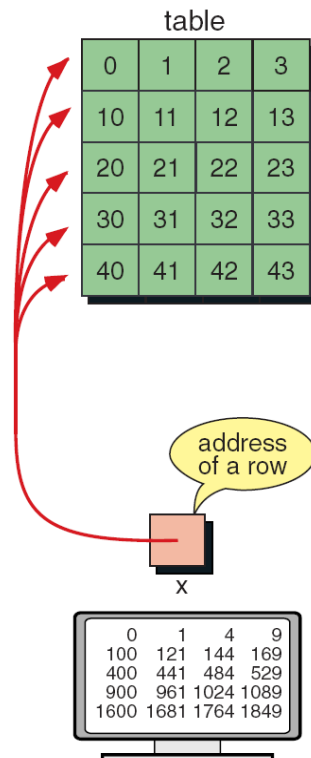
```

```

void print_square (int x[])
{
    for (int col = 0; col < MAX_COLS; col++)
        printf("%6d", x[col] * x[col]);
    printf ("\n");
    return;
} // print_square

```

FIGURE: Calculate Average of Integers in Array



| | | | | | |
|----|----|----|----|----|---|
| 0 | 1 | 1 | 1 | 1 | 1 |
| -1 | 0 | 1 | 1 | 1 | 1 |
| -1 | -1 | 0 | 1 | 1 | 1 |
| -1 | -1 | -1 | 0 | 1 | 1 |
| -1 | -1 | -1 | -1 | 0 | 1 |
| -1 | -1 | -1 | -1 | -1 | 0 |

FIGURE: Example of Filled Matrix

PROGRAM: Fill Matrix

```

1  /* This program fills the diagonal of a matrix (square
2     array) with 0, the lower left triangle with -1, and
3     the upper right triangle with 1.
4     Written by:
5     Date:
6  */
7  #include <stdio.h>
8
9  int main (void)
10 {
11 // Local Declarations
12     int table [6][6];
13
14 // Statements
15     for (int row = 0; row < 6; row++)
16         for (int column = 0; column < 6; column++)
17             if (row == column)
18                 table [row][column] = 0;
19             else if (row > column)
20                 table [row][column] = -1;
21             else
22                 table [row][column] = 1;
23
24     for (int row = 0; row < 6; row++)
25     {
26         for (int column = 0; column < 6; column++)
27             printf("%3d", table[row][column]);
28         printf("\n");
29     } // for row
30     return 0;
31 } // main

```

Three Dimensional (3D) ARRAY in C Language

- A 3D array is essentially an array of arrays of arrays: it's an array or collection of 2D arrays, and a 2D array is an array of 1D array.

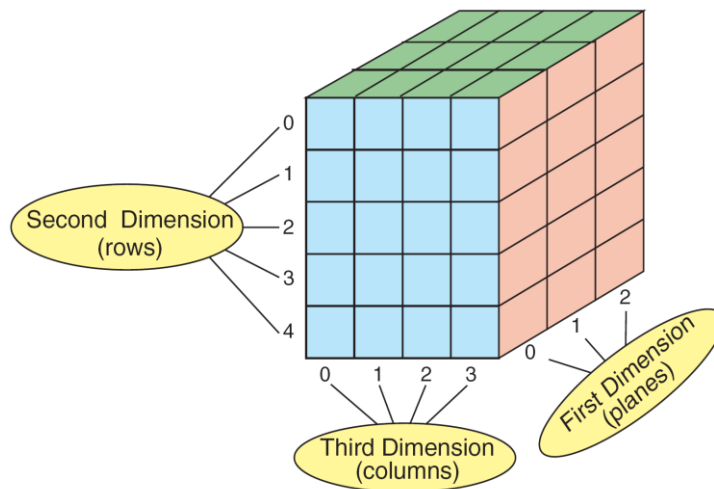
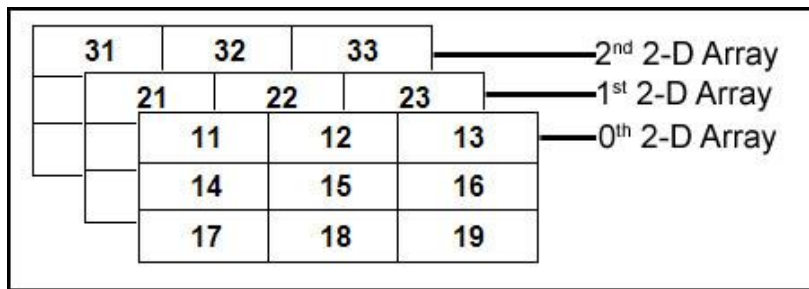


FIGURE: A Three-dimensional Array (3 x 5 x 4)

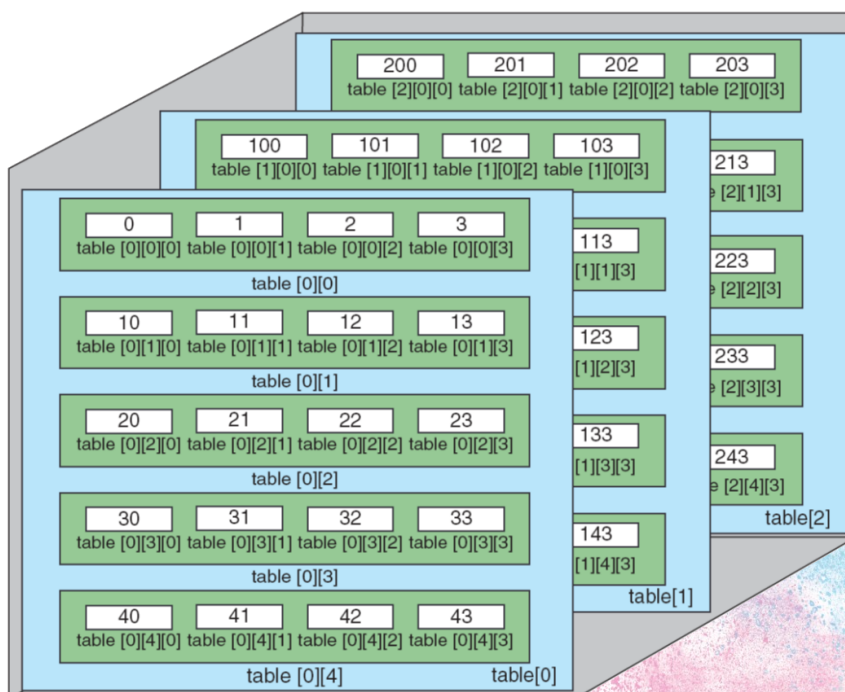


FIGURE: C View of Three-dimensional Array

Declaration and Initialization 3D Array



