

An Introductory Lecture on

CSCMJ101: Introduction to Programming using C

Dr. Vandana Kushwaha

Department of Computer Science
Institute of Science, BHU, Varanasi

Course Objectives

CSCMJ101	Introduction to Programming using C	Theory	Practical	Cumulative
Major/Minor		3	1	4

- The course aims to provide exposure to problem-solving through programming.
- It aims to train the student to the basic concepts of the C-programming language.
- Learn to develop simple algorithms and flow charts to solve a problem.
- Learn to develop logics which will help to create programs, applications in C.
- By learning the basic programming constructs you can easily switch over to any other programming language in future.
- This course involves a lab component which is designed to give the student hands-on experience with the concepts.

Syllabus

- **UNIT I**
- Introduction to Programming, Program Concept, Characteristics of Programming, Stages in Program Development, Algorithms, Flowcharts, Types of Programming Methodologies, Introduction to C Programming Basic Program Structure in C, Variables and Assignments, Input and Output, Selection and Repetition Statements.
- **UNIT II**
- User defined functions, Standard library functions, Passing values between functions, Calling convention: Call by value and Call by reference, Recursive functions. Introduction to Arrays, Declaration and Referring Arrays, Arrays in Memory, Initializing Arrays. Arrays in Functions, Multi-Dimensional Arrays.

Syllabus

- **UNIT III**
 - Introduction to String, Declaration and Initialization, Reading and Writing Strings, Arrays of Strings, String and Function, Standard String Library Functions. Basics of Pointers, Pointers and One-dimensional Arrays, Pointer Arithmetic, Pointers as Function Arguments, Pointers and Strings, Pointers and two-dimensional arrays, Arrays of Pointers, Storage Classes.
- **UNIT IV**
 - Basics of Structures, Structures and Functions, Arrays of Structures, Pointers to Structures, Unions, Preprocessor, File Inclusion, Macro, Conditional Compilation, Dynamic Memory Allocation, Command Line Arguments, File Handling.

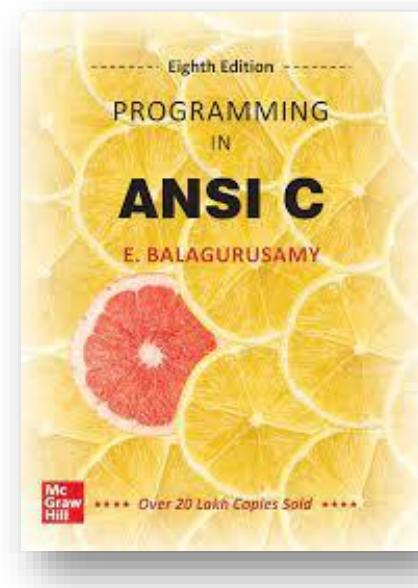
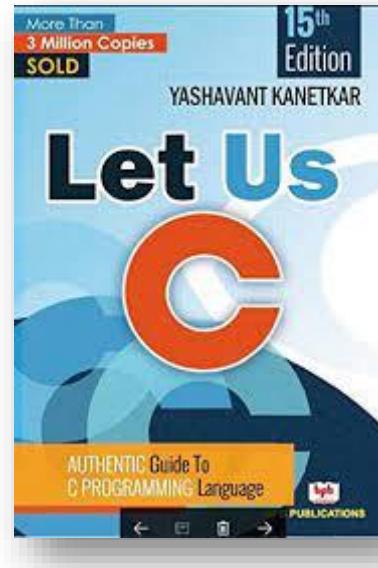
Learning Outcomes

- Apply problem-solving knowledge and skills to write small, well-documented, effective C programs.
- Choose the right data representation formats based on the requirements of the problem.
- Able to write simple programs in the corresponding programming language to solve a task, given the constraints on the inputs.
- Able to manually trace through a program to identify logical errors;
- Able to differentiate between logical errors, syntax errors, and run-time errors.
- Able to write code following good programming style (clear comments, naming convention, indentation, etc.)

Suggested Readings

BOOKS

- B. W. Kernighan and D. M. Ritchie, *the C Programming Language*, PHI.
- Y. Kanetkar, *Let Us C*, BPB Publications.
- E. Balagurusamy, *Programming in ANSI C*, McGraw-Hill.



C Programming Lab

- Programming Exercises using C Programming.
- **Software Required(PC)**



Dev-C++ 6.3
4 ★★★★☆ (13692 votes)



Code::Blocks 20.03
3.6 ★★★★☆ (1772 votes)

- **Apps(Smartphone)**
 - C4droid - C/C++ compiler & IDE
 - CppDroid – C/C++ IDE
- **Online C Compilers**

Class Schedule

Day	Paper	Time	Venue
Monday	<i>Practical</i> <i>(Major Students)</i>	11 a.m. to 1 p.m.	Lab at Computer Science Department
Tuesday	<i>Practical</i> <i>(Minor Students)</i>	11 a.m. to 1 p.m.	Lab at Computer Science Department
Wednesday	<i>Theory</i>	2 p.m. to 4 p.m.	GH-2 at NLTC
Thursday	<i>Theory</i>	2 p.m. to 3 p.m.	GH-2 at NLTC

Lecture 1

Introduction to Programming

CSCMJ101: Introduction to Programming using C

Dr. Vandana Kushwaha

Department of Computer Science
Institute of Science, BHU, Varanasi

What is a Program?

- A **program** is a **set of instructions** written in a **specific programming language** that a **computer** or **digital device** can **execute** to perform a **particular task** or **solve a problem**.
- These **instructions** tell the **computer** :
 - **What operations** to perform,
 - **How** to **process data**, and
 - **What outputs** to produce.

Key Components of a Program

- **Instructions:** The commands or steps that the computer follows, written in a **programming language** (e.g., Python, Java, C++).
- **Input:** Data that the **program receives** from the **user** or another system.
- **Processing:** The **actions** the **program takes** to **manipulate** or **work** with the **input** (such as *calculations, logic, or data handling*).
- **Output:** The **result** the **program produces**, which could be **displayed** to the **user**, **saved** to a **file**, or **sent** to another system.
- **Control Flow:** The **order** in which **instructions are executed**, which can be influenced by **conditional statements** (like *if and else*), loops, or functions.

What is Programming?

- **Programming** is the process of **writing instructions**, known as **program/code**, that a computer or other digital device follows to perform specific tasks or solve problems.
- These **instructions** are **written** in a **programming language** that the computer can understand and execute.
- **Programming** enables **computers** to:
 - **Process data**,
 - **Automate tasks**, and
 - **Create software applications, websites, games**, and more.

Problem Solving using Computer

- Computers are used for solving various day-to-day problems and thus problem solving is an essential skill that a Computer science student should know.
- Computers themselves cannot solve a problem.
- Precise step-by-step instructions should be given by us to solve the problem.
- Thus, the success of a computer in solving a problem depends on how correctly and precisely we define the problem, design a solution (algorithm) and implement the solution (program) using a programming language.
- Thus, problem solving is the process of:
 - Identifying a problem,
 - Developing an algorithm for the identified problem and finally;
 - Implementing the algorithm to develop a computer program.

Stages in Program Development

1. Problem Definition

2. Problem Analysis

3. Algorithm Design

4. Coding

5. Testing and Debugging

6. Maintenance

1. Problem Definition

- First we try to understand the **problem** to be solved in totally.
- Before with the next stage or step, we should be **absolutely sure** about the **objectives** of the given problem.
- In order to **solve** the **problem**, it is **very necessary** to **define** the **problem** to get its proper understanding.
- For **example**, suppose we are asked to write a code for “ **Compute the average of three numbers**”.

2. Problem Analysis

- **Problem analysis** helps to **formulate the solution** of the problem in a better way.
- By analysing a problem, we would be able to figure out what are the **inputs** that our program should accept and the **outputs** that it should produce.
- Once a problem has been defined, the problem's specifications are then listed.
- **Problem specifications** should definitely include :
 - **INPUT** : What is the input set of the program?
 - **OUTPUT** : What is the desired output of the program and in what form the output is desired?
 - **Format and Constraints** on input/output.

3. Algorithm Design

- It is essential to **device a solution** before writing a **program code** for a given problem.
- The **solution** is represented in **natural language** and is called an **algorithm**.
- We can imagine an **algorithm** like a **very well-written recipe** for a dish, with **clearly defined steps** that, if followed, one will end up preparing the dish.
- We start with a tentative solution plan and keep on **refining the algorithm** until the **algorithm** is able to capture all the aspects of the desired solution.
- For a given problem, more than one **algorithm** is possible and we have to select the most suitable solution.

4. Coding

- After finalising the **algorithm**, we need to **convert the algorithm** into the format which can be **understood by the computer** to generate the desired solution.
- Different **high level programming languages** like **C, C++ or JAVA** can be used for writing a **program**.
- It is equally important to **record** the details of the **coding procedures** followed and document the solution.
- This is helpful when revisiting the programs at a later stage.
- This process is referred as '**documentation**'.

5. Testing and Debugging

- The **program** created should be **tested** on **various parameters**.
- The **program** should **meet the requirements** of the **user**.
- It **must respond** within the **expected time**.
- It should **generate correct output** for all possible **inputs**.
- In the presence of **syntactical errors**, **no output** will be obtained.
- In case the **output generated is incorrect**, then the **program** should be checked for **logical errors**, if any.
- The **errors or defects** found in the **testing phases** are **debugged** or **rectified** and the **program** is again **tested**.
- This **continues till** all the **errors are removed** from the **program**.

6. Maintenance

- Once the **software application** has been **developed**, **tested** and **delivered** to the user, still **problems** in terms of **functioning** can come up and **need to be resolved** from **time to time**.
- The **maintenance** of a **program** involves the following :
 - **Detection and Elimination of undetected errors** in the existing program.
 - **Modification of current program to enhance its performance and adaptability.**
 - **Enhancement of user interface.**
 - **Enriching the program with new capabilities.**

Lecture 2

Algorithm and Flow Chart

CSCMJ101: Problem Solving through C

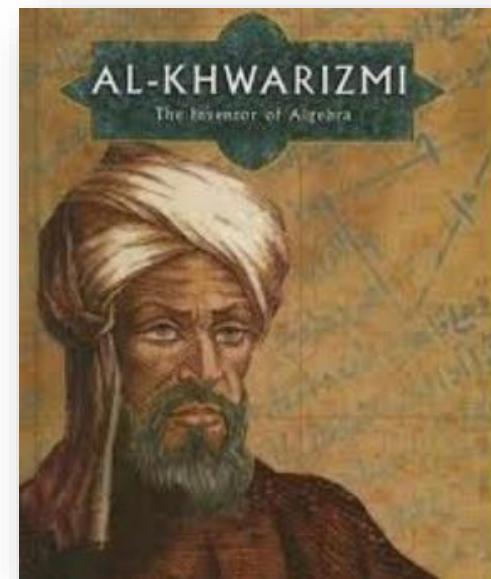
Programming

Dr. Vandana Kushwaha

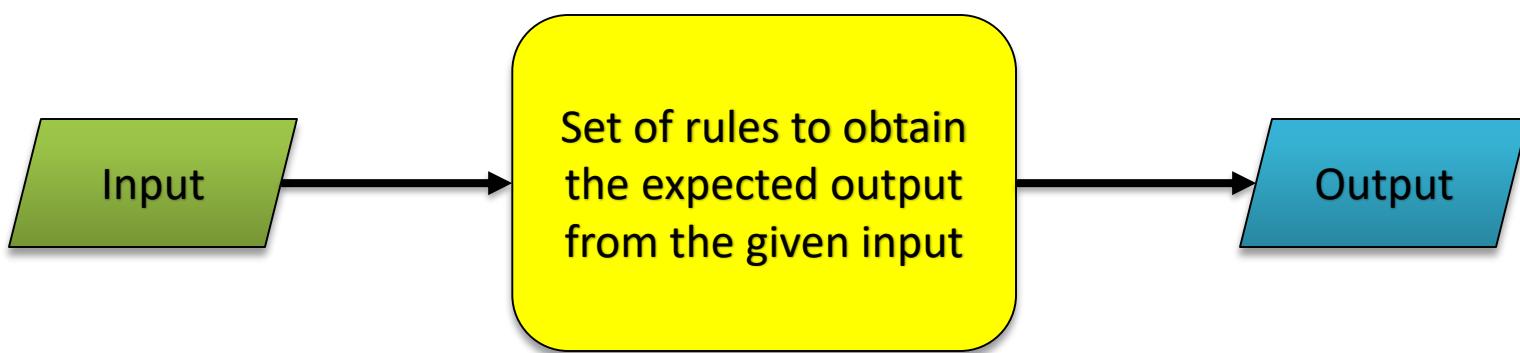
Department of Computer Science
Institute of Science, BHU, Varanasi

Algorithm

- The origin of the term **Algorithm** is traced to **Persian astronomer and mathematician, Abu Abdullah Muhammad ibn Musa Al-Khwarizmi** (850 AD) as the Latin translation of **AlKhwarizmi** was called '**Algorithmi**'.
- An **Algorithm** is a **precise sequence of instructions for solving a problem**.
- The **Algorithm** designed are **language-independent**, i.e. they are **just plain instructions** that can be **implemented** in **any language**, and yet the **output** will be the **same**, as expected.



What is an Algorithm?



Algorithm

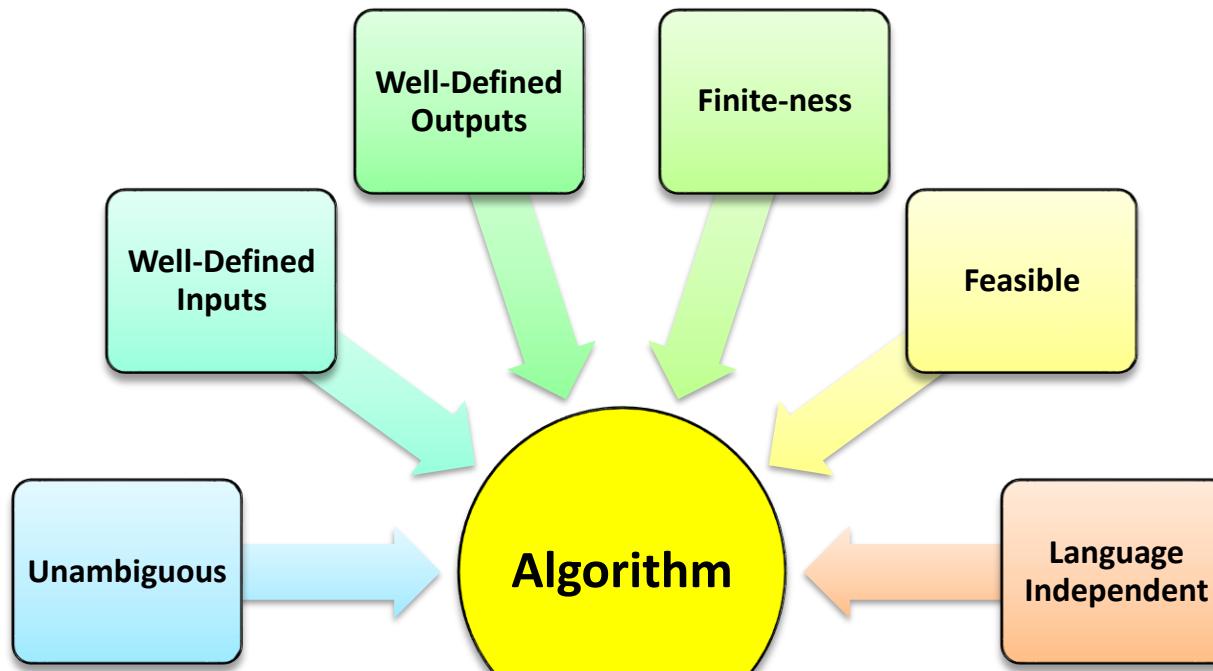
Dr. Vandana Kushwaha, Dept. of Computer
Science, BHU

Characteristics of an Algorithm

- **Clear and Unambiguous:** Algorithm should be **clear** and **unambiguous**. Each of its **steps** should be **clear in all aspects** and must lead to **only one meaning**.
- **Well-Defined Inputs:** If an **algorithm** says to take **inputs**, it should be **well-defined inputs**.
- **Well-Defined Outputs:** The **algorithm** must **clearly define** what **output** will be yielded and it should be **well-defined** as well.
- **Finite-ness:** The **algorithm** must be **finite**, i.e. it **should not** end up in an **infinite loops**.
- **Feasible:** The **algorithm** must be simple, generic and practical, such that it can be **executed upon the available resources**.

Characteristics of an Algorithm

- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just **plain instructions** that can be **implemented in any language**, and yet the **output will be same, as expected.**



Algorithm to print sum and average

- **Step 1: Start.**
- **Step 2 :Read the three number** suppose "a","b","c" form the user.
- **Step 3:** Declare the variables "**sum**" and "**avg**".
- **Step 4 : $\text{sum}=\text{a}+\text{b}+\text{c};$**
- **Step 5: $\text{avg}=\text{sum}/3.$**
- **Step 6:Display "sum " and "avg".**
- **Step 7: End .**

Representation of Algorithms

- There are **two common methods** of representing an **algorithm** :
 - **Flowchart**
 - **Pseudocode.**
- Either of the methods can be used to represent **an algorithm** while keeping in mind the following:
 - it **showcases the logic of the problem solution, excluding any implementation details.**
 - **it clearly reveals the flow of control during execution of the program.**

Flowchart – Visual Representation of Algorithms

- A **flowchart** is a **visual representation** of an **algorithm**.
- A **flowchart** is a **diagram** made up of **boxes, diamonds and other shapes**, connected by **arrows**.
- Each **shape** represents a **step** of the **solution** process and the **arrow** represents the **order or link** among the steps.
- There are **standardised symbols** to draw **flowcharts**.

Shapes or symbols to draw flow charts

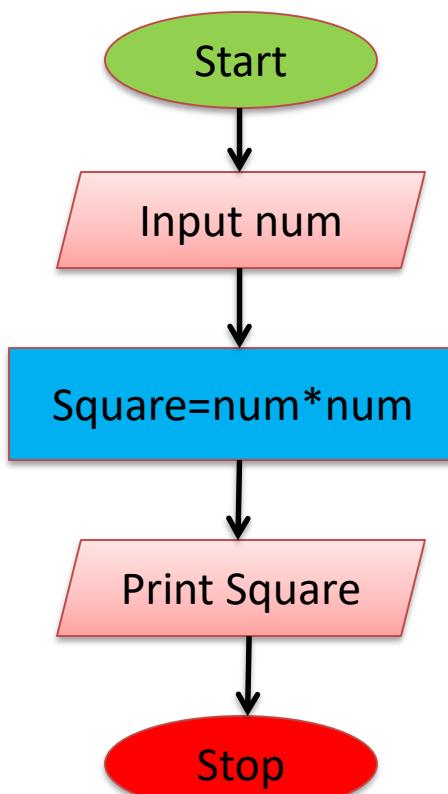
Flowchart symbol	Function	Description
	Start/End	Also called “Terminator” symbol. It indicates where the flow starts and ends.
	Process	Also called “Action Symbol,” it represents a process, action, or a single step.
	Decision	A decision or branching point, usually a yes/no or true/ false question is asked, and based on the answer, the path gets split into two branches.
	Input/Output	Also called data symbol, this parallelogram shape is used to input or output data
	Arrow	Connector to show order of flow between shapes.

Example 1

- ***Write an algorithm to find the square of a number.***
- Before developing the algorithm, let us first identify the input, process and output:
 - **Input:** Number whose square is required
 - **Process:** Multiply the number by itself to get its square
 - **Output:** Square of the number
- **Algorithm** to find square of a number.
- **Step 1:** Input a number and store it to num
- **Step 2:** Compute num * num and store it in square
- **Step 3:** Print square

Example 1

- The algorithm to find square of a number can be represented pictorially using flowchart as shown in Figure below:



Pseudocode

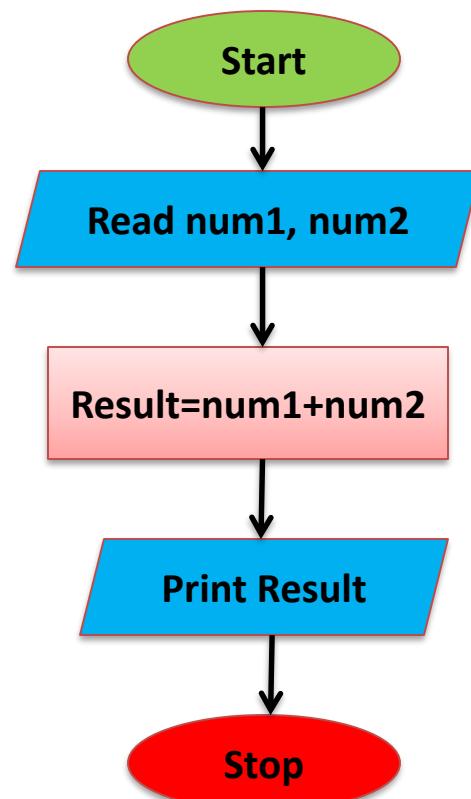
- A **pseudocode (pronounced Soo-doh-kohd)** is another way of representing an **algorithm**.
- It is considered as a **non-formal language** that helps **programmers** to write algorithm.
- It is a **detailed description** of **instructions** that a computer must follow in a particular order.
- It is intended for **human reading** and **cannot be executed** directly by the computer.
- No **specific standard** for writing a **pseudocode** exists.
- The word “**pseudo**” means “**not real**,” so “**pseudocode**” means “**not real code**”.

Pseudocode

- Following are some of the frequently used **keywords** while writing **pseudocode**:
 - INPUT
 - COMPUTE
 - PRINT
 - INCREMENT
 - DECREMENT
 - IF/ELSE
 - WHILE
 - TRUE/FALSE

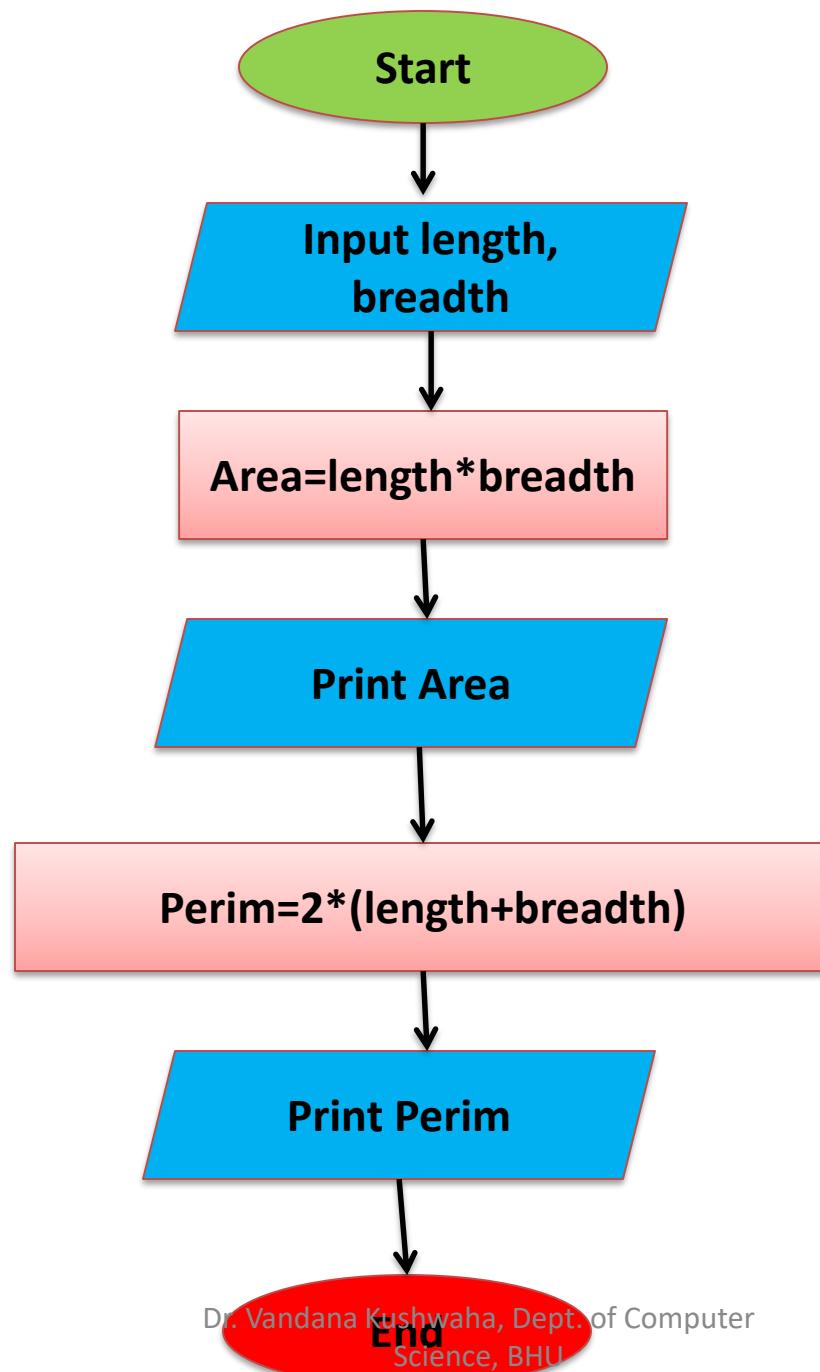
Example 2

- Write an algorithm to display the sum of two numbers entered by user, using both pseudocode and flowchart.
- Pseudocode for the sum of two numbers will be:
 1. INPUT num1
 2. INPUT num2
 3. COMPUTE Result = num1 + num2
 4. PRINT Result



Example 3

- *Write an algorithm to calculate area and perimeter of a rectangle, using both pseudocode and flowchart.*
- **Pseudocode** for calculating area and perimeter of a rectangle.
 1. **INPUT** length
 2. **INPUT** breadth
 3. **COMPUTE** Area = length * breadth
 4. **PRINT** Area
 5. **COMPUTE** Perim = 2 * (length + breadth)
 6. **PRINT** Perim



Benefits of Pseudocode

- Before writing codes in a **high level language**, a **pseudocode** of a **program** helps in representing the **basic functionality** of the intended **program**.
- By writing the **code first** in a **human readable language**, the **programmer** safeguards against leaving out any **important step**.
- Besides, for **non-programmers**, actual programs are **difficult** to read and understand.
- But **pseudocode** helps them to **review the steps to confirm that the proposed implementation** is going to achieve the **desire output**.

Flow of Control

- The **flow of control** depicts the **flow of events** as represented in the flow chart.
- The events can flow in a **sequence**, **selection**(decision) and **repetition**(loop).

1. Sequence Control

- The statements are executed **one after another**, i.e., in a **sequence**.
- Such algorithms where all the steps are executed one after the other are said to execute in sequence.
- However, statements in an **algorithm** may not always execute in a sequence.
- We may sometimes require the **algorithm** to either do some routine tasks in a **repeated manner** or behave differently depending on the **outcomes** of previous steps.
- Example 1, example 2 and example 3 are the cases of **sequence control**.

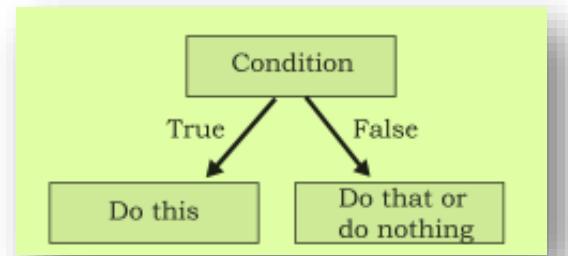
Flow of Control

2. Selection Control

- The problem which involves some **decision-making** based on certain **conditions**.
For example:
- **Checking eligibility for voting.**
- *Depending on their age, a person will either be allowed to vote or not allowed to vote:*
 - If age is greater than or equal to 18, the person is eligible to vote
 - If age is less than 18, the person is not eligible to vote
- *In these examples, any one of the alternatives is selected based on the outcome of a condition.*
- The program checks one or more conditions and perform operations (sequence of actions) depending on true or false value of the condition.
- *These true or false values are called binary values.*

Flow of Control

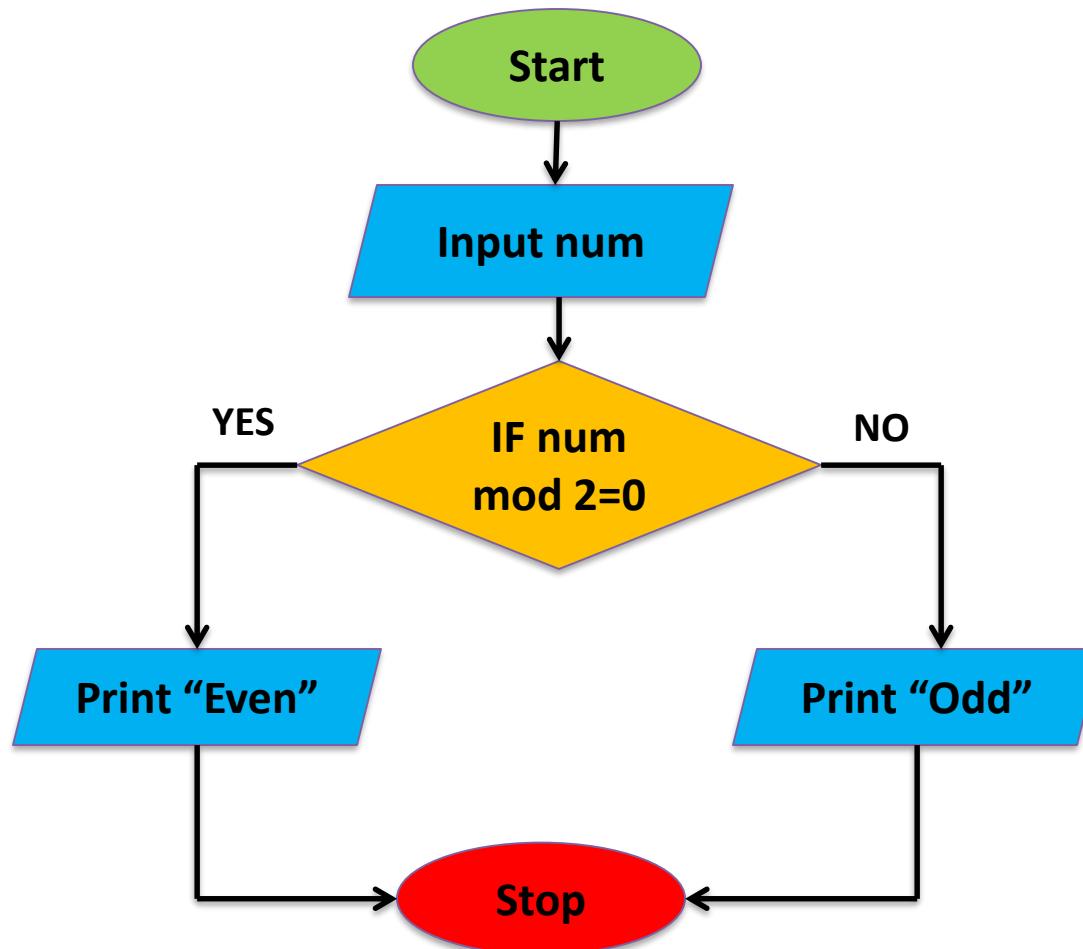
- Conditions are written in the algorithm as follows:
 - If <condition> is true then
 - steps to be taken when the condition is true/fulfilled
- There are situations where we also need to take action when the condition is not fulfilled .
- To represent that, we can write:
 - If <condition> is true then
 - steps to be taken when the condition is true/fulfilled
 - otherwise
 - steps to be taken when the condition is false/not fulfilled
- In programming languages, 'otherwise' is represented using Else keyword.



Example 4

- Write an algorithm to check whether a number is odd or even.
 - **Input:** Any number
 - **Process:** Check whether the number is even or not
 - **Output:** Message “Even” or “Odd”
- Pseudocode of the algorithm can be written as follows:
 1. PRINT "Enter the Number"
 2. INPUT number
 3. IF number MOD 2 = 0 THEN
 4. PRINT "Number is Even"
 5. ELSE
 6. PRINT "Number is Odd"

Example 4

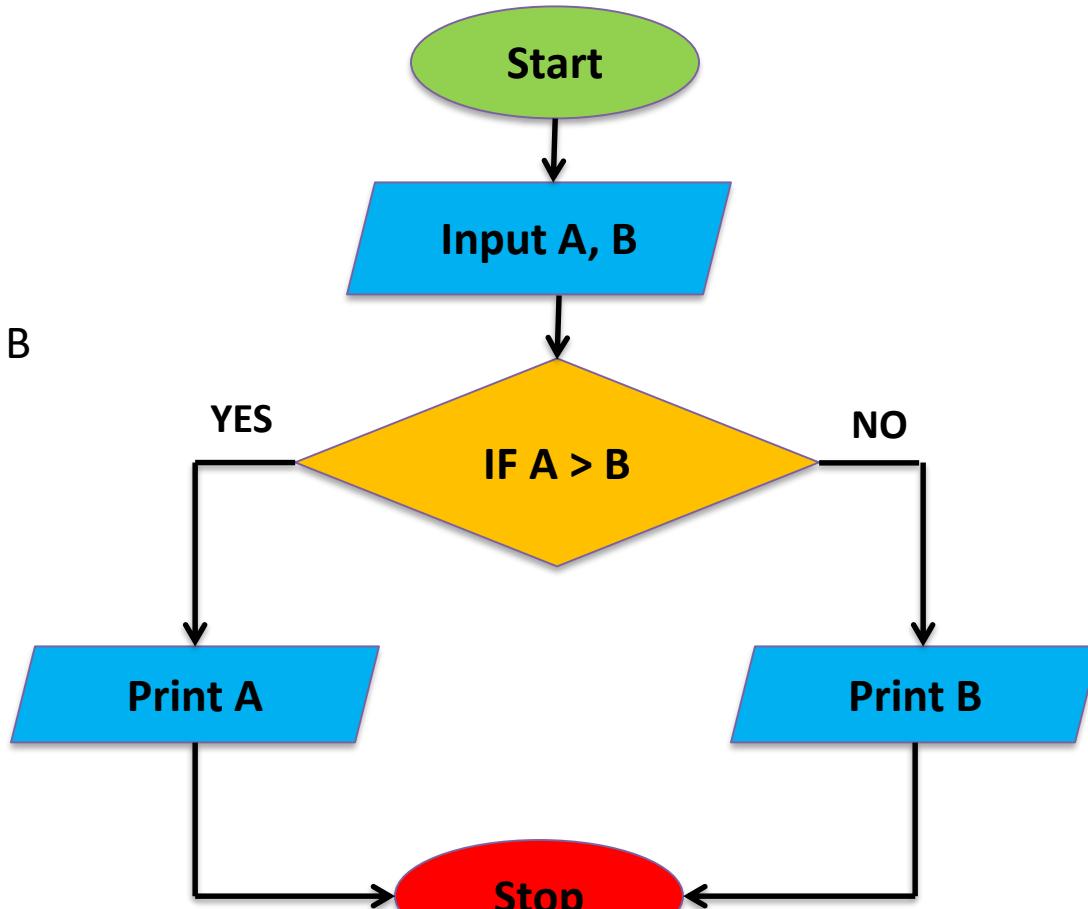


Example 5

- Algorithm & Flowchart to find the largest of two numbers.

Algorithm

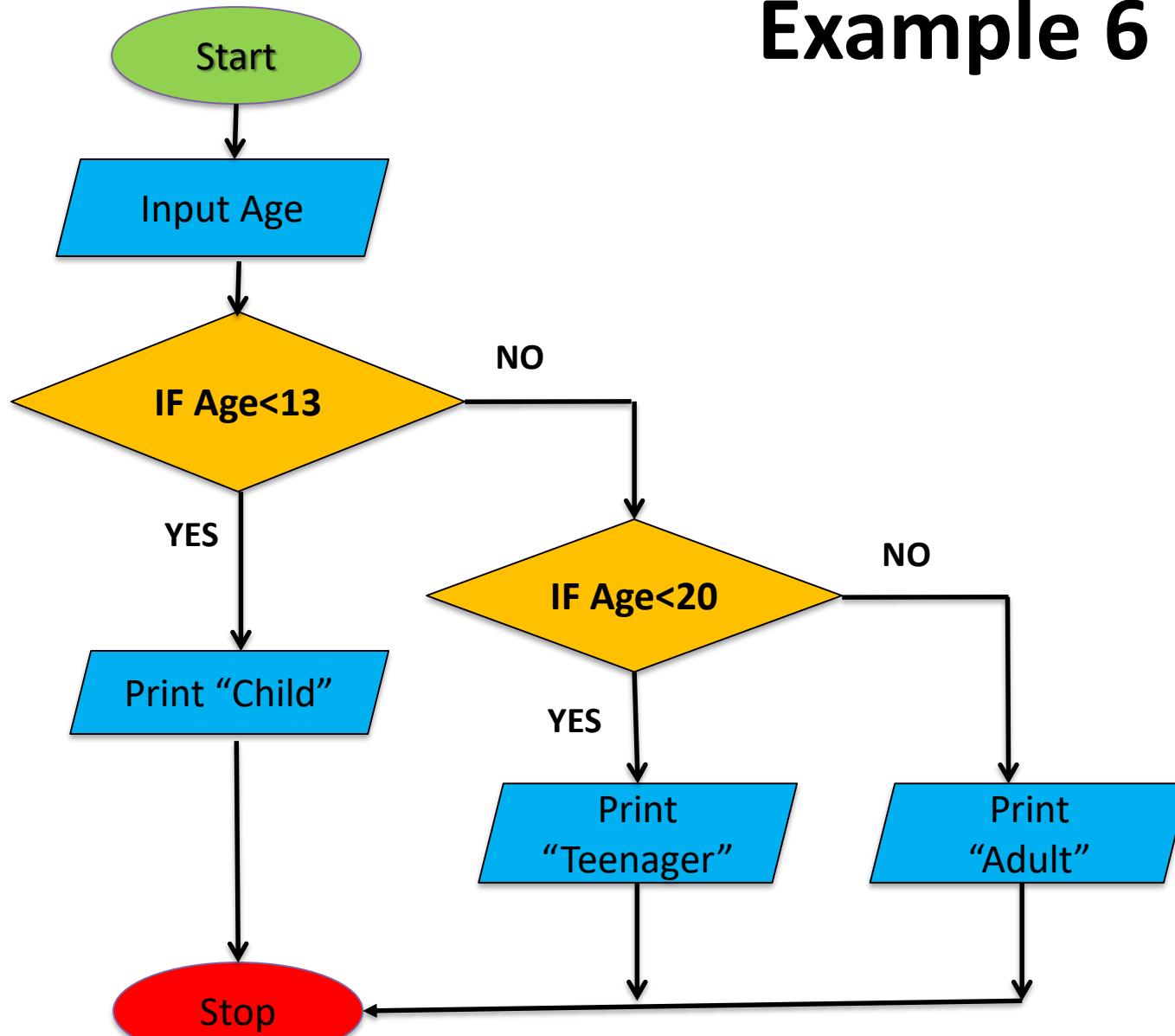
1. Start
2. Input two numbers say A, B
3. IF A > B THEN
4. print largest is A
5. ELSE
6. print largest is B
7. ENDIF
8. Stop



Example 6

- Let us write a **pseudocode** and draw a flowchart where multiple conditions are checked to categorise a person as either child (<13), teenager (≥ 13 but < 20) or adult (≥ 20), based on age specified:
- Pseudocode** is as follows:
 1. INPUT Age
 2. IF Age < 13 THEN
 3. PRINT "Child"
 4. ELSE IF Age < 20 THEN
 5. PRINT "Teenager"
 6. ELSE
 7. PRINT "Adult"

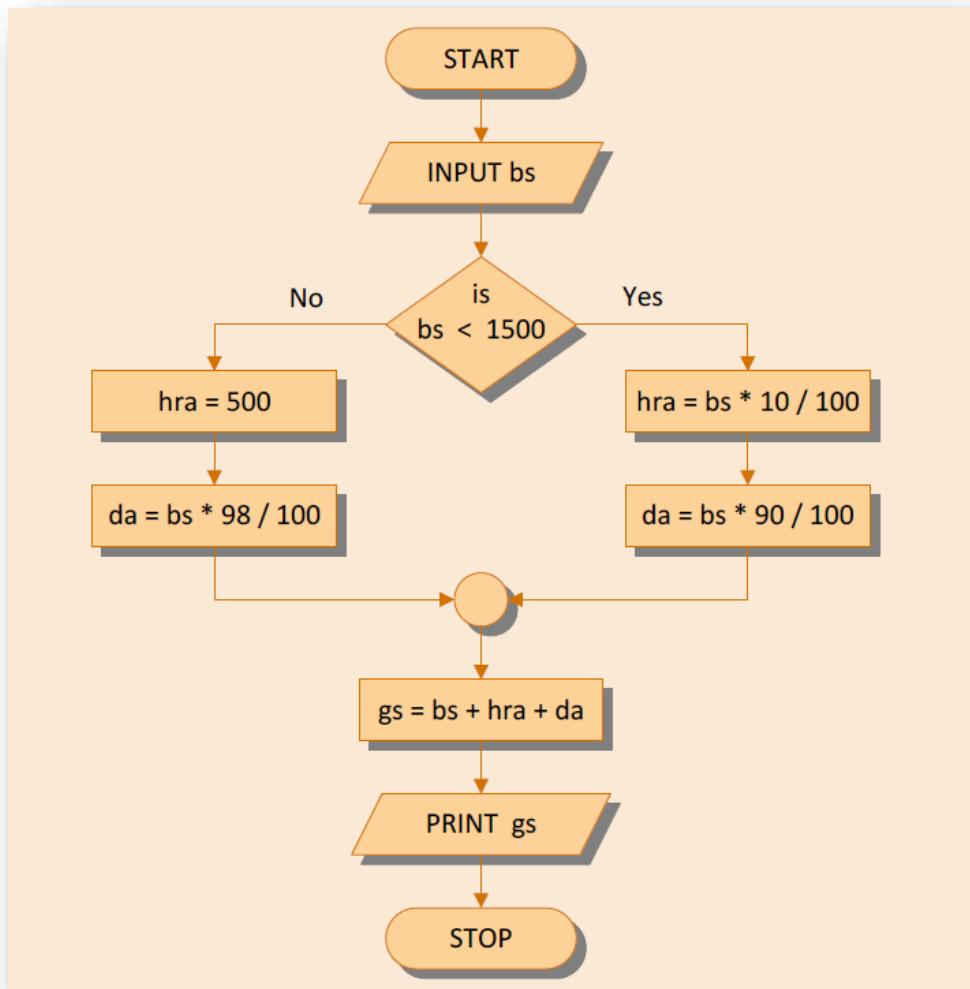
Example 6



Example 7

- In a company an employee is paid as under:
 - If his **basic salary** is less than **Rs. 1500**, then **HRA = 10% of basic salary** and **DA = 90% of basic salary**.
 - If his salary is either equal to or above **Rs. 1500**, then **HRA = Rs. 500** and **DA = 98% of basic salary**.
- If the **employee's salary** is **input** , Draw a **Flowchart** to find his **gross salary**.

