

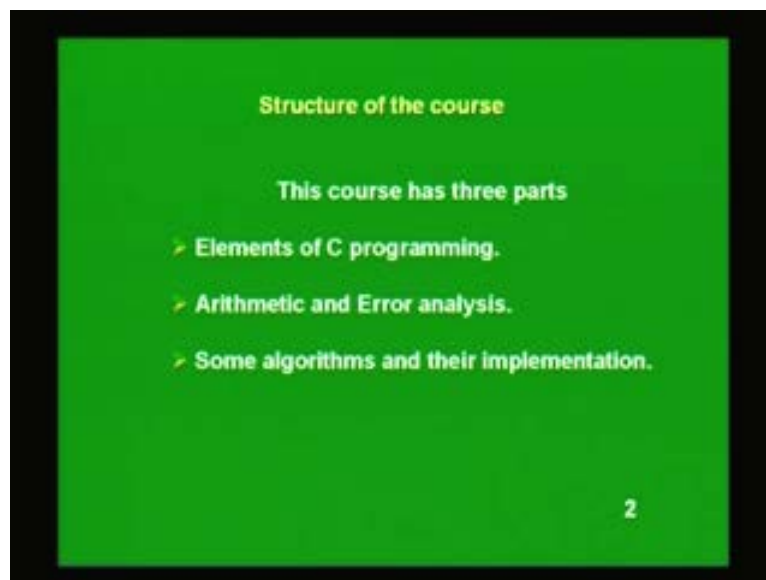
**Numerical Methods and Programming**  
**P. B. Sunil Kumar**  
**Department of Physics**  
**Indian Institute of Technology, Madras**  
**Lecture - 1**  
**Programming - Basics**

Hi. This course on numerical methods and programming, we will cover some of the basic aspects of both programming and about algorithms. We will learn, in this course, how to convert some of your ideas into calculations on your computer using some standard algorithms, and probably also learn how to design algorithms towards the end of the course. Before we actually go into the algorithms, we need to know how to implement that on a computer. That is, we need to know some elements of the programming.

So the first part of the course will cover that. So basic structure of the course is as follows. We would first look at, this course has 3 parts, structured into three main parts. Each has its subdivisions, so the first part is just elements of C programming. You will only see about C programs. There are many other languages like FORTRAN, etcetera, but we would concentrate only on C, C language. And the second part of the course is numerical arithmetic and error analysis.

That is what the second part of the course would be. And the third part of the course would be some of the algorithms and its implementation. So we have basically three different parts. The first part would be C programming. So, you would have learnt by the end of the first part, that, at the end of the first part, you would have learnt how to implement an algorithm.

(Refer Slide Time: 02:54)



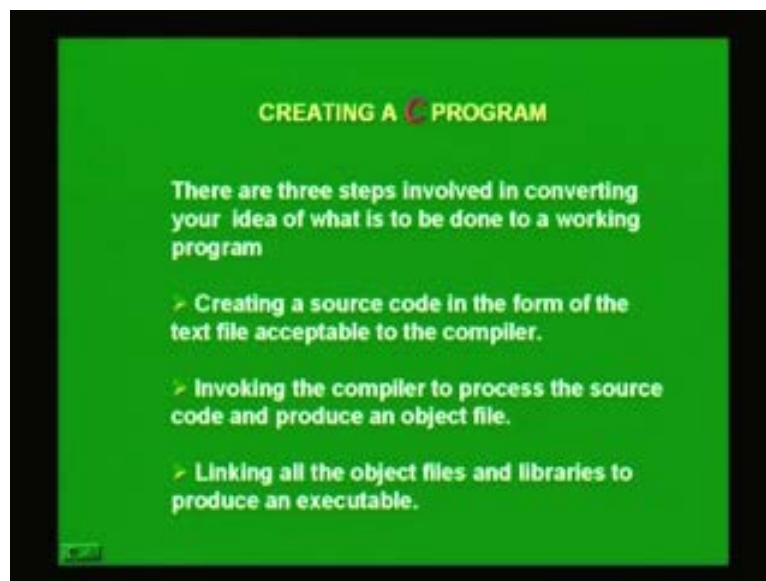
So we will first go through very basic features of C programming. We will not go through most of the complicated and deeper aspects of C programming, but some of the basic features of C programming. And then you would go through the concept of

numbers on a computer, and what are the errors they can come, how the error can propagate, and **what are the care**, what is the care one has to take when one is actually writing a code, etcetera. And the third part would be just the implementation of the algorithm and some of the algorithms.

So, let us start by looking at how do we create a C program. There are basically 3 steps involved in converting your idea into what is to be done to a working program. So, that is, first you have to create a source code okay in the form of some text file which is acceptable to a computer, using one of the editors. You would type in this source code, and then you have to invoke a compiler. In this case, you would use a C compiler. There are many C compilers.

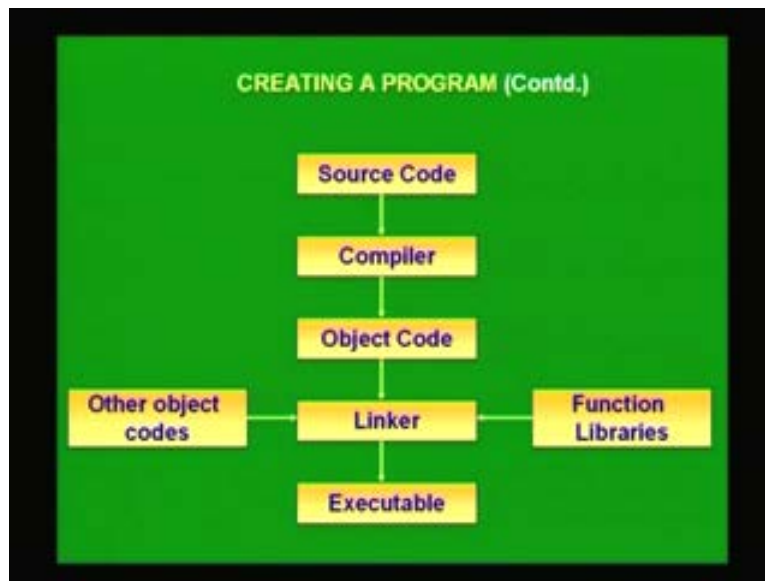
You would use the C compiler to process the C code and make it into an object file. You could have many different parts of the code into different files. We will go through all those details soon, and those will be all compiled separately or together to form object files, and then you would link all these object files together with other libraries and header files, etcetera, to form what is known as an executable, and it is an executable which we run on a computer. So that is the basic aspect, that is, basic 3 steps involved in running a code. So I summarize that in form of a flow chart, here.

(Refer Slide Time: 04:44)



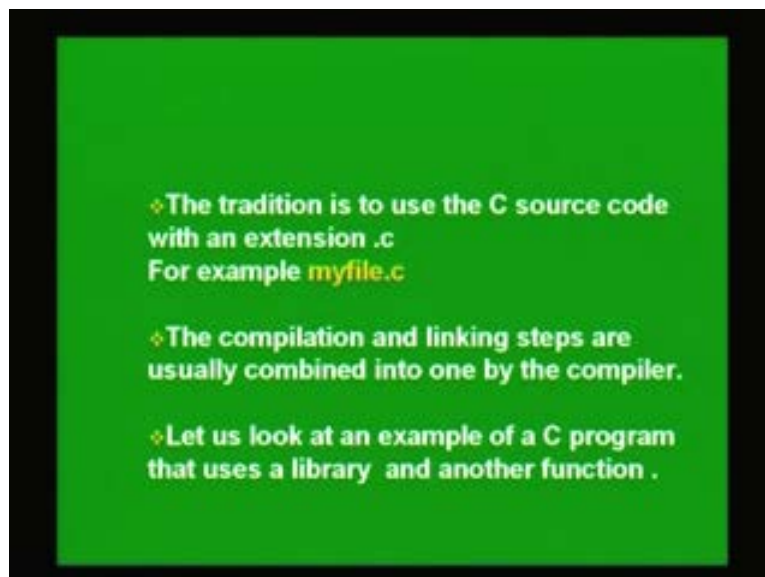
First we have a source code, and then we would invoke a compiler, and then produce an object file that creates an object code, and then we would use a linker to link this object code with other object codes and also with other libraries, function libraries and then we would get an executable. So I guess that is clear. So this is what we would do to run a program.

(Refer Slide Time: 05:32)



So the tradition is to use, to create, this source code with an extension, “.c”, when we are using a C program, it is not necessary, but that is a tradition. So we will always write a file name as with “.c”. For example, if you want to quickly create my file, “my file. c” is an example of a C program file, and the compilation and linking steps okay, can be, as I said, you first compile and you would produce an object code, and then you link it, okay, or you could have both of them combined into one step. We will see that in the later part of the course.

(Refer Slide Time: 06:21)



So let us look at a C program that uses a library and another function as an example. The best way to learn programming is through examples. So we would just look at an example of a C program. Here is a simple C code. So it has different parts okay. We can see there are some “include” files here, they are called the header files, and then you have the main part of the program here okay.

(Refer Slide Time: 06:31)

```
A Sample code "sample.c"

#include <math.h>
#include <stdio.h>
main()
{
    float x,y;
    while(x++<10.0)

    {
        y=sqrt(x);

        printf("%f\n",y);

        printit();
    }
}
```

And this program is basically computing the square root of a variable, x, and it is printing it out, and then it is calling a function. It has some basic features of a complete C program. It has a main part, and it does some computation here, and then it calls another function okay. So that is what this program is.

