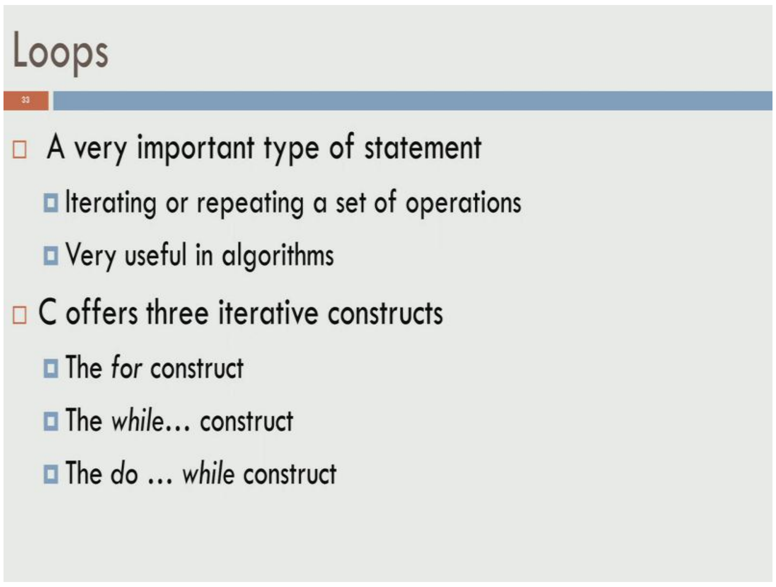
Programming, Data Structures and Algorithms
Prof. Shankar Balachandran
Department of Computer Science and Engineering
Indian Institute Technology, Madras

Module – 07
Lecture – 07
Contents
Repetitive statements or loops
Counter or sentinel
For loops with examples
Logical operators-AND, OR
Nested for loops
While loops with examples-GCD of two numbers by euclid's algorithm
Do-while loops with examples
Where to find the IDE Dev C++ (Sourceforge)

So, in module three, we will look at reparative statements. So far we have seen conditional statements both if and select using switch statements. We will look at repetitive statements.

(Refer Slide Time: 00:26)

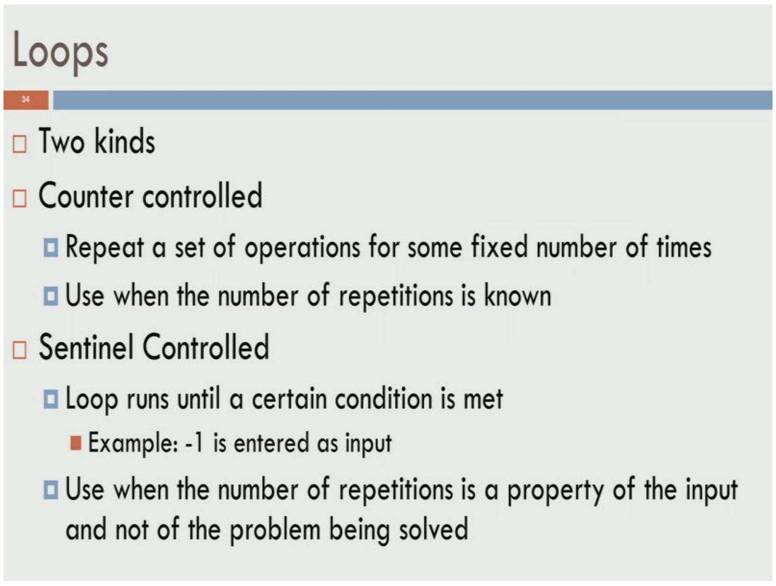


Loops

- A very important type of statement
  - Iterating or repeating a set of operations
  - Very useful in algorithms
- C offers three iterative constructs
  - The *for* construct
  - The *while*… construct
  - The *do* … *while* construct

So, repetitive statements are also called loops. It forms a very important class of statements, because you get to iterate or repeat over a set of operations. This is useful in lots of algorithms and it is a very basic construct. So, C offers three different kinds of repetitive statements namely for statement, the while statement and the do while statement. We will see each one of them and see motivating examples for each one of them.

(Refer Slide Time: 00:54)

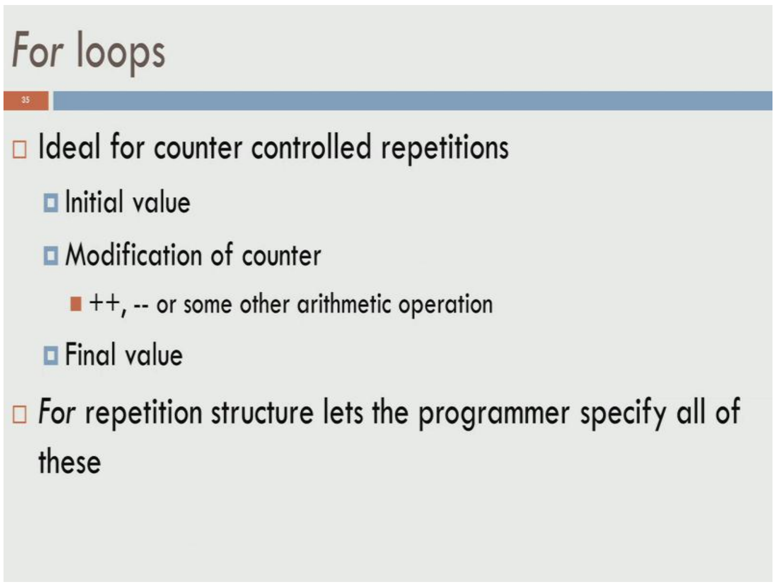


Loops

- Two kinds
- Counter controlled
  - Repeat a set of operations for some fixed number of times
  - Use when the number of repetitions is known
- Sentinel Controlled
  - Loop runs until a certain condition is met
    - Example: -1 is entered as input
  - Use when the number of repetitions is a property of the input and not of the problem being solved

So, in any loop construct, no matter what language you pick, there are two different kinds of loops that you can run. You can either have them counter controlled or what is called sentinel controlled. So, in counter controlled, what you have is you have a set of operations and you want to repeat that a certain number of times. And this number of times, you have to repeat; it is already known to you. Whereas, in sentinel controlled, you keep looping until a certain condition is met. So, for example, you may wait till the user inputs minus 1 and you want to do a repeated set of actions till the user inputs minus 1; maybe you want to find the factors of a number; but, the user could keep on giving these numbers. At some point of time, if the user inputs minus 1, you want to stop the program. So, if you want something like that, upfront you do not know how many inputs the user is going to give. In such cases, the sentinel controlled structures are useful. And this structure is usually used when the number of iterations is dependent on the input and not on the problem being solved.