Example 7



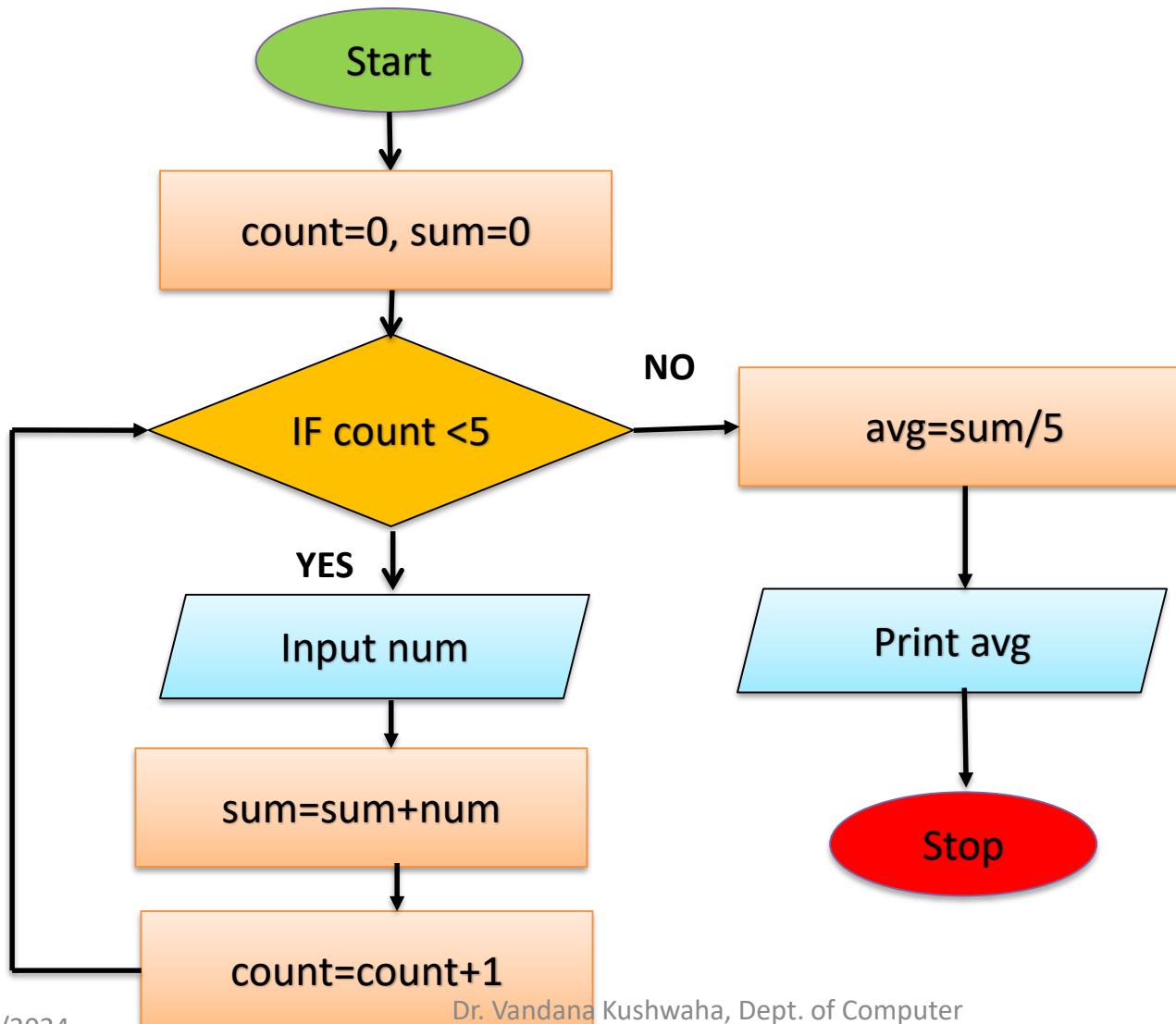
3. Repetition Control

- In programming, **repetition** is also known as **iteration** or **loop**.
- A **loop** in an **algorithm** means **execution** of some program statements **repeatedly till some specified condition is satisfied**.
- **Example 8**
- Write **pseudocode** and draw a flowchart to accept 5 numbers and find their average.
 - Step 1: SET count = 0, sum = 0
 - Step 2: WHILE count < 5 , REPEAT steps 3 to 5
 - Step 3: INPUT a number to num
 - Step 4: sum = sum + num
 - Step 5: count = count + 1
 - Step 6: COMPUTE average = sum/5
 - Step 7: PRINT average

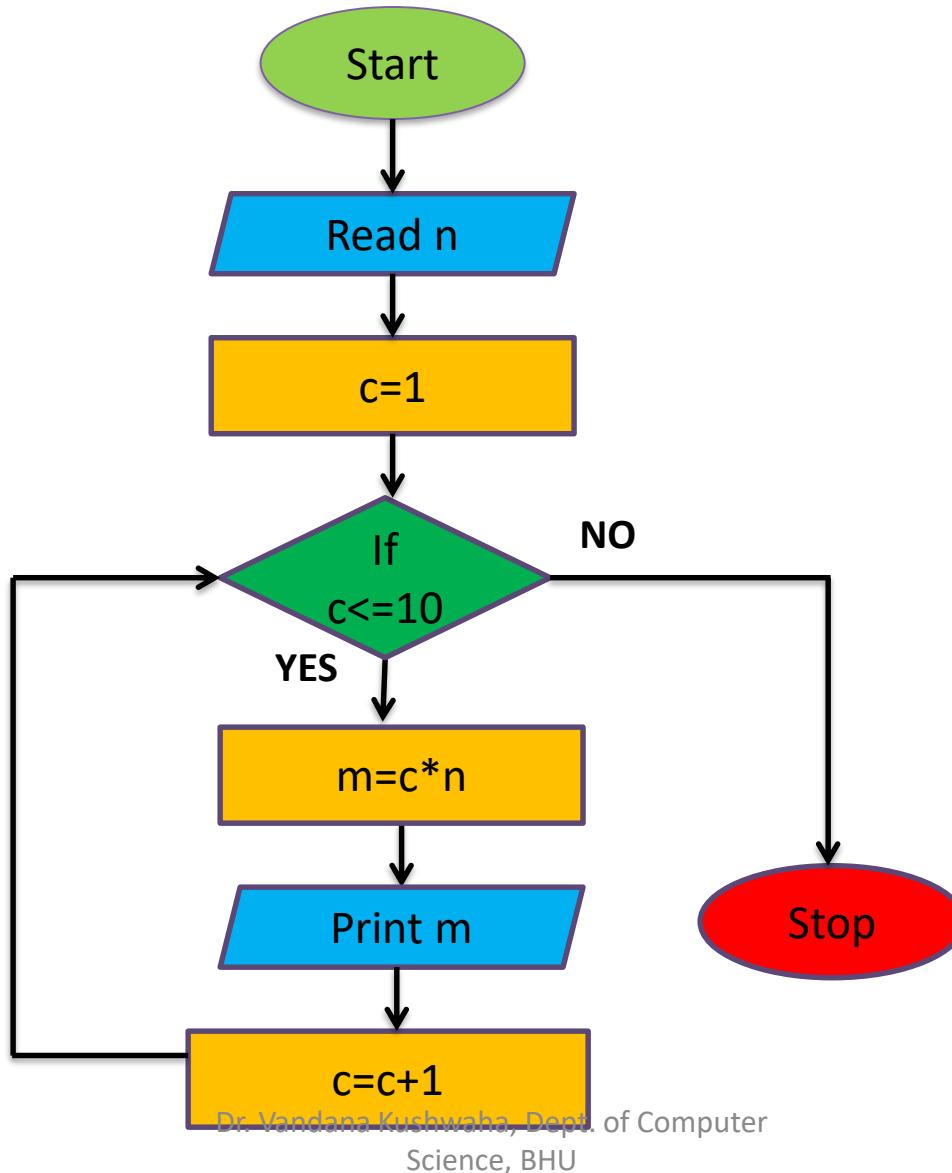
3. Repetition Control

- In **example 8**, a counter called “**count**” keeps track of number of times the loop has been repeated.
- After every **iteration** of the **loop**, the value of count is **incremented by 1** until it performs the set number of repetitions, given in the iteration condition.

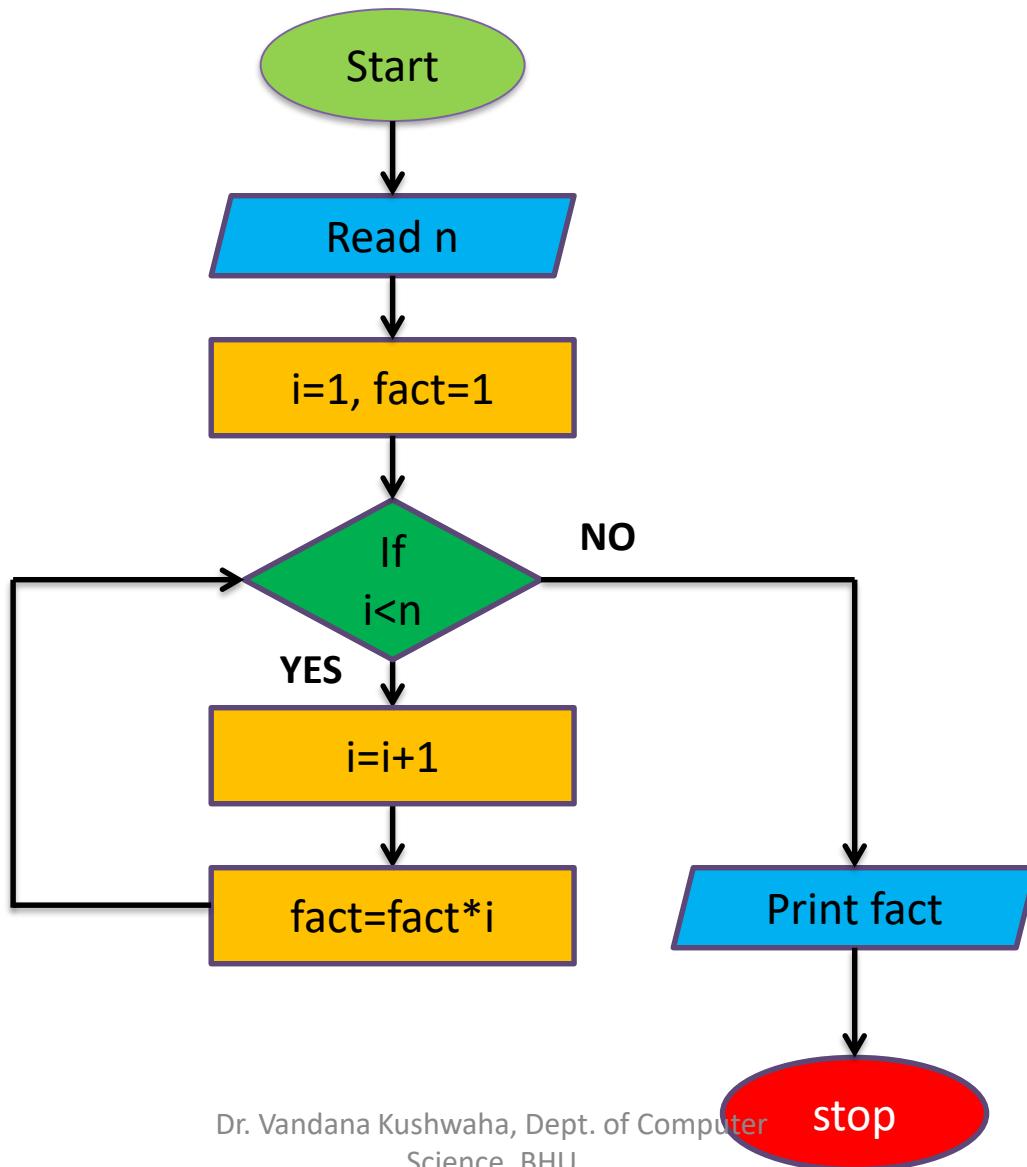
Example 8



Multiplication Table of a number



Flowchart : Factorial of a number



Notations used in Algorithm and Flowchart

- Assignment Symbol (\leftarrow or $=$) is used to assign value to the variable.
- For example to assign value 5 to the variable HEIGHT, statement is **HEIGHT \leftarrow 5** or **HEIGHT = 5**.
- The symbol ' $=$ ' is used in most of the programming language as an **assignment symbol**, the same has been used in all the algorithms and flowcharts in the manual.
- The statement **C = A + B** means that add the value stored in variable A and variable B then assign/store the value in variable C.
- The statement **R = R + 1** means that add **1** to the value stored in variable **R** and then assign/store the new value in variable **R**, in other words increase the value of variable **R** by **1**.

Mathematical Operators

Operator	Meaning	Example
+	Addition	A+B
-	Subtraction	A-B
*	Multiplication	A*B
/	Division	A/B
%	Modulo division(remainder)	A%B
^	Power	A^3 for A ³

Relational Operators

Operator	Meaning	Example
<	Less than	A<B
<=	Less than equal to	A<=B
= or ==	Equal to	A==B
!=	Not equal to	A!=B
>	Greater than	A>B
>=	Greater than equal to	A>=B

Logical Operators

Operator	Example	Meaning
AND	$A < B$ AND $B < C$	Result is True if both $A < B$ and $B < C$ are true else false
OR	$A < B$ OR $B < C$	Result is True if either $A < B$ or $B < C$ are true else false
NOT	NOT ($A > B$)	Result is True if $A > B$ is false else true

Selection control Statements

Selection Control	Example	Meaning
IF (Condition) Then ... ENDIF	IF (X > 10) THEN Y=Y+5 ENDIF	If condition X>10 is True execute the statement between THEN and ENDIF
IF (Condition) Then ... ELSE ENDIF	IF (X > 10) THEN Y=Y+5 ELSE Y=Y+8 Z=Z+3 ENDIF	If condition X>10 is True execute the statement between THEN and ELSE otherwise execute the statements between ELSE and ENDIF

Note: We can use keyword **INPUT** or **READ** or **GET** to accept input(s) /value(s) and keywords **PRINT** or **WRITE** or **DISPLAY** to output the result(s).

Loop control Statements

Selection Control	Example	Meaning
WHILE (Condition) DO ENDDO	WHILE (X < 10) DO print x x=x+1 ENDDO	Execute the loop as long as the condition is TRUE

Advantages of using Flowcharts

- **Communication** : Flowcharts are better way of communicating the logic of a system to all concerned.
- **Effective Analysis** : With the help of flowchart, problem can be analysed in more effective way.
- **Proper Documentation** : flowcharts serve as a good program documentation, which is needed for various purposes.
- **Efficient Coding** : The flowcharts act as a guide or blueprint during the systems analysis and program development phase.
- **Proper Debugging** : The flowchart helps in debugging process.
- **Efficient Program Maintenance** : The maintenance of operating program becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on that part.

Lecture 3

Programming Languages

CSCMJ101: : Introduction to Programming

using C

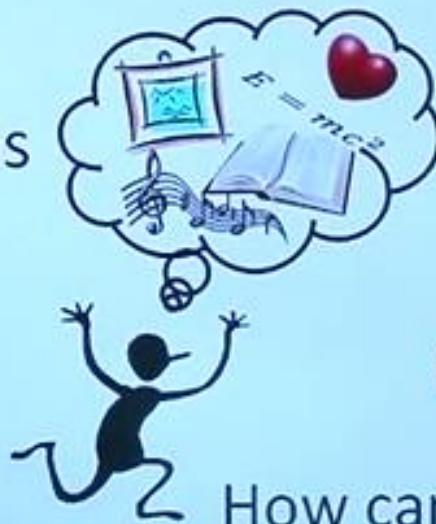
Dr. Vandana Kushwaha

Department of Computer Science
Institute of Science, BHU, Varanasi

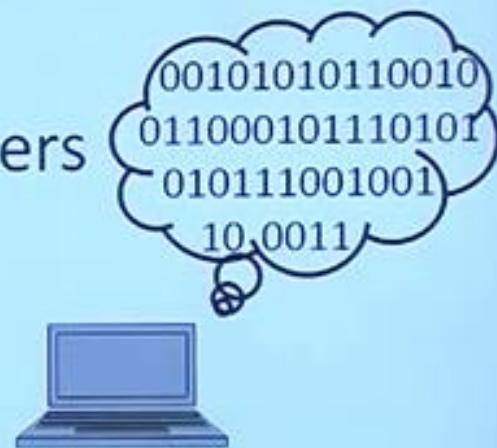
Introduction

Computer Languages

Humans



Computers



How can humans “talk to”
(instruct) computers?

Answer: Computer languages (e.g., Java, C,
Python, FORTRAN, Basic, C++, Lisp, Ruby, ...)

Introduction

- To communicate with a computer, we need a language that a computer understands.
- A computer needs step-wise instructions to perform any action.
- A program is a set of instructions that tells the computer what to do.
- After preparing an algorithm and a flowchart for a particular software it has to be written in the computer coded language.
- Programming languages are used in computer programming to implement specific algorithms.
- A programming language is a set of written symbols that instructs the computer hardware to perform specific tasks.
- Typically, a programming language consists of a vocabulary and a set of rules (called syntax) that the programmer must learn.

Generation of Programming languages

- Programming languages can be classified into the following categories:
 - **Machine Language (First Generation) – 1940s**
 - **Assembly Language (Second Generation) – 1950s**
 - **High Level Language (Third Generation)- 1960s**
 - **Fourth Generation Language (4GL)- 1970s through the 1990s,**
 - **Fifth Generation Language (5GL)**

MACHINE LANGUAGE (FIRST GENERATION)

- Machine language is the **only language** that a **computer understands**.
- Unlike humans, **computers can understand language of only 0s and 1s** (binary digits), i.e., **binary number system**.
- A **program** written in **0s and 1s** is called **machine language program** or binary language program.
- **Machine language was the first generation computer language.**
- A **sample code** in machine language is shown below:
 - **ADD A B** is written as
 - 00010001 00111011 00001100

MACHINE LANGUAGE

- Machine language is faster in executions since the computer directly starts executing it.
- At the same time it is very difficult to write, debug and understand as it is a time consuming process.
- There is not, however, one universal machine language because the language must be written in accordance with the special characteristics of a given processor.
- Each type or family of processor requires its own machine language.
- For this reason, machine language is said to be machine-dependent (also called hardware-dependent).

MACHINE LANGUAGE

- Thus a **machine language program** written on one computer may or **may not run on another computer.**
- Because of this **very few people** opt for **specialization in machine language.**
- In the computer's **first generation**, programmers had to use **machine language** because no other option was available.
- Machine language is also known as **low-level language.**
- Since the **programmer** must **specify every detail** of an **operation**, a **low-level language** requires that the **programmer** have **detailed knowledge** of how the **computer works.**

ASSEMBLY LANGUAGE

- As it was very difficult to generate code in machine language, the **assembly language** was developed which consisted of small meaningful terms.
- They are also classified as **low-level languages** because detailed knowledge of **hardware** is still required.
- This is considered to be the **Second generation language**.
- **Assembly languages** use **mnemonic operation codes** and **symbolic addresses** in place of **1s** and **0s** to represent the **operation codes**.
- A **mnemonic** is an **alphabetical abbreviation** used as memory aid.
- This means a **programmer** can use **abbreviation** instead of having to remember lengthy **binary instruction codes**.

ASSEMBLY LANGUAGE

- An **example** of an **assembly language program** for adding two numbers **A** and **B** and storing the result in some memory location:
 - LDA A (Load value of A in accumulator)
 - ADA B (Add value of B in the value of accumulator)
 - OUT A (Display the content of accumulator on output device)
- A **machine** cannot execute an **assembly languages** program directly as it is not in a **binary form**.
- An **assembler is needed in order to** translate an **assembly language** program into the **object code executable** by the machine.

ASSEMBLY LANGUAGE

- Writing a program in **Assembly language** is **more convenient** than in **Machine language**.
- **It is more readable.**
- But **assembly language** has **some complications** associated with it like:
 - Assembly language is **not portable**. It means that **assembly language program** written for one **processor** will not work on a different **processor**.
 - Assembly language program is **not as fast as machine language** because it has to be first **translated** into **machine (binary) language code**.
- **Machine language and Assembly language** are referred to as **low level languages**.

HIGH LEVEL LANGUAGE

- High level language is quite similar to the English language thus user friendly.
- **BASIC, C, PASCAL, COBOL, FORTRAN** and **Java** are some popular **high level languages**.
- These are called **High level languages (HLL)** or **Third generation languages**.
- The program shown below is written in **BASIC** to obtain the sum of two numbers.
 - 1 X=7
 - 2 Y=10
 - 3 PRINT X + Y
 - 4 END
- The time and cost of creating machine and assembly languages was quite high and this was the prime motivation for the development of high level languages.

HIGH LEVEL LANGUAGE

- A **high level source program** must be **translated** first into a form which the **machine can understand**.
- Such **translation** is done by **software called *Compiler and Interpreter***.

Characteristics:

1. Readability
2. Portability
3. Easy Debugging
4. Easy Software development

FOURTH GENERATION LANGUAGE(4GL)

- Fourth generation languages are **closer to human language** than other **high level languages**.
- These are **highly user friendly** and **independent of any operating system**.
- Designed to **reduce time** and the **cost of programming**.
- In **4GL**, the user has to specify only the required output they want , while the computer determines the sequence of instructions that will accomplish those results.

FOURTH GENERATION LANGUAGE

- These are used mainly in **database programming** and **scripting**.
- **Example of these languages include Perl, Python, Ruby, SQL, MATLAB etc.**
- The query languages like **SQL**(Structured Query Language), which are used to answer queries or question with data from a database.
- **For example:**
 - **SELECT Name, Address**
 - **FROM Personnel**
 - **WHERE Name = “ Krishna”;**

FIFTH GENERATION LANGUAGE(5GL)

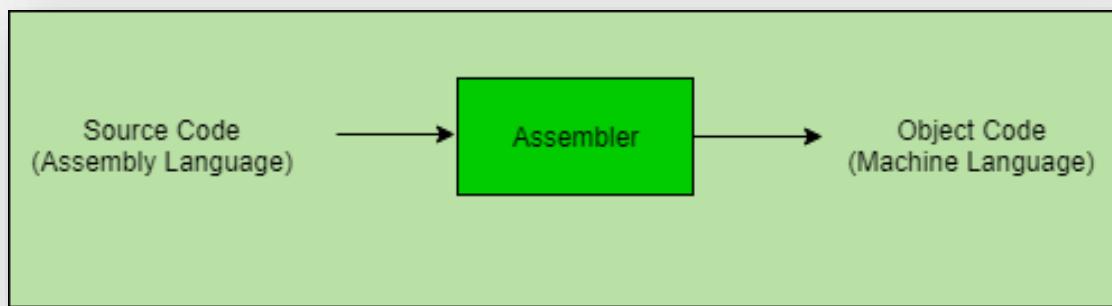
- **Fifth generation programming languages** will permit the user to give commands in a more conversational way.
- User can **voice input** devices rather than traditional keyboards or pointing devices.
- **Fifth generation languages** are used mainly in **Artificial intelligence research**.
- Example of **fifth generation languages**: **PROLOG, OPS5, and MERCURY** are the best known fifth generation languages.
- In **artificial intelligence**, an **expert system** is a computer **system** emulating the **decision-making ability** of a human expert.
- **Expert systems** are designed to solve **complex problems** by reasoning through knowledge.
- **Fifth-generation languages** are designed to make the computer solve a given problem **without the programmer**.

The Language Processor

- The **Machine level language** is very complex to understand and code, therefore the users prefer the **High-Level Language** for coding.
- These **codes** need to be **converted** into the **machine language** so that the **computer** can **easily understand** and work accordingly.
- This operation is performed by the **Language Processor**(**Language Translator**).
- There are **three** different types of language processor:
 - 1. Assembler**
 - 2. Interpreter**
 - 3. Compiler**

Assembler

- The **Assembler** is used to **translate** the program written in **Assembly language** into **Machine code**.
- The **Source program** is a **input** of **Assembler** that contains **Assembly language instructions**.
- The **output** generated by **Assembler** is the **Object code** or **Machine code** understandable by the computer.



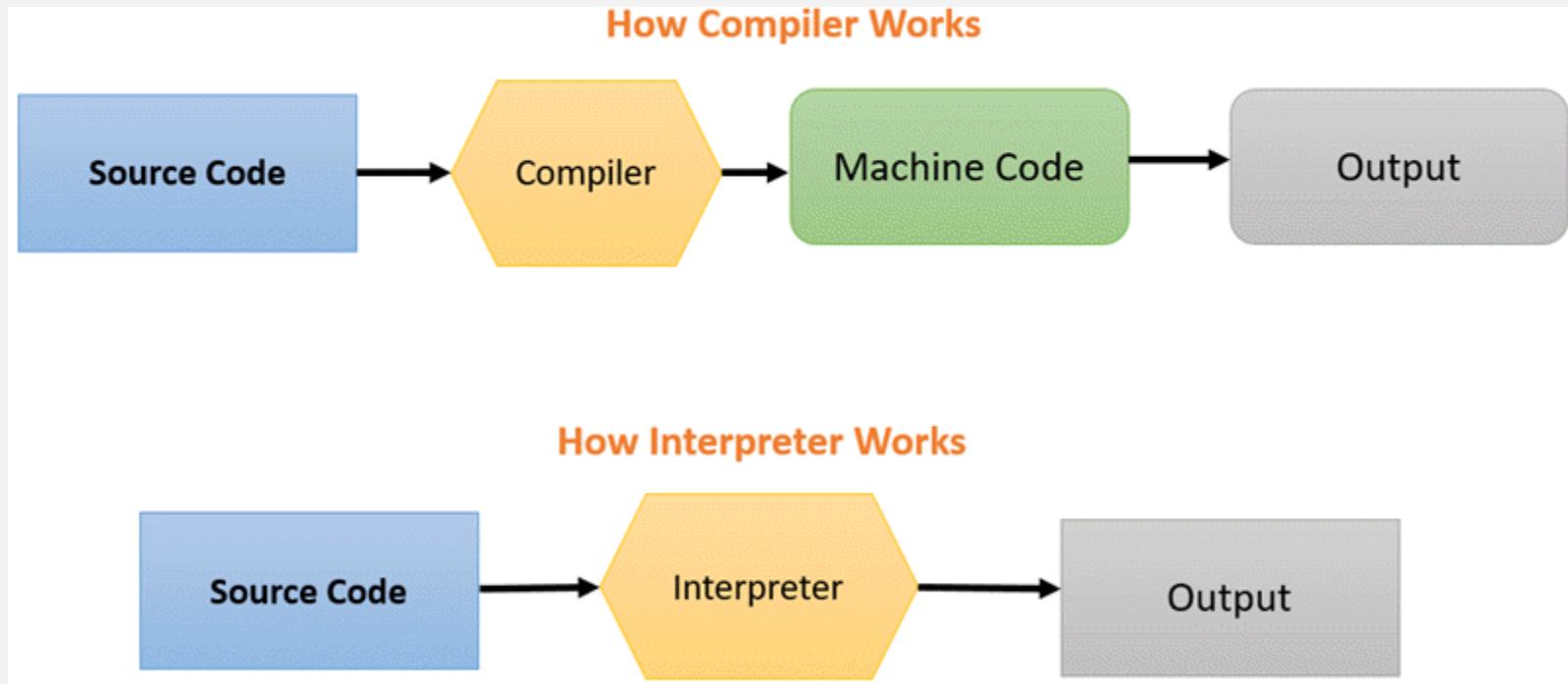
Interpreter

- The **interpreter** translates a **single statement** of **source program** written in **high level language** into **machine code** and **executes it immediately** before moving on to the next line.
- If there is an **error** in the statement, the **interpreter** terminates its translating process at that statement and displays an error message.
- The **interpreter** moves on to the **next line** for execution only after **removal** of the **error**.
- **Example:** Perl, Python and MATLAB etc.

Compiler

- The **language processor** that reads the **complete source program** written in **High level language** as a **whole in one go** and translates it into an **equivalent program** in **Machine language** is called as a **Compiler**.
- **Example:** C, C++, C#, Java.
- In a **compiler**, the **source code** is translated to **object code** successfully if it is **free of errors**.
- The **Compiler** specifies the **errors** at the **end of compilation** with line numbers when there are any errors in the **source code**.
- The **errors** must be **removed** before the **compiler** can **successfully recompile** the **source code** again.

Compiler vs. Interpreter



Compiler vs. Interpreter

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
Interpreters usually take less amount of time to analyze the source code. However, the overall execution time is comparatively slower than compilers.	Compilers usually take a large amount of time to analyse the source code. However, the overall execution time is comparatively faster than interpreters.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Programming languages like JavaScript, Python, Ruby use interpreters.	Programming languages like C, C++, Java use compilers.

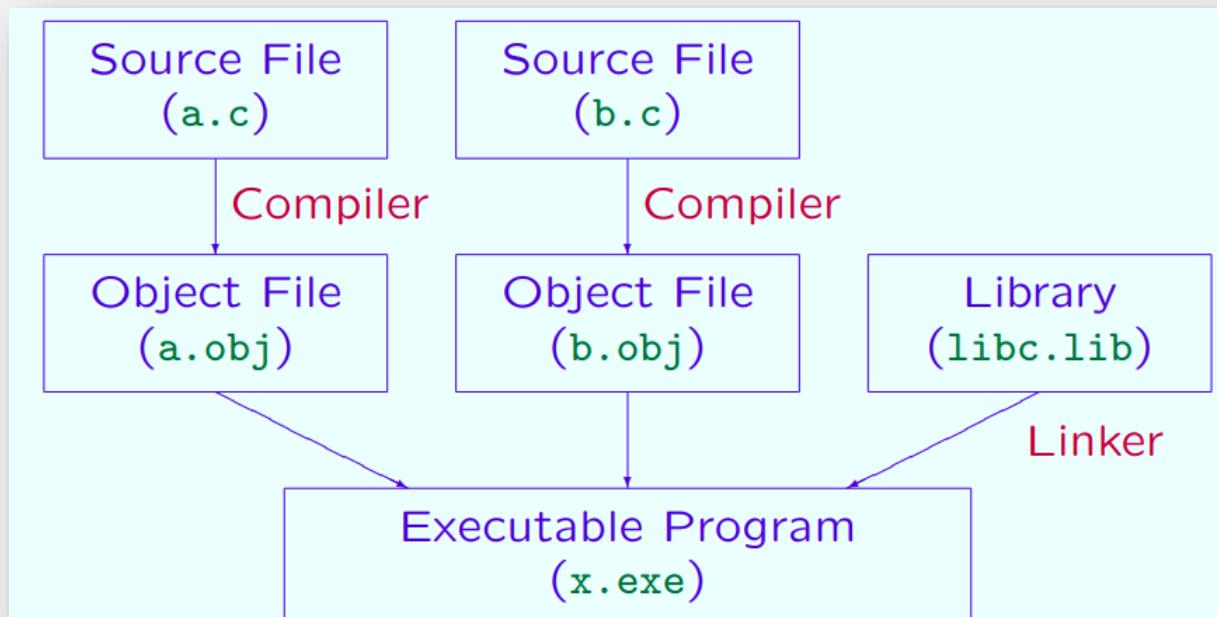
Linker

- In computer science, a **linker** is a **computer program** that takes one or more **object files** generated by a **Compiler** and combines them into one, **executable program**.
- A large **C** programs are usually made up of **multiple modules** that span **separate object files**, each being a compiled computer program.
- The **program** as a whole refers to these separately **compiled object files**.
- The **linker** combines these **separate files** into a **single, unified program**.
- The **object code** is **combined** with required **supporting code** to make an **executable program**.
- This step typically involves adding in any **libraries** that are required.

Linker

- The **compiler** does not directly produce an **executable program**.
- It produces an **Object file** (a file with object code).
- The **Object file** contains **machine code**, but with **gaps** (for the addresses of unknown functions).
- It is input to another **program**, the **linker**.
- The **Linker** adds **machine code** for the **needed functions** from a **library** (the standard C library).
- It puts the **addresses** of these **functions** in the **places** where they are **called**.
- But it is possible (and indeed very common) to write **programs** that consist of **several source files**.

Linker



Lecture 4

Introduction to C Programming

CSCMJ101: : Introduction to

Programming using C

Dr. Vandana Kushwaha

Department of Computer Science
Institute of Science, BHU, Varanasi

Introduction

- C is a **programming language** developed at **AT & T's Bell Laboratories** of USA in **1972** by **Dennis Ritchie**.
- C **language** was created for a specific purpose i.e designing the **UNIX operating system**.
- In the **mid-1980s** it became important to establish an **official standard** for C, since it was being used in **projects** subject to **commercial** and **government** contracts.
- In **1983** the **American National Standards Institute (ANSI)** set up a **committee** that further **amended and standardized the language. (ANSI C)**
- C **Programming** is near to **machine** as well as **human** so it is called as **Middle-level Programming Language**.

Why was Name "C" given to Language?

- CPL was **Common Programming Language**.
- In **1967**, BCPL Language (Basic CPL) was created as a scaled down version of **CPL**.
- BCPL and CPL are the earlier ancestors of **B Language**.
- Many of C's **principles** and ideas were derived from the earlier language B. (**Ken Thompson** was the developer of **B Language**.)
- As many of the **features** were **derived** from "B" Language that's why it was named as "C".
- After 7-8 years **C++** came into existence which was first example of **object oriented programming** .

Why 'C' is important?

- C is a **powerful language** that includes a **collection of in-built functions** and **operators** that help in writing any **complex program**.
- C programs are very **efficient** because they contain a variety of **data types** and **robust operators**.
- The **capabilities** of an **assembly language** and **high level language** are combined together in the **C compiler**, which makes it the most **suitable language** for writing **system software** as well as **commercial software**.
- C is **fast, flexible, portable** and structured programming language having a **rich library**.

Applications of 'C'

- **Operating Systems Development**
- **C language** was originally developed to **write UNIX operating system**.
- Also, the execution time of the **programs** written in **C language** is equivalent to that of **assembly language**, which has made **C language** the most crucial part in the **development of various operating systems**.
- **Unix-Kernel, Microsoft Windows utilities and operating system applications**, and a big segment of **Android operating system** have all been **written in C language**.

Applications of 'C'

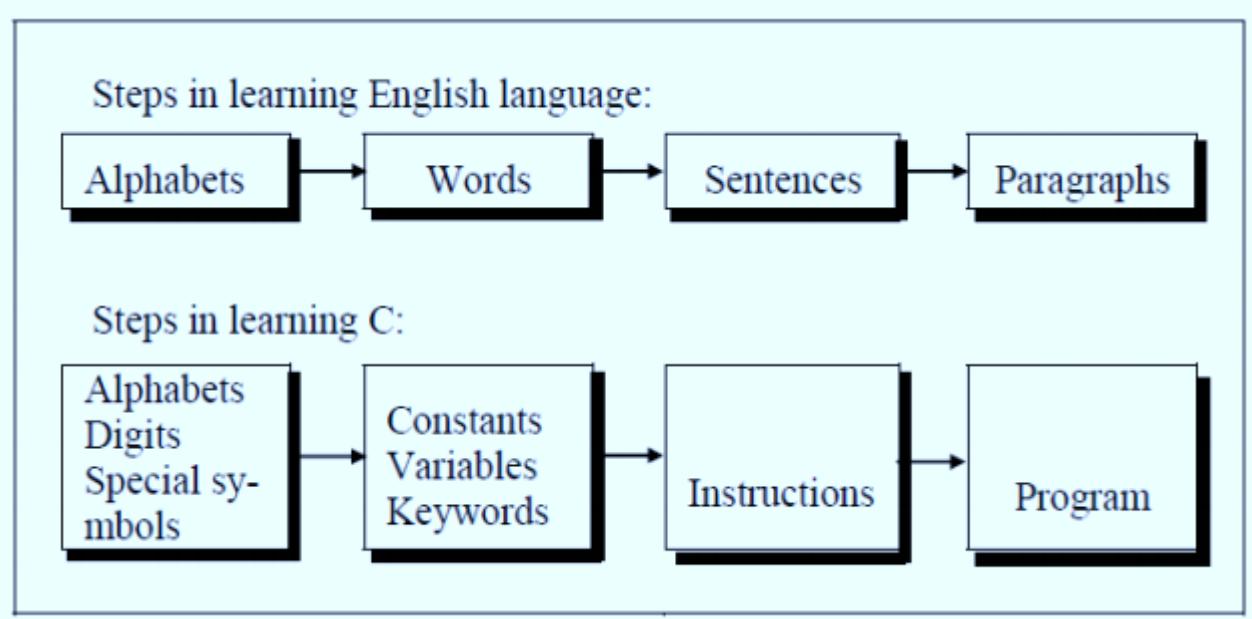
- **Development of New Languages**
- The code written in **C language** is simple and execution efficient.
- As a result, the development of various languages has been influenced by **C language**.
- These languages are C++ (also known as C with classes), C#, Python, Java, JavaScript, Perl, PHP, Verilog, D, Limbo and C shell of Unix etc.
- Every language uses **C language** in variable capacity.
- For example, **Python** uses **C** for creating **standard libraries**, whereas the syntaxes and control structures of languages like C++, PHP and Perl are based on C.

Applications of 'C'

- **Computation Platforms**
- The **swift implementation of algorithms** and **data structures** in **C** facilitates **quicker computation** in programs.
- Thereby enabling the usage of C in those applications which require calculations of higher degree such as **Mathematica**, **MATLAB** etc.
- **Embedded Systems**
- **C language** is the preferred choice for **writing the drivers of embedded systems** along with embedded systems applications.
- The reason why this language is the most preferred one is its access to **machine level hardware APIs**, along with the presence of C compilers, dynamic memory allocation and deterministic use of resources.

Learning C programming

There is a close analogy between learning English language and learning C language.



Character set

- A character denotes any alphabet, digit or special symbol used to represent information.
- Valid alphabets, numbers and special symbols allowed in C are:

Alphabets	A, B,, Y, Z a, b,, y, z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ` ! @ # % ^ & * () _ - + = \ { } [] : ; " ' < > , . ? /

- The alphabets, numbers and special symbols when properly combined form constants, variables and keywords.

'C' Tokens

- Tokens are **the smallest elements of a program**, which are meaningful to the compiler.
- The following are the types of **tokens**:
 - Keywords,
 - Identifiers,
 - Constant,
 - Strings,
 - Special Symbols
 - Operators, etc.

Keyword

- There are certain words reserved for doing specific task, these words are known as **reserved word or keywords**.
- These words are predefined and always written in **lower case or small letter**.
- These keywords can't be used as a variable name as it assigned with fixed meaning.
- C language supports **32 keywords**.
- Some examples are int, short, signed, unsigned, default, volatile, float, long, double, break, continue, typedef, static, do, for, union, return, while, do, extern, register, enum, case, goto, struct, char, auto, const etc.

Identifiers

- Identifiers are user defined word used to name of entities like variables, arrays, functions, structures etc.
- **Rules for naming identifiers are:**
 1. Name should only consists of alphabets (both upper and lower case), digits and underscore (_) sign.
 2. First characters should be alphabet or underscore
 3. Space is not allowed in identifier naming
 4. Name should not be a keyword
 5. Since C is a case sensitive, the upper case and lower case considered differently, for example code, Code, CODE etc. are different identifiers.
 6. Identifiers are generally given in some meaningful name such as value, net_salary, age, data etc.

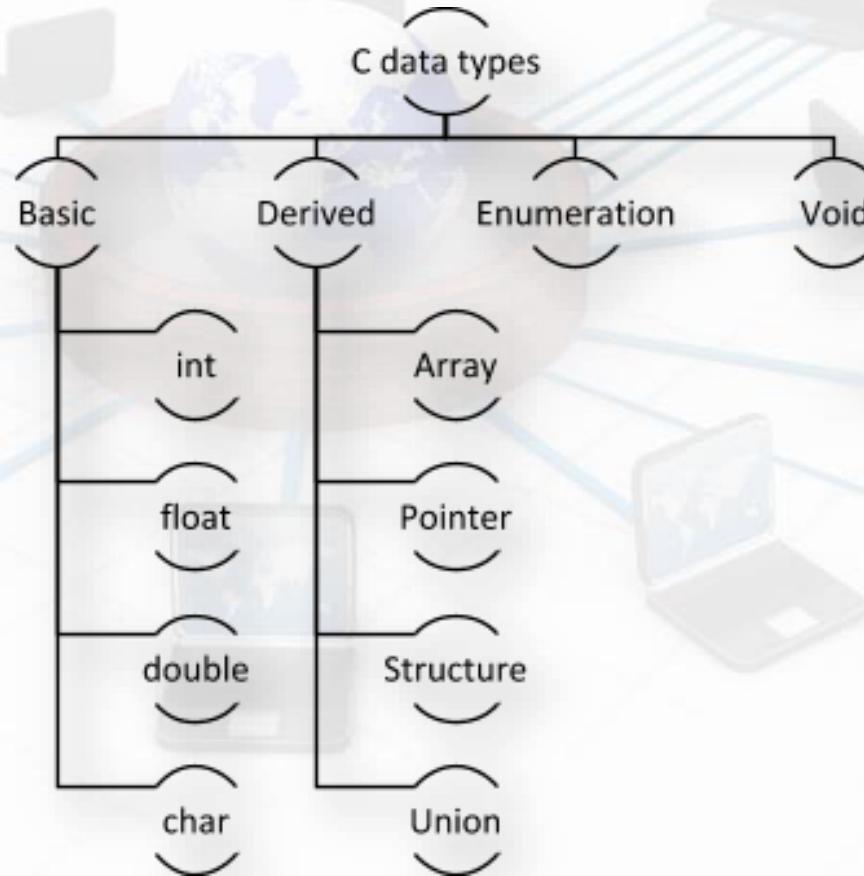
Identifiers

- An identifier name may be long, some implementation recognizes only first eight characters, most recognize 31 characters.
- Some **invalid identifiers** are :
 - 5cb (name can't start with a number)
 - int (name can't be a keyword)
 - res# (special character other than underscore(_) is not allowed)
 - avg no (space is not allowed)

Data types

- Data types refer to an extensive system used for declaring variables or functions of different types before its use.
- The data type of a variable determines which type of value it stores, how much space it occupies in storage.
- C has the following data types :
 1. **Basic data types:** int, float, double, char.
 2. **Derived data type:** pointer, array, structure, union.
 3. **Void data type:** void means no data type.
 4. **Enumerated**

Data types



Data types

- **Integer data type:** A variable declared to be of type **int** can be used to contain integral values only—that is, values that do not contain decimal places.
- **Float data type:** A variable declared to be of type **float** can be used for storing **floating-point numbers** (values containing decimal places) eg. 45.50, 20.00 etc.
- **Double data type:** The **double** type is the same as type **float**, only with roughly **twice the precision**.
- **Character data type:** The **char** data type can be used to store a single character, such as the letter **a**, the digit character **6**, or a special character semicolon ;
- **Qualifiers:** There are **two types of type qualifier** in C
 - **Size qualifier:** short, long
 - **Sign qualifier:** signed, unsigned

Data types

- When the qualifier **unsigned** is used the **number** is **always positive**, and when **signed** is used **number** may be **positive or negative**.
- If the **sign qualifier** is **not mentioned**, then by **default sign qualifier** is assumed.
- The range of values for **signed data types** is **less** than that of **unsigned data type**.
- Because in **signed type**, the left most bit is used to represent sign, while in **unsigned type** this bit is also used to represent the value.
- The size and range of the different data types on a **16 bit machine** is given on next slide:

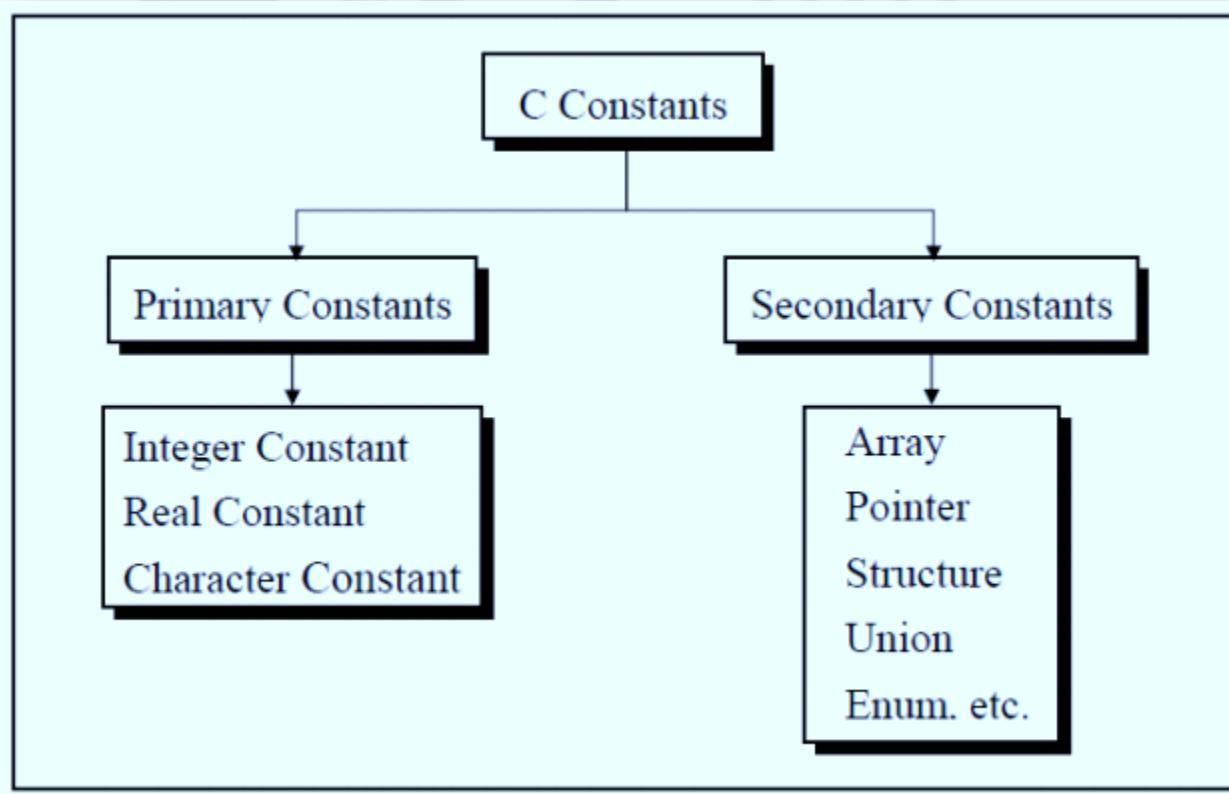
Data types

Basic data type	Data type with type qualifier	Size (byte)	Range
char	char or signed char	1	-128 to 127
	Unsigned char	1	0 to 255
int	int or signed int	2	-32768 to 32767
	unsigned int	2	0 to 65535
	short int or signed short int	1	-128 to 127
	unsigned short int	1	0 to 255
	long int or signed long int	4	-2147483648 to 2147483647
	unsigned long int	4	0 to 4294967295
float	float	4	-3.4E-38 to 3.4E+38
double	double	8	1.7E-308 to 1.7E+308
	Long double	10	3.4E-4932 to 1.1E+4932

Constants

- **Constant** is a any value that cannot be changed during program execution.
- In C, any number, single character, or character string is known as a constant.
- A **constant** is an **entity** that **doesn't change** whereas a **variable** is an **entity** that **may change**.
- For example, the number **50** represents a **constant integer** value.
- The character string "Programming in C is fun.\n" is an example of a constant character string.
- **C constants can be divided into two major categories:**
 - Primary Constants
 - Secondary Constants

Constants



Primary Constants

- Primary Constants are fundamental constants in C and the Secondary constants are the constants which is created with the help of Primary Constants.
- **Integer Constants:** Integer constants are numeric integer values, like 2, 5, 245, -540, 0, -3 etc.
- **Real Constants:** Real Constants are numeric fractional values, like 3.0, 8.5, 1964.98 etc.
Note: 2.0 is a Real constant in C.
- **Character Constants:** Any single symbol with single quote is called character constants in C, like 'a', '+', '2', 'B' etc.
- **Note:** '-3' is not a character constant because there is two symbols (- and 3) similarly '3.0' is also not a character constant because it has three symbols (3, . and 0), there must be a single symbol under single quote to be a character constant in c.