(Refer Slide Time: 07:06)

```
Create print.c

printit()
{
    printf ("The program is Over");
}
,
```

So, what is this function? This function is in this form. So here is a function which I call "print. c", and that function simply prints this program. So that is what this function is going to print. We will just see this program, and see how it can be compiled, and how it can be run, etcetera. This is "sample. c". So here is the program okay.

(Refer Slide Time: 7:18)

```
sample.c
#include<math.h>
main()
{
    float x,y;
    FILE *FP;
    FP=fopen ("sample.dat","w");
    while(x++<3.0)
    {
        y=tanh(x);
        fprintf(FP,"%f\n",y);
    }
    printinit();
    fclose(FP);
}
```

(Refer Slide Time: 08:00)

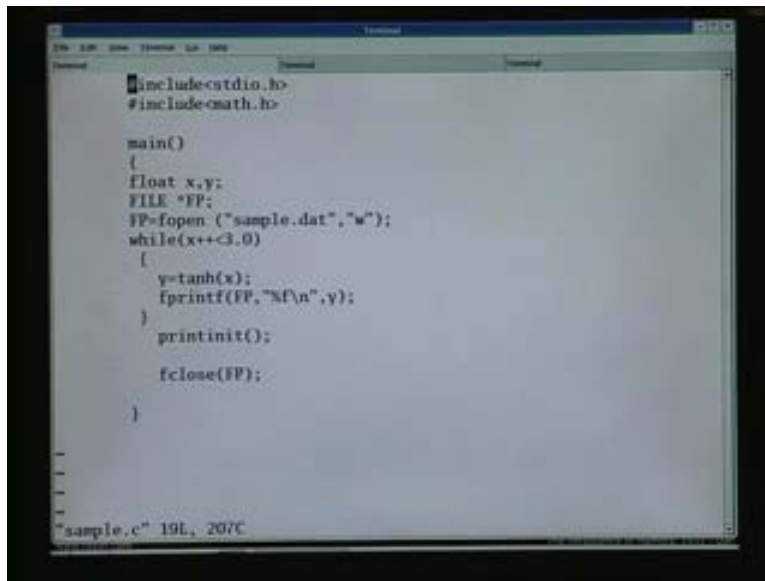
To compile these programs we could type  
`cc -c sample.c` and `cc -c print.c`  
separately.

This commands will create `sample.o` and  
`print.o` files. We can create the executable by  
typing the command.

```
cc sample.o print.o -lm -o a.out
```

So it has that “include” files, and the “stdio. h” and “math. h”. Here I am not computing the square root of x, I am computing tan hyperbolic of x, and it writes, instead of writing it into the screen, it writes it into a file called “sample. dat”. This is little more complicated than the simple code I showed. It opens a file called “sample.dat”, and then it prints it into that file, and then closes the file. That is what it does, and again, it calls this function. So now the question is, how do we compile this? Okay, there are two ways of compiling this. As I said here that you can first compile the programs separately to produce object files that is what we will do. So we will say “cc minus c sample. c”. So, that would create, so it would create a file called “sample. o” and we would say “cc minus c print. c”, so that would create a file called “print. o”.

(Refer slide time: 08:10)

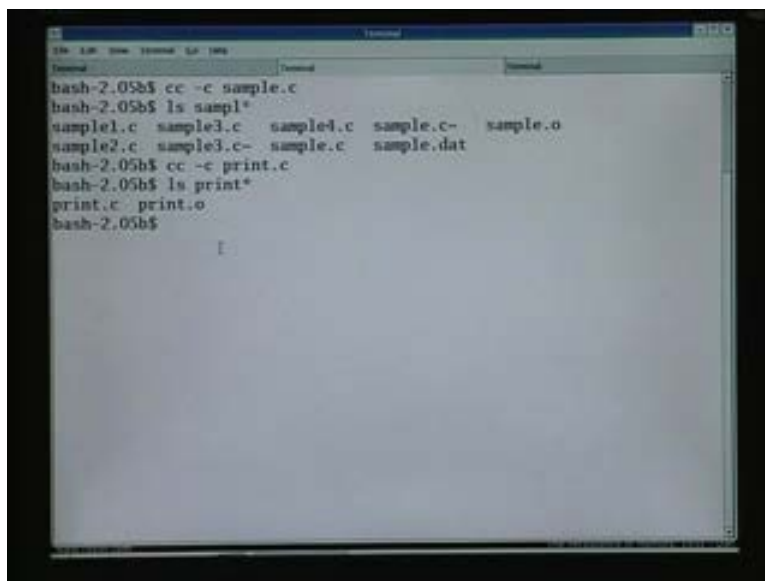


```
#include <stdio.h>
#include <math.h>

main()
{
    float x,y;
    FILE *FP;
    FP=fopen ("sample.dat","w");
    while(x<=3.0)
    {
        y=tanh(x);
        fprintf(FP,"%f\n",y);
    }
    printf(" ");
    fclose(FP);
}
```

So, we compiled this “print. c” and we got a file called “print. o”. And then we would now put them together and produce an executable, and in the process, we also link it to a library called minus lm. So we link it to a math library by using this command, and we say minus o which means the output file and that goes into this “a. out”. You can put any name here, like I could put “my file” here and you will get an executable “my file”. So we will do that. So we can say cc sample. C, sample. o sorry, print. O, minus lm, minus o, a. out. So that would create this file called “a. out”. “a. out” is the file which has been created by this compilation okay. So you could also do this in one step. I will show you.

(Refer Slide Time: 09:28)



```
bash-2.05b$ cc -c sample.c
bash-2.05b$ ls sampl*
sample1.c sample3.c sample4.c sample.c- sample.o
sample2.c sample3.c- sample.c sample.dat
bash-2.05b$ cc -c print.c
bash-2.05b$ ls print*
print.c print.o
bash-2.05b$
```

I will remove this “a. out”, and I could say cc sample. C, print. C, minus lm, minus o, a. out, and then you can see that it has created “a. out”. Okay so to run the program, I would just say ./a. out. So it is printed on the screen, the print program is over, that is

because I had given in the print. c. So remember, the print. c is just printing “the program is over”. So what does this program do? It has actually computed tan hyperbolic x for all values of x, and it wrote it into a file called “sample.dat”, and then it called this program function, this function “print in it”, print it, and the “print it” is just basically printing it out “this program is over”. That is all it is, yeah? So that’s what we did, and we saw the two ways we can compile it. We can actually compile it by invoking minus c command which will only produce, only do the compilation, and if you do not use this minus c and then it would compile and link. So, both will be done together here.

(Refer Slide Time: 10:50)

```

bash-2.05b$ cc -c sample.c
bash-2.05b$ ls sampl*
sample1.c sample3.c sample4.c sample.c- sample.o
sample2.c sample3.c- sample.c sample.dat
bash-2.05b$ cc -c print.c
bash-2.05b$ ls print*
print.c print.o
bash-2.05b$ cc sample.o print.o -lm -o a.out
bash-2.05b$ ls a.out
a.out
bash-2.05b$ rm a.out
bash-2.05b$ cc sample.c print.c -lm -o a.out
bash-2.05b$ cc sample.c print.c -lm -o a.out
bash-2.05b$ ls a.out
a.out
bash-2.05b$ ./a.out
The program is over
bash-2.05b$

```

(Refer Slide Time: 12:03)

```

bash-2.05b$ cc -c sample.c
bash-2.05b$ ls sampl*
sample1.c sample3.c sample4.c sample.c- sample.o
sample2.c sample3.c- sample.c sample.dat
bash-2.05b$ cc -c print.c
bash-2.05b$ ls print*
print.c print.o
bash-2.05b$ cc sample.o print.o -lm -o a.out
bash-2.05b$ ls a.out
a.out
bash-2.05b$ rm a.out
bash-2.05b$ cc sample.c print.c -lm -o a.out
bash-2.05b$ cc sample.c print.c -lm -o a.out
bash-2.05b$ ls a.out
a.out
bash-2.05b$ ./a.out
The program is over
bash-2.05b$ cc sample.c print.c -o a.out
/tmp/cc8wcd11.o(.text+0x54): In function 'main':
: undefined reference to 'tanh'
collect2: ld returned 1 exit status
bash-2.05b$

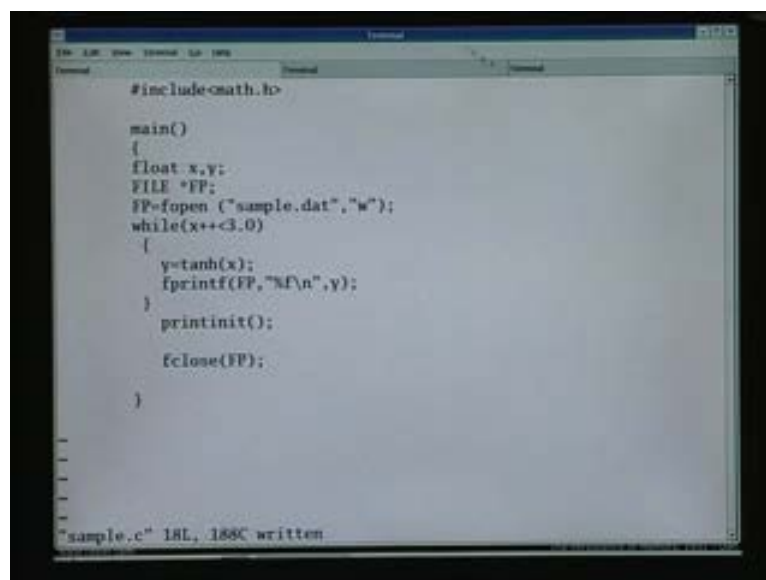
```

For example, here we are not linking it that is why we do not have to use minus lm command. Here we are linking. so we are using minus lm command because we are linking to a math library. So let us see what happens if we do not link it. So let us say

I do not like linking this program with another library, so I just do this okay. So then it says it cannot understand tan hyperbolic okay.

