

```
complement(int L...){  
    void main(void)  
    {  
        int A[max][max], B[max][m...  
        system("clear");  
        printf("\n\tRandom Graph g...  
        printf("\n\tEnter number of
```

INTRODUCTION TO PROGRAMMING IN C

Prof. Satyadev Nandakumar
Computer Science & Engineering
Indian Institute of Technology, Kanpur



INDEX

S.No	Topic	Page No.
1	Intro - Process of programming	01
2	Intro - GCD	08
3	Intro - Programming cycle	20
4	Intro - Tracing a simple program	31
5	Intro - Variables	38
6	Intro - Operators	47
7	Loops - While	55
8	Loops - While example	59
9	Loops - While GCD example	64
10	Loops - Longest 1	70
11	Loops - Longest 2	76
12	Loops - Longest 3	81
13	Loops - Do-while	84
14	Loops - Matrix using nested loops	89
15	Loops - For	97
16	Loops - Matrix using nested for loops	106
17	Loops - Break statement	111
18	Loops - Continue statement	122
19	Loops - Continue statement example	128
20	Data types in C	134
21	ASCII code	142
22	Operators Expressions Associativity	150
23	Precedence of operators	158
24	Expression evaluation	164
25	Functions - Introduction	176
26	Functions - How functions are executed	190
27	Functions - Examples - 1	198
28	Functions - Examples - 2	210
29	Arrays in C	218
30	Initializing arrays	233

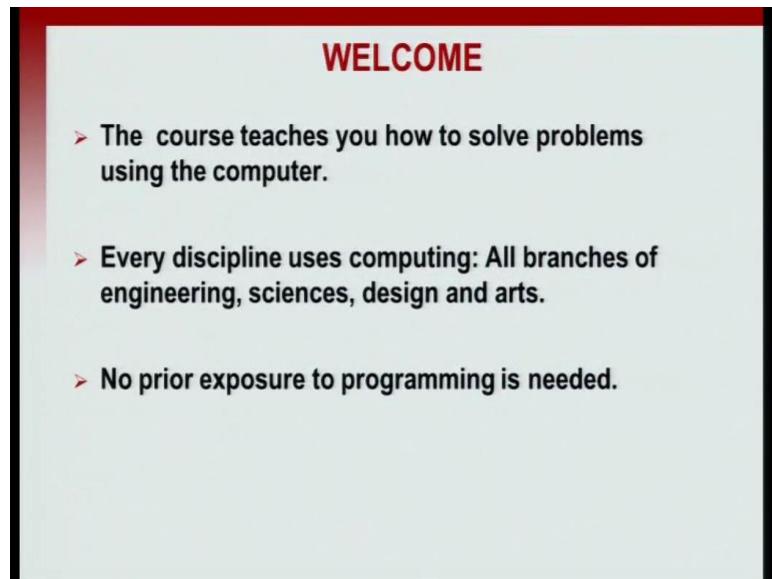
31	Initializing character arrays	240
32	Pointers in C	245
33	Pointer arithmetic	254
34	Function with pointer arguments	261
35	Example - copy a subarray	270
36	Programming using arrays and pointers	279
37	Sizeof operator	288
38	Returning pointers from functions	295
39	Example - return duplicate of a string	305
40	Recursion - Linear Recursion	313
41	Recursion - Linear Recursion - 2	322
42	Recursion - Two-way Recursion	335
43	Multidimensional Arrays	345
44	Multidimensional Arrays and Pointers	359
45	Multidimensional Arrays and Pointers - continued (2)	369
46	Multidimensional Arrays and Pointers - continued (3)	376
47	File Handling	385
48	Some other file-handling functions	395
49	Structures in C - 1	401
50	Structures in C - 2	411
51	Singly Linked Lists	416
52	Doubly Linked Lists - introduction	428
53	Organizing code into multiple files - 1	438
54	Organizing code into multiple files - 2	449
55	Pre and post increment	460

Introduction to Programming in C

Department of Computer Science and Engineering

Welcome to the introductory programming course on NPTEL MOOCs. The goal of this is to learn how to code basic programs in the C programming language.

(Refer Slide Time: 00:18)



Basically the aim of this course is to teach you how to solve problems using a computer. And by the end of this course, we will hope that you can write medium-sized programs – maybe running to a couple of 100 lines of code comfortably in the C programming language. Programming nowadays is considered a basic skill similar to mathematics that is needed across all disciplines like engineering, in the sciences, and nowadays even in the arts. So, little bit of programming skill is an enhancement to any other skillset that you might already have. This course we will start from the ground up; we do not assume any prior experience in programming whether in C or in any other language. So, the focus will be to start from the basics; and to use C as a medium of program.

(Refer Slide Time: 01:21)

Process of Programming

1. Define and model the problem. In real-life this is important and complicated. For example, consider modeling the Indian Railways reservation system.
2. In this course, all problems will be defined precisely and will be simple.

A couple of words about the process of programming; it involves two basic steps. One is to define the problem; often you get real-world problems, which are not precise enough to write a program for. So, the first step would be to define and model the problem. And this is a very important step in large scale software development; however we will not focus on this as part of this course. During this course, you will not write large software system like the Indian railways reservation system; those are extremely complex problems involving multiple programmers. In this course, we will assume that the problem is well-defined and already provided to you. So, they will be precise and they will be fairly short and simple. So, this is the first step of programming, which is definition of the problem, which you can assume will be given.

(Refer Slide Time: 02:22)

Process of Programming: Step 2

- Obtain a logical solution to your problem.
- A logical solution is a finite and clear step-by-step procedure to solve your problem.
- Also called an Algorithm. We can visualize this using a flowchart.
- Very important step in the programming process.

Now, comes the second step, which is to obtain logical solution to your problem. And what do we mean by a logical solution? A logical solution is a finite sequence of steps; do this first, do this next; if a certain condition is true do this; otherwise, do something else. This is called an algorithm. So, an algorithm is basically a finite step-by-step procedure to solve a problem. One way to visualize an algorithm is using a flowchart. If you are new to programming, it is recommended that, you draw flowcharts to define the solution to your problem. Experienced programmers very rarely draw flowcharts, but that is not a reason for beginning programmers to avoid flowcharts.

(Refer Slide Time: 03:29)

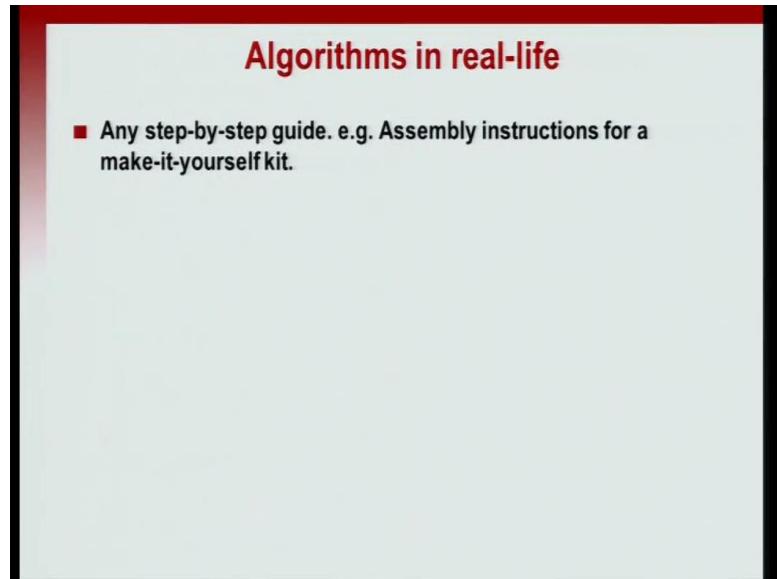
Algorithms in Ordinary Life? (Recipes!)

- An algorithm is a familiar concept: cooking recipes are almost algorithms! (not quite precise enough for a computer!)
- Ingredients
 - 1 liter (33 oz) ice cream (any flavor).
 2. Crushed cereal, such as corn flakes, frosted flakes, cinnamon squares, or puffed rice.
 3. Flour (a small bowl of it, approx 1/2 cup).
 4. Oil (use an unflavoured oil that has a high heat point).
 5. 2 eggs (beaten in a bowl large enough for dipping).
 6. Cinnamon and/or sugar (optional).
- Instructions
 1. Prepare the two baking sheets by lining with a silicon liner or parchment paper. Then place the sheets in the freezer for half an hour prior to making the ice cream balls.
 2. Scoop symmetrical balls of ice cream. Try to make each scoop about as large as your fist. Make as many scoops as will fit on the baking sheets.
 3. Harden the scooped ice cream balls in the freezer.
 4. Set out the bowls for dipping. Place a bowl of flour, a bowl of beaten egg and a bowl of crushed cereal or fine cookie/cracker crumbs in the workspace, in a formation that makes it easy to dip, in order.
 5. Coat the ice cream.
 6. Place the ice cream balls back on the baking sheets, then back in the freezer.
 7. Fry the coated ice cream balls. Heat up the oil until it shimmers - approx 185C.
 8. Serve the ice cream balls.

So, defining a problem is there; then the process of coming up with an algorithm. This is a very important step in the programming process. And followed by this, there is a third step, which is to implement the algorithm in a usual programming language. So, is the concept of an algorithm a new concept? I would claim that, it is not. An algorithm is a very familiar concept; the most important example that you can think of are cooking recipes. Now, cooking recipes are written in a way that, they are almost algorithms. They are not quite precise enough for a computer, but they come quite close. For example, let us take an unnamed dish – a desert and let us look at how things specified in a recipe. And we will see that, this analogy is quite deep. There is a very strong similarity in the way that recipe is written and a program is written. So, usually, they will have a list of ingredients upfront. For example, you have ice cream, crushed cereal and so on. And then once you have all the ingredients in place, then you have instructions to say how do you start and how do you end up with the dish.

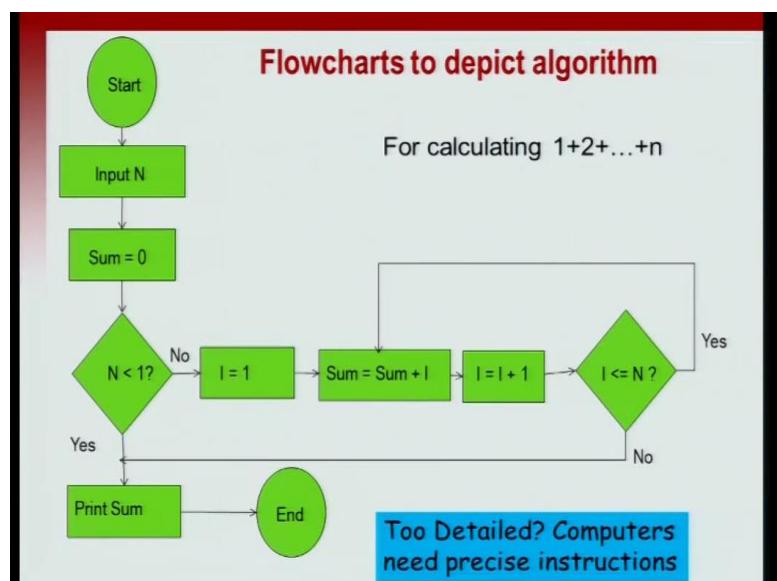
Now, those instructions will be fairly precise; of course, you assume that, the person preparing the dish is a fairly-experienced cook, so that certain instructions need not be given in very precise detail. For example, you can say do this, heat oil and so on. And it is assumed that, a person knows how to heat oil. Even so you will see that, certain recipes are fairly vague and other recipes are fairly detailed. And in any recipe, you can see certain things, which are vague and will cause confusion to most people. For example, here is a term, which says try to make each scoop about as large as your fist. Now, that of course, is a vague term, because my fist could be a different size than yours. And then you will see that, in a formation that makes it easy do dip in order. So, this is fairly vague and it is not very helpful to a cook, who is making this for the first time. So, think of algorithms as similar to recipes, but mentioned in a more precise manner.

(Refer Slide Time: 06:12)



Another way you can be familiar with algorithms is when you have the – when you buy a make it yourself kit for a furniture or something like that; and you will be provided with a step-by-step instructions on how to assemble the kit. Often when you buy disassembled table or something like that, it will come with a sheet telling you how to start with the components and build a table. Those are also similar to an algorithm.

(Refer Slide Time: 06:41)

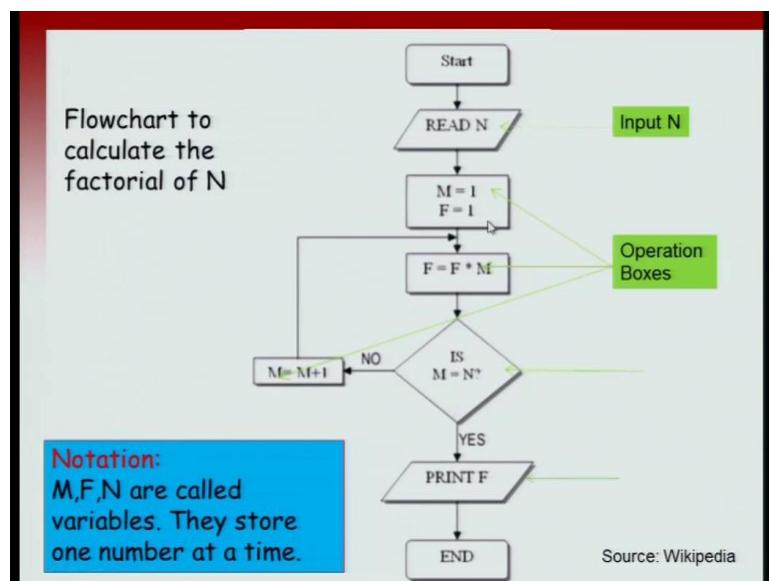


So, let us look at a flowchart to depict a mathematical algorithm and we will use this flowchart to explain certain conventions about how algorithms can be described. So,

every flowchart will have a start and an end; and it will have a finite number of boxes. So, this is the finite number of instructions that I was talking about. There are certain conventions in drawing flowcharts; the start and the end are often described in circles. Then there are ordinary boxes and then there are diamonds. We will shortly describe what they mean. So, suppose you want to write an algorithm for adding the first n numbers; all of you know how to do it. The point is how do you describe this step-by-step to somebody who does not know it already.

So, first you have to take what is the upper limit N and then you have to sum them up. So, one way to sum them up is start with an initial sum of 0 and then add numbers one by one. So, increment a counter from 1 all the way up to n. So, you start with I equal to 1 and then add the I-th number to the sum; and then increment I; if I is already N, then you are done; if I is not N, then you go back and do the sum all over again until you hit an I. When you reach I equal to N, you come out the program; print the sum; and end the program. So, this is a very simple flowchart. So, initially, if N is less than 1, you have nothing to do; if N is greater than 1, you start a counter from I equal to 1 to n and add the numbers one by one until you hit the N-th number.

(Refer Slide Time: 08:56)



If you wanted to compute a slightly different problem, which is let us say the factorial of N, which is just a product of the first N numbers, the flowchart will look fairly similar; the only difference is that instead of adding numbers, you will multiply them. So, this

flowchart is similar to the previous flowchart; you will first input in N, and then increment N until you hit N equal to M. If so you will finally, print the factorial; otherwise, you go back to the loop. So, here are the conventions used. The start symbol is often ((Refer Time: 09:45)) as a circle or an oval; the input symbol and the output symbol are often represented as parallelograms; and the normal operation boxes are represented as rectangles; and the test box to see whether you have hit a limit to test some condition in general, they are represented as diamonds.

Introduction to Programming in C

Department of Computer Science and Engineering

In this session, we will write another algorithm to solve a mathematical problem. If you do not know this algorithm already, that is fine; it is more for the purpose of demonstrating, if you know a solution, how do you come up with the algorithm to tell a computer how to solve it.

(Refer Slide Time: 00:27)

GCD

- An algorithm to find the greatest common divisor of two positive integers m and n , $m \geq n$.
- A naïve solution – Described *informally* as follows.
 1. Take the smaller number n .
 2. For each number k , $n \geq k \geq 1$, in descending order, do the following.
 1. If k divides m and n , then k is the gcd of m and n
- This will compute gcd correctly, but is VERY slow (think about large numbers m and n).
- There is a faster way...

The algorithm is for finding the greatest common divisor or the highest common factor, this is known under two names of two positive integers: m and n . So, this is an algorithm you probably know. How do you solve this? Let us first try a naive solution. And before writing an algorithm, let us see what do I mean by the simple solution of GCD. So, you are asked to find the greatest common divisor of m and n ; take the smaller number n ; and now you start looking for each number k between 1 and n , remember that n is the smaller number; in descending order, do the following. What you do is if k divides m and n , then k is the greatest common divisor of m and n .

And this is obvious by the definition of greatest common divisor; if k divides m and n , then it is obviously a divisor of m and n . Also we are coming in descending order; we start from n and go down to 1. So, the first divisor that you hit when you go down is going to be the greatest common divisor of m and n . So, this algorithm obviously works. It will compute the GCD correctly, but it is very slow. And think about a very large

numbers: m and n; and you will see that, it may go n steps before reaching the correct GCDs. So, compute the GCD of two very large numbers, which are relatively prime to each other; that means that the GCD of m and n are 1. Now, if you pick such a pair, this algorithm will compute the GCD correctly, but it will take n steps, because you have to go down all the way from n to 1 before you will hit the GCD. Can we do better? There is a faster way and it is a very old algorithm.

(Refer Slide Time: 02:40)

GCD Algorithm - Intuition

To find gcd of 8 and 6. Consider rods of length 8 and 6. Measure the longer with the shorter. Take the remainder if any. Repeat the process until the longer can be exactly measured as an integer multiple of the shorter.

$\text{Gcd}(8, 6) = 2.$

The algorithm is due to Euclid. We will see a slightly modified version of that algorithm. So, before we go into Euclid's algorithm for GCD, we will describe what it does and give you a slight intuition of why it works. So, consider the GCD of 8 and 6. Now, you can consider two rods: one of length 8, and another of length 6. Now, obviously, if a number divides 6 and 8, then I should be able to make a stick of that length, so that I can measure 6 exactly with that shorter rod; and I can measure 8 exactly with that shorter rod. This is the meaning of a common divisor, and we have to find the greatest common divisor.

So, first, what we will do is we will measure the longer rod using the shorter rod. Now, it may not measure the longer rod exactly. For example, in this case, 6 does not measure 8 exactly; there will be a small piece of length 2 left over. So, take that remainder. And now, repeat the process; now, 2 has become the shorter rod and 6 has become the longer rod. Now, see if 2 measures 6 exactly; it does. So, you are done. And then you can say that, 2

is the GCD of 8 and 6. The reason why this works is – by the nature of this algorithm, it is clear that 2 divides 6, because that is why we stop the algorithm. And also, we know that, 8 is basically 6 plus 2. So, it is obviously, a multiple of 2. So, it is a common divisor. And with a slightly more elaborate argument, we can argue that, it is the greatest common divisor. So, this is an algorithm, which is essentially due to Euclid. So, it was known for at least 2000 years.

(Refer Slide Time: 05:00)

GCD Algorithm - Intuition

$$\begin{array}{r}
 & 102 \\
 & \overline{)102 \text{ mod } 21 = 18} \\
 21 & \\
 \overline{)21} & 21 \text{ mod } 18 = 3 \\
 18 & \\
 \overline{)18} & \\
 3 &
 \end{array}$$

$\text{Gcd}(102, 21) = 3$

Let us pick a slightly more elaborate example. Let us say we want to find the GCD of 102 and 21. The process of taking remainder is what is known as the modulo operator in mathematics. So, 102 modulo 21 is the remainder of integer division of 102 by 21. So, the remainder of when you divide 102 by 21 is 18. So, that is the shorter rod for the next stage. Now, 21 mod 18 is 3. And that becomes the rod for the next stage; the shorter rod for the next stage. And 18 mod 3 is 0; that is when you stop the algorithm. So, when the modulo operator gives you 0 result; that means that, the shorter number exactly divides the larger number; that means that, the shorter number is a divisor of the larger number and you stop the algorithm. Now, you say that, GCD of 102 and 21 is 3.

(Refer Slide Time: 06:18)

Euclid's method for gcd

Euclid's algorithm (step-by-step method for calculating gcd) is based on the following simple fact.

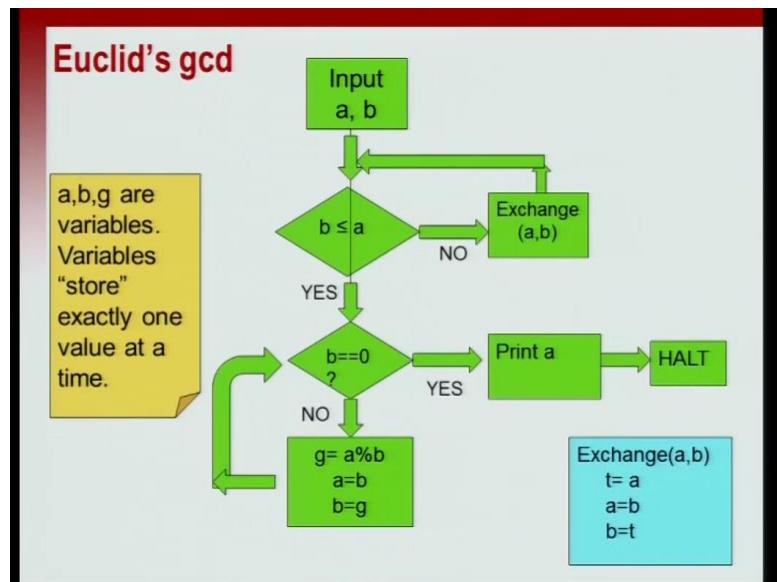
Suppose $a > b$. Then the gcd of a and b is the same as the gcd of b and the remainder of a when divided by b .

$$\gcd(a, b) = \gcd(b, a \% b)$$

To see this consider division of a by b
 $a = bq + r$

So, this is a slight modification of the classical Euclid's method for GCD. And so, it is based on the following simple fact, which we have described. And you can prove this mathematically as well. So, suppose you take two positive numbers: a and b ; where, a is the larger number; then GCD of a and b is the same as GCD of b and the remainder when you divide a by b . So, it is written by the equation $\text{GCD}(a, b)$ is $\text{GCD}(b, a \% b)$. The modulo operator is represented as the percentage sign, because this is the convention that we will use in C. And this equation can be seen by our previous slide; a was the bigger rod; b was the shorter rod. This was the first stage. The second stage was when b is the shorter rod. And the shorter rod for the next stage is modulo – is given by the modulo operator. To prove this, you can start by considering the division of a by b and writing a as $bq + r$. But, we will not go into the proof. From elementary properties of natural numbers, it is possible to prove that, Euclid's method correctly computes the GCD.

(Refer Slide Time: 07:48)

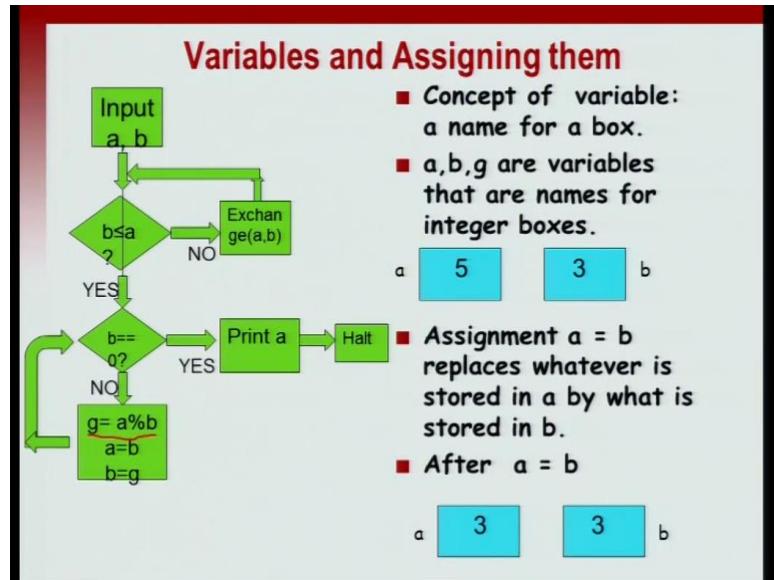


Right now, we will move into how do we write the GCD algorithm in the form of an input. So, here is a slightly abbreviated picture. I have skipped the start state; but the start state is there. Let us focus on what happens during the algorithm. You have two numbers: a and b. The first thing to ensure is that, a is the larger number. The reason we do that is that, if a is the larger number, then the modulo operator is properly defined. So, if a is the larger number, then we are fine; we can go into the GCD algorithm. If a is not the larger number, you merely swap a and b, so that whatever is the larger number, you called it a. So, exchange a and b; means that you say that, the value of a is stored in a temporary variable; then the value of b is stored in a; and then the value of b is stored in t with the value of t stored in b. So, here is a way to exchange the values of a and b. So, ensure at first that, a is the larger number. Once you do that, you get into the code for the proper utility in GCD.

First you test whether b is 0. If b is 0, then there is nothing to do; a is the GCD of a and b; $\text{GCD}(4, 0)$ is 4; $\text{GCD}(4, 0)$ is 8, and so on. So, if the smaller number is 0, then there is nothing to do in the algorithm; the algorithm is over; and you say that, print a. If b is not 0, then we do the Euclidean equation. You take $a \% b$; store it in a variable g; then assign the value of b to a and assign the value of g to b. So, this corresponds to the operation of taking b and $a \% b$ as the next step. After you do that, you again test the condition whether b has now become 0. If it is 0, then we are done and a is the GCD; otherwise, we do another round of taking $a \% b$ and setting $a = b$ and $b = g$. So, a, b and g are what are

known as variables. And variables are used in programming to store exactly one value at a time. So, at any particular time, it will have one value; then after the execution of another instruction, it will have a new value and so on.

(Refer Slide Time: 10:43)



Now, for the purposes of describing an algorithm, imagine that, the variable is a box; and it is a name of a box; and the value is stored inside the box. For example, a, b and g are the variables that we have used in the program. And they are the names for these integer boxes. So, if we are computing, let us say the GCD of 5 and 3, then you might start with a equal to 5 and b equal to 3. The second operation that we have used in the code is the assignment operation. So, this is what an example of the assignment operator. And when we do an assignment, what we mean is that, you take the left variable, which is g in this case and assign it the value of what is the expression on the right-hand side, which is $a \% b$. So, assignment $a = b$ replaces whatever is stored in a by what is stored in b. So, take the right-hand side; take the value of that; and put it into the variable that the left-hand side represents. For example, if a was 5 and b is 3; after $a = b$, you would take the value of b and put it in a. So, a will now become 3 and b will remain 3.

(Refer Slide Time: 12:17)

Sequential assignments

$g = a \% b;$
 $a = b;$
 $b = g;$

initially
a b g
10 6 ??

- Semi-colons give a sequential order in which to apply the statements.
- Variables are boxes to which a name is given.
- We have 3 variables: a, b, g. This gives us three boxes. Initially, a is 10, b is 6 and g is undefined.

Another small thing that we have used in the code is sequential assignment. So, if you write a bunch of statements one after the other, let us say separated by semicolons; then this means that, the instructions are to be executed one after the other in sequence. So, first, you do g equal to $a \% b$; then you do $a = b$; and after that you do $b = g$. So, initially, let us say that a is 10 and b is 6; g is undefined.

(Refer Slide Time: 12:50)

Sequential assignments

$g = a \% b;$
 $\hat{a} a = b;$
 $b = g;$

After $g = a \% b$
a b g
10 6 4

- Semi-colons give a sequential order in which to apply the statements.
- Variables are boxes to which a name is given.
- We have 3 variables: a, b, g. This gives us three boxes. Initially, a is 10, b is 6 and g is undefined.
- Run statements in sequence.
- \hat{a} Next statement to run

After you run the statement g equal to $a \% b$, you take 10 modulo 6; you will have 4.

(Refer Slide Time: 12:59)

Sequential assignments

```
g = a%b;
a = b;
b = g;
```

After $a = b$

a 6	b 6	g 4
--------	--------	--------

- Semi-colons give a sequential order in which to apply the statements.
- Variables are boxes to which a name is given.
- We have 3 variables: a , b , g . This gives us three boxes. Initially, a is 10, b is 6 and g is undefined.
- Run statements in sequence.
- Next statement to run

And then $a = b$; the value of b will be stored in a . So, a becomes 6. And then $b = g$; the value of g will be stored in b . So, b will become 4.

(Refer Slide Time: 13:12)

Running the program

```
Input a, b
IF b <= a THEN
    Exchange(a,b)
    IF b == 0 THEN
        Print a
        Halt
    ELSE
        g = a%b
        a = b
        b = g
        Go to step 2
END IF
```

Program counter. At the next step to be executed. Initially at beginning.

State of the program is variables : boxes with names.

a	b	g
---	---	---

Now let us start running the flowchart. One step at a time.

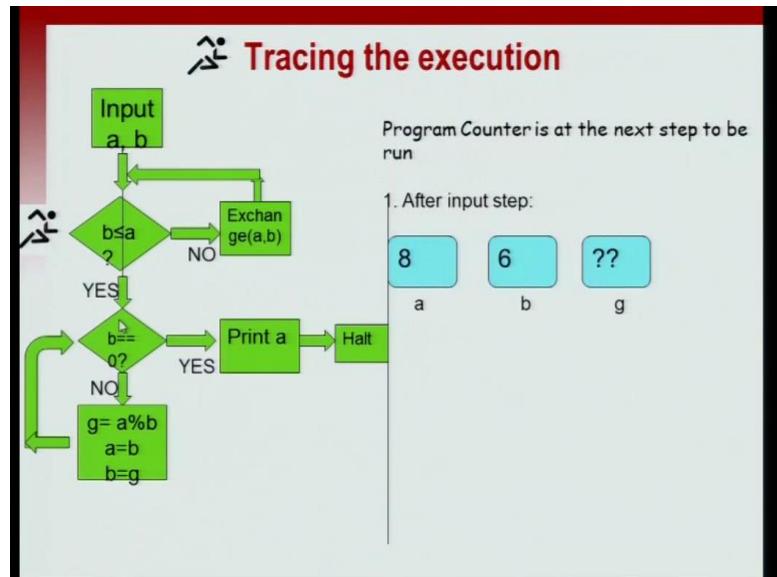
1. After input step:

8	6	??
a	b	g

Now, let us just dry run the program or the algorithm and see how it computes the GCD of two numbers. So, I will denote the currently executing statement with an icon and I will call this the program counter. So, this is at any point, it is the next step to be executed. Initially, it is at the beginning of the code; where, you take the input. And we will have three variables, which will represent the current state of a program. So, suppose

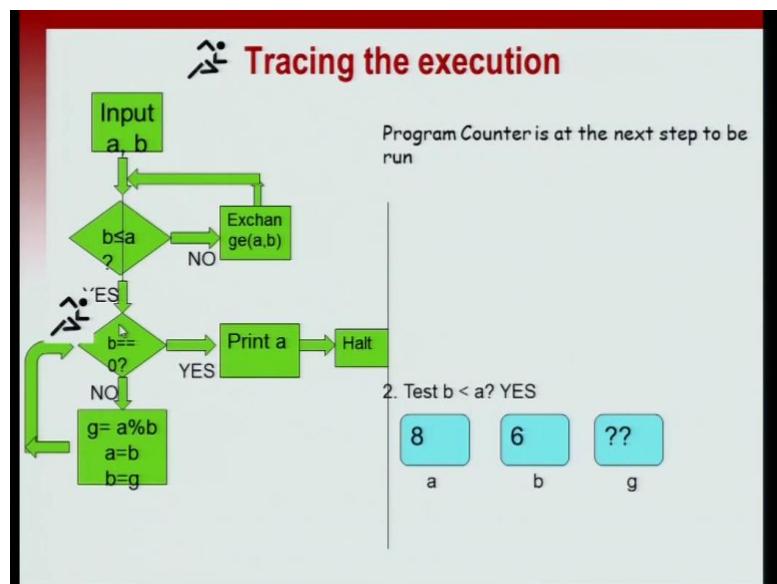
you want to compute the GCD of 8 and 6. So, you have a equal to 8; b equal to 6. You know that a is greater than b.

(Refer Slide Time: 14:02)



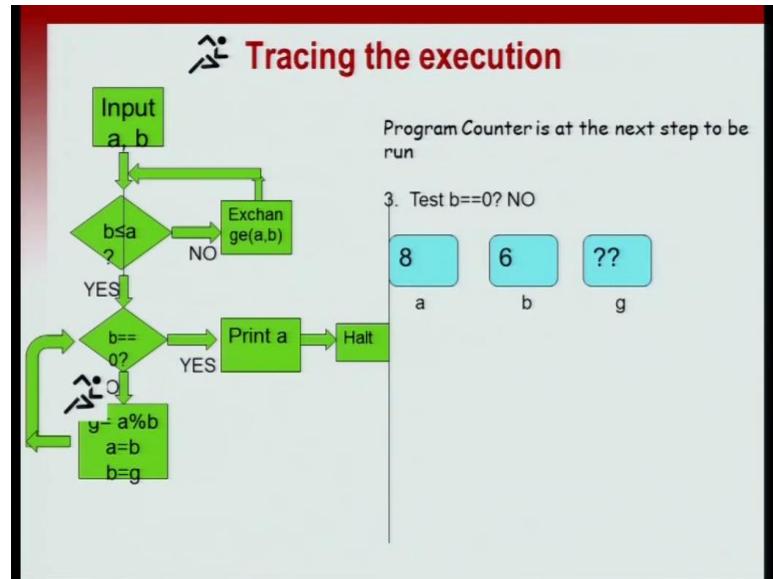
So, you proceed.

(Refer Slide Time: 14:09)



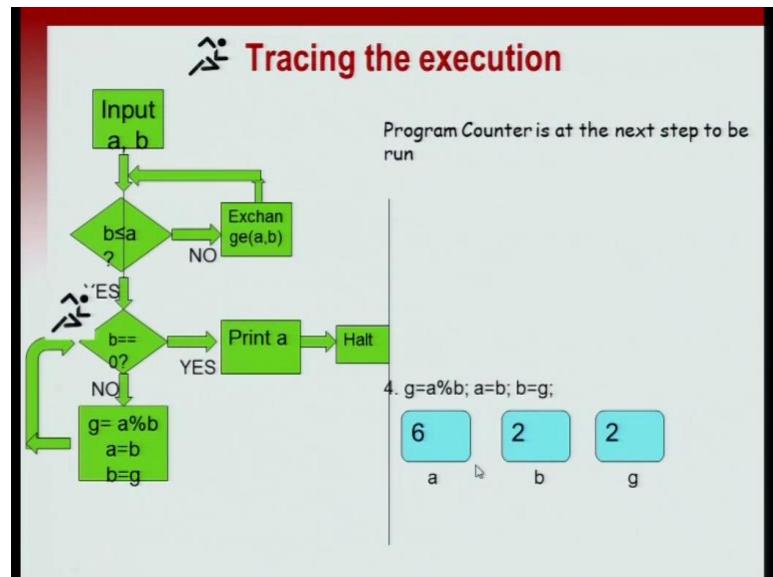
Now, you test whether b is 0.

(Refer Slide Time: 14:17)



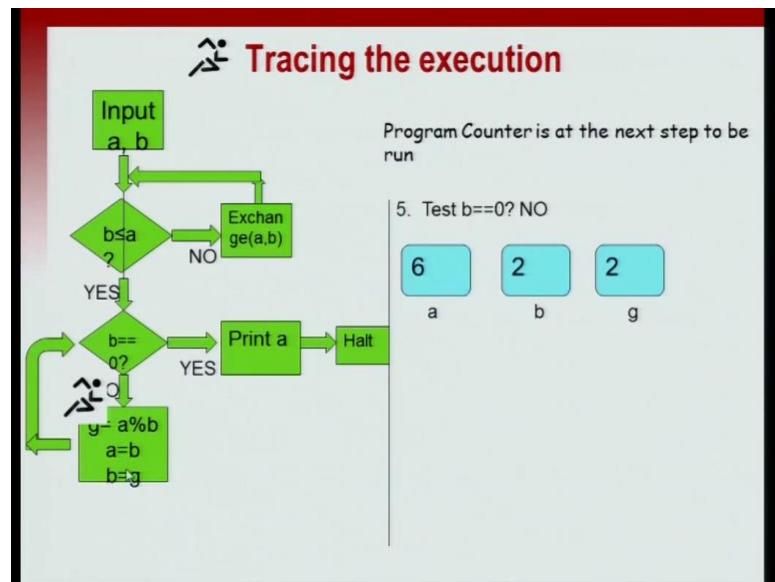
So, since b is non zero, you go into the main body of the loop.

(Refer Slide Time: 14:22)



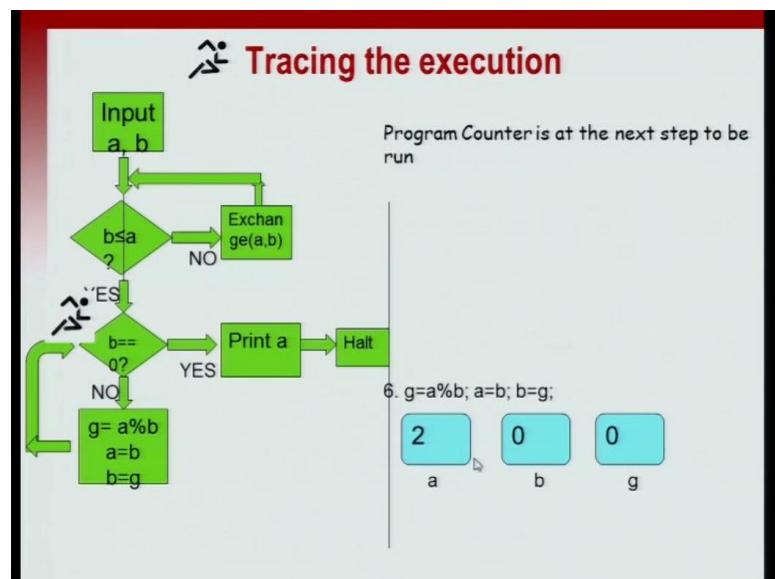
So, you do g equal to $a \% b$; $a = b$; $b = g$, this step once. So, you will end up with a is now 6; b is 2; and g is 2. You again comeback to the discussion and test whether b is 0 or not; b is not 0.

(Refer Slide Time: 14:44)



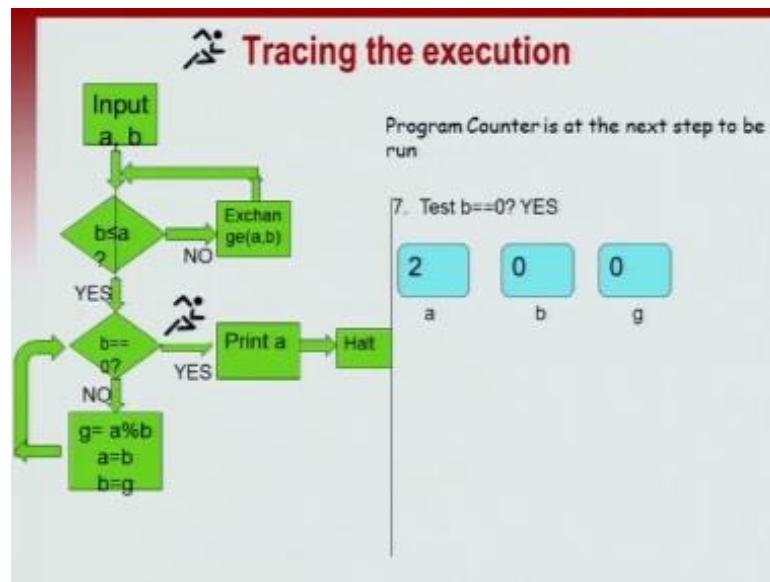
So, you go back into the body of the loop again. So, you have g to be $a \% b$. So, 6 modulo 2 should be 0.

(Refer Slide Time: 14:58)



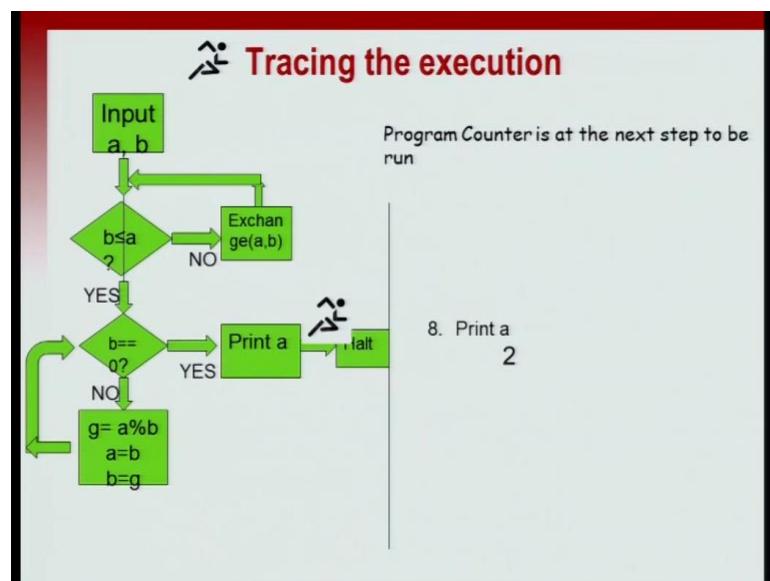
And then you do $a = b$ and $b = g$. You will have a equal to 2; b equal to 0; and g equal to 0. At this point, b is now 0.

(Refer Slide Time: 15:20)



So, you say that a is actually the GCD of three numbers – of the numbers 8 and 6.

(Refer Slide Time: 15:22)

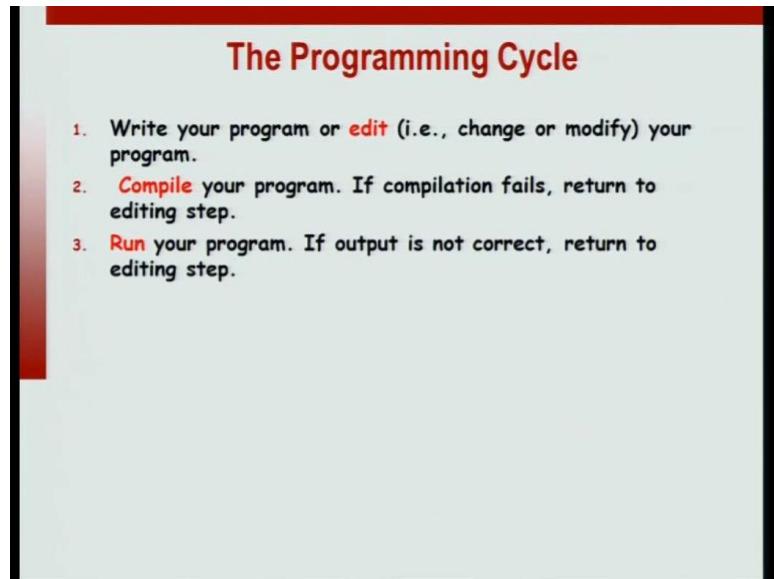


So, you can ensure that, it computes the GCD correctly.

Introduction to Programming in C

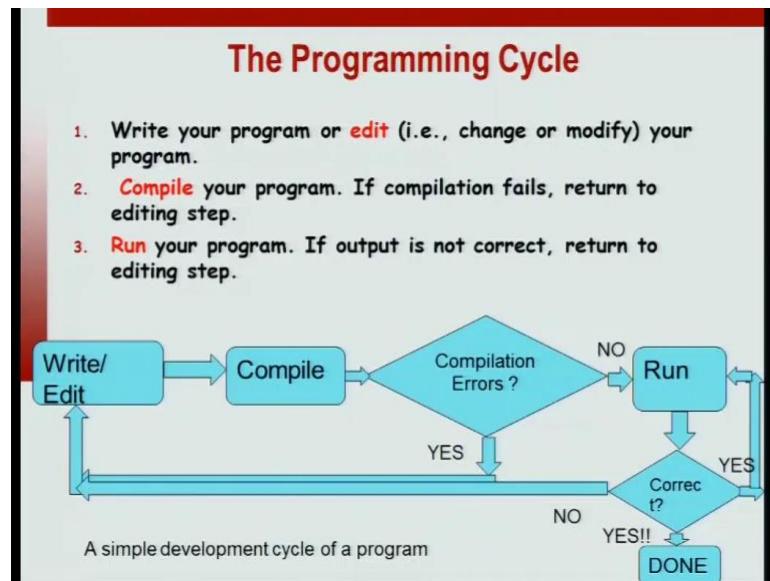
Department of Computer Science and Engineering

(Refer Slide Time: 00:22)



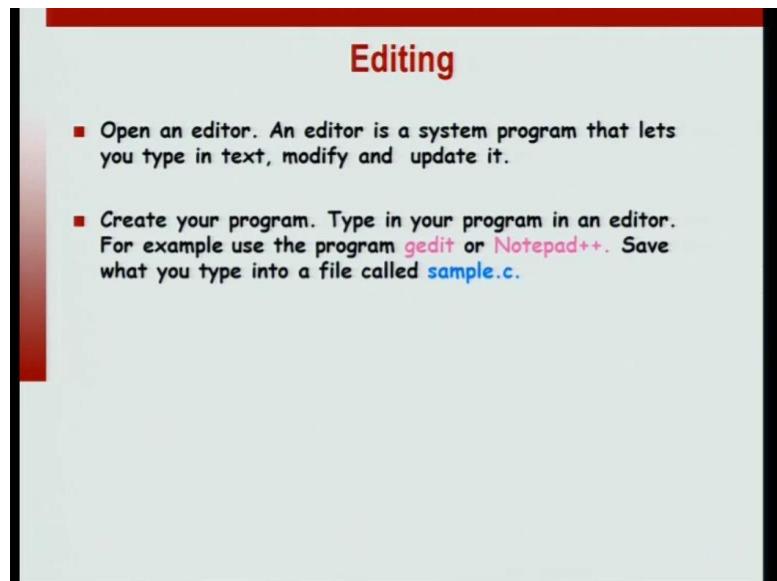
Once we are understood what algorithms are, we will start writing a few simple programs in the C programming language. Before we begin, we will give a brief introduction to the process of programming. When you are programming, you follow typically, what is known as the programming cycle and this contains three parts. One is the process where you write the program or edit the program, and after you are done editing the program, you save it and then you compile your program. If your compilation succeeds, you are ready to run the program. If your compilation fails, then you return to the editing step and correct the errors and compile again. Once compilation process succeeds, then you can run the program and check whether the output is correct. If the output is correct, you are done; if not you go back to the edit process.

(Refer Slide Time: 00:59)



So, this is why it is known us the edit, compile, run cycle. So, you edit the program first, then compile it. If there are compilation errors you go back and edit it again, otherwise you run the program. When you run the program, if the logic is correct, then you are done. If your logic is incorrect, then you go back and make changes to the program, compile it and run it again. So, this is the process, that we have to follow in the, when we program. We look at each of the steps one by one.

(Refer Slide Time: 01:38)



In editing, it is typically done in what is known as an editor. Now, editor is a program that lets you create a text file, make changes to the text file and update the text file, later save it. So, in order to create a program, pick up particular editor of your choice. If you are on Linux, I would recommend a simple editor like G Edit. If you are on windows, there is a free editor called `Notepad ++`. Be careful that this is not the usual notepad that comes along with the system. Write your code in, in editor of your choice and save it into a file. Let us call it `Sample.c`.

(Refer Slide Time: 02:25)

Step 2: Compile

- After editing, you have to **COMPILE** the program.
- The computer cannot execute a **C** program or the individual statements of a **C** program directly.
- For example, in **C** you can write **$g = a \% b$**

$g = a \% b$

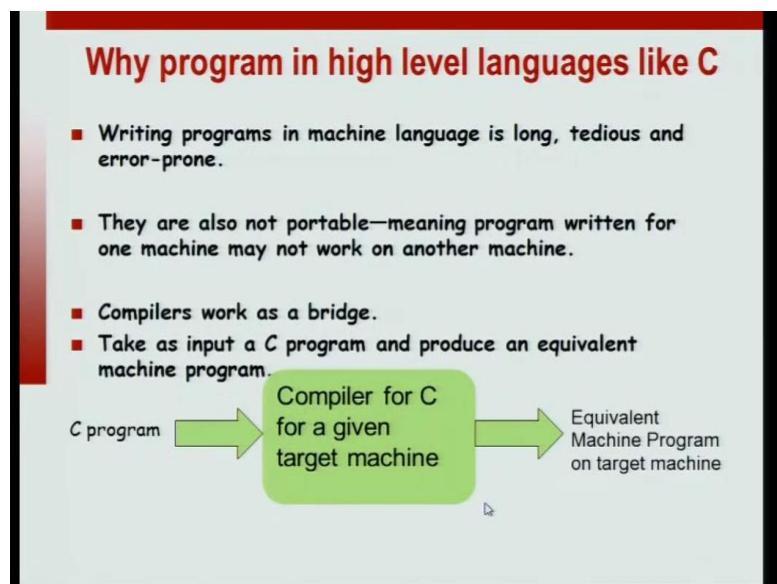
- The microprocessor cannot execute this statement. It translates it into an equivalent piece of code consisting of even more basic statements. For example
 - ▼ Load from memory location **0xF04** into register **R1**
 - ▼ Load from memory location **0xF08** into register **R2**
 - ▼ Integer divide contents of **R1** by contents of **R2** and keep remainder in register **R3**
 - ▼ Store contents of **R3** into memory location **0xF12**.

Once your code is saved, you have to compile a program. Now, why do we have to compile a program? Why is this step necessary? The computer does not understand C per say, it cannot execute a C program or the individual statements in a C, in C language correctly. For example, let us say that in C you can write **$g = a \% b$** . The percentage operation stands for modulo. So, this statement says that you take **a % b** and assign it to the variable **g**.

Now, the microprocessor, the processor in the computer cannot execute this statement because it does not understand this C programming language. So, it translates it into an equivalent piece of code consisting of even more basic statements. For example, a, this is just for the purpose of illustration and it is not important that you understand exactly what is going on, but in a statement like **$g = a \% b$** , can be translated into bunch of statements saying load data from particular memory location into particular register, load the second piece of data from another memory location to the second register, divide the contents of these two registers, store the remainder in a third register and then finally, take the result and store it into a third memory location.

So, the simple statement that we wrote, **$g = a \% b$** or **$g = a \% b$** , becomes a bunch of basic statements, that the microprocessor can understand and then it executes these statements.

(Refer Slide Time: 04:17)



So, why not program in the microprocessor language or in assembly language? Writing programs in machine language is very tedious. One line in a higher programming language like C translates into multiple lines of machine language. So, writing machine language code is very long and it is very tedious and is particularly prone to errors. Also, they are not portable. If you write machine code for a particular processor, let us say, you are writing the code for an Intel processor and you translate it to an AMD machine, it might not work. Whereas, if you take your C code and compile it in another machine, it will run on the machine.

So, compilers work as a bridge. What they do is, take a high level C programming language and translate it into the equivalent machine code. So, think of them as a translator. So, you, the input is a C program and then you give it to a compiler. The output of the compiler will be the equivalent machine program for whichever machine you want to run it on. So, compiler is a translator, which translates from C to machine code.

(Refer Slide Time: 05:36)

How do you compile?

- On Unix/Linux systems you can **COMPILE** the program using the **gcc** command.

```
% gcc sample.c
%
```
- If there are no errors, then the system silently shows the prompt (%).
- If there are errors, the system will list the errors and line numbers. Then you can edit (change) your file, fix the errors and recompile.
- As long as there are compilation errors, the **EXECUTABLE** file is not created.

How do you compile? We have just seen why we bother with compilation and on UNIX system or Linux systems, you can compile the program using the gcc compiler. So, gcc stands for the gnu c compiler. So, for example, if you have edited and saved your file as a **Sample.c**, you can just type on the comment prompt on the terminal **gcc Sample.c**.

If your code does not have any errors, then the system will silently say, that the compilation is done and it will show you the prompt. If there are errors, the system will list the errors and so, you can go back to the editor, edit you code to correct errors and come back and compile again. As long as there are compilation errors, there will be no executable file created. So, the executable file is the code, is the file that you can finally run. And if there are compilation errors, the compiler will not produce executable code.

(Refer Slide Time: 06:43)

Compilation

- We will use the compiler **gcc**. The command is
`% gcc yourfilename.c`
- gcc** stands for Gnu C compiler.
- If there are no errors then **gcc** places the machine program in an executable format for your machine and calls it **a.out**.
- The file **a.out** is placed in your current working directory.
More on directories in a little bit!

So, name your file as whatever you want, let us call it, **yourfilename.c** and then **gcc yourfilename.c**. It will produce the executable file. If you are on Linux, the executable file that it creates is something called **a.out**. If there are no errors and look at your directory, there will be a new file called **a.out** in your directory and we will explain the directory structures in another session, ok.

(Refer Slide Time: 07:16)

Simple! Program

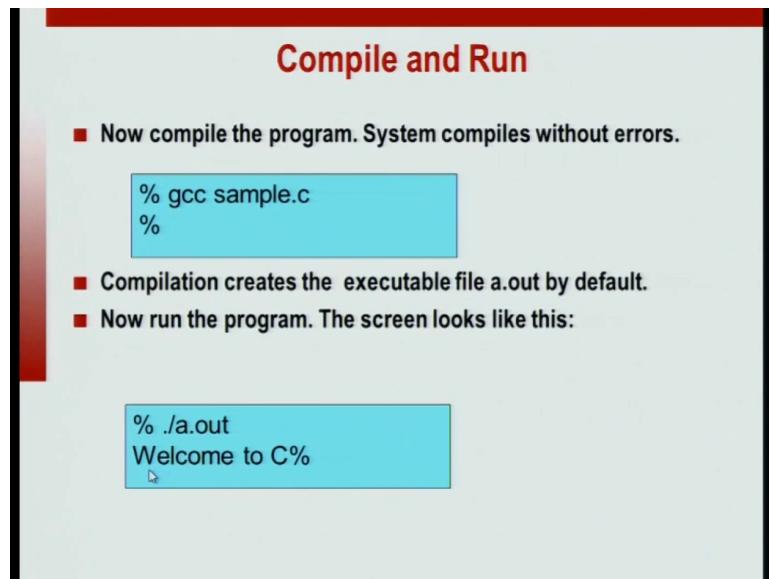
- We will see some of the simplest C programs.
- Open an **editor** and type in the following lines. Save the program as **sample.c**

```
# include <stdio.h>
main () {
    printf("Welcome to C");
}
```

sample.c: The program prints the message "Welcome to C"

Let us look at a very simple C program. Open your editor depending on which system you are in. So, let us write a very simple program. It is, it is very short. What it has is, are three lines of code and some punctuation. This is known as the C syntax. Let us examine this code. What this code does is, it prints a particular message, which is, welcome to C.

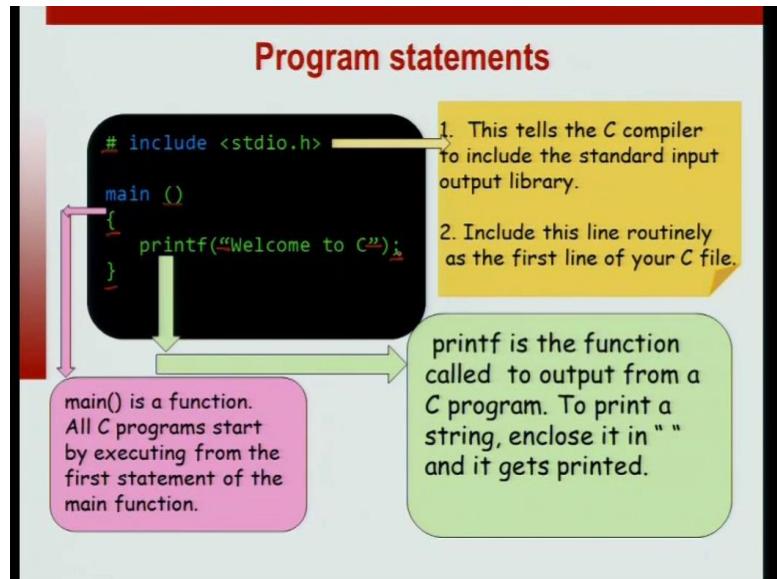
(Refer Slide Time: 07:51)



And it has various components, you type it into an editor as it is, make no punctuation mistakes, syntax errors. Now, if you compile the program and you have typed the program correctly, then a new file called **a.out** will be created. So, if you type, **gcc Sample.c** and if there are no errors, it will just say nothing. If there are, if it says something, then there is a compilation error.

Compilation creates an executable **a.out** and now you can run the program by typing, and this is important, **./a.out**. So, this syntax is important, what you type is, **./a.out** and then when you run the program it will say, welcome dot, Welcome to C, because that is what the program is supposed to do...

(Refer Slide Time: 08:47)



Let us look at the program little more carefully. What are its components? It had three lines, the first line said `# include <stdio.h>`. So, it has multiple components. One is the first symbol, which is, has the first symbol, which is the hash. Please do not forget to include that. And actually, there is no space between the hash and the first i, so there is no space here. So, `# include <stdio.h>`. This line is supposed to tell C that please include the standard input-output library. The standard input-output library is what has the print routines, which will print output messages on to the terminal.

So, if you want to have any input output component of your program, then you should include `<stdio.h>`. Include this line routinely in your, in the first line of your C file because in the course of this class, we will often need `scanf` and `printf` statements. So, we will often need input statement and output statements. So, include this by default.

Now, if you look at the second line, we will have a function called `main`. And again, note the parenthesis here that is also part of the syntax. So, `main` is supposed to be a function. All C programs start by executing the `main` function and it starts from first statements of the `main` function. Now, what dose the `main` function have? It has a single line, which says `printf "Welcome to C"`. So, `printf` is the function called to output from a C program. So, to print a particular message you enclose it in double quotes. So, whatever is

enclosed in the double codes, will be printed.

So, to repeat again, please note the extra punctuation symbols, which tell you, that these are valid C statements. So, all the underline statements are, all the underline symbols are important. So, in the line printf Welcome to C, this is what is known as a statement in C and statements in C end in a semicolon. So, this semicolon is also important because it tells you, that this is where the statements ends, what typical errors do we have when we code in C.

(Refer Slide Time: 11:34)

The slide has a red header bar with the word 'Errors' in white. Below the header is a bulleted list of five common errors:

- Let us systematically enumerate a few common errors.
- 1. Forgetting to include stdio.h.
- 2. Forgetting main function
- 3. Forgetting semicolon
- 4. Forgetting open or close brace ({ or }).
- 5. Forgetting to close the double quote.

Below the list is a note: "Try deliberately making these mistakes in your code. Save them and try to compile. Study the error messages for each."

At the bottom of the slide is a footer note: "Familiarity with error messages will help you find coding errors later."

Let us systematically enumerate a few common errors that could happen in even a simple program like what we have seen. For example, you could forget to include `<stdio.h>`. If you do not include the standard input library, then the compiler will give you an error message. You may forget to include the main function, then also you will get some error message. You could forget to include the semicolon in the statement, you could forget to include the braces, the curly braces in main or forget to close the double quote, open or close the double quote in the printf statement. So, these are a few errors that you could make even in a simple code like what we have seen.

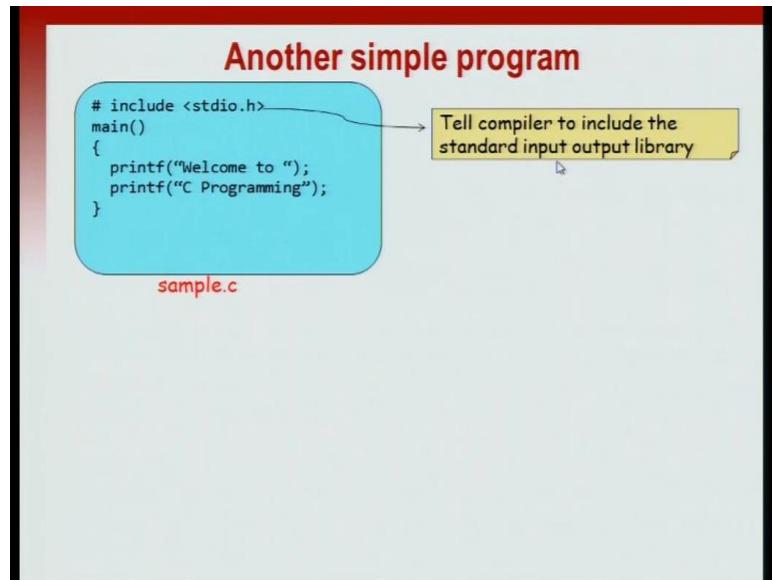
We have only three lines, but they could also have errors. I would advise you to try

deliberately making these mistakes in your code, try compiling them and study the error messages. Once you are familiar with error messages, this will help you later in your coding, because when you see the error messages you can guess what errors did you possibly make in your code. So, go back to the code and correct it.

Introduction to Programming in C

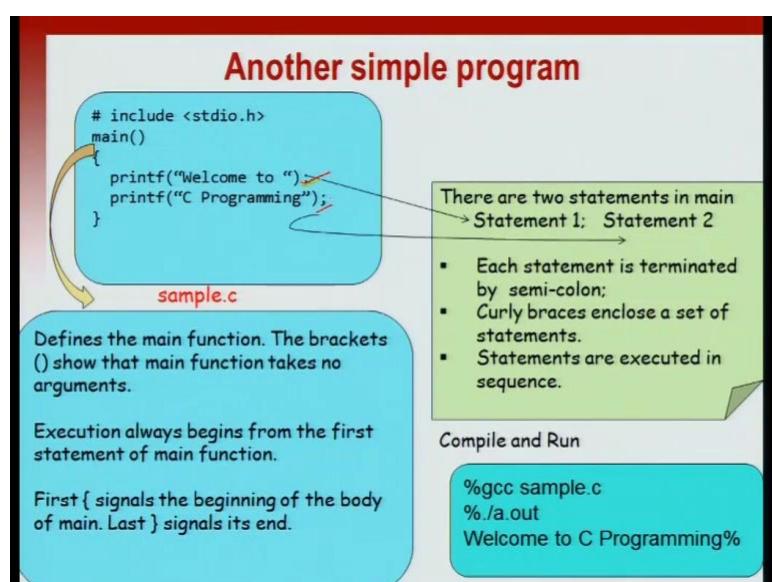
Department of Computer Science and Engineering

(Refer Slide Time: 00:09)



In this session let see another simple program, and try to study what is going on in there. So, here I have slightly more complicated program, then what we just saw. This has two printf statements; once is welcome to, and the second says C programming. So, it is slightly more sophisticated than the code that we have seen. So, to recap the first line **# include <stdio.h>**, tells the compiler to include the standard output library.

(Refer Slide Time: 00:42)



Then we have the main function; the open and close brackets immediately after name show that main is a function, execution always begins at the first line of the main statement. Then the body of the function the logic of the function is enclosed within two curly braces; the first curly brace signals - beginning of the function, and the last curly brace says that the function is over here.

This particular name function has two statements; earlier we have just one statement. The each statement as I said before is terminated with a semicolon. So, this is the first semicolon, and this is the second semicolon. The first semicolon says that the statement printf welcome to ends that point, and then followed by the second statement. And the second statement is also terminated by semicolon. Every statement in C is terminated with the semicolon. Curly braces enclose a set of statement, and each statement in a sequence is executed in the exact sequence that we wrote in the code. Now, once we edit this in an editor save the file, now compile, and run the file. So, let us call it **sample.c** or you may rename it any file you want. And then once the compilation is successful, you can run it using **./a.out**, and then it will print welcome to C programming, which was essentially the same messages as we seen before.

(Refer Slide Time: 02:36)

Tracing the Execution

<p>Line No.</p> <pre>1 # include <stdio.h> 2 main() 3 { 4 printf("Welcome to "); 5 printf("C Programming"); 6 }</pre> <p>Output: After lines 5,6 Welcome to C Programming%</p>	<ul style="list-style-type: none">■ Program counter starts at the first executable statement of main.■ Line numbers of C program are given for clarity.■ Let us run the program, one step at a time.■ Program terminates gracefully when main "returns".
---	---

Let us try's what happens when we execute the program. By tracing we mean step by step looking at each statement, and C's see what happens when the program executes. We have what is known as a program counter, which says here is the currently executing

line of program. The program counter starts executing at the first statement of the main, for reference I have given line numbers in the code. Now this is given just for clarity. Now let us just see, what happens when we run the program line by line.

So first we execute the first line of the code, after we are done executing the line 4. So, after we are done executing lines 3 and 4, the message welcome to will be printed on the terminal. This will be followed by the next line, so the next lines is C programming. So, after the next line executes, it will print C programming %. I have given this in two different colors to highlight that one was printed by the first line, and otherwise printed in second line, otherwise the colors have no meaning, no special meaning. The program terminates when the main finishes execution, and this is what is typically known as returning from the function, we will see this terminology later in the course.

(Refer Slide Time: 04:19)

Program Comments

```
# include <stdio.h>
/* a simple C program */
main()
{
    printf("Welcome to ");
    printf("C Programming");
    /* first print */
    /* second print */
}
```

These are called COMMENTS.

- Any text between successive /* and */ is a comment and will be ignored by the compiler.
- Comments are NOT part of the program.
- They are written for us to understand or explain the program better.
- Comments can be short or long. Any number of comments may be included.
- It is a very good idea to comment your programs. For larger programs, industry, this is a must. Will help you and other developers understand and maintain programs.

Now, when you code in addition to the statements which are actually executed, you may also give a few additional remarks; these are what are known as program comments. For example, the lines a simple C program first print and second print; these are the comments in the code. So, whatever is highlighted in red in the code is what are known as comments. Any text between forward /*, and then later followed by a */. So, any text between successive /* and */ is a comment, and it will be ignored by the compiler. So, as far as the compiler is concerned a code with comments is the same as a code without comments. It does not effect the logic of the code. So, comments are not

part of the program; however, it is highly recommend that any program you write, you should comment the code. This is show that other people can understand your code also you yourself looking at the code 4 months later or five months later, it is it may be difficult to understand what you wrote? Much before and comments help you understand the logical of the program.

Now, it is a very good idea to comment your programs, and for lager program it is a must to comment the programs. This is standard industry practice, and even if you participating in large programming project like free software projects, comments are highly encouraged, because it will understand other developers, other programmers to understand your code. So, we will try to follow our own advice most of the programs that we will see in this code, we will comment it, so that it easy to follow the logic of the code.

(Refer Slide Time: 06:15)

Notes*

- Just as main() is a function, printf("...") is also a function. printf is a library function from the standard input output library, which is why we inserted the statement

```
# include <stdio.h>
```
- printf takes as arguments a sequence of characters in double quotes, like "Welcome to". A sequence of characters in double quotes is called a *string constant*.
- We "call" functions that we define or from the libraries.

Now, a few notes just as a main is a function printf is also a function. Printf is a library function which means that it is given by the C programming language, and we wanted to tell the compiler to include this library function. The statement which set that is this `# include <stdio.h>`. So, `# include <stdio.h>` is the line telling that I want the standard input output library, because that is the library from which I will get the function printf. Now what does printf do? Printf takes two arguments, just like arguments to mathematical

function. So, it takes an input argument which in our first case was welcome to. So, this was the printf first printf statement in the program that we just soft.

Now, this was enclosed in double codes, right. So, it was enclosed in open double quote, and then ended with a close double quote. A sequence of characters in double quotes it is what is known as a string constant. Now we can call the functions that we define or we can call the functions that the library provide. This is how once you define a function, you can call a function. Now we will see an additional concept, the printf statements that we have seen so far, we will print a message and it will print there. And then the prompt the terminal prompt will come immediately after the print. Now typically what we want to do this will print a message, then say tell the prompt to appear on the next line.

(Refer Slide Time: 08:14)

Printing in different lines

The newline character

- All letters, digits, comma, underscore are called characters. There are 256 characters in C.
`a' ... `z' `A' .. `Z' `0' .. `9' `@' `.' `/` `!` `?` `~`
`^` `&` etc..
- There is a special character called newline. In C it is denoted as '\n'
- When used in printf, it causes the current output line to end and printing will start at the next line.

So, for this we need what is known as the new line character. All letters digits, gamma, punctuation symbol; these are called characters in the C programming language. There are the total of 256 characters in C, 256 is 2 to the 8. So, for example this small letters a to z, capital letters A to Z, 0 to 9, the at symbol, other punctuation symbols like dot, gamma, exclamation mark, and so on; are all characters in C. Now in addition to this there are certain things call special characters. So, there is a special character call new line, in the C programming language it is denoted as \n. So, there are notice that there are two kinds slashes; /, and \ on your keyboard. And the new line character is denoted as \n.

So, even though it is single character, it is denoted by two letters. When used in printf it causes the current output line to end, and then printing will start from the new line. So, it is something which says the current line has enter, now whatever you have to print, print it in the next line.

(Refer Slide Time: 09:34)

The newline character

- Newline character '\n' is like any other letter and can be used multiple times in a line
- "... \nC..." is treated as "...'\n' followed by 'C'.

```
#include <stdio.h>
main()
{
    printf("Welcome to \n");
    printf("C programming\n");
}
```

When we compile and execute,

```
$ ./a.out
Welcome to
C programming
$
```

The new line character `\n` is like any other letter, and can be used multiple times in any particular line. For example, if you have something to print followed by `\n`, followed by C, followed by something to print. Now this will be treated as, so many characters and then a new line followed by C. So, let see a particular example, if you have the old program that we just wrote, but we end each message with a `\n`. So, we have `printf welcome to \n`, `printf C programming \n`. When we compile and execute, we will see something new. So, when we run this `./a.out`, it will print `welcome to`, and then the next thing to print is a `\n` which is a new line. So, printing will start from the next line, and then it will print the next message with `C programming`. So, it will print that followed by new line. So, the prompt will appear on the line after words. So, new line character is something that is used to make your output messages a little more ((Refer Time: 10:58)) here.

(Refer Slide Time: 11:00)

Last on newlines

■ To repeat, newline character '\n' is like any other character. It can be used multiple times. Another example.

```
#include <stdio.h>
main()
{
    printf("Welcome to\n\nC\n");
}
```

■ When we compile and execute, we have the following.

```
$ ./a.out
Welcome to

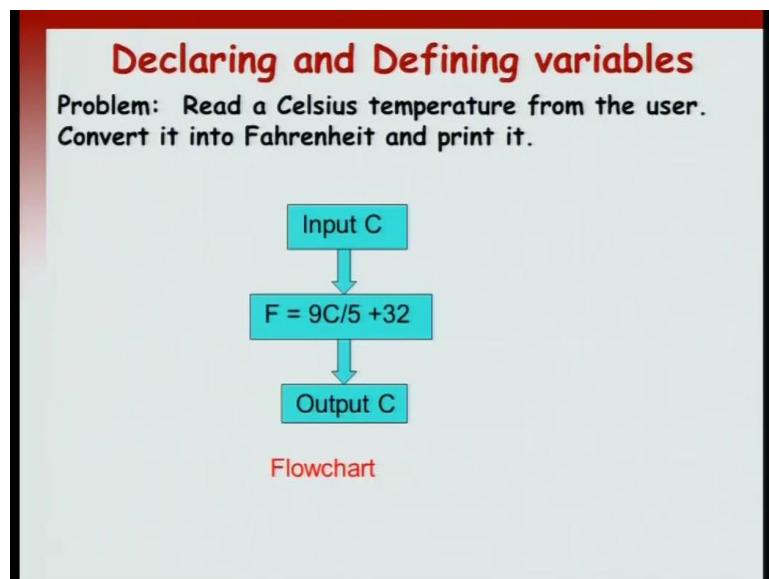
C
$
```

So, let us just conclude by saying one more thing about new lines, the new line character \n is like any other character, and you can use it multiple times even within single message. For example, if I do the same program, but let us say I have welcome to \n \n C \n. So, I have repeated occurrences of \n in the same message, what it will do is if will print the message welcome to, then it will print a new line, and then it will print another line, and then it will print C followed by new line. So, when you run program you will have welcome to new line, then the blank line, then C, then another line. So, new lines are just like any other characters, the difference is that, because it is a special character, it is represented by two letter. So, it is not represented by single letter it is represented by \n. So, they are together one character in C, call the new line character.

Introduction to Programming in C

Department of Computer Science and Engineering

(Refer Slide Time: 00:13)



In this session we will see slightly more sophisticated programs. Recall that in our discussion about flowcharts, we talked about variables, which were conceptually seen as little boxes in which you can hold values. So, let us see how to write simple C programs in which we make use of variables. So, we will illustrate with the help of a sample program.

So, we have this following program, which is very simple, read a Celsius temperature and convert it into the equivalent Fahrenheit temperature. This is something that all of you must know. So, the flowchart is very simple, you have an input C, which is the current Celsius that you want to convert. Then you apply the formula F, which is **9C/5 + 32**. In this session, we well see how to write simple C programs, which makes use of variables.

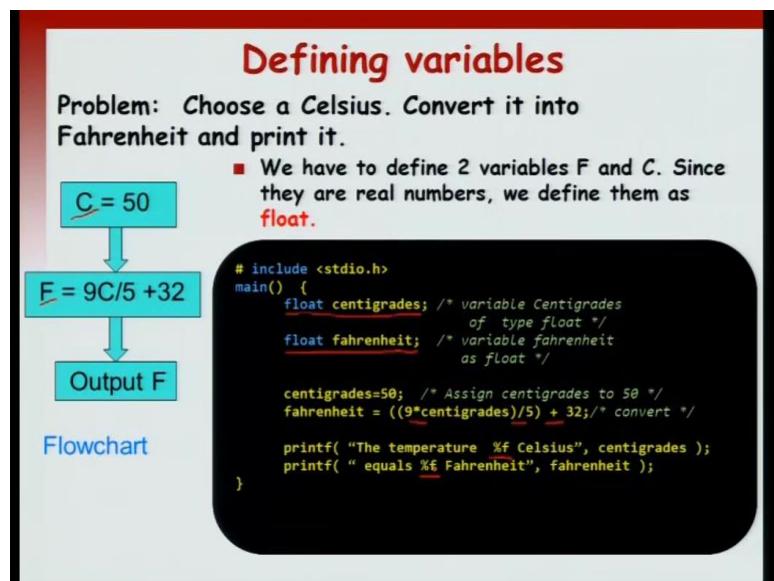
Recall that in our discussion about flowcharts we talked about variables, which were conceptually seen as little boxes in which you can hold values. So, let us illustrate a simple C program making use of variables with the help of a program. So, we have a small problem, which is convert a Celsius temperature into the equivalent Fahrenheit temperature. This is the formula that all of you must know. So, let us write a C program for it.

So, we will draw the simple flowchart for doing the program. You input the temperature in C, in Celsius, convert it into Fahrenheit according to the formula $9C/5 + 32$. Once you have done that, the variable F holds the Fahrenheit value, so you output the F. So, here is the simple flowchart that we want to implement.

(Refer Slide Time: 02:03)

Defining variables

Problem: Choose a Celsius. Convert it into Fahrenheit and print it.



```

graph TD
    A[C = 50] --> B["F = 9C/5 + 32"]
    B --> C[Output F]
  
```

Flowchart

- We have to define 2 variables F and C. Since they are real numbers, we define them as float.

```

#include <stdio.h>
main()
{
    float centigrades; /* variable Centigrades
                           of type float */
    float fahrenheit; /* variable fahrenheit
                        as float */

    centigrades=50; /* Assign centigrades to 50 */
    fahrenheit = ((9*centigrades)/5) + 32; /* convert */

    printf("The temperature %f Celsius", centigrades );
    printf(" equals %f Fahrenheit", fahrenheit );
}
  
```

Now, how do we write the equivalent C code. So, we, in the flowchart we have seen, that we have two variables, C and F. These are the variables that we would want to implement in the C code. So, let us see how to do it. So, we write the following C program in which now we have two more components, one is the variable declaration, float centigrade and then the second is another variable, float Fahrenheit. So, centigrade corresponds to C in the flowchart and Fahrenheit corresponds to F in the flowchart. So, I write the following code, which is supposed to implement the flowchart on the left.

So, let us say, that the input is 50 degree Celsius and then the Fahrenheit, the formula is exactly the same as what we have in the flowchart. We have $9C/5 + 32$. Notice here, that these are arithmetic operators. So, the * arithmetic operator stands for multiplication, / stands for division and + stands for addition. So, this is exactly as in the flowchart except, that here in the flowcharts multiplication symbol is being swallowed, but in C you have to specify it using a * operator. So, Fahrenheit equal to $9C/5 + 32$ is exactly

similar to the analogous line in the flowchart.

And finally, for outputting we will use printf statement. So, here is something new in the printf statement. We use what are known us format specifiers, this %f symbols are new and we will describe them shortly.

(Refer Slide Time: 04:05)

Variable definitions

```
# include <stdio.h>
main() {
    float centigrades; /* variable centigrades of type float */
    float fahrenheit; /* variable fahrenheit as float */

    centigrades=50; /* Assign centigrades to 50 */
    fahrenheit = ((9*centigrades)/5) + 32; /* convert */

    printf( "The temperature %f Celsius", centigrades );
    printf( " equals %f Fahrenheit", fahrenheit );
}
```

The statement float centigrades; defines a variable centigrades. It creates a box capable of storing a real number* and names the box centigrades. Similarly, float fahrenheit; defines a variable fahrenheit to store another real and creates a box called fahrenheit.

So, let us look at the program in little more detail. So, we have two statements, which are of interest, in the beginning of the code, which are what are known as the definition of the two variables. Recall from our discussion on flowcharts that variables are boxes and each box has a name associated with it. So, you have two concepts associated with variable as far as flowcharts were concerned, one was the box and the second was the name of the box. Now, when we come to C, we will associate a third concept with variable, which is the type of the box.

So, if you look at the first statement it says, float centigrade;. Now, this defines a variable centigrades. It creates a box capable of storing a real number and names the box the centigrades. So, the box is of type float. Type float means, that box can hold real number. Similarly, fahrenheit is also a box, which can hold real number. So, you declare that the type of that variable is float. So, these are supposed to be the first two lines of the code.

(Refer Slide Time: 05:29)

The screenshot shows a code editor with a dark theme. A red callout box highlights a specific line of code: `centigrades=50;`. Below the code editor, there is a list of four bullet points explaining the code:

- `centigrades =50`; assigns to variable centigrades the value 50.
- `fahrenheit = ((9*centigrades)/5) + 32` is an arithmetic expression.
- `*` is the multiplication operator, `/` is division, `+`,`-` are usual.
- Brackets (and) in expressions has its usual meaning.

Now, `centigrades = 50` that is the line, which assigns the value 50 to the variable centigrades. So, once you execute the code, the box associated with name centigrade will hold the value 50 followed by the line, which computes the value of fahrenheit. So, fahrenheit equal to $9C/5 + 32$. It is an, it associates an arithmetic expression. So, it evaluates an arithmetic expression, takes its value and stores it in the box associated with the fahrenheit. And as we just saw before, `*` is the multiplication operator, `/` is the division operator and `+` is the additional operator.

Now, the brackets in an arithmetic expression are just like brackets in mathematics. So, they group together a particular thing. Now, let us just try to the program. Let us see what happens step by step when we run the program.

(Refer Slide Time: 06:48)

The slide shows a C program named sample2.c. The code includes declarations for float variables centigrades and fahrenheit, initializes centigrades to 50, calculates fahrenheit as (9*centigrades)/5 + 32, and prints the results. A callout box explains that microprocessors use finite precision for real numbers, represented by a limited number of digits after the decimal point, with an example of 12.3456789 being stored as 1.23456789E+1. It also notes the use of scientific notation. Below the code, a terminal window shows the execution of gcc sample2.c and the resulting output: "The temperature 50.000000 Celsius equals 122.000000 Fahrenheit". A callout box points to the "%f" format specifier in the printf statements, explaining that it specifies printing a real number in decimal notation.

```
sample2.c
#include <stdio.h>
main() {
    float centigrades;
    float fahrenheit;
    centigrades=50;
    fahrenheit= ((9*centigrades)/5)+32;
    printf("The temperature ");
    printf("%f", centigrades);
    printf("Celsius equals");
    printf("%f", fahrenheit);
    printf("Fahrenheit");
}

%gcc sample2.c
./a.out
The temperature 50.000000 Celsius equals 122.000000 Fahrenheit
```

• Microprocessors represent real numbers using **finite precision**, i.e., using *limited number of digits after decimal point*.
• Typically uses scientific notation:
12.3456789 represented as 1.23456789E+1. Bit more later.

centigrade fahrenheit
50.000 122.0000

"%f" signifies that the corresponding variable is to be printed as a real number in decimal notation.

Let us say that we save the file as `sample2.c` and then run it as `./a.out`. So, first we will have two boxes created, one for centigrade and one for fahrenheit. These can store float numbers. Now, what are float numbers? Basically, they are real numbers, which are saved by the microprocessor. Now, the microprocessor can store variable real number only using finite precision. So, this is different from actual real numbers that we encounter in mathematics. So, we have only a limited number of digits after the decimal point, but other than that you can think of them as real numbers. We will see floating point number later in the course in greater detail, right. For now, think of them as the machine representation of a real number.

So, once you finish the declaration statements what you have are two boxes one first centigrade one for Fahrenheit and because you declare the types to be float it is understood that those boxes will hold real number. So, let us execute the first executable assignment here. `centigrades = 50` and you will see that the box contains 50.000 something. Even though we specified it as an integer, it will convert it into real number, floating pointing number and store it. Then this is followed by the calculation of the Fahrenheit value, and let us say that you compute 9 times 50 divided by 5 plus 32, it comes out as 122. After that line is executed, the box associated with Fahrenheit will contain 122.

Then, the next line says, print, printf the temperature, there is no new line, so the next printf will start from where this printf ended and here you see something new, which is the **%f** symbol. So, these are what are known as format specifiers. So, the **%f** symbol say, that take the corresponding variable, which is given centigrades here. Now, print it as a float, print it as a real number.

So, notice the difference between first printf and the second printf. The first printf just had a string between “, the second printf has two arguments, one is a string between “ and the string is **%f** and then the second argument is centigrades. So, it says take value of the centigrade and print it as a floating point number. So, it does that and you see 50.000 in the output. There is no new line. So, the next printf starts from the previous line where the previous printf left off, 50 Celsius equals, it prints that.

And now, you have another format specifier. It says printf **%f** fahrenheit. fahrenheit is 122 and it will print it as floating point number or as real number. So, it will print it as 122.000 printf fahrenheit. So, the final message, that will be printed will be the temperature, 50 centigrade, 50 Celsius equals 122 Fahrenheit.

So, the new thing we have seen in the program include variable definitions, how they have an associated type and similarly, how do we print these variables. So, we do not want to print the names of the variables, we want to print the, we want to print the content of the variable. We want to print what is stored in the box. For that we use the format specifiers like these **%f**.

(Refer Slide Time: 10:59)

Introduction to types in C

■ Variables are the names of boxes to store values in. We can give them any names we like. Almost!

■ But! Are all boxes the same?... No! Not in C (or C++/Java).

■ Types: Variables (boxes) are defined with a type. Some basic types.

int a;	defines a variable a that can hold an integer
float b;	defines a variable b that can hold a real value (single precision)

Computers cannot store arbitrarily large integers. The type integer can store all numbers between $-M+1, -M+2, \dots, 0, 1, \dots, M$ where, M is a constant depending on the machine and compiler. Usually $2M = 2^{32}$

So, let us briefly introduce what are types in C. So, variables are the names of the boxes in which to store values, but these boxes are special. Certain boxes can hold only certain kinds of values, so all boxes are not the same. There are different kinds of boxes. Now, types are basically saying, that a particular box can hold a particular kind of data. So, variables are defined with an associated type and we will use some basic types during the course of this program language tutorial.

One of the two common type, two of the common types that we see in this program are int, which stands for an integer and float, which stands for a floating point number, which stands for real number. Notice, that machine can hold only a fix number of bits. So, that does not mean, that the integer can go from minus infinite to infinite. It goes from a certain vary small negative number to a very large positive number. Similarly, floating point also is limited by a particular range. This is because machines cannot represent arbitrary values. The type of integer can store all numbers from $-M+1, \dots, 0, 1, \dots, M$. So, there will be some large M, for which defines the upper limits and the lower limits of the particular machine. Now, that limit may depend on which particular machine that you use. On a, on a 32-bit machine it will be 2^{32} , M will be 2^{32} .

(Refer Slide Time: 13:03)

Always define a variable before use.

Names of variables:

Names are made of letters and digits; the first character must be a letter. The underscore character `_` counts as a letter. So csquare is a valid name.
Other valid names: "c_sq", "csq1", "c_sq_1", "C_sq_1" etc..
Names are sensitive to Upper/Lower Case.

Assignment operation is not the same as mathematical '='.

For example, suppose b=3, and a=2.
Then a=b will set a to 3.
b is unchanged.

Assignment copies the value in the right to the variable on the left.

A few final words about variables. Just like in a cooking recipe, you well never mention a step, which involves ingredient without mentioning, that ingredient is needed in the first place. So, you will never say, that use salt and if you look at the list of ingredients, you were, you will see, that there is no salt in the list of ingredients. Such recipes are considered bad. So, when you write a typical recipe, you list out all the ingredients first and then write the steps for the cooking. Similarly, in a program you define whatever variables that you need before those variables are used by any statement in the program. Always define a variable before use.

Now, a word about names of variables in the C programming language. The names are consisting of numbers, letters and an underline symbol, an underscore symbol. And there is a particular convention that a variable cannot start with a number. So, the initial letter has to be a letter or an _, it cannot be number, but further can be either capital letter, small letter or numbers or an _. So, there are valid names like **c_sq**, **csq1**, **c_sq_1**. So, all these are valid. One thing to note is, that the names are sensitive to upper and lower case. So, for example, capital C Centigrade is different from a centigrade, which starts with a small c. So, these are two distinct variables that is the common source of errors when we start programming.

Another thing to note or to watch out for is, that the assignment operation, which is equal to is not the same as mathematical equal. So, when mathematically we say a equal to b, it means, that a and b are the same quantity. So, a equal to b is the same as saying b equal to a. This is not true in C. For example, let us say, that you have the statements **b = 3**; and then later you have **a = 2**; and further you have the statement **a = b**; So, the statement a equal to b will set a to b's value. So, b's value is 3 and that value will be copied to a. So, it will set a to 3 and b will be unchanged. So, watch out for this.

If, if you were expecting the mathematical operator, after the operation **a = b**, a and b we will have the same value, but that is not the case. The meaning of the symbol equal to is, that take the value on the right hand side of the expression and copy that into the box specified by the left side. So, copy the value in the right hand side to the variable on the left.

Introduction to Programming in C

Department of Computer Science and Engineering

We have seen comparison operators, like less than, equal to, less than or equal to and so, on. We will see bunch of few more operators in this session. So, let us consider the modulo operator which we have already seen in when we discussed utility in GCD.

(Refer Slide Time: 00:27)

The % operator over integers

- For integers a and b , the operator % is the remainder operator. $a \% b$ gives the integer remainder when a is divided by b.
- Example: test if a is divisible by 6.

```
#include <stdio.h>
main () {
    int a;
    scanf( "%d", &a );
    if ( (a % 6) ==0 ) /* test if a is divisible by 6 */
    {
        printf( "Input %d is divisible by 6", a );
    } else {
        printf( "Input %d is not divisible by 6", a );
    }
}
```

So, $a \% b$ gives the remainder when a is divided by b. So, suppose we have the following problem, we get a number a and we want to check whether the given number is divisible by 6. If it is divisible by 6 $a \% 6$ will be 0 the reminder will be 00. So, we will write a simple code, you have int a, a is of type int. Then, scan the number using `scanf("%d", &a)`. And then, you test whether a is divisible by 6 to test whether a is divisible by 6 you check whether a mod 6 is 0. If it is divisible, you say that input is divisible by 6 `%d` a. Otherwise, else, `printf` the input is not divisible by 6 very simple operation.

(Refer Slide Time: 01:29)

Check divisibility by 6 and 4

- Example: Read a. Determine if a is divisible by 6 and 4. (No number theory now).
- A simple solution using nested-if statements (nested means an if within an if).

```
# include <stdio.h>
main () {
    int a;
    scanf("%d", &a); /* define and read a*/

    if ((a%6) == 0) { /* a is divisible by 6 */
        if ((a%4) == 0) { /* a is also divisible by 4 */
            printf("%d is divisible by 6 and 4\n", a);
        }
    }
}
```

Now, let us make it slightly more elaborate. Suppose, you have to test whether, this a slight variant. Suppose, you have to test whether a give number is divisible by 6 and by 4 two numbers. How do you do this? So, you scanf the number and you test whether a is divisible by 6. So, **a % 6** is 0. If that is true, then you also check whether **a % 4** is 0. If both are true, then you print that the given number is divisible by 6 and 4. So, percentage is divisible by 6 and 4 a.

So, you can argue about this program and see that, if it is divisible by 6, but not by 4 then, it will enter the first if, but not enter the second if. Therefore, it will not print that it is divisible by 6 and 4. Similarly, if it is not even divisible by 6 it will not even enter the first if condition. So, you will in any case not print that it is divisible. So, convince yourself that this particular code will print a number is divisible by 6 and 4 if and only if the given number is divisible by both 6 and 4.

Now, that piece of code was slightly long is there any way to write the same code with a fewer number of lines. And for this c provides what are known as logical operators. Now, there are three logical operators in Boolean logic which are Boolean AND, Boolean OR and Boolean NOT. So, there are three logical operations AND, OR and NOT and C provides all of them. So, the same if condition that we wrote before, we could have easily said if it is divisible by 6 and if it is divisible by 4 then print the output.

(Refer Slide Time: 03:44)

AND in C: The Logical Operator &&

- Another Solution: Use && to logically combine two expressions into a single expression.

```
if ( ((a%6) == 0) && ((a%4) == 0) ) {  
    /* (a is divisible by 6) AND  
       (a is divisible by 4) */  
  
    printf ( "%d is divisible by 6 and 4\n", a );  
}
```

- && (consecutive ampersands, no blanks) is the C operator corresponding to the mathematical AND function (on booleans 0/1).
- It takes two values as input and returns a zero if any of the values is 0. Otherwise, it returns the second input value.

So, for this C provides an operator which is the Boolean AND operation. So, the Boolean AND operation in C is given by two ANDs. So, by now you should be familiar with the fact that certain operations in C have repeated characters. For example, we already have seen the equality operations which was equal, equal. Similarly, the Boolean operation and it is actually the and symbol on the keyboard. But, you have to have two of them that represents the logical AND.

So, this expression says if **a % 6** is 0. So, this expression is what test for a is a multiple of 6. And this is the expression which test whether a is the multiple of 4. So, if both conditions are true, then you say that the given number is divisible by 6 and by 4. So, consecutive ampersand signs, that is the and symbols without any blanks in between is the C operator corresponding to the mathematical and the logical AND function.

So, it takes two values as input and returns a 0. If any of the values is 0, if both values are 1 then it returns a 1. So, this is the same as the logical AND. If either of them is 0 then the result is 0, if both of them are 1, then the result is 1.

(Refer Slide Time: 05:39)

Logical Operator for AND: &&

The table defines $a \&\& b$ where a and b are int/float or are expressions of type int/float.

a	b	a && b	Comments
Non-zero	Non-zero	1	
0	Any value	0	b is not evaluated.

Recall that in C, 0 is FALSE and non-zero is TRUE.
So $a \&\& b$ is the logical operation a AND b



Every expression has a type. $a \&\& b$ is of type **int** irrespective of the types of a or b . But it takes values 0 or 1. You can also print it as an int. Try it out on the computer.

```
printf("%d", a && b);
```

So, the truth table for the operation AND is as follows if a is a non-zero value and b is a non-zero value, then C considers that both are true. So, the output value is of a and b is 1. If a is 0 and b is any value at all the output is 0 and b is not evaluated. So, this is the same as logical end. The only think to notice that, if in evaluating a and b you already know that a is 0, then you know the result is 0. So, C will not bother to evaluate b . Because, it knows that the result is already 0.

Every expression has a type a and b is of type int regardless of the types of a and b . This is because a and b is a logical assertion. The type of a logical assertion is that, it is either true or false, it that it corresponds to a Boolean value. Therefore, at the type of an a and then b regardless of what a and b are the result is always 0 or 1. So, it is of type int. Now, you can print the result as an int, you can say `printf %d` a and then b .

(Refer Slide Time: 07:04)

Logical Operator for OR: ||

■ Let a, b denote two expressions in C. Then

$a \parallel b$ evaluates to non-zero (TRUE) if either a is non-zero (TRUE) OR b is non-zero (TRUE). Otherwise, $a \parallel b$ evaluates to 0 when both a and b are 0.

a	b	$a \parallel b$	Comments
zero	zero	0	
Non-zero	any value	1	b not evaluated

$a \parallel b$ is the logical operation $a \vee b$

 a $\parallel b$ is an expression of type int. Takes values 0 or 1. You can print it, use it anywhere as an int

Now, there are three logical operations as I mention. So, there is also OR in C it is denoted by two vertical bars which are there on your keyboard. So, a or b which is $a \parallel b$ evaluates to non-zero if either a is non-zero or b is non-zero. If both of them are zero, then the result is zero. So, this the meaning of a logical OR operation, if both of them are false, then a or b is false. If at least one of them is true then a or b is true.

So, you can write the truth table for that. If a and b are 0 then the output is 0, if a is non zero and b is any value. Then, in already know that the output of a or b is 1. So, the output is 1 and b will not be evaluated. This is similar to in the case of AND. If a was 0 and b was any other value, then you know that the output of and is 0. Therefore, b will not be evaluated. Similarly, if here if a is non-zero, then you know the value is 1. So, b will not evaluated and as before a or b is of type int.

(Refer Slide Time: 08:33)

Not in C !

- (Last) logical operation: NOT. Denoted as the unary operator !
- ! is logical complement.
- Unary means takes one argument. Binary operator takes two arguments, e.g., &&, ||, +, -, *.
- If a is an expression then !a is an expression (of type int).

a	!a
0	1
Non-zero	0

- Is a not divisible by 3? Expression in C:

```
if (!(a%3) == 0) {
    printf("Not divisible by 3");
}
```

So, the third logical operation is NOT. Now, NOT in C is denoted as the exclamation mark. So, let us see an example of that. So, NOT is the logical complement and it takes only one argument, this is different from the previous two that we have seen a or b and a and b both took two arguments it is not takes only one arguments. So, it is called a unary operators. So, NOT of a is an expression of type integer and the value is the negation of a. So, if a is 0 NOT of a will be 1 and if a is non zero then NOT of a will be 0.

So, for example, if I want to say that a is not divisible by 3 I will just write NOT of a mod 3 equal to 0. You know that a mod 3 equal to 0 test for a being divisible by 3. So, negation of that it will say that the given number is not divisible by 3.

(Refer Slide Time: 09:43)

The logic of leap years

- A normal year has 365 days. Leap Year has 366 days (extra day is the 29th of February).
- Why? 1 year = 365.242375 rotations of earth.



365.25
Every 4 years
you lose ~1 day.
1 day every 4 years.

In 100 years you would add 25 days.

Every 100 years, skip adding the extra day

Every 400 years, add 1 day..

Let us finish this by slightly complicated example which is that of leap years. So, I am given a particular year number and I want say whether the given number is a corresponds to leap year or not. Now, what is a leap year it is that you add a few years will have February 29th in February all other years will have 28 days in February. So, what is the logic of a leap year. So, roughly an average solar year is 365.242375 rotations. So, in particular is not an integer.

So, we normally say that year has 365 days that is not quite true, this it is a rounding. So, how much are we losing. So, you can calculate it as follows. The remaining number after the decimal point is a roughly 0.25. So, at a rough cut let say that every 4 years. Because of this 0.25 you will lose one day. So, every year you are losing about a quarter of a day. So, if every 4 years you will add a day. Now, when you do that you go back to... So, just a minute.

So, you have 365 point let us say 25. So, every 4 years you would lose about a day approximately 1 day. So, in order to compensate for that you add 1 day every 4 years. So, in 100 years you would have added 25 days. But, that is 1 day too much. Because, remember that this number is only 365.24 something. So, in 100 days you should have added only 24 days. But, now you added a 25 days. So, to compensate for that every 100 year skip adding the extra day.

So, every 4 years you have add 1 extra day,, but every 100 years every 100th year you do not add that extra day, you skip it. Because, you would have added 1 more day then you. And then again you can look at what remains, what remains is roughly .24 which means that every 400 years if you do this adjustment you are losing about a day. Because, every 100 years you are losing about quarter of a day from this 0.2375 part. So, you do the same logic again every 400 years. So, every 400 years add an extra day. So, this is the logic of the leap year that we all know. So, how do you decide whether year will be leap year.

(Refer Slide Time: 14:01)

The slide has a title 'The logic of leap years' at the top. It contains several bullet points:

- A normal year has 365 days. Leap Year has 366 days (extra day is the 29th of February).
- Why? 1 year = 365.242375 rotations of earth.
- How to decide whether a year will be a Leap year:
 1. if 4 evenly divides Year
 2. if 100 evenly divides Year
 3. If 400 evenly divides Year
- Every 4th year is a leap yr, but not every 100th year unless it is a 400th year.

Below the bullet points is a C code snippet:

```
int year;
if ( (year %4) == 0)
    && ( ( !(year % 100) == 0) ||
        ( (year % 400) == 0) )
    { printf ("Year %d is a leap year", year); }
```

So, the logic that I have outlined just now says that, if a year is a multiple of 4 then it is a leap year. But, if a year is a multiple of 100 as well then it is not a leap year. But, if it is a multiple 400 than it is a leap year. So, here is a pretty complicated expression. So, every 4th year is a leap year. But, skip every 100th year unless it is also a 400th year. And you can write this expression in C, it is slightly complex has you can imagine.

So, if the first line the first expression says that, wise year is a multiple of 4. So, if year is divisible by 4 also the following should be true, it should not be a multiple of 100 unless it is a multiple of 400. So, it should not be divisible by 100 that should be true or it should be true that, it should be a multiple of 400. For example, if you have 400 then it is a leap year. So, what will happen is that year modulo 4 400 modulo 4 is 0.

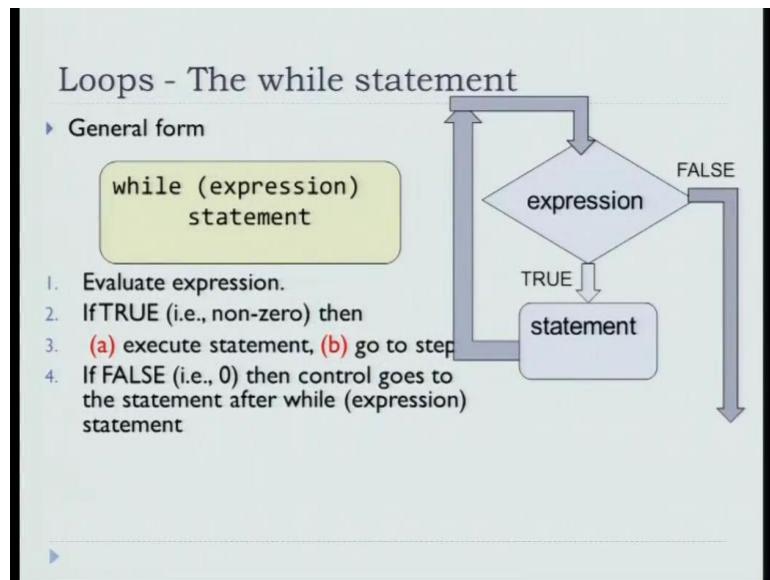
Then, what happens is that you have 400 modulo 100 that is 0. So, this is equal to 0 that is 1 NOT of 1 is 0. So, this part is entirely 0, but it is divisible by 400, 400 divided by 400 is 0. Therefore, this part is true, this or 0 or 1 is true. Therefore, the whole expression becomes 1 and 1. So, it is true. So, this logical expression slightly complicated logical expression encodes the logic for saying that the given year is a leap year. So, try this out yourself this is a slightly tricky expression. And convince yourself that, this exactly encodes the logic of the leap year.

Introduction to Programming in C

Department of Computer Science and Engineering

In this session, we will look at loops in the C Programming language. And we will start with very basic kind of loop which is known as the while statement.

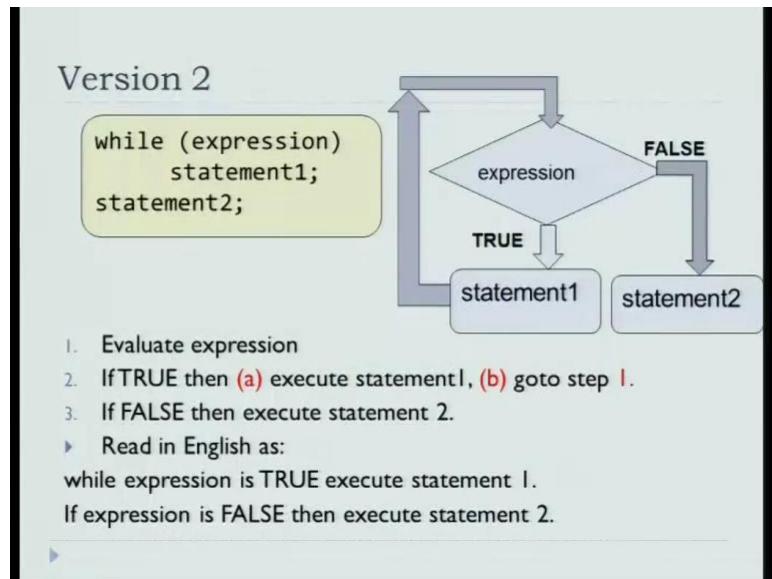
(Refer Slide Time: 00:04)



So, the general form of a while statement this similar to that of and if which is that there is an expression and you say while that expression, then do this statement. So, the flow chart corresponding to the while expression will be, you test whether the expression is true or false. If it is true you do this statement, if it is false you exit out of the loop and execute the next statement outside the loop.

So, if the expression is true in C that is the expression is non-zero, then execute the statement and go to the step outside the loop. If it is false then directly go outside the next statement after the while loop. This is similar if you recall to the, if block without the else. So, loops are a new thing that explicitly there was no loop construct in a flow chart, we just had this way of going back to the expression. But, in programming languages loops are such a basic programming need that in addition to the if block, you have loop construct as well.

(Refer Slide Time: 01:39)



So, slightly different variant of the while expression will be that while expression statement 1 and then statement 2. So, the flow chart here is easy to follow if the... So, if first test whether the expression is true. If the expression is true then you execute statement 1. And then after you execute statement 1, then go to go back to the expression. If the expression is false then you go to statement 2. So, while the expression is true execute statement 1 and if the expression is false, then execute statement 2.

So, the difference in the, if condition will be that if this was an if block. Then, if the expression is true you do statement 1 and you exit out of the while loop. And that is not done in the case of a normal while loop. After you execute the statement you go back to the expression. So, as long as the expression is true you keep executing statement 1 and if the expression becomes false then you execute statement 2.

(Refer Slide Time: 02:47)

Example 1

1. Read a sequence of integers from the terminal until -1 is read.
2. Output sum of numbers read, not including the negative number.
 - » First, let us write the loop, then add code for sum.

```
#include <stdio.h>
main() {
    int a;
    scanf("%d", &a); /* read into a */
    while ( !( a == -1) ) {
        scanf("%d", &a); /* read into a inside Loop*/
    }
}
```

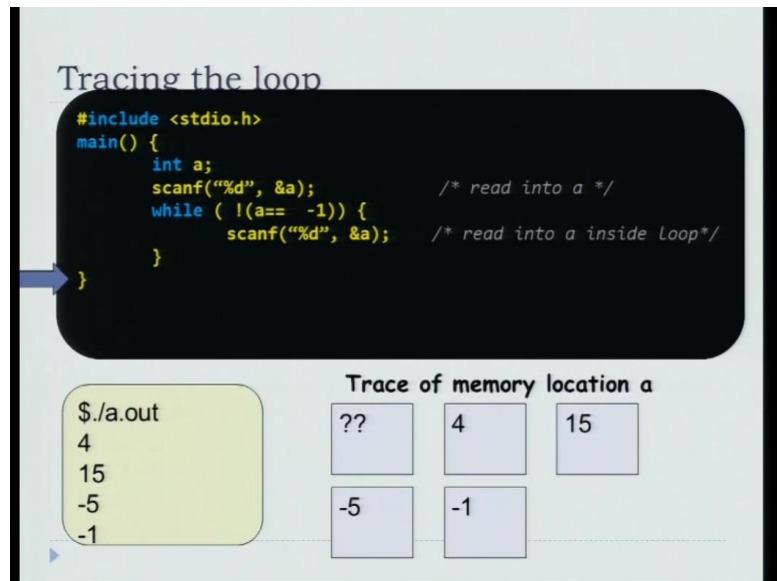
So, let us illustrate the use of a while loop with the help of a program. So, we will introduce a very simple problem which is, read a sequence of integers from the terminal until **-1** is encountered. So, **-1** signals that the input is at end. Now, what I want do is that sum up all the numbers until the **-1** and output the sum. It is a very simple program. What you have to do is to read a sequence of numbers, until you hit the first **-1** and then add this numbers and output their sum.

So, let us first introduce the very simple loop which will do only the basic thing of reading the numbers until a **-1** is encounter. So, how do you write the loop you have **stdio.h**. And then you declare an integer variable and read the variable. So, this is supposed to be the first number. If that number is **-1** then you do not have to read any more numbers. So, if the number is not **-1**. So, if **a = -1** is false then you read one more number.

After you read one more number you do not finish the loop, you go back and test whether the loop condition is still true. So, you go back and check whether the second number you read was **-1** or not. And then, you keep on reading it until you hit a **-1**. At some point when you hit a **-1** you go back to the loop and the condition that **a = -1** will be true. So, NOT of that will be false and you will exit the loop.

So, read the first number if it is a **-1** do not enter the loop; otherwise, keep on reading numbers until you hit a **-1**. That is the meaning of the while loop.

(Refer Slide Time: 04:50)



So, let us just trace the execution of the loop on a sample input to understand how it works. So, in a box I will represent the memory location a and its current content. So, I run the program after compiling `a.out` and let us say that I enter the number 4. Now, you scan the number 4. So, memory location a becomes 4. Now, 4 is not `-1`. So, you enter the loop. So, then let us say the next number is 50, you read the number into a. So, memory location a is now 15, 15 is not `-1`.

So, you again enter the loop, you enter `-5`, `-5` is not `-1`. So, you enter the loop again. At this point you enter, you scan the number into a and a becomes `-1`. So, you go back to the loop again and now the test that so, `a = -1`, so, naught of that is false. So, the while condition becomes false at this point you exit the program. So, this is a very simple part of the program that we want to write, recall that we want to read a bunch of numbers and sum them. And the end of the numbers is represented by a `-1`. Until now we have just read those numbers.

Introduction to Programming in C

Department of Computer Science and Engineering

In this session, we will continue the program that we were writing. Recall that, we were writing a while loop, which will read a bunch of numbers. And it is supposed to sum them up, until you hit a **-1**. In the loop that we have seen, so far we just read the numbers until **-1** was encountered. So, let us now complete the program and compute their sum as well. So, for computing there sum, how do we normally do it?

We will add numbers two at a time. So, the first two numbers will be added. Then, that sum will be added to the third number and so on, until you hit a **-1**. So, let us try to do that, in the course of a while loop.

(Refer Slide Time: 00:46)

The slide has a title "Add numbers until -1" and a bullet point: "Keep an integer variable s. s is the sum of the numbers seen so far (except the -1).". Below is the C code:

```
#include <stdio.h>
main() {
    int a;
    int s;
    → s = 0;
    scanf("%d", &a);           /* read into a */
    while ( !(a == -1) ) {
        → s = s + a;
        scanf("%d", &a);   /* read into a inside Loop */
    }
    /* one could print s here etc. */
}
```

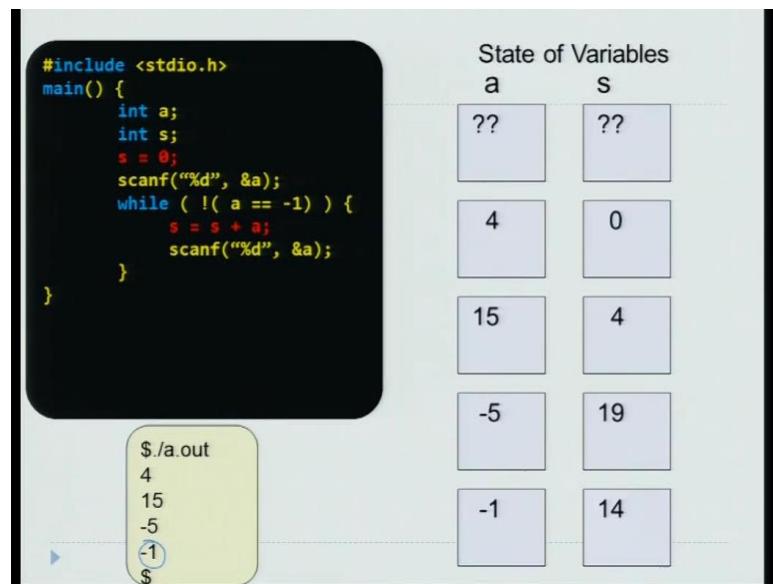
What I will declare is, I will declare a new variable **s**. So, here is the new variable **s** that, I have declared. **s** is supposed to the hold the sum of the variables that, I have read so far. Now, it is very, very important that, when you declare a variable, you should initialize them properly. In the case of **a**, we did not initialize it because, we were reading the first number, as soon as was declared.