Variables

- **Variable** is a data name which is used to store some data value or symbolic names for storing program computations and results.
- The value of the variable can be change during the execution.
- The **rule for naming the variables** is same as the naming identifier.
- Before used in the program it must be declared.
- **Declaration of variables** specify its **name**, **data types** and **range** of the value that variables can store depends upon its data types.
- **Syntax:** Data type **variable_name;**
- **Example:**
 - int a;
 - char c;
 - float f;

Variable initialization

- When we assign any initial value to variable during the declaration, is called initialization of variables.
- When variable is declared but contain undefined value then it is called **garbage value**.
- The variable is initialized with the assignment operator such as :
- **Syntax:** **Data type variable_name = constant;**
- **Example:**
 - int a=20;
- Or**
- int a;
 - a=20;

Structure of C Language program

Following are the main **components** of a **C program**:

- 1) Comment line
- 2) Pre-processor directive
- 3) Global variable declaration
- 4) **main function()**
- 5) User defined function()

Structure of C Language program

```
// Name of Program → Documentation section  
  
#include<stdio.h>  
#include<conio.h> } → Preprocessor Directives  
  
#define max 100 → Definition section  
void add(); } → Global declaration section  
int x=100; }  
  
int main() → main () Function section / Entry Point  
{ int a=100; → Variable declaration  
  
printf("Hello Main"); } → Body of Main function  
return 0;  
}  
  
void add(){ → Function Definition  
printf("Hello add");  
}
```

Structure of C Language program

- **Comment line**
- Comment line It indicates the purpose of the program.
- It is represented as /* */
- Comment line is used for increasing the readability of the program.
- It is useful in explaining the program and generally used for documentation.
- Comment line can be single or multiple line but should not be nested.

Structure of C Language program

- **Pre-processor Directive**
- #include tells the compiler to include information about the standard input/output library.
- It is also used in symbolic constant such as #define PI 3.14(value).
- The stdio.h (standard input output header file) contains definition & declaration of system defined function such as printf(), scanf() etc.
- Generally printf() function used to display and scanf() function used to read value.
- **Global Declaration:**
- This is the section where variable are declared globally so that it can be access by all the functions used in the program.
- And it is generally declared outside the any function.

main() Function

- **main()** It is the **user defined function** and every program has one main() function from where actually program is started.
- The **main() function** can be anywhere in the program but in general practice it is placed in the first position.

Method 1:

```
void main()  
{ .....  
.....  
}
```

- The main() function **return no value** when it declared by data type as **void**.

main() Function

Method 2:

```
int main( )  
{  
---  
return 0;  
}
```

- The main() function **return zero value** when it declared by data type as **int**.
- The program execution start with opening braces and end with closing brace.
- And in between the two braces declaration part as well as executable part is mentioned.
- And at the end of each line, the semi-colon is given which indicates statement termination.

First C program

- /*First c program with return statement*/

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

```
int main ( )
```

```
{
```

```
    printf ("Welcome to C Programming language.");
```

```
    return 0;
```

```
}
```

- **Output:**

Welcome to C Programming language.

Program 2

```
#include <stdio.h>
int main ( )
{
    int v1, v2, sum; //v1,v2,sum are variables and int is data type declared
    v1 = 150;
    v2 = 25;
    sum = v1 + v2;
    printf("%d", sum);
    return 0;
}
```

Output:

- 175

Program 3

```
#include <stdio.h>
int main ( )
{
    int v1, v2, sum; //v1,v2,sum are variables and int is data type declared
    v1 = 150;
    v2 = 25;
    sum = v1 + v2;
    printf ("The sum of %d and %d is= %d", v1, v2, sum);
    return 0;
}
```

Output:

- The sum of 150 and 25 is=175

Program 4

```
#include <stdio.h>
int main ( )
{
    int v1, v2, sum; //v1,v2,sum are variables and int is data type declared
    printf("Enter the two numbers");
    scanf("%d %d", &v1, &v2);
    sum = v1 + v2;
    printf ("The sum of %d and %d is= %d", v1, v2, sum);
    return 0;
}
```

Output:

Enter the two numbers **20**

30

The sum of 20 and 30 is=50

Lecture 5

Operators in C

**CSCMJ101: Problem Solving through C
Programming**

Dr. Vandana Kushwaha

Department of Computer Science
Institute of Science, BHU, Varanasi

Introduction

- C language supports a rich set of built-in operators.
- An operator is a symbol that tells the compiler to perform a certain mathematical or logical manipulation.
- Operators are used in programs to manipulate data and variables.
- Some operator requires two operands to perform operation(called binary operator) while some requires single operand (called unary operator).
- Several operators are there in C:
 - *Arithmetic operators *Relational operators
 - *Logical operators *Assignment operators
 - *Conditional operators *Special operators

Arithmetic Operators

- Assume variable **A** holds **10** and variable **B** holds **20** then –

Operator	Description	Example
+	Adds two operands.	$A + B = 30$
-	Subtracts second operand from the first.	$A - B = -10$
*	Multiplies both operands.	$A * B = 200$
/	Divides numerator by de-numerator.	$B / A = 2$
%	Modulus Operator and remainder of after an integer division.	$B \% A = 0$
++	Increment operator increases the integer value by one.	$A++ = 11$
--	Decrement operator decreases the integer value by one.	$A-- = 9$

- The **modulus operator %** cannot be applied with **floating point** operand.
- There is **no exponent operator** in C.

Integer float Arithmetic

- When **both** the **operand** are **integer** then it is called **integer arithmetic** and the **result** is always **integer**.
- When **both** the **operand** are **floating point** then it is called **floating arithmetic** and the **result** is always **float**.
- When **one operand** is of **integer** and **another** is of **floating point** then it is called **mix type** or **mixed mode arithmetic** and the **result** is in **float type**.

Operation	Result	Operation	Result
$5 / 2$	2	$2 / 5$	0
$5.0 / 2$	2.5	$2.0 / 5$	0.4
$5 / 2.0$	2.5	$2 / 5.0$	0.4
$5.0 / 2.0$	2.5	$2.0 / 5.0$	0.4

Assignment Operator

- A **value** can be **stored** in a **variable** with the use of **assignment operator**.
- The **assignment operator(=)** is used in assignment statement and assignment expression.
- **Operand** on the **left hand side** should be **variable** and the **operand** on the **right hand side** should be **variable** or **constant** or any **expression**.
- **Example:**

```
int x=2;
```

```
int y=x;
```

```
int a=x+y;
```

```
int a=3+5-2;
```

Type Conversion in Assignments

- Sometime it may happen that the **type** of the **expression** and the **type** of the **variable** on the **left-hand side** of the **assignment operator** may **not be same**.
- In such a case the **value** of the **expression** is **promoted** or **demoted** depending on the **type of the variable** on **left-hand side of assignment operator =**.
- **Example**

```
int i ;  
float b ;  
i = 3.5 ; //the float 3.5 is demoted to an int 3 and then stored in variable i.  
b = 30 ; //the int 30 is promoted to float 30.0 and then stored in variable b.
```

- If we try to **print** the values of variable **i** and **b** as `printf("%d %f", i,b);`
- **Output:** 3 30.0

Assignment Operators

Operator	Description	Example
=	assigns values from right side operands to left side operand	a=b
+=	adds right operand to the left operand and assign the result to left	a+=b is same as a=a+b
-=	subtracts right operand from the left operand and assign the result to left operand	a-=b is same as a=a-b
=	mutiply left operand with the right operand and assign the result to left operand	a=b is same as a=a*b
/=	divides left operand with the right operand and assign the result to left operand	a/=b is same as a=a/b
%=	calculate modulus using two operands and assign the result to left operand	a%=b is same as a=a%b

Relational operators

- Assume variable **A** holds 10 and variable **B** holds 20 then –

Operator	Description	Example
<code>==</code>	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	$(A == B)$ is not true.
<code>!=</code>	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	$(A != B)$ is true.
<code>></code>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	$(A > B)$ is not true.
<code><</code>	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	$(A < B)$ is true.
<code>>=</code>	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	$(A >= B)$ is not true.
<code><=</code>	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	$(A <= B)$ is true.

Logical Operators

- Assume variable **A** holds **1** and variable **B** holds **0**, then –

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero(true), then the condition becomes true.	(A && B) is false.
 	Called Logical OR Operator. If any of the two operands is non-zero(true), then the condition becomes true.	(A B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

- In **C** every **nonzero value** is considered as **true(T)** and **zero value** is considered as **false(F)**.

Logical Operators

Truth Table of Logical Operator

C1	C2	C1 && C2	C1 C2	!C1	!C2
T	T	T	T	F	F
T	F	F	T	F	T
F	T	F	T	T	F
F	F	F	F	T	T

Special operator

Operator	Description	Example
sizeof	Returns the size of a variable	<code>sizeof(x)</code> return size of the variable x
&	Returns the address of a variable	<code>&x</code> ; return address of the variable x
*	Pointer to a variable	<code>*x</code> ; will be pointer to a variable

sizeof() operator

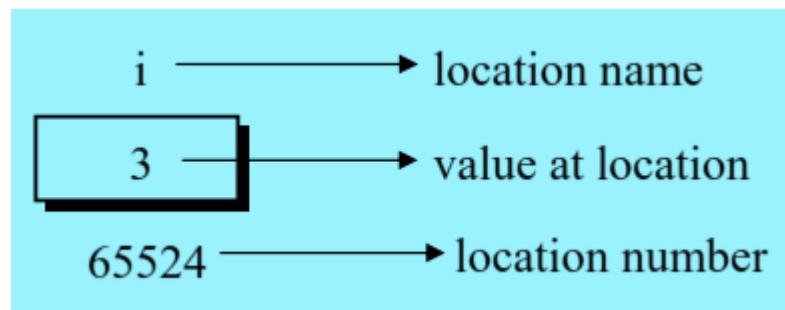
```
int main()
{
    int x=3;
    float y=4.7;
    char ch='a';
    printf("%d %d %d", sizeof(x),sizeof(y),sizeof(ch));
    return 0;
}
```

Output:

4 4 1

& Operator

- Consider the declaration, int i = 3 ;
- This declaration tells the C compiler to:
 - (a) Reserve space in memory to hold the integer value.
 - (b) Associate the name **i** with **this memory location**.
 - (c) Store the value 3 at this location.
- We may represent **i's location in memory by the following memory map**.



& Operator

- & is an ‘Address of’ operator, gives the location **address** used by the variable in memory.
- **&a means address of variable a.**

```
void main( )
{
    int i = 3 ;
    printf ( "\nAddress of i = %u", &i ) ;
    printf ( "\nValue of i = %d", i ) ;
}
```

Output:

Address of i = 65524

Value of i = 3

Pre-increment Operator

- A **pre-increment operator** is used to **increment** the **value** of a **variable** before using it in an expression.
- In the **Pre-Increment**, **value** is **first incremented** and then used inside the expression.
- **Syntax:** **a = ++x;**
- Here, if the value of 'x' is 10 then the value of 'a' will be 11 because the value of 'x' gets modified before using it in the expression.
- **Example:**

```
int main()
{
    int x=1;
    int a=++x;
    printf("x=%d  a=%d",x , a);
```

OUTPUT:

x=2 a=2

Post-increment Operator

- A **post-increment operator** is used to **increment** the **value of the variable after** executing the expression completely in which post-increment is used.
- In the **Post-Increment**, **value** is first used in an expression and then incremented.
- **Syntax:** **a = x++;**
- Here, suppose the value of 'x' is 10 then the value of variable 'a' will be 10 because the old value of 'x' is used.
- **Example:**

```
int main()
{
    int x=1;
    int a=x++;
    printf("x=%d  a=%d",x , a);
}
```

OUTPUT:

x=2 a=1

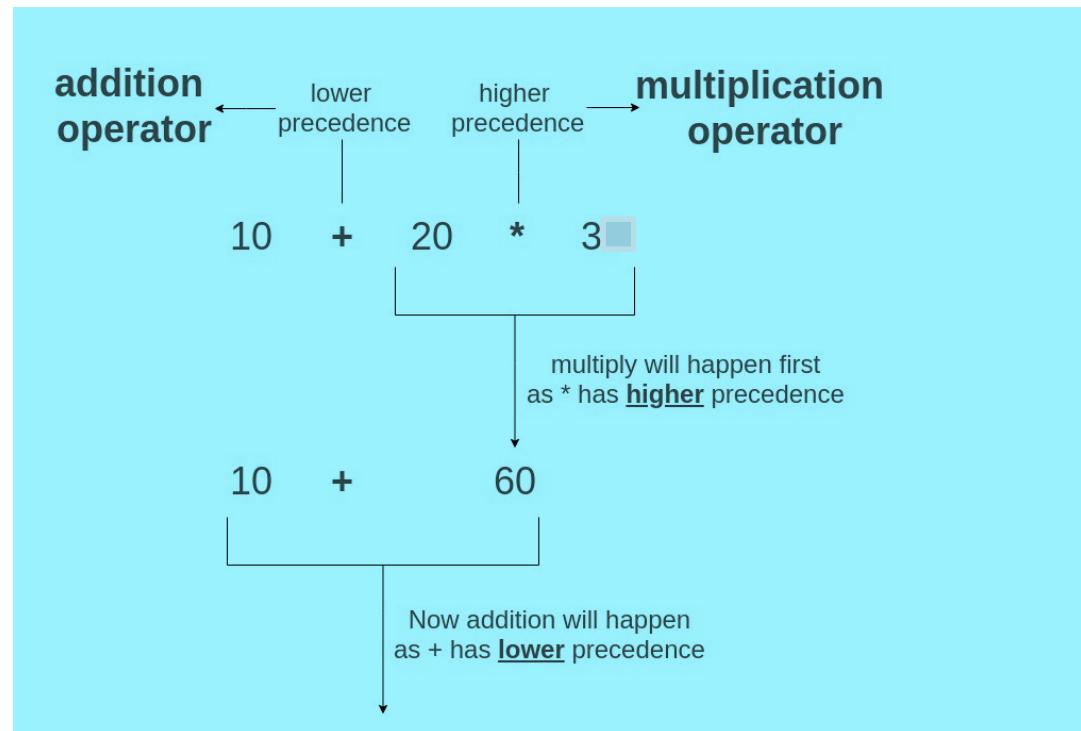
Operator Precedence

- **Operator precedence** determines which **operator** is **performed first** in an **expression** with more than one operators with different precedence.

Priority	Operators	Description
1 st	* / %	multiplication, division, modular division
2 nd	+ -	addition, subtraction
3 rd	=	assignment

Operator Precedence

- **Example:** Solve $10 + 20 * 30$



Operators Associativity

- **Operators Associativity** is used when **two operators of same precedence** appear in an **expression**.
- **Associativity** can be either **Left to Right** or **Right to Left**.
- Consider the **expression**

a = 3 / 2 * 5 ;

- Here there is a tie between operators of same priority, that is between / and *.
- We will **evaluate the expression** from **left to right**.

- **Example:**

- a = 3 / 2 * 5
- =1*5
- =5

Example 1

- Determine the hierarchy of operations and evaluate the following expression:
- $i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8$
- $i = 6 / 4 + 4 / 4 + 8 - 2 + 5 / 8$ operation: *
- $i = 1 + 4 / 4 + 8 - 2 + 5 / 8$ operation: /
- $i = 1 + 1 + 8 - 2 + 5 / 8$ operation: /
- $i = 1 + 1 + 8 - 2 + 0$ operation: /
- $i = 2 + 8 - 2 + 0$ operation: +
- $i = 10 - 2 + 0$ operation: +
- $i = 8 + 0$ operation : -
- $i = 8$ operation: +

Example 2

- Consider one more expression $a = b = 3$;
- Here **both assignment operators** have the **same priority and same associativity (Right to Left)**.
- Thus we will evaluate it from **right to left**:

$a = b = 3$;

- First **b** is **initialized** with **3** i.e. $b=3$
- After that the **value of b** (which is **3**) is **initialised to a**.
- After **evaluation** both **a** and **b** are equal to **3**.

Precedence	Operator	Description	Associativity
1	<code>++ --</code> <code>()</code> <code>[]</code> <code>.</code> <code>-></code> <code>(type){list}</code>	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access Structure and union member access through pointer Compound literal(C99)	Left-to-right
2	<code>++ --</code> <code>+ -</code> <code>! ~</code> <code>(type)</code> <code>*</code> <code>&</code> <code>sizeof</code> <code>_Alignof</code>	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of Alignment requirement(C11)	Right-to-left
3	<code>* / %</code>	Multiplication, division, and remainder	Left-to-right
4	<code>+ -</code>	Addition and subtraction	Left-to-right
5	<code><< >></code>	Bitwise left shift and right shift	
6	<code>< <=</code> <code>> >=</code>	For relational operators <code><</code> and <code>≤</code> respectively For relational operators <code>></code> and <code>≥</code> respectively	
7	<code>== !=</code>	For relational <code>=</code> and <code>≠</code> respectively	
8	<code>&</code>	Bitwise AND	
9	<code>^</code>	Bitwise XOR (exclusive or)	
10	<code> </code>	Bitwise OR (inclusive or)	
11	<code>&&</code>	Logical AND	
12	<code> </code>	Logical OR	
13	<code>? :</code>	Ternary conditional	Right-to-Left
14	<code>=</code> <code>+= -=</code> <code>*= /= %=</code> <code><<= >>=</code> <code>&= ^= =</code>	Simple assignment Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	Left-to-right
15	<code>,</code>	Comma	

Algebraic expression in C

- How to convert a **general arithmetic statement** to a **C statement**.
- Some of the **examples of C expressions** are shown below:

Algebraic Expression	C Expression
$a \times b - c \times d$	$a * b - c * d$
$(m + n) (a + b)$	$(m + n) * (a + b)$
$3x^2 + 2x + 5$	$3 * x * x + 2 * x + 5$
$\frac{a + b + c}{d + e}$	$(a + b + c) / (d + e)$
$\left[\frac{2BY}{d+1} - \frac{x}{3(z+y)} \right]$	$2 * b * y / (d + 1) - x / 3 * (z + y)$

Lecture 6

Control Instructions in C

**CSCMJ101: : Introduction to
Programming using C**

Dr. Vandana Kushwaha

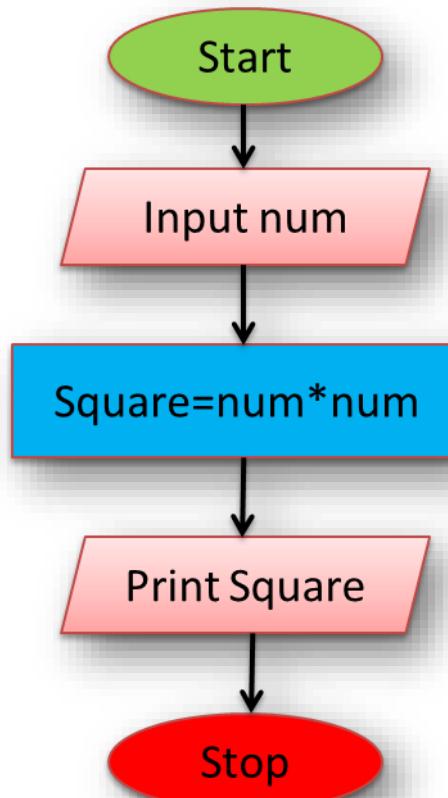
Department of Computer Science
Institute of Science, BHU, Varanasi

Introduction

- The ‘**Control Instructions**’ enable us to specify the **order** in which the **various instructions** in a **program** are to be **executed** by the **computer**.
- In other words the **control instructions** determine the ‘**flow of control**’ in a **program**.
- There are **four types** of **control instructions** in C. They are:
 - (a) Sequence Control Instruction**
 - (b) Selection or Decision Control Instruction**
 - (c) Repetition or Loop Control Instruction**
 - (d) Case Control Instruction**

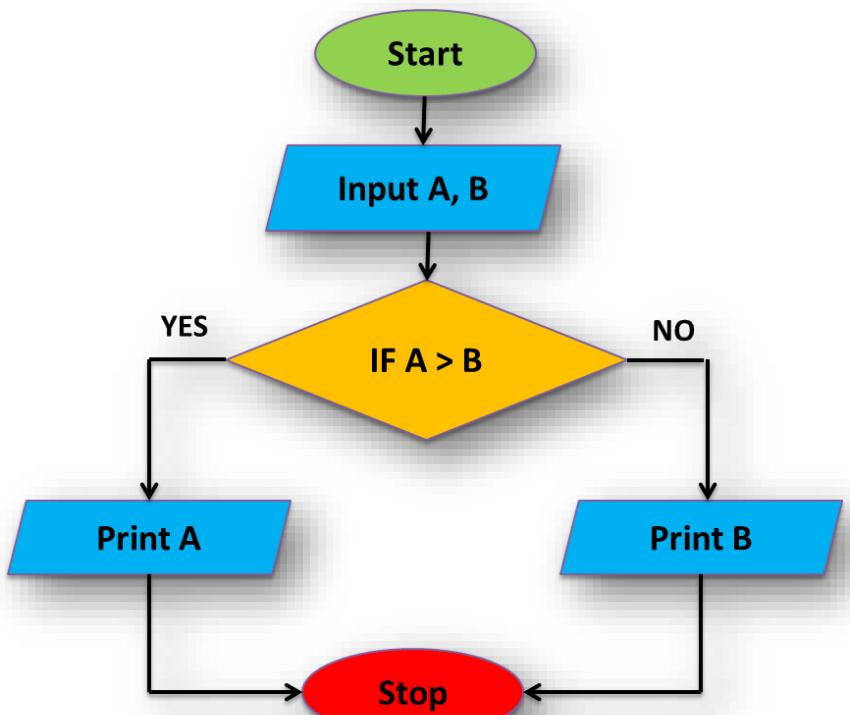
Sequence control instruction

- The **Sequence control instruction** ensures that the **instructions** are **executed** in the **same order** in which they **appear** in the **program**.



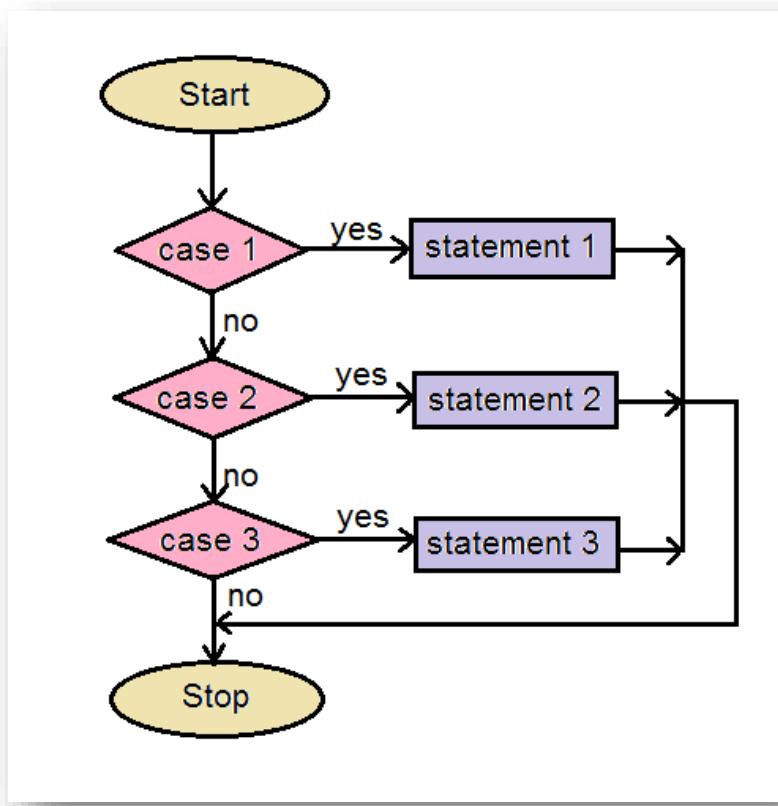
Decision control instructions

- **Decision control instructions** in C are statements that allow programmers to execute specific groups of statements when certain **conditions** are met.



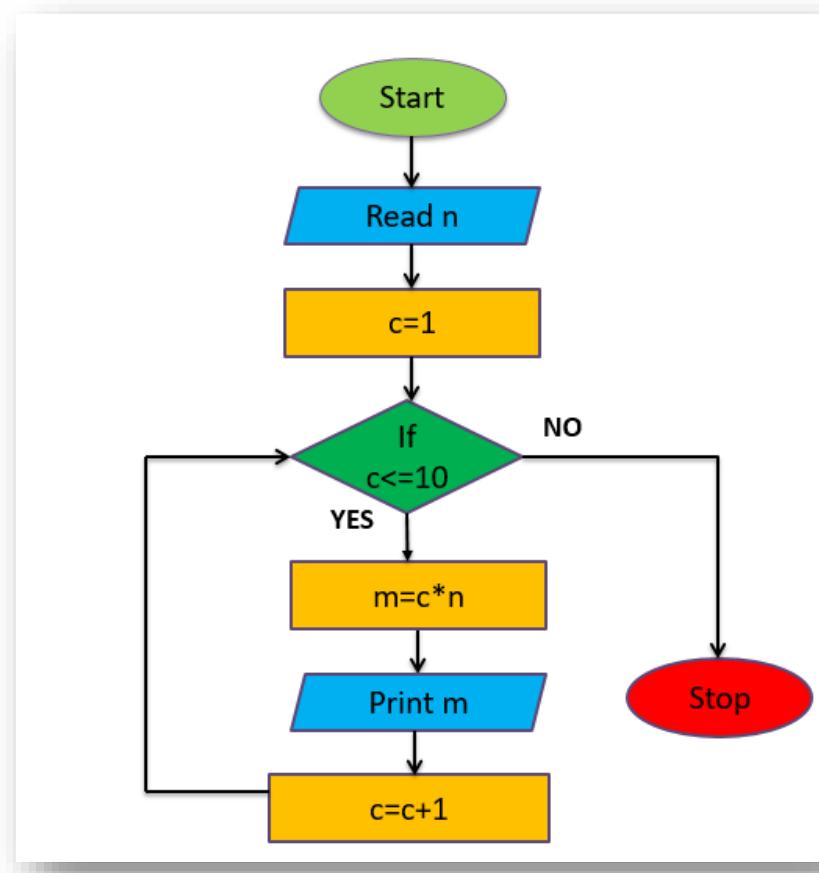
Case control instructions

- A **Case control instruction** in C is a switch statement that allows a program to select between **multiple options** based on the value of an expression.



Loop control instruction

- The **Loop control instruction** helps computer to execute a group of statements repeatedly.



Decision Control instructions

- In any **programming language**, there is a need to **perform different tasks** based on the **condition**.
- For **example**, consider an online website, when you enter wrong id or password it displays error page and when you enter correct credentials then it displays welcome page.
- So there must be a **logic** in place that **checks** the **condition** (id and password) and if the **condition returns true** it **performs a task** (displaying welcome page) **else** it **performs a different task**(displaying error page).
- Using **decision control statements** we can **control** the **flow of program** in such a way so that it executes **certain statements** based on the **outcome** of a **condition** (i.e. **true or false**).

Decision Control instructions

- In C Programming language we have following decision control statements.

1.The **if statement**

2.The **if-else statement**

3.The **else if statement**

The if Statement

- Like most languages, C uses the **keyword if** to implement the **decision control instruction**.
- **Syntax:**

```
if ( condition )
execute this statement ;
```

- The keyword **if** tells the compiler that what follows is a **decision control instruction**.
- The **condition** following the **keyword if** is always **enclosed within a pair of parentheses ()**.
- If the **condition**, whatever it is, is **true**, then the **statement** is **executed**.
- If the **condition** is **not true** then the **statement** is **not executed**; instead the program **skip** these **instruction(s)**.

The if Statement

- We express a **condition** using C's relational operators (==, !=, <, >, <=, >=).
- The **relational operators** allow us to **compare two values** to see whether they are equal to each other, unequal, or whether one is greater than the other.
- /* Demonstration of if statement */

```
void main( )
{
    int num ;
    printf ( "Enter a number less than 10 " ) ;
    scanf ( "%d", &num ) ;
    if ( num <= 10 )
        printf ( "What an obedient person you are !" ) ;
}
```

The if Statement

- One more possible way of writing **if statement** is:

```
if ( expression )
execute this statement ;
```

- Here the **expression** can be **any valid C expression**.
- We can **even use arithmetic expressions** in the if statement.
- Corresponding to each **C expression** one **truth value** is associated either **true** or **false**.
- The **truth value** of an **expression** evaluating to a **nonzero** value is **true**.
- The **truth value** of an **expression** evaluating to a **zero** value is **false**.

The if Statement

- **Example 1** `if (3 + 2 % 5)`
 `printf ("This works") ;`
- **Example 2** `if (a = 10)`
 `printf ("Even this works") ;`
- **Example 3** `if (a = 1-1)`
 `printf ("Even this works") ;`
- **Example 4** `if (-5)`
 `printf ("Surprisingly even this works") ;`

Multiple Statements within *if*

- It may so happen that in a **program** we want **more than one statement** to be **executed** if the **expression** following **if** is satisfied.
- If such **multiple statements** are to be **executed** then **they must be placed** within a **pair of braces** as illustrated in the following example.

```
/* Calculation of bonus */
main( )
{
    int bonus, cy, yoj, yr_of_ser;

    printf ( "Enter current year and year of joining " );
    scanf ( "%d %d", &cy, &yoj );

    yr_of_ser = cy - yoj;

    if ( yr_of_ser > 3 )
    {
        bonus = 2500 ;
        printf ( "Bonus = Rs. %d", bonus );
    }
}
```

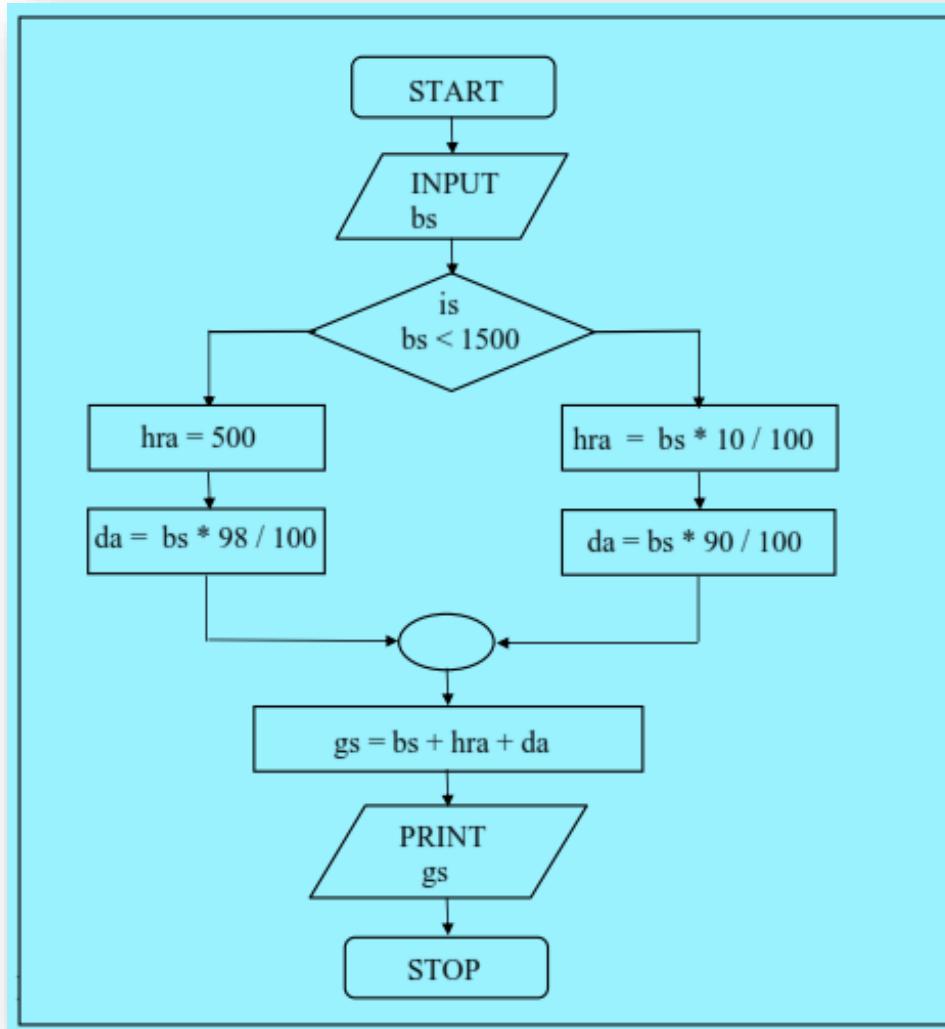
The if-else Statement

- The **if statement** by itself will execute a **single statement**, or a **group of statements**, when the expression following if evaluates to **true**.
- It **does nothing** when the expression evaluates to **false**.
- Can we **execute** another **group of statements** if the **expression** evaluates to **false**?
- Of course! This is what is the purpose of the **else statement** that is demonstrated in the following example:
- **Example:** In a company an employee is paid as under: If his basic salary is less than Rs. 1500, then HRA = 10% of basic salary and DA = 90% of basic salary.
- If his salary is either equal to or above Rs. 1500, then HRA = Rs. 500 and DA = 98% of basic salary. If the employee's salary is input through the keyboard write a program to find his gross salary.

Example:/* Calculation of gross salary */

```
void main( )
{
    float bs, gs, da, hra ;
    printf ( "Enter basic salary " ) ;
    scanf ( "%f", &bs ) ;
    if ( bs < 1500 )
    {
        hra = bs * 10 / 100 ;
        da = bs * 90 / 100 ;
    }
    else
    {
        hra = 500 ;
        da = bs * 98 / 100 ;
    }
    gs = bs + hra + da ;
    printf ( "gross salary = Rs. %f", gs ) ;
}
```

Flowchart:/* Calculation of gross salary */



Nested *if-else*

- If we write an entire **if-else** construct within either the **body** of the **if statement** or the **body** of an **else statement**, this is called '**nesting**' of if.
- **Example:**

```
void main( )
{
    int i ;
    printf ( "Enter either 1 or 2 " ) ;
    scanf ( "%d", &i ) ;
    if ( i == 1 )
        printf ( "You have entered one." ) ;
    else
    {
        if ( i == 2 )
            printf ( "You have entered two." ) ;
        else
            printf ( "You have not followed my instruction !" ) ;
    }
}
```

Forms of *if statement*

if (condition) do this ;	if (condition) { do this ; and this ; }	if (condition) do this ; else do this ;
if (condition) { do this ; and this ; } else { do this ; and this ; }	if (condition) do this ; else { if (condition) do this ; else { do this ; and this ; } }	if (condition) { if (condition) do this ; else { do this ; and this ; } } else do this ;

Use of Logical Operators

- C allows usage of **three logical operators**, namely, **&&**, **||** and **!**.
- These are to be read as '**AND**' '**OR**' and '**NOT**' respectively.
- The first two operators, **&&** and **||**, allow **two or more conditions** to be **combined** in an **if statement**.
- **Example :** The marks obtained by a student in 5 different subjects are input through the keyboard. The student gets a division as per the following rules:

Percentage above or equal to 60 - First division

Percentage between 50 and 59 - Second division

Percentage between 40 and 49 - Third division

Percentage less than 40 - Fail

Write a program to calculate the division obtained by the student.

Program using Nested if-else statements

```
void main( )
{
    int m1, m2, m3, m4, m5, per ;
    printf ( "Enter marks in five subjects " ) ;
    scanf ( "%d %d %d %d %d", &m1, &m2, &m3, &m4, &m5 ) ;
    per = ( m1 + m2 + m3 + m4 + m5 ) / 5 ;
    if ( per >= 60 )
        printf ( "First division " ) ;
    else
    {
        if ( per >= 50 )
            printf ( "Second division" ) ;
        else
        {
            if ( per >= 40 )
                printf ( "Third division" ) ;
            else
                printf ( "Fail" ) ;
        }
    }
}
```

Program using logical operator

```
void main( )
{
    int m1, m2, m3, m4, m5, per ;
    printf ( "Enter marks in five subjects " ) ;
    scanf ( "%d %d %d %d %d", &m1, &m2, &m3, &m4, &m5 ) ;
    per = ( m1 + m2 + m3 + m4 + m5 ) / 5 ;
    if ( per >= 60 )
        printf ( "First division" ) ;
    if ( ( per >= 50 ) && ( per < 60 ) )
        printf ( "Second division" ) ;
    if ( ( per >= 40 ) && ( per < 50 ) )
        printf ( "Third division" ) ;
    if ( per < 40 )
        printf ( "Fail" ) ;
}
```

The else if Statement

- In **else if statement** every **else if** is associated with its **previous if**.
- The **last else** goes to work only if all the conditions fail, which is **optional**.
- Note that the **else if clause** is nothing different, It is just a way of **rearranging** the **else** with the **if** that follows it.
- This would be evident if you look at the following **code**:

```
if ( i == 2 )
    printf ( "With you..." );
else
{
    if ( j == 2 )
        printf ( "...All the time" );
}
```

```
if ( i == 2 )
    printf ( "With you..." );
else if ( j == 2 )
    printf ( "...All the time" );
```

The else if Statement

- There is one more way in which we can write program using else if statement:

```
void main( )
{
    int m1, m2, m3, m4, m5, per ;
    printf ( "Enter marks in five subjects " ) ;
    scanf ( "%d %d %d %d %d", &m1, &m2, &m3, &m4, &m5 ) ;
    per = ( m1+ m2 + m3 + m4+ m5 ) / 5 ;
    if ( per >= 60 )
        printf ( "First division" ) ;
    else if ( per >= 50 )
        printf ( "Second division" ) ;
    else if ( per >= 40 )
        printf ( "Third division" ) ;
    else
        printf ( "fail" ) ;
}
```

The ! Operator

- This **NOT operator (!)** reverses the **result** of the **expression** it operates on.
- For **example**, if the **expression** evaluates to a **non-zero value(true)**, then applying **! operator** to it **results** into a **0(false)**.
- Vice versa, if the **expression** evaluates to **zero(false)** then on applying **! operator** to it makes it **true**, a **non-zero value**.
- **Example:** Consider the following **conditional expression**:
- $!(y < 10)$
- This means “**not y less than 10**”.
- In other words, if y is less than 10, the expression will be **false**, since $(y < 10)$ is **true**.
- We can express the same **condition** as $(y \geq 10)$.

A Word of Caution

```
void main( )
{
    int i ;
    printf ( "Enter value of i " ) ;
    scanf ( "%d", &i ) ;
    if ( i = 5 )
        printf ( "You entered 5" ) ;
    else
        printf ( "You entered something other than 5" ) ;
}
```

And here is the **output** of two runs of this program...

Enter value of i 200

You entered 5

Enter value of i 5

You entered 5

Example: if statement

```
int main()
{
    int x=0;
    if(x)
        printf("Sorry not allowed");
    if(x=0)
        printf("Again not allowed");
    if(x==0)
        printf(" Shall I allow you??");
    return 0;
}
```

Output:

Shall I allow you??

Conditional operator for Decision Control instruction

- Consider an instruction with **conditional operator**:
- **Syntax**
 - **expression 1 ? expression 2 : expression 3**
- Corresponding **instruction** using decision control instruction **if-else** can be written as:

if(expression 1)

expression 2;

else

expression3;

Conditional Operator

- The **Conditional operators** in C language are known by **two more names**
- **Ternary Operator or ? : Operator**
- **Syntax:**
 - **expression 1 ? expression 2: expression 3**
- The first expression (**expression 1**) generally returns either **true** or **false**, based on which it is decided whether (**expression 2**) will be executed or (**expression 3**)
- If (**expression 1**) returns **true** then the expression on the left side of " :" i.e (**expression 2**) is executed.
- If (**expression 1**) returns **false** then the expression on the right side of " :" i.e (**expression 3**) is executed.

Conditional operator

- **Example 1:**
 - int x=7;
 - $y = (x > 5 ? 3 : 4) ;$
 - Here on the right hand side of = there is **expression1** - $x > 5$ which is **true**
 - Thus **expression2** which is **3** will be the final value of variable **y**.
- **Example 2:**
 - int x;
 - $y = (x = 0 ? 3 : 4) ;$
 - Here **expression1** is $x = 0$ which is **false**(in C zero value is equivalent to false).
 - Thus **expression3** which is **4** will be the final value of variable **y**.

