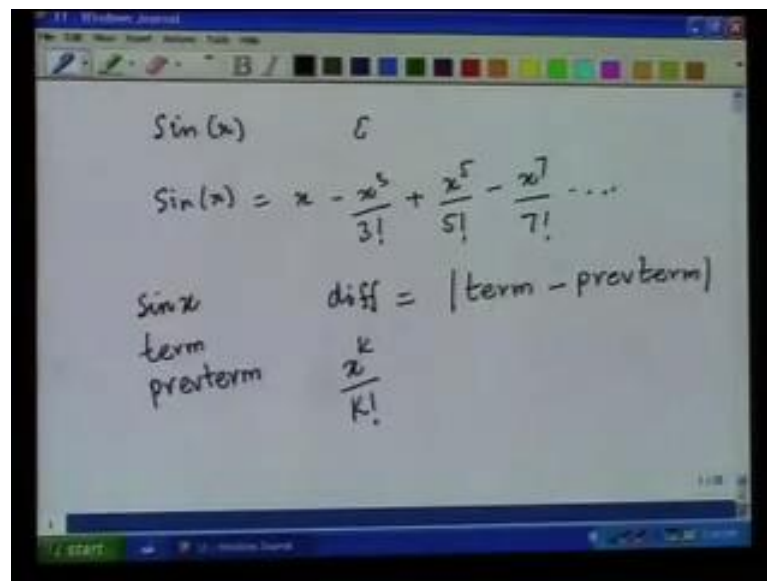


Introduction to Problem solving, and Programming
Prof. Deepak Gupta
Department of Computer Science Engineering
Indian Institute of Technology, Kanpur

Lecture – 6

(Refer Slide Time: 02:37)



The image shows a digital whiteboard with the following content:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

Below the expansion, the difference between terms is defined:

$$\text{diff} = |\text{term} - \text{prevterm}|$$

A table-like structure shows the relationship between terms:

term	prevterm
$\frac{x^k}{k!}$	$\frac{x^{k-1}}{(k-1)!}$

So before we start a new topic, today let us look again at the problem that I have given you at the last time to solve it the problem, if you recall was compute the value of $\sin x$ given the value of x . To a certain accuracy ϵ , and to compute this you have to use the Taylor series expansion for the $\sin x$ which I given is x minus x cube by three factorial plus x power 5 by 5 factorial etcetera.

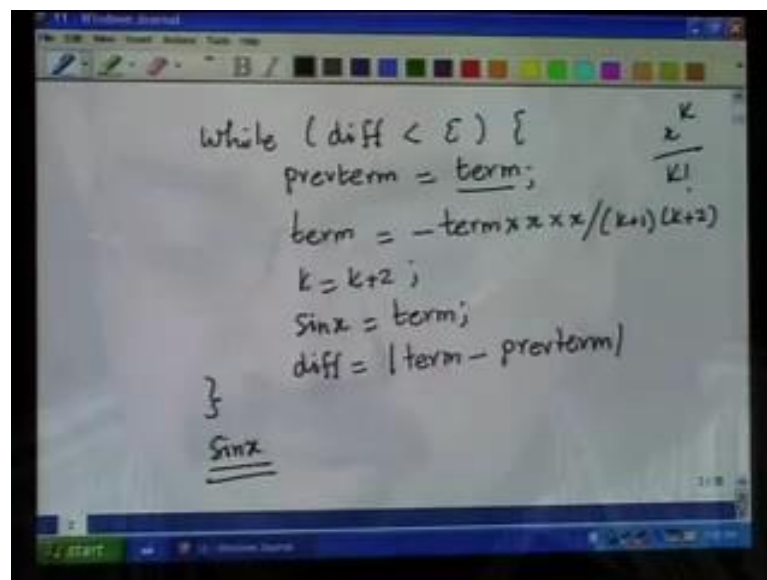
So, how do we solve these problem well as I said we just keep computing terms of this series 1 by one, and keep adding them to a sum which will be the final value of $\sin x$, and this is an infinite series we will have to stop the solution when the absolute difference of 2 consecutive terms becomes less than ϵ . So, before looking at the program lets try to work out an formal algorithm for doing this. So, let us say will we use this variable $\sin x$ will carry the some of the terms which will finally, of course, give the value of the \sin of x , and the term is the most recently computed term the last computed term in the series that we are computed. And let us say let us use the variable prevterm for the term prev to the most recently term. So, the difference between the 2 term is $\text{term} - \text{prevterm}$

prevterm, and the absolute value of that is the difference. So, we will have to keep computing till the value of difference becomes less than epsilon.

Now, 1 thing that you should note in this series is that when I am computing that x to the power k over k factorial where k is some odd number, then I do not have to compute x to the power k , and k factorial all over again. Because if I do that I have to multiply x k times, then also multiply k by k minus 1 k minus 2 etcetera up to 1 which is the number of the multiplication if. So, that you can use the fact that we already know what the previous term was.

So, previous term that we have computed was x to the power k minus 2 divided by k minus 2 factorial. So, if we just multiplied this by x twice, and divided by k minus 1 times k that is the value for the next term. So, let us now try to write an informal algorithm using this idea. So, the main loop we will look something like this while difference is less than epsilon. So, the term that you computed in the previous iteration will become the previous term for the iteration.

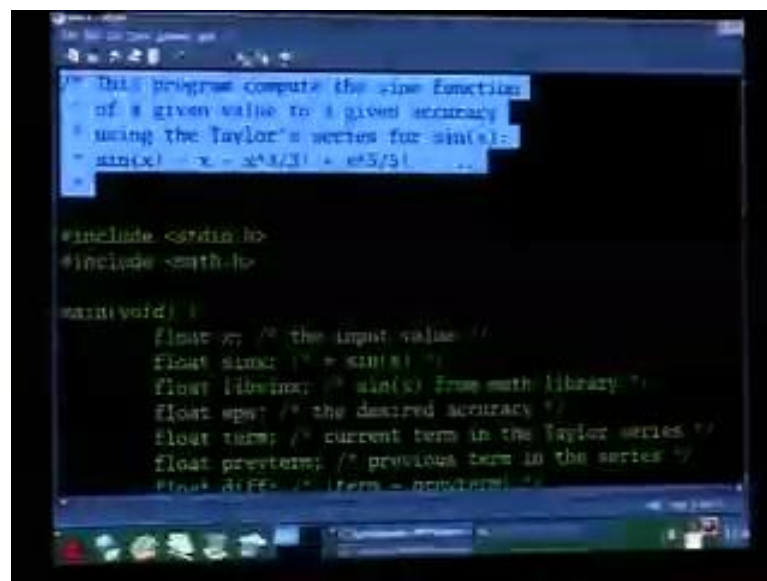
(Refer Slide Time: 04:49)



So, previous term will become equal to the current value of the term, and new value of term will have to compute its assume that the previous value of the term use the value k that the previous the term that we computed earlier was k power k by k factorial, then the new value of term would be minus term into x into x divided by k plus one, and k plus two.