So the tan hyperbolic function is built into a library, math library, so we need this math library to compile this program. So for many of those mathematical functions we need this math library. So for some other functions you might need some other libraries, so you would use those libraries or give the minus l command. So these libraries are all stored in a different path of the computer, so it will be stored in a region called, for example, in a Linux platform it will be stored in /users/lib. So it could be in a different program. In a different operating system it will be in different place. So all these libraries are stored there and we are just linking them to that library, linking the program to the library. So we need to use minus lm command if you are using math functions. So similarly, we saw in this program that we are using these header files.

(Refer Slide Time: 13:09)



```
#include <math.h>

main()
{
    float x,y;
    FILE *FP;
    FP=fopen("sample.dat","w");
    while(x<+3.0)
    {
        y=tanh(x);
        fprintf(FP,"%f\n",y);
    }
    printinit();

    fclose(FP);
}
```

"sample.c" 18L, 188C written

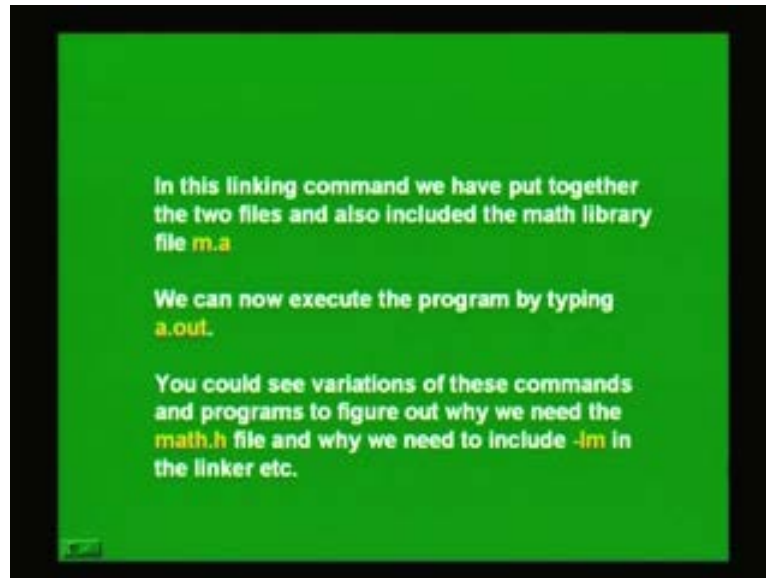
So what is the use of these header files? We can see that by just removing this header file, let us say we remove the stdio. h and then try to compile now with minus lm. Of course, so then it says it cannot understand these things minus file, fp, etcetera. So all that definitions, that is, we say this “file” statement, and this “open” statement, all the statements are actually put into the stdio. h file. So that is required if you want to use these statements, of course, for the simple “printf” statement you do not need it.

In some of the compilers, all these statements are actually built into the compiler, so we do not need a separate header file okay. So but it is a good practice to put this in here because some compilers need it and some compilers do not need it. So if you want your program to be kind of compiler independent as much as possible then you put this into the program. So that is the basic idea of how you create a program and how do you compile it and how do you run it okay. So we saw that we have this very simple code, and with another function we compiled that and we could run it okay. So that is what it is okay.



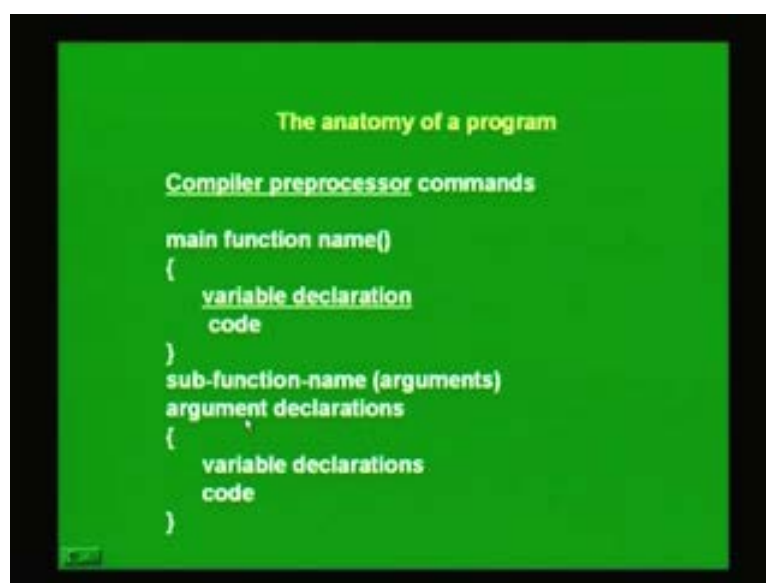
So as I said, in this linking command, to summarize what I said, in this linking command we use the library okay, an archive, a math archive called `m.a` that is what we did. So, we also saw what is the need of this header files, and what is the need of this `lm` command in this thing.

(Refer Slide Time: 14:49)



Okay so now, let us look at the anatomy of a program. To summarize this, we had some compiler preprocessor commands, and that is what I said, all these “include” files here, and then you had a main function, some name to a function, you can give some name to this main function, and then you had some variable declaration right, and then you have the code, the main code, and then you could have sub-functions. In the previous example, I had put the sub-function separately.

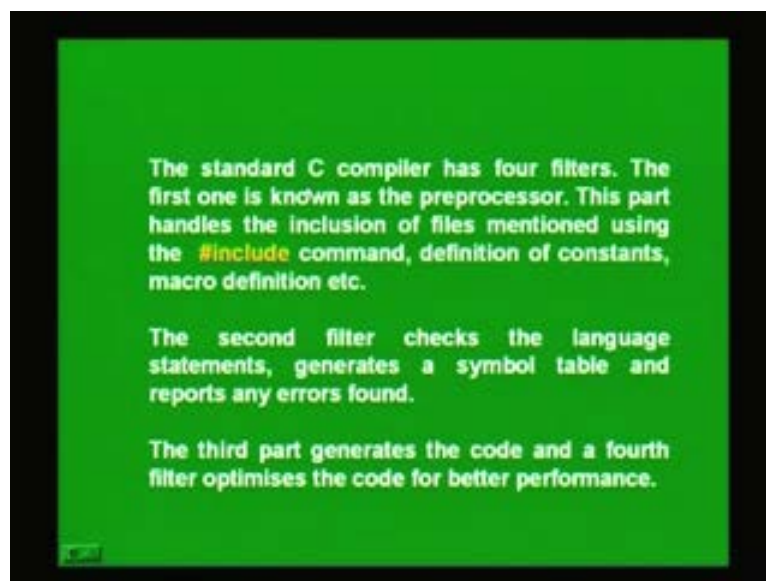
(Refer Slide Time: 15:43)



You can also include it into the main code okay. In one file, in the main file, in one file, you could have the main program or the main function I call okay, and the sub-functions, or you could have them separately. Again, it is a good idea to have the sub-function separately, and you compile them separately, and then you link them together. That is always a better idea. I will tell you why later. It is always safer to do that, to have the sub-function separately, and then do that for such small programs as the one I am going to show here. It may not be important, but for big programs it is always good to have sub-functions separately and compile separately and in the sub-function, it has the same structure. It might have some arguments which you pass into it, and then you have the argument declarations here, and then you have the variable declarations which are private to the sub-function, and then you have the main code of the sub-function. We will see examples of this, how we do this okay.

So, before that, let us go through each one of these in little more detail. For example, let's see what a compiler preprocessor does. As I said, the compiler preprocessor basically handles the first part of the program okay, so that is, `stdio. h`, `math. h`, and all these header files, are actually handled by the compiler preprocessor, everything with an "include" command. And then the second filter of the compiler. As I said here, the compiler has about 3 or 4 filters. The first filter is the preprocessor which handled that, and the second filter actually checks the code. It checks all the codes, symbols, statements. All things are checked by the second filter and the third generates the code, a code which is understandable to the machine.

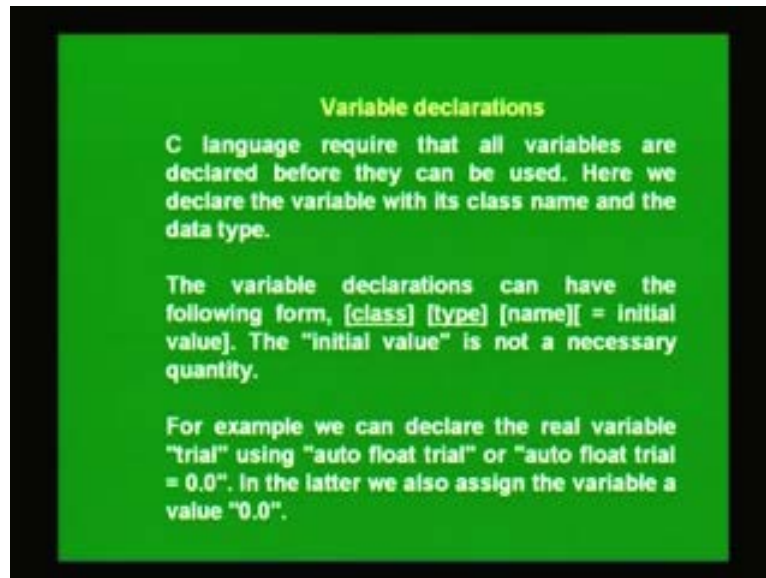
(Refer Slide Time: 16:46)



And there can be a fourth filter which actually does the optimization which improves the code. It actually modifies the code okay and it improves the code such that it will run faster. That is about these 4 filters. The fourth one need not be there, it depends on whether you invoke it or not. So, that is the 4 parts into the code. So, let us look at that is the basic compiler preprocessor you looked at. Now let us look at the variable declarations. What is meant by the variable declaration? We have seen in this code. We have said that `float x y`. These are the only variable declarations. Are these are the only type of variable declaration if you can make? No, there are many types of

variable declarations. In general, it has this form, that is, it has a class, and it is a type, and then it needs a name. It could also have an initial value given to it. So the C language, unlike other languages, requires all variables to be declared before you start the program okay. All variables have to be declared, which is good or bad, depending upon what your use is. So everything has to be declared okay.