Example 1

```
int x, y ;
scanf ( "%d", &x ) ;
y = ( x > 5 ? 3 : 4 ) ;
```

```
int x, y ;
scanf ( "%d", &x ) ;
if ( x > 5 )
    y = 3 ;
else
    y = 4 ;
```

Example 2

```
int i ;
scanf ( "%d", &i ) ;
( i == 1 ? printf ( "Hello!!" ) : printf ( "Good Bye!!" ) ) ;
```

```
int i ;
scanf ( "%d", &i ) ;
if( i == 1)
    printf ( "Hello!!" );
else
    printf ( "Good Bye!!" ) ;
```

Some more examples

```
void main( )
{
int a = 300, b, c ;
if ( a >= 400 )
b = 300 ;
c = 200 ;
printf ( "\n%d %d", b, c ) ;
}
```

Output:
2457 200

```
void main( )
{
int x = 3, y, z ;
y = x = 10 ;
z = x < 10 ;
printf ( "\nx = %d y = %d z = %d", x, y, z ) ;
}
```

Output:
x=10 y=10 z=0

Some more examples

```
int main( )
{
int k = 35 ;
printf ( "\n%d %d %d", k == 35, k = 50, k > 40 ) ;
getch();
return 0;
}
```

Output: 0 50 0

```
int main( )
{
int i=65 ;
char j='A';
if ( i == j )
printf( "C is WOW");
else
printf( "C is a headache" ) ;
getch();
return 0;
}
```

Output:
C is WOW

ASCII Table

0	<u>NUL</u>	16	<u>DLE</u>	32	<u>SP</u>	48	0	64	@	80	P	96	'	112	p
1	<u>SOH</u>	17	<u>DC1</u>	33	!	49	1	65	A	81	Q	97	a	113	q
2	<u>STX</u>	18	<u>DC2</u>	34	"	50	2	66	B	82	R	98	b	114	r
3	<u>ETX</u>	19	<u>DC3</u>	35	#	51	3	67	C	83	S	99	c	115	s
4	<u>EOT</u>	20	<u>DC4</u>	36	\$	52	4	68	D	84	T	100	d	116	t
5	<u>ENQ</u>	21	<u>NAK</u>	37	%	53	5	69	E	85	U	101	e	117	u
6	<u>ACK</u>	22	<u>SYN</u>	38	&	54	6	70	F	86	V	102	f	118	v
7	<u>BEL</u>	23	<u>ETB</u>	39	'	55	7	71	G	87	W	103	g	119	w
8	<u>BS</u>	24	<u>CAN</u>	40	(56	8	72	H	88	X	104	h	120	x
9	<u>HT</u>	25	<u>EM</u>	41)	57	9	73	I	89	Y	105	i	121	y
10	<u>LF</u>	26	<u>SUB</u>	42	*	58	:	74	J	90	Z	106	j	122	z
11	<u>VT</u>	27	<u>ESC</u>	43	+	59	;	75	K	91	[107	k	123	{
12	<u>FF</u>	28	<u>FS</u>	44	,	60	<	76	L	92	\	108	l	124	
13	<u>CR</u>	29	<u>GS</u>	45	-	61	=	77	M	93]	109	m	125	}
14	<u>SO</u>	30	<u>RS</u>	46	.	62	>	78	N	94	^	110	n	126	~
15	<u>SI</u>	31	<u>US</u>	47	/	63	?	79	O	95	_	111	o	127	<u>DEL</u>

Lecture 7

CSCMJ101: : Introduction to Programming using C

Case control structure in C

Dr. Vandana Kushwaha

Department of Computer Science
Institute of Science, BHU, Varanasi

Switch Case instruction

- The **control statement** that allows us to make a **decision** from the **number of choices** is called a **switch**, or more correctly a **switch case-default**, since these **three keywords** go together to make up the **control statement**.
- They most often appear as follows:

```
switch ( integer expression )
{
    case constant1 :
        do this ;
    case constant2 :
        do this ;
    case constant3 :
        do this ;
    default :
        do this ;
}
```

Switch Case instruction

- The **integer expression** following the keyword **switch** is any **C expression** that will yield an **integer value**.
- It could be an **integer constant** like 1, 2 or 3, or an **expression** that evaluates to an **integer**.
- The **keyword case** is followed by an **integer** or a **character constant**.
- Each **constant** in each **case** must be **different** from all the others.
- The “**do this**” lines in the above form of **switch represent** any valid C statement.

Switch Case instruction

- What happens when we run a program containing a **switch**?
- First, the integer expression following the keyword **switch** is evaluated.
- The **value** it gives is then matched, one by one, against the **constant values** that follow the **case statements**.
- When a match is found, the program executes the **statements** following that **case**, and all **subsequent case** and **default statements** as well.
- If no match is found with any of the **case statements**, only the **statements** following the **default** are executed.

Example

```
void main( )
{
    int i = 2 ;
    switch ( i )
    {
        case 1 :
            printf ( "I am in case 1 \n" ) ;
        case 2 :
            printf ( "I am in case 2 \n" ) ;
        case 3 :
            printf ( "I am in case 3 \n" ) ;
        default :
            printf ( "I am in default \n" ) ;
    }
}
```

Output:

```
I am in case 2
I am in case 3
I am in default
```

Example

```
void main( )
{
    int i = 2 ;
    switch ( i )
    {
        case 1 :
            printf ( "I am in case 1 \n" ) ;
            break ;
        case 2 :
            printf ( "I am in case 2 \n" ) ;
            break ;
        case 3 :
            printf ( "I am in case 3 \n" ) ;
            break ;
        default :
            printf ( "I am in default \n" ) ;
    }
}
```

Note:

- The break statement is used inside the switch to terminate a statement sequence.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

Output:

I am in case 2

Example

It is allowed to use **char values** in **case** and **switch** as shown in the following program:

```
void main()
{
    char c = 'x' ;
    switch ( c )
    {
        case 'v' :
            printf ( "I am in case v \n" ) ;
            break ;
        case 'a' :
            printf ( "I am in case a \n" ) ;
            break ;
        case 'x' :
            printf ( "I am in case x \n" ) ;
            break ;
        default :
            printf ( "I am in default \n" ) ;
    }
}
```

Example

At times we may want to execute a common set of statements for multiple **cases**.

```
void main( )
{
    char ch ;
    printf ( "Enter any of the alphabet a, b, or c " ) ;
    scanf ( "%c", &ch ) ;
    switch ( ch )
    {
        case 'a' :
        case 'A' :
            printf ( "a as in apple" ) ;
            break ;
        case 'b' :
        case 'B' :
            printf ( "b as in brain" ) ;
            break ;
        case 'c' :
        case 'C' :
            printf ( "c as in cookie" ) ;
            break ;
        default :
            printf ( "wish you knew what are alphabets" ) ;
    }
}
```

Note:

Once a **case** is satisfied the control simply falls through the **case till** it doesn't encounter a **break statement**.

Output:

Some facts about switch case

- Even if there are multiple statements to be executed in each **case** there is no need to enclose them within a pair of braces (unlike **if**, **and** **else**).
- The disadvantage of **switch** is that one cannot have a case in a **switch** which looks like:

```
case i <= 20 :
```

- All that we can have after the case is an **int constant or a char constant** or an expression that evaluates to one of these constants.
- Even a **float is not allowed**.
- The advantage of **switch over if** is that it leads to a more structured program and the level of indentation is manageable, more so if there are multiple statements within each **case of a switch**.

Program to create a simple calculator

```
#include <stdio.h>
void main()
{
    char operator;
    float n1, n2;
    printf("Enter an operator (+, -, *, /): ");
    scanf("%c", &operator);
    printf("Enter two operands: ");
    scanf("%f %f",&n1, &n2);
    switch(operator)
    {
        case '+':
            printf("%f + %f = %f",n1, n2, n1+n2);
            break;
        case '-':
            printf("%f - %f = %f",n1, n2, n1-n2);
            break;
        case '*':
            printf("%f * %f = %f",n1, n2, n1*n2);
            break;
        case '/':
            printf("%f / %f = %f",n1, n2, n1/n2);
            break;
        default:
            printf("Error! operator is not correct");
    }
}
```

Difference between switch and if

- **if statements** can evaluate **float conditions**; **switch statements** cannot evaluate **float conditions**.
- **if statement** can evaluate **relational operators**; **switch statement** cannot evaluate **relational operators** i.e. they are not allowed in **switch statement**.

Example

```
#include <stdio.h>
void main()
{
    char ch;
    printf("Enter an alphabet: ");
    scanf("%c",&ch);
    switch(ch)
    {
        case 'A':
        case 'E':
        case 'I':
        case 'O':
        case 'U':
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            printf("%c is a VOWEL.\n",ch);
            break;
        default:
            printf("%c is a CONSONANT.\n",ch);
    }
}
```

Lecture 8

CSCMJ101: : Introduction to Programming using C

Loop Control Structure in C

Dr. Vandana Kushwaha

Department of Computer Science
Institute of Science, BHU, Varanasi

Introduction

- Sometimes we need to perform a set of instructions **repeatedly**.
- This involves **repeating some portion** of the program either a **specified number of times** or until a **particular condition** is being **satisfied**.
- This **repetitive operation** is done through a **loop control instruction**.
- There are **three methods** by way of which we can repeat a part of a program. They are:
 - (a) For loop*
 - (b) While loop*
 - (c) Do-while loop*

Types of Loop

- There are mainly **two types of loops**:
- **Entry Controlled loops:**
 - In this type of loops the **test condition** is tested **before entering the loop body**.
 - **For Loop** and **While Loop** are **entry controlled loops**.
- **Exit Controlled Loops:**
 - In this type of loops the **test condition** is tested or evaluated at the **end of loop body**.
 - Therefore, the **loop body** will execute **at least once**, irrespective of whether the **test condition** is **true** or **false**.
 - **Do – while loop** is **exit controlled loop**.

Types of Loop

Loops

Entry Controlled

for

```
for( initialization ; condition; updation)  
{  
}
```

while

```
while( condition )  
{  
}
```

Exit Controlled

do-while

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

The While Loop

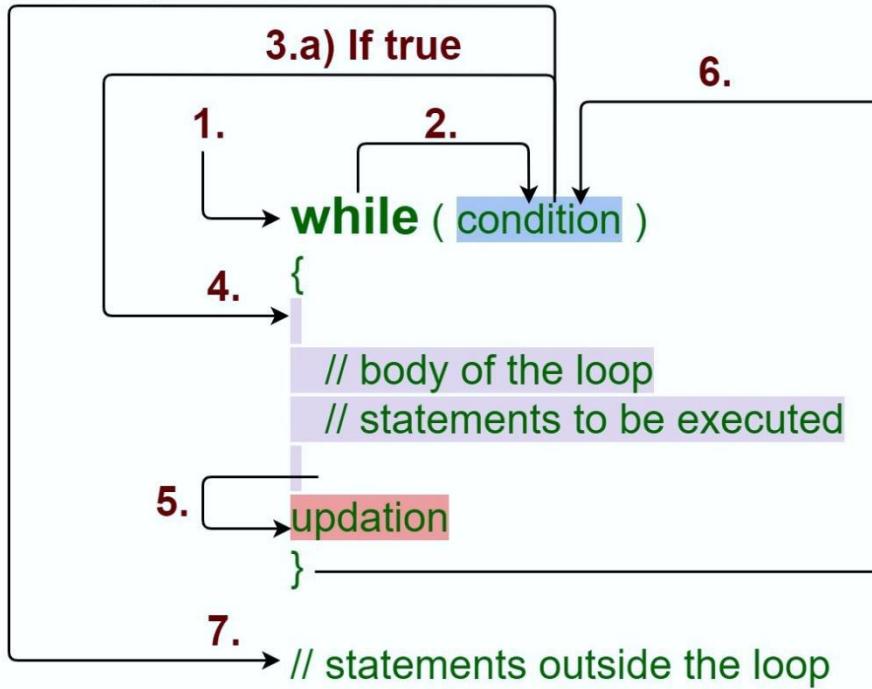
- It is often the case in programming that we want to do something a **fixed number of times.**
- Perhaps we want to calculate gross salaries of ten different persons, or we want to convert temperatures from Centigrade to Fahrenheit for 15 different cities.
- The **while loop** is ideally suited for such cases.
- **Syntax:**

```
initialize counter variable;  
while(test_expression)  
{  
    //body of the loop;  
    statement to be executed;  
    update counter variable;  
}
```

The While Loop

While Loop

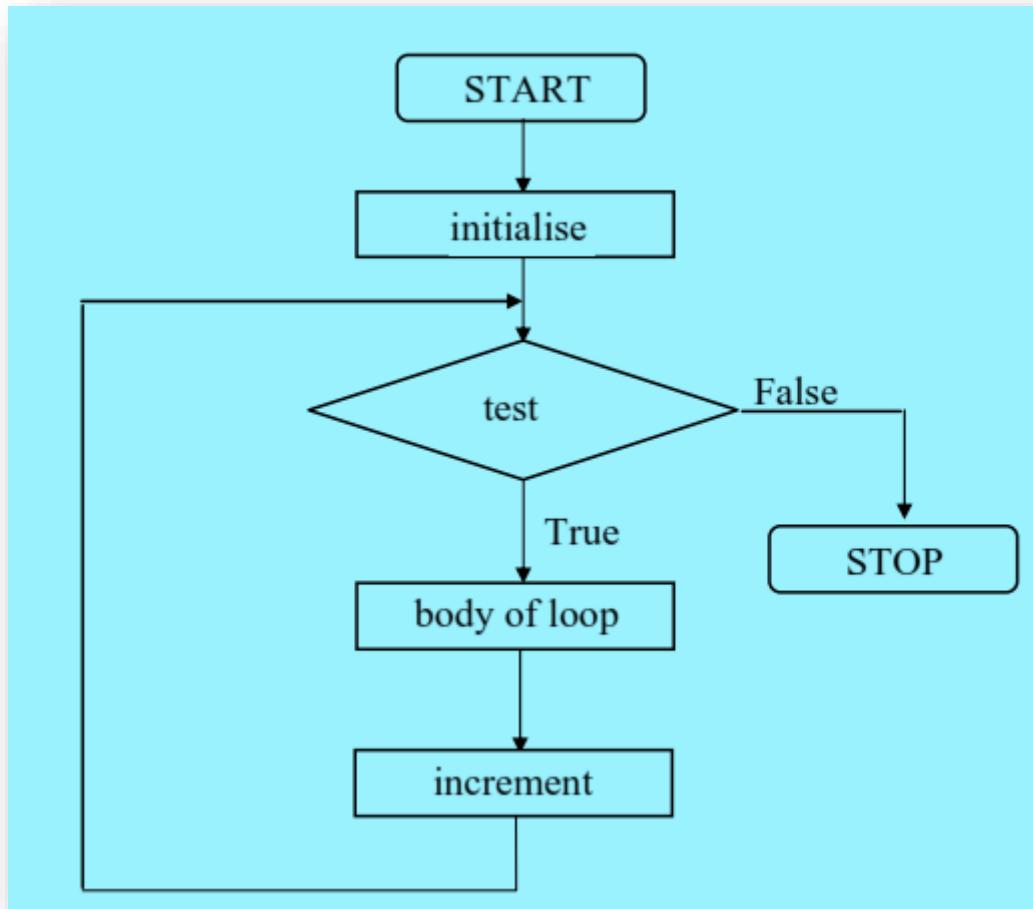
3.b) If false



How while loop works?

1. The **while loop** evaluates the **test expression** inside the parenthesis () .
2. If the **test expression** is **true**, statements inside the **body** of **while loop** are **executed**.
3. After executing the **loop body**, **counter variable** is **updated(increments/decrements)**.
4. Then, the **test expression** is **evaluated** again.
5. The process goes on until the **test expression** is evaluated to **false**.
6. If the **test expression** is **false**, the **loop terminates** (ends) and control goes to next instruction after the loop(if any).

While loop Flowchart



The While Loop

- While loop having **single statement** in its body:

```
while( i <= 10 )  
    i = i + 1;
```

OR

```
while( i <= 10)  
{  
    i = i + 1 ;  
}
```

Infinite Loop

- As a rule the **while** must **test** a **condition** that will eventually become **false**, otherwise the **loop** would be **executed forever, indefinitely**.
- Following is **an indefinite/infinite loop**, since **i** remains equal to **1** forever.

```
void main( )
{
    int i = 1 ;
    while (i <= 10 )
        printf ( "%d\n", i ) ;
}
```



Infinite loop

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

finite loop

The While Loop

- Instead of **incrementing** a **loop counter**, we can even **decrement** it and still manage to get the body of the loop executed **repeatedly**. This is shown below:

```
void main( )
{
    int i = 5 ;
    while (i>=1 )
    {
        printf ( "\n Welcome") ;
        i=i-1;
    }
}
```

- It is not necessary that a **loop counter** must only be an **int**. It can even be a **float**.

What will be the output??

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



```
main( )  
{  
    int i = 1 ;  
    while (i <= 10 )  
    ; //Null Statement  
    {  
        printf ( "%d\n", i ) ;  
        i=i+1;  
    }  
}
```

Infinite loop

Example: print even numbers<10

```
void main()
{
    int n, i=2;

    printf("Even numbers less than 10 are: ");

    while(i < 10)
    {
        printf("%d \n", i);

        i=i+2;
    }
}
```

OUTPUT:

```
Even numbers less than 10 are:
2
4
6
8
```

Example: Multiplication table

```
void main()
{
    int n, i=1;
    printf("Enter an integer: ");
    scanf("%d", &n);
    while(i <= 10)
    {
        printf("%d * %d = %d \n", n, i, n * i);
        i=i+1;
    }
}
```

OUTPUT:

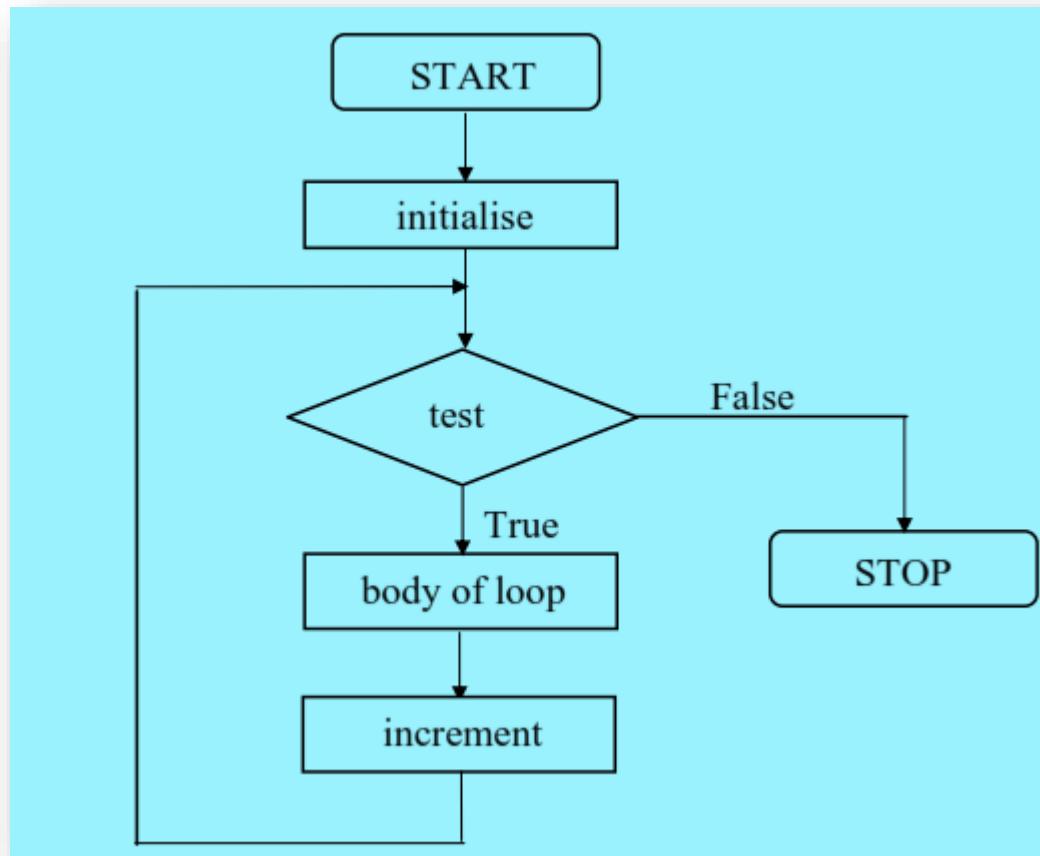
```
Enter an integer: 5
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

For loop

- **For loop** is the **compact** form of **while loop**.
- **Syntax:**

```
for (initialize counter variable; test_expression; update counter variable )  
{  
    //body of the loop  
    statement to be executed;  
    statement to be executed;  
}
```

Flow chart: for loop



For loop

- There are **three parts** of a **for loop** statement:
- **Initialization Expression:**
 - In this expression we have to **initialize the loop counter** to some value.
 - For example: **int i=1;**
- **Test Expression:**
 - In this expression we have to **test the condition**.
 - If the **condition** evaluates to **true** then we will execute the **body of loop** and go to **update expression** otherwise we will **exit** from the **for loop**.
 - For example: **i <= 10;**

For loop

- **Update Expression:**
- After executing **loop body** this expression **increments/decrements** the **loop counter variable** by some value.
- For **example**: **i++ or i-- etc.**

For loop

For Loop

3.b) If false

3.a) If true

1. 2.

6.

4. for (initialization ; condition ; updation)

{

// body of the loop

// statements to be executed

}

5.

7. → // statements outside the loop

Example: print numbers from 1 to 10

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

```
void main( )  
{  
    int i ;  
    for ( i = 0 ; i <= 9 ; i = i + 1 )  
        printf ( "%d", i+1 ) ;  
}
```

```
void main( )  
{  
    int i ;  
    for ( i = 1 ; i < 11 ; i = i + 1 )  
        printf ( "%d", i ) ;  
}
```

Output:

1 2 3 4 5 6 7 8 9 10

Nesting of Loops

- The way **if statements** can be **nested**, similarly **while** and **for** can also be **nested**.

```
main( )
```

```
{  
int r, c, sum ;  
  
for ( r = 1 ; r <= 3 ; r++ ) /* outer loop */  
{  
    for ( c = 1 ; c <= 2 ; c++ ) /* inner loop */  
    {  
        sum = r + c ;  
  
        printf ( "r = %d c = %d sum = %d\n", r, c, sum ) ;  
    }  
}  
}
```

OUTPUT:

```
r = 1 c = 1 sum = 2  
r = 1 c = 2 sum = 3  
r = 2 c = 1 sum = 3  
r = 2 c = 2 sum = 4  
r = 3 c = 1 sum = 4  
r = 3 c = 2 sum = 5
```

Different forms of 'for' statement

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

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

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

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

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

Multiple Initialisations in the *for Loop*

- The **initialisation expression** of the **for loop** can contain more than one statement separated by a **comma**.
- **For example:**

```
for ( i = 1, j = 2 ; j <= 10 ; j++ )
```

- **Multiple statements** can also be used in the **updation expression** of **for loop**; i.e., we can increment (or decrement) two or more variables at the same time.
- However, **only one expression** is allowed in the **test expression**.
- This **expression** may contain **several conditions** linked together using **logical operators**.

Program to print sum of first ten natural numbers

```
void main()
{
int j, sum = 0;
printf("The first 10 natural number is :\n");

for (j = 1; j <= 10; j++)
{
printf("%d ",j);
sum = sum + j;
}

printf("\nThe Sum is : %d\n", sum);
}
```

OUTPUT:

The first 10 natural number is :

1 2 3 4 5 6 7 8 9 10

The Sum is : 55

Sum of digits program in C

```
void main()
{
int n,sum=0,m;
printf("Enter a number:");
scanf("%d",&n);

while(n>0)
{
m=n%10;
sum=sum+m;
n=n/10;

}

printf("Sum is=%d",sum);
}
```

OUTPUT:

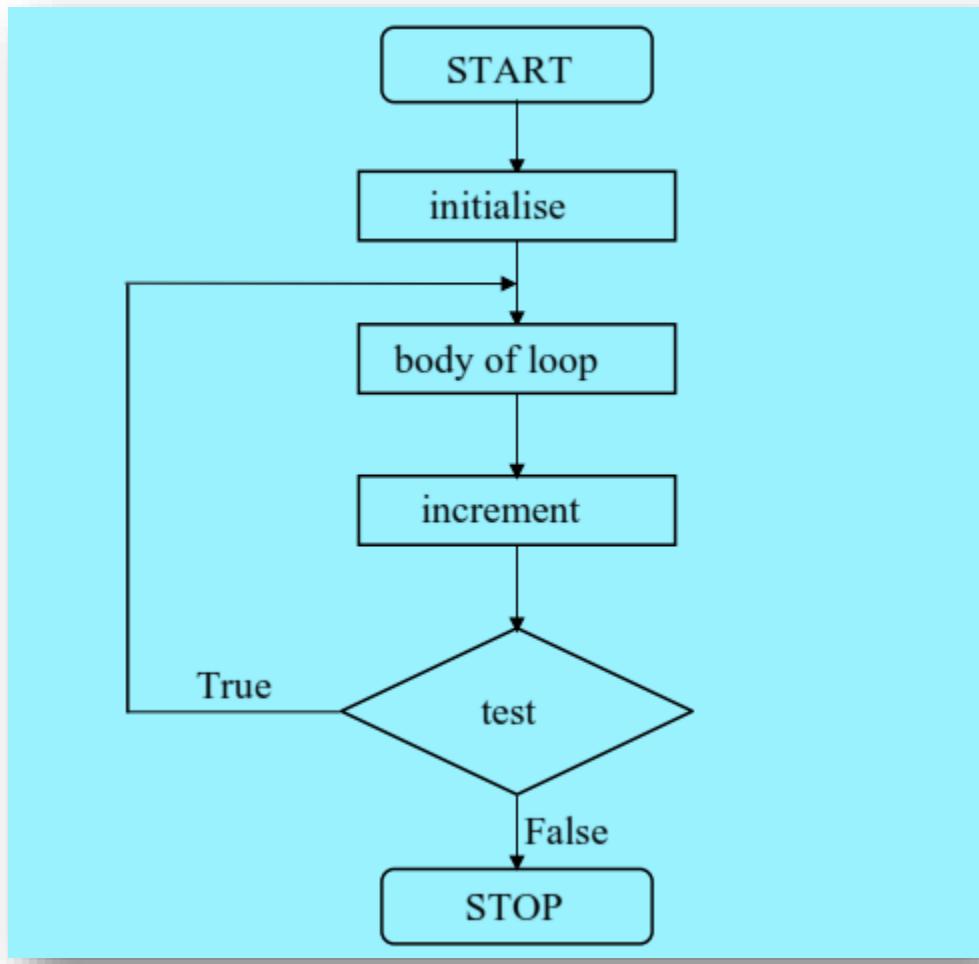
```
Enter a number:654
Sum is=15
```

Do while loop

- In **do while loop** the **condition** is **tested** at the **end of loop body**, i.e **do while loop** is **exit controlled loop**.
- In **do while loop** the **loop body** will **execute at least once** irrespective of **test condition**.
- **Syntax:**

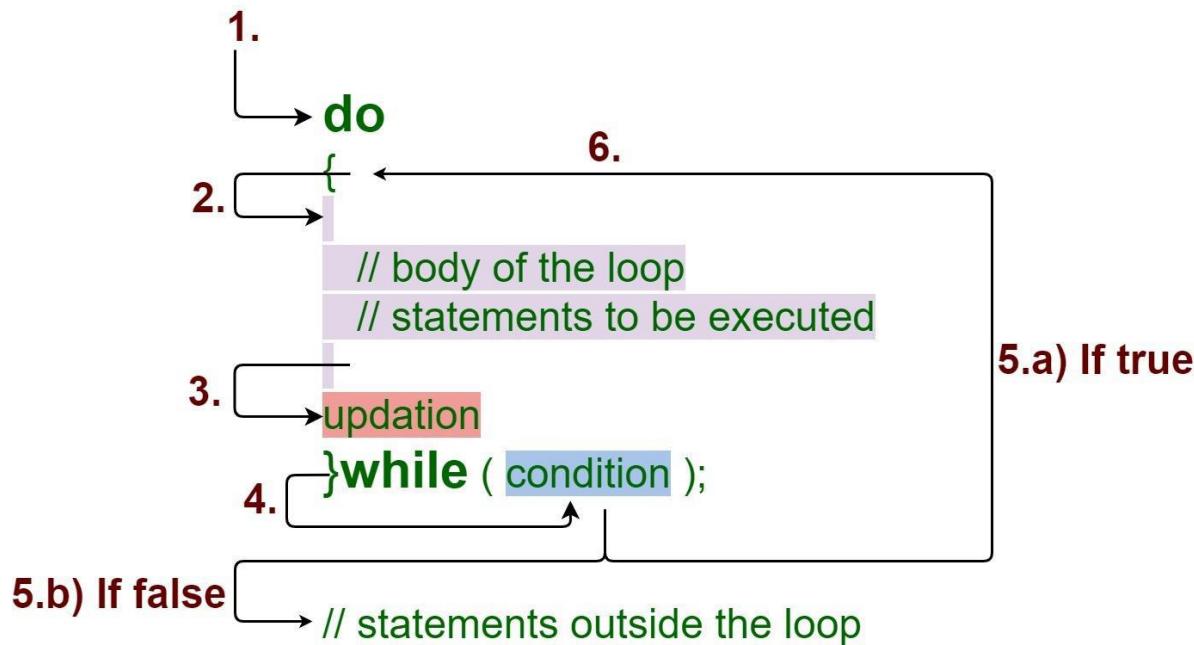
```
initialization expression;  
do  
{  
// statements  
update_expression;  
} while (test_expression);
```

Do while loop



Do while loop

Do - While Loop



Example

```
#include<stdio.h>
int main()
{
    int num=1;
    do
    {
        printf("%d\n",2*num);
        num++;
    }while(num<=10);
    return 0;
}
```

OUTPUT:

```
2
4
6
8
10
12
14
16
18
20
```

Example

```
/* Execution of a loop an unknown number of times */  
main( )  
{  
    char another ;  
    int num ;  
    do  
    {  
        printf ( "Enter a number " ) ;  
        scanf ( "%d", &num ) ;  
        printf ( "square of %d is %d", num, num * num ) ;  
        printf ( "\nWant to enter another number y/n " ) ;  
        scanf ( " %c", &another ) ;  
    } while ( another == 'y' ) ;  
}
```

OUTPUT:

Enter a number 5

square of 5 is 25

Want to enter another number y/n y

Enter a number 7

square of 7 is 49

Want to enter another number y/n n

C program to find factorial of a number

```
void main()
{
    int num,i;
    long int fact=1;
    printf("Enter a nonzero integer number: ");
    scanf("%d",&num);
    for(i=num; i>=1; i--)
        fact=fact*i;
    printf("\nFactorial of %d is = %ld",num,fact);
}
```

OUTPUT:

```
Enter a nonzero integer number: 8
Factorial of 8 is = 40320
```

C program to print pattern

```
void main()
{
int i,j;
int n=5;
for(i=0; i<n; i++)
{
for(j=0;j<=i;j++)
{
printf("*");
}
printf("\n");
}
}
```

OUTPUT

```
*
```

```
**
```

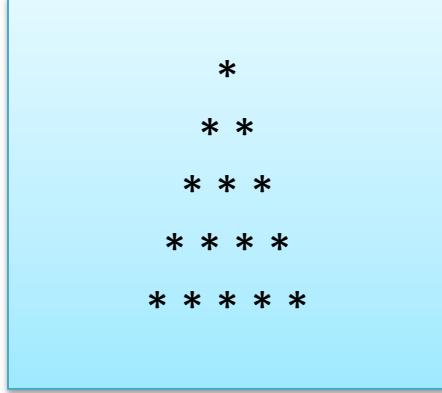
```
***
```

```
****
```

```
*****
```

C program to print pattern

```
main()
{
int i,j,space=4;
for(i=0;i< 5;i++)
{
/*loop for initially space, before star printing*/
for(j=0;j< space;j++)
{
printf(" ");
}
for(j=0;j<=i;j++)
{
printf("* ");
}
printf("\n");
space--;/ * decrement one space after one row*/
}
}
```



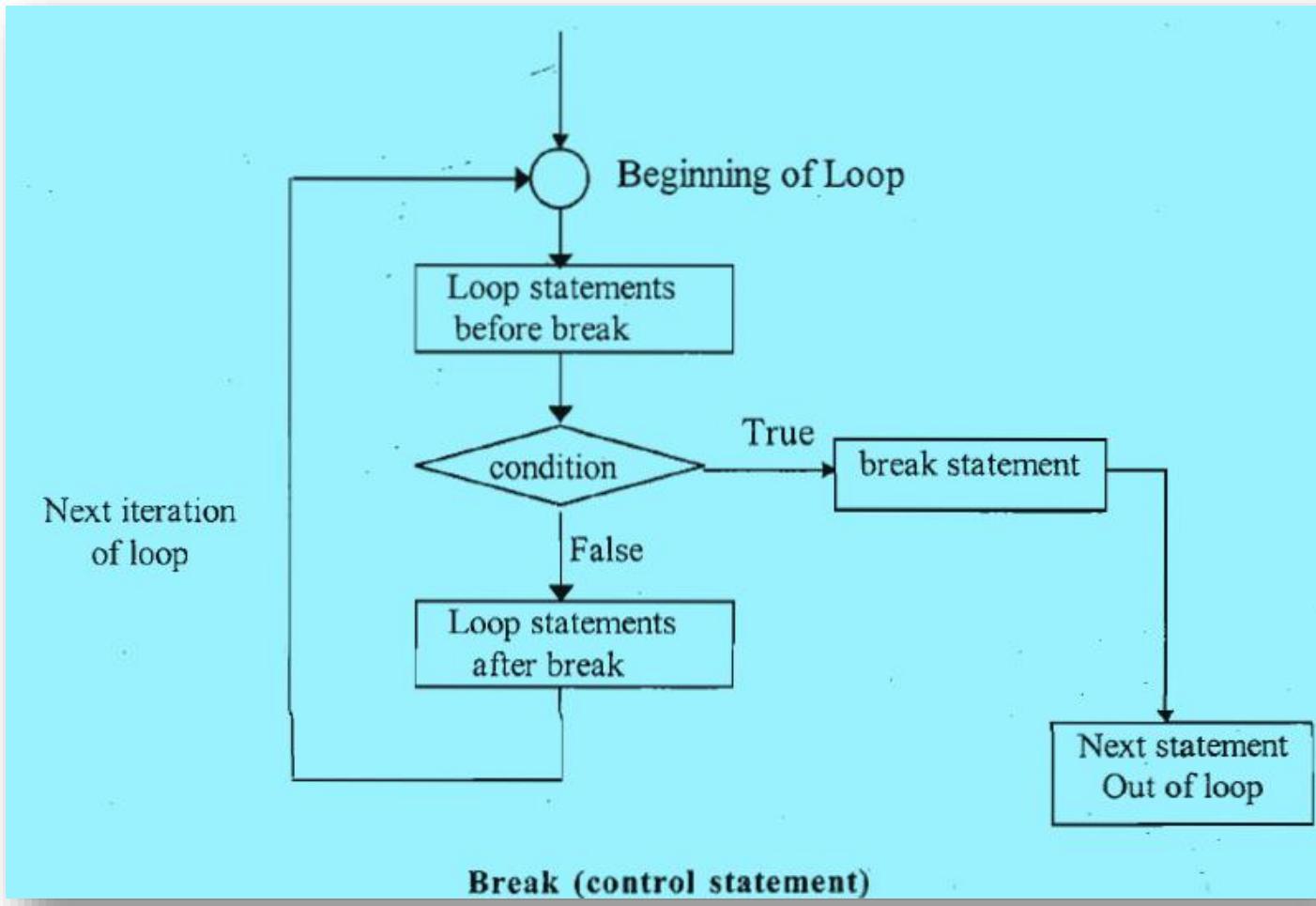
```
*  
* *  
* * *  
* * * *  
* * * * *
```

The *break Statement*

- We often come across situations where we want to **jump out** of a **loop instantly**, without waiting to get back to the conditional test.
- The keyword **break** allows us to do this.
- When **break** is encountered inside any loop, control automatically passes to the first statement after the loop.
- A **break** is usually associated with an **if**.
- Example:

```
for(initialization ; condition ; updation)
{
    ....;
    if(condition)
        break;
    ....;
}
```

The *break* Statement



Example 1

```
void main( )
{
    int i;
    for ( i = 1 ; i <= 10 ; i++ )
    {
        if ( i ==5 )
            break;
        printf("\n%d", i) ;
    }
}
```

OUTPUT:

```
1  
2  
3  
4
```

Example 2

```
void main( )
{
    int i, j;
    for ( i = 1 ; i <= 2 ; i++ )
    {
        for ( j = 1 ; j <= 2 ; j++ )
        {
            if ( i == j )
                break;
            printf ( "\n%d %d\n", i, j );
        }
    }
}
```

OUTPUT:

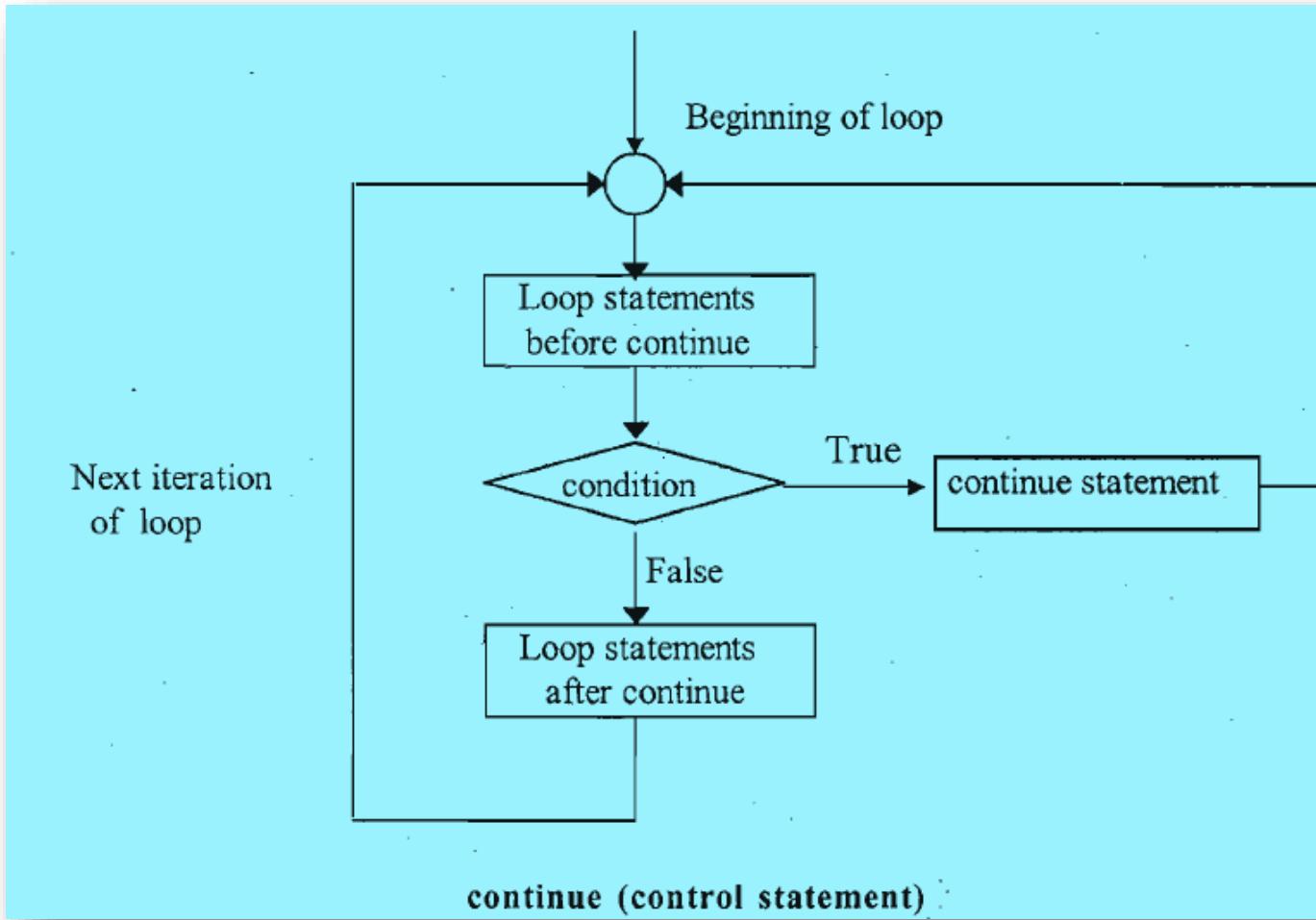
2 1

The *continue* Statement

- In some programming situations we want to take the **control** to the **beginning of the loop**, bypassing the statements inside the loop, which have not yet been executed.
- The keyword **continue** allows us to do this.
- When **continue** is encountered inside any loop, control automatically passes to the next iteration of the loop after skipping the instructions written just after the **continue** statement.
- A **continue** is usually associated with an **if**.
- Example:

```
for(initialization ; condition ; updation)
{
    ....;
    if(condition)
        continue;
    ....;
}
```

The *continue* Statement



Example 3

```
void main( )
{
    int i;
    for ( i = 1 ; i <= 10 ; i++ )
    {
        if ( i ==5 )
            continue;
        printf("\n%d", i) ;
    }
}
```

OUTPUT:

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

Example 4

```
main( )
{
    int i, j ;
    for ( i = 1 ; i <= 2 ; i++ )
    {
        for ( j = 1 ; j <= 2 ; j++ )
        {
            if ( i == j )
                continue ;
            printf ( "\n%d %d\n", i, j ) ;
        }
    }
}
```

Output:
1 2
2 1

CSCMJ101: : Introduction to Programming using C

Lecture 9

Function in C

Dr. Vandana Kushwaha

Department of Computer Science
Institute of Science, BHU, Varanasi

Introduction

- A **function** is a **group of statements** that together perform a **specific task**.
- A **function** is a set of statements that **take inputs**, do some specific **computation** and **produces output**.
- The **idea** is to put some commonly or **repeatedly done task together** and make a **function** so that instead of writing the same code again and again for different inputs, we can **call the function**.
- Every C program has at least one function, which is **main()**, other than that a program can define additional functions.

Advantage of functions in C

- Generally a **complex problem** is **divided** into **sub problems** and then **solved**.
- This **divide and conquer technique** is **implemented** in C through **functions**.
- A **program** can be **divided** into **functions**, each of which **performs** some specific **task**.
- So the use of C **functions modularizes** and **divides** the **work** of a **program**.
- When **some specific code** is to be **used more than once** and at **different places** in the **program** the use of **functions** avoids **repetition** of that **code**.
- The **program** becomes **easily understandable**, **modifiable** and **easy to debug** and **test**.
- **Functions** can be stored in a **library** and **reusability** can be achieved.

Example

```
main( )
{
    message( ) ; //function call
    printf ( "\nCry, and you stop the monotony!" ) ;
}
```

```
void message( ) //function definition
{
    printf ( "\nSmile, and the world smiles with you..." ) ;
}
```

OUTPUT:

Smile, and the world smiles with you...

Cry, and you stop the monotony!

Example

- Here, **main()** itself is a **function** and through it we are calling the function **message()**.
- When **main()** ‘calls’ the function **message()** the **control** passes to the function **message()**.
- The activity of **main()** is **temporarily suspended**; it falls asleep while the **message()** function wakes up and goes to work.
- When the **message()** function runs out of statements to execute, the **control returns to main()**, which comes to life again and begins executing its code at the exact point where it left off.
- Thus, **main()** becomes the ‘**calling**’ function, whereas **message()** becomes the ‘**called**’ function.

Calling more than one function

```
main( )
{
    printf( "\nI am in main" );
    italy( );
    brazil( );
    argentina( );
}
```

```
void italy( )
{
    printf( "\nI am in italy" );
}
```

```
void brazil( )
{
    printf( "\nI am in brazil" );
}
```

```
void argentina( )
{
    printf( "\nI am in argentina" );
}
```

OUTPUT:

I am in main

I am in italy

I am in brazil

I am in argentina

One function can call another function

```
main()
{
    printf ( "\nI am in main" ) ;
    italy( ) ;
    printf ( "\nI am finally back in main" ) ;
}

void italy( )
{
    printf ( "\nI am in italy" ) ;
    brazil( ) ;
    printf ( "\nI am back in italy" ) ;
}

void brazil( )
{
    printf ( "\nI am in brazil" ) ;
    argentina( ) ;
}

void argentina( )
{
    printf ( "\nI am in argentina" ) ;
}
```

OUTPUT:

I am in main
I am in italy
I am in brazil
I am in argentina
I am back in italy
I am finally back in main

Facts about Function

- C program is a **collection of one or more functions**.
- A **function gets called** when the **function name** is followed by a **semicolon**.