This minus comes, because if you observe the term in the have alternative sign plus, and minus, and after this the value of k will become k plus 2 we get incremented by k, and to the value of sinx we will add the value of term, and what else we need to do well we need to update the value of difference. So, difference will now be the new value of the difference which term minus the previous term, and that is it, and at the end of this the value of sinx is the desired output.

(Refer Slide Time: 05:01)

A screenshot of a code editor showing a C program. The program includes comments at the top explaining its purpose: to compute the sine function using Taylor's series. It includes headers for stdio.h and math.h. The main function declares several float variables: x (input value), sinx (result), libsinx (reference from math library), eps (accuracy), term (current term), preterm (previous term), and diff (difference between terms). The code is partially visible, showing the initial setup and the start of a loop structure.

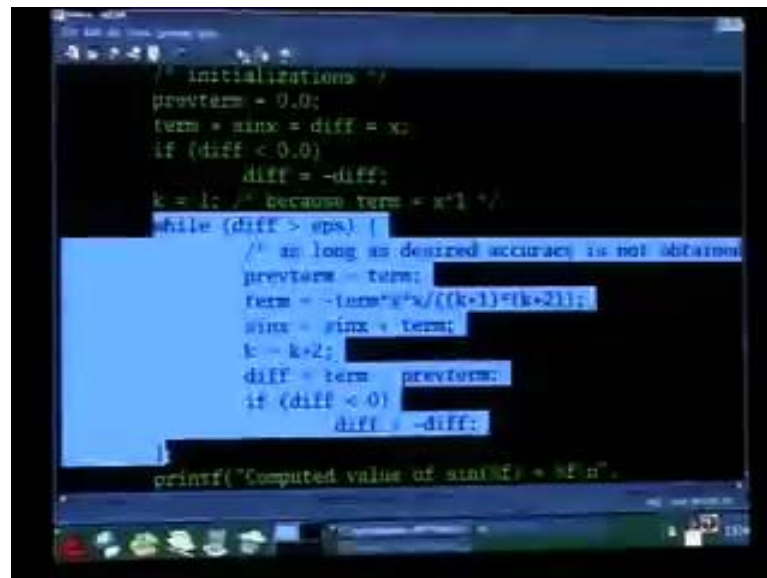
```
/* This program compute the sine function
of a given value to a given accuracy
using the Taylor's series for sin(x):
sin(x) = x - x^3/3! + x^5/5! - ... */

#include <stdio.h>
#include <math.h>

main(void) {
    float x; /* the input value */
    float sinx; /* = sin(x) */
    float libsinx; /* sin(x) from math library */
    float eps; /* the desired accuracy */
    float term; /* current term in the Taylor series */
    float preterm; /* previous term in the series */
    float diff; /* term - preterm */
```

So let us now look at the C program to do all these here the C program to compute sinx note that we have a big comment at the front, which says what the program does, and how it does it if you remember I told earlier that writing comments in the program is very important, because it will help us understand the program even the person who himself written the program may need to understand some difficult parts of written comments, and we can go that before we look at the other things in the program lets look at the main loop of the program here is the main loop.

(Refer Slide Time: 05:30)

A screenshot of a C program in a terminal window. The code implements a Taylor series for sine. It starts with initializations for prevTerm, term, sinx, and diff. A while loop continues as long as the absolute difference is greater than a small epsilon. Inside the loop, the previous term is assigned to term, a new term is calculated using the formula $term = -term * x^2 / ((k+1)*(k+2))$, sinx is updated by adding the new term, k is incremented by 2, and the difference is updated as term minus prevTerm. The final result is printed as the computed value of sin(x).

```
/* initializations */
prevTerm = 0.0;
term = sinx = diff = x;
if (diff < 0.0)
    diff = -diff;
k = 1; /* because term = x^1 */
while (diff > eps) {
    /* as long as desired accuracy is not obtained
    prevTerm = term;
    term = -term*x*x/((k+1)*(k+2));
    sinx = sinx + term;
    k = k+2;
    diff = term - prevTerm;
    if (diff < 0)
        diff = -diff;

    printf("Computed value of sin(x) = %f\n", sinx);
```

And it quiet quiet same as what we just discussed while the difference is greater than the value of epsilon that is as long as the desired accuracy is not obtained you assign previous term is assigned to the term. Because the most recent computed term will now become previous term, and the user will computed the new term is computed from the last computed term is minus term times x square divided by k plus time k plus 2.

Note that the expression k plus 1 times k plus 2 has to be enclosed in an extra set of brackets that is, because the otherwise it will become equivalent to term into x square divided by k plus one, but multiplied by k plus 2 instead of multiplication symbol instead of division by k plus 2.

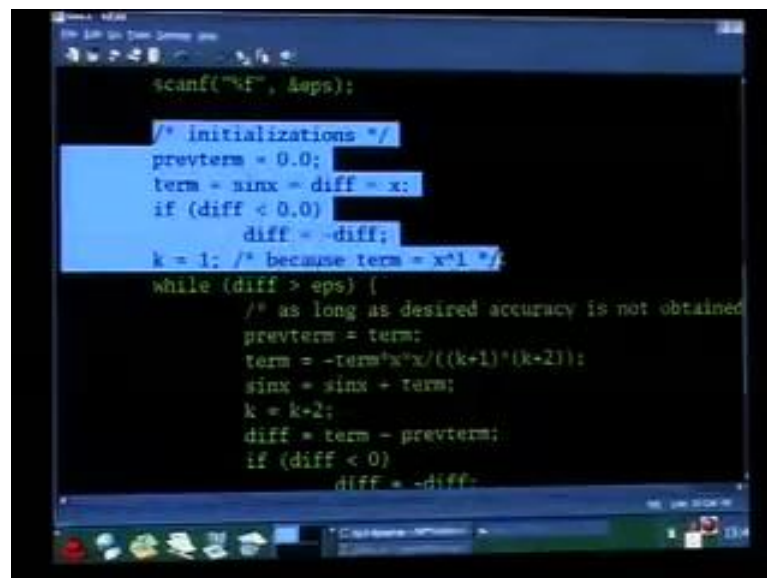
That is, because really the division, and the multiplication operator have the same what is known as precedent which we talked about in detail later on, and essentially they are carried out in the left to right order, but we will not worried about this lets for the time being.