(Refer Slide Time: 19:03)



**Variable declarations**

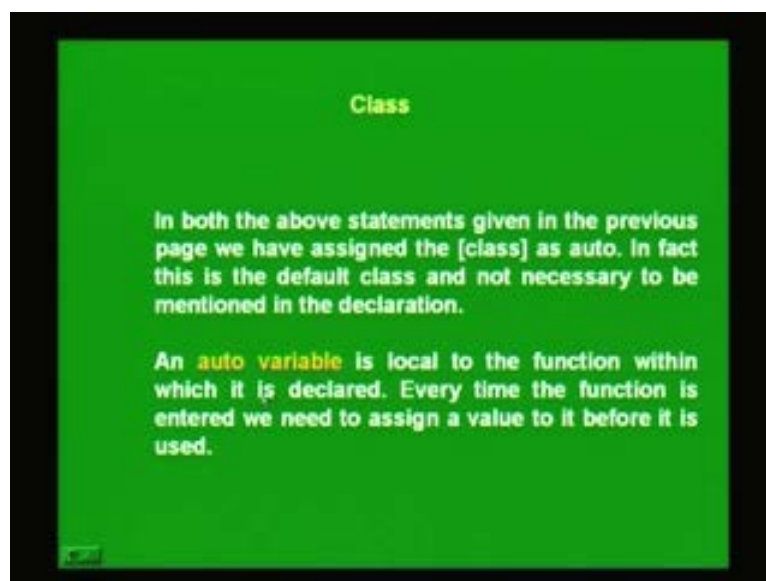
C language require that all variables are declared before they can be used. Here we declare the variable with its class name and the data type.

The variable declarations can have the following form, [class] [type] [name][ = initial value]. The "initial value" is not a necessary quantity.

For example we can declare the real variable "trial" using "auto float trial" or "auto float trial = 0.0". In the latter we also assign the variable a value "0.0".

So and the declaration is of this form, that is, class, type, name, and an initial value. The initial value is not compulsory. For example, here I give an example. "Auto" is a class, "float" is a type okay, and then you have "trial" as a name okay. So you could declare it as "auto float trial" is equal to 0.0, or simply "auto, float, trial". Both are possible. Now what happens is, "auto", we will see that in the next slide. It has more details about the class.

(Refer Slide Time: 19:38)

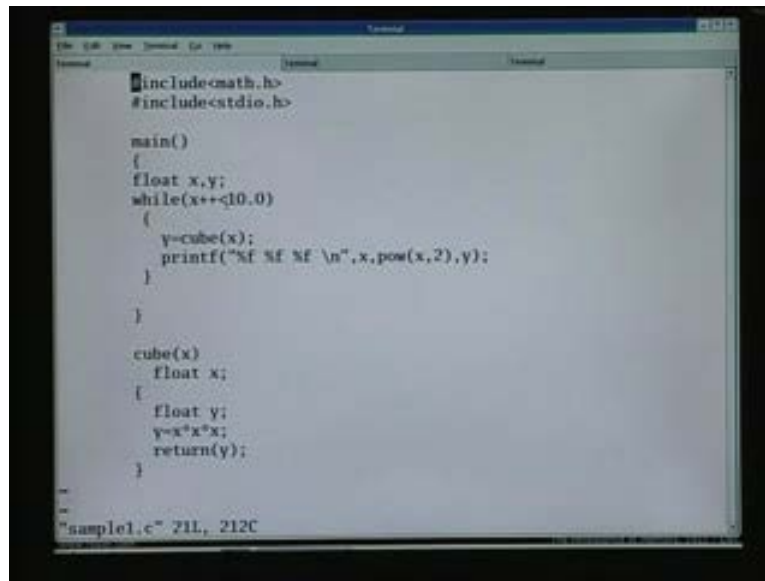


**Class**

In both the above statements given in the previous page we have assigned the [class] as auto. In fact this is the default class and not necessary to be mentioned in the declaration.

An **auto variable** is local to the function within which it is declared. Every time the function is entered we need to assign a value to it before it is used.

(Refer Slide Time: 20:00)



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

main()
{
    float x,y;
    while(x++<10.0)
    {
        y=cube(x);
        printf("%f %f %f \n",x,pow(x,2),y);
    }
}

cube(x)
float x;
{
    float y;
    y=x*x*x;
    return(y);
}
```

So “auto” variable is actually a default. If you say “float”, it is taken as “auto” automatic okay. So, that is a default variable declaration. Let us look at some of these examples here. Okay so here I have declared a similar program as the one before, so here are the header files, etcetera. And then there is a main function here, and we also have the sub-function here, and I have declared the variables “float x y”. No initial value is given okay. This actually means “auto float x”. If you have to declare more than one variable in one line you may just separate them by a comma.

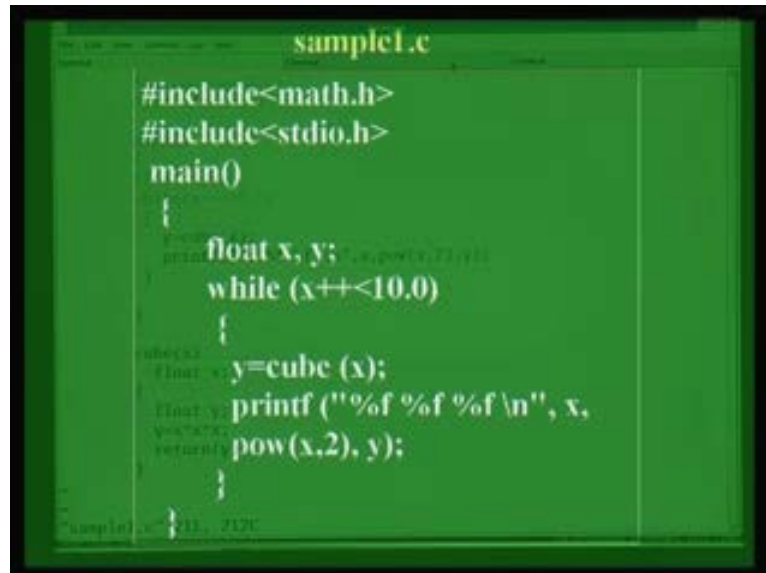
So you can have any number of variables put in here. I have done two here, and in the C program all lines are separated by a semi colon. So, here is a declaration of the “auto” variable. This variable declared as “auto” variable is private to this function. Unless you make an effort to pass it to something else it will not go out of this function. You could have the same variable declared in another function. Here is a sub-function I have declared okay.

For example, “float y”. The y value it takes inside this is different from the y value it takes from this unless you have passed it back, and in this function we are actually doing that okay. So here is an example of the use of an auto variable in which you declare the function. Here the x value takes, starting from 0, in steps of 1, it increases the value. It is a floating point variable. Now, what is a floating point, and what is an integer, etcetera? We will see in the later class when we talk about arithmetic; how is a floating point actually stored in the computer; how is an integer actually stored in the computer. We will see that later okay.

Right now, we say floating point is a real number, and it is an integer. That is what we have – two different types – and we will go through that later, a little later again. So here is a floating point, and that takes the value 0, 1.0, 2.0 etcetera here, and this function actually computes the cube of that, so it finds the cube of the variable, and returns it to this function, and this spreads it. It also uses a built in math function called power. It finds the second power of the variable x okay, and I have here made a program to compute the cube. So if you actually execute the program you will get the

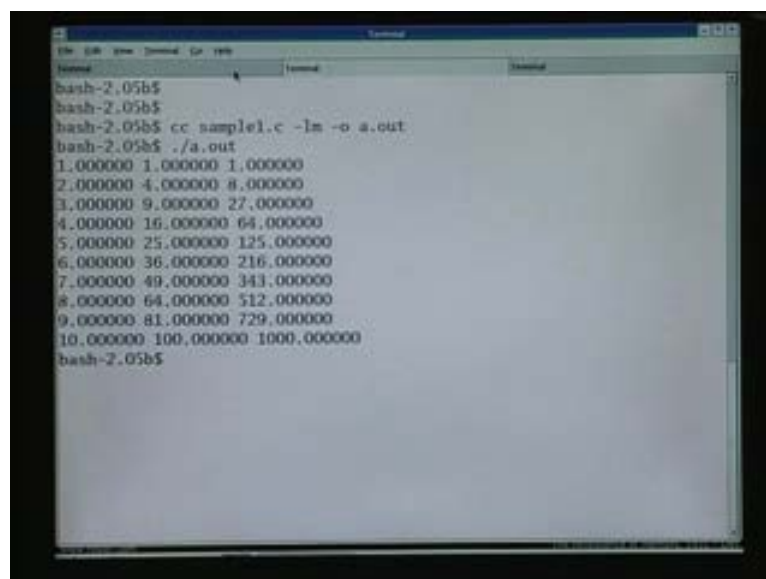
square and the cube, both, So we will see that here. Okay so we say cc sample1.c minus lm again we need to do and minus o, a. out, or I could call it something else, a. out I call, So if I do that, it prints out the number x value and its square and its cube.

(Refer Slide Time: 22:29)