But, in the case of sum, you would use **s** to maintain the sum, as you read numbers. So, it is important that, you start with **s = 0**. So, the initialization step marked by this arrow is quite important. If you do not initialize it properly, then the sum may not be correct, as

we will see soon. So, we keep a variable s, which is supposed to hold the sum of n numbers, sum of these numbers and initialize the sum to 0.

Then, the difference from the loop that, we have seen so far is highlighted in red. So, earlier recall that, we were reading the number. And just testing, whether the number is **-1**, if it was not **-1**, you read one more number so, that was the loop. Now, inside the loop, what we will do is, we will keep up, running sum of the numbers that we have seen so far. So, initially s sum is initialized to 0. Then, if the first number is not **-1**, you add the first number to s. So, s will now be the first number. Now, read the second number. If the second number is not **-1**, you will enter the loop again. So, you will add the second number to s. So, s is now first number plus second number. And this keeps on going until you hit **-1**, in the input. So, let us continue with this.

(Refer Slide Time: 02:47)



Let us try to trace the execution of this program, on a sample input and try to understand, how it works. Let us say that, I compile the program successfully and run the program. So, I run a dot out and let us as before, let us the first number be 4. So, after initialization, when you declare the variable a is undefined and s is also undefined. After the initial statement, **s = 0**, s is now 0. And then, you scan the variable a. So, a becomes 4, because 4 was the input. And sum is still 0. You enter the loop and you say, **s = s + a**. So, sum becomes 0 plus 4, which is 4. And you read the next number. Let us say the next number was 15. So, a becomes 15, a is not **-1**. Therefore, you enter the loop again. And

sum is now 4 plus 15, which is 19. So, sum at any point of time is the sum of the number that, we have read so far. So, we have read 4 and 15. So, the sum is 19.

Now, you read the next number. Let us say, the next number was **-5**. **-5** is not **-1**. Therefore, you enter the loop again s equal to s plus **-5**. So, s becomes 14. Then, you read the next number. And let us say, the next number was **-1**. So, since the number read is **-1**. You go back to the loop. And this condition becomes false. So, you exit out of the loop. And then print that, the sum is let us say 14.

So, when you verify it by hand, you would see 4 plus 15 plus **-5** is 14. So, you have, the program has executed correctly. The important thing to note is, the final **-1** is not summed up. So, that is, it is used as the end of the input and you should not compute the sum of the numbers, including **-1**. **-1** is excluded. Then, the program executed, correctly.

(Refer Slide Time: 05:42)

The screenshot shows a terminal window with a black background. On the left, there is a code editor window containing the following C code:

```
#include <stdio.h>
main() {
    int a;
    int s;
    s = 0;
    scanf("%d", &a);
    while ( !( a == -1 ) ) {
        s = s + a;
        scanf("%d", &a);
    }
}
```

On the right, the terminal window shows the output of the program. It starts with the command `$./a.out`, followed by three inputs: `4`, `15`, and `-5`. After these inputs, the program exits because the input is `-1`. The terminal ends with the prompt `$`.

Terminology

- ▶ **Iteration:** Each run of the loop is called an **iteration**.
- ▶ In example, the loop runs for 3 iterations, corresponding to inputs 4, 15 and -5.
- ▶ For input -1, the loop is broken, so there is no iteration for input -1.

We will introduce a few terminology associated with the notion of a loop. Each execution of a loop is known as an iteration. So, in the above loop, when the input was 4, 15 **-5 -1**, the loop runs for three iterations, corresponding to the inputs 4, 15 and **-5**. So, for input **-1**, the loop is broken. So, you do not enter the loop. So, you do not count an iteration corresponding to **-1**. So, you entered four numbers including the **-1** and the loop executed three times. So, you say that, the loop had three iterations. So, this is a technical term associated with the loops.

(Refer Slide Time: 6:16).

Loop invariant

```
#include <stdio.h>
main() {
    int a;
    int s;
    s = 0;
    scanf("%d", &a);
    /* Invariant: s holds the
       sum of all values read
       except the last one */
    while ( !( a == -1) ) {
        s = s + a;
        scanf (" %d", &a );
    }
}
```

- ▶ Loop invariant: A property relating values of variables that holds at the beginning of each iteration of loop.
- ▶ A good way of thinking about loops and proving correctness—i.e., program meets its specifications.

And here is a concept that, I will introduce to help you argue about the correctness of a loop. So, there is a notion known as a loop invariant. Now, a loop invariant is a property relating values of the variable that holds at the beginning of each loop. So, thus bit abstract let me just illustrate with the example, that we just saw. So, loop invariants are a good way of thinking about the correctness of loops that, you have to do.

So, in our program what will be the loop invariant? Let us look at the property of that, we are interested in. There are two variables in the program, s and a. And both of those variables are involved in the loop. But, the interesting property that we have relates to s. What is the property that, s holds with respect to the loop. So, we can see that, s holds the sum of all values read so far, except the last value is that true, the first time that we enter the loop? Yes, because s was initialized to 0. And you had actually read the number.

So, it is true that, s holds the sum of all values, except the first one. So, that is true, when you first enter the loop. And that, any point when you enter the loop, you sum the last value that was the read. And read one more number. So, you will see that, s still holds the sum of all values read so far, except the last one. So, this is the loop invariant in the program. And loop invariants help you, argue about the correctness of the loops.

(Refer Slide Time: 08:13)

A flavor of program correctness

```
#include <stdio.h>
main() {
    int a;
    int s;
    s = 0;
    scanf("%d", &a);
    /* Invariant: s holds the
       sum of all values read
       except the last one */
    while ( !( a == -1 ) ) {
        s = s + a;
        scanf( "%d", &a );
    }
}
```

- ▶ If invariant is correct, then the value of s upon termination must be correct.
- ▶ Because: loop terminates when it reads -1, so, the value of s is the sum of all numbers read except the last -1, by invariant.

So, if the loop invariant is correct and the program maintains loop invariant, then the value of s when the program stops, will be correct. Why is that? Because, the loop terminates, because the last value read was a **-1**. And the invariants says that, s holds the sum of all values, except the last value. So, this means that s holds the sum of all numbers, except the **-1**.

Therefore, when the program ends that is, you exit out of the loop, s holds the sum of all number that you were supposed to add. So, here is how arguing about loop invariant and seeing, whether loop invariant holds in the loop that you have written, helps you argue that the program is correct.

Introduction to Programming in C

Department of Computer Science and Engineering

We will see a few more examples, because loops are really important. Let us go back to the first problem that we discussed, which was the problem of computing the greatest common divisor of two positive numbers. So, the problem is to read the two numbers, find their GCD and compute the output.

(Refer Slide Time: 00:37)

More examples

- Problem: read two numbers, find its gcd and output.
- We had a flowchart of the problem based on the fact that, if $a \geq b$ then $\text{gcd}(a,0)$ is a
 $\text{gcd}(a, b)$ equals $\text{gcd}(b, a \% b)$ where,
 $a \% b$ is the remainder when a is divided by b .

Now, we had a flowchart of the problem based on the fact that, if $a \geq b$, then $\text{GCD}(a,0)$ if b is 0, then $\text{GCD}(a,0)$ is a . Otherwise, $\text{GCD}(a,b)$ is the same as $\text{GCD}(b, a \% b)$, with $a \% b$ is $a \% b$ is the remainder, when a / b . So, let us now try to write the program in C using a while loop. So, we have to do a few preliminary things.

(Refer Slide Time: 01:12)

The slide has a light blue background with a dark blue header bar. The title 'Writing gcd program (first half)' is in the header. Below it is a dark blue rounded rectangle containing C code. To the right of the code is a list of bullet points and some handwritten notes.

Code:

```
#include <stdio.h>
main () {
    int a;
    int b;
    int t; /* stores temporary value */

    /* Now read input values */
    scanf( "%d%d", &a, &b);

    /* Ensure that a is the Larger of a
       and b, if not exchange a with b */
}
```

Notes:

- Exchanging values of a and b cannot be done by
 $a = b; b = a;$
- After $a = b$, the old value of a is lost.
- The cyclic exchange

$a = 3; \quad b = 4;$

$\boxed{a = b; \quad b = a;}$

$a = b;$ results in $a = 4;$

So, let us call up the first half of the program. In the first half I declare three variables, a, b and there is another variable t, whose need we will see right now. But, let us say that I need an extra variable for now, let us just take it on faith. So, what I will do is scan two variables a and b. Now, recall in the GCD equation that we saw right now, we assume that $a \geq b$. Now, what if the user is unaware of his condition and enter the lesser number first.

So, he just entered the numbers in such a way that, $a < b$. So, we need to correct that, we need to make sure that a is the greater number. So, we need to exchange the values of a and b, if it is true that a is less than b. Now, how do we do this? So, the first thing to note to try will be to say that for example, if I say that let us say a was 3 and b was 4. And suppose, I just said $a = b$ I want to exchange the values of a and b. I just said $a = b$ and $b = a$.

What will be the effect of this? Note that, this is the assignment statement. So, after I execute this line, b is 4 so, a will be 4, $a = b$ results in a equal to 4. After the execution of this line. So, what this situation that we will have is that b equal to 4 and a equal to 4. And we will have no memory of, what was the original value of a? That is lost. So, it is just simply lost.

So, this idea that we can exchange two values by just writing a equal b, $b = a$ does not work. So, what is the correct way to do it? So, we have an idea known as the cyclic exchange and this is a really neat idea. The idea is that... So, how can I motivate it? Let

us say that you have two rooms and these two rooms are full of steps. And I want to change the contents of the first room to the second and the second room to the third.

One way I can do it is that I will move the contents of the first room to a different room. So, have a temporary room and then copy the contents of the second room to the first and copy the contents of the third room to the second. So, this is a very nice intuition and it almost is similar to what we need to do.

(Refer Slide Time: 04:28)

Writing gcd program (first half)

```
#include <stdio.h>
main () {
    int a;
    int b;
    int t; /* stores temporary value */

    /* Now read input values */
    scanf( "%d%d", &a, &b);

    /* Ensure that a is the Larger of a
       and b, if not exchange a with b */
    if (a < b) {
        t = a;
        a=b;
        b = t;
    }

    /* now a is >= b */
    /*... continued on next slide */
}
```

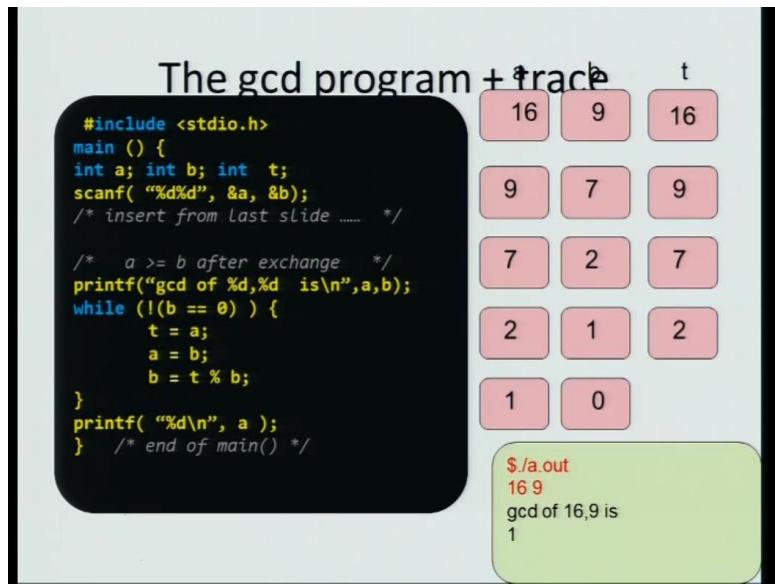
- Exchanging values of a and b cannot be done by $a = b; b = a;$
- After $a = b$, the old value of a is lost.
- The cyclic exchange

```
graph LR
    S1[t=4] --> S2[a=6]
    S2 --> S3[b=6]
```

So, we have two variables a and b that we need to swap. So, one thing we can do is, I will keep a third variable t. First what I will do is, I will copy the value of a to t. So, now I have a backup copy. Now, then I will say $a = b$. So, at this point the value of b will be copied to the a. So, now, a is 6, b is 6. But, still we have a memory of what was a before? Because, the old value of a was stored in t.

So, in order to complete the routine, all you need to do is to copy the value of t to b and that can be completed by the equation by the assignment statement b equal to t. So, this idea is known as cyclic exchange. So, recall the physical intuition of swapping the contents of two rooms which are almost full. You have a third room and you move the contents of the first room to the third room, move the contents of the second to the first and move the contents to the third to the second. So, this is similar to what we did. In the case of physical contents, we cannot copy, in the case of variables, we can copy.

(Refer Slide Time: 05:52)



So, now let us complete the GCD program. We have just done the first part of the program which is to ensure that a is actually the greater number. If it was not the greater number, you swap or exchange. Now, So, after exchange we have ensure that $a \geq b$. Now, we have to write the main loop for the GCD function. So, just by translating the flowchart, what we will do is while b is not 0. What you do is, you say that store the value of a and t. Assign a to b, $a = b$ and b becomes $t \% b$.

Recall the equation was written as follows that. So, recall that the equation was $\text{GCD}(a, b)$ is the same as $\text{GCD}(b, a \% b)$. So, when you assign a to b, the old value of a is lost and we can no longer do $a \% b$. So, the way to do that is, you use the idea of temporary variable, store the old value of a and t, before you do $a = b$. So, that finally, $a \% b$ can be done by $t \% b$. I do not want the new value of a, I want the old value of a.

So, let us just trace the execution of this program. Let us say that I scan two numbers a and b and the user was correct in entering it. So, he actually enter the greater number first. So, we have 16 and 9, a equal to 16 and b equal to 9 and t is undefined. So, after you read the numbers, you just say GCD of... After these you enter a message which is printed message which is the GCD of a and b is,. So, GCD of 16 and 9 is and then you enter the loop.

So, in the initial execution of the loop you have t equal to a, which will store t equal to 16, $a = b$, which is a will become 9. And then, but you want to compute the modulo 16 % 9, but 16 was lost in a, because a is not 9. So, you have do $t \% b$,. So, 16 % 9 which is 7.

So, you go back to the while loop and then you see that b is not 0, b is 7. So, you enter the loop again. T is equal to a,. So, t is 9, $a = b$,. So, a become 7, b becomes 2.

Again b is not 0, So, you enter the loop again. So, t is 7, a equal to 2 and b equal to 1. Again you enter the loop, b is not 0, t is 2, a becomes 1 and b becomes 0 at this point, you exit the loop and at this point a that we ended with is the GCD of these numbers. So, 16 and 9 are relatively prime. Therefore, their GCD is 1. Now, let us think a minute about what is the loop invariant in this program?

(Refer Slide Time: 09:42)

```
#include <stdio.h>
main() {
    int a; int b; int t;

    scanf( "%d%d", &a, &b);

    if (a < b) { /* exchange */
        t=a; a=b; b = t;
    }
    printf( "gcd of %d,%d  is\n", a, b);

    /*Invariant: gcd (A,B) = gcd(a,b)*/
    while ( !(b == 0) ) {
        t = a;
        a = b;
        b = t % b;
    }

    printf( "%d\n", a );
} /* end of main() */

```

Loop Invariant

- Let A,B be the values input to a,b where, A,B > 0.
- Invariant:
 $\text{gcd}(A,B) = \text{gcd}(a,b)$
 holds at the beginning of each iteration of loop.
- This guarantees correctness.
- How many times does the loop run?
 More complicated.

What is it that? We have a central while loop which computes the GCD. What was the invariant in that loop? So, for this I will just introduce a slight notation which makes it easier to discuss this invariant. So, let capital A and capital B be the original numbers that I input. And little a and little b represent the numbers which are involved in the loop. So, capital A and capital B are the original input and the invariant that I have is that at every stage, the GCD of the original inputs are the same as the GCD of little a and b.

We call that little a and b are the loop variables involved in the loop. So, little a and b keep changing through the loop, whereas, capital A and B are fixed, they are the input. So, the invariant that I have is that every time you enter the loop, the GCD of the input where the same as the GCD of the variable. Now, this guarantees the correctness. Because, when you exit out of the loop, you will correctly compute. You exit out of the loop, because b equal to 0, and by the original equation, you know that when b is equal to 0, a is the GCD of a and b. So, this guarantee is correctness.

Now, you could also ask other questions like, how many times has the loop run? And this question is big complicated, because you have to compute it based on the input numbers. So, such questions are of interest to computer science. But, we will not going to computing the efficiency or the performance of this code. But, it is also a very crucial question.

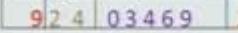
Introduction to Programming in C

Department of Computer Science and Engineering

So, we will see one more example for writing loops, we see as slightly trickier example and will cover this over multiple sessions. So, the problem is the following.

(Refer Slide Time: 00:17)

Example

- Read a sequence of numbers until a -1 is read. Output the length of longest contiguous increasing subsequence.
- Example input:
9 2 4 0 3 4 6 9 2 -1
- The increasing contiguous sub-sequences are:

The largest one is 0 3 4 6 9 and has length 5.
- Example 2 : 11 9  15 -1
Largest contiguous sub-sequence is
7 8 11 12 15 and has length 5.

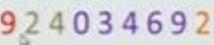
We have to read as sequence of numbers until a **-1** is read, **-1** indicates that then the sequence of sent it. And the question is we have to output the length of the longest contiguous increasing subsequence. So, that is a lot of words let us illustrate it with an example. Let us say that the input is of the following numbers - 9 2 4 0 3 4 6 9 2, and then finally a **-1**. And we have to output the length of the longest contiguous increasing subsequence. So, let us say what do I mean by a contiguous increasing subsequence? So, I say that 9 is an increasing subsequence, then the next number is 2; 2 is less than 9. So, 9 and 2 cannot be part of a subsequence, where the numbers keep on increasing.

So, 2 is the start of a new sequence, again the next number is 4. So, 2 and 4 form an increasing sequence. So, you can continue increasing a sequence. The next number is 0, 0 is less than 4, so break the sequence there. Then when you look at then succeeding numbers 0 3 4 6 9; they form an increasing sequence. And the last number is 2, which is lesser than 9. So, the increasing sequence stops here. So, these are the increasing contiguous subsequences; contiguous means together occurring adjacent to each other. So, the largest of the longest contiguous subsequence is obviously, 0 3 4 6 9, and the length of that sequence is 5.

Let us take another example 11 9 7 8 11 12 15 15 and -1. So, just to illustrate the point 11 is greater than 9. So, that cannot be an increasing sequence, 9 is greater than 7, so that is another the increasing sequence is just 9, but then 7 8 11 12 15; these are increasing. And I decided to stop here even though the next number was 15, because I am interested in and increasing subsequence. So, 15 and 15 are equal numbers. So, we break it ((Refer Time: 02:59)). So, the longest increasing subsequence is 7 8 11 12 15 and its length is 5. So, this the longest contiguous increasing subsequence.

(Refer Slide Time: 03:23)

Basic Property

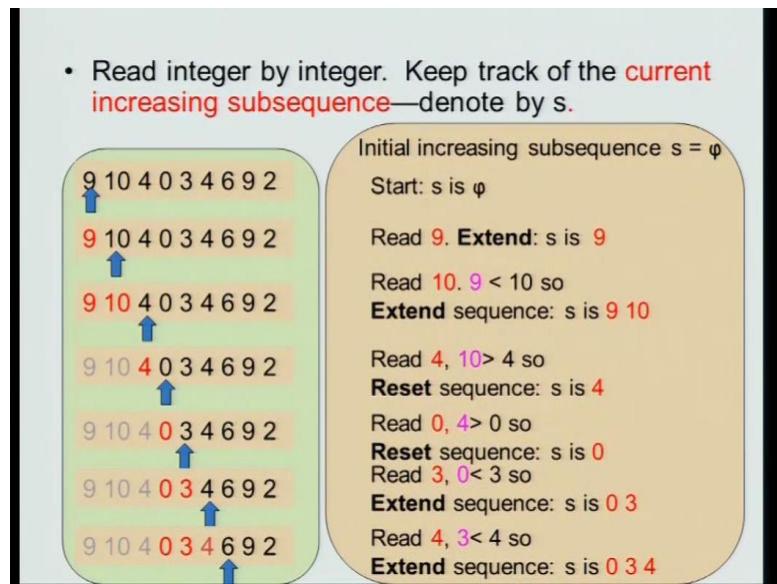
- Every sequence decomposes (breaks) into a collection of contiguous increasing subsequences.
- Example


There are 4 increasing sub-sequences: (i) 9 (ii) 2 4 (iii) 0 3 4 6 9 and (iv) 2.
- We will use this property to design the loop.

So, here is a basic property given any sequence of numbers, we can break it into a collection of increasing contiguous subsequences. For example the numbers that the sequence that we have seen. So, 9 2 4 0 3 4 6 4 6 9 2, and that the length of the increasing, the longest increasing contiguous subsequence is 5. So, we have to write a program to do this, given a sequence of numbers find the length of the longest increasing subsequence.

So, how do we do it? We do it in the way that we have been writing loops so far, like adding n numbers and finding this sum and so on. The idea was that you start from the first number and keep on adding the numbers until you hit -1 at which point you have this sum. So, the idea of this algorithm was that you start from the first, and you keep reading, until certain condition happens. We will adopt that idea to solve our current problem.

(Refer Slide Time: 04:28)



So, what we need to do is to keep track of the current increasing subsequence. Let us say that it is denoted by s . So, before we get into the code, let see how we will do it by hand. So, initially the increasing subsequence is s , and let say that it is empty. After you read 9, you have an increasing subsequence which consist of exactly 1 number. So, s is 9. Now, the next number is 10; 10 is greater than 9. So, you extend this. Read the next number 4; 4 is less than 10. So, 9 10 4 cannot be an increasing subsequence, therefore you say that you break the subsequence there, so 9 10 is a different subsequence.

Now you start a new subsequence which is 4. So, the current subsequence is just 4. So, 0 is less than 4, so you break it there, the current increasing subsequence become 0. Read the next number 3, 3 is greater than 0. See you extend the subsequence s is now 0 3 4, 4 is greater than 3. So, the sequence becomes 0 3 4 and so on. So, what are we doing here? We are reading the read, we are reading the numbers integer by integer, and we are keeping track of the current increasing subsequence. So, this is part of what we want to do?

(Refer Slide Time: 06:16)

- It would suffice to store the previous number (p) read and compare with current number (c).
- If $p < c$, then extend the sequence else reset.
- Extend means add 1 to length of current increasing subsequence, reset means set this to 1.

etc.

Initial increasing subsequence $s = \emptyset$

Start: s is \emptyset

Read **9**. **Extend**: s is **9**

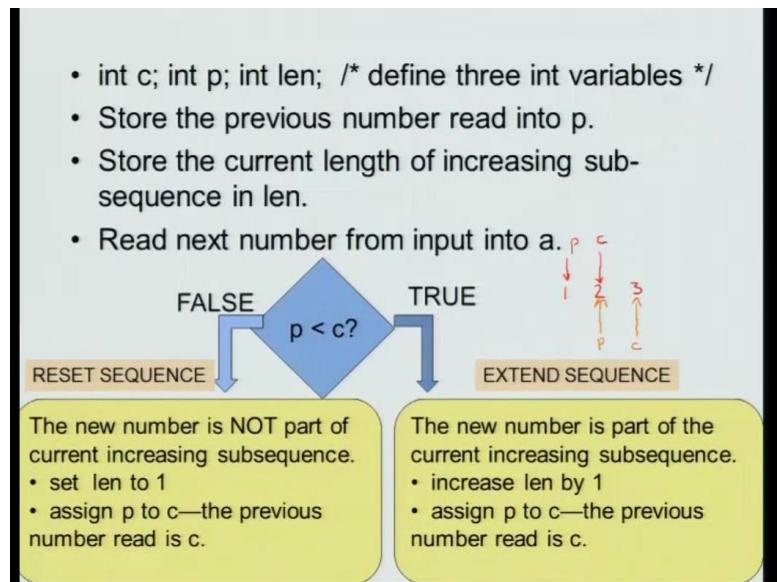
Read **10**. **9 < 10** so
Extend sequence: s is **9 10**

Read **4**, **10 > 4** so
Reset sequence: s is **4**

Now, in order to decide whether we should increase the subsequence extend it or break the subsequence, and start a new sequence what we need to do is we should remember what was the previous number we saw, and what is the current number that we have seen. So, with two variables we can say whether the subsequence should be extended or should be broken at this point.

So, if the previous number is less than the current number, then we should increase the subsequence. If the previous number is greater than or equal to the current number then we should break the subsequence and start a new sequence. So, extend the sequence means add 1 to the length of the current increasing subsequence, and add that number to the subsequence. Reset means you start a new sequence of length 1.

(Refer Slide Time: 07:15)



So, from the current from the description that we have seen so far, we need the following variables, we need C which is for the current rate c number, p which is the previous number that we have seen, and length which is the length of the current increasing sequence. So, we store the previous number into p, store the current length of the increasing subsequence into length, and read the next number into c.

So, if the previous number is less than the current number. So, we take the true branch in which case we extend the sequence. So, the new number that we have read is part of the currently increasing subsequence. So, increase the length of the sequence by 1 and now we move lockstep. So, what we do is? So, we are at a stage where suppose we have numbers 1 2 and 3, suppose p was pointing to 1, c was pointing to 2. So, since 2 was greater than 1, we extend the sequence. After extending the sequence, we have to proceed and see what will happen with the next number. So, when you do that you can do the following, I will extend the current sequence by doing the following, I will now set p equal to 2, and c equal to 3.

So, this is the idea that we will advance both the variables by 1 number each. So, that it is always true that previous is 1 number behind current. So, I hope this idea is clear that in order to ensure that p is 1 number behind current, you have to advance both p and c. So, assign p to c, this will advance p and then read the next number. So, that will become c. Now what happens if $c \geq p$, then the new number is not part of the current increasing

subsequence. So, you start a new sequence which is of length 1 and again do the same assign p to c, which is advancing the pointer and read the next number. So, here is the method that we will follow in order to keep track of the current increasing subsequence. Now what is left is to find the longest of all the increasing subsequences that we find.

Introduction to Programming in C

Department of Computer Science and Engineering

(Refer Slide Time: 00:31)

```
#include <stdio.h>
main () {
    int prev;      /* previously read integer */
    int curr;      /* currently read integer */
    int len=0;     /* Length of current incr. subseq.*/
    scanf("%d",&prev);           /* increasing subseq. */
    if (!(prev == -1)) {
        len = 1; /* Length of current incr. subseq is 1*/
        scanf("%d",&curr);

        while (curr != -1) { /* valid number read */
            if (prev < curr) { /* extend sequence */
                len = len + 1;
            }else{
                len = 1; /* reset sequence*/
            }
            prev = curr;          /* always do this */
            scanf("%d", &curr);   /* read next int */
        }
    }
}
```

In this session we will try to code up the C code for finding the length of the longest increasing contiguous sub sequence. So, let us first examine what we need to do, we will write a code, and from the previous discussion we saw that we need at least three variables; one for a storing the previous number, one for storing the current number, and the third for storing the length of the current decreasing sub sequence. So, we start by declaring all those three variables and initializing length to 0. So, here is a new construct that we are seeing for the first time, which is that when you declare a variable, you can also initialize it immediately by saying len equal to 0. So, this is a very intuitive notation. So, this will declare a variable and immediately initialized it to 0. Once we declare these three variables, let say that we scan the first variable into previous.

Now, let us focus on, on the main body of the program. If the currently read number is if the currently read number is $\neq -1$, then you say that you start with length 1. So, the length of the current increasing sub sequences 1, and then you scan the next number into curr. So, here is current number. So, this part of the code is just to initialize. So, if the current, if the first number is -1 , then there is no point in getting into the program, because the its equivalent to the empty inputs. So, there is no increasing sub sequence to be found. So, you just exit out of the program ((Refer Time: 01:58)). So, initially we just

check to see whether the first number is **-1** or not. If the first number is **$! = -1$** , you scan the next number, so current will be the second number. And if current is **$! = -1$** while the currently read number is **$! = -1$** . What you do is exactly the logic that we were discussing before. If the previous number is less than the current number then you extend the length by 1.

So, length equal to length plus 1 says that I am continuing the current increasing sub sequence by increasing its length. Otherwise that is current is less than or equal to previous, you break the sequence and say length equal to 1. Then we have this step previous equal to current, which is the advancing both variables by 1. So, previous becomes the currently read number, and current becomes the next number to be ((Refer Slide: 03:02)). So, recall from the diagram that previous and current were at some position, and we will advance both of them by 1. And when the loop condition is check the next time we will check whether the currently seen number is **-1**. So, so far we have coded up part of the logic, which is the part of the logic dealing with when the current when the next number is read do we extend the sequence or do we break the sequence and start a new sequence. So, this is just part of the work that we need to do to solve the problem. So, lets continue with the logic.

(Refer Slide Time: 03:40)

```
#include <stdio.h>
main () {
    int prev;
    int curr;
    int len = 0;
    scanf("%d",&prev);
    if (prev != -1) {
        len = 1;
        scanf("%d",&curr);
        while (curr != -1){
            if (prev < curr){
                /*extend */
                len = len + 1;
            }else{
                /*reset*/
                len = 1;
            }
            prev=curr;
            scanf("%d", &curr);
        }
    }
}
```

- Let us try on some boundary cases first (usually a good idea).
- Now that it works on a boundary case, we can try on other cases.

5 >
 -1
 ↑
prev curr len
5 -1 1 ✓

So, let us start with a few boundary cases, and let see that whether these works. If it works we can try or logic on other cases. So, let us say that by boundary cases I mean

may be very long inputs or very short inputs. So, these are cases where your code normally breaks. So, when you test your code it is always a good idea to check boundary cases. And one thing that makes programming difficult is that in or when we do things by hand, we know how to handle the boundary cases elegantly, but in a program unless you say how to handle the boundary cases, the program might break. And the lot of testing and the lot of errors come from incorrectly handling the boundary cases. So, it is always good to handle the boundary cases, let us try test our code on very small inputs.

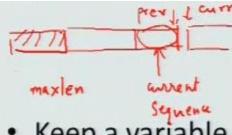
So, let us say that I enter a sequence **5 - 1**. So, previous becomes, so length is 0; previous becomes 5 and then since **prev != 1**, I have used an abbreviation here which is the not equal to operator, this is the same as saying not of previous equal to equal to 1. So, previous not equal to 1 is the same as saying not of previous equal to equal to 1, its an operator in C, then you say that the len equal to 1, then you say that the length is 1, because you have seen 1, 1 number and scan the next number. The next number is **-1**. So, you scan the next number, and the next number is **-1**, so you break the sequence. So, immediately exit out of the sequence, and the length of the increasing sub sequence that we saw, so 5 is the only increasing sub sequence. And when we exited the length was 1. So, we handle the boundary case of an extremely small sequence, a sequence with exactly 1 element correctly. This gives us confidence that the code could be correct, of course we have handle only the boundary case. Now, we see need to test it for other cases as well.

(Refer Slide Time: 06:17)

```

int prev; int curr; int len=0;
int maxlen = 0;
scanf("%d",&prev);
if(prev != -1) {
    len = 1; maxlen = 1;
    scanf("%d",&curr);
    while(curr != -1) {
        if(prev < curr) {
            len = len + 1;
        }else{
            if(maxlen < len){
                /* Longer subseq. found*/
                maxlen = len;
            }
            len = 1;
        }
        prev = curr;
        scanf("%d",&curr);
    }
    /*when the last subseq is longest*/
    if(maxlen < len){
        maxlen = len;
    }
}

```



- Keep a variable **maxlen**, initialized to 0.
- When a new increasing subsequence is found, compare its length (**len**) with **maxlen**.
- If **maxlen < len**, we have a new larger incr. subseq.

So far the program is not doing anything useful, because we are just extending the sequence and breaking the sequence. What recall that the what we was suppose to do was to fine the length of the longest increasing sub sequence. So, this is the main, this the main thing that we have to do in the logic. So, to do this what we do is something simple. We keep track of the maximum length sequence that we have seen so far, keep track of the length of the longest sequence that we have seen so far. Also we have the current sequence. Now all we need to do is whether to check whether the current sequence is longer than the previously known longest one. So, for this what we do is keep track of the maximum length that we have seen so far. So, this is a standard technique in program. And how do I do that.

So, let us modify the program a little bit. So, earlier we resend out that we need at least three variables. Now in order to keep track of the length the maximum length that we have seen so far, I need a new variable. So, this part we have already done before. And here is the **maxlen = 0**. So, that is the new variable which is the maximum length that we have seen so far. When we start the program we have not seen any increasing sub sequence, and therefore the length of the longest increasing sequence, the current increasing sequence is 0; that is **len** equal to 0. And the length of the maximum length that we have seen so far is also 0.

Then you scan the new number; if the new number is not **-1** you continue. So, length equal to 1, now max length equal to 1, because currently the longest sequence that we have seen so far is 1 1 long. You scan the num next number. If you... So, here is the

main body of the loop, and what we need to do is the following, if the currently read number is greater than the previous number, we extend the sequence. So, this logic is the same as before. Otherwise which means that current number is that less than or equal to previous. So, we are starting the new sequence. So, the situation is the following we have some maxlen sequence somewhere in the past. So, maxlen is the length of the sequence that we have seen somewhere in the past.

Now we are scanning in the sequence we have a current sequence. And we have decided to break this sequence. So, we have we are now starting a new sequence starting at current. So, we are at this part of the logic. So, we have decided to start a new sequence, that is because the current sequences last number is greater than or equal to the current number. So, here is previous, and this is current. So, we are deciding to start a new sequence what we need to see is whether this sequence is longer than the previously known maximum length. If the sequence that we just stopped is longer than the previously known maximum length sequence. So, if `maxlen < len` notice that length is then sequence that length of the sequence that we just stopped. Then we say that `maxlen = len`. So, if the current sequence is longer than the previously known maxlen, what we do is that maxlen becomes the length of this sequence. Otherwise if the current sequence was shorter than the previously known maximum length, we do not do anything, so maximum length is the same...

Introduction to Programming in C

Department of Computer Science and Engineering

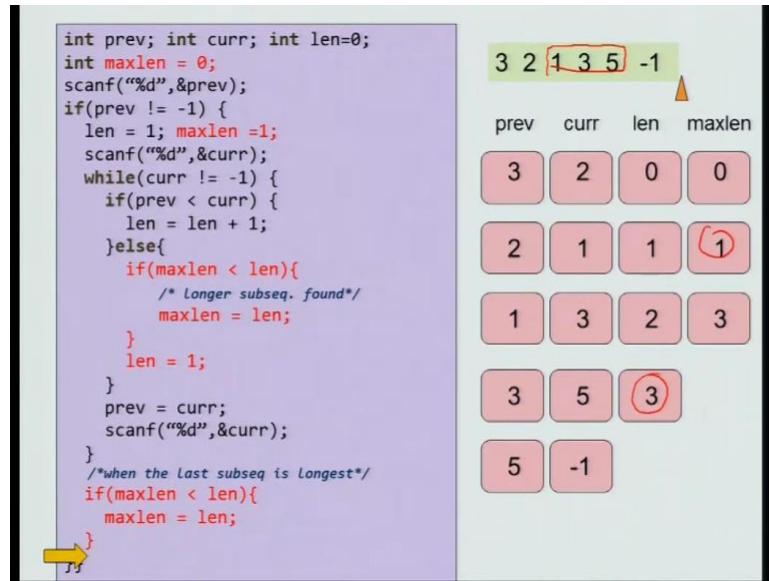
(Refer Slide Time: 00:27)

```
int prev; int curr; int len=0;
int maxlen = 0;
scanf("%d",&prev);
if(prev != -1) {
    len = 1; maxlen =1;
    scanf("%d",&curr);
    while(curr != -1) {
        if(prev < curr) {
            len = len + 1;
        }else{
            if(maxlen < len){
                /* Longer subseq. found*/
                maxlen = len;
            }
            len = 1;
        }
        prev = curr;
        scanf("%d",&curr);
    }
    /*when the Last subseq is Longest*/
    if(maxlen < len){
        maxlen = len;
    }
}
```

- Keep a variable **maxlen**, initialized to 0.
- When a new increasing subsequence is found, compare its length (**len**) with **maxlen**.
- If **maxlen < len**, we have a new larger incr. subseq.

So, when we extent the sequence we don't have to do anything special when we break a sequence, and we start a new sequence, then all we have to do is you check whether the currently say the sequence that you just saw was longer than the previously known longest sequence. If that is the case then the sequence that just ended is becoming the longer sequence we have seen so far. Otherwise you maintain the max length. So, just forget about the currently stop sequence. Now there is a... So, that this loop, and at the end we have to do slight tricky logic, it could so happen that the sequence ends with a longest sequence increasing sub sequence. In that case, we will never reset the max length. So, if the last sequence is the longest, you also have to handle the case separately. So, we will see an example where, if you exit out of the loop that is you have already seen a **-1**, you just have to check whether the last increasing sequence that you saw was in fact the longest. So, there is a small if block at the end to do that.

(Refer Slide Time: 01:25)



In this part we will just see, small tracing of this program on a sample input. So, that the logic of the program become slightly more clear. So, I have picked a particular input 3 2 1 3 5 -1, and you will see that the longest increasing sequences are 3. So, the increasing sequences are 3, then 2, then 1 3 5. So, 1 3 5 is going to with a longest increasing subsequence, and let us see how our program will find that out. So, initially you have a bunch of variables which should be declare. So, len = 0, maxlen = 0, and previous and current are undefined. Then you first read previous. So, previous becomes 3, it is not -1, so you enter the if condition, at which point you set length and max length to 1.

Now you scan the current number. So, current becomes 2. So, remember that previous is now 3, and current is 2. So, current is not -1, therefore you enter the while loop. Prev < curr is false, because previous is 3 and current is 2. Therefore, you enter the else part, maxlen < len is false; both are 1. Therefore, you start a new sequence with length equal to 1. Now you continue the loop with previous becoming current.

So, previous is now 2 and current you read the next number which is 1. So, previous and current have both more 1 step. So, current is not -1, prev < curr is again false, because 2 is greater than 1. So, you enter the else part. Max length and length there is no change. So, you reset the length to 1, previous is current. So, current previous becomes 1, and you scan the next number which is 3. Now at this point previous is 1, and current is 3. So, the if condition is true. So, you extend the length; length increases by 1. Again you

advance previous and current. So, previous becomes 3, current becomes 5. Again 3 is less than 5, so increase the length we are extending the sequence. So, the length becomes 3. Advance, so previous becomes 5, and current becomes -1 at this point you exit the loop.

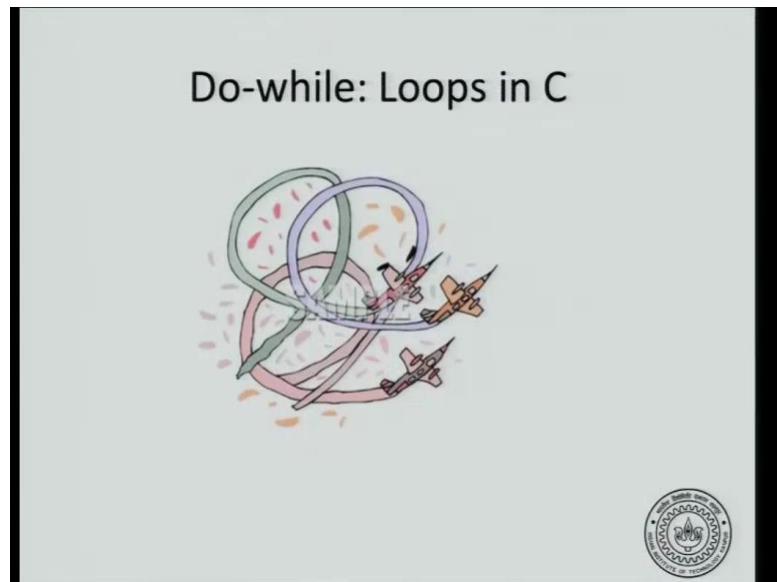
And now you encounter the situation that max length, which is the length that we have seen so far, recall that it is one, but the length of the sequence that we just stop the see the input with is 3; that is that happen, because the longest increasing ((Refer Time: 05:09)) contiguous sub sequence, well was at the end of the input. So, it happen right at the end. So, when we exit the loop we have to do 1 additional check, we cannot simply say that the maximum length that we have seen in the sequence is 1, because max length is the length of the longest sequence we have seen before the current 1. The current 1 was the 1 that we just stop to with it had a length of 3. So, we just check, if max length equal to length is less than the length, then we set max length to be the length. So, once you do that max length becomes 3. This is just to handle the case when the longest increasing sub sequence is the last. Now you can exit out of the exit out of the if condition, and then print that the maximum length that you have seen is 3.

Introduction to Programming in C

Department of Computer Science and Engineering

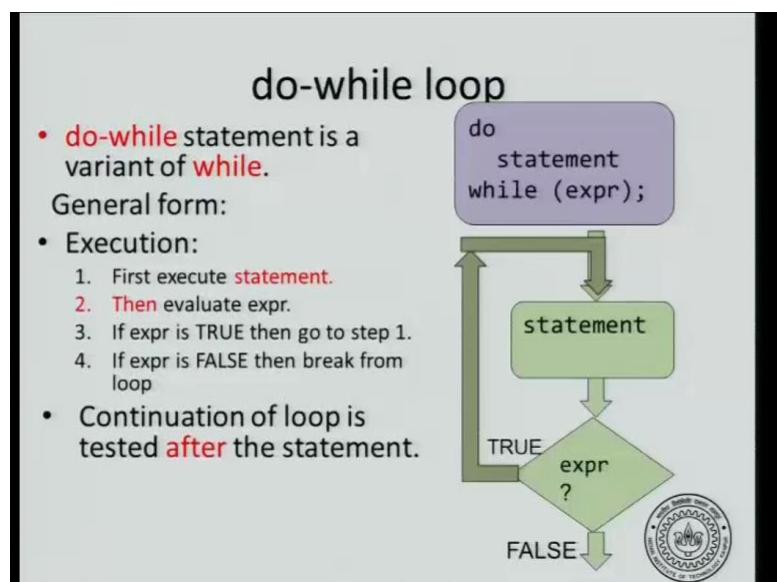
So far we have been using while loops in C, now C programming language also provides you other kinds of loops.

(Refer Slide Time: 00:12)



Let's look at some of them. The first alternative loop mechanism in C that we will look at is what is known as a do-while loop?

(Refer Slide Time: 00:20)



And so it is a variant of a while loop, I am the general form is what you see here, you have do statement followed by while expression. And here is an important syntactic difference which causes some syntax errors, when you code. The do while terminates within semicolon, where is the while loop does? So, the while loop has the following a form which is while expression, and then statement; the difference is that here the statement is occurring before the while the test expression. So, the way it execute is the following. We first execute the statement, then evaluate the expression. If the expression is true, you go back to step 1; that is execute this statement. If the expression is false, then you get out.

So, you execute the x statement then test whether the expression is true or not, if it is true you go back and the execute the statement again, so you loop. If the statement is false, you get out of the loop. The difference from while loop and do while loop is the following, you have statement that will be executed without testing the expression even once. So, when you start executing the loop, you will first execute the statement without testing the expression, and after testing the expression you will go back and test the loop expression, if it is true and you start executing the loop again. So, the first execution of the statement there is no test done for that.

(Refer Slide Time: 02:13)

Comparative Example-I

- Problem: Read integers and output each integer until -1 is seen (include -1 in output).
- The program fragments using while and do-while.
- The while construct and do-while are equally expressive (whatever one does, the other can too).

<p style="text-align: center;">Using do-while</p> <pre style="background-color: black; color: cyan; padding: 10px;">int a; /*current int*/ do { scanf("%d",&a); printf("%d ",a); } while (a != -1);</pre>	<p style="text-align: center;">Using while</p> <pre style="background-color: black; color: cyan; padding: 10px;">int a; /*current int*/ scanf("%d",&a); while (a != -1) { printf("%d ",a); scanf("%d",&a); } printf("%d ", a);</pre>
--	---



So, let us see the comparison between a loop while loop and do while loop. So, we will look at the following problem, you have to read numbers and output in integer until a **-1**

is seen. Now the difference is that in this problem you have to include the **-1**. So, read all the numbers up to an including **-1**, and print all the numbers. So, we will have the following programs using while loop and do while loop. Now the important thing to notice is that the while construct and the do while construct are equally expressive. So, you cannot right any more new programs using the do while construct, then you could use the while construct, but certain kinds of programs are easier using or shorter using the do while construct. For example let us solve this problem using the while construct. So, what you do initially is, you declare a variable then scan the variable; if the variable is **-1**, you immediately exit out of the loop, and print **-1** and finish the program.

If the number is not **-1**, you print the value and scan the next number. If the number you scan is not **-1**, you just print it and repeat the loop. If it is **-1**, you exit out of the loop and print the **-1** that you saw. So, here is the logic using the do while loop, in using the while loop. And notice that when we existed out of the loop we needed a printf statement, and before you yes, enter the loop you needed a scanf statement. So, this was the structure of the program. This problem can be elegantly solved using the do while loop. What you initially need to do is to declare a variable, then scan the variable and print it any way. Either the number is **-1** or it is not. In any case we need to print it.

So, go ahead and print it then test whether the number was **-1**. If it is **-1**, you're done and you exit out of the program. If it is not **-1**, you go back and scan the next number and print it. So, this is a program that we have seen where you could do this same think with the while loop. The only difference is that the do while program is shorter. And please be careful about the syntactic difference between the while loop and the do while loop, notice the semicolon at the end this causes a lot of confusion when you compile the program it is easy to miss this.

(Refer Slide Time: 05:05)

Programming Tip

- If you are new to C, cultivate your loops programming using one of while or do-while.
- When you are comfortable with one of them, the other construct becomes easy.



If you are new to C programming, you can stick to one particular loop. As I said before you cannot write any new programs that you can do is using the do while loop, then you could previously do using the while loop. So, you can write the same logic, you can write the same number of programs using the while loop, and the do while loop it gives you no further power. So, it is recommended that you stick to one loop pick while or pick do while whatever you do, but stick to that loop in when you right the program. When you are comfortable with one of the loops programming using the other loop becomes easy.

(Refer Slide time: 05:51)

```
int prev; int curr; int len;
int maxlen;
scanf("%d",&prev);
maxlen = 1; len=1;
if (!(prev == -1)) {
    do {
        scanf("%d",&curr);
        if (prev < curr) {
            len = len + 1; /*extend */
        }else{
            if (maxlen < len) {
                maxlen = len;
            }
            len = 1;
        }
        prev = curr;
    } while ( !(curr == -1) );
}
if (maxlen < len) {
    maxlen = len;
}
```

- Find length of longest increasing subsequence ending in -1, including -1 in sequence.



So, let us try to solve a problem that we have already seen, which is to find the length of the longest contiguous increasing subsequence ending in **-1**. The difference that we have is that earlier we did not include **-1** in the sequence when you computed the length of the sequence, now we will include **-1**. So, here is the program to do that and the logic - the core logic, so here is the initialization, and here is the loop logic, and the final check.

So, if you recall from the lecture which covered the problem solving the longest increasing subsequence, then you will see that the main structures in the code. The main lines of logic in the code are pretty much the same. All I have done is to change the while logic to the do while logic. And let see what that is accomplish for us. So, what this does is that you will scan a particular number, if the particular number is bigger than the previous number, then you extend the sequence. If it is less than or equal to the previous number, then you stop this sequence and started new sequence, this was the logic.

And when you start a new sequence the length is new start with 1. Then you say current equal to the next number, and previous equal to the number that was just read. So, the logic here is that the testing for whether the currently rate number is **-1** is done at the end of the loop. So, is the first number is **-1**, you just do all this and then say that the length of the increasing subsequence is 1, then you test if the currently read number is **-1** or not. If the currently read number is **-1**, then you are already done and you exit out of the loop. Then you check whether max length is less then length as before. ((Refer Time: 08:26)) difference between this logic, and the logic that we have seen before is that we do this execution without testing whether the currently read number is **-1**. So, automatically what happens is that if the number is **-1**, all these steps will be performed before we test that the sequence has ended. So, automatically we ensure that **-1** is also included when we calculate the increasing subsequence.

Introduction to Programming in C

Department of Computer Science and Engineering

In this session, let us look at a matrix problem and the problem is given as follows.

(Refer Slide Time: 00:06)

A Matrix Problem

- The first line of the input consists of two positive integers m and n.
- This line is followed by m lines, each containing n integers, signifying an $m \times n$ matrix A. We have to calculate the sum of the squares of the sum of numbers in each row and print it.

$$\sum_i (\sum_j A_{ij})^2, i=0..m-1, j=0..n-1.$$

	3	4		
row	4	7	11	2
i	1	1	2	4
	2	9	0	-1

e.g. $A_{20} = 2, A_{12} = 2$

Desired output

$$(4+7+11+2)^2 + (1+1+2+4)^2 + (2+9+0+(-1))^2$$

You have a bunch of lines of input. The first line of the input consists of two numbers, m and n. This line is followed by m lines, each containing n integers. So, this is supposed to represent the matrix of size m time m cross n. We have to calculate the sum of the squares of the sum of numbers in each row, that is quite a mouthful, let us look at the formula. So, what I want to do is, $\sum_j A_{ij}$. So, once you do this sum, you square that and then do the \sum_i .

So, i is an index which goes over the rows. In C, we adopt the convention that the first row is starting with 0 and so, it goes from 0 to m minus 1. Similarly, the first column starts with index 0, so it goes on till n minus 1. So, the input will be given as follows. The first number m represents that there are 3 rows, the second number represents the number of columns in each row and then this is followed by a matrix of size 3 cross 4.

Now, the numbering is given as follows. For example, if you have A_{20} , this means second row zeroth column. Note that, second row means we start with row 0, then row 1, then row 2 and zeroth column is this, the first column. So, A_{20} is this number which is 2. Similarly, A_{12} is first row, row number 1 column number 2, which is also 2. So, the

desired output that we have is $(4 + 7 + 11 + 2)^2$ and so on. So, how do we do this by hand? Let us just look at the calculation.

(Refer Slide Time: 02:15)

$$\sum_{i=0}^2 \left(\sum_{j=0}^3 A_{ij} \right)^2$$

$$\left(\sum_{j=0}^3 A_{ij} \right)$$

4	7	11	2	$24^2 = 576$
1	1	2	4	64
2	9	0	-1	<u>100</u>

$$\sum_{i=0}^2 \left(\sum_{j=0}^3 A_{ij} \right)^2 = 740$$

So, we have 4 7 11 2 1 1 2 4 and 2 9 0 minus 1. Notice that the formula that we have to calculate is i going from 0 to 2, j going from 0 to 3 A_{ij} squared. So, how do we do this?

First, we sum the numbers in each column so, I will name that as j going from 0 to 3 A_{ij} and then square. So, if you sum all this, you see 11 22 24^2 which is 576 and similarly, this is 64 8^2 and this is 10^2 , which is 100.

So, what I have tallied on the right hand side is, for each row you sum the numbers take that sum and square it. And then finally in order to compute what we want, which is $\sum_{i=0}^2 (\sum_{j=0}^3 A_{ij})^2$. In order to calculate this, all we need to do is to sum these numbers up and this turns out to be 740.

So, notice when we did by hand, we did the following, we first calculated row wise, we summed over all the numbers in that row. Take the sums, square it. Then, you repeated the same operation for the next row and then for the third row. So, we have three numbers and then we added them in sequence. So, we will see how we will code this up?

(Refer Slide Time: 04:22)

Double loops

- Need something of a double loop here (loop inside a loop).
- One loop to do the row sum of each row.
- Once a row is finished, we square the row sum.
- Another (outer) loop to add the squares of row sum over all rows that have been fully read.



So, what we need here is something called an inner loop or a double loop, we need a loop inside a loop. Now, the inner loop is doing what we did first? It is taking a row and adding all the numbers in that row, then squaring it. So, we need one loop to do the rowsum of each row. Once a row is finished, we square the rowsum. Once that is done, remember that once we tallied numbers on the right hand side and squared them. We have to add those numbers up. So, we need another loop, an outer loop to add these squares of rows sums.

(Refer Slide Time: 05:07)

Inner loop: Row sum

- Easy part first: assume we are at the beginning of a row (have not read any numbers yet) and write a loop to calculate the row sum.

```
int a;           /* the current integer */
int colindex;   /* index of current column */
int rowsum=0;    /* sum of row entries read so far */
int rowsumsq=0;  /* square of the sum of row entries */

while (colindex < n) {      /* not finished reading n cols*/
    scanf("%d", &a);        /* read next number */
    rowsum = rowsum + a;    /* add to rowsum */
    colindex = colindex + 1; /* increment colindex */
}

rowsumsq = rowsum * rowsum; /*square rowsum */
```

So, let us do this program in stages. First, let us write the inner loop, this is the loop, so that for a given row you sum up all the numbers in that row. Now, let us assume that we are at the beginning of a row and we have not read any numbers yet. Now, what we have to do is to start reading the numbers. So, we write a while loop. We declare four variables `a`, column index, then we need something for the `rowsum` and some integer variable for `rowsumsq`.

Now, what you do is you go along the row and add the numbers in each column. So, while the column index is less than `n`, recall that the matrix size was `m` cross `n`. You scan the next number, the next number is added to the `rowsum` and then increment the column index. Until you hit `n`, recall that the last column is `n` minus 1, because we start the column numbers from 0.

Now, once you have done you have the sum of the numbers in that row and what you need to do is to square that number. So, we have `rowsum` times `rowsum` will be `rowsumsq`.

(Refer Slide Time: 06:33)

Outer Loop Structure

- We have a code that reads the next `n` integers from the terminal and sums them.
- Modify it so that it reads the next `m` integers **from the output of the previous code**, specifically the value of `rowsumsq` and sums them.

That completes the inner loop, which is what we did, when we added the numbers along a given row and then finally, squared the sum. Now, what we need is an outer loop structure over these. So, we have a code that reads the next `n` integers from the terminal and sums them. Now, what we need is some further code, that takes the output of the previous code and then sums all those numbers up.

Remember, when we did this by hand, this was the second operation we did, we went over the right most column and added all those numbers up and that was the result that we wanted.

(Refer Slide Time: 07:13)

- Task: Modify code below so that it reads the next m integers **from the output of the previous code**, specifically the value of **rowsumsq** and sums them.

```

int a;          /* the current integer */
int colindex;   /* index of current column */
int rowsum = 0;  /* sum of row entries read so far */
int rowsumsq = 0; /* square of the sum of row entries */

while (colindex < n) {      /* not finished reading n cols*/
    scanf("%d", &a);        /* read next number */
    rowsum = rowsum + a;     /* add to rowsum */
    colindex = colindex + 1; /* increment colindex */
}
rowsumsq = rowsum * rowsum; /*square rowsum */

```

So, how do we modify the code?

(Refer Slide Time: 07:17)

- Previous code modified to read the next m integers **from the output of the previous code**, specifically the value of **rowsumsq** and Outer Loop: Still in Design Phase: incomplete and informal

```

int rowindex=0; /* index of current row being read */
int sqsum=0; /* sum of col entries read so far */

while (rowindex < m) { /* not finished reading m rows*/
    sqsum=sqsum+`rowsumsq'; /* add to colsum */
    rowindex = rowindex + 1; /* increment colindex */
}
printf("%d ",colsum);

```

rowsumsq comes from previous code. Let's insert that code here.

So, let us what we need is something like this. Assume that we have the output available from the previous code in some variable called rowsumsquare. And we need a loop over

that, which is going from rowindex 0 to m minus 1 and tallying up all the numbers in rowsumsquare. So, for each row you will end up with a rowsumsquare and you have to add all those rowsumsquare to get in. So, in this the rowsumsquare comes from the previous code.

So, this is how we will visualize the outer loop. Now, note that this is not completely specified code. This is just a very intuitive picture, that instead of rowsumsquare, it should come from some inner loop which actually calculates it. So, we should plugin the output from the previous inner loop and this is the outer loop over it.

(Refer Slide Time: 08:36)

Insert inner loop code into outer

```
int rowindex = 0; /* index of current row being read */
int colsum = 0; /* sum of col entries read so far */
int a;
int colindex;
int rowsum;
int rowsumsq;
while (rowindex < m) { /* not finished reading m rows */
    rowsum = 0; colindex=0; /* re-initialization */
    while (colindex < n) { /* not finished reading n cols */
        scanf("%d", &a);
        rowsum = rowsum + a;
        colindex = colindex + 1;
    }
    rowsumsq = rowsum * rowsum; /*square rowsum */
    colsum = colsum + rowsumsq; /* add to colsum */
    rowindex = rowindex + 1; /* increment colindex */
}
```

So, here is how we put these two loops together, we have a while loop inside the while loop. Remember that, this was the previous loop that we had written. So, this was the inner loop that we have written, where at the end of the inner loop you have the rowsumsquare. Now, at the end of the code what you will end up is the exact rowsumsquare of a particular column. So, you can imagine that after this inner loop finishes execution, the rowsumsquare is the correct rowsum is the correct square of the sum of the elements in the row.

Now, the outer index does the following, you start from rowindex 0 and sum the rowsumsquare over all columns. So, this is the structure of the code. Let us look at the code in slightly more detail. We have a rowindex, a column index, a column sum, a rowsum and the rowsumsquare, a is supposed to be the current number that we are read

in.

Suppose, we know that the size of the matrix is m cross n. So, rowindex can go from 0 to m minus 1. So, the termination condition of this while loop is rowindex equal to m. Now, for all that you are currently at a particular row. So, you have to initialize the rowsum to 0 and the column index to 0. Because, for a given row you have to start from row 0 for a given row you have to start from column 0 and you go on, until column n minus 1.

Also this previous rowsum should not influence the next row. So , for every row you have to initialize the rowsum variable. Once that is done, you go over the columns of that given row and you scan the numbers add the number to the rowsum and increment the column index, until you hit n columns, column 0 through n minus 1. When you read all the columns in that row, you have the correct rowsum. So, that rowsumsquare can now be calculated.

Once rowsumsquare is calculated, you have the rowsumsquare for that particular row. So, add the rowsumsquare to the previously computed columns. So, If you go back and think about how you did this by hand, you can convince yourself that this is exactly the coding of the logic that we had earlier.

(Refer Slide Time: 11:25)

The screenshot shows a C program on the left and a step-by-step execution table on the right.

C Program:

```
int m,n;
int rowindex = 0;
int sqsum = 0;
int a, colindex, rowsum;
scanf("%d,%d", &m,&n);
while (rowindex < m) {
    rowsum = 0;
    colindex=0;
    while (colindex < n) {
        scanf("%d", &a);
        rowsum=rowsum+a;
        colindex=colindex+1;
    }
    sqsum=sqsum +
        (rowsum * rowsum);
    rowindex=rowindex+1;
}
```

Input: 2 3
1 0 -1
0 1 1

Output should be 4

Execution Table:

a	m	rowindex	colindex	rowsum	sqsum
1	2	0	0	0	0
0	2	0	1	1	0
-1	2	0	2	1	4
0	2	1	0	0	4
1	2	1	1	0	4
1	2	1	2	1	4
	3				4

So, let us try this on a small example to see exactly, how the code works? Suppose, the input is 2 3 followed by two rows of three numbers each. So, the input matrix size is 2

cross 3 and the entries are 1 0 minus 1 and 0 1 1. Let us see, how the code executes on this? So, the output should be 4, if you do it by hand and let us see, the variables are m, n, a, rowindex, column index, rowsum and square sum. Finally, the result should be in square sum.

You start with rowindex equal to 0, column index equal to 0 and you scan m and n. So, you already know the size of the matrix, when you scanned m and n. So, m becomes 2 and n becomes 3. Now, rowindex is 0 which is less than 2. So, it starts the loop which reads the row 0. So, notice the arrow here, you are starting to read this particular row, the first row, which is row 0. Or you initialize rowsum equal to 0, column index equal to 0 and while column index is less than n, you scan the next number which is 1.

Add a to the rowsum. So, rowsum becomes 1, increment the column index. So, it reaches column 1 row 0 read that number. Add it to the rowsum, go to the second column and read the number and add it to the rowsum. So, once you are done, now column index is 3. So, just means that we have read all the entries in the row 0. So, we have got the correct rowsum. What we will do is, add the rowsumsquare to the square sum.

So, rowsum is 0, 0 square to square sum, so square sum remains 0. Now, you go to the second row. So, increment rowindex. Now, rowindex is less than 2 rowindex is 1. So, it is less than 2 we are reading row 1 and you repeat the same execution. We reinitialize the rowsum to 0, column index to 0. and then, scan the next number which is 0. Add it to the rowsum, increment the column index, scan the next number which is 1 and so on, until you finish reading the second row as well.

So, once you read the second row, you will find that the rowsum is 2 and square sum would be square sum plus 2 square which is 4. After you do that, you increment rowindex and rowindex becomes 3, which is greater than the given rowindex. So, you exit the loop. So, we have correctly computed the sum that we wanted.

Introduction to Programming in C

Department of Computer Science and Engineering

In this session we will see a very popular loop construct in C.

(Refer Slide Time: 00:07)

For statement in C

- General form
 - `for (init_expr; test_expr; update_expr)
statement;`
- `init_expr` is the initialization expression.
- `update_expr` is the update expression.
- `test_expr` is the expression that evaluates to either TRUE (non-zero) or FALSE (zero).
- Execution:
 1. First evaluate `init_expr`;
 2. Evaluate `test_expr`;
 3. If `test_expr` is TRUE then
 - 4. execute `statement`;
 - 5. execute `update_expr`;
 - 6. go to Step 2.



We have already seen while loops and do while loops, will see that do while loops are not all that common in C code, when C programmers code. Among the most popular loop construction C is this, for loop. So, let say what it stands for? The expression for the general form of the for statement, the slightly more complex than that of a while loop. While loop was very simple, while as the certain expression was true, you execute the statement and when the expression becomes false, you exit out of the loop, for loop is slightly more complex.

So, it has the following components, it has an initialization expression, then the test expression, this the expression corresponding to the expression inside the while loop and then there is an update expression, followed by the loop statement. This looks complex at first,, but it is quite intuitive once you start using it. The execution is as follows, first you execute the initialization expression, then you test whether the test expression is true or not.

If the test expression is true, you execute the statement and then come back and execute the update expression. After you execute the update expression go back to step 2, which is go to the test expression. So, init expression is the initialization expression, update expression is the update expression and test expression is the expression, that is evaluates to either true or false.

(Refer Slide Time: 01:54)

The diagram illustrates the equivalence between a for loop and a while loop. It starts with a blue box containing the general form of a for loop:

```
for (init_expr; test_expr; update_expr)
    statement;
```

Below this, a bulleted list states: "Execution is (almost) equivalent to". An orange box contains the corresponding while loop code:

```
init_expr;
while (test_expr) {
    statement;
    update_expr;
}
```

Finally, another bulleted list notes: "Almost? Exception if there is a continue; inside statement - this will be covered later." A small circular logo of the Institute of Technology is visible in the bottom right corner of the slide.

So, if you look at the flow of how the code goes, then it is first you start from the initialization expression, then you go to the test expression. If the test expression is true, you go to the statement, then you go to the update expression and you go to the test expression again. So, the loop is here you test the expression, execute the statement, update and test again, initialization is done only once. So, this is the first step and here is the loop, this sounds bit complex at first,, but it is quite simple to use, once you get the hang of it.

So, the execution of the for loop can be understood in terms of the while loop. The execution of the for loop is almost equivalent to the following while loop, you have the initialization expression before the while loop, then the test expression, while test expression, then you have statement and then you have the update expression. So, if you have a for loop you can write the equivalent code using while loop. So, if you say that I do not want to use for loops, here is how you have a for loop and you can write the equivalent while loop in the following way.

Or if you have a while loop, you can write a equivalent for loop by looking at the this form and how it is translate to the corresponding for loop? Now, there why did I say execution is almost equivalent, we will see this later in the course. Whenever, there is a continues statement or a break statement, you will see that we need to modify this as equivalents between the for loop and the while loop. But, for now for with the features of see that we have seen so far. The for loop is equivalent to the while loop and we will have to modify this slightly later.

So, the init expression maps to the first part of the for loop, the test expression maps to the second part and the update expression maps to the third part. One important thing to notice is that, the update expression is after the statement. So, we have the following first we execute the initialization expression, then we test whether the expression is true. If it is true, you execute the statement, update expression and then again go to the test expression, if it is true you execute statement, update and then test again.

So, you initialize the expression then when the test the test expression if it is true, you execute the statement after the statement is true, after the statement is executed you update the expression and go back to the test expression. Because, that is how you execute it in the while loop? You first initialize, then test whether it is true execute the statement, update and then go back to the test expression. So, this is how a while loop can be translated to a for loop and **vice versa**.

(Refer Slide Time: 05:23)

Enough definitions

- Let's do some examples.
- Print the sum of the reciprocals of the first 100 natural numbers.

$$1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{100}.$$


So, let us do some examples very simple think, let us say that print the sum of reciprocals of the first 100 natural numbers. So, what do I want to do? I want to do the following, I want to do $1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{100}$. So, how would I do it? I would initialize a variable call sum, sum will be initialized 1 and then 2 sum I will add $\frac{1}{2}$, then to that I will add $\frac{1}{3}$ and keep on going until $\frac{1}{100}$.

(Refer Slide Time: 06:08)

Enough definitions

- Let's do some examples.
- Print the sum of the reciprocals of the first 100 natural numbers.

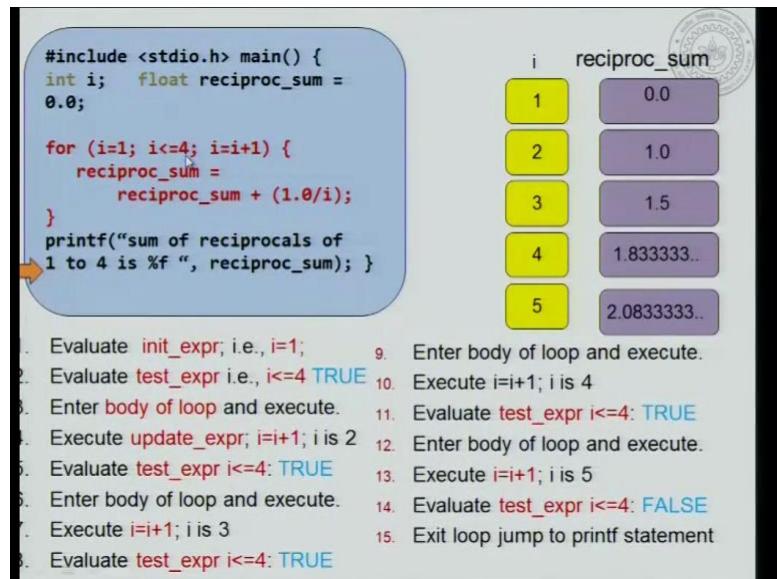
```
int i; /* counter: runs from 1..100 */
float reciproc_sum = 0.0; /* sum of reciprocals */
for (i=1; i<=100; i=i+1) { /* the for loop */
    reciproc_sum = reciproc_sum + (1.0/i);
}
printf("sum of reciprocals of 1 to 100 is %f ", reciproc_sum);
```

So, let see how to code this in C using the for loop. So, I have a variable call reciprocal sum and even though I am summing over integers, we know that the reciprocal numbers will be real numbers. So, in order to keep the reciprocal sum I need a floating point number, floating point variable and then I have an integer variable, which goes from 1 to 100.

So, here is how I will do the loop? First initialize i to 1, if $i \leq 100$, you enter the loop and do reciprocal sum equal to the current reciprocal sum plus 1 over i. After doing that you update by saying $i = i + 1$. So, increment i. Once the increment is done, you test whether the new number is less than or equal to 100, if it is less than or equal to 100, you do the reciprocal sum come back update, until you reach 101.

At the point where you reach 101 you test whether $i \leq 100$ that becomes false and you exit. So, you will see that when you exit out of the loop, the reciprocal sum will be the sum of reciprocals of numbers from 1 to 100. So, here is how the for loop functions.

(Refer Slide Time: 07:45)



So, instead of 100 let us try to executed on a particularly very small number to see how this for loop executes. So, let us instead of summing from 1 to 100, let us sum from 1 to 4. So, first you have the initialization expression. So, `i` is undefined before you enter the while loop, reciprocal sum is of course initialize to 0. So, I can after initialization `i` will be 1, as soon as it is initialized we will test whether it is less than or equal to 4, 1 is less than or equal to 4 that is true.

So, you will enter the for loop, then you add to the reciprocal sum 1 over `i`, `i` is 1,. So, 1 over 1 is 1. So, reciprocal sum will be updated 2 reciprocal sum plus 1. So, reciprocal sum would be 1, then you go to the update expression, at this point you have `i = i + 1`, So, `i` becomes 2. Now, test whether `i <= 4` yes it is and enter the loop. So, 1 plus 0.5 then go back to the update expression `i` becomes 3 now and test whether 3 is less than or equal to 4 it is,. So, enter the loop.

So, you add 1.5 plus $\frac{1}{3}$, 1.833 and. So, on, update again you have 4, 4 is less than or equal to 4 that is true. So, you enter the loop one more time and add 1 over 4.25 to the current number. So, you get 2.0833 and. So, on update again `i` becomes 5, at this point 5 is not less than or equal to 4,. So, you exit out of the for loop. Now, you say that print that the sum of reciprocals from 1 to 4 is reciprocal sum, which is 2.0833.

So, even though the for loop looks complicated, once you start using it, it is very nice to right, you have a initialization expression, you have a test expression and then you have the update expression, that you should do after every execution of the loop, after every

iteration you should have the update expression. As soon as the update is over, you test whether I can execute the loop one more time, if I can enter the loop update and test again and. So, on, until the loop condition is false.

(Refer Slide Time: 11:00)

Simple example

- Input is given in two lines.
 - The first line contains a single number m that specifies the number of integers that follow.
 - The second line contains m integers. Output the sum of the m numbers.

Sample input	5 2 -1 15 3 6
--------------	------------------

Strategy is simple:

1. Read the first integer into m .
2. Keep a variable say sum that is intended to store the sum of all the numbers in the second line read so far. Initialize sum to 0.
3. Run a for loop, reading a new number and adding it to sum —this loop is set to run m times.



Let us take another example, you have two lines, the first line contains a single number m , which specifies how many numbers are there in the second line. The second line contains m integers and we have to just output the sum of the m numbers. Now, we know how to do this, we have already done this using a while loop, let us try to do it using a for loop. So, the sample input is let say the first line is 5 and then I have 5 integers on the second line.

The strategy is very simple, you read the number on the first line into m and then have a variable called sum , which will start with the first number and keep on adding the subsequent numbers, until you have read m numbers initialize sum to 0. So, run a for loop from the first number to the m th number and keep adding the numbers to sum . So, this loop will run for m times.

(Refer Slide Time: 12:11)

```
Sample input 5  
          2 -1 15 3 6
```

```
int m, i, sum, num; /* num stores the next integer, i is used as for loop counter, sum is the partial sum of numbers read. */  
scanf("%d", &m); /* read the length of seq. */  
sum=0;  
for (i=0; i < m; i = i+1) /* for loop: to run m times */  
    scanf("%d", &num); /* read the next number */  
    sum = sum+num; /* add to running sum */  
printf("sum of given %d numbers is %d",m, sum);
```

i	0	1	2	3	4	5
num	2	-1	15	3	6	
sum	0	2	1	16	19	25

Output(on one line)
sum of given 5 numbers is 25

So, let us code this up, you have m , i , sum and $numbers$ which are all integers. First you scan the number m , initialize sum to 0 this is important. Because, if sum is not properly initialized it is sum garbage value and you keep adding numbers to it, you will get garbage value as the output. So, initialize the number sum properly to 0 and then here is the for loop, what the for loop does is, you start with $i = 0$ and go on until i less than m .

Now, you could also do the following could start with i equal to 1 and go on until exactly m . So, if you start with i equal to 1 you will say i less than or equal to m , you can adapt either convention, in C it is more popular to start from 0 and go on until m minus 1. So, you break the loop when i is equal to m . So, here is the test condition for the loop and then you have the loop body, which is you read the number and add the number to sum and after you have done that, you have the update expression which is $i = i + 1$.

So, here is the how the for loop looks you start from 0 and go on until i becomes m , you add the number and just increment i , which is i is the number of integers we have seen so far. Let us do trace of this execution, you start you have this integer variables and you first read m which is 5, the number on the first line and then we do things in order, you have initialized sum to 0, you start with $i = 0$.

Once you do the initialization expression i become 0, i is less than m 0 less than 5 that is fine. So, you execute the loop, scan that next number, which is 2 add it to the sum . So, sum becomes 2 now update, update is increment i .. So, i becomes 1 and test whether 1 is less than 5 it is. So, you read the next number add it to the sum .. So, this sum becomes 1

update again and keep repeating this, until you have read all 5 numbers.

So, when you read the 5th number i will be 4, after that you add the 5th number to the summation. Once you done i will be incremented to 5, 5 is not less than 5, 5 is equal to 5. So, you will exit out of the loop, at this point you will have the correct sum,. So, the correct sum will be 25 and you exit on. So, the printf will come out on one line, it will say that the sum of given 5 numbers is 25.

So, what I will recommend is, write the same program using a while loop and a for loop and see how you can easily go from while to for and for to while. The advantage of the for loop and the reason why for loop become,. So, popular among programmers is that, in comparison to the while loop, it is first of all it is easier to read. Because, you have all the initialization expression, the update expression and the test expression all on one line. So, you see what the loop is about. The second is that, it involves fewer lines of code, then the corresponding while loop. So, it is a very popular loop among programmers.

(Refer Slide Time: 16:45)

Initializing multiple variables using

Earlier Program

```
int m, i, sum, num;
scanf("%d", &m);
sum=0;
for (i=0; i < m; i = i+1) {
    scanf("%d", &num);
    sum = sum+num;
}
printf("sum of given %d
numbers is %d",m, sum);
```

Equivalent Program

```
int m, i, sum, num;
scanf("%d", &m);

for(sum=0,i=0; i<m; i= i+1) {
    scanf("%d", &num);
    sum = sum+num;
}
printf("sum of given %d
numbers is %d",m, sum);
```

The program on right is equivalent to the original prog. (left).
The expression for initialization is sum=0, i=0
It assigns to the variable sum the value 0 and then assigns to the variable i the value 0.

Now, here is a syntactic convenient that C providers and let me make this remark as the final thing in this session. So, notices that we had to initialize two variables here. So, the first is sum was initialized to 0 and the second was that i was initialize to 0. Now, would not be convenient, if I could do this together and that is what C provides us. So, I have something known as the comma operator. So, the normal comma that we have seen.

So, in order to initialize multiple variables at the same time, I can say sum equal to 0 comma i = 0. So, C will initialize the variables in the order, that it is given, first it in will

initialize sum to 0 and then it will initialize the $i = 0$. So, here is a very synthetically convenient notation that C provide for as the advantage again is that you end up with fewer lines of code.

Introduction to Programming in C
Department of Computer Science and Engineering

In this section, we will use the, for loop to code of the matrix problem. So, remember that we have seen while loop and we have seen a do while loop. Inside while loops we have written nested loops or double loops. So, let us look at a for loop which problem, where the solution involves a nested loop.

(Refer Slide Time: 00:25)

A matrix problem

- For loops are a good choice when the number of iterations is known ahead of time. Good example is matrices. Here is a problem.
- The first line of the input has a positive integer n , the next n rows, each have n floating point numbers each, denoting an $n \times n$ matrix (A_{ij}) where, $0 \leq i, j \leq n-1$.
- The problem is to compute the trace of the matrix, that is, the sum of the diagonal elements

$$tr(A) = \sum_{i=0}^{n-1} A_{ii}$$


So, the for loops are a good choice when the number of iterations is known in advance. So, a good example of such a condition is when you program for matrices, because the dimensions of a matrices are known in advance. So, let us consider a sample problem. So, the first line of the input has a number n now the matrix size is $n \times n$ and there are n floating point numbers in the matrix given row by row, each line contains a distinct row. Now, the problem is to compute the trace of the matrix, the trace of the matrix is the sum of the diagonal elements. So, it is defined as $\sum_{i=0}^{n-1} A_{ii}$. Notice that the matrix row indexing starts from 0, similarly the matrix column indexing also starts from 0.

(Refer Slide Time: 01:34)

The image shows a terminal window with a dark background and light-colored text. It contains a C program to calculate the trace of a matrix. Below the terminal is a 3x3 grid of numbers with red annotations:

```
# include <stdio.h>
main () {
    int i, j;      /* i runs over rows, j over cols */
    int n;        /* dimension n X n of the matrix */
    int a;        /* the current matrix entry read */
    float trace = 0.0; /* sum of diag elements seen so far */

    scanf("%d", &n);      /* read dimension of matrix */

    for(i=0; i<n; i++) { /* for each row do */
        for( j=0; j<n; j++) { /* for each col of row i do */
            scanf (" %d", &a); /* read Aij into a */
            if ( i==j )
                trace = trace + a;
        }
    }
}
```

① . 2 3 Trace = 1 + 5 + 9
4 ⑤ 6
7 8 ⑨

So, let us write a c program to solve this problem. Now, you should be familiar with how we compute the trace of a matrix. So, for example, if that matrix is given as let say 1 2 3 4 5 6 7 8 9. So, the way we do it by hand is, look at the first row only this element goes into the trace. So, it is trace is 1 +, no other element of the row goes into the trace, in the second row the second element goes into the trace. So, it is 5 + and then no other remaining element goes into that trace and you go to the third row. And the third element goes in to the trace so, 1 + 5 + 9, this is how we do it by hand. You go row by row and then pick out for each row pick some element which goes into that trace only the diagonal element will go into the trace. Let us try to code this.

So, in this we have two variables i and j which I will use to iterate over the row indices and the column indices n is the designator for the size of the matrix. For example, the dimension of the matrix is **n X n**. Now, a is the variable in to which I will read the current entry and then trace is the sum of the diagonal elements seen so far. I assume that it is an integer matrix, it is not general enough you can use a float variable as well. I will first scanf the size of the matrix n, the matrix is of dimension **n X n**. Once I have done that, here is what I was talking about in the previous slide, once you scan the number n you know that the matrix is **n X n**. So, the number of times that you are going to iterate is known in advance. So, the number of times that you have to iterate is known before you enter the for loop. In such cases the for loop is more convenient to write than the while loop.

So, the outer loop is for each row from **i = 0**, to **i = n** excluding **i = n** you increment the

row. Similarly, for $j = 0$ to n you increment the column index so, j is supposed to be the column index. Now, you scan the number a now if $i = j$ remember that we wanted to add the only the diagonal elements. So, the diagonal elements will be when the row index is the same as the column index. So, when the row index is the same as the column index, you should add the corresponding numbers to the trace. So, once j becomes $n - 1$, you will fail the test $j < n$. So, we will exit out of the inner for loop, and you will go to the outer for loop. In the outer for loop you have i iterating over the row indices. So, you will go to the next row and do the same processing for the next row, until you hit row index n at which point you will exit out of the outer for loop.

(Refer Slide Time: 05:23)

```
# include <stdio.h>
main () {
    int i, j;      /* i runs over rows, j over cols */
    int n;          /* dimension n X n of the matrix */
    int a;          /* the current matrix entry read */
    float trace = 0.0; /* sum of diag elements seen so far */

    scanf("%d", &n);           /* read dimension of matrix */

    for(i=0; i<n; i++) { /* for each row do */
        for( j=0; j<n; j++) { /* for each col of row i do */
            scanf (" %d", &a); /* read Aij into a */
            if ( i==j ) |
                trace = trace + a;
        }
    }
}
```

Example i ↓

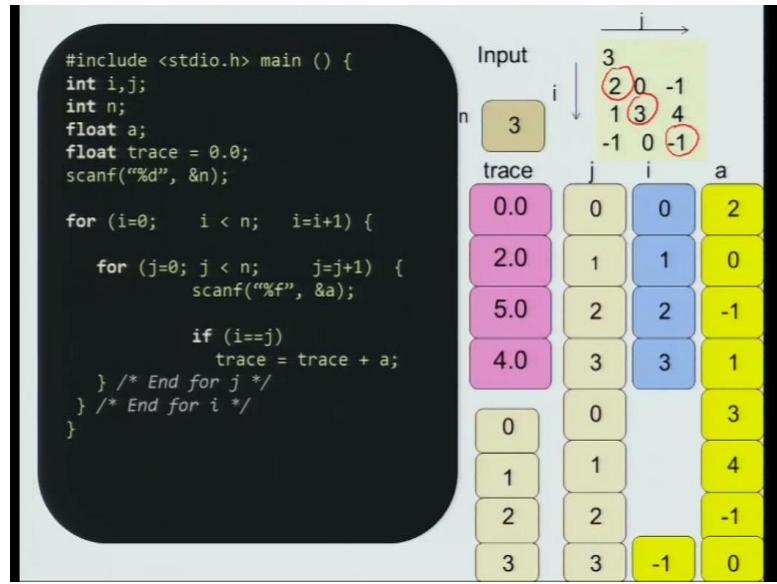
		j →
	3	
Input	1	2 3
	1	3 3
	-1	0 -1

 Tip

- No need for braces if there is a single statement in body of loop, if, else etc..
- Empty body is signified by ;

So, let us look at a sample input let say that you have 1 2 3 1 3 3 and -1 0 -1. Here, is a particular convenience that c gives you which I have used in this code. So, notice that this if block I did not put the braces. So, it could have been necessary to put the braces according to the syntax that we have discussed so, far. But if there is only a single statement in the if block then, we do not need to put the braces and it is syntactically correct to do so.

(Refer Slide Time: 06:05)



So, let us just run the program on a sample input. So, we have some sample array 2 0 -1 1 3 4 -1 0 1. So, initially there is this number 3. So, you know that it is a **3 X 3** matrix. So, once you do that you know that n is 3 so, representing that it is a **3 X 3** matrix. So, then you start with **i = 0** and go on until i less than n incrementing i by 1 each time i is 0 **i < n**, because n is 3. So, you enter the outer loop the first statement of the outer loop is itself a for loop, you start with **j = 0 j < 3**. So, you enter the inner loop you scan a number a, which is a floating point number and if **i = j**. So, remember that we are looking for diagonal elements. So, we are currently at this point and **i = 0** and **j = 0**.

So, we are entering we are scanning the zeroth element of the zeroth column of the zeroth row. So, that element has to go into the trace. So, **i = j** is true and then you say that **trace = trace + a** trace was initialize to 0 so, trace becomes now 2. Once you do that, you iterate the inner for loop. So, you go to the updates statement in a inner for loop j becomes **j + 1**. So, you go to the next column and the **j < 3**. So, you scan the next number 0 if **i = j** that is false now, because i is 0 and **j = 1**. So, you do not execute the if statement and go to the update statement. So, j becomes 2, **2 < 3**. So, you scan 1 more number which is -1, i is not j. So, you update again j becomes 3, now 3 is not less than 3, so, you exit out of the inner loop. When you exit out of the inner loop there are no more statements to execute. So, you go directly to the update statement in the outer loop which becomes **i = i + 1**. So, you are reading the first row, row number 1 you are finished reading row number 0. Again you scan the numbers when **j = 1** that is the second number in the second row, you will see that **i = j**, because i is 1 and **j = 1**. So, you will add it to the

trace. So, that is $2 + 3$ which is 5. So, trace gets updated and after you do that you scan the remaining entry in the same row, but it does not go to the trace, and then you have done with the row.

After that again you go to the outer loop you update the row index of the row index is less is 2 which is less than 3. So, you exit so, you enter the if condition and you execute the inner loop when $i = 2$ and $j = 2$ you will find an element which is -1 which will go in to the trace. So, the elements that will be added to that trace are when 2 3 and -1 . Once you are done you get out of the inner loop and then you go into the outer loop and update it, but then i becomes 3 it is no longer true that 3 is less than 3. So, you have done reading all the rows. So, you exit the program when you exit the program you have the correct trace which is 4.

Introduction to Programming in C
Department of Computer Science and Engineering

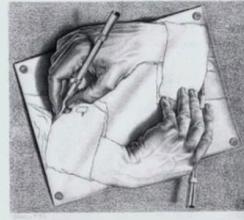
(Refer Slide Time: 00:07)

**Loopy constructs in C:
break and continue**



In this session we will see one more feature that is present in C associated with loops.

(Refer Slide Time: 00:09)

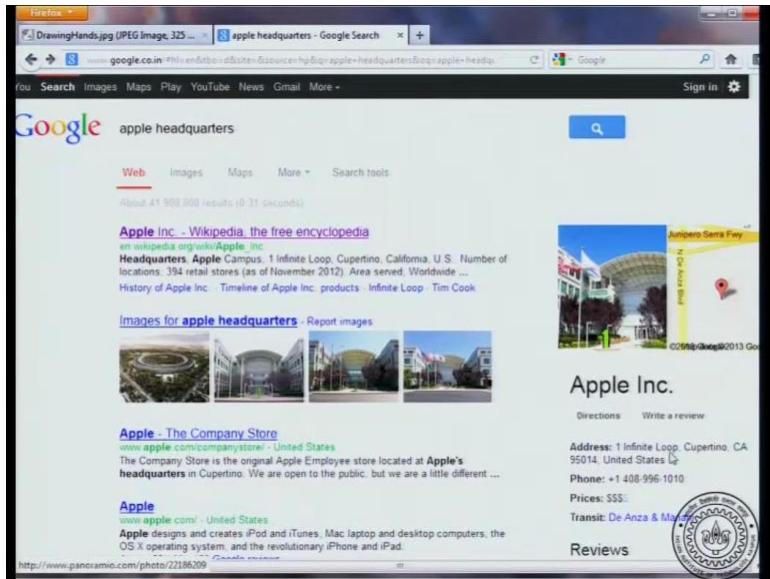


M. C. Escher, *Drawing Hands*, 1948



So, we will motivate these statements using the concept of an infinite loop. Here is a drawing that supposed to be a representation of an infinite loop.

(Refer Slide Time: 00:22)



And a trivia, for example, the apple head quarters; the address is one infinite loop.

(Refer Slide Time: 00:33)

Infinite loop

- Beware! The following is probably an infinite loop.

```
while (1)
    statement;
```

- The test_expr of the while loop is the constant 1. This always evaluates to 1 (TRUE). So if the test is executed, then control enters the body of the loop.

```
while (1) {
    printf("Hi! I am an infinite loop\n");
}
```

```
Hi! I am an infinite loop
```

So, let us see what is an infinite loop? So, the basic or the simplest kind of infinite loop is when you have a while statement. And the test condition, you can see that it will never be false. So, remember that 1 is true in C. So, this statement means that you will enter the while loop, you will test the condition. The test is true, so, you will execute the statement. You will go back and test the condition again, it is again true, does not

changed; it is 1. Therefore, you will enter the statement again. So, you will have an infinity loop.

If the test is executed then the control enters the body of the loop, and this happens without any change. So, let us look at this simple while loop which is while 1. The statement is print f Hi! I am an infinite loop. So, if you will run this code, compile and run this code after you write the main function and all that, then the program will keep on printing the same message over and over again. And you cannot exit out of the program. If you are running a Linux system you can press control c, and the program will exit immediately. But here is an infinite loop; it executes an infinite number of times.

(Refer Slide Time: 02:01)

Break from loop!

- C allows one to explicitly break out of a loop by executing the statement **break;**
- When the **break;** statement is encountered, execution breaks out of the inner-most enclosing loop body (while, do-while, for or switch) and control goes to the statement after that loop body.
- Following program reads all numbers until a -1 is read and sums them (except the last -1).

```
int a, sum = 0;
while (1) {
    scanf("%d", &a);
    if (a == -1) {
        break;
    }
    sum = sum + a;
}
printf("%d ", sum);
```

while (a != -1)

So, is there a statement which helps us to exit from a loop? Now, this is useful not just to handle infinite loops, even when you write normal loops it is important to have these constructs; they make your programming easier. So, C allows a programmer to explicitly break out of a loop using a particular statement known as **break**. When the **break** statement is encountered, the execution breaks out of the inner most loop. So, what is a loop? So far we have seen while loop, do-while loop, and for loop; later we will see a construct called **switch**.

So, whatever is the inner most loop, notice that we have talked about double loops, we have talked about the while loop within a while loop, we have talked about a for loop

within a for loop, whatever is the inner most for loop within which a particular break occurs, it will exit out of that.

So, let us write a very simple program which reads all numbers still -1 is seen and adds them up; -1 is excluded. So, you will, you can write a while loop; you have written this before where the while loops test condition was somewhat most sophisticated. Earlier we wrote something like while, if you recall, if you, we had written a loop saying, **while (! (a == -1))**. So, this was the earlier loop that we had written.

And in this case, let us write a similar program, but with a simpler test expression which is just while 1. So, you always enter the loop, no matter what number you read. So, initialize the variable sum to 0, declare the integer variable a, and then you enter the while loop because the test is true; you scan a number; and here is a use of the break statement.

If the scanned number is -1 you break out of the loop; if it is not -1, you go to the next statement which is **sum = sum + a**. So, you add the number. Again you go back to the loop; the test condition is always true, so, you enter and read the next number. So, the net effect of the loop is that whenever you see a -1, it immediately exits out of a loop; otherwise, it adds that number to the loop.

(Refer Slide Time: 04:40).

Break from loop!



- C allows one to explicitly break out of a loop by executing the statement **break;**
- When the **break;** statement is encountered, execution breaks out of the inner-most enclosing loop body (while, do-while, for or switch) and control goes to the statement after that loop body.
- Following program reads all numbers until a -1 is read and sums them (except the last -1).

```
int a, sum = 0;
while (1) {
    scanf("%d", &a);
    if (a == -1) {
        break;
    }
    sum = sum + a;
}
printf("%d ", sum);
```

5 3 2 -1

a	5	sum	0
	3		5
	2		8
-1		Output	10\$

So, let us look at using a sample input. Initially, a, is undefined; it is just declared. So, it has some garbage value. And sum is initialized to 0. Let us say that the input is 5, 3, 2, -

1. While 1, so, 1 is true; therefore, you enter the while loop; you scan f, the first number. So, a, becomes 5; a, is not -1. Therefore, you go to **sum = sum + a**. So, sum becomes 5.

And then you go back to the while loop; the test condition is still true, while 1. So, you read the next number 3; 3 is not -1. So, you add it to the sum; sum becomes 8. And you go back; the same thing occurs. So, you have the third number read which is 2; add it to the sum, and sum becomes 10. Then you read the next number; and now, a, is -1. So, what happens? The, if condition, the expression within the if statement is true; and you execute the statement inside the if condition, the statement is break.

So, recall that the rule of break says that exit out of the inner most loop. So, in particular, what is the inner most loop? You look, you starting from here, and imagine that you are going outwards towards the top of the program. The first loop that you will encounter on its way that is the loop that you will exit out of... In particular, break does not mean that exit out of the if condition; break means that exit out of the first loop that you see when you start from the statement and work outwards. So, that is this while loop. Break means you will exit out of that while loop and print this statement. So, you will print that the output is 10.

(Refer Slide Time: 07:02)

Another example of break

- Read integer `maxchars`, followed by some characters. Exit loop if either limit exceeded, or a blank line is entered.

```
#include <stdio.h> main(){  
    int maxchars; /* max number of chars */  
    int i; /* number of chars read so far */  
    char current='\\n'; /* current char being read */  
    char previous; /* previous char read */  
  
    scanf("%d",&maxchars);  
    getchar(); /* reads a character */    scanf ("%c",&—);  
  
    for(i=0; i<maxchars; i=i+1){  
        previous = current; /* store previous char */  
        current=getchar(); /* read next char */  
        if((current=='\\n') && (previous=='\\n')){  
            /* Empty Line encountered */  
            break;  
        }  
        printf("%d\\n",i);  
    }  
}
```

So, let we have been dealing with integers for a long time. Let us write a small program using characters. So, here is a problem, and let us say that we are writing a very simple editor. Now, the editor has the following property. There are a particular number of maximum characters that you can read; maybe it is 1000. So, you can type in a bunch of

characters until one of the two conditions occur; either you enter a blank line by itself which is indicating that I am done entering the text or you enter more than the maximum number of characters available.

So, recall, there are two conditions for exiting out of our so called editor; you can type a lot of characters, if your limit was 1000 and you exit 1000 then you cannot type in any more characters, and you exit. Otherwise, if you are within 1000 characters but you entered a blank line that is indicating that you are done, you have nothing more to enter, then also you should exit. So, there are two conditions. Let us try to write this code.

So, you have maximum characters. And let us say, I scan that. Then an, i, which counts how many characters I have read so far; so, i should initialize to 0. And then there is a current character, and then there is a previous character. So, I will initialize current to the new line character. Now, there is a particular reason for that which will become clear later. So, you should initialize current to a particular character.

And then what I do is, use the getchar function. So, getchar function reads a particular character from the input and stores it in some variable if you need to. Instead you can also say something like scanf **%c**, and some, into some variable. So, you can do either of these two things. And they are almost equal. So, you read one more character. Now, what should you do? You initialize by starting from 0. So, you have read no characters until now. And until you read maximum member of characters, so you execute this loop.

Remember that I said that for loop is good when you know the number of iterations in advance. So, we know that atmost we will execute maximum number of character times because that is the maximum number of characters we are allowed to field. So, for loop is slightly better than a while loop. You can also do it using a while loop if you want. So, you say for **i = 0**, i less than maximum characters, **i = i + 1**. Now, we will do this programming style that we should be familiar with right now. So, previous becomes current and current becomes the next character; so, previous equal to current. So, this will store the current character into the variable previous. Then you read the next character using getchar.

(Refer Slide Time: 10:52).

Another example of break

- Read integer `maxchars`, followed by some characters. Exit loop if either limit exceeded, or a blank line is entered.

```
#include <stdio.h> main(){}
int maxchars;      /* max number of chars */
int i;             /* number of chars read so far */
char current='\\n'; /* current char being read */
char previous;    /* previous char read */

scanf("%d",&maxchars);
getchar(); /* reads a character */

for(i=0; i<maxchars; i=i+1){
    previous = current; /* store previous char */
    current=getchar(); /* read next char */ scanf("%c",&current)
    if((current=='\\n') && (previous=='\\n')){
        /* Empty Line encountered */
        break;
    }
}
printf("%d\\n",i);
}
```

And as I said before, you can also write equivalently `scanf ("%c", & current)`. So, both these are almost equivalent that is a slight difference, but we will it is not important as of now.

(Refer Slide Time: 11:27)

Another example of break

- Read integer `maxchars`, followed by some characters. Exit loop if either limit exceeded, or a blank line is entered.

```
#include <stdio.h> main(){}
int maxchars;      /* max number of chars */
int i;             /* number of chars read so far */
char current='\\n'; /* current char being read */
char previous;    /* previous char read */

scanf("%d",&maxchars);
getchar(); /* reads a character */

for(i=0; i<maxchars; i=i+1){
    previous = current; /* store previous char */
    current=getchar(); /* read next char */
    if((current=='\\n') && (previous=='\\n')){
        /* Empty Line encountered */
        break;
    }
}
printf("%d\\n",i);
}
```

This is a sentence. I press enter. \\n

Now, if current is new line and the previous was new line, so, when will that happen? Suppose I write this is a sentence, I will explicitly represent the new line. So, when I press enter I will have a new line character here. And when will a blank line occur? When the next character is also new line. So, by a blank line what I mean is that the

current sentence is over, so I press a new line; and the next character on the next line is also a new line; that is what is actually meant by a blank line.

So, when that happens then we know that an empty line has been encountered; and here is the important thing break because one of the conditions to exit out of that loop was that either at maximum number of characters is encountered or a blank line is encountered. So, you may not have encountered maximum number of characters, but you have encountered a blank line. So, you should exit out of the proof, exit out of the loop. Again the rule is that break out of the inner most for loop, inner most loop, which in this case is just for loop. So, you get out of that loop and printf a new line.

(Refer Slide Time: 12:44)

Standard way of avoiding break;

```
#include <stdio.h> main(){
    int maxchars;
    int i;
    char current='0';
    char previous;

    scanf("%d",&maxchars);
    getchar();

    for(i=0;i<maxchars;i=i+1){
        previous = current;
        current=getchar();
        if((current=='\n') &&
           (previous=='\n')){
            break;
        }
        printf("%d\n",i);
    }
}

#include <stdio.h> main(){
    int maxchars;
    int i;
    char current='0';
    char previous;
    int flag=0; /*flag guards Loop*/

    scanf("%d",&maxchars);
    getchar();

    for(i=0;(i<maxchars) &&
        (flag != 1); i=i+1 ){
        previous = current;
        current=getchar();
        if((current=='\n') &&
           (previous=='\n')){
            flag=1;
        }
        printf("%d\n",i);
    }
}
```

Now, as with many constructs in C, you can avoid break all together. You can write code if you have used break, you can right equivalent logic without using break. So, here is a standard way to do it. So, here is the code that we just dealt with. It had 2 exit conditions - one is that the number of characters that you read is greater than the maximum allowed; another exit condition was that you had entered a blank line. So, here we used the break statement.

And now I want to write an equivalent loop without using the break statement. And here is a very standard programmatic style. These are known as flags. So, flag is just a variable which indicates that a particular condition has occurred. Initialize flag to just 0. In our code, what flag is supposed to do is that it will indicate whether a blank line has occurred or not. So, let us first look at the body of the loop; without looking at the loop

head first. Let us just look at the body of the loop. So, it is similar to what went before. Instead of the break statement, what I will do is, if I realize that an empty line has happened then I will set flag to 1; notice that flag was initially 0.

So, **flag = 1** will indicate that an empty line has been seen. Now, I will modify the loop as follows. Remember that the test condition here is just that maximum number of characters has occurred. Instead, I will check for two conditions in the for loop. I will check that maximum number of characters have not occurred, and I will also check that flag is not 1 because flag is 1 means that a new line, a blank line has been encountered. So, I will check for both these conditions in the for loop itself.

If either of them is true that is; sorry, if either of them is false that is if i is greater than or equal to maximum characters, or **flag = 1**, then the test condition will become false and you will exit out of the loop. So, here is a standard way to avoid a break. And notice that this condition is negated in the for loop because the condition in the for loop is the condition for entering the loop. So, to exit out of the loop you need **flag = 1**.

So, in summary, what I want to say is that if you want to write a code using break, you can also write it without using break. One of the standard way to do it is by using a flag variable for whatever condition that we want to check. You can pick either of this style whichever suits you more.

(Refer Slide Time: 16:35)

To break or not to!

- Use of break sometimes can simplify exit condition from loop.
- However, it can make the code a bit harder to read and understand.

- Tip: if the loop terminates in at least two ways which are sufficiently different and requires substantially different processing then consider the use of termination via break; for one of them.

- Using break; is not essential. You can write all loops using the usual while, do-while or for loops



So, how do we decide whether to use the break statement or not? Sometimes the use of the break statement can simplify the exit condition. And on the other hand, it could also make the code a bit harder to read. What do I mean by harder to read? When I see the for loop in the code on the right hand side, it is clear that there are two ways to exit out of the for loop - one is i greater than or equal to maximum characters, the other is $\text{flag} = 1$. Just by looking at the for loop, I can say that, ok, here are the two conditions for which the loop will terminate - i greater than are equal to maxchar, or $\text{flag} = 1$.

On the other hand, if you look at this left hand side code, I actually have to look at the body of the code to realize what are the ways of exiting out of loop. So, you have to understand the body of the loop in order to see what are the conditions for the loop to exit. It is not just i greater than or equal to maxchar. So, in that sense, the code with break is harder to understand than the code without break. It still recommended to use break when you have two or more exit conditions out of a for loop. So, typically programmers do use break and it is just a matter of style whether you we will use break or not; I myself prefer using a break.

(Refer Slide Time: 18:07)

Note

- break exits the loop immediately – does not go to the update expression

```
#include <stdio.h>
main()
{
    int i;
    for(i=0; i<10; i=i+1){
        if((i%2) == 1){
            break;
        }
    }
    printf("%d\n", i);
}
```

./a.out
1

not 2, since $i=i+1$ is not done after break

One final thing about the break statement; when you use break statement initially, it is important to notice that break causes an exit immediately out of the loop. So, remember when you have a for loop, the normal execution order is you initialize, then you test. So, this is step 1, this is step 2, then you execute the body of the loop that step 3, and then

you update this is step 4, and then go back to the test condition. So, this is the normal execution order of the loop.

When you encounter a break, you exit immediately out of the loop. In particular, when you break you do not go back to the update statement. So, let us examine what this code will do? You have, $i = 0$, $i < 10$; increment i . So, you start with $i = 0$; $i \% 2$ will be $0 \% 2$ which is 0. So, it will say, ok fine, you need not get into the if condition.

Then $i = i + 1$; so, i equal to 1; $1 < 10$; you enter the for loop; $1 \% 2$ is 1; so, you will break. When you break you immediately get out of a loop. So, when you print this then i will be 1. So, in particular, i is not 2, which is what will happen if you go back and update $i = i + 1$, before exiting out of the loop. So, the important thing to notice is that it is not 2, since $i = i + 1$ is not done when you break. When you break you get out the loop immediately without doing the update state.

Introduction to Programming in C
Department of Computer Science and Engineering

We have seen the break statement, which is a statement used when you are in the middle of a loop and you encounter a condition and you want to exit the inner most loop. There will also be occasions in a program, when you are in the middle of a loop and you encounter some condition and then, you realize that you do not need to execute this iteration, you just can go to the next iteration. So, skip the current iteration.

The break statement was, you encounter a condition and you say, I am done, I will exit out of the inner most loop. Here, it is not exiting out of the inner most loop, it just skipping the current iteration.

(Refer Slide Time: 00:45)

The continue; statement



For this, we will see the continuous statement and let us motivate this by an example.

(Refer Slide Time: 00:48)

Example

- continue; statement causes the next iteration of the enclosing for, while, or do loop to begin.
- E.g., read numbers skipping negative numbers until a non-digit is found.

```
int a;
while ( scanf( "%d", &a ) ==1 ) {
    if (a < 0) { /* negative number */
        continue; /* skip remainder of loop and */
    } /* go to loop test */
    /* code for processing positive a */
}
```

- scanf returns the number of conversions successfully made.



So, continue the statement causes the next iteration of the closest enclosing for while or do while loop. Let us motivate it with a very simple example. Let us say that we are reading numbers coming in a stream and what we have to do is to skip the negative numbers. So, we have to read all the positive numbers and reading should be finally over, when you encounter some input which is not a number. How do we do this?

Let us imagine that you have the main and things like that written and the central part of the code can be analyzed as follows. So, you have integer variable a and let us examine the code in closer details. So, what we need to do is, we may have an input sequence that looks like this, 1, -1, 2 and then ... So, let us say that the input sequence is something like this. What we will do is, we will do this scanf operation to get the numbers.

So, scanf operation will read the first entry as 1, it will read the second entry as 1. The third entry as -1 and the third entry is 2 and so, on. So, that is what is `scanf("%d", &a)`, we are already familiar with this. But, what does = 1 mean? So, this is something that we have not uncounted. So, far the scanf statement has a return value, it gives you the number of inputs that was successfully read.

For example, we are trying to read an integer in the %d specifier. So, when we try to read the first entry, it should succeed. So, this will succeed, when you try to read the second entry, it should succeed, when you try to read the third entry, it should succeed. In all these, the scanf %d will return a 1. Because, one entry has been read correctly. Here, it will fail, because it tries to read a natural number here, but what it see is a ., a full stop

character and that is not a number.

So, `scanf %d` will simply fail. So, this is what I said, it returns the number of conversions that have been successfully made. So, when you try to read an input like 1, -1, 2, . it was succeed in a first three `scanf` and the last `scanf`, it will fail. So, that is what the `scanf` is supposed to do. So, as long as you have read a number.

So, while you have read a number, you examine whether it is a positive number. If it is a negative number that is, if $a < 0$, then you say continue which is saying that, I do not need to execute the remaining part of the loop. So, this part of the loop will be skipped, if $a < 0$. Continue means, go from here and start executing the next iteration of the loop. Let us go head and complete the code.

(Refer Slide Time: 04:31)

- E.g., read integers until a non-digit is found and find the largest of the positive integers.

```
int a; /* current integer read */
int max = -1; /* running maximum so far */
while ( scanf("%d", &a) == 1 ) {
    if (a < 0) { continue; }
    if (max < a) { max = a; }
}
```

max = 0;
max = 1
max = 2

1 ↑ -1 2 .

↑ ↑ ↑



So, let us modify the problem as a little bit, read the integers until a non digit is found. And let us do something with the positive integers. Let us say that we have to find the largest of the positive integers. So, what should we do? Again, let us try to do it by hand to get a feel for, what I will be doing? So, I have 1 -1 2 .. Let us say that I initialize the maximum to some reasonable value. Since, we are looking at the largest of the positive integers I can initialize maximum to 0.

Then, I look at the first one, the maximum read. So, for. So, it is a positive entry.. So, I will update `max = 1`. Then, I read the next number and it is a negative numbers,. So, skip it. Then, I read the third number which is a positive number. So, I will update the maximum to 2. So, this is the part that we want to focus, if it is a negative number, skip.

So, here is the code for doing that, while the currently read input is a number, that is why the `%d` succeeded and one entry was correctly read.

So, if the number was read, check whether the number is negative. If the number is negative, continue. Continue means go to the next iteration of the loop. Do not do, what is remaining in the loop. So, if the currently read number is non negative, what you will check whether their current maximum is less than the new number. If it is less than the new number, you reset the maximum to the new number.

So, this is the code that we have written similar to other codes that we have seen. So, you update the maximum number and go and read the next number. If the currently read number is negative, then we will say continue. So, we will not update the maximum. This is what the continue is supposed to be.

(Refer Slide Time: 06:58)

- E.g., read integers until a non-digit is found and find the largest of the positive integers.

```
int a; /* current integer read */
int max = -1; /* running maximum so far */
while ( scanf("%d", &a) == 1 ) {
    if (a < 0) { continue; }
    if (max < a) { max = a; }
}
```

- Without using continue; -- adds one level of nested if

```
int a; /* current integer read */
int max = -1; /* running maximum so far */
while (scanf("%d", &a) == 1) {
    if (a >= 0) {
        if (max < a) { max = a; }
    }
}
```

Now, as in break you can also write equivalent code without using the continuous statement. So, let us try to do that and for doing that all we have do is, make sure that the maximum is updated only if it is a non-negative number. So, this says if it is a negative number, do not do the next statement. This says, if it is a non-negative number, then update maximum if necessary. So, it can be written with one more level of nested if. So, this is that if a is non-negative, then execute the next statement. Here. it says that if a is negative, then continue which means skip to the next statement.

So, notice that these two conditions are the negations of each other. The long and short of it is that continue is not really necessary. But, if you have it, then it is useful and it makes

the code clearer in certain occasions.

(Refer Slide Time: 08:10)

continue; in for loop

- In for loop, continue; causes control to jump to the update_expr of for loop.

```
for ( initialization_expression ;  
      test ;  
      update ) {  
    continue;  
    body  
}
```

What happens to continue in a for loop? Noticed that, for loop has the following form, you have for, then there is an initialization expression. Then, there was a test and finally, there was update and then, you have the body of the loop. What happens if you encounter a continue in the middle of the loop? In the case of a while loop, it is very clear, you go to the test expression, you go to the next iteration.

The only contention is, in the case of a for loop, do you go to the update statement? And the answer is yes, then you skip the remaining part of the loop. So, this is the remaining part of the loop that you would skipped. When you skip that you go directly to the update statement. Notice that, when you do the break. So, if the statement has a break, you break immediately out of the loop without doing the update. In the case of a continue, you have to do the update.

(Refer Slide Time: 09:37)

continue; in for loop

- In for loop, continue; causes control to jump to the **update_expr** of for loop.
- Useful if there is already few levels of nesting of if statements inside.



And as with the break statement, the continue statement is also redundant, you can program without using the continue statement as well. But, it is useful if whereas, already a few levels of nesting of the if statements inside it. We saw in the previous slide that, you could avoid continue statement by using an extra level of nested if statement. Now, if you do not want to complicate the code in that way, you can use a continues statements.