Let us just put extra brackets here. So, the value of sinx gets updated by the value of this new term just get added to the old value k becomes k plus 2 the new differences is term minus previous term, but remember that we want the absolute value of the difference. So,

how do we compute that that is straight forward if the value of distance turns out to be less than 0, then we simply changes the sign.

Let us now look at the initializations set we need to perform to start of this group let us assume that we already read the value of x, and epsilon above these statement in the program I will show them to you in a minute. So, here are the initialization.

(Refer Slide Time: 07:26)



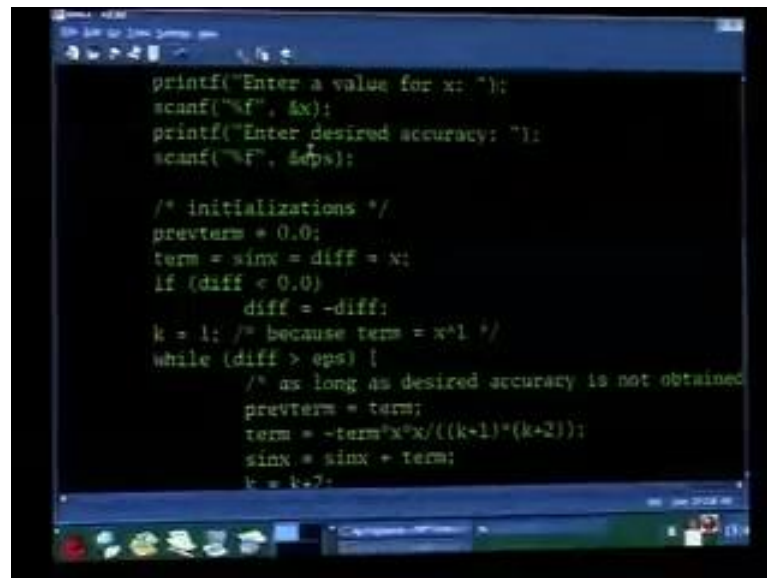
```
scanf("%f", &eps);

/* initializations */
prevterm = 0.0;
term = sinx = diff = x;
if (diff < 0.0)
    diff = -diff;
k = 1; /* because term = x^1 */
while (diff > eps) {
    /* as long as desired accuracy is not obtained
    prevterm = term;
    term = -term*x*x/((k+1)*(k+2));
    sinx = sinx + term;
    k = k+2;
    diff = term - prevterm;
    if (diff < 0)
        diff = -diff;
```

So, we know the first term is x. So, therefore, we can assign the term to be x, and the initial value of sinx is also x, because the term is already included in that the prevterm the 1 before this is of course, 0 therefore, the value of previous term initialize to zero.

And the value of difference is also x, because term is x, and the previous term is zero. So, difference is term minus previous term which is x, but of course, this x can be negative. So, this difference can be negative as well and... So, therefore, this difference is less than 0 we change the sign of the difference. And because the current value of the term is x to the power 1 divided by 1 factorial therefore, k should be initialized to 1 let now see what else do we need to do this is the standard set of statement for reading some input. So, we want to read the value of x, and the value of epsilon.

(Refer Slide Time: 08:22)

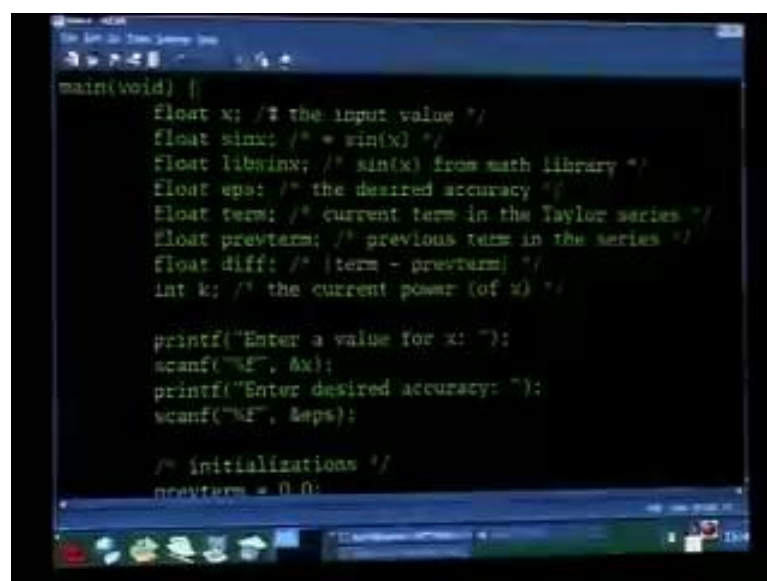
A screenshot of a C program in a Windows command prompt. The program prompts the user to enter a value for x and a desired accuracy (epsilon). It then calculates the sine of x using a Taylor series approximation. The code includes initializations for prevterm, term, and diff, and a while loop that continues until the difference between terms is less than epsilon. The loop updates prevterm, term, and diff, and increments k.

```
printf("Enter a value for x: ");
scanf("%f", &x);
printf("Enter desired accuracy: ");
scanf("%f", &eps);

/* initializations */
prevterm = 0.0;
term = sinx = diff = x;
if (diff < 0.0)
    diff = -diff;
k = 1; /* because term = x^1 */
while (diff > eps) {
    /* as long as desired accuracy is not obtained
    prevterm = term;
    term = -term*x*x/((k-1)*(k+2));
    sinx = sinx + term;
    k = k+2;
```

So, a prompt enter value for x read the value of x using scanf percent f note that since x is a simply floating point number of x float. Therefore we use percent f here, and similarly we use percent f to read the value of epsilon here are the declarations of the various variables.