```
sample1.c
#include<math.h>
#include<stdio.h>
main()
{
    float x, y;
    while (x++<10.0)
    {
        y=cube (x);
        printf ("%f %f %f \n", x,
        pow(x,2), y);
    }
}
```

(Refer slide time 23:06)

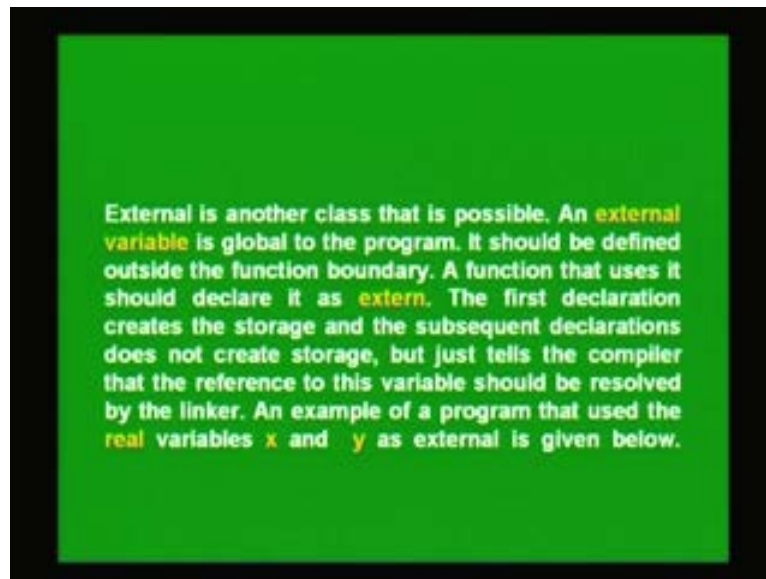


```
bash-2.05b$
bash-2.05b$
bash-2.05b$ cc sample1.c -lm -o a.out
bash-2.05b$ ./a.out
1.000000 1.000000 1.000000
2.000000 4.000000 8.000000
3.000000 9.000000 27.000000
4.000000 16.000000 64.000000
5.000000 25.000000 125.000000
6.000000 36.000000 216.000000
7.000000 49.000000 343.000000
8.000000 64.000000 512.000000
9.000000 81.000000 729.000000
10.000000 100.000000 1000.000000
bash-2.05b$
```

Now, when I pass this variable, this function actually calls it, when I call this function I pass the variable x. So I said that in the beginning, that when you have sub-function, you have to declare variable declaration of the variable which is being passed on to that function, that has to be done here. Suppose we do not do that. Let us say we do not have this, then what happens? So if you do not do that do we get the same answer? So we compile that again, and we run that, and we get 0 okay. So basically, it does not know what the value of x is because we have not passed it. So it has to be passed. When you have a sub-function this auto variable has to be declared inside that

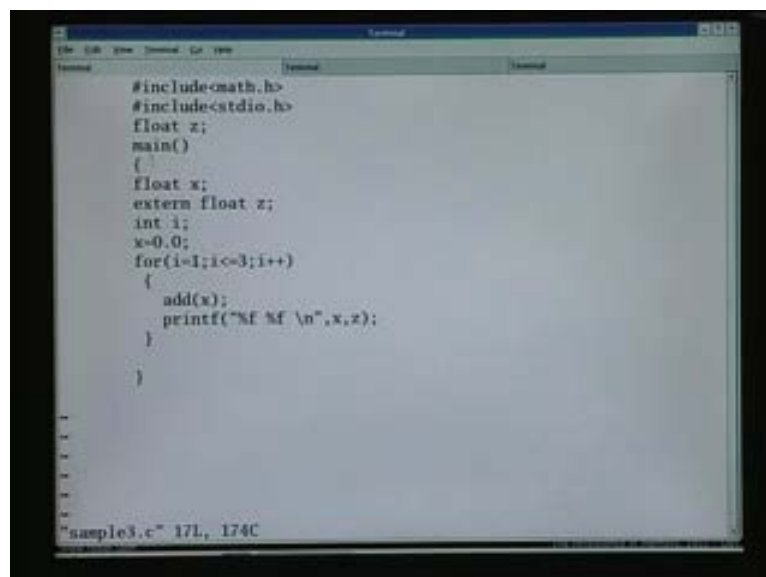
sub-function separately. That is, the auto variable's use. Then you can have external variables. External variables are common to the whole program okay.

(Refer Slide Time: 24:30)



So, you have many sub-functions and functions, and you want a variable to be shared by all these programs, all these functions, then you could declare that as external, then it will be shared, then you have to tell the sub-function that this variable is declared external, otherwise it will not know okay. So that, we will see that with another example. That is the best way to look at this.

(Refer Slide Time: 25:15)

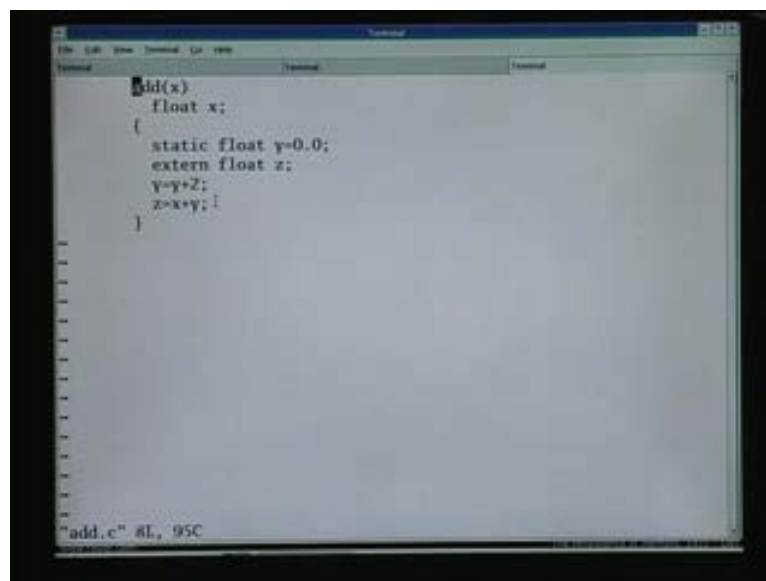


So here is a function which uses the external variable, and this is the main program. Now, I have declared "float z". This is again a floating point, the z is declared as a floating point, and I have declared it outside the main function. That is where you display the external variables, you declare them outside. You do not have to say

external; you just declare the variable which you want to use outside that function, outside the main program. Now this program actually calls, so I have declared that inside here as external float set because its main function is going to use this thing, so I have declared that variable as external in the main function.

Now there is another sub-function here which this program calls, and that is called add (x) some other function called add. So that function which I would show you here has also been using the variable x, variable z, so I would say external float z inside that function. I have to say this. If I do not say this this function will not know what z is okay. It is an external variable, and it shares that value with it. So if the z value is changed here, in this function, it will also get changed in this sub-function.

(Refer Slide Time: 26:42)



```
add(x)
float x;
{
  static float y=0.0;
  extern float z;
  y=y+z;
  z=x*y;
}
```

add.c" 8L, 95C

Now if you want to compile this function: cc sample3. C. I am not creating the object file, I will just do everything together. So it creates, it runs this program. So let us see what happens if I remove this here. So I do not have the declaration here, and then I of course it does not understand what that is. We could change the value of the function here. Okay so we could have, What we could do is, we could give it a value here, say, z equal to 6.0 okay, and then we save it, and we declare that here. Then we will see that what value it gets now would be different, so because we have changed it in the main program itself and it has been shared with this sub-program here okay. C program is case-sensitive, so you have to be careful with that. I am printing out something different. I should print out inside this program. I will make a "printf" statement here.

It is just to see that when we change it in the main program it also gets changed here. So it prints out "6" here. Because in the main program we change it, it changes here also. Okay that is what we wanted to see. So that is the external variable. Then you can have what is called, this is again the summary of how we use external variable. So we declared it outside the main function, and any function which it uses it, will declare it external. So that is the external variable. And then you could have the static variables. Okay now static variables are also very special definitions. If you declare a

variable as static inside a subprogram okay, now that variable is changed exactly once okay, that is the first time it enters the subprogram okay. That variable is changed. After that it remains static. We will see that again with another example.

(Refer Slide Time: 29:03)

```
sample2.c
#include<math.h>
#include<stdio.h>
main()
{
    float x,y;
    int i;
    x=0.0;
    for(i=1;i<=3;i++)
    {
        y=add(x);
        printf("%f %f\n",x,y);
    }
}
```

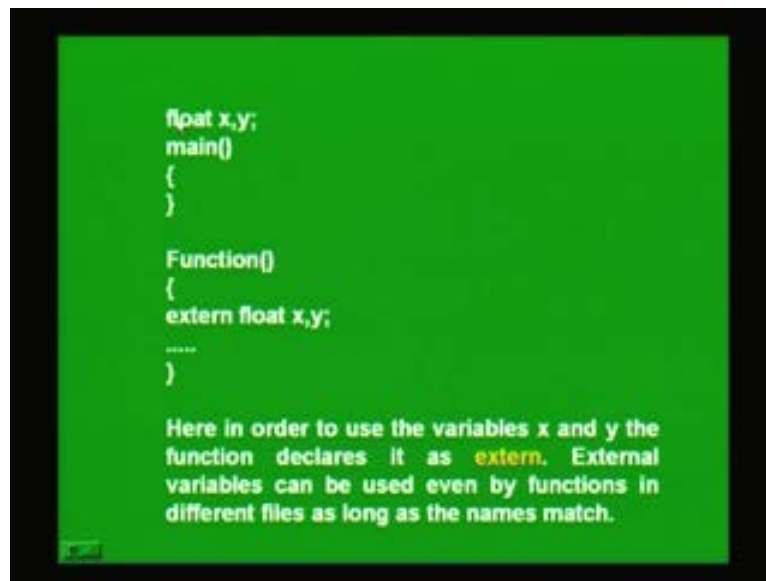
(Refer Slide Time: 29:17)