```
main( )  
{  
    argentina( );  
}
```

- A **function is defined** when **function name** is followed by a **pair of braces** in which one or more statements may be present.

```
return_type argentina()  
{  
    statement 1 ;  
    statement 2 ;  
    statement 3 ;  
}
```

Facts about Function

- A **function** can be **called** any number of times.

```
main( )
{
    message( );
    message( );
}

void message( )
{
    printf ( "\nJewel Thief!!" );
}
```

Facts about Function

- The **order** in which the **functions are defined** in a **program** and the **order** in which they **get called** need not necessarily be same.

```
main( )
{
    message1();
    message2();
}

void message2( )
{
    printf ( "\nBut the butter was bitter" );
}

void message1( )
{
    printf ( "\nMary bought some butter" );
}
```

Facts about Function

- A **function** can **call itself**, such a process is called ‘**recursion**’.
- A **function** can be **called from other function**, but a **function cannot be defined in another function**.
- Thus, the following **program code** would be **wrong**, since **argentina()** is being defined inside another function, **main()**.

```
main( )
{
    printf ( "\nI am in main" ) ;
    void argentina( )
    {
        printf ( "\nI am in argentina" ) ;
    }
}
```

Types of Function

- There are basically **two types of functions**:
 - **Library functions** Ex. `printf()`, `scanf()`, `sqrt()` etc.
 - **User-defined functions** Ex. `argentina()`, `brazil()` etc.
- **Library functions** are commonly required functions grouped together and stored in what is called a **Library(header file)**.
- This **library of functions** is present on the disk and is written for us by people who write **compilers** for us.
- Almost always a **compiler** comes with a **library of standard functions**.
- The **procedure of calling** both types of functions is exactly same.

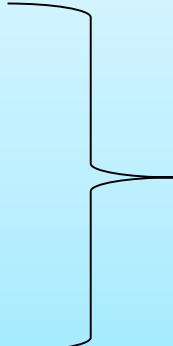
User Defined Functions

- Users can **create** their own **functions** for **performing** any **specific task** of the program.
- These types of **functions** are called **user-defined functions**.
- To create and use these **functions**, we should know about these **three** things-
 - 1. Function Definition .***
 - 2. Function Declaration/Prototype***
 - 3. Function Call***

Function Definition

- The **function definition** consists of the **whole description** and **code** of a function.
- It tells **what the function is doing** and what are its **inputs** and **outputs**.
- A **function definition** consists of two parts - a **function header** and a **function body**.
- The **general syntax** of a **function definition** is:

```
return type func_name( type1 arg1, type2 arg2,..... )  
 {  
 local variables declarations;  
 statement;  
 return(expression) ;  
 }
```



The code is annotated with yellow boxes and curly braces. The first line 'return type func_name(type1 arg1, type2 arg2,.....)' is labeled 'function header'. The entire block from '{' to '}' is labeled 'function body'.

Function Definition

- The **return_type** denotes the **type** of the **value** that will be returned by the function.
- A function can return either **one value or no value**.
- If a function does not return any value then **void** should be written in place of **return type**.
- **func_name** specifies the **name of the function** and it can be **any valid C identifier**.
- After function name the **argument declarations** are given in **parentheses**, which mention the **type** and **name** of the **arguments**.
- These are known as **formal arguments** and used to **accept values**.
- A function can take **any number of arguments** or even **no argument** at all.
- If there are **no arguments** then either the **parentheses can be left empty** or **void** can be written inside the parentheses.

Body of the Function

- The **body of function** consists of **declarations of variables** and **C statements** followed by an **optional return statement**.
- The **variables declared inside the function** are known as **local variables**, since they are **local** to that **function only**, i.e. they have existence only in the function in which they are declared, they can not be used anywhere else in the program.
- The **return statement** is **optional**, It may be **absent** if the **function does not return any value**.
- Note that a **function definition cannot be placed inside another function definition**.
- **Function definitions** can also be **placed in different files**.

Function Definition

- The **function definition** can be placed anywhere in the **program**, but generally all **definitions** are **placed after the main() function**.
- **Example:**

```
void drawline (void)
{
    int i;
    for(i=1;i<80;i++)
        printf ("--");
}
```

- Here the **function is not returning** any value so **void** is written at the place of **return_type**, and since it does not accept any **arguments** so **void** is written inside parentheses.
- The **int variable** is declared inside the **function body** so it is a **local variable** and can be used inside this function only.

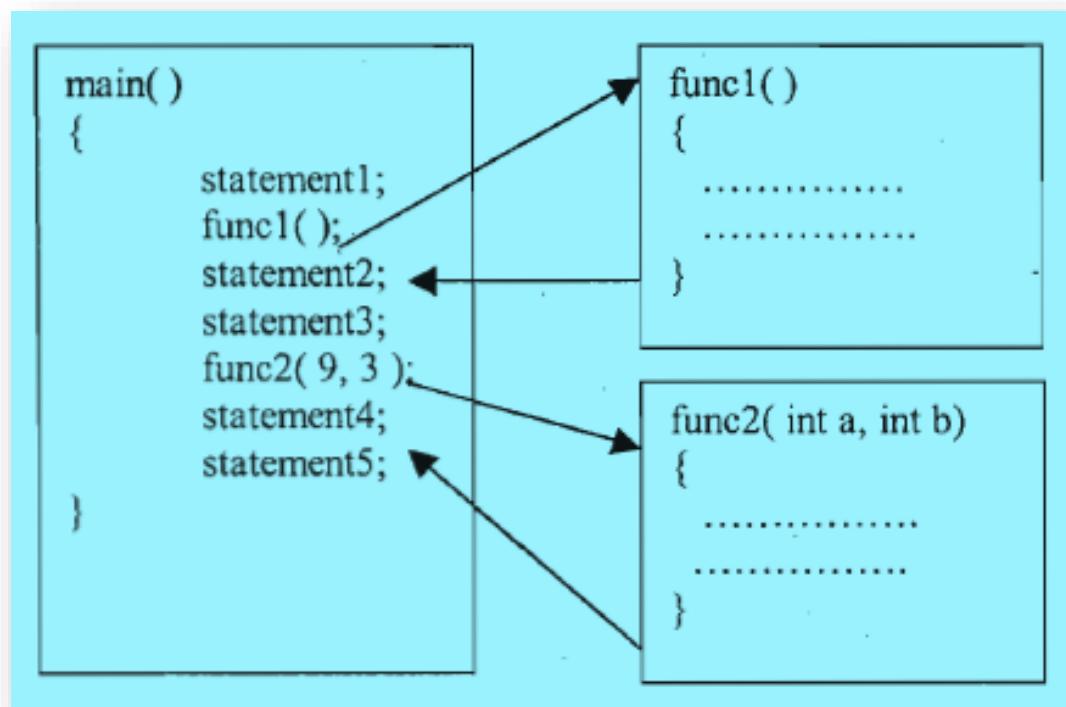
Function Call

- The **function definition** describes **what a function can do**, but to actually **use** it in the program the **function should be called** somewhere.
- A **function** is **called** by simply **calling its name followed by the argument list (if any) inside the parentheses followed by a semicolon(;)**.

```
func_name(arg1, arg2, arg3 ...);
```

- These **arguments** **arg1, arg2, ,...are called actual arguments.**
- Here **func_name** is known as the **called function** while the function in which this function call is placed is known as the **calling function**.
- When a **function** is **called**, the **control passes** to the **called function**, which is executed and after this , the control is transferred to the statement following the function call in the **calling function**.

Transfer of control when function is called



Function Declaration/Prototype

- The **calling function** needs information about the **called function**.
- If definition of the called function placed before the calling function, then declaration is not needed.
- But generally the function **main()** is placed at the top and all other functions are placed after it.
- In this case, function declaration is needed.
- The **function declaration** is also known -as the **function prototype**, and it informs the **Compiler** about the following **three** things:
 1. Name of the function
 2. Number and type of argument it received by the function.
 3. Type of value returned by the function.

Function Declaration/Prototype

- The general **syntax** of a **function declaration** is:

```
return_type func_name(type1 arg1, type2 arg2, .., .....);
```

- This looks just like the **header of function definition**, except that there is a **semicolon** at the end.
- The **names of the arguments** while declaring a function are **optional**.
- These are only used for descriptive purposes.
- So we can write the declaration in this way also :

```
return_type func_name(type1 , type2 , .. , .....);
```

Return statement

- The **return statement** is used in a function to **return a value** to the calling function.
- It may also be used for immediate **exit from the called function to the calling function** without returning a value.
- This statement can appear anywhere inside the body of the function.
- There are **two ways** in which it can be used-

```
return;  
return ( expression );
```

- The first form of **return statement** is used to terminate the function without returning any value, in this case only **return keyword** is written.
- We can use multiple return statements in a function but as soon as first return statement is encountered the function terminates and all the statements following it are not executed i.e. **function can return only one value at a time**.

Function Arguments

- The calling function sends some values to the called function for communication; these values are called **arguments** or **parameters**.
- **Actual arguments**
- The **arguments** which are mentioned in the **function call** are known as **actual arguments**, since these are the values which are *actually sent* to the **called function**.
- **Actual arguments** can be written in the form of variables, constants or expressions or any function call that returns a value, For example-

func (x);

func(a*b, c*d+k);

func(22, 43);

func(1, 2, sum(a, b));

Function Arguments

- **Formal arguments**
- The name of the **arguments**, which are mentioned in the **function definition** are called **formal arguments** since they are used just to hold the values that are sent by the calling function.
- These, **formal arguments** are simply like other **local variables** of the function which are created when the function call starts and are destroyed when the function ends.
- However there are **two differences**, First is that **formal arguments** are **declared inside parentheses** while other local variables are declared at the beginning of the function block.
- The second difference is that **formal arguments** are **automatically initialized** with the values of the **actual arguments** passed, while other local variables are assigned values through the statements written inside the function body.

Example

```
int calsum (int x , int y , int z );
main( )
{
    int a, b, c, sum ;
    printf ( "\nEnter any three numbers " ) ;
    scanf ( "%d %d %d", &a, &b, &c ) ;
    sum = calsum ( a, b, c ) ;
    printf ( "\nSum = %d", sum ) ;
}

Int calsum (int x , int y , int z )
{
    int d ;
    d = x + y + z ;
    return ( d ) ;
}
```

OUTPUT:

```
Enter any three numbers 10 20 30
Sum = 60
```

Passing Values between Functions

- In this program, from the function **main()** the values of **a**, **b** and **c** are passed on to the function **calsum()**, by making a call to the function **calsum()** and mentioning **a**, **b** and **c** in the parentheses:

sum = calsum (a, b, c);

- In the **calsum()** function these values get collected in three variables **x**, **y** and **z**:
- calsum (int x, int y, int z)**
- The variables **a**, **b** and **c** are called ‘**actual arguments**’, whereas the variables **x**, **y** and **z** are called ‘**formal arguments**’.
- **Any number of arguments can be passed to a function being called.**
- However, the type, order and number of the actual and formal arguments must always be same.

Passing Values between Functions

- Instead of using **different variable** names **x, y** and **z**, we could have used the **same variable** names **a, b** and **c**.
- But the **compiler** would still treat them as different variables since they are in different functions.

Example

```
int multiply(int x, int y);
int sum ( int x , int y);
void main( )
{
int m=6,n=3;
printf("%d\n",multiply(m,n));
printf ( "%d \n" ,multiply(15,4));
printf("%d\n",multiply(m+n,m-n));
printf("%d\n",multiply(6,sum(m,n))) ;
}
int multiply(int x, int y)
{
int p;
p=x*y;
return p;
}
int sum ( int x , int y)
{
return x+y;
}
```

OUTPUT

18

60

27

54

Types Of Functions

- The **functions** can be **classified** into **four categories** on the basis of the **arguments** and **return value**.
 1. Functions with **no arguments** and **no return value**.
 2. Functions with **no arguments** and a **return value**.
 3. Functions with **arguments** and **no return value**.
 4. Functions with **arguments** and a **return value**.

Functions with no arguments and no return value.

```
#include<stdio.h>

void display();      //function prototype

void main()
{
    printf("Do you like programming in C language??");
    display();      //function call
}

void display()      //function definition
{
    printf("\nYes I like C programming language");
}
```

OUTPUT:

```
Do you like programming in C language??
Yes I like C programming language
```

Functions with no arguments and no return value

```
#include<stdio.h>

void cube(); //function prototype
```

```
void main()
{
    printf("First 5 nonzero integers and their cubes are: ");
    cube(); //function call
}
```

```
void cube() //function definition
{
    int i;
    for(i=1;i<=5;i++)
        printf("\n%d %d", i, i*i*i);
}
```

OUTPUT:

```
First 10 nonzero integers and their cubes are:
1  1
2  8
3  27
4  64
5  125
```

Functions with no arguments and a return value

```
#include<stdio.h>

int nsum();          //function prototype

void main()
{
    int i;
    printf("\nSum of 10 natural numbers= %d",nsum());      //function call
}

int nsum()           //function definition
{
    int i, sum=0;
    for(i=1;i<=10;i++)
        sum=sum+i;
    return sum;
}
```

OUTPUT:

Sum of 10 natural numbers= 55

Functions with arguments and no return value.

```
#include<stdio.h>

void nsum(int);      //function prototype

void main()
{
    int i=10;
    nsum(i);          //function call
}

void nsum(int x)    //function definition
{
    int i, sum=0;
    for(i=1;i<=x;i++)
        sum=sum+i;
    printf("\nSum of 10 natural numbers= %d",sum);
}
```

OUTPUT:

Sum of 10 natural numbers= 55

Functions with arguments and a return value.

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

```
int nsum(int); //function prototype
```

```
void main()
```

```
{
```

```
    int i=10;
```

```
    printf("\nSum of 10 natural numbers= %d",nsum(i)); //function call
```

```
}
```

```
int nsum(int x) //function definition
```

```
{
```

```
    int i, sum=0;
```

```
    for(i=1;i<=x;i++)
```

```
        sum=sum+i;
```

```
    return sum;
```

```
}
```

OUTPUT:

Sum of 10 natural numbers= 55

More about Function Declaration

- If **actual arguments** are more than the **formal argument** then the extra actual arguments are just **ignored**.
- If **actual arguments** are **less** than the **formal arguments**, then the extra formal arguments receive **garbage value**.

```
func (int a, int b, int c)  
{  
}
```

- **func(1, 2, 3, 4, 5); /*Actual arguments more than formal*/**
Here the last two arguments are just ignored. a = 1, b = 2, c = 3
- **func(1, 2); /*Actual arguments less than formal*/**
Here the third argument receives garbage value a = 1, b = 2,c = garbage value

More about Function Declaration

- If there is a **type mismatch** between a corresponding **actual** and **formal argument**, then **compiler** tries to **convert** the **type** of **actual argument** to the **type** of the **formal argument** if the **conversion** is **legal**.
- Otherwise a **garbage value** is passed to the **formal argument**.

Library Functions

- The **library functions** are supplied with every **C compiler**.
- The **source code of the library functions** is not given to the user.
- These **functions** are **precompiled** and the user gets only the **object code**.
- This **object code** is **linked**, to the **object code** of your **program** by the **linker**.
- Different categories of **library functions** are grouped together in separate **library files**.
- When we call a **library function** in our **program**, the **linker** selects the **code** of that **function** from the **library file** and adds it to the **program**.

Using Library Functions

- To use a **library function** in our **program** we should know about:
 1. *Name of the function and its purpose*
 2. *Type and number of arguments it accepts*
 3. *Type of the value it returns*
 4. *Name of the header file to be included*
- Example

```
#include<math.h>

void main()
{
    Int x=16;
    printf("squareroot of %d is %d",x, sqrt(x));
}
```

Recursion

- Recursion is a **powerful technique** of writing a **complicated algorithm** in an easy way.
- According to this technique a **problem is defined in terms of itself**.
- The **problem** is solved by **dividing it into smaller problems**, which are **similar in nature** to the **original problem**.
- These **smaller problems** are solved and their solutions are applied to get **the final solution** of our original problem.
- To implement **recursion technique** in programming, a **function** should be capable of **calling itself** and this facility is available in C.
- The **function that calls itself (inside function body) again and again is known as a recursive function**.
- In **recursive function** the **calling function** and the **called function** are **same**.

Recursive Function

```
void main()
{
    .....
    func();
    .....
}

void func()      //recursive function
{
    .....
    if ( ... )      //terminating condition
    .....
    func();        //recursive call
    .....
}
```

- There should be a **terminating condition** to *stop* recursion, otherwise **func()** will keep on calling itself **infinitely** and will **never return**.

Recursive Function

- Before writing a **recursive function** for a **problem** we should consider these points:
 - We should be able to **define the solution** of the problem in terms of a similar **type of smaller problem**.
 - At **each step** we get closer to 'the final' solution of our **original problem**.
 - There should be a **terminating condition** to stop recursion.
- **Some problems** which can be **solved** using **recursive functions** are:
 - ***Factorial***
 - ***Power***
 - ***Fibonacci numbers***
 - ***Tower of Hanoi***

Factorial using Recursion

- Factorial of a positive integer n can be found out by multiplying all integers from 1 to n .
 - $n! = 1 * 2 * 3 * \dots * (n-1) * n$
- This is the **iterative definition of factorial** and its program can be written using **loop**.
- **The recursive definition of factorial:**
- We know that $6! = 6 * 5 * 4 * 3 * 2 * 1$
- We can write it as- $6! = 6 * 5!$
- Similarly we can write- $5! = 5 * 4!$
- So in general we can write $n! = n * (n-1)!$

Factorial using Recursion

- Now problem of finding out **factorial of (n-1)** is similar to that of finding out **factorial of n**, but it is definitely **smaller in size**.
- So we have **defined the solution of factorial problem in terms of itself**.
- We know that the **factorial of 0 is 1**.
- This can act as the **terminating condition**.
- So the **recursive definition of factorial** can be written as:

$$n! = \begin{cases} 1 & n=0 \\ n * (n-1)! & n>0 \end{cases}$$

Factorial using Recursion

```
#include<stdio.h>
long fact(int n);
void main ( )
{
    int num;
    printf ("Enter a number ") ;
    scanf("%d",&num);
    printf ("Factorial of %d is %ld\n", num, fact (num) );
}
long fact (int n)
{
    if (n==0)
        return(1) ;
    else
        return(n*fact(n-1)) ;
}
```

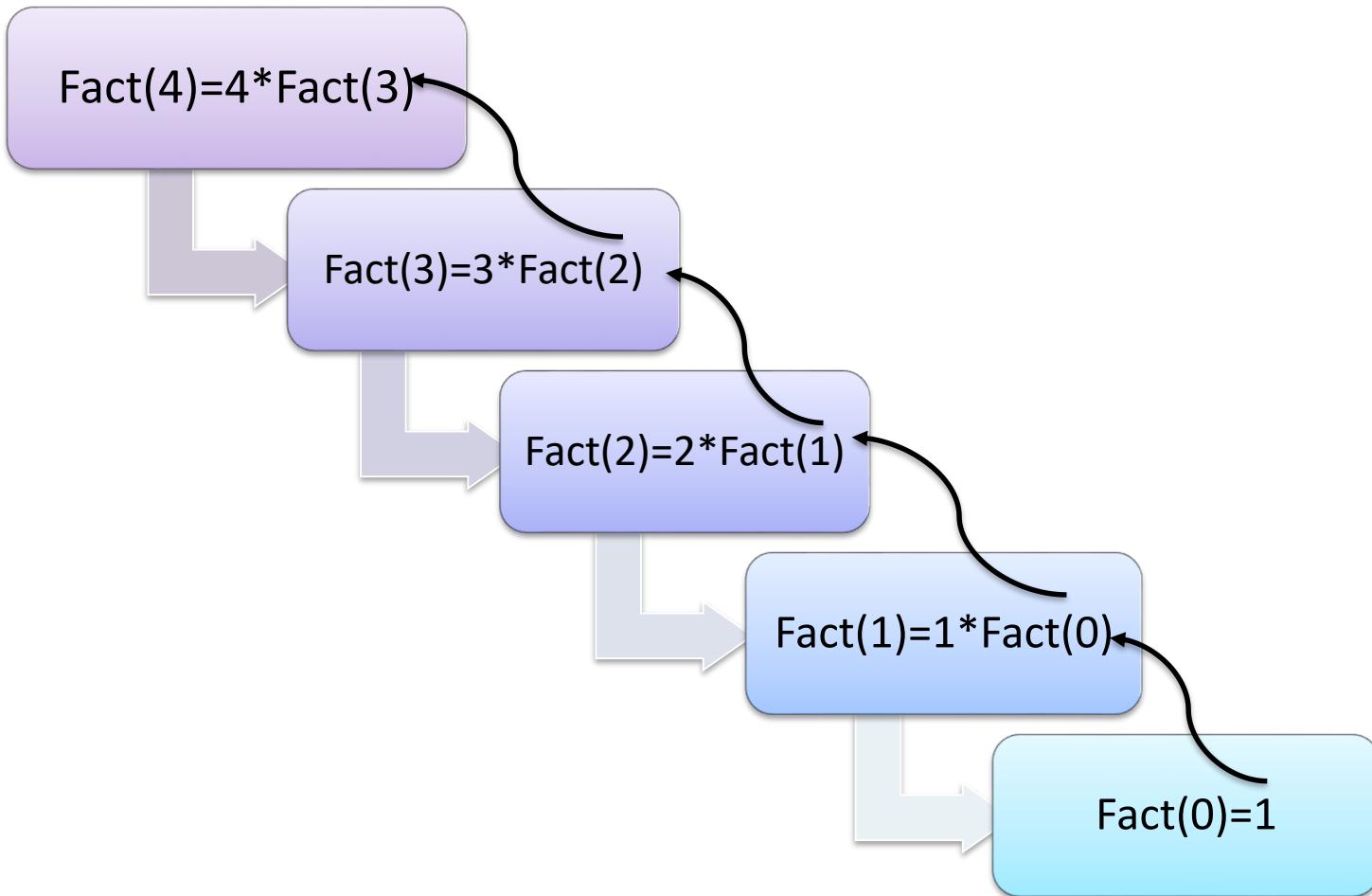
Factorial using Recursion

- This function **returns 1** if the **argument n** is **0**, otherwise it **returns n*fact(n-1)**.
- To return $n * \text{fact}(n-1)$, the value of **fact(n-1)** has to be **calculated** for which **fact()** has to be **called again** but this time with an **argument of n-1**.
- This process of **calling fact()** continues **till** it is called with an **argument of 0**.
- Suppose, we want to find out the **factorial of 4**.
 - Initially **main()** calls **factorial(4)**
 - Since $4 > 0$, **factorial(4)** calls **factorial(3)**
 - Since $3 > 0$, **factorial(3)** calls **factorial(2)**
 - Since $2 > 0$, **factorial(2)** calls **factorial(1)**
 - Since $1 > 0$, **factorial(1)** calls **factorial(0)**

Factorial using Recursion

- When **factorial()** is called with **n=0** then the **condition** inside **if** statement becomes **true**, so now the **recursion stops** and **control returns** to **factorial(1)**
- Now **every called function will return the value to the previous function.**
- These **values are returned** in the **reverse order of function calls**.
- In the above case the function **factorial()** is called **5 times**, but there is only one copy of that function in memory.
- Each **function call** is different from another because the argument supplied is different each time.

Factorial using Recursion



Recursive function for finding a^n

- The **iterative definition** for finding a^n is

$a^n = a * a * a * \dots \text{n times}$

- The **recursive definition** can be written as-

$$a^n = \begin{cases} 1 & n=0 \\ a * a^{n-1} & n>0 \end{cases}$$

float power (float a, int n)

{

```
if(n==0)
return(1);
else
return(a*power(a,n-1));
}
```

Advantages and Disadvantages of Recursion

- The use of **recursion** makes the **code** more **compact** and **elegant**.
- **It simplifies the logic** and hence makes the program **easier to understand**.
- But the code written using **recursion** is **less efficient** since **recursion** is a **slow process** because of **many function calls** involved in it and **needs more memory**.
- Most problems with **recursive solutions** also have an **equivalent non-recursive(generally iterative) solutions**.
- A **non recursive solution** increases performance while a **recursive solution** is simpler.

Local Variables In Recursion

- Each **function** has some **local variables** that exist only **inside that function**.
- When a **function** is **called recursively**, then for **each call** a **new set of local variables** is **created(except static)**, their name is same but they are **stored at different places** and **contain different values**.
- These **values** are **remembered** by the **compiler** till the **end of function call**.

Calling Convention

- Calling convention indicates the order in which arguments are passed to a function when a function call is encountered.
- There are two possibilities here:
 - (a) Arguments might be passed from left to right.
 - (b) Arguments might be passed from right to left.
- C language follows the second order.
- Consider the following function call:
`fun (a, b, c, d);`
- In this call it doesn't matter whether the arguments are passed from left to right or from right to left.
- However, in some function call the order of passing arguments becomes an important consideration.

Example

```
int a = 1 ;  
  
printf ( "%d %d %d", a, ++a, a++ ) ;
```

- It appears that this **printf()** would output **1 2 3**.
- This however is not the case. Surprisingly, it outputs **3 3 1**.
- This is because C's calling convention is from **right to left**.
- That is, firstly 1 is passed through the expression **a++** and then **a** is incremented to 2.
- Then result of **++a** is passed.
- That is, **a** is incremented to 3 and then passed.
- Finally, latest value of **a**, i.e. 3, is passed.
- Thus in right to left order 1, 3, 3 get passed.
- Once **printf()** collects them it prints them in the order in which we have asked it to get them printed (and not the order in which they were passed).
- Thus **3 3 1** gets printed.

Types of Function Call

- Arguments can generally be passed to functions in one of the two ways:
 - (a) sending the values of the arguments- **Call by value**
 - (b) sending the addresses of the arguments- **Call by reference**
- (a). **Call by value:**
 - With this method the changes made to the **formal arguments** in the **called function** have **no effect** on the **values of actual arguments** in the **calling function**.

Example: Call by Value

```
main( )
{
    int a = 10, b = 20 ;
    swapv ( a, b ) ;
    printf ( "\na = %d b = %d", a, b ) ;
}

void swapv ( int x, int y )
{
    int t ;
    t = x ;
    x = y ;
    y = t ;
    printf ( "\nx = %d y = %d", x, y ) ;
}
```

OUTPUT:

```
x = 20 y = 10
a = 10 b = 20
```

- Note that values of **a** and **b** remain **unchanged** even after exchanging the values of **x** and **y**.

Lecture 10

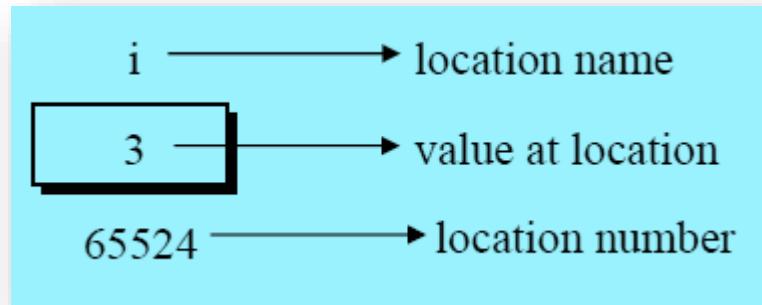
Pointer in C

Dr. Vandana Kushwaha

Department of Computer Science
Institute of Science, BHU, Varanasi

Introduction

- **Pointer** is a variable which stores **address** of another variable.
- **Pointer Notation**
- Consider the declaration, int i = 3 ;
- This **declaration** tells the **C compiler** to:
 - (a) Reserve space in **memory** to **hold** the **integer value**.
 - (b) Associate the **name i** with this **memory location**.
 - (c) Store the **value 3** at this **location**.
- We may represent i's location in memory by the following memory map.



Pointer

- We see that the computer has selected memory location **65524** as the place to store the value **3**.
- The location number **65524** is not a fixed number, because some other time the computer may choose a different location for storing the value **3**.
- The important point is, **i's address** in memory is a **number**.
- We can print this **address number** through the following program:

```
main( )  
{  
    int i = 3 ;  
  
    printf ( "\nAddress of i = %u", &i ) ;  
  
    printf ( "\nValue of i = %d", i ) ;  
}
```

OUTPUT:

Address of i = 65524

Value of i = 3

Pointer

- ‘**&**’ used in this statement is C’s ‘**address of**’ operator.
- The expression **&i** returns the **address of the variable i**, which in this case happens to be 65524.
- Since 65524 represents an address, there is no question of a sign being associated with it.
- Hence it is printed out using **%u**, which is a **format specifier** for printing an **unsigned integer**.
- The other **pointer operator** available in **C** is ‘*****’, called ‘**value at address**’ operator.
- It gives the **value stored at a particular address**.
- The ‘**value at address**’ operator is also called ‘**indirection**’ operator.

Example

```
main( )  
{  
    int i = 3 ;  
  
    printf ( "\nAddress of i = %u", &i ) ;  
  
    printf ( "\nValue of i = %d", i ) ;  
  
    printf ( "\nValue of i = %d", *( &i ) ) ;  
  
}
```

OUTPUT:

```
Address of i = 65524  
Value of i = 3  
Value of i = 3
```

- Note that printing the value of ***(&i)** is same as printing the value of **i**.
- The expression **&i** gives the address of the variable **i**.
- This address can be collected in a variable, by saying, **j = &i** ;
- **j** is not an ordinary variable like any other integer variable.

Example

- It is a **variable** that contains the **address of other variable (i in this case)**.



- We can't use **j** in a program without declaring it.
- And since **j** is a **variable** that **contains the address of i**, it is **declared** as, **int *j ;**
- This **declaration** tells the **compiler** that **j** will be used to **store the address** of an **integer value**.

Example

```
main( )
{
    int i = 3 ;
    int *j ;
    j = &i ;
    printf ( "\nAddress of i = %u", &i ) ;
    printf ( "\nAddress of i = %u", j ) ;
    printf ( "\nAddress of j = %u", &j ) ;
    printf ( "\nValue of j = %u", j ) ;
    printf ( "\nValue of i = %d", i ) ;
    printf ( "\nValue of i = %d", *( &i ) ) ;
    printf ( "\nValue of i = %d", *j ) ;
}
```

OUTPUT:

Address of i = 65524

Address of i = 65524

Address of j = 65522

Value of j = 65524

Value of i = 3

Value of i = 3

Value of i = 3

Pointer Declaration

- Consider the following declarations,

```
int *alpha ;
```

```
char *ch ;
```

```
float *s ;
```

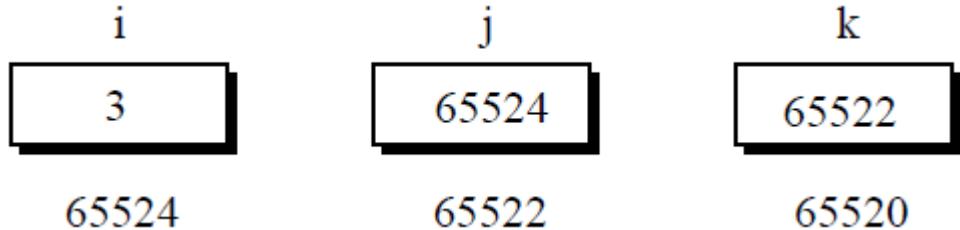
- Here, **alpha**, **ch** and **s** are declared as **pointer variables**, i.e. **variables** capable of holding **addresses**.
- Remember that, **addresses (location nos.)** are always going to be **whole numbers**, therefore **pointers** always contain **whole numbers**.
- Now we can put these two facts together and say—**pointers** are variables that **contain addresses**, and since **addresses** are always **whole numbers**, **pointers** would always contain **whole numbers**.

Pointer Declaration

- The declaration **float *s** does not mean that **s** is going to contain a **floating-point value**.
- What it means is, **s** is going to contain the **address of a floating-point value**.
- Similarly, **char *ch** means that **ch** is going to contain the **address of a char value**.
- Or in other words, the **value at address** stored in **ch** is going to be a **char**.
- Even **pointer** can contains **another pointer's address**.

Pointer to pointer

```
main()
{
    int i = 3, *j, **k ;
    j = &i ;
    k = &j ;
    printf ( "\nAddress of i = %u", &i ) ;
    printf ( "\nAddress of i = %u", j ) ;
    printf ( "\nAddress of i = %u", *k ) ;
    printf ( "\nAddress of j = %u", &j ) ;
    printf ( "\nAddress of j = %u", k ) ;
    printf ( "\nAddress of k = %u", &k ) ;
    printf ( "\nValue of j = %u", j ) ;
    printf ( "\nValue of k = %u", k ) ;
    printf ( "\nValue of i = %d", i ) ;
    printf ( "\nValue of i = %d", * ( &i ) ) ;
    printf ( "\nValue of i = %d", *j ) ;
    printf ( "\nValue of i = %d", **k ) ;
}
```



OUTPUT:

Address of i = 65524
Address of i = 65524
Address of i = 65524
Address of j = 65522
Address of j = 65522
Address of k = 65520
Value of j = 65524
Value of k = 65522
Value of i = 3
Value of i = 3
Value of i = 3
Value of i = 3

Pointer to pointer

- Variables **i**, **j** and **k** have been **declared**,

`int i, *j, **k;`

- Here,
 - **i** is an ordinary **int**,
 - **j** is a **pointer** to an **int** (often called an integer pointer), whereas
 - **k** is a **pointer** to an **integer pointer**.
- We can **extend** the above program still further by creating a pointer to a pointer to an integer pointer.

Example

- Let us consider the following declarations:

`int a = 87;`

`float b= 4.5;`

`int *p1 = &a;`

`float *p2 = &b;`

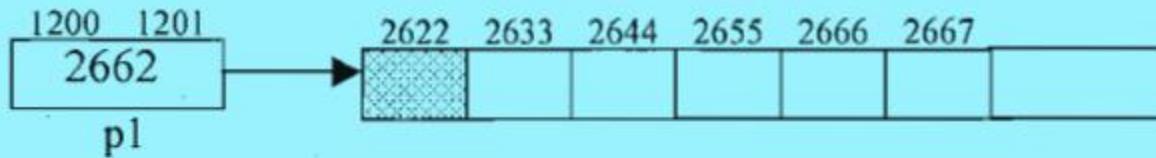
- `*p1= 9;` is equivalent to `a = 9;`
- `(*p1)++;` is equivalent to `a++;`
- `x = *p2 + 10;` is equivalent to `x = b + 10;`
- `printf("%d %f", *p1, *p2);` is equivalent to `printf("%d %f", a, b);`
- `scanf("%d%f", p1, p2);` is equivalent to `scanf("%d %f ", &a, &b);`

Pointer

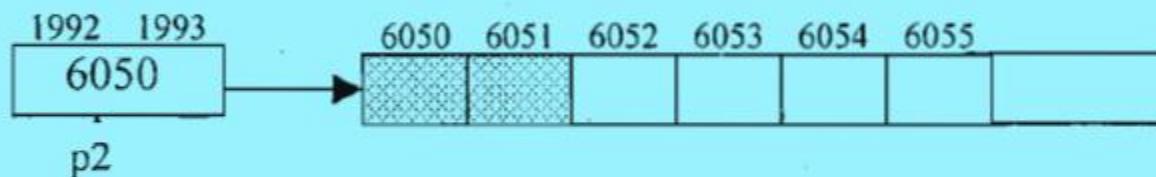
- While **declaring a pointer variable**, we have to mention the **data type**.
- The reason is that the **number of bytes reserved** will be different for different **data types**.
- The **address** stored in the **pointer** only tells the **address of starting byte**. For example suppose we have a **pointer ptr** which contains the **address 2000** and when we write ***ptr**.
- The compiler knows that it has to access the information **starting at address 2000**.
- So the **compiler** will look at the **data type** of the **pointer** and will **retrieve** the information depending on that **data type**.
- For example if **data type** is **int** then **2 bytes** information will be **retrieved** and if **data type** is **float, 4 bytes** information will be **retrieved** and so on.
- The **size of pointer variable** is **same for all type of pointers** but the memory that will be accessed while dereferencing is different.

Pointers of different data type

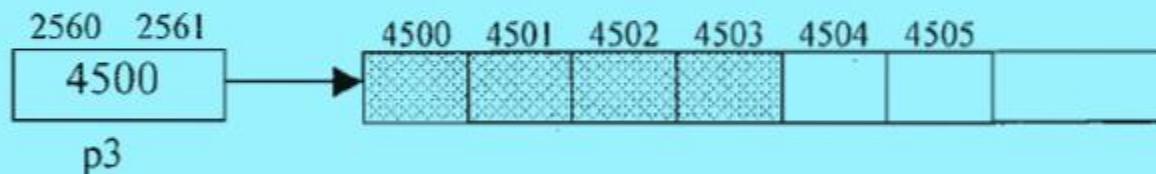
```
char *p1;
```



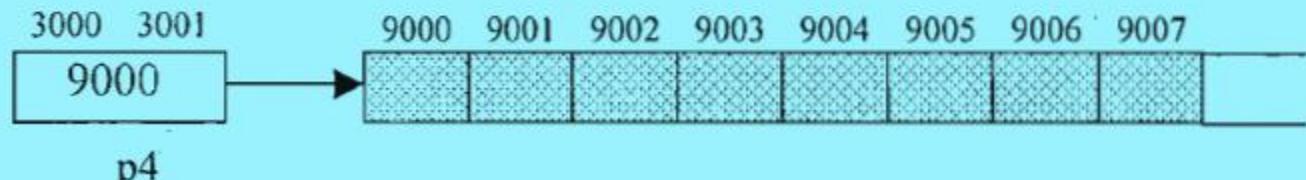
```
int *p2;
```



```
float *p3;
```



```
double *p4;
```



Pointers of different data type

```
#include<stdio.h>
main( )
{
    char a='x', *p1=&a;
    int b=12, *p2=&b;
    float c=12.4, *p3=&c;
    double d=18.3, *p4=&d;
    printf("sizeof(p1) = %d , sizeof(*p1)= %d\n",sizeof(p1),sizeof(*p1));
    printf("sizeof(p2) = %d , sizeof(*p2) = %d\n",sizeof(p2),sizeof(*p2));
    printf("sizeof(p3) = %d , sizeof(*p3) = %d\n",sizeof(p3),sizeof(*p3));
    printf("sizeof(p4) = %d , sizeof(*p4) = %d\n",sizeof(p4),sizeof(*p4));
}
```

OUTPUT:

sizeof(p1) = 2 , sizeof(*p1) = 1

sizeof(p2) = 2 , sizeof(*p2) = 2

sizeof(p3) = 2 , sizeof(*p3) =4..

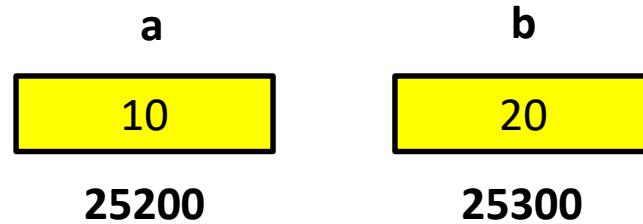
sizeof(p4) = 2 , sizeof(*p4) =8

Call by Reference

- In the second method (**call by reference**) the **addresses** of **actual arguments** in the **calling function** are copied into **formal arguments** of the **called function**.
- This means that using these **addresses** we would have an **access** to the **actual arguments** and hence we would be able to **manipulate** them.