```
1  #include<stdio.h>
2  #include<conio.h>
3
4  void main()
5  {
6  int i, j, k;
7  int arr[3][3][3]=
8      {
9      {
10         {11, 12, 13},
11         {14, 15, 16},
12         {17, 18, 19}
13     },
14     {
15         {21, 22, 23},
16         {24, 25, 26},
17         {27, 28, 29}
18     },
19     {
20         {31, 32, 33},
21         {34, 35, 36},
22         {37, 38, 39}
23     },
24 };
25 clrscr();
26 printf(":::3D Array Elements:::\n\n");
27 for(i=0;i<3;i++)
28 {
29     for(j=0;j<3;j++)
30     {
31         for(k=0;k<3;k++)
32         {
33             printf("%d\t",arr[i][j][k]);
34         }
35         printf("\n");
36     }
37     printf("\n");
38 }
39 getch();
40 }
```

A screenshot of the Turbo C++ IDE. The title bar reads "Turbo C++ IDE". The main window shows the output of the program, which is a 3x3x3 grid of numbers. The output is as follows:

```
:::3D Array Elements:::
11      12      13
14      15      16
17      18      19

21      22      23
24      25      26
27      28      29

31      32      33
34      35      36
37      38      39
```

Array using a Loop



```
1  #include<stdio.h>
2  #include<conio.h>
3
4  void main()
5  {
6  int i, j, k, x=1;
7  int arr[3][3][3];
8  clrscr();
9  printf(":::3D Array Elements:::\n\n");
10
11 for(i=0;i<3;i++)
12 {
13     for(j=0;j<3;j++)
14     {
15         for(k=0;k<3;k++)
16         {
17             arr[i][j][k] = x;
18             printf("%d\t",arr[i][j][k]);
19             x++;
20         }
21         printf("\n");
22     }
23     printf("\n");
24 }
25 getch();
26 }
```

Terbitan



e ISBN 978-967-2240-27-3