Otherwise, in other cases you may want to exit out of the loop, in that case you can use the break statement. So, they are extra feature that the C language provides, they are not really necessary, but they are used with.

Introduction to Programming in C
Department of Computer Science and Engineering

Let us do a sample program using continue statements, I will introduce the problem initially, the problem is that of finding Pythagorean triples.

(Refer Slide Time: 00:13)

continue - Finding Pythagorean Triples

3 4 5
$$3^2 + 4^2 = 5^2$$

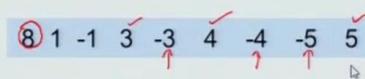




By the way Pythagorean triples are numbers like are triplets of numbers like 3, 4 and 5. Because, you know that $3^2 + 4^2 = 5^2$. So, the Pythagorean triples because there can be a right triangle, where let say the base 3, the altitudes is 4 and the hypotenuse is 5. So, 3, 4 and 5 could be the sides of right triangle, because they satisfy the Pythagorean identity.

(Refer Slide Time: 00:53)

continue – Finding Pythagorean Triples

- E.g., Read n, assume $n \geq 2$. Read n integers, and print triplets of consecutively positive input integers that are Pythagorean, skipping negative ints. For input

Output should be 3 4 5



So, here is a problem we are given a stream of numbers and let us say there are n numbers. So, the initial number says how many other numbers there are. So, 8 says that there are 8 numbers to process, after you read $n, n \geq 2$, you have to read n integers and then you have to identify Pythagorean triplets occurring consecutively. By consecutively we will say that consecutive positive integers. Because, in the middle there could be negative numbers so you have to just ignore them.

For example, you have that 3, 4 and 5 are consecutive, positive entries in this data. Because, -3, -4 and -5 are negative numbers. So, consecutive in this context need not mean that they occur together, it just means that, if we ignore the negative numbers and between them they are together. So, we have to identify all such Pythagorean triples. So, in this case the Pythagorean triple in the input sequence is 3, 4 and 5.

(Refer Slide Time: 02:24)

Code – Part I

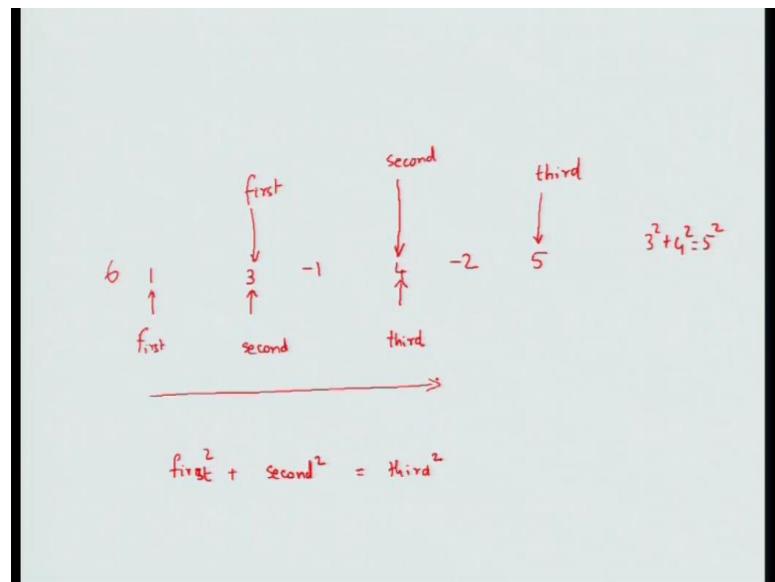
```
#include <stdio.h>
main()
{
    int curr, prev, pprev; /* current, prev, pprev positive nos. */
    int n;                 /* number of integers */
    int i;                 /* for Loop counter */
    int count = 0;          /* no. of positive ints seen yet */
    scanf("%d", &n);

    /* the Loop (continued on next slide) */

} /* End of main */
```

So, let us try to code it. I hope you see how it can be done? So, let us try to do it my hand.

(Refer Slide Time: 02:39)



So, let us say that I have and then some negative numbers in between and so on. So, some positive numbers, some negative numbers in between till I find the... So, I have let us if four, six numbers. So, the input is of the following for what I need to do is, at any point I may have to remember some triple. So for example, the first triple that I will find is the following. So, this is the first number, this is the second number and this is the

third number and what I have to do is to check whether, $\text{first}^2 + \text{second}^2 = \text{third}^2$. So, this is what I have to check?

Now, suppose that 1, 3 and 4 are not a Pythagorean triple, they are not. Because, $1^2 + 3^2$ is not 4^2 . Then, what do you have to do? You have to advance all these first, second and third variables. So, let us try to advance the third variable, the next interesting number is 5, because that is the next positive number. So, the next iteration should check for the following, this should be the third number, 4 should be the second number and 3 should be the first number.

If you do that, then you know that $3^2 + 4^2 = 5^2$ and you will identify a Pythagorean triple. So, what we do is, that we have to shift all these variables, first, second and third by one positive entry. So, this is what we have to do, we have to remember three numbers, the current number that we have seen, the previous positive number that we have seen and the previous to previous positive number that we have seen. So, this is one situation where you need to remember three variables.

And once you check whether the current triplets satisfy it, if you satisfy it fine. If you do not satisfy it, you have to advance the variables by one. So, first will take over second, second will take over third and third will go to the next positive number. So, this is the method of programming this, let us try to code this out ((Refer Time: 05:30)). So, we will write the code as follows, we need three variables, the current number, the previous number and the previous to previous number.

Currently will leave all of them undefined, n is the number of integers to read, i is we will eventually try to do. So, for the for loop we need a counter. So, I will basically count from 1 to n to ensure that n numbers have been read. I will also have an extra variable call count, i is suppose to count the numbers seen so far and count will count the positive numbers seen, so far. So, I need two half them in this code I mean in that. Now, after you do that you scan the n, which tells you how many numbers are there in the input? Now, a for loop has to go here which will do most of the work in the code. So, let us see what that loop looks like?

(Refer Slide Time: 06:48)

The Loop

```
for (i=0; i < n ; i = i+1) {  
    scanf("%d", &curr);  
    if (curr <= 0) { continue; } /* skip non-positive nos. */  
    if (count == 0) {  
        pprev = curr; count = 1;  
    } else {  
        if (count == 1) {  
            prev = curr; count = 2;  
        } else { /* count is 2 and will remain 2 */  
            if (pprev*pprev + prev*prev == curr*curr){  
                /* Pythagorean triple found */  
                printf( "%d %d %d\n", pprev, prev, curr);}  
                pprev = prev;  
                prev = curr;  
            }  
        }  
    }  
}
```

$pprev^2 + prev^2 = curr^2$

So, recall what we did by hand, you will look at the current number which is the next number to read, if the next number is 0 or less than 0 you say continue. So, this is the application of the continue statement here. So, if says if the current number is not positive, you just go onto the next iteration of the loop. Now, here is some logic which is not easy to read, but we can motivate it the following, if the current number that I have seen is the first positive number. When obviously, then this was the first number that I have read.

Therefore, there was no previous number and there was no previous to previous number. So, I will because this the first positive number that I am reading, then I will just set there the previous to previous number is the current number, also I have seen one positive number. So, I will say increment count, **count = 1**. So, if the current number that I have seen is positive and it is not the first positive number.

That means, if **count = 1** I already seen one positive number, then what to you do is, you know that there is a previous to previous number, you set the previous number to the current number and you continue the loop setting that **count = 2**, which says that I have seen two positive numbers. So, I have a previous to previous number and I have a previous number, now I will read the next number. This is because in order to identify a triple, you need at least three numbers.

So, previous to previous and previous should already been to some positive values in the input, this is why we initially said that, we need at least two inputs. So, we will go back to the loop if `count = 1`, otherwise let us say that count is at least 2, so it is 2 or more. So, in this case we will just say that as for as count is consent I do not need to keep track of how many positive numbers are needed? It was used only to see that I have at least two positive numbers to begin with. So, that I can add the next number as the possible third number in the triple.

So, I will not update count from now one, you can also do that, but count after words serves no purpose. So, I will say that count is 2 and I will just adopt the convention that it will remain to. So, I will seen at least two positive numbers, now I have also a third number in the current. So, you have previous to previous, you have previous and you have current. So, these are the three numbers that you have.

So, what you have to check is, whether $\text{pprev}^2 + \text{prev}^2 = \text{curr}^2$. So, that is what we will check, we will check whether previous to previous square plus previous square is equal to current square, if that is true then you have found the Pythagorean triple. So, you will just say that I will printf that I have found the Pythagorean triple, which is found by previous to previous, previous and current.

Now, what I will do if the Pythagorean triple is found is that I will advance previous to previous by one. So, previous to previous will become previous, previous will become current. So, recall that figure that I first true and then we will go back to the loop. So, this is the code for kind identifying the Pythagorean triples and the encodes exactly the logic that we did by hand.

Introduction to Programming in C

Department of Computer Science and Engineering

In this session, we will learn about one more fundamental data type in C. So, far we have seen ints and floats. Ints are supposed to represent integers and floats are supposed to represent real numbers. We will see the third most important data type which is character. So, it is called char in c or char.

(Refer Slide Time: 00:23)

Characters in C: char

- C allows a few more basic types than int and float.
- The **char datatype** is one byte (i.e., 8 bits) wide and can hold **exactly one** character, e.g.,
`'0'...'9' 'a' ... 'z' 'A' ... 'Z'`
`'?' '@' '#' '$' '^" '+' '-' '_' '(' ')' '='` and so on.

```
# include <stdio.h>
main() {
    char ch;
    ch= 'A';
    printf("%c\n", ch);
}
```

```
# include <stdio.h>
main() {
    char ch ='A';
    printf("%c\n", ch);
}
```

Output:

A	\$
---	----

Character constants are enclosed
in single quotes, e.g., 'A' , '0' ...

C allows a character data type to be 1 byte that is 8 bits wide, and 1 byte can hold exactly one character. For example, a character may be a digit like 0 so, on up to 9. It can be lower case letter like a up to z, it can be upper case letter like capital A through capital Z and so, on. Similarly, there are other characters question marks and sharp and so, on. So, how do you declare a character variable, how do you assign it and how do you print or scan it. So, these are the basic operations that you can do with any data type. So, you declare a character variable using the data type `char ch` will declare variable of name ch and of data type char. In order to assign it to any particular constant, any particular character, what you have to do is, you write `ch = 'A'`. So, this is how you would assign any character in constants. All the character in constants are supposed to be enclosed in this single code. For example, `'0'` stands for the character 0 and not the number 0 and similarly, a within single code stands for character a.

Now, how do you prints print a characters you can use the format specifier `%c`. So, recall that `%d` prints an integer and `%f` prints of float, we have the third fundamental data type

which is character which can be printed using a `%c`. So, if you say print f `%c` ch, it will print a. There is also an abbreviator notation where as soon as you declare the variable, you can initialize it using character ch equal to a. This is similar to saying int I equal to zero. It is the same concept.

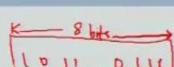
(Refer Slide Time: 02:49)

The char data type

What can we do with a variable of the char data type?

1. We can assign character constants 'A'-'Z' '0'-'9' etc. to a variable of type char. What does this mean?

1. The value of a character constant is the integer value of that character in the machine's character set, which is universally the ASCII set.
2. ASCII stands for the American Standard Code for Information Interchange. It is the standard mapping used by computers and devices today for characters.



Now, what can we do with a character data type? For example, we can assign character constants to those characters variables. Now, what does a character variable mean? Here is the first surprise. The value of a character constant is an integer that the machines represent, machine stores which is usually the ASCII set. What does this mean? The machine deals with fundamentally bits. So, you have a data field which is 8 bits wide and this is sequence of bits say 1 0 1 1 0 1 1 1.

Now, here is the bit pattern and if you see that this bit pattern is a char, then the machine takes this integer, takes this bit pattern as an integer and looks up a table known as the ASCII set table and sees which character it is. So, the value of the character constant is actually an integer. What does that integer represents? The integer represents a particular entry in an ASCII character table and what entry is in that particular location, that is the character constant. So, think of it like the following. The character is just an uninterpreted sequence of bites. If you tell the machine, please read this as an integer, it will read this as an integer. If you read this, if you tell the machine please read this as a character, it will take that integer, go look up the ASCII table and see that this integer stands for the character c and prints that. So, by itself the bit pattern can be interpreted in multiple ways.

So, here is a surprising thing which is different from natural language. There are certain natural languages where this does not typically happen with Indian languages, but there are certain languages where you have a character and how you read it depends on where you saw it. So, if it was in the middle of a text, then this is an alphabet. If you saw this in the middle of a numbers sequence, then it is a number. What happens in the machine is somewhat similar. You have a bit sequence and this thing is interpreted as a character by looking up the ASCII set. ASCII stands for American Standard Code for Information Interchange, and it is one of the popular encodings for characters used in computers. So, the code chart looks something like this.

(Refer Slide Time: 05:56)

$(76)_{10} = 7 \times 16 + 6$															
ASCII Code Chart															
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
!	"	#	\$	%	&	'	()	*	+	,	-	.	/		
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_		
.	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{ }	-	DEL		

Table entries are in hexadecimal (base 16):
8 rows X 16 columns = 128 entries.
Base 16: the symbols are 0-9, A-F, where, A stands for 10, B for 11, C for 12, D for 13, E for 14 and F for 15.
hex number 7A equals in decimal $7 \times 16 + 10 = 112$.
hex number 23 equals in decimal $2 \times 16 + 3 = 35$



You have 256 characters and characters can be looked up in a table. The table entries are in hexadecimal so, base 16. We will come to that little in the course why basic 16 is convenient,, but there are 8 rows and 16 columns in the table. So, in base 16 notation, a stands for 10, b stands for 11, c for 12 so, on up to f for 15. So, this is what is meant by base 16 notation.

So, let us look at what does the number 7 a represent. 7 a is row 7 column number 10. So, that is the number that I am interested in. What does 7 a represents? It means 7 times 16 plus 10. So, in base 10 notations, the number 76 let us say so, if I have this number 7 in base 10 notation, this; obviously, stands for the numerical values 7 into 10 plus 6. Similarly, in base 16 notation, 7 a stand for 7 into 16 plus 10. Remember that a is 10. So, you have 112 and similarly, hexadecimal 2 3. So, row 2 column 3 for example, hexadecimal 2 3 means look up 2 time 16 plus 3, the 35th entry in the table.

(Refer Slide Time: 07:51)

ASCII Code Chart															
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0 NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1 DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2 !	"	#	\$	%	&	'	()	*	+	,	-	.	/		
3 @	0	1	2	3	4	5	6	7	8	9	:	;	<	=	> ?
4 @	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5 P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	-		
6 .	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7 p	q	r	s	t	u	v	w	x	y	z	{ }	-	DEL		

The first 32 characters (rows 1 and 2) with ASCII code 0–31 correspond to special characters (functions) and are not printable. Code x20 (decimal 32) corresponds to the space character. Code x21 (decimal 33) corresponds to the "!" character, etc.

Now, here is the structure of the ASCII code set that you use in c, the first 32 characters basically from 0 0 hexadecimal to 1 f hexadecimal. So, these 32 characters which are shaded, are what are known as special characters, and they are not printable. They are required by the computer for certain special purposes. Code 2 0 that is decimal 32, 2 0 is 2 times 16 plus 0. So, this particular entry corresponds to the space characters. So, this is just a blank space. Code 21 corresponds to the exclamation character and so, on.

(Refer Slide Time: 08:40)

ASCII Code Chart															
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0 NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1 DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2 !	"	#	\$	%	&	'	()	*	+	,	-	.	/		
3 @	0	1	2	3	4	5	6	7	8	9	:	;	<	=	> ?
4 @	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5 P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	-		
6 .	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7 p	q	r	s	t	u	v	w	x	y	z	{ }	-	DEL		

1. Printable characters: ASCII codes 32 onwards till 126.
 2. Upper case 'A' - 'Z' have consecutive ASCII codes 65–90.
 3. Lower case 'a'-'z' have consecutive ASCII codes 97–122.
 4. Digits '0'-'9' have successive ASCII codes 48–57.

So, the printable characters in the ASCII code are hexadecimal 20, that is decimal 32 until 126. So, what is enclosed in the green parenthesis, these are all printable characters. Now, out of this, the capital letters start from x 41 which is 65 in decimal and go on up

till decimal 90. Small letters start from 97 and go on till 122 and so, on digits 0 to 9 occur before any character. So, why we need this information? This is how the characters are stored in the computer and do we really need to know it?

(Refer Slide Time: 09:46)

ASCII Code Chart

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
!	"	#	\$	%	&	,	()	*	+	,	-	.	/		
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^ _			
~	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{ }	-	DEL		

What do we (C Programmers) need this table for?

There are some ideas behind the design of the table that C uses. There is no need to memorize.

© Original Artist. Reproduction rights reserved. www.CartoonStock.com

The point is not that you have to memorize this table. You do not need to memorize the table, but you need to remember certain abstract properties of the table. We will make that precise in a moment. We do not have to say that the ASCII code for a is 65 or 42 that is a waste of our memory. So, let us just see what we can do with this table without really remembering what that table looks like. So, there are some ideas behind the design of the table, how the table is structured which c programmers can use. There is no need to remember that a particular character has a particular ASCII value.

(Refer Slide Time: 10:37)

Character constants

A character constant is an integer, namely the ASCII code for that character. The following two code segments are equivalent, that is, ch gets the same value.

```
char ch;  
ch = 'A';
```

```
char ch;  
ch = 65;
```

- A character is printed as a character using the %c option in printf. scanf reads a character using %c option

```
char ch;  
ch = 'A';  
printf("%c ",ch);  
printf("%d", ch);
```

Output: A 65

- Printing a character as an integer prints its ASCII code.



So, let us just recall. A character constant is an integer, namely the ASCII code for that character now which means that I will emphasize this with a very strange code. I can declare character ch and say **char ch = 'A'** that; obviously, initializes the character to a. It assigns the value a to the variable ch, but I could also do the following characters **ch = 65**. Why 65? The ASCII value for a was 65. So, instead of writing it as a within single code, I can write **ch = 65**, and it will be the correct ASCII character anyway. Now, this means that the same character can also be interpreted as an integer if you really want to think of it that way.

So, for example, I can say **%f %c ch** if I do it in print f, it will print it as. So, the first print f will print a, but I could also take a character variable and ask c to print it as an integer using **%d**, it will print 65. So, remember that the external form that we see in some sense is the letter a. The internal representation is the number 65 because 65 is the entry in the ASCII table corresponding to the character a.

(Refer Slide Time: 12:22)

The slide has a light gray background with a dark gray sidebar on the right. The title 'Using arbitrary characters' is at the top. A bulleted list follows: 'Any 8-bit character with hexadecimal representation xhh can be specified as '\xhh''. Below the list is a code block in C:

```
# include <stdio.h>
main () {
    char bellch, vtabch, ch;
    /* sounds a bell when printed */
    bellch = '\x7';
    /* prints vertical space when printed */
    vtabch = '\xb';
    /* hex 41=decimal 65: ASCII code for 'A' */
    ch='\x41';
    printf( "%c%c%c", bellch, vtabch, ch );
}
```

To the right of the code is a green rounded rectangle containing '\$./a.out' and 'A\$'. Below it is a yellow rounded rectangle with the text 'first a bell rang! then a tab was printed, followed by 'A''. At the bottom right is the IIT Bombay logo.

Now, one more thing is that you can print arbitrary numbers, even non-printable characters you can sort of print them using c and one way to do that is I can print any 8 bit character with a hexadecimal representation like \s, \x followed by the hexadecimal to digit. For example, \x followed by 7 is the bell character. So, let me go back a couple of times, couple of slides. So, if you look at the 7th entry in the ASCII table, it is represented as bell. It is a small bell in your system. So, if you ask the system to print the 7th character in the ASCII table, what will happen is that your computer will make a small beep sound. So, there are certain non-printable characters which can also be printed directly using... ok.

Similarly, let say \xb is the 11th number in the ASCII table, it is a vertical space. So, if you print that character, it prints a vertical space. Similarly, if I ask it to print hexadecimal 41 using \x41 so, x 41 is $4 * 16 + 1$ which is $64 + 1 = 65$ and we just saw that ASCII value 65 was the character a. So, if I ask it to print ch which is hexadecimal 41 as a character, then it will print the value a. So, when you run this program, what it will do is, first because you ask it to print a bell character, it will beep once, it will ring the bell and then, it will print the second character which is a vertical space. So, it will print a vertical space and then, the third character was a printable character a, it will print a. So, you can ask the system to print arbitrary entries in the ASCII table. If it is a printable character, it will print that corresponding character. If it is non-printable character, it might take a suitable action.

(Refer Slide Time: 15:07)

Escape Sequences in C	
Escape sequences refer to character constants with special meaning. They start with a '\' followed by a single character. For e.g., '\n' refers to the newline character. It looks like two characters but represents only one.	
Escape sequence	Meaning
\a	Alert (bell)
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
Escape sequence	Meaning
\\	Backslash
\?	Question mark
'	Single Quote
"	Double Quote
\xhh	hexadecimal number xhh
\ooo	Octal number
\0	NULL character



So, just for information sake, instead of printing it as \x followed by the x code, c provides certain escape characters, some special sequences as well in order to print these non-printable characters. First of all until now we have seen one such number which is \n. So, \n is the new line character. It is a non-printable character, but it corresponds to some ASCII corrected. Similarly, for the other non-printable characters, c has some escape characters. For example, back slash a is the bell character and so, on.

Introduction to Programming in C

Department of Computer Science and Engineering

In the previous session, we were talking about ASCII character set. And I said that, we do not need to remember the ASCII table. But, we need to remember some general properties of the ASCII table.

(Refer Slide Time: 00:15)

ASCII Code Chart															
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
!	"	#	\$	%	&	,	()	*	+	,	-	.	/	
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
.	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{		}	-	DEL

So, what are those general properties? First, we know that the initial 32 characters of the ASCII table are non printable characters. Then, the remaining or rather from ASCII value 32 to ASCII value 126 are printable values. Among them, you know that the integers, the digits are occurring together. Similarly, the capital letters occur consecutively, one after the other. And the small letters occur consecutively, one after the other.

So, this is an abstract property of the ASCII code chart that, helps us in writing some useful code. We will see in a minute, what kind of use we can obtain using these general properties. Rather, than knowing the specific things like, the character value of A is hex value 41 or decimal value 65. This kind of information, we need not remember this.

For example, let us write a small program that prints... In our previous session, we had said that, we do not need to remember the exact ASCII code of certain characters. We just need to remember some abstract properties of the ASCII table. For example, abstract properties like all the digits occur together from 0 to 9. All the capital letters from A to Z

occur together in the table, in the alphabetical order.

Similarly, all the small letters occur together in consecutive locations in the ASCII table. Also, another property that you can observe is that, the small letters occur after all the capital letters. Let us see, how we can write some interesting code using these properties. And not by remembering the exact ASCII code of certain letters.

(Refer Slide Time: 02:28)

Print the Alphabet



- The ASCII codes of upper case letters are consecutive and in the English alphabet order. Same is true for lower case letters and digits.
- E.g., to print upper case letters in alphabetic order:

```
main () {  
    char ch;  
    for ( ch='A'; ch<='Z'; ch=ch+1 ) {  
        printf( "%c",ch );  
    }  
}
```

Output ABCDEFGHIJKLMNOPQRSTUVWXYZ\$



So, let us write a simple program, to print the alphabet. The ASCII codes of the upper case letters are consecutive and the ASCII codes of the lower case letters are consecutive. This is the property that, we will exploit in order to print the alphabet. So, for example, let us say that, we are going to print the letters of the alphabet in capital letters. So, for that we can use the following program using a for loop. So, what you have to do is, to initialize a particular character variable to capital letter A so, the ASCII character A. So, note that A within single quotes stands for the character constant A.

If you look at the integer value, then it is the ASCII code for A. We are not particularly interested to know, what exactly the number is. Now, we can write the for loop in an interesting way. We can say that, start from capital A and then, print the characters until you hit capital Z. And the update statement is, after printing go to the next ASCII letter. So, what this is doing is, starting from A and then, it will go to **A + 1**, which is the ASCII code for B.

Then, it will go to **B + 1**, which is the ASCII code for C, so on up till Z. So, once you reach Z, it will print that character. It will update once more, where it is the ASCII character one more than, the ASCII character next to Z in the ASCII table. We do not really need to know, what it is. But, certainly it will be greater than the ASCII value of Z and at that point, we will exit the code. So, the output of it will be consecutively A to Z.

(Refer Slide Time: 04:38)

```
char ch;
for ( ch='A'; ch<='Z'; ch=ch+1 ) {
    printf( "%c", ch );
}
```

Output

ABCDEF	GHIJK
LMNOP	QRSTU
VWXYZ	\$

1. Characters are stored as 8 bit integers.
2. They can be assigned as integers, incremented, decremented etc.,
3. Suppose 'A' has ASCII code 65. ch = 'A' sets ch to 65.
4. ch = ch+1 sets ch to 66.
5. printf("%c",ch) prints ch as a character, this prints 'B'.

Relational operations <, >, >=, <= are defined on chars by comparing their integer representations (ASCII codes). So 'A' < 'B' since 'A' is represented as 65 and 'B' as 66.



Let us look at, what is happening here in greater detail. All the characters are stored as 8 bit integers. Now, they can be assigned as integers, incremented, decremented, etcetera because, essentially they behave like integers. So, suppose A has ASCII code 65, but we are not concerned about that. Now, so ch equal to character constant A, sets ch equal to 65. Now, **ch + 1** is the number 66, which corresponds to the ASCII code of B.

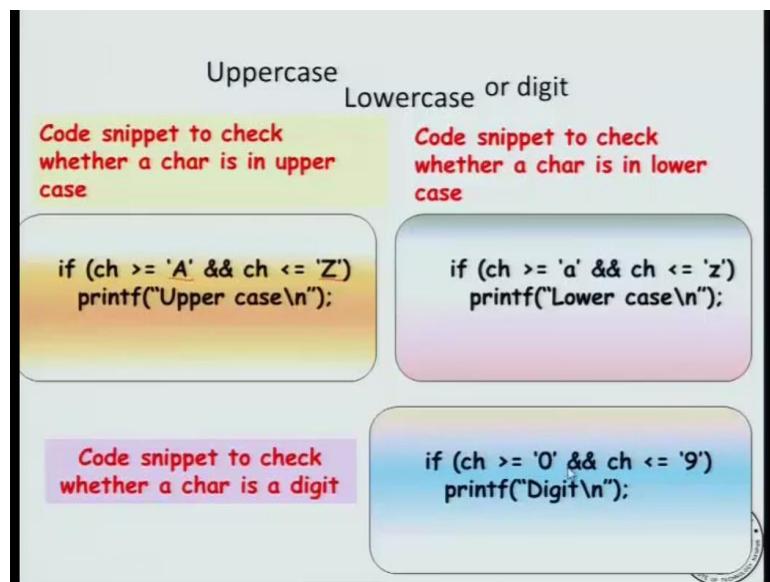
So, addition, subtraction all these can be performed on character values because, internally they are represented as 8 bit integers. Similarly, relational operations like less than, greater than, **<=, >=**, all of these also make sense. So, for example, if we use the relational expression capital letter A, ASCII constant A less than character constant B. Then, notice that A is the ASCII value 65 and B is the ASCII value 66. So, A less than B is correct.

Now, for realizing that A less than B is correct, we do not need to know that, A is 65 and B is 66. All we know is that, the abstractly in the ASCII table, the character code for A is less than the character code for B because, B occurs after A. So, if it is 65 and 66 or it

is 0 and 1, the answer is still the same.

Now, let us write a few more interesting programs, where the spirit is that, we do not need to understand what the exact ASCII code of a letter is. But, just we want to remember the layout of the ASCII table.

(Refer Slide Time: 06:45)



For example, suppose I want to write a conditional expression an if condition, which says that, if the given character is capital letter, then print that, it is in upper case. So, all I need to do is, if the character value is \geq the character constant A and \leq the character constant Z. Then, you print that, the given letter is in upper case. Again, please remember that we did not need to know that, this was 65 and this was, whatever it is 90.

It could as well have been 0 and 25. It would still have worked because, all we are need to remember in the ASCII table is that, A through Z occurs in consecutive locations in the standard alphabetical order. From that we can understand that, if I write this if expression, it will print up the message upper case, only if the given character ch is an upper case letter. Similarly, let us say that, if you want to check whether a character is in lower case. You can analogously write, character is \geq 'A', 'a'. And it is \leq little z, in single quotes. If that is true, then you print that, it is in a lower case. Now, if you want to check whether a given character is a digit, similarly you can say that, it is \geq the character 0. And this is \leq the character 9. Now, here is a subtle point which I hope, you notice.

The character 0 is the ASCII constant, ASCII character constant 0. So, it corresponds to some particular ASCII value. It is different from the number 0. So, this is something that... So, we are looking for the ASCII value corresponding to 0 it is \geq that and \leq the character value corresponding to that character 9. So, if that is true, then the given character is a digit.

(Refer Slide Time: 09:10)

if ch is a lower case letter of the English alphabet, then it converts it to upper case. Otherwise it does nothing

```
if (ch >='a' && ch <='z') {
    ch = ch -'a' + 'A';
}
```

ch = 'b'; ch = 'a'; 'a' = 100
 'A' = 65,
 ch = ch - 100 + 65;

ch = ch - 100 + 65;



Now, here is a snippet that, I would advise you to take a look at it. And tell me, what it actually does. So, take a moment yourself and try to figure it out. So, what it does is, the given character ch, if it is a lower case letter. Remember, this is the example that we just saw. This condition checks, whether the given character is a lower case letter, a small letter between a and z, little a and little z. If it is true, then what you do is, add capital A - a, to the character.

So, what does it accomplish? So, let us say that, we actually had c h equal to little a. Now, for the purpose of illustration let us say that little a, was ASCII value 100. I do not know, whether that is true. But, it is not important. That is, what I want to illustrate. Now, what does capital A represent. It represents some ASCII value let us say 65. So, if the given character was little a, what I would do is, I would say character = ch - 100 +65

Similarly, if ch was character constant b, I would still add ch = ch - 100 +65 So, it is adding a constant difference to the given character regardless of what, whether it was a or b. The additive constant that we are adding is still the same. And if you think about,

what is happening it is adding, exactly the difference between little a and capital A. Notice, that the difference between little b and capital B is the same as little a and capital A.

Why? Because, all the capital letters occur consecutively and all the small letters occur consecutively. So, suppose a minus z, little a minus capital A is... Let us say 35, then little b minus capital B will also be 35, because you advance one in each case. So, if you think for a minute, you will see that what this code does is... Take the ASCII code corresponding to the small letter. And add a constant difference. What is that difference? That difference is, what will take you to the capital letter, corresponding capital letter.

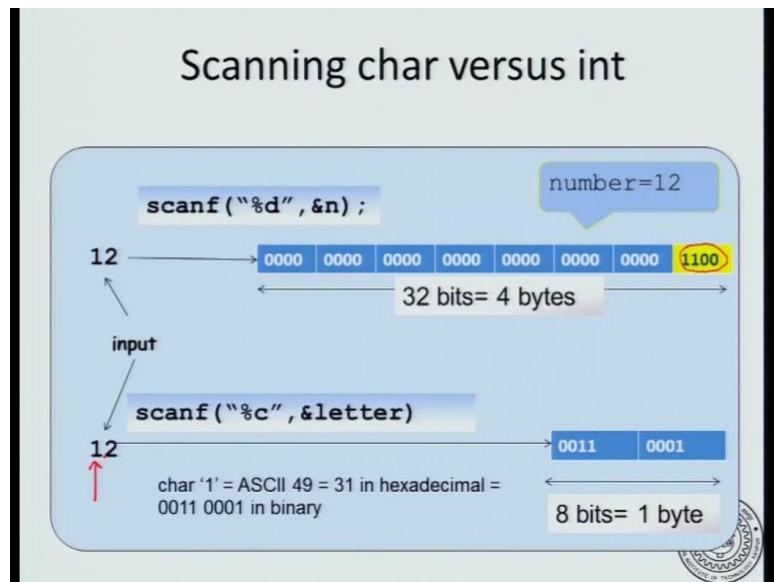
(Refer Slide Time: 12:30)

if ch is a lower case letter of the English alphabet,
then it converts it to upper case. Otherwise it does
nothing

```
if (ch >='a' && ch <='z') {  
    ch = ch -'a' + 'A';  
}
```

So, in short what this does is, to convert the given character in lower case letter to an upper case letter. So, if it is a lower case letter, it will convert it to an upper case letter. Otherwise, it does nothing. Now, let us think about, what we mean by scanning a character verses scanning an integer.

(Refer Slide Time: 12:49)



So, let us say that I have two variables, in number and character letter. So, let us say that I am scanning now *n*, which is a number and the input was 12. So, when I say `scanf("%d", &n)`, I am trying to read 12 into an integer variable. Now, on a typical machine an integer variable may occupy 32 bits. or 4 bytes. So, it has 32 bits in it. And if you know the binary notation, you will see that 1100 in binary is the number 12.

So, this is $8 + 4$, which is 12. So, when you see that, the input is 12. And then, I ask in the C program, I am doing `scanf("%d", &n)`. What will happen is that, *n* is an integer with 32 bits wide. And it will have the following pattern encoded into it. This is what, it means by scanning an integer. And if you try to print it out, it will try to interpret *n* as a decimal number. And it will print and the digit 12, here the number 12.

On the other hand, for the same input, here is the difference I want to emphasize. If the code was saying, `%c` and *letter*, so scan the input 12 using the `scanf` statement, `scanf %c` and *letter*. What will happen is that, the C program is looking at the first character, which is the digit 1 and scanning it in. Now, character 1 is ASCII 49 it is not important, you remember that. But, it has some ASCII value, and that ASCII value 49 is 31 in hexadecimal because, it is $3 * 16 + 1$, which is $48 + 1 = 49$. So, that character 1 is 31 in hexadecimal. And hence, what will be stored? Remember, that a character ASCII character is 8 bits wide. So, it will store 31 in hexadecimal. So, it will be 3. This is the number 3 and this is the number 1. So, when you scan the input into a character variable

called letter. What will happen is that, letter will have the number, hexadecimal 31 or ASCII value 49, which corresponds to the number, which corresponds to the character constant 1.

So, this is the difference between scanning a given input as a number and scanning a given input using a character. So, when you scan it using a number, this entire thing will be scanned. It will be converted into binary and you will store it in an integer variable. When you scan it as a letter, it will scan the first digit only because, that is the character and then store the ASCII value, inside the letter variable, inside the character variable. So, this corresponds to the letter variable 1 within a single quote, the character constant 1.

Introduction to Programming in C

Department of Computer Science and Engineering

In the session, we will discuss operators and expressions. So, we have already used C expressions in our programs before.

(Refer Slide Time: 00:09)

C expressions

- We have used expressions in our programs before. Expressions in C are close to our notion of mathematical expressions.
- Expressions are the basic unit of evaluation. They return a value (of a type).



And expressions in C are similar to expressions in mathematics and they follow the same rules, similar to what mathematical expressions also follow. They are a basic unit of evaluation and each expression has a value. Say, that an expression returns a value of a particular type.

(Refer Slide Time: 00:38)

C expressions

```
int a = 3; int b = 4; int c;
c = (a*a) + (b*b);
```

- The right hand side of the assignment operator =, that is, $(a*a) + (b*b)$ is an expression. It is made up of sub-expressions $a*a$ and $b*b$.
- Expressions are made up of constants and/or variables—these are the “atoms”. They then get combined using operators, unary (e.g., - or !) or binary (+ or - or * etc.).



So, let us consider a few example expressions. For example, I have the following, `a` is 3, `b` is 4 and I have a variable `c`, which is just declared to be of type A. And then, say that `c = (a * a) + (b * b)`. So, the right hand side of the assignment operator, this is a assignment operator. And the right hand side is an expression and that expression has sub expressions, `a * a` and `b * b` within parenthesis.

So, an expression can be made up of variables, it can be made up of constants. These are the atoms or the basic components of an expression. And sub expressions can be combined into bigger expressions, using operators. Now, operators can be unary that is, they take one argument operation. For example, on unary operators the examples can be `-`, which is the unary `-`. For example, `-3` is a negative number. Similarly, NOT operator that we have seen in connection with logical operations so, NOT of zero, for example, the logical negation operator. Both of these operations take one argument. Now, there is also the binary operations like `+`, `-`, `*`, etcetera. So, `+` takes two arguments. For example, an expression like `2 + 3` and here is the binary `-`. So, if I say `2 -3`, this is actually a binary operator which takes two arguments, which are 2 and 3.

Similarly, the binary multiplication `2 * 3` would be the product of 2 and 3. So, notice the difference between... It is the same sign for the unary `-` and the binary `-`. But, the unary `-` takes only one argument and the binary `-` takes two arguments. We have used the assignment operation many times and let us understand that in, somewhat more detail.

(Refer Slide Time: 03:12)

Assignment Operator

- We have used the assignment operator many times.

```
int a;
int b = 4;
int c;
b= b+1;
```
- E.g., `b = b+1`; above. If we remove the semi-colon ; then we get the expression
`b= b+1`



For example, if you consider the expression `b = b + 1`. Now, if you remove the semicolon

at the end. So, the statement is **$b = b + 1$** semicolon. And if you omit the semicolon, what you get is an assignment expression, **$b = b + 1$** without the semicolon.

(Refer Slide Time: 02:28)

The Assignment Operator

- Here is an example expression involving = operator. Assume a and b are defined to be of type int.
 $a = (\underline{b = 10})$
- Recall: assignment operator assigns to the left operand (which must be a variable) the value of the expression on the RHS and then returns that value (and type).



So, how does the assignment operation work? For example, consider an expression like **$a = (b = 10)$** . What does this do? So, assume that a and b are integer variables. Now, assignment assigns to the left hand variable, left hand operand, the value of the expression on the right hand side. For example, in this assignment operation there are two assignment expressions.

One is the expression **$b = 10$** . And the second is the expression **$a = b = 10$** . So, the first assignment expression is supposed to do the following, assign 10 to b. So, what it does is, it assigns the value of the right expression which is 10 in this case to the left hand side operand that is one thing, it does. And also, it returns the value after the assignments. So, 10 has been assigned to b. And the return value of this expression is 10.

(Refer Slide Time: 04:47)

The Assignment Operator

- Here is an example expression involving = operator. Assume a and b are defined to be of type int.
 $a = (b = 10)$
- Evaluation of $a = (b = 10)$ is done as follows.
 - First evaluate the expression in parenthesis.
 - This expression ($b=10$) has an = operator. Evaluate the expression on its RHS. Its value is 10 (constant) and of type int. So b gets the value 10. And the expression ($b=10$) evaluates to 10, this is returned.
 - The original expression now reduces to $a = 10$. This is evaluated as usual, a gets the value 10. The value 10 is returned.



So, we can now analyze $a = b = 10$, as follows. First, evaluate the expression in parenthesis. The expression $b = 10$ has an assignment operator. So, evaluate the expression on the right hand side and then assign it to b. Now, that operation returns. So, that expression returns a particular value, which is 10. Now, the original expression can be thought of as, just $a = 10$. This is evaluated as usual.

So, you take 10 and assign it to a. And the return value of the whole expression becomes 10. So, when executing this expression, when evaluating this expression, two variables are assigned their values. One is b, which is assigned the value 10 and the second is a, which is also assigned the value 10.

(Refer Slide Time: 05:49)

Assignment =

- Most commonly used to initialize a number of variables to the same value (similar to mathematics). Example

```
int a,b,c,d,e;  
a=b=c=d=e=0;
```
- There is ambiguity here since we did not put brackets. Should the assignment be treated as
 $((((a=b)=c)=d)=e)=0$
or should it be
 $a=(b=(c=(d=(e=0))))?$

$\frac{\text{LHS}}{\uparrow \text{var}} = \frac{\text{RHS}}{\uparrow \text{var/const/expr}}$



The assignment operation can be used to initialize a number of variables, in one shot. For example, if I write a statement like $a = (b= (c = (d= (e=0))))$. What does this mean? Now, here there is some ambiguity here, because we do not know which order to evaluate this. Should, we evaluate from left to right. Should, we evaluate from right to left. Does it matter?

So, should the assignment be treated as the following, where $a = b$ is done first, then, $= c$, then, $= d$ and. So, on. Or should it be the opposite way right to left, where $e = 0$ is first done. Then, $d =$ that, then $c =$ that,. So, on until a.

(Refer Slide Time: 06:39)

Evaluation from right for =

- Consider expression
 $a = (b = (c = (d = (e = 0))))$
- This assigns e,d,c,b,a successively to 0, which is the standard mathematical convention. Why?
 1. The expression in the inner most parenthesis is evaluated first. This is $e = 0$. It assigns e to 0 and returns 0.
 2. The next outer expression is the expression $d = (e = 0)$. $e = 0$ has been evaluated and the result is 0. So d is assigned to 0, and the expression returns 0. So on ...



So, the expression is evaluated from right to left, in the case of the assignment operation. For example, the above expression that we just saw will be done as, $a = \dots$ So, $e = 0$, first and then backward, until a is assigned. Now, this is also the standard mathematical convention. We are not introducing a new strange rule, here. Why is this? First, we will evaluate the inner most expression, which is $e = 0$.

So, e will be assigned 0, then the return value of this sub expression. So, this sub expression will return the value 0. So, this becomes $d = 0$, d is assigned the value 0. And the return value of this sub expression becomes 0. So, then we have $c = 0$ and. So, on. So, finally, every variable here will be assigned the value 0. So, the reason for doing this is that, if you try to do it in the opposite way, you will see that uninitialized variables are initialized to other uninitialized variable.

For example, if you go from left to right, in the previous. This simply does not make any

sense, because you have just declared a b c and. So, on. And when you say $a = b$, a and b are not initialized yet. So, this assignment hardly makes any sense. The basic rule of assignment is that, left hand side = right hand side. So, the left hand side is some value that can be assigned to.

For example, this is a variable. The right hand side can be anything, variable, constant or it can be an expression. So, all these are valid assignment. So, what is an invalid assignment? So, $a = 0$ can be a valid assignment but, $0 = a$. So, the assignment operation is evaluated, right to left.

(Refer Slide Time: 09:11)

Associativity of Operators

- C defines the assignment operator = to be right-associative. That is,

$$a=b=c=d=0$$
 has the same meaning as

$$a=(b=(c=(d=0))).$$
- Operator binary + is left associative. That is,

$$a + b + c + d$$
 is evaluated by grouping from the left as

$$((a+b)+c)+d$$
- Associativity of an operator tells the order in which to evaluate operators if there are multiple occurrences of identical operators in an expression.



Now, we have the concept of associativity of operators. So, what does associativity mean? It is, we have just argued that, $a = b = c = d = 0$. An expression like that will be evaluated from right to left. So, it is as though, we have parenthesized the expression as $d = 0$, inner most. Then, $c =$ that, then $b =$ that and. So, on. So, on the other hand, if you take an operator like binary + the addition symbol, then the usual custom is that you parenthesis from left to right.

So, the evaluation is done, $a + b$ first. Then, that sum is added to c. Then, that is added to d. So, the assignment operation goes right to left. The addition symbol operates left to right. So, this concept of associativity of an operator tells us, the order in which we evaluate the operations, if there are multiple occurrences of the same operator. So, the first there are multiple occurrences of the = sign. In the second, there are multiple occurrences of the addition symbol..

So, associativity rules tells you that, if there are identical operators in an expression, in which order do you evaluate them? Do you evaluate them from left to right? If you do, then it is called a left associative operator. If you evaluate from right to left, in the case of, for example, the assignment, then it is called a right associative operator.

(Refer Slide Time: 11:00)

Binary – is also left associative

- $a - b - c - d$ is evaluated as $((a-b)-c)-d$.
`printf("%d", 10-5-15);`
Output -10
- Evaluated as $((10-5)-15) = -10$.

 General principle: C defines an associativity for EACH operator of C. Let us see a part of the table.



Binary - is also left associative. For example, $a - b - c - d$ is evaluated as $a - b$, then c then $- d$. So, for example, if you say $10 - 5 - 15$, what will be done is $10 - 5$ and then $- 15$. So, this is $5 - 15$, which is $- 10$. Whereas, if the parenthesis had been in the opposite way, it would be $10 - 5 - 15$, which case it could be $10 -$, this is $- 10$ which is 20 .

Notice that, this is not how you are supposed to do it, even in mathematics. So, the way that C handles the associativity of the binary operation, is correct. So, the correct parenthesis is $10 - 5$ and then $- 15$. In general, for every operator c defines an associativity. So, let us see the part of the associativity of operations in C.

(Refer Slide Time: 12:23)

Operator type	Operator	Associativity	
Parenthesis	()	Left to right	
Boolean Not, unary -	! -	Right to left	
Multiplication, division, remainder	* / %	Left to right	
Add, Subtract (binary)	+ -	Left to right	
Relational comparison	< <=	Left to right	
	> >=		Keep this table handy while programming. Useful to know precedence among common ops.
Equality comparison	==	Left to right	
Logical AND	&&	Left to right	
Logical OR		Left to right	
Assignment	=	Right to left	



There are several operations that we have seen,. So, far. The parenthesis, the Boolean naught, the logical naught and the unary -, the binary multiplication, division and. So, on. Addition symbol, comparison less than, less than or = and. So, on. Equality, logical AND, logical OR and then the assignment operator. We have seen, all these operations,. So, far. And of this, the typical associativity is left to right.

There are couple of exceptions, one we have already seen. Assignment operation is right to left. The unary operations are also right to left. Most of the other operations are left to right. So, if you think for a little bit, you can see that the associativity for unary operations is also easily seen to be right to left. That makes more sense.

So, the idea is not that you should memorize this table but, you should understand. Given the table, can I understand, what will happen with an expression? How c will evaluate it? It is not that, you should remember this. But, rather if you are given the table and an expression, can you correctly calculate what the value of the expression will be.

Introduction to Programming in C

Department of Computer Science and Engineering

This one more concept that we have to understand, before we really understand how C evaluates expressions, that is the concept of precedence.

(Refer Slide Time: 00:12)

Precedence

- Let us think of ourselves momentarily as the C compiler. Suppose we are given the expression
 $a = b + c$
- How would we know whether the expression should be evaluated as
 $(a=b)+c$
or as
 $a = (b+c) ?$
- To decide such situations, C also defines another notion: that of **precedence BETWEEN operators**.



So, what do we mean by a precedence? Let us pick a expression which involves multiple operators. Like for example, in this expression you have two operations, the assignment operation and the addition operation. Now, how do we know, how to evaluate this expressions. So, what are the two ways in which the above expression can be interpreted, the first way is you could say $a = b$ and then say $+ c$ or you can say $a = b + c$.

To decide which of the above possibilities to really do, C also defines what is known as a precedence between operators. So, we have already seen in the notion of associativity which is what happens, when the many occurrences of the same operator occur in an expression. Precedence on the other hand is to mediate between two different or multiple different operations in the same expression.

(Refer Slide Time: 01:30)

Example

- The $+$ operator is given more precedence than the $=$ operator. So
 $a = b + c$
is evaluated as $a = (b + c)$ $\times (a=b) + c$
- The binary $+$ and binary $-$ operators have the same precedence, both associate left to right.
- Addition and Subtraction have lower precedence than multiply $*$ and divide $/$ and modulo $\%$ which have equal precedence and associate left to right. So
$$\begin{array}{c} a + b - c * d \% e / f \\ \uparrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\ \text{is evaluated as} \\ (a+b) - ((c * d) \% e) / f \end{array}$$

Operations: $+$ $-$ $$ $\%$ $/$*
Precedence $$ $\%$ $/$*
 $+$ $-$

$$(a+b) - (((c * d) \% e) / f)$$

So, let us see what is an example of using precedence. So, in c the $+$ operation is given more precedence than the $=$ operation. So, I really want to interpret this expression as $a = b + c$. So, this is how I want to interpret their operation and not as $a = b + c$. So, I want to avoid this and I want to do it in this way. So, one way I can ensure that is by saying that, please do $b + c$ first, then take that result and assign it to it.

So, one way of doing that is to say, whenever $=$ and $+$ appear together give more importance to $+$, do that first. So, proceed means going first. So, $+$ has a greater precedence over assignment $=$ and $-$ have the same precedence and both have the same associativity we have seen this, addition and subtraction have the same precedence, but multiplication and division have a higher precedence.

So, if I want to evaluate a complicated expression, let us say that $a + b - c * d \% e / f$. So, suppose I have a fairly complicated expression, what I can do is, what are the operations here? So, the operations are $+$, $-$, $*$, $\%$ and $/$. Now, according to the precedence I know that $*$, $\%$ and $/$ have equal precedence about $+$ and $-$.

So, I know that these operations have to be done before $+$ and $-$. So, they have a lower precedence. So, these have to be done first, but among them how do you know which to do first, for that we use the left to right associativity of these operations. So, as far as with in the same precedence is concerns, let us just simplify this situation in a little bit and think of them as the same operation, their different operations of the same precedence.

But, I will just ((Refer Time: 04:26)) the thing a little bit to say that, let say that they are the same operation, all of them have left or right associativity. Therefore, I will according to the associativity rule I will do $c * d$ first, then that $\%$ e and then that $/ f$, because that is what the left to right associativity it says. So, by precedence we will know that these three operations have to be done first, among them how do you do this, $*$ occurs first when you scan from left to right. So, $c * d$ has to be done first and then the $\%$ operation and then the $/$ operation.

So, and once you done there then you come to $+$ and $-$. So, currently once we have finished with this, you will have some situation like this, $c * d \% e / f$ and then on the remaining side you have a $+ b -$ this. And now you have to decide, which may you will do the $+$ and $-$, again we know that they have the same precedence.

So, let us found a little bit and think of them as the same operator, both of them have the left or right associativity. So, I do $a + b$ first and then the $-$. So, with in the same precedence level, you will decide which operation to do first purely based on the left to right associativity.

(Refer Slide Time: 06:16)

Operator class	Operator type	Operator	Associativity
	Parenthesis	()	Left to right
Unary	Boolean Not, unary -	! -	Right to left
Arithmetic	Multiply, divide, remainder	* / %	Left to right
	Add, Subtract (binary)	+ -	Left to right
Comparison	Relational comparison	< <= > >=	Left to right
	Equality comparison	==	Left to right
Logical Operators	Logical AND	&&	Left to right
	Logical OR		Left to right
Assignment	Assignment	=	Right to left
Comma	Comma	,	Left to right

So, let us take a look at the precedence of associativity table. Again I want to emphasize is not to memorise, it is just that if you are given this table, you should be able to understand how an expression is going to be evaluated? So, parenthesis is above all because once a parenthesis an expression, then you really saying this is the order that I want. So, it over writes any other precedence or associativity rule.

Then, you have the unary operations which have the second higher precedence, then the arithmetic operations, then the comparison operation, logical operators, assignment and so, on. The comma is an operation we will see later. So, with in the arithmetic operations multiply, divide and % operator have higher precedence over + and -. + and - have higher precedence over relational operations, like <, < or = and so, on. So, we will see a few examples of how to use this table to understand what will happen with an expression?

(Refer Slide Time: 07:28)

Operator type	Operator	Associativity	Examples
P R E C E D E N C E	Parenthesis	()	Left to right
	Boolean	! -	Right to left
	Not, unary	-	
	Multiplication	* / %	Left to right
	on, division, remainder		
	Add,Subtraction (binary)	+ -	Left to right
	Relational comparison	< <= > >=	Left to right
	Equality comparison	==	Left to right
	Assignment	=	Right to left
	Comma	,	Left to right

Operations : = + * %
Precedence * %
+ =

$a = 10 + ((5 * 4) \% 2)$



So, let us take $10 + 5 * 4 \% 2$ and assign to a, let us examine what will happen here. So, what I will do is I will make a list of operations. So, they are $=, +, *, \%$ and then precedence I know that multiplication and $\%$ have very high precedence. Then, the next level is $+$ and then assignment has the least precedence.

Now, both of these occurring in this expression $*$ and $\%$, how do we decide which goes first, both of them have left to right associativity. So, whatever happens first in the looking from left to right, we will do that first. So, among all these operations we know that $5 * 4$ will happen first, then this will be followed by $\% 2$ and then this will be followed by $10 +$. And finally, the last which is that you do all these operations get the value and assign it way. So, this is the way in which the above expression will be evaluated. So, the above expression corresponds to giving the parentheses in the wave that we have done.

(Refer Slide Time: 09:16)

	Operator type	Operator	Associativity	
P	Parenthesis	()	Left to right	
R	Boolean Not, unary	! -	Right to left	
E	Multiplicati on, division, remainder	* / %	Left to right	
C	Add,Substra ct (binary)	+ -	Left to right	
E	Relational compariso n	< <= > >=	Left to right	
N	Equality compariso n	==	Left to right	
C	Assignmen t	=	Right to left	
E	Comma	,	Left to right	

Examples

- $a = 10 + 5 * 4 \% 2$
- Evaluated as $a = (10 + (5 * 4 \% 2))$
- Why? The operators are = * and %. We have to evaluate operators in order of precedence.
- Among these * and % have the highest equal precedence (see table). Since they have left to right associativity, we evaluate $5 * 4 \% 2$ as $(5 * 4) \% 2$. Then we evaluate the binary addition.
- Finally we evaluate the assignment operator (lowest in precedence).



So, once you do that a will get the value 10.

Introduction to Programming in C

Department of Computer Science and Engineering

Let us see a few more examples of expression evaluation in C; what kinds of expressions are allowed, what kind of errors do people usually make, and so on.

(Refer Slide Time: 00:14)

Operator Type	Operator	Associativity	
Parenthesis	()	Left to right	• Example: Evaluate the expression
Boolean Not, unary -	! -	Right to left	<code>int a = 1,b = 1,c = 2;</code>
Multiplication, division, remainder	* / %	Left to right	<code>a <= b && b >= c</code>
Add,Subtract (binary)	+ -	Left to right	<i>Operations: <= && >=</i>
Relational comparison	< <u><=</u> > <u>>=</u>	Left to right	<i>Precedence: <= >= &&</i>
Equality comparison	==	Left to right	<i>(a <= b) && (b >= c)</i>
Logical AND	<u>&&</u>	Left to right	<i>① ② ③</i>
Logical OR	<u> </u>	Left to right	
Assignment	=	Right to left	



Let us say that we have given an expression $a = 1$, $b = 1$, $c = 2$. And then we have an expression $a < b$ and then $b >= c$. So, this is the expression that we want to see how it will be evaluated. So, let us just go through it systematically. The operations on are $<=$, then we have the logical AND operation the $>=$ symbol. Of these, the relational comparison operations $<=$ and $>=$ – have greater precedence over the logical AND. So, the precedence will be AND. And among operations of the same precedence level, we have left to right. So, whatever happens first when looking from left to right will be evaluated first. So, these two operations have the same precedence. So, we will have $(a <= b)$; then $(b >= c)$; these have to be done first and then AND. So, this will be done first, this will be done second, and this is the third operation. Conceptually, using just precedence and associativity rules, this is how the expression should be evaluated.

(Refer Slide Time: 01:51)

Operator Type	Operator	Associativity	
Parenthesis	()	Left to right	• Example: Evaluate the expression
Boolean Not, unary -	! -	Right to left	<code>int a = 1,b = 1,c = 2;</code>
Multiplication, division, remainder	* / %	Left to right	<code>a <= b && b >= c</code>
Add,Subtract (binary)	+ -	Left to right	• Answer: Relational Operators <code><=</code> and <code>>=</code> have higher precedence than binary logical operators <code>&&</code> and <code> </code>
Relational comparison	< <= > >=	Left to right	• Expression is evaluated as $(a \leq b) \&\& (b \geq c)$
Equality comparison	==	Left to right	equals 1 $\&\&$ 0
Logical AND	&&	Left to right	equals 0
Logical OR		Left to right	
Assignment	=	Right to left	

IIT Madras Logo

So, when we evaluate it, $a == b$ is 1 $<$ are $=$ 1. So, that is 1. $b \geq c$ is 1 $>=$ 2. So, that is 0. So, this becomes 1 and 0; in which case, it is 0. Now, let us look at a few tricky examples.

(Refer Slide Time: 02:14)

Operator Type	Operator	Associativity	
Parenthesis	()	Left to right	• What are the values of a and c after the following if statement is run?
Boolean Not, unary -	! -	Right to left	<code>int a,b = 2, c;</code>
Multiplication, division, remainder	* / %	Left to right	<code>if (a == b > 1) { c=1; }</code>
Add,Subtract (binary)	+ -	Left to right	<i>Operations : > =</i>
Relational comparison	< <= > >=	Left to right	<i>a = b > 1 will be evaluated</i>
Equality comparison	==	Left to right	<i>as a = (b > 1)</i>
Logical AND	&&	Left to right	<i>a = 1</i>
Logical OR		Left to right	<i>has value 1.</i>
Assignment	=	Right to left	<i>if (1) { c=1; }</i>

IIT Madras Logo

So, if you have an expression of the following form, if $a == b > 1$; then $c = 1$. So, let us see what happens here. We will do the same thing; operations sorted by precedence is... There is greater-than symbol, which has a higher precedence over the equal-to symbol. So, the expression $a == b > 1$ will be evaluated as $b > 1$, because that has higher

precedence. So, this goes first. And then $a = b > 1$. Now, b is 2. So, $b > 1$ is 1. So, you have $a = 1$. And $a = 1$ is an assignment expression. It assigns the value 1 to a. And the return value is 1 because a is assigned to 1.

(Refer Slide Time: 03:44)

Operator Type	Operator	Associativity	
Parenthesis	()	Left to right	• What are the values of a and c after the following if statement is run?
Boolean Not, unary -	! -	Right to left	<code>int a,b = 2, c;</code>
Multiplication, division, remainder	* / %	Left to right	<code>if (a = b > 1) { c=1; }</code>
Add, Subtract (binary)	+ -	Left to right	• Consider $(a = b > 1)$.
Relational comparison	< <= > >=	Left to right	Operators are = and >. > has higher precedence than =.
Equality comparison	==	Left to right	• So grouping is $a = (b > 1)$
Logical AND	&&	Left to right	1. Simplifies to $a=1$ (b is 2, 2 > 1)
Logical OR		Left to right	2. a=1 assigns a to 1 returns the value assigned which is 1.
Assignment	=	Right to left	3. Now body of if is executed and c is 1.

a
c

So, then this whole if expression becomes if $1 - c = 1$; in which case, we know that, $c = 1$; that statement will be executed.

(Refer Slide Time: 03:59)

<ul style="list-style-type: none"> previous slide: intention was probably to assign a the value of b and check if this value is > 1 $a = b > 1$ $\text{if} ((a = b) > 1) \{ \dots \}$
<ul style="list-style-type: none"> Above is a common idiom (common usage) in C. $a = (b > 1)$
<ul style="list-style-type: none"> E.g., read all integers from the terminal (Control-D marks end of terminal) until a -1 is read.
<ul style="list-style-type: none"> Note: <code>scanf(...)</code> returns the number of operands successfully read.
<pre>int a; while ((scanf("%d",&a)>0 && !(a == -1)) { /* do something */ }</pre>
<ul style="list-style-type: none"> Note: <code>&&</code> evaluates its left operand first. If left operand evaluates to FALSE, the right operand is not evaluated. This is used in above. a is checked only if scanf has read a new integer a.

Now, typically, what is expected... The typical programming style is to say something like a assigned to b; and if that result is > 1 . So, we may want to deliberately violate the

precedence. How do you do that? So, C does it some way; if you do not parenthesize it, you can always change the order of evaluation in C by introducing parenthesis, so that the meaning is very clear. So, if you do not parenthesize it, then $a = b > 1$ is the same as $a = b > 1$. But, what if you really want to do $a = b$ and then that > 1 ? So, that case, you parenthesize it. Why? Because parenthesis has the highest precedence. So, whatever is within parenthesis will be evaluated first. So, $= b$ will be evaluated first and b is 2. So, a will get the value 2. So, the assignment $a = b$ will have returned the value 2. And 2 is > 1 . So, it will execute ((Refer Time: 05:17)) One particular way in which such an expression can be seen; we have already seen such an example is – you read all integers from the terminal until a **-1** is read.

(Refer Slide Time: 05:36)

- previous slide: intention was probably to assign a the value of b and check if this value is > 1
 $\text{if } ((a = b) > 1) \{ \dots \}$
- Above is a common idiom (common usage) in C.
- E.g., read all integers from the terminal (Control-D marks end of terminal) until a -1 is read.
- Note: `scanf(...)` returns the number of operands successfully read.

```
int a;
while ((scanf("%d", &a) > 0) && !(a == -1)) {
    /* do something */
}
```

3 2 -1 ..

- Note: `&&` evaluates its left operand first. If left operand evaluates to FALSE, the right operand is not evaluated. This is used in above. a is checked only if `scanf` has read a new integer a .



So, suppose the input is of the form 3 2 **-1**; and then let us say dot or something of that sort. So, what this expression does is `scanf` returns a value, which is the number of tokens that – number of inputs that, it was successfully able to read. So, if you try to read a character as an integer, it may not succeed. And so, as long as you have correctly written the integer and the integer is not **-1**, then you do a particular ((Refer Time: 06:19)) So, this is the kind of expression that is often used; where, you assign some value to a using the assignment statement. Or, maybe you want to check the return value of a function whether it is positive or not. And based on that, you want to write a condition. So, the logical and operation does operates in the following way. It evaluates the left operand first. If this condition is false, then you know that, the whole expression is going to be

false. If at least one of the terms is false, then you know that, the whole thing is false. So, it will not even evaluate the second operand.

On the other hand, if the operation is true, then it will check whether the second operand is true. If the second operand is also true, then the whole expression is true. If the second operand is false, then the whole expression is false. This method of evaluation is also called short-circuiting because it may not evaluate the whole expression in order to get the result. So, if I know that, this expression is false; then there is no need to evaluate this, because I know that, the whole expression is going to be false.

(Refer Slide Time: 07:44)

The slide has a title 'Infinite loop' in bold at the top right. On the left, under the heading '• A risky test:', there is a code block:

```
int a = 2;
while (1 < a < 5) {
    printf("%d\n", a);
    a = a+1;
}
```

To the right of the code, a yellow sticky note-like box contains the output of the program:

```
3
4
5
6
7...
```

Below the code and output, there is a bulleted list:

- (1 < a < 5) is evaluated as $((1 < a) < 5)$ since < associates left to right.
- 1 < a is either 0 or 1, both are less than 5, so the condition is always TRUE (1).
- Better to write the test as $(1 < a \&\& a < 5)$

In the bottom right corner of the slide area, there is a circular logo of the Indian Institute of Technology (IIT) Kharagpur.

Here is a common mistake that people do, because this is similar to mathematical notation. When you want to check a condition that a is between 1 and 5; what happens if you right $1 < a < 5$? Because this is the way we do it in mathematics. C will apply the precedence and the associativity. In this case, it is the same operation. So, only associativity applies. And according to associativity, it is left to right. So, this will be evaluated as $1 < a < 5$. Now, a is 2. So, $1 < a$ is false. So, this becomes 0. So, the whole thing is $0 < 5$. So, it is true. So, if you execute this code, it will eventually become an infinite loop, because this is an expression that always evaluates to true. Now, what you probably mean is that, I want to check that, a is between 1 and 5; a is 2. So, the correct way to write such an expression would be $1 < a$ and $a < 5$; that will check the betweenness condition. So, notice that, this is different from the way we normally write in

mathematics. This is how we would write such a test in mathematics. But, that will cause an infinite loop. This is because C will apply the precedence and the associativity rules and not what you think it should do.

(Refer Slide Time: 09:35)

	high Operator type	Operator	Associativity	
P	Parenthesis	()	Left to right	• Is this expression legal? If so evaluate it. int a = 5, b=6, c = 4 ; c = a=b% c- a = a+1;
R	Boolean Not, unary -	! -	Right to left	• Syntax error! Won't compile!
E	Multiplication, division, remainder	* / %	Left to right	• Why? Highest priority is b%c, this is $6\%4 = 2$. Expression becomes $c = a = (((b\%c) - a) = (a + 1))$
C	Add,Subtract (binary)	+ -	Left to right	• The next highest priority is – and both same, and associates left to right.
E	Relational comparison	< <= > >=	Left to right	• $2 - a$ is -3 , $a + 1$ is 6 . Expression becomes $c = a = -3 = 7$
D	Equality comparison	==	Left to right	
N	Assignment	=	Right to left	
E	Syntax Error!			
■ = associates from right to left, so expression becomes $c = (a = (-3 = -7))$. LHS of = must be a variable.				

Now, let us look at can there be expressions, which make no sense? We have seen several examples, where you can always make sense out of it. So, let us take this expression. Again, list out the operations; see you have $= =$; then you have the $\%$ operation, which is highest precedence; then you have minus; then you again have an $=$; and then you have a $+$. So, these are the operations in the expression. So, what needs to be done first? $b \% c$. And then you have $-a$; and then you have a $+ 1$. This is by following precedence and associativity rules.

Now, we come to the assignment statement. Assignment statements are done right to left. So, the first thing that you would try to do is the following. So, you try to do the... So, here is a sub expression; here is a sub expression; here is a sub expression; and here is a sub expression. So, it is like assigning four terms. And the innermost thing will be done first; the rightmost thing will be done first. So, the rightmost assignment is $b \% c -a$ is assigned to $a + 1$. Now, this is a syntax error. So, what happen is as we just discussed if you work out the whole assignment; if you workout the whole expression, it becomes something like this. And somewhere when you work out the assignment from right to left, you will see that, it is trying to assign a number -3 to -7 . That does not make any

sense. The left-hand side of an assignment statement should be an assignable value, which is essentially a variable. And in this case, you are trying to assign a number to another number, which does make sense. So, here is a syntax error.

(Refer Slide Time: 11:57)

Comma operator: used in for statement

- Comma as an operator is a binary operator that takes two expressions as operands.
$$\boxed{\text{expr1} , \text{expr2}}$$
- Think of **,** as just like **+** or **-** or ***** or **/** or **=** or **==** etc.. Some examples,
1. i+2, sum=sum-1
2. scanf("%d",&m), sum=0, i=0
- Execution of **expr1, expr2** proceeds as follows.
- Evaluate **expr1**, discard its result and then evaluate **expr2** and return its value (and type).



We will conclude the discussion on operations with one more operation, which is quite common in C; which is the **;** operator. Now, this is not very common in mathematics. But, let us just discuss what does it mean in C. So, let us say that, we have two expressions: expression 1 and expression 2 separated by a **;**. Now, think of the **;** as an operation just like any other operation like **+** or minus. So, it must have a precedence it must have an associativity and so on. So, what will happen when we have an expression like **i + 2 ; sum = sum -1**. So, how does it follows? First, you evaluate the expression 1. So, first, in this case, you evaluate **i + 2**; then you evaluate **sum = sum -1**; and return the value of the lost expression. So, the whole – the **;** operation is involved in an expression called the **;** expression. Every expression has a value and the value of the **;** expression will be expression 2.

(Refer Slide Time: 13:22)

Comma operator: used in for statement

- Comma as an operator is a binary operator that takes two expressions as operands.
 $\boxed{\text{expr1}, \text{expr2}}$
- Think of , as just like + or - or * or / or = or == etc.. Some examples,
1. $i+2, \text{sum}=\text{sum}-1$
2. $\text{scanf}(\%d, \&m), \text{sum}=0, i=0$
 $\boxed{\text{scanf}(\%d, \&m)}, \underbrace{\text{sum}=0}_{0}, \boxed{i=0}$
- Execution of **expr1, expr2** proceeds as follows.
- Evaluate **expr1**, discard its result and then evaluate **expr2** and return its value (and type).



So, what if you have multiple expressions? You figure out what is the associativity of the ; expression. The ; expression associates left to right. So, this expression will become `scanf` and so on; `sum = 0; i = 0`. So, this... For the first ;, this is expression 1 and this is expression 2. So, this expression evaluates to the result of `sum = 0`; which is 0 as we know. Now, the second level is you have `0; i = 0`. So, the first ; expression is evaluated and its result is expression 2 of that expression, which is a value of `sum = 0`, which is 0. So, the outer expression becomes `0; i = 0`. The value of that expression is the value of expression 2 in that bigger expression, which is the value of `i = 0`. So, here is how you will apply the rule that, it is the value of the second expression for a more general expression involving multiple commas.

(Refer Slide Time: 14:46)

Comma Operator execution

- Take for e.g., the expression `sum=0, i=0`.
 - First evaluate the expression `sum =0`, which assigns 0 to sum and returns 0. This value is discarded.
 - Then the expression `i=0` is evaluated. This assigns 0 to i and returns 0. This value is returned (of type int).
 - In the for statement `for (sum=0, i=0; ...) {...}` it has the same effect as `sum=0; for (i=0;...) {...}`

So, what you do is – first, evaluate the first expression and it has some value. For example, in this case, it is an assignment expression. So, it will have value 0. And then the second expression is evaluated. And the value of ; expression is the value of the second expression. Note that, you may... At first sight, you may see multiple commas in the same expression; but the way you do it is that, you group them using associativity rules into a sequence of ; expressions, where each ; expression has exactly two terms. This is what we did in the previous example. Now, ; expression is very convenient, because you can do things like when you want to initialize multiple variables in a for loop for example, you can just say `sum = 0, ; i = 0`. It will initialize both values at the same time; both variables at the same time.

(Refer Slide Time: 15:49)

Comma Operator execution

- Commas are evaluated from **left to right**. That is, `scanf("%d",&m), sum=0, i=0` is executed as `(scanf("%d",&m), sum=0), i=0`
- The comma operator has the lowest precedence of all operators in C. So
 - `a=a+5, sum = sum + a`
is equivalent to
`(a=a+5), (sum = sum + a)`
No need for explicit parentheses.

```
int a = 1;
1 int sum = 5;
a=a+5, sum = sum + a;
```



So, ; are evaluated left to right. This is what I just worked out an example of the following form. So, if you have multiple sub expressions in a ; expression; if we have multiple ;, what you do is you associate them just like you did with + and star; you have multiple ; expressions. And then group them two at a time. So, it becomes two ; expressions. And then evaluate them. Now, the ; expression has the lowest precedence of any operator in C. So, if you have an operation like `a = a + 5 ; sum = sum + a`, what will happen is you do this expression `a = a + 5`; then do this expression `sum = sum + a`. And then evaluate the ; expression. And therefore, when you have a ; expression, you do not need explicit parenthesis, because the precedence takes care of it; it has the lowest precedence. So, it will never get swallowed into a bigger expression, which involves other operations. So, it will always be evaluated at the end.

(Refer Slide Time: 17:01)

	Operator type	Operator	Associativity
high	Parenthesis	()	Left to right
	Boolean Not, unary -	! -	Right to left
	Multiplication, division, remainder	* / %	Left to right
	Add, Subtract (binary)	+ -	Left to right
	Relational comparison	< <=	Left to right
		> >=	
	Equality comparison	==	Left to right
	Assignment	=	Right to left
low	Comma	,	Left to right

So, just to remind you, here is the table once again. And notice that, as we discussed the ; operation is the lowest precedence and the associates left to right.

(Refer Slide Time: 17:10)

Comma Operator vs Separator

- The symbol , is used both as an operator and as a separator.
- We met the separator version earlier in multiple declarations and/or initializations.
`int sum =0, i=0, j=0;`
- Also used in functions e.g., scanf and printf for separating operands
`scanf("%d%d", a, b); printf("%d %d", a, b);`
- The use of comma here is as a separator—not as an operator. In function calls (e.g., scanf, printf), variable definitions and initializations.
- Always clear from context whether comma is being used as a separator or as an operator.



This is also a slightly different meaning of the ; in C. We will just mention that in passing. There is also the normal separator. So, the separator can be seen in multiple occasions in C. When you initialize an expression; when you say `sum = 0, ; = zero, ; j = 0;` this is not the ; expression; it is just a separator as in English. So, similarly, when you call a function, you have ; to separate out the arguments. That does not mean that, the

arguments are inside a ; expression. Here ; is just a separator as in English. And it is always clear from the context whether a ; is a separator or an operator. As an operator, it has a particular value; as a separator, it does not do anything other than saying that, this first and then this. So, we have seen several operators in C and discussed the concepts of precedence and associativity. And what is important is – given the precedence and the associativity tables, can you understand an expression; see whether it is a valid expression, and if it is a valid expression, what will be its value.

Introduction to Programming in C

Department of Computer Science and Engineering

(Refer Slide Time: 00:11)

Functions

In this session we are going to introduce a new concept of programming in C called functions. So, initially, let us just try to motivate why we need functions, and then we will try to see whether programming becomes easier, if we have functions.

(Refer Slide Time: 00:26)

Why use functions

- Break up complex problem into small sub-problems. (Top-down design!)
- Solve each of the sub-problems separately as a function, and combine them together in another function.
- The main tool in C for modular programming.
- We have seen functions before
 - main() is a special function. Execution of program starts from the beginning of main().
 - scanf(...), printf(...) are standard input-output library functions.

So, let us say that, why do we need functions? There are essentially two different reasons for it. And I will mention these reasons one after the other. The first reason is to break up a complex problem into simple sub problems. All of us, for example, like to drop to do

less saying that these are the things I wish to accomplish today. So, step 1, you know, get to college, step 2 - attend classes, step 3 - finish home work or something like that. And then each of those main task will have several sub task. In order to get to college, maybe you need to renew the ticket subscription, get on the bus, get to college, and so on.

So, each of those higher level task involves several search smaller sub task. And conceptually, it is cleaner to say that these are the big level things that I want to do. Each of those big level task have several sub tasks, so that I can think of it, what I want to accomplish in a layer wise manner. So, this is something that we do intuitively. We always break up complex problem into simpler sub problems so that we can analyze the simpler sub problem and perform it completely, and then come back to the bigger problem. So, we need to solve it each separately.

And the main tool for this programming in C which allows you to accomplish breaking up a complex sub problem into simpler sub problems is what is known as functions. So, this enables you to do what is known as modular programming in c. And functions are not new. We have already seen three functions in particular - main was a function that we always wrote, and then we have printf and scanf which we use for outputting and inputting respectively. So, let us just motivate the notion of functions by using the second motivation that I was talking about.

(Refer Slide Time: 02:48)

$$C_n^k = \frac{n!}{k! (n-k)!}$$

```

main()
{
    int a, b, c;    float result;
    a = n! * k!
    b = k! * (n-k)!
    result = a / (b * c);
}

```

So, suppose, you have, you want to say, I want to compute $\binom{n}{k}$, which is $\frac{n!}{k!(n-k)!}$, correct?

So, this is the definition of n choose k for $\binom{n}{k}$ as it is known. Now, suppose I want to write this, n code this into C program, so I will have, let us say, a main function. And then inside the main function I will have, let us say, three variables – a, b, c, and then float result because the result of a division will be a float. So, I will have, what should I do intuitively, one way to do it is I will have a block of code which says it will calculate $n!$ which is the numerator, then I will say that $a = n!$; at end of this, let us say, that a stores $n!$.

Then, I will have another block of code which says that I will calculate $k!$. And then this will say, let us say, $b = k!$. And the third block of code will calculate $n - k!$; let us say, I will store this in c. And then I will say, ok result $= \frac{a}{(b*c)}$, some code that looks like this.

And you would notice what is inconvenient about it; all these three blocks of code, once we complete it, will look very similar. They are all calculating the! of a particular number.

But, there is nothing in c, which will, that using the features that we have seen so far, which will tell us that this code, this code and this code are essentially the same, and I need to write that similar code only once. So, there is no simple way to use loops to accomplish these. So, it seems like this redundant business of writing similar code again and again can be avoided. So, this is the second motivation for introducing the notion of functions which is basically to avoid duplication of code.

(Refer Slide Time: 05:39)

So, here is a side benefit of functions, avoid code replication. We have already seen loops

Why use functions

- Break up complex problem into small sub-problems. (Top-down design!)
- Solve each of the sub-problems separately as a function, and combine them together in another function.
- The main tool in C for modular programming.
- We have seen functions before
 - `main()` is a special function. Execution of program starts from the beginning of `main()`.
 - `scanf(...)`, `printf(...)` are standard input-output library functions.

– Avoid code replication

to some extend avoid code replication. But, here is a newer method to avoid code replication in a greater unit. So, the second reason why we write functions is to avoid writing similar code again and again.

So, let us try to write functions by motivating it with the help of an example. This example will show the benefit of how we can avoid code duplication using functions, and also how we can breakup a complex problem into simpler sub problems. So, in this I will introduce the problems similar to what have we seen before.

(Refer Slide Time: 06:41)

An example

- Problem: first line of input is n —the number of numbers to follow in the next line. Count the number of successive pairs of numbers that are relatively prime (i.e., gcd is 1).
- Example Input

8	4	6	16	7	8	9	10	11
---	---	---	----	---	---	---	----	----
- Relatively Prime Pairs are:

16	7	7	8	8	9	9	10	10	11
----	---	---	---	---	---	---	----	----	----
- The pairs 4 6 and 6 16 are not relatively prime.
- Answer 5

We have a sequence of numbers. The first number tells you how many inputs there are. And then what we need to do is to pick out the numbers which are relatively prime in these sequences. So, two numbers are relatively prime if their gcd is 1. So, 16 and 7 are relatively prime; 4 and 6 are not because they have a common factor of 2; 6 and 16 are not, they have a common factor of 2; 16 and 7 do not have a common factor other than 1; 7 and 8 are similarly relatively prime; 8 and 9 are relatively prime; 9 and 10 are relatively prime; and 10 and 11 are relatively prime.

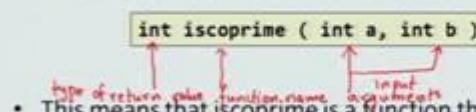
So, these are the relatively prime pairs. And we need to write a function which given a sequence of these numbers count how many pairs, how many successively occurring numbers are relatively prime to each other. In this case there are 5 such pairs.

(Refer Slide Time: 07:43)

Design the program

- Suppose we write a **function** `iscoprime(a, b)` that takes two positive integers `a` and `b` and returns 1 if `a` and `b` are co-prime, and returns 0 otherwise.
- The function declaration in C would be

`int iscoprime (int a, int b)`


- This means that `iscoprime` is a function that takes two integer arguments, the first one called `a`, the second one called `b`. The function itself returns an integer value.

So, in this problem we can clearly see that there is a sub problem which is, given two numbers are they relatively prime? That is one sub problem. And if we have the solution to that sub problem then we can compose the solution to the whole problem as follows. Given two numbers, I check whether they are relatively prime. If they are relatively prime I will increment the count of the relatively prime pairs I had seen so far, otherwise I will skip to the next pair and see whether they are relatively prime. So, for each new pair of numbers I am seeing that is the sub task of checking whether they are relatively prime.

So, let us say that suppose we have a function; a function is something that we will see in

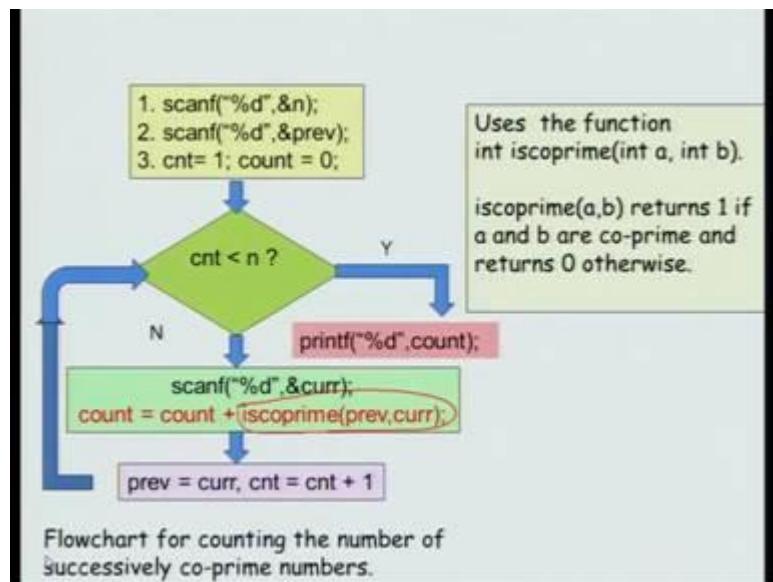
a minute. Suppose we have a small component which will perform the task of testing iscoprime a, b. So, iscoprime a, b, that function will take two numbers a and b and check whether they are relatively prime or not. If a and b are relatively prime it evaluates to 1. It is, we say that it returns 1 if they are relatively prime; and if they are not co-prime to each other, if there not relatively prime, then it has to return as 0. So, it has to evaluate to 0.

Now, associated with every function there are three concepts. We will see them one by one. There is this declaration of a function which says what does the function look like, what is the type of the function. So, the declaration of the function will be written in the following way. It will be written as `int iscoprime (int a, int b)`. This means that iscoprime is the function name, and then it takes two arguments - a and b which are of type int; so int a and int b. If we had written another function which takes a float a and int b, we would say, function int, float a, int b.

So, in this case we are taking two integers as arguments, so you have to say, int a, int b. A small syntactic point that you have to notice, that, you cannot abbreviate this as int a, b; so that is not allowed. Each variable needs to have a separate type signature. So, these are called the input arguments. So, that is the second part of the declaration. The first part of the declaration, the first, which says that, it is an int, is actually the type of the return value. So, the return value is 1 if the pairs is co-prime, and it is 0 if the pair is not co-prime. So, the return value is an integer. So, we need a function name, we need a declaration of the input arguments. The arguments need to be named, and the return value of the output.

So, let us say how do we design the higher level function? So, here is how you use functions when you program. You assume that the function is already available to you, and it does what it is supposed to do. Using that how do I build the solution to the whole program? So, in this case, let us just assume that we have written int iscoprime; we have written that function. And we are interested in, how do we build the solution to the entire problem using that?

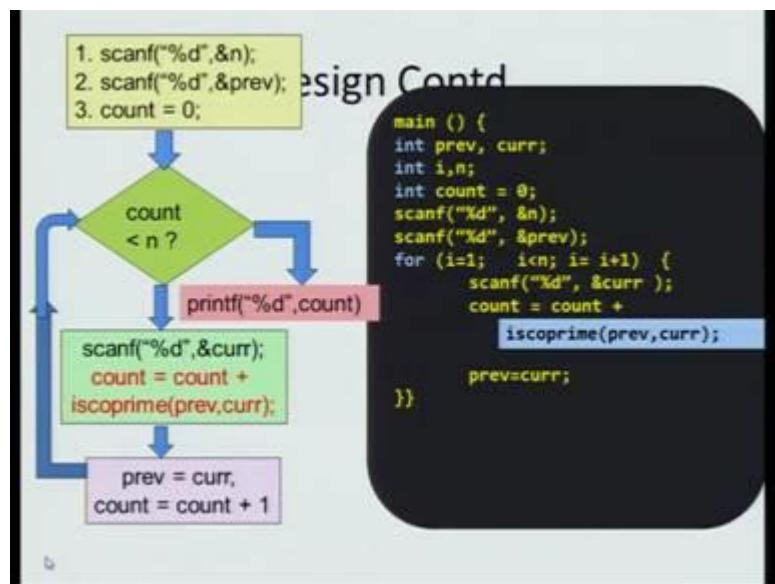
(Refer Slide Time: 11:12)



So, how do you do that? Use, have a flow chart which reach numbers one by one. And count is the number of co-prime pairs that you have seen so far. So, you check whether you have seen n numbers. If you have not seen n numbers then you read the next number and check whether the previous number and the current number form a co-prime pair. So, you give iscoprime prev current; it will return 1, if they are co-prime. So, that will get added to the count. If there not co-prime, they will, it will return a 0. So, count will remain as it is.

Once you do that you say prev is = current, and indicate that you are going to read the next number. This is similar to other problems where we scanned this bunch of numbers and did some function based on that. The new think here is the iscoprime function which we just assumed that it is correctly written, and it does what it is supposed do. So, this is the function declaration.

(Refer Slide Time: 13:10)



Now, how do you code this up? You basically code this up in c, just as you did it with other program, other functions like scan f. You just say, count + iscoprime previous current. So, this is how you can encode the flowchart including the function called as a C program.

(Refer Slide Time: 13:34)

```
int iscoprime(int a, int b) {
    /* returns 1 if gcd of a and b is 1 and 0 otherwise */
    /* a and b assumed > 0 */
    /* write the gcd code */
    int t;
    if(a < b){t=a; a=b; b=t;} /* exchange a and b */
    while ( !(b == 0) ) {
        t = b;
        b = a%b;
        a = t;
    }
    /*a is the gcd. For co-primality, test if a is 1*/
    if (a==1)
        return 1; /* important: functions */
    else
        return 0; /* return value using keyword return */
}
```

Now, let us come to the interesting part which is, how do we design the int iscoprime function? So, the top is the declaration part of the function where I say that what is its

type. So, the function name is `iscoprime`. It takes two variables `a` and `b`; `a` is of type `int`, `b` is of type `int`. And it is supposed to written in integer value. So, that much is clear from the type declaration the type signature so called of `iscoprime`.

Now, what you do with it? You say that, so this is the classic gcd code; you declare a `t` variable; if `a` is less than `b`, you swap `a` and `b`. And this part of the code is just calculating the gcd. This is code that we have seen before. And at the end of that, `a`, will become the gcd. If `a` and `b` are co-prime then `a`, will be 1. If, `a`, is any number greater than 1, then they are not co-prime. So, if, `a` is `= 1`, you return 1. And for returning, you use the keyword `return`. So, you return the value 1; otherwise you return the value 0. So, this is how you write the function `iscoprime`.

(Refer Slide Time: 15:11)

```
#include <stdio.h>
int iscoprime(int a, int b)
{
    int t;
    if (a < b) {
        t = a;
        a = b;
        b = t;
    }
    while ( !(b == 0) ) {
        t = b;
        b = a%b;
        a = t;
    }
    if (a==1)
        return 1;
    else
        return 0;
}

main () {
    int prev, curr;
    int i,n;
    int count = 0;
    scanf("%d", &n);
    scanf("%d", &prev);
    for (i=0; i<n; i++) {
        scanf("%d", &curr );
        count = count +
            iscoprime(prev,curr);
        prev=curr;
    }
}
```

In sequence in the same file

So, now we have to put both these go together. So, I will say, include `<stdio.h>`; this is the first line of the code. Then I will have the source code for `iscoprime`. So, I will write that. And afterwards write name function, so that, when `main` calls `iscoprime` function, then we already have the code for `iscoprime` available. First this line, then the `iscoprime` function, and then the `main` function.

(Refer Slide Time: 15:53)

```
#include <stdio.h>
int iscoprime(int a,
int b){
```

a, b are the **formal parameters** of iscoprime function. Specified to be of type int each. Formal parameters are viewed as variables inside function body.

```
    int t;
    if (a < b) {
        t = a;
        a = b;
        b = t;
    }
    while (b != 0) {
        t = b;
        b = a%b;
        a = t;
    }
    if(a==1)
        return 1;
    else
        return 0;
}
```

return is used to pass back the function's value

So, let us look at the function in somewhat greater detail; a and b are what are called the formal parameters of the function. They are viewed as variables. Now, the formal parameters are visible only within the function. So, we say that their scope is inside the function.

(Refer Slide Time: 16:22)

```
#include <stdio.h>
int iscoprime(int a,
int b){
```

Functions can be called. E.g.,
main () { int x;
x = iscoprime(5,6);
printf("%d",x);
}

5, 6 are referred to as actual parameters of the call.

Once a function call is encountered.

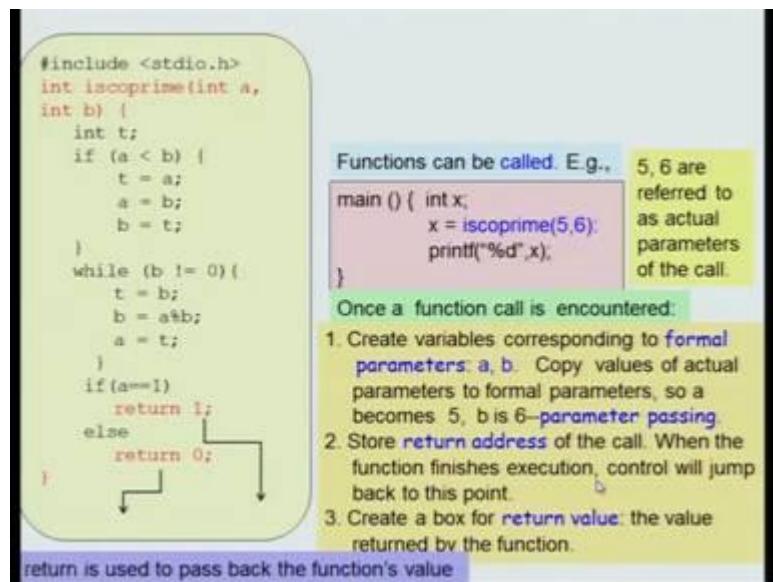
```
    int t;
    if (a < b) {
        t = a;
        a = b;
        b = t;
    }
    while (b != 0) {
        t = b;
        b = a%b;
        a = t;
    }
    if(a==1)
        return 1;
    else
        return 0;
}
```

return is used to pass back the function's value

Now, there is, this is what is known as the declared definition of the function. Every function can be called. Notice that we have already called the functions like print f and

scan f. So, once you define a function you can call a function; calling a function will be evaluating that function with particular arguments; you can do that. So, when you call a function you execute the function with the given arguments. So, 5 becomes a, and 6 become b.

(Refer Slide Time: 16:55)



Once a function call is encountered what happens is that formal parameters are mapped to actual parameters. So, a becomes; so the value 5 is copied to a, and the value 6 is copied to b. This process of copying values is known as parameter passing. Then what you do is, you store the return address of the call. The return address is the line of the main function where the function was called. So, let us say that it was called in the second line of main. Once the function finishes it has to come back to this point.

Now, in addition, we also create a box for storing the return value. At the end of function either 1 or 0 will be returned. So, we also need some space in memory to store that return value.

(Refer Slide Time: 17:53)

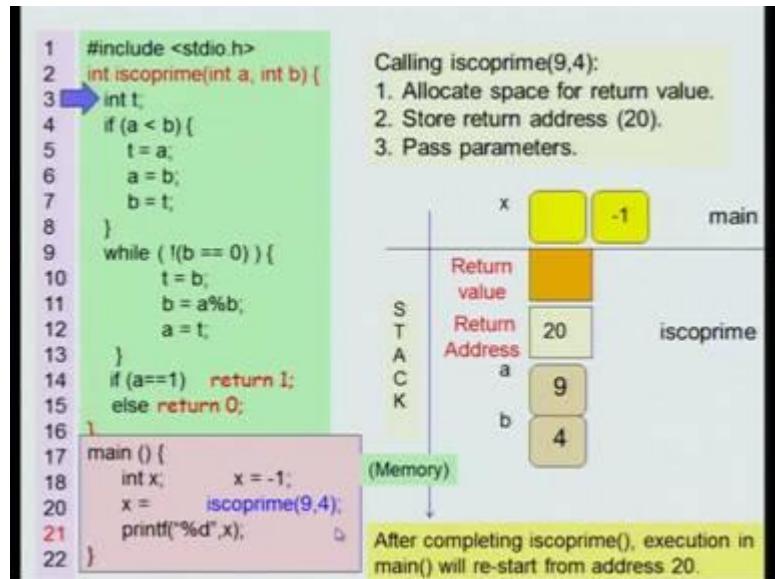


(Refer Slide Time: 17:55)

<pre>1 #include <stdio.h> 2 int iscoprime(int a, int b) { 3 int t; 4 if (a < b) { 5 t = a; 6 a = b; 7 b = t; 8 } 9 while (!(b == 0)) { 10 t = b; 11 b = a%b; 12 a = t; 13 } 14 if (a==1) return 1; 15 else return 0; 16 } 17 main () { 18 int x; x = -1; 19 x = 20a iscoprime(9,4); 20 printf("%d",x); 21 } 22 }</pre>	<ul style="list-style-type: none">• Steps when a function is called: <code>iscoprime (9,4)</code> in step 20a.• Allocate space for (i) return value, (ii) store return address and (iii) pass parameters.1. Create a box informally called “Return value” of same type as the return type of function.2. Create a box and store the location of the next instruction in the calling function (main)— return address. Here it is 20. Execution resumes from here once function terminates.3. Parameter Passing- Create boxes for each formal parameter, a, b here. Initialize them using actual parameters, 9 and 4.
---	---

So, to look at it in slightly greater detail, so let us say that `iscoprime 9, 4` is called in step 20 a. So, this is the address; 20 a by which I mean it is line 20 and some location a. So, now, you have to allocate the space for the return value; store the return address and pass the parameters. Now, at, when you pass the inputs, 9 and 4, the space is allocated for, a = 9, and b = 4. This is the process of parameter passing.

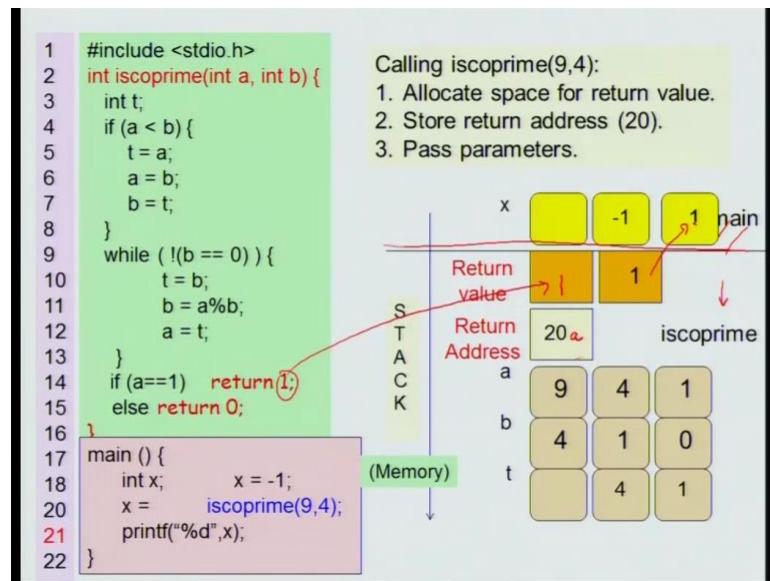
(Refer Slide Time: 18:36)



So, we visualize the memory as a stack. So, when you start the programs you start executing from line 1 of many; so `x` is initialized to `-1`. And then you come to the function called `iscoprime` 9, 4. So, when you execute this you do the following: you allocate the space for the return value, you pass the parameters and then execute the function, and finally pass back the return value.

So, when you execute the function you imagine that the stack is now divided into a separate space. So, here is a clean separation between the memory that is required for `main`. So, above here is `main`, and below here is the memory required for `iscoprime`. So, in that I have stored a box for return value. I have stored the return address which is 20 a. And then I have, `a = 9`, and `b = 4`. Now, I will execute the function as though memory is limited to here.

(Refer Slide Time: 20:00)



So, I will declare `t`, and then execute the gcd algorithm. So, this is stuff that we have seen before. And finally, `a` is the gcd which is 1. If `a` is 1 we have to return 1. So, the value 1 will be copied to the return value, and that is the value that will be passed back; so `x` will be 1. So, the return value will be copied back to the main function.

Introduction to Programming in C

Department of Computer Science and Engineering

When comes to C functions, we have seen the following concepts. One is the declaration in the definition of the function by which I mean the declaration is what type is the function? What are the input arguments? What types are the input arguments? And what is the result return type? So, these form the declaration. Definition is the logic of a function. So, this is what is known as the declaration and the definition of the function and we do it only once. So, function is defined only once. Once we define a function we can of course, call the function multiple times. So, definition is done only once and calling can be done any number of times.

(Refer Slide Time: 00:49)

Stack

- We referred to stack.
- A stack is just a part of the memory of the program that grows in one direction only.
- The memory (boxes) of all variables defined as actual parameters or local variables reside on the stack.
- The stack grows as functions call functions and shrinks as functions terminate.



■ Direction of stack growth doesn't matter. We will usually show it growing downward.

Now, we refer to a stack which is, what is the central concept in executing a function. Stack is just a part of the memory, that goes only in one direction. So, that is what it is supposed to mean. Basically, you can think of it as a stack of boxes or a stack of paper on a table or a stack of plates. So, it grows in one direction. So, the stack grows as the main calls of a particular function, that function calls a different function and so, on.

And you can imagine the stack is growing upwards or growing downwards. It does not matter. As functions get called it either grow keeps growing upwards or keeps going downwards. We will usually represent it us keeping growing downward.

(Refer Slide Time: 01:44)

```
# include <stdio.h>
int fact(int r) { /* calc. r! */
    int i;
    int ans=1;
    for (i=0; i < r; i=i+1) {
        ans = ans *(i+1);
    }
    return ans;
}

main () {
    int n, k;
    int res;
    scanf("%d%d", &n, &k);
    res = (fact(n)/ fact(k))/fact(n-k);
    printf("%d choose %d is",n,k);
    printf("%d\n",res);
}
```

- Define a factorial function.
- Use to calculate nC_k
- Let us trace the execution of main().
- Add temporary variables for expressions and intermediate expressions in main for clarity.

So, let us look at this function that we were talking about earlier. So, n choose k is n factorial upon k factorial times n minus k factorial and let us try to code this up. We know that factorial is something that we will need over and over in this program. So, let us say that I write factorial as a function. So, factorial takes an integer and returns an integers. So, the declaration is int fact int r, r is a input argument and the return type is int.

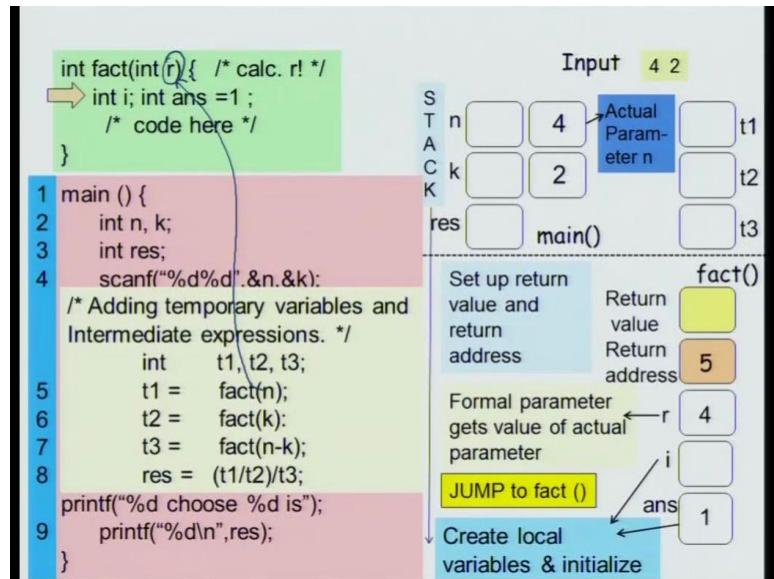
Now, inside that we will write the code for factorial. All variables declared inside the factorial or local or private to the factorial function, they cannot be seen outside. So, the input argument as well as any variables declared inside factorial or private or local to the factorial function. So, I have i and this encodes the logic of factorial that we have seen earlier.

So, you start with the product equal to 1 and keep on multiplying the numbers, till you reach r factorial. So, once you reach r you return the r factorial. This logic is something that we have seen before. Now, we will see how do we put this together in order to produce the function. So, what we need to do is? We will just encode this solution that we have. So, it is $(\text{fact}(n)/\text{fact}(k))/\text{fact}(n-k)$. So, here are the encoded just a logic.

So, even though division is involved I know that when I do nC_k the result is always going to be an integer. So, I can declare it us int res. So, this part is known as the definition of the factorial function. So, this part is what is known as definition and each of this are what are known as calls. Now, let us try to see what happens when we execute this

program? So, regardless of how many functions have been defined, whenever you start executing a programming it always executes the first line of main. So, let us try to add some temporary variables. Because, we call this function three times in this main. Let us try to separate them out into three separate calls, just for the sake of clarity.

(Refer Slide Time: 04:54)



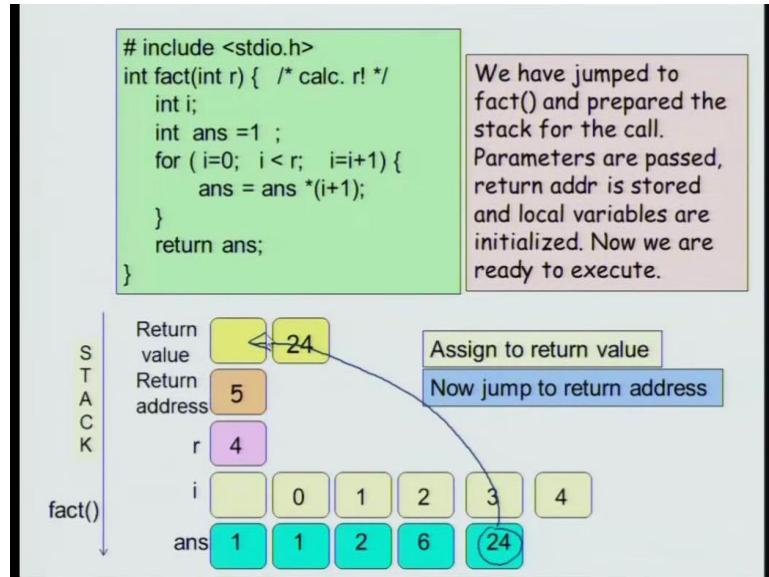
So, I will add slightly larger code, this is not proper c code. I let us say that I have three extra variables which have declared of int **t1**, **t2** and **t3**. Now, **t1** will be factorial of **n**, **t2** will be factorial of **k** and **t3** will be **fact(n-k)** , have separated this out. So, that, I can clearly explain what happens when the code executes. Let us say that I want to calculate 4 c 2. Now, first when the program starts executing, you start with code on the first line of the main.

So, you **scanf n and k**. So, **n** is 4 and **k** is 2. Now, use do **t1 = fact(n)** . So, when **t1 = fact(n)** is called, what you do is, you set up the return value and return address. So, return value is not yet decided to return address is 5, because, you have to go back to line 5 of the code. So, that is why the return value is 5. Also what you need to do, you need to copy the parameter value which is 4. So, this is the actual parameter 4 and you have to copy it to the input argument **r**.

So, **r** is the input argument, **r** should be assign to the value **n** here, **n** is 4. So, that is known as passing the argument. Now, once that is done the code can be seen us jumping to factorial. So, as soon as the function is called, you actually pass the execution to the factorial function. Now, inside the factorial function you have two in local variables **i** and

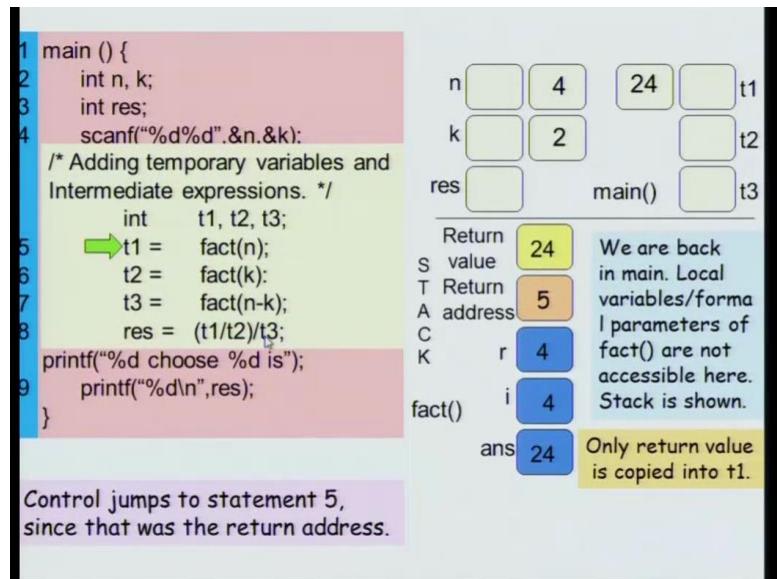
ans which is answered. And we start executing the factorial function. So, let us see what happens, when we execute the factorial function. So, far we have passed the arguments and so, on.

(Refer Slide Time: 07:22)



Now, I have just hidden the part of the stack that was used for name. And let us focus just on the factorial function. This computes the factorial function, that we are familiar with this nothing new here. So, it has a variable `i` which keep track of how many times it has loop has executed and `r` is notice 4. So, you compute the factorial of 4. Finally, when `r = 4 ; ans = 24` now, this 24 value we say return the answer value. So, answer value is 24. So, this will be copied to the return value location. So, the return value will get the value 24 and now jump back to return address. So, return address is line 5.

(Refer Slide Time: 08:28)



So, will jump back to line 5 and there we will say that $t1 = 24$. Only the return value is copied back to the main program all other things are irrelevant. So, the correct way to imagine what happens. When the function has returned is that, the stack that was allocated to main to the execution of fact is completely erased. So, once we go back to main as soon as the function returns back to the main. You should imagine that the entire stack is deleted, and only the memory that was originally allocated to main remains.

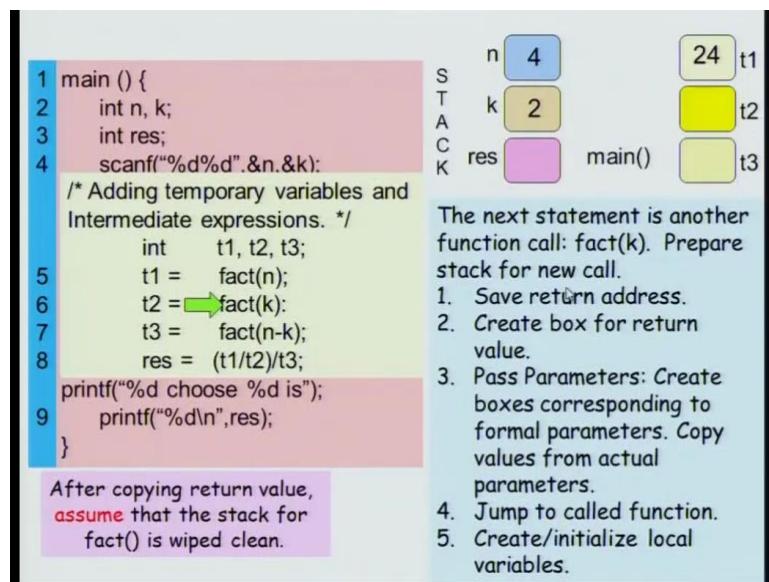
So, the correct way to think about a function executing, you can imagine that, you are main and you have a friend, who can calculate factorial for you. Now, you can ask your friend to calculate factorial for you and things are done in a very hygienic manner. So, what you do is, you write on a piece of paper the number 4 and give it your friend. Now, your friend is another room. So, he has at his disposal some black board. So, he looks at the number 4 and using the private local variables that he has, which is `i` and a result or answer, he calculates the factorial of these numbers. Once see does that, he copies the result back on to a piece of paper. So, 4 factorial is 24 and brings it back to you. Before he does that, he erases the black board and he will bring back the number 24 on a piece of paper.

Now, you can imagine that the space that your friend used to compute 24 has now been wiped clean. And all that remains is the value 24 which you can copy back on to your note book. So, this allegory tells you exactly what happens in the case of function execution. You write down what you want the factorial of on a piece of paper, pass it to your friend, he will go to a separate room. And he will calculate whatever he wants. Once he does that,

he will clean his black board, right down the result on a piece of paper and bring that paper back to you.

So, as far as you are concerned you are least bothered with how he is computing the factorial function. All you want is the result. And this is the basic way to think about functions. You should be able to reason out a bigger program by saying, what does a smaller program, what does a smaller function do regardless of how that function does it.

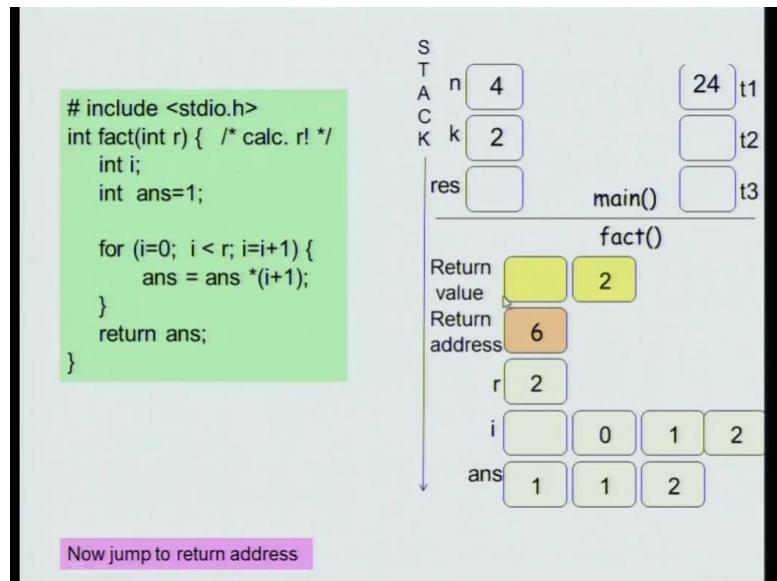
(Refer Slide Time: 11:45)



Now, let us get on with the remaining execution. We have just computed factorial of 4. Now, we need to calculate factorial of 2 and factorial of 4 minus 2. So, we go to the next line, the next line also involves the call to factorial of k. So, we do the same things again, we save the return address. Now, the return address is 6. Because, we are executing line 6, then we create a box for the return value and pass the parameters, and finally, jump to the called function.