```
main( )
{
int a = 10, b = 20 ;
swapr ( &a, &b ) ; //swapr(25200,25300);
printf ( "\na = %d b = %d", a, b ) ;
}

swapr( int *x, int *y ) // x=25200 y=25300
{
int t ;
t = *x ;           //t= *25200 -> t=10
*x = *y ;         // *25200 = *25300 -> a=20
*y = t ;          // *25300 = 10 -> b=10
}
```



OUTPUT:
a = 20 b = 10

Call by Reference

- Using a **call by reference** intelligently we can make a function **return more than one value at a time**, which is not possible ordinarily.
- Thus, we have been able to **indirectly return two values** from a **called function**.
- And hence, we have overcome the **limitation** of the **return statement**, which can **return only one value** from a function **at a time**.

Mixed Call

```
main( )
{
int radius ;
float area, perimeter ;
printf ( "\nEnter radius of a circle " ) ;
scanf ( "%d", &radius ) ;
areaperi ( radius, &area, &perimeter ) ;
printf ( "Area = %f", area ) ;
printf ( "\nPerimeter = %f", perimeter ) ;
}
areaperi ( int r, float *a, float *p )
{
*a = 3.14 * r * r ;
*p = 2 * 3.14 * r ;
}
```

OUTPUT:

Enter radius of a circle 5

Area = 78.500000

Perimeter = 31.400000

Lecture 11

Array in C

Dr. Vandana Kushwaha

Department of Computer Science
Institute of Science, BHU, Varanasi

Introduction

- The **variables** that we have used till now are **capable of storing only one value** at a time.
- Consider a situation when we want to store and display the **age of 100 employees**.
For this we have to do the following-
 1. Declare **100 different variables** to store the age of employees.
 2. Assign a value to each variable.
 3. Display the value of each variable.
- Although we can perform our task by the above three steps but just imagine how **difficult** it would be to handle so many variables in the program and the program would become very lengthy.
- The concept of **arrays** is useful in these types of situations where we can **group similar type of data items**.

Array

- An **array** is a **collection of similar type of data items** and each data item is called an **element of the array**.
- The **data type of the elements** may be any **valid data type** like **char, int or float**.
- The **elements of array** share the **same variable name** but **each element** has a **different index number** known as **subscript**.
- For the above problem we can take an **array variable age[100]** of type **int**.
- The **size of this array variable** is **100** so it is capable of storing 100 integer values.
- The **individual elements of this array** are- .
- age[0], age[1], age[2], age[3], age[4],..... age[98], age[99]
- In C the **subscripts start from zero**, so **age[0]** is the **first element**, **age[1]** is the **second element** of array and so on.

Array

- Arrays can be single dimensional or multidimensional.
- The **number of subscripts** determines the **dimension** of array.
- A **one-dimensional array** has **one subscript**, **two dimensional array** has **two subscripts** and so on.
- The **one-dimensional arrays** are known as **vectors** and **two-dimensional arrays** are known as **matrices**.

One Dimensional Array

- **Declaration of 1-D Array**
- Like other simple variables, arrays should also be declared before they are used in the program.
- The **syntax for declaration** of an array is:
 - *data_type array_name[size];*
- Here **array_name** denotes the name of the array and it can be any valid C identifier, **data_type** is data type of the elements of array.
- The **size** of the array specifies the **number of elements** that be stored in the array.
- It may be a **positive integer constant** or constant integer expression.

One Dimensional Array

- Here are some **examples of array declarations-**
 - int age[100]; .
 - float sal[15];
 - char grade[20];
- Here **age** is an integer type array, which can store 100 elements of integer type.
- The array **sal** is a floating type array of size 15, can hold float values and third one is a character type array of size 20, can hold characters.
- The **individual elements** of the above **arrays** are:
 - age[0], age[1], age[2],..... age[99]
 - sal[0], sal[1], sal[2],sal[14]
 - grade[0], grade[1], grade[2],..... grade[19]

Accessing 1-D Array Elements

- The **elements** of an **array** can be **accessed** by specifying the **array name** followed by **subscript in brackets**.
- In C, the array **subscripts start from 0**.
- Hence if there is an **array of size 5** then the valid **subscripts** will be from **0 to 4**.
- The **last valid subscript is one less than the size of the array**.
- This **last valid subscript** is sometimes known as the **upper bound** of the **array** and **0** is known as the **lower bound** of the **array**.
- Example
- int arr[5]; /*Size of array arr is 5, can hold five integer elements*/
- The **elements of this array** are :
- arr[0], arr[1], arr[2], arr[3], arr[4]
- Here **0 is the lower bound and 4 is the upper bound of the array**.

Accessing 1-D Array Elements

- In **C** there is **no check on bounds** of the **array**.
- For **example** if we have an **array arr[5]** , the **valid subscripts** are only **0, 1, 2, 3, 4** .
- And if someone tries to access elements **beyond these subscripts** like **arr[5]** , **arr[10]** or **arr[-1]**, the **compiler will not show any error message** but this may **lead to run time errors**, which can be very **difficult to debug**.
- So it is the **responsibility of programmer** to provide **array bounds checking** wherever needed.

Processing 1-D Arrays

- For **processing arrays** we generally use a **for loop** and the **loop variable** is used at the place of **subscript**.
- The **initial value of loop variable** is taken **0** since **array subscripts start from zero**.
- The **loop variable** is **increased by 1** each time so that we can access and process the *next element* in the array.
- The **total number of passes in the loop** will be equal to the **number of elements in the array** and in each pass, we will process **one element**.
- Suppose **arr[10]** is an array of **int** type-
- (i) **Reading values in arr[10]**

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

```
scanf("%d", &arr[i]);
```

Processing 1-D Arrays

- (ii) Displaying values of arr[10]

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

```
printf("%d ", arr[i]);
```

- (iii) Adding all the elements of arr[10]

```
sum = 0;
```

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

```
sum += arr[i];
```

Program to input values into an array and display them

```
main( )
{
    int arr[5],i;
    for(i=0;i<5;i++)
    {
        printf ("Enter the value for arr[ %d] " , i) ;
        scanf("%d",&arr[i]) ;
    }
    printf ("The array elements are: \n");
    for(i=0;i<5;i++)
        printf("%d\t",arr[i]);
    printf ("\n");
}
```

Output:

```
Enter the value for arr[0] 12
Enter the value for arr[ 1] 45
Enter the value for arr[2] 59
Enter the value for arr[3] 98
Enter the value for arr[4] 21
The array elements are :
12 45 59 98 21
```

Initialization of 1-D Array

- After **declaration**, the **elements** of an **array** have **garbage value** while the elements of **global** and **static arrays** are **automatically initialized to zero**.
- We can **explicitly initialize arrays** at the time of **declaration**.
- **Syntax**
- ***data_type array_name[size]={value1, value2..... valueN };***
- Here **array_name** is the name of the array variable, **size** is the size of the array and **value1, value2,.....valueN** are the constant values known as initializers, which are assigned to the array elements one after another.
- These **values** are **separated by commas** and there is a **semicolon** after the ending braces.
- **Example**
- **int marks[5] = {50, 85, 70, 65, 95};**

Initialization of 1-D Array

- While **initializing 1-D arrays**, it is **optional to specify** the **size** of the **array**.
- If the **size** is **omitted** during **initialization** then the **compiler assumes** the **size** of **array** equal to the **number of initializers**.
- **Example-**
- int marks[] = { 99, 78, 50, 45, 67, 89};
- Here the **size of array** marks is assumed to be **6**.
- If during **initialization** the **number of initializers** is **less than the size of array** then, all the remaining elements of array are assigned value **zero**.
- **Example-** int marks[5] = { 99, 78};
- Here the **size of array** is **5** while there are only **2 initializers**.
- After this initialization the value of the elements are as-
- marks[0] : 99, marks[1]: 78, marks[2]: 0, marks[3]: 0, marks[4]: 0

Initialization of 1-D Array

- So if we initialize an **array** like this `int arr[100] = {0};` then all the elements of arr will be initialized to zero.
- If the **number of initializers** is **more than the size** given in brackets then **compiler** will show an **error**.
- For example `int arr[5] = {1, 2, 3,4,5,6,7, 8}; //Error`

Initialization of 1-D Array

- We **can't copy** all the elements of an array to another array by simply **assigning** it to the other array.
- For example if we have two arrays **a[5]** and **b[5]** then

```
int a[5] = {1, 2, 3, 4, 5};
```

```
int b[5];
```

```
b = a; /*Not valid*/
```

- We'll have to **copy all the elements** of array **one by one**, using a **for loop**.

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

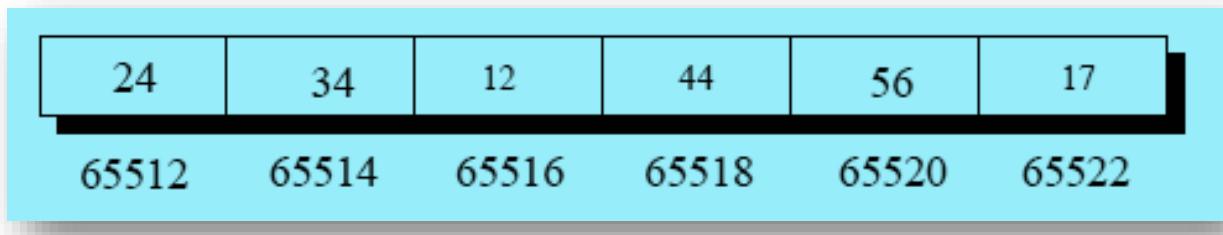
```
b[i] = a[i];
```

Program to search for an item in the array

```
main ( )
{
    int i,arr[10]={23,12,56,98,76,14,65,11,19,45};
    int item;
    printf ("Enter the item to be searched ") ;
    scanf("%d",&item)
    for(i=0;i<10;i++)
    {
        if(item==arr[i])
        {
            printf ("%d found at position %d\n", item, i+1);
            break;
        }
        if (i==10)
            printf("Item %d not found in array\n" , item) ;
    }
}
```

Array Elements in Memory

- Suppose we have an array **num[] = { 24, 34, 12, 44, 56, 17 }.**
- The following figure shows how this array is located in **memory**.



- 16 bytes get immediately reserved in memory, 2 bytes each for the 8 integers .**
- And since the **array** is not being initialized, all eight values present in it would be **garbage values**.
- Whatever be the initial values, all the **array elements** would always be present in **contiguous memory locations**.

Array Elements in Memory

```
main( )
{
    int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    int i ;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        printf ( "\nelement no. %d ", i ) ;
        printf ( "address = %u", &num[i] ) ;
    }
}
```

OUTPUT:

element no. 0 address = 65512

element no. 1 address = 65514

element no. 2 address = 65516

element no. 3 address = 65518

element no. 4 address = 65520

element no. 5 address = 65522

Accessing the array elements using Pointers

```
void main( )
{
    int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    int i, *j ;
    j = &num[0] ; /* assign address of zeroth element(base address) */
    for ( i = 0 ; i <= 5 ; i++ )
    {
        printf ( "\naddress = %u ", j ) ;
        printf ( "element = %d", *j ) ;
        j++ ; /* increment pointer to point to next location */
    }
}
```

OUTPUT:

address = 65512 element = 24
address = 65514 element = 34
address = 65516 element = 12
address = 65518 element = 44
address = 65520 element = 56
address = 65522 element = 17

Accessing the array elements using pointers

- Accessing **array elements** by **pointers** is **always faster** than accessing them by **subscripts**.
- However, from the **point of view of convenience** in programming we should observe the following:
 - Array **elements** should be **accessed using pointers** if the **elements** are to be **accessed in a fixed order**, say from **beginning to end**, or from **end to beginning**, or **every alternate element** or any such definite logic.
 - Instead, it would be **easier to access the elements** using a **subscript** if there is **no fixed logic** in accessing the **elements**.

Passing Array Elements to a Function

- We can pass individual array elements as arguments to a function like other simple variables.
- **Using Call by value**

```
main( )
{
    int i ;
    int marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;
    for ( i = 0 ; i <= 6 ; i++ )
        display ( marks[i] ) ;
}
display ( int m )
{
    printf ( "%d ", m ) ;
}
```

Passing Array Elements to a Function

- Using Call by reference

```
main( )
{
    int i ;
    int marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;
    for ( i = 0 ; i <= 6 ; i++ )
        disp ( &marks[i] ) ;
}
disp ( int *n )
{
    printf ( "%d ", *n ) ;
}
```

- Here, we are passing **addresses of individual array elements** to the **function display()**.
- Hence, the **variable** in which this **address is collected (n)** is declared as a **pointer variable**.

Pointer Arithmetic

```
void main( )
{
int i = 3, *x;
float j = 1.5, *y ;
char k = 'c', *z ;
printf ( "\nValue of i = %d", i ) ;
printf ( "\nValue of j = %f", j ) ;
printf ( "\nValue of k = %c", k ) ;
x = &i ; y = &j ; z = &k ;
printf ( "\nOriginal address in x = %u", x ) ;
printf ( "\nOriginal address in y = %u", y ) ;
printf ( "\nOriginal address in z = %u", z ) ;
x++ ; y++ ; z++ ;
printf ( "\nNew address in x = %u", x ) ;
printf ( "\nNew address in y = %u", y ) ;
printf ( "\nNew address in z = %u", z ) ;
}
```

OUTPUT:

Value of i = 3

Value of j = 1.500000

Value of k = c

Original address in x = 65524

Original address in y = 65520

Original address in z = 65519

New address in x = 65526

New address in y = 65524

New address in z = 65520

Pointer Arithmetic

- Every time a **pointer** is **incremented** it **points** to the **immediately next location** of its **type**.
- That is why, when the **integer pointer x** is **incremented**, it **points** to an **address** two locations after the current location **65526**, since an **int** is **2 bytes** long.
- Similarly, **y** points to an address 4 locations after the current location and **z** points **1** location after the current location.
- The way a **pointer** can be **incremented**, it can be **decremented** as well, to **point** to **earlier locations**.

Pointer Arithmetic

(a) **Addition of a number to a pointer.** For example,

```
int i = 4, *j, *k ;
```

```
j = &i ;
```

```
j = j + 1 ;
```

```
j = j + 9 ;
```

```
k = j + 3 ;
```

(b) **Subtraction of a number from a pointer.** For example,

```
int i = 4, *j, *k ;
```

```
j = &i ;
```

```
j = j - 2 ;
```

```
j = j - 5 ;
```

```
k = j - 6 ;
```

Pointer Arithmetic

Subtraction of one pointer from another.

- One **pointer variable** can be subtracted from another provided **both variables point to elements of the same array**.
- The **resulting value** indicates the **number of bytes** separating the corresponding array elements.
- Example:

```
main( )
{
    int arr[ ] = { 10, 20, 30, 45, 67, 56, 74 } ;
    int *i, *j ;
    i = &arr[1] ;
    j = &arr[5] ;
    printf ( "%d %d", j - i, *j - *i ) ;
}
```

Pointer Arithmetic

- Here **i** and **j** have been declared as **integer pointers** holding addresses of **first** and **fifth element** of the array respectively.
- Suppose the array begins at location **65502**, then the elements **arr[1]** and **arr[5]** would be present at locations **65504** and **65512** respectively, since each integer in the array occupies **two bytes** in memory.
- The expression **j - i** would print a value **4** and not **8**.
- This is because **j** and **i** are pointing to locations that are **4 integers apart**.
- What would be the result of the expression ***j - *i** ? **36**, since ***j** and ***i** return the values present at addresses contained in the pointers **j** and **i**.

Comparison of two pointer variables

- Pointer variables can be compared provided both variables point to objects of the same data type.
- Such comparisons can be useful when both pointer variables point to elements of the same array.
- The comparison can test for either equality or inequality.
- Moreover, a pointer variable can be compared with zero (usually expressed as NULL).

Comparison of two pointer variables

```
main( )
```

```
{
```

```
    int arr[ ] = { 10, 20, 36, 72, 45, 36 };
```

```
    int *j, *k ;
```

```
    j = &arr [ 4 ] ;
```

```
    k = ( arr + 4 ) ;
```

```
    if ( j == k )
```

```
        printf ( "The two pointers point to the same location" ) ;
```

```
    else
```

```
        printf ( "The two pointers do not point to the same location" ) ;
```

```
}
```

A word of caution

- Do not attempt the following operations on pointers... they would never work out:
 - (a) Addition of two pointers
 - (b) Multiplication of a pointer with a constant
 - (c) Division of a pointer with a constant

Passing an Entire Array to a Function

Method 1:

```
int main( )
{
    int num[ ] = { 24, 34, 12, 44, 56, 17 };
    display ( &num[0], 6 ) ; //display(num,6);
}

void display ( int *j, int n )
{
    int i ;
    for ( i = 0 ; i < n ; i++ )
    {
        printf ( "\nelement = %d", *j );
        j++ ; /* increment pointer to point to next element */
    }
}
```

- Note that the **address** of the **zeroth element** is being passed to the **display()** function.

Passing an Entire Array to a Function

- Thus, just **passing the address of the zeroth element of the array** to a **function** is as good as passing the **entire array** to the function.
- It is also **necessary to pass the total number of elements in the array**, otherwise the **display()** function would not know **when to terminate the for loop**.
- Note that the **address of the zeroth element (base address)** can also be **passed** by **just passing the name of the array**.
- Thus, the following **two function calls** are same:
 - `display (&num[0], 6);`
 - `display (num, 6);`

Passing an Entire Array to a Function

Method 2: Passing the array using **array notation**:

```
int main() {  
    int arr[] = {1, 2, 3, 4, 5};  
    display(arr, 5);  
    return 0;  
}
```

```
void display(int arr[], int size) {  
    for (int i = 0; i < size; i++)  
        printf("\n%d ", arr[i]);  
}
```

Pointer and 1-D Array

- Consider the **array declaration**:

```
int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
```

- By mentioning the **name of the array** we get its **base address**.
- Thus, by saying ***num** we would be able to refer to the **zeroth element** of the **array**, that is, **24**.
- One can easily see that ***num** and ***(num + 0)** both refer to **24**.
- Similarly, by saying ***(num + 1)** we can refer the **first element** of the **array**, that is, **34**.
- In fact, this is what the **C compiler** does internally.
- When we say, **num[i]**, the **C compiler** internally converts it to ***(num + i)**.
- This means that all the following **notations** are same:
 - num[i] ≡ *(num + i) ≡ *(i + num) ≡ i[num]**

Accessing array elements in different ways

```
main( )  
{  
    int num[ ] = { 24, 34, 12, 44, 56, 17 } ;  
  
    int i ;  
  
    for ( i = 0 ; i <= 5 ; i++ )  
    {  
        printf ( "\naddress = %u ", &num[i] ) ;  
  
        printf ( "element = %d %d %d %d", num[i], *( num + i ), *( i + num ), i[num] ) ;  
    }  
}
```

OUTPUT:

```
address = 65512 element = 24 24 24 24  
address = 65514 element = 34 34 34 34  
address = 65516 element = 12 12 12 12  
address = 65518 element = 44 44 44 44  
address = 65520 element = 56 56 56 56  
address = 65522 element = 17 17 17 17
```

Two Dimensional Arrays

- It is also possible for **arrays** to have **two or more dimensions**.
- The **two dimensional array (2D Array)** is also called a **matrix**.
- **Declaring a 2-Dimensional Array**
- A **2D array** with **m rows** and **n columns** can be created as:

```
data_type arr_name[m][n];
```

- **data_type**: Type of data to be stored in each element.
 - **arr_name**: Name assigned to the array.
 - **m**: Number of rows.
 - **n**: Number of columns.
- **Example**, we can declare a **two-dimensional** integer array with name ‘arr’ with 10 rows and 20 columns as: **int arr[10][20];**

Initialising a 2-Dimensional Array

- We can **initialize** a **2D array** by using a **list of values** enclosed inside ‘{ }’ and separated by a comma as shown in the example below:
- `int arr[3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}`
- or
- `int arr[3][4] = {{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}};`
- It is important to remember that while **initializing a 2-D array** it is necessary to **mention the second (column) dimension**, whereas the **first dimension (row)** is **optional**.
- Thus the following **declaration** is perfectly **acceptable**:
- `int arr[][3] = { 12, 34, 23, 45, 56, 45 } ;`

Memory Map of a 2-Dimensional Array

- In **memory** whether it is a **one-dimensional** or a **two-dimensional array** the **array elements** are **stored** in one **continuous chain**.
- **Elements of a two-dimensional array are stored row-wise at contiguous memory location.**
- The **arrangement of array elements** of a **two-dimensional array in memory** is shown below for the array:
- **int s[4][2]= {1234, 56, 1212, 33, 1434, 80, 1312, 78};**

s[0][0]	s[0][1]	s[1][0]	s[1][1]	s[2][0]	s[2][1]	s[3][0]	s[3][1]
1234	56	1212	33	1434	80	1312	78
65508	65510	65512	65514	65516	65518	65520	65522

1234	56
1212	33
1434	80
1312	78

Processing 2-D Arrays

- For **processing 2-D arrays**, we use **two nested for loops**.
- The **outer for loop** corresponds to the **row** and the **inner for loop** corresponds to the **column**.
- int arr[4][5];
- (i) **Reading values in arr**

```
for( i = 0; i < 4; i++ )  
  
for( j = 0; j < 5; j++ ).  
  
scanf("%d ", &arr[i][j]);
```

- (ii) **Displaying values of arr**

```
for( i = 0; i < 4; i++ )  
  
for( j = 0; j < 5; j++ )  
  
printf( "%d ", arr[i][j]);
```

Program to input and display a matrix

```
#define ROW 3
#define COL 4
main ( )
{
    int mat [ROW] [COL], i, j;
    printf ("Enter the elements of matrix (%dx%d) row-wise ; \n" ,ROW, COL) ;
    for(i=0;i<ROW;i++)
        for(j=0;j<COL;j++)
            scanf ("%d", &mat[i] [j]);
    printf ("The matrix that you have entered is ; \n") ;
    for(i=0;i<ROW;i++)
    {
        for(j=0;j<COL;j++)
            printf("%5d",mat[i] [j]);
        printf("\n") ;
    }
}
```

Pointers and 2-Dimensional Arrays

- Each **row** of a **two-dimensional array** can be thought of as a **one-dimensional array**.
- This is a **very important fact** if we wish to **access array elements** of a **two-dimensional array** using **pointers**.
- Thus, the **declaration**, `int s[5][2]` ; can be thought of as setting up **an array of 5 elements**, each of which is a **one-dimensional array** containing **2 integers**.
- We refer to an **element of a one-dimensional array** using a **single subscript**.
- Similarly, if we can imagine **s** to be a **one-dimensional array** then we can refer to its **zeroth element** as **s[0]**, the next element as **s[1]** and so on.
- More specifically, **s[0]** gives the **address of the zeroth one-dimensional array**, **s[1]** gives the **address of the first one dimensional array** and so on.

2-D array as an array of 1-D arrays

```
int main( )
{
    int s[4][2] = {
        { 1234, 56 },
        { 1212, 33 },
        { 1434, 80 },
        { 1312, 78 }
    };
    int i ;
    for ( i = 0 ; i <= 3 ; i++ )
        printf ( "\nAddress of %d th 1-D array = %u", i, s[i] ) ;
```

}

OUTPUT:

Address of 0 th 1-D array = 65508
Address of 1 th 1-D array = 65512
Address of 2 th 1-D array = 65516
Address of 3 th 1-D array = 65520

2-D array as an array of 1-D arrays

- The **compiler** knows that **s** is an array containing **4 one-dimensional arrays**, each containing **2 integers**.
- Each **one-dimensional array** occupies **4 bytes** (two bytes for each integer).
- These **one-dimensional arrays** are **placed linearly** (zeroth 1-D array followed by first 1-D array, etc.).
- Hence each **one-dimensional arrays** starts **4 bytes** further along than the **last one**, as can be seen in the **memory map** of the **array** shown below:

s[0][0]	s[0][1]	s[1][0]	s[1][1]	s[2][0]	s[2][1]	s[3][0]	s[3][1]
1234	56	1212	33	1434	80	1312	78
65508	65510	65512	65514	65516	65518	65520	65522

2-D array as an array of 1-D arrays

- The expressions **s[0]** and **s[1]** would yield the addresses of the **zeroth(65508)** and **first(65512)** one-dimensional array respectively.
- Suppose we want to refer to the element **s[2][1]** using **pointers**.
- We know that **s[2]** would give the address **65516**, the **address of the second one-dimensional array**.
- Obviously (**65516 + 1**) would give the address **65518**.
- Or (**s[2] + 1**) would give the address **65518**.
- And the **value at this address** can be obtained by using the **value at address operator**, saying ***(s[2] + 1)**.
- A one-dimensional arrays that **num[i]** is same as ***(num + i)**.
- Similarly, ***(s[2] + 1)** is same as, ***(*(s + 2) + 1)**.

2-D array as an array of 1-D arrays

- Thus, all the following **expressions** refer to the **same element**:

- $s[2][1] = * (s[2] + 1) = * (* (s + 2) + 1)$

```
main( )
```

```
{
```

```
int s[4][2] = {{ 1234, 56 },  
                { 1212, 33 },  
                { 1434, 80 },  
                { 1312, 78 }};
```

```
int i, j;
```

```
for ( i = 0 ; i <= 3 ; i++ )
```

```
{
```

```
    printf ( "\n" );
```

```
    for ( j = 0 ; j <= 1 ; j++ )
```

```
        printf ( "%d ", *( *( s + i ) + j ) );
```

```
}
```

```
}
```

Pointer to an Array

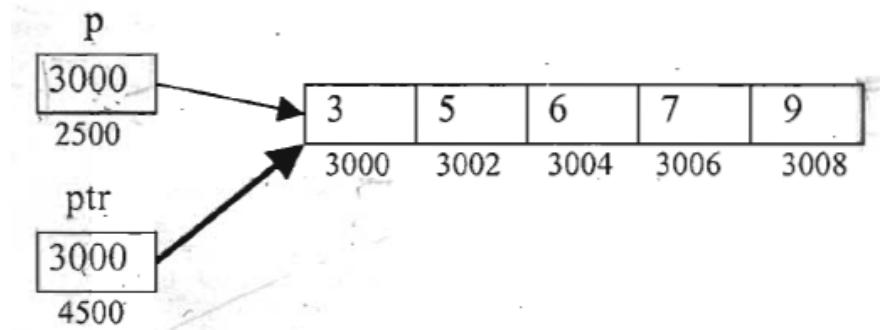
- If we can have a **pointer to an integer**, a **pointer to a float**, a **pointer to a char**, then can we not have a **pointer to an array**?

```
main( )
{
    int s[5][2] = {{ 1234, 56 },
                    { 1212, 33 },
                    { 1434, 80 },
                    { 1312, 78 }} ;
    int ( *p )[2];           //p is a pointer to an array of two integers
    int i, j, *pint;
    for ( i = 0 ; i <= 3 ; i++ )
    {
        p = &s[i];
        pint =(int *) p ;
        printf ( "\n" );
        for ( j = 0 ; j <= 1 ; j++ )
            printf ( "%d ", *( pint + j ) );
    }
}
```

Note: The entity pointer to an array is immensely useful when we need to pass a 2-D array to a function.

Pointer to an Array

```
void main()
{
    int *p; /*Can point to an integer*/
    int (*ptr)[5]; /*Can point to an array of 5 integers*/
    int arr[5];
    p=arr; /*Points to 0th element of arr*/
    ptr=arr; /*Points to the whole array arr*/
    printf("p = %u, ptr = %u \n",p,ptr);
    p++;
    ptr++;
    printf("p = %u, ptr = %u \n",p,ptr);
}
```



Output:

p = 3000 ptr = 3000

p = 3002 ptr = 3010,

Passing 2-D Array to a Function

```
void main( )
{
    int a[3][4] = {
        1, 2, 3, 4,
        5, 6, 7, 8,
        9, 0, 1, 6
    };
    show ( a, 3, 4 ) ;
}
```

Method 1

Using pointer to array

```
show ( int ( *q )[4], int row, int col )
{
    int i, j ;
    for ( i = 0 ; i < row ; i++ )
    {
        for ( j = 0 ; j < col ; j++ )
            printf ( "%d ", *( ( q + i )+j ) );
            //printf ( "%d ", q[i][j] );
            printf ( "\n" );
    }
}
```

Passing 2-D Array to a Function

```
void main( )
{
    int a[3][4] = {
        1, 2, 3, 4,
        5, 6, 7, 8,
        9, 0, 1, 6
    };
    print( a, 3, 4 );
}
```

Method 2

Using pointer to array

```
print( int q[ ][4], int row, int col )
{
    int i, j;
    for ( i = 0 ; i < row ; i++ )
    {
        for ( j = 0 ; j < col ; j++ )
            printf ( "%d ", q[i][j] );
        printf ( "\n" );
    }
}
```

Passing 2-D Array to a Function

```
void main( )
{
    int a[3][4] = {
        1, 2, 3, 4,
        5, 6, 7, 8,
        9, 0, 1, 6
    };
    display(a,3,4);
}
```

Method 3

Using single pointer

```
display ( int *q, int row, int col )
{
    int i, j;
    for ( i = 0 ; i < row ; i++ )
    {
        for ( j = 0 ; j < col ; j++ )
            printf ( "%d ", * ( q + i * col + j ) );
        printf ( "\n" );
    }
    printf ( "\n" );
}
```

Array of Pointers

- The way there can be an **array of int** or an **array of float**, similarly there can be an **array of pointers**.
- Since a **pointer variable** always **contains an address**, an **array of pointers** would be nothing but a **collection of addresses**.
- The **addresses** present in the **array of pointers** can be **addresses of isolated variables** or **addresses of array elements** or any other addresses.
- All rules that apply to an ordinary array apply to the **array of pointers** as well.

Array of Pointers

```
main( )
```

```
{
```

```
int *arr[4] ; /* array of integer pointers */
```

```
int i = 31, j = 5, k = 19, l = 71, m ;
```

```
arr[0] = &i ;
```

```
arr[1] = &j ;
```

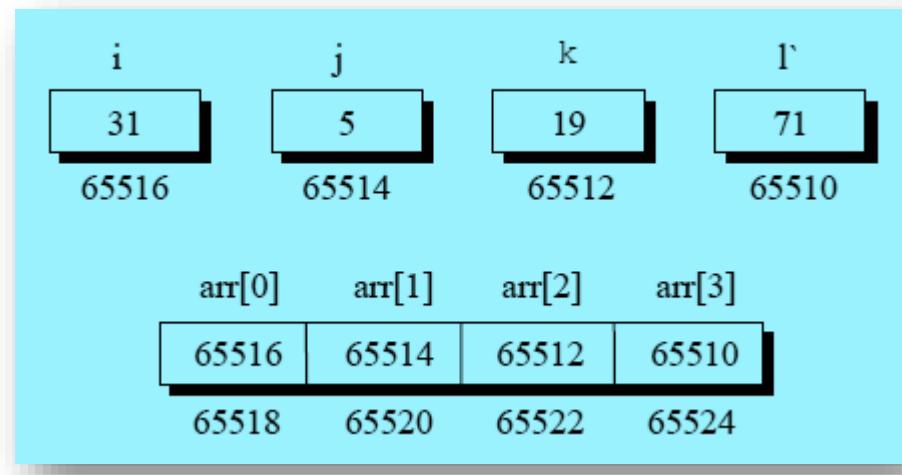
```
arr[2] = &k ;
```

```
arr[3] = &l ;
```

```
for ( m = 0 ; m <= 3 ; m++ )
```

```
printf ( "%d ", * ( arr[m] ) ) ;
```

```
}
```



Array of Pointers

- An **array of pointers** can even contain the **addresses of other arrays**.
- The following program would justify this.

```
main( )  
{  
    static int a[ ] = { 0, 1, 2, 3, 4 } ;  
  
    int *p[ ] = { a, a + 1, a + 2, a + 3, a + 4 } ;  
  
    printf ( "\n%u %u %d", p, *p, * ( *p ) ) ;  
}
```

OUTPUT:

6487584 4206608 0

Sorting Array Elements

- **Sorting** an array means **arranging the elements** of the array in a certain order either **increasing** or **decreasing** order.
- **Popular Sorting Algorithms**
 - *Bubble Sort*
 - *Selection Sort*
 - *Insertion Sort*
 - *Merge Sort*
 - *Quick Sort*
 - *Heap Sort*

Bubble Sort

- The **Bubble Sort algorithm** works by **comparing** the **adjacent elements** and **swapping** them if they are in the wrong order.
- In this **algorithm**, the **largest element** "**bubbles up**" to the **end** of the **array in each iteration**.

Bubble Sort

- **First Pass**

- Compare **first** and **second** elements



No Swap since $5 < 8$.

- Compare **second** and **third** elements



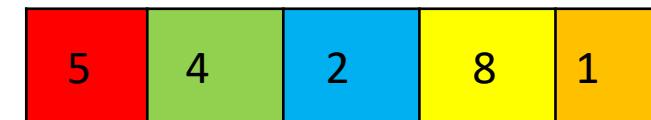
Swap since $8 > 4$.

- Compare **third** and **fourth** elements

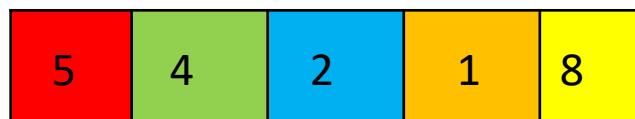


Swap since $8 > 2$

- Compare **fourth** and **fifth** elements



Swap since $8 > 1$



Bubble Sort

- **Second Pass**
- Compare **first** and **second** elements



- Compare **second** and **third** elements

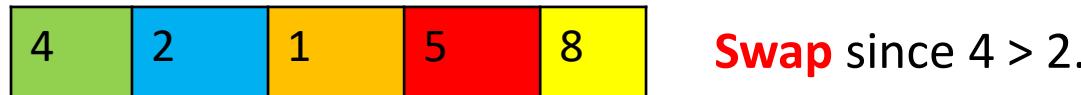


- Compare **third** and **fourth** elements



Bubble Sort

- **Third Pass**
- Compare **first** and **second** elements



- Compare **second** and **third** elements



Bubble Sort

- **Fourth Pass**
- Compare **first** and **second** elements

-  **Swap** since $2 > 1$.



- **Array** elements are now **sorted**.

Implementation of Bubble Sort

```
int main() {  
    int arr[] = {5,1,4,2,8};  
    int i, j;  
    int n = sizeof(arr) / sizeof(arr[0]);  
    for (i = 0; i < n - 1; i++)  
        for (j = 0; j < n - i - 1; j++) {  
            if (arr[j] > arr[j + 1]) {  
                int temp = arr[j];  
                arr[j] = arr[j + 1];  
                arr[j + 1] = temp;  
            }  
        }  
    printf("Sorted array: ");  
    for (i = 0; i < n; i++)  
        printf("%d ", arr[i]);  
}
```

Lecture 12

String in C

Dr. Vandana Kushwaha

Department of Computer Science
Institute of Science, BHU, Varanasi

Introduction

- The way a **group of integers** can be **stored** in an **integer array**, similarly a **group of characters** can be **stored** in a **character array**.
- Character arrays** are many a time also called **strings**.
- Character arrays** or **strings** are used by programming languages to manipulate **text** such as **words** and **sentences**.
- A **String** is a **one-dimensional array of characters** terminated by a **null character** '**\0**'.
- For example, **char name[] = { 'H', 'A', 'E', 'S', 'L', 'E', 'R', '\0' };**
- Each **character** in the **array** occupies **one byte of memory** and the **last character** is always '**\0**' is called **null character**.
- ASCII value** of '**\0**' is **0**.

H	A	E	S	L	E	R	\0
65518	65519	65520	65521	65522	65523	65524	65525

Introduction

- The **string** can also be initialized as:

```
char name[ ] = "HAESLER" ;
```

- Note that, in this **declaration** '**\0**' is not necessary as **C** inserts the **null character** automatically.

```
/* Program to demonstrate printing of a string */
```

Method 1:

```
main( )
{
    char name[ ] = "Klinsman" ;
    int i = 0 ;
    while ( i <= 7 )
    {
        printf ( "%c", name[i] ) ;
        i++ ;
    }
}
```

Program to demonstrate printing of a String

Method 2:

```
main( )
{
    char name[ ] = "Klinsman" ;
    int i = 0 ;
    while ( name[i] != '\0' )
    {
        printf ( "%c", name[i] ) ;
        i++ ;
    }
}
```

Program to demonstrate printing of a String

Method 3:

```
main( )
{
    char name[ ] = "Klinsman" ;
    char *ptr ;
    ptr = name ; // store base address of string
    while ( *ptr != '\0' )
    {
        printf ( "%c", *ptr ) ;
        ptr++ ;
    }
}
```

Program to demonstrate printing of a string

- Method 4:

```
main( )  
{  
    char name[ ] = "Klinsman" ;  
  
    printf( "%s", name ) ;  
}
```

- The **%s** used in **printf()** is a **format specification** for printing out a **string**.

Program to demonstrate input of a String

- **scanf()** function can be used to **read(input)** a **string**.

```
main( )
{
    char name[25];
    printf ( "Enter your name " );
    scanf ( "%s", name );
    printf ( "Hello %s!", name );
}
```

- The declaration **char name[25]** sets aside **25 bytes** under the array **name[]**, whereas the **scanf()** function fills in the **characters** typed at **keyboard** into this **array** until the **enter key is hit**.
- Once **enter** is hit, **scanf()** places a '**\0**' in the **array**.
- We pass the **base address** of the **array** to the **scanf()** function.

Limitations of scanf()

- While entering the **string** using **scanf()** we must be **cautious** about few **things**:
 - The **length of the string** should **not exceed** the **dimension** of the **character array**.
 - This is because the **C compiler** doesn't perform **bounds checking** on **character arrays**.
 - Hence, if you **carelessly exceed the bounds** there is always a **danger of overwriting** something important.
 - **scanf()** is **not capable** of receiving **multi-word strings**.
 - Therefore names such as '*Debashish Roy*' would be **unacceptable**.
 - The way to get around this **limitation** is by using the function **gets()**.

gets() and puts()

- The usage of functions **gets()** and its counterpart **puts()** is shown below.

```
main( )
```

```
{
```

```
    char name[25] ;  
    printf ( "Enter your full name " ) ;  
    gets ( name ) ;  
    puts ( "Hello!" ) ;  
    puts ( name ) ;  
}
```

- OUTPUT:**

Enter your name Debashish Roy

Hello!

Debashish Roy

gets() and puts()

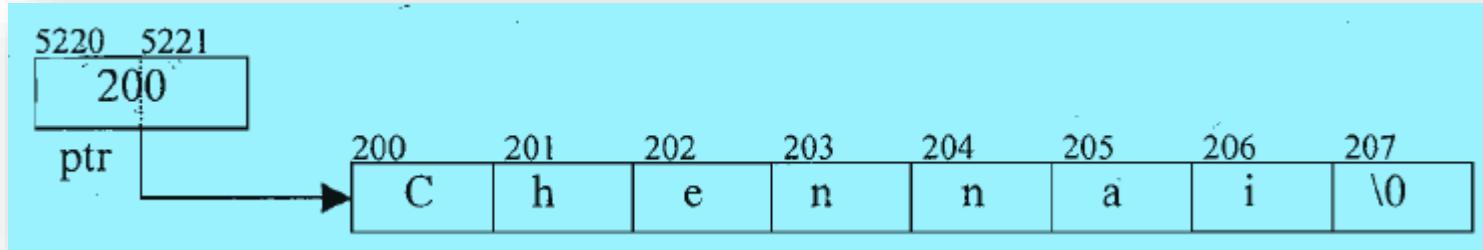
- **puts()** can **display only one string** at a time.
- **puts()** places the cursor on the **next line after printing the string**.
- Though **gets()** is **capable of receiving only one string at a time**, the plus point with **gets()** is that it can receive a **multi-word string**.

Pointers and Strings

- A string can be declared using character pointer as:

```
char *ptr = "Chennai" ;
```

- Here pointer 'ptr' will contain the **base address** of **string**.



- So in this case total **10 bytes** are reserved, **2 bytes** for the **pointer variable** and **8 bytes** for the **string**.
- We cannot assign a string to another, whereas, we can assign a char pointer to another char pointer.
- Also, once a string has been defined it cannot be initialized to another set of characters but such an operation is perfectly valid with char pointers.

Pointers and Strings

Example 1

```
main( )
{
    char str1[ ] = "Hello" ;
    char str2[10] ;
    char *s = "Good Morning" ;
    char *q ;
    str2 = str1 ; /* error */
    q = s ; /* works */
}
```

Pointers and Strings

Example2

```
main( )  
{  
    char str1[ ] = "Hello" ;  
  
    char *p = "Hello" ;  
  
    str1 = "Bye" ; /* error */  
  
    p = "Bye" ; /* works */  
}
```

Standard Library Functions for String

- With every **C compiler** a large set of **useful string handling library functions** are provided using header file **string.h**.
- strlen() function:**
- This function **returns the length of the string** i.e. the **number of characters** in the **string excluding the terminating null character ('\0')**.

```
main( )
{
    char arr[ ] = "Bamboozled" ;
    int len1, len2 ;
    len1 = strlen ( arr ) ;
    len2 = strlen ( "Humpty Dumpty" ) ;
    printf ( "\nstring = %s length = %d", arr, len1 ) ;
    printf ( "\nstring = %s length = %d", "Humpty Dumpty", len2 ) ;
}
```

OUTPUT:

string = Bamboozled length = 10

string = Humpty Dumpty length = 13

Implementation of strlen()

- The function **xstrlen()** keep counting the **characters** till the **end of string** is not met.
- Or in other words **keep counting characters till the pointer s doesn't point to '\0'.**

```
int xstrlen ( char *s )
{
    int length = 0 ;
    while ( *s != '\0' )
    {
        length++ ;
        s++ ;
    }
    return ( length ) ;
}
```

strcpy() function

- This function **copies the contents** of one **string** into another.
- The **base addresses** of the **source** and **target strings** should be **supplied** to this function.

```
main( )
```

```
{  
  
    char source[ ] = "Sayonara" ;  
  
    char target[20] ;  
  
    strcpy ( target, source ) ;  
  
    printf ( "\nsource string = %s", source ) ;  
  
    printf ( "\ntarget string = %s", target ) ;  
  
}
```

OUTPUT:

source string = Sayonara

target string = Sayonara

strcpy() function

- **strcpy()** goes on **copying the characters in source string into the target string till it doesn't encounter the end of source string ('\0').**
- It is **our responsibility** to see to it that the **target string's dimension is big enough to hold the string** being copied into it.
- Thus, a **string gets copied** into another **character by character**.
- There is **no short cut** for this.

Implementation of strcpy() function

```
xstrcpy( char *t, char *s )
```

```
{  
    while ( *s != '\0' )  
    {  
        *t = *s ;  
        s++ ;  
        t++ ;  
    }  
    *t = '\0' ;  
}
```

- Note that **having copied** the **entire source string** into the **target string**, it is **necessary to place a '\0'** into the **target string**, to mark its end.

strcat() function

- This **function concatenates the source string at the end of the target string.**

```
main( )  
{
```

```
    char source[ ] = "Folks!" ;  
    char target[30] = "Hello" ;  
    strcat ( target, source ) ;  
    printf ( "\nsource string = %s", source ) ;  
    printf ( "\ntarget string = %s", target ) ;  
}
```

OUTPUT:

```
source string = Folks!  
target string = HelloFolks!
```

- Note that the **target string** has been made **big enough to hold the final string.**

Implementation of strcat() function

```
char *xstrcat(char *str1,char *str2)
{
    while(*str1!='\0') /*Check for the end of first string*/
        str1++;
    while(*str2!='\0') /*Add second string at the end of first*/
    {
        *str1=*str2;
        str1++;
        str2++;
    }
    *str1='\0';
    return str1;
}
```

- The function **xstrcat()** returns a **pointer** to the **first string**.

strcmp() function

- This is a **function** which **compares two strings** to find out whether they are **same or different**.
- The **two strings** are **compared character by character** until there is a **mismatch** or **end of one of the strings** is reached, whichever occurs first.
- If the **two strings** are **identical**, **strcmp()** returns a **value zero**.
- If they are not, it returns the numeric difference between the ASCII values of the **first non-matching pairs of characters**.
- Thus the function **strcmp(s1, s2)** returns:
 - i. a value < 0 when **s1 < s2**
 - ii. a value = 0 when **s1= = s2**
 - iii. a value > 0 when **s1 > s2**

strcmp() function

```
main( )
```

```
{
```

```
    char string1[ ] = "Jerry" ;
```

```
    char string2[ ] = "Ferry" ;
```

```
    int i, j, k ;
```

```
    i = strcmp ( string1, "Jerry" ) ;
```

```
    j = strcmp ( string1, string2 ) ;
```

```
    k = strcmp ( string1, "Jerry boy" ) ;
```

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

```
}
```

OUTPUT:

0 4 -32

Implementation of strcmp() function

```
int xstrcmp(char *str1,char *str2)
{
    while(*str1]!='\0' && *str2!='\0' && *str1==*str2)
    {
        str1++;
        str2++;
    }
    if(*str1==*str2)
        return 0;
    else
        return(*str1 - *str2);
}
```

Two-Dimensional Array of Characters

- Strings are **character arrays** so **array of strings** means **array of character type arrays** i.e. **a two dimensional array of characters**.
- Suppose we declare and initialize a **two-dimensional array of characters** as
`char arr[5][10] = { "white", "red", "green", "yellow", "blue" };`
- Here the **first subscript** of array denotes **number of strings** in the array and **second subscript** denotes the **maximum length** that each string can have.
- The **space reserved for this two-dimensional array is 50 bytes**.

2000	w	h	i	t	e	\0				2009
2010	r	e	d	\0						2019
2020	g	r	e	e	n	\0				2029
2030	y	e	l	l	o	w	\0			2039
2040	b	l	u	e	\0					2049

Two-Dimensional Array of Characters

- In internal storage representation of array of strings, **2000** is the **base address** of the **first string**.
- Similarly, **2010** is the **base address** of the **second string**.
- Here **10 bytes** are **reserved in memory** for **each string**.
- We can see that **first string** takes **only 6 bytes**, so **4 bytes are wasted**.
- Similarly **2nd string** takes only **4 bytes** and **6 bytes are wasted**.
- The **total number of bytes** occupied by the array is **50** while the strings use only **28 bytes** so **22 bytes of memory is wasted**.

Two-Dimensional Array of Characters

```
void main()
{
    char arr[ 5 ] [10] ={ "white", "red" , "green", "yellow" , "blue" };
    int i;
    for(i=0;i<5;i++)
    {
        printf("String = %s \t",arr[i] );
        printf("Address of string = %u \n", arr[i]);
    }
}
```

Output:

String = white Address of string = 2000

String = red Address of string = 2010

String = green Address of string = 2020

String = yellow Address of string= 2030 .

String = blue Address of string = 2040

Two-Dimensional Array of Characters

- Suppose we **don't initialize** it at the **time of declaration** and try to **assign strings** to it afterwards like this:

```
arr[0] = "white"; /*Invalid*/
```

```
arr[1] = "red"; /*Invalid*/
```

- Similarly we **cannot write**:

```
arr[0] = arr[ 1]; /*Invalid*/
```

- If we want to **assign values to strings** we will have to use **strcpy()** or **scanf()** function.

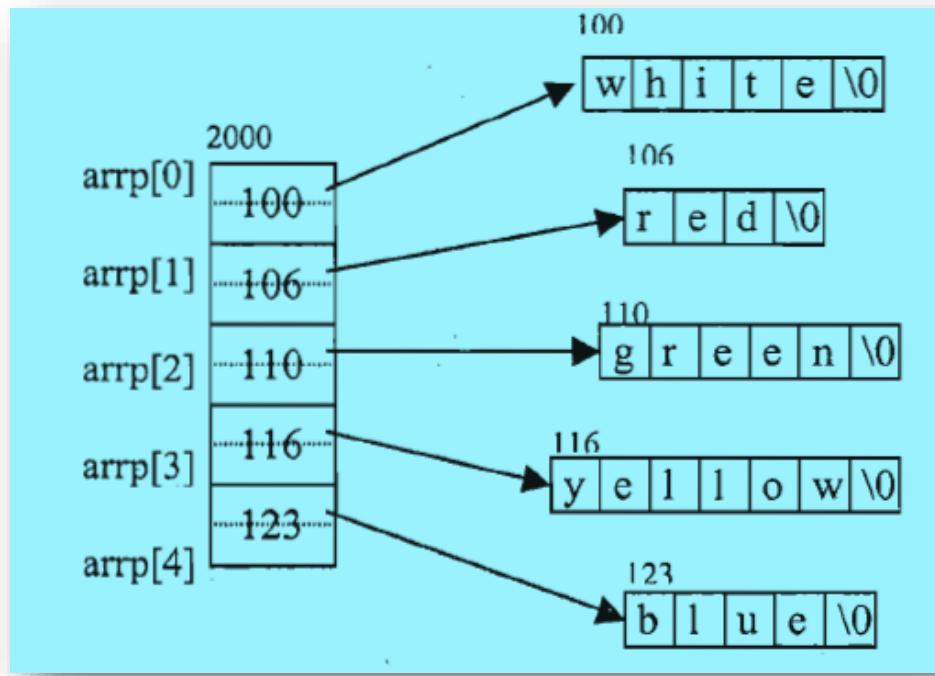
```
strcpy( arr[0], "white"); /*Valid*/
```

```
scanf(" %s", arr[0]); /*Valid*/
```

```
strcpy(arr[0], arr[1]); /*Valid*/
```

Array of pointers to strings

- **Array of pointers to strings** is an **array of char pointers** where each **pointer** points to the **first character** of a **String** i.e. each element of this **array** contains the **base address** of a string.
- `char *arrp [] = { "white", "red", "green" , "yellow" , "blue" };`



Array of pointers to strings

- Here all strings occupy **28 bytes** and **10 bytes** are occupied by the **array of pointers**.
- So the total byte occupied are **38 bytes**.
- In **two-dimensional array**, total bytes occupied were **50 bytes** for these same strings.
- So we can see here saving of **12 bytes**.