(Refer Slide Time: 01:57)

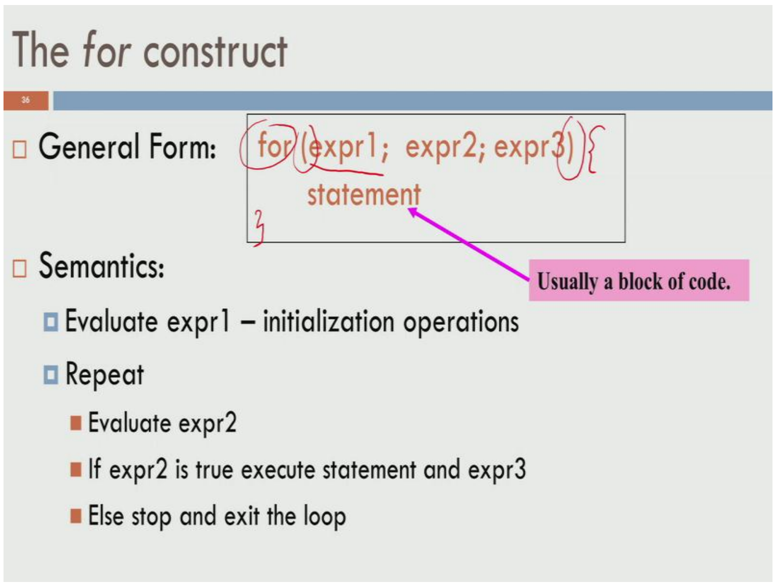


For loops

- Ideal for counter controlled repetitions
  - Initial value
  - Modification of counter
    - ++, -- or some other arithmetic operation
  - Final value
- For repetition structure lets the programmer specify all of these

So, let us start with the first one namely, the for loops. So, for loops is actually the… For loop construct is a counter controlled repetition structure. What you have is you start with an initial value of some counter and you modify the counter as we go along till you reach a final value. So, maybe I start with number 1; go in steps of 2. And let us I stop at 43; in that case, I would be using all the odd numbers. So, I am going in steps of two from 1. So, I would go 1, 3, 5, 7 and so on up till 43. Or I may say just for 20 times print hello. In that case, I would start the initial value at 1; the counter would go by increments of 1 and it stops at 20. So, this for loop – the way C does; it lets the programmers specify an initial value, a modification to the counter as well as a final value; and on top of that, it also lets you put in a body or what you would really want to do so many times.

(Refer Slide Time: 03:03)



The for construct

- General Form:

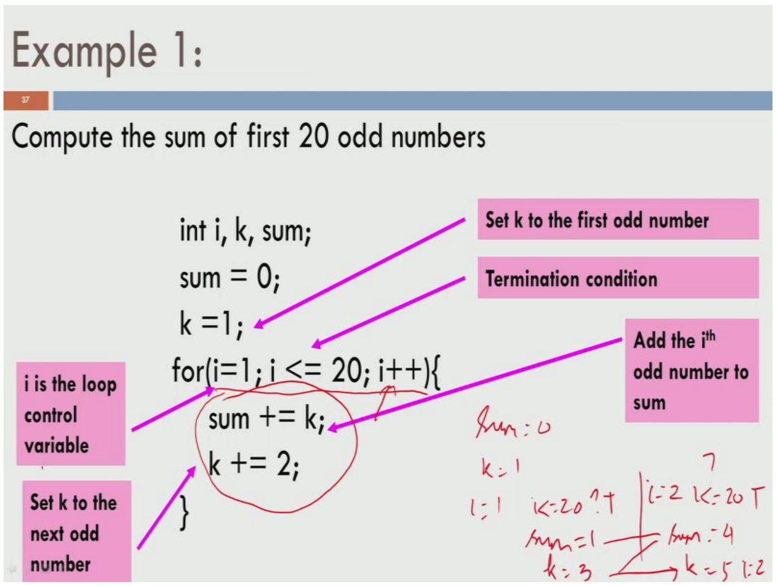
```
for (expr1; expr2; expr3) {
    statement
}
```

Usually a block of code.

- Semantics:
  - Evaluate expr1 – initialization operations
  - Repeat
    - Evaluate expr2
    - If expr2 is true execute statement and expr3
    - Else stop and exit the loop

Say let us look at the basic expression. You have a for loop and it starts… it works like this. So, you have for, which is a key word. And you have these two parenthesis and you have three expressions: expression 1, expression 2, and expression 3. And the body of the loop goes inside this statement here. So, the meaning of this is you evaluate expression 1. So, the expression 1 is just the initialization. And you repeat something. And what you repeat? You evaluate expression 2. If it is true, you execute statement; if it is false, you exit the loop. So, when you execute the statement, you check if the expression 2 is true; if it is true, you execute the statement and evaluate expression 3. If it is false, exit from the loop. Typically, the statement that you have is a block of code. Therefore, usually, you have to put this within braces. So, you need braces here and here typically, because that makes a code block.

(Refer Slide Time: 04:23)



Example 1:

Compute the sum of first 20 odd numbers

```
int i, k, sum;
sum = 0;
k =1;
for(i=1; i <= 20; i++){
    sum += k;
    k += 2;
}
```

Set k to the first odd number

Termination condition

Add the i^{th} odd number to sum

i is the loop control variable

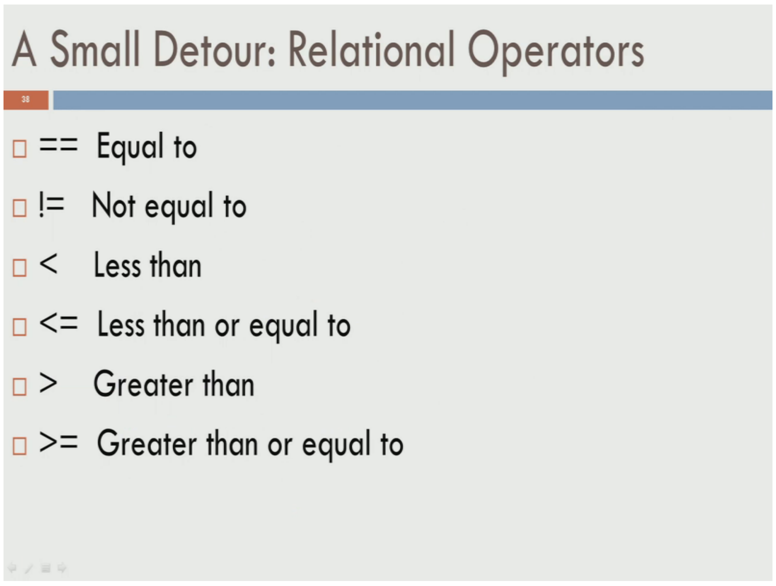
Set k to the next odd number

sum = 0
k = 1
i = 1   i <= 20 ? T | i = 2 i <= 20 T
sum = 1   sum = 4
k = 3   k = 5 i = 2

So, let us take this small example computing the sum of first 20 odd numbers. So, to do so what we are going to do is we are going to have a variable called sum, which is initialized to 0; and to that, we are going to add k repeatedly; and k is just going to be a sequence of odd numbers starting at 1. So, let us see the loop structure in place. If you notice the structure of the for loop, the for loop runs from i equals 1; i less than or equal to 20 i plus plus; which means this body that you are seeing here is going to be executed exactly 20 times. So, let us check what the overall structure of the program is. We set k to be the first odd number, which is 1; and i is the loop iterator or the loop control variable; and i less than or equal to 20 is the termination condition. Since we are incrementing i in terms of 1 using i plus plus, this loop will run exactly 20 times. And the body of the loop is in such a way that, k tracks the i-th odd number; the very first iteration k is 1; and the second iteration – k is supposed to be 3 and so on. It tracks the i-th odd number and we are adding that to sum. And we are incrementing k in increments of 2; which means from 1 we will go 3, 5, 7, 9 and so on. And this is the whole setup.