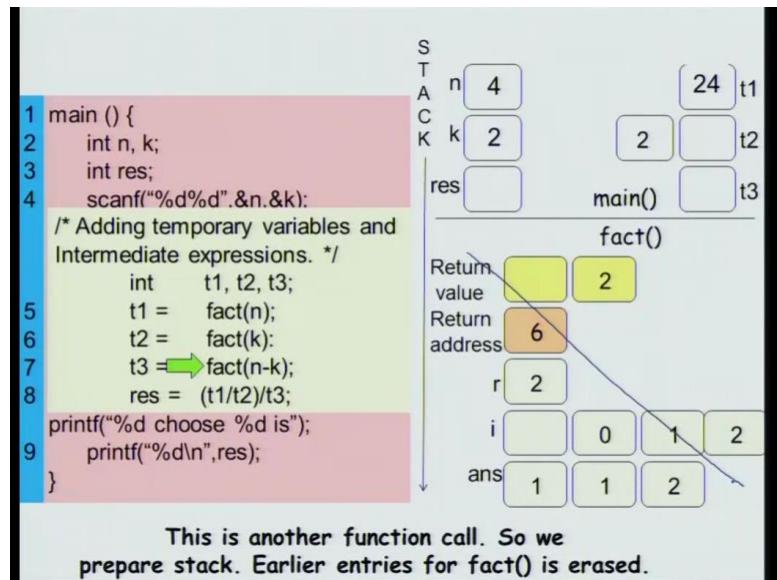
So, we do all that we have some memory for main. But, we allocate a new space in the stack for executing factorial. At this point return address is 6. Because, it is a second factorial that is being called, r is 2, because, k is 2 and you execute the factorial function.

(Refer Slide Time: 12:43)



So, you again go to the factorial function and calculate 2 factorial, 2 factorial is 2. So, that will be transferred back to the return value

(Refer Slide Time: 13:00)



And now you can imagine that, you will get back to the address 6, where `t2` will have the value 2. So, once you do that again the thing to imagine is that, this slate is wiped clean. And all the memory that you allocated to the stack is now free. So, all once you are back in main all you have as the memory for me. Now, there is the third call to factorial. **Fact(n-k)** and it is done in exactly the same manner without much elaboration. So, it will create **n - k is 4 - 2** which is also 2 and the return address is 7 equation n.

And once you do that, it will execute the factorial code again, and calculate the factorial of 2 which is again 2 and return to line 7. So, 2 will be copied as the return value and once the execution finishes, you return to line 7 of main program. At this point, you say that $t3 = 2$. And you can imagine that the stack allocated to factorial is now erased. So, at this point main has $t1 = 24$, $t2 = 2$ and $t3 = 2$. You have all the information that you need in order to calculate your result.

So, you calculate $(24/2)/2$ and the answer is 6 which is 4 choose two. So, this illustrates how do you write a function? How do you define a function? And how do you call it? And what actually happens when you execute a main function? So, the execution of a function can be visualized as a stack. A stack is a part of memory, that is allocated as private to a new function that is being called. Once that function finishes execution, the stack is erased and you go back to the previous function. And you go back to the calling function.

Introduction to Programming in C

Department of Computer Science and Engineering

We have been talking about designing programs using functions. And the general philosophy is that, you have a large task that you want to accomplish and you break it in to sub task, may be each of those sub task are split it in to smallest sub task and so on.

(Refer Slide Time: 00:19)

A general principle of program development

- 1. Break up your task into smaller sub-tasks, and those sub-tasks into still smaller sub-tasks and so on until each sub-task is easily solvable in a function.**
- 2. Write a function for each of the sub-tasks.**
- 3. Design your program from the top-down, big task to the small tasks.**
 - I. Debug/test your program bottom-up. Debug functions that perform elementary tasks, and then move to testing more complex functions.**
 - II. elementary tasks, and then move to testing more complex functions.**

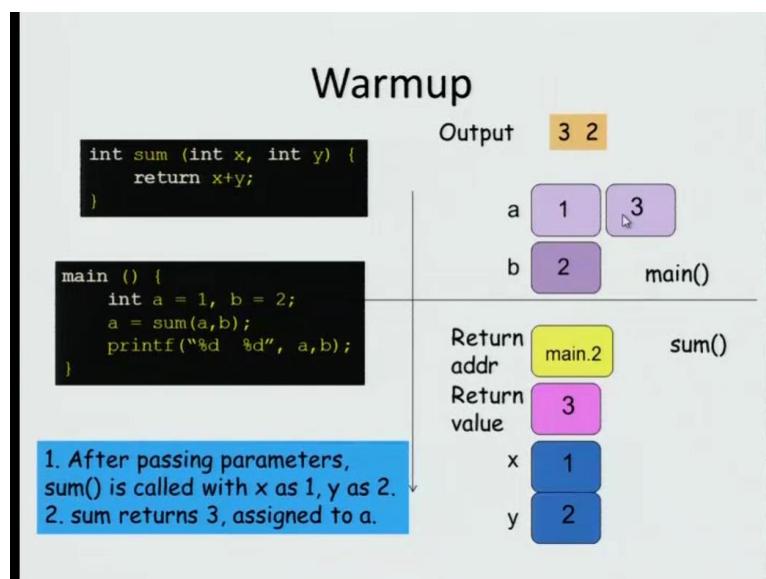
So, break them until some sub task can be easily solved by single function. And then, you put all these function together in order to solve the whole problem. So, design your program from top down, big task decomposing into small task and so on. And debug your program or make sure that they are free of errors from the bottom up. So, test each functions thoroughly and then test the overall program.

(Refer Slide Time: 00:50)

- How does C pass arguments to functions?
[Call-by-value]
 - Evaluation order
 - Side Effects
- Returning values

In this, we will discuss a few technical details about how C executes its functions. In particular we will see how C passes arguments to its functions, and also how does it return values. When passing arguments will talk about issues like evaluation order, in what order are arguments evaluated, if there are multiple arguments. And we will discuss what are known as side effects.

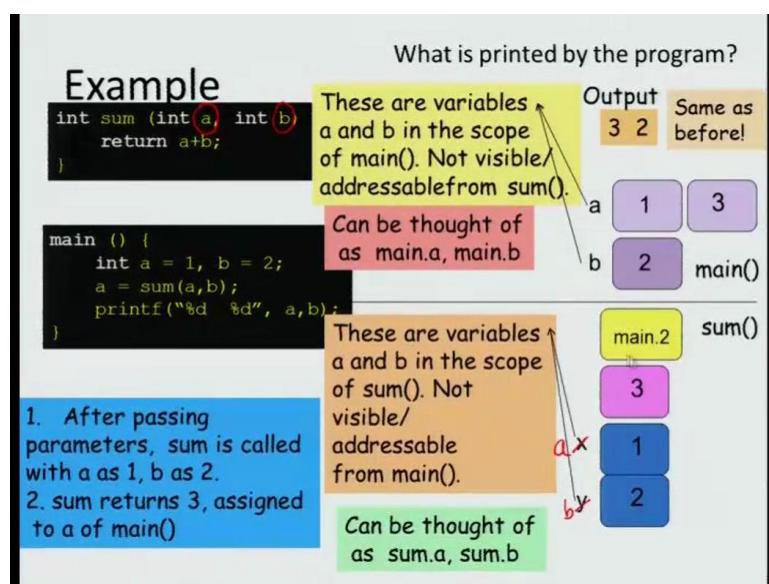
(Refer Slide Time: 01:23)



So, let us start by considering a very simple example. I have a very simple function called sum which just adds up two variables x and y which are integers. Therefore, the return

value is also an integer. Now, inside the main program I will call `a = sum(a,b)`. A is 1 and b is 2. And then, you will sum this up and the return value is assigned to a. So, this is suppose to do 1 plus 2 3 and a is assigned the value 3. So, after passing the parameters sum is called with x as a, which is 1 and y as b which is 2. So, sum returns 3. So, the return value is 3 and the 3 is assigned back to a. So, when you print it, the output will be 3 followed by 2. So, this is simple enough.

(Refer Slide Time: 02:32)



Now, let us try, slightly more tricky example. So, here is the novelty in this example. Some instead of being declared with x and y are now being declared with two variables called a and b. The main program also has two variables, named a and b. So, what will happen here? The output is the same as before. So, 3 2 if you compile the program and execute it, it will be same as before. So, what really happened here?

After passing the parameters sum is call with a as 1 and b as 2. And so it returns a value 3 assigned to the a of mean. Now, these variables are called a and b in main and they are called a and b in sum as well. Now, the variables a and b inside sum are different from the variables a and b inside main. So, the scope of these variable is mean and the scope of these variables a and b is the sum function.

So, in other words the a and b inside sum has scope just this function, they are not visible or addressable outside especially in main. So, if you want to think of it, you can think of them as `sum.a`, `sum.b`. So, they are the a variable belonging to sum and the b variable

belonging to sum. So, even though you would think that this a and this a may get confused they are actually different variables. One is the a variable belonging to main, and the other is the a variable inside sum and they are different, even though they have a common name.

(Refer Slide Time: 04:47)

Example

What is printed by the program?

```
int sum (int a, int b) {
    return a+b;
}

main () {
    int a = 1, b = 2;
    a = sum(sum(a,b),b);
    printf("%d %d", a,b);
}
```

Evaluation of sum(sum(a,b),b)

1. First evaluate inner sum(a,b)
for a = 1, b=2.
2. This is 3.
3. So sum(sum(a,b),b) becomes
sum(3,2).
4. This is 5.
5. So output is 5 2

Pure expressions do not change the state of the program, e.g.,

1. $a - b * c/d$
2. $f(f(a,b), f(f(a,b),a))$

Expressions with side-effects change the state of the program for example,

1. $a = a + 1$
2. $f(a=b+1, b=a+1)$

So, now let us try a slightly more elaborate program, what happens if you have **sum(sum(a,b),b)**, this is the program. In this case what will happen? So, first evaluate the in a program, in a function sum of a b. So, a is 1 and b is 2. So, that will return 3, then you add b again to it b is 2, you have 5 as the total sum. So, the total the complete output is a will be assigned 5 and b is still 2.

So, this is similar to evaluating a normal mathematical expression. One thing that we need to take care of is to handle expression with side effects. Now, what are expression with side effects? So, let us classify expression into two kinds, one is what are known as pure expressions. So, they are the normal mathematical expression, like $a - b * c/d$ and so on. Similarly evaluating function, these normally do not have any effect other than returning you the value.

So, they will be correctly evaluated and they will return some value, other than that, they have no effect. Now, expressions with side effects change the state of the program. For example, when I execute an expression $a = a + 1$. Now, this is an expression, it has a value. So, let us say that a was 1 before $a = a + 1$. A plus 1 has value 2

and a is assign the value 2. The state of the program involves, for example, what values are stored in the variables. When you execute the expression $a = a + 1$, the value of the variable a changes. Contrast this with previous expression, like $a - b * c / d$. You can see that, unless you assign to something no variables value is changing, it will just evaluated and the value will be return. Here, the value will be returned also variable a is changing. Here, in this second function you have two arguments, two function f. The first is the expression $a = b + 1$, the second is an expression $b = a + 1$.

This might sound like a very strange way to code. But, you know that any expression can be given as arguments. So, in particular assignment expressions can be given as arguments. For example, $a = b + 1$ is an assignment expression, which is given as an argument to the function. Now, such expressions are called expressions with side effects, because, the change the state of the program.

(Refer Slide Time: 08:01)

Example

What is printed by the program?

```
int minus(int a, int b) {
    return b-a;
}
```



BUT!

Rule: All arguments are evaluated before function call is made.

C doesn't specify order, in which arguments are evaluated. This is left to the compiler.

Let us evaluate function arguments in left to right order.

a	2	2
b	1	3

Evaluate sum($a=b+1, b=a+1$). How should we evaluate it?

When you have side effects you should be careful. For example, what will happen in the following program? You have function `int minus(int a, int b)` and it returns $b - a$. Now, in this program main calls the minus function with two expressions as arguments $a = b + 1$ and $b = a + 1$. They are expressions with side effects, because, once evaluate these arguments, you know that the variable a will change in the first expression and the variable b will change the second expression.

So, what will happen in this program? So, how should we evaluate it? The general rule is that all arguments are evaluated before the function call is made. So, before the function is executing, we know that $a = b + 1$ and $b = a + 1$ both will be executed. But, and here is the major problem, we know that both have to be executed. But, C does not specify in which order they have to be executed, so, it was left to the compiler.

So, let's evaluate it in first in left right order. So, this expression first and then $b = a + 1$. So, what will happen then? $a = b + 1$ b is 1. So, a will get the value 2, $b = a + 1$ will be executed after that a is now 2. So, b will get the value 3. Now, you execute minus. So, you will return $3 - 2$ which is 1 and b has value 3. So, this is the expected output. But, when you run it on some machines, you may get the output -1 3.

(Refer Slide Time: 10:07)

Left-right OR right-left

```
int minus(int a, int b) {
    return b-a;
}

main () {
    int a = 2, b = 1;

    a = minus( a=b+1,    b=a+1);

    printf ("%d %d", a,b);
}
```

We used left to right evaluation. Expected output:

1 3

Let us compile and run.
On some machine, output is:

 **-1 3**

b = a+1 b [3] What happened?
The compiler evaluated right to left. Output is

So, what happened here? Now, this happens for example, when the compiler would evaluate it right to left. So, when you evaluate it right to left what will happen is that $b = a + 1$ will be executed first. So, $b = a + 1$, b gets the value 2 plus 1 3. And then, you will execute $a = b + 1$, b is now 3. So, a gets the value 4. So, when you call minus of 4 comma 3 minus will return $3 - 4$ which is -1 . So, in this case you know that b gets the value 3, a gets the value 4 and the result will be -1 .

(Refer Slide Time: 11:10)

What was the mistake?

- So what was the mistake?
- Actually, C does not specify the order in which the arguments of a function should be evaluated.
- It leaves it to the compiler. Compilers may evaluate arguments in different orders.
- Both answers are consistent with C language!! What should we do?

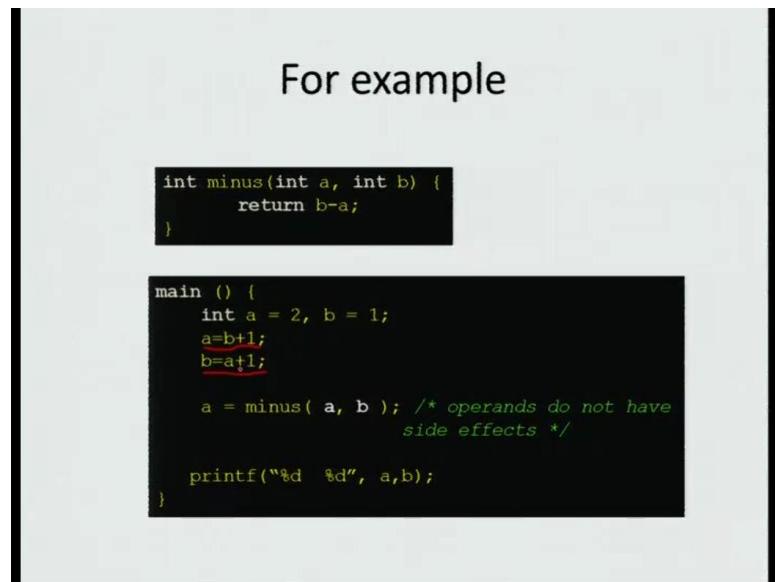
Write your arguments to functions so that the result is not dependent on the order in which they are evaluated. Better still, write them so that the operand expressions are side-effect free.



So, what was the mistake? The mistake was that we assume that both arguments will be evaluated before the function is called. But, we assume that it will be evaluated left to right. And the first expression will be evaluated before the second expression, that is the reasonable assumption to make. But, C does not guarantee you that, C leaves this decision to the compiler. Now, compilers may evaluate arguments in different orders. For example, a very common order is right to left.

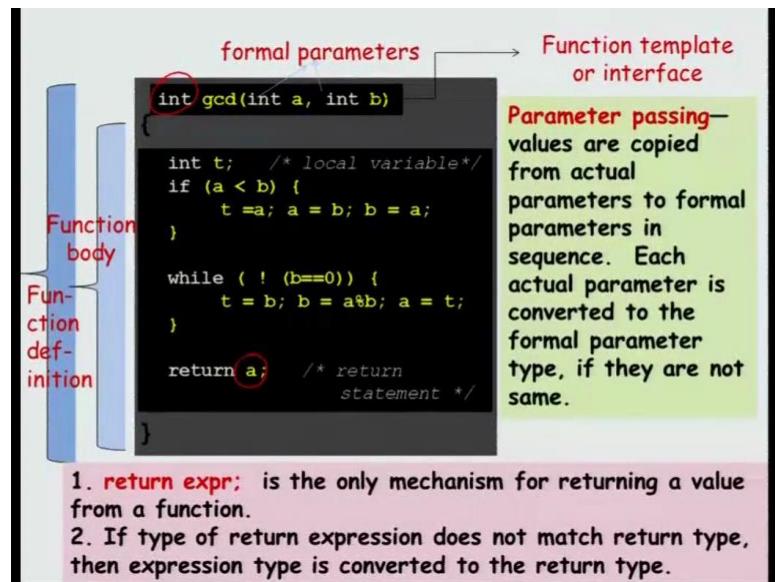
So, both answers like 1 and 3 and -1 and 3 are both consistent with the C specification. Now, this is the very troubling scenario, what should we do? So, we should write this function in such a way that, they do not depend on whether the arguments are evaluated left to right or whether they are evaluated right to left. So, write expressions in such way, that they are free of side effects, when you pass them into functions.

(Refer Slide Time: 12:13)



So, how do we do that? We can do the following. For example, if we really wanted the left to right order, that is if you want $a = b + 1$ to happen first and then $b = a + 1$. Why not write them explicitly that order in the main function. So, first this will be executed then $b = a + 1$ will be executed. So, a will get the value 2 and then b will get the value 3. So, $\text{minus}(a, b)$ will execute as $3 - 2$ in which case you will get 1. So, the important thing to note is that, in this particular function called, the arguments do not have any side effects. Because, we explicitly coded them up before to specify that, this is the order in which I want. If put it here, then it is up to the compiler, the compiler can do whatever is best in for several criteria.

(Refer Slide Time: 13:21)



So, now let us come back to what the function, we have the function definition which is the entire function. The logic of the function is what is known as the function body. And the heading is what we call the type signature. The type signature has for example, two arguments a and b these are call the formal parameters. Now, we focus on the return expression. So, return followed by some expression is the only mechanism for returning the value from a function.

If the type of return expression does not match the declare type of the return. So, if for example, a is of a variable which is different from int. In this case they are the same, then it is fine. But, otherwise the return expression is converted to this type and then returned. So, it might lead to some undesirable variable. Now, we have discussed parameter passing's in when passing parameters in c, the values from the calling function are copied to formal parameters in the called function. So, the actual parameters are converted to the formal parameter type and separate copies made. So, this is known as call by value.

(Refer Slide Time: 15:03)

More definitions

```
int gcd (int a, int b) {  
    int t; /* local variable */  
    if (a < b) {  
        t = a; a = b; b = t;  
    }  
    while ( ! (b==0)) {  
        t = b; b = a%b; a = t;  
    }  
    return a; /* return statement */  
}
```



© Ross Larson - www.Clipart.com/1048200

Important: executing return anywhere will immediately exit from function and transfer control back to the calling function at return address.

Formal parameters and local variables are visible and accessible only within the function.

Memory for formal parameters and local variables is allocated only when the function is called. This memory is freed as soon as the function returns. (with the exception of static variables).

So, formal parameters and local variables are accessible only within the function, we have already seen in this. And memory for the formal parameters and the local variables of the called function will be erased as soon as function returns. So, executing return anywhere inside the function will immediately return from the function. And transfer control back to the calling function at the specified return address.

(Refer Slide Time: 15:37)

Returning tips

- Executing return expression; will cause function to immediately return to its return address
- We can use return in main(). Executing it will cause main () to return—this means that the program will terminate.
- Value returned by a function can be ignored by the calling function (e.g. below).

```
int f (float a, int b) {  
    /* code here */  
}
```

```
main() { int x, float y;  
    /* some code */  
    • f(y,x);  
    /* some code */  
}
```



But, then why call in the first place?

Because functions can have side-effects!
e.g. scanf - side effect: assigns input to a variable.

So, when you execute there are few things keep in mind. Whenever, you execute any return expression, it will cause the function to immediately return. Now, main is a

function so, we can use return statement inside main what; that means, the main will immediately stop execution. That is the whole program will stop execution. Now, when you return a particular value, the calling function may choose to ignore the value.

For example, let us say that I write some dummy function int f and it takes two argument float a and int b and we some code here. And then, I have the main function in which I have two variable int x and float y. Then, I have some code and here is the interesting thing, I call **f(y,x)**, y is an float x is an int. So, I am find, but this function returns an integer value. But, I am not assigning it to anything. So, I am not saying something like **x = f(y,x)**. So, this is not required.

So, if this is the case, then why call the function in the first place? This is, because the function also may have side effects. So, remember that side effects are something some expressions, we change the state of the program. So, functions may have side effects, you already seen one such function which has the side effect for examples, scanf. So, the side effect of calling scanf is that the input from the keyboard is copied into some variable. So, function may have side effect, this is why you can call the function and choose to ignore the output or the return value.

(Refer Slide Time: 17:42)

- Executing **return;** will cause function to immediately return to its return address (i.e., in the calling function). return value is unpredictable.

What is the output of the program?

```
float f( int a, float b ) {
    float t=a+b;
    return; /* no expr given with return */
    /* return value is unpredictable: garbage! */
}
main () {
    int a = 1;
    float b =2.0;
    printf("%d\n", f(a,b));
}
```

Printed value is unpredictable.

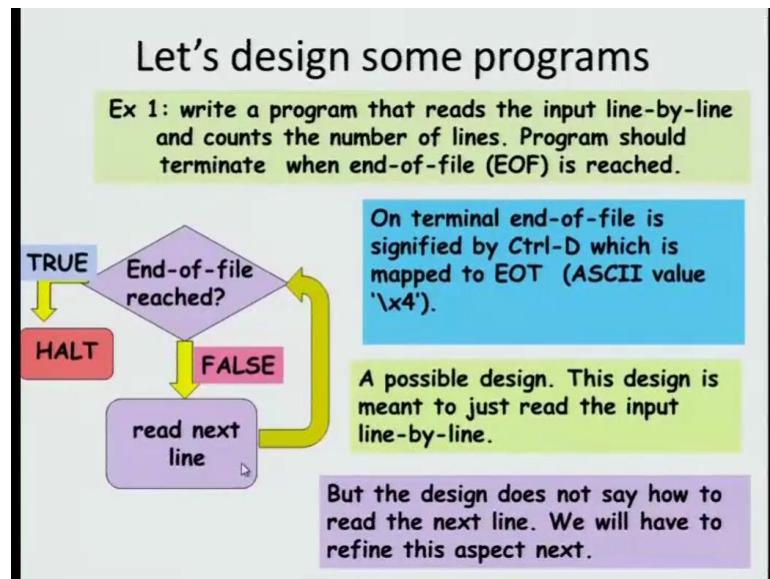
Now, just for curiosity sake executing return will calls the function to immediately return to the return address. Now, the return value if you omit it, then the return value is unpredictable. So, here is a example, you should in general avoid doing things like this.

But, just for completeness, I am supposed to return a float value instead if I just say a return, the program will compile. But, when you execute some unpredictable behavior may result. So, the printed value in this case can in general will not predictable.

Introduction to Programming in C

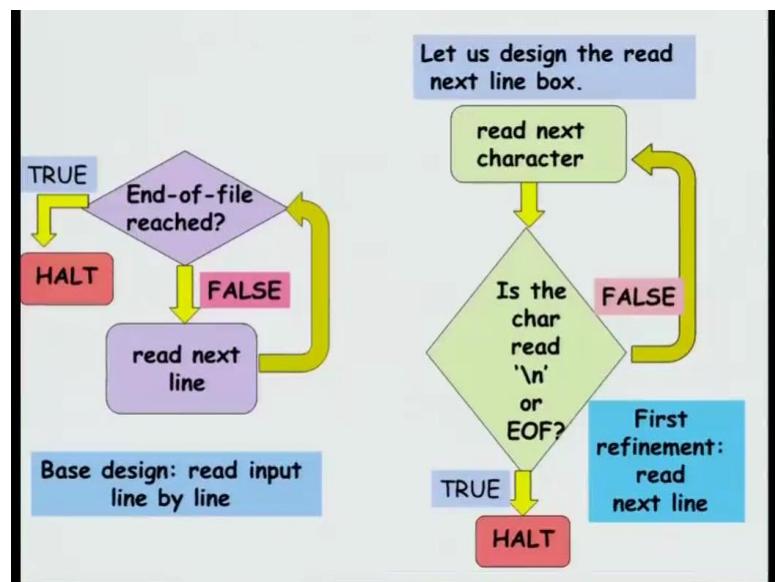
Department of Computer Science and Engineering

(Refer Slide Time: 00:14)



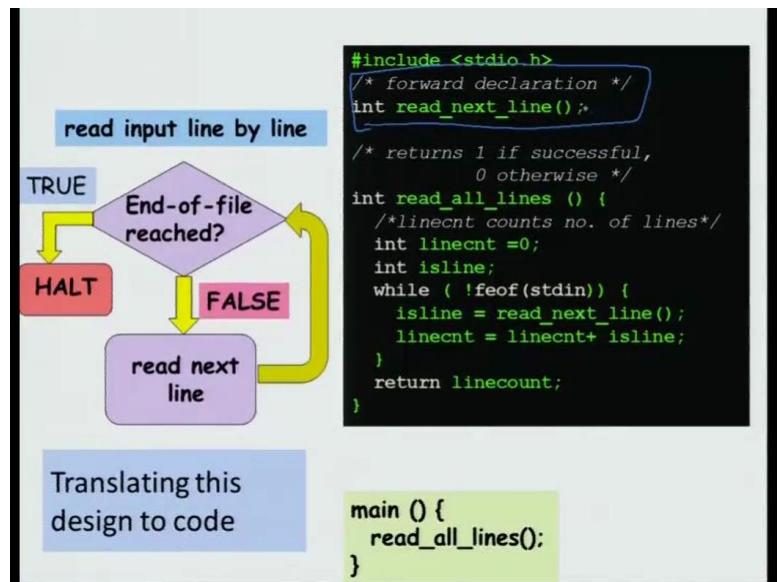
With the concepts we have seen so far, let us design a sample program. So, the... What we ... The problem that we want to solve is we want to write a program that reads the input line-by-line, and counts how many lines has the user input. Program should terminate when the end-of-file character is encountered. So, we will try to solve this problem. By the way, the end of file is a character, which you can enter using control-D if you are running Linux. So, the flowchart at the very top level can be envisioned as follows. So, we will just check has the end-of-file been reached. If the end-of-file has not been reached, you read the next line. If it is has been and check again. If the end-of-file has been reached, then you halt; otherwise you read another line. So, here is the very top-level picture of what we want to do. So, this design is just meant to read the input line-by-line. So, it is a very vague flowchart, but at the top level, this is what we want to do. So, let us say more details about how we are going to accomplish this. In particular, we want to see how we can read and put line-by-line.

(Refer Slide Time: 01:33)



So, here is the top-level design. And now we are going to essentially expand this box. We want to say how do we read the next line. So, let us design the read next line box. So, the read next line box, first you read a character and then you check whether the character read is new line character; that means that the user has pressed an enter. So, the line is ended at that point or the user can enter a bunch of characters; and instead of pressing enter, press control D. So, the user can enter end-of-file. If either of these are true, then the line has ended. So, you halt. Otherwise, if the character is neither new line nor end-of-file, then you read the next character. So, here is the design for the function to read the next character – next line. So, you read character-by-character; after every character, you check whether a new line or an end-of-file has been encountered. If either of them happen, then the line has ended; otherwise, you go back and read another character.

(Refer Slide Time: 02:54)

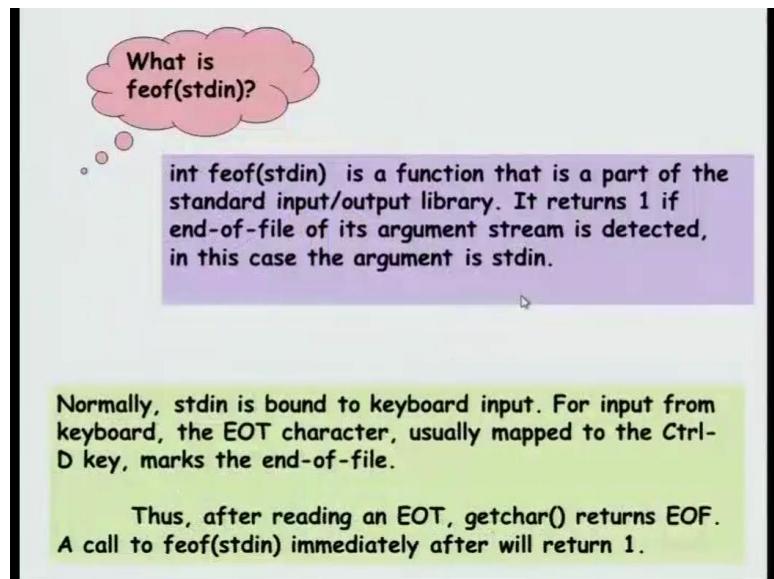


So, let us start by writing the top-level function. So, let us translate the top-level function into code. So, here we will introduce a new concept called what is known as a forward declaration. So, when you define a function, you can either give the logic – the full function body when you define the function or you can just say that, here is what the function will look like; here is the type signature; basically, it is taking no arguments and it will return an integer value. And I will terminate that statement by using a semicolon; which says that, this function... I will currently just say the type of the function; I will define the function later. This is done, so that we can write a function, which uses this particular function. So, when we write a function, which uses that function, the type of the function should be known. For that we can just declare the type of the function. This is what is known as a declaration of a function.

Unless you define the function, you cannot use it; but in order for another function to just see what the function looks like, declaration is sufficient. So, let us design the top-level function. So, we declare this function that, we will use in this function that we are about to write. So, this user function will be called read all lines. Now, in that, we will keep a line count initialized to 0; and then I will keep a flag called isline. Now, what this will do is we have to check for whether an end-of-file has been reached or not. For that, I will use the function feofstdin. We will see that in a minute. While the end-of-file has not been encountered, you say that, read next line; read next line will return a 1 if a line has been encountered; otherwise, it will return a 0. So, line count will be incremented by 1 if

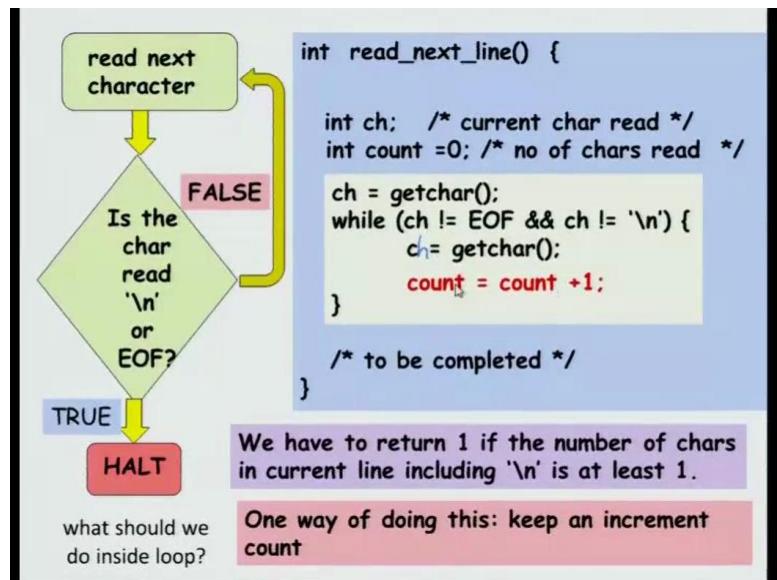
I read another line; otherwise, it will remain as it is. Finally, you return the number of lines read. So, this is a realization of the flowchart on the left. Now, there are a couple of things that require explanation. First is that even though the read next line function has not yet been defined, just based on the declaration, I can say that, it is going to return an integer and I can use the integer here.

(Refer Slide Time: 06:09)



The other thing is what do we mean by `feof(stdin)`? So, what do we mean by the function `feof`? So, `feof(stdin)` is a function that is part of the `stdio` library. We have already used other functions from that library. For example, `printf` and `scanf`. Now, the `feof` function – what it does is – it returns a function; it returns a value 1 if the end-of-file has been encountered in the input argument. So, `stdin` means that, I am using the standard input, which is the keyboard input. So, if an end-of-file has been entered via the keyboard, then `feof(stdin)` will return 1. So, `stdin` is usually the keyboard input. And usually, if the user enters the control D character, then `feof` will say 1, because end-of-file has been entered.

(Refer Slide Time: 07:13)

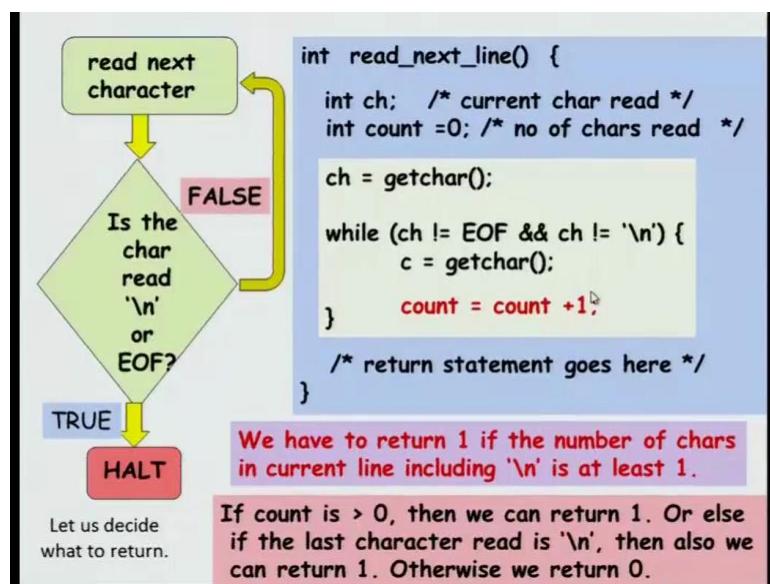


Now, let us design the function to read a line. We earlier wrote a function, which assumed that, there is a function, which will read the next line; and based on that, I will keep on reading lines until the end-of-file is encountered. So, we are now about to write the bottom function. So, we want to read a line. So, we have already drawn the flowchart for that. Now, let us try to make it into code. So, we have to design a few variables; we will have `int ch` for reading a character; we will come to that in a minute; then we will keep a count of how many characters have been read. And let us write the basic loop. So, we will just write the loop corresponding to the flowchart; `ch` will be `getchar`. So, get the next character. And while `ch` is... While the read character is neither end-of-file nor new line, you should keep reading characters. So, if neither of this is true, then you should read the next character, which is what the flowchart says. A slight... a small point here is that, `getchar` returns an integer. This is a technicality because end-of-file is negative 1.

ASCII characters if you remember, go from 0 until 127 or something like that; whereas, end-of-file is defined to be **-1**. So, because of this **-1**, you cannot keep the return value of `getchar` as a character; it technically has to be an integer. Now, this is a technicality. So, keep that in mind. Now, we need to do something further in the loop. So, we will complete this in a minute. So, what should we do inside the loop? This should be character. So, what is this function supposed to do overall? We have to return a 1, if the number of characters in the current line that we have read is at least 1. So, if the current line contains at least a character, then we have to return 1. For example, if the user just

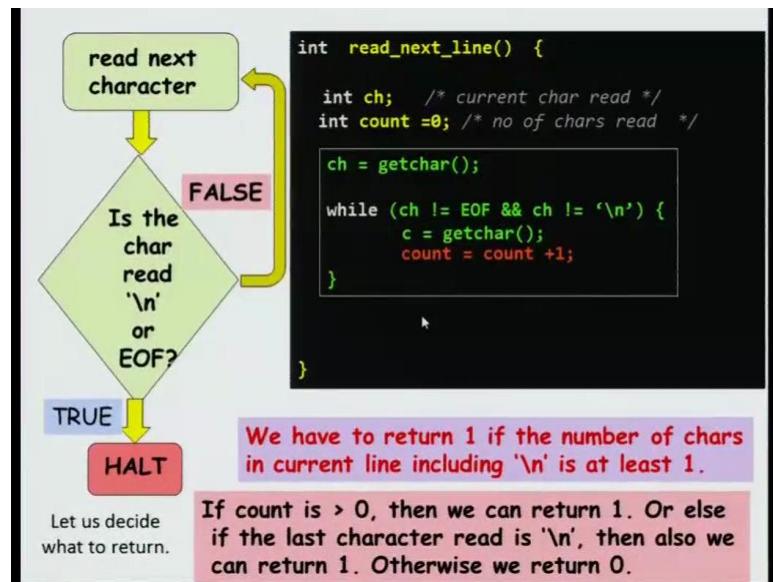
entered a new line, which is just press the enter key, there is a blank line. In that case, we would not say that, we have read a line, because it was a blank line. So, if there is at least one character, which is neither new line nor end-of-file in that line, we have to return a 1; otherwise, let us say we return a 0. So, one way to do that is to keep a count of the number of the characters we have read. So, for every character read, we will keep a count of every character, which is neither end-of-file nor a new line; we will keep a count of characters. So, notice the way that, the loop has been return. So, if the first character is a new line, it will not enter the loop. Hence, count remains 0. At the same time, the way the loop is returned; count will count exactly those characters, which are neither new line nor end-of-file.

(Refer Slide Time: 11:04)



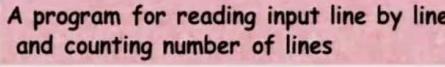
So, now, let us decide what should be the return value. We have to return a 1 if the number of characters in the current line including new line is at least 1. So, if count is greater than 0, we can return a 1. If exactly 1; if the last character was end-of-file without having any other characters, then will return a 0. So, how we do that? We can check whether at least a character has been read by just checking the value of count.

(Refer Slide Time: 11:45)



So, if count is greater than 0, then at least one character has been entered; otherwise, for example, we can also say that, if the user has just entered a blank line, then also we can say that, one more line has been entered. So, that is up to the way you want to do it; you can also take the stance that, maybe a blank line does not count as a line. If that is the case, then you do not have to do it; but in this case, let us just assume that, if at least a character has been entered, which is either a normal character on a new line, we will say that, return 1. If the only character entered in that line is end-of-file, we will say that, there is no more new line. So, what we have to do is return count greater than 0; this tells you how many non-new-line, non-end-of-file characters have been entered. So, this should be at least 1; or, there is exactly one character entered, which is a new line. So, neither these cases we will return a 1; otherwise, we will return a 0.

(Refer Slide Time: 13:03)



```
#include <stdio.h>
int read_next_line();
int read_all_lines () {
    int linecnt =0;
    int isvalid;
    while ( !feof(stdin)) {
        isvalid=read_next_line();
        linecnt = linecnt+
            isvalid;
    }
    return linecount;
}
int read_next_line() {
    int ch;
    int flag = 0;
    ch = getchar();
    while (ch != EOF &&
           ch != '\n') {
        c = getchar();
        flag =1;
    }
    return flag ||
        (ch == '\n');
}
main() { read_all_lines(); }
```

So, we can put these programs together by concatenating all the code that we have written. Notice one thing that declare the function first; we use the function here. So, here is a top-level function, which will use read next line. When read all lines uses read next line; read next line has not been defined yet. So, you can go here after read all lines has been defined, you can define read next line. So, here is a function here. So, this is function 1, this is function 2, and finally you have made. Read all lines does not need any forward declaration, because when main uses read all lines, it has already been defined. That was not the case here. When read all lines used to read next line, read next line was not defined yet. That is why we needed a forward declaration. In this program, you can reorder the code such that read next line code can be written before in which case you do not need the forward declaration. But the concept of forward declaration is useful for later discussion. So, I have just introduced that.

Introduction to Programming in C

Department of Computer Science and Engineering

(Refer Slide Time: 00:08)

Arrays in C

Dictionary meaning of the word array

arr-ay: noun

1. a large and impressive **grouping** or **organization** of things: He couldn't dismiss the **array** of facts.
2. **regular order or arrangement; series:** an array of figures.

◀



This session will learn about arrays in C. Now, what is the word array mean, it means a grouping or a collection of objects. So, for example, you could say that he could not dismiss the array of facts. So, that means, a collection of facts and it also implies a regular order or arrangement that is in the case of a series. So, what do we mean by an array? And why do we need it?

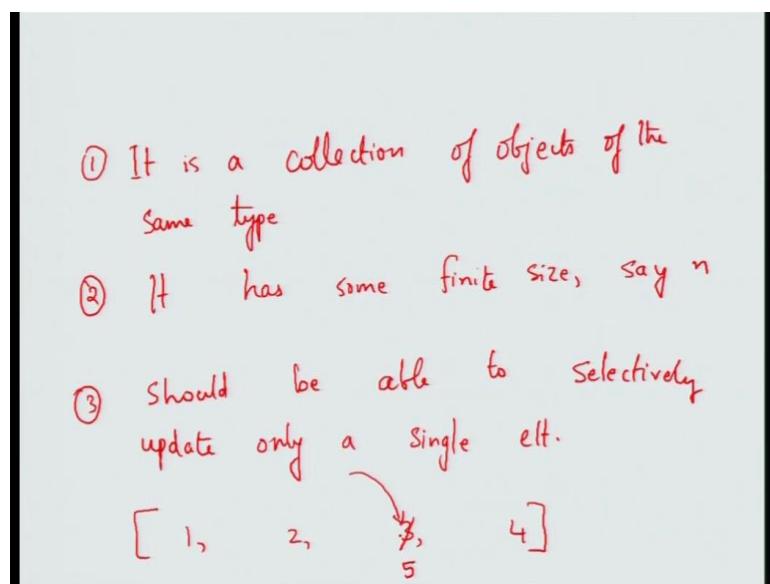
(Refer Slide Time: 00:45)

A hand-drawn diagram of an array. It consists of four numbers enclosed in square brackets: [1, 2, 3, 4]. Above each number, there is a red letter and a red arrow pointing downwards to the corresponding number. The letters are 'a' above 1, 'b' above 2, 'c' above 3, and 'd' above 4.

So, let us consider that I have a bunch of numbers say 1,2,3,4 and I want to consider

them as being part of the sequence. Let us say 1, 2, 3, 4. So, the first element is 1 and so on. Now, I want them to be stored and one way I can do it is that, I can store them in to separate variables. For example, I can say that a is 1, b is 2 and c is 3 and d is 4. But, when I do it in that way, they are separate variables and the relationship between those variables, the fact that b comes after a and things like that is something that the programmer knows, but it is hard for somebody else looking at the code to figure out. Often we need to store sequence as a sequence itself.

(Refer Slide Time: 01:47)



So, an array has the following properties, one it is a collection of objects of the same type. Second, is that it has some size, some finite size say n there are n elements in the array and the third is that, I should be able to selectively update only a single element in the array. By which I mean, suppose I have four elements in the array I should be able to say that, I want to replace the third element by 5. So, 3 will be replaced by 5 without touching the other elements, that is what the third thing is about

The first thing says that I want a collection such that it is a collection of objects of all of type integer and not of any other type. And the second says that, it has a some finite size so, it is not an infinite collection, so, that is what an array is supposed to do. Now, let us see how we can define arrays, will try to motivate why arrays are needed by introducing certain problems. And I will try to convince you that it is easy to do using arrays and whereas, it was difficult to do without using arrays, using only the facilities in C that we have seen so, far.

(Refer Slide Time: 03:58)

Defining arrays

An array is defined in C similar to defining a variable.

```
int a[5];
```

The square parenthesis [5] indicates that a is not a single integer but an array, that is a consecutively allocated group, of 5 integers.

It creates five integer boxes or variables

a[0] a[1] a[2] a[3] a[4]

The boxes are addressed as a[0], a[1], a[2], a[3] and a[4].
These are called the **elements** of the array.

So, an array is defined in c, similar to how we define a variable. If we had an integer variable we would say int a semicolon instead of that when we declare an array we have int a 5. So, this would declare that it is an array containing 5 integers. Now, one thing that is certain about arrays in c is that, the five integers which makeup the array will be allocated consecutively in memories so, they will happen one after the other. Also one think to note is that, arrays in c start with index 0 so, the first element is a 0. So, if we have an array of five elements it will go from a 0 to a 4. So, have we seen arrays and mathematics for example, you can think of vectors similarly matrices these are all arrays and c arrays will have similarities to mathematical vectors and mathematical arrays. But, note that in mathematics, it is customary to start from index 1, here it is from index 0. So, the boxes are addressed as a 0 to a 4 these are called the elements of array. The array is the whole collection of boxes and each box in it will be called an element of array.

(Refer Slide Time: 05:35)

The diagram shows a C program code on the left and its execution state on the right. The code is:

```
include <stdio.h>
main () {
    int i;
    int a[5];
    for (i=0; i < 5; i = i+1) {
        a[i] = i;
    }
    printf("%d", a[i]);
}
```

Annotations explain the code: 'The program defines an integer variable called i and an integer array with name a of size 5' points to the declaration of `a[5]`. 'This is the notation used to address the elements of the array.' points to the assignment statement `a[i] = i;`. A note below states: 'The variable i is being used as an "index" for a.' and 'Similar to the math notation a_i '.

Below the code, a horizontal bar represents the array `a` with elements labeled `i`, `a[0]`, `a[1]`, `a[2]`, `a[3]`, and `a[4]`. The first element `a[0]` is highlighted in yellow. The `i` label above the bar corresponds to the variable `i` in the code. The `a` logo of the Institute of Technology is at the bottom right.

Now, let us consider a simple program using an array. So, I mentioned that the third requirement that I want for in array is that... So, that first requirement was that all elements of the array are of the same type. Second requirement was that, it has a finite size and that the third requirement is that I should be able to selectively update only one element of the array without touching the other elements. So, let us see a program where we can do all that. So, here is a simple program it declares an integer I and integer array a five and then a for loop. So, let see what the for loop is supposed to do. So, the for loop starts from `i = 0` and then goes from `i = 0` to 5, filling in the elements by sing the statement `a[i]` equal to `i`. So, let us see what that is supposed to do.

So, this is the notation `a[i]` is the notation used to address the elements of the array. So, notice the similarity here a 5 when you declare it say's that it is an array of size 5. `a[i]` is saying that I want the `I` th element in the array. So, when `i = 0` it will refer to the 0th element in the array until `i = 4`. It will go on and till the fourth element of the array. So, a of 5 similar to a of 5 the way we row declare the array say's I want the `i`th element of the array. So, the variable `i` is being used as an index for `a`, that means, if I say `a[i]` will pick the `i`th cell, in the `i`th element in the array. Now, this is similar to the mathematical notation `a subscript i`, which is what we normally use for vector and matrices.

(Refer Slide Time: 07:46)

The diagram shows a C code snippet and its memory representation. The code initializes an array `a` of size 5 with values from 0 to 4. A callout box says, "Let us trace through the execution of the program." Another box states, "Fact : Array elements are consecutively allocated in memory." Below the code, the array `a[0]` to `a[4]` is shown with indices 0 to 4 below it. To the right, a variable `i` is shown with values 0 to 4, and the array elements `a[0]` to `a[4]` are shown with values 1 to 5 respectively.

```
include <stdio.h>
main () {
    int a[5];
    int i;

    for (i=0; i < 5; i= i+1) {
        a[i] = i+1;
    }
}
a[0]   a[1]   a[2]   a[3]   a[4]
1       2       3       4       5
i      0       1       2       3       4
```

So, let us run through the program once to see what is doing. So, first we declare a 5, which is five consecutively allocated integers in the memory. And we also have a variable `i`, `i` starts with 0 and for this 0th iteration `a[i]` is allocated let say `i + 1`, so, a 0 will be 1 then we update `i`. So, this statement becomes `a[1] = 1 + 1` which is 2. So, and then execute it a 2 becomes 3, a 3 becomes 4 and a 4 becomes 5. So, notice that because we have indices, and indices can be numbers, they can also be replaced by integer expressions,, this is the trick that we have used here. So, a `a[i]` goes from a 0 all the way up to a 4.

(Refer Slide Time: 09:05)

The diagram shows a C code snippet defining two arrays: `float num[100];` and `char s[256];`. A callout box says, "One can define an array of float or an array of char, or array of any data type of C. For example". Arrows point from the code to the memory representation. The `num` array is defined as an array of 100 floating-point numbers indexed from 0 to 99 and named `num[0]...num[99]`. The `s` array is defined as an array of 256 characters indexed from 0 to 255 and named `s[0]...s[255]`. Below the code, the `num` array is shown as an array of 100 floats, and the `s` array is shown as an array of 256 characters.

```
main() {
    float num[100];
    char s[256];
    /* some code here */
}
array of 100 float      num[0]  num[1]  num[2] ...  num[99]
array of 256 char        s[0]   s[1]   s[2]   s[3] ...  s[254] s[255]
```

Now, it is only required that a single array can be objects of same type, so, we have dealt

with integer arrays so, far. Now, we can also deal with floating arrays and float arrays and character arrays and things like that. So, in general you can declare an array of any data type in c for example, you can say float num 100. So, that will declare array of hundred floating point numbers, similarly char s 256 will declare, a character array of size 256. So, you can declare floating point array and you can visualize it as hundred floating point boxes, allocated consecutively that is important part. The consecutive location in the memory one after the other will be allocated for the same array. Now, for a character array similarly hundred boxes are allocated one after the other. Now, depending on the size of the data type involved, obviously, the size of the array will be different. So, the float array will be of size hundred times the size of the single float and the character array will be of size 256 times with the size of a single character and so on.

(Refer Slide Time: 10:26)

Mind the size(of array)

Consider program fragment:

```
int f() {
    int x[5];
    ...
}
```

This defines an integer array named x of size 5.

Five integer variables named x[0] x[1] ... x[4] are allocated.

The variables x[0], x[1] ... x[4] are integers, and can be assigned and operated upon like integers! OK, so far so good!

But what about x[5], x[6], ... x[55]? Can I assign to x[5], increment it, etc.?

Why? x[5], x[6], and so on are undefined. These are names but no storage has been allocated. Shouldn't access them!

NO! Program may crash.

So, one thing is, we have to take care of the size of the array. For example, if we have an integer array of size 5x this means that 5 integer variables named x0 to x4 are allocated. Now, the variables x0 to x4 are integers and they can be assigned and also they can be operated on, they can be part of other expressions and so, on. Now, what about arbitrary integers, we know that 0 to 4 are valid integers what about 5 and so, on. What happens to x5, x66 something like that. Similarly, what happens what will happen if I right **x[-1]** what are these valid. So, the answer is no, you cannot in general assume that indices other then 0 to 4 make any sense. Your program may crash and this is the most important thing in c programming when we use the array it is the main part of it, because it is not even guaranteed that a program will crash. So, you may run the program once with x of 5

let us say and the program will work fine. And you will be under the false impression that everything is correct in our program, but the next time you run it, may be your program will crash. So, it is not even guaranteed that it will crash, if it is guaranteed that it will crash, then of course you can know that there is an error, and you can go back to the code. In this case you just you have to be careful when you write the code. So, $x[5]$ and $x[6]$ are undefined, these are names, but there are no storage location that they correspond to, so, you should not access them.

(Refer Slide Time: 12:42)



Q: Shouldn't I or couldn't I access array elements outside of the array range declared?



```
int f() {
    int x[5];
    x[0] = 0;
    x[1] = 1;
    ...
    x[4] = 4;
    x[5] = 5;
    x[6] = 6;
}
```

All good ✓

Both these statements are not recommended.

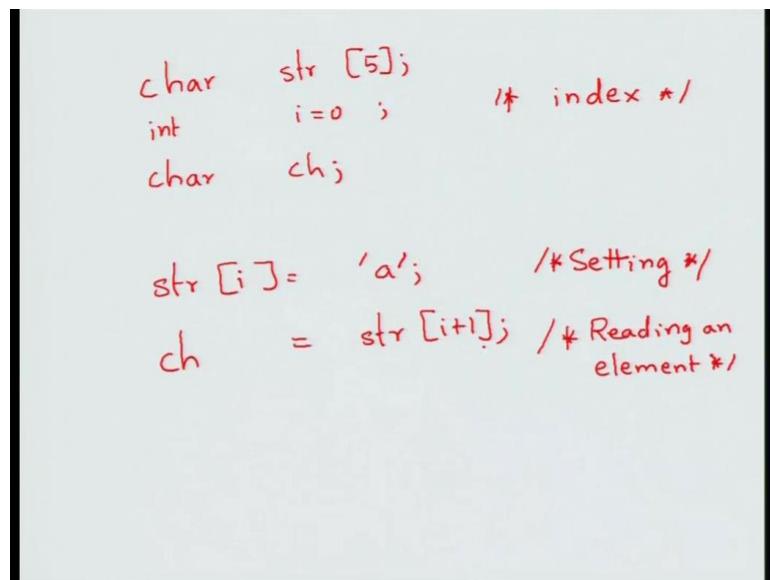
Will it compile? Yes, it will compile. C compiler may give a warning.

But, upon execution, the program may give "segmentation fault: core dumped" error or it may also run correctly and without error.

So, if you ask a very specific question, shouldn't I access them are can't I access them. So, what will happen if I write a code like this where I declare an integer array of size 5 then I know that $x[0]$ to $x[4]$ are valid locations they are the first five locations. But the problem comes with statements like $x[5] = 5$ and $x[6] = 6$ do not refer to valid locations in the array so, what will happen. So, the initial statements up to $x[4]$ are all fine, but the last two statements $x[5] = 5$ and $x[6] = 6$ lead to arrays will it compile? Yes, if you just give the source code, with these erroneous locations it will compile, but c compiler does not check that the indices are within the proper range. So, it will compile and the compiler will not tell you that there is anything wrong with there, but when you run a program the program will give something called a segmentation fault it may or may not give that. So, this is one of the most notorious errors when you program in c. So, we will see this error in greater detail when we understand something called pointers. But in general when you exceed the bounds of the array, when you go beyond the locations permissible in the array, your code may crash and the code will crash usually

with the error segmentation fault. So, if you run the program and if you see a segmentation fault this is a good indication that maybe you are referring to locations in your array that do not exist. So, you should go back and rectify the code, but the danger is that, it may not always crash. So, the only way to be really sure is to go through your source code and examine it. Your program may crash, so, we have seen certain aspects of arrays in C so far.

(Refer Slide Time: 15:04)



So, for example, let say that I declare a character array str of size 5, so, it has five characters inside it. And let us say that I use the variable i as an index into the array. So, str[0] to str[4] can be addressed using the index i. So, if I have the index, I know that I can set particular values as str[i] = 'a'. Since, i is 0 this will set the 0th element in the array to character a. Similarly, if I say ch equal to str[i + 1] it will take whatever is in the first cell in str[1] and assign it to the variable ch. So, we can set particular element in an array like this. Similarly, we can also read the value in an element and then assign it to something else. So, these are possible with the help of an array now let us consider a particular example, which is the problem is as follows. We want a character array let say a size 100 and then we have to read input which is from the key board and store them in the array, After we have stored it we should stop, once at least hundred character have been written, because that is array size or when the user first end a file. Remember that you can press <Ctrl-D> to enter the end of file.

(Refer Slide Time: 16:49)

Example with arrays: Print backwards

Problem:

Define a character array of size 100 (upper limit) and read the input character by character and store in the array until either

- 100 characters are read or
- an EOF is encountered on the terminal

Now print the characters backward from the array.

Example Input 1	Output 1
Me or Moo <Ctrl-D>	oom ro eM
Example Input 2	Output 2
Eena Meena Maine Mo	oM eniaM aneem aneE



Now, what we have to do is take the error, take the array and print it in the reverse order. Now, if you think for a little bit you can see that it is difficult to do this without an array. Instead of an array, if you are storing it in a single character, there is no way to store hundred characters in one variable and then print them in the reverse order right. Because the first character has to be printed at the end and last character entered has to be printed first. So, you need to remember all the characters, this is an intuitive reason why arrays are important for this problem. So, what is an example problem let say that we have m e or then new line then **Moo <Ctrl-D>**. So, when you reverse it, you will have oom then the new line then or emn and so, on. So, you have to reverse everything input. Similarly, if you have a string what you have to output is the exact reverse of the string including the spaces.

(Refer Slide Time: 18:09)

Read and print in reverse

1. We will write the program using just main () now.
2. There will be two parts to main: read_into_array and print_reverse.
3. read_into_array will read the input character-by-character up to 100 characters or until a <Ctrl-D> is read.
4. print_reverse will print the characters in reverse.

Overall design

```
main() {  
    char s[100];  
    /* read_into_array */  
    /* print_in_reverse */  
}
```



So, let us design the program we will just try to write the program using may. Now, there are two parts in this program the first is just to read what has been input into the array and the second part is to print the array in reverse. So, the read into array that part of the program will read the input character by character, until one of two events happen. The first is hundred characters have been input because you have declared the array of size only hundred. So, you can read only hundred characters, so, once you reach that you should stop. Otherwise, before you reach hundred characters may be the use of first a <Ctrl-D> to say that I am done with input ok. In that also, you have to say that I have done by reading the input. Now, print reverse we print the characters in reverse ok.

(Refer Slide Time: 19:07)

Let us design the program fragment read_into_array.

Keep the following variables:

1. int count to count the number of characters read so far.
2. int ch to read the next character using getchar().

**Note that getchar() has prototype int getchar()
since getchar() returns all the 256 characters and the integer EOF**

```
int count = 0;  
int ch;  
read the next character into ch using getchar();  
while (ch is not EOF AND count < 100) {  
    s[count] = ch;  
    count = count + 1;  
    read the next character into ch using getchar();  
}  
s[count]=ch;
```



An initial design (pseudo-code)

So, let us design the program for reading into the array. So, keep the following variables, one is to keep the count of how many characters I have read so far. And then I will keep a variable to store the currently read character. Now, we have touched upon the topic once, I am going to declare it int ch instead of char ch. So, I am not going to do this, this is because to getchar will give you whatever character has been read just now from the input. So, in particular they are character that can be entered can be an ASCII value which is from 0 to 255 or something. And then it can also enter the eof character the end of file character which is actually -1. So, -1 does not correspond to ASCII character. So, getchar can also read an end of file character, this is the reason why, if you are reading character through getchar and we are doing this because, the user can also enter the -1 then in order to hold at value you need an int ch rather than a character ch.

(Refer Slide Time: 22:50)

The slide illustrates the process of translating pseudo-code into actual C code. It features three main sections:

- Initial design pseudo-code:**

```
int count = 0;
int ch;
read the next character into ch using getchar();
while (ch is not EOF AND count < 100) {
    s[count] = ch;
    count = count + 1;
    read the next character into ch using getchar();
}
```
- Overall design:**

```
main() {
    char s[100];
    /* read_into_array */
    /* print_in_reverse */
}
```
- Translating the read_into_array pseudo-code into code.**

```
int count = 0;
int ch;
ch = getchar();
while ( ch != EOF && count < 100) {
    s[count] = ch;
    count = count + 1;
    ch = getchar();
}
```

A red oval highlights the "initial design pseudo-code" section. A green box at the bottom contains the text "Translating the read_into_array pseudo-code into code."

So, this is what we just mentioned, the end of file character is usually -1. So, it is not a valid key value. So, the code at the top level looks like this, we have the logic to read a next character into the ch using getchar, and then we have a let us say while loop which says that, while the character is not the end of file and the number of characters read count is less than 100. You store the character into the array increment count and then read the next character. So, please look at the structure of the loop very carefully the s is a character array. So, technically it cannot hold end of file, but then if you think about it little bit you will see that we will never encounter the situation where, you will store end of file into the s array, because suppose first character is end of file then we will not even enter the loop. Now, at any point when we enter end of file, it will be at this point right

we will read the character only here, before storing it into the array we will actually check whether it is end of file. So, we will not accidentally set the array to **-1** at any point, so, character array suffices. So, think carefully about the way this loop has been interpreted.

In particular, if I had just done this as the last line before the loop ended then, you would run into problems because you could store the end of files character into the s array by mistake so, just think about that issue. Now, here is an initial design and so, the overall design is that first you have to read into the array and then you have print it in reverse. So, let us make the read into array little bit more precise. So, we have **ch = getchar()** and because you are using the getchar function we have int ch, because it could also be an end of file. Now, the while loop says that while the ch is not end of file and the number of characters read is strictly less than hundred increment. So, you first set **s[count]** equal to the character read,, the increment count and then get the next character. So, this loop keeps on filing characters into the character array until you see either end of file or you have enter 100 characters.

(Refer Slide Time: 23:54)

Now let us design the code fragment print_in_reverse

Suppose input is HELP<Ctrl-D>

Note: width of the data-types has not been drawn to scale. char is 1 byte wide, int is 4 bytes wide (usually).

The array char s[100] `H' `E' `L' `P' [] []

s[0]	s[1]	s[2]	s[3]	s[99]		
------	------	------	------	-------	--	--

index i runs backwards in array count 4

```

int i;
set i to the index of last character read.
i = count-1;
while (i >= 0) {
    print s[i]
    i = i-1; /* shift array index one to left */
}

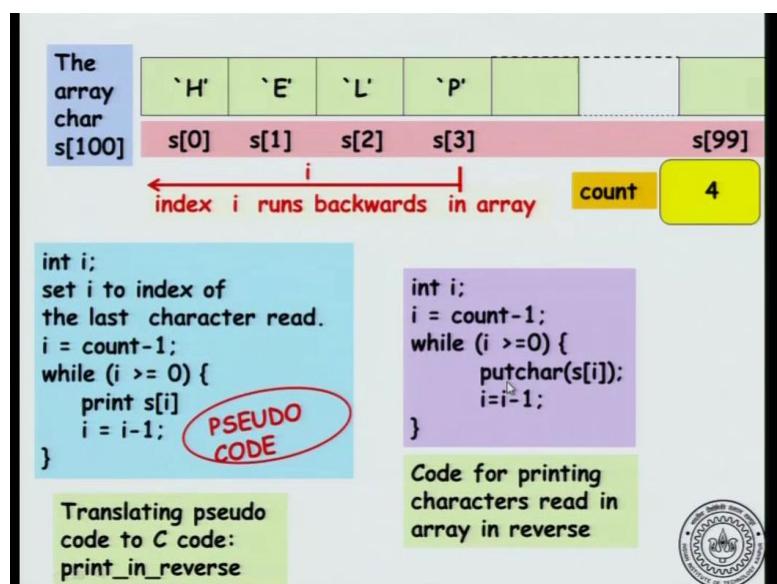
```

PSEUDO CODE

Now, let us design the remaining thing which is print in reverse. So, suppose the input is let us say to be concrete HELP and then <Ctrl-D> the end of file character. So, once you run the reading to array part, it will lead to the array looking like this s 0 will be the character h, s 1 will be e, s 2 will be l, s 3 will be p at this point you will read end of file and you will not store the end of file in the array right. So, s 0 to s 3 are valid characters and at this point, if you go back to the code count always keep tracks of how

many characters have been read. So, in particular count will be 4 when you exit the array. Now, to print it reverse, all you have to do is to start printing from s 3, then s 2, then s 1, then s 0 right. So, you read in this direction and you print in the reverse direction. So, we should be somewhat careful at this point suppose you have read the array before you enter this part, then you declare i which is the array index that we are going to use now i should be set be the index of the last character read. So, here is the tricky part notice that count is 4. So, four characters have been read therefore, the last character read is at index count **-1**. So, it is not at count index, if you say s of four that is an invalid index whereas s of three is where we should start from. So, start from **i = count - 1** that we will start at this character, now, while **i >= 0**. So, we will start from **s[3]** then print **s[5]**. So, we will print s 3 then, decrement I because you have to go back to the next to the previous character. So, i becomes 2, **i >= 0**, so, you will print s 2 decrement I, so, i becomes 1. So, you will print e, then decrement i you will come to i become 0 and you will print h you decrement once again i becomes **-1**. So, you will exit the while loop ok.

(Refer Slide Time: 26:44)



So, this is the array that we were doing and so, here is the code for printing the characters in reverse. So, here is the pseudo code where we said print s of I instead of that in c we have a particular function which will print the character which is put char. So, due to this the dual function of **getchar**. So, put char takes an character as an argument and prints it on to the standard output. So, you have int i i is set to be count **-1** because that way we will get the last index of the character in the array and then you start counting down until use print the first character and till the end of the array ok.

(Refer Slide Time: 27:37)

Overall design Putting it together



The code fragments we have written so far.

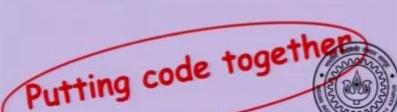
```
main() {  
    char s[100];  
    /* read_into_array */  
    /* print_in_reverse */  
}  
  
int count = 0;  
int ch;  
ch = getchar();  
while ( ch != EOF && count < 100) {  
    s[count] = ch;  
    count = count + 1;  
    ch = getchar();  
}  
  
read_into_array code.  
  
int i;  
i = count-1;  
while (i >=0) {  
    putchar(s[i]);  
    i=i-1;  
}  
  
print_in_reverse  
code
```



So, putting these two together, you have the read into array part and then you have the reverse part print in reverse part. So, when you put these two together the first thing you do is, bring all the declarations together.

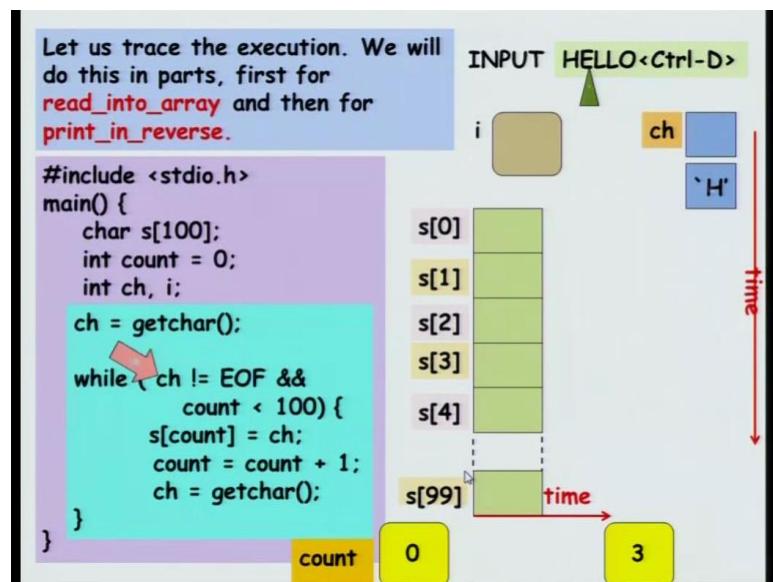
(Refer Slide Time: 27:51)

```
#include <stdio.h>  
main() {  
    char s[100]; /* the array of 100 char */  
    int count = 0; /* counts number of input chars read */  
    int ch; /* current character read */  
    int i; /* index for printing array backwards */  
  
    ch = getchar();  
    while ( ch != EOF && count < 100) {  
        s[count] = ch;  
        count = count + 1; /*read_into_array */  
        ch = getchar();  
    }  
  
    i = count-1;  
    while (i >=0) {  
        putchar(s[i]);  
        i=i-1;  
    } /*print_in_reverse */
```



So, this is the declarations for read into array as well as print to put together. Similarly, first you have to print, you have to put the code for the read into array part and then the code for the print in reverse part ok.

(Refer Slide Time: 28:13)

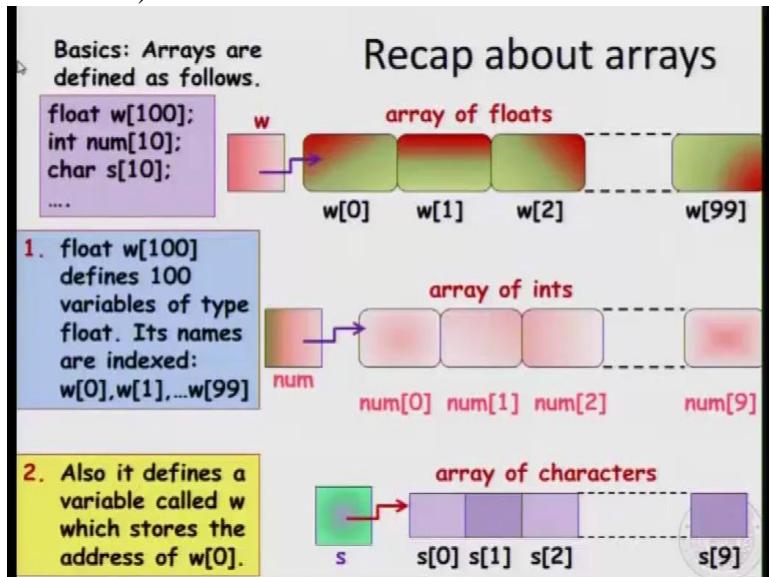


So, let us trace the execution for a small sample input. So, then we have the input is hello and then the user presses `<Ctrl-D>` for end of file, let see what will happen. So, you start reading into the array. So, `s[count]` with count equal to 0 starts setting the array. So, `s[0]` will be h and then `s[1]` will be e and so, on. So, once `ch` becomes `<Ctrl-D>` the end of file character you will exit the loop. So, the character array is hello.

Introduction to Programming in C

Department of Computer Science and Engineering

(Refer Slide Time: 00:07)



In this lecture will just talk about how to initialize arrays. So, recall that we have defined arrays as follows, if you declare an array float $w[100]$, it will declare an array of floats 100 floats consecutively allocated in memory. And we have also mentioned the fact that there is a separate box w , which will point to the first location in the array. So, it contains the address of the first location. In $num[10]$ will declare an integer array of 10 integers plus one box which will hold the address of the first location, and so on. So, the arrays names, the cells of the array or the elements of the array are index from $w[0]$ through $w[99]$ the indices start from 0. And we also mentioned that conceptually there is a separate variable called w , the name of the array which stores the address of $w[0]$.

(Refer Slide Time: 01:11)

Recap about arrays

We can specify size of arrays using constant expressions.

array of floats

float w[10*10];

w

w[0] w[1] w[2] ... w[99]

What about using variables?

```
int size;
float w[size];
scanf ( "%d", &size );
for ( i=0; i<10; i+=1 ) {
    scanf ( "%f", &w[i] );
}
```

Not allowed in Ansi C.

Allowed in C99 and later.

Let's avoid this feature.

Now, it is not important that we use numbers as the size of the arrays, we can also use constant expression; for example, we can say `float w[10 * 10]`. So, instead of saying 100, we can give an arithmetic expression which evaluates 200, and this has the same effect it will evaluate an array of 100 elements starting from `w[0]` to `w[99]`. And there is a separate box called `w` which moves to the address of the first location, but what about using variables or variables size arrays, this is feature that we often wish we had. So, what do I mean by that? I could declare the following code `int size;`, and then `float w[size]` and I could say `scanf ("%d", &size);`. So, user enters the size of the array, and then I can enter 10 elements into the array for example, but here the size of the array itself is a variable which depends on the user input. And we often wish that we would be able to allocate variable size arrays, but this is not allowed in Ansi C, it is allowed in the latest versions of C 99 C 11 and so on. We will avoid this feature for the purposes of this course, let us assume that array means they are declared to be of constant size. By constant size you can give the size as a particular number or you can give it as a constant expression, that is an arithmetic expression involving constants, but not general expressions.

(Refer Slide Time: 03:16)

How can we create an int array num[] and initialize it to:



Method 1 `int num[] = {-2, 3, 5, -7, 19, 103, 11};`

1. Initial values are placed within curly braces separated by commas.
2. The size of the array need not be specified. If unspecified, it is set to the number of values we provide.
3. Array elements are assigned in sequence in the index order. First constant is assigned to array element [0], second constant to [1], etc..

Method 2 `int num[10] = {-2, 3, 5, -7, 19, 103, 11};`

Specify the array size. size must be at least equal to the number of initialized values. Array elements assigned in index order. Remaining elements are set to 0.

Now, let us just look at how can we create an integer array num and also initialize it to particular values; for example, I want the num array to look like the following, it contains 7 cells having the values - 2, 3, 5, - 7, and so, on. Now I know that if C did not allow me to initialize arrays when I declared it, I could declare the array as int num 7 and then I will just write `num[0] = - 2, num[1] = 3, and so on until num[6] = 11`. So, here is a way that I can create an array and ensured that this state is reached, but is there more convenient way of doing it. Can I start off the array with these contents. So, C allows you two ways do it. The first is I declare an int `num[]` and then specify what are the initial values, so `- 2` so on up to `- 11` within `{}`.

So, this is one way to that C allows you to do this. The initial values are placed within curly braces and separated by `,`, the size of the array need not be specified. So, I need not say that `num[]` has size 7, it will allocate an array with enough space to hold 7 integers. Array elements are assigned in the order that you specified. So, `num[0]` will be `- 2`, `num[1]` will be `3`, and so on. So, it is done in a reasonable manner. This is also another way to do it, which is slightly different from way above, I can declare the size of an array. So, I declare an array of size 10, and then give this initial value. What will happen in this case, is that it will make sure that the size of the array is at least equal to the size of list that I have given. So, I have given 7 elements, and I have declared an array of size 10, 7

is less than 10. So, it is fine. So, I can declare an array of size 10, I should give a value, I should give values at most 10 in number. So I can give a 10 or below. In this case, I give 7 numbers. So, what happens is that, array is initialized in the order elements given **num[0]** will be - 2, **num[1]** will be three and so on, until **num[6]** will be 11. 7 elements are filled; the remaining 7 elements are unspecified. So, they will be initialized to 0.

Now let me just remained you that if I had just declared an array **int num[10]**, and then put a semicolon. So, I had just declare an array without saying any initialization at all, then you should assume that the array contains or arbitrary values, you should assume that array contain Junk values, but if you initializes an array of size 10, and give only 7 initialization values, then the C standard gives you the guarantee that the remaining elements are initializes to 0. So, they are not junk.

(Refer Slide Time: 06:52)

Recommended method: array size determined from the number of initialization values.

```
int num[] = {-2, 3, 5, -7, 19, 103, 11};
```



Is this correct? `int num[100] = {0, -1, 1, -1};`

YES! Creates num as an array of size 100. First 4 entries are initialized as given. `num[4] ... num[99]` are set to 0.

num	0	-1	1	-1	0	0	...	0
-----	---	----	---	----	---	---	-----	---

Is this correct? **NO! it won't compile!**

```
int num[6] = {-2, 3, 5, -7, 19, 103, 11};
```

Why?

1. num is declared to be an int array of size 6 but 7 values have been initialized.
2. Number of initial values must be less than equal to the size specified.

The recommended method to initialize an array is to give the list of initial values, and let the compiler decide what the size of the array it should be. So, if you give 7 initial values, it will decide that the array is of size 7. Now is the following code correct, if I declare an array of size 100 num and give four initial values. So, this is correct, it creates num as an array of size 100, the first four entries will be initialized as given. So, **num[0]** will be 0, **num[1]** will be - 1, **num[2]** will be 1, **num[3]** will be - 1, and then **num[4]** until

`num[99]`, they are all set to 0. So, after the initializations the array will look as follows; the first four values are what we given and the remaining value are 0's. Now is the following code correct, `num[6] =` and then you give a list of 7 values to initialize, is this correct? The answer is no, it will not compile. So, if you right this code, and compile it using gcc, it you will get a completion error. Why is that? We have declared an array of size 6, but we have given 7 initial values. So, there is no way to do this. So, the rule of thumb is that either give no size for the array, and let the compiler figure out or if you do give a size it has to be at latest 7, which is the number of values that you give, it can be 10, it can be 100, but it cannot be less than 7. Now just like size can be not just numbers it can also be constant expressions, we can also have constant expressions as initialization values ok.

(Refer Slide Time: 08:54)

Initialization values could be constants or **constant expressions**. Constant expressions are expressions built out of constants.

```
int num[] = { 109, 'A', 7*25*1023 +'1' };
```

Type of each initialization constant should be promotable/demote-able to array element type.

E.g.,

```
int num[] = { 1.09, 'A', 25.05};
```

Float constants 1.09 and 25.05 downgraded to int

Would this work?

```
int curr = 5;
int num[] = { 2, curr*curr+5};
```

YES! ANSI C allows constant expressions AND simple expressions for initialization values.
"Simple" is compiler dependent.

A cartoon character with a question mark above their head is shown next to the last point.

So, for example, I can give `num[] = 109`, then the character value A, character value A means it will take the ASCII value of A, 65 or whatever it is. So, the first number will be 109, the second number will be 65, let us say if the ASCII value A is 65, and the third value will be `7 * 25 * 1023 + '1'`. So, whatever the ASCII value of the character one is let say 90 or something. So, it will be added two this, constant expression, and it will be initialize to that value; `num[]` two will be the result of evaluating this expression. So, the type of each initialization constant should be promotable or demutable to the array

element type. So, the each value in the initialization list should be compatible with let us say integer, because we have declared the array of size of type integer. So, what do I mean by that, for example I can initialize an array `num[]` with initialization list 1.09, then , A , 25.25. So, this is ok, because the floating point values can be downgraded to integers. So, may be this will be initialize to one then whatever the ASCII value, A is let us say 65, and then 25.

Now, these are about constant expression. What about expression involving variables when we initialize an array. So, can we do something like this. `int curr = 5,` and then the `num[]` array is initialize with `{2, curr*curr+5}`, will this work. The answer surprisingly is yes that it will work on most compilers. So, the ANSI C allows constants expressions, and simple expressions for initialization values. Now simple is of course dependent on which compiler we are using. So, if you write a code, and compile using gcc with such an initialization may be or code will compile, and the movement you compile your code with a different compiler it may not compile.

So, earlier I had said that the size of the array cannot be initialized using variable expressions. In ANSI C that is forbidden, but the initialization value, so the value that goes in to the array can involve variable expressions, this may or may not be supported. So, it is safe to assume that both the size of the array, and the initialization value can be done only using constant expressions, even though some compilers allow simple initialization values using variable expressions.

(Refer Slide Time: 12:08)

Character array initialization

Character arrays may be initialized like arrays of any other type. Suppose we want the following char array.

We can write: `s[] = {'I', ' ', 'a', ' ', 'm', ' ', 'D', 'O', 'N', '\0'};`

BUT! C allows us to define **string constants**. We can also write:

`s[] = "I am DON";`

1. "I am DON" is a **string constant**. The '\0' character (also called **NULL char**) is automatically added to the end.
2. **Strings constants in C are specified by enclosing in double quotes e.g. "I am a string".**

Now how do we initialize character arrays? Character arrays can be initialized like arrays of any other type, suppose we want the following array. `s[] = 'I', ' ', 'a', 'm', '`, etcetera. So, I can initialize it just like initializing the other array, I will not specify their size of `s` and then give these characters, I am DON. So, this is another way to specify and the last character is a null character, but C also allows you to define what are known as string constants. So, we can also write `s[] = "I am DON"`, but now with in double quotes. So, this is known as a string constant, the null character is an implicit ending character inside a string constant. So, it is automatically added to the int. Now the string constants in C are specified by enclosing it in double quotes.

Introduction to Programming in C

Department of Computer Science and Engineering

(Refer Slide Time: 00:05)

Character array initialization

Character arrays may be initialized like arrays of any other type. Suppose we want the following char array.

s		'I'	' '	'a'	'm'	' '	'D'	'O'	'N'	'\0'
		s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]

We can write: `s[] = {'I', ' ', 'a', 'm', ' ', 'D', 'O', 'N', '\0'};`

BUT! C allows us to define **string constants**. We can also write:
`s[] = "I am DON";`

1. "I am DON" is a **string constant**. The '\0' character (also called NULL char) is automatically added to the end.
2. **Strings constants in C are specified by enclosing in double quotes e.g. "I am a string".**

In this video, will talk about initializing character arrays which are a special kind of arrays and has more features then, when it comes to initialization as compared to arrays of other type. So, they may be initialized just like any other array and suppose I want to initialize an array to the following values, the first is I second is a space character then a then m and so on. And the final character is a null character which will be given as '`\0`' now we can write s. So, character `s = []` without specifying a size and then followed by the list of characters. Notice that each character is enclosed in single quotes right.

So, the space character is a '' and so on until the last which is a null character, which is '`\0`'. But C also allows us to define what are known as string constants. So, in order to initialize an array a character array I can also write character `s[] = "I am DON"`. So, I am DON is a string constant in a every string constant that is a string enclosed in double quotes the null character is automatically added to the end. So, I want you to note this difference that, here we explicitly gave a null character at the end here we do not have to give that. Now C; C string constant are specified by enclosing some text with in double quotes for a example, "`I am a string`"

(Refer Slide Time: 02:18)

Printing strings

We have used string constants many times. Can you recall?

printf and scanf: the first argument is always a string.

1. printf("The value is %d\n", value);
2. scanf("%d",&value);

Strings are printed using %s option.

E.g. 1 printf("%s", "I am DON");
Output
I am DON

E.g. 2 char str[]="I am GR8DON";
printf("%s",str);
Output
I am GR8DON

str str[0] str[2] str[4] str[6] str[8] str[11]

'I' ' ' 'a' 'm' ' ' 'G' 'R' '8' 'D' 'O' 'N' '\0'

State of memory after definition of str in E.g. 2. Note the NULL char added in the end.

This NULL char is not printed.

Now, we will talk about how do we print strings we have used string constants many times. So, just take a moment to think back to see whether you can figure out where we have used string constants. So, we have used for them for example, in printf and scanf the first argument of a printf or a scanf was always a string constant, because if you recall we had some text in which involved special characters like new line. It involves formats specifiers as like `%d`, but whatever it was, it was a bunch of characters. So, it was a text inside a pair of double quotes that is a string constant. So, the first argument is the string constant followed by what all arguments we want to print. Similarly, even for scanf we had some say formats specifier enclosed in double braces, so, that is a string constant and then you say and value. So, strings are printed using the `%s` option. So, any of the basic data types in C can be easily printed using the printf statement if you give the correct format specifier.

So, if you have a string constant you can print it using the `%s` option. For example, if I want to print the string “I am DON” then what I can do is I can say `printf("%s", "I am DON")`. And this will print I am DON which is exactly what I wanted to print. Now, what if I initialize a character array character `str[]` without specifying the size. I initialize it using a string constant I am great DON within double quotes. Then I print it using `printf("%s", str)`, will this work? And the answer is yes it will work, because C will consider this as a string constant and it will print it using `%s` and you will get the correct output. So, state of memory after definition of this string in example two is that, it has a

list of all these characters ‘I’, ‘ ‘, ‘a’, ‘m’, ‘ ’ and so on and note the implicit null at the end. So, even though the double quotes ended just after n when you stored it in an array there is an implicit null that is inserted at the end of the array. So, when you print it will print until the null character. So, null character itself at the end of the string is not printed when you print it using `%s`.

(Refer Slide Time: 05:36)

Consider the fragment.

Strings

This defines a constant string, i.e., character array terminated by, but not including, '\0'.

What is printed? Let us trace the memory state of str[].

str	str[0]	str[2]	str[4]	str[6]	str[8]	str[11]					
'I'	' '	'a'	'm'	' '	'G'	'R'	'8'	'D'	'O'	'N'	'\0'

Output
I am

1. A string is a sequence of characters terminated by '\0'. This '\0' is not part of the string.
2. There may be non-null characters after the first occurrence of '\0' in str[]. They are not part of the string str[] and don't get printed by printf("%s", str);.

Now, let us look at the following fragment to understand slightly in a deeper way what `%s` thus when you print it using `printf`. So, suppose I declare an character array using character `str[] = "I am GR8DON"`. So, this is initialized using the string constant which means that after the last end, there will be a null character in the array. Now, I initialize it. I said `str[4] = '\0'` note that there are 11 non-null characters in the string constant. So, this goes from `str[0]` to `str[10]` followed by `str[11]` which is a null character. So, now, I said `str[4] = '\0'`. So, somewhere in the middle of the string I put a null character. What will happen if I print it using `printf %s`? So, let us see what happens here I declare the array and initialize it using a string constant.

So, it has all these letters followed by a null at the end. Then when I said `str[4] = '\0'` what it does is it goes to the fourth location in the array and changes that to null. So, what that does is there was a space there before, but now you insert a null character there. After the null character there are other non-null characters and then there is a second one. What will happen when you print? It will just print I am and stop that it will

not print the remaining characters and why does that happen? So, string of C as far as C is concerned is a sequence of characters terminated by a null, this null is not part of the string. So, they may be non-null characters after the first occurrence of null in str, but they are not considered part of the string str their part of the character array. But when you look at str as a string it is just till the first null character. So, when you print it using `%s` only the part until the first null is printed. So, that is considered the string the character array is bigger.

(Refer Slide Time: 08:07)

So do we lose the chars after the first '\0' ? Where did they go?

Of course not, they remain right where they were. They were not printed because we used %s in printf. Let's take a look.

<code>str</code>	<code>str[0]</code>	<code>str[2]</code>	<code>str[4]</code>	<code>str[6]</code>	<code>str[8]</code>	<code>str[11]</code>						
	'I'	' '	'a'	'm'	'\0'	'G'	'R'	'8'	'D'	'O'	'N'	'\0'

```
char str[]="I am GR8DON";
str[4]='\0';
printf("%s",str);
```

Output: I am

```
int i;
for (i=0; i < 11; i++) {
    putchar(str[i]);
}
```

Output: I amGR8DON

The character '\0' may be printed differently on screen depending on terminal settings.

So, it will just print I am and stop there. So, do I lose the characters after the first null and where do they go? Well, of course they do not go anywhere they remain where they were. So, what is the new state of the array? The new state of the array is **I am** and then there is a null and then there are some other characters. So, if I print it using `%s` it will only come up to I am and then stop there. So, is there any way to print the remaining characters? Of course, there is a way right. So, if I print that using `%s` I will get I am, but I could easily write a loop like this. I will say `int i` and then for `i = 0`, until `11`, `i++` and then `putchar(str[i])`. So, this will print the character `str[0]` str 1 and so on up to `str[11]`, regardless of whether that character is null or not if it is null it will do something, but it will still go on to the next character.

If you run this what you will see is, it will print the first character which is I, then it will print the second character which is space, then it will print the third character which is a.

So, these three are printed as they are and then m and the fifth is a null character. What do you mean by printing a null character? It may not print anything. So, it may be just kept, but then it goes on to the next character GR8DON and there it stops, because it does not print the eleventh character. So, the null character in this example is not printed. Now, the way the null character is treated on different terminals may be different. So, on some Linux terminals if you ask to print null character it will just not print anything, but other character terminals may print them in different ways.

Introduction to Programming in C

Department of Computer Science and Engineering

(Refer Slide Time: 00:09)



In this video, we will *t with a topic that is considered one of the trickiest concepts in C. These are what are known as pointers. We will just recap what we know about arrays, because arrays and pointers are very closely related in C.

(Refer Slide Time: 00:23)

The memory allocated to array has two components:

A consecutively allocated segment of memory boxes of the same type, and

A box with the same name as the array. This box holds the address of the base (i.e., first) element of the array.

OK, Consider the definition.

`int num[10];`

`num num[0]num[1]num[2] ... num[9]`

This definition for `num[10]` gives 11 boxes, 10 of type int, and 1 of type address of an int box.

1. We represent the address of a box x by an arrow to the box x . So addresses are referred to as pointers.
2. The contents of an address box is a pointer to the box whose address it contains. e.g., `num` points to `num[0]` above.

too much theory. Give examples please

Hmm..

The memory allocated to any array has two components. First is there are a bunch of consecutively allocated boxes of the same type. And the second component is there is a box with the same name as the array. And this box contains the address of the first element of the array. So, let us be clear with the help of concrete example. So, let us consider a particular array of size 10 declared as int `num[10]`. Conceptually, there are 10 boxes from `num[0]` through `num[9]`. These are all containing integers. Plus there is an additional eleventh box – `num`. So, it has the same name as the name of the array. And it contains the address of the first location of the array. So, it contains the address of `num[0]`. These are `num[0]` through `num[9]` are located somewhere in memory. So, maybe this is memory location 1000. So, `num` contains the number 1000, which is supposed to indicate that, the address of the first location in the array is 1000 or `num` points to the memory location 1000. So, conceptually, this gives 11 boxes, which are 10 integer boxes plus 1 box, which holds the address of the first box.

Now, we represent the address of a box `x` by an arrow to the box `x`. So, addresses are referred to as pointers. And this is all there is to C pointers. Pointers in some sense are variables, which hold the addresses of other variables. That is an exact description of the concept of pointers. Now, we will see now what that means and what can we do with pointers.

(Refer Slide Time: 02:23)

What can we do with a box? e.g., an integer box?

```
int num[10];
```

True! But we can also take the address of a box. We do this when we use `scanf` for reading using the `&` operator.

ptr would be of type address of int. In C this type is `int *`.

```
int * ptr;
ptr = &num[1];
```

That's simple. We can do operations that are supported for the data type of the box.

For integers, we can do + - * / % etc. for each of `num[0]` through `num[9]`.

OK. Say I want to take the address of `num[1]` and store it in an address variable `ptr`.

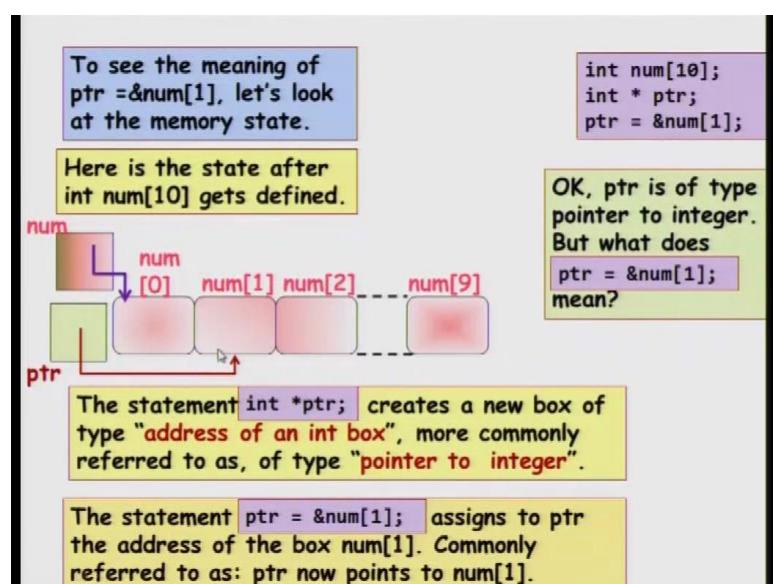
```
ptr = &num[1];
```

But what is the type of `ptr`? And how do I define `ptr`?

Let us just step back a minute and say what can we do with a particular box or particular variable in memory, which is an integer. So, that is very simple. For example, you can scanf into that box; you can print the value in that box; you can do arithmetic operations on that box like plus, division, %, and so on. And you can do this for each of the boxes from `num[0]` through `num[9]`, because each of them by itself is an integer. But, we will also see a new operation, which is that, you can take the address of a box. So, we have already done this when we did scanf. So, we mentioned & of a variable. So, we will see these & operator in somewhat more detail.

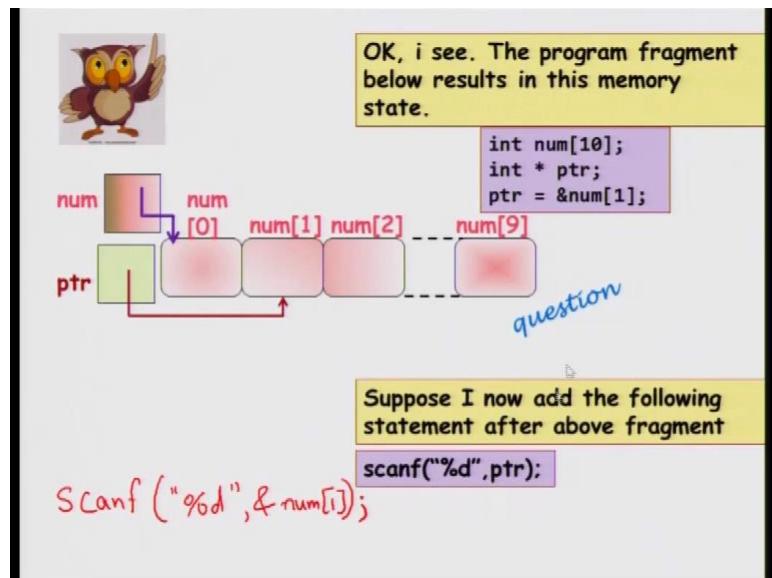
So, suppose I want to take the address of `num[1]` and store it in an address variable `ptr`. So, what I am essentially saying is that, you can say `ptr = &num[1]`. So, `num[1]` is an integer box; it is an integer variable; & of `num[1]` is the address of that integer in memory. So, you assign it to the variable `ptr`. But, every variable in C needs to have a type. What is the type of `ptr`? And how do you declare or define such a type – such a variable? Now, `ptr` holds the address of an integer. In C, you denote that by saying that, the type of `ptr` is `int *`. So, here is a new type that we are seeing for the first time. We are saying `ptr` is of type `int *`. Just like you can say that, if I have `int a`, you can say that, `a` is of type `int`. In this case, we can say `ptr = &num[1]`.

(Refer Slide Time: 04:40)



We have discussed right now we have `int num[10]`, `int *ptr`, and `ptr = &num[1]`. So, `ptr` is the pointer to an integer. But, what does `ptr = &num[1]` really mean? So, let us look at the memory status once we declare this array. So, we have `num`, which is the address of the first location. And then we have somewhere in memory, we have 10 consecutive locations corresponding to the array – `num[0]` through `num[9]`. Now, I declare `int *ptr`. So, I create a box. Now, this box is supposed to hold the address of some integer variable. So, `ptr` is of type address of an integer box or more commonly referred to as pointer to integer. The statement `ptr = &num[1]` says that, now, points to `num[1]` or `ptr` contains the address of `num[1]`. And pictorially, we denote an arrow from `ptr` to `num[1]` just like I denoted an arrow going from `num` to `num[0]`, because the name of the array is a pointer to the first location of the array. The name of the array is a box, which holds the address of the first location of the array. Similarly, `ptr` is a box, which holds the address of `num[1]`. So, we say that, `ptr` points to `num[1]`. And we denote it pictorially by an arrow.

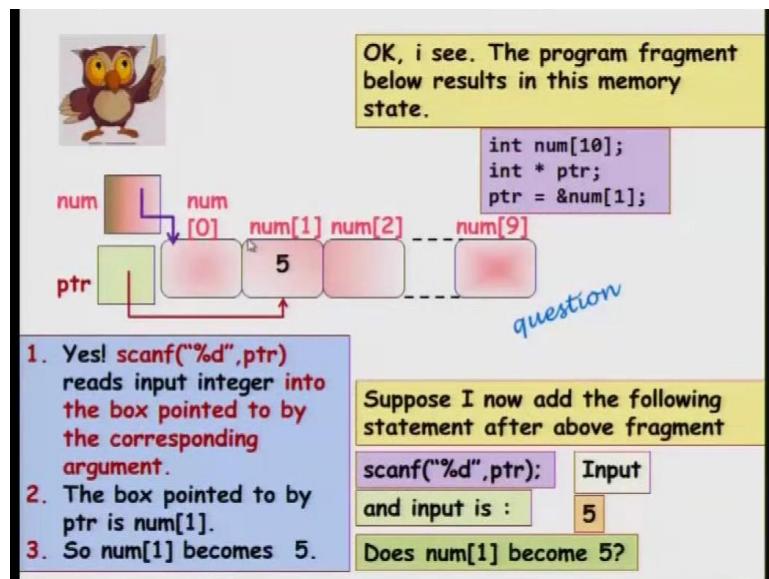
(Refer Slide Time: 06:11)



The program status is like this – state is like this. Now, suppose I add one more statement after all these three statements; I say `scanf("%d",ptr)`. Now, earlier when we declared an array and we read into an array directly, I said that, you can do the following. I can write `scanf("%d",&num[1])`. So, this will value whatever the user input into the first array

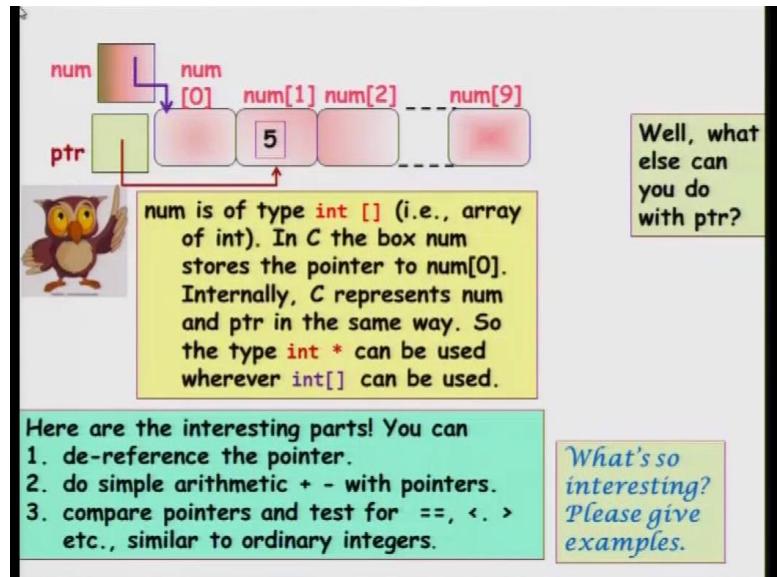
using the `&` operator. Now, `ptr = &num[1]`. So, it is a reasonable thing to ask – can I say `scanf("%d",ptr)`? There is no `&` operator here because `ptr` is `&num[1]`. This was our original statement and this is our new statement.

(Refer Slide Time: 07:15)



And the answer is yes, you can do it. Suppose the input is 5, does `num[1]` become 5? So, `scanf("%d",ptr)` really does work like `scanf("%d",&num[1])`. So, it reads the value input by the user and it looks up `ptr`. So, it is an address. So, it goes to that address and stores it there. So, now, we can clarify a long standing mystery, which is the `&` operator in the case of `scanf`. So, we can say that, `scanf` second argument is a pointer; which says where should I put the input by the user? For example, if I have float variable and I `scanf` as `%f` and then sum address of a float variable, it is done similar to reading an integer into an integer variable. What `scanf` takes is an address of int variable or float variable as it may be. If you have a `%d`, then it takes a pointer to an integer variable and takes the input value by the user and puts it into that address. So, as far as `scanf` is concerned, it does not matter whether you gave it as `&num[1]` or whether you initialized `ptr` to `&num[1]` and then gave `ptr`. It is an address and it will put the integer input by the user into that location. So, `num[1]` indeed does become 5.

(Refer Slide Time: 08:55)

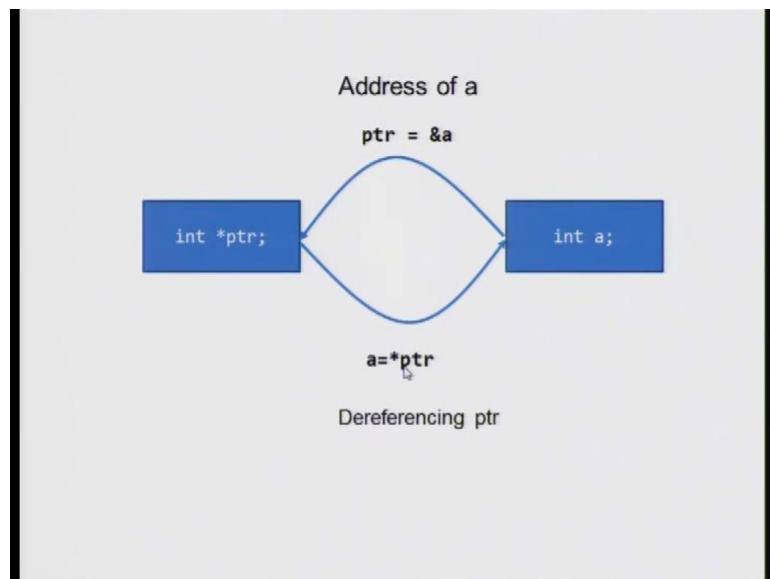


So, the location of the `num[1]` is now containing the value 5. Now, what else can you do with a pointer? Num is an array; it is of type `int []` – pair of square brackets. And in C, the box num contains the address of the first location of the array, which is `num[0]`. So, internally, as far as C is concerned, the address of `num[0]` is just like address of any other integer location. So, the type `int *` can be interchanged with `int []`. So, you can think of num itself as just a pointer to an integer; or, you can say that, it is a pointer to an array; which gives you the additional information that, the next 10 values are also integers. If you just say pointer to an integer, the next location may be something else. But, internally as far as C is concerned, an array name num can also be treated as pointer to an integer.

Now, here are some other interesting things that you can do with pointers. Whenever you declare a data type, you also define what all operations can you do with a date type. So, 2 and 3 are fairly simple; we have already seen it with integers, floating points and so on. You can do simple arithmetic `+` and `-` with pointers. You cannot do `*` and `/`. You cannot do that. But, you can do `+` and `-`. Similarly, if you have two pointers, you can test for `==`, you can test for `<`, you can test for `>` and so on as though you are comparing ordinary integers. So, 2 and 3 are what we have seen before; except that, in 2, you cannot do multiply and `/` and `%`. All these things are not done with pointers. But, addition and

subtraction can be done. But, there is a new operation, which is dereferencing a pointer. We have not seen this operation before with earlier data types.

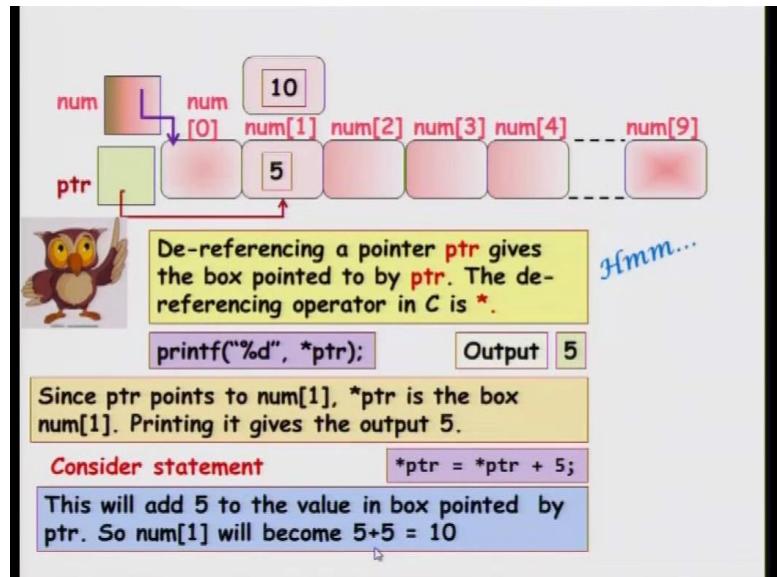
(Refer slide Time: 11:10)



What is dereferencing? Let me pictorially represent what it does. Suppose you have `int *ptr` and `int a`. So, `a` is an integer variable and `ptr` is a pointer to `int`. If I want to store the address of a `int` `ptr`, I do it as follows: `ptr = &a`. So, this means that, take the address of `a` and store it in `ptr`. So, now, you can say that, `ptr` points to `a`. Now, I can also think of a reverse operation; which is `ptr` contains some address. Go look up that address; so that will be an `int`. And store that value in `int`.

So, that is what is known as the `*` operator – `a = *ptr`. This means that, `ptr` is an `integer` pointer. So, `ptr` will point to a location, which contains an `integer`. `*ptr` will take the contents of that location and store it in `a`. So, this is known as the dereferencing operator. So, the address operator takes an `integer` variable and stores the address in a pointer. The dereferencing operation takes a pointer; looks up that address; and stores the value in `a`. So, you can visualize the `&` operator and the `*` operator as sort of reverse operations of each other. `&` takes an `integer` and takes the address of that; `*` takes a pointer and takes the value of the address pointed to that.

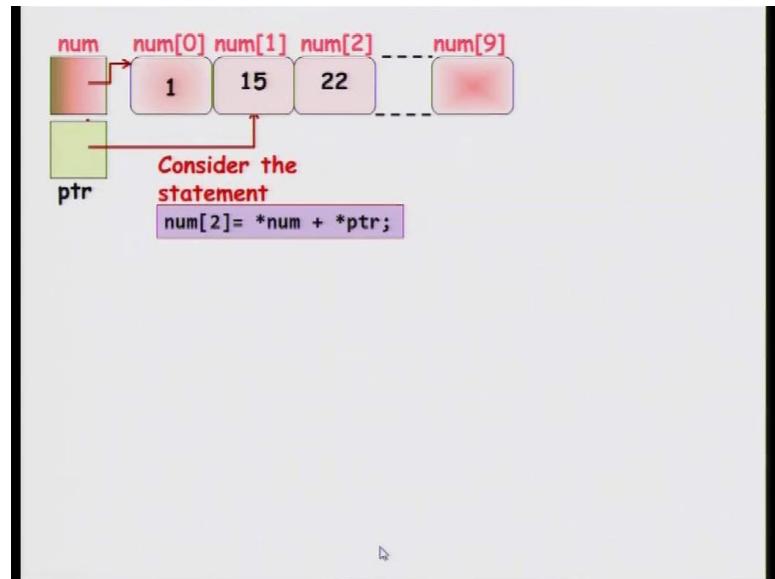
(Refer Slide Time: 12:52)



Dereferencing a pointer therefore gives you the value contained in the box pointed to by the pointer. The dereferencing operator is `*`. So, if I say `printf %d` * pointer, what it will do is – look up the location pointed to by `ptr`. In this case, it is this integer box. The contents of that box is 5 and it will be printed. So, the output will be 5. Not for example, the content of `ptr`. So, the content of `ptr` may be like 1004; it will not print 1004; but what it is supposed to do is look up the location 1004; it contains the value 5; print that value. So, `*ptr` is the box `num[1]`. And printing it gives you the output 5.

Now, can I consider a statement like `*ptr = *ptr + 5`? This is perfectly legal. What this will do is `*ptr` is an integer value. It is equal to 5, because look up this location `ptr`; that is an integer; take that value; which will be 5. So, this will be $5 + 5 - 10$. And where do you store it? You store it in the integer variable corresponding to `*ptr`. The integer variable corresponding to `*ptr` is `num[1]`. So, I would have normally said `num[1]` equal to `*ptr + 5`; but `num[1]` is the same as `*ptr`. So, I can say `*ptr = *ptr + 5`. So, this will look up that location; add 5 to its contents; and store it in that location. So, `num[1]` will now become 10.

(Refer Slide Time: 14:46)



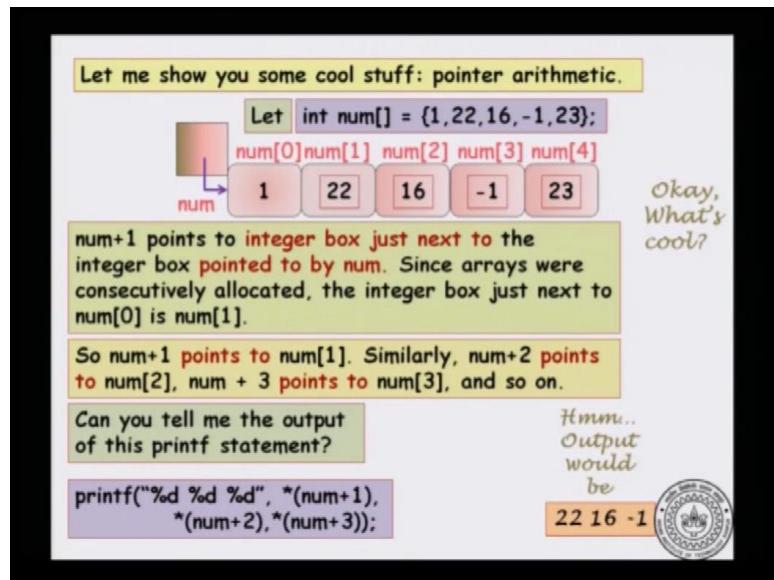
Similarly, you can consider other examples. For example, I can consider a statement like num of 2 equal to `*num + *ptr`. The novelty here is that...