```
main( )
{
    int i;

    char *arrp [ ] = { "white", "red", "green" , "yellow" , "blue" };
    for(i=0;i<5;i++)
    {
        printf ("String %s \t" , arrp[ i ] );
        printf("Address of string %u\t" ,arrp[i] );
    }
}
```

Array of pointers to strings

- Instead of **initializing** we could have **assigned strings after declaration** as:

```
arrp[0] = "white"; /*Valid*/
```

```
arrp[ 1] = "red"; /*Valid*/
```

- Here it is **valid** because **arrp[0]** is a **pointer variable** and it can be assigned **address of any string**.
- This **type of assignment** was **invalid** when we used **2-D array**.
- When we are using an **array of pointers to strings** we can **initialize the strings at the place** where we are **declaring the array**, but we **cannot receive the strings from keyboard** using **scanf()**.
- We have to **first allocate memory** through **malloc() function**.

Array of pointers to strings

- Thus, the following **program** would **never work out**.

```
main( )
{
    char *names[6];
    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        printf ( "\nEnter name " );
        scanf ( "%s", names[i] );
    }
}
```

- The **program doesn't work** because; when we are **declaring the array** it is **containing garbage values**.
- And it would be definitely wrong to send these **garbage values** to **scanf()** as the **addresses** where it should keep the strings received from the keyboard.

Example

```
void main()
{
    char arr[5] [10];
    char *arrp[5];
    arr[0]="January"; /*Invalid*/
    arrp[0]="January"; /*Valid*/
    strcpy(arr[1] , "February"); /*Valid*/
    strcpy(arrp[1] , "February"); /*Invalid*/
    scanf ( "% s" , arr[ 2]); /*Valid*/
    scanf("%s",arrp[2] ); /*Invalid*/
    arrp[3]=(char *)malloc(10);
    strcpy(arrp[3] , "March"); /*Valid*/
    arrp[4]=(char *)malloc(10);
    scanf("%s",arrp[4] ); /*Valid*/
}
```

Lecture 13

Storage Class in C

Dr. Vandana Kushwaha

Department of Computer Science
Institute of Science, BHU, Varanasi

Introduction

- In addition to **data type**, each **variable** has one more attribute known as **storage class**.
- The proper use of storage classes makes our program **efficient** and **fast**.
- We can **specify** a storage class while **declaring** a variable.
- The general **Syntax** is:

```
storage_class datatype variable_name;
```

- There are **four types of storage classes**:
 1. **Automatic**
 2. **External**
 3. **Static**
 4. **Register**
- The **keywords auto, static, register, extern** are used for these **storage classes**.

Storage Class

- So we may write **declaration statements** like this:

```
auto int x, y;
```

```
static float d;
```

```
register int z;
```

- When the **storage class specifier** is not present in the **declaration**, compiler assumes a **default storage class** based on the place of declaration.
- **Storage class decides about these four aspects of a variable-**
 - **Lifetime** - Time between the **creation** and **destruction** of a variable.
 - **Scope** - Locations where the variable is **available** for use.
 - **Initial value** - Default value taken by an **uninitialized** variable.
 - **Place of storage** - Place in **memory** where the storage is allocated for the variable.

Automatic storage class

- The **variables** declared inside a **block/function** without any **storage class specifier** are called **automatic variables**.
- We may also use the **keyword auto** to declare **automatic variables**, although this is generally not done.
- The following two **declaration statements** are equivalent and both declare **a** and **b** to be **automatic variables** of type **int**.

```
func( )  
{  
    int a,b;  
}  
func( )  
{  
    auto int a,b;  
}
```

Automatic storage class

- The **uninitialized automatic variables** initially contain **garbage value**.
- The **scope** of these **variables** is **inside function or block** in which they are declared and they can't be used in another **function/block**.

```
main( )  
{  
func( );  
func( );  
func( );  
}  
  
func( )  
{  
int, x=2,y=5;  
printf("x=%d,y=%d",x,y) ;  
x++;  
y++;  
}
```

OUTPUT:

x = 2, y = 5

x = 2, Y = 5

x = 2, y = 5

Automatic storage class

- Since **automatic variables** are known **inside a function or block** only, so we can have **variables of same name** in **different functions or blocks** without any **conflict**.

```
main ( )
{
int x=5;
printf ("x %d\n",x);
func ( );
}
func ( )
{
int x=15;
printf ("x %d\t",x);
}
```

Output:

x = 5 x = 15

- Here the **variable x** declared inside **main()** is **different** from the **variable x** declared inside the **function func()**.

Automatic storage class

- There are **different blocks** inside **main()**, and the **variable x** declared in **different blocks** is **different**.

```
void main ( )  
{  
    int x=3;  
    printf("%d\t",x) ;  
    {  
        int x=10;  
        printf("%d\t",x);  
    }  
    {  
        int x=26;  
        printf("%d\t",x);  
    }  
    printf("%d\n",x);  
}
```

OUTPUT:

3 10 26 3

External Storage Class

- The variables that have to be used by many functions and different files can be declared as **external variables**.
- The initial value of an **uninitialized external variable** is zero.

Difference between Definition and Declaration.

- The **declaration** of an **external variable** declares the **storage class, data type** and **name** of the **variable**.
- The **declaration** will be written as: **extern float salary;**
- The **definition** of an **external variable** defines the **data type** and **name** of the **variable**.
- While **definition reserves storage** for the **variable**.
- The **definition** of an **external variable** can be written as: **float salary;**

Definition of External Variables

- **Definition** creates the variable, so **memory is allocated** at the time of **definition**.
- There can be only **one definition**.
- The **variable** can be **initialized** with the **definition** and **initializer** should be **constant**.
- The keyword **extern** is **not specified** in the **definition**.
- The **definition** can be written only **outside functions**.

Declaration of External Variables

- The **declaration** does not create the variable, it only **refers** to a **variable** that has already been created somewhere, so **memory is not allocated** at the time of **declaration**.
- There can be **many declarations**.
- The variable **cannot be initialized** at the time of **declaration**.
- The **keyword extern** is always specified in the **declaration**.
- The **declaration** can be placed inside functions also.

Example1

```
int x=8; //external variable definition
```

```
main ( )
```

```
{
```

```
.....
```

```
}
```

```
func1( )
```

```
{
```

```
.....
```

```
}
```

```
func2( )
```

```
{
```

```
.....
```

```
}
```

- In this program the **variable x** will be **available to all the functions**, since an **external variable is active** from the **point of its definition** till the **end** of a program.

Example2

- Now if we **change the place of definition** of **variable x** like this-

```
main( )  
{  
.....  
}  
func1( )  
{  
}  
int x=8; //external variable definition
```

```
func2 ( )  
{  
}
```

- Now **x** is **defined after main() and func1()**, so it **can't be used by these functions**, it is **available only** to function **func2().**

Example3

- Suppose we want to use this **variable** in function **main()** then we can **place declaration** in this **function** like this-

```
main( )
{
    extern int x; //external variable declaration
    .....
}

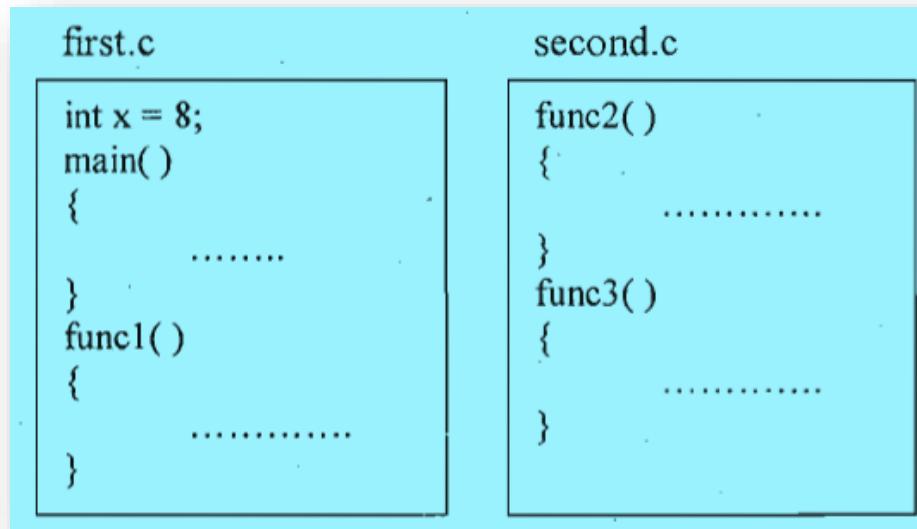
func1( )
{
    .....
}

int x=8; //external variable definition
func2()
{
    .....
}
```

- Now **x** will be **available** to **functions func2()** and **main()**.

External Storage Class for Multi file Program

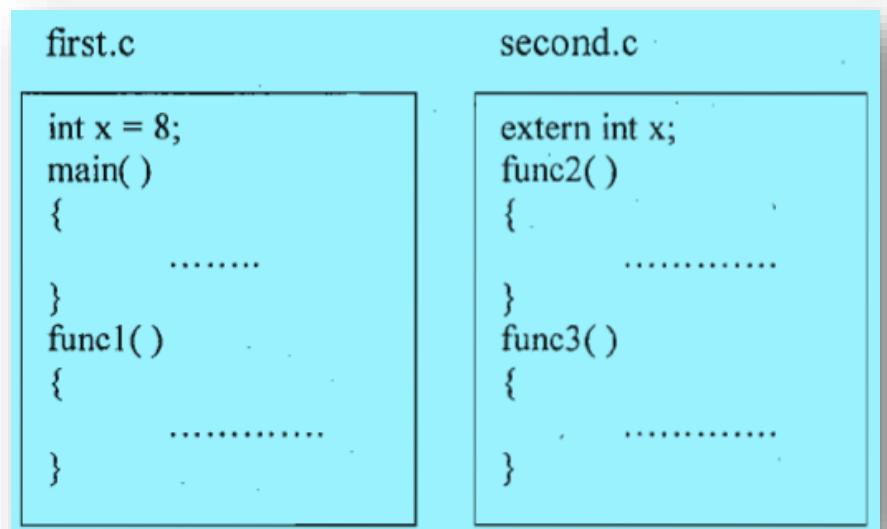
- When the **program** is **large** it is written in **different files** and these files are **compiled separately** and **linked together** afterwards to form an **executable program**.
- Now we will consider a **multi file program** which is **written in two files** viz. **first.c** and **second.c**



- Here in the file **first.c**, an **external variable x** is **defined** and **initialized**.
- This **variable** can be **used** both in **main()** and **func1()** but it is **not accessible** to **other files**.

External Storage Class for Multi file Program

- Suppose file **second.c** wants to **access** this **variable** then we can **put** the **declaration** in this file as:



- Now this **variable** can be **accessed** and **modified** in files **first.c** and **second.c** and any changes made to it will be visible in **both files**.
- If this variable is needed by **only one function** in the file **second.c**, then the **extern declaration** can be **put inside** that **function**.
- So the **declaration** of an **external variable** using **extern keyword** is used to **extend** the **scope** of that **variable**.

Example

main.c

```
#include <stdio.h>

float area(float x);

float pi=3.14; //definition

main()

{

    float r;

    printf("Enter the radius");

    scanf("%f",&r);

    printf("The area is %.2f\n", area(r));

}
```

area.c

```
extern float pi; //declaration

float area(float a)

{

    return (pi*a*a);

}
```

C:\Dev-Cpp\MinGW64\bin>gcc main.c area.c

C:\Dev-Cpp\MinGW64\bin>a.exe

Enter the radius 4.5

The area is 63.58

Static Storage Class

- The **scope** of a **local static variable** is same as that of an **automatic variable** i.e. it can be used **only inside the function or block** in which it is **defined**.
- The **lifetime of a static variable is more than that of an automatic variable**.
- A **static variable** is **created at the compilation time** and it **remains alive till the end of a program**.
- It is **not created and destroyed each time** the control enters a function/block.
- Hence a **static variable** is **created only once** and **its value is retained** between **function calls**.
- If it has been **initialized**, then the **initialization value** is placed in it **only once** at the **time of creation**.
- It is **not initialized each time** the function is called.

Static Storage Class

- A **static variable** can be **initialized** only by **constants** or **constant expressions**.
- Unlike **automatic variables**, we can't use the values of **previously initialized variables** to initialize static variables.
- If a **static variable** is **not explicitly initialized** then **by default** it takes **initial value zero**.

```
int x = 8;
```

```
int y = x; /*Valid*/
```

```
static int z = x; /*Invalid, initializer should be constant*/
```

Example 1: Static Storage Class

```
main( )
{
    func ( );
    func ( );
    func ( );
}

func ( )
{
    static int x=2, y=5;
    printf("x=%d,y=%d\n",x,y) ;
    x++;
    y++;
}
```

OUTPUT:

x = 2, y = 5

x = 3, y = 6

x = 4, y = 7

Example 2: Static Storage Class

```
main( )  
{  
    int n,i;  
    printf {"Enter a number :"};  
    scanf ("%d" ,&n) ;  
    for(i=1;i<=10;i++)  
        func(n);  
    printf("\n") ;  
}  
func (int n)  
{  
    static int step; /* Automatically initialized to 0 * /  
    step=step+n;  
    printf("%d\t",step);  
}
```

OUTPUT:

```
Enter the number : 4  
4 8 12 16 20 24 28 32 36 40
```

Register Storage Class

- Register storage class can be applied only to local variables.
- The scope, lifetime and initial value of register variables are same as that of automatic variables.
- The only difference between the two is in the place where they are stored.
- Automatic variables are stored in memory while register variables are stored in CPU registers.
- Registers are small storage units present in the processor.
- The variables stored in registers can be accessed much faster than the variables stored in memory.
- So the variables that are frequently used can be assigned register storage class for faster processing.
- For example the variables used as loop counters may be declared as register variables since they are frequently used.

Register Storage Class

- Example

```
main()
{
    register int i;
    for(i=0;i<20000;i++)
        printf("%d\n",i);
}
```

- Register variables don't have memory addresses so we can't apply address operator(&) to them.
- There are limited number of registers in the processor hence we can declare only few variables as register.
- If many variables are declared as register and the CPU registers are not available then compiler will treat them as automatic variables.

Register Storage Class

- The **CPU registers** are generally of **16 bits**, so we can specify **register storage class** only for **int, char** or **pointer types**.
- If a **variable** other than these **types** is declared as **register variable** then **compiler treat** it as an **automatic variable**.

Summary

	Automatic	Register	Static	External
Storage	Memory	CPU registers	Memory	Memory
Default initial value	Garbage value	Garbage value	Zero	Zero
Scope	Local to the block in which the variable is defined	Local to the block in which the variable is defined.	Local to the block in which the variable is defined.	Global
Life	Till the control remains within the block in which the variable is defined	Till the control remains within the block in which the variable is defined.	Value of the variable persists between different function calls.	As long as the program's execution doesn't come to an end.

Lecture 14

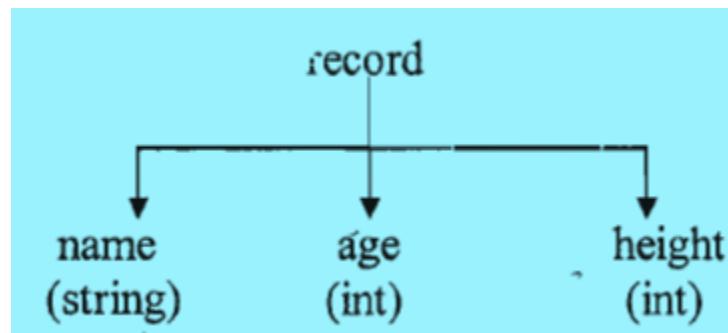
Structure in C

Dr. Vandana Kushwaha

Department of Computer Science
Institute of Science, BHU, Varanasi

Introduction

- Array is a **collection** of same type of elements but in many **real life applications** we may need to **group different types** of logically related data.
- For example if we want to create a **record of a person** to contains **name, age and height** of that **person**, then we **can't use array** because all the **three data elements** are of **different types**.



- To **store** these **related fields of different data types** we can use a **structure**, which is **capable of storing heterogeneous data**.
- Data of different types can be grouped together under a single name using **structure**.
- The **data elements** of a **structure** are referred to as **members**.

Defining a Structure

- **Definition** of a **structure** creates a **template** or **format** that describes the characteristics of its **members**.
- **General syntax of a structure definition** is:

```
struct tagname {  
    data type member1;  
    datatype member2;  
    .....  
    datatype memberN;  
};
```

- Here **struct** is a **keyword**, which tells the **compiler** that a **structure** is **being defined**.
- member1, member2.....memberN are known as **members** of the **structure** and are **declared** inside **curly brackets**.

Defining a Structure

- There should be a **semicolon(;) at the end of the curly braces.**
- These **members** can be of any **data type** like **int, char, float, array, pointers or another structure type.**
- **tagname** is the **name of the structure**, it is used further in the **program to declare variables** of this **structure type.**
- **Definition of a structure provides one more data type** in addition to the built in **data types.**
- We can **declare variables** of this **new data type** that will have the **format** of the **defined structure.**
- **Definition of a structure template does not reserve any space in memory** for the **members;**
- **Memory space is reserved only** when **actual variables** of this **structure type** are **declared.**

Defining a Structure

- Although the **syntax declaration of members** inside the template is identical to the syntax we use in declaring variables, **members are not variables**, they **don't have any existence** until they are attached with a structure .
- The **member names** inside a **structure** should be different from one another but these names similar to any other variable name declared outside the structure.
- The **member names of two different structures** may also be **same**.
- **Example:**

```
struct student{
```

```
    char name [20] ;
```

```
    int rollno;
```

```
    float marks;
```

```
};
```

Defining a Structure

- Student is the **structure tag** and there are **three members** of this **structure** :
 - name, rollno and marks.
- Structure template can be **defined globally** or **locally** i.e. it can be **placed before all functions** in the **program** or it can be **locally present** in a **function**.
- If the **template is global** then it can be used by all functions while if it is local then **only the function containing it can use it**.

Declaring Structure Variables

- By **defining** a **structure** we have **only created** a **format**, the **actual use** of **structures** will be when we **declare variables** based on this format.
- We can declare structure variables in two ways-
 - *With structure definition*
 - *Using the structure tag*

With structure definition

```
struct student{  
    char name [20] ;  
    int rollno;  
    float marks;  
}stu1,stu2,stu3;
```

- sru1, stu2 and stu3 are **variables** of type **struct student**.

Declaring Structure Variables

- **Using Structure Tag**
- We can also **declare structure variables** using **structure tag**. This can be written as

```
struct student{  
    char name [20] ;  
    int rollno;  
    float marks;  
};  
  
struct student stu1, stu2;
```
- Here **stu1** and **stu2** are **structure variables** that are **declared** using the **structure tag student**.
- **Declaring a structure variable reserves space in memory.**
- Each **structure variable** declared to be of type **struct student** has **three members** viz. *name, rollno and marks*.

Declaring Structure Variables

- The **compiler** will **reserve space for each variable sufficient to hold all the members.**
- For example each variable of type **struct student** it occupy **26 (20+2+4) bytes.**

Initialization of Structure Variables

- The **syntax of initializing structure variables** is similar to that of **arrays**.
- All the **values** are given in **curly braces** and the **number, order and type** of these values should be **same** as in the **structure template definition**.
- The **initializing values** can only be **constant expressions**.

```
struct student {  
    char name [20] ;  
    int rollno;  
    float marks;  
}stu1={"Mary",25,98};  
struct student stu2={ "John ", 24, 67. 5};
```

- Here **value of members** of stu1 will be "Mary" for name, 25 for rollno, 98 for marks.
- The **value of members** of stu2 will be "John" for name, 24 for rollno, 67.5 for marks.

Initialization of Structure Variables

- We cannot initialize members while defining the structure.

```
struct student {  
    char name [20] ;  
    int rollno;  
    float marks=99; /* Invalid */  
}stu;
```

- This is invalid because there is no variable called marks, and no memory is allocated for structure definition.
- If the number of initializers is less than the number of members then the remaining members are initialized with zero.

Initialization of Structure Variables

- For example if we have this **initialization**:

```
struct student stu1 = {"Mary"};
```

- Here the **members rollno and marks** of **stu1** will be **initialized to zero**.
- This is **equivalent** to the **initialization**:

```
struct student stu1 = {"Mary", 0, 0};
```

Accessing Members of a Structure

- For **accessing any member of a structure variable**, we use the **dot (.) operator** which is also known as the **membership operator**.
- **The format for accessing a structure member is structvariable.member**
- Here on the **left side** of the **dot** there should be a variable of structure type and on **right hand side** there should be the name of a member of that structure.
- **For example consider the following structure-**

```
struct student {  
    char name [20] ;  
    int rollno;  
    float marks;  
};  
struct student stu1,stu2;
```

name of stu1 is given by – stu1.name
rollno of stu1 is given by – stu1.rollno
marks of stu1 is given by – stu1.marks
name of stu2 is given by - stu2.name
rollno of stu2 is given by - stu2.rollno
marks of stu2 is given by - stu2.marks

Example

```
struct student{  
    char name [20] ;  
    int rollno;  
    float marks;  
};  
void main()  
{  
    struct student stu1= {"Mary" ,25,68) ;  
    struct student stu2, stu3;  
    strcpy(stu2.name, "John");  
    stu2.rollno=26;  
    stu2.marks=98;  
    printf("Enter name, roll no. and marks of stu3");  
    scanf("%s %d %f", stu3.name,&stu3.rollno,&stu3.marks);  
    printf("stu1%s %d %.2f\n",stu1.name,stu1.rollno,stu1.marks);  
    printf("stu2%s %d %. 2f\n", stu2. name, stu2.rollno, stu2.marks) ;  
    printf("stu3%s %d %. 2f\n", stu3 .name, stu3. rollno, stu3.marks);  
}
```

Assignment of Structure Variables

- We can **assign values** of a **structure variable** to **another structure variable**, if both variables are defined of the **same structure type**.
- **For example-**

```
struct student {  
    char name [20] ;  
    int rollno;  
    float marks;  
};  
main ( )  
{  
    struct student stu1={"Oliver",12,98};  
    struct student stu2;  
    stu2=stu1;  
    printf("stu1%s %d %.2f\n", stu1.name, stu1.rollno, stu1.marks) ;  
    printf("stu2%s %d %.2f\n", stu2 .name, stu2. rollno, stu2 .marks) ;  
}
```

Array of Structures

- We can declare **array of structures** where **each element of array** is of **structure type**.

- **Array of structures can be declared as:**

```
struct student stu[ 10];
```

- Here **stu** is an **array of 10 elements**, each of which is a **structure of type struct student**, means **each element of stu has 3 members**, which are name, rollno and marks.

- **These structures can be accessed through subscript notation.**

- To access the **individual members** of these **structures** we will use the **dot operator** as usual.

- **All the structures of an array are stored in consecutive memory locations.**

Storage of Structures in Memory

- The **members of structures** are stored in **consecutive memory locations**.

```
void main ( )
```

```
{
```

```
struct student {
```

```
char name [5] ;
```

```
int rollno;
```

```
float marks;
```

```
}stu;
```

```
printf ("Address of stu.name = %u \n" , stu.name) ;
```

```
printf("Address of stu.rollno = %u\n",&stu.rollno);
```

```
printf("Address of stu.marks = %u\n" ,&stu.marks);
```

```
}
```

OUTPUT:

Address of stu.name = 65514

Address of stu.rollno = 65519

Address of stu.marks = 65521

Array of Structures

```
struct student {  
    char name [20] ;  
    int rollno;  
    float marks;  
};  
void main ( )  
{  
    int i;  
    struct student stuarr [10] ;  
    for(i=0;i<10;i++)  
    {  
        printf ("Enter name, roilno and marks ");  
        scanf("%s%d%f",stuarr[i] .name,&stuarr[i] .rollno,&stuarr[i] .marks);  
    }  
    for(i=0;i<10;i++)  
        printf("%s %d %f \n",stuarr[i] .name,stuarr[i] .rollno,stuarr[i] .marks);  
}
```

Arrays Within Structures

- We can have an **array** as a **member of structure**.
- In structure student, we have taken the member name **array of characters**.
- Now we will declare another array inside the structure student.

```
struct student {  
    char name [20] ;  
    int rollno;;  
    int submarks [4] ;  
};
```

- The array **submarks** denotes the marks of students in 4 subjects.
- If **stu** is a variable of type **struct student** then-
- **stu.submarks[0]** - Denotes the marks of the student in first subject
- **stu.submarks[1]** - Denotes the marks in second subject.
- If **stuarr[10]** is an **array of type struct student** then
- **arr[0].submarks[0]** - Denotes the marks of first student in first subject
- **arr[4].submarks[3]** -Denotes the marks of fifth student in fourth subject.

Nested Structures

- The **members of a structure** can be of **any data type** including **another structure type** i.e. we can include a **structure within another structure**.
- A **structure variable** can be a **member of another structure**.
- This is called **nesting of structures**.

```
struct tag1 {  
    member1;  
    member2;  
    struct tag2 {  
        member1;  
        member2;  
    }var1;  
}var2;
```

- For accessing **member1** of inner structure we will write **var2.var1.member1**

Example Nested Structures

```
struct student {  
    char name [20] ;  
    int rollno;  
    struct date{  
        int day;  
        int month;  
        int year;  
    } birthdate;  
    float marks;  
}stu1,stu2;
```

- Here we have defined a **structure date** inside the **structure student**.
- This **structure date** has **three members** day, month, year and birthdate is a variable of type **struct date**.

Nested Structures

- We can access the **members** of **inner structure** as:

stu1.birthdate.day - day of birthdate of stu1

stu1.birthdate.month - month of birthdate of stu1

stu1.birthdate.year - year of birthdate of stu1

stu2.birthdate.day- day of birthdate of stu2

- We could have **defined it outside and declared its variables inside the structure student** using the tag.
- But **remember if we define inner structure outside, then this definition should always be before the definition of outer structure.**
- Here in this case the **date structure** should be **defined before the student structure.**

Nested Structures

```
struct date{  
    int day;  
    int month;  
    int year;  
};
```

```
struct student{  
    char name [20] ;  
    int rollno;  
    float marks;  
    struct date birthdate;  
}stu1,stu2;
```

- The advantage of defining date structure outside is that we can declare variables of date type anywhere else also.

Nested Structures

- Suppose we define a **structure teacher**, then we can **declare variables** of **date structure inside** it as:

```
struct teacher{  
    char name [20] ;  
    int age;  
    float salary;  
    struct date birthdate, joindate;  
}t1,t2;
```

- The **nested ,structures** may also be **initialized** at the time of **declaration**.
- For **example**
- struct teacher t1= { "Sam", 34, 9000, {8, 12, 1970}, { 1, 7, 1995 } };

Pointers to Structures

- We can have **pointer to structure**, which can **point to the start address** of a **structure variable**.
- **These pointers are called structure pointers** and can be **declared as**:

```
struct student {  
    char name [ 20 ] ;  
    int rollno;  
    int marks;  
};
```

```
struct student stu, *ptr;
```

- **Here ptr is a pointer variable that can point to a variable of type struct student.**
- We will use the **& operator** to access the **starting address** of a **structure variable**, so **ptr can point to stu** by writing:

ptr= &stu;

Pointers to Structures

- We can use the **arrow operator** `->` which is formed by **hyphen symbol** and **greater than symbol** for accessing the **members** as:

`ptr->name`

`ptr->rollno`

`ptr->marks`

Example

```
struct student {  
    char name [20] ;  
    int rollno;  
    int marks;  
};  
  
struct student stu={"Mary",25,68};  
  
struct student *ptr=&stu;  
  
printf ("Name%s\t",ptr->name);  
  
printf ("Rollno%d\t",ptr->rollno);  
  
printf ("Marks%d\n",ptr->marks) ;  
}
```

Structure and Function

```
struct book
{
    char name[25] ;
    char author[25] ;
    int callno ;
};

main( )
{
    struct book b1 = { "Let us C", "YPK", 101 } ;
    display ( &b1 ) ;
}

display ( struct book *b )
{
    printf ( "\n%s %s %d", b->name, b->author, b->callno ) ;
}
```

OUTPUT:

Let us C YPK 101

Lecture 15

Union in C

Dr. Vandana Kushwaha

Department of Computer Science
Institute of Science, BHU, Varanasi

Introduction

- Union is a **derived data type** like **Structure** and it can also contain **members** of different **data types**.
- The **syntax** used for **definition** of a **union**, **declaration** of **union variables** and for **accessing members** is **similar** to that used in **structures**, but here **keyword union** is used instead of **struct**.
- The main **difference** between **union** and **structure** is in the way **memory** is **allocated** for the members.
- In a **structure** each **member** has **its own memory location**, whereas **members** of **union share the same memory location**.
- When a **variable** of type **union** is **declared**, **compiler** allocates **sufficient memory** to **hold the largest member** in the **union**.
- Since all members **share the same memory location** hence we can **use only one member at a time**.

Union

- Thus **union** is used for **saving memory**.
- The concept of **union** is **useful** when it is **not necessary** to use **all members** of the **union** at a time.
- **Syntax of definition of a union is:**

```
union union_name{  
    data type member1;  
    data type member2;  
    .....  
}
```

- **Syntax of Union variable declaration :**

```
union union_name {  
    datatype member1;  
    data type member2;  
}variable_name;           OR
```

```
union union_name variable_name;
```

Union

- We can **access** the **union members** using the **same syntax** used for **structures**.
- If we have a **union variable** then the **members** can be **accessed** using **dot(.) operator**, and if we have a **pointer to union** then the **members** can be **accessed** using the **arrow -> operator**.

Example

```
void main() {  
union result{  
int marks;  
char grade;  
float per;  
}res;  
res.marks=90;  
printf("Marks %d\n",res.marks);  
res. grade= 'A' ;  
printf("Grade %c\n",res.grade);  
res.per=85.5;  
printf("Percentage %f\n",res.per);  
}
```

Output:

Marks: 90

Grade: A

Percentage: 85.500000

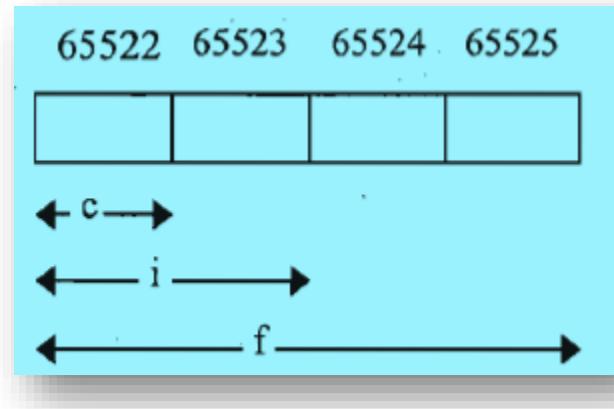
Union

- Only **one member** of **union** can **hold value** at a time, don't try to use all the members simultaneously.
- So a **union variable** of type result can be treated as either an **int** variable or **char** variable or a **float** variable.
- It is the **responsibility** of the **programmer** to **keep track** of **member** that **currently holds the value**.
- **Union variables** can also be **initialized**, but there is a **limitation**.
- We know that **due to sharing of memory** all the **members can't hold values simultaneously**.
- So during **initialization** also **only one member** can **given an initial value**, and this **privilege** is given to the **first member**.
- Hence **only the first member** of a **union** can be given an **initial value**.
- The **type** of the **initializer** should **match** with the **type** of the **first member**.

Memory allocation for Union

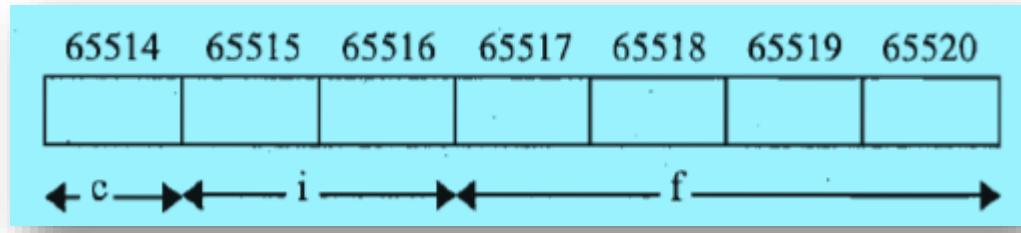
```
union utag
```

```
{  
char c;  
int i;  
float f;  
}uvar;
```



```
struct stag
```

```
{  
char c;  
int i;  
float f;  
}svar;
```



Example

```
void main()
{
union a{
    int i;
    char ch[2];
}key;
key.i=512;

printf("key.i= %d\n",key.i);

printf("key.ch[0] %d\n",key.ch[0]);

printf("key.ch[1] %d\n",key.ch[1]);

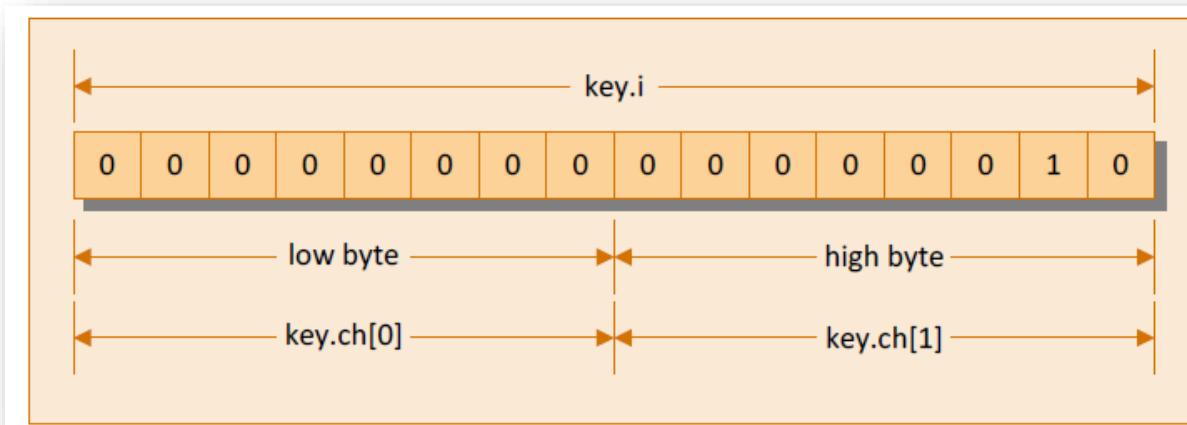
}
```

Output:

```
key.i= 512
key.ch[0] 0
key.ch[1] 2
```

Example

- 2 byte(16 bit) binary equivalent of **512** is = **0000 0010 0000 0000**
- Intel CPUs follow ***little-endian architecture*** while storing a two byte number in computer memory.
- In ***little-endian architecture*** the low byte is stored before the high byte.



- Thus `key.ch[0]` is printed as **0** and `key.ch[1]` is printed as **2**.

Example

- We can't assign **different values** to the **different union elements** at the **same time**.
- That is, if we **assign a value** to **key.i**, it gets **automatically assigned** to **key.ch[0]** and **key.ch[1]**.
- **Vice versa**, if we assign a value to **key.ch[0]** or **key.ch[1]**, it is bound to get assigned to **key.i**.

Example

```
# include <stdio.h>
int main( )
{
union a
{
short int i ;
char ch[ 2 ] ;
} ;
union a key ;
key.i = 512 ;
printf ( "key.i = %d\n", key.i ) ;
printf ( "key.ch[ 0 ] = %d\n", key.ch[ 0 ] ) ;
printf ( "key.ch[ 1 ] = %d\n", key.ch[ 1 ] ) ;
key.ch[ 0 ] = 50 ; /* assign a new value to key.ch[ 0 ] */
printf ( "key.i = %d\n", key.i ) ;
printf ( "key.ch[ 0 ] = %d\n", key.ch[ 0 ] ) ;
printf ( "key.ch[ 1 ] = %d\n", key.ch[ 1 ] ) ;
return 0 ;
}
```

Output:

```
key.i = 512
key.ch[ 0 ] = 0
key.ch[ 1 ] = 2
key.i= 562
key.ch[ 0 ] = 50
key.ch[ 1 ] = 2
```

Utility of Unions

- Suppose we wish to store **information** about **employees** in an **organization**.
- The **items of information** are as shown below:

Name
Grade
Age
If Grade = HSK (Highly Skilled)
 hobbie name
 credit card no.
If Grade = SSK (Semi Skilled)
 Vehicle no.
 Distance from Co.

Utility of Unions

- Since this is **dissimilar information** we can **gather** it **together** using a **structure** as shown below:

```
struct employee
{
    char n[ 20 ];
    char grade[ 4 ];
    int age;
    char hobby[ 10 ];
    int crcardno;
    char vehno[ 10 ];
    int dist;
};
struct employee e;
```

Utility of Unions

- Though **grammatically** there is **nothing wrong** with this **structure**, it **suffers from a disadvantage**.
- For any **employee**, depending upon his/her **grade**, either the fields **hobby** and **credit card no.** or the fields **vehicle number** and **distance** would get used.
- **Both sets** of fields would **never get used**.
- This would **lead** to **wastage of memory** with every **structure variable** that we **create**, since **every structure variable** would have **all the four fields** apart from **name, grade and age**.
- This can be **avoided** by **creating** a **union** between these sets of fields.

Utility of Unions

```
struct info1
{
    char hobby[ 10 ];
    int crcardno ;
};
```

```
struct info2
{
    char verno[ 10 ];
    int dist ;
};
```

```
union info
{
    struct info1 a;
    struct info2 b;
};
```

```
struct employee
{
    char n[ 20 ];
    char grade[ 4 ];
    int age ;
    union info f;
};
struct employee e;
```

Lecture 16

Dynamic Memory Allocation

Dr. Vandana Kushwaha

Department of Computer Science
Institute of Science, BHU, Varanasi

Introduction

- The **memory allocation** that we have done till now was **static memory allocation**.
- The **memory** that could be used by the **program** was **fixed** i.e. we **could not increase or decrease** the **size** of **memory** during the **execution of program**.
- In **many applications** it is **not possible** to **predict** how much **memory** would be **needed** by the **program** at **run time**.
- For **example** if we declare an **array of integers** `int emp_no[200];`
- Now **two types** of **problems** may occur.
- The **first case** is that the **number of values** to be **stored** is **less than** the **size of array** and hence there is **wastage of memory**.
- For **example** if we have to store only **50 values** in the above array, then space for **150 values**(300 bytes) is **wasted**.

Introduction

- In **second case** our **program fails** if we want to **store more values** than the size of array, for **example** If there is need to store **205 values** in the above array.
- To **overcome these problems** we should be able to **allocate memory at run time**.
- The **process of allocating memory at the time of execution** is called **dynamic memory allocation**.
- The **allocation** and **release** of this **memory space** can be **done** with the help of some **built-in-functions** whose **prototypes** are found in **alloc.h** and **stdlib.h** header files.
- These **functions** take **memory** from a **memory area** called **heap** and **release** this **memory** whenever not required, so that it can be used again for some other purpose.
- **Pointers** play an **important role** in **dynamic memory allocation** because we can access the dynamically allocated memory only through pointers.

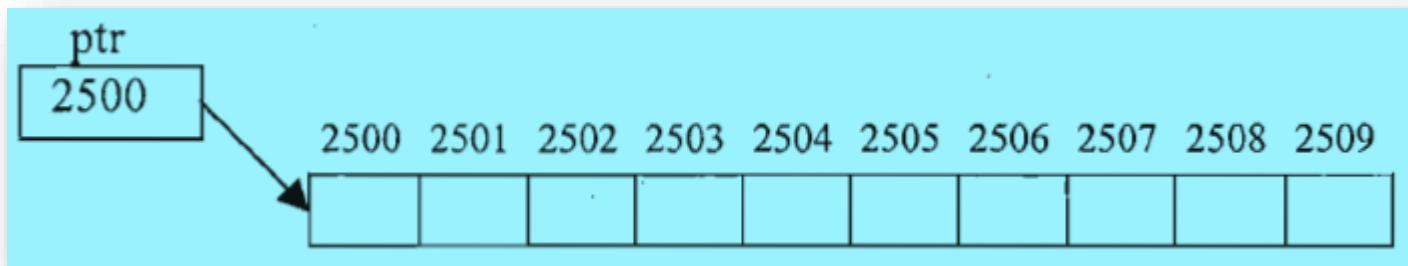
malloc() function

- The **malloc()** function reserves a **block of memory** of the **specified number of bytes**.
- And, it **returns a pointer** of type **void** which can be **casted** into **pointers of any form**.
- Syntax:**

```
ptr = (castType*) malloc(size);
```

- For **example**:

```
int *ptr;  
ptr = ( int * ) malloc (10);
```
- And, the pointer **ptr** holds the **address** of the **first byte** in the **allocated memory**.



malloc() function

- We can use **sizeof operator** to make the program portable and more readable.