Let us say we want to mentally simulate this; so initially, sum would be 0 and… Initially, sum is 0 and k is 1. We start the loop; when we start the loop i equals 1 and we are going to check if i is less than or equal to 20; this is actually true. So, therefore, sum plus equal to k. So, this plus equal to is essentially a shortcut for sum equals sum plus k. So, sum is 0. Now, we are adding 1 to it. So, sum becomes 1. And k – we are adding 2 to it. So, k becomes 3. At the end of it, we go and evaluate expression 3, which is i plus plus. So, now, i becomes 2. So, this is one body of the loop. At this point, we again go and check is i less than or equal to 20 or what; it is true. So, now, we add sum plus equal to k. So, we add 1 and 3; sum becomes 4 and k is incremented by 2. So, k becomes 5. And finally, at the end of this iteration, i becomes 2 and so on. So, from… So, essentially, what we are doing is we are incrementing i and we will do this exactly 20 times. So, we will go from number 1, 3, 5, 7 and so on 20 times. So, it is a fairly simple program. But, it shows you the basic structure of a for loop.

(Refer Slide Time: 07:14)

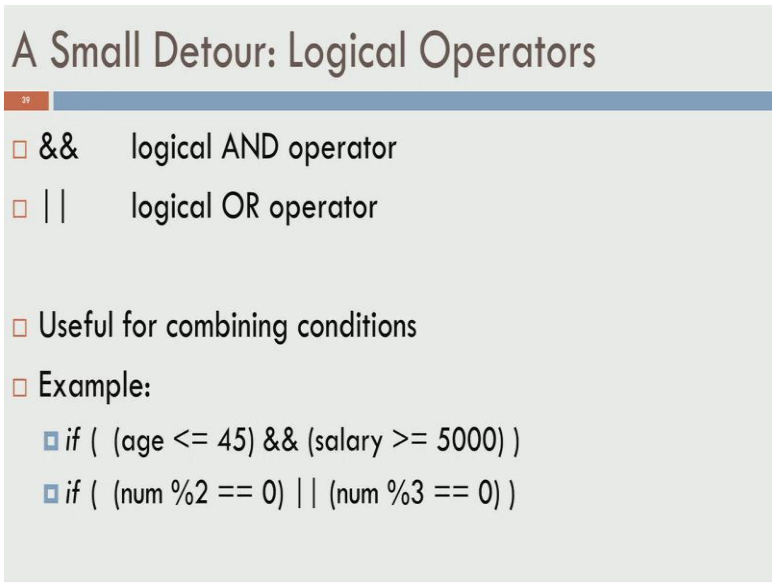


A Small Detour: Relational Operators

- == Equal to
- != Not equal to
- < Less than
- <= Less than or equal to
- > Greater than
- >= Greater than or equal to

So, let us take a small detour and look at what are the different kinds of operators that you can use. We already saw equal to equal to in an earlier lecture. We also saw greater than in the lecture, where we are looking at a largest of the numbers and so on. There are other operators namely, not equal to or less than or equal to, greater than, and so on. So, these are the six operators that you can use on numbers. So, you can use them on both integers and floating point numbers.

(Refer Slide Time: 07:46)



A Small Detour: Logical Operators

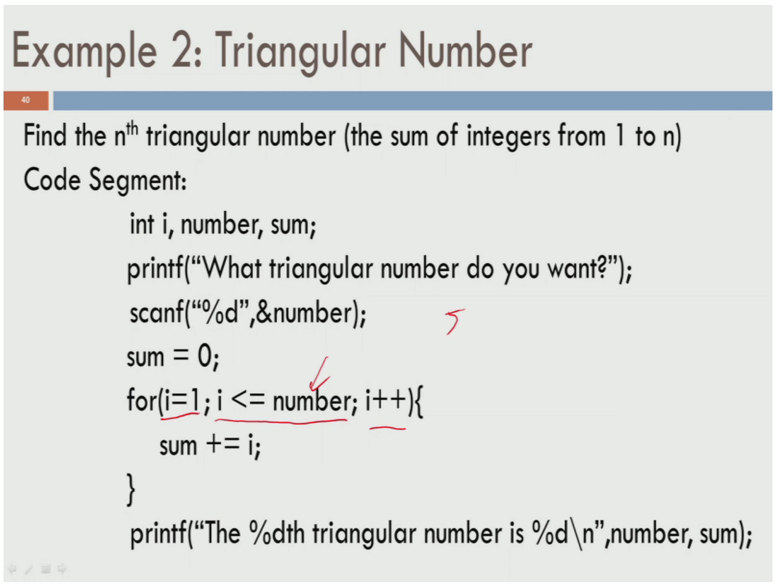
- && logical AND operator
- || logical OR operator

- Useful for combining conditions
- Example:
  - if ((age <= 45) && (salary >= 5000))
  - if ((num %2 == 0) || (num %3 == 0))

But, more importantly, we also need to understand this notion of what are called logical operators. So, in all these things that we have seen so far, we have only one condition for the expression that we are evaluating. Sometimes we need more than a few conditions. So, let us look at this example; where, let us say if age is less than equal to 45 and salary is greater than or equal to 5000, I want to do some work. So, I cannot… So, I want to combine this into one condition and you can do this using this operator called AND. The AND operator when you type, you type ampersand ampersand; and be careful; it is twice; it is not a mistake.

We have to type ampersand twice. That is a logical AND operation. Sometimes I may need something like a logical OR condition; either condition a is true or condition b is true; in which case, we want logical OR operation. And you get that using what is called the pipe operator. So, you press… you have this pipe twice; you usually see this pipe on top of the back slash symbol. So, back slash symbol and pipe are in the same key usually. And for example, if I want to see if a number is a multiple of 2 or a multiple of 3; then I could use this condition num percentage 2 equals 0; checks whether it is a multiple of 2; num percentage three equal to equal to 0 checks whether the number is a multiple of 3. And by sticking in this OR operator in between, I am checking if a number is either a multiple of 2 or a multiple of 3. So, this is a very useful thing; and we may use it in the next few lectures.

(Refer Slide Time: 09:26)



Example 2: Triangular Number

Find the nth triangular number (the sum of integers from 1 to n)

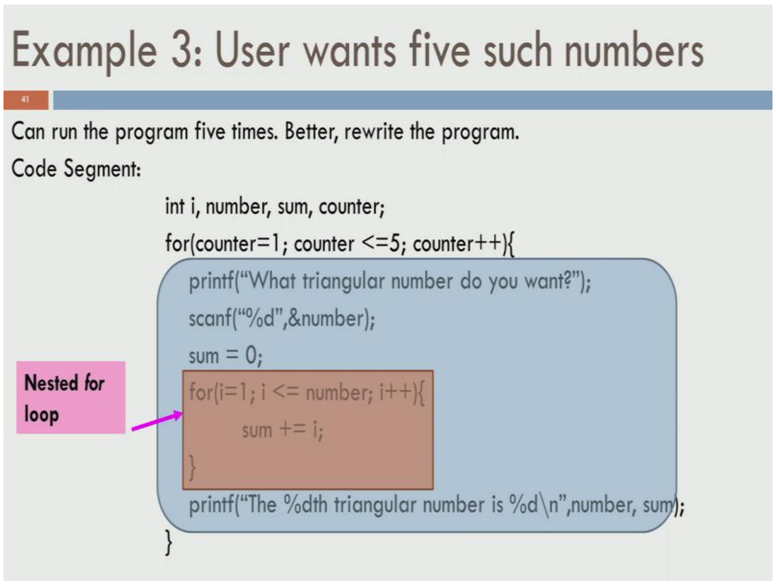
Code Segment:

```
int i, number, sum;
printf("What triangular number do you want?");
scanf("%d",&number);          5
sum = 0;
for(i=1; i <= number; i++){
    sum += i;
}
printf("The %dth triangular number is %d\n",number, sum);
```

So, I want to give another example of a for loop. Let us say I want to print the n-th triangular number. So, in the previous example, the number of iterations was actually known in the program itself. So, we had i equal to 1; i less than or equal to 20. So, I said whenever we will know the number of iterations, it is better to use a for loop. But, I want to show an example here; where, we really do not know the number of iterations, because the user is going to input it. But, we still know that, once the number of iterations – once the user gives n, the number of iterations is fixed. So, let us see this small example. Find the n-th triangular number; a triangular number is defined as a sum of integers from 1 to n. So, I am showing a small segment of a code here. Let us say i as before is the iterator and number is the number of the n-th triangular number that the user is asking for; and sum is the result that we want to give to the user. So, we start with prompting the user what is that triangular number that you want; and we are going to scan it from the user; we will start with sum equal to zero. So, at this point, I want to run it as many times as the number that the user wanted.