Introduction to Programming in C

Department of Computer Science and Engineering

In this video let me show you some cool stuff which is pointer arithmetic which helps you to understand the relationship between pointers and arrays in C.

(Refer Slide Time: 00:13)



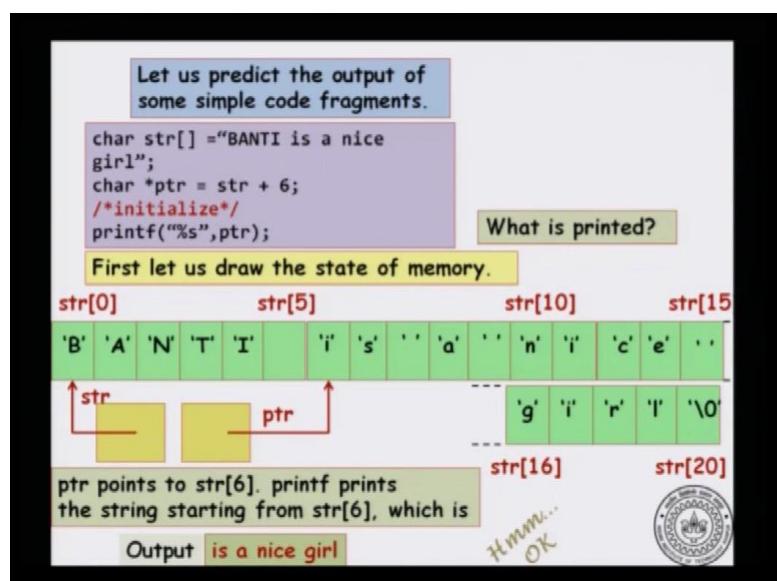
So, let us consider an array declared as follows `int num`, and then it has 5 numbers in the initialization list. So, the array will be initialized as follows; there are 5 consecutive integer locations in memory with the given elements + there is a 6th cell which points to the first location in the array. So, `num` points to the first location in the array. If `num` points to the first location, then you can do the following operator `num + 1`. So, `num + 1` with point to the integer box write mix to the integer box pointed 2 by `num`.

And we also know that arrays are consecutively located. So, the integer box next to `num` is exactly `num[1]`. So, `num + 1` points to `num[1]`. Similarly `num` of `num+2` points to `num[2]`, and so on. Until `num+4 = num[4]`. So, this particular box, for example, `num[4]` can be accessed in two ways; you can write `num[4]` or you can write `*(num+4)`. Can you tell me the output of the following printf statement. So, think about this for a minute, you have 3 integers to print using `%d %d %d`, and what are to be printed are `*(num+1)`, `*(num+2)` and `*(num+3)`. So, think about it for a minute...

`num+1` is the address, which is the second integer box in the array, `num + num` points to

the first location therefore num + 1 points to the second location, star is the dereference operator on a pointer. So, * of this pointer means go to that location which is this location, and get the value in that location, which is 22. Similarly num+2 is the box 2 boxes away from the first box in the array. So, 2 boxes away from num, that happens to be num+2 which is the... And then get the value there which is 60. Similarly *(num+3) will give you -1. so the output would be 22 16 -1. So, in this printf statement we have used two concepts. One is getting to a different pointer from a given pointer using pointer arithmetic operator +, so we have used + here. The second operator that we have used is * on a given on a given pointer. So, + will tell you go to the next integer location, and * will tell you for a given integer pointer give me the value in that location.

(Refer Slide Time: 03:31)



Now, let us look at the slightly different array. What happens if you have a character array. So, I have char str array which is initialized to let say given string BANTI is a nice girl, and then I have a character pointer. So, char *ptr and it is assigned str + 6, it is initialized to str + 6, what will happen here?

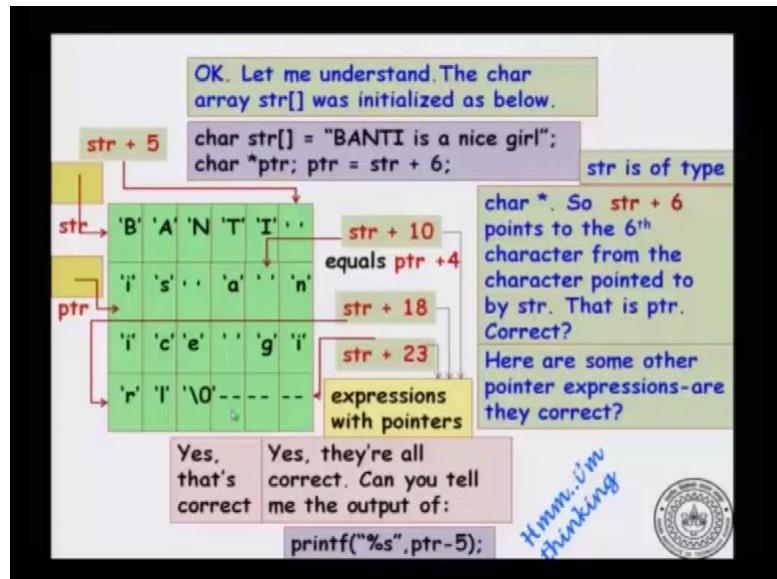
What is different about this example is that, earlier I said that in an integer array + 1 for example, would go to the next integer location in memory. So, wherever num was num+1 would go to the next integer location here, str is a character array. So, it has to go

to the next character location, and that is exactly what it does. So, what is printed? Lets first consider the state of the memory. So, you have an array which is a character array, it starts from `str[0]`, and goes on up to `str[20]`. So, there are 19 characters followed by the null character. Why is the null character there, because I initialize the two a string constant; every string constant has a null character implicitly at the end.

So, this is the straight of the str array. Now I say that I declare a pointer, the pointer is pointing to a char. So, it is a `char *` pointer and what is the location it points to it points to `str + 6`. Str is a point out to the first location of the character array, and `+ 6` would jump 6 character locations away from `str[0]`. So, you would reach this character. The important difference between this example at the previous example is that, if you declared an integer array `+ 1` would jump 1 integer location `+ 6` would jump 6 integer locations. Here since such a character array `str + 6` would jump 6 character locations. So, how the `+` operator is interpreted in the cases of pointer depends on what array am I pointing to right now? Now what will happen with the `printf` statement? So, if I say `printf %` as `ptr` what will happen?

So, `ptr` points to `str[6]`. So, `printf` will print whatever string is starting from that location until the first null character. So, it will start printing from this `i`, and then go on tip printing till it reaches the null characters. So, the output will be just is a nice girl. So, when you want to `printf` it is not important that you start from the absolute beginning of the array. We can start from arbitrary location in the character array, and if you say `printf %`, it will start from there and go on and print until the first null character.

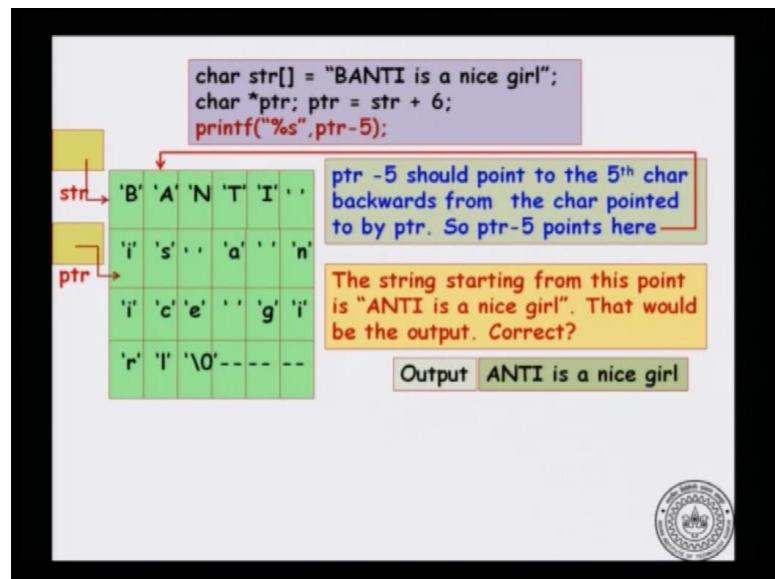
(Refer Slide Time: 06:42)



So, let us look at it once more. So, it was the code that you had, and let say that the one-dimensional array for this seek of convenience, I will just... So, it like this. It is actually in a row, but here is the first part, here is the second part, and so on. So, when I say str, str is a character array, and ptr + 6 would go 6 locations away from the first location. So, str is pointing to then first location in the array, it will go to the 6th location in the array ptr, and ptr is pointing to the 6 location.

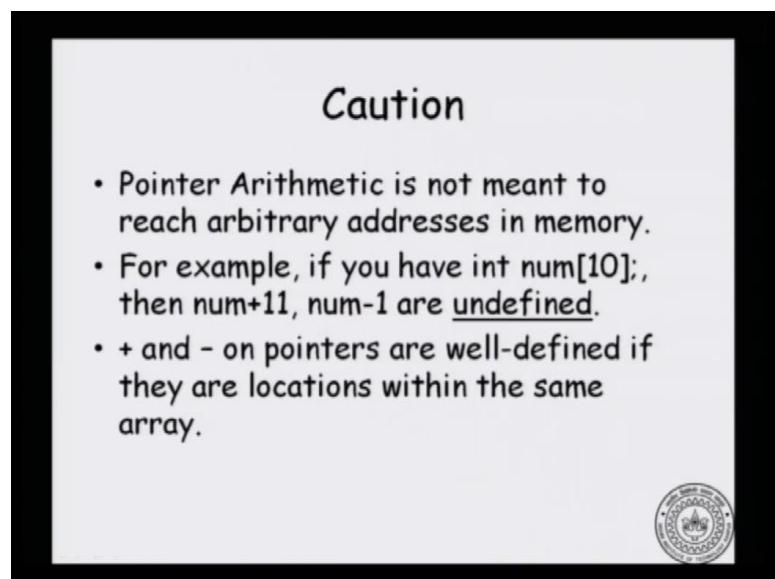
So, you can ask more expressions do the make sense, can I say str + 5 is this location. Similarly can I say str + 10 is this location, and so on. So, these are all correct expressions. Now can you tell the output of printf "%s" ptr - 5, we have talked about + operator on pointers. So, it will whatever the nature of array that the pointer is pointing to it will jump n locations away from it. So, if I say ptr + n, it will jump n locations of that type away from it. So, by the same logic can I argue that if I do -5 ptr - 5 can I say that it will go 5 locations previous to what ptr is pointing to right now. And the answer is yes.

(Refer Slide Time: 08:30)



So, it will behave exactly as you expect. So, ptr is pointing to this location, here is the previous location. So, it will jump to 5 locations before the location pointer 2 by ptr, I will happen to be A. So, the location which is str[1], that is = ptr-5. So, if you printf on that location, it will say BANTI is a nice girl, that is the output.

(Refer Slide Time: 09:10)



Before I proceed this one thing that I want to emphasize, and it is often not emphasized when you see online material on pointer arithmetic. C pointer arithmetic is not suppose to be meant for navigating the array, meant for navigating arbitrary locations in the memory. So, you cannot take a pointer. Let say character pointer and just say pointer $+ 1000$. It will give you some location in the memory, but the behavior of the program will be undefined. So, the c pointers are well defined, pointer arithmetic using C pointers are well defined only when the pointers are pointing to locations within an array. So, within an array $+ n$ will take you n locations away from the given pointer, $-n$ will give you $-n$ away from the give behind the given pointer and so on. Whatever type the character of whatever type the given pointer is pointing 2.

For example, if you have int num 10, and then you have $\text{num}+11$, you know that $\text{num}+11$ is not a valid location in the array. Similarly $\text{num}-1$ the num arrays starts at num of 0, which is equivalent to $\text{num}+0$. So, $\text{num}-1$ is also out of the given array, therefore these two locations are actually undefined, because c does not guarantee you, that if you try to deference these pointers, you will get any meaningful information. So, $+$ or $-$ and pointers are well defined, and their behavior is easy to predict exactly when you are navigating within the bounce of an array.

(Refer Slide Time: 11:14)

Main Point

- Suppose you have int num[10];

num[i] is equivalent to *(num+i)



So, the main point of lecture was that if you have let say for example, an integer array int num 10, then num of 5 which is the array notation is exactly equivalent to *(num+5). And I am not saying this that you can think of num of 5 as *(num+5), it is not an analogy this is exactly what C actually does. So, num of 5 is translated to *(num+5). So, arrays and pointers in C are very intimately related.

Introduction to Programming in C

Department of Computer Science and Engineering

(Refer Slide Time: 00:05)

Recap

- A pointer is a variable that contains the address of another variable. We say the pointer “points to” another variable.
- & : take the address of any variable.
- * is the operator to dereference a pointer (get the content of the variable that the pointer points to.)
- Pointer arithmetic (+,-)
- Relation between arrays and pointers – array name is a pointer to the first entry in the array. Also,

`array[i]` is equivalent to
`*(array+i)`



So, here is the stuff that we have seen about pointers. First we have defined what is a pointer? A pointer is just a variable that holds the **&**another variable. We say that pointer points to another variable. And depending on what variable it points to, the type of that target, we say it is an int pointer or a character pointer or a float pointer and so on. So, this is the first thing what is a pointer? And then we have seen what all can you do with a pointer; what are the operations that you can do in a pointer. So, if you have a normal variable, you can take the **&**that variable using the **&** operator. If you have a pointer, then you can dereference the pointer by using ***(ptr)**. That will go to the location pointer 2 by ptr and take the value of that target.

Further we have seen pointer arithmetic involving **+** and **-**. And I have introduced you with the caution that they are meant to navigate within arrays; they are not meant to navigate to arbitrary locations in the memory. If you do that, it may or may not work. And further we have touched up on the intimate relationship between arrays and pointers in C. As captured by the formula, **array[i]** is ***(array+i)**. A special case of this is to say that the name of the array is an **&**the first entry in the array. For example, **array[0]** is the same as ***(array+0)**. We have seen this and think about them once more to get comfortable with the notion.

(Refer Slide Time: 01:47)

The slide contains two blocks of C code. The left block is for the main function:

```
int main() {
    int a = 1, b = 2;
    swap(a,b);
    printf("From main");
    printf(" a = %d",a);
    printf("b = %d",b);
}
```

The right block is for the swap function:

```
void swap(int a, int b) {
    int t;
    t = a; a=b; b =t;
    printf("From swap ");
    printf("a = %d",a);
    printf("b = %d\n",b);
}
```

Below the code is a hand-drawn diagram illustrating pointer interaction. It shows three boxes labeled A, B, and T. Arrows indicate the flow of data: arrow 1 points from A to T, arrow 2 points from B to A, and arrow 3 points from T to B.

In this video, we will talk about how pointers interact with functions. When we introduced arrays, we first said here are arrays; here is how you write programs with arrays. And then we introduced... Here is how you pass arrays into functions. Let us do that the same thing with pointers. So, here are pointers. And how do you pass them to pointers? Before coming into how do you pass them to pointers, we will go into – why should you pass pointers to functions. So, let me introduce this with a very standard example. This is a classic example in C. How do you exchange two variables? We have seen the three-way exchange; where, I said that, if you have three rooms: A, B... I have two full rooms: A and B.

And then I want to exchange the contents of these rooms; then I can use a third room. First, move the contents of A to T; that is your first move. Then move the contents of B to A; that is your second move. And then afterwards, A now contains the contents of B; and B is empty; T is containing the contents of A. So, the third move is – move T to B. So, the net effect will be that, B contains the whole contents of A; A contains the whole contents of B; B contains the whole contents of A; and T is empty. So, this was the three-way exchange, which we did within main function. This is long back when we discussed GCD algorithm.

(Refer Slide Time: 03:44)-

The slide shows the following C code:

```
int main() {
    int a = 1, b = 2;
    swap(a,b);
    printf("From main");
    printf(" a = %d",a);
    printf(" b = %d",b);
}

void swap(int a, int b) {
    int t;
    t = a; a=b; b=t;
    printf("From swap ");
    printf(" a = %d",a);
    printf(" b = %d\n",b);
}
```

The output window displays:

```
Output: From swap a = 2 b = 1
        From main a = 1 b = 2
```

A callout box contains two points:

1. Passing int/float/char as parameters does not allow passing "back" to calling function.
2. Any changes made to these variables are lost once the function returns.

A handwritten note on the right says: "OK, i remember now".

Now, let us try to do that using a function. So, I have a swap routine, which takes two integer arguments: a and b; and it is meant to exchange the values of a and b. So, inside main, I have $a = 1$, $b = 2$. And I call swap a and b. And swap a and b – what it does is this three-way exchange that, we have discussed. Now, just to test whether things are working, I have a bunch of printf statements, which says what is the value of swap, what is the value of a and b after swap has executed. Similarly, when I come back, I will just print the values of a and b to see what has happened after swap. So, when you call swap and you output it within swap, it is very clear that, $a = 2$, $b = 1$. So, the three-way exchange would work as you expect. And you have whatever was passed, which is swap1, 2. So, it will exchange those variables and it will print $a = 2$ and $b = 1$.

Now, within main, a was 1 and b is 2. Now, when you print these statements inside main, surprisingly, you will find that, $a = 1$ and $b = 2$. So, the effect of swap is completely absent when you come back to main. Within swap, they were exchanged. But, when you come back to main, they were not exchanged. Why does this happen? This is because remember that, some space is allocated to a function; and whatever space is allocated to the swap function, all the variables there is erased – are erased once you return from the swap function. So, within swap function, a and b are exchanged. But, all that is gone when you return to main. So, passing integer, float, character variables as parameters, does not allow passing back to the calling function; you have only the return value to return back. Any changes made within the called function are lost once this function

returns. So, the question is can we now make a new function such that work done within that function will be reflected back in main.

(Refer Slide Time: 06:39)

```
main()
{
    int num[2];
    num[0] = 1;
    num[1] = 2;
    swap1(num);
}

int swap1(int arr[])
{
    int t;
    t = arr[0];
    arr[0] = arr[1];
    arr[1] = t;
    return 0;
}
```

Now, here is an intermediate solution. We know that, if we pass arrays, then work done in the called function will be reflected back in the calling function. So, you could think of the following intermediate function. So, if I have int num 2 and then I say that, **num[0]** is 1, **num[1]** is 2. This is in the main function. And then I call swap of num. Now, we will call it **swap1(num)**; I have a new function. Now, what swap1 does is – so int swap1 int arr. So, suppose I have this function; inside that, I will just say that, I will have an intermediate variable t; and then have **t = num** or **arr[0]**. Then **arr[0] = t**; **arr[0] = arr[1]**; and **arr[1] = t**. Suppose I have this function. And now, you can sort of argue that, this will also swap the two cells in the num array. So, the dirty trick that I am doing is that, I want to swap two variables; instead, I will say that, instead of these two variables, I will insert them into a array of size 2; and then call swap1 on that array. Now, what swap1 does is – it will exchange – it will do the three-way exchange on the array.

Now, I know that because of the way arrays are passed in C, any change that happens to the array arr inside swap1 will be reflected back in main. So, when I print these num array back in main, I would see that, **num[0]** is now 2 and **num[1]** is 1. So, this is an intermediate trick in order to write the correct swap function. But, you will agree that, this is a kind of a dirty trick, because in order to swap two variables, I created an array;

and then depended on the fact that, swap will change the array in such a way that, the change will reflected back in main. So, is there a nicer way to do it? That is what we are interested in. And the answer is let us just think about that array trick. What we did was – when we passed an array, we were of passing the &the array.

(Refer Slide Time: 09:40)

Here is the changed program.

```
void swap(int *ptrA, int *ptrB) {
    int t;
    t = *ptrA;
    *ptrA = *ptrB;
    *ptrB = t;
}

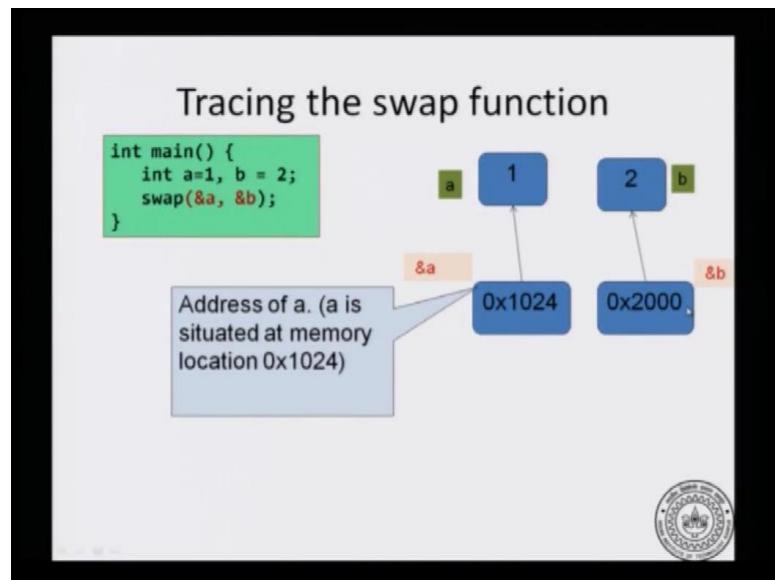
int main() {
    int a=1, b=2;
    swap(&a, &b);
}
```

1. The function swap() uses pointer to integer arguments, int *a and int *b.
2. The main() function calls swap(&a, &b), i.e., passes the addresses of the ints it wishes to swap.

This is how arrays are passed to functions. So, now, let us just take that idea that, we are passing the address. So, let us try to write a swap function, where you are passing the &variables instead of the variables themselves. So, here is the correct swap function. And what I write is void swap. So, void is a new keyword that you will see; but it is not a big deal; it is just a function that does not return a value; it just performs an action without returning a value. So, such functions you can write it as void – void swap int *ptrA, int *ptrB. So, ptrA and ptrB are pointers.

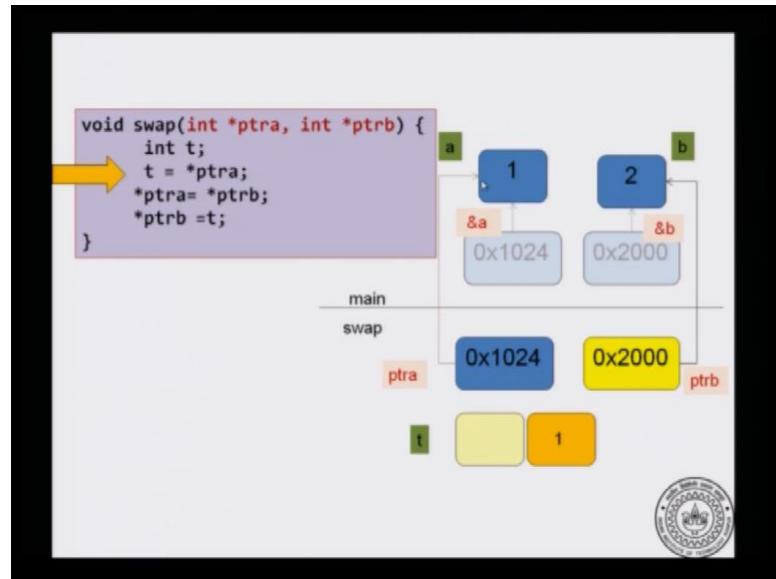
Now, inside the code, you have something that looks like a three-way exchange. It is very carefully return, because the obvious way to quote the function is not right. So, you have to be slightly careful; you have to declare an integer variable. Now, t contains *ptrA; *ptrA = *ptrB; and *ptrB = t. The obvious way to write it seems to be – you declare an integer *ptr t and then do this. It is not quite right; we will come to that later. So, here is the swap function. And how do you call the function? You declare two integer variables in main: a = 1 and b = 2; and then pass the addresses using &a and &b.

(Refer Slide Time: 11:27)



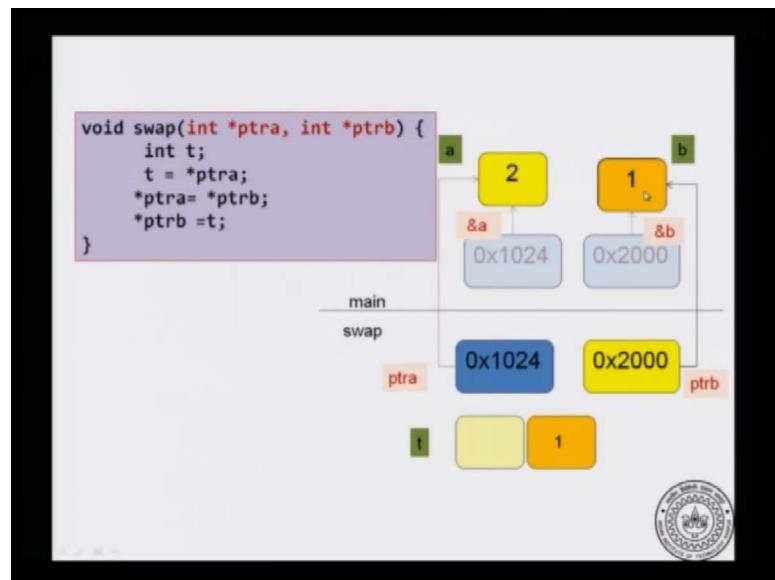
So, let us just trace the function. You have two variables in main; `a = 1`, `b = 2`; and call `swap(&a and &b)`. Now, just to denote that, these are addresses, I will say that, these are... `a` is situated at location 1024 in hexadecimal. So, this is some location in memory – hexadecimal 1024. And this is some other location in memory; `b` is say at hexadecimal location 2000. Now, do not be distracted by the hexadecimal notation if you are uncomfortable with it; just write 1024 in an equivalent decimal notation; and you can say that, it is at that location. So, it is at that location. And I am representing the location in hexadecimal, because it leads to shorter addresses. And this is also an address. So, when I will take `&a`, I will get 1024x in... When I take the `&b`, I will get 2000x. So, this is the `&a`. And it is located at memory location 1024 when represented in the hexadecimal notation.

(Refer Slide Time: 12:49)



What happens when you call the swap function? So, here is the state of main. And when you call the swap function, a new bunch of memory – a new block of memory is allocated on the stack. So, first, the formal parameters are copied their values from the actual parameters. So, ptra will get &a, which is 1024; ptrb will get &b, which is 2000. Now, I declare a new variable t; t = *ptra. So, what does that mean? ptra is an address – dereference the address; which means go look up that address. So, it will go to this location and get that value. So, t will now become 1. And the next statement is somewhat mysterious; please understand it very slowly. So, on the right-hand side, you have *ptrb. This means dereference ptrb. So, we are saying ptrb is address 2000; when you dereference it, you will get the value 2. Now, where do I have to store that value 2? For that, dereference ptra. So, 1024 – dereference it; you will go to this box. That is where you have to store 2.

(Refer Slide Time: 14:19)



So, 2 will go to that location. So, what has happened due to that is that, a in name has now changed. Why? Because within the swap function, we were dealing with pointers. So, as a result of the statement `*ptrA = *ptrB`, it has taken 2 from the main function's b and put it back into the main function's a. And that was accomplished through variables inside swap. So, think about it for a while. And the last statement of course is `*ptrB = t`. So, dereference ptrB and put the value 1 there. So, here is a three-way exchange that works through variables only in swap. But, since they were pointer variables, you ended up changing the locations in the main as well.

(Refer Slide Time: 15:26)

Homework ☺
Will the following code perform swap correctly?

```
void swap(int *ptrA, int *ptrB) {  
    int *ptrT;  
    ptrT = ptrA;  
    ptrA = ptrB;  
    ptrB = ptrT;  
}
```

And once you return, all the memory corresponding to swap will be erased. But then when you return to main, a and b will have changed. a and b were 1 and 2 before. Now, a is 2 and b is 1. So, it has correctly swapped. Now, as an exercise, I said that, the obvious way to write the swap function is as follows. Void swap a int *ptrb; ptra and int *ptrb. And then I declare int *ptrt. And then I write these statements. This is a very obvious way to code swap. This does not work. So, try to draw these pictures as we have done with a swap function that actually worked. Try to draw the picture of what happens in main and what happens in the swap function. And understand why this particular swap function does not work.

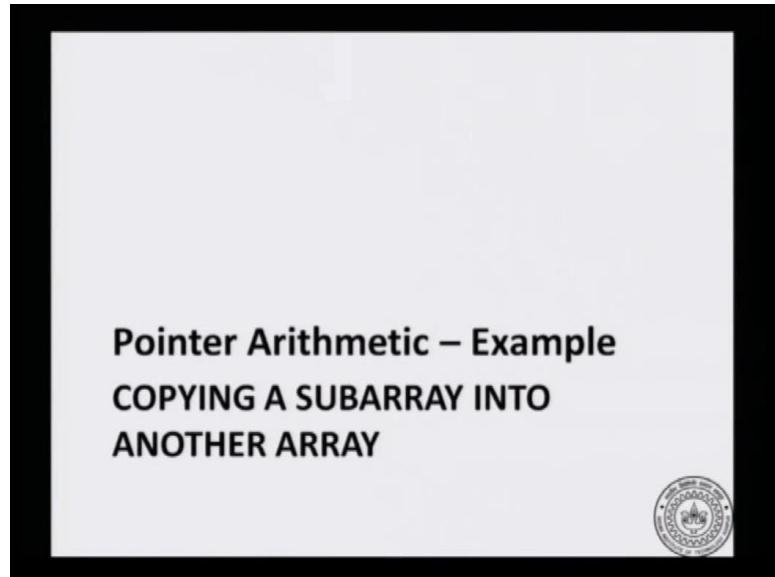
One final word about passing pointers to functions; C has something called a call by value mechanism. What is meant by call by value is that, when you call a function, remember the original picture that, your friend came with his note book and copied down the numbers in your page. So, your friend created a separate copy of your arguments; then computed what had to be computed and returned you a value. That picture is essentially still correct. Even though you are now dealing with functions, which can manipulate memory inside main, the passing mechanism is still call by value. It is just that, what is being copied are the addresses. So, when you manipulate the addresses through dereferencing, you end up changing the location inside main. So, even with pointers in C, what happens is call by value.

Introduction to Programming in C

Department of Computer Science and Engineering

Since pointer arithmetic is a tricky concept let us solve one more problem to try to get comfortable with that notion.

(Refer Slide Time: 00:07)



So, the problem here is copying a sub array into another array. Now, let us explain what that means?

(Refer Slide Time: 00:17)

The slide contains a text box with the following message:
I have written a function `copy_array(int a[], int b[], int n)`. It copies `n` successive index elements from `a[]` and puts into `b[]`.

Below the message is a code snippet:

```
int copy_array(int a[], int b[], int n) {
    int i;
    for (i=0; i < n; i = i+1) {
        b[i] = a[i];
    }
    return 0;
}
```

Underneath the code are three numbered steps:

1. Suppose there are two arrays `from[]` and `to[]`.
2. I want to copy `n` numbers from the array `from[]` starting at index `i` into the array `to[]` starting at index `j`.
3. I need a declaration like this below.

```
int copy_array_2( int from[], int i,
                  int to [], int j,
                  int n);
```

At the bottom left, the question 'Can you write this function?' is displayed.

Suppose, I have written a function `copy_array()` which has three arguments an integer array `a[]` and integer array `b[]` and `n` which is the size, I want to copy `n` successive index

elements from a and put it into b. So, a is 0 through a $n - 1$ have to be copied to b. I can easily write it in the following function has int copy_array int a, int b, int n and then I have one variable to keep track of the index and that variable goes from 0 to n, for $i = 0, i < n, i = i + 1$ and then I simply say $b[i] = a[i]$ within the loop.

So, this would copy whatever $a[i]$ is into the location $b[i]$. So, once the loop executes, I would have copied n elements from the array a to the array b. But, this is not general and I want to solve the following problem, I have two arrays let us name them `from()` and `to()` and I want to copy n numbers from the array `from[]` to `to[]`. But, I have an additional requirement, I want to copy n elements from index i.

So, the earlier code solve the problem from index 0 in general I want to copy from index i of `from[]` in the elements into the locations starting at index j in `to[]`. So, the earlier function assume that i and j were both 0. In the general function I want arbitrary i and arbitrary j. So, I need a declaration like the following, I have int copy_array_2. So, this is the second function I am writing and I have from i to j and then n is the number of elements to copy. So, what I have to do is from i from $i + 1$. So, on up to from $i + n - 1$ have to be copied to `to[j], to[j+1]` so, on up to `to[j + n - 1]`.

So, for the purposes of this lecture let us just assume that `from[]` and `to[]` are big enough. So, that you will never ever over suit the arrays by taking $i + n - 1$ and $j + n - 1$. Can you write this function? Now; obviously, you can write a separate function to solve this. Now, the trick is can you use the `copy_array()` function, the `copy_array()` functions copied n elements starting from index 0 of a to index $n - 1$ to the array b starting at index $b[0]$ to $b[n - 1]$. So, that is what is it means.

And this should be strange, because if you think about it in a mathematical way, you are saying that a general function is being solved in terms of s particular functions. So, you are reducing a general case to a special case at sounds a bit strange. But, we can do this with pointer arithmetic.

(Refer Slide Time: 03:58)

The screenshot shows a code editor with two functions defined:

```
int copy_array_2( int from[], int i,
                  int to[], int j,
                  int n);
```

OK! You can write the new function using your own function

```
int copy_array(int a[], int b[], int n) {
    int i;
    for (i=0; i < n; i = i+1) {
        b[i] = a[i];
    }
    return 0;
}
```

```
int copy_array_2( int from[], int i, int to[], int j, int n) {
    copy_array(from+i,to+j,n);
    return 0;
}
```

A watermark of a university seal is visible at the bottom right.

So, let us try to solve with using our own function, and here is the guess that I am making and then I will justify that this works. So, I will just say here is my function `from[], i, to[], j, n` that will be defined as `copy_array(from + i, to + j, n)`. So, `copy_array()` will start from `a[0]` and copy `n` elements afterwards to `b[0]` so, up to `b[n-1]`. And that function I am calling using the address `from + i, to + j` and `n`, `n` is the number of elements I want to copy. Now, I will justify that this works.

(Refer Slide Time: 04:54)

Problem: Given two int arrays `f[]` and `t[]`, copy `n` numbers starting from index `i` in `f[]` to index `j` in `t[]`.
So $t[j] = f[i], t[j+1] = f[i+1], \dots, t[j+n-1] = f[i+n-1]$.

```
void copy_array( int a[],
                  int b[], int n) {
    int i;
    for (i=0; i < n; i = i+1) {
        b[i] = a[i];
    }
    return 0;
}
```

```
int copy_array_2( int f[], int i,
                  int t[], int j, int n) {
    copy_array(f+i,t+j,n);
    return 0;
}
```

State: execution at start of `copy_array_2`

Memory State Diagram:

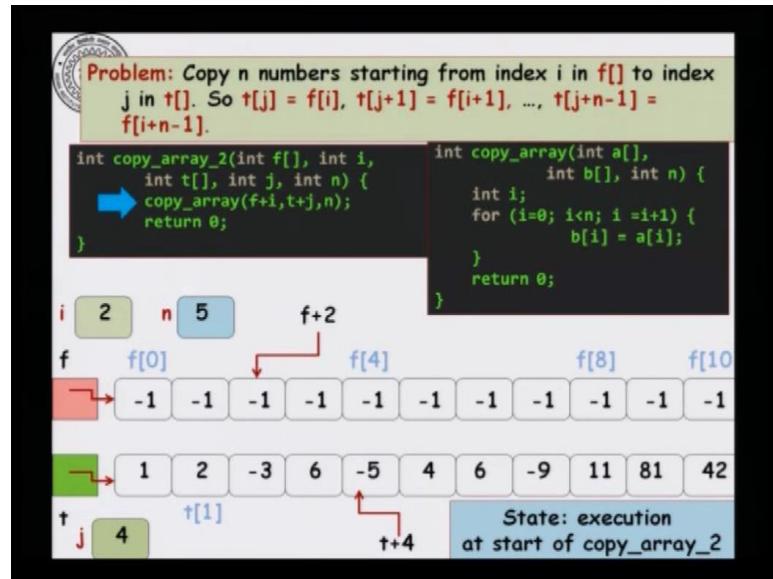
<code>f</code>	<code>f[0]</code>	<code>f[4]</code>	<code>f[8]</code>	<code>f[10]</code>							
	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	
	1	2	-3	6	-5	4	6	-9	11	81	42
	<code>t[0]</code>	<code>i</code>	<code>2</code>	<code>4</code>	<code>j</code>	<code>5</code>	<code>n</code>	<code>t[7]</code>	<code>t[10]</code>		

So, here is a problem that I have and I want $t[j] = f[i], t[j + 1] = f[i + 1]$ so, up to $t[j + n - 1] = f[i + n - 1]$. So, let us try to see what happens in this function? Suppose, I call `copy_array_2()` from name, using the arrays `f, t, i, j, n` and that function nearly calls the

old `copy_array()` function, using $f + i$, $t + j$ and n . So, this state of execution at the start of `copy_array_2()`, let say that $f[]$ is an array with say 10 elements and $t[]$ is an array with say 10 elements arbitrary and what I want is I also assume that i is 2 and j is 4.

So, I want to copy 5 elements starting from the second location or the third location in f $f[2]$ onwards to the fifth location in t onwards. So, here is what I want to... So, I want to copy this - 1, - 1, - 1, - 1, - 1, to $t[4]$ onwards. So, I will copy them here, so, 5 elements are to be copied.

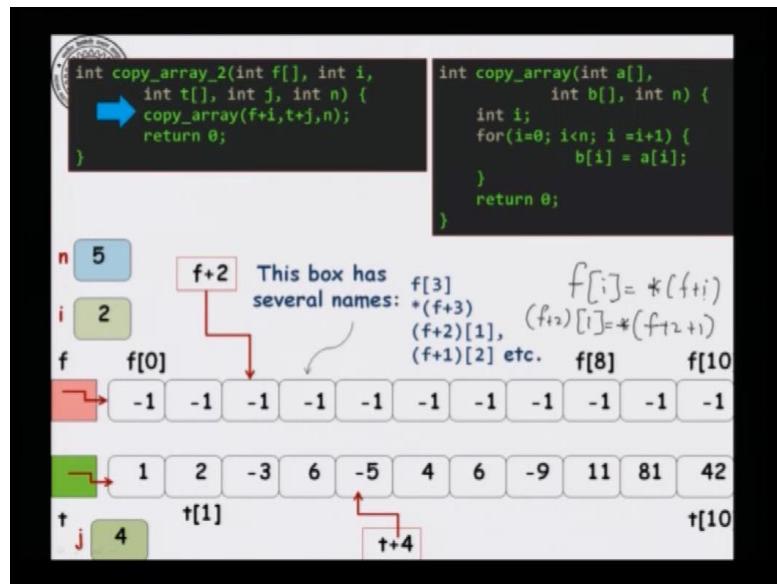
(Refer Slide Time: 06:33)



Let see; how our, function is able to do this. So, $t + 4$ is this location $f + 2$ is just location. So, what I am calling is the old `copy_array()` function with $f + i$. So, f is the address of the first location of the array. Therefore, $f + 2$ using pointer arithmetic is the second integer box after that. So, it is pointing to $f[2]$, similarly $t + 4$, $t + j$ in this case is pointing to the fourth location after the location pointer 2 by t , t is an array. So, t points to the first location in the array therefore, $t + 4$ will point to the fifth location in the array.

So, when I say $f + 2$ $f + 2$ is a pointer 2 here and $t + 4$ is a pointer 2 here and I am calling `copy_array()` function with these as the arguments and n is the number of elements I want to copy.

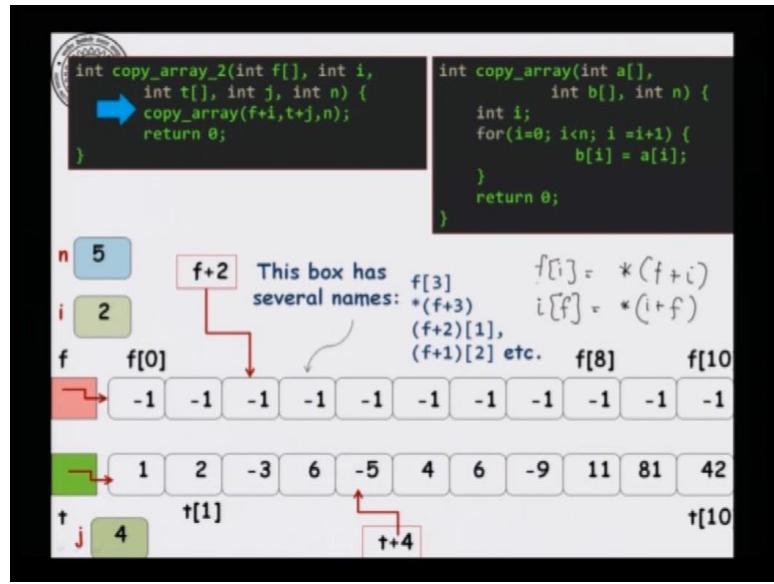
(Refer Slide Time: 07:45)



So, here is the state just before I call, `copy_array()` function. Now, for example, this particular box has several names, the most common name for it will be `f[3]`. But, I can also write it as `*[f + 3]`, this says jump 3 integer boxes after `f` and then dereference that address. Now, if you are comfortable with the notion that let us say `f[i]` is the same as `*[f + i]`.

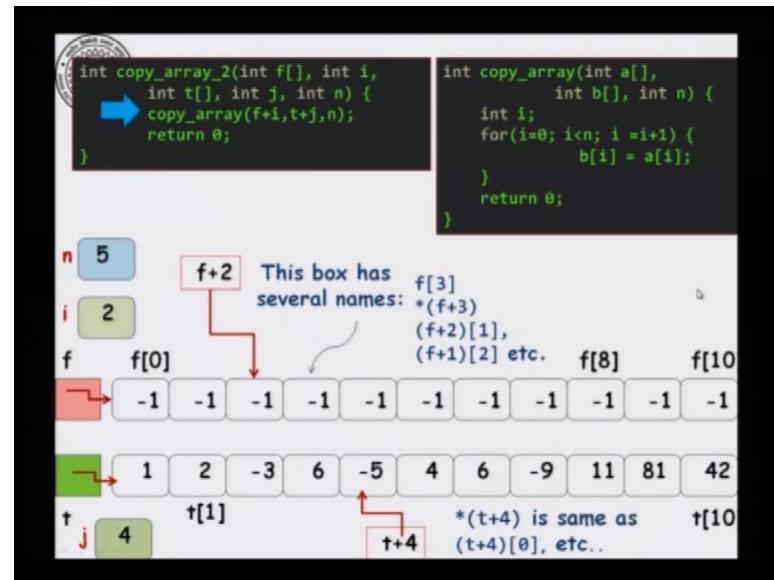
If you are comfortable with that notion, then you should be easy to see that `(f + 2)[1]` is just `*[f + 2 + 1]`. It is the same formula that I am using and this happens to be `f + 3`, which happens to be `f[3]`. So, `*(f+3)` would be `f[3]` and so, on. So, this formula that `f[i]` is the same as dereferencing the address `f + i`, `f[i] = *(f + i)` is applicable even for more strange looking expressions.

(Refer Slide Time: 09:06)



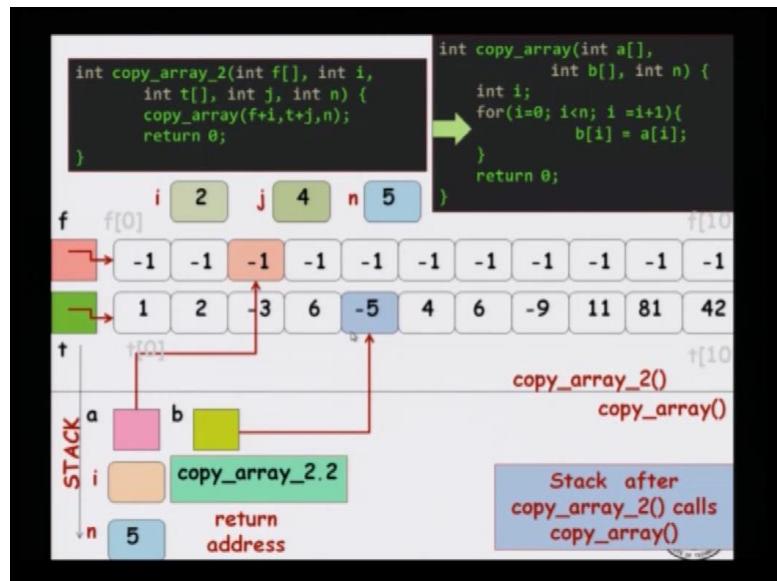
Now, here is a trivia about c, that because of the way it is defined. So, if you say that **f[i]** is the same as ***(f+i)**, then you could think that this is the same as ***(i+f)**. So, I can write this as i of f never do this, but it will actually work. So, **f[i]** you can also write it as i of f. For example, 3 of f and it will also work, because internally c translates it to ***(f+i)** and we know that ***(f+i)** is a same as ***(i+f)**. So, never do this, but this helps you to understand that **f[i]** is being translated by c into this format.

(Refer Slide Time: 10:27)



So, now, that we know this, similarly you can argue about ***(t+4)** and **(t+4)[0]** and. So, on all of them refer to the same box.

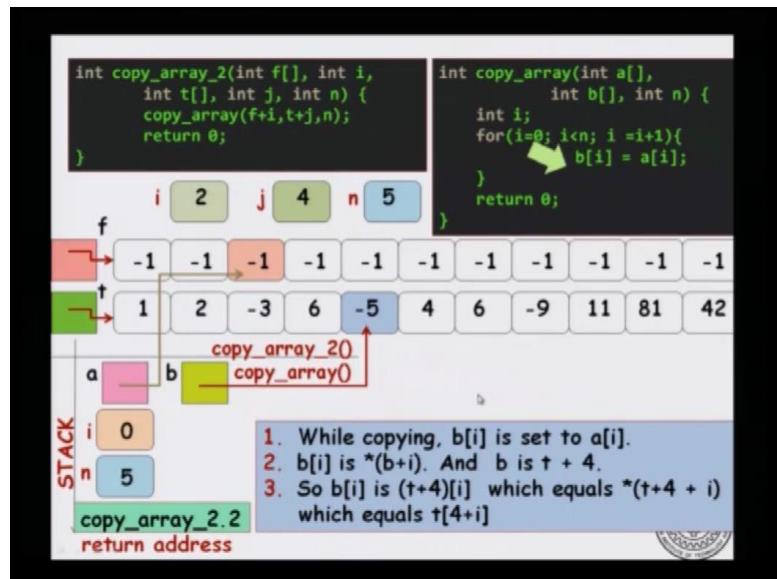
(Refer Slide Time: 10:36)



Now, let us see what happens when we call `copy_array()`. So, we have the stack space for `copy_array_2()` and `copy_array_2()` calls `copy_array()`. The formal parameters are `a` and `b`, `a` copies the address it was passed to, it was passed the address `f + i`. So, `a` points to `f + 2`, similarly `b` points to `t + 4`. So, `b` points to this. Now, as for as `copy_arrays` concerned it is not too bothered by the fact that, it was not passed the absolute first address of the array. It will think that whatever address it has been passed is the start of an array and it will work from there.

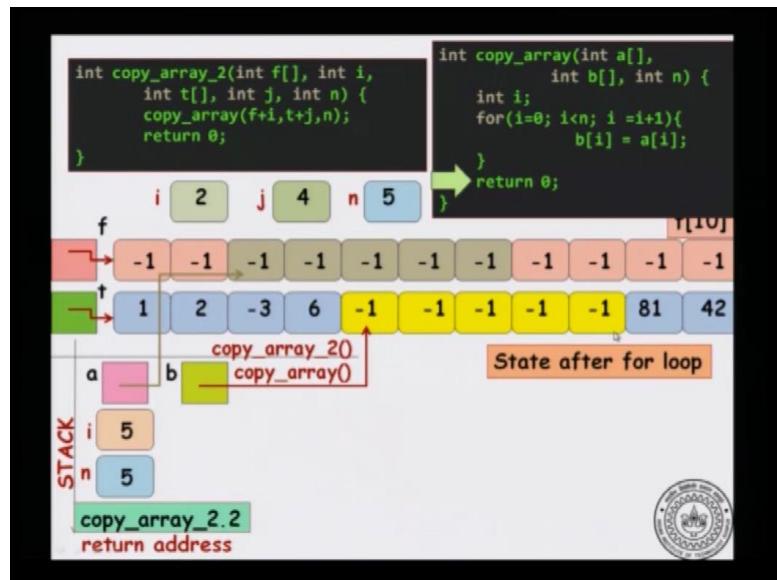
So, `copy_array()` does not bother the fact that, I was given the second element of the array, rather than the first element of the array and so, on, it will work as though the array started from there. And here is where we are exploiting that fact so, you now copy `n` elements from this location to this array, so, `n` is 5.

(Refer Slide Time: 12:06)



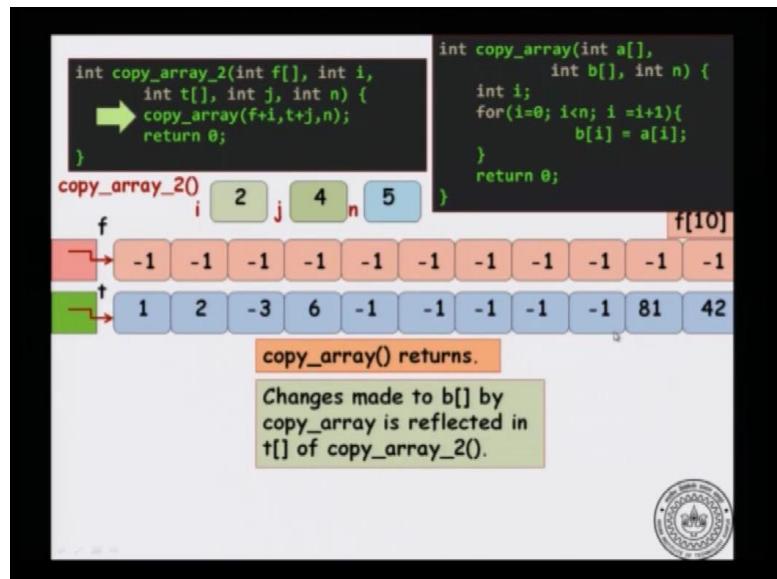
Now, when you execute that you will have `copy_array_2()`, should copy 5 elements starting from here to 5 locations here. So, that is what it will actually do.

(Refer Slide Time: 12:18)



And when you execute the loop, it will start from this location copied to `t + j` then `f + i + 1` will be copied to `t + j + 1` and so, on. So, it will copy these five locations to here in the `t` array.

(Refer Slide Time: 12:42)



And after you do this `copy_array()` returns and every variable that was allocated to `copy_array()` is erased. But, then because of pointers it was actually working with the arrays in `copy_array_2()`. So, even when you erase all the memory allocated to `copy_array()`, once you return these arrays would have been changed. These five locations starting from `f + 2` have been copied to these five locations starting from `t + 4`. So, changes made to `b[]` by `copy_array()`, is still maintained after you returns to the calling function `copy_array_2()`.

Introduction to Programming in C

Department of Computer Science and Engineering

In this lecture will see some more pointer arithmetic operators, and we will introduce those by talking about them through a problem.

(Refer Slide Time: 00:11)

More Pointer Operations - Reversing an Array

Write a function to reverse an array

The declaration of the function is

```
void rev_array ( int a[], int n );
```

The function takes an integer array and reverses the array *in place* - that is, without using any extra space.

before calling rev_array

a[0] | a[1] | a[2] | ... | a[n-2] | a[n-1]

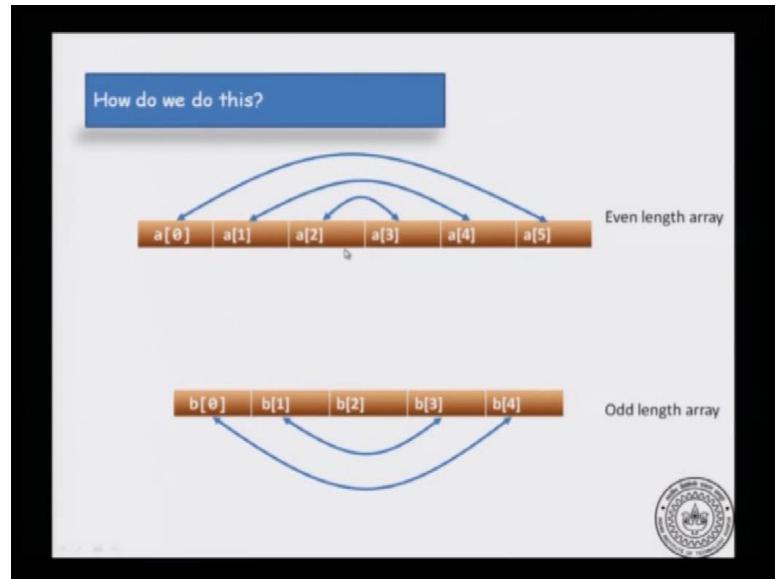
After calling rev_array

a[n-1] | a[n-2] | ... | a[2] | a[1] | a[0]

So, the problem that I have is that of reversing an array. So, we have to write a function to reverse an array, and let say that the declaration of the function is `void rev_array (int a[], int n[])`. Now if you have to reverse an array, what is one way to do it you take the array copy to into another array, and then copy back in the reverse fashion. So, you have an array `a[]` copy all the values in to `b[]`, and now you copy those values back to `a` in the following way that the `b`, `b` is last value will go to `a[0]`, `b` is second value last value will go to `a[1]` and so on. Now, let us try to do it slightly more cleverly, we want to take an integer array and reverse the array in place.

That means that essentially using no extra space. So, do not use and extra array in order to reverse it, reverse it within a itself. So, the array before calling reverse array will look like `a[0]` upto `a[n-1]` in this way. And after calling reverse array it should look like `a[n-1]` `a[n-2]`, etcetera up to `a[0]`, and we doing that we should not use an extra array.

(Refer Slide Time: 01:38)



So, how do we do this, let us look at bunch of couple of concrete examples to see how do it by hand, and then we will try to code that algorithm. So, in this there are 2 cases. So, for example, what happens when you having even length array, when you have let say 6 elements. Then $a[0]$ and $a[5]$ have to the exchange, $a[5]$ goes to $a[0]$, $a[5]$ goes to the 0 th location, $a[0]$ goes to the sixth location of the fifth location. And then $a[1]$ and $a[4]$ have to be exchanged, $a[2]$ and $a[3]$ have to be exchanged, after this you should stop, that will correctly reverse the array. Now what happens if the case of in the case of an odd length array, suppose you have an array b which has only 5 elements. In that case to reverse the array you have to exchange $b[0]$ and $b[4]$ $b[1]$ and $b[3]$, and you can stop there, because does no need to exchange $b[2]$ with itself. So, the case of an odd length array, you will end of with n element which does not need to be touched, in the case of an even length array, you have to exchange until you reach the middle of the ((Refer Time: 03:00)).

(Refer Slide Time: 03:05)

```
void rev_array ( int a[], int n )
{
    /* pointer to last element of a */
    int* b = a+n-1;
    while ( b > a ) {
        swap ( a, b );
        a = a+1;
        b = b-1;
    }
}
```

```
void swap ( int* ptra,
            int* ptrb )
{
    int t = *ptra;
    *ptra = *ptrb;
    *ptrb = t;
}
```

OK, but what does $b > a$ mean?

1. a initially points to the first element (left end) of the array.
2. b initially points to the last element (right end) of the array.
3. a moves forward, b moves backward.
4. In the loop, exchange $*a$ with $*b$. Repeat this until a and b cross over, or until a and b become equal.

Let us try to code this up. So, how is the reverse array return first I need... So, remember how we did this by hand, we exchange the 0 th location with the last location, then we exchange the first location within second last location, and so on. So, it is easy to code, if you have two pointers; initially one pointers starts to at the beginning of array, the second pointer is to the last of the array exchange those values, then the first pointer goes to the next location, and the second pointer goes to the previous location, that is how you did it by hand. So, let us try to code that up I will have a pointer $*b$, which points to the last element of array $a + n - 1$.

Now, the loop is as follows, I will discuss this in a minute while $b < a$. So, remember in the example by hand, we had to exchange till we reach the middle of the array. How do you find the middle of the array, I will just write it as $b > a$, and I will explain it in a minute. So, while this is true that you have not a ((Refer Time: 04:25) the middle of the array. You exchange swap a and b, here we use the swap function which we have seen in the previous lecture. So, for example, it will swap the 0 th element with the $n - 1$ first element. After that is done you increment a and you decrement b. So, the design logic is that a initially points to that first element of the array - the left end of the array, and b points to the right end of the array.

In general while the algorithm happens then a is moving forward, and b is moving backward. Inside the loop you exchange $*a$ with $*b$; that is what is accomplish by calling `swap(a, b)`, because swap a and b will dereference goes locations and exchange

the values there. So, do this repeatedly until a and b cross over, because then a is moving forward and b is moving backward, then the middle of the array is the point where a and b cross over or the point where a and b meet. So, this is the very simple logic for reversing an array. Now I have left one thing unexplained. What do I mean by $b > a$, b and a are pointers. So, what do I mean by pointer $b <$ pointer a , we need to explain that. We are introducing an operation on operator and two pointers, what does it mean?

(Refer Slide Time: 06:07)

```

void rev_array ( int a[], int n )
{
    /* pointer to last element of a */
    int* b = a+n-1;
    while ( b > a ) {
        swap ( a, b );
        a = a+1;
        b = b-1;
    }
}

void swap ( int* ptra,
            int* ptrb )
{
    int t = *ptra;
    *ptra = *ptrb;
    *ptrb = t;
}

```

Relational Comparison between Pointers

Let a and b be pointers to variables of the same type (e.g. `int *a; int *b;`). Then

1. We can compare pointers using `==` and `!=`.
2. $a == b$ is true if and only if they are pointing to the same box in memory.
3. Otherwise $a != b$ is true.

So, we are seeing a new concept which is relational comparison between two pointers. If a and b are pointers to variables of the same type like `int *a, int *b`. We can compare them, compare these pointers using `=`, and not `=`. This can be done for arbitrary locations a , and b as long as those locations are of the same type. So, $a = b$ is true, if and only if a and b are pointing to the same location; that is natural to expect. Otherwise if they are pointing to different locations $a \neq b$ is true. Now there is another case, if a is pointing to an integer let say, and b is pointing to a float then equal to and not equal to are undefined. So, notice that even though this behavior looks natural, it is natural only if they are pointing to this same type. So, here are operations equal to and not equal to.

(Refer Slide Time: 07:15)

```
void rev_array ( int a[], int n )
{
    /* pointer to last element of a */
    int* b = a+n-1;
    while ( b > a ) {
        swap ( a, b );
        a = a+1;
        b = b-1;
    }
}
```

Relational Comparison between Pointers

```
void swap ( int* ptra,
            int* ptrb )
{
    int t = *ptra;
    *ptra = *ptrb;
    *ptrb = t;
}
```

Comparing pointers using <, <= etc.

1. We can compare two pointers *a* and *b* using <. For this, they must be pointing to locations in the same array.
2. *a < b* is true if and only if *a* is pointing to an array location with index less than the index of the array location pointed to by *b*.
e.g. We can say *a+1 < a+2* in an array *int a[10]*;

OK, but what does *b > a* mean?

What about less than less than or equal to greater than greater or equal to and so on. And this is surprising, because here is something that you do not expect. You cannot compare less than less than equal to on arbitrary locations in the memory. We can compare *a* and *b* using less than for this they must be pointing to the same locations in the array. Earlier when we discussed + and -, we were saying that + and - are well behaved only when you are navigating within an array.

Similarly, when we are comparing 2 pointers using greater than greater than or equal to less than less than or equal to, then they should all be pointing then *a* and *b* should be pointing to the same array, different locations in this same array. If that is true then *a < b*, if *a* is pointing to a location which is before *b* in the same array. Similarly *a <= b* is true if *a* is pointing to a location which is *b* or before *b*, and so on. So, for example, we can say that if you have an array *int a[10]*, then *a + 1 < a + 2*, that is clearly true. Because *a + 1* is pointing to the location one in array and *a + 2* is pointing to location two in the array.

(Refer Slide Time: 08:57)

The diagram shows two code snippets. On the left, the `rev_array` function contains a loop where `b` starts at `a+n-1` and decreases towards `a`. An annotation points to the condition `b > a` with the question "OK, but what does b > a mean?". On the right, the `swap` function swaps the values at `*ptrA` and `*ptrB`. Below the code, an array `a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | a[8] | a[9]` is shown. Two pointers, `ptrA` and `ptrB`, are pointing to elements `a[1]` and `a[3]` respectively. A callout box states "Here, `ptrA < ptrB` is true".

So, if you have an array for an example let say `a[0]` through `a[9]`, and `ptrA` is pointing to location one, and `ptrB` is pointing to location three. Then `ptrA < ptrB` here the comparison is well defined and it is true.

(Refer Slide Time: 09:20)

The diagram is similar to the previous one, showing the `rev_array` and `swap` functions. The question "OK, but what does b > a mean?" remains. Below, two arrays are shown: `a[0] | a[1] | a[2] | a[3] | a[4]` and `b[0] | b[1] | b[2] | b[3]`. Pointer `ptrA` points to element `a[1]`, and pointer `ptrB` points to element `b[1]`. A callout box states "Here, `ptrA < ptrB` is undefined (they are not pointing to the elements in the same array.)".

But on the other hand let say that `ptrA` is pointing to `a[1]`, and `ptrB` is pointing to `b[1]`. In this case `ptrA < ptrB` is undefined, because their pointing to two different arrays. So, maybe in memory `a` is lead out before `b` and so on, but that is not what the `<` or `=` operation is supposed to do. It is suppose to compare pointers only within the same array.

(Refer Slide Time: 09:55)

The slide displays two code snippets in a dark-themed code editor. The left snippet is for reversing an array, and the right snippet is for swapping two integers. Below the code is a memory diagram illustrating the first iteration of the swap operation.

Code Snippets:

```
void rev_array ( int a[], int n )
{
    /* pointer to last element of a */
    int* b = a+n-1;
    while ( b > a ) {
        swap ( a, b );
        a = a+1;
        b = b-1;
    }
}
```

```
void swap ( int* ptra,
            int* ptrb )
{
    int t = *ptra;
    *ptra = *ptrb;
    *ptrb = t;
}
```

Memory Diagram:

First Iteration: swap(a, b)
Then advance a and retract b by 1 each.

The diagram shows a memory array with six locations containing the values 101, 21, -1, 121, -101, and 0. Pointer 'a' points to the first location (101), and pointer 'b' points to the last location (-101). Arrows indicate the movement of pointers: 'a' moves right (advancing) and 'b' moves left (retracting) during the first iteration.

So, with this understanding let us understand how the reverse array works. So, in the first iteration, you have an array **a[]**, let say that array is 101, 21, and so on, it has 6 locations. And we will run through that trace of the execution for an even length array, and I would encourage you to create an odd length array, and trace to the executions to ensure that the code works for odd length arrays as well. So, in this lecture will do it for an even length array. So, a is initially pointing to the beginning of the array, b is pointing to the end of the array, $a + n - 1$ will go to the end of the array. Now $b < a$ that is true. So, we will enter the loop and in the first iteration we will swap a and b.

(Refer Slide Time: 10:49)

The slide displays two code snippets in a dark-themed code editor. The left snippet is for reversing an array, and the right snippet is for swapping two integers. Below the code is a memory diagram illustrating the first iteration of the swap operation.

Code Snippets:

```
void rev_array ( int a[], int n )
{
    /* pointer to last element of a */
    int* b = a+n-1;
    while ( b > a ) {
        swap ( a, b );
        a = a+1;
        b = b-1;
    }
}
```

```
void swap ( int* ptra,
            int* ptrb )
{
    int t = *ptra;
    *ptra = *ptrb;
    *ptrb = t;
}
```

Memory Diagram:

First Iteration: swap(a, b)
Then advance a and retract b by 1 each.

The diagram shows a memory array with six locations containing the values 0, 21, -1, 121, -101, and 101. Pointer 'a' points to the first location (0), and pointer 'b' points to the fifth location (-101). Arrows indicate the movement of pointers: 'a' moves right (advancing) and 'b' moves left (retracting) during the first iteration.

So, it will go to the swap function, and this is the swap function that actually works from the previous video. So, you can assume that $a[0]$ will be swapped with $a[1]$. So, they were initially 101 and 0, and after swap they will be 0 and 101. Come once that happens a advances by one integer location b goes back by one integer location. So, this is the state after the first iteration. In the second iteration you start with a at 21, and b at -101, again $b < a$, so is swap.

(Refer Slide Time: 11:41)

The slide contains two code snippets and a diagram:

- Code Snippet 1:** A C-like code snippet for `rev_array`.

```
void rev_array ( int a[], int n )
{
    /* pointer to last element of a */
    int* b = a+n-1;
    while ( b > a ) {
        swap ( a, b );
        a = a+1;
        b = b-1;
    }
}
```
- Code Snippet 2:** A C-like code snippet for `swap`.

```
void swap ( int* ptra,
            int* ptrb )
{
    int t = *ptra;
    *ptra = *ptrb;
    *ptrb = t;
}
```
- Diagram:** A diagram showing the state of an array and pointers a and b .

Second Iteration: `swap(a, b)`
Then advance a and retract b by 1 each.

The array elements are shown in orange boxes: 0, -101, -1, 121, 21, 101. Pointer a points to the element 21, and pointer b points to the element -101. Arrows indicate the movement of pointers: a moves up to point to -101, and b moves down to point to 21.

So, 21 - 101 becomes -101 and 21, so they are swapped. And you advance a by 1 and you take back b by 1, again $b < a$.

(Refer Slide Time: 12:02)

The slide displays two code snippets and a diagram. The first snippet is a function `rev_array` that reverses an array. It uses a pointer `b` to point to the last element of the array. The second snippet is a function `swap` that swaps the values at two memory locations. A callout box indicates the current state is the "Third Iteration: swap(a, b) Then advance a and retract b by 1 each." Below this, a diagram shows an array of integers [0, -101, 121, -1, 21, 101]. Two blue arrows originate from pointers `a` and `b`. The arrow for `a` points to the value 121, and the arrow for `b` points to the value -1. A circular seal of the Indian Institute of Technology (IIT) Kharagpur is visible in the bottom right corner.

```
void rev_array ( int a[], int n )
{
    /* pointer to last element of a */
    int* b = a+n-1;
    while ( b > a ) {
        swap ( a, b );
        a = a+1;
        b = b-1;
    }
}

void swap ( int* ptra,
            int* ptrb )
{
    int t = *ptra;
    *ptra = *ptrb;
    *ptrb = t;
}
```

Third Iteration: swap(a, b)
Then advance a and retract b by 1 each.

Diagram showing array elements: 0, -101, 121, -1, 21, 101. Pointers a and b are shown with arrows pointing to 121 and -1 respectively.

So, you go to the third iteration. In the third iteration this is the state of the beginning of the iteration, and the swap these contents. So, you swap -1 and 121 it becomes this becomes the state of the array, and once that is done b over suits. So, b goes before a, and a goes after b, swap denoted there with two colored arrows. So, here is the b arrow, it goes to location 121, and a arrow goes to location -1. When this happens b is now $<$ a. So, this means that you have cross the middle of the array. Therefore, you should stop now. So, now b $<$ a, and the loop terminates, and we have seen that this correctly reverse the array. So, here is how the reverse array works, we have seen the concept of relational comparison operators using pointers. How do make sense when they are pointing to locations with them the same array, and how that can be used to write code to using arrays.

Introduction to Programming in C

Department of Computer Science and Engineering

In this video, we will see the sizeof operator, which is a slightly advanced topic, in relation to pointer arithmetic. This is explaining, how pointer arithmetic freely works. And it is also important to understand one topic that, we will see later on and called malloc.

(Refer Slide Time: 00:23)

sizeof operator

■ The **sizeof** operator gives the number of **bytes** that any value with a type occupies. The values depend on the machine.

Example: `sizeof(int) = 4`
`sizeof(float) = 4`
`sizeof(char) = 1`



So, the sizeof operator and note that, it is an operator and that highlighted that in red, it looks like a function call, but it is not. The operator gives the number of bytes that any value with the given type occupies. So, sizeof is an operator which takes the name of a type of as an assigned argument, it can also take other kinds of arguments, we will see that. So, you could ask, what is the sizeof an int? What is the sizeof a float? What is the sizeof a character?

And the answer, the value that it will come out to be will depend on some particular machine. So, the reason why we use the sizeof operator is that, it helps you to write the code that is general enough for any machine, we will see, what that means? So, right now you just returns you the size of any given data type.

(Refer Slide Time: 01:22)

sizeof() operator

- This has an effect on the way data is allocated. (depends on the machine)
- For example, in a character array, the cells are 1 byte apart:

0x1000	0x1001	0x1002	0x1003	0x1004	0x1005	0x1006	0x1007
'S'	'u'	'c'	'c'	'e'	's'	's'	'\0'
char s[8]							
- In an integer array, the cells are 4 bytes apart.

0x2000	0x2004	0x2008	0x200c
1	2	100000	14
int a[4]			

c in base16
= 12

So, the sizeof operator has an effect on the way data is allocated and the way details allocated depends on the machine, we will see that. So, for an example if you have a character array, the cells are 1 byte apart. So, sizeof operator returns you the number of bytes that a data type occupies. So, in the case of a character, the character occupies 1 byte. So, if you have a character array declared as `char s[8]`, what you have are, 8 cells and each of those cells occupy a width of 1 byte.

So, let us say that the character arrays starts at hexadecimal address 1000. So, the next cell will be at the next byte, which is byte address hexadecimal 1001. And this goes on until the last cell which is hexadecimal 1007. So, this contains a null terminator character array with the letters s u c c e s s and then followed by a null. What happens with an integer array? So, in an integer array, sizeof an int is 4 bytes. So, the successive elements of an integer array are 4 bytes apart.

So, let us say that I have declared an integer array as `int a[4]`. This means, that a 0 let us say, it starts at hexadecimal address 2000. Then, a 1 will start 4 bytes away, because the size of an int is 4 bytes. So, it should start at address hexadecimal 2004 and I have sort of indicated it pictorially I want to say that a character is a narrower data type than an integer, an integer occupies 4 bytes. So, the next integer cell, the next integer element in the array will start at hexadecimal address 2004.

The third element will start at hexadecimal address 2008 and the last at hexadecimal address 200c. So, notice that I should have started it at 2012. But, 12 in hexadecimal

addressing scheme is c. So, in base 16, c is the same as decimal 12. So, that is why I have written it as hexadecimal 200c.

(Refer Slide Time: 04:18)

Pointer arithmetic in detail

- Suppose you declare an int *ptr; Then ptr+i is equal to the byte numbered
$$\text{ptr} + i * \text{sizeof(int)} = \text{ptr} + i * 4$$
- Suppose you declare a char *ptr; Then ptr+i is equal to the byte numbered
$$\text{ptr} + i * \text{sizeof(char)} = \text{ptr} + i$$
- In general, if you have type *ptr; Then ptr+i is the byte numbered
$$\text{ptr} + i * \text{sizeof(type)}$$
- Why? This ensures that $\text{array}[i] = *(\text{array} + i)$ points to the i^{th} element in the array.
???



Now, let us look at pointer arithmetic in greater detail with our current understanding of the sizeof operator. So, suppose you have an int *pointer. If you have an int *pointer and then you want to say that $\text{ptr} + i$ is equal to, what it should? So, notice that + make sense, when you are navigating within an array. So, ptr is let us say pointing to some cell within the array and $\text{ptr} + i$ should go to the ith cell after ptr, that is what, it should do. Now, the i^{th} cell after ptr means the i^{th} integer after ptr.

So, we should skip 4 i bytes in order to reach the i^{th} integer cell after ptr. So, thus is what we have written here $\text{ptr} + i$ is the byte number, $\text{ptr} + i * \text{sizeof(int)}$, the machine addressing goes in terms of bytes. So, in order to jump to the i^{th} integer cell, we have to know, how many bytes to skip? And the size of an integer is 4 bytes. So, this means we have to skip ahead 4 i bytes, in order to reach $\text{ptr} + i$.

Now, if we have declared character *ptr, then $\text{ptr} + i$ is supposed to jump to the i^{th} character after ptr size of a character is 1 byte. So, $\text{ptr} + i * \text{sizeof(char)}$ would be $\text{ptr} + i * 1$, it is the same as $\text{ptr} + i$. So, notice that let us say that, machine understands only byte addresses. So, in order to execute $\text{ptr} + i$ correctly, we have to tell which byte should I go to, should the machine go to? And in order to do that, you utilize the sizeof operator. So, since we have declared character *ptr, you know that it is sizeof character, $i * \text{sizeof}$ character those many bytes I have to skip.

In the previous case, I have declare int *ptr. So, in that case I have to skip $i * \text{sizeof}(\text{int})$, in order to reach the correct cell. So, here is the actual reason why $\text{ptr} + i$ would magically work correctly. Whether, it was an integer array or it was a character array? This is because, at the back of it all, you translate everything to byte addresses using star sizeof whatever type. So, in general if you have type $*\text{ptr}$, then $\text{ptr} + i$ is the byte number $\text{ptr} + i * \text{sizeof}(\text{type})$.

So, this type is the same as the declared type of pointer, ptr is a pointer to that type. Therefore, you multiply it with `sizeof(type)` and this is the general formula for pointer arithmetic. Now, one of the side effects of that or one of the consequences of this kind of addressing is that, $\text{array} + i$ is $*(\text{array} + i)$ and it will correctly jump to the i th location in that array, regardless of whatever type the array was. Why is that?

Because, $\text{array} + i$ is then translated to $\text{array} + i * \text{sizeof}$ whatever type the array has been declared to be. So, you will correctly jump to the byte address corresponding to the i th element in the array. So, here is how array arithmetic in c works, in full. What do we mean by this? Let us see that with the help of an example.

(Refer Slide Time: 08:30)

An Example

- Suppose you have `int a[10]`; starting at address `0x2000`.
- $a[2] = *(a+2)$ is the content located at `byte` address
 $a+2 * \text{sizeof}(\text{int}) = a+2 * 4 = 0x2008$.

<code>0x2000</code>	<code>0x2004</code>	<code>0x2008</code>	<code>0x200c</code>
<code>1</code>	<code>2</code>	<code>100000</code>	<code>14</code>
<code>int a[4]</code>			

Why C arrays
start at index 0!



Suppose, you have an integer array declared as `int a[10]` and it starts at the address 2000. And I want to know, how is it that you get this third element of the array `a[2]`. So, `a[2]` we know is $*(a + 2)$, a is a pointer to the first element of the array. And you have to now understand, how $+ 2$ is executed? So, $+ 2$ should be the content located at byte address $a + 2 * \text{sizeof}(\text{int})$. Why is this? A is been declared as integer array. And in c, integer array

has the same type as `int *`. So, `a` is a pointer to `int`. Therefore, we know that, we have to do `a + 2*sizeof(int)`.

Whatever the argument is, it will do $2*\text{sizeof}$ the type pointer 2 by that pointer. So, we will do `a+2*4`, which is hexadecimal address 2008, `a` was 2000. So, if you have an array `int a[4]`, let say and it started at address 2000, then you will jump to array address 2008. And this is the reason, why C arrays start at index 0? Because, it is a very easy formula, `a[0]` would be `*(a+0)`, which is simply `*a`.

In that case, you have the consistent explanation, that the name of the array is a pointer to the first element of the array, you do not need a special rule to do that. Think of what would have happened, if arrays started at 1. Then, `a[2]` would be `a+1*sizeof(int)`. So, `a[n]` would be `a + n - 1` times `sizeof` whatever and that is an uglier formula than what we have here. So, it is better for arrays to start at location 0, because it makes the pointer arithmetic easier.

(Refer Slide Time: 11:01)

Summary - `sizeof()`

- Thus `sizeof()` operator is used in pointer arithmetic.
- We will see another common use soon.



So, in summary the `sizeof()` operator is used in pointer arithmetic and we will see one more common use of the `sizeof()` operator, very soon.

(Refer Slide Time: 11:13)

General Usage

- `sizeof(expression)` – size of the data type of the expression in bytes. e.g. `sizeof(10)` = size of an int
- `sizeof(typename)` – size of the data type in bytes. e.g. `sizeof(int)` = 4 on a particular machine.
- `sizeof(array)` – size of the array in bytes. e.g. If the array is `int num[10];` then `sizeof(num)` = 40 (which is $10 * sizeof(int)$)
$$\frac{\text{Size of } (\text{num})}{\text{Size of } (\text{num}[0])} = \frac{40}{4} = 10$$

So, the general usage is you can give size of an expression. What will it do is, it will take the type of that expression. So, if I say `sizeof(10)`, then 10 is an int,. So, it will execute `sizeof int` and let us say that, on a particular machine it is 4 bytes. Similarly, you could also say `sizeof type name`. So, for example, I could say `sizeof(int)`. Rather than giving an integer as an argument, I could also say `sizeof(int)`, where int is the name of the type and it will return 4 on some particular machine.

A less common usage is, you could give `sizeof array`, if the array is some particular array and it will return you the size of the array in bytes and this is important. It will not return you exactly the number of elements in the array, it will return the total size of the array in bytes. What do I mean by that? If I say, `int num[10]` and then I say `sizeof(num)`, it will return is 40 because, there are 10 integers, each integer occupying 4 bytes.

So, in order to calculate the number of elements in the array, for example, you could do the following, you could say `sizeof(num) / sizeof(num[0])`. So, this would evaluate to `40/4` which is `10`. So, size of the operator on the array does not exactly give you the number of elements in the array, it will give you the total number of bytes in the array. But, if you also know how many bytes, the particular element in the array occupies, then you can easily figure out the size of the array, in terms of the number of elements.

So, also note that c does not say that, an integer is 4 bytes or float is 4 bytes and so, on. What it specifies is the relationship between the sizes of various types and we will not get in to it, right now. But, just keep in mind that the size of a particular type is depended

on, which machine you are running the code on.

Introduction to Programming in C

Department of Computer Science and Engineering

In this video, we will discuss slightly advanced usage of pointers. Even though, the title of the first slide is, how to return pointers from a function? That is just a motivation for introducing a slightly more advance topic in pointers.

(Refer Slide Time: 00:10)

How to return pointers from a function

- “dangling pointer” example:

```
int *increment(int n)
{
    int temp;
    int *ptr = &temp;
    temp = n+1;
    return ptr;
}

int main(){
    int *p = increment(1);
    printf("%d\n", *p);
}
```

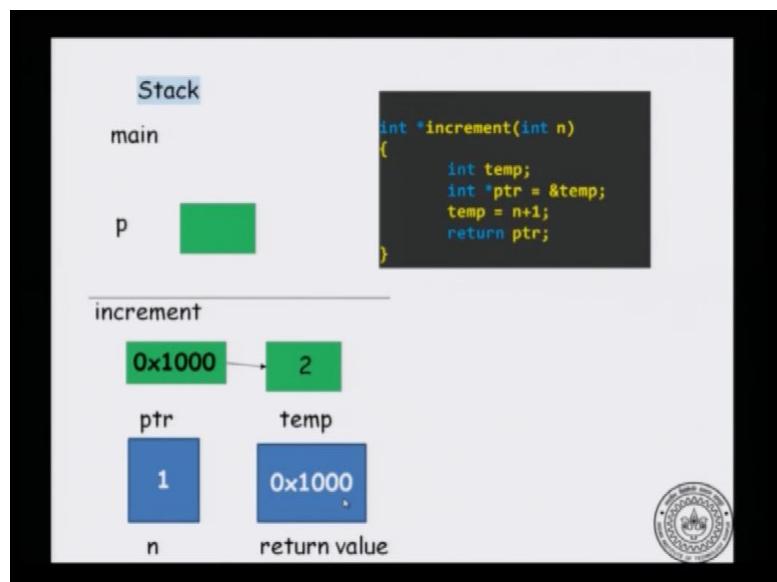
- Returns the address of temp, but temp is erased as soon as increment(n) returns.
- So the return value is not a meaningful address to the calling function,



So, let us just see what is the problem with returning a pointer from a function? We know that, any variable can be passed as argument to a function, can be declared as a local variable within a function, and can also be a return from a function. So, is there something and we have already seen that, in the case of the swap function how we pass pointers to a function and what new kinds of functions does this enable us to write?

So, in this lecture let us talk about what will happen, when we return pointers from a function? I have written a very silly function here, you do not need a function to do this. But, this illustrates a point I have a main function, in which I have an integer pointer p and I will make it point to the return value of `increment(1)`. And the increment function, what it does is, it takes an argument, it increments the value makes a pointer and points to the incremented value and returns that pointer, we will see a pictorial representation in a minute. Now, this leads to a very notorious error in c known as a dangling pointer.

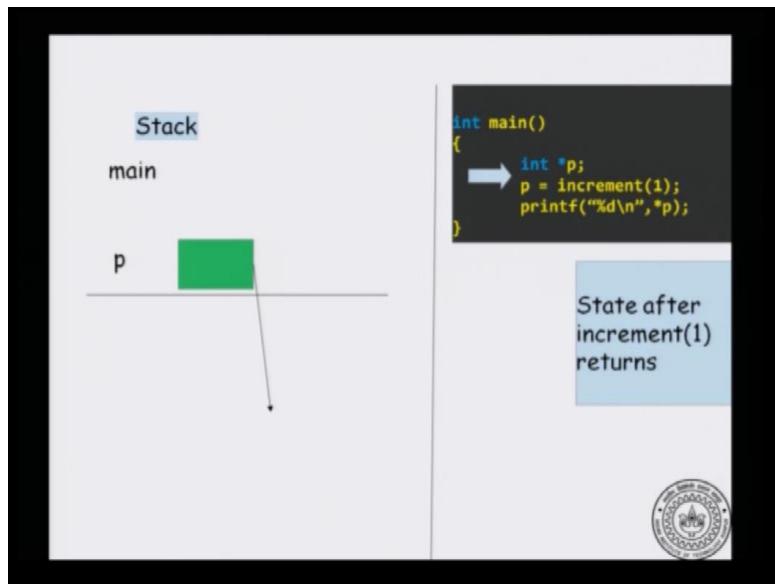
(Refer Slide Time: 01:41)



Let us examine that in slightly greater detail. What happens when we call the increment function? Inside the main function, we have an int pointer p and then it is declared, it is not pointing to anything and immediately, you will call **increment(1)**. So, you call **increment(1)**, n is a local variable in increment, it is the argument. So, n is 1 and then, you declare a temp and then, you declare a pointer to temp,. So, there is a ptr. Let us say that, temp is a address hexadecimal 1000. So, ptr contains 1000 and it points to temp, temp is at address 1000.

Now, in the next statement you increment temp, you set temp to $n + 1$. So, n is 1 and temp is 2. Now, ptr points to temp and now, I will return ptr. So, the return value is 1000, which is the address of temp.

(Refer Slide Time: 03:01)



Now, what happens when you return to main? As soon as increment finishes and we have said this several times before, as soon as any function finishes, the memory with that is allocated to the function is erased. So, when you return to the main function, what happens is that, you have p and p will contain the address 1000. So, it is meant to point to temp. But, the space meant for temp has already been erased.

So, p is pointing to a junk value in memory, it is pointing to an arbitrary location in memory. So, this is known as a dangling pointer. Hopefully, the picture is a representative of a dangling pointer. The fact that, it points to a location, which is no longer meaningful. Notice that, what I am talking about is the ideal situation, when you code it in c and try to run it, may be p does point to the location with address two.

This is because, c may not be aggressive in re cleaning the memory. But, you should always assume that, the safe thing is to assume every location that was allocated to increment is erased immediately after increment returns. In practice, it may not be the case,, but you should never assume that, you still have the temp variable. In general, what you will have is a dangling pointer. Because, p points to a location which no longer contains any relevant information.

So, when you print p, you will have a danger. So, how do you return pointers from a function? So, we have seen what a dangling pointer means? And here is a very silly function, which will create a dangling pointer. Now, what is the problem with this function, it returns the address of a local variable temp. But, temp is erased as soon as

increment n returns. So, the return value is not a meaningful address to the calling function, which is mean. Can we get around this?

(Refer Slide Time: 05:11)

■ Problem: Anything allocated for the called function on stack is erased as soon as it returns.
■ Is there a way to meaningfully return pointers to "new" variables?
■ We use a new concept – a globally accessible memory, called "heap".
■ Idea: If we allocate a value on the global memory (heap), it is not erased when the function returns.

IIT-Bombay Logo

So, the main problem here is that, anything that is allocated to the called function on the stack is erased as soon as it returns. Is there any way at all to meaningfully return pointers to new variables? Then, use a new concept that is a globally accessible memory called heap, we have already seen a stack. Now, we will understand what a heap is. So, roughly the idea is that, if you allocate value on the global memory, it is not erased when the function returns.

(Refer Slide Time: 05:56)

An Intuition

- Think of executing a function as writing on a classroom blackboard.
- Once the function finishes execution (the class is over), everything in the blackboard is erased.
- Suppose we want to retain a message, after class is over.
- Solution could be to post essential information on a "notice board", which is globally accessible to all classrooms.
- The blackboard of a class is like the stack memory (erased after finishing), and the notice board is like the heap (not erased when the function finishes).

IIT-Bombay Logo

I will explain this with the help of a slightly broad analogy. Hopefully, this is indicative of what actually happens with the heap? So, think of executing a function as writing on a classroom blackboard, when a lecture is going on. Once the function finishes execution, which is like the class is over, everything on the blackboard is erased. Suppose, you want to retain a message after the class is over. Now, the solution could be that you can post things on a notice board, which is global to all class rooms.

So, it is common to all class rooms. So, things on the notice board are not removed as soon as a class is over. If you write something on the blackboard, which is similar to storing something on the stack, as soon as the class is over, it will be erased. So, if you have something to communicate back to another class, may be you can post it on a notice board. Now, the notice board is globally accessible to all class rooms. The black board is like a stack and the global notice board is like a heap and contents on the heap is not erased, when the function finishes.

(Refer Slide Time: 07:16)

Allocating on the heap

- We can allocate space on the heap using a library function called malloc, in stdlib.h
- malloc(n) allocates n bytes of memory on the heap.
- It returns a pointer, which can be cast to a pointer of any type.

```
#include <stdlib.h>

int main()
{
    int *ptr;
    /* Allocate 10 integers on the heap
     * Treat the return address as an
     * integer pointer
     */
    ptr = (int *) malloc (10*sizeof(int));
}
```



So, how do you allocate things on the heap? There is a standard library function called malloc, in the file **stdlib.h** which can be used to allocate space on the heap. Roughly, this is what it does, if you ask for **malloc(n)**, there n is a positive integer, it will allocate n bytes of memory on the heap and it will return a pointer to the first location of the allocated space. Now, that pointer can be converted to pointer of any type, malloc just allocates n bytes.

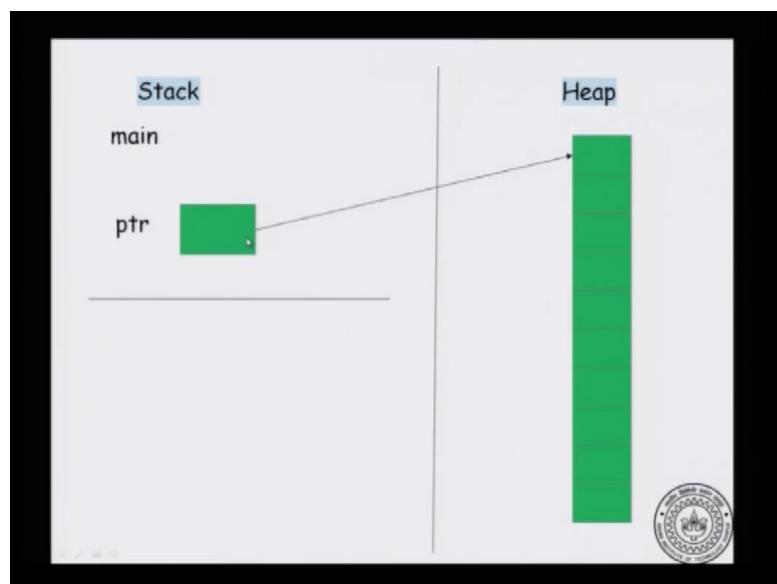
Now, you may want to interpret those bytes as n divided by 4 integers. In that case, it

will return a pointer,. So, you convert that pointer to an int pointer. Let us see an example I may have an ints pointer for ptr and now, I want to allocate 10 integers on the heap. How do you I do that? I will allocate `10 * sizeof(int)`. So, this will allocate on some particular machine, let us say 40 bytes and it will return an address of the first location. Now, that address I want to treat it as an integer address. So, I will convert it to an int as `int *` and then `malloc (10*sizeof(int))`.

So, this style of writing code means the code portable. Because, suppose you write the code and on a machine, where integer was 4 bytes and you take your code and go to a bigger machine, which has 8 bytes as the size of an integer. Then, you compile the code on that machine and your code will still allocate 10 integers. Why? Because, on the new machine `sizeof(int)` will be automatically 8. So, it will allocate 80 bytes.

So, in order to write portable code, you can use `sizeof(int)`, instead of assuming that, integer is 4 bytes. So, I want to allocate `malloc (10*sizeof(int))`; this will allocate 10 integers no matter, which machine you do it all. So, and it will return you the address of the first byte in that allocated space, that address you convert to an integer array, integer pointer. Here is, how you allocate memory on the heap.

(Refer Slide Time: 10:04)



So, when we think pictorially, think of heap has a separate space in the memory. In this case, ptr will be allocated some space on the heap. Let us say 10 integers on some particular machine, it will say 40 bytes and it will return the address of the first byte. Now, that first byte you treat it as a pointer to int, that is done through the conversion int

*

(Refer Slide Time: 10:42)

The slide has a title 'Removing from the heap' and a bullet point: 'The values allocated on the heap can be removed using `free(ptr)`, where ptr is a pointer which points to values allocated on the heap.' Below the bullet point is a code snippet:

```
#include <stdlib.h>

int main()
{
    int *ptr=0;
    /* Allocate 10 integers on the heap
     * Treat the return address as an
     * integer pointer
     */
    ptr = (int *) malloc (10*sizeof(int));

    /* remove the space allocated on heap */
    free(ptr);
    ptr=0;
}
```

Now, it is nice that you can allocate space on the heap. But, in order to be hygienic, you should also remove the allocated space, once you have done with it. There should be a reverse operation to allocate and that is free, it is in the same library, `stdlib.h`. And if I just say `free(ptr)` and ptr was originally allocated using `malloc`. Then, it will correctly remove,; however, many bytes were originally allocated.

So, let us say that I have `int *ptr` and then ptr, I allocate 10 integers on the heap and ptr is the address of the first allocated location. Now, I may do a bunch of processing here and once I have done, it is just nicer me to de allocate things on the heap. This is like, saying that things on the notice board once some condition occurs, where you know that the notices no longer needed, you just remove that posting from the notice board for that, we use `free` of ptr.

Now, notice the asymmetry here, `malloc` took the number of bytes to be allocated free just wanted to say, which pointer is to be free? It does not ask for, how many bytes to free? So, you can imagine that `malloc` does some kind of book keeping, where it says that I allocated 40 bytes and that was return to ptr. So, if I just say `free(ptr)` it automatically knows that, 40 bytes are to be free, you do not have to give the extra argument saying, how many bytes to free?

Once you free the pointer, you just set it back to null, this is just a safe practice and it is not absolutely necessary,, but it is recommended.

(Refer Slide Time: 12:57)



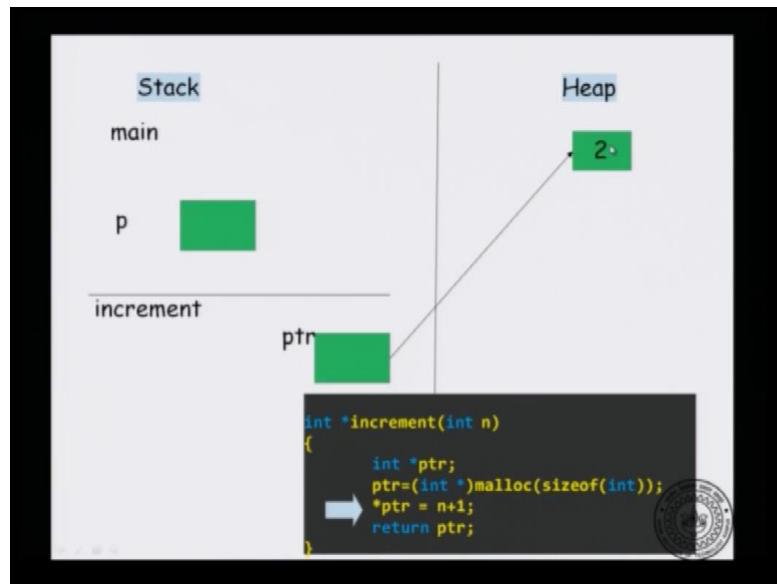
```
■ A Solution to our earlier problem:  
#include <stdio.h>  
#include <stdlib.h>  
  
int *increment(int n)  
{  
    int temp;  
    int *ptr;  
    ptr=(int *)malloc(sizeof(int));  
    *ptr = n+1;  
    return ptr;  
}  
  
int main()  
{  
    int *p;  
    p = increment(1);  
    printf("%d\n",*p);  
    free(p);  
    p=0;  
}
```

So, let us solve our earlier problem using malloc. Our earlier problem was that, ptr was pointing to some location within the stack. So, as soon as the function returned, the return address no longer meant any meaningful address. So, let us now solve this problem. I have included `stdlib.h`, because I will allocate memory on the heap. So, the increment function is modified as follows strictly speaking, I do not need a temp variable any more.

I have an int pointer and I will use the pointer to allocate one integer on the heap, this is a really wasteful practice but, it just illustrates a point. So, it will allocate one integer on the heap and then, return that address and treat that address as `int *`. Now, I will use `* ptr = n + 1` to dereference that location on the heap and set the value to `n + 1`. Once I am done, I will return the ptr I will return the address on the heap.

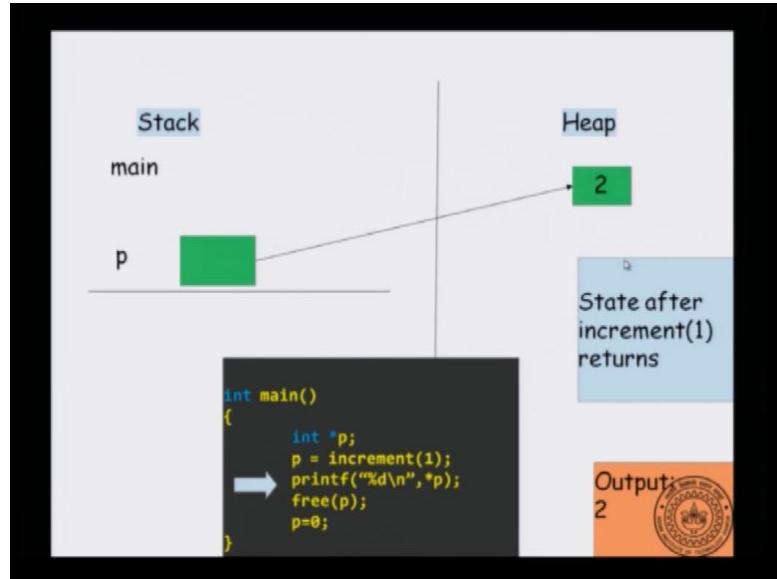
There are differences here is that, their return address is on the heap and only the stack is erased. So, things on the heap are not erased, unless you explicitly ask it to be erased, via `free`. So, returning a heap address and,. So, p will point to a meaningful location on the heap, when you print it, you will get too. And once you have done, you can say `free(p)`. So, here is a strange use which you have not seen before something was done. The `malloc` was done in the increment function and the `free` is being done in the main function. Now, if you think back to the physical analogy, it is not really surprising somebody can post something on the notice board and a different person can remove it.

(Refer Slide Time: 15:12)



What happens here is that, the increment function `ptr` points to some location on the heap, using `malloc`. So, one integer is allocated on the heap and when you say, `*ptr = n + 1`'s, then the location in the heap will contain 2.

(Refer Slide Time: 15:34)



And here is the catch, earlier `p` was just dangling, it was just pointing to an arbitrary location in the memory. But, `increment` allocated something on the heap and returned that address. As soon as `increment` returns, the stack is a waste. So, everything that was allocated on the stack for `increment` is erased,, but things that are allocated on the heap remain. So, `p` points to a meaningful address on the heap, then once you are done you can say `free(p)` and things will be erased, when you print it, the output will be 2.

(Refer Slide Time: 16:22)

Common Errors using malloc and free

- Forgetting to malloc
- Not allocating enough space in malloc (e.g. Allocating len characters instead of len+1.)
- Forgetting to free memory after use (called a memory leak.)
- Freeing the same memory more than once. (is a runtime error)



Malloc and free are prone to a lot of errors and a lot of programming errors in c, can be traced back to incorrect use of malloc and free. So, there are some categories of errors for example, you may forget to malloc in the first place. So, you will lead to dangling references or dangling pointers, as we saw in the first example. Now, you could allocate some space. But, you may not allocate enough space, that is a very common error.

Commonly, you could allocate of by one errors I wanted to allocate really $\text{len} + 1$ number of bytes. But, instead I allocated only len number of bytes. Another very common error is something known as a memory leak, which is that you allocate things on the heap,, but you forgot to free memory after use, this is called a memory leak. Notice that, if you allocate space on the stack, it will always be cleaned up as soon as the function returns.

So, memory leaks usually happen, when you malloc space on the heap. But, you forget to free them, once you have done and a lot of software ships with memory leaks and this is a major concern in the industry. This is also an obscure error, which is freeing the same memory more than once. This is uncommon when a single programmer is working on a code. But, when multiple programmers are working on the same piece of code, you may end up freeing the same memory twice, this will lead to some run time errors.

Introduction to Programming in C
Department of Computer Science and Engineering

(Refer Slide Time: 00:06)

Return the duplicate of a string

■ Write a function to take a string as input, and return a copy.

1) Let the input string be s. s ends in a '\0'

2) Find the number of non-null characters of s. Let this be len.

3) Allocate $\text{len} + 1$ characters on the heap. Store the address in t. (Make sure we have space also for the '\0'.)

4) Copy the contents of s, including '\0', to t.



In this lecture let us look at an application of malloc and free to solve some problem that we are interested in. So, the problem that I will define is to write a function to return the duplicate of a string; a string is given us the argument and you have to return the duplicate of that string. So, we have to write a function to take a string as input and return the copy. Now let us assume that the input string is s, and it ends in a null character. Assume that we can find the number of null non null characters in the string. So, this is will be refer to us the length of the string. What we will do is allocate $\text{length} + 1$ characters. So, there are length non null characters, and then one more for storing the null characters. So, we will allocate $\text{len} + 1$ characters on the, heap using malloc, and we will copy the contents of s to that space on the heap. And finally, return the address of that location. So, that will be the t th the new array. So, notice that the original array may be on the stack, and the new array the duplicate array will be on the heap. Let us write this function.

(Refer Slide Time: 01:29)

```
char *duplicate( char *s)
{
    int i=0;
    int len; /* Length of s, excluding '\0' */
    char *t; /* new string */

    /* Step 2: find Length of s */
    for(i=0; s[i] != '\0'; i++)
        ;
    len = i;

    /* Step 3: Allocate memory for t */
    t = (char *)malloc( (len+1) * sizeof(char) );

    /* Step 4: Copy the contents of s into t */
    for(i=0; i<len;i++){
        t[i] = s[i];
    }
    t[i] = '\0';

    return t;
}
```

So, I will call it duplicate it takes one array which is the same as a pointer. So, I can declare it has `char *`s or `char`s with square brackets, it does not matter. So, I will just declare it has a character `*s`, and what will it return? It will return another array, and array is the same as a pointer. So, I will return character star. So, the input argument is an array, and the output is also an array. I will declare 3 variables; `i` which is for the loop, `len` which will store the number of non null characters in `s`. So, let us be very specific, I do not want to store the number of characters in `s`, because I want to say that I do not want to count null.

Now if you want to count null as well in the length then you will modify the code, but typical convention is that when you mention the length of a string, you do not count the null character. I will also declare a `char *t`. Now the code proceeds and stages; first I have to write a loop to find the length of the string, I can write a very simple loop to do that I can say for `i = 0` as long as `s[i]` is not null, do increment `i`. So, as soon as I see the first null in `s`, I will stop. When I exit out of the loop, `i` will be the number of non null characters in `s`. So, I can say `len = i`. So, in the first step of the function, we just find the length of the string excluding the null character at the end.

Now comes the important thing, we have to copy that array to somewhere. If we copy

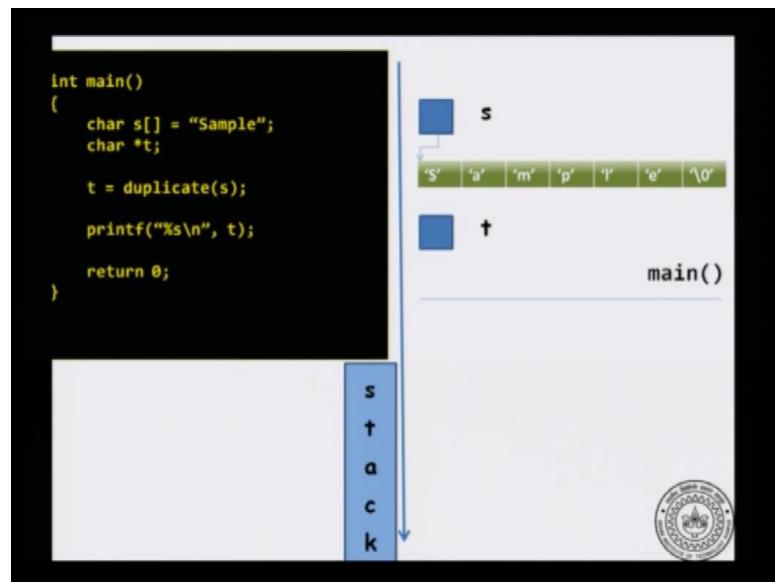
that array to the stack itself that is if I copy that array to some space within the duplicate functions stack, it will be erased when I return. So, I should allocate the space on the heap. I can allocate space on the heap using the malloc function. So, let us look at the malloc function. I want to allocate a bunch of space on the heap, how much do I have to allocate, I have to allocate $\text{len} + 1$ number of characters.

In other words, I have to allocate $\text{len} + 1$ times `sizeof` a single character; these many bytes on the heap. Notice that it is not len times `sizeof(char)`, because if I allocate only that much then I will not have to space to copy the last null character. So, I should make sure the input is a null terminated character, it is duplicate should also be the null terminated. So, I should make space for all characters including the null character on the heap. So, I will allocate `(len+1) * sizeof(char)` many bytes on the heap, it will return you the address of the first byte and that address I will convert to a char star. So, malloc returns a kind of an unsorted. So, here are these many bytes. Now it will return you the address of the first byte that was located, now I want to treat that as a character pointer. So, I will be able to do that using the casting operator.

Why do I have to do that think about it for a minute, because you want pointer arithmetic to work. When I say `t[i]`, I should correctly execute star of `t + i`. So, go back to that lecture and understand why it is important that you know the it is not just a byte address, it is a character pointer. Once you do the allocation, you can copy s array into t array. We do not really care about the fact that t is not on the stack, t is on the heap, because copying is done exactly the same way. So, I can say `i = 0, i < len, i ++ t[i] = s[i]`. And then finally, this will copy all the non null characters, and finally I will say `t[i] = null`, the last character will be the null character.

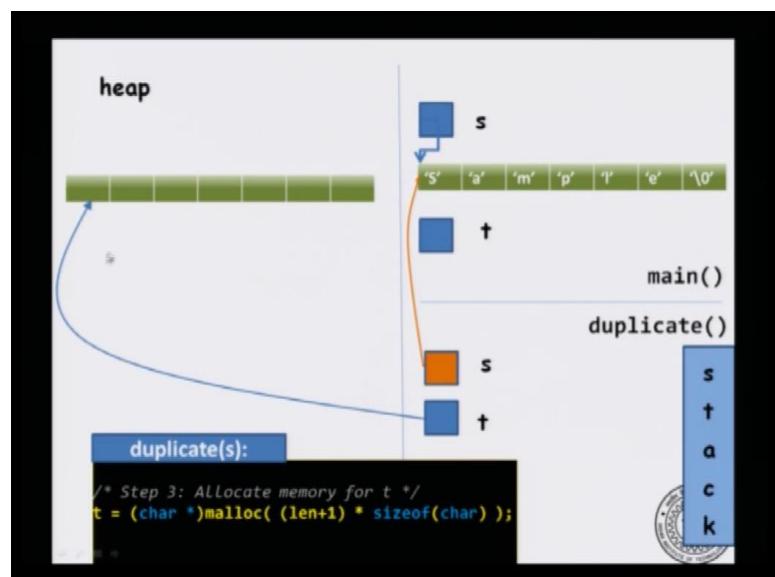
Now, if you want to understand it in slightly greater detail understand why the character star cast was required in order for `t[i]` to work properly. Once I have done copying the array, I can just return t, and I will not leap it will not lead to a dangling pointer because t is allocated on the heap.

(Refer Slide Time: 06:37)



So, let us pictorially understand what happens during the execution of this program. I have main function, and I allocate a char array. Now this is allocated on the stack. As soon as I declare a character array and initialized it with in main, it is allocated in the stack corresponding to main. So, `s` is a pointer to the first location in the array. And I declare another `*t`, and then I call `t = duplicate(s)`, I should return a separate copy of `s`.

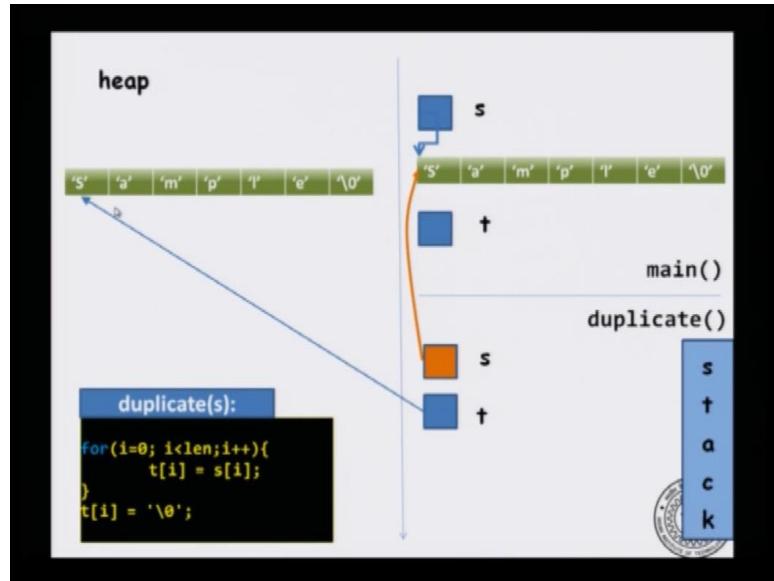
(Refer Slide Time: 07:19)



Let see what happens in the `duplicate` function? We do allocation for all the local

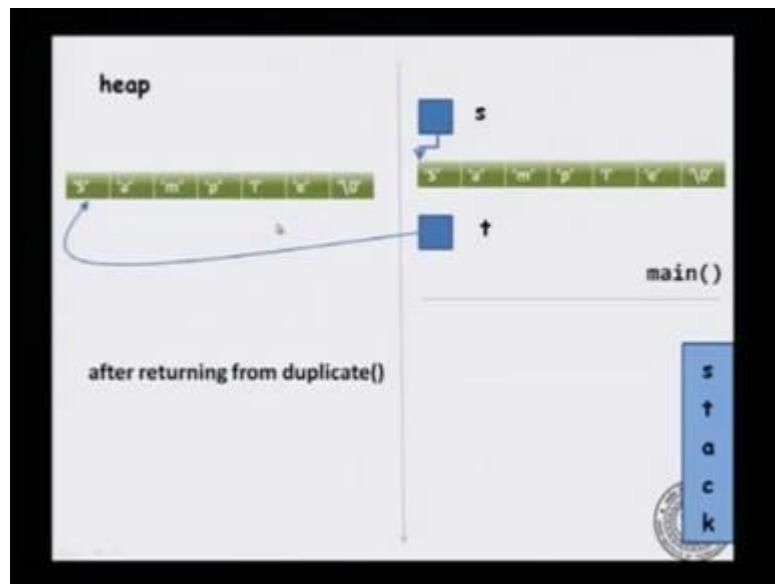
variables all that, but important thing is that we have s and t which are new pointers. Now s is the input argument to duplicate, and it will be pointing to the array in the main function, because I call `duplicate(s)`. So, duplicates s will point to the same array as the s of mean. So, it is pointing to the array on the stack, Now as soon as I allocate memory for t on the heap, which was step 3 of duplicate, I would say `t = (char *)malloc((len+1) * sizeof(char))`. What is len here? len is 6; there are 6 non null characters len + 1 is 7. So, I allocate 7 characters on the heap, and its return address will be cost to a character pointer. So, t is now pointing to this space on the heap.