```
add(x)
float x;
{
    static float y=0.0;
    float z;
    y=y+2;
    z=x+y;
    return(z);
}
```

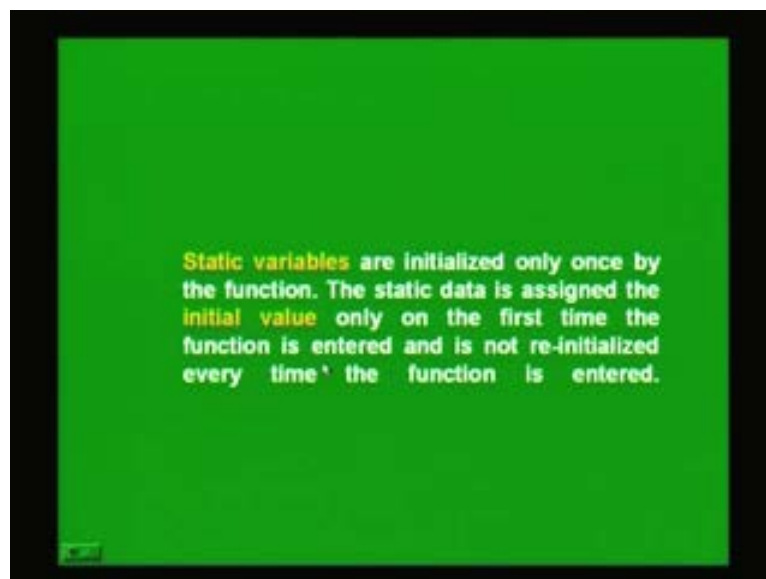
So here is the function which uses the variable static, static variable here. I declare the variable here as y equal to 0 as static. So that means it would initialize this variable the first time it enters this code okay, and after that it will not. What happens if you run this? So we will run this. Cc sample 2.c; it prints 2, 4 and 6. That is what it is doing. First time it enters it got 0, then it's adding 2 to it, and then printing 2, 4 and 6. Let us see what happens if you don't declare it as static okay. Since it is initialized every time it will just keep printing 2. That is what it will do. That is what we expect it to do.



(Refer Slide Time: 29:43)

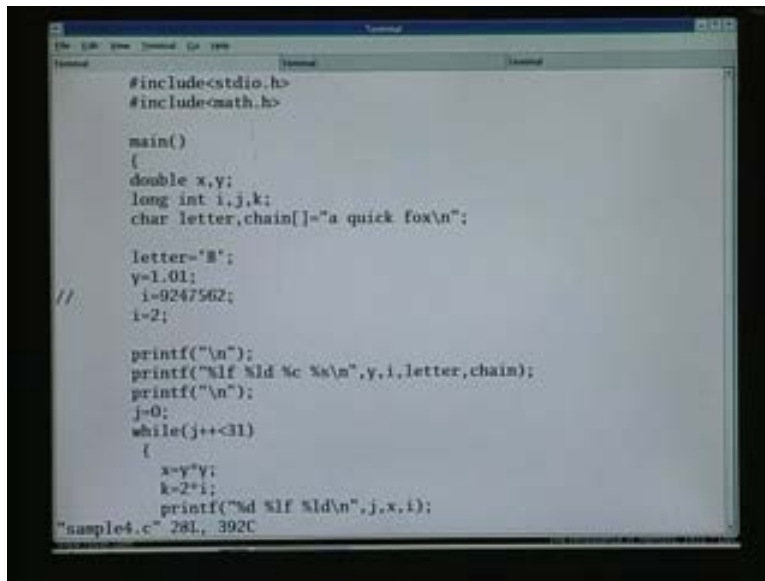


(Refer Slide Time: 29:54)



So that is what it does. So that is a static variables do. It is very useful to have this definition. There are many cases. For example, in the case of random number generator you have some kind of a seed and use that seed to generate a random number, and the next time you call a random number it does not want to again initialize the whole process, it wants to generate a new random number not the same random number. So it wants to initialize it only once. For the first time you call this thing. For that kind of function it is extremely useful to have something defined as static okay. So we need this definition of static.

(Refer Slide Time: 30:24)



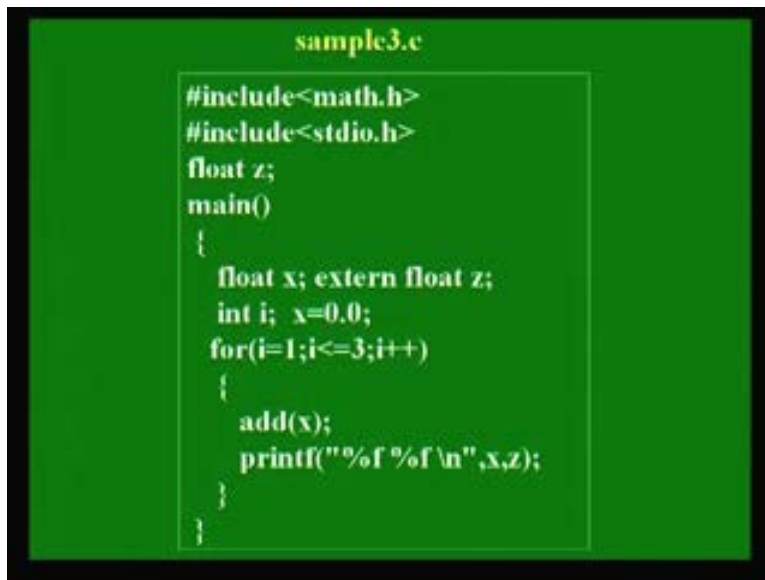
```
#include<stdio.h>
#include<math.h>

main()
{
double x,y;
long int i,j,k;
char letter,chain[]="a quick fox\n";

letter='B';
y=1.01;
//
i=0247562;
i=2;

printf("\n");
printf("%lf %ld %c %s\n",y,i,letter,chain);
printf("\n");
j=0;
while(j++<31)
{
x=y*y;
k=2*i;
printf("%d %lf %ld\n",j,x,i);
}
}
```

(Refer Slide Time: 32:16)



```
sample3.c
#include<math.h>
#include<stdio.h>
float z;
main()
{
float x; extern float z;
int i; x=0.0;
for(i=1;i<=3;i++)
{
add(x);
printf("%f %f\n",x,z);
}
}
```

So we just now saw that we have auto variables, and we have external variables, and we have static variables. I think that is the basic three classes which you need to look at. So going back to our definition, so we had the class, then we have type, and your name. Name, we already saw, x, y, anything as a name okay. We can put in many names, but we have to be careful as it is case sensitive okay. In the C program, the variable names are case sensitive, so use of a upper case and lower case letters are different, variable.

And initial value we saw that we can put the initial value, we may not put the initial value. So that also is up to the programmer. So the second thing is the type okay so we also saw, I already showed you some type here, for example, "float", okay that is one type. What are the main types of variables which we can declare? So here is the summary or the chart of various data types. You could have an integer, or we could

have a floating point right. So that is what we saw, a real number or an integer okay or we could have characters. The characters are declared as “char”; integers are declared as “int” and floating point as “float”.

(Refer Slide Time: 32:31)

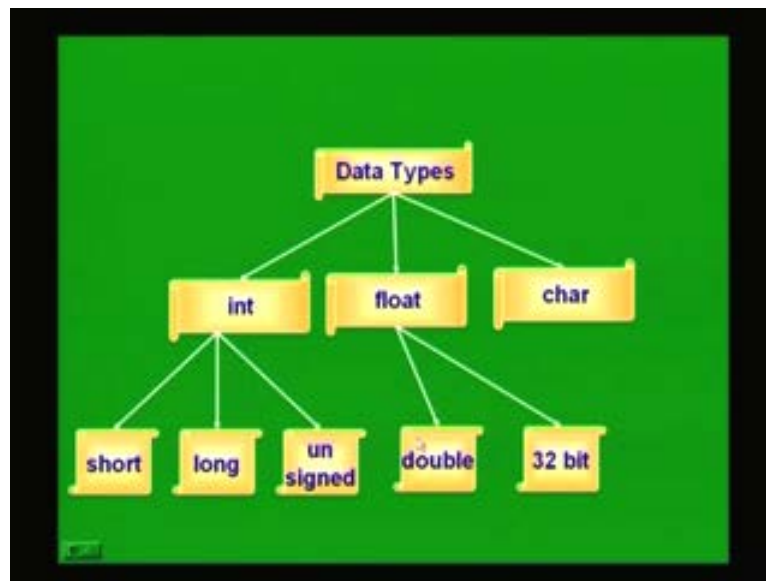
```
add ( x )
float x;
{
    static float y=0.0;
    extern float z;
    y=y+2;
    z=x+y;
    printf("%f\n", z);
}
```

Now, there are two different types of integers which we can declare. We can have long integers and short integers. So now we saw that in most of the recent compilers, the present day compilers, when you declare an integer “int”, it is always a long integer. So that means if you have a 32 bit computing, then you have 2 to the power of 32 is the largest integer which we can represent, unsigned integer okay. One bit is actually used for the sign. So we can actually have 2 to the power 30 or 31 as size.

So we have the third type, that is actually the unsigned. We can also use that bit which is reserved for a sign to represent a number, so that is why unsigned. So, unsigned can be little longer than the long integer. And then we have the floating point. The floating point also has two types. You could have the single precision or the double precision. So when you say “float”, that is a single precision okay. And then you said double, that is a double precision okay. We will see the difference in a program; [35:34] that’s the best way to see it okay or you could have characters okay.

You got 32 bit, or a double precision in the float point, and then you have characters. That is the basic data types which we will have. Let us use a, let us look at a program which uses all this data types. So here it is okay. S you have double x and y, have been declared as double. That is the floating point double okay; auto variables. And then I have declared them as long integers. As I said, in this, especially in the compiler which I have in this machine, whether I use integer or long integer it gives the same thing. It may not be true in all machines. If you are using large numbers, it is advisable to use it as long. Long integers are also represented as just long. You do not need to write “long int”. These have been declared as integers. And then I have a character, which is the letter, it is called a character. So this has been declared as a character. This is the name.

(Refer slide time: 35:48)



(Refer Slide Time: 35:56)

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

main()
{
double x,y;
long int i,j,k;
char letter,chain[]="a quick fox\n";

letter='B';
y=1.01;
//
i=9247562;
i=2;

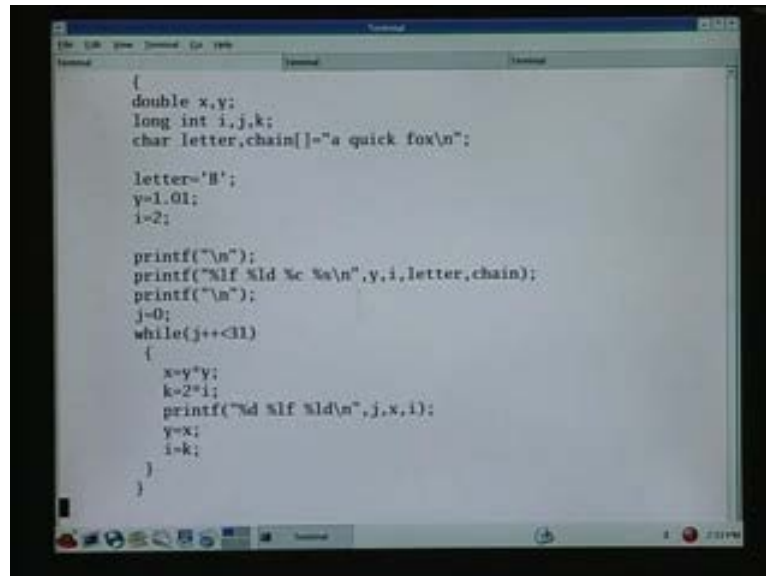
printf("\n");
printf("%lf %ld %c %s\n",y,i,letter,chain);
printf("\n");
j=0;
while(j++<31)
{
x=y*y;
k=2*i;
printf("%d %lf %ld\n",j,x,i);
}
```