So, now, what we have is we have for i equals 1, i less than or equal to number i plus plus. The key thing that I want you to notice is that, this number is not a constant. In the previous example, we had it as a constant; now, it is not a constant. But, the moment this is received from the user, the number of iterations is fixed and that makes it suitable for a for loop. So, for i equals 1, i less than or equal to number; i plus plus – sum plus equals i. So, the way this is going to work is you are going to check if i, which is equal to 1 is less than or equal to the number or not. Let us say the number that the user inputs is 5; 1 is less than or equal to 5; you will add 1 to sum; then i gets incremented to 2; 2 is less than or equal to 5. Then 2 is added to sum and so on. So, you will add numbers 1, 2, 3, 4 and 5. And the result would be 15. So, the fifth triangular number is 15; that would be the print out of this program.

(Refer Slide Time: 11:51)



Example 3: User wants five such numbers

Can run the program five times. Better, rewrite the program.

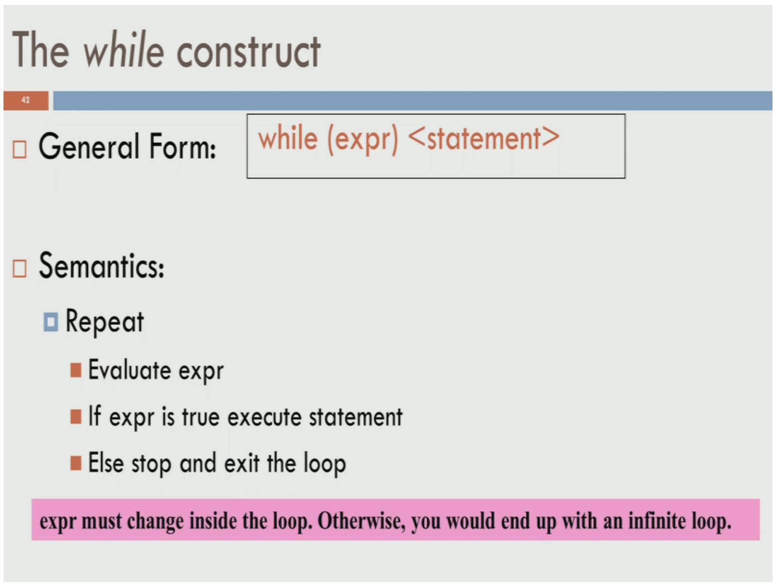
Code Segment:

```
int i, number, sum, counter;
for(counter=1; counter <=5; counter++){
    printf("What triangular number do you want?");
    scanf("%d",&number);
    sum = 0;
    for(i=1; i <= number; i++){
        sum += i;
    }
    printf("The %dth triangular number is %d\n",number, sum);
}
```

Let us look at another example, where the user is going to ask for this multiple times. So, the user is going to ask for sum n-th triangular number. But, he wants five such numbers. So, one way in which you can do this is you can ask the user to run the program five times. But, that is not the most desired way to do it. Instead, if we know upfront that, the user wants five triangular numbers; but, each time, the user may ask for a different triangular number. And that is what we have here. So, we have a for loop here and this for loop is going to take care of the fact that, we are going to ask for five numbers from the user. And within each body of this loop outside, we have this finding the n-th triangular number.

So, we are doing five iterations of asking n from the user and finding the n-th triangular number and printing it. And if you notice, we have this blue body, which is on this outer for loop; and this orange body, which is the inner for loop. So, what we have done is we have nested a for loop inside another for loop. And this is also a very useful thing to do. So, we are having this outermost loop running on counter and the innermost loop running on i. So, we have two iterators. And what this program does is or what this code segment does is – it is going to ask the user five times for what is the triangular number that they want. And each time once the number is entered, it will find the triangular number and print it out. So, I suggest that, you go and write this code and see what happens.

(Refer Slide Time: 13:35)



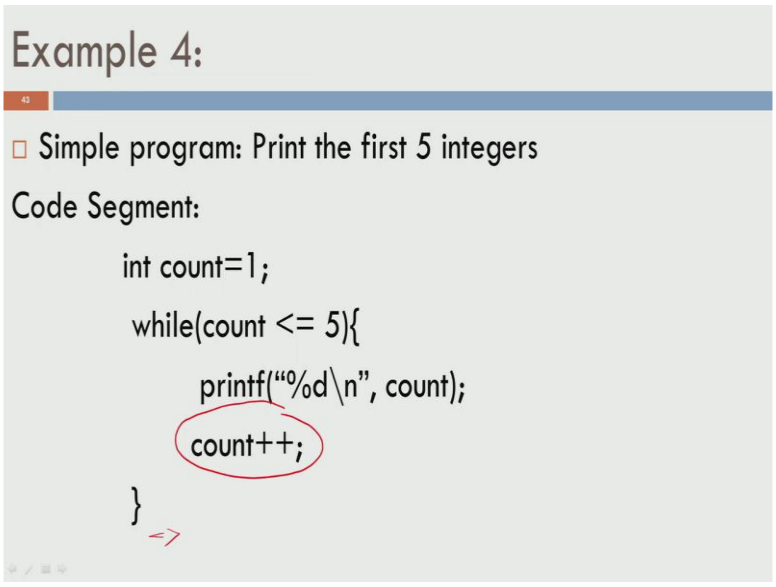
The *while* construct

- General Form: `while (expr) <statement>`
- Semantics:
  - Repeat
    - Evaluate expr
    - If expr is true execute statement
    - Else stop and exit the loop

expr must change inside the loop. Otherwise, you would end up with an infinite loop.

So, let us move on to the next construct, which is the while construct. So, the general form of the while construct is while expression statement. And as before, the statement is a block of code. So, typically, you put that within braces. The semantics is that, you repeat this following process; you evaluate the expression first; if the expression is true, then you execute the statement; if the expression is false, you exit the loop. So, in the for loop, exiting the loop is known upfront; you run it for a certain number of iterations and you exit the loop; whereas, in while loop, you exit once the expression becomes false; which means the expression must change inside the loop. If the expression does not change inside the loop, you would end up within infinite loop, because the expression would always be true; we will keep executing the body of the loop infinitely.