(Refer Slide Time: 08:28)



Now, once I am done ((Refer Time: 08:33)) creating the space on the heap, what I can do is, I can copy the location the s array into the t array on the heap. So, once that loop executes it will look like this, here is the s array inside main, here is the heap the array allocated by duplicate, and you will just copy `t[i] = s[i]` for. So, you will copy 's' 'a' 'm' 'p' 'l' 'e', that is within the loop. And then finally, I will say that `t[6] = '\0'`. So, here is an array of size 7, it has a 6 non null characters and the last element is null.

(Refer Slide Time: 09:23)



And then once I am done, I will return from `duplicate`. Again keep in mind what is erased is the stack. Everything that was allocated to `duplicate` on the stack is erased. Those `s` `t` and the local variables in `duplicate` no longer exist, but the work that was done by allocating on the heap that still remains. So, the return value `t`, that is return value which is the address of the array in heap will be assigned to `t`. So, `t` now points to heap. Notice how it executed `s` was allocated on the stack, and the effect of the `duplicate` function will be that, the `duplicate` of the array will be created on the heap.

(Refer Slide Time: 10:09)

Summary

- **sizeof operator – to know the number of bytes needed to store a datatype.**
 - ▼ Used in pointer arithmetic
 - ▼ Used in array index calculation
 - ▼ Used when allocating memory on the heap.
- Allocating memory on the heap can be done by malloc.
- Memory allocated using on the heap can be removed by using free.



So, notice what we understood about the **sizeof** operator. **Sizeof** operator was used to know the number of bytes needed at the stored data type. It is used in pointer arithmetic, it is used in array index calculation, and it is also used when allocating memory on the heap, because malloc needed to know how many bytes to allocate. And suppose I have wanted to allocate 10 integers, instead of me saying that on this machine I know that an integer is 4 bytes. So, you go ahead and allocate 40 bytes. The problem with doing that is if you take your code to another machine, and that machine integer is 8 bytes; and your code will no longer allocate sufficient space. So, the real way to write portable code would be to say 10 times **sizeof int**, that code will work regardless of which machine you execute on.

So, here is the use of **sizeof** operator when you call malloc, it helps you to write portable code which will execute on any machine. So, we have seen that allocating memory on the heap can be done using malloc, we have understood what it means to allocate memory on the heap the difference between stack and heap; stack is erased as soon as a function returns, heap is not erased when a function returns you have to explicitly say that i ((Refer Time: 11:39)) now freeing that using free function. Again remember the asymmetry within malloc and free; malloc needed to know how many bytes you allocate, free just needed to know which pointer to de allocate, which pointer to free. Did not want

to know how many bytes to free, it does that automatically.

(Refer Slide Time: 12:03)

Common Errors using malloc and free

- Forgetting to malloc
- Not allocating enough space in malloc (e.g. Allocating len characters instead of len+1 in the previous code.)
- Forgetting to free memory after use (called a memory leak.)
- Freeing the same memory more than once. (is a runtime error)

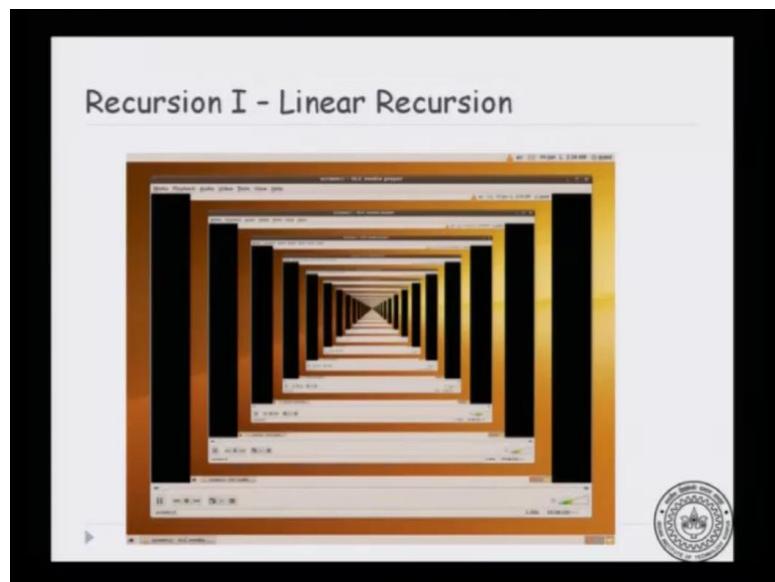


And to repeat common errors using malloc you could forget to malloc. Now you could not allocate enough space in heap. For example, in the code that we have just seen, suppose you are allocated just len number of characters instead of len + 1. Then you would not have enough space on the heap to copy the last null character. So, you will violate that t is an exact **duplicate(s)**. Now you could forget to free memory after use, this is called a memory leak, and you could have the sub square error of freeing the same memory twice, that leaves to run time errors.

introduction to Programming in C
Department of Computer Science and Engineering

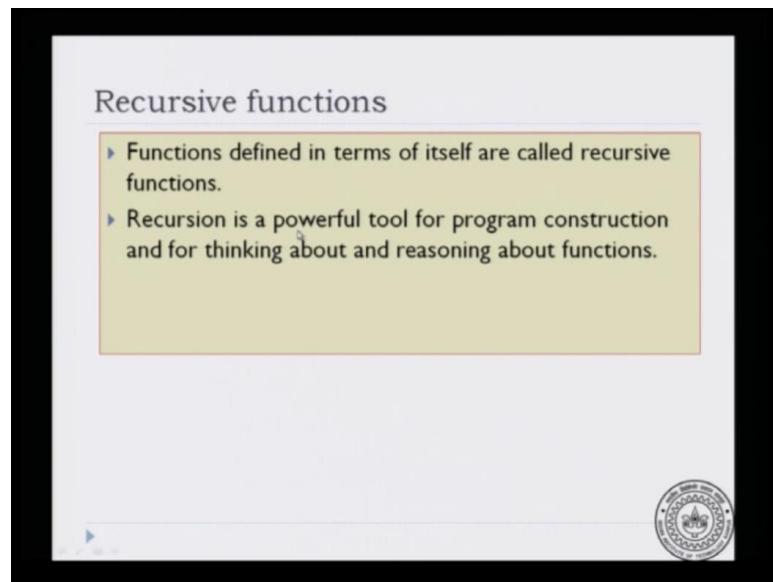
Today's video will talk about an important concept in computer science which is recursion, and we will slowly approach this by looking at various kinds of recursion.

(Refer Slide Time: 00:09)



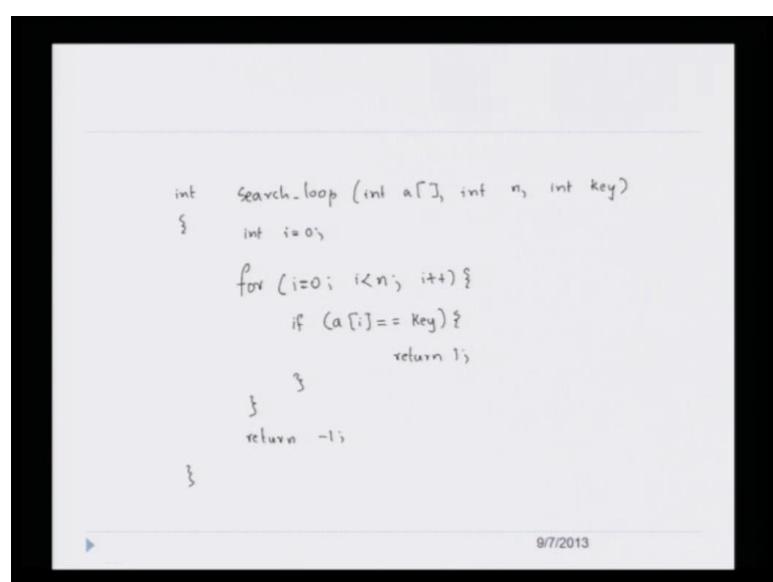
Now, recursion is usually something that is completely new. it is a new way of thinking about problems that might sound unfamiliar at first, but eventually it is a more natural way of solving problems than other techniques. So, we will carefully examine what recursion means. So, this is the video of a media player having a copy of itself inside the video and it goes on forever. We will see what does this have to do with recursion.

(Refer Slide Time: 00:59)



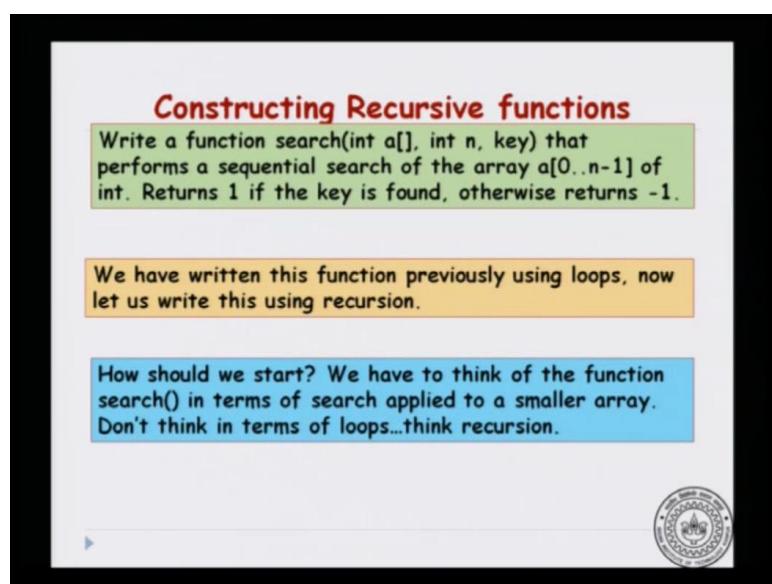
So, recursion in English means roughly say again i am function defined in terms of itself are called recursive functions. Now, this is not completely accurate. We want to say that functions defined in terms of itself in a particular way, these are valid recursions. Recursion is a powerful tool for program construction and for thinking about and reasoning about functions in general. So, it is a general purpose technique of programming, and you can do any kind of program using only just recursion. We will not see such general types of recursion in this course, but we will see fairly common examples for recursions.

(Refer Slide Time: 01:53)



So, for example, let us consider a very simple function which will search for a key within a given array and we know how to write this. What i will do is, i will take an integer, initialize it to 0, for $i = 0$ to n . N is the size of the array. i will increment i and if at any i, i find the key, i will return 1 indicating that i have found the key. if i have not found the key and i have reached the end of theory, i will return -1. This is a typical way to search for a key inside a given array. Now, we will approach the idea of recursion by looking at a recursive solution to this. Hopefully, while seeing this program, we will get an idea of what recursion means.

(Refer Slide Time: 02:52).



So, what do we mean by a recursive solution to this, right. Rather than defining it and describing abstract properties of recursion, why not let write an actual program which is defined in the recursive manner and through these kinds of examples will eventually get the hang of recursion. So, we have to write a function search it will return whether a key is found or not. if the key is found, it returns 1. if the key is not found, it returns -1 and you have to search an array a of size n for the key. Now, we have written this function just now using loops. Now, let us write this using recursion.

Now, what do we mean by solving it in recursive manner? We have to think of the function search in terms of the same function applied to a smaller instance of the problem. So, we have to solve the problem of searching for a key in an array of size n. Can we think of this, in terms of solving the sub problem for a smaller array? This is the

basic question that you have to ask when you want to design a recursive function. So, let us try to in very abstract terms think of how to solve this in a recursive manner.

(Refer Slide Time: 04:24)

```
A General Scheme
▶ search ( a[], n, key )
    n==0      =>      -1      /*array is empty */ BASE
    else      =>
        { a[0] == key => 1. /*key found */
        { else         => search ((a[1]), n-1, key)
                                /*search subarray */
        }
    Inductive Case
```

So, let us say that i will in some unspecified syntax, this is not going to be valid c, but this is just so that we see the idea in a very clear manner. i have to search for an array of size a of size n for key. Now, if the array is empty that is n is equal to 0, you can have more conditions here. N can be < 0 as well, but let say that empty array is n is equal to 0, then you say that i have not found the key because it is an empty array. So, you give back the value-1. So, $n = 0$ implies the value to be returned this-1. That is what this notation is supposed to stand for.

Suppose n is not 0, so, this means that an array is non-empty. Now, how do we solve this, recursively right. So, this look for the first element whether it is the key or not. if the first element is the key, we do not have to do anything further. We know that the key is present in the arrays, so you return 1. So, the key has been found and you return 1, and now is the big step for recursion. How can we search for the key in an array of size smaller than n? So, if $a[0]$ is not equal to key, then this means that key can be somewhere in a 1 through a $n-1$ or it is absent in the array. in any case what we now have to do is search for the arrays starting at a 1, so by a 1 this is not strictly c notation. What i mean is the sub arrays starting at a $+ 1$. So, search in the sub array starting at a $+ 1$. Now, the sub

array has one element less because we already know if you are here that a 0 is not equal to key, so there are only $n-1$ element in the smaller sub problem.

What do we have to search for, we have to search for the key. So, this says that either the key is present as the first element of the array or you have to solve the sub problem of searching in the sub array of size $n-1$ for the same key. So, here is the key to thinking about a problem in recursive terms. What you first do is, consider the case when you have the trivial array which is the empty array in this case. So, we have the base case and then, these are the recursive case. So, the recursive case consists of doing something at size n . So, in this case, it is search whether the first element is the key or not. if it is true, then we do not have to do anything further, we have found the key, otherwise solve the sub problem.

Now, the sub problem is a smaller copy of the old problem. So, this is what is known as the inductive case or the recursive case. The reason i am calling it inductive case is that recursion has very tight connections to the idea of mathematical induction. if you know how to write a proof by mathematical induction, what you normally do is you consider a base case. So, you have a theorem and you want to prove this by mathematical induction. You consider the base case probably $n = 1$ or $n = 0$. These will be the base cases for an association about natural numbers and then, if the base case is true, then you say that i assume that the problem is true for size n and now, i want to prove that the theorem is true for size $n + 1$. This is how a mathematical induction proof looks like and in the case of a recursive program, there is a very tight analogy.

Recursion in fact is just a mathematical induction in the context of writing programs. We have to solve a problem. First we will see what is a problem in the base case and the base case is a very trivial case usually, but it is important that you think about base case. You say that if the array is empty and then, i will return -1 because the key cannot be in the array. Then, you say that i will now define the problem of size n in terms of a sub problem of size $n-1$ for example. So, we will solve the same. We will solve the bigger problem in terms of a smaller copy of itself and this is the key to thinking about recursive programs.

(Refer Slide Time: 09:52).

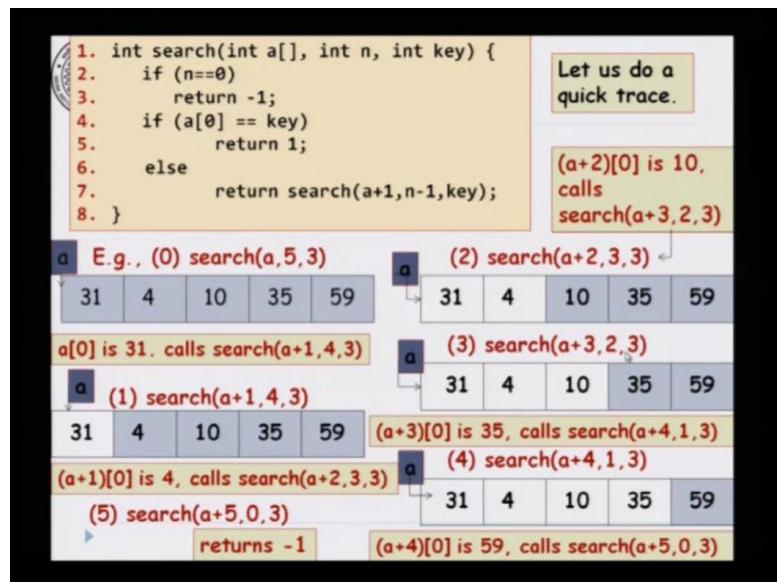
```
/*
=====
Recursive function to search for key in array a with
elements. Returns 1 if key is present in array, -1 otherwise.
=====
1. int search ( int a[], int n, int key ) {
2.     if ( n==0 )
3.         return -1;                      /* Base case */
4.     if ( a[0] == key )
5.         return 1;                       /* found */
6.     else
7.         return search(a+1,n-1,key); /* recursive call */
8. }
```

9/7/2013

Let us code this in c. So, we code this in a very straight forward manner. i will write a int search, **int a[]**, int n which is the size of the array a, int key which is the key we are searching for. if $n = 0$, then return-1 because the key is not found. This is the base case and otherwise $n > 0$, so you can search for a 0 is equal to key or not. So, you can search for whether the first element is the key. if it is, then you have found the key, otherwise what you do is you call search a + 1 which is the sub arrays starting at size 1. The sub array has size $n-1$ and key.

So, when you search or write a recursive program, there are a few things that you want to check. The first is that the base case is properly handled. The second is that when you define the sub problem, you want to ensure that it really is a sub problem because if you solve the problem in terms of an equal size problem or even a bigger size problem, your program may not terminate. We will see this in a moment. So, this part which is highlighted in green which is calling search itself, but on a smaller sub problem is a + 1 $n-1$. This is what is known as a recursive call to the same function. So, we have seen functions that can call other functions. Now, we have seen functions which can call themselves and this is what is known as recursion.

(Refer Slide Time: 11:46).

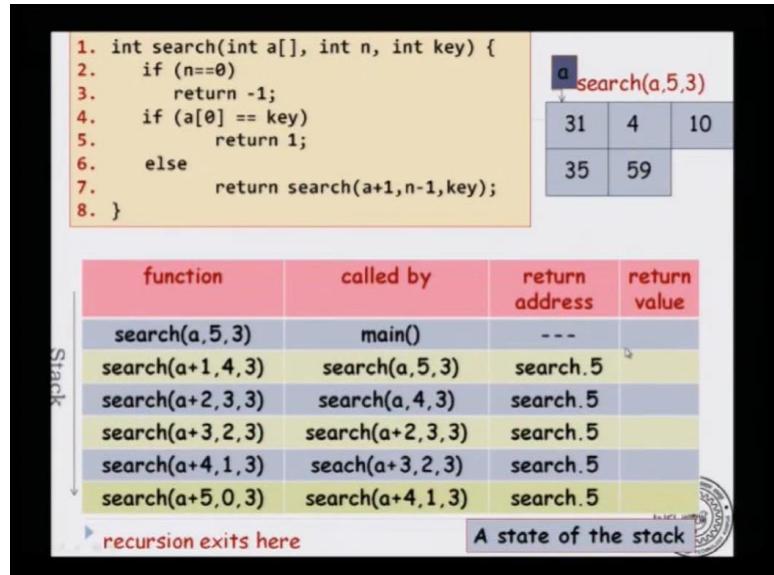


Let's see how this function behaves. Now, before we go into the execution trace of this function, I want to add a word of caution. The actual way to understand recursion is not to think about the stack and how functions are calling other functions. The real way to understand recursion is to think about this program as a problem defined in terms of sub instance. But in any case we will just see the execution of this function through the stack trace just to get comfortable with what happens at the back of all of this. So, let us do a quick trace.

Suppose we have an array 31 4 10 35 59. It is an array of size 5 named a, and we are searching for the key 3. Now, we know that this key is not present in the array, but let see how the function executes. So, first we call `search(a,5,3)`. A 0 is 31 which is not the key. So, it calls `search(a+1,4)` because now we are searching in the sub array of size 4 for the same key. So, that is in effect, the same as calling the same search function on this sub array highlighted in grey. This is because the answer to search in the whole array is now the same as answer to the search in the sub array. That is what the recursive statement is. Now, `(a+1)[0]` is 4 this is the first element of the sub array. A 4 is not 3 and a, and at this point you call the sub sub problem which is `search(a+2,3)`, the sub array of size 3 for the key 3. Here is the sub array of size 3 and you are searching for 3 in this sub array. Again the first element of the array is 10, which is not 3. So, you call the sub problem of this which is `a+3`. Now, the array is of size 2 and you will search for 3 and this goes on until you find that you have exhausted the array. Finally, the array is of size

0 and you will finally say that since the array is of size 0, i have not found the key. So, you return -1.

(Refer Slide Time: 14:30)



Let us just look at this stack of function cause and see how it looks like. **Search(a,5,3)** is called by mean and let say that it has some return address. We do not care about it right now, but **search(a,5,3)** calls **search(a+1,4,3)** and the place to return is some line in search function. This calls the sub sub problem **a + 2 3** that calls **a + 3 2** that calls **a + 4 1**, and that calls **a + 5 0** at which point you realize that the sub problem now is empty and then, you return **a-1**. So, at this point you have reached the base case. if **n = 0**, return-1. So, that will return **a-1**. Where will it return to? it will return to the function which immediately called it which is **search(a+4,1,3)**. So, this guy gets **a-1**. Therefore, it just returns **-1**, return the value of whatever is returned by the sub problem, ok. So, it is **-1** and **-1** gets returned. So, it gets bubbled up all the way back to main, and main you can realize that the element is not present in the array because the return value of **search(a,5,3)** was **-1**. At this point, the call stack terminates.

(Refer Slide Time: 16:26)

The screenshot shows a slide from a presentation. At the top left, there is a code snippet for a recursive search function:

```
1. int search(int a[], int n, int key) {
2.     if (n==0)
3.         return -1;
4.     if (a[0] == key)
5.         return 1;
6.     else
7.         return search(a+1,n-1,key);
8. }
```

To the right of the code, there is a call stack diagram with the label "search(a,5,3)" at the top. The stack contains four boxes representing the state of the variables:

a	search(a,5,3)	
31	4	10
35	59	

Below the call stack, a green box contains the text: "search(a, 5, 3) returns -1. Recursion call stack terminates." A small circular logo is visible in the bottom right corner of the slide.

So, what was special about the recursion call stack? it was just that most of the stack was involved by a function calling itself over and over, but each time the function called itself, it was calling on a smaller version of the problem. Here is how, you think about a very simple program in terms of recursion. Earlier, we saw how to solve this using iteration which was using a loop and we have seen the problem how to be solved using recursion.

Now, a word of caution, we will see this in further examples. it is very important that you handle the base case properly. Now, this is something that we are not used to in normal way of thinking. When we think about solving a problem, we are thinking about solving substantial sizes of the problems. We are not concerned too much with what happens with an empty array, what happens when n is -1 and things like that, but even in this problem, we know that when we call `search(a + 5,0,3)` we know that the function terminated because we had a base case which said that if n equals to 0, then return -1. if we did not have this case, you could see that probably it will go on calling itself infinite number of times. So, just like when you are writing for loop or a while, loop you have the case of infinite loops. in the case of recursion, you can have an infinite recursion and you have to guard against that. The only way to guard against that is to get the base case correct. So, here is something in counter intuitive about programming recursive functions. You know almost half of your intellectual effort is in handling the base case properly, and only the remaining is involved in solving the recursive case.

Introduction to Programming in C

Department of Computer Science and Engineering

(Refer Slide Time: 00:06)

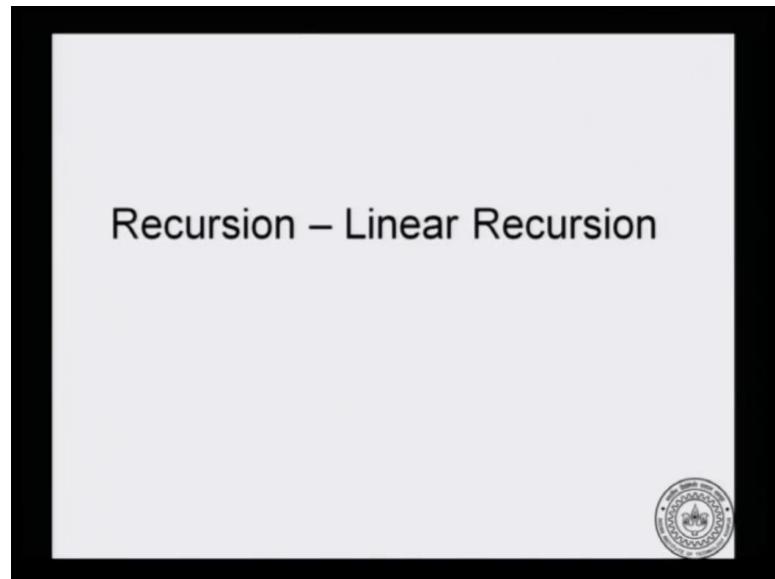
Designing recursive programs

- While designing recursive programs, think recursively...not in terms of the stack.
- Stack is created and used for execution (tracing).
- Depth of recursion refers to the maximum size of stack (i.e., maximum number of pending function calls).
- Memory used by a recursive programs includes the local memory + stack depth



In this video will look at linear recursion in a bit more depth, while I describe what I mean by linear recursion. We have mentioned earlier that when designing recursive programs, think about the problem in recursive terms, do not think in terms after stack that is used in execution. When it is actually executed that will be a stack created and use for the execution, and that depth of recursion is a term which means the maximum size of the stack, while you execute the program on given input. The memory used by the programs includes the local memory of the function, + the depth of the stack.

(Refer Slide Time: 00:51)



So, let us look at linear recursion in a bit more detail. By linear recursion I mean problems which can be solved by calling, and instance of the sub problem, exactly one instance of some sub problem. We will see more general kinds in later videos.

(Refer Slide Time: 01:11)

Example 2: In-place reversing an array

Write a function `reverse(int a[], int n)` that reverses the values contained in the first n indices of $a[]$. That is, $a[0]$ and $a[n-1]$ are exchanged, $a[1]$ and $a[n-2]$ are exchanged, and so on.

reverse (a,n): formulating the problem recursively

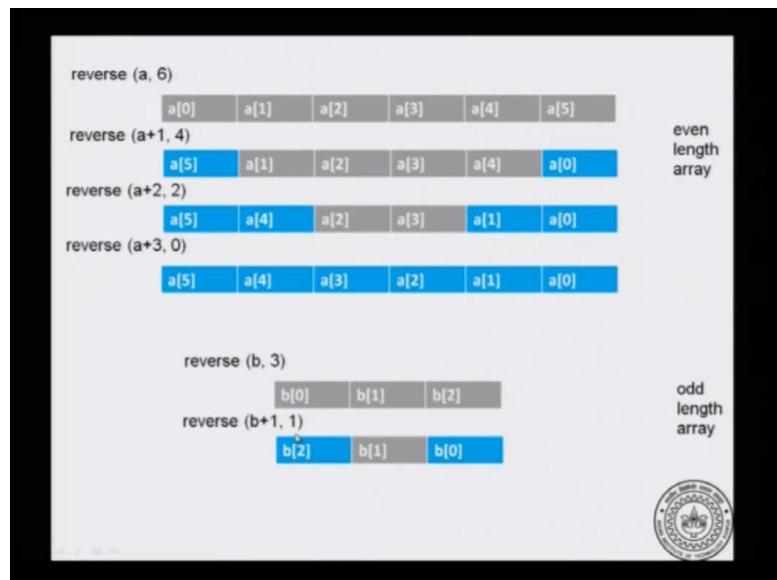
Basic idea:

1. if n is 0 or 1, return. Nothing to reverse.
2. Otherwise,
 - a) exchange $a[0]$ with $a[n-1]$.
 - b) call `reverse` on array starting at position 1 and of size $n-2$.

Let us look at an example that we have seen before, which is reversing an array in place. So, we had to reverse an array a with n elements, and it is suppose to reverse the values contain in the first n indices of a ; that is $a[0]$ is exchange with $a[n-1]$ $a[1]$ is in exchange with and so on. So, we have to do $n/2$ exchanges approximately. So, let us look

at the problem recursively, we had earlier solve it using loops. Now the basic idea of the recursive solution to, in place reversal, is the following; if n is $a[0]$ or 1 if thus arrays either empty are it contains exactly one element, then we do not need to do any think to reverse theory; otherwise it contains at least two elements. In this case exchange $a[0]$ with $a[n-1]$; that will be the first layer. Now call this sub problem. We have to solve one more sub problem which is reverse on an array, which is from $a[1]$ through $a[n-2]$. Notice that we had already solved the problem of swapping $a[0]$ and $a[n-1]$. So, earlier we had seen a program which involved linear recursion, which just bend left to right. In the case of reversal f on array, it is still linear recursion in the sense that there is only one call to a sub problem, but the way in which you call the sub problem is slightly different.

(Refer Slide Time: 03:02)



Let see with an example. We will consider actually two examples; one for even length array, and another for an odd length array. Let say that a is an even length array with six elements and we want to reverse it, using the function $\text{reverse } a[6]$, and we have to do it in a recursive way. So, what you do is first swap $a[0]$ with $a[5]$. And now what is the sub problem left be solved. We have to solve reverse of this intermediate array, which starts from $a[1]$ and contains four elements. So, we have to reverse the array with starts from $a + 1$, and there are four elements to be reversed. So, in one step even though we have only a single call to a sub problem, we have actually reduced the size by 2. Now use a reverse or rather swap a and $a[1]$ and $a[4]$, and now the sub problem that remains is, to reverse this sub routine which is $a + 3$, and you have two elements to reverse. So, you do this,

and in this point you have a sub array which starts at $a + 3$ and has zero elements to reverse. At this point that is nothing but.

Now, for in odd length array let us take a very small array which contains three elements, and we have to reverse it. What you do is, you reverse you swap $b[0]$ with $b[2]$. At this point you have a sub problem which has exactly 1 element, and you do need to reverse that array, that arrays it is soon reverse. So, the problem just stops there. So, notice that difference between the even length array the odd length array. In the case of even length array, the step, just before the last step involved an array of size 2, and you still had to reverse that is array. In the case of an odd length array this, the last of involves has a single length array, which is soon reversed. So, you do not have to do anything. So, there are two base cases to worry about; one is where the sub array is of size zero, and another is where the sub array is of size 1, 0 corresponds to even length arrays, and one corresponds odd length arrays.

(Refer Slide Time: 05:31)

Example 2: In-place reversing an array

reverse(int a[], int n) to reverse the values contained in the first n indices of a[].

```
void reverse(int a[], int n) {
    if (n==0 || n==1) return ;
    else {
        swap(a,a+n-1);
        reverse(a+1,n-2);
    }
}
```

Space requirement is : linear in n.
Why? because stack depth is $\text{ceil}(n/2)+1$.



Let us write this code now. So, we have reverse a containing n elements, and we have return type void, which means that this function is not going to return you value, but it is going to do something. So, if $n=0$ or $n = 1$ return, because in that case a is, it is on reverse; otherwise you swap the first element with a last element, that is this operation a and $a + n - 1$. So, notice that swap is a function that takes two pointers to int and exchanges them. Once you do that you call the sub problem, which is $\text{reverse}(a + 1, n -$

2). Notice that unlike the previous examples we have discussed, the sub problem reduces by 2 insides. Even though you have a only single call, the sub problem is not of size $n - 1$, it is of size $n - 2$. So, look at the case of the odd length array and the even length array that we have seen before. And you can notice that the sub problem reduces by 2 in size for every step. Now what is the depth of the stack. you know that ruffle n upon to calls will be done, because you start at a size n , the next call will be of size $n - 2$ and so on until you hit either one or zero. So, you can work out that there will be about $n/2$ steps, before you reach one or zero. The accurate expression is, ceiling of the expression $n / 2 + 1$. So, many calls will be there, before you hit 1 or 0. So, each function call will take, let us a constant among space and there are about $n/2$ function calls. So, the stack depth is $n/2$, and therefore, the wholes space which is stack depth times the number of variables at each function that will be about $n/2$.

(Refer Slide Time: 07:58)

Example 3: Array Maximum

Find the maximum of the numbers in an array.

```
int max_array(int a[], int n);
If n is of size 0 then we return some really large -ve value.
If n is of size 1 then just return a[0].
If n has size >=2...let us see an example.
```

$\max \{1, 2, 3\} = 3$
 $\max \{1, 2, 3, \phi\} \geq \max \{1, 2, 3\}$
 $\max S \geq \max \phi$
 We set $\max \phi = -\infty$


So, now let us consider a third example which is, computing the size, the maximum of a particular array. For concreteness let us consider in integer array, and we have to compute the following function int max array. It takes two arguments; one is the array itself, and the second is m, which is the number of elements in the array. Again let us think about the problem recursively, we have return loops to solve the problem earlier, but now let us think about it in a recursive manner. If the array contains 0 elements, then what is the maximum. So, here it may be slightly counter intuitive if you are saying for the first time. The maximum of an empty array is some large a negative value. Think of

it has minus infinity. Why do we do this this is, because let us take a concrete example 1 2 3. We know that the maximum of this array is three. Now, what happens when you take a larger array or list of numbers. So, what happens if you take, let us keep this unspecified a is an int. You know that if a is less than 3 then the maximum of this array is going to be three. If $a > 3$ then the maximum of this the second one is going to be greater than that. So, in any case, whatever be the nature of a you can always say that $\text{maximum } \{1 2 3\}$ a is going to be \geq the $\text{maximum } \{1 2 3\}$. Now what; that means is that, if you take a larger set, its maximum is always going to be \geq the maximum of a sub set.

Note that this is independent of a , because you can analysis my cases, if $a \leq 3$ then this maximum will 3 itself and $3 \geq 3$. If $a > 3$, then this maximum is strictly greater than the previous maximum. So, maximum is always monotone according to the sub set relation. Now this means that what will be the maximum of the empty set. The empty set is a sub set of every set. So, no matter which s I pick, maximum s has to be \geq maximum of the empty set. This means that a reasonable value for maximum of empty set is minus infinity. So, the set... So, this is a reasonable convention; that is why when n is of size zero, we returns some really large negative value. By which I mean the absolute value of the number is really big, because we are trying to say that it essentially minus infinity. If n is of size 1, then you just return $a[0]$, because the array contains only 1 element, it is maximum will be a zero. If n has size at least 2. Now we are in business, we have to solve the problem in terms of a sub problem. So, here was an example where the base cases had to be really thought of, but now we are at the case where we are thinking about the recursion. So, what is the recursive step here.

(Refer Slide Time: 12:10)

Example 3: Array Maximum

Find the maximum of the numbers in an array.

```
int max_array(int a[], int n);  
If n is of size 0 then we return some really large -ve value.  
If n is of size 1 then just return a[0].  
If n has size >=2...let us see an example.
```

a

recursive call: `max_array(a+1, n-1)`

- 1 maximum value is the larger of `a[0]` and the maximum in the range `a[1..n-1]`.
- 2 This is computed by a recursive call:
`max_array(a+1, n-1)`.



So, let us take a concrete array. We have array a , which contains the numbers 2 4 3 7 5 23, -3 and 9, some concrete array. And I want to calculate the maximum of the array a in terms of some sub problem. The natural sub problem that we can think of, is the sub problem of finding the maximum of this sub array, which start from $a[1]$ and goes on until the last element. So, recursive call should be something like max array $a + 1$, and there are $n - 1$ elements in it, because we omit the first element. Now, maximum value, how can we solve the whole problem in terms of the sub problem. Suppose we note what is the maximum value in the tail; $a + 1$, 2 containing $n - 1$ elements. The maximum of the whole array will be the greater of the two numbers, which two numbers, the maximum of this sub array and $a[0]$. So, maximum value is the large of a zero and the maximum of the tail sub array, which is $a + 1$ to $a + n - 1$. Now in order to compute the sub problem we called a recursive call to the same function, looks for the max array from $a + 1$ containing $n - 1$ elements.

(Refer Slide Time: 13:55)

Array Maximum

Find the maximum of the the numbers in an array.

```
int max_array(int a[], int n);
If n is of size 0 then we return some really large -ve value.
If n is of size 1 then just return a[0].
If n has size >=2...let us see an example.
```

a [25 | 4 | 3 | 7 | 5 | 23 | -3 | 9]

recursive call: max_array(a+1,n-1).

It should return 23.
We return the larger of the returned value and a[0].

Let us write this code.



So, in this example, the maximum of the tail sub array will be 23. And let say that a zero is 25. So, the maximum of the whole array will be the greater of the two numbers 25 and 23. So, in this case, the maximum value will be 25 which is **a[0]**.

(Refer Slide Time: 14:26)

Find the maximum of the the numbers in an array.

```
int max_array(int a[], int n) {
    int maxval;
    if (n == 0) return -99999; /* some Large -ve number*/
    if (n==1) return a[0];    /* 1 element array */
    /* otherwise n >=2. */
    /* Find the Largest element in the array a[1..n-1] */
    maxval = max_array(a+1, n-1);
    return max(a[0],maxval);
}
```

How good is this program? Is it better, equal or worse than a standard iterative program we would write.

The questions are?

- 1. How much time does the recursive program take?
- 2. How much space (including stack depth) does it consume?

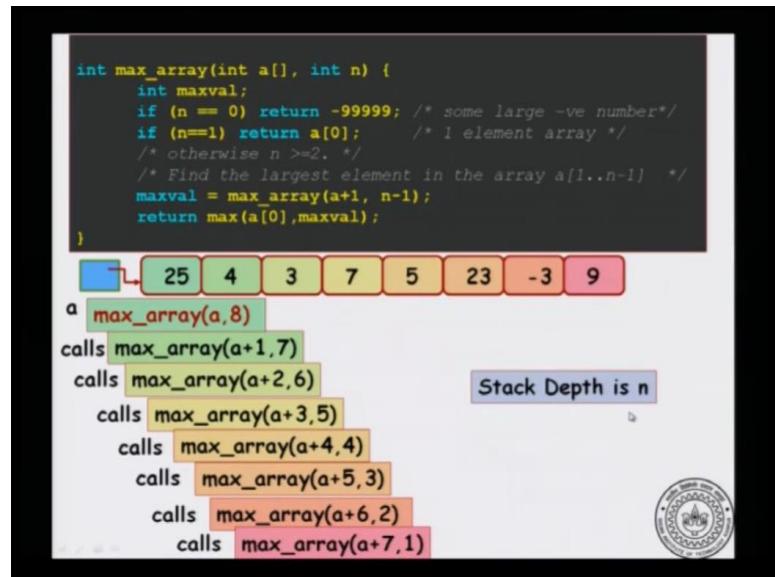


So, now let us write this code. So, the recursive function is very simple, and this is one of the reasons why people like to write recursive functions, because from a recursive function it is very clear what the function is going to do. Usually recursive functions are shorter than their loop versions, and they are easier to understand when you read

someone else code. So, let us solve max array using the recursive function in c. We have int max array, because it is finally, going to return in int value which is the greatest value in the array. Now you have an int array a , and n is the size of the array. Let say that we set some max val if n is 0, then the maximum is simply something like minus infinity. Let us keep it a very large number - 9 9 9 9 9. So, - 5 9, some large value does not matter, and then if $n = 1$ then the array contains only one element and therefore, it is the maximum. So, you just return $a[0]$; otherwise n is at least 2. So, in this case, you say that the maximum value of the sub problem is max array $a + 1, n - 1$. So, this is the maximum of the tail array.

Now once you have the maximum of the tail array, the maximum of the whole array is the greater of the two numbers which is $a[0]$ and max val. So, we return $\max\{a[0], \text{max val}\}$. Now max is a function that is already there in the standard math library in c, but if you want to write it, it is not a difficult function to write it, you can take two integers and return the greater of the true integers. Now we can think about is a better then the loop version of the program. The advantage of the recursive program is that, it is easier and in some sense it contains fewer number of lines then the loop program. The disadvantage is that it takes subs more space while executing. So, the questions are how much time does the function take, how much space does the function take. So, these are things which are concrete and can be measured, there is also software question which is, how you see is set for programmer to look at this function and understand what it does. In the second criteria and it is the recursive function that is course. In the first criteria it is often the iterative function, the loop function that is course.

(Refer Slide Time: 17:39)



So, please think about the questions, and you can work through it and say that in order to solve `max_array` of an array of size n . Let us take an array size 8, you will see that these other recursive calls it will make (a + 2,6) so on up to (a + 7,1), and when you hit an array of size 1 you get to one of the base cases, which is that when you have an array which contains a single element, the maximum is the only element in the array. So, once you hit here, you will start returning. So, the maximum depth of function calls in this will be the size of the array. So, you can say that stack depth is n .

(Refer Slide Time: 18:25)

Recursive procedures are general. They need not be used only for array computations.

The questions are?

1. How much time does the recursive program take?
2. How much space (including stack depth) does it consume?

Any guesses: for 1 and 2 ?

Now, recursive programs are general programs, just like look loop programs are general programs. You have return loops even before you saw what are arrays in C? Similarly you can write recursive programs which deal with general data, not just array data. And in all of these questions, you can ask the following question how much time does the function take, and how much space does the function take.

(Refer Slide Time: 19:01)

Recursive functions are very general. They need not be used only for array computations only.

E.g.. Write a program that reads n , and then n numbers and returns their maximum. (no arrays are used.)

```
int read_max(int n) {
    int x;
    if (n == 0) return -99999; /* some large -ve number*/
    scanf("%d",&x);
    if (n==1)
        return x;
    /* otherwise n >=2. */
    /* Find the largest element in the next n-1 inputs */
    return max(x, read_max(n-1));
}

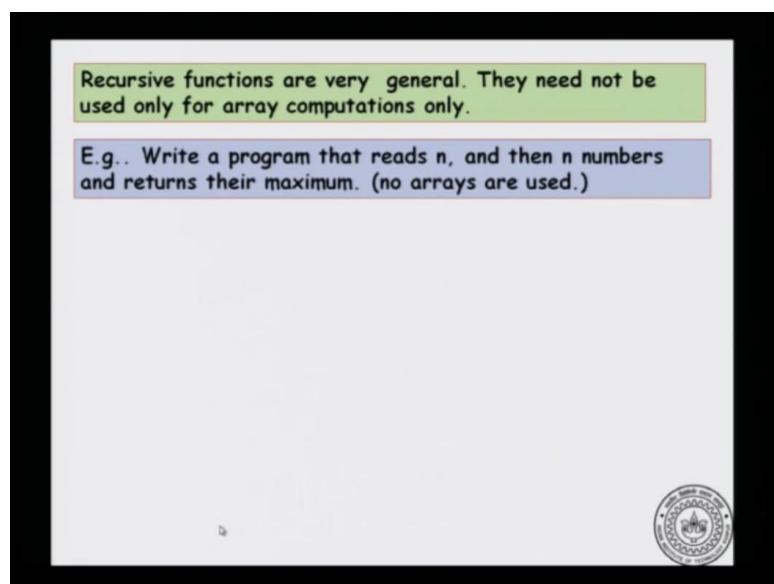
int main() {
    int n; scanf("%d",&n);
    printf("%d ", read_max(n));
}
```

We will see an example for a recursive function, that will read n numbers and returns the maximum. Before we came to know of C arrays, this is the kind of loop functions that we used to write, we would take n numbers. So, first you will read how many numbers to read, then you will read exactly those many numbers and find their maximum using a loop. Now Let us try to do that using recursion. We are not going to use any arrays. So, what we have to do is, write a function to `read_max`, it takes n elements. And the logic is the same as finding the maximum of an array, but we will do it without using arrays. How do you do this. If you have zero numbers to read then you return minus infinity, or some approximation, some large negative value; otherwise you read the first number. If $n = 1$; that is we have to read only one number, then you just say that x the maximum; otherwise $n \geq 2$, and we have read one number.

So, you say that return the maximum of the two values, which is x and what goes inside, inside you have to solve a sub problem, which is the sub problem of reading $n - 1$ numbers and returning the maximum. Go back and compare the program with finding

the array maximum, and the recursion works exactly in the same way. So, we will read $n - 1$ numbers, and return the maximum of those, and then you compare maximum of the first number and the maximum of the sub problem. This is exactly as before except that we did not use any arrays. And how do you call this function, you just declare a main function with n , you scan it how many numbers to read, and call the function read max n . Finally, it will return the maximum of the n numbers read and you just print the value. So, think about this for a minute, and see why we did not need to use arrays.

(Refer Slide Time: 21:29)



(Refer Slide Time: 21:40)

Standard arithmetic functions are sometimes defined recursively.

E.g.. A gcd function (Euclid's algorithm).

```
int gcd(int a, int b) {
    /* assumes a >= b */
    if (b == 0)
        return a;
    return gcd(b, a%b);
}

int main() {
    int g;
    ...
    if (a < b)
        swap(&a,&b);
    g = gcd(a,b);
}
```

Which is better? the recursive formulation or the iterative formulation?

- 1 Logic is the same.
- 2 So the number of steps is of the same order.

Space used?

- 4 Iterative program uses 3 variables.
- 5 Recursive program: What is the stack depth for this program?

Now, there are other functions which are typically return in a recursive manner. We just saw that you can use recursion with arrays. We saw problems where, you do not need to use arrays, but you can still write a recursive routine. We now will come to arithmetic functions, and many arithmetic functions are of an recursively defined. For example, let us take to function Euclid's algorithm, and you can write the **GCD** function as follows. You first ensure that **a >= b** using the swap function, and then you just call **gcd(a,b)**. And **gcd(a,b)** is defined a recursively as follows. If **b = 0** then **gcd(a,b) = a**. If **b** is non zero then you just **return gcd(b, a%b)**. So, this is how you write recursive **GCD** routine. And I will make the clean that this routine is cleaner, then the iterative routine. In the iterative routine, remember we had to use an intermediate variable, which will store the value of, let say **a** and then did a careful three way exchange in order to accomplish **b, a . a % b**.

Here the code is very simple, if **b** is zero then we know the **gcd(a,b) = a**. If **b** is non zero then we know the **gcd(a,b)** is **gcd(b, a%b)**. So, it is a very concise way of writing the function. Now you can ask the question which is better, is the recursive formulation or the iterative formulation. Logic is the same, so it will take the same number of steps. So, the time taken will roughly be the same. And we have also made the clean that there recursive version is easier to understand. The disadvantage may be the following that, the recursive function may use very deep stack. So you can ask the question like how deep will be the stack in the case of the recursive program. So, in the following video, we will talk about more general kinds of recursion. In this video and the previous video, we have seen recursive problem. So, it can be solve by one call to a sub problem, and we will see more general kinds of recursion.

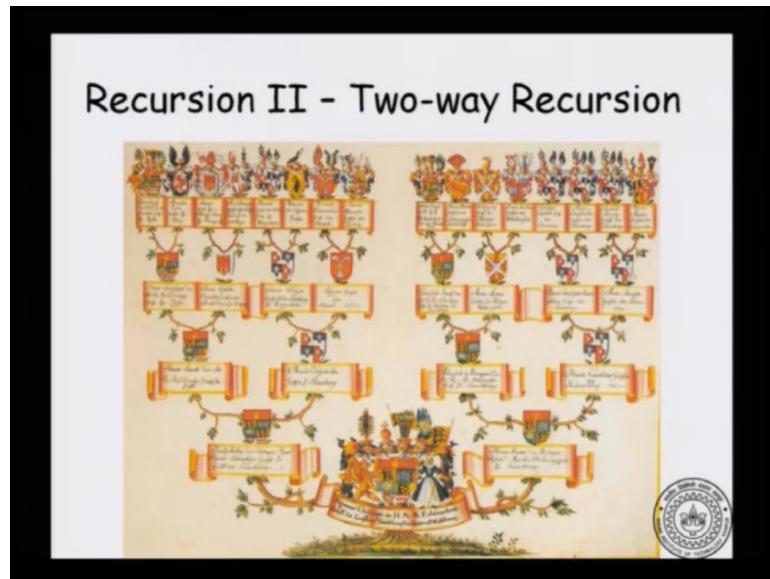
Thanks.

Introduction to Programming in C

Department of Computer Science and Engineering

In this video we will look at slightly more general way of defining problems through recursion.

(Refer Slide Time: 00:10)



We will, for the lack of a better name, I will call it just two-way recursion. These are problems which are solved by calling two-sub instances. This is the picture of a family tree, and we will see that the call stack for a two-way recursive functions looks somewhat similar to a family tree.

(Refer Slide Time: 00:32)

Find the maximum value in an array

Can we reduce the stack depth?

- 1 Divide the array a into about two equal halves: $a[0 \dots n/2 - 1]$ and $a[n/2 \dots n-1]$.
- 2 Recursively find the maximum element in each half and return the larger of the two maxima.
- 3 Base cases: If n is 1 then return $a[0]$, if n is 0 return $-\infty$.

Let us revisit a problem that we have seen which is to find the maximum value in an integer array. We saw that the stack depth in our earlier solution was order n , because each problem of size n called once at problem of size $n - 1$. Now, can we reduce the depth of the stack from something close to n to something smaller than n .

So, here is an alternate way to look at the problem which can be described in a very simple way. Instead of looking at the maximum of the first element and then the tail, what I can do is, take an array of size n and split it roughly in 2 halves. So, there is left half and a right half, each of size n over 2.

Now, imagine that you have the solution for the greatest element in the first half, let us call that x . And imagine that you have the greatest element of the right half, let us call that y . Now, whichever is greater among x and y , is going to be the greatest in the whole array. And this is the idea that we are going to implement right now.

So, divide the array into about 2 equal halves. The first half is 0 to a $n/2 - 1$; this contains $n/2$ elements. And the second half is, a $n/2$, so on, upto $n - 1$; this is the right half. Now, recursively find the maximum element of each half. And let us say that you have x which is the maximum in the left half and y which is the maximum in the right half, then you just return the larger of x and y , that should be the largest element of the array.

While doing this we have to take care of the base cases. This is as before for the linear case; when $n = 1$ then the only element in the array is the maximum element, so, return

a[0]. If **n = 0** that is the array is empty, you return minus infinity. So, let us consider a concrete array; **a**, is an integer array with these elements.

(Refer Slide Time: 02:55)

```
Find the maximum of the numbers in an array.  
(linear version)
```

```
int max_array(int a[], int n) {  
    int maxval;  
    if (n == 0) return -99999; /* some large -ve number */  
    if (n==1) return a[0];      /* 1 element array */  
    /* otherwise n >=2. */  
    /* Find the largest element in the array a[1..n-1] */  
    maxval = max_array(a+1, n-1);  
    return max(a[0],maxval);  
}
```

Just to remind you, the linear version was done as follows: if **n = 0**, you return something like **-infinity**, a very large negative value. Now, if **n = 1**, you return **a[0]** which is the only element in the array. Otherwise, you have atleast 2 elements. And earlier what we did was, you call the sub problem, **a + 1**, so, the array which starts with the second element in the array. And now the sub problem has **n - 1** elements because you are considering a 0, the first element as a separate thing.

Now, what you want it to return was maximum of whatever was returned in the sub problem. So, let that be some **maxval**. And whichever is greater, **a[0]** and **maxval**, that is going to be the greatest element in the array. Now, we saw that the stack depth for this problem was **n** because size **n** problem is being reduced to a size **n - 1** problem. So, in each step we are reducing the size of the problem by 1, and increasing the stack depth by 1. So, in total that stack depth would be **n** because there will be about **n** calls or **n - 1** calls; however, you want to count.

(Refer Slide Time: 04:21)

Find the maximum of the numbers in an array.
(two-way recursive version)

- 1 Divide the array a into about two equal halves: $a[0 \dots n/2 - 1]$ and $a[n/2 \dots n-1]$.
- 2 Recursively find the maximum element in each half and return the larger of the two maxima.
- 3 Base cases: If n is 1 then return $a[0]$, if n is 0 return $-\infty$.

```
int max_arr(int a[], int n) {
    if (n==0) return -INFTY;
    if (n== 1) return a[0];
    return max(max_arr(a,n/2),
               max_arr(a+n/2, n-n/2));
}
```

Is this better than the previous recursive definition of max_array?
And how does it compare with the standard iterative definition?



Now, let us look at the two-way recursive version. So, here is the algorithm that we discussed; and let us just code this up. So, we will have `int max_array;` and then `int a,` which is the array containing n elements. And let us say that we have some constant $-\infty$, we have defined elsewhere in the program. Later we will see how to do this.

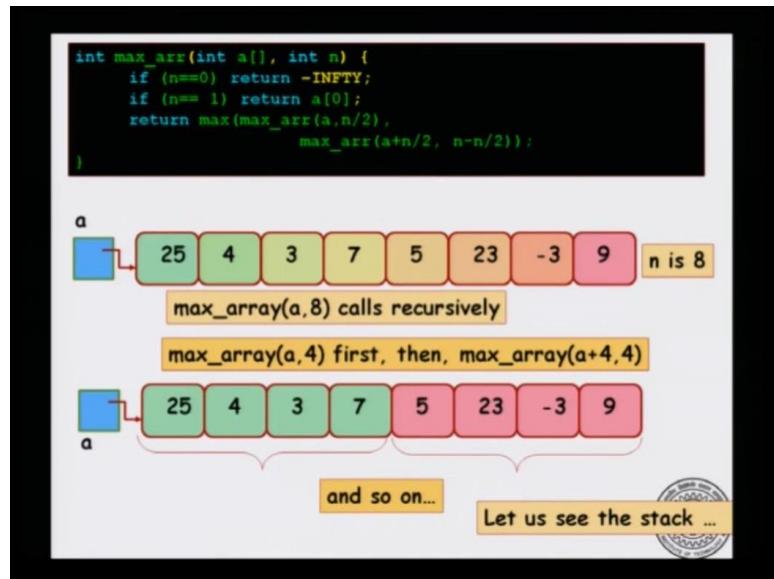
Let us say that if $n = 0$, you return $-\infty$, some large number, some large negative value. And if $n = 1$, you return the only value in the array. So, these are the base cases as before. The changes here; if you have at least 2 elements then you return maximum of the values returned by the 2 sub problems. What are the 2 sub problems? The first is the left half of the array which starts from, a , that is the, at the first location in the array, and contains $n/2$ elements.

Then we need to compute the maximum of the right half; how do we find the right half? So, we need to skip, $n/2$ elements, which went to the left half, to get to the first index in the right half. So, we do that by saying, $a + n/2$. If, a , is the address of the first location of the whole array then $a + 1/2$, is going to be the first address of the first location of the right half.

And how many elements does the right half contain? $n/2$ elements went to the left. Therefore, what we are left with is $n - n/2$. So, notice, how we call the left half starting from, a , and containing $n/2$ elements; and right half which is starting from, $a + n/2$, and containing $n - n/2$ elements.

Now, let us examine whether this is better than the previous recursive call, where we reduce the problem of size n to a problem of size $n / 2$. It was called linear recursion because we called one sub problem in order to solve the whole problem. Here, we have, we are roughly dividing it into halves and then calling 2 version, 2 sub problems, each of size about $n / 2$. Now, surprisingly, we will see that there is a huge improvement if you do this. And this is one of the most elementary tricks in computer science which is called divide and conquer; and here is a very simple example of that.

(Refer Slide Time: 07:00)



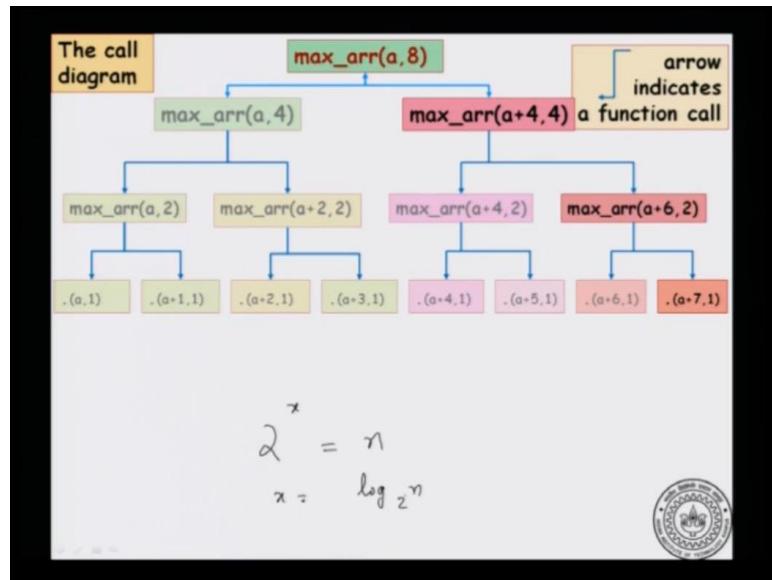
So, if you look at the concrete array that we had, and we call, `max_array(a, 8)`, because this contains 8 elements. Now, we say that it will recursively call 2 sub problems which is maximum, `max_array(a, 4)`. So, that will be the first 4 elements starting from, a 0. And then `max_array(a + 4, 4)`, which are the 4 elements starting from, a 4 which is the fifth element in the array.

Now, let us just look at the stack. Now, notice what types I repeatedly mentioned which is that in order to think about a recursive problem you just think about the formulation of the problem, and then what you have to convince yourself is if I solve the sub problems correctly then I will get the correct solution to the main problem. So, I will have, I will divide my work into 2 sub problems.

So, both of them will report their results back to me. Now, what I have to do is to figure out how do I put these 2 solutions together in order to solve the whole sub problem. So,

think about it in terms of the design of the algorithm, and not about the execution stack. But, we will show why this is a major improvement over the linear recursion version of the same solution by looking at the stack.

(Refer Slide Time: 08:33)



So, let us just look at the stack; `max_array(a, 8)`, calls, `max_array(a, 4)`. Now, the way function calls in C works, you will go to the second half of this problem which is, `(a + 4, 4)`, only after `max_array(a, 4)`, is completely done, right. So, let us now see how, `max(a, 4)`, will execute? It has 2 sub problems again. And let us look at the first sub problem which is, `max_array(a, 2)`, that itself has a sub problem, `max_array(a, 1)`. In order to abbreviate I will just put at dot there, but that dot is supposed to signify `max_arr`.

Now, once you have solved this, suppose this is the base case, now it contains only 1 element, so, the only element is the maximum; so, it returns that value to, `max_array(a, 2)`; that is one of the sub problems for, `max_array(a, 2)`. So, now, this, `max_array(a, 2)`, calls the second sub problem that it has, which is, `max_array(a + 1, 1)`. Again, it is a base case; it contains only 1 element; that single element is the greatest element in that.

So, you have 2 values now - one coming from the left and one coming from the right. And you just compare these 2 values, and that will be the greatest value in the first two elements of the array. So, once you do this, you return; and one you return, you get the value, `max_array(a, 2)`. So, suppose, all of that happens, and then you return to,

`max_array(a, 4)`. At this point, this function will call its second component which is, `max_array(a + 2, 2)`, and the recursion continues.

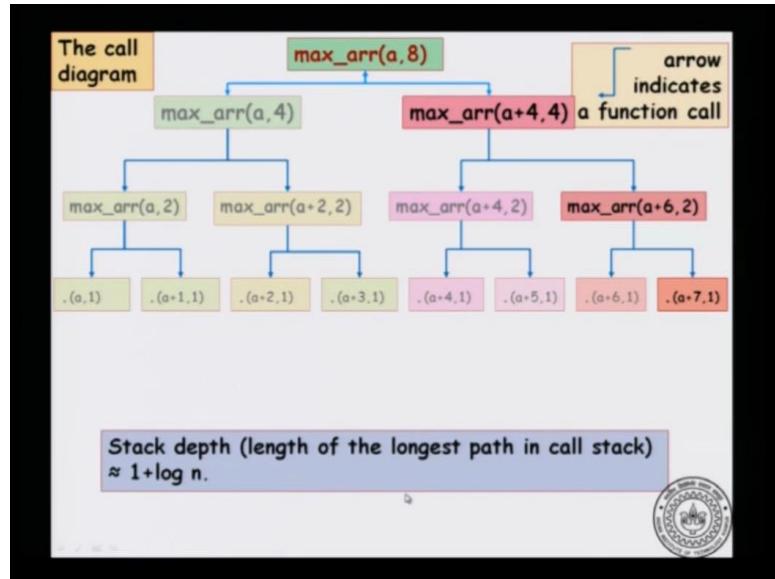
So, as soon as a function returns its stack will be erased; I am showing that by dimming out that particular function call, ok. And this proceeds. So, once this value is obtained you can return to, `max_array(a + 2, 2)`. Now, this function is finished because it has called both its sub problems. So, this will return. And this problem has returned, has finished with both its sub problems. So, you will, after this function is done you will eventually unwind all the way back up to the top.

And now, we are ready to call the second sub problem of, `max_array(a, 8)`, which is `max_array(a + 4, 4)`. And, you do it similarly. Now, one thing you can notice here, is that, at any point the active path, what are active on the stack, the functions which are not yet returned are the highlighted entries in the call tree, ok.

So, for example, at the very end the call stack contains 4 functions. Before you eventually return and compute the loss, compute the maximum of the whole array, the worst case depth of the stack is 4. And we had 8 elements, so, you would think that based on this experience that the depth of a stack is about n over 2. But, if you think more carefully about it what happens is that, at every sub problem, at every level, I am dividing the problem by 2.

So, the depth of the stack is the maximum length part in this tree. And at every step of the tree I am dividing the problem by 2. How many times do I have to divide in by 2 in order to reach 1, that will be the depth of the tree. Equivalently, you can think about, how many times do I have to double in order to reach n if I start from 1, that is the bottom of the way. So, if I start from 1 and I double every level, how many times do I have to double in order to reach n , that is the solution to the equation $2^x = n$. So, what I have to find is, how many times do I have to double? So, how many times do I have to multiply 2 with itself in order to reach n ? And you will see that the solution is $\log_2 n$. So, this is going to be the height of the call graph or the call tree.

(Refer Slide Ti me: 13:29)



So, the stack depth here is about, $1 + \log n$, that is approximately correct, which is a huge improvement over n . If you think of n as something like 1024 which is 2^{10} , we are saying that the stack depth is about 10. Notice that, in the linear case we would have a stack depth of about 1024, instead we are doing about 10. So, this is the huge improvement in the case of stack depth.

So, with a very simple idea which is instead of solving one sub problem of size $n - 1$, what if you split it into 2 halves, roughly about size $n/2$. You will see that you get a huge improvement in the stack depth. This is one of the simple ideas that we repeatedly use in computer science.

(Refer Slide Time: 14:25)

Standard arithmetic functions may be defined recursively but when implemented directly, can be very inefficient.

E.g.. Fibonacci numbers
 $F_0 = 1, F_1 = 1, F_n = F_{n-1} + F_{n-2}$

```
int fib(int n) {
    if (n==0 || n==1)
        return 1;
    else return fib(n-2) + fib(n-1);
}
```

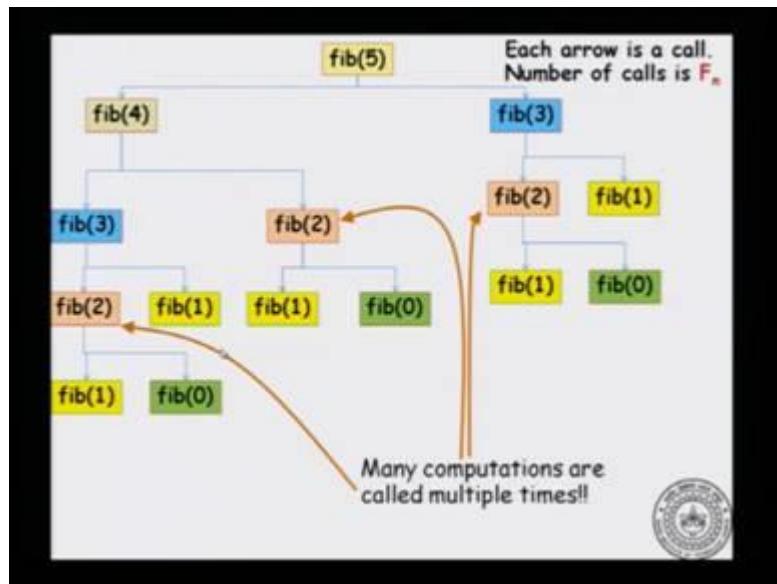
This is a very inefficient (but correct formulation). Let us trace the calls.



Now, there are standard arithmetic functions also which can be defined in terms of the two-way recursion. A very classic example is Fibonacci numbers. So, for example, they are defined as, $F_0 = 1, F_1 = 1$. And for $n \geq 2$, they are defined as $F_n = F_{n-1} + F_{n-2}$. So, if you code this out, so, a very simple function, `int fib(int n)`, if $n = 0$ or $n = 1$, you return 1. Otherwise, you return Fibonacci, so, $\text{fib}(n - 2) + \text{fib}(n - 1)$.

So, it is a very simple arithmetic sequence which is defined in terms of a two-way recursion. So, this is the very simple way to write it, but it is a very inefficient way to do it. So, we will see why it is inefficient in a moment.

(Refer Slide Time: 15:20)



If you just think of how you trace the function, in the case of a later, of a concrete Fibonacci number; let us say, we want to calculate the fifth Fibonacci number. Now, that depends on $\text{fib}(4)$ and $\text{fib}(3)$; $\text{fib}(4)$ depends on $\text{fib}(3)$ and $\text{fib}(2)$; $\text{fib}(3)$ depends on $\text{fib}(2)$ and $\text{fib}(1)$, and so on. So, this is the call graph that you will have, the call tree that you will have if you consider the calculation of Fibonacci 5.

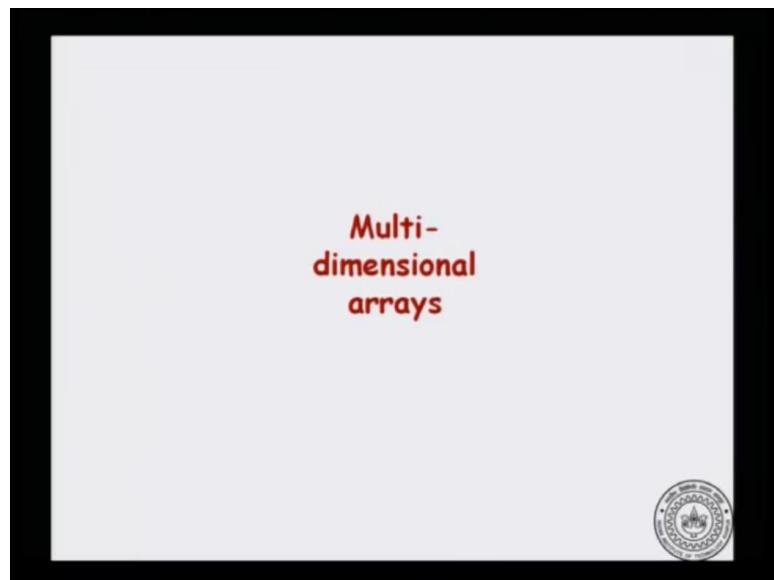
Now, what is the problem here? You will see that many computations are unnecessarily done multiple times. So, if you look at Fibonacci 2 in the call graph, it is evaluated multiple times. So, Fibonacci 2 is evaluated when $\text{fib}(3)$ is called. It is also called when $\text{fib}(4)$ is called. And $\text{fib}(3)$ is called in a different context. When you want to calculate $\text{fib}(5)$, even there $\text{fib}(2)$ is called. So, you will see that $\text{fib}(3)$ is called 2 times, $\text{fib}(2)$ is called 3 times, and $\text{fib}(1)$ is called 5 times, and so on.

So, we are unnecessarily repeating the work. And there is tricks in computer science to alleviate, to remove this kind of unnecessary work. But, that is strictly, it is not an idea that strictly falls into the concept of recursion, and is slightly outside the scope of this course. So, we will not cover this in this course, but I just want to point out that even though it is natural to consider this arithmetic sequence in terms of two-way recursion it may not be the most efficient way to do it.

Introduction to Programming in C
Department of Computer Science and Engineering

In this video will look at multi-dimensional arrays.

(Refer Slide Time: 00:03)



In particular, let us look at two dimensional arrays. Because, that will give you an idea how multi-dimensional arrays work. Initially, let us look at them as arrays and in a subsequent video will look at the connection between multi-dimensional arrays and pointers.

(Refer Slide Time: 00:22)

Multidimensional Arrays

Multidimensional arrays are defined like this:

double mat[5][6]; OR int mat[5][6]; OR float mat[5][6]; etc.

1. The definition states that mat is a 5 X 6 matrix of doubles. It has 5 rows, each row has 6 columns, each entry is of type double.

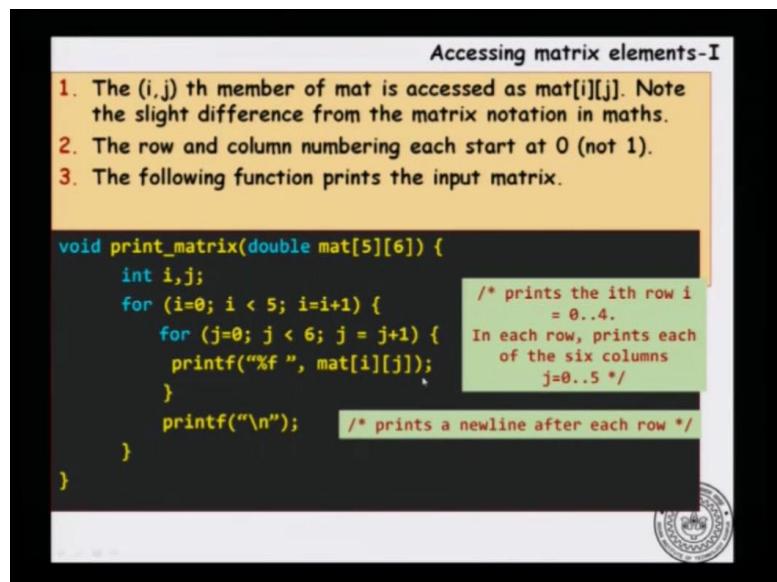
2. The type **double** in C stands for double precision floating point. If you are doing lot of floating point computations, use **double** instead of **float**.

2.1	1.0	-0.11	-0.87	31.5	11.4
-3.2	-2.5	1.678	4.5	0.001	1.89
7.889	3.333	0.667	1.1	1.0	-1.0
-4.56	-21.5	1.0e7	-1.0e-9	1.0e-15	-5.78
45.7	26.9	-0.001	1000.09	1.0e15	1.0

So, multidimensional arrays can be defined in the similar to the following, you can say `double mat[5][6]` or `int mat[5][6]` or `float mat[5][6]`, this is similar to the mathematical notation of multidimensional arrays are matrixes. So, let us look at the first example, we have that the definition states that mat is a 5×6 array of double entries. So, this means that mat has 5 rows, each row contains 6 entries and all the entries are of type double. Double is what is known as double precision floating point numbers.

And if you are doing a lot of floating point computations, then instead of float you could use double because, you might need a lot of precision in your computation. So, the matrix 2D array might look like this, this looks like a mathematical matrix of size 5×6 . So, it has 5 rows, rows 0 through row 4 and each row has 6 columns, column 0 through column 5.

(Refer Slide Time: 01:46)



Now, the i j th member of matrix is accessed as `mat[i][j]` this is slightly different from the mathematical notation. In mathematical notation you will write matrix and then a square bracket and then you will write i comma j followed by close bracket. So, this is different in C, you would write the indices separately in their own square brackets. Now, the row and the column numbering begin at 0, this is similar to one dimensional arrays we saw that one dimensional arrays start with index 0.

Let us look at a function which prints the input matrix. So, I have a function it returns void. So, it does not return anything it just performs an action, which is to print a matrix. Now, the function is called `print matrix`, it takes a double matrix `mat` of size 5×6 , 5 rows 6 columns each. I first declare `i` and `j`, `i` is suppose to iterate over the rows and `j` is suppose to iterate over the columns. Now, how do you iterate over the whole matrix.

Well, first you would take each row `i`. So, you need an `i` outer loop for that based on the variable `i`. I will go from 0 to 4, so the for loop goes from 0 until you hit 5. Now, for each row what do we have to do, we have to take the elements in the column. Now, the columns are numbered 0 through 5. So, for each `i` th through we have to take column 0, column 1, column 2, column 3, column 4 and column 5. So, all these entries and then you have to print that entry.

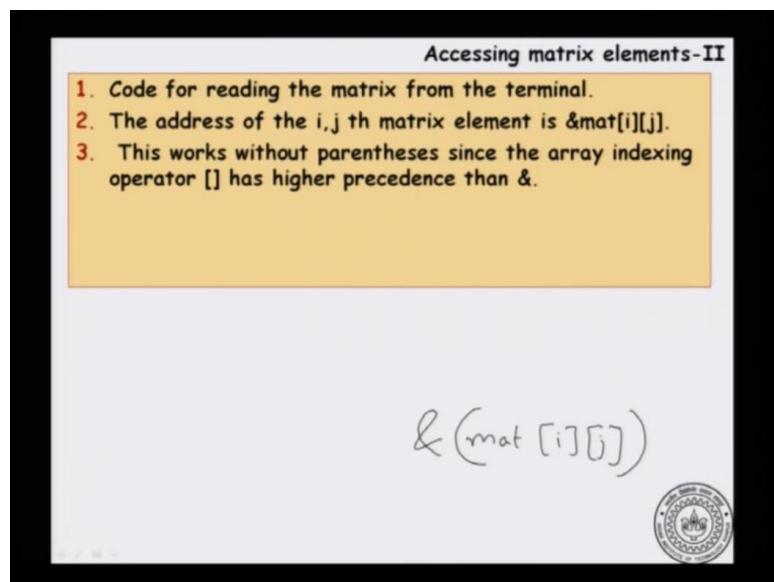
We have just mention that the i j th element in the matrix is accessed as `mat[i][j]`, `i` in square bracket and `j` in square bracket. Therefore, you will say `printf("%f", mat[i][j])`. So,

this will take the entry in the i th row, j th column. One more thing that is worth noting is that, even though you had a double matrix, you still print it as `%f` as though you were printing a float and the language will take care of printing the double precision.

So, here is the loop to print the columns of a row. Once you are finished with the row, you would print a new line. Because, then you can start at the beginning of the next line for the next row, so here is the loop. So, what the loop does is prints the i th row, row starting from 0 and ending in 4 and for each row print each of the 6 columns 0 through 5. Now, at the end of each row you would print a new line. So, here is the code to print a matrix, the lesson here is how to access the $i j$ th element. You would access it as `mat[i][j]`.

Now, the dual operation of printing is of course, reading in the input from the user, we have done it using `scanf`. So, let us try to use `scanf` to read in elements which are input by the user.

(Refer Slide Time: 05:29)



Now, one of the things with the `scanf` is that the argument to which variable we have to read it in, we usually give and x , if you have to read it into a particular variable x when we will say `scanf` whatever format it is and then say and x , which says the address of x . Similarly, I could guess that in order to read to the $i j$ th element of a matrix, I would need `&mat[i][j]` and that is actually correct, you do not need a parentheses here to right.

So, you do not need to right `&(mat[i][j])` with `mat[i][j]` in square bracket. Because, the address operator has lower precedence then the array indexing operator. So, when you see an expression like this, the parentheses will be in such a way that `mat[i][j]` is evaluated first and then the address is taken. Because, it is a lower precedence operator. So, this is similar to a one dimensional array that you have already seen, you would have read it using `&` of a `i`. So, we have the two dimensional analog of that.

(Refer Slide Time: 06:44)

Accessing matrix elements-II

1. Code for reading the matrix from the terminal.
2. The address of the i, j th matrix element is `&mat[i][j]`.
3. This works without parentheses since the array indexing operator `[]` has higher precedence than `&`.

```
void read_matrix(double mat[5][6]) {
    int i, j;
    for (i=0; i < 5; i=i+1) {
        for (j=0; j < 6; j = j+1) {
            scanf("%f", &mat[i][j]);
        }
    }
}
```

/ reads the i th row $i = 0..4$.
In each row, reads each of the six columns $j=0..5$ */*

scanf with %f option will skip over whitespace.

0	1	2	3	4	10
5	6	7	8	9	11
:					

So, let us look at the code and the code looks exactly as the print routine, except that we are now scanning a number. So, you have an outer loop which will go through all the rows and then in inner loop which will go through the columns of the i th row and how do you scan. You says `scanf("%f", &mat[i][j])`. So, remember if it was just a double variable instead of an array, you would have just said `&` of the variable name. Similarly, we have `&mat[i][j]`.

Again, note that even though we have a double array, you read it in exactly as though it were a float array, using `%f` format. So, read in the i th row and i th row goes from 0 through 4 and for each row read the j th column, column goes from 0 to 5. Now, the way this `scanf` works, the `scanf` with `%f` option will skip over the white space and it will skip over any white space.

So, in effect what it means is that, if I had to enter a matrix of size 5×6 , I can enter it in multiple ways, I can enter it in the most natural way which is 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

and so on. So, 5 rows each row has is entered in a line and each line has 6 entries, so let us call this may be 10 and 11. So, each row has 6 entries and there are 5 rows, this is a most natural way to enter it.

But, as far as scanf is concerned any white space will be skip. So, instead I could just enter one number in one line. So, I could enter it one number per line and it put be read exactly in the same manner, that is a property of scanf.

(Refer Slide Time: 08:53)

Accessing matrix elements-II

1. Code for reading the matrix from the terminal.
2. The address of the i, j th matrix element is $\&\text{mat}[i][j]$.
3. This works without parentheses since the array indexing operator $[]$ has higher precedence than $\&$.

```
void read_matrix(double mat[5][6]) {  
    int i,j;  
    for (i=0; i < 5; i=i+1) {  
        for (j=0; j < 6; j = j+1) {  
            scanf("%f ", &mat[i][j]);  
        }  
    }  
}  
/* reads the ith row i  
   = 0..4.  
   In each row, reads  
   each of the six  
   columns j=0..5 */  
scanf with %f option will  
skip over whitespace.  
So it really doesn't matter whether the entire input is given  
in 5 rows of 6 doubles in a row or all 30 doubles in a single  
line, etc..
```

So, it really does not matter whether the entire input is given in 5 rows of 6 doubles or just 30 doubles each number in a single row by itself. So, that is you both of them are fine. We have seen how to print an array. We have seen how to read elements into an array. Now, let see how to initialize a multi-dimensional array.

(Refer Slide Time: 09:17)

The slide has a title 'Initializing 2 dimensional arrays' at the top left. Below it, a text box says 'We want $a[4][3]$ to be this 4×3 int matrix.' To its right is a 4x3 grid of numbers: 1 2 3, 4 5 6, 7 8 9, 0 1 2. Next to the grid, the word 'Initialize' is written in red with an arrow pointing to the word 'as'. To the right of the grid is a code snippet in C-like syntax:

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

 At the bottom of the slide is a handwritten note:

```
int b [3] = {0,1,2};
```

 A circular logo is visible in the bottom right corner.

So, we want to initialize let us a 4 by 3 array in the following way, it should be 1, 2, 3, 4, 5, 6, 7, 8, 9 and 0, 1, 2, let say this is the array that I want to enter. Now, we have seen initialization of one dimensional arrays, if I let say `int b[3]` how did we initialized, we could initialized it us 0, 1, 2. So, we summary of this is that, it is a list of numbers separated by commas and the list is enclosed in curly braces, this is the case for a one dimensional array.

So, it is natural to generalized the notation in the following way, if I have to initialize a 4 by 3 array, I can just say curly brace. And so here is a list of elements and each element is basically a row. So, what is a number here will be a row? So, it will be a list of rows and each row being an somewhat like an array, each row will be given by a list. So, the array initialization on the right hand side is exactly the array that is shown here. So, it will come out to 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2. So, the notation is consistent and it is a generalization of the one dimensional array notation.

(Refer Slide Time: 10:51)

The slide has a title 'Initializing 2 dimensional arrays' at the top left. It contains several text boxes and code snippets:

- A blue box on the left says: 'We want $a[4][3]$ to be this 4×3 int matrix.'
- A yellow box next to it shows a 4x3 grid of numbers:

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

 with the text 'Initialize as' written above it.
- A blue box on the right contains C code:

```
int a[][] = { {1,2,3}, {4,5,6}, {7,8,9}, {0,1,2} };
```
- An orange box labeled 'Initialization rules:' lists four points:
 1. Most important: values are given row-wise, first row, then second row, so on.
 2. Number of columns must be specified.
 3. Values in each row is enclosed in braces {}.
 4. Number of values in a row may be less than the number of columns specified. Remaining col values set to 0 (or 0.0 for double, '\0' for char, etc.)
- A blue box at the bottom left shows C code:

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

 with the text 'gives this matrix for $a[3][3]$ ' written below it.
- A yellow box on the right shows a 3x3 grid of numbers:

1	0	0
2	3	0
3	4	5

.
- A small circular logo of a university or institution is in the bottom right corner.

So, there are some initialization rules, similar to what we are seen for one dimensional arrays, values are given row wise. The row number 0 is the first entry, number of columns needs to be specified, we need to know how many columns there are? Now, value of each row is enclosed in {} and the number of values in a row, may be less than the total number of columns, this is allowed.

This was similar to how we saw that, even though you had declared the size of an array, you could give one dimensional array, you could be less than that number of values as the initial values. The remaining values will just be 0, same case occurs in the multidimensional array. So, let us watch an example, if I have an array a number of rows unspecified, number of columns 3.

But, each row let say I have 3 rows, each row does not have exactly 3 elements, one the row 0 has just 1 element, row 1 has only 2 elements and so on, it will be initialized as 1 0 0 because in row 0 I have given only 1 element. So, that will be the first and the remaining will be 0, 2 3. So, I am short of 1 element that would be 0, 3 4 5 I have 3 columns and I have given 3 values. So, it will be initialized as 3 4 5. So, the initialization on the left hand side results in the matrix on the right hand side, here is all initialization words.

(Refer Slide Time: 12:40)

Accessing matrix elements-II

```
void read_matrix(double mat[5][6]) {
    int i,j;
    for (i=0; i < 5; i=i+1) {
        for (j=0; j < 6; j = j+1) {
            scanf("%f ", &mat[i][j]);
        }
    }
}
```

/* reads the ith row i = 0..4.
In each row, reads each of the six columns j=0..5 */

Could I change the formal parameter to mat[6][5]? Would it mean the same? Or mat[10][3]? Question?

That would not be correct. It would change the way elements of mat are addressed. Let us see some examples. Answer

Now, let us look at the access mechanism in somewhat greater detail. So, let us ask the following question, instead of matrix **5×6** I have return and function to read a matrix of size **5×6**, can I give a 6 by 5 matrix? So, this is a matrix of 5 rows, 6 columns each instead can I give a matrix of 6 rows 5 columns each, the total number of elements is till 30 would it be the same or would it be even a matrix of **[10][3]**, 10 rows 3 columns each all of these have 30 elements.

Now, as far as C is concerned are all these the same, the answer is that no, it is not correct neither it should be. But, we will say that the answer depends on the way the array elements are accessed. So, we will see this in greater detail.

(Refer Slide Time: 13:40)

Passing two dimensional arrays as parameters

Write a program that takes a two dimensional array of type double [5][6] and prints the sum of entries in each row.

```
void marginals(double mat[5][6])
{
    int i,j; int rowsum;
    for(i=0; i < 5; i=i+1) {
        rowsum = 0.0;
        for (j=0; j < 6; j = j+1) {
            rowsum = rowsum+mat[i][j];
        }
        printf("%f ", rowsum);
    }
}
```

Question?

But suppose we have only read the first 3 rows out of the 5 rows of mat. And we would like to find the marginal sum of the first 3 rows.

Answer:

That's easy, we can take an additional parameter **nrows** and run the loop for **i=0..2 instead of 0..5**.

So, in order to motivate there let us introduce the problem of passing an array to a function and let us look at the issue in greater detail. Suppose, I want to take two dimensional array of type double [5][6] and print the sum of entries in each row. So, this is similar to a matrix program, that we have seen much, much before given a 2D matrix, for each row you have to find the sum of elements in each row and just print it out.

So, in mathematics this is often called marginal's. So, let us just compute the marginal's, we have a function void marginal's, it takes matrix [5][6], it has int i j, i is over the rows, j is over the columns and I also have a row sum variable to keep track of the sum of a row. So, what do I do, I have an outer loop which goes through all the rows, for each row I initialize the sum to 0. Now, for each row I have to some all the elements in the i th row. So, I have to sum all the elements in the columns j 0 through 5 of matrix i j.

So, I will go through the elements and add them into the row sum. Once I am done with the last column of row i, I have the row sum for row i and I will print that. So, this printf is happening in the loop for row height. Now, let us look at a slight modification, we say that instead of printing 5 rows I currently have only 3 rows of entries available. So, can you print me the row sum of the first 3 rows, instead of all the 5 rows.

Now, this is very simple let us just modify the function a little bit, it takes an additional parameter saying, how many of the initial rows do you want me to sum? So, that is an additional parameter, let us call it n rows.

(Refer Slide Time: 15:54)

The slightly generalized program would be:

```
void marginals(double mat[5][6], int nrows)
{
    int i,j; int rowsum;
    for (i=0; i < nrows; i=i+1) {
        rowsum = 0.0;
        for (j=0; j < 6; j = j+1) {
            rowsum = rowsum+mat[i][j];
        }
        printf("%f ", rowsum);
    }
}
```

In parameter double mat[5][6], C completely ignores the number of rows 5. It is only interested in the number of cols: 6.

We declared mat to be of type double [5][6]. Does this mean that nrows should be ≤ 5 ? We are not checking for it!

Let's see an example...



So, here are the number of rows for which I have to take this sum. And that function is a very small modification of the function that we have already seen. The difference is that, we now take n rows which is like, how many rows do we have to add and then for $i = 0$, earlier I would go from $i = 0$ to 5. Because, the matrix had 5 rows, but in now I will just say I will go up to n rows and the logic is the same as before, nothing else changes.

So, his strange things he completely ignores the number of columns, for as far as the c languages concern, if you have a 2D array, the number of columns is crucially it has to be specified. But, the number of rows is not really important. So, c completely ignores the 5 part, the number of rows. Now, this means that we could pass less than 5 rows into the same function. Since, we are not checking for example, that encloses ≤ 5 .

(Refer Slide Time: 17:11)

The following program is exactly identical to the previous one.

```
void marginals(double mat[][6], int nrows)
{
    int i,j; int rowsum;
    for (i=0; i < nrows; i=i+1) {
        rowsum = 0.0;
        for (j=0; j < 6; j = j+1) {
            rowsum = rowsum+mat[i][j];
        }
        printf("%f ", rowsum);
    }
}
```

1. Why? because C does not care about the number of rows, only the number of cols.
2. And why is that? We'll have to understand 2-dim array addressing.

This means that the above program works with a k X 6 matrix where k could be passed for nrows.

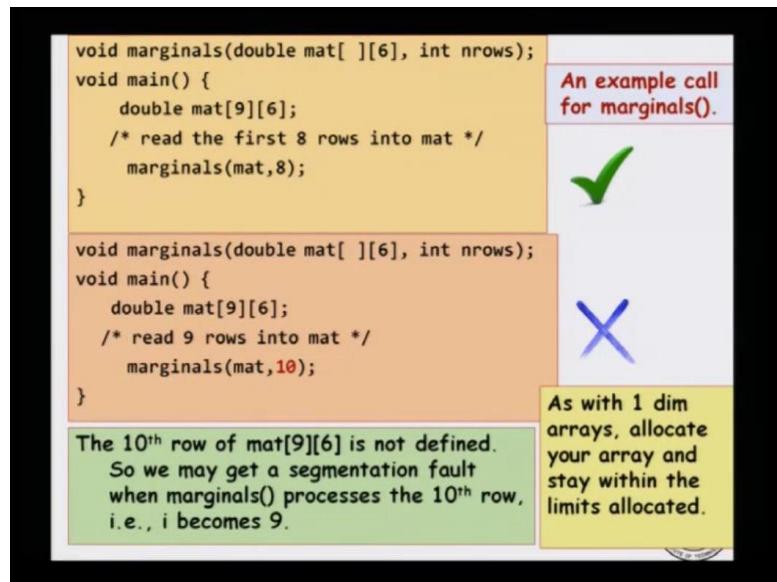
Example...

So, let see an example here is the completely surprising example, that this code is the same as before, though only difference is highlighted in read, that I have now omitted what is the number of rows? Please relate this back to codes, that we used to right for arrays. Earlier, I said that for an array, you do not need to specify the number of elements in the array, when you write a function taking an array as parameter I could just say int arr and then empty pair of [] with no size in between.

So, we have a similar phenomenon for 2D arrays, except you are not allowed to omit both rows and columns, you have to specify the number of columns. But, you have the flexibility that you are allow to omit the number of rows. So, the number of rows is not important, you could omit it and just given empty pair of brackets and the code will work as before.

So, this means that the above program actually works for any $k \times 6$ matrix, where k could be the number of rows. And this is because c does not care about the number of rows, only about the number of columns and y is this asymmetry, why is said that it case about the number of rows, but not the number of columns, will see this using the two dimensional array addressing.

(Refer Slide Time: 18:43)



Let say that I have return code for computing marginal's and it takes these parameters double mats empty pair. So, the number of rows is un specify, the number of columns is 6 and then it takes an additional parameter n rows, which says how many rows do should I add. And then I am calling this function, suppose I have define the function elsewhere and I am calling this function from name. So, I declare a matrix 9 by 6 and then I will call marginal's on just the first 8 rows not the 9th row.

So, I passes subset of the rows, this is 5. Because, I have declared as matrix of size 9 by 8, but I am passing only 8 rows to marginal's and that is fine, I can passes subset of the rows. What is definitely not fine is, suppose you declare a matrix of size 9 by 6 and say that I want you to find the marginal's of the first 10 rows, then this is unsafe. Because, it is true that the marginal's function does not really care about the number of rows.

So, it will work for any $k \times 6$ matrix. But, you cannot hope to pass arbitrary junk values to that matrix. For example, you have just declared a [9][6] matrix. Now, the 10th row of the matrix is basically invalid. So, if you pass it you could expect your code to receive a segmentation evaluation, when your run the code. So, when it processes the 10th row what it, it was basically cross the limits of the array. So, the code may have a segmentation evaluation.

So, note the difference between saying that it could marginal's could work with arbitrary $k \times 6$ matrices, it is till true that if you pass junk values to the matrix, your code will

crash. If your code is a valid matrix, then you can pass an arbitrary number of rows inside the matrix. So, the summary is that as with one dimensional arrays, you should allocate your array and stay within the limits allocated, within those limits the number of rows does not matter. But, it does not mean that you can over suit the limit and hope that your code will work, it may not and it can crush.

Introduction to Programming in C

Department of Computer Science and Engineering

In this video, we will look at the relation between Multi-dimensional Arrays and Pointers.

(Refer Slide Time: 00:03)



And this is by far one of the trickiest topics in the entire course.

(Refer Slide Time: 00:18)

The slide contains several text boxes and a code block:

- A yellow box: "We will now discuss how address arithmetic (pointer arithmetic) works for 2-dim matrices.
This just might be a tiny bit complicated!"
- A green box: "BUT you can do a lot of matrix computations by just remembering what we discussed so far."
- A yellow box: "When you declare a 2-dim array as a formal parameter to a function, you must specify the number of columns, but need not specify the number of rows."
- A code block in a light blue box:

```
void make_identity10( double m[][10]) {  
    int i,j;  
    for (i=0; i < 10; i = i+1) {  
        for (j=0; j < 10, j = j+1) {  
            if (i==j) {  
                m[i][j] = 1.0;  
            }else {  
                m[i][j] = 0.0; }  
        }  
    }  
}
```
- A purple box on the right: "make_identity10 () assigns to m[][10] the identity matrix in its first 10 rows. It assumes m is at least 10X10."

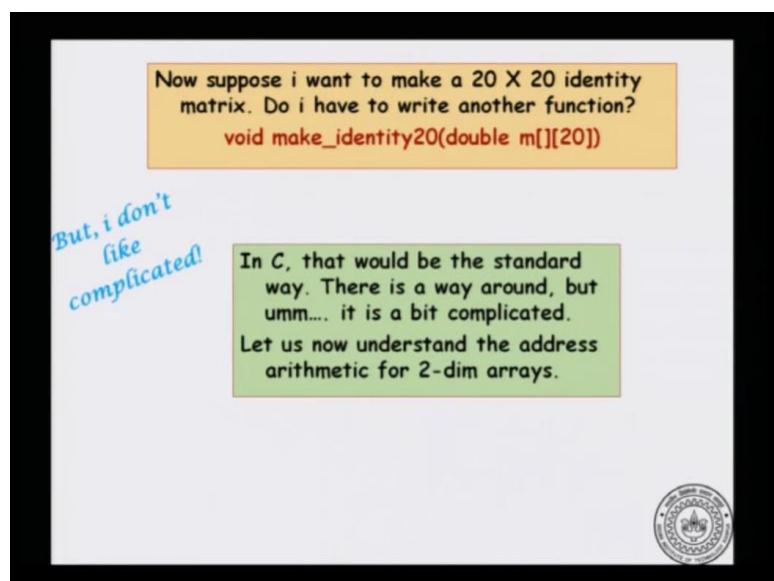
And you can code multidimensional arrays without actually understanding the exact relation between multidimensional arrays and pointers. But, understanding this gives you a better grasp of how C treats multidimensional arrays. So, we will now discuss how pointer arithmetic works with two dimensional matrixes. Because, as soon as we had discussed one dimensional matrix, the next thing we did was we discussed the relation between pointers and 1D arrays.

So, let us try to see what is the relation between pointers and 2D arrays. Now, this is more complicated than it looks at first sight. And you can do a lot of matrix computations by not understanding this. Except that, understanding this gives you a better grasp for what is happening. We have seen that, when you declare a 2D array as a parameter to a function, then you should specify the number of columns, but not the number of rows.

So, let us look at a function which makes an identity matrix, an identity matrix is a matrix that has one along its diagonal and zero everywhere else. So, we have `void make_identity10(double m[][10])`. Since, identity matrixes are square matrixes, this essentially says that the code will work for a 10×10 matrixes.

Then, I have a for loop going from $i = 0$ to 10 and a for loop going from, for the columns going from $j = 0$ to 10. And the code just says that, if I am at a diagonal element that is $i = j$, then $m[i][j]$ is 1 for all other elements, $m[i][j]$ is 0. So, this creates a matrix of size 10×10 .

(Refer Slide Time: 02:23)



Now, this is a very strange code. Because, it is a function that essentially makes exactly one matrix. It would have been nice, if I would have a function that can create arbitrary size identity matrixes. For example, if I wanted a 20×20 matrix, it looks like, I have to write another function `make_identity20(double m[][20])`, the rows are unspecified, the number of columns is 20.

This is the standard way to do it. But, there is a slightly more complicated way to actually accomplish a function which can take an arbitrary size. So, let us see how these things can be done by understanding, how pointer arithmetic works with 2D arrays.

(Refer Slide Time: 03:15)

Addressing the $[i][j]$ element in a 2-d array

Consider the definition `int mat[3][5];`

We view it as a 3×5 matrix of int, something like this.

mat	0	1	2	3	4	the matrix view
	5	6	7	8	9	
	10	11	12	13	14	

C views mat as a long linear array of size 15 in "row-major" form, row 1 then row 2 then row 3...

mat	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

the row-major view (internal to C)

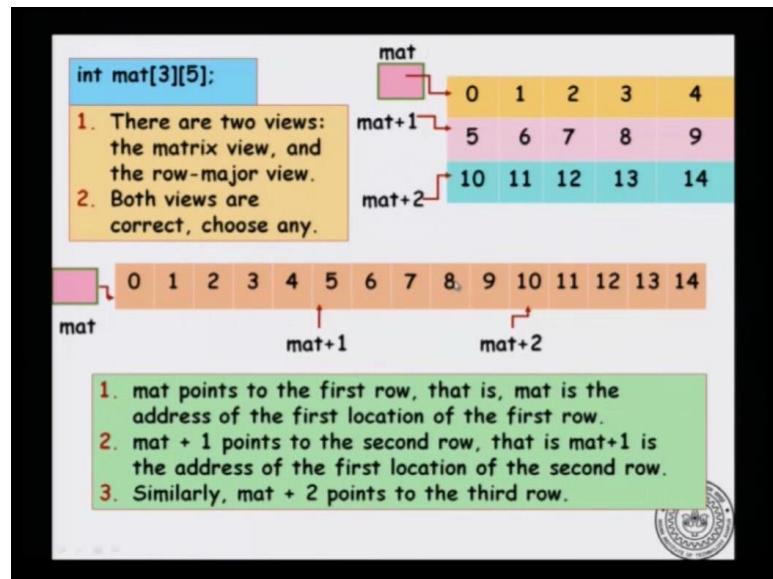
BUT the difference between mat as a 2d array with 5 columns and a 1d array of integers is in the pointer arithmetic.

So, let us go back to, how do I address the i j th element in a 2D array? Now, we can view it as a 3×5 matrix of integers, something like this. So, it may be an array $0 \ 1 \ 2 \ 3 \ 4$, that is row 0 and $5 \ 6 \ 7 \ 8 \ 9$, that is row 1, $10 \ 11 \ 12 \ 13 \ 14$, that is row 2. So, this is the matrix view which is 3 rows, each with 5 columns, this is the standard view. But, internally C views this as a long linear array of size 15, in what is known as the row major form.

So, let us just look at what it is? Internally, C looks at the array in the following form. It is basically C 0 through 14, lead out in a single row. So, this is the row major view, it is called row major, because first all elements of row 0 will be lead out, then all elements of row 1 will be lead out. And finally, all invents of the lost row will be lead out. But, it is lead out as a linear way.

Now, the natural question to ask is, in that case is a 2D array really at the heart of just a 1D array. So, the difference between a 2D array seen in the row major view point and an actual one dimensional array will come in the pointer arithmetic.

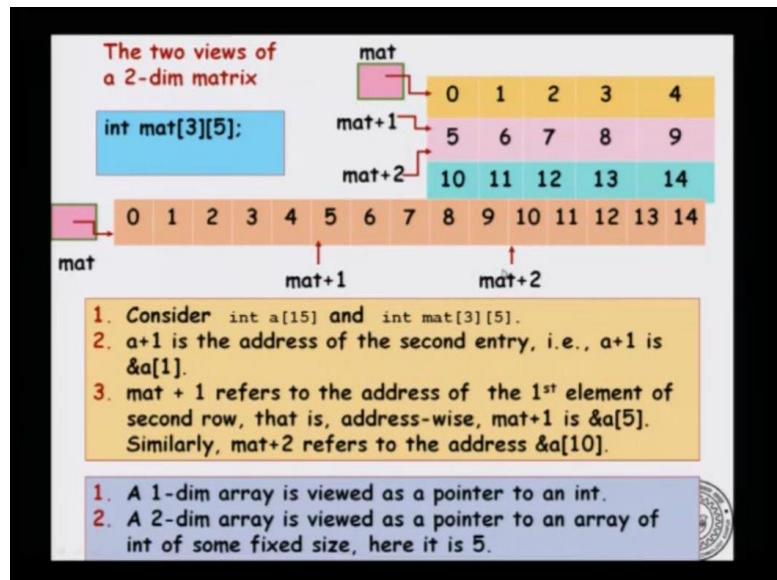
(Refer Slide Time: 04:52)



So, as I just mentioned there are two views, the matrix view and the row major view and both views are correct. So, if I have the matrix view, mat is a pointer to the first row. So, **mat + 1** will be a pointer to the second row and **mat + 2** will be a pointer to the third row. So, row number 3 or row indexed with 2, in the row major view point, here is the difference mat points to the first row and **mat + 1** should point to the second row.

So, we cannot say that mat is pointing to the first element here and **mat + 1** should point to therefore 1. Now, that is not what happens? It has to be consistent with the matrix view. So, the pointer arithmetic **mat + 1** should point to the same element, regardless of whether you are looking at it using the matrix view point or whether you are looking at it using the row major view point. So, **mat + 1** will still point to 5 and **mat + 2** will still point to 10. So, these two view points are consistent.

(Refer Slide Time: 06:11)



Now, here is the difference with one dimensional arrays. So, we have just repeated the viewpoints here, the matrix view point and the row major view point. Now, had `mat` actually being a one dimensional array, `mat` would point to the first element in the array. Therefore, `mat + 1` which should point to the second element in the array. So, that is not what happens, it is actually the row major representation of a 2D array and `mat + 1` should skip exactly 5 elements, because that is the size of the column.

So, `mat + 1` should skip 5 elements and go to the element `mat[1][0]`. So, here is why you need to know the number of columns? Because, in the row major view point I have to implement `mat + 1`. So, I have to say how many elements should I skip, in order to get to the first element of the second row? And that number is exactly the number of columns in the array. So, the number of columns in the array is 5. So, to get to `mat + 1` from `mat`, I would skip 5 elements.

Similarly, to get to `mat + 2` from `mat + 1`, I would skip exactly 5 elements. So, this is the reason why the number of columns is an important information. Because, that tells me in the row major representation, how many elements do I have to skip in order to get to the correct entry in this second row or third row? So, here is the pointer arithmetic for the row major representation. And notice that, this is considerably different from the pointer arithmetic for a 1D array, in a 1D array, `array + 1` will go here, the first element of the array.

(Refer Slide Time: 08:12)

A 2-dim array is viewed as a pointer to an array of int of some fixed size, here it is 5.

Can you guess what the type of mat would be?
Which one below looks most right?

int *mat;
int **mat;
int *mat[5];
int (*mat)[5];

Recall that array indexing operator [] has higher precedence than *, the dereferencing operator

int * (mat [5]);

float arr[5];

Now, can you try to guess what will be the type of mat? So, here are four candidates and let us go through them to see, what is the most likely candidate? And we will see this in a greater detail, `int *mat`, mat is a pointer to int. Now, we have seen that, that is approximately an array of integers and that is definitely incorrect. Because, this is supposed to be a 2D array, not an array of integers. Pointer to pointer to mat, we have seen that so far and that looks like a likely candidate.

So, what about the third and the fourth? The third and the fourth looks confusingly similar. What do they mean? So, here is a hint, the array indexing operator `[]` has higher precedence than `*`, the dereferencing operator. So, in this case the first says that, so what is this mean? The first declaration is actually `int *mat[5]` and the second declaration is `int (*mat)[5]`. So, what does this say?

So, let us compare with the standard declaration like `float arr[5]`. This means, that array arr is an array of size 5, each entry of type float. Similarly, this means the matrix mat is an array of size 5, each entry being a pointer to integer. So, it will be some matrix like this, it has five elements and each of them is a pointer. So, here is the view point for declaration 3. Now, what about declaration 4?

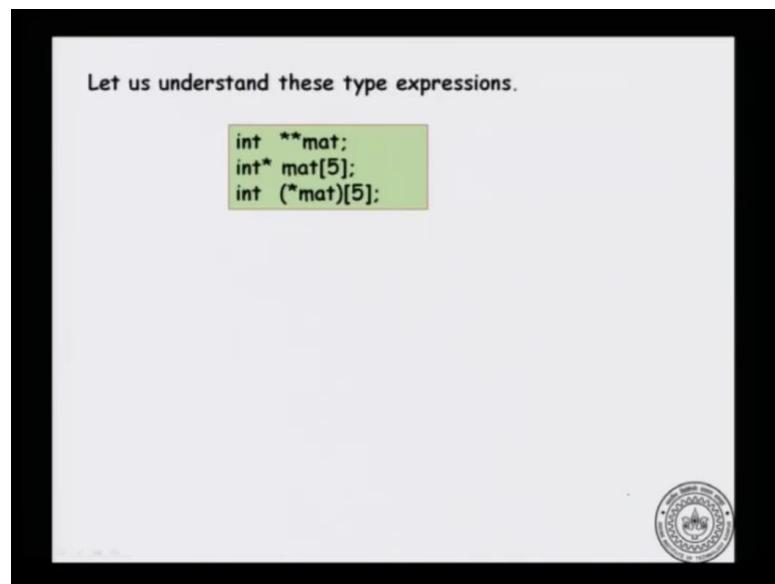
(Refer Slide Time: 10:26)

The diagram shows a horizontal array of 15 integers from 0 to 14. A pink box labeled 'mat' contains a pointer arrow pointing to the first element (0). Below the array, three pointers are shown: 'mat' points to index 0, 'mat+1' points to index 5, and 'mat+2' points to index 10. A blue box contains the text: 'A 2-dim array is viewed as a pointer to an array of int of some fixed size, here it is 5.' A pink box asks: 'Can you guess what the type of mat would be? Which one below looks most right?' Below are four declarations: `int *mat;`, `int **mat;`, `int *mat[5];`, and `int (*mat)[5];`. The fourth declaration is marked with a green checkmark. To the right, a yellow box states: 'Recall that array indexing operator [] has higher precedence than *, the dereferencing operator'. Below the text are two diagrams: one showing a pointer to a function with parameters, and another showing a pointer to an array of 5 integers.

So, there let us see this, so let us compare it with a standard declaration like, let us take a standard declaration like `int arr[5]`. Again, this says that arr is an integer array of size 5. So, it contains 5 elements, each of type int correspondingly, what this means is that `*mat` is an integer array. So, here is an integer array containing 5 elements, these are integers. Now, this means that if we deference mat. So, mat is a pointer to an array of size 5 and this is exactly the actual representation of a two dimensional array.

So, notice the difference between these two representations, the first says that mat 5, mat is an array of 5 entries and each entries is a pointer to an int, so it look like this. So, it is an array of 5 pointers to int. The lost declaration says that, `*mat` is an array of int of size 5. So, mat is a pointer to an array of integers of size 5. So, here is the difference and we will argue that the fourth definition is essentially what we want and we will see this in a greater detail.

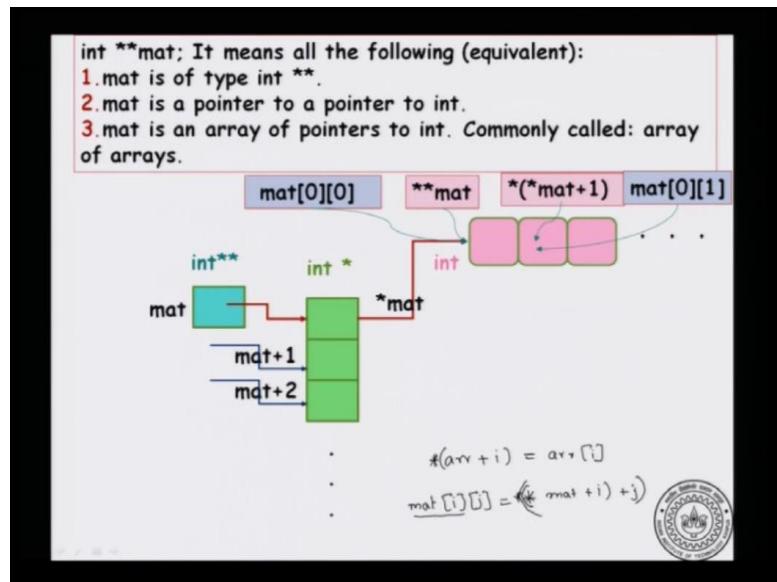
(Refer Slide Time: 12:24)



So, let us understand these type expressions in greater detail and we will see this in the further video also. We particularly pick on one representation here. So, we have argued that the 2D array is similar to the last declaration here, I eliminated the most obviously wrong declaration, which is in `*mat`, that is basically a one dimensional array. So, I have just eliminated that, we will examine all the others.

What I have just said is that a 2D array is similar to the last declaration. But, even the previous two declarations do make sense. And there may be situations, where you need to use such variables. Let us examine them in greater detail.

(Refer Slide Time: 13:05)



So, let us look at the first one which is `int **mat` and it means all of the following equivalent ways. So, all of these are an equivalent ways of looking at the same thing. You could say that a matrix of type `int **` or you could say that matrix is a pointer to a pointer to an int. Since, arrays are pointers approximately, you could also say that mat is an array of pointers to int and this is also commonly called array of arrays.

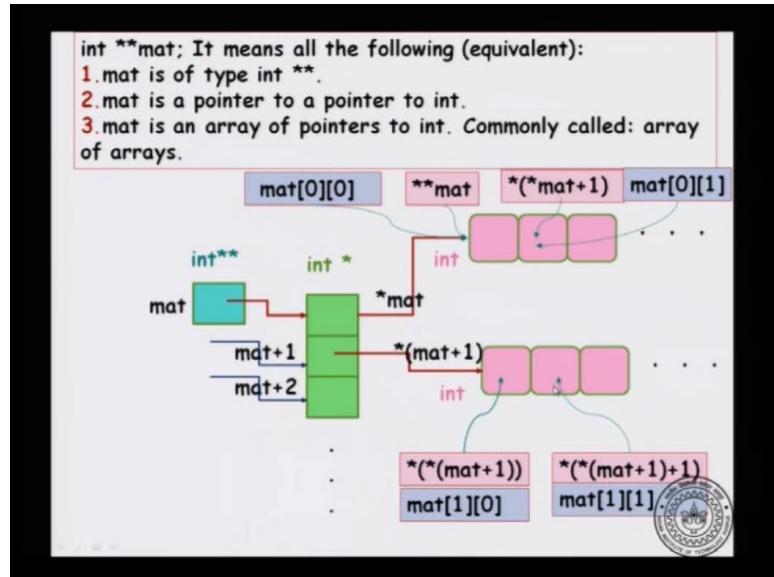
So, you have `mat int **`. Now, this is a pointer to an array of integers. Now, every pointer to an int is essentially a pointer to an array, you can look at it like that. So, you can say that mat will point to an array of pointers and this array of pointers, each of them may point to a different array. So, you dereference mat once, you will get a pointer to an integer and again you deference once more, you will get the actual array.