(Refer Slide Time: 08:45)

A screenshot of a C program in a Windows command prompt. The program starts with a main function that declares variables for x, sinx, libsinx, eps, term, prevterm, diff, and k. It then prompts the user to enter a value for x and a desired accuracy (epsilon). The code includes initializations for prevterm and term, and a while loop that continues until the difference between terms is less than epsilon. The loop updates prevterm, term, and diff, and increments k.

```
main(void) {
    float x; /* the input value */
    float sinx; /* = sin(x) */
    float libsinx; /* sin(x) from math library */
    float eps; /* the desired accuracy */
    float term; /* current term in the Taylor series */
    float prevterm; /* previous term in the series */
    float diff; /* |term - prevterm| */
    int k; /* the current power (of x) */

    printf("Enter a value for x: ");
    scanf("%f", &x);
    printf("Enter desired accuracy: ");
    scanf("%f", &eps);

    /* initializations */
    prevterm = 0.0;
```

X is the input value note that in the program I have given a comment for explaining every variable use in the program this helps in understandability of the program.

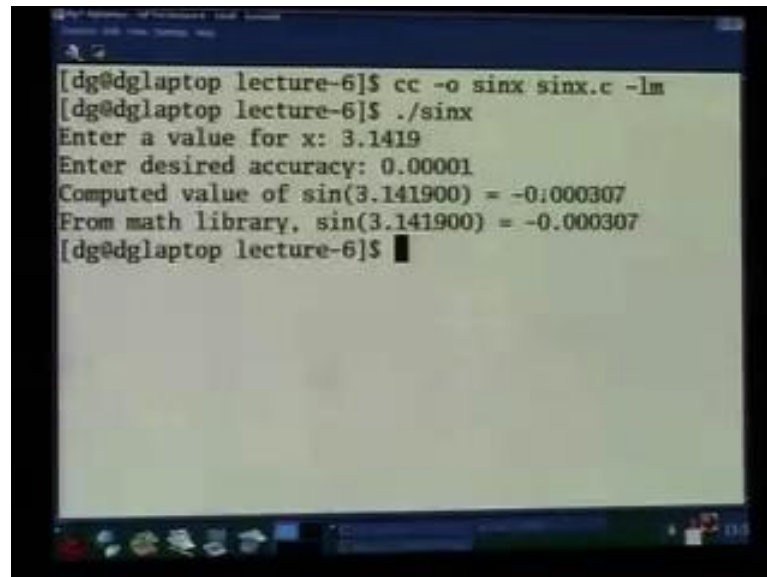
So, x is the value that is given to be input to us variable `sinx` will hold the value of \sin of x this is comment says `sinx` will be equal to \sin of x, we can not use \sin of x with bracket itself as the variable name, because remember that variable name cannot contain bracket character.

So, just to verify that our answer is correct we are also going to use the math library functions `sinx` which is built in into the c math library just to verify that our answer is correct. We also compute the value using `sinx` function from the math library to the variable `lib sinx` is going to contain the value of `sinx`, obtain using the math library sine function `epsilon` is of course, the desired accuracy the term, and the previous term you know already what they are term is the current term in the Taylor series previous term is the previous term in the series `diff` is the difference between the term, and `prevterm`.

Actually there is `(())` and k is the current power of x. So, always term is x power k divided by k factorial. Now since we are also going to use the math function in the sine from the math library. We are also have to include we have another include statement the `hash include math dot h` this statement is that we are going to use a function from the math library namely sine function remember that the `hash include stdio dot h` is there in almost every program that we write, because in almost every program that we write we need to use the standard input, and output function.

So, after the loop run, and terminate finally, we can print the value of `sinx` to the computed value of \sin of whatever. So, we want to print the actual value of x here. So, just put percent f that will get the place value of actual value of f, and the actual value is another real number. So, therefore, another percent f, and the corresponding value is \sin of x, and here we are computing the same value using the math library functions `sin`. So, `libsindx` is \sin of x. So, this will directly give us the value of `sinx` from the math library, and we are also printing this library just to compare that the value that we have computed is correct.

(Refer Slide Time: 13:02)

A terminal window with a dark background and light-colored text. The text shows the compilation and execution of a C program. The user enters the command to compile 'sinx.c' with the math library, then runs the program. The program prompts for a value for x and a desired accuracy. The user enters 3.1419 and 0.00001. The program outputs the computed value of sin(3.141900) and compares it to the value from the math library, showing they are identical.

```
[dg@dgllaptop lecture-6]$ cc -o sinx sinx.c -lm
[dg@dgllaptop lecture-6]$ ./sinx
Enter a value for x: 3.1419
Enter desired accuracy: 0.00001
Computed value of sin(3.141900) = -0.000307
From math library, sin(3.141900) = -0.000307
[dg@dgllaptop lecture-6]$
```

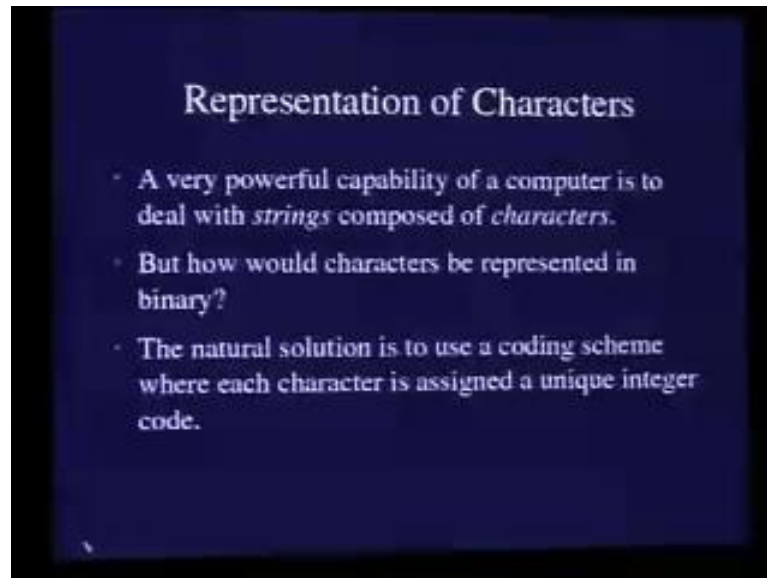
So, let us compile this program, and see whether it works correctly 1 new thing in this in this command line for compiling the program that you should notice is this minus lm that we have used over here this minus lm is required, because you have to tell the compiler that using a function from the math library, and therefore, the math library should be linked along with the program by the compiler. This is not necessary for this standard c library function such as printf, scanf, etcetera that we are been using, because the c compiler always link the standard c library along with the every program that it compiles, but that is not true for the math library or many other specialized library we might be used in our program.