(Refer Slide Time: 14:25)



Example 4:

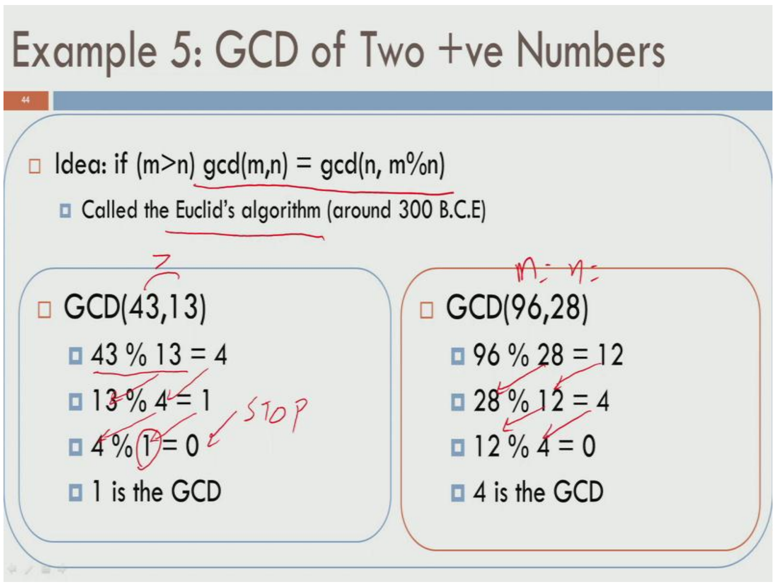
- Simple program: Print the first 5 integers

Code Segment:

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

Let us see a small example. Let us say I want to just print the 5 integers – the first 5 integers – the first 5 positive numbers. So, I can do this with a for loop running from 1 to 5 and printing it. I am showing a small code segment, which does something different. So, I have int count equals 1. So, right then we declare the initialized count equal to 1. This is possible in C. Then while count is less than equal to 5; print count and count plus plus. So, the key thing that you have to notice is that, this count that we have here is actually changing. If we do not have this, count would be 1 forever while 1 is less than equal to 5; print count. We would be printing 1 in an infinite loop. However, this count plus plus is changing the value of count. So, the next time you would check if 2 is less than or equal to 5; then you would check if 3 is less than or equal to 5 and so on. At some point, count would become 6; 6 less than or equal to 5 would become false. At that point, you exit the loop and you move to this location. You exit the while loop. So, this is a natural use for a while loop. So, this can be returned using a for loop; but, this is a simple example to show how the while loop works.

(Refer Slide Time: 15:45)



Example 5: GCD of Two +ve Numbers

- Idea: if (m>n) gcd(m,n) = gcd(n, m%n)
  - Called the Euclid's algorithm (around 300 B.C.E)

- GCD(43,13)
  - 43 % 13 = 4
  - 13 % 4 = 1
  - 4 % 1 = 0
  - 1 is the GCD

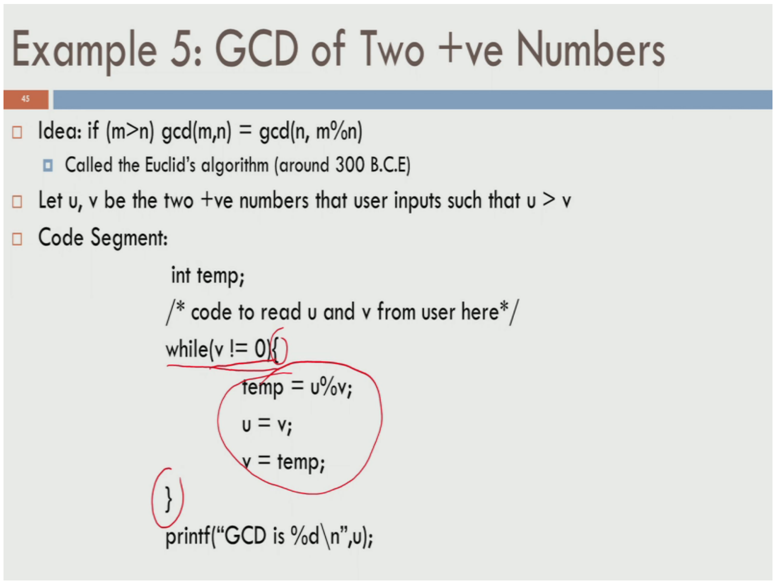
- GCD(96,28)
  - 96 % 28 = 12
  - 28 % 12 = 4
  - 12 % 4 = 0
  - 4 is the GCD

Let us see a more concrete example; where, the number of iterations is actually not known. So, the problem that I am going to pose is finding the greatest common divisor of two positive numbers. Let us say the user gives two positive numbers. Let us also assume that, m is greater than n. There are two numbers: m and n that are given to you. And let us assume that, m is actually greater than n. If it is not, you can ask the user to give it in that order or you could change m and n, so that m actually become greater than n. That is not the crux, but I want you to look at this notion of GCD. We are going to use an algorithm called the Euclid's algorithm. And this dates back to 300 years before BC. So, it is a very old algorithm and it is a very neat algorithm. So, if I want to find out GCD of two positive numbers, then GCD of m comma n is the same as GCD of n comma m percentage n; where, percentage stands for modulo.

So, I have two examples below to show that. So, let us say I want to find out GCD of 43 and 13. See in this case, m is 43 and n is 13 and m is actually greater than 13; so the condition that we want to satisfy. We start with 43 percentage 13. So, 43 percentage 13 is 4, because 13 times 3 is 39. So, 43 divided by 13 gives a reminder of 4. And now, you take 13 and the reminder 4; you do 13 percentage 4; that is 1. And then you take 4 percentage 1; this is 0. So, at this point, you stop and you report this as the GCD. So, 43 and 13 are both prime numbers. There is no common factor. So, the common factor is indeed 1. Therefore, 1 is the GCD.

Let us look at a difference example; let us say m equals 96 and n equals 28. We start with 96 percentage 28. So, 28 times 3 is 84. Therefore, 12 is the reminder. Then you take 28 modulo – the reminder that you got in the previous iteration. 28 percentage 12 is 4; and 12 percentage 4 is 0. And therefore, 4 is the GCD. So, even though in both these examples, we do know that, the number of iterations is 3 because we have worked it out; upfront you may not know what is the number of iteration that is required? So, given two numbers, you do not know the number of iterations that is required upfront. In such cases, the while loop is a natural choice.

(Refer Slide Time: 18:34)



Example 5: GCD of Two +ve Numbers

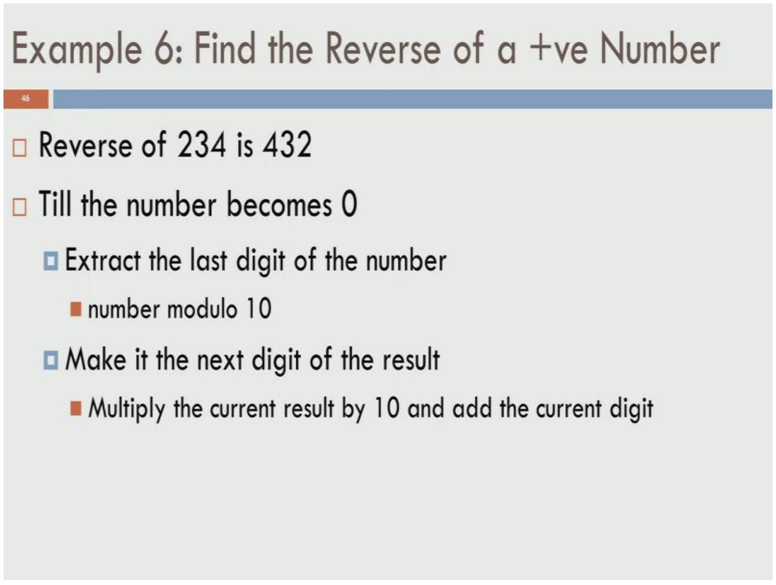
- Idea: if (m>n) gcd(m,n) = gcd(n, m%n)
  - Called the Euclid's algorithm (around 300 B.C.E)
- Let u, v be the two +ve numbers that user inputs such that u > v
- Code Segment:

```
int temp;
/* code to read u and v from user here*/
while(v != 0){
        temp = u%v;
        u = v;
        v = temp;
}
printf("GCD is %d\n",u);
```