So, what happens when I do `mat + 1`? It will go to the second entry in the array of integers. Now, that may be a different array all together. So, `mat[0][0]` is similar to `**mat`, this is just a way, address arithmetic works. And both of them are addresses, both of them are pointing to this location, both of them mean the content of this location. Similarly, `*(mat + 1) = mat[0][1]`. So, in the case of one dimensional arrays we have just mentioned the equation that `arr[i]` is the same as `*(arr + i)`.

And what we are saying here essentially is that, `mat[i][j]` is the same rule applied twice. So, I could say `mat[i] = *(mat + i)`. So, that will give me an array and then, I need the jth

element of that. So, you can again do $\ast(\ast(\text{mat} + i) + j)$. So, these are two ways of looking at this array.

(Refer Slide Time: 15:50)



So, `mat + 1` will be the next element in the pointer to integers and it is the same as and when you deference it, you will get another array. So, in order to get the first element of this array, I could say `mat[1][0]` or using the pointer notation, I have `*(* mat + 1)`, these are the same and similarly for other elements of the array. So, one of the advantages of this kind of `int **mat` is that, I have freedom in both dimensions.

You can see these as the rows of a matrix and these as the columns of a matrix. If you see that, then you can see that I have a lot of freedom here, first of all the number of rows is not limited. Because, it is just `int **mat`, I could have any number of rows here. Now, another main advantage and the reason why this is somewhat popular is that, the length of row 0 need not be the same as the length of row 1, these are just pointers to integers.

So, the first pointer to integer may be pointing to a row of size 2, the second pointer may be pointing to a row size 3 and so on. So, the row lengths need not be the same. So, think of an array where row 0 is two elements long and row 1 has three elements in the row and so on. So, if you have extremely ragged arrays, then `int **mat` is a nice representation to take.

Introduction to Programming in C

Department of Computer Science and Engineering

In this video, we will look at one of the other expressions.

(Refer Slide Time: 00:03)

Let us understand these type expressions.

```
int **mat;
int* mat[5];
int (*mat)[5];
```

→


```
int arr[5];
int (*mat)[5]
```

↓

mat is a pointer to an array of ints of size 5.



In particular, we will look at the third one, which is `int (*mat)[5]`. So, if I had written `int arr[5]`, this means that array is an integer array of size 5. So, similarly I can read `int (*mat)[5]` as star mat is an integer array of size 5. So, in other words mat is a pointer to an array of size 5, array of int of size 5. We can look at in this way and let us see, what this really means.

(Refer Slide Time: 00:54)

`int (*mat)[5];` mat is a pointer to an array of size 5. How did we read this? mat is an address type. If you dereference mat, i.e., take `*mat`, it is of type array of size 5.

mat points to the 1st row of 5 ints. `*mat` is an array of size 5.

`mat[0][0]` is same as `(*mat)[0]` is same as `*(*mat)` or `**mat`

`mat + 1`
points to the
2nd row of 5
ints.
`*(mat + 1)` is
an array of
size 5.

`(int [5]) *`

mat

`int [5]`

mat + 1

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

$\underline{\underline{mat[i][j] = *(*(mat + i) + j)}}$

Note: all boxes are allocated.

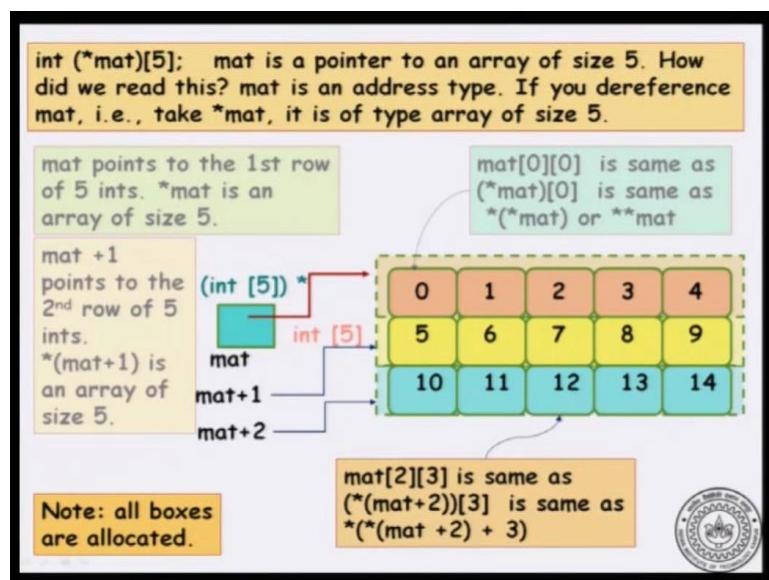


So, we can picturize in this way, if you dereference mat that is, if you take `*mat`, you will get some array of size 5 of integers. Now, let us look at the pictures. So, mat may be pointing to some array of size 5, which means that the next subsequent location will be another array of size 5, if it is a valid address. Now, for the first location we can refer to it as `mat[0][0]` or it is the same as `(*mat)[0]` or it is the same as `*(*mat)`.

So, remember the general formula that we had was, if I have the notation `mat [i][j]`, I can look it up as `*mat`. So, first let me translate `mat [i]`. So, that we have seen that this is simply dereferencing `mat + i`, that address. So, now we have one more subscript. So, in order to decode that, I will do the formula for a second time, so, this `+ j`. So, remember that this is the general form. So, similarly if you have `mat[0][0]`, I can write it as `*mat[0]` or I can write it as `*mat`, because i and j are both 0s.

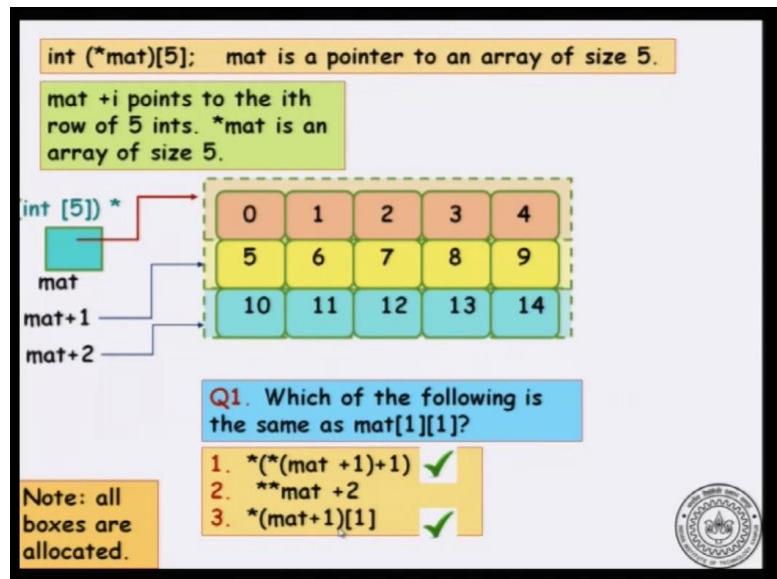
So, this is just a special case of the general form, `mat + 1` points to the second row of 5 integers. So, remember that the type of mat is, it is a pointer to an array of size 5 of integers. So, the next pointer location when you do `mat + 1` goes to the next array of size 5. So, `mat + 1` is another array of size 5. In particular, it may be the second row of a two dimensional array, where you have 5 columns, `mat + 2` will be similarly the third row and so, on.

(Refer Slide Time: 03:10)



So, `mat[2][3]` for example, if you apply the formula, it will come out to be `*(*(mat + 2 + 3)). Notice that, all boxes are allocated in this example.`

(Refer Slide Time: 03:25)



Now, $\text{mat} + i$ points to the i^{th} row of 5 integers and $*\text{mat}$ is an array of size 5, this is what we have seen. Now, you can in order to get comfortable with a notation, you can look at these formulas and try to decode. Like for example, you could try, what is the arithmetic way of representing the location $\text{mat}[1][1]$. So, you can see that it is definitely the first case, where it is $*(\text{mat} + 1)$. So, that is definitely true, because this is just the formula that we just now discussed.

But, if I do not decode both the subscripts, I decode only one subscript using pointer arithmetic and leave the other subscript as it is, then I know that it is also equivalent to 3. So, 3 is also another way of representing it and tried to convince yourself, why the second is not correct?

(Refer Slide Time: 04:28)

int (*mat)[5]; mat is a pointer to an array of size 5.

(int [5]) *

mat

mat+1

mat+2

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

Q2. We are given a function int search(int a[], int n, int key). How can we use it to search for key in the second row of mat[]?

Which is correct?

1. search(mat+1, 5, key);
2. search (* (mat+1), 5, key);
3. search(mat[1], 5, key);

1. mat is a pointer to an array of size 5. Same with mat+1, mat +2 etc.. mat+1 is the pointer to the next array of size 5 after mat.

2. Argument to search should be of type int *.

Now, let us understand this in somewhat more detail by considering a tricky question, we have a function int search. So, here is a function int search, int a, int n, int key. So, what does this function do? It will search for key inside array a of size n, a is an array with n elements and you have to search for it, search inside for it for the element key. If it is found, then you return the index where it is found, if it is not found, you return -1.

Because, -1 can never be a valid index in an array. So, when you return -1, you know that it is not present in the array. Now, can we use this, a function to search inside a 2D array. So, we are using a one dimensional function, in order to search inside a 2D array. Now, the basic idea is that we can search row by row, each row of a two dimensional array is somewhat like a one dimensional array. So, we will call search multiple times, once for each row in the array, until we either find it or we are done with all rows. The algorithm is, search it row by row.

Now, the question is which of the following is actually doing that? So, we have three expressions, `search(mat + 1, 5, key)`, `search *(mat + 1, 5, key)` and which of these will do it. Now, let us look at second, mat is pointing to an array of size 5. Therefore, `mat + 1` is also a pointer to an array of size 5, when we dereference that, we get an array of size 5,. So, that is the right type.

So, the first argument to search the second statement will be an array of size 5. So, therefore, the second call is valid. What about the third call? Again, mat of 1 is simply `*(mat + 1)`, if you translated into pointer arithmetic. So, the third line is just the second

line in discussed, instead of using pointer arithmetic notation, we are using subscript notation so, 2 and 3. In fact, are equivalent, so, 2 is correct. Therefore, 3 is also correct.

Now, think about why statement 1 does not make sense. So, `mat + 1` is actually a pointer to an array of size 5. Therefore, it is not the right type, it is not an array of size 5, it is a pointer to an array of size 5. So, it is not the correct type and therefore, the first call is not valid, the first option is a big delicate. So, I would encourage you to stop here and think about, why it is not correct?

(Refer Slide Time: 07:37)

`int (*mat)[5];`

`(int [5]) *`

`mat`

`mat+1`

`mat+2`

`0 1 2 3 4`

`5 6 7 8 9`

`10 11 12 13 14`

Q3. Write a function that searches for any occurrence

of a key in a $k \times 5$ integer matrix, k is a parameter. Use a 1-dim int search(int a[], int n, int key) as a sub-routine.

declaration of the function is:

```
int search_2d_5 ( int (*mat)[5], int nrows, int key,
                  int *row,           int * col)
/* row and col are the output parameters, returns -1, -1 if
the key is not found */
```

Design check each row of mat using the function search(). If search returns that key is found then we return the coordinates. Otherwise, we set row and col to -1 respectively.

Now, let us utilize the function in order to write our routine to search inside a 2D array. So, once again we are utilizing a one dimensional search routine in order to search inside a two dimensional array. So, let us say that, we are given this int search function which can search inside a one dimensional array for a key. Now, I will write a 2D function, a function which can search inside a 2D array.

Now, the correct declaration of the function would be `int *mat[5]`, `int n` rows `int key`, `n` rows is going to be the number of rows in the array. Key is the key, we are searching for and `int *row` and `int *col`. So, I want to focus on the first argument and the last two arguments. The first argument says that, I will pass you a pointer to an array of size 5, this is exactly what we should do because, then a two dimensional array can be just traverse by using `mat + 1`, `mat + 2` and so, on.

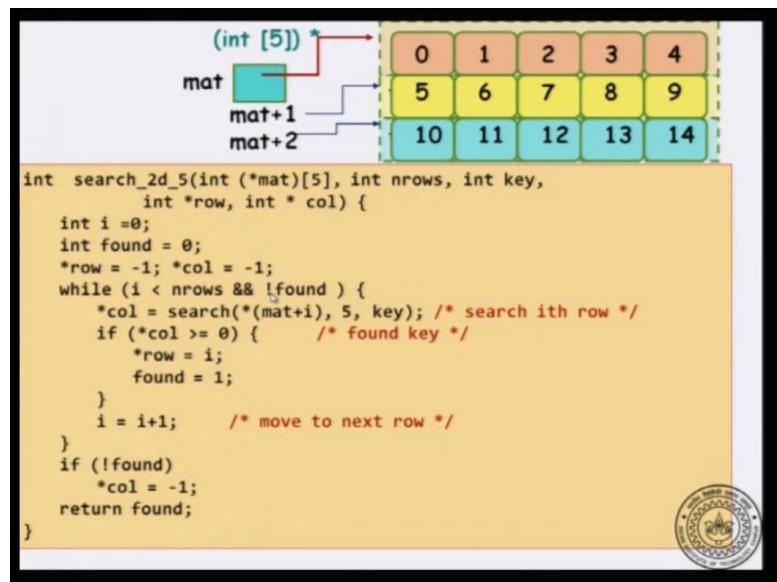
So, here is the correct type declaration that should accompany the 2D search routine, `n` rows is just the number of rows, key is the key. Why are we saying, `int *row` and `int *`

column? We want to return two things, if a key is found, we want to return its row index and its column index. Now, unfortunately a function can return only one value. So, how will you return two values?

So, we will say that we will not return two values. What we will do is, give me a pointer and I will write in that address, the correct row and the correct column, if it is found. Here is a standard way in C, where you might encounter a situation where you need to return two values and instead, what you pass are the pointers. The algorithm is what we have discussed before. You check each row of mat using the function search. If search returns success, then that will be the column index in that row, because search is searching inside a 1D array.

So, wherever it returns that will be the column index in the i th row. So, now you say that the column index is that and the row index is the i that I had. If it is not found in any of the rows, you return **-1**.

(Refer Slide Time: 10:16)



So, let us write the function, we have an i to go traverse for the rows, we have $found = 0$, this will be the flag indicating whether the key is found or not. And initially, you just set $*row = -1$ and $*col = -1$ to indicate that I am not yet found it, found the key. Now, you write the main loop which is going through the rows one by one. You start with row 0 and you go on, until both these conditions are true. That is, you have not seen all the rows, i is less than n rows and you have not found the key, so, not found.

What should you do to the i th row? I should say that search the i th row. So, the way I

say it is, `search *(mat + i)`. This is the same as saying `search mat[i, 5]`, which is the number of columns and key, which is the key that I want to search for, the return value is stored in `*call`. So, you dereference call and store the return value there. Now, search can return either you if the key is found, it will return the correct column index or it will return `-1`.

So, you just check for that, if `*col` is a non-negative number, then you say that it has been found. So, you say that the `row = i`, So, `*row` is `i` and `found` is now 1. So, at the next iteration you will exit out of the loop, because you have found the key. And then, the last statement in the loop will be just to increment the `i` variable. Finally, if you have done with all the rows and if you have exited out of the while loop, you check whether you exited out of the while loop, because you exhausted all the rows.

So, there are two conditions to exit the while loop, one is `i >= n` rows, that is one condition. The second is that `found = 1`, if you exited because, `found = 1`, then you can return the correct value without any problem. If you exited before, if all the rows were exhausted and you still did not find the key, then you have to say that column is `-1`. So, here is a brief code which will do this.

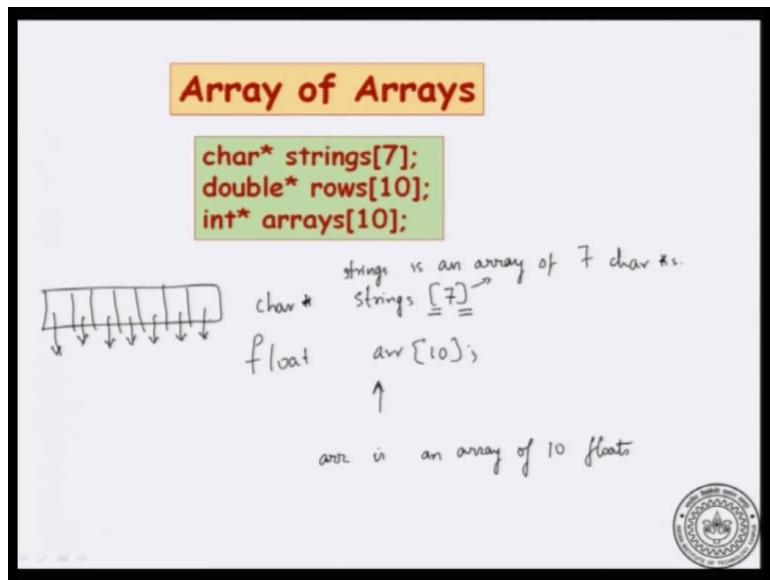
So, this code utilizes our understanding of two dimensional arrays as basically an pointer to an array of size 5 and here is why the number of columns is important. Because, in order to do `mat + 1` correctly, we need to know how many bytes to skip and this is crucially depended on the number of columns. The number of rows actually does not matter. Because, you can keep on incrementing the rows as long as the array is valid. The number of columns is important, because that is how you get to the next row.

Introduction to Programming in C

Department of Computer Science and Engineering

In this video will look at the last possibility with respective multi-dimensional arrays in pointers, this is known as an array of arrays.

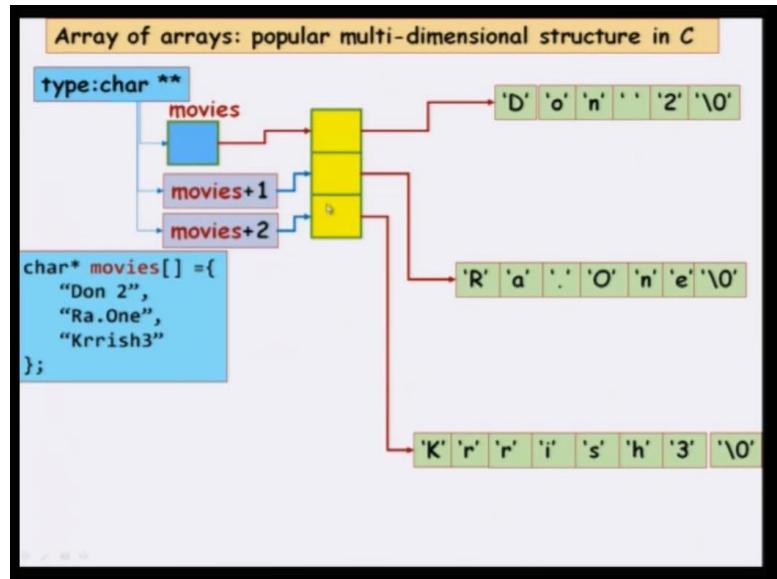
(Refer Slide Time: 00:04)



In order to understand this, let us just look at something we are comfortable with, if I had just a `float arr[10]` elements, then how would I read this, I will say that array is arr is an array of 10 floats. So, this is how I would read it, if I have more complicated declaration like `char*` as strings. So, notice that the precedence for this `[]`, is higher than that of the precedence for `*`.

So, this would actually be read as strings is an array of 7 character stars. So, that is how it could be read, because 7 would bind closer to strings. So, strings will become in array of size 7 and what type is it, it is `char *`. So, you replace float with `char *` and it is roughly the same phenomenal. So, the pictorially you can think of it like this. So, you have 7 cells in strings and each entry is a `char *`. So, each entry is a character pointer, you can think of it as a string, you can think of it is a character array whatever. So, here is the pictorial representation. Let us look at why we would need such a structure and what is the advantage of it? This is very popular structure almost as popular as two dimensional arrays themselves.

(Refer Slide Time: 02:14)

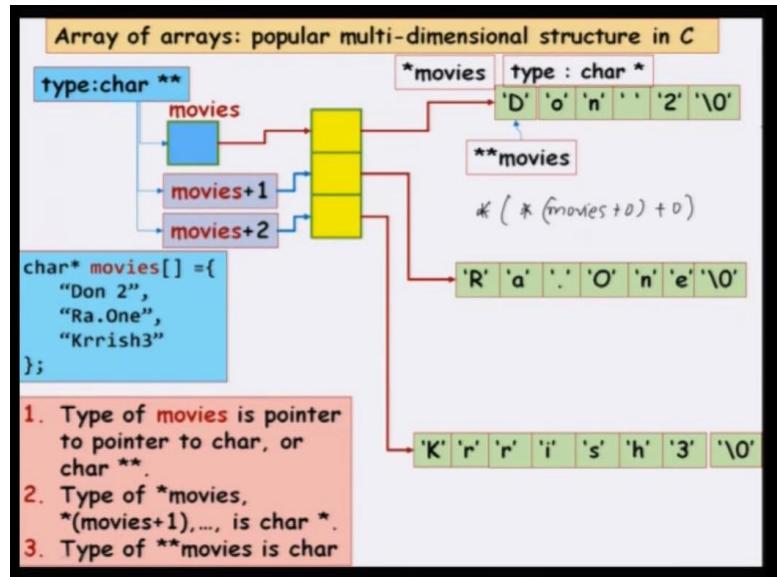


So, let us look at what it means for when we allocate character^{*} an array of arrays. So, we may want to let say store the names of several movies. And one of the things is the there is no maximum limit to the name of a movie and it can be as long as we want it can be as short as you want and suppose you want to store all of these is in a data. So, let say that we have `char *movies` and I declare it as an array of arrays and it contains the first array is Don 2, the second is Ra.one and. So, on.

Now, how will we do this, So, one way to do this is you say that `movies` is pointing to an array of arrays. So, `movies + 1` is pointing to another character array, `movies + 2` is pointing to another character array and. So, on. So, this is how we pictorially represent it, there are three entries and each entry is a character pointer. So, it can point to any character array what. So, ever.

And here you see the distinct advantage of this kind of representation over 2D matrices. Why? Because, in 2D matrices the whole point was the number of columns was fixed, that is how the pointer arithmetic worked. Here, the number of columns in one row can be different from the number of columns in another row. So, this representation is actually more useful when you have what are known as ragged arrays, that is one row and the next row may have very different lengths. And here is natural situation of storing strings when you need such a facility.

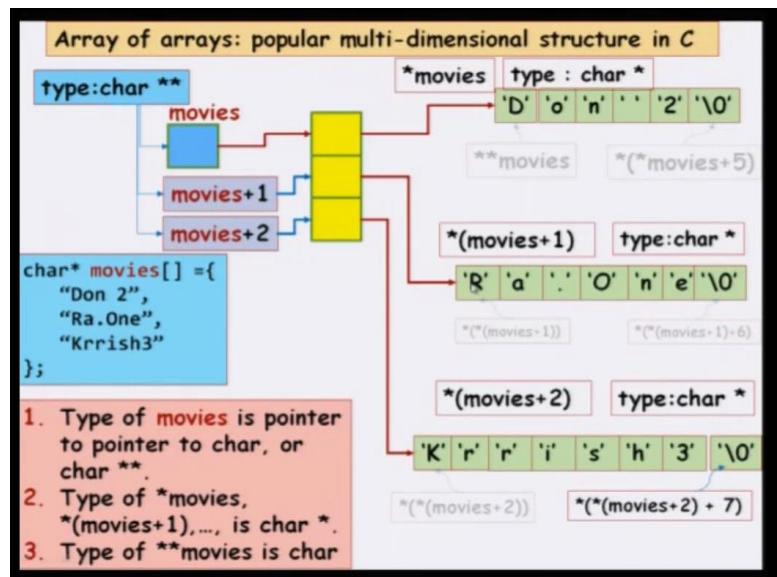
(Refer Slide Time: 04:08)



So, let us see what this means type of the variable `movies` is a pointer to a pointer to character or `char**`. Now, type of `*movies` is `char*`, because you dereference one level and type of `**movies` is `char`. So, let us look at it once more. So, `*movies` has type `char *`. So, in particular `*movies` will be this array, it is pointing to this array. So, `**movies` will be what is it according to the general formula, this will be `*(*(movies plus 0) + 0)`. So, this will be the pointer arithmetic version of accessing this cell which contains D.

But, instead you could also write `movies[0][0]`. Similarly, in order to get to the last cell here, you could say `*(*(movies + 5))`, it is the particular application of the general formula.

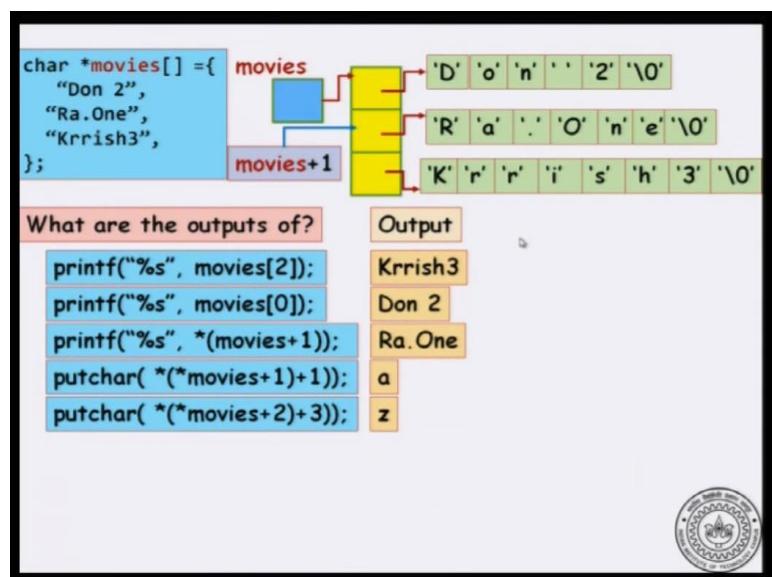
(Refer Slide Time: 05:32)



The second row will be `*(movies + 1)` again try to think in which ever notion you comfortable with. Because, you can also right this as movies 1, you will get the same result. So, `*(movies + 1)` will come to the second row and you have `*(*(movies + 1))` that would come to the first element in the second row and. So, on. So, `*(movies + 2)` would be the third array in the structure and here is how you access different elements in the third array.

So, notice the picture is slightly different here, even the representation suggest that, these rows need not be contiguous in memory. So, then location after this row n's need not be this row. So, the second row can be located arbitrarily far away in memory, the advantage due to that is that these rows can be of different length, they are not packed as in the 2D array.

(Refer Slide Time: 06:54)



Let us look at this particular thing in detail. So, that you get comfortable with in. So, suppose you have that array and I considered what is `printf("%s", movies [2])`, movies 2 will be the third character array, that is present in the structure. So, it will print Krrish 3. Similarly, movies 0 will print the first string and if you say `printf("%s", *(movies + 1))` by pointer notation, this is the same as the subscript notation `movies[1]`. So, that would print Ra.One.

Now, what happens if you have put `char *(movies + 1 + 1)`. So, again if you are more comfortable with a subscript notation, you can translate back in to the subscript notation, this will become `movies[1][1]`. So, what it will print is, this letter which is small a,

similarly for the last one. So, it will print whatever it will print the i. So, here is a whereas...

(Refer Slide Time: 08:24)

Array of arrays allows us to have a 2-dimensional structure with variable number of columns per row.

Write a program that takes a number between 1 and 12 and prints the name of the month.

```
void print_month(int month) {
    char *month_names[] = { "January", "February",
                           "March", "April", "May", "June",
                           "July", "August", "September",
                           "October", "November", "December" };

    if (1 <= month && month <= 12)
        printf("%s\n", month_names[month-1]);
    else
        printf("Invalid month\n");
}
```



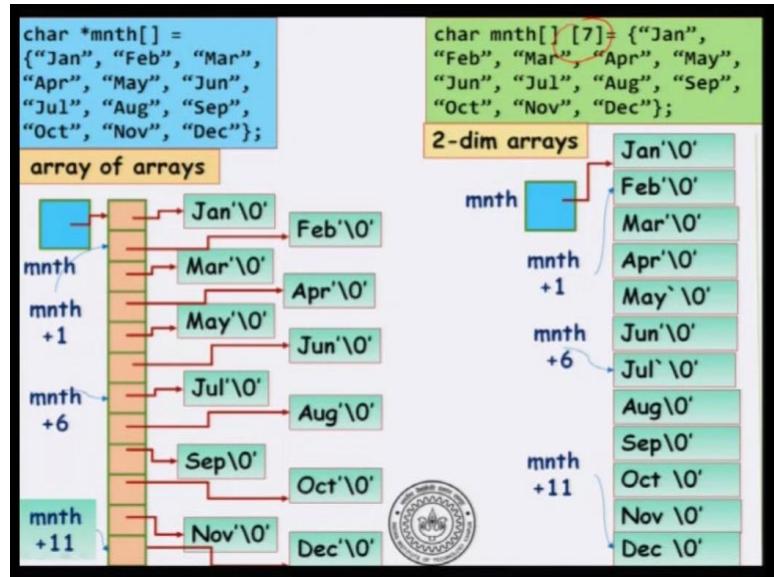
Array of array is now allows us to have a two dimensional structure with different number of elements per row and this is the advantage that it has. Now, let say that we want to right a very natural program, which is it takes a number between 1 and 12 and it prints out what is that month name corresponding to that number. So, I want to store in months and here is the problem different months have different lengths. We right now saw is solution to this problem, which is to store arbitrary length a strings in one structure, we would make an array of arrays.

So, you can say that `char *month_names[]`. So, this is an array of arrays of character and then you can just initialize it to the month names, you do this and then I will write the code. So, you can write the code in anyway. So, you can say that 0 is January and. So, on up to 11 is December. But, maybe it is more natural to say that 1 is January and. So, on up to 12 is December. So, I will check if the given month index is between 1 and 12, then I will print the month name month minus 1.

So, if you give the month as 1 you will print month names 0 which is January, if you give the month as 2 you will print month names 1 which is February and so, on. Now, if the month is not in this range it is in invalid month. So, you just print that an exist, So, here is a very simple program with illustrates what advantage you get out of this kind of array of array structure. You can store with in the same data structure different strings of

completely different lengths, this is not possible in a 2D array because, all you have to calculate something like the maximum column length. So, the maximum width month name for example, it could be September and then all the other names have to have exactly that width.

(Refer Slide Time: 10:48)

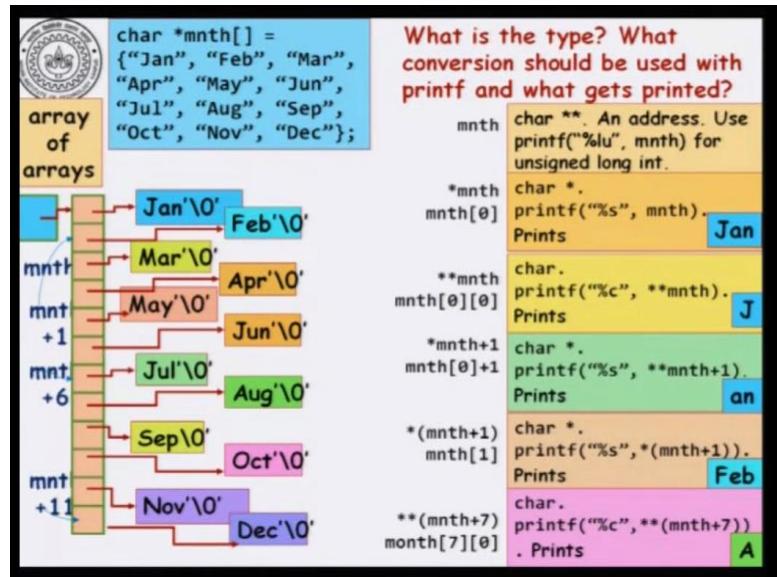


So, let us look at this the array of arrays picture is like this, you have an array of characters stars. Now, each of those characters stars may be pointing to different months. In this every month is exactly three characters long,, but you get the picture basically in this these rows can be of different lengths. So, contrast this with two dimensional arrays, where the chief feature of the two dimensional array is the following, you have to specify the number of columns.

So, the number of columns have to be specified and no matter what the exact string is, it will occupy 7 characters now. So, the remaining will be null fill or something. So, also notice that pictorially I have tried to represent it, the very next memory cell after the first row will be the beginning of the second row. So, after row 0 it will immediately start with row 1. Whereas, in the case of array of arrays row 0 and row 1 may be located arbitrarily far apart in memory.

The only connection is that the pointers to these rows are consecutively located in the pointer array, that is not the case here, it is actually located together in memory and it is represented in row major fashion, where each row will take exactly 7 letters. So, I hope the limitation of the two dimensional arrays in this case is clear.

(Refer Slide Time: 12:31)



So, you can try a few exercises in order to understand this notation a little bit, this concept of array of arrays a little bit better. So, let us look at the types of various concepts. So, if I have month, month is actually a `char **`, it is an address. So, if you want to print out month, I mean it is very rare that you need to print out month, you would use something like `%lu`, which is long unsigned for printing the unsigned long int. What happens if you access `*month`? Now, you dereferencing one level below. So, it will be a `char *`.

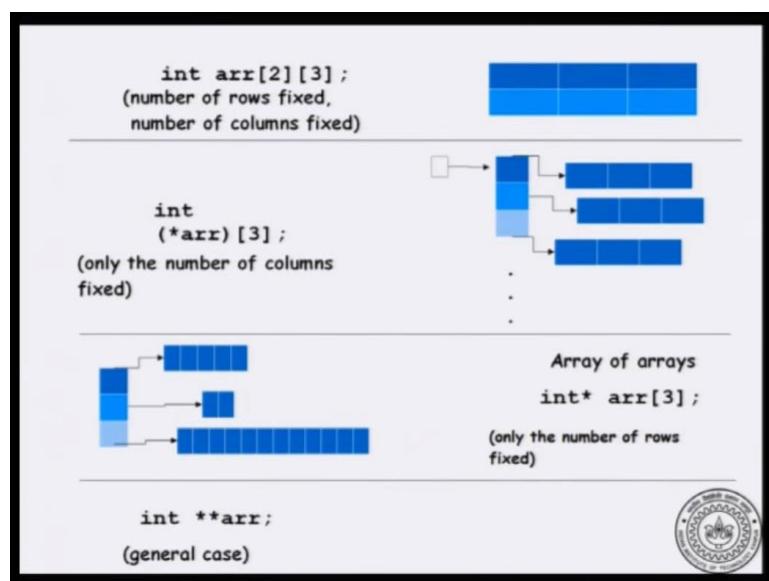
So, now, if you print percentages `*month`, it will print January. If you print a `**month`, you have dereference two levels and you will get the first letter of the first array, which is j and you can try out a few other thinks, you can say the `*(month + 1)`. So, one way you can do it is you translated to subscript notation and try to see what it will print and there are... So, I would encourage you to try out these examples, in order to get that translation between arrays and pointers correct.

(Refer Slide Time: 14:01)

Comparison	
Array of Arrays <code>char *mnth[12];</code> Individual array sizes can be different. E.g., mnth[6] is "July" (size 5). mnth[8] is "September" (size 10).	2-dim arrays <code>double matrix[10][10];</code> All rows must have same length.
Useful in string/word processing and representing graphs, non-uniform sized structures. Not so useful for matrices.	Natural for matrix computations. Not so good for non-uniform sized structure.

So, the comparison between array of arrays in two dimensional arrays on the one hand individual array sizes can be different in the case of array of arrays. In the case of two dimensional arrays all the rows must have exactly the same number of columns. So, array for array is useful in a lot of string processing routines, in representing graphs and something like that. But, two dimensional arrays are more advantages when you deal with matrices, because mathematical matrices typically have a fix number of columns.

(Refer Slide Time: 14:40)



So, here is wrong picture, but it short of gives you an idea of how to look at these structures. So, if I have `int array[2][3]` you can think of it as the number of rows is fixed and the number of columns is fixed, this is not actually what happens in C, in C actually

the number rows does not matter, the number columns matters. But, you can for a moment to make it easier to think about, think that if you declare it in this way, this is when the number of rows is fixed in the number of columns are fixed.

So, in particular if you know beforehand that your data structure has a fixed number of rows and fixed number of columns, then it is probably better to you said 2D array. Now, if you have `int *arr[3]`, now this means that arr is a pointer to an array of size 3. So, here the number of columns is fixed. But, the number of rows is variable, it you can have any number of rows, on the other hand the third case `int *arr[3]`. So, it is an array of 3 elements each of type `int *`.

So, you can see that this is one situation, where you have 3 pointers,, but each of them can point to arrays of arbitrary length. So, this is a situation where the number of rows can be seen as fixed and the number of columns is variable. And the general case can be `int **`, which is where the number of rows and the number of columns are both waiting.

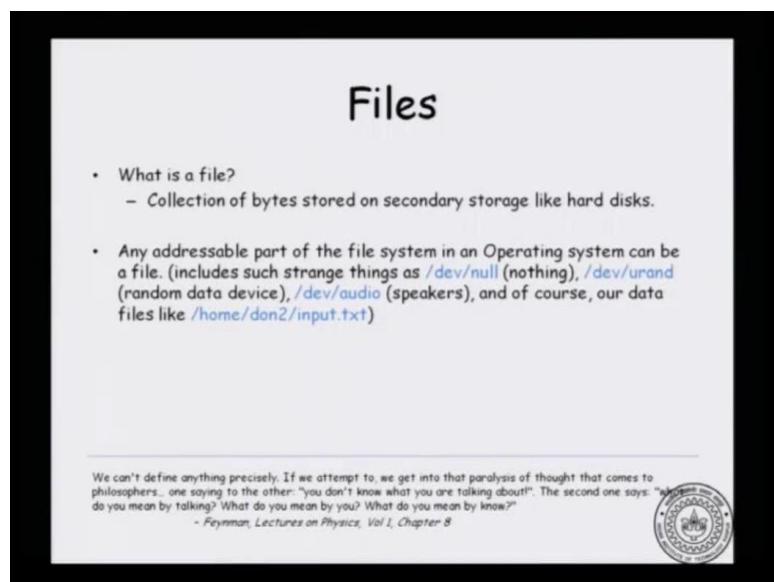
So, you can think of it and this way, this is not a correct picture. But, when you want to modal data, this is probably and you know that you are in a situation, where the number of columns is variable. But, the number of rows you know beforehand probably you should go for array of arrays. If you are in a situation, where you know that the number of columns is fixed, but you do not know how many data there are, then you can go for `int*` array of size 3. So, you can go for the second alternative and so, on. So, this picture is not quite accurate,, but it is indicative of the usage.

Introduction to Programming in C

Department of Computer Science and Engineering

In this video, we will look at a few basic things about file handling in C. This is a vast topic in itself and we will see just the basics of this. So, let us begin by describing what are files.

(Refer Slide Time: 00:15)



Now, you would think that the most natural way to define what a file is, it is a set of bytes or a collection of bytes sitting in secondary storage, like may be your hard drive, may be your CD ROM drive or DVD drive or something, it is known as a secondary storage device. But, the actual description is that, any addressable part of the file system in an operating system is a file. Now, this includes extremely bizarre strange things.

For example, `/dev/null` in Linux, this stands for nothing. So, if you write to `/dev/null`, it is like discarding the data. Similarly, `/dev/urand`, this is the random data device. If you read from here, you will get random data, `/dev/audio` is speakers. So, if you write some data into that, it will be heard on the speakers. And of course, plain old data files, for example, in a home directory you may have `/home/don 2/input.text`.

So, `input.text` is just a collection of bytes. So, we will not bother with defining what a file is, but it is something that can be manipulated using the file system interface. So,

you can open a file system interface to that file, you can read from that file, you can write into that file, you can close that file and so on.

(Refer Slide Time: 01:53)

File Access

- Recall (long back): we remarked that 3 files are always connected to the program :
 - stdin : the standard input, from where scanf, getchar(), gets() etc. read input from (usually has *file descriptor 0*)
 - stdout : the standard output, to where printf(), putchar(), puts() etc. output to. (*file descriptor 1*)
 - stderr : standard error console. (*file descriptor 2*)



Now, recall that in one of our earlier lectures, I said that there are three files works which are available by default to all c programs. So, these are standard input stdin. This is associated usually with a key board and this is where scanf, getchar, gets this kinds of functions get their input from, it has file descriptor 0. Similarly, standard output stdout is where printf, putchar, puts all these functions output the data to. This is usually visible on the terminal under screen, this has file descriptor 1. We also have a third file, which is known as standard error. This is the standard error console and which has file descriptor 2. Usually, you can print error messages to stderr. We have not seen how to print error messages to stderr, so far we will see that in this video.

(Refer Slide Time: 03:00)

Reading input using the standard file descriptors

- `./a.out < inputfile`
 - This will use inputfile as the source, instead of the keyboard
- `./a.out > outfile`
 - This will use outfile as the output, instead of the terminal
- `./a.out 2>errfile`
 - This will redirect error messages to errfile, instead of stderr.



Now, how do you read input using standard file descriptors, but from sources other than key board and so on. So, for example if you are running `a.out` and you want to take the input not from the key board, but, from an input file. You can say `/a.out < input file`, this says that the input is coming from the file, input file. So, this is the input file as the source, instead of the key board.

Similarly, if you want to redirect, so if you want to redirect the output to a particular output file, instead of the screen, you can says `/a.out > out file`. So, this will use the out file as the output instead of the terminal. And if you want to output something the error messages to error file, you can say `/a.out 2 > error file`, 2 stands for the standard error console. So, if you say redirect this to this file, you will say ok.

The standard output should get the standard output messages and the standard error messages should go to err file.

(Refer Slide Time: 04:17)

Some usage scenarios

- What if we want to read input from multiple files, and redirect output to multiple files in the program?
- The redirection mechanism is provided by the Linux shell, and is not a part of the C programming language
- Does C provide any functions to handle files?



So, Linux gives you some facilities to take input from other files using the standard input and the output. So, what you say is that, instead of the standard input, you can use this <, > arrows, in order to redirect input from some file or output to another file or error to another file. So, this is the facility that Linux gives you. But, consider the general situation, when you have a program, you want to read the input from multiple files and may be output to multiple files.

So, this is the general situation, we just saw how to take input from one particular input file, how to output to another output file using the redirection operator, the < and > operation on Linux? So, the redirection mechanism is provided by the Linux shell and is not a part of the C program in language. So, is there a way to do it in C itself, rather than using the facilities of Linux. So, can we read from other files, other than the standard key board? Can we write into other files, other than writing on to the screen, standard output and so on.

(Refer Slide Time: 05:30)

General Scheme of File handling in C

1. Open the file for reading/writing etc. - this will return a file pointer - this points to a structure containing information about the file: location of a file, the current position being read in the file, and so on.
 - FILE* fopen (char *name, char *mode)
2. Read/Write to the file
 - int fscanf(FILE *fp, char *format, ...)
 - int fprintf(FILE *fp, char *format, ...)
3. Close the File.
 - int fclose(FILE *fp)

Compare with
scanf and printf
- first argument
fp is missing

So, we will look at the general scheme of file handling in C, all these functions that I am going to talk about are in stdio.h, itself. So, you do not need to include any more files. So, if you want to open the file for reading or writing etcetera, we need to first open the file. The three standard files stdout, stdin and stderr are available to the program. Any other file, you have to open the file. And the function to do that is fopen takes two arguments name and mode, we will see what these are soon and it returns something called a file pointer.

A file pointer is a pointer to a structure and that structure contains a lot of information about the file. For example, where is it situated, the current position being read in the file. So, may be you are read 1000 bytes and you are about to read the 1000 and first byte. So, it has that information and various maintenance information, about the file. Now, in order to read or write into the file, you can use fscanf or fprintf. These are the analogs of scanf and printf, which allow you to right to an arbitrary files.

It takes three arguments, at least two arguments, the first is the file pointer, where you want to write the file, where you want to read from the file and so on. And then, there is a format specified, just as a normal scanf or normal printf and then further arguments. So, the difference here is that, whereas scanf and printf started with a format specifies, we have an additional file pointer in the beginning.

So, compare with a scanf and the printf, the first argument fp is missing. This is because, scanf just assumes that the file it has to read from is this standard input. And printf assumes that it has to print to the standard output and to close the file, you say fclosefp. And notice the way, the fscanf, fprintf and fclose work, they do not take the file name as input, only fopen takes the name of the file as input.

Whatever fopen returns the file pointer, those are the arguments to fscanf, fprintf and fclose. This is because, once a file has been opened, all the information that fscanf, fprintf and fclose need are already in the structure pointer 2 by fp.

(Refer Slide Time: 08:18)

- Write a program that will take two filenames, and print contents to the standard output. The contents of the first file should be printed first, and then the contents of the second.
- The algorithm:
 1. Read the file names.
 2. Open file 1. If open failed, we exit
 3. Print the contents of file 1 to stdout
 4. Close file 1
 5. Open file 2. If open failed, we exit
 6. Print the contents of file 2 to stdout
 7. Close file 2



Now, let us write a very simple program, it takes the names of two files and what it does is, it first prints the contents of the first file and then prints the contents of the second file and these will be output to the standard output. What is the algorithm? It is very simple, you have to first read the file names, then open file 1, if open fails, we exit. Now, you print the contents of file 1 to stdout, after you have done, you close file 1. Then, you open file 2, check whether open has succeeded, if it has failed, we exit. Then, print the contents of file 2 to stdout close file 2 and that is it.

(Refer Slide Time: 09:08)

The slide has a title 'Opening Files' and a list of bullet points:

- FILE* fopen (char *name, char *mode)
- The first argument to fopen is the name of the file - can be given in short form (e.g. "inputfile") or the full path name (e.g. "/home/don/inputfile")
- The second argument is the mode in which we want to open the file. Common modes include:
 - "r" : read-only mode (Any write to the file will fail). File must exist.
 - "w" : write mode. The first write happens at the beginning of the file, by default. Thus, may overwrite the current content. A new file is created if it does not exist.
 - "a" : append mode. The first write is to the end of the current content. File is created if it does not exist.

UNIVERSITY OF KERALA
Kerala, India

So, let us see what each of these steps in slightly more detail. How do you open the file? We open it using a standard call called fopen, fopen takes two arguments, the name and the mode as character pointers and returns a file point. The first argument name is the name of the file and the name of the file can be given in short form. Suppose, you are already in a directory, where that file is situated. Then, you can just give the name of the file. For example, input file or you can give the full path name of that file in the operating systems.

So, for example input file may be in the directory `/home/don`. So, in that case you can give the name as `/home/don/input file`. So, this will be the full path file, either of this is accepted. Now, the second argument is the mode, this is the way in which you want to open the file. So, what are the common modes? For example, if you give r, this will open the file in read only mode. This is, if you want to just read a file and not write to that file.

There are also other situations, where the medium itself may not support writing. For example, if you have a CD ROM disk then you cannot write to that. So, it can only be opened in a read only mode, if you give w, this is the write mode. Now, the first write happens at the beginning of the file. So, if the file already exists, it will be over written. If a file does not exist, so this is the name of a new file that we support commonly is known as the append mode, we specify that by saying the mode is a.

So, if you open the file for append mode, then instead of writing at the first location of the file, it will write at the end of the current file. So, if the file does not exist, then it will start from the first location. If the file exists, it will go to the end of the file and start writing from there. So, append does not overwrite the file.

(Refer Slide Time: 11:38)

Opening Files

- If successful, fopen returns a *file pointer* - this is later used for fprintf, fscanf etc.
- If unsuccessful, fopen returns a NULL.
- It is a good idea to check for errors (e.g. Opening a file on a CDROM using "w" mode etc.)



We have seen the arguments of fopen. Now, let us look at what it returns? If successful fopen returns what is known as a file pointer. This is later used for fprintf, fscanf, fclose as I just mentioned. If unsuccessful, the file may be you try to open a nonexistence file for reading or you try to write to a file which cannot be return to. For example, it is a file sitting inside a CD ROM drive and you are not allowed to write to it. So, if you try to open the file in write mode, then you have a problem.

So, for whatever reason if the file open does not succeed, then the fopen returns in null and it is always a good idea to check for these errors. So, just try opening a file and always check whether it has return the null.

(Refer Slide Time: 12:33)



The Program

```
int main()
{
    FILE *fp1, *fp2;
    char filename1[128], filename2[128];
    gets(filename1);
    gets(filename2);
    if( ( fp1 = fopen( filename1, "r" ) ) == NULL ) {
        fprintf( stderr, "Opening File %s failed\n",
                 filename1 );
        return;
    }
    copy_file( fp1, stdout );
    fclose( fp1 );
    if ( ( fp2 = fopen ( filename2, "r" ) ) == NULL ) {
        fprintf ( stderr, "Opening File %s failed\n",
                  filename2 );
        return;
    }
    copy_file ( fp2, stdout );
    fclose ( fp2 );
    return 0;
}
```

So, let us write the program that we were discussing, which will take two input files and print one file and then print the other file. So, the program is fairly simple, we have a main function, we have two file pointers `*fp1` and `*fp2`. And then, two file names `filename1` and `filename2`, you get the `filename1` from the input, you get `filename2` from input using `gets` functions. Now, what we have to first do is, write the contents of the first file.

So, try opening the file, so `if (fp1 = fopen(filename1, "r"))`. Because, we just need to read from the file, we do not need to write into it. So, open it in r mode, if it is successful that is, if it is or rather if it has failed. So, if it has returned a null, then you just say `printf` that it has failed. And here is for the first time, we have seeing how to print to this standard error. So, `stderr` is any other file is similar to any other file, you can just say `fprintf stderr`. And then, opening file failed `filename1`.

So, we try to open `filename` with `filename1` as the name, but there was some error. So, you print that to the error terminal, which is `stderr`. Now, once you do that will call the function `copy_file(fp1, stdout)`. So, here is a function that we will write, which will copy from a source file to a destination file and what it takes are pointers to those files.

Once you have done, you close the file 1 and then, you repeat the whole process, the exactly the same process for file 2. So, try to open it, if there is an error, you print the error message to `stderr`, then `copy_file(fp2, stdout)` and finally, close the file. Once you

have done, you can return from main. So, now what is left is, what is this copy file function?

(Refer Slide Time: 14:59)



copy_file function

```
void copy_file ( FILE *fromfp, FILE *tofp )
{
    char c;

    while ( !feof ( fromfp ) ) {
        fscanf ( fromfp, "%c", &c );
        fprintf ( tofp, "%c", c );
    }
}
```

So, let us look at the copy file function. Now, there are two ways to start writing any function which takes files as arguments. One is you can take the file name as the argument itself and within the function, try to open the file. So, you will get a file pointer and you can start reading from the file using fscanf I am writing to the file using fprintf, this is possible.

It is somewhat more convenient to say that I am assume that the files are already open and I am getting the file names as the point using file pointers. So, this avoids duplication of work, the main does not have to open the file. And then, every function has to open the file again and again. Instead, what you can just say that, I assume that the caller function has already is a file open and I will just take a file pointer as the argument.

So, let us look at this function, it is a void function. So, it does not return anything, it just performs an action, it is name is copy file takes two arguments, fromfp which is a file pointer to after to the source file and tofp, which is file pointer to the designation file. And what is a function do? We have a character c and here is a function, we will see in a later video. But, right now it just checks whether fromfp has encountered enter file.

So, feof just tells you whether you have done with the from file. So, if you are not done with a from file what you do is, you scan one character from the from file, so `fscanf(fromfp, "%c", &c)`. So, this will read one character from the source file fromfp and read it into the variable c. What we have to do is, to print that to tofp. So, you say `fprintf (tofp, "%c", c)`. So, this is exactly like scanf and printf, but taking one extra argument.

So, in the case of scanf you just says, what is the source file that is the file pointer argument. In the case of fprintf, you have to take the designation file which is tofp, that is the extra argument in that expression.

Introduction to Programming in C
Department of Computer Science and Engineering

(Refer Slide Time: 00:06)

Some other file handling functions

- **int feof (FILE* fp);**
– Checks whether the EOF is set for fp - that is, the EOF has been encountered. If EOF is set, it returns nonzero. Otherwise, returns 0.

- **int ferror (FILE *fp);**
– Checks whether the error indicator has been set for fp. (for example, write errors to the file.)



In this video we will see some more common file operations; these are by no means the only file facilities that C provides you, but in common programming practice these are the functions that people of in use. So, we have seen this in the code that we wrote. The first function is feof, and then it takes a file pointer. What it does is, it is checks whether you have encountered end of file while operating on fp. So, maybe you are trying to read the file, and you have already reach the end of file. So, if you have already reach end of file, that is if EOF is set, then feof returns a non zero value. If feof is not set, that is you have not completed the file yet by seeing end of file, then feof returns 0. So, in order to check whether a file has still has some data, you can just say not of feof fp. So, that will check for the fact that the files still has some data.

Now another useful file function is f error. So, the f error function what it does is it takes the input file pointer, it takes the file pointer *fp, and checks whether you encounter some error while reading the file. So, error may be of many kinds, for example you are trying to write to a read only medium like cd or maybe you trying to write to a file system, and the file system is full. You are trying to write to a hard drive and the hard drive is full. So, then you might encounter an errors. So, there are various errors that you encounter in files operations, and ferror checks for some of these errors. So, if the error indicator has been set for fp, then ferror returns a non zero value, otherwise it is says 0.

(Refer Slide Time: 02:14)

Some other file handling functions

- int fseek(FILE *fp, long int offset, int origin);
 - To set the current position associated with fp, to a new position = origin+offset.
 - Origin can be:
 - SEEK_SET: beginning of file
 - SEEK_CUR: current position of file pointer
 - SEEK_END: End of file
- int ftell(FILE *fp)
 - Returns the current value of the position indicator of the stream.

*fseek(fp, 10, SEEK_SET);
↓
10 bytes from beginning*



Now here is some couple more interesting functions whose who they are very useful, and a call by commonly use whenever we deal with files. For example, we can have something called fseek. fseek is a function which allows you to start reading from, our start writing to arbitrary locations in the file. So, often we may want to read into the 10000 byte directly, and we do not want to be bother with reading the first 9999 characters discarding them, and then coming to the 10000 th character. This may be lot of wasted time. It will be more convenient, if I can directly jump to the 10000 th location in the file. So, is there a function that allows to you do to that yes, there is just thing call fseek. Now what it takes is the file pointer, and it takes two arguments; one is known as an offset, and the other is known as the origin.

So, let us look at the offset and the origin in greater detail. So, suppose I want to read from the 10 th byte of the file. So, I could say fseek, and suppose by file pointer is fp, I will just say let say I want to read from the 10 th point from the beginning of the file. What I can say is SEEK_SET. So, if I do this what will happen is that? It will start from the beginning, SEEK_SET is the beginning of the file. So, it will add 10 bytes to from the beginning of the file, and it will start from there. So, if I know that I want read from the 10 th byte, then I can say that start from the beginning of the files SEEK_SET says origin of the beginning of the file plus 10 bytes. So, this is 10 bytes from beginning.

Now, there are other situations, for example you might want to say that I want to start from the 10 th byte from the current location. I have already read many bytes. Now I want to skip the next 10 bytes.

(Refer Slide Time: 05:12)

Some other file handling functions

- int fseek(FILE *fp, long int offset, int origin);
 - To set the current position associated with fp, to a new position = origin+offset.
 - Origin can be:
 - SEEK_SET: beginning of file
 - SEEK_CUR: current position of file pointer
 - SEEK_END: End of file
- int ftell(FILE *fp)
 - Returns the current value of the position indicator of the stream.

fseek (fp, 10, SEEK_CUR);
↓
10 bytes from current position



So, is there way to do that again what you can do is, if you say fseek fp and let us say 10 itself, but SEEK_CURRENT. So, there is a typo over here, this is just CUR. So, if I say this, then what I need to do is what it will perform is, it will say 10 bytes from the current position. So, I have already read 100 bytes from the file, and then I say fseek 10 bytes from the current location. What it will do is jump to 110 th location.

(Refer Slide Time: 06:07)

Some other file handling functions

- int fseek(FILE *fp, long int offset, int origin);
 - To set the current position associated with fp, to a new position = origin+offset.
 - Origin can be:
 - SEEK_SET: beginning of file
 - SEEK_CUR: current position of file pointer
 - SEEK_END: End of file
- int ftell(FILE *fp)
 - Returns the current value of the position indicator of the stream.

*fseek (fp, -10, SEEK_End);
↓
10 bytes before the end of file*



Now I could also say something like... So, here is a very common situation, I want to start reading from the 10 th byte from the end. So, I want to regardless of the size of the file I want to jump to the end, and then rewind 10 bytes and start from there. So, in that case I can say the origins SEEK_END. So, that is the end of the file and where do I start from SEEK_END plus something does not make any sense, because it is it will refer to something that does not exist in the file. So, you could say SEEK_END **-10**. So, this is 10 bytes before the end of the file. So you can use fseek in several ways and is a very convenient function, because it allows you to jump to arbitrary location in the file. And it will work as long as the target location origin plus offset is a valid location in the file. Now there is also something called ftell, which will tell you the current position in file. So, if it will take a file pointer as the argument ***fp**, and it will return you where in the file your currently at.

(Refer Slide Time: 07:35)

Opening Files

- There are other modes for opening files, as well.
 - "r+" : open a file for read and write (update). The file must be present.
 - "w+" : write/update. Create an empty file and open it both for input and output.
 - "a+" : append/update. Repositioning operations (fseek etc.) affect next read. Output is always at the end of file.



So, with this let us take a look at a few more modes in the file operations. So, when you open the file we saw that you could open it in mode r w a. Now there are also some other special modes that see give see, for example there is something called r+. This says you can a open file for reading and writing. So, this is essentially an update mode. w+ will be write an update. So, create an empty file and update that file. And there is something call a + which is appended update, this is somewhat strange. If you do any fseek after you open the file in a + mode, then the read will be effected. So, suppose I am at 100 th location, I have write 99 bytes, I am at the 100 th byte. If I read, if I now do an fseek to 10 bytes ahead. So, now I will be at the 110 th byte.

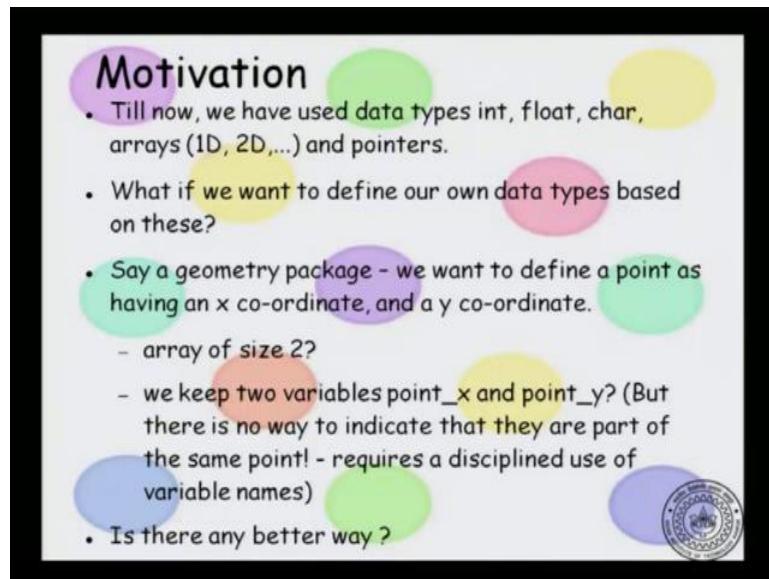
Now there are two possibilities now, I can read from here or I can do and fprintf, fscanf will start from the 110 th byte, it will be obey the fseek. fprintf will always print at the end of the file. So, that is the append part of it. So, fprintf is always output is always at the end of the file, and reading will be depended on any fseek that you do. So, fseek will never affect the where you print, it will always be the end of the file. So, a + is a very special for it. These are some additional file operations that you might find useful while coding in C.

Introduction to Programming in C

Department of Computer Science and Engineering

So, in this lecture will talk about structures and C, which is syntactic feature that C provides in order to define new data types.

(Refer Slide Time: 00:10)

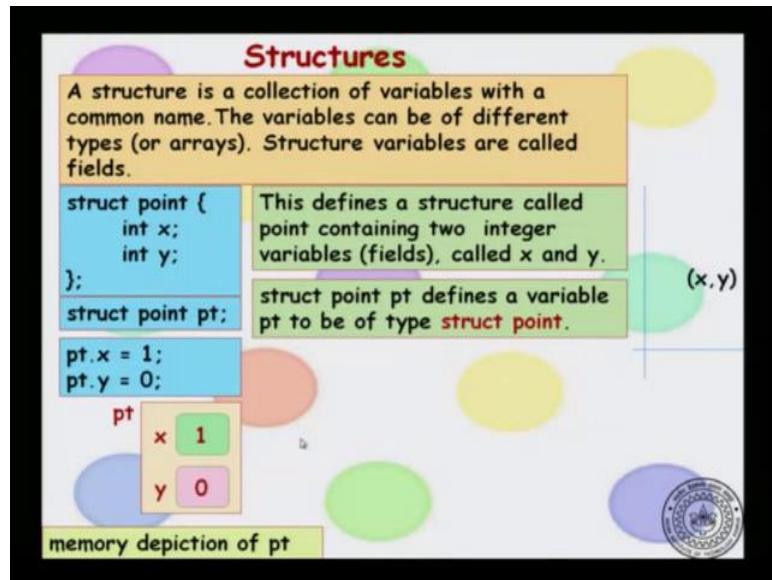


So, let us look at the motivation. Till now we have used the data types that C language has already provided like int, float, character, and also we have seen data types like arrays and we could define arrays of end arrays of character and so on, we have also seen pointers which can hold the address locations of other variables. Now, what if we want to define our own data types using the data types that are already available. So, if you want to define custom data types does the language provide any feature to do it. Before we reach there we will just take a look at why you would want to define such a data type. So, let say that we have designing a geometry package, and we want to define a point on the plane as having a x coordinate and a y coordinate. Now thus a easy way to do it if you have arrays, you could hold a point inside an array of size 2.

Now you would keep the first coordinate in the the x coordinate as the zeroth element in the array, and the y coordinate as the first element in the array, this is one way to do it. Another way is to keep two variables point underscore x and point underscore y, and these are the x coordinates and the y coordinate of a single point, this is another way to do it. But in both these solutions this no way to indicate that these two are in intended to

be the x coordinate and the y coordinate of the same point. That programmer has to impose considerable discipline in coding in order to maintain this meaning. So, is there a more natural way to do it in C. So, we want to define a point data type, and a point data type internally has two integers; one- two floating point, one in x coordinate and another a y coordinate.

(Refer Slide Time: 02:18)

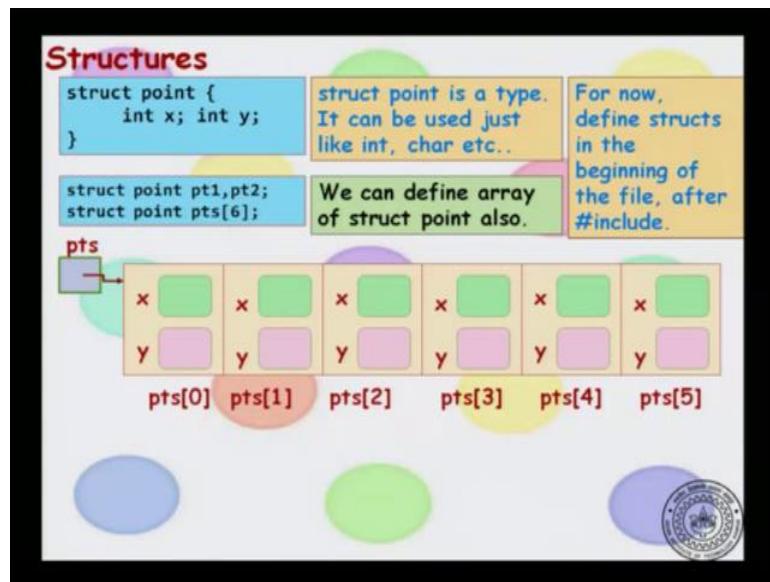


So, now we will define a structure as a collection of variables with a common name. Now the variables can be of different types, this is unlike arrays. We know that in an array you can hold multiple values in a contiguous sequence, but all the values have to be on the same time. So, what is different here with structure is that you can hold multiple values in the same data type, also the same data type can hold multiple sub types within itself. So, structural variables are called fields. So, let us look at a example structure, we define something called a struct point, and it has an int x and a int y coordinate.

And how do you... This is the data type notice that the data type declaration has a semicolon at the end. Now how do you define a variable with this data type, you can say struct point pt. So, pt is one variable that is of data type struct point, you cannot say point pt you have to say that it is a struct point. Now how do I assign values to this data type. So, structure is a composite data type that is two internal components; one is x, and the other is y. So, in order to say that the internal components will have certain values I need to say how to you get to these internal components, and this is done by the dot operator.

So, you can say `pt.x=1`, that would assign the x field inside the pt structure to 1. Similarly `pt.y=0`, what it will do is it will take the y field of pt structure and assign it to 0. So, the internal memory representation after executing the statements will be that pt is a structure; it has two sub fields - x and y, and x will be assigned to 1 and y will be assigned to 0. Struct point is the name of the data type, and pt is the name of the variable.

(Refer Slide Time: 04:42)

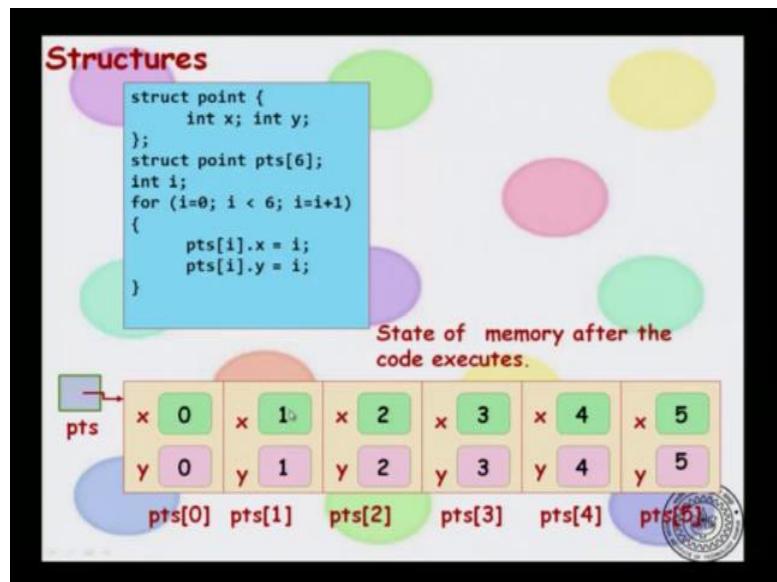


Now as far as C is concerned this structure data type that you define is just like any data type that it provides. So, user defined data types especially structures at ((Refer Time: 04:57)) more or less in the same way as an ordinary data type. And we will see that with an ordinary data type there are multiple things that you can do with it, you can initialize, you can declare a variable to be of that data type, we have seen already how to do it? You can initialize a variable of a particular data type, we will see how to do it?

Similarly you can pass a data of a particular type to a function, and return a variable of that type from a function, you will see that all these are possible with structures as well. So, we will see these with examples. Now struct point is just a type and it can be used like any other standard C data type; even though you as a programmer have defined it. For now how do you define a structure, usually you define all the structures that you need at the very top of the file, just after the `#include`. Now, you know that with the standard data type you can define an array of the data type, if you have you can declare array of int, array of characters and so on.

Similarly, you can define a struct point array, C is says that user defined data types are treated in the same way as the standard data types that it provides. So, here let us look at this example. So, we have struct point pt 1, pt 2. So, this say is that pt 1 and pt 2 are two variables which are of type struct point. What about pts 6, it is an array of size 6 where each element in the array is of type struct point. So, to visualize it you would imagine that it is like this; there are 6 cells - each cell contains a struct point. So, each cell will have two fields - x and y. Now how do I assign values to these elements in the array.

(Refer Slide Time: 06:58)



So, you would write a loop, for example of the following form. So, you would say for $i = 0$, $i < 6$, $i = i + 1$ and $\text{pts}[i].x = i$. So, $\text{pts}[i]$ is the i th element in the array. In the elements in the array are of type struct point. So, $\text{pts}[i]$ will be a struct point variable, and that variable has 2 fields - x and y. So, I can say $\text{pts}[i].x = \text{something } i$, and $\text{pts}[i].y = i$. So, at the end of execution of this loop the result will be of the following form pts 0 is a structure, and its x and y coordinate are 0; pts 1 is another structure, its x and y coordinates are 1, and so on. So, what are the characteristics features of an array, that it contains cells which are contiguously allocated, so they will be allocated one of to the other in memory, and each cell is of the same type as the others in the same array. So, both those are maintained. These structures will be stored one after the other in memory also everything in the array is a struct point. So, an array of structs is similar to an array as ints except for the fact that within a cell you will have values that can have subfields.

(Refer slide Time: 08:34)

The slide has a title "Functions returning structures" at the top right. On the left, there is a code snippet:

```
struct point {  
    int x; int y;  
};  
  
struct point make_point(int x, int y)  
{  
    struct point temp;  
    temp.x = x;  
    temp.y = y;  
    return temp;  
}  
int main()  
{  
    int x, y;  
    struct point pt;  
    scanf("%d%d", &x,&y);  
    pt = make_point(x,y);  
    return 0;  
}
```

On the right side, there are three numbered points:

1. **make_point(x,y)** creates a struct point given coordinates (x,y).
2. Note: **make_point(x,y)** returns struct point.
3. Functions can return structures just like int, char, int *, etc..

Below the code, a note says: "Given int coordinates x,y, make_point(x,y) creates and returns a struct point with these coordinates." There is also a small logo in the bottom right corner.

Now we will see what else can you do with structures. Just like a variable of a int data type, you can return it from the function. If you declare an int x, you can say return x. Now let say it can be a right functions which can return struct point. So, the theme of this lecture is that user defined structures are treated by C pretty much the same way as the standard data types. So, the behavior should be consistent, you should be able to return a value of a struct of type struct from the function, let see an example. So, here the user defines a struct point, and then when you run the program, you say can give two integer values - x and y, and I want to create a point struct with subfields x and y which the user has input. So, I have return a function called **make_point**; **make_point** takes two arguments int x and int y, and what it returns is a variable of type struct point. So, this is the return type of the this whole name is basically the return type of the function, the name of the function is **make_point**. Now how do you define it? So, for example, you can define a variable temp of type struct point, and then say **temp.x = x, temp.y = y**, and return temp.

Now, if you forget about this code, particular code, if temp had been an int variable, you would say return temp. If the function was if return type int. Here it is the function is returning struct point, and you would do it exactly in the same way as a function returning int. So, what else can we do with a normal data type, you could for example pass it as a parameter two or function. So, if you have int variables, you can pass

functions taking int arguments. Similarly can you right functions taking struct parameters, and we will see that is can be done, yeah.

(Refer Slide Time: 10:55)

```

Functions with structures as parameters

# include <stdio.h>
# include <math.h>
struct point {
    int x; int y;
};
double norm2( struct point p) {
    return sqrt ( p.x*p.x + p.y*p.y);
}
int main() {
    int x, y;
    struct point pt;
    scanf("%d%d", &x,&y);
    pt = make_point(x,y);
    printf("distance from origin
        is %f ", norm2(p));
    return 0;
}

norm2(struct point p) returns
Euclidean norm of p.

```

Using the math library

- 1: The norm2 or Euclidean norm of point (x,y) is $\sqrt{x^2 + y^2}$
- 2: To make things easier, we are using the math library whose header file is included by line `#include <math.h>`
- 3: The double `sqrt(double)` function is in math library.
- 4: Use `gcc file.c -lm` to compile file.c with the math library.

So, we will take an example that is fairly easy to understand. So, you take a point p and calculate the norm of the point p. So, what is the norm of the point p? The norm of the point (x, y) in the Euclidean plane is simply $\sqrt{x^2 + y^2}$. So, you will just have to calculate a function which does this. For this we will use the math library in C. So, I will say `include<math.h>`, and then I will defined a function norm of struct point p. So, let us call this norm 2 and less ignore why it is call norm 2. So, it is just a function that takes a point p and calculates the `norm(p)`. So, for this what do I do? I will say that returns square root. So, `sqrt` is the square root function provided by `math.h`. So, `return sqrt(p.x * p.x + p.y * p.y)`.

So, the in the main what you would do is user defines user gives an input x and input y. You make a point using the earlier function that we wrote `make_point` x y. So, pt will be a point with x coordinate x, and y coordinate y. Now for that point pt you define `norm2(pt)`. So, `norm2(pt)` here you would pass a point a struct point as a parameter, and the function with calculate the norm of the func of the point and return the norm. So, the way you would passes structure is the same as the way you would pass an int or a character or something.

(Refer slide Time: 12:54)

The slide has a title 'Structures inside structures' at the top. It contains the following C code:

```
struct point {  
    int x; int y;  
};  
  
struct rect {  
    struct point leftbot;  
    struct point righttop;  
};  
struct rect r;
```

To the right of the code, there is a list of three points:

1. Recall, a structure definition defines a type.
2. Once a type is defined, it can be used in the definition of new types.
3. struct point is used to define struct rect. Each struct rect has two instances of struct point.

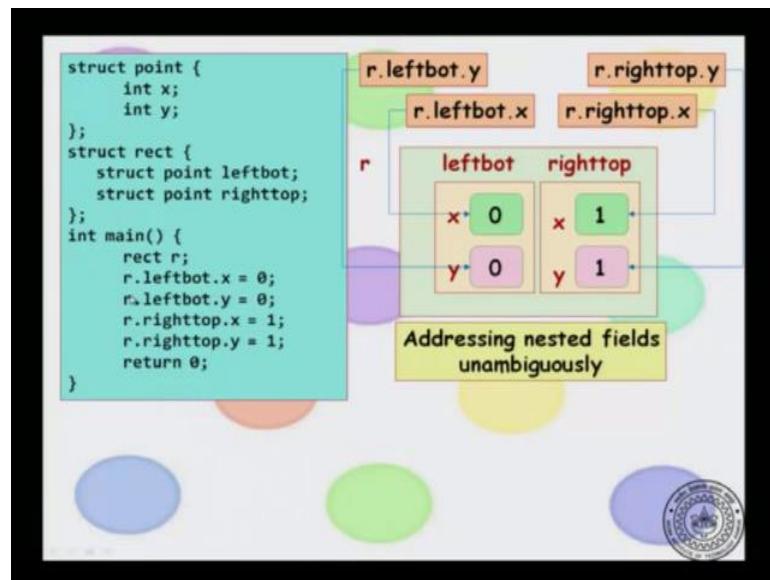
Below the code, there is a diagram labeled 'r' which represents a rectangle. It is divided into four quadrants. The top-left quadrant is labeled 'leftbot' and the top-right quadrant is labeled 'righttop'. Each of these quadrants contains a green square labeled 'x' and a pink square labeled 'y', representing the x and y coordinates of the point.

On the right side of the slide, there is a green box containing the text: 'rect1 is a variable of type struct rect. It has two struct point structures as fields.' Below this, another box asks: 'So how do we refer to the x of leftbot point structure of r?' A small circular logo is visible in the bottom right corner.

Now let us take the game a bit further. You know that I can you know that you can defines structures whose subfields are standard C data types. Now, if user defined data types, user defined structures are of the same category as standard data types then I should be able to defines structures whose internal fields or themselves structures. So, we have seen structures whose internal fields can be basic data types, now will see structures whose internal fields are structures themselves. So, let us look at a very reasonable use case in which this can be occurring. So, suppose you want to extend your geometry package, and you want to define a rectangle. Now a rectangle is define by two points; two diagonally opposite points, let say the left bottom and the right top; these two points define a rectangle So, the left bottom and the right top are themselves points. So, they have some fields which are x coordinate and y coordinate.

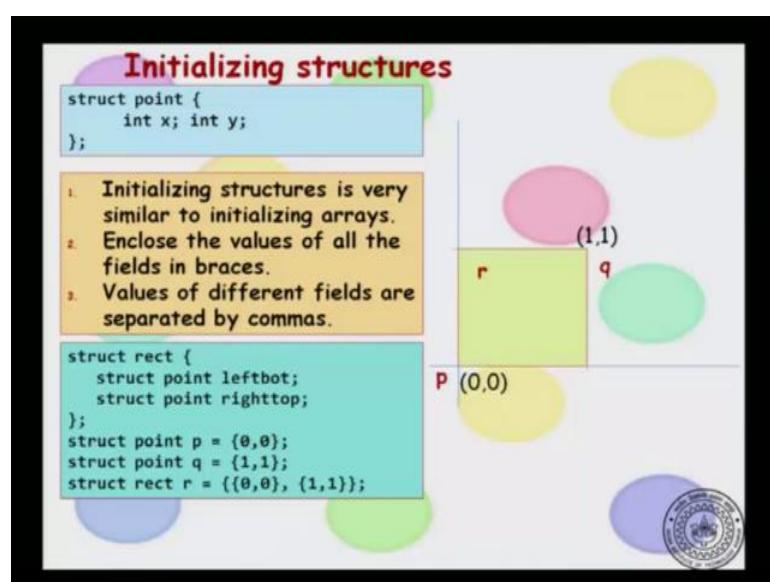
So, this is how you would imagine the picture, r is a rectangle - it has two points; left bottom right top, and this left bot and right of themselves have two subfields x and y, this is the composite picture. Now how do I manipulate this rectangle, how do I say that this rectangles left bottom is the point one one, suppose I want to say something like this.

(Refer Slide Time: 14:39)



So, again you would use the dot notation, so, is a very consistent representation. So, I would say that in the inside the code of name, I would say `struct rect r` and then I would say `r.leftbot . x = 0`, this says that take the left bottom subfield of `r`. Now, since that is a structure `r.leftbot` itself has subfields which is `x` and `y`, its `x` coordinate is assigned to 0. Similarly `r.leftbot . y = 0` and so on. So, I will say that this rectangle's left bottom is 0, and its righttop is 1 1. So, after running this code, this is the state of the memory. So, I will have a rectangle `r`, and its `leftbot.x = 0`, its `leftbot.y = 0`, its `righttop.x = 1`, and its `righttop.y = 1`.

(Refer slide Time: 15:47)



And now we will also see how to initialize structures. So, we know that normal basic data types like int, char, and all that when you declare a variable, you can also initialize, can you initialize a user defined structure in this way. So, the wave define it is similar to the way you define you initialize arrays. So, initializing structures is very similar to initializing arrays, enclose all the values of the fields in braces, and the values are given in the same order that you they are defined in the structure. Suppose you have struct point int x and int y, how would I initialize it I would say **struct point p = {0, 0}**. So, this means that the first field int point, that is x is assigned 0. The second field that is y is assigned 0 as well. Similarly if I say **struct point q = {1, 1}**, it is says **q.x = 1** and **q.y = 1**

Now, you can do the same thing with nested structures. So, this is very nice. So, if I want to define a rectangle. Remember that a rectangle has two 2 fields, which are themselves structures their points. So, if I say **r = {{0,0},{1,1}}**. What happens is that r first field which is the left bottom, it is will get the value **(0, 0)**; that means, that it is the left bottom subfield x will get 0, and the left bottoms y will get 0. Similarly the r is second field is the right top, it will get **(1, 1)**. So, **righttop.x** will be 1, **righttop.y** will be 1 as well. So, this is how you would initialize an a initialize a structure as very similar to initializing an array. The only thing to remember is that the values must be given in the same order then they are declared in the type declaration

(Refer slide Time: 18:10)

Assigning structure variables

```
int main() {
    rect r,s;
    r.leftbot.x = 0;
    r.leftbot.y = 0;
    r.righttop.x = 1;
    r.righttop.y = 1;
    s=r;
    return 0;
}
```

1. We can assign a structure variable to another structure variable.
2. The statement **s=r;** does this

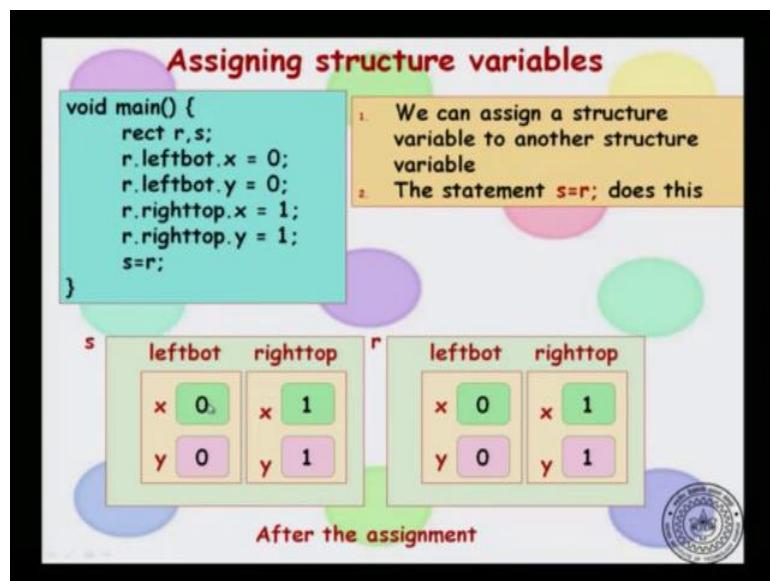
s	leftbot	righttop
	x y	x y

r	leftbot	righttop
	x y	x y

Before the assignment

Now we know that variables can be assigned to other variables. So, natural question is can we assign to struct variables, and the answer is yes, suppose you have a rectangle `r` whose left bottom is `0, 0`, either initialize it or assign the values, and its righttop is `1, 1`. So, I define another variable `s` which is also a rectangle, and if I say `s = r`. Let see what happens? So, at before the assignment `r` is as follows. So, you have `x 0 0 x` `xy 0 0` and `x y 1 1` left bottom on the right top, and `s` is uninitialized it has just been declared, but no value has been assigned it.

(Refer Slide Time: 19:06)



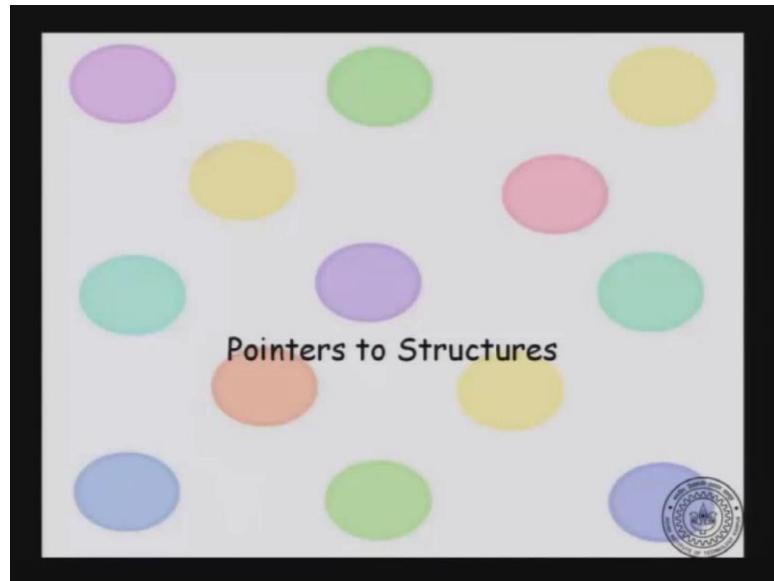
So, this is the state before the assignment. When you do `s = r` it is very nice, what it does is `s` is `leftbot.x` will be assigned 0, `s` is `leftbot.y` will be assigned 0, `s` is `righttop.x` will be assigned 1, and `righttop.y` will be assigned 1. So, what happens is it goes into the structure `r` and copies it entirely in its full depth into `s`. So, it is not just that a left bottom and right top are copied, its internal fields are also copied into `s`.

Introduction to Programming in C

Department of Computer Science and Engineering

This lecture will continue our discussion of structures. So, if you remember from earlier lecture we were saying that user defined structures or user defined types which will be treated by C, in pretty much the same way as the basic data types.

(Refer Slide Time: 00:23)



So, we will continue on that theme and look at topic on pointers to structures, we know that for a basic data type you can define a pointer to that type, I can declare `int *` or `char *` and so on. Similarly does it make sense to talk about struct points `*` for instance.

(Refer Slide Time: 00:42)

Passing structures..?

```
struct rect { struct point leftbot;
    struct point righttop;
};

int area(struct rect r) {
    return
        (r.righttop.x - r.leftbot.x) *
        (r.righttop.y - r.leftbot.y);
}

int main() {
    struct rect r = {{0,0}, {1,1}};
    area(r);
    return 0;
}
```

We can pass structures as parameters, and return structures from functions, like the basic types int, char, double etc..

But is it efficient to pass structures or to return structures?

Usually NO. E.g., to pass struct rect as parameter, 4 integers are copied. This is expensive.

So what should be done to pass structures to functions?

The slide features a diagram of a rectangle structure 'r' with fields 'leftbot' and 'righttop', each containing an 'x' and a 'y' coordinate. A circular seal of the Indian Institute of Technology (IIT) Kharagpur is visible in the bottom right corner.

Let us look at an example where it makes sense. So, let us go back to the example of struct point and struct rectangle from the earlier lecture. So, let us say that **struct rect** has two points, which are **leftbot** and **righttop** and both of them are struct point. Now, we want to calculate the area of a rectangle. So, you have a rectangle **r** which is initialized to **{0, 0}**, **{1, 1}**. So, leftbottom will be **{0, 0}** and righttop will be **{1, 1}** and I want to compute its area.

Now, the area function is defined as follows, it is a function that returns an integer, it takes as parameter a struct rectangle and it does the following, it does **(r.righttop.x - r.leftbot.x) * (r.righttop.y - r.leftbot.y)**. So, it does that particular function and it returns it. So, we know that we can pass structures as parameters and also return structures from functions, but is it efficient to pass structures or to return structures? And the answer is usually no, because copying a structure involves copying all its fields.

So, generally when you call a function the value has to be copied onto the function's scope and we have seen this when discussing functions. So, when you pass a structure the entire structure has to be copied. Similarly, when you return a structure the entire structure that was created inside the function has to be copied back, this is usually an expensive operation. So, one way to get around it is to pass a pointer to the structure. So, what should be done to pass a structure to a function in an efficient manner.

(Refer Slide Time: 02:44)

The diagram illustrates the concept of passing structures by pointer. On the left, a C code snippet shows a struct rect definition and a main() function that calculates the area of a rectangle. A pointer *pr is used to pass the address of the rectangle r to the area() function. On the right, a large green box contains the title "Passing structures..?" and three callout boxes: 1) "Instead of passing structures, pass pointers to structures." 2) "area() uses a pointer to struct rect pr as a parameter, instead of struct rect itself." 3) "Now only one pointer is passed instead of a large struct." Below the code and the green box is a diagram showing a variable r pointing to a large green rectangle labeled "leftbot" and "righttop". Inside this rectangle are two smaller green rectangles labeled "x" and "y", representing points. The entire diagram is set against a background featuring the Indian Institute of Technology logo.

```
struct rect { struct point leftbot;
    struct point righttop;
};

int area(struct rect *pr) {
    return
        ((*pr).righttop.x - (*pr).leftbot.x) *
        ((*pr).righttop.y - (*pr).leftbot.y);
}

int main() {
    struct rect r = {{0,0}, {1,1}};
    area( &r );
    return 0;
}
```

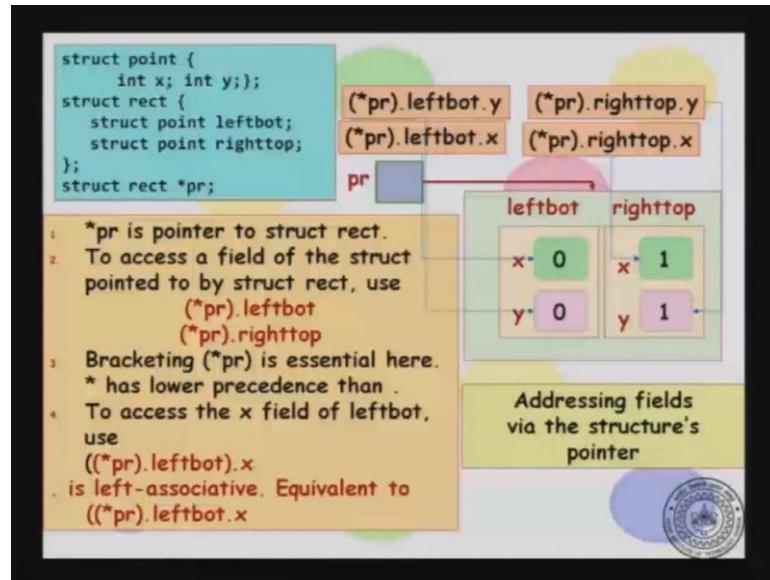
Now, one way to do it would be to define, what is known as a pointer to a structure, how do you define a pointer to a structure, you define it pretty much the same way as pointer to any other data type had this been an integer, you would declare `int *pr`. So, if you want to declare a variable which is a pointer to a structure, you would define `struct rect *pr`. So, `pr` is a pointer to struct rectangle.

So now, how would you pass the argument, you would say address of the rectangle `r`. So, you would say `area` and the parameter is address of `r`. Now, inside the function earlier you remember it was `*pr.righttop.x`, now `pr` in this case is just a pointer to a rectangle. So, we have to access the variable in that address, how do you do it using the `*` operator, this is the same as addressing any basic data type, you would say that `*pr` would be the variable in that location.

So, in this case it would be `*pr`, `*pr` would be a struct rect and that `struct rect .righttop.x - (*pr).leftbot.x` and so on. So, the lesson here is that instead of passing structures, you pass pointers to structures and now whatever be the size of the structure. So, you have a struct rectangle which inside has two points and so on. So, you may want to pass a very large structure and copying that will take a long time. But, instead what you do is, you pass just a pointer, now regardless of the size of the structure only one pointer is copied.

So, this same principle goes for returning structures as well, when on a structure from the function what you would do is to return a pointer to that structure. Of course, now the structure has to be allocated on the heap rather than the stack.

(Refer Slide Time: 04:55)



Now, let see how the memory depiction of this looks like. So, `pr` when you define `struct rect *pr`, `pr` is a pointer to a structure of type rectangle and then what is happening here is there, if you want to access the `y` coordinate of the left bottom, you would say `((*pr).leftbot).y`. So, it will come to the `leftbot` field of rectangle and pick its some field `y`. So, `((*pr).leftbot).y` would refer to this location in the memory. Similarly, `((*pr).righttop).y` would be this location in the memory and so on. So, you can address the sub fields of address the fields of a structure using pointer.

(Refer Slide Time: 05:54)

The slide has a title 'Structure Pointers' at the top right. On the left, there is a code snippet:struct rect {
 struct point leftbot;
 struct point righttop;
};
struct rect *pr;A list of points follows:

1. Pointers to structures are used so frequently that a shorthand notation is provided.
2. To access a field of the struct pointed to by struct rect, use `(*pr).leftbot` or `pr->leftbot`
3. `->` is one operator. To access the `x` field of `leftbot`, use `(pr->leftbot).x`
4. `->` and `.` have same precedence and are left-associative.
Equivalent to `pr->leftbot.x`

To the right, there is a diagram of a wooden ladder leaning against a wall. Next to it is the text 'Highest precedence: [] (array subscript) function call () -> and . Left-right'. Below the ladder is the text 'Operator Precedence'.

There is one syntactic convenience that C provides you, because addressing structures is a fairly common occurrence. And because in this case by associativity, you cannot omit the parenthesis, you cannot say `*pr` without parenthesis, because it means `pr.leftbot` and `*` of that. So, that is not what you want, you want to say that take the structure in the location `*pr` and take it is `leftbot`. So, in this case by it is associativity and precedence rules, you have to include these parenthesis, you cannot omit them and this is inconvenient.

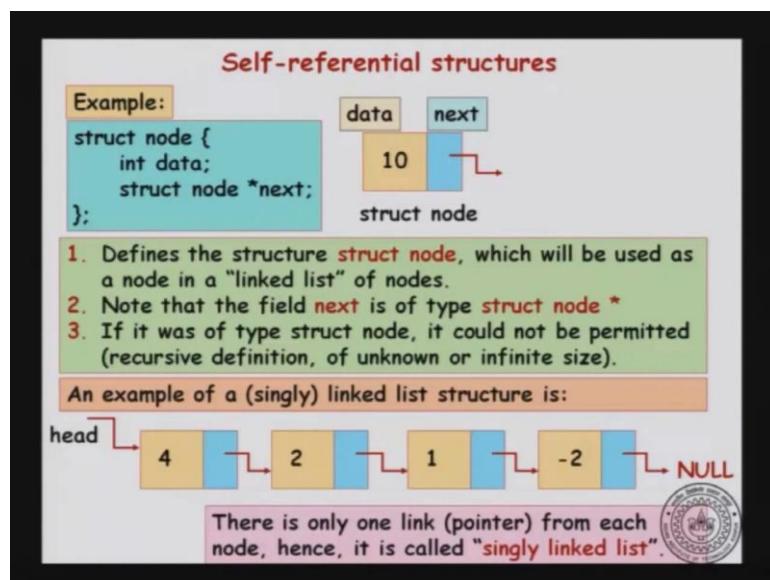
Therefore, C provides a syntactic convenience, which is pr arrow, arrow is actually two characters it is a `-` and a `>`. So, `pr -> leftbot` is the same as within bracket `(*pr).left bot`. So, there are two ways to address the fields of the location pointed to by `pr`. So, `pr` is a pointer to a struct rect you can access its `leftbot` by saying `(*pr).leftbot` or `pr -> leftbot`. Notice that, these two characters `-` and `>` is just a single operator and they have... So, `- >`, that is the `- >` operator and `.` have the same precedence and both are left associated.

Introduction to Programming in C

Department of Computer Science and Engineering

Once we know structures and pointers to structures, we can introduce some very important data structure called link list. So, we will first see what link list are, how to operate on them, and then argue why link list are useful.

(Refer Slide Time: 00:20)



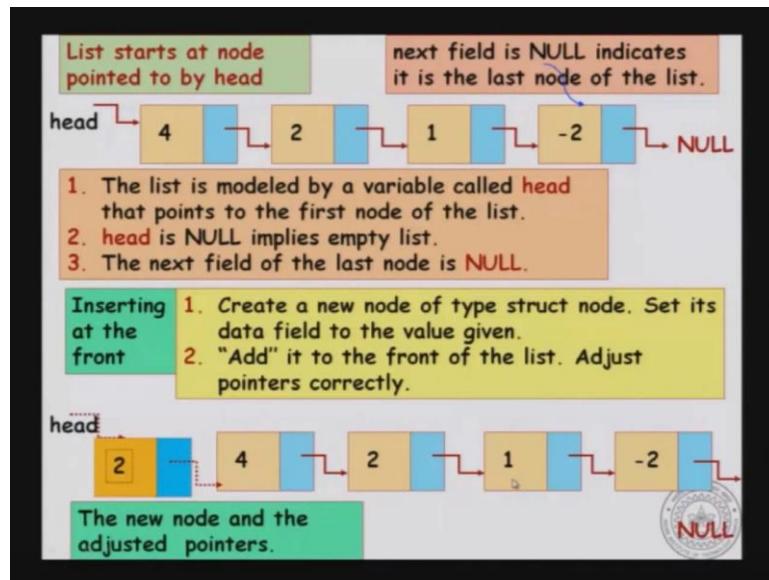
So, let us just introduce this notion called self referential structures. So, we are defining a struct node that has two fields - one is an int data, and the next field is the surprising one, it is a pointer two type struct node. So, this data structure, this c structure is called a self referential structure because internally there is a pointer to an object of the same type. So, in that sense it refers to some other object of the same type. So, it is called self referential.

So, an example would be like this where the data field has 10, and the next field points to something else which should be a struct node So, then field next is of type struct node; now, there is a subtle point to be emphasized; instead of **struct node *** had I return struct node next then this is not allowed, because the definition of struct node has an internal struct node inside it, so which essentially has infinite size. So, we are cleverly avoiding that by including just a pointer to the next node.

So, using this structure we can define what is known as a singly link list. So, an example of a singly link list structure would be where we have pointer which we will call the head

of the list, head points to the first struct which is 4, which has data 4; and it is linked to another struct which has data 2, that is linked to another struct which has data 1, and so on. The last struct in the list will be linked to null. So, there is only one link from each node, hence the name singly link list.

(Refer Slide Time: 02:17)



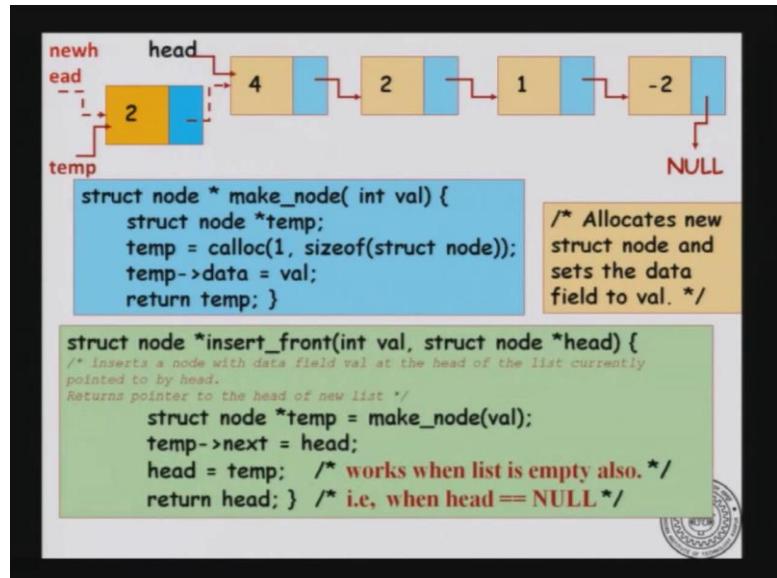
So, the fact that the next field is null indicates that that is the last node in a link list. And a link list is always identified by its head which is the pointer to the first node in the link list. Once we reach the first node we can travel the list by using just the next links. So, once we have a link to 4, we can always say, 4, 2, 1, -2, and so on. So, the list is made modeled by the variable called the head that points to the first node in the list. And if the head is null then that means the list is empty. And then you have a bunch nodes; and when once we reach a node with the next field null then that is the last node in the link list.

Now, let us look at certain simple operations on singly link list. Suppose you want to insert a node at the front of the list, so we have a list, 4, 2, 1, -2, and we want to insert something else in the beginning. So, what you do is you create a new node of type struct node and set its data field to whatever number that you want to store. Now, add it to the front of the list; we will see how this can be done.

So, suppose that the head is now pointing to 4, and the list is 4, 2, 1, -2, and you want to add a new node, so 2 is the node, the data field is 2, how do you add? You do two operations – first you say that 2 is next is the first node in the old list. So, that would

insert 2 here. And then now the list has changed because the first element is now 2, so head moves to 2; head was previously 4 and head now moves to 2. So, this is abstractly how you would insert a node at the beginning of a list.

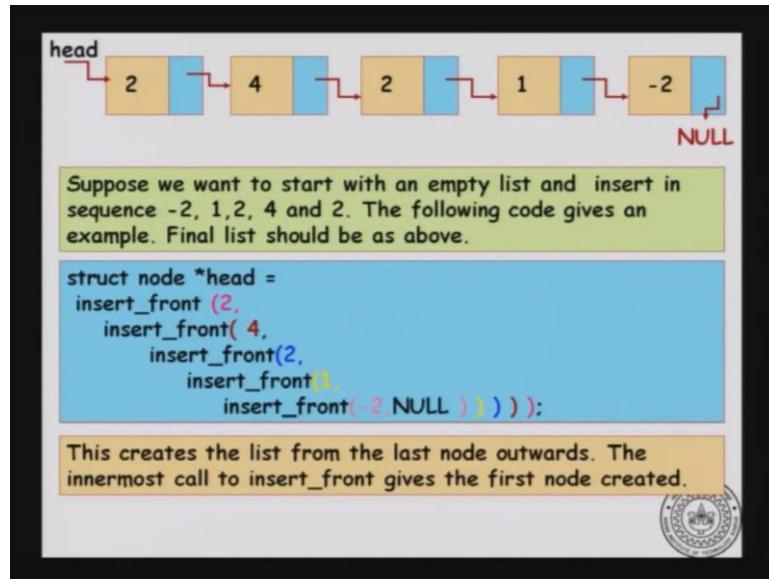
(Refer Slide Time: 04:20)



So, now, let us try to code it and see. So, first we need a code small function to make a node with the given data. So, we will say, `struct node * make_node` in 12. Now, we will create a pointer `struct node * temp`, then use one of the malloc function called `malloc`; so 1, size of `struct node`. So, this will allocate memory enough to create one node. Now, this, that memory its data field will be set to `val` which is what we are given as argument to the function and then you return the node.

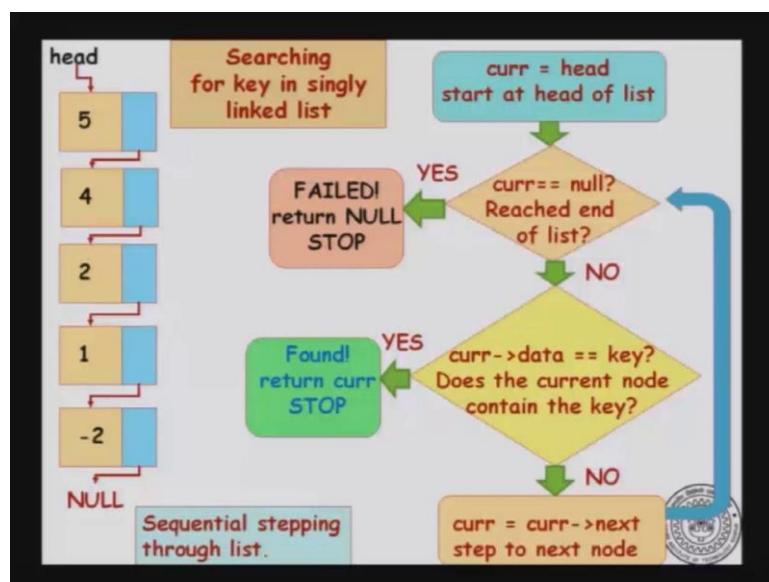
So, we have created a node; and how do you insert in the front? Once you have, once you receive an value to be inserted at the beginning of a list. So, the list is identified by the head. So, we have to create a node which contains the value and insert it at the beginning of this list. So, what we do is we first create a node with the value using `make_node` function; now `temp next` is set to `head`; so this link is activated. So, 2's next will be 4. So, that is the first step. The second step is that the `head` now has to move to 2 because the first element in the new list is 2, not 4. So, I will just say, `head = temp`, and return `head` which is the head of the new list.

(Refer Slide Time: 06:04)



And now, you can call this function multiple time. Suppose you want to start with an empty list and insert -2, then insert -1, then insert 2, then insert 4, then insert 2, you can call these functions one after the other. So, I can just say `struct node *head = insert_front(2, insert_front(4, insert_front(2, insert_front(1, insert_front(-2, NULL)))))`. So, this is the function; this is the sequence of functions and this is the list that you will end up with. So, once you have the function to insert at the beginning of a list, you can use that function multiple times to build up the list.

(Refer Slide Time: 06:46)

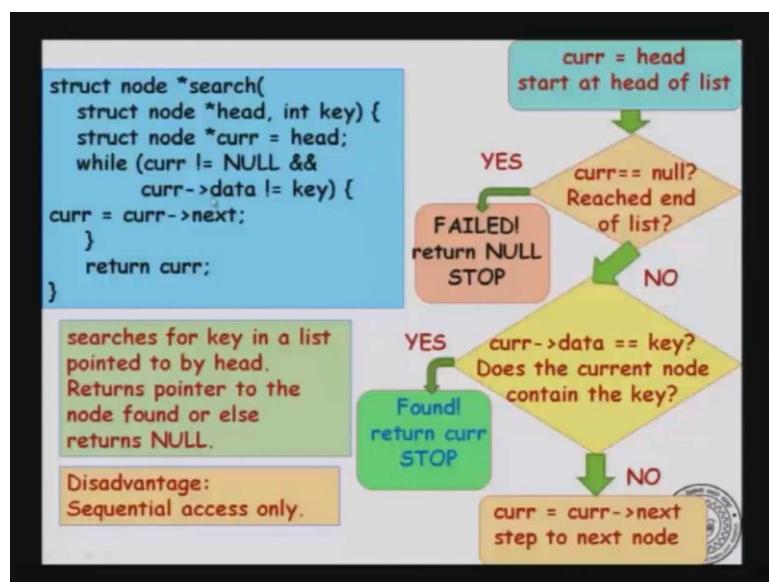


Now, let us look at some useful functions to be done on link list. So, once you have a link list it will be good if you can search the link list to see whether an element is present or not. So, we will look at a very simple algorithm. So, we want to search for a particular key that is an element in a singly link list. So, how do you do it? Abstractly, what you do is you start with the head, see whether the data field in the head node is the key that you want.

If it is, then you are done and say, that I have found it; if it is not what you do is, you go to the next node through the next link and then search; ok, you said the data field of the next node if it is you are done; if it is not, you search and follow the next node. You follow this procedure until you reach the last node. Suppose, you have not found the key even in the last node, and you follow the next link and it is null; so once you reach null then you know that you have reached the end of the list. So, you cannot take null next that will cause your code to crash. So, once you know that your node is null, you can end the search and then say that the key is not present in the list.

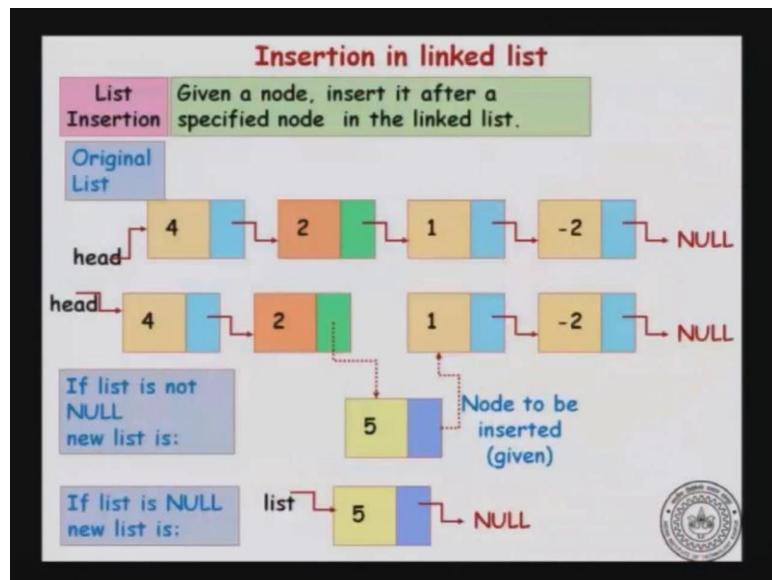
Here is a flowchart corresponding to that; you start with `*curr = head`; now, is `curr` null, if the `curr` is null then that means you have reached the end of the list and you have not found the key. So, if the `curr != NULL` then there is data still to be searched. So, you see whether `curr` data is equal to key; if it is yes then you have found the key, otherwise you follow the next node link to go to the next node in the link list; and again, repeat the procedure, and you can code this in a straight forward manner.

(Refer Slide Time: 08:45)



So, I will write `struct node *search`; I need the head of the list and I need the key. You start with `*curr = head`. If `curr !=NULL && curr ->data != key`, you follow the next link, `curr = curr->next`, and you repeat the procedure. So, when you exit the list either `curr` will be null or `curr` data will be key. So, what is the condition when you reach the return? So, if the key is absent then you are returning null, if the key is present you are returning the pointer to the node, pointer to the first node that contains the key. So, convince yourself that the code works.