I have another character, which is a character array, or a character string okay. For example, "a quick fox "; that is a string which I call chain. So these are examples of character declaration. You can declare a single letter character, you can use any name. I just use letter here or you could have said "chain", or an "array", or "a string of characters" okay. So in this program now I define letter as "b". Usually in single quotes, character is assigned. So this letter which has been declared as a character has been assigned a character "b" okay and this letter, this integer "i" has been assigned a number 2, and then I am printing them out.

So this example also demonstrates how do you use print statement. Okay so "printf" is actually print on the screen, as I have shown you in the previous example, you could open a file and use "s printf", that is, printing into a file. This example also demonstrates that we can print it on to a screen okay and now here is the format of the

print. So if you are printing a double precision number okay, you use “lf”, or you could use “e”, you will see what the differences are okay. For a long integer, you use “ld”, okay “percentage ld”. That is what you need. And the C program, if you give wrong type here for the print, it will print basically junk; it will not screen, but it will print junk in a C program. The characters are printed using a “percentage c”, and strings are printed using a “percentage s”. That is the format. Now we will run this, and then we will see what it prints. So here and the program has, the program continues.

(Refer Slide Time: 38:56)

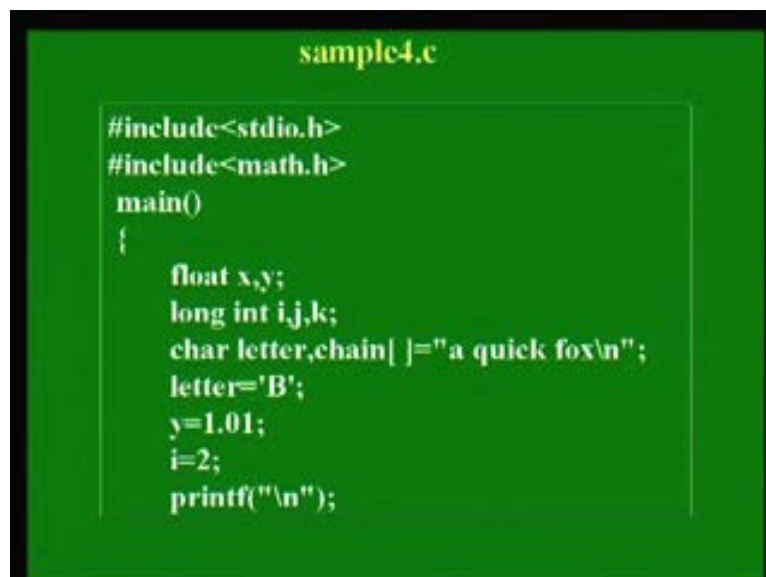


```
{
double x,y;
long int i,j,k;
char letter,chain[]="a quick fox\n";

letter='B';
y=1.01;
i=2;

printf("\n");
printf("%lf %ld %c %s\n",y,i,letter,chain);
printf("\n");
j=0;
while(j++<31)
{
x=y*y;
k=2*i;
printf("%d %lf %ld\n",j,x,i);
y=x;
i=k;
}
}
```

(Refer Slide Time: 39:58)



```
sample4.c
#include<stdio.h>
#include<math.h>
main()
{
float x,y;
long int i,j,k;
char letter,chain[]="a quick fox\n";
letter='B';
y=1.01;
i=2;
printf("\n");
```

It has a loop in which the j values increase from 0 to 31. j value is initialized to 0, and it goes up by steps of 1 up to 31 okay and the x value the y value, which is equal to 1.01, to start with, it computes the square of that and assigns that value to y okay. So basically, it keeps computing the square, that is 1.01 square, and square of 1.01

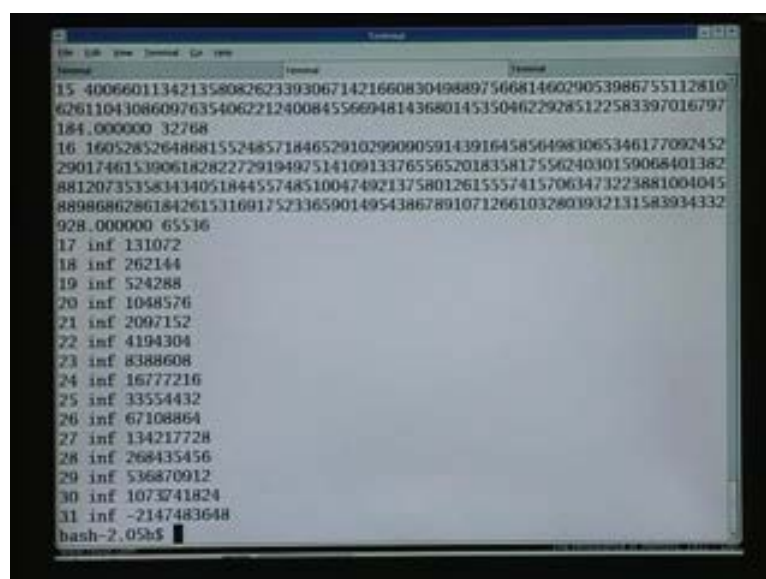
square, etcetera. So we will see how long it can go before it blows up beyond the capability of the computer. So what I want to show you is that this floating point, and these integers, there is a limit to the size of these numbers which the computer can handle, because it has a finite precision. That is what we are going to see.

(Refer Slide Time: 40:14)

```
printf("%lf %ld %c %s\n", y, i, letter, chain);
printf("\n");
j=0;
while (j++<31)
{
    x=y*y;
    k=2*i;
    printf ("%d %lf %ld\n", j, x, i);
    y=x;
    i=k;
}
}
```

So let us look at that in this example. Let us compile this and run it. So we will compile this again, this time, sample 4. Here I have not given any minus o; by default, it creates “a. out”. If you want some other name then you have to give minus o, otherwise it would always create “a. out”. So, “./a. out”, now, this prints, you can see that it prints as the loop goes, but first it prints the number y okay, and the integer I, and the letter which I assigned is b, and the string “a quick fox”. It has printed all that. Then it enters into this loop.