So, in the case of math library for example, we have to in the program itself gives the include system just we saw in addition we have to use this minus lm plus, because the compiler that the maths library is also be linked with the program ok. The program has been compiled lets now try to run it. So, try to give some familiar value for x know that sin pi is zero. So, try to give the value of pi as the value of x next we want to have an accuracy up to 5 decimal points you can see the answer is almost 0 and in fact the math's library function has given precisely the same value.

Let us try some other value let us say pi by 2 sin pi by 2 is 1 the value of pi by 2 is approximately 1 point 5 seven, and we get the exactly 1 this time. So, this program is

working. Let now talk about another couple of type of value that are very useful in program in the first such values that we will talk about are character.

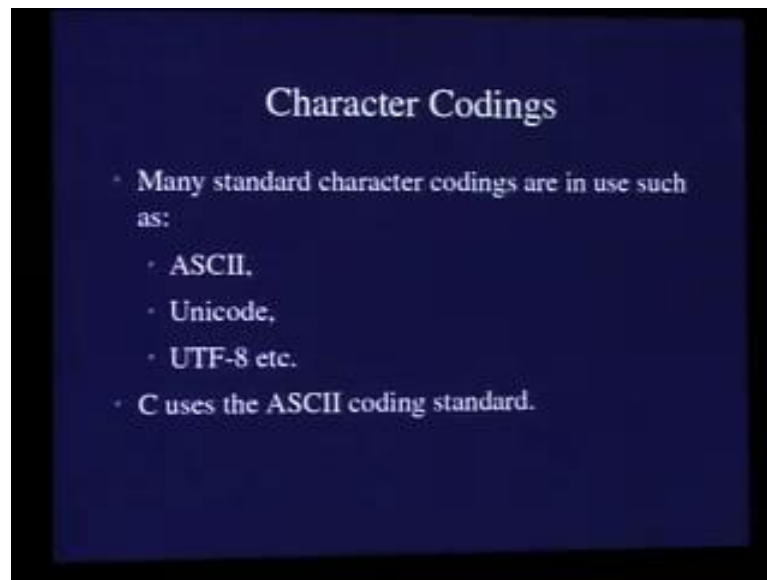
(Refer Slide Time: 13:41)



By characters is what meant by really always alphabet numerals, and other special symbols colon semicolon full stop less than greater than etcetera, that you find in the key board 1 very important capability of a computer is to be is that it can deal with character also apart from just number, and in particular it can deal with string which are which is a string is nothing but a sequence of a character will not talk too much about handling of characters in this lecture, and leave that too later lecture, but we want to talk about little bit about the representation of character inside the computer.

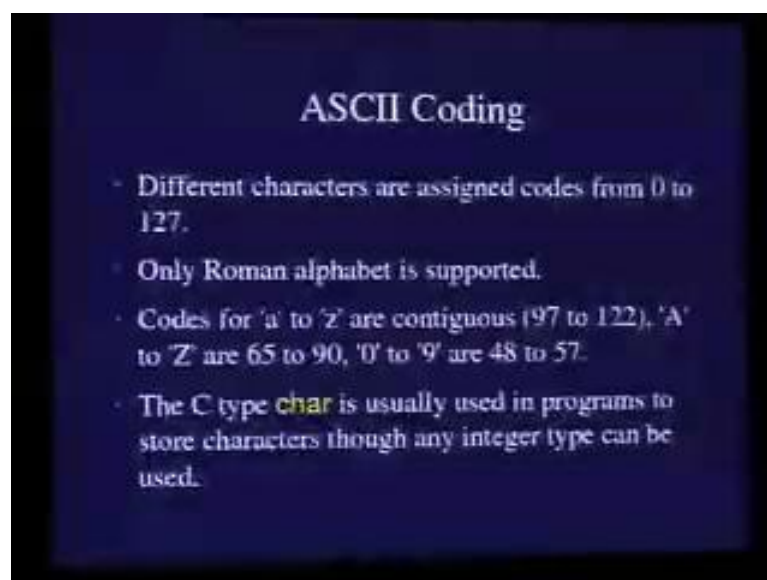
So, how as you know that every data every piece of data inside the computer is represented as the sequence of bit which are nothing but the zeros, and ones the question is how would characters represented in binary, and the natural answer is that you simply assign some code to different character. Let us say integer codes to various characters, and when you want to store a character in memory you just store the corresponding integer code.

(Refer Slide Time: 15:10)



There are several standard coding which have been defined in the various coding, and many such coding used, and some of the coding are called ASCII Unicode UTF-8 etcetera etcetera. The c programming languages occur using uses the ASCII standard which essentially is a standard code defining integers to corresponding to all characters that you can see on the key board. So, in the ASCII coding.

(Refer Slide Time: 15:19)



The different characters are assigned codes from 0 to 1 twenty seven so; that means, a total of 1 twenty eight characters can be represented only the roman alphabet is

supported in some of the other coding schemes. For example Unicode, and utf eight you can. In fact, support the characters from many different steps like, many other steps, so but we will be focusing on ascii for this course, because that is what C uses there are some some examples of ASCII code the codes for the characters a to z are contiguous 1 after the other the codes for small a character a small a is ninety seven that for small z is 1 twenty 2 b is ninety eight c is ninety nine and so on.

Capital A to capital Z are also 1 after the other capital A is sixty five capital B sixty six. So, on capital Z is ninety, and similarly the characters 0 to nine have the c code forty eight to fifty seven most carefully that we should always be careful in distinguishing the character zero, and the integer 0 the character 0 would be represented by the ascii code for 0 which is forty seven which is forty eight I am sorry, whereas the integer 0 would be stored internally just as the 0 in binary which is all bit back to zero. Therefore, the character zero, and the integer 0 are very very different the c type character are we have already visited in the last lecture is usually used in programs to store the characters might be reading from the input or you might be manipulating this characters and so on.