```
ptr = ( int * ) malloc ( 5 * sizeof ( int ) );
```

- This **allocates** the **memory space** to hold **five integer values**.
- If there is **not sufficient memory** available in **heap** then **malloc()** returns **NULL**.
- So we **should always check** the **value returned** by **malloc()**.
- **Example:**

```
ptr = (float *) malloc(10*sizeof(float) );
if ( ptr == NULL )
printf("Sufficient memory not available");
```

The calloc() function

- The **calloc() function** is used to allocate **multiple blocks of memory**.
- **Syntax:** `ptr = (castType*)calloc(n, size);`
- It is somewhat similar to **malloc() function** except for **two differences**.
- The **first one** is that it takes **two arguments**.
- The **first argument** specifies the **number of blocks** and the **second one** specifies the **size of each block**.
- **Example**

```
ptr = ( int * ) calloc ( 5 , sizeof(int) );
```

- This **allocates 5 blocks of memory, each block contains 2 bytes** and the **starting address** is stored in the **pointer** variable **ptr**, which is of type **int**.
- An **equivalent malloc() call** would be

```
ptr= ( int * ) malloc ( 5 * sizeof(int) );
```

The calloc() function

- The other **difference** between **calloc()** and **malloc()** is that the **memory allocated** by **malloc()** **contains garbage value** while the **memory allocated** by **calloc()** is **initialized to zero**.
- Like **malloc()**, **calloc()** also **returns NULL** if there is **not sufficient memory** available in the **heap**.

realloc() Function

- If the **dynamically allocated memory** is **insufficient** or **more** than required, we can **change** the **size** of previously allocated memory using the **realloc()** function.
- **Syntax:**

```
ptr= (castType*) realloc( ptr, newsize);
```

- The function **realloc()** is used to **alters** the **size** of the **memory block** without **losing** the **old data**.
- This function takes **two arguments**, **first** is a **pointer** to the **block of memory** that was **previously allocated** by **malloc()** or **calloc()** and **second** one is the **new size** for that block.
- For **example**

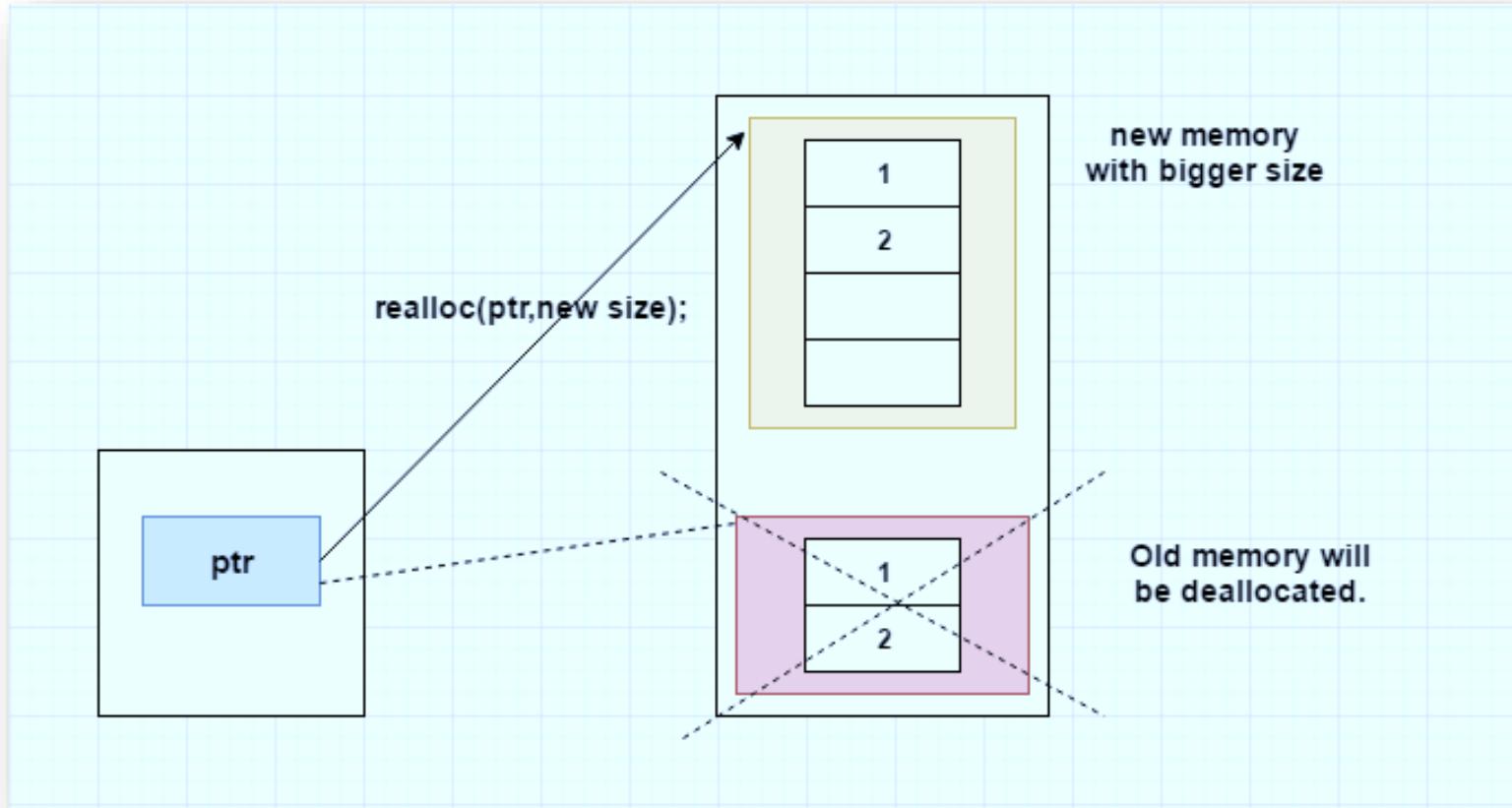
```
ptr= (int *) malloc ( size );  
ptr = (int *) realloc ( ptr , newsize );
```

realloc() Function

- If **realloc()** request **success**, it will **return** a **pointer** to the **newly allocated memory**.
- Otherwise, it will **return** **NULL**.
- If the **given size** is **smaller** than the **actual**, it will **simply reduce** the **memory size**.
- If the **given size** is **bigger** than the **actual**, it will check whether it can **expand** the **already available memory**.
- **If it is possible** it will **simply resize** the **memory**.
- Otherwise, **realloc()** will **create** the **new block** of **memory** with the **larger size** and **copy the old data** to that **newly allocated memory** and than it will **deallocate** the **old memory area**.
- For **example** assume, we have **allocated memory** for **2 integers**.
- We are going to **increase** the **memory size** to store **4 integers**.

realloc() Function

- If realloc() is **unable** to expand the memory size, it will do something like :



Example: Array using DMA

```
void main()
{
    int *ptr,size,i;
    size = 2;
    ptr = malloc(size * sizeof(int));
    for(i = 0; i < size; i++)
        scanf("%d",ptr + i);
    printf("Array Elements\n");
    for(i = 0; i < size; i++)
        printf("%d\n",ptr[i]);
    size = 5;
    printf("Enter three more elements: \n");
    ptr = realloc(ptr, size * sizeof(int));
    for(i = 2; i < size; i++)
        scanf("%d",ptr + i);
    printf("New Array Elements\n");
    for(i = 0; i < size; i++)
        printf("%d\n",ptr[i]);
}
```

OUTPUT:

```
1
2
Array Elements
1
2
Enter three more elements:
3
4
5
New Array Elements
1
2
3
4
5
```

free() function

- The **dynamically allocated memory** is **not automatically released**; it will **exist till the end of program**.
- If we have **finished** working with the **memory allocated dynamically**, it is **our responsibility to release that memory** so that it can be **reused**.
- The function **free()** is used to **release the memory** space allocated **dynamically**.
- The memory released by **free()** is made **available to the heap again** and can be used for some other purpose.
- **Syntax:** **free(ptr);**
- Here **ptr** is a **pointer variable** that **contains the base address** of a **memory block** created by **malloc()** or **calloc()**.
- Once a **memory location** is **freed** it should not be used.

Lecture 17

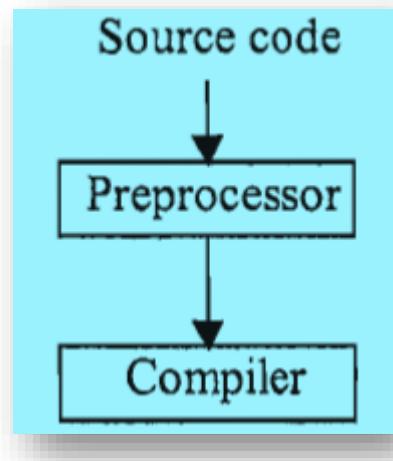
Preprocessors in C

Dr. Vandana Kushwaha

Department of Computer Science
Institute of Science, BHU, Varanasi

Introduction

- The **source code** that we write is translated into **object code** by the **compiler**.
- But **before being compiled**, the **code** is passed through the **C preprocessor**.
- The **preprocessor** scans the whole **source code** and **modifies** it, which is then given to the **compiler**.



- The **lines starting** with the **symbol #** are known as **preprocessor directives**.
- When the **preprocessor** finds a **line** starting with the **symbol #**, it considers it as a **command** for itself and works accordingly.

Introduction

- All the **directives** are **executed** by the **preprocessor**, and the **compiler** does not receive any line starting with **# symbol**.
- **Features of Preprocessor directives :**
- Each **preprocessor directive starts** with a **# symbol**.
- There can be only **one directive on a line**.
- There is **no semicolon** at the **end of a directive**.
- To continue a **directive** on **next line**, we should place a **backslash(\)** at the **end of the line**.
- The **preprocessor directives** are usually **written** at the **beginning** of a **program**.
- A **directive** is **active** from the **point** of its **appearance** till the **end** of the **program**.

Types of Preprocessor directives

- **Types of Preprocessor directives** are:
 1. Simple Macro Substitution
 2. Macros with arguments
 3. Conditional Compilation
 4. File Inclusion

Simple Macro Substitution

- These **directives** are used to **define symbolic constants**.
- The **general syntax** is- `#define macro_name macro_expansion`
- Here **macro_name** is any **valid C identifier**, and it is generally taken in **capital letters** to distinguish from other variables.
- The **macro_expansion** is the **value** to be **replaced** with **macro name**.
- A **space** is **necessary** between the **macro name** and **macro expansion**.
- The **C preprocessor** **replaces** all the **occurrences** of **macro name** with the **macro_expansion**.

```
#deflne TRUE 1
```

```
#define FALSE
```

```
#define PI 3.14159265
```

```
#define MAX 100
```

```
#define RETIREAGE 58
```

Examples

- We can have **macro expansions** of any sort.
- **Examples:**

```
#define AND &&
```

```
#define OR ||
```

```
#define BEGIN main( ){
```

```
#define END }
```

```
#define INFINITE while( 1 )
```

```
#define NEW_LINE printf("\n");
```

```
#define ERROR printf("An error has occurred\n");
```

Macros with Arguments

- The **#define directive** can also be used to define **macros with arguments**.
- **General syntax:**

```
#define macro_name(arg 1, arg2, ..... ) macro_expansion
```

- Here **arg1, arg2** are the **formal arguments**.
- The **macro_name** is **replaced** with the **macro_expansion** and the **formal arguments** are **replaced** by the corresponding **actual arguments** supplied in the **macro call**.
- For **example** suppose we define these **two macros-**
- `#define SUM(x, y) ((x) + (y))`
- `#define PROD(x, y) ((x) * (y))`

Macros with Arguments

- Now suppose we have these **two statements** in our programs

s= SUM(5, 6);

p = PROD(m, n);

- After passing through the **preprocessor** these statements would be **expanded** as

s= ((5) + (6));

p = ((m) * (n));

- Since this is just **replacement** of text, hence the **arguments** can be of **any data type**.

- For **example** we may use the **macro SUM** to find out the **sum** of **long** or **float types** also.

- The statements **s= SUM(2.3, 4.8);** would be expanded as **s= ((2.3) + (4.8));**

Examples

- `#define SQUARE(x) ((x)*(x))`
- `#define MAX(x, y) ((x) > (y) ? (x) : (y))`
- `#define CIRCLE_AREA(rad) (3.14 * (rad) *(rad))`
- Note that there should be **no space** between the **macro_name** and **left parenthesis**, otherwise the **macro expansion** is considered to start from the **left parenthesis**.

- For example if we write a **macro** like this-

```
#define SQUARE (x) ( (x)*(x) )
```

- Now any call like **SQUARE(5)** would be **expanded** as
$$(x) ((x)*(x))(5)$$
- Here **SQUARE** is considered as a **macro without arguments** and the text **(x) ((x)*(x))** is regarded **macro expansion**.
- SO **SQUARE** is **replaced** by the **macro expansion** and **(5)** is written as it is.

Example

- The entire **macro expansion** should be **enclosed within parentheses**.

```
#define SQUARE(n) n * n
```

```
main( )
```

```
{
```

```
int j;
```

```
j = 64 / SQUARE ( 4 );
```

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

```
}
```

- The **output** of the above program would be: **j = 64**
- Whereas, what we **expected** was **j = 4.**

Macros Vs Functions

- Macros with arguments can perform tasks similar to functions.
- A macro is expanded into inline code so the text of macro is inserted into the code for each macro call.
- Hence macros make the code lengthy and the compilation time increases.
- On the other hand the code of a function is written only at one place, regardless of the number of times it is called so the functions makes the code smaller.
- In functions the passing of arguments and returning a value takes some time and hence the execution of the program becomes slow while in the case of macros this time is saved and they make the program faster.
- So functions are slow but take less memory while macros are fast but occupy more memory due to duplicity of code.

Macros Vs Functions

- If the **macro is small** it is **good** but if it is **large** and is **called many times** then is **better to change it into a function** since it may increase the **size of the program** considerably
- So whether to use a **macro** or a **function** depends on the **memory available**, your requirement and nature of the task to be performed.
- **Macros** can be used with **arguments of different types** .
- **Functions** perform **type checking** so **separate functions** have to be written for each **data type**.
- But **lack of type checking** facility in **macros makes** them more **error prone**.

Including Files

- The **preprocessor directive #include** is used to **include a file** into the **source code**.
- We have already used directive to include **header files** in our programs.
- The **filename** should be **within angle brackets** or **double quotes**.
- **Syntax -**
- **#include "filename"** OR **#include<filename>**
- The **preprocessor replaces** the **#include directive** by the **contents** of the **specified file**.
- After **including** the **file**, the **entire contents** of **file** can be used in the **program**.
- If the **filename** is in **double quotes**, first it is searched in the **current directory** (where the source file is present), if **not found** there then it is searched in the **standard include directory**.

Including Files

- If the **filename** is **within angle brackets**, then the **file is searched in the standard include directory only.**
- Generally **angled brackets** are used to **include standard header files** while **double quotes** are used to **include header files** related to a particular program.
- We can also **specify** the **whole path** instead of the **path name**.
- **For example-**

```
#include "C:\mydir\myfile.h"
```

Conditional Compilation

- There may be **situations** when we want to **compile some parts** of the **code** based on **some condition**.
- Before **compilation** the **source code** passes through the **preprocessor**.
- So we can direct the **preprocessor** to **supply only some parts** of the **code** to the **compiler** for **compilation**.
- **Conditional compilation** means **compilation of a part of code** based on some **condition**.
- These **conditions** are **checked during** the **preprocessing** phase.
- The **directives** used in **conditional compilation** are-
- **#ifdef #ifndef #if #else #endif**

#if and #endif

```
#define FLAG 8
```

```
main( )
```

```
{
```

```
int a=20,b=4;
```

```
#if FLAG >= 3
```

```
a=a+b;
```

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

```
#endif
```

```
print f ("a = %d,b %d \ n" , a , b) ;
```

```
}
```

OUTPUT:

a = 24, b = 96

#if and #endif

- In this program **FLAG** is defined with the **value 8**.
- **First** the **constant expression FLAG >= 3** is **evaluated**, since it is **true**(non zero).
- Hence all the **statements** between **#if** and **#endif** are **compiled**.
- Suppose the value of **FLAG** is changed to **2**, now, the **constant expression FLAG >= 3** would **evaluate to false**,
- Hence the **statements** between **#if** and **#endif** will **not be compiled**.
- In this case the **output** of the code would be **a = 20, b = 4**.

#else directive

- #else is used with the #if preprocessor directive.
- It is analogous to if, else control structure.

#if constant-expression

statements

#else

statements

#endif

- The constant expression evaluates to non-zero then the statements between #if and #else are compiled otherwise the statements between #else and #endif are compiled.

#else directive

```
#define FLAG 2
main ( )
{
    int a=20,b=4;
    #if FLAG>3
        a=a+b;
        b=a*b;
    #else
        a=a-b;
        b=a/b;
    #endif
    printf ("a %d, b = %d\n",a,b);
}
```

OUTPUT:

a = 16, b = 4

#ifdef and #ifndef

- We can, if we want, have the **compiler** skip over part of a **source code** by inserting the **preprocessing commands** **#ifdef** and **#endif**, which have the general form:

#ifdef macroname

statement 1 ;

statement 2 ;

statement 3 ;

#endif

- If **macroname** has been **#defined**, the **block of code** will be **processed as usual**; **otherwise not.**

Uses of #ifdef and #ifndef

- To “comment out” obsolete lines of code.
- It often happens that a program is changed at the last minute to satisfy a client.
- This involves rewriting some part of source code to the client’s satisfaction and deleting the old code.
- But veteran programmers are familiar with the clients who change their mind and want the old code back again just the way it was.
- One solution in such a situation is to put the old code within a pair of /* */
- But we might have already written a comment in the code that we are about to “comment out”.
- This would mean we end up with nested comments.
- Obviously, this solution won’t work since we can’t nest comments in C.

Uses of #ifdef and #ifndef

- Therefore the **solution** is to use **conditional compilation** as shown below.

```
main( )  
{  
    #ifdef OKAY  
        statement 1 ;  
        statement 2 ; /* detects virus */  
        statement 3 ;  
        statement 4 ; /* specific to stone virus */  
    #endif  
        statement 5 ;  
        statement 6 ;  
        statement 7 ;  
}
```

- Here, **statements 1, 2, 3 and 4** would **get compiled** only if the **macro OKAY** has **been defined**, and we have **purposefully omitted** the **definition** of the **macro OKAY**.

Uses of #ifdef and #ifndef

- At a **later date**, if we **want** that **these statements** should also **get compiled** all that we are required to do is to **delete** the **#ifdef** and **#endif** statements.
- A more sophisticated use of **#ifdef** has to do with making the **programs portable**, i.e. to make them work on two totally different computers.
- Suppose an **organization** has **two different types of computers** and we are **expected to write a program** that **works on both** the **machines**.
- We can do so by **isolating the lines of code** that **must be different for each machine** by marking them off with **#ifdef**.

Uses of #ifdef and #ifndef

```
main( )  
{  
    #ifdef INTEL  
        code suitable for a Intel PC  
    #else  
        code suitable for a Motorola PC  
    #endif  
    code common to both the computers  
}
```

- When we **compile** this **program** it would **compile only the code** suitable for a **Motorola PC** and the **common code**.
- This is because the **macro INTEL** has **not been defined**.

Uses of #ifdef and #ifndef

- Note that the working of **#ifdef - #else - #endif** is similar to the ordinary **if - else control instruction of C**.
- If we want to run your program on a **INTEL PC**, just **add a statement** at the top **saying, #define INTEL.**

```
#define INTEL
main( )
{
    #ifdef INTEL
        code suitable for a Intel PC
    #else
        code suitable for a Motorola PC
    #endif
        code common to both the computers
}
```

Lecture 18

File Input/Output in C

Dr. Vandana Kushwaha

Department of Computer Science
Institute of Science, BHU, Varanasi

Introduction

- The **input** and **output** operations that we have **performed** so far were done through **screen** and **keyboard** only.
- After **termination** of **program**, all the **entered data** is **lost** because **primary memory** is **volatile**.
- If the **data** has to be **used later**, then it becomes **necessary** to **keep it** in **permanent storage device**.
- C supports the **concept** of **files** through which **data** can be **stored** on the **disk** or **secondary storage device**.
- The **stored data** can be **read** whenever **required**.
- A **file** is a **collection of related data placed on the disk**.

File Operations

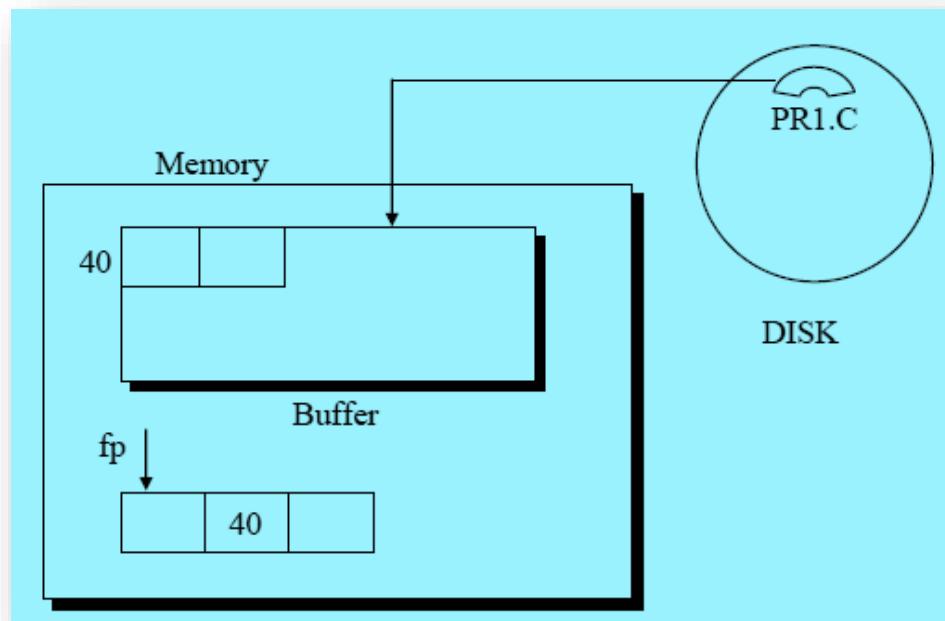
- There are **different operations** that can be carried out on a **file**. These are:
 1. *Creation of a new file*
 2. *Opening an existing file*
 3. *Reading from a file*
 4. *Writing to a file*
 5. *Moving to a specific location in a file (seeking)*
 6. *Closing a file*

Opening a File

- Before we can **read** (or **write**) information **from** (**to**) a **file** on a **disk** we must **open** the **file**.
- To **open** the **file** we need to **call** a function **fopen("PR1.C", "r")**.
- It would **open** a **file** "**PR1.C**" in **read mode**, which tells the **C compiler** that we would be **reading** the **contents** of the **file**.
- Note that "**r**" is a **string** and not a **character**; hence the **double quotes** and not **single quotes**.
- In fact **fopen()** performs **three important tasks** when you **open** the **file** in "**r**" **mode**:
 - (a) Firstly it **searches** on the **disk** the **file** to be opened.
 - (b) Then it **loads** the **file** from the **disk** into a **place** in **main memory** called **buffer**.
 - (c) It **sets up** a **character pointer** that **points** to the **first character** of the **buffer**.

Need of buffer

- **Reading** every character from a **file** on a **disk** takes a **long time** to **complete** the **reading operation**.
- It would be more **sensible** to **read the contents** of the **file** into the **buffer(memory area in RAM)** while **opening** the **file** and then **read the file** **character by character** from the **buffer** rather than from the **disk**.



Opening a File

- First we define a **structure pointer** of type **FILE**:

FILE *fp ;

- The **fopen()** function returns the **address** of this **structure**, which we collect in the **structure pointer** called **fp**, declared as **SYNTAX**:

fp=fopen(const char *filename, const char *mode);

- **fopen()** function takes **two strings** as **arguments**, the **first one** is the **name of the file** to be **open** and the **second one** is the **mode** that decides which **operations** (**read, write, append** etc) are to be performed on the **file**.
- On success, **fopen()** returns a **pointer** of **type FILE** and on **error** it returns **NULL**.
- The **return value** of **fopen()** is assigned to a **FILE pointer fp** declared previously.

Opening a File

- For **example**-

```
FILE *fp1, *fp2; . ,  
fp1 = fopen ( "myfile. txt", "w" );  
fp2 = fopen ( "yourfile.dat", "r" );
```

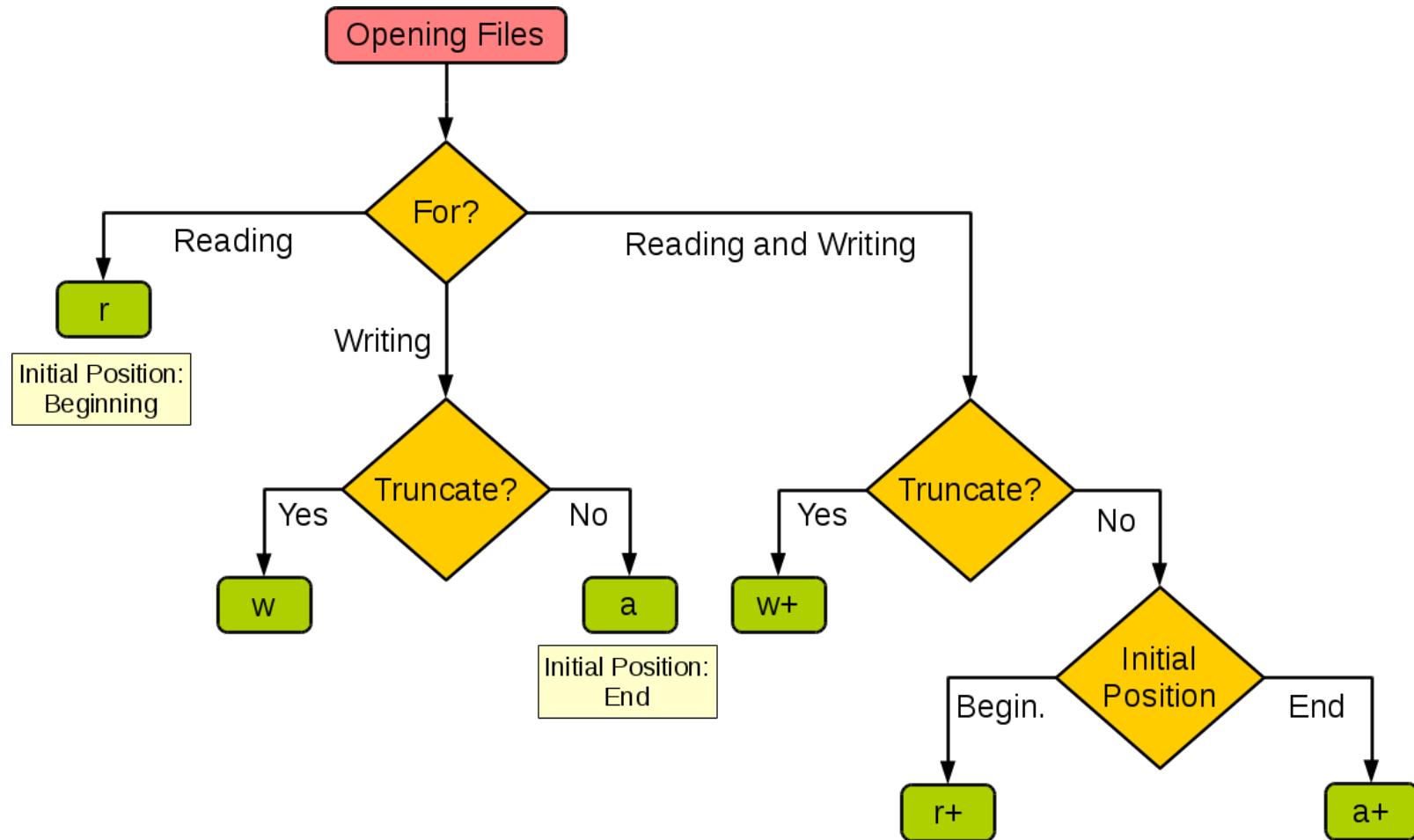
File opening modes

- **w" (write)**
- If the **file doesn't exist** then this mode **creates a new file** for **writing**, and if the **file already exists** then the **previous data is erased** and the **new data entered** is written to the **file**.
- **"a"(append)**
- If the **file doesn't exist** then this mode **creates a new file**, and if the **file already exists** then the **new data entered** is **appended** at the **end of existing data**.
- **"r" (read)**
- This mode is used for **opening an existing file** for **reading purpose only**.
- The **file to be opened must exist** and the **previous data of file is not erased**.

File opening modes

- **"w+" (write+read)**
- This mode is **same** as "**w**" **mode** but in this **mode** we can also **read and modify** the **data**.
- If the **file doesn't exist** then a **new file** is **created** and if the **file exists** then **previous data is erased**.
- **"r+" (read+write)**
- This mode is **same** as "**r**" **mode** but in this **mode** we can also **write and modify** **existing data**.
- The **file to be opened must exist** and the **previous data** of file is **not erased**.
- Since we can **add new data** to **modify existing data** so this **mode** is also called **update mode**.

File opening modes



Errors in Opening Files

- If an **error** occurs in **opening** the **file**, then **fopen()** returns **NULL**.
- We can **check for any errors** in **opening** by checking the **return value** of **fopen()**.

FILE *fp;

fp = fopen("text.dat","w");

If(fp==NULL)

{

printf("Error in opening file");

exit () ;

}

Errors in Opening Files

- **Error** in opening a file may **occur** due to **various reasons**, for **example-**
- If we try to **open a file** in **read** or **update mode**, and the **file doesn't exist** or we do **not have read permission** on that **file**.
- If we try to **create a file** but there is **no space** on the **disk** or we **don't have write permission**.
- If we try to **create a file** that **already exists** and we **don't have permission** to **delete** that **file**.
- **Operating system** **limits** the **number of files** that **can be opened** at a time and we are **trying to open more files** than that **number**.

Closing a File

- The **file** that was **opened** using **fopen() function** must be **closed** when no more operations are to be performed on it.
- After **closing** the **file**, **connection** between **file** and **program is broken**.
- **SYNTAX:** *int fclose(FILE *fptr);*
- On **closing** the **file**, all the **buffers** associated with it are **flushed** i.e. **all the data** that is in the **buffer** is **written** to the **file**.
- The **buffers** allocated by the system for the file are **freed** after the **file** is **closed** so that these **buffers** can be **available** for **other files**.
- **fclose()** returns **EOF** on **error** and **0** on **success**(**EOF** is a **macro constant** defined in **stdio.h** and its value is **-1**).

Functions used for file I/O

- The **functions** used for **file I/O** are:
 1. **Character I/O** - `fgetc()`, `fputc()`
 2. **String I/O** - `fgets()`, `fputs()`
 3. **Integer I/O** - `getw()`, `putw()`
 4. **Formatted I/O** - `fscanf()`, `fprintf()`
 5. **Record I/O** - `fread()`, `fwrite()`

Reading/Writing Character from/to a File

- To **read** the file's contents **character by character** from **memory** there exists a function called **fgetc()**.
- **Syntax-**

*ch = fgetc (FILE *fp);*

- **fgetc()** reads the **character** from the **current pointer position**, **advances** the **pointer position** so that it now **points** to the **next character**, and **returns** the **character** that is **read**, which we **collected** in the variable **ch**.
- To **write** **character** contents to a **file** there exists a function called **fputc()**.
- **Syntax-**

*fputc(char ch, FILE *fp);*

- Where **ch** is a **name** of **char variable** that we **write** in a **file** and **fp** is the **file pointer**.

Display contents of a file on screen

```
int main( )
{
    FILE *fp ;
    char ch ;
    fp = fopen ( "PR1.C", "r" ) ;
    while ( 1 )
    {
        ch = fgetc ( fp ) ;
        if ( ch == EOF )
            break ;
        printf ( "%c", ch ) ;
    }
    fclose ( fp ) ;
}
```

Count chars, spaces, tabs and newlines in a file

```
main( )
{
    FILE *fp ;
    char ch ;
    int nol = 0, not = 0, nob = 0, noc = 0 ;
    fp = fopen ( "PR1.C", "r" ) ;
    while ( 1 )
    {
        ch = fgetc ( fp ) ;
        if ( ch == EOF )
            break ;
        noc++ ;
        if ( ch == ' ' )
            nob++ ;
```

Count chars, spaces, tabs and newlines in a file

```
if ( ch == '\n' )
    nol++ ;
if ( ch == '\t' )
    not++ ;
}
fclose ( fp ) ;
printf ( "\nNumber of characters = %d", noc ) ;
printf ( "\nNumber of blanks = %d", nob ) ;
printf ( "\nNumber of tabs = %d", not ) ;
printf ( "\nNumber of lines = %d", nol ) ;
}
```

A File-copy Program

- The function **fgetc()** which **reads characters** from a **file**, Its counterpart is a function called **fputc()** which **writes characters** to a file.

```
main( )
{
FILE *fs, *ft ;
char ch ;
fs = fopen ( "pr1.c", "r" ) ;
if ( fs == NULL )
{
puts ( "Cannot open source file" ) ;
exit( ) ;
}
ft = fopen ( "pr2.c", "w" ) ;
```

A File-copy Program

```
if ( ft == NULL )
{
puts ( "Cannot open target file" ) ;
fclose ( fs ) ;
exit( ) ;
}
while ( 1 )
{
ch = fgetc ( fs ) ;
if ( ch == EOF )
break ;
else
fputc ( ch, ft ) ;
}
fclose ( fs ) ;
fclose ( ft ) ;
}
```

String (line) I/O in Files

- To **read** the **file's contents** **line by line** from memory there exists a **function** called **fgets()**.
- **Syntax-** `char * fgets (char *str, int n, FILE *fp);`
- **str** : Pointer to an array of chars where the string read is copied.
- **n** : Maximum number of characters to be copied into **str**
- **fgets()** **reads a line** and stores it into the **string** pointed to by **str**.
- It **stops** when either **(n-1) characters** are read, the **newline character** is read, or the **end-of-file** is reached, whichever comes first.
- To **write character** contents to a **file** there exists a function called **fputs()**.
- **Syntax-** `fputs(char *ch, FILE *fp);`
- Where **ch** is a name of **char pointer** variable to a string that we write in a file and **fp** is the **file pointer**.

String (line) I/O in Files

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int main( )
{
FILE *fp ;
char s[8] ;
int len;
fp = fopen ( "POEM.TXT", "w" ) ;
if ( fp == NULL )
{
puts ( "Cannot open file" ) ;
exit(0) ;
}
```

String (line) I/O in Files

```
printf ( "\nEnter a few lines of text:\n" );
```

```
while (1)
{
    gets(s);
    len=strlen(s);
    if(len==0)
        break;
    fputs ( s, fp ) ;
    fputs ( "\n", fp ) ;
}
fclose ( fp ) ;
}
```

Reads strings from the file and displays them on screen

```
int main( )
{
    FILE *fp ;
    char s[80] ;
    fp = fopen ( "POEM.TXT", "r" ) ;
    if ( fp == NULL )
    {
        puts ( "Cannot open file" ) ;
        exit( ) ;
    }
    while ( fgets ( s, 79, fp ) != NULL )
        printf ( "%s" , s ) ;
    fclose ( fp ) ;
}
```

Integer I/O

```
main ( )
{
FILE *fptr;
int i,number;
fptr=fopen("num.txt", "w");
for(i=1;i<=5;i++)
{
scanf("%d", &number);
putw(i,fptr);
}
fclose(fptr);
fptr=fopen("num.txt", "r");
while ( (i=getw(fptr) ) !=EOF)
printf("%d\t",i);
fclose(fptr) ;
}
```

Program will write integers from 1 to 5 into the file "**num.txt**".

Program: Even-Odd

```
int main()
{
    FILE *f1, *f2, *f3;
    int number, i;
    printf("Contents of DATA file\n\n");
    f1 = fopen("DATA.txt", "w"); /* Create DATA file */
    for(i = 1; i <= 30; i++)
    {
        scanf("%d", &number);
        if(number == -1) break;
        putw(number,f1);
    }
    fclose(f1);
    f1 = fopen("DATA.txt", "r");
    f2 = fopen("ODD.txt", "w");
    f3 = fopen("EVEN.txt", "w");
}
```

Program: Even-Odd

```
while((number = getw(f1)) != EOF)
{
    if(number %2 == 0)
        putw(number, f3); /* Write to EVEN file */
    else
        putw(number, f2); /* Write to ODD file */
}
fclose(f1);
fclose(f2);
fclose(f3);
f2 = fopen("ODD.txt","r");
f3 = fopen("EVEN.txt", "r");
printf("\n\nContents of ODD file\n\n");
while((number = getw(f2)) != EOF)
    printf("%4d", number);
printf("\n\nContents of EVEN file\n\n");
while((number = getw(f3)) != EOF)
    printf("%4d", number);
fclose(f2);
fclose(f3);
}
```

Formatted I/O

```
int main( )
{
FILE *fp;
char name[10];
char text[50];
int age;
fp=fopen (" rec.txt", "w") ;
printf ("Enter your name and age") ;
scanf("%s%d",name,&age) ;
fprintf(fp,"\\nMy name is %s and age is %d",name,age);
fclose (fp) ;
fp=fopen (" rec.txt", "r") ;
while(fscanf(fp, "%s", text)!=EOF){
    printf("%s ", text );
}
fclose(fp);
}
```

Lecture 19

Miscellaneous Topics in C

Dr. Vandana Kushwaha

Department of Computer Science
Institute of Science, BHU, Varanasi

Command Line Arguments

- In any C program, there may be multiple functions, but the **main()** function remains the **entry point** from where the **execution starts**.
- While the **other functions** may have **one or more arguments** and a **return type**, the **main() function** is generally written with **no arguments**.
- It is possible to **pass arguments** from the **command line** to the **main() function** when the **program is executed**.
- These **arguments** are called **command line arguments**.
- **Command line arguments** are important for your program, especially when you want to **control your program** from **outside**, instead of **hard coding** those values inside the code.

Command Line Arguments

- Let us suppose we want to write a C program "**hello.c**" that prints a "**hello**" **message** for a **user**.
- Instead of **reading** the **name** from **inside the program** with **scanf()**, we wish to **pass the name** from the **command line** as follows:

C:\users\user>hello Prakash

- The **string** will be used as an **argument** to the **main() function** and then the "**Hello Prakash**" message should be displayed.
- To facilitate the **main() function** to **accept arguments** from the **command line**, we should define **two arguments** in the **main() function** – **argc** and **argv[]**.
- argc** refers to the **number of arguments** passed and **argv[]** is a **pointer array** that **points to each argument** passed to the **program**.

Command Line Arguments

- **Syntax**

```
int main(int argc, char *argv[]) {  
    ...  
    ...  
  
    return 0;  
}
```

- The **argc** argument should always be **non-negative**.
- The **argv** argument is an **array of character pointers** to all the **arguments**, **argv[0]** being the **name** of the **program**.
- After that **till "argv [argc - 1]"**, every element is a **command-line argument**.

Command Line Arguments

```
int main (int argc, char * argv[])
{
if (argc != 3)
{
printf("You have forgot to type numbers.");
exit(1);
}
int c = atoi(argv[1]) + atoi(argv[2]);
printf("addition: %d", c);
return 0;
}
```

- Now **execute** this **program** through **command prompt** by entering the following **command**:

C:\Users\user>add 10 20 (press enter key)

addition: 30

- atoi()** function **converts** the **string** representation of a **number** to an **integer**.

Enumerated Data Type

- The **enumerated data type** give us an opportunity to **invent our own data type** and **define** what **values** the **variable** of this **data type** can take.
- This can help in making the **program** more **readable**, which can be an **advantage** when a program gets complicated or when **more than one programmer** would be **working** on it.
- Using **enumerated data type** can also help you **reduce programming errors**.
- As an **example**, one could invent a **data type** called **mar_status** which can have **four possible values** ***single, married, divorced, widowed***.

Enumerated Data Type

- The **format** of the **enum definition** is similar to that of **structure**:

```
enum mar_status
{
    single, married, divorced, widowed
};
enum mar_status person1, person2;
```

- Like **structures**, this **declaration** has **two parts**:
- (a) The **first part** declares the **data type** and **specifies** its **possible values**, called '**enumerators**'.
- (b) The **second part** declares **variables** of this **data type**.

Enumerated Data Type

```
enum mar_status
{
    single, married, divorced, widowed
};
enum mar_status person1, person2;
```

- **Internally**, the **compiler** treats the **enumerators** as **integers**, starting with **0**.
- Thus, in our example, **single** is stored as **0**, **married** is stored as **1**, **divorced** as **2** and **widowed** as **3**.
- This way of assigning **numbers** can be **overridden** by the **programmer** by **initializing** the **enumerators** to **different integer values** as shown below:

```
enum mar_status
{
    single = 100, married = 200, divorced = 300, widowed = 400
};
enum mar_status person1, person2;
```

Uses of Enumerated Data Type

- **Enumerated variables** are usually **used** to clarify the **operation** of a **program**.
- For **example**, if we need to use **employee departments** in a **payroll program**, it makes the **listing easier to read** if we use **values** like **Assembly, Manufacturing, Accounts** rather than the **integer values 0, 1, 2**, etc.

```
int main( )
{
    enum emp_dept
    {
        assembly, manufacturing, accounts, stores
    };
    struct employee
    {
        char name[ 30 ];
        int age ;
        float bs ;
        enum emp_dept department;
    };
    struct employee e ;
```

Uses of Enumerated Data Type

```
strcpy ( e.name, "Lothar Mattheus" );
e.age = 28 ;
e.bs = 5575.50 ;
e.department = manufacturing ;

printf ( "Name = %s\n", e.name ) ;
printf ( "Age = %d\n", e.age ) ;
printf ( "Basic salary = %f\n", e.bs ) ;
printf ( "Dept = %d\n", e.department ) ;

if ( e.department == accounts )
    printf ( "%s is an accountant\n", e.name ) ;
else
    printf ( "%s is not an accountant\n", e.name ) ;
return 0 ;
}
```

OUTPUT:

```
Name = Lothar Mattheus
Age = 28
Basic salary = 5575.50
Dept = 1
Lothar Mattheus is not an accountant
```

Uses of Enumerated Data Type

- The **program** first **assigns values** to the **variables** in the **structure**.
- The statement, **e.department = manufacturing** ; assigns the value **manufacturing** to **e.department** variable.
- This is much **more informative** to anyone **reading the program** than a statement like, **e.department = 1** ;
- One important **weakness** of using **enum variables**... there is **no way to use** the **enumerated values** directly in **input/output functions** like **scanf()** and **printf()**.
- The **printf() function** is not smart enough to perform the translation; the **department** is **printed out** as **1** and **not** **manufacturing**.

Renaming Data types with Typedef

- **typedef declaration**
- Its **purpose** is to **redefine** the name of an **existing variable type**.
- For **example**, consider the following statement in which the **type unsigned long int** is **redefined** to be of the type **TWOWORDS**:
- **typedef unsigned long int TWOWORDS ;**
- Now we can **declare variables** of the **type unsigned long int** by writing:
- **TWOWORDS var1, var2 ;** instead of **unsigned long int var1, var2 ;**
- Thus, **typedef** provides a **short and meaningful way** to use a **data type**.
- Usually, **uppercase letters** are used to make it clear that we are dealing with a **renamed data type**.

Renaming Data types with `typedef`

- It can be **significant** when the **name** of a **particular data type** is **long** and **unwieldy**, as it **often** is with **structure declarations**.

- For **example**,

```
struct employee
{
    char name[ 30 ] ;
    int age ;
    float bs ;
};

struct employee e ;
```

- This **structure declaration** can be made **more handy** to use when **renamed** using **`typedef`** as shown below:

```
typedef struct employee EMP ;
EMP e1, e2 ;
```

Renaming Data types with `typedef`

- Thus, by **reducing the length and apparent complexity of data types**, `typedef` can help to **clarify source code** and **save time and energy spent in understanding a program.**
- The previous `typedef` can also be written as:

```
typedef struct employee
```

```
{
```

```
    char name[ 30 ] ;
```

```
    int age ;
```

```
    float bs ;
```

```
} EMP ;
```

```
EMP e1, e2 ;
```

Renaming Data types with `typedef`

- `typedef` can also be used to **rename pointer data types** as shown below:

```
struct employee
```

```
{
```

```
    char name[ 30 ] ;
```

```
    int age ;
```

```
    float bs ;
```

```
}
```

```
typedef struct employee * PEMP ;
```

```
PEMP p ;
```

```
p -> age = 32 ;
```

Typecasting

- Sometimes we are required to **force** the **compiler** to **explicitly convert the value of an expression to a particular data type.**

- **Example:**

```
int main( )
{
    float a ;
    int x = 6, y = 4 ;
    a = x / y ;
    printf ( "Value of a = %f\n", a ) ;
    return 0 ;
}
```

- **Output:**

- Value of a = 1.000000

Typecasting

- The answer turns out to be **1.000000** and not **1.5**.
- This is because, **6** and **4** are **both integers** and hence **6 / 4 yields an integer, 1.**
- This **1** when stored in **a** is **converted** to **1.000000** .
- But what if we don t want the quotient to be truncated?
- One solution is to make either **x** or **y** as **float**.
- Let us say that other requirements of the **program** do not permit us to do this.
- In such a case what do we do? Use **typecasting**.

Typecasting

```
int main( )
{
    float a ;
    int x = 6, y = 4 ;
    a = ( float ) x / y ;
    printf ( "Value of a = %f\n", a ) ;
    return 0 ;
}
```

- And here is the output...
- Value of a = 1.500000
- This program uses **typecasting**.
- The **expression (float)** causes the **variable x** to be **converted** from type **int** to type **float** before being used in the division operation.

Typecasting

```
int main( )
{
    float a = 6.35 ;
    printf ( "Value of a on type casting = %d\n", ( int ) a ) ;
    printf ( "Value of a = %f\n", a ) ;
    return 0 ;
}
```

- **Output:**
- Value of a on type casting = 6
- Value of a = 6.350000
- **Note** that the **value** of **a** does not get permanently **changed** as a result of **typecasting**.
- Rather it is the value of the expression that undergoes type conversion whenever the **cast appears**.