(Refer Slide Time: 09:30)

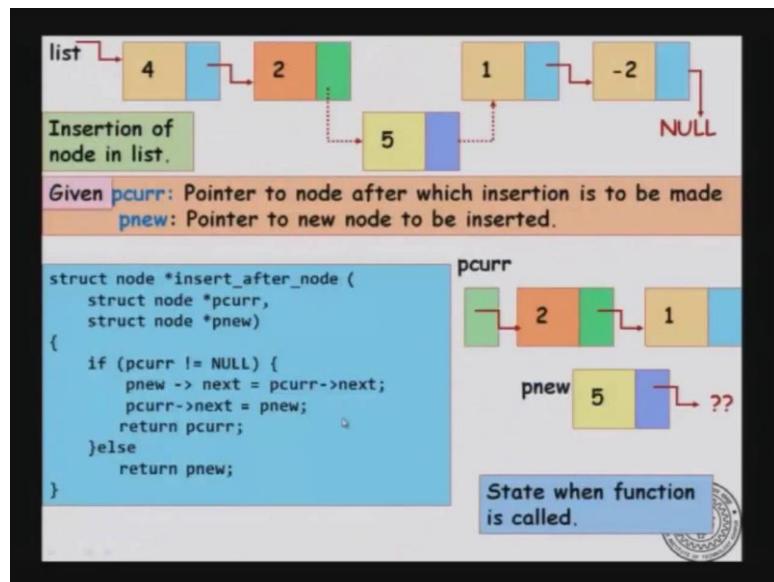


Now, let us look at slightly more involved operations. We have seen insert at the head of the list and that was fairly simple, now suppose you want to insert into the middle of the list; you do not want to insert right at the front, you want to insert somewhere in the middle. Now, there are two cases here. If the list is null, that is the easy case; if the list is null then insert in the middle is essentially insert at the front. So, we already have seen the code for that.

Now, if the list is not null, now it is a new algorithm. So, let us look at an example. So, suppose the list is 4, 2, 1, -2, and I want to insert a node 5 after node 2. So, how do I do it? 2's next link was 1. And what we need to do in this case is I want to say that I have to insert this node 5; 5's next node will be 1; that is, so think about this as it link in a chain. So, you need to disconnect this link say that 2 is now connected to 5, and 5 is then connected to 1.

Now, the only thing to be noted is that the links have to be detached in a particular sequence. So, first I need to say that 5's next is 1, and then I need to say that 2's next is 5. So, convince yourself that the opposite sequence where I say that 2's next is 5, now your code will have no way to proceed because you have lost the how to traverse from 2 to 1. If you say that 2's next is 5, then 5 has no way to know what was the original next node of 2. So, you have to do it in a particular sequence - 5's next is 1, and then 2's next is 5. We will see this code in a minute.

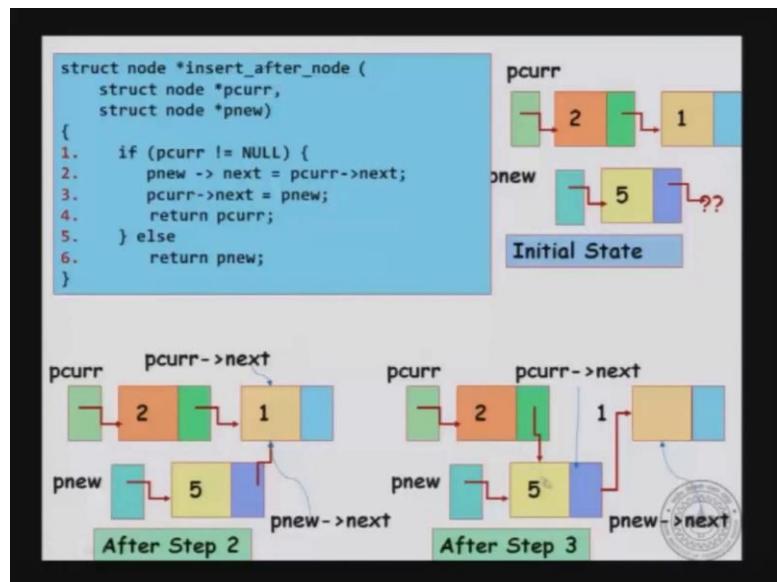
(Refer Slide Time: 11:31)



So, what we do is we want to insert after a node, so **pcurr** is the node after which we have to insert; and **pnew** is pointing to the new node that we have to insert. If **pcurr** is null, then essentially the list is basically **pnew**; this is a case that we have seen before. If **pcurr** is not null, that means the list is not empty, then what you do is, the new node's next node is, **pcurr**'s next node. So, 5's next node is the old 2's next node which is 1. So, 5's next will be set to 1.

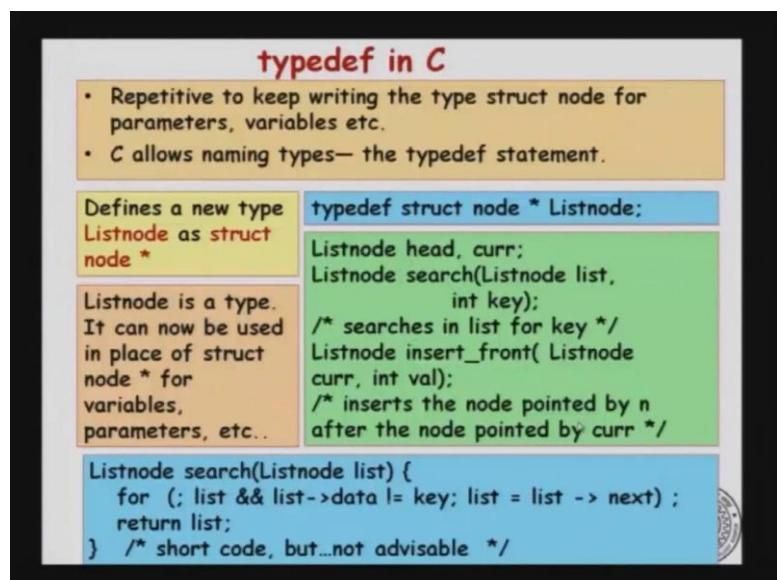
And after that I will say, **pcurr->next = pnew**; then I will say that **pcurr** which is 2's next will be set to 5. So, convince yourself that if I swap these 2 lines, if I swap the lines order then the code will not work. So, see this for 5 minutes and you convince yourself that that will not work.

(Refer Slide Time: 12:40)



So, let us just see how this works. So, initially, let us say that I want to insert after 2, and new is the new node. So, initial state is something like this; the 5s next node is pointing to something, maybe some arbitrary location. Now, after line two that is `pnew -> next = pcurr -> next`, this is the state of pointers. Please look very carefully. So, 5s next will point to 1, and then this point 2s next is also pointing to 1. So, there are 2 nodes whose next is 1 which is fine because we have not completely inserted 5 into the list now. Now, at this point, I will just detach 2s next and make it point to 5. So, there we go. So, after step 3 you have essentially inserted 5 into the list.

(Refer Slide Time: 13:40)



Now, let us look at some syntactic conveniences that c provides you. So, repetitively you are typing the struct node, and things like that is it is just too much to type, and c allows you to define short names for types. So, if I want to say, like **struct node *** I want to use the name list node. So, I will just say struct; instead of **struct node *head**, I will just say list node head. So, it is a shorter way to do it.

So, how do I write this? This is using what is known as a **typedef** keyword in c. So, if I say, **typedef struct node * list node**, what it means is that list node is another name for the long type **struct node ***. So, this is something that you may use if you want to. It is not something that is that you should use, but it is just convenient. So, if I say, list node head, this is the same as saying **struct node *head, curr**.

(Refer Slide Time: 14:50)

Why linked lists

The same numbers can be represented in an array. So, where is the advantage?

1. Insertion and deletion are inexpensive, only a few "pointer changes".
2. To insert an element at position k in array:
 - a) create space in position k by shifting elements in positions k or higher one to the right.
3. To delete element in position k in array:
 - a) compact array by shifting elements in positions k or higher one to the left.

Disadvantages of Linked List

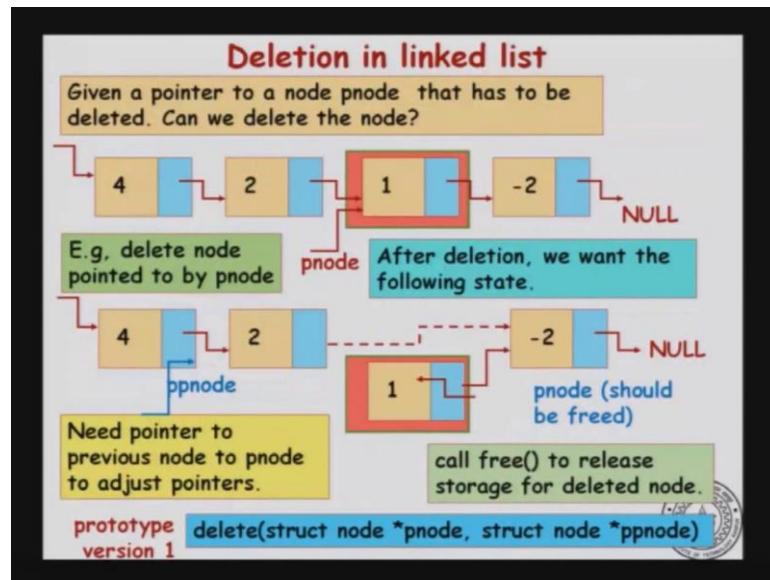
Direct access to kth position in a list is expensive (time proportional to k) but is fast in arrays (constant time).

Now, let us see why link lists are important. So, first of all it is one of the first non trivial data structures that you learn. In earlier days when c had only fixed size arrays, link list was important when you needed variable size storage. Nowadays, c has variable size storage, so you can, in arrays, so that is not so important any more. But, here is one important thing, one difference between link list and arrays which are very important.

Like, insertion and deletion in link list are fairly cheap. In the case of an array, so if you want to insert an element at position k in an array, you have to copy all elements from k to $n - 1$, to the last element in the array; move each of them backwards, makes space for it, and then insert the k th array. So, this involves, in the worst case, it involves moving all the elements of the array by one element, one position each.

Similarly, for delete; suppose, you want to delete an element from array, then what you have to do is, you have to take the remaining elements of the array and move them one position to the left. So, this will involve moving n elements in the array if array has n elements in the worst case. Whereas, note that in the link list case, to insert or delete any element, a new node, whether at the beginning or in between, it just takes one operation; the other elements need not be manipulated.

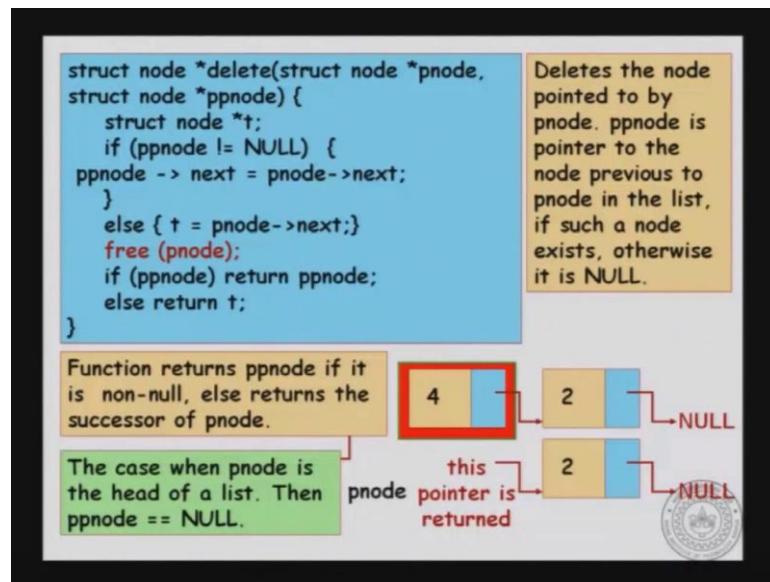
(Refer Slide Time: 16:40)



So, let us just quickly see how to delete a particular node in a link list. So, we will just say that 4, 2, 1, -2, is the link list, and I want to delete this particular node, is that possible? So, I cannot straight forward delete this node because the previous nodes next element should point to this guys next element. So, if I want to delete 1, what do I have to do? I have to say that 2s next node should be -2.

But, in a singly link list there is no way to go back; from 1 you cannot easily get to 2. So, this is slightly; so deletion requires slightly some more information. So, if I can delete a node, if I also have a handle a pointer to its previous node, then it is very easy to say that 2s next node will be -2.

(Refer Slide Time: 17:40)



And that is what we will do. So, we will say that let us have a delete function, *t* node is the node that I want to delete, and *ppnode* is its previous node. And what I will do is, if there is a previous node I will say, previous node's next is the current node's next. So, *2*'s next link will go to *-2*. If there is no previous node then I will say that, *t* equal to at the current node's next; and then once that is done, you delete the current node, *pnode*. So, this is how you would delete an element from the link list.

(Refer Slide Time: 18:25)

Linked Lists: the pros and the cons		
list	Singly Linked List	Arrays
Arbitrary Searching.	sequential search (linear)	sequential search (linear)
Sorted structure.	Still sequential search. Cannot take advantage.	Binary search possible (logarithmic)
Insert key after a given point in structure.	Very quick (constant number of operations).	Shift all array elements at insertion index and later one position to right. Make room, then insert. (linear time)

So, just recap searching in a link list will take order n time, in the case of a link list, that is you have to search all the elements in the worst case which is the same in an array. Now, suppose you sort an array, you have faster search techniques available; you can do binary search in an array. Unfortunately, in a link list, even if you sort the link list, there is no way to do a binary search in the link list; why is this? Because you cannot reach the middle element of the link list in one shot.

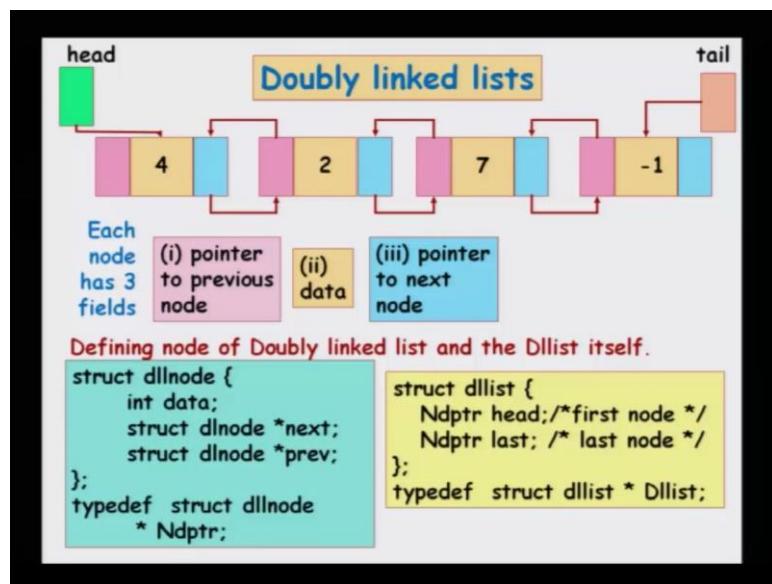
In an array, you can just say a mid, and it will go to the... So suppose you say that $\text{mid} = 0 + \frac{n-1}{2}$, you can go to the middle element of the array. But, there is no way to do that in a link list, you have to go one after the other. So, sorting does not help in searching when you are looking at singly link list. But on the other hand, insertion and deletion are very quick in a link list, whereas they are very slow in an array.

Introduction to Programming in C

Department of Computer Science and Engineering

In this lecture, we will see slightly more advanced data type, then a singly link list. We will briefly go over one or two functions to manipulate the data structure. The principle of manipulating the data structure for the other operations is similar.

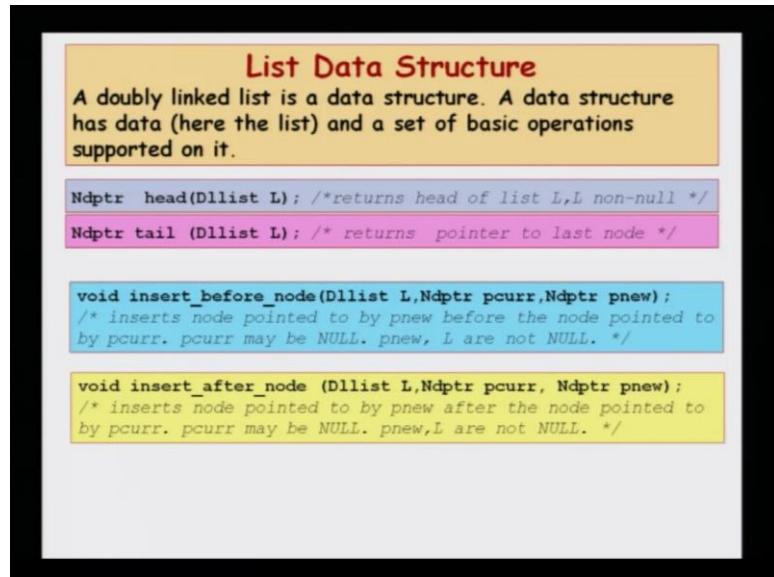
(Refer Slide Time: 00:18)



So, in the case of a singly link list, we have seen that every node has one link to its next neighbour, and we have seen this problem in a singly link list that if you are at a current node in a link list, you can always go forward, but there is no way to go back. There is one the only way took get to its previous node is to start all over again from the beginning of the list and traverse until you reach a list traverse, but traverse until you reach the previous node. So, we can easily remedy this by thinking of a data structure, a slightly more involve data structure where every node has two links; one... So, look at this node 2. So, it has two links; one is two it is neighbour successive neighbour. So, it is it is next node, there is another link which goes back to it is previous neighbour. So, in this data structure there are 2 links per node, therefore it is known as a doubly link list. And this list, obviously you can go from a current node, you can go forward or backward, so easily...

So, now the variation is this in each node has three fields; one is a pointer to the previous node, the second is the data in the node, and third is the pointer to the next node. So, how will the definition look like, it will say something like struct **dllnode**, doubly linked list node, it will have one filed which is data in data let us say, and then two nodes - struct **dllnode** next, and struct **dllnode** previous. So, one to the go to next node and another to go to the previous node. Now we will need two pointers typically for a doubly link list. One is the pointer to the beginning of list which is usually called the head, and then another to the end of list which is usually called tail. So, I will use a type def in order to the short term the name, I will just say **typedef struct dllnode * Ndptr**. And then I will say that the list has two node pointers; **Ndptr heads**, and node pointers last. So, doubly link list. Each node in the doubly link list has two list; one to its previous node, and another to is next node. And the list itself has two pointers; one to the beginning of the list - call the head, and another to the end of the list - call the tail.

(Refer Slide Time: 03:00)



So, now a doubly link list is another data structure notice that we have seen two or three data structure so far arrays are one, which see already provides. We have already seen singly link list, now we have seen a third link list - third data structure which is a doubly link list. Now data structure has data and a bunch of operation defined on it. So, let us look at a typical operations that can be defined on a doubly link list, and we will go over the implementation of two or three them. So, **Ndptr head**. So, this is a function that should return the head of the list. Similarly node point of tail, they should return the tail

of the list. Insert before, so this is like a insert before node in the case of a singly link list. So, here we are given a current node and we have to insert before a current node in the doubly link list.

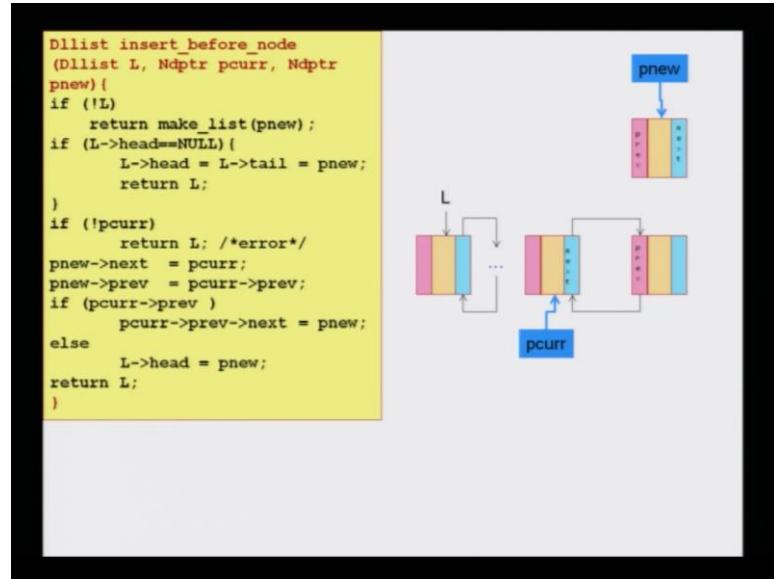
Notice thus this was difficult in singly link list, because there was no way to go from a current node to a previous node. We could always go to the next node. So, if I say that here is node and insert before that node in a singly link list is a difficult. You need some extra information, but in a doubly link list you have the current node and you can use the previous link in order to go before that. **Insert_after_node** also can be done this could also be done in a singly link list.

(Refer Slide Time: 04:41)



And then you can think of several other common like, you can think of a make node, you can think of a **make_list** with us single with a single node pointer two by **pnew**. You can make an empty list, you can check whether I given list is empty, you can write functions to copy a doubly link list to an new doubly link list, you can concatenate to doubly link list. You can do a deep concat, we will see this in a features slide, you can append to link list, and so on. Similarly we can have insert, since we have insert functions we can also have delete functions, you can delete a particular node, you can extract a node in the sense that. So, delete would take out a node and free the memory allocated to the node, extract would just take out the node from the link list, but you retain the node, you can deleted entire list, and so on.

(Refer Slide Time: 05:52)



So, let us look at a couple of these functions; other functions can be return in similar manner. So, suppose let us take insert before load. This was a function that was not easy with the singly link list. So, I am given a link list L , and given a current node p_{curr} , and a new node to insert before the current node. So, what are the things to check? If the list is empty then insert before the current node just means that you create a new node, and return the new list. Now, if the head of the list is null, then you just say that now the new list contains only one load, L head will point new, L tail will point a new. So, if the list itself was null, then what you do is you create a new node, now the new list contains only one elements. So, the head will point to that and the tail will also point you that, and you return that. Now you come to the non trivial case, suppose there is a list; and the list has some elements. So, if p_{curr} is not equal to null then what to you do is, sorry if p_{curr} equal to null then you return L , this is an error. If p_{curr} is not equal to null then what you do is the following.

So, now you have to insert p_{new} in to the list. So, how do you do this? So, we say that the new nodes next will be... So, we are trying to insert p_{new} before p_{curr} . So, the new nodes next will. So, the new nodes next will be p_{curr} ; p_{curr} previous will go to p_{new} . And so the p_{new} next will go to p_{curr} , and p_{curr} previous will go to p_{new} . Similarly we have to say that the previous node, the node before p_{curr} it has to point to p_{new} . So, p_{curr} previous that nodes next will good point to p_{new} , and then you return the new list, so this can be done by looking at pointers and handling pointers carefully.

(Refer Slide Time: 08:23)

The diagram shows three cases for deleting a node in a Doubly Linked List (Dllist).
Case 1: Deleting the head node (p). The head pointer is updated to point to the next node, and the previous node's next pointer is set to NULL.
Case 2: Deleting the tail node (p). The tail pointer is updated to point to the previous node, and the next node's previous pointer is set to NULL.
Case 3: Deleting a middle node (p). The node's previous and next pointers are adjusted to skip over it, and the free(p) function is called to release the memory.

```
void delete_list_hdr(Dllist L)
{ if (L) free(L); }

file list.c

void delete_node(Dllist L, Nodptr p)
{
    if (IsEmpty(L) || !p)
        return;
    if (L->head == p) {
        L->head = p->next;
        L->head->prev = NULL;
    }
    if (L->tail == p) {
        L->tail = p->prev;
        L->tail->next=NULL;
    }
    if (p->next)
        p->next->prev = p->prev;
    if (p->prev)
        p->prev->next = p->next;
    free (p);
}
```

So, now let us see how to delete a particular node in a list. So, if you have to delete header of the list, then if there is a list you just delete the header, and you just free the entire list. Now if you have to delete a particular node in the middle of a list, what do you do? So, let us look at the various cases. So, in case one the node that you want to delete is the head of the list. So, in this case suppose you want to delete p, what would you do? You would make head point to the next element and **free(p)**.

(Refer Slide Time: 09:10)

The diagram shows three cases for deleting a node in a Doubly Linked List (Dllist).
Case 1: Deleting the head node (p). The head pointer is updated to point to the next node, and the previous node's next pointer is set to NULL.
Case 2: Deleting the tail node (p). The tail pointer is updated to point to the previous node, and the next node's previous pointer is set to NULL.
Case 3: Deleting a middle node (p). The node's previous and next pointers are adjusted to skip over it, and the free(p) function is called to release the memory.

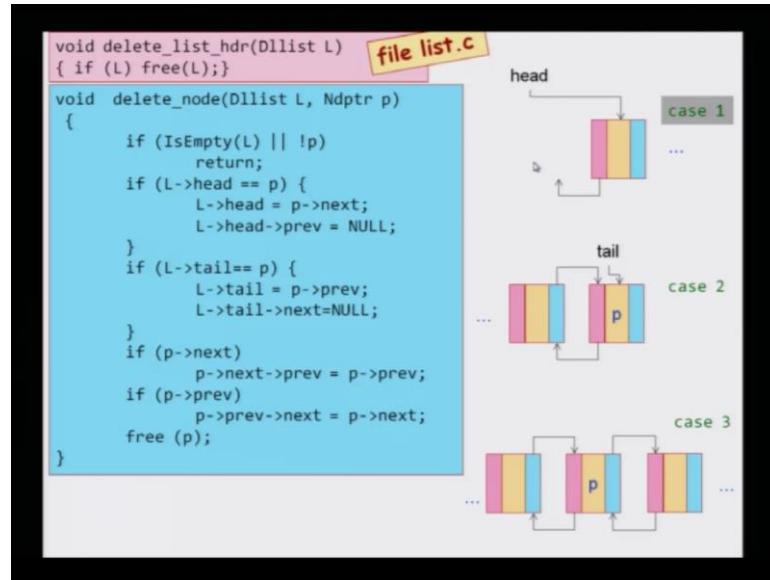
```
void delete_list_hdr(Dllist L)
{ if (L) free(L); }

file list.c

void delete_node(Dllist L, Nodptr p)
{
    if (IsEmpty(L) || !p)
        return;
    if (L->head == p) {
        L->head = p->next;
        L->head->prev = NULL;
    }
    if (L->tail == p) {
        L->tail = p->prev;
        L->tail->next=NULL;
    }
    if (p->next)
        p->next->prev = p->prev;
    if (p->prev)
        p->prev->next = p->next;
    free (p);
}
```

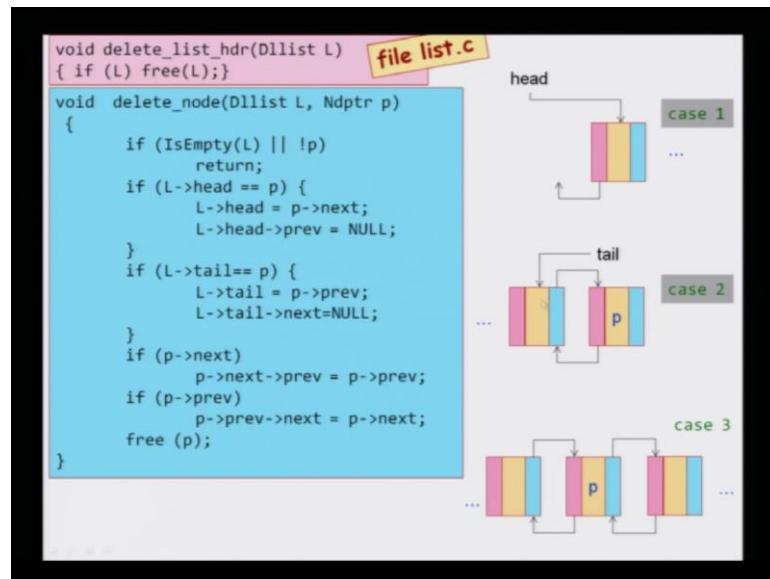
So, head will be made the point to p next. So, this is the line here, L head will go to p next. Now this guy's previous will be set to null, because we are going to delete this node.

(Refer Slide Time: 09:26)



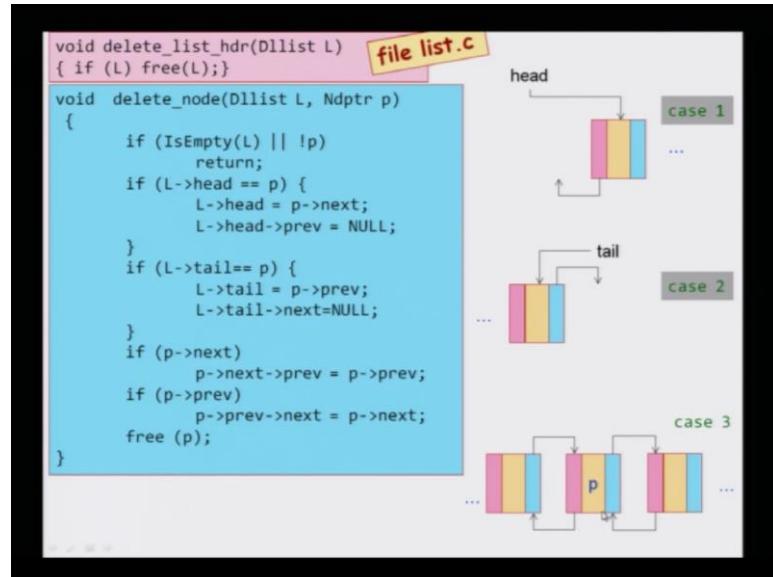
So, this guy's previous will be set to null. So, now it does not point anything and then you win **free(p)**. So, this is the first case, where p the node to be deleted was the head of the list. Similarly, if you want to delete the tail of the list. So, now what should you do here, the tail should go to p previous. So, in case two when p is the end of the list that we want to delete.

(Refer Slide Time: 09:52)



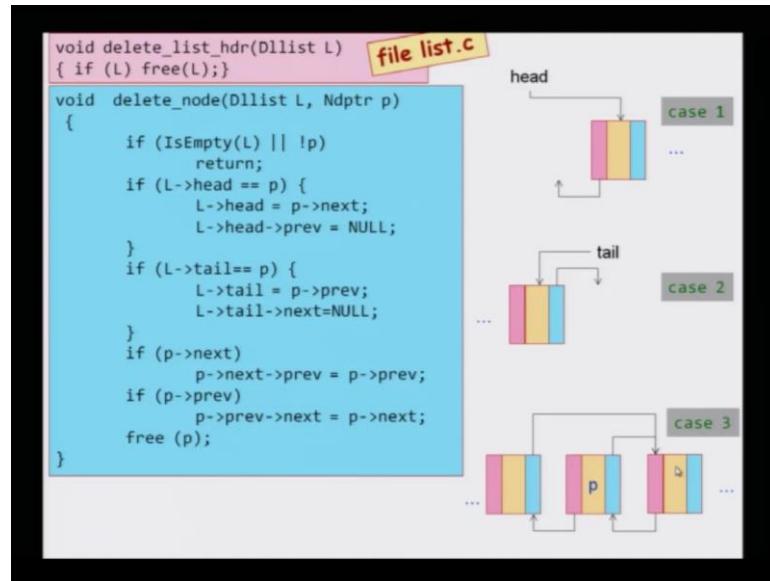
Then tail should go into previous. Now this guy's next to will now point to null, because we are going to delete this node.

(Refer Slide Time: 10:02)



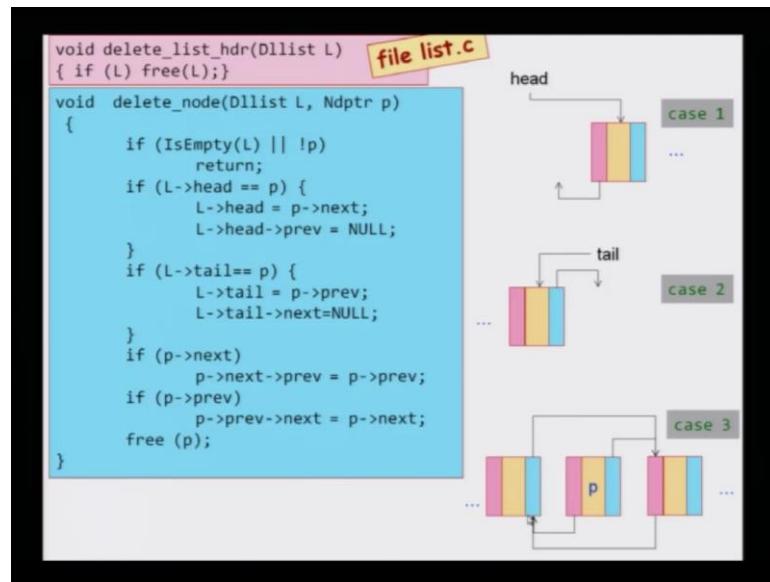
And finally we will `free(p)`. So, *L tail* will go to *p previous*, *L tail next* will be null, and then finally you will `free(p)`. So, we have seen two easy cases; one is delete a head, and other is delete a tail, and now we will see the difficult case where *p* is an intermediate node. So, in this case what we will do? So, we will we have to remove this node. So, *p previous next* node should be the next node of *p previous*. So, this link should point to the node after *p*. So, that is the first thing.

(Refer Slide Time: 10:45)



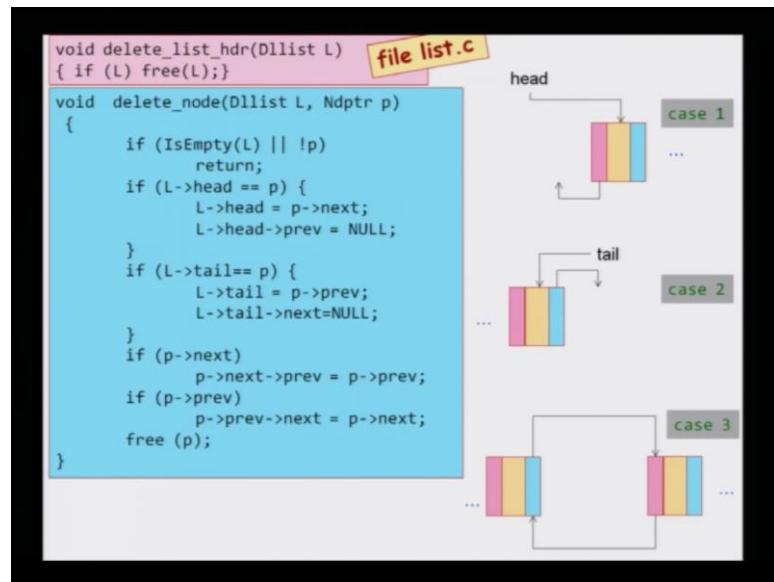
So, we will make this point to the node after *p*, and this node previous should point to the node before *p*.

(Refer Slide Time: 10:56)



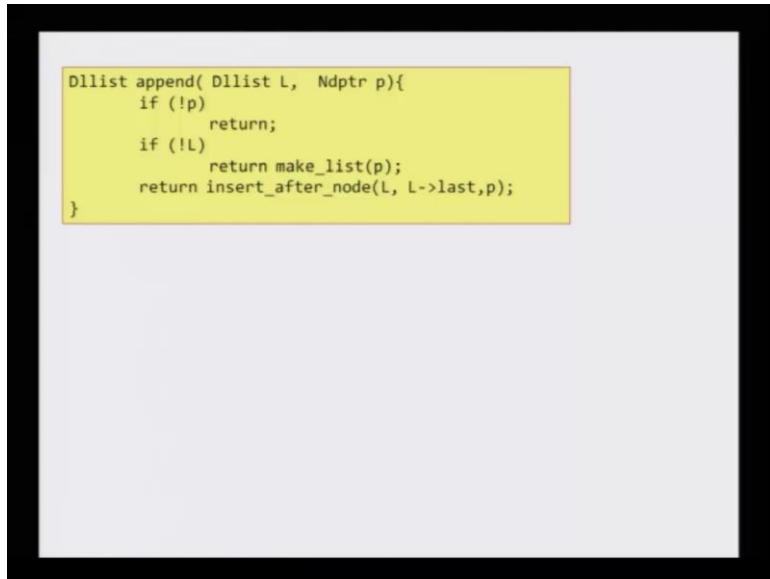
So will reset the links. Now, if you look at the link this guys next is the one after *p*, this guys previous is the one before *p*. So, now *p* can be safely removed.

(Refer Slide Time: 11:09)



So, this is how you would delete a node in the intermediate list. So, if there is the next node, then p next previous will be p previous, that is this backward link. And if there is a the previous node then p previous next will be p next, that is this forward loop. And finally, after that you will **free(p)**. So, this is how you would be delete a node from a link list from a doubly link list, and other operations can be done in a similar manner and some of these operation will be asked in their excise problem that you will be assigned. Similarly, you can think of an extract node, the code will be exactly identical to before except at the end, you will instead of freeing p, you will return p. You do not free the p node, we will just the return the p node.

(Refer Slide Time: 12:08)



Now, let us look at one more example, how do you attend append one node to the end of the list. So, first we will check that the node is pointing to a normal node, if it is pointing to a null node that is nothing to be the done. So, there is nothing to be appended. So, you have return. Now, if there is a list, then what to you do is if there is no need list what you do is you make an list with only one node which is p. Now if there is a list, you can in order to append the node at the end, what you could do is call [insert_after_node\(L, L->last,p\)](#) So, append will be the same as insert the node p at the end of the list. So, you will say insert after L last, what is the node to be inserted p? So, if you have in [insert_after_node](#) or an insert before node, you can do this to implement other functions.

So, this is the brief introduction to doubly link list which are similar to singly link list, but facilitate forward as well as backward traveling from a current node, using that you can implement more functions easier than a singly link list. At the same time, it has all the advantages of a singly link list in the sense that if you want to insert a node, it can be done using a constant of number operation, if you want to delete a node it can be done in the constant number of operations. So, those advantages are similar to a singly link list. At the same time the disadvantages are also similar to a singly link list. In the sense that if you want to search through even a shorted doubly link list, you have to search through all the elements.

Introduction to Programming in C
Department of Computer Science and Engineering

(Refer Slide Time: 00:07)

**Practices for writing larger
programs using C**

In this lecture we will see some practices for writing larger programs using c. As far as we have seen so far, we always wrote our code in a single file; and this is not practical for very large programs running into C 1000 of lines or millions of lines. So, we will see what is the usual practice for organizing a code when we have larger programs.

(Refer Slide Time: 00:32)

Structuring large programs

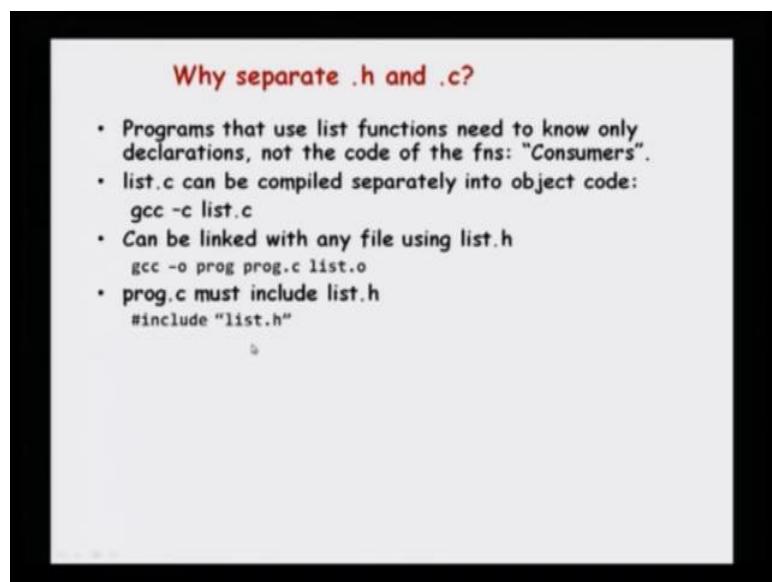
1. Separate function prototype declarations from function definitions.
2. Place structure defns, typedefs and function prototypes in .h file.
3. Place C code of function definitions in .c file.
 - a) E.g., list.h contains the struct dllnode, struct dlist and the typedefs.

So, one of the basic principles is to separate the function prototype declarations from the function definitions. We have seen that when we have a function we have two things to do - one is to declare a function which is just the types involved in the function, and then the definition of the function which is actually the code of the function. So, one way to structure it, one principle in structuring is that we will separate out the function prototype from the function definition.

Now, place all prototype definitions, structure definitions, typedefs, so just the declarations, you will place it in a file with suffix **.h**. So, right now we have been coding in a file call **.c**. So, right now what we are proposing is that the declarations alone we will place it in a separate file with suffix **.h**. You have already seen such an example which is **stdir.h**, we never bothered about what is inside a **stdir.h**. Now, we are talking about how to write these header files.

Now, decelerations are only half the function, right. I mean we have to write the definition of the function, the code of the function; the actual code of the function you place it in a **.c** file separately. So, for example, **list.h** contains the definitions of the struct **dllnode** for doubly link list and so on, and **list.c** would contain the bodies of the function.

(Refer Slide Time: 02:05)



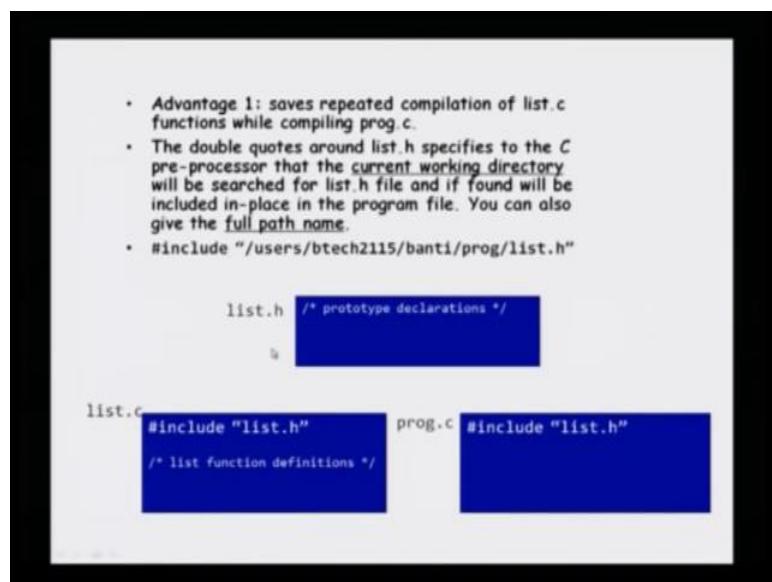
Now, we will see why separate **.h** and **.c**? Programs that use the doubly link list need to know only the declarations actually. These are, think of these programs as the consumers of this code. Now, it does not need to know how the code is implemented, just what to call and what is the declaration of the function. Now, if we do this, then **list.c** can be

compiled separately into object code. So, we can say `gcc -c list.c`, this will produce just file call `list.o`. `List.o` is not executable, but it can be used in other programs to create executables.

So, how can we do this? This is the procedure known as linking. So, we can link the `list.o`. So, notice the difference here. When we see `gcc -c list.c`, what it could produce is a `.o` function, `.o` file; and this `.o` file can be included to produce output. So, this says that we are compiling `prog.c` file, with `list.o` object file, and the output we will produce is called `prog`. So, `gcc -o prog` means, the output file we will produce will be called `prog`. So, if we omit `-o prog`, and simply say `gcc prog.c list.o`, then the file that we will get is, `a.out`. If you specify an output file we will get that output file.

Now, inside `prog.c`, let's say that we need to use `list` functions. So, `prog.c` will include “`list.h`”; this is the important thing. It will not say include “`list.c`”, it will just say include “`list.h`”. This is similar to what we have seen with `stdio`; we did not bother about whether there was an `stdio.c` file. We said we will include `<stdio.h>`. Also, notice the difference that we are using double quotes instead of angular brackets. So, when we wrote `<stdio.h>`, what we had was angular brackets, open angular bracket and close angular bracket, here we have quotes; why use that? We will see this.

(Refer Slide Time: 05:04)



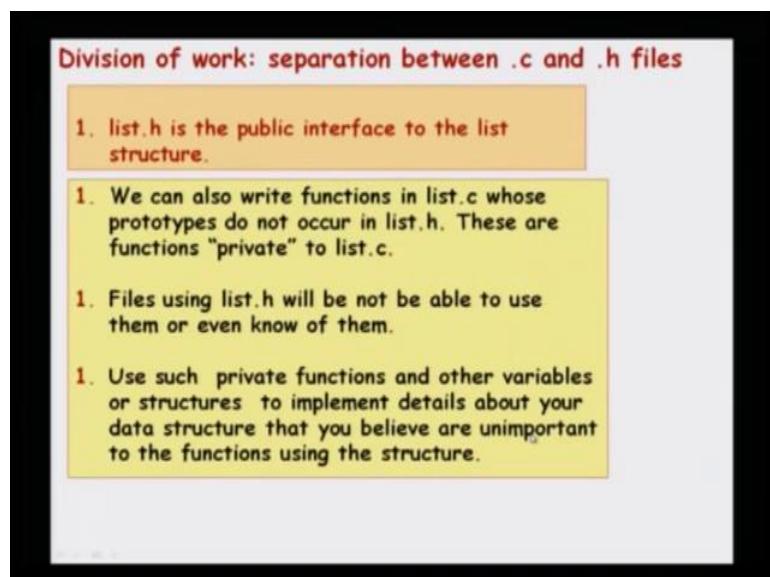
So, what is the advantage of separating `list.c` from `prog.c`, why break it up into multiple files? One advantage is that it saves a repeated compilation of `list.c` functions while

compiling `prog.c`. So, the “`list.h`” specifies to the c pre-processor; we will see this in a subsequent lecture, that the current working directly will be searched for `list.h` file.

So, since we are saying that `include "list.h"` within double quotes, what it means is that, where is `list.h` found? It will be found in the current directory. If it is not found in the current directory it will search for some standard library parts, some standard header file parts. So, in the case of `<stdio.h>` we put `<stdio.h>`; that means, that `<stdio.h>` will be found not in the current directory, but in some standard header directories.

So, when you use double quotes you can also use some full path names. Suppose, your full path name in a Linux system is “`/users/btech2115/banti/prog/list.h`”, you can specify the hole path as well. This is the more general notation. So, currently the structure is as follows: you have a `list.h` file, it has just the prototype declarations; `list.c` will define those functions. So, first inside `list.c` you would say, `include "list.h"`, and then have all the function definitions. `prog.c` needs `list.c` functions, but instead of saying `include "list.c"` it will say `include "list.h"`. Now, we will see how to compile such a setup.

(Refer Slide Time: 07:00)

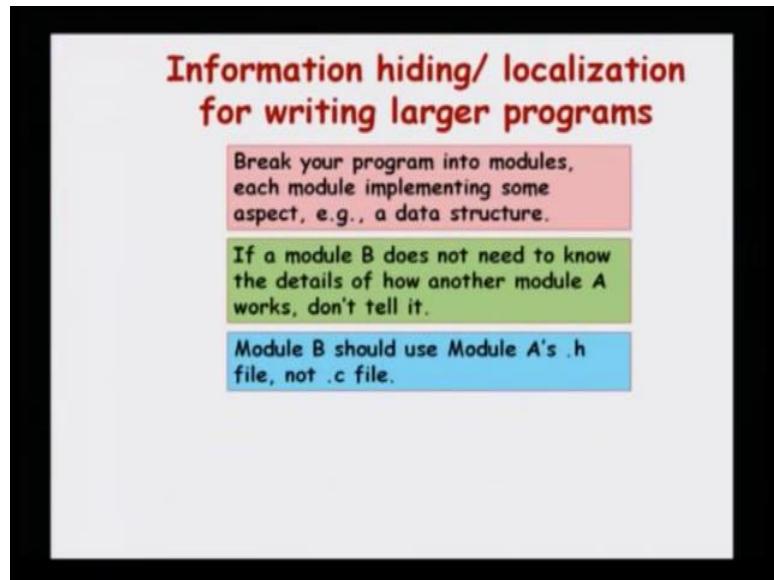


So, what is the division of work? What is the separation between the `.c` and the `.h` files? `.h` file is the public interface; that, if anybody else, any other program wants to use the list functions what you would do is, include the `list.h` functions, include the `list.h` file. Now, `list.c` implements, defines all the functions that `list.h` has declared. In addition, it can also define other functions, but these functions will not be available to other

programs that are using the **list.h** file.

So, files using **list.h** will not be able to use these extra functions or even know about these functions. These are thought of as private functions. So, this can be used to implement certain details of your code that other uses of this program need not know about.

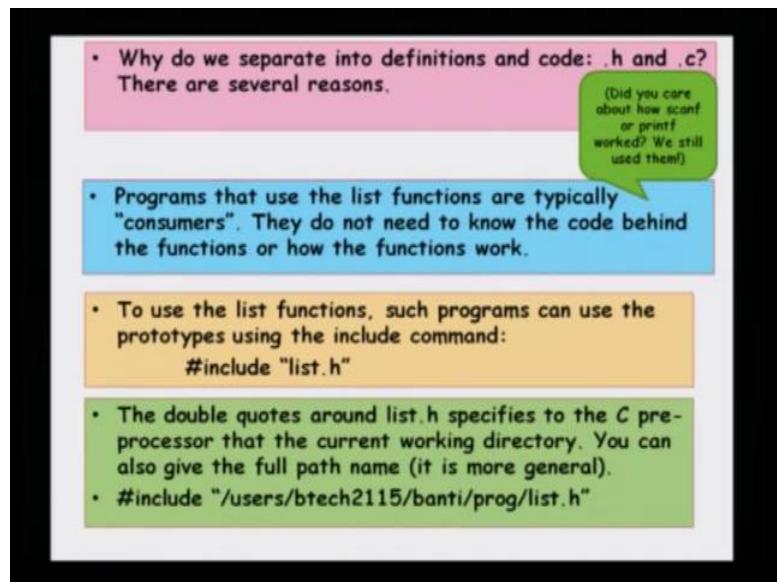
(Refer Slide Time: 08:03)



So, the general principle is what is known as information hiding or localization. So, break your programs into modules. We have already seen one way to break a program into modules, which is by writing functions. Now, this is another way to, this is another level of module array station where you say that take a collection of functions and put them in a file and have multiple files.

Now, each module implementing some aspect; for example, data structures like a link list. Now, if a module B does not need to know the details of how another module A works, then we do not need to tell B about how it is done. But, module B should use module A's **.h** file, not the **.c** file.

(Refer Slide Time: 08:54)



Why do we separate definitions into **.h** and **.c**? There are some reasons. Programs that use the list functions, for example, are typically consumers, and they do not need to know the exact details behind how these functions work. And we have already done this in other, we do not know about how scanf or printf worked. We just know that scanf needs these two arguments, for example, it needs a format string and it needs the variable to be printed; the printf needs the format string and the variable to be printed. Similarly, scanf needs the format string and the variables to be assigned.

So, we just knew that, we do not know anything about how scanf or printf is actually defined or implemented; we just know that it needs these arguments, and therefore we can call them. So, this is the kind of separation of detail that we are hoping to achieve. Now, so if some program wants to use the list functions, such programs can use the prototypes using the include command, **#include “list.h”**. So, again to remind the double quotes specify that it is the current working directory that **list.h** is present in; you can also give full paths.

(Refer Slide Time: 10:17)

.h vs .c (fast compilation)

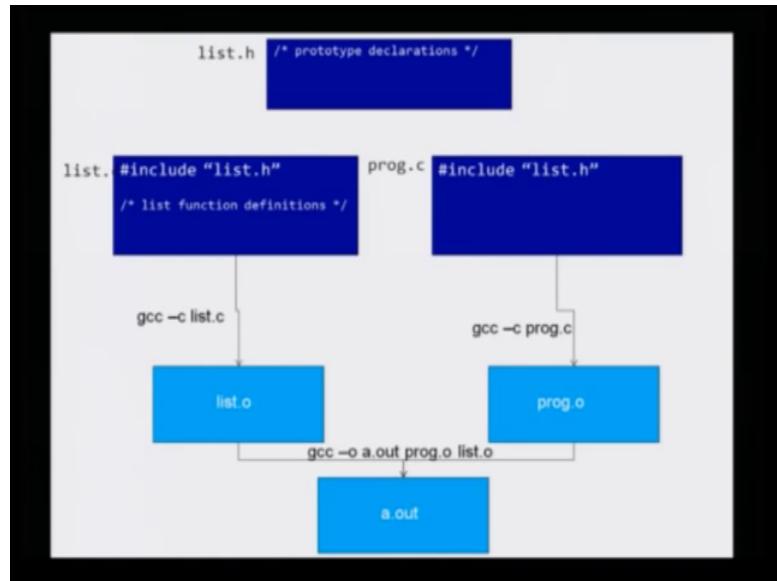
- The `list.c` program will contain all the function definitions.
- Usually header files `.h` are much smaller than `.c` files.
- If `list.h` and `list.c` are separated, then, `list.c` can be compiled ahead of time. This can be done by generating the object code (but not executable code)
`gcc -c list.o list.c`
- The `list.c` program is complete, except for a `main()` function. So it can be compiled into an object code file, usually ending in the suffix `.o` in Unix environments. The above compilation creates an object file `list.o` from `list.c`.
- Suppose we have written a function called `prog.c` that uses many of the `list` functions and includes `list.h`.
- We can compile `prog.c` to generate object code:
`gcc -c prog.o prog.c`

`List.c` program will contain all the actual function definitions. Now, usually header files are much smaller than the c files. If `list.h` and `list.c` are separated, then `list.c` can be compiled ahead of time, and you can generate the object file. Now, notice that `list.o` in this case will be not executable; it is just an object file that can be used to build executables.

Now, `list.c` programs is complete, except for a main function. So, it has a lot of functions; it defines all the functions there `list.h` has declared, plus optionally some more functions. And it can be compiled to produce an object code, but it cannot be done, it cannot be compiled into an executable code because it does not have a main function.

Now, suppose we have written function called `prog.c` that uses many of the `list` functions, that uses `list.h`. We can compile `prog.c` to generate an object code: `gcc -c prog.o prog.c`. So, now, we have 2 object files, `list.o` and `prog.o`; and then we can use these two object files to create the executable file.

(Refer Slide Time: 11:47)

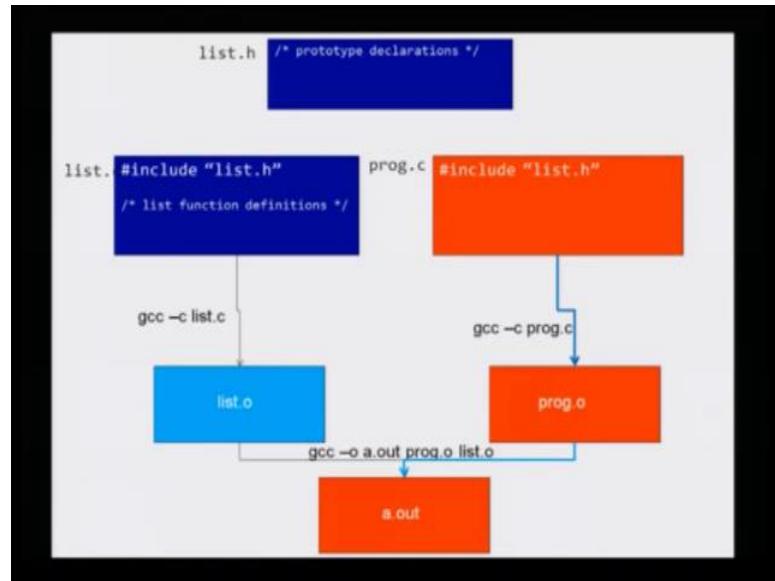


Let us look at a diagram which will hopefully be helpful. So, `list.h` includes a prototype declarations, but not the function definitions; `list.c` defines these functions. So, in order to define these functions, first it says, `include "list.h"`, therefore, it will get all the declarations, and then it has this code which implements the list function definitions. `prog.c` is a consumer which needs these functions. So, how does it do it? It does not say `list.c`, it says `include "list.h"`. So, the declarations of all the functions are available to `prog.c`.

Now, I separately compile `list.c` into `list.o` using `gcc -c`, and `prog.c` into `prog.o` using `gcc -c`. So, now, I have 2 object files, `list.o` and `prog.o`. And these will be combined using `gcc -o`. So, this says that the output file will be called, `a.out`; the compilation units that I need are `prog.o` and `list.o`. So, use these two files, in order to create the output file, `a.out`.

And what is the big advantage here? Let us consider a scene where the `prog.c` file changes. I need some changes to be made into `prog.c`; maybe I add some more functions, modify some functions and all. So, now I need to recompile and produce the output file. I have changed `prog.c`, but not `list.c`.

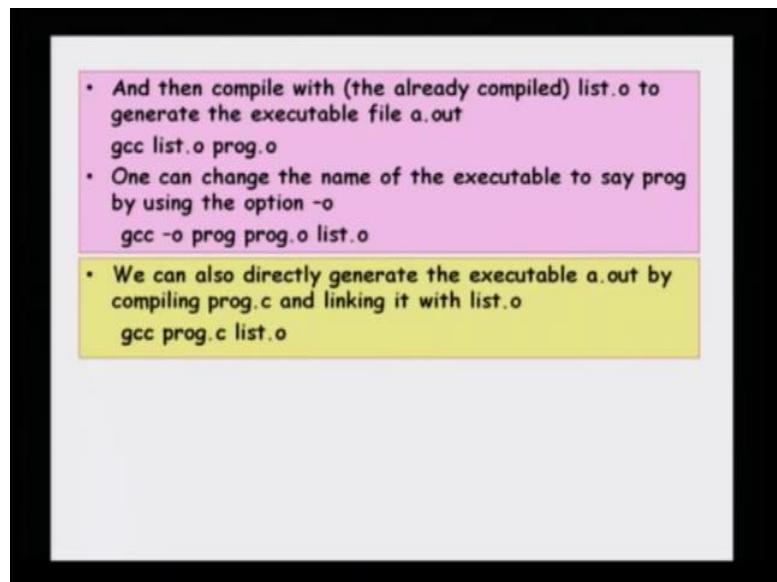
(Refer Slide Time: 13:30)



So, let us say `prog.c` has changed. What should I do now? I should compile only the `prog.c`. I can say `gcc -c prog.c`; now, I will produce a new `prog.o` file. Notice, `list.c` has not changed. So, we do not need to recompile `list.c`. So, we can just say, `gcc -c prog.c`; `list.o` is same as before. And then I can use the new `prog.o`, the old `list.o`, in order to produce the new `a.out`. So, notice the, `a.out` depends on `prog.o` and `prog.c` has changed.

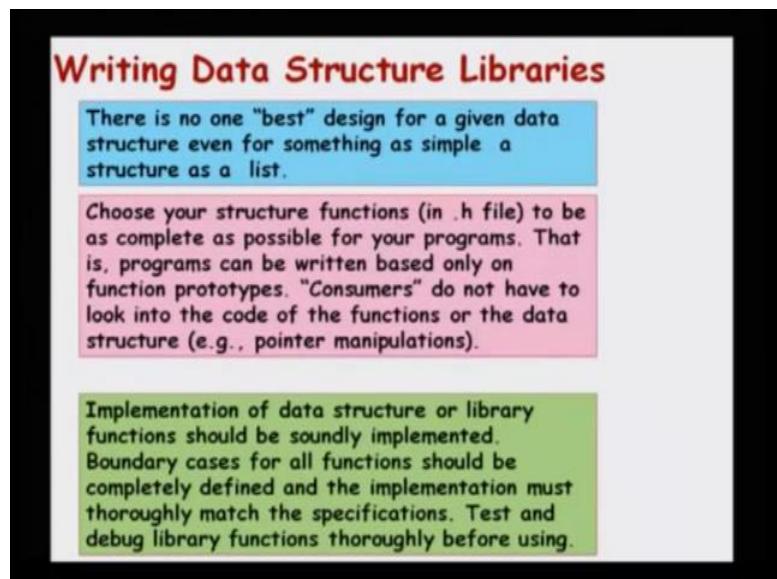
So, only this path gets recompiled which is saving a lot of effort. And in large programs, when one particular file changes and you recompile the project, only the necessary files get recompiled. It does not recompile the whole project which will take a lot of time, instead it will compile only those files which are necessary. So, this is the huge advantage.

(Refer Slide Time: 14:37)



So, this just repeats what was said in the last slide.

(Refer Slide Time: 14:49)

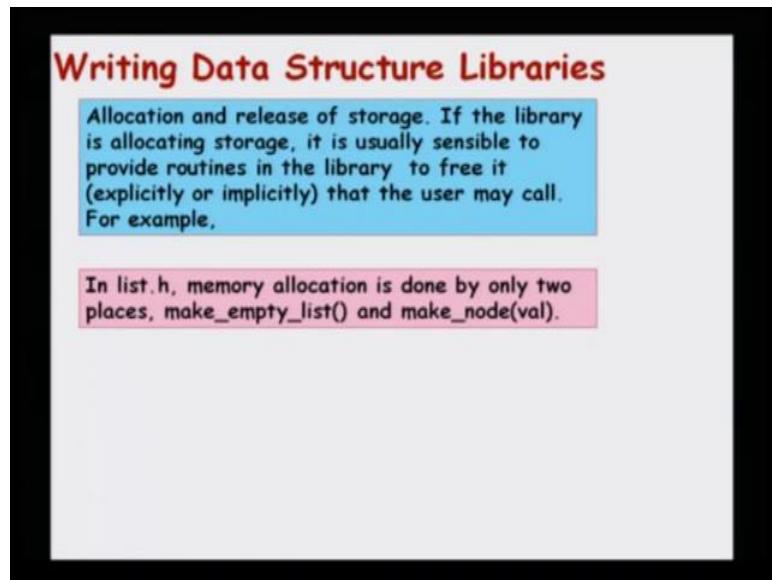


Now, couple of thoughts about writing data structure libraries. There is no one design which is best for a data structure library. Choose your structure functions to be as complete as possible for your programs. Now, programs can be based only on the function prototypes. Suppose I write a program which needs a list function, I can just look at the function prototypes in the .h file and then write my program. Consumers do not need to know how the program is implemented; just what the functions are, what are

its arguments, not the details about how it is implemented.

Now, implementation of libraries should be very sound. All boundary cases should be completely defined and the implementation should thoroughly match the specifications. So, libraries need to be tested and debug thoroughly before other users can use it.

(Refer Slide Time: 15:50)



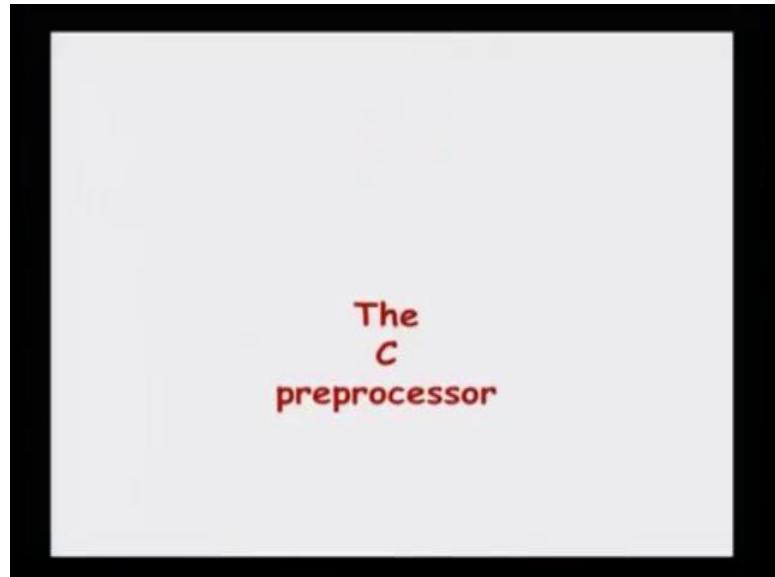
Also one more thing, allocation and release of storage. If the library is allocating storage, it is only sensible to provide routines in the library itself which can free those storage. So, it cannot be that libraries allocating a storage and the freeing of storage has to be done outside the library, that is not a sensible design. So, if the library itself is allocating storage, you give library functions to free the storing as well. For example, in `list.h`, memory allocation is done only in two places, make empty list and make node. So, to deallocate that you should provide a free functions for these functions, corresponding to these functions.

Thanks.

Introduction to Programming in C

Department of Computer Science and Engineering

(Refer Slide Time: 00:09)



In this lecture, let us **look** at one particular part of the C compiler which is very important, namely the preprocessor. Technically speaking, the preprocessor is the step before compilation. So, let us understand this in detail.

(Refer Slide

Time:

00:18)

The C preprocessor

We have used statements such as
`#include <stdio.h>`
`#include "list.h"`

You may have also seen the use of `#define` in C programs
`#define PI 3.1416`
`#define MAX 9999`

Lines in a C program that start with the hash character `#` are viewed as macros by the C preprocessor and are transformed by it.

The C preprocessor implements the macro language used to transform C programs **BEFORE** they are compiled.

1. As part of the compilation, first the C preprocessor runs and transforms macros.
2. The resulting file, including the transformed macros is compiled by the C compiler.

We use statement such as `#include <stdio.h>`; also `#include "list.h"`. And you may have seen C code which looks like this. You say `#define PI` to be 3.1416; `#define MAX` to be 9999, something like this. So lines in a C program, that is start with a hash symbol are called macros. And they are viewed and they are processed by the C preprocessor. Now, the C preprocessor implements what is known as a macro language; part of C. And it is used to transform C programs before they are compiled. So, C preprocessor is the step just before compilation. We do not explicitly call the C preprocessor. But when you write `gcc`, some file name, `gcc` the first step is the preprocessor step. So, as part of the compilation, first the C preprocessor runs and then transforms the macros. The resulting file, including the transformed macros is compiled by the C compiler.

(Refer Slide Time: 01:33)

Header files

- A header file is a file containing C declarations and macro definitions to be shared between several source files.
- header files are included in your program using C preprocessing directive '`#include`'. For example,

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

Header files serve two purposes.

1. System header files declare interfaces to parts of the operating system (system calls, libraries).
2. Your own header files : contain declarations for interfaces between the source files of your program.

<http://gcc.gnu.org/onlinedocs/gcc-3.0.1/>

So, let us look at header files. A header file is a file containing C declarations, macro definitions etcetera to be shared between several source files. Header files are included in your program using C preprocessor directives “hash include”. For example, we have seen `<stdio.h>` and within quotes `list.h`. So, header files serves two purposes that we have seen. First is that it could be a system header files. This declares interfaces to part of the operating system including system calls, C libraries and so on. Or you could have your own header files, which you have written to contain declarations of your code of the functions in your code.

(Refer Slide Time: 02:31)

Header files

What does including the header file do?

Including a header file produces the same results as copying the header file into each source file and at the exact place where the corresponding #include command was written.

Advantages

1. Related declarations appear in only one place.
2. Single file to change, modify etc.. Modifications are automatically seen by all files that include it.

<http://gcc.gnu.org/onlinedocs/gcc-3.0.1/>

Now, what happens exactly when you include a header file in a C file? Including a header file produces the same results as copying the header file into each source files. So, when you say `include<stdio.h>`, it is essentially taking the contents of the `<stdio.h>` file and copy, pasting in to your source code. So, this happens at exactly the place where the corresponding hash include command was written. The advantages of having these header file and hash include is that related declarations appear only in one place. And if you want to change a particular function or declaration of a function, you just have to change it in a single file. And all files which include that header file will automatically see the change.

(Refer Slide Time: 03:24)

The slide has a red header 'Include Syntax: #include'. Below it is a bulleted list of rules for the #include directive:

- `#include <file>`: used for system header files. It searches for a file named `file` in a standard list of system directories.
- `#include "file"`: used for header files of your own program.
 - It searches for a file named `file` first in the directory containing the current file, then in the same directories used for `<file>`.
- The argument of `#include` behaves like a string constant.
- Comments are not recognized, and macro names are not expanded. Thus, `#include <x/"y>` specifies inclusion of a system header file named '`x/"y`'.

At the bottom of the slide is a yellow URL box containing the text: <http://gcc.gnu.org/onlinedocs/gcc-3.0.1/>

So, here is something that we have mentioned in the previous video. If the difference between angular bracket in the double quotes; so the angular bracket is usually used in system header files and it searches for the file named let say `<stdio.h>` in a standard list of system directories. If you say within double quotes, on the other hand like `list.h`, it searches for this `list.h` first in the current directory. If it is not found in the current directory, then it goes again in to the standard list of that. Now, the argument for hash include; whether you include it in a angular bracket or in a double quotes, it behaves like a string constant and it is literally put there. So, if you have like comments, the comments are not recognized as just comments. If you have a `*` symbol, for example, it will be just put exactly like a `*` symbol. So, it is just treated as a string constant and no interpretation is done.

(Refer Slide Time: 04:30)

Example

Suppose there is the following header file named header.h

The file p.c includes this header file as follows.

After C preprocessor processes the file, the C compiler will see the input which would be same as p.c written as follows:

header.h

```
char *error="Overflow";
```

p.c

```
int x;
#include "header.h"
void main() {
    printf("%s", test());
}
```

```
int x;
char *error="Overflow";
void main() {
    printf("%s", test());
}
```

1. Included files are not limited to declarations and macro definitions; those are merely the typical uses.
2. Any fragment of a C program can be included from another file.

<http://gcc.gnu.org/onlinedocs/gcc-3.0.1/>

Now, let us look at a very special case that could happen in the header file. Typically, you would not do this. So, suppose you have; within the “header.h” you have a text **char *error=“Overflow”**. Typically, you do not initialize variables in a header file. But, let us say that in a particular “header.h”, we have this **char *error=“Overflow”**.

Now, in **p.c** I write this very peculiar thing. I write **int x** and then in the middle of the code I say **#include “header.h”**. Till now we have always used hash include headers in the beginning of the file. But, suppose what happens if I do it in the middle? Now, after the C preprocessor processes the file, what happens is that whatever text is there in “header.h” is copied, pasted at that position. So, for example, this code will be transformed by the C preprocessor to look like this. It will say **int x** and the “header.h” contain the single line; **char *error=“Overflow”**. So, that text will come here. Now, this transformed text is what the C compiler sees and it will compile in and produce the object code or the executable.

So, included files are not limited to declarations and macro definitions, these are merely the typical uses. You can put any text there. And when you include that header file, the text will be copied, pasted into the position. Typically though you would want to avoid this, you would want only declarations in a header file.

(Refer Slide Time: 06:13)

The slide has a yellow header bar with the title '#define—the object-like macros'. Below the title is a bulleted list of three items:

- An object-like macro is an identifier that will be replaced by a code fragment. It is called object-like because it looks like a data object in code that uses it.
- Commonly used to give symbolic names to numeric constants. Created using '#define' directive.
- For example,
 #define BUF_SZ 1024
defines a macro named BUF_SZ as an abbreviation for the token 1024.

At the bottom of the slide is a URL: <http://gcc.gnu.org/onlinedocs/gcc-3.0.1/>

Now, let us look at some other features that the C preprocessor gives. We have seen in some code, this use of `#define`. So, `#define` is used for what are known as object-like macros. An object-like macro is basically an identifier and it will be replaced by some code text. It is called object-like because it looks like an object. So, its most common use is to give symbolic names to numeric constants. Suppose, you want to write a program in which the maximum array size is let us say `1024`, instead of putting `1024` in several places, the typical usage in a program would be to say `#define BUF_SZ`; so buffer size to be `1024`. So, you have used `#define` to define this identifier `BUF_SZ` and `BUF_SZ` will be assigned the text `1024`. So, this says that I am defining a macro named `BUF_SZ`. And it is an abbreviation, the short form for the token `1024`.

(Refer Slide Time: 07:34)

#define—the object-like macros

- If somewhere **after** the
 `#define BUF_SZ 1024`
 directive there is a C statement say:
 `char *str = malloc (BUF_SZ, sizeof(char));`
- Then the C preprocessor will recognize and *expand* the
 macro `BUF_SZ`.
- The C compiler will see the same input as it would if you
 had written
- `char *str = malloc (1024, sizeof(char));`

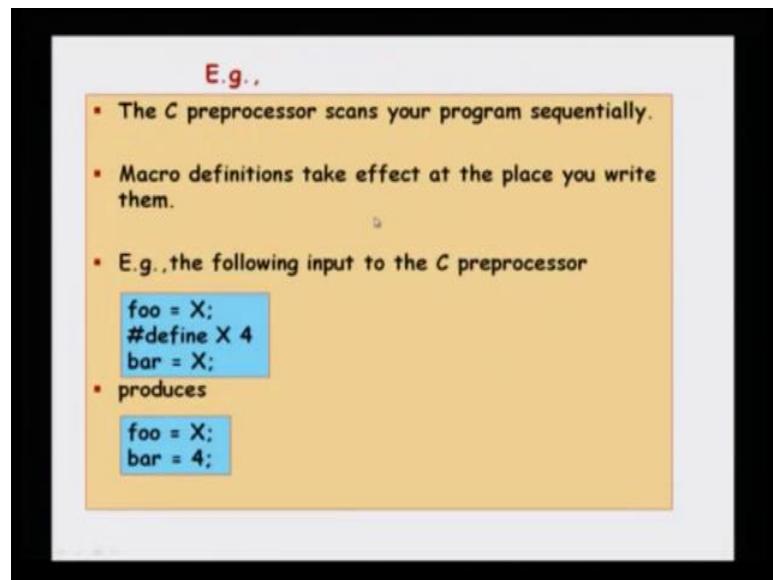
• By convention, macro names are written in upper case.
Programs are easier to read when it is possible to tell at
a glance which names are macros.

<http://gcc.gnu.org/onlinedocs/gcc-3.0.1/>

Now, if somewhere in your code if you say `#define BUF_SZ 1024`, in all places after that, suppose you say like `char *str= malloc` or `calloc(BUF_SZ, sizeof(char))`, what the preprocessor will do is that it will look at this string constant. It is the name of a macro. And it will replace it with `1024` which is value of the macro. So, the transformed text will look like this; `char *str= malloc` or `calloc (1024, sizeof(char))`.

Now, by convention macro names are return in upper case, so that somebody who reach the code will be aware that code this could be a macro; because if I write it in lower case, there are chances that somebody would think, that it is a variable name and look for the variable. So, writing it in capital letters is a way of indicating to the user that this is actually a macro. So, please look at in a header file for example.

(Refer Slide Time: 08:44)



So, the C preprocessor scans through a program sequentially. This is an important thing to understand. And macro definitions take effect at the place you write them. So, let us look at a very specific example to illustrate this point. So, suppose you write `foo = X`, after that you have a line say `#define X 4` and then `bar = X`. What will the C preprocessor do? It will look through the file and say `foo = X`. fine. It does not know what `X` is. It will not transform that line. Then it sees the `#define X 4`. Now, it knows that `X` is a macro and it has the value four. And then it sees `bar = X`, but now `X` is a macro. The preprocessor knows about this. So, it will replace `X` with four. So, that transformed text will be `foo = X; bar = 4`. It is natural to imagine that I would have `four = four`. But, that is not what happens; because the way the source code was written, the `#define` happened after `four = X`. So, anything that happens before the macros was defined is not changed.

(Refer Slide Time: 10:13)

**A typical project management problem:
#ifndef**

- Suppose we have created a file `list.h` and `list.c`.
- There is a file `p1.c` that needs `list` functions and creates a new set of functions. So we create `p1.h` and include "list.h" in `p1.h`.
- There is a file `p2.c` that needs `list` functions and some of the functions created by `p1.c`. So we create `p2.h` and include both "list.h" and "p1.h" in `p2.h`.
- When we compile `p2.c`, we include `list.h` twice, once from `list.h` and another from `p1.h`. Structure definitions are re-defined—this is a problem.

```

graph TD
    list[list.h] --> p1[p1.h]
    p1 --> p2[p2.h]
  
```

Now, let us conclude this discussion of preprocessor with a very typical project management problem. And we will see a third macro, that is, third operation that is done by the C preprocessor. This is something called `#ifndef`. This is used typically when you have multiple files in your project and you need to include multiple header files into a single source file.

So, let us discuss what is the problem with the particular example? Suppose, we have a `list.h` and the `list.c`. So, I have this header file `list.h`. Now, I have a program `p1.c`; that needs the `list` functions and also creates a bunch of new functions. So, its declarations will be included in `p1.h`. Now in `p1.h`, I would say include `list.h`. So, this is okay. I will have a corresponding `p1.c`, which will we just say include `p1.h`. Now, suppose that I have another file `p2.c`; the `p2.c` needs some functions in `list.h` and some functions n should `p1.h`. So now, there, when I write `p2.h`, I will say include `p1.h` and include `list.h`. Now, what happens is that when we compile `p2.c`, `list.h` gets included twice. First, because it indirectly includes `list.h`, and second because it includes `p1.h` which itself includes `list.h`. So, `list.h` code will be a copy, pasted twice in `p2.h`.

So, for example, this is the problem, because if `list.h` contains a structure definition, it will be included twice and the compiler will produce an error. So, this is the standard problem in large projects; where you want to include a file, but you do not want to include it multiple times. So, in this particular example I want to improve `list.h`, but I want to avoid the possibility that `list.h` is included multiple times leading to compiler errors.

(Refer Slide Time: 12:33)

Simple solution

- For every .h file, create a macro. E.g., for list.h create LIST_H (some name). Change list.h as

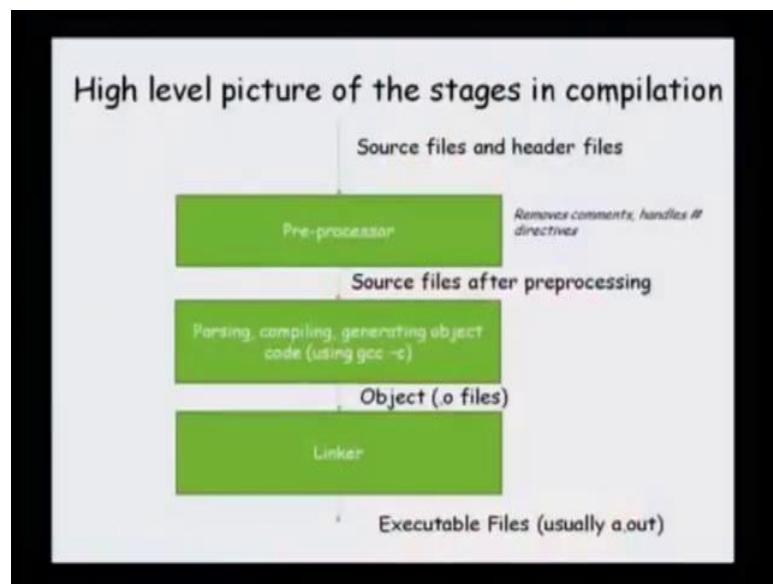
```
#ifndef LIST_H  
#define LIST_H  
/* all the statements/directives in list.h */  
#endif
```

- if LIST_H macro (not variable) is not defined as yet, then define it, and process all statements until the matching #endif.
- If List_H is defined, then, all statements until #endif are skipped and not processed.

So, how do we solve this? So, this is a facility provided by the C preprocessor. You can say you can use this `#ifndef`. So, it is saying that if this macro is not defined, then do something. So in `list.h`, I will write the following: `#ifndef`. This is a macro that I will define. So, usually if a file is `list.h` the macro will be named in capital letters; `LIST_H`. If this macro is not defined, then `#define list.h`. So, this says define `list.h` from me and then all the remaining statements in `list.h`. And then it will be enclosed in an end if.

So, now, what happens is that suppose `list.h` is included for the first time in `p1.h`, then `list.h` is not defined. So, it will define `list.h` and then include `list.h` in `p1.h`. Now, `p2.h` includes `p1.h` and `list.h`. So, now when `list.h` is included for the second time, the C preprocessor will look at this statement; `ifndef LIST_H`. That has been defined because `p1.h` has already defined. It calls it to be defined. So, it says that `LIST_H` is defined, so I will skip the entire thing until `ifndef`. So, this is one way to say that. So, if `LIST_H` macro is not defined as yet, then define it and process all statements until the matching `ifndef`. If it is already defined, this happens when you are trying to include it for the second time, then all statements until the `ifndef` are skipped. So, you do not copy, paste it for the second time. So, this is the standard way to avoid including one file multiple types.

(Refer Slide Time: 14:40)



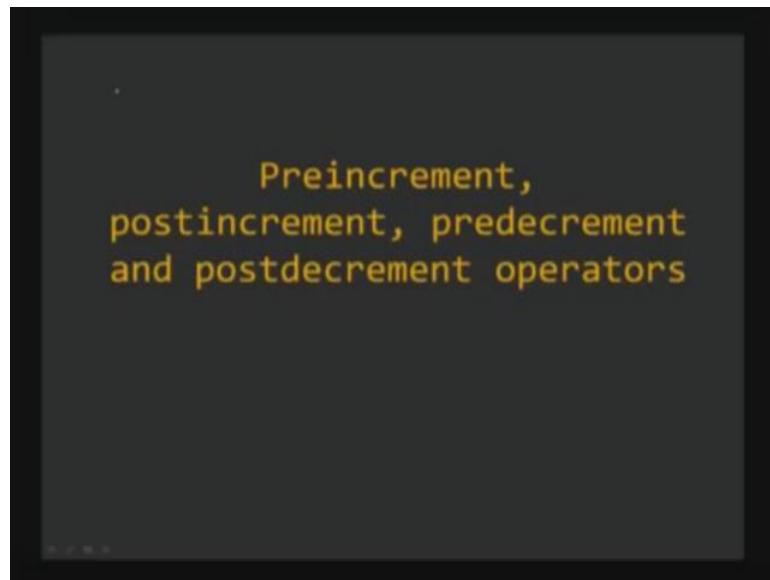
So, the high level picture of the stages in compilation. You have; we take up. So, the high level picture of the stages in compilation. You have source files and then it runs through this preprocessor, it produces the transformed files. And then after compilation using `gcc -c`, it produces object files. And after the object files are done, they are linked to produce the executable files. So when you press `gcc`, some source file, internally it first runs the C preprocessor, then it runs the compiler and then it runs the linker. And `gcc` provides facilities to stop the compilation at any stage. And so for example, we have seen in the previous video that you can stop the compilation just after the compilation itself by using `gcc -c`. So, it will produce `.o files`. And several `.o files` can be later linked to produce the `a.out` file.

Introduction to Programming in C

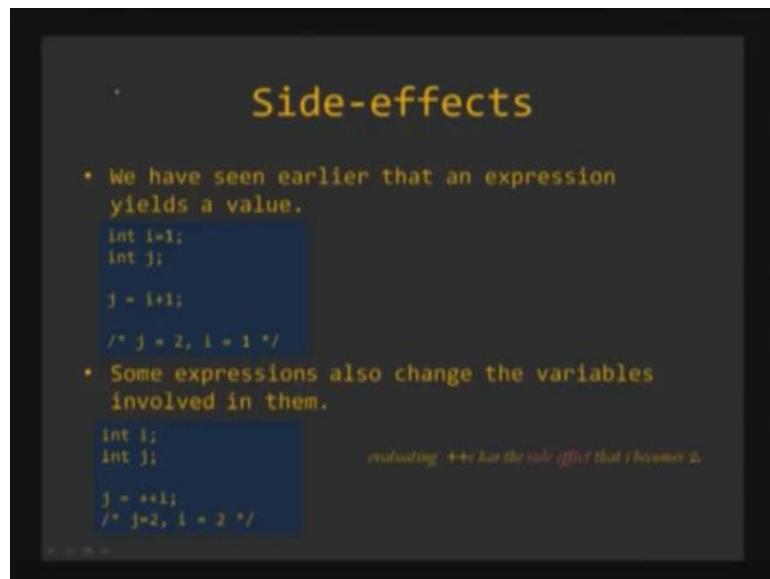
Department of Computer Science and Engineering

In this video will talk about how pre increment, post increment and operators like that work in c.

(Refer Slide Time: 00:00)



(Refer Slide Time: 00:09)



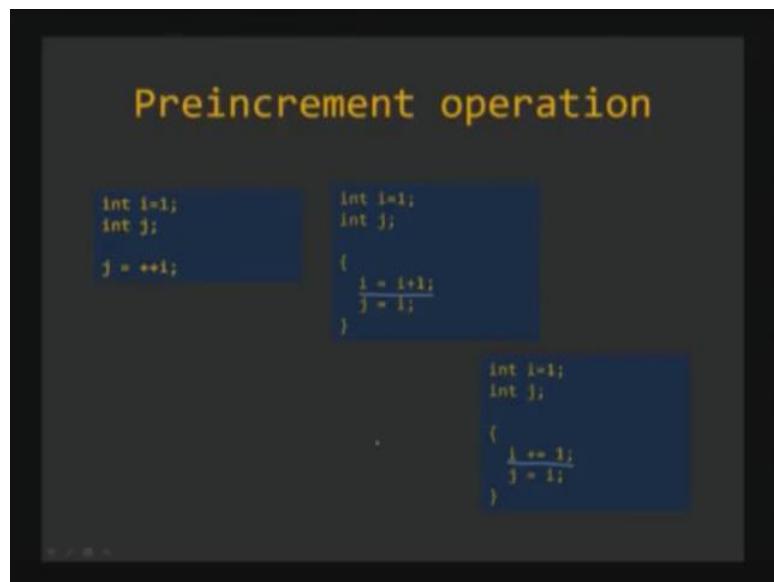
So, will first introduce the notion of side effects. Now, we have earlier seen that any expression in c yields a certain values. So, let us look at a particular example, if you have integer variables i and j, i is assigned to 1 and then, you say that j is assigned to i plus 1. What happens is that, you take the value of i add 1 to it and then result of the expression i

plus 1. So, the result of the expression will be 2, which is assigned to j. The value of i itself is unchanged due to an expression like i plus 1 is just that you read the value of a variable and then, return the value of i plus 1.

Now, some expressions in C also change the variables involved in them. For example, if you have a code like int i. Let us say i is initialized to 1 int j and then, you say j equal to plus plus i. In this case what happens is that, you take the value of i increment it. So, you will get i equal to 2 and then, that incremented value is then assigned to j. So, evaluating the expression plus plus i has the side effect, that i becomes 2. So, it not only takes the value of i increments it by 1 and gives it to j, it also has the additional effect that i's value is incremented.

So, contrast the first example and the second example, in the first example when you said i plus 1, the value of i was unchanged and in the second when you said plus plus i, the value of i is changed. So, this is known as a side effect, because in addition to returning the value it also changes the variable involved in plus plus i.

(Refer Slide Time: 02:09)



So, now, let us look at the operation in slightly more detail. So, when you say int i equal to 1 in j and then j equal to plus plus i. The effect of this plus plus i can be understood in terms of an equivalent code. So, what you do is, consider a code where you have i equal to i plus 1 and then assign j equal to i. So, in this case i will become 2 after i equal to i plus 1 and then j will be assigned the value 2. So, this is the effect of the pre increment operations.

So, pre increment operation is called. So, because before you use the value of i, you would increment the value of i. So, that is one way of understanding this and these two codes are equivalent in effect. There is a slightly different way of writing this, which is a short form for writing i equal to i plus 1. So, instead of doing this, you can say i plus equal to 1. So, plus equal to 1 says the effect i equal to i plus 1. So, it is a short form of writing it. So, all these codes are have equivalent effect. So, plus plus i is called a pre increment operator, because before you use the value of i its value is incremented.

(Refer Slide Time: 03:31)

The slide has a dark background with yellow text. The title 'Preincrement and postincrement' is centered at the top. Below the title is a bulleted list of three items. At the bottom of the slide is a code snippet in a light blue box.

Preincrement and postincrement

- `++i` is an example of an expression with the preincrement operation.
- You can also use `i++`, with the postincrement operation.
- How do we interpret complicated looking expressions like

```
int i=1;
int j;
j = (i++) + (++i);
```

Now, there is also the post increment operator. So, plus plus i is an example of an expression with the pre increment operation. And you can also use i plus plus, which is known as the post increment operation and confusing thing is how do we interpret fairly complicated expressions like the following. So, suppose you have int i equal to 1 in j and then j equal to i plus plus plus plus plus i. So, what should we expect in this case is this allowed behavior what does it mean? What will be the result? Which is stored in j? So, let us look at these things in slightly greater detail.

(Refer Slide Time: 04:16)

The slide has a dark blue background with yellow text. At the top, it says "Some simple examples". Below that is a block of C code. Handwritten annotations are present: a circle around "i++" in the first line, another circle around "i++" in the third line, and a bracket under "i++" in the fourth line pointing to the value "3".

```
int i=1;
int j, k;

j = i++; /* j is assigned the old value of i, and i is incremented after expression */
printf("%d %d\n", i, j); /* prints 2 1 */

k = ++ i; /* i is incremented, and the new value is returned. */
printf("%d %d\n", i, k); /* prints 3 3 */

k = ++ i + j++; /* i=3 j=1 */

/* i is incremented, and the new value is returned.
 * the old value of j is returned and j is incremented after the
 * expression. k=4+1
 */
printf("%d %d %d\n", i, j, k); /* prints 4 2 5 */
```

So, first let us look at some simple examples and try to understand the behavior. So, suppose you have i equal to 1 and then two variables j and k and first you say `int j = i + i;`. So, this is the post increment operator. So, what happens here is that, you take the value of i assign into j . So, that is i equal to 1, the current value of i will be assigned to j and after the expression is over i will be incremented. So, then i will become 2. The old value of i is assigned to j and then the value of i will be incremented. So, it is the post increment operator.

So, when you `printf(i, j)` here, i will be 2 and j will be 1. Because, old value of i was what was **stored** in j . Now, let us look at `plus plus i`. So, if you say `k = plus plus i`, i is now 2 when it starts and you pre increment i . So, you increment i , i becomes 3 and that value is stored in k . So, it is the pre increment operator,. So, the value will be incremented before the assignment will take place.

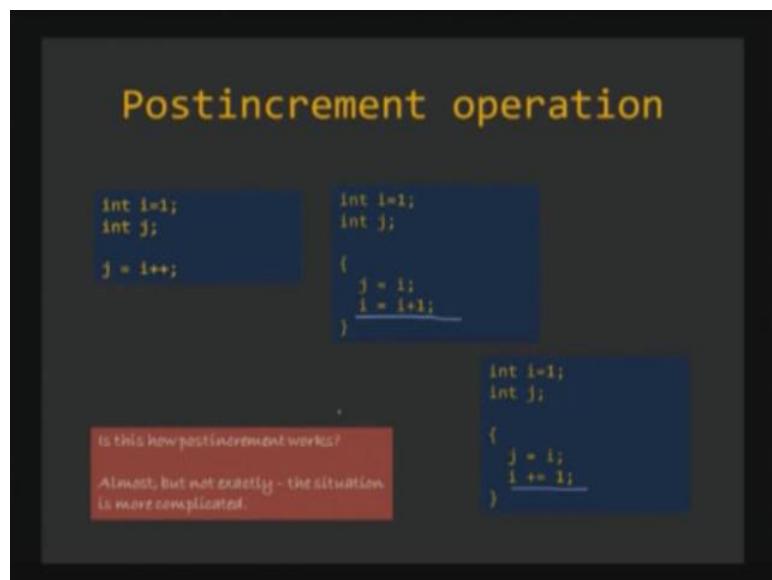
So, when you `print(i, k)`, i will be 3 and k will also be 3. So, notice the difference between the first case and the second case, the pre increment versus the post increment. Now, let us look at slightly more complicated examples. So, at this point what do we have? We have i equal to 3 and j equal to 1 at this point and then you say `k = plus plus i + j + plus plus`.

So, take a minute and think about what will happen here, you pre increment i . So, the value of this expression, that is used to add will be 4. Because, the value of i will be incremented before it is used in the plus expression, where as this is the post increment

expression. So, the old value of j will be used and then j will be incremented.

So, here the value that will be used will be 4 and here the value that will be used will be the old value of j which is 1. So, k will be 4 plus 1 which is 5, i will be incremented. So, i becomes 4 and after this expression is over j will be incremented,. So, j becomes 2. So, when you print this you will say that i is 4, j is 2 and k is 5. So, understand why k is 5? Because, it is 4 plus 1 rather than 4 plus 2. So, this is fairly simple can be understood in terms of the pre increment and the post increment operator.

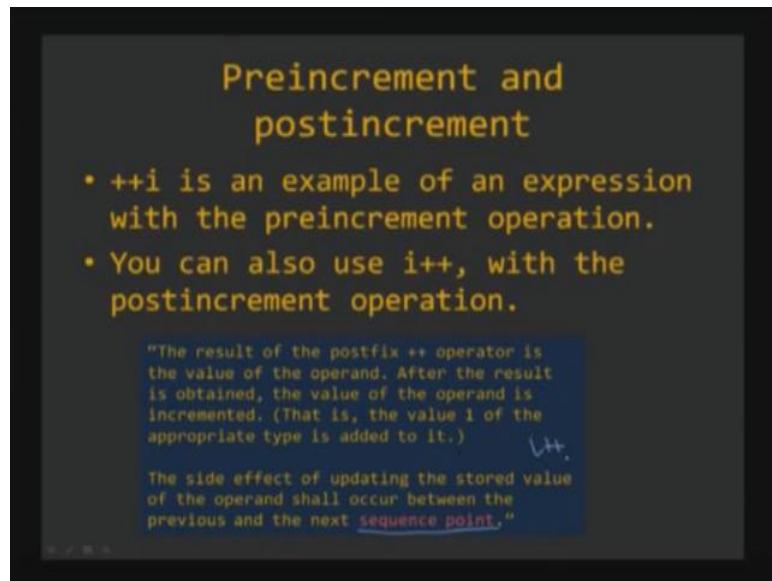
(Refer Slide Time: 07:25)



So, let us look at some code that is equivalent to the post increment operation. So, suppose you have j equal to i plus plus, you can think of it like the following, you can say that j is assign to i. So, the old value of i is assign to j and then, the value of i is increment i equal to i plus 1. If you want to use the compound assignment operation, then what you can do is j equal to i and i equal to i plus 1. So, this is equivalent to i plus equal to 1 is equivalent to i equal to i plus 1.

So, contrast with the pre increment operation, there i equal to i plus 1 will be done before j equal to i, here j equal to i will be done before i equal to i plus 1. So, can we see that this is exactly how post increment works? And the answer is yes, in this particular case this is exactly how it works. But, if you want to understand the general case, we have to understand a slightly more advanced concept in c. And note that, this is not something that strictly false in to an introductory course. But, in case you want to understand exactly how it works, then we will look at the general case.

(Refer Slide Time: 08:42)



So, to understand the general case instead of writing a few examples and compiling it and saying, one way to do it would be to go to the c standard and say what is the standard say and here it is slightly surprising. So, the result of the postfixes operator is the value of the operands. So, this is the old value of the operand will be return, after the result is obtain, the value of the operand is incremented, this is what we saw in the last let.

Now, when is the operand increment, we loosely said last time that after the expression is over then the value of `i` will be incremented. But, what is the precise point at which the value of `i` will be incremented, this is slightly surprising. So, the side effect of updating the stored value of the operand shall occur between the previous and the next sequence point. So, when you have an `i` plus plus operation, it will not be immediately updated, it will be updated only after a place known as the sequence point.

So, let us just understand briefly what is mean by a sequence point. So, before we get into it let me emphasize, we are trying to understand. So, the post increment operation will say that, the old value of `i` will be used and the value of `i` will be incremented after the expression, we are time to precisely understand after what point can we say that `i`'s value would have been incremented.

(Refer Slide Time: 10:23)

The slide has a dark background with yellow text. At the top, the title 'SEQUENCE POINTS' is written in a bold, sans-serif font. Below the title, there is a bulleted list of sequence points:

- A point in the code by which all pending side effects are guaranteed to take place.
- Some prominent sequence points
 - End of a full declarator
 - full expressions
 - immediately before a library function returns
 - after each formatted input/output specifier in an input/output function call.
 - for a full list, see C11 standard, Annex C.

So, a sequence point has defined in the standard is a point in the code by which all pending side effects are ensured to be over. So, this is very technical definition and it is to be understood by compiler writers. But, we will briefly understand what does it mean? So, some prominent sequence points include end of full decelerators. So, for example, if I have a declaration int i equal to 0 comma j equal to 0, then a full decelerated gets over after i equal to 0. So, after i equal to 0 there is sequence point here.

So, if there are any pending side effects, then it will be incremented at this point, this is another full decelerator. So, it will after that again any pending side effects will be ensured to be done. Then, the surprising think is suppose you have full expressions. So, suppose you have like i plus plus plus plus 3. So, the major think to understand will be when is this i plus plus suppose to happen, will it happen immediately after i plus plus and the think is then the c standard does not say that, that has to happen.

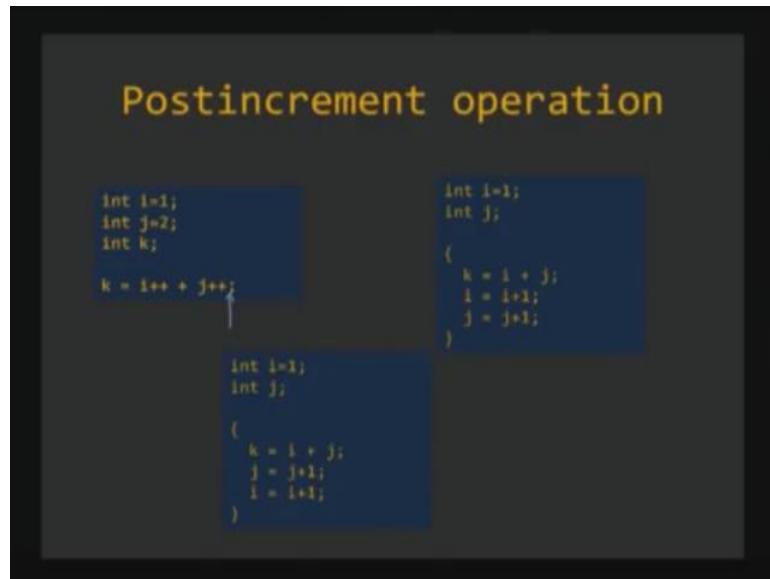
The c standard says that, the next sequence point is the semicolon. So, when you see the semicolon operation, you will know that, this hole think is what is known as a full expression j equal to i plus plus plus plus 3. So, that is known as a full expression. So, after you encounter a full expression any **pending** side effects. So, this is the pending side effect, that will be updated. So, only at that point c standard says that, by now you should have updated the i plus plus operation.

Before that the compiler is free to do what it works, it may or may not updated. So, this is actually slightly confusing and contrary to the popular understanding of when should i

plus plus have to again the general cases slightly confusing, it is not what would expect, it just says that by the next sequence point in the code, all pending side effects should be taken in place.

Now, it does not say that exactly at the end of the sequence point, you will update all side effects, compilers are free to do what it wants, all it says is that by the time you need the next sequence point pending side effect should take place in whatever all. So, this is slightly technical. Now, for a full standard list of course,, you have to refer to the C standard, which is not really recommended I mean,, but it is just that if you want to understand that, then you can look at the standard.

(Refer Slide Time: 13:29)

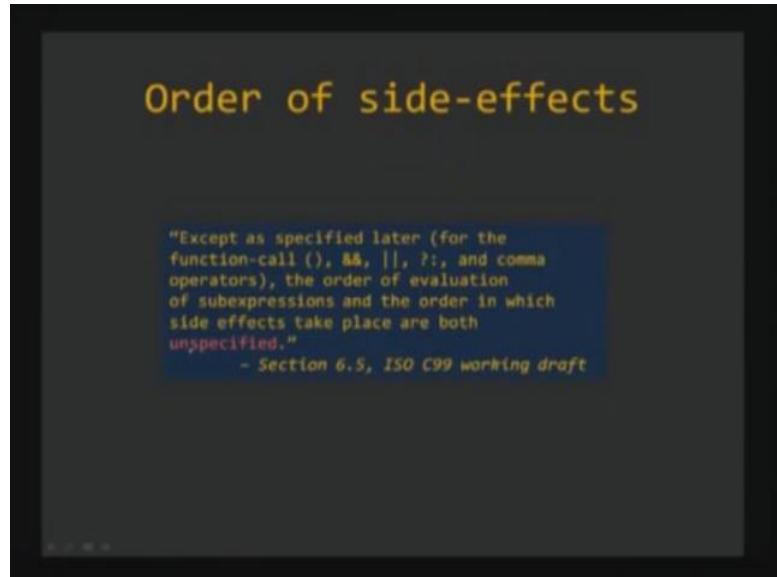


So, let us try to understand the post increment operation. So, it's again slightly greater detail. So, if you say that $k = i + j$, now, there are two ways to do it. Of course, $k = i + j$, the old values of i and j are used and then, you calculate k assign it to k . And then, you can do $i = i + 1$, $j = j + 1$, because the standard says that by the time you see the full expression pending updates must happen.

So, you can say that by the time you see the semicolon operation I will do $i = i + 1$ and $j = j + 1$. Now, if you think a minute you could also do update j first and then i . So, I know that by the time you see the semicolon pending update should happen,, but in what order should it happen is it $i = i + 1$ first and $j = j + 1$ next or is it the other way round. And the answer is that, the C standard does not say.

So, it leaves it deliberately unspecified,. So, that the compiler can do what it wants.

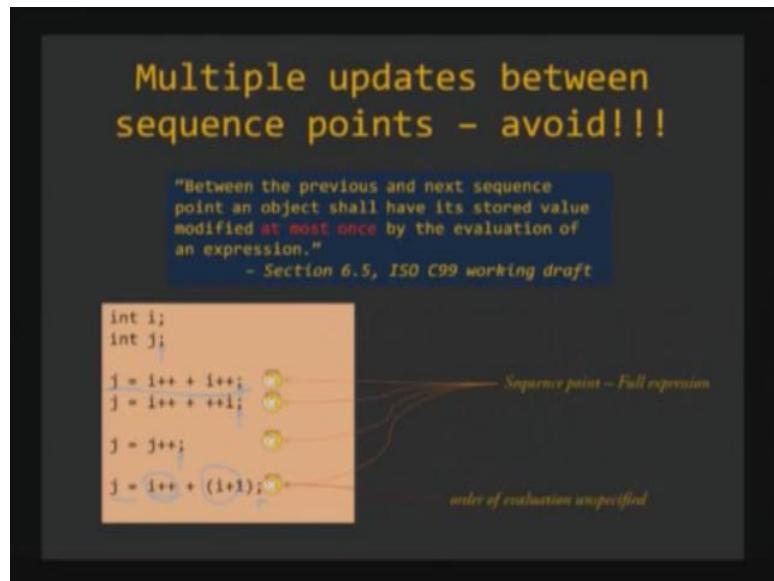
(Refer Slide Time: 14:41)



So, here is the second certlity in this whole business. So, if you say, what is the order of the side effects? There are certain operations, where the sequence is specified. For example, the function call the logical AND operator, the logical OR operator, the conditional operator and the comma operator. So, for very specific operators the sequence is specified. But, in all other operations the order of evaluation of sub expressions is unspecified.

And similarly, the order of side effects is also unspecified. So, in the previous slide doing i equal to i plus 1 before j equal to j plus 1 is valid, as also j equal to j plus 1 and i equal to i plus 1. So, both these orders are valid and the c standard does not say that, what should really happen? So, what in practice you will notice is that, in one compiler a certain order may happen, in another compilers certain other order may happen. So, it is left to the compiler and you cannot assume anything about, what really happens? Which order it happens?

(Refer Slide Time: 15:50)



Further and here is the most important think as for as the sequence points are conserved. The c standard says that, an object or a variable can have it is stored value modified at most once by the evaluation of an expression between two sequence points, this is very important. So, between two sequence points, if a variable is to be updated by a side effect, then it should be updated at most once. Beyond that, if it is updated multiple times, the c standard says that the result is actually unspecified.

So, let us look at a few specific examples to see, what is actually happening here? So, let us take the first expression j equal to i plus plus plus i plus plus. So, we know that, here is a sequence point and we know that, here is a sequence point. These are full expressions between these two sequence points, the value of i is updated more than once. Here is i plus plus plus i plus plus and the c standard says that, the behavior is unspecified, this is somewhat surprising.

Because, you may try it out multiple times and you will see that consistently some behavior is happening. But, what the c standard says is that, if you take the code and compile it with a different compiler, the result may be different. So, the result of this expression is actually unspecified. Similarly, let us look at the next example. So, here the sequence point is a full expression, let us look at the next expression.

So, i plus plus plus plus i . So, post increment and then pre increment. Again even in this case, the result is unspecified, because these two are the sequence points here between this full expression and between these full expressions. So, you have two

sequence points and between these two sequence points, the value of i is updated more than once. So, the result is unspecified according to the c standard.

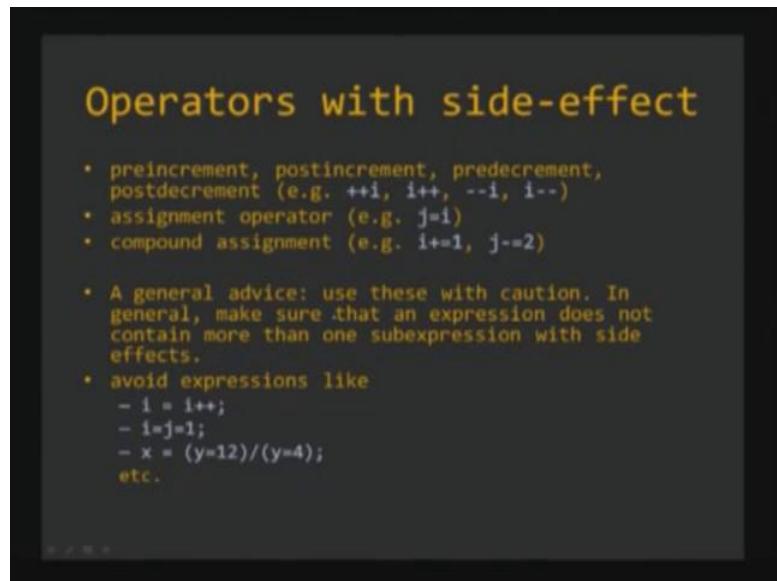
Let us look at this interesting example, j equal to j plus plus. Again result is unspecified, because you can have these two sequence points and between these, the value of j is updated twice. First, by the post increment operator and then by the assignment operator. So, the value of j is updated more than once, the result is unspecified. The last expression, it is interesting.

So, if you look at the two sequence points here, you have one full expression here, another full expression here. Between these, the value of i is updated only once, here and the value of j is updated only once, here. So, it is not that the value of i is updated more than once,. So, the value of j is updated more than once. But here, it say which of these sub expressions happened first? Is it i plus plus that happens first or i plus 1 that happens first.

So, according to the c standard that is actually unspecified. So, the order of evaluation of the sub expressions is unspecified, according to the c standard. So, let us just go back to that and this is, what it says, the order of evaluation of sub expressions is also unspecified. So, if you look at this expression j equal to i plus plus plus i plus 1, it is not clear which happens first, i plus plus or it is i plus 1, that is also unspecified.

So, here are the sequence points which end at full expression and the specific case of the last example, it is not that values of variables are updated more than once. It is just that the sub expressions may be evaluated in whatever order, it may occur.

(Refer Slide Time: 20:03)



So, all this is slightly confusing. So, let us just summarize something that you can take away for, as per as programming is concerned. So, let us list out a few operators with side effects. So, let us say pre increment, post increment, pre decrement, post decrement all of these have side effects. In addition to returning the value, it also updates the variable. The assignment operator clearly has side effects.

So, if you say `j = i;;` obviously, the value of `i` will be assigned to `j` and we have earlier seen that, as an operation it returns the value that was assigned. So, that has the side effect, because it updates `j` and also **return** the value, which is the value of `j`. We have also seen this compound assignments. So, you can say `i plus equal to 1`, which is the same as `i = i + 1` and `j minus equal to 2`, which is as same as `j = j - 2`. So, all these operators have side effects.

And the general advice is that, use operators with side effects with extreme caution. In general, if you use them make sure that a single full expression does not contain more than one sub expression with side effects. So, make sure that even if you want to use these expressions with side effects, make sure that one full expression contains at most one side effect.

So, avoid expressions like `i = i plus plus`, as we have seen before, this has two updates on `i`. So, the result is unspecified `i = j = 1`. Well, here is a full expression that has two side effects, technically the result is you can predict what the result is,, but as a programming practice, please avoid these kind of expressions.

Because, this is an expression that involves multiple updates and it is not really that the result is unspecified. Because, the updates are on different variables.

But still, as a good coding practice avoid such expressions. So, let us look at the third example, you have x equal to y equal to 12 divided by y equal to 4. Again, it is not clear, which of the sub expressions y equal to 12 or y equal to 4, which will happen first? So, the result of this expression is very difficult to interpret. So, in general do not use full expressions that have more than one side effects, even if they are on the single variable.

If it is multiple updates on a single variable, then the C standard clearly says that, the result is unspecified. But, even if they are on multiple variables, try to avoid writing such expressions. You can always write slightly longer code, where the meaning of the code will be perfectly clear and the result will be completely specified.

Thank you.



**THIS BOOK IS NOT FOR SALE
NOR COMMERCIAL USE**



(044) 2257 5905/08

|



nptel.ac.in

|



swayam.gov.in