And the reason for using this type char is remember that the character is an integer type for which exactly 1 byte is used for storage which means that the numbers from minus 1 twenty eight to plus 1 twenty seven can be represented, and since the ascii code has source form from 0 to 1 twenty seven any character can be represented using the char type.

But in principle any integer type can be used to hold a character, because ultimately remember that a character is internally stored as an integer which is the ascii code for that particular character, and we can of course, also use the unsigned char, because the range for unsigned char is from 0 to 2 fifty five. Another important type of values that will often we need to deal with in our program are what are known as Boolean values, and the Boolean values is nothing, but just true or false.

We often need to remember just true or false are of some condition in the program we are going to shortly see an example of the program which uses this function these values which are just true or false they called Boolean values after the famous mathematician called bool; however, there is no Boolean type in C, and really to store Boolean values that is to store true or false what we can do is to store any integer type character integer

long short any of them, and in the integer representation from the Booleans in C the value 0 denotes false where as any non 0 value which may be negative or positive denotes true. So another example problem that we are going to solve today that is to find out whether the given integer n which is greater than 0 is prime or not.

So, the prime testing problem has been actually a very fascinating problem for mathematician for long time, and people have try to come up with the efficient algorithm for solving these problem, and only recently has last year this problem was solved in a very efficient fashion.

However those algorithms are fairly complicated will use very easy very easy algorithm which will be familiar to all of you to solving this problem, and that algorithm follows directly from the definition of the multi you know that a number is prime you can only use the it is visible by no other number other than one, and itself.

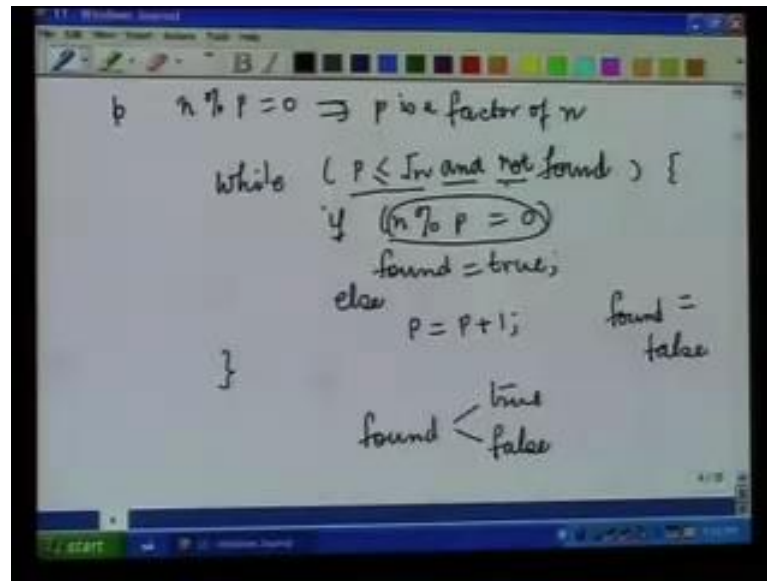
So, if there is some other factor of n other than one, and n itself, then n is not a prime. So, essentially the idea to find whether n is prime or not what we need to do is to test whether n has any factor other than one, and n are not.

So, we need to test four factors from 2 to n minus one; however, we do not need to test all the number to check whether they are any of them is factor of n or not, because you know that if there is a factor of n lets say p which is greater than square root of n , then there must be another factor q which is less than square root of n .

Why is that if you take simply q to be n by p since c is the factor of n therefore, q must be an integer that is p evenly divides n , and since c is greater than square root of n , and p times q is n therefore, q must be less than square root of n what is that tell us that tells us that we need to look for it divisor or the factor of n only from 2 up to square root of n .

If we cannot find the factor of n from 2 to square root of n ; that means, that cannot be any factor greater than square root of n either apart from n , of course itself . So, how do we use this facts to fashion the algorithm since the idea is very simple, let us use this variable called c which is the potential factor that we are going to get whether it is a factor of n or not do we test whether c is the factor of n or not.

(Refer Slide Time: 24:37)



We just have to check the remainder of the division of n by p , and test whether the remainder is 0 or not. So, if p is the factor the remainder would be zero, and in `c` there is an operator called the percent operator which percent `c` will give us the exactly the value of the remainder when n is divided by p . So, if n percent p is equal to 0 this implies that p is a factor of n . So, what we need to do is to have a loop run a loop from 2 to square root of n , and `c` will varies from 2 to square root of n in that loop, and if at any step p is the factor of n , then we should stop the loop.

So, the loop will look like this while some condition if the reminder of dividing the remainder obtained dividing n by p is zero, then we know that n is not a prime indeed p is the factor of prime which is other than the one, and n itself, then we need to do something, then we need to stop the loop. Otherwise you just have to increment p by p plus 1 as I put a question mark here, because we do not really know what we need to write here ok.

So, the question really is how long this loop should run now clearly this loop has to keep the has to has to `(())` as soon as value of p becomes greater than square root of n therefore, it must be running only as long as p is less than equal to square root of n , but if you find 1 factor of p 1 factor of n , then you do not need to test further integers of square root of n , because finding 1 factor of n is would enough to say that n is not a prime.

Therefore we must stop the loop as soon as we find the factor, if we find the factor if we do not find the factor, then of course, the loop will be terminate when ultimately p becomes greater than square root of n it will finally, becomes greater than square root of n , because in every iteration incrementing p by one.

So, now we need to in this case remember this loop or falsity of the condition that is n percent p is equal to 0 or not. So, we use a Boolean variable to store this value.

Let us call that Boolean variable as found. So, found will be either true or false we will start out as the false. So, true will be initialized to false, and am sorry false will be initialized to false, and this will become true as soon as we find 1 sector of n .