(Refer Slide Time: 42:42)



```
15 40066011342135808262339306714216608304988975668146029053986755112810
62611041086097635406221240084556694814368014535046229285122583397016797
184.000000 32768
16 16052852648881552485718465291029909059143916458564983065346177092452
29017461539061828227291949751410913376556520183581755624030159068401382
88120735358343405184455748510047492137580126155574157063473223881004045
88986862861842615316917523365901495438678910712661032803932131583934332
928.000000 65536
17 inf 131072
18 inf 262144
19 inf 524288
20 inf 1048576
21 inf 2097152
22 inf 4194304
23 inf 8388608
24 inf 16777216
25 inf 33554432
26 inf 67108864
27 inf 134217728
28 inf 268435456
29 inf 536870912
30 inf 1073741824
31 inf -2147483648
bash-2.05b$
```

You remember, that is the first statement was to print all those, things that is, y I, and the letter, and the chain okay. That is printed and then it goes into this loop and it is printed every time. Every time it squares it prints that. So that is the next part. The next part you can see, that is what it is printing. It prints 1.02, and 2, and it squares that, and it keeps squaring it. Here it takes the power 2 to the power of n, is what is being printed here and you can see that it can go, this is being declared in double precision, so you can see that it goes all the way up to 16, and you cannot handle it anymore. By 17, by the 17th loop, okay it has gone into infinity. As far as the computer is concerned, it is infinity. Infinity has been reached. And the integer continues to square power 2 to the power of n, it can go all the way up to 30, 2 to the power of 30 is the largest integer which it could print, it cannot do beyond that. Then it gives junk. This is a warning to C program users, that you have to be careful when you are using large numbers okay.

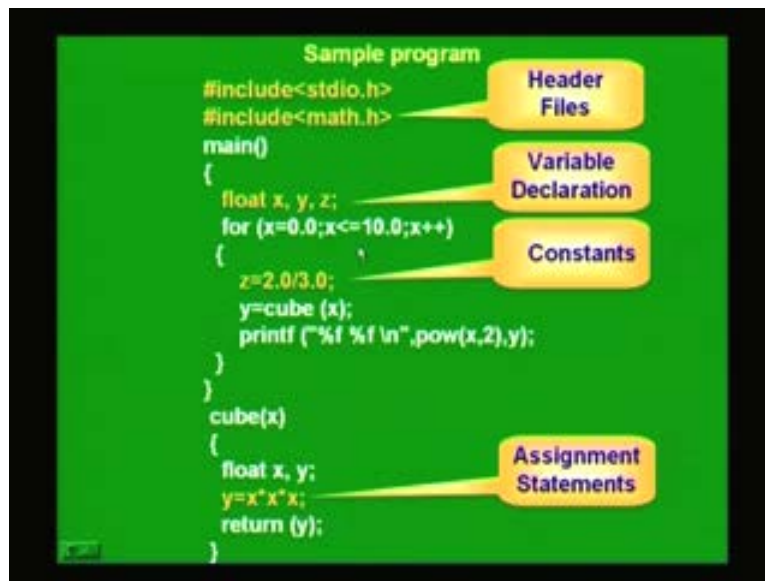
So, you get 2 to the power of 31, it is not screening, it is not saying that it is infinity, it's not telling you that it is not a number, nothing. It is just giving you a minus 2147483648. That makes no sense to you. So that is the thing. Let us change this and then let us make this just a float, that is, single precision, that is float okay. Now let us see what it gives. We compile it again and we run it okay. Now you see it could not go beyond 13. It went up to 16 last time, so beyond 13 it cannot go. It is already infinity.

So, if you want to use larger numbers, and if you also want to keep more precision, that is, the number of decimals you want to keep, number of digits you want to keep after the decimal, then you have to use floating, you have to use double. That is the two different data types of the floating point, and this integer types which I want to show. Unfortunately, I cannot demonstrate the difference between short, long, and unsigned on this computer, at least on this compiler, by default it uses long integer, and if you use any of this you will get 2 to the power of 30 as the largest integer which it can handle. But that may not be true in other machines, in other compilers okay.

So, going back to the basic program, so that is the summary of the program which we have so far seen, we had again, say that we had "include" files and we saw the use of a main function and a sub function, and then we saw how you declare variables, and how do you use print statements, etcetera. So as I said, these are the header files which have been handled by the preprocessor, and here are the variable declarations, and now I want to tell you something more here. This is a constant assignment. We are assigning here a constant z equal to 2.0 by 3.0 okay. So now, z is a floating point.

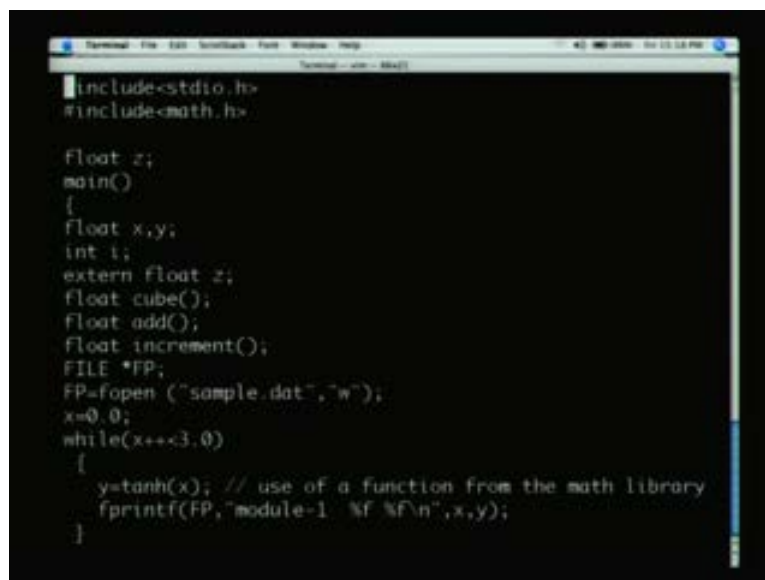
So now for us, if you write z is equal to 2 by 3, or 2.0 by 3.0, it is the same. It is a floating point, it is a real number. We know it is point something, right .6 something, we know that, but for a computer, they are different. If you write 2 by 3, and then if you take ratio of two integers, and then the computer assumes the result is an integer, or the nearest integer, so if I declare z as 2 by 3 without .0 by 3.0, then you are going to get 0. Just say z is a floating point and it is 2 by 3, then it's 0 for the computer okay. So you have to be careful. You have to say 2.0 by 3.0, and that is constant assignment, and we can also have, for example here, this also, this is an assignment statement, you are assigning some value to y. These are called assignment statements okay. So that is the basic summary of the code.

(Refer Slide Time: 46:25)



We will conclude today's lecture by looking at this anatomy of a program once again, that is, here is the summary of the program. First, we start with compiler preprocessor commands, which you might remember. It means that a standard C compiler has 4 filters and the command which take care of the first part is the preprocessor command which includes, which is actually the inclusion files, etcetera. So we will look at that program again, and we will see once again, what these things mean and then we have the main function with the function name given there, and the main function will have variable declarations and the main code.

(Refer Slide Time: 47:56)



And then you could have many such sub functions, or may not have any functions at all, sub functions at all. So let us look at what is meant by variable declaration. In variable declarations, we would declare the variable with a class and a type and a name, and sometimes with an initial value also given. This is not necessary, but



sometimes this also can be given. So we look at all of this through a sample summary program that is shown here in this which I have made this program, which summarizes what all things which we have looked at now.

First, here is the preprocessor commands which says that here include the standard “io” library and the math library, and then I have one declaration here of a variable, so we will look at what these declarations are one by one. So here is the main program. Sometimes, we do as we saw earlier. We have external variables. We declare the variables outside the function, and that is shown here outside the main function, and if it is not external then they are declared after the main program starts. So here is the declaration of floating point variables x and y, with type, float, and class, auto; and then you have a integer variable declared.

Now I am using this external variable so as you remember, as you may remember, that we have to declare them again here, so that is external type variable which is external class, and then “float” is the type, and then z is the variable. So “float” is declared outside the main function because it is an external variable, and then we have functions, sub functions here. There are 3 sub functions here. They are called “cube”, “add” and “increment”. Note that these sub functions also have a type floating point variable, or floating type. The reason for that is these sub functions actually return something to the main program. As you can see here, for example, the cube, this sub function and this cube actually returns the variable y which is of the type “float”, so the function itself has to be declared of that type.

So that is the floating point sub function calls, and then here is another declaration, so this is declaration of the sub functions, and the declaration of the variables, the auto class, the external class okay and then here you have the declaration of a file pointer “fp”, and then this after the declaration you have the first assignment statement which is “fp” is now a file pointer, is now used to open a file called “sample. dat”, and here is an assignment which gives x as value 0.0. So now, as I said earlier, remember that you could also use this initialization. You could also do it here. You could say float x equal to 0.0 to initialize this function.

And then this program has a small loop here in which you compute tan hyperbolic of the variable x. So that is the assignment statement here, y which is declared as the floating point is now assigned to tan hyperbolic of x, x is another floating point. So tan hyperbolic is now here a function. Now this function is not declared anywhere here because it is part of the standard math library, and that is what we have used here using “math. h”. It is part of the standard math library. now this demonstrates the use of the function from the math library and here is a way of using, once we calculated this, here is the method to actually write it on to a file, so I use “f printf” to write it into a file “fp”, and in that “fp”, I write apart from some comments which says module 1, that is this part of the program which is writing this, and then there are the 2 floating points which is now commented by percentage f and percentage f.

So, that is to say that that is floating point which you are writing and that is x and y and then we close that file “fp” and now here is another function which I call, this is another function which doesn’t return anything. It is actually void. So if I actually declare it here, I should declare it as void, but many of the compilers you do not have to declare it, but many of the older compilers you will have to say “void”, for

example, here you will have to say “void”; you have to say here “void print init”. That is the correct way of doing it, but many of the compilers assume if you don't declare it that it is indeed a void function. So here is another loop.

Now this is a way of calling a function which is now this loop again, similar to the earlier one, this loop, but here now we are calling this function cube of x, and that is something which I have returns, and that returns a value, a floating point, and that floating point is assigned to this. This program now simply prints it out on the screen. You have to note that I am using “printf” instead