So, let us see this basic code. We start with if m is greater than n; if m is greater than n, GCD of m comma n is GCD of n comma m percentage n. This is what we wanted to do. And the code segment is as follows. So, if v is already 0; then there is nothing to do; however, otherwise, what we do is we have a temporary variable called temp; we find out u percentage v, which is u modulo v; that goes as temp. And once you do that, u becomes v itself and v becomes temp; which is what we did here. So, it is called m and n here; it is called u and v in the program. So, what we did was we took the reminder and we will call it v. And take the previous divisor; call it u and we repeat this. At the end of it, v will become 0 at some point of time and you report u as the GCD.

So, now, let us pay attention to the construct itself and not the logic of the program. So, what we have is we have while v is not equal to 0. So, we keep doing this till we violate this condition. And we have two braces here indicating that, this is a body of the loop. And this body of the loop – upfront we do not know the number of iterations; we can see that, there is nothing, which is doing an iteration count; however, we are hoping that, v will become 0 at some point of time. So, v does not become 0 at any point of time. This while loop will become an infinite loop; but, because of the property of the numbers and the operations that we are doing here, v will indeed become 0 at some point of time and u is the GCD of two numbers. So, again I suggest that, you go and try the different examples and ensure that, this is actually correct; we go and verify that, this algorithm is actually correct. So, finally, I give one last example for a while loop.

(Refer Slide Time: 20:41)



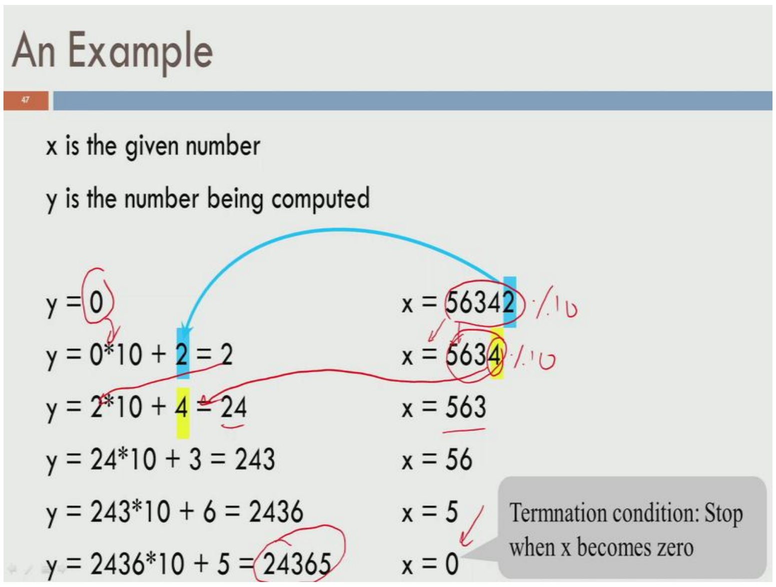
Example 6: Find the Reverse of a +ve Number

46

- Reverse of 234 is 432
- Till the number becomes 0
  - Extract the last digit of the number
    - number modulo 10
  - Make it the next digit of the result
    - Multiply the current result by 10 and add the current digit

So, again in this, at this point, we do not know the number of iterations that we are going to execute. So, the problem is finding the reverse of a positive number. So, if I want the reverse of the number 234, it is 432. So, I need three iterations to find out the reverse. But, I could give a number, which is 19574 and this would be a five digit number; you need five iterations. In general, if I give you an n-digit number, you need n iterations; upfront you do not know the number of digits that the user is going to give. So, the algorithm that we are going to follow is as follows. So, till the number becomes 0, extract the last digit of the number by finding out the number modulo 10. See you take a number; find number modulo 10; that gives you the last digit. Make it the next digit of the result by multiplying the result by 10 and adding the current digit. So, I will show you simple example to make this clear.

(Refer Slide Time: 21:38)



An Example

x is the given number

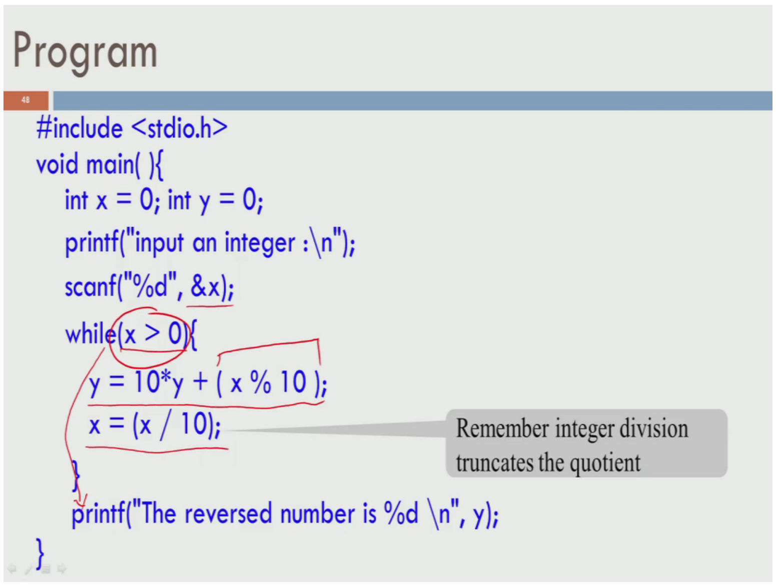
y is the number being computed

y = 0	x = 56342 %10
y = 0*10 + 2 = 2	x = 5634 %10
y = 2*10 + 4 = 24	x = 563
y = 24*10 + 3 = 243	x = 56
y = 243*10 + 6 = 2436	x = 5
y = 2436*10 + 5 = 24365	x = 0

Termnation condition: Stop when x becomes zero

Let us say I want to… Let x be the given number and y be the number that is being computed. Let us say the user input 56342 as the number, which has to be reversed. So, 2 is the first digit of the resultant number. We want 2 to be the first digit of the resultant number. The way we do that is we take 56342 and do modulo 10; that will get the reminder 2 alone. It will extract only the last digit. You take that; multiply the previous one by 10 and add this to it; that will result in 2. And what you do is you take 56342; divide that by 10 to get 5634. What I mean by divide is you get only the quotient and not the fractional part. So, 56342 divided by 10; the quotient is 5634. At this point, you do percentage 10 again; you will get only 4. You take 4 and add it to two times 10. The number is 24. So, so far, we have taken the last two digits and reversed it. Then you take 5634 itself and again divide it by 10 to get 563 and so on. If you keeping doing this, at some point, you get 24365, which is the reverse of 56342. But, at that point, x becomes 0 and it is a termination condition. So, you will keep doing this till x becomes 0; and this is a natural way of using a while loop.

(Refer Slide Time: 23:19)



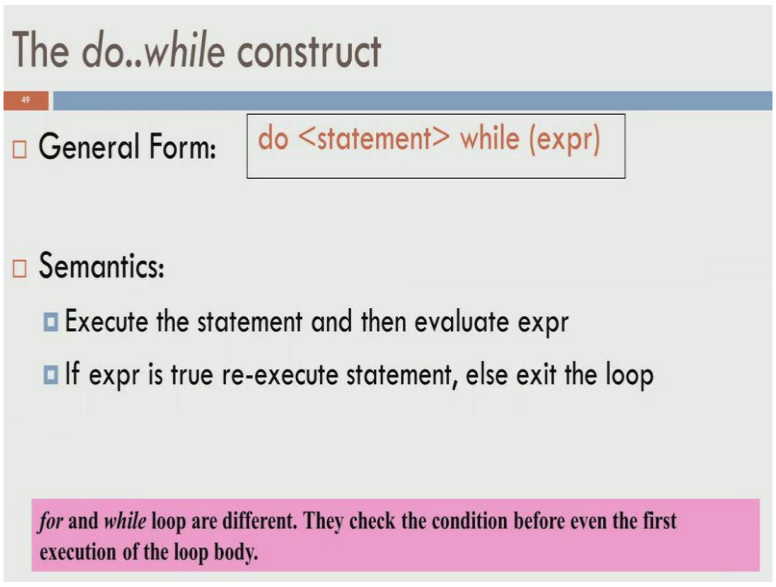
Program

```
#include <stdio.h>
void main( ){
   int x = 0; int y = 0;
   printf("input an integer :\n");
   scanf("%d", &x);
   while(x > 0){
      y = 10*y + ( x % 10 );
      x = (x / 10);
   }
   printf("The reversed number is %d \n", y);
}
```

Remember integer division truncates the quotient

So, let us look at the program itself. We have x equals 0 and y equals 0; and you expect the user to give an input number. So, input an integer and you scan it from the user. If the user inputs 0 itself, you check the condition while x is greater than 0. Let us say the user gives 0 as the input; then the while loop is not even executed once; you can directly go and print y equals 0. So, we also assume that, the user inputs only positive numbers or 0. So, if the user inputs 0, this while will not be executed. Therefore, the reverse number is 0 itself. However, if the user inputs any number greater than 0; then we have y equals 10 times y plus x mod 10. So, you remember the evaluation order; we have parenthesis here; this will be evaluated first; and that is doing modulo 10 of x; that extracts the last digit, but it adds it to 10 times y. And x itself is diminished by a factor of 10. Remember – x being an integer – the integer division, you have integer divided by integer; that gives you only the quotient and it truncates the fractional part. So, x keeps diminishing by a factor of 10 every time. At the end of it, you have x equals 0. At that point, this while condition is not true anymore; when it becomes false, you come to the end of the loop and you print the reverse number. So, this is a full program segment. If you type it in your editor and compile, it should work.

(Refer Slide Time: 25:10)



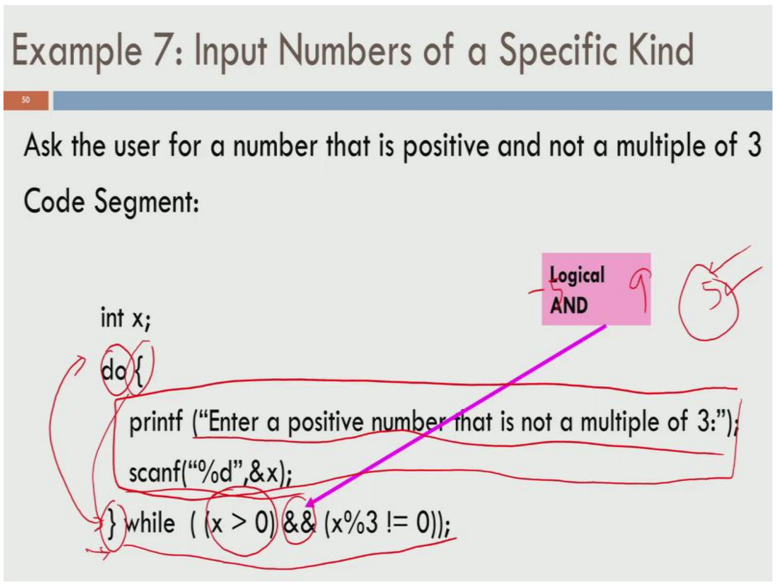
The do..while construct

- General Form: do <statement> while (expr)
- Semantics:
  - Execute the statement and then evaluate expr
  - If expr is true re-execute statement, else exit the loop

for and *while* loop are different. They check the condition before even the first execution of the loop body.

Let us move to the third construct, which is a do-while construct. And the general form is do statement – while expression. The semantics is as follows. Execute the statements and then evaluate the expression. So, the key concept is that, you execute it once and then you evaluate the expression. If the expression is true, you re-execute the statement; otherwise, exit the loop. So, this do-while construct is different from the for and while construct. The for and while constructs – both check the condition once even before executing the body of the loop even once. However, in do while, you execute the loop body at least once before you go into checking the condition.

(Refer Slide Time: 25:52)



Example 7: Input Numbers of a Specific Kind

Ask the user for a number that is positive and not a multiple of 3

Code Segment:

```
int x;
do {
    printf ("Enter a positive number that is not a multiple of 3:");
    scanf("%d",&x);
} while ( (x > 0) && (x%3 != 0));
```

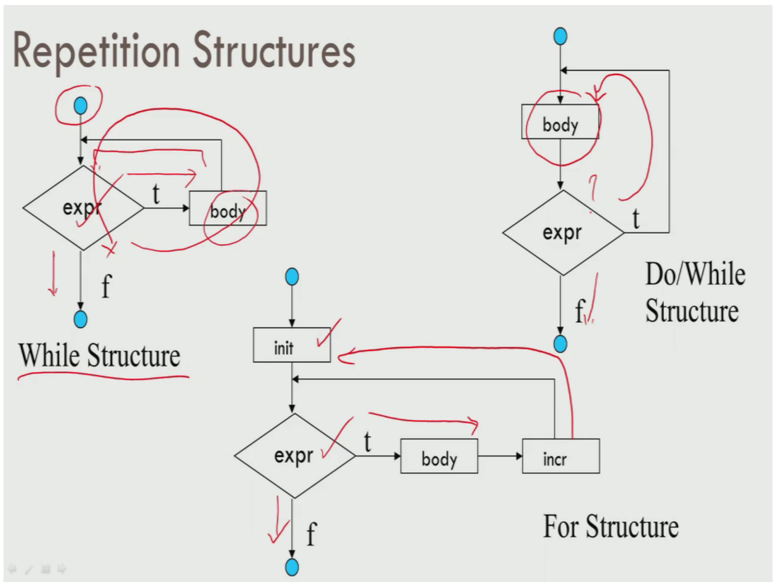
Logical AND

So, let us look at a specific example, where do while is natural. Let us say I want the input numbers to be of a specific kind. So, I showed you an example earlier; where, we were checking if the number that is input by the user is a multiple of 3 or not. If it is not, we prompted the user to give this input once more and we scanned it. But, we had no guarantee that, the user actually input a number, which is a multiple of 3; which means we have to repeatedly keep asking the user till the user actually inputs a multiple of 3. So, this is a natural thing to do with a do-while construct. So, let us see what is happening in this setup here.

Let us as before, look at the logic first and then we look at the structure itself. So, the very first time you have a do, there is nothing to check; you prompt the user by printing this on the screen; and the user is expected to enter a number. Let us say the user inputs minus 5; then after this, you go and check the loop. So, now, we see an example of ampersand ampersand or the logical AND. So, while x is greater than 0 and x percentage 3 not equal to 0; so the user let us say entered minus 5; minus 5 is not greater than 0. So, this condition evaluates to false and this condition also is actually true. So, minus 5 mod 3 is not equal to 0; however, since this evaluate to false and that evaluates to true, the overall expression evaluates to false. Therefore, the user has to enter another number. At this point, you go back here and there is nothing to check; you go and print this on the screen once more; you scan the user input once more.

Let us say the user now input 9. If the input is 9, x greater than 0 would be true; however, x mod 3 would still be 0. Therefore, this while condition is still false; you would go back and do the print f statement. Let us say now the user inputs 5. So, we did minus 5 first; we did 9 next. Let us say the user inputs 5 now. And at this point, 5 is greater than 0 is true and 5 mod 3 is not equal to 0, is also true. Therefore, 5 is a valid user input. At this point, the condition evaluates to true; and therefore, you stop here. So, you keep doing this till a certain condition is satisfied. So, let us look at the construct itself; we have the key word do followed by a block of code; and the body of the loop is this one. So, that is the overall structure for the setup here.

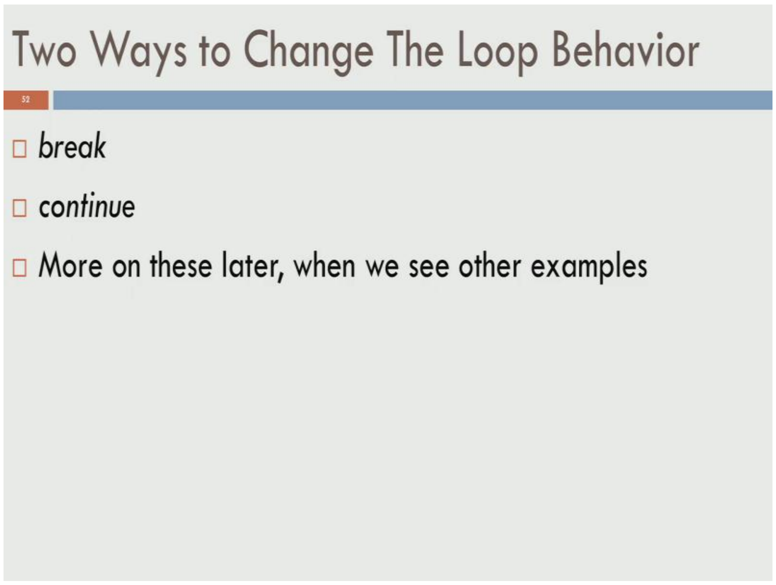
(Refer Slide Time: 28:54)



So, let us look at this in summary. In the for loop, what we have is we start with an initialization condition; we first check an expression; if it is true, you evaluate the body; you do some loop increment and go back. And when you go back, you have to evaluate the expression once more. At some point, this loop increment that you are doing will violate the expression; at that point, you exit; and that is the false branch. If you look at the while loop, the while loop starts with some body and it checks the expression first; if it is true, you execute the loop body and comeback and evaluate the expression; if it is false, you exit the loop. So, the key thing to notice is that, this body that you have here must contain something that will change the evaluation of the expression. If it does not change the evaluation of expression, you will keep doing this loop forever, which we want to avoid.

And this picture clearly shows how do while is different from both these. We have a body of the loop that is executed even before the expression is checked once. So, once you execute the body, you then check expression; if it is true, you go back and execute the body once more; if it is false, you exit the loop. And these are the three basic constructs that is available in C. And depending on the problem, you need to pick one appropriately. So, in summary, for loops are used whenever you know the number of iterations that you have to run the body of the loop; a while loop is used if you want to check on a condition at least once even before you get to the body, but you do not know the number of iterations that you need upfront; and do while is used if you want to execute a body at least once even before you check the expression, but you do not know the number of times the body is supposed to be executed.

(Refer Slide Time: 31:06)



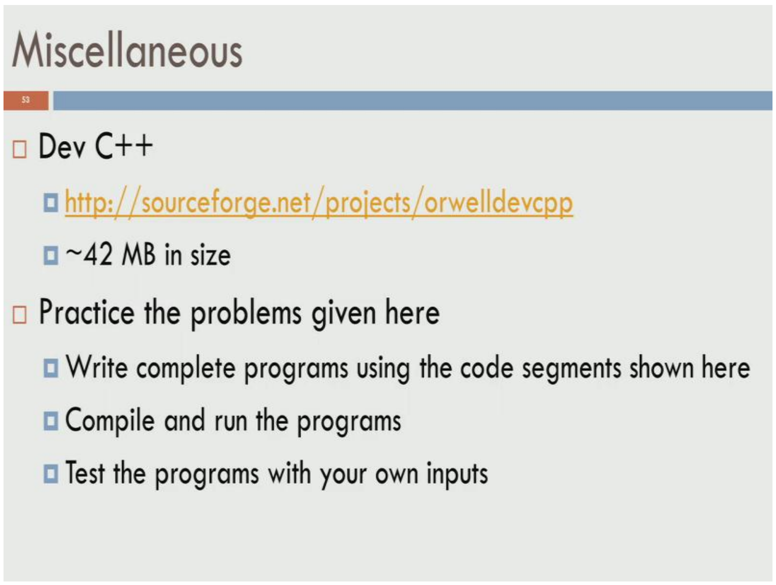
Two Ways to Change The Loop Behavior

52

- *break*
- *continue*
- More on these later, when we see other examples

So, there are two ways to change the loop behavior. There are two key words namely, break and continue. I am not talking about this in this module; we will see this later as we see the need for them. So far, I have been using dev C plus plus to show you program segments; I promised earlier that, I will actually point out where you can get this dev C plus plus.

(Refer Slide Time: 31:35)



Miscellaneous

53

- Dev C++
  - http://sourceforge.net/projects/orwelldevcpp
  - ~42 MB in size
- Practice the problems given here
  - Write complete programs using the code segments shown here
  - Compile and run the programs
  - Test the programs with your own inputs

It can be found at sourceforge website – at sourceforge dot net slash projects slash orwelldevcpp. It is roughly about 42 mega bytes in size; you need to download the compiler, the debugger, the graphical environment and so on. It takes about 42 MB in size. I suggest that, you download and start using that. And it takes a few minutes to set up and not more. So, that is a very useful thing to download. So, of course, there are several other development environments; if you are already using some IDE, go ahead and use it. If you are coming from the Linux world, you are probably used to the shell and you very likely have a GCC or some such compiler installed already; you can go ahead and use those also if you are familiar with them. But, since I am showing a demo using dev C plus plus, it may make sense to actually use it, so that you can track what you are doing. So, this is especially useful if you are new to programming.

I suggest that, you actually practice the problems that I have given so for. At times, I have given complete problem statements; at times I am giving only code segments; but, you have to go and write a complete program. You write complete programs whenever you can. And whenever it is needed, run the programs; test with your own inputs and ensure that, the programs that I have return so far are indeed correct. So, it is possible that, I have made some mistakes somewhere along. So, it is better that, you go and check these answers once before you move on to the next lecture. So far, in this week, we have seen basics of programming in C; we saw the notions of variables; we saw the if statement; we saw the switch statement; and we saw for while and do while statements. We also saw basic operators and how to print and scan inputs. From next week onwards, we will see more sophisticate uses of these constructs in solving different kinds of problems. So, we are at the end of lecture 2.

Thank you.