So, therefore, it should be set to true whenever you find that n percent p has become equal to zero. So, this question mark should now be replaced with found assign true, and this condition which we still have not completed. So, we must keep running the loop as long as p is less than equal to n less than equal to square root of n , and we have not found the factor right, because as soon as we find the factor we should terminate the loop. So, this is the basic idea of the algorithm note that we are use the Boolean variables found here which of course, in the c program as we have just seen will be represented using a variable of type int only, because there are no Boolean type in c, and we have also used 2 operations on Boolean values, and not.... So, the operation not essentially in word the truth value that is if the value is true not of that value is false, and vice versa.

So, the value of found is true, then not found is false, and similarly the value of found is false, then not found is true, and the operator AND is a binary operator that is it has 2 operands. So, in this example here are the 2 operands p less than equal to square root n , and not found. So, essentially what the AND operator does is that it gives the value is true is both the operands have the value true right.

So, if p is less than square root of n less than equal to square root of n , and not found is true meaning that found is false, then this entire expression will have the value true, and therefore that is the case, then the both will continue where as if either of these 2 conditions is false that is either c becomes greater than square root of n in which case we should terminate the loop, because we have not been able to find any factor, and we have to state all possible factors from 2 to square root of n or if you have found the factor, then also we should terminate the loop, because we know now that p is n is not a prime,

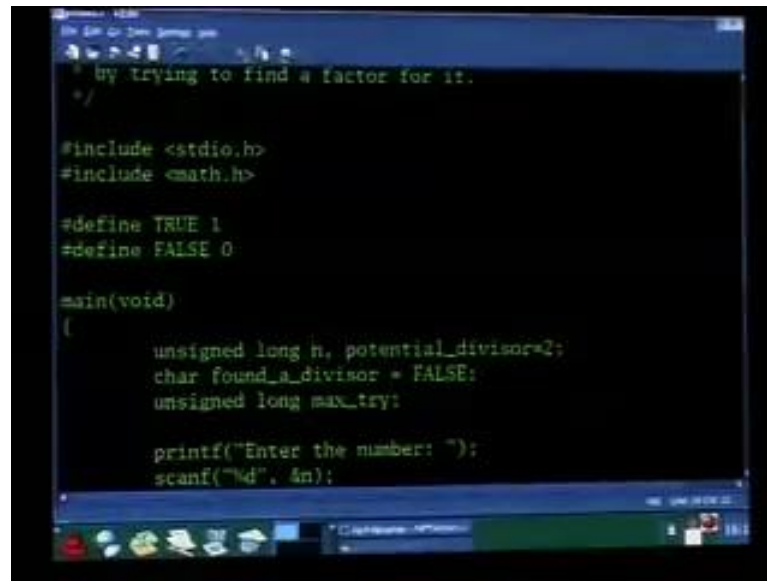
and `c` is the factor for `n`. So, if `sound` is true, then `not found` is false, and gives the value true only if both the operations are true it will give the value false, and therefore, the loop will terminate.

So, let us now convert these ideas into a c program. So, here is the program based on the algorithm that we just discussed. So, starts out with the comment that usual thing what the program does.

So, it finds whether the non negative integer is prime or not by trying to find a factor for it we have to include the `stdio dot h` file as usual `math dot h` they have included, because they are going to use the square root functions from the math library ok.

As we know now that c language does not define any Boolean type if we use the integer type of or Boolean type as well. So, we are going to represent we are going to use the value 1 to represent true, and 0 to represent false we call that every non 0 value is considered true, and 0 is false, but it is usually a convention to use 1 as the value for the true, and here is 1 more way of defining constant in the program. In the program that we have written in the last lecture for computing area of the circle we have defined constant by declaring them as variables, but prefix in the declaration with the word `const` this is 1 more way. So, if you say `#define true one`, and `#define false 0` what these essentially needs is that in the rest of the program wherever the word true or the word false appear that should be replaced by the value 0 or by the value 1 as the case might be.

(Refer Slide Time: 29:36)

A screenshot of a code editor window showing a C program. The code includes standard headers, defines TRUE and FALSE, and has a main function that declares variables for a number, a divisor, and a max try count. It prompts the user to enter a number and reads it using scanf.

```
/* by trying to find a factor for it.
*/

#include <stdio.h>
#include <math.h>

#define TRUE 1
#define FALSE 0

main(void)
(
    unsigned long n, potential_divisor=2;
    char found_a_divisor = FALSE;
    unsigned long max_try;

    printf("Enter the number: ");
    scanf("%ld", &n);
```

Where is the main program this time, we are using slightly different type we are using the type unsigned long for n. So, that n can be big as we can manage as big as we can handle, and n has to be always non negative. So, therefore, we can make it unsigned.

Similarly, the potential divisor is unsigned long, and the initial value for the potential divisor is 2. So, this potential divisor is the variable key that we have been using in the algorithm in the program we are using more robust name for variables longer these variable.

So, that we know precisely what a variable stands for, and what is the function that is performing the program, and the variable found that we are using in the algorithm is called found a divisor in this particular program, and it is precisely through a as you can see the names precisely specifies or indicate exactly what it meant by this variable what is the purpose of this variable.

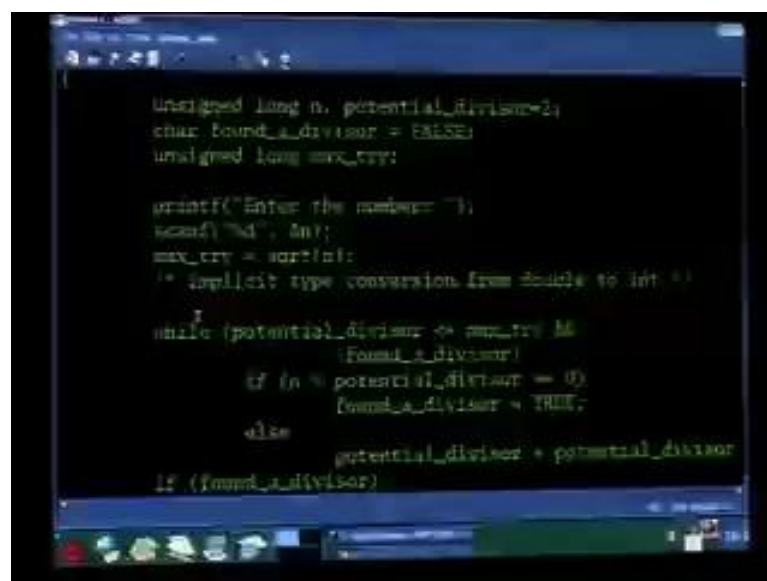
Now, these has to be conceptually it is the type Boolean, because it only hold the true or false value, and so but in c we have to use some integer type for this variable. So, that we used the type char we used the type we could have used any integer type, but we have used the type char, because the type char requires the least amount of storage among the integer type. So, using the char type char for representing Boolean variables is a good practice, because the program does not in that case you unnecessary storage, and of

course, the we just need to represent the value zero, and one. So, char type is good enough.

Now, max try is going to the value of square root of n are. In fact, the number the integer which is just below the square root of n in other words max size going to be the square root of n note that the square root of n need not to be an integer. For example if m is n, then the square root of m is three point something in that case max try should be three we no need to test for any divisor above three, because if there is a divisor above three, then there will be a divisor below three as well ok.

Starting of the program we read the number, and straight away we compute the square root of n by using the square root function in the math library this square root function actually gives the double value, and we are assigning with the double value to a variable of type unsigned long remember that max try is unsigned long. So, here the compiler automatically converts this double value to an unsigned long value this is called implicit type conversion which will not talk in too much detailed in this lecture. We will talk about it later, but intuitively what is happening here is that out of the double value when it is converted to an unsigned long integer this fractional part of the real number will be and only be integer part will retain.

(Refer Slide Time: 32:45)



```
Unsigned long n, potential_divisor=2;
char found_a_divisor = FALSE;
unsigned long max_try;

printf("Enter the number: ");
scanf("%ld", &n);
max_try = sqrt(n);
/* implicit type conversion from double to int */

while (potential_divisor <= max_try &&
       !found_a_divisor)
{
    if (n % potential_divisor == 0)
        found_a_divisor = TRUE;
    else
        potential_divisor = potential_divisor + 1;
}

if (found_a_divisor)
```

So, here is the main loop pretty much what we solve in the algorithm the condition for while loop is while potential divisor is less than max less than equal to max try and. So,

this here of this 2 ampersand denotes the, and logical or Boolean operation in c, and this blank character or the exclamation mark denotes the negations operation negation Boolean operation or the not Boolean operation.

So, the loop condition is while potential divisor is less than equal to max try, and we have not. So, far found the divisor note that the loop will terminate when either potential divisor becomes greater than max try or we found the divisor.

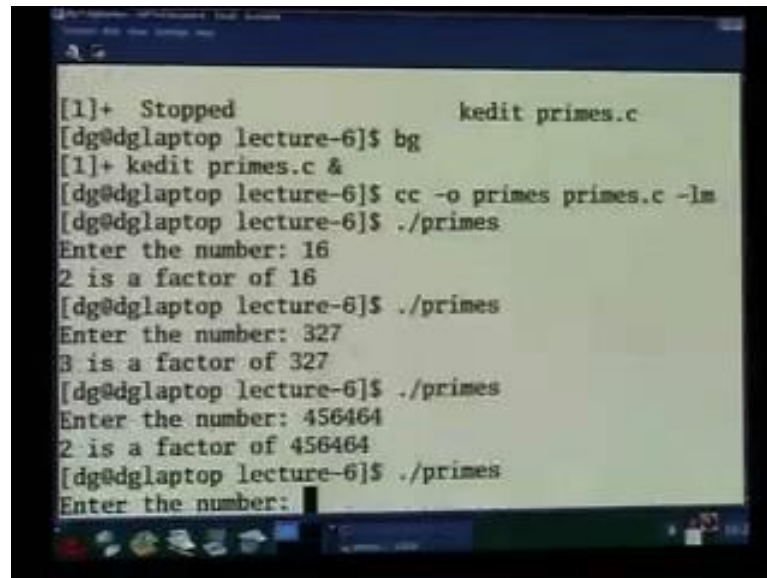
So, is n percent potential divisor is zero, then found a divisor is set to true otherwise potential divisor is simply incremented by 1 note that the body of the while loop is the single if else statement. So, this entire thing is a single if else statement, and therefore, we do not need braces around the body of while loop, because this is just 1 statement, and similarly the, then part of the if statement is a similar assignment statement we do not need braces here, and the else part is also a single assignment statement. So, we do not need braces around this either.

Of course, if we put braces there is no problem with that. So, this is the end of the loop at the end when the loop terminates if we have found the devices that is that n is not a prime so. In fact, we just saying n is not a prime we also said that n is not a prime, because here is the factor of n which is not 1 which is not n.

So, percent d is a factor of percent d first percent d is replaced by the potential divisor which of course, is the actual which is that actual divisor that we found or the actual factor that we found, and this percent d is replaced by the value of n if the variable found in the divisor is false, then we print that the even number n is a prime.

So, it is a pretty straight forward program lets now try to compile the, and see whether it was correctly again we have to use the minus lm to compile the program, because we are using the square root function from the math library, and if you remember we had also included math dot h in the program side itself ok.

(Refer Slide Time: 36:00)



```
[1]+ Stopped                  kedit primes.c
[dg@dgllaptop lecture-6]$ bg
[1]+ kedit primes.c &
[dg@dgllaptop lecture-6]$ cc -o primes primes.c -lm
[dg@dgllaptop lecture-6]$ ./primes
Enter the number: 16
2 is a factor of 16
[dg@dgllaptop lecture-6]$ ./primes
Enter the number: 327
3 is a factor of 327
[dg@dgllaptop lecture-6]$ ./primes
Enter the number: 456464
2 is a factor of 456464
[dg@dgllaptop lecture-6]$ ./primes
Enter the number:
```

So, let us now just execute this program lets try with some non prime number, let us say sixteen 2 is a factor of sixteen, let us try with the larger number three twenty four of course, 2 will again be the factor. Let us try three twenty seven this time three is the factor 2 is of course, the factor.

Let us try some prime number seventeen is a prime nineteen is a prime twenty three is a prime and so on this program is working correctly this brings us the end f the lecture. In the next lecture we will start working about operators in c of various signs. So, we will start with arithmetic operators, and talk about different other different kinds of operators like Boolean operators such that in more detail. We will see how complicated expression can be formed out of by using, and combining these different operators in various ways we are already been using the these operators in the programs that we have been writing, but we will discuss them more firmly in the next lecture, and also discuss some of the certain details associated with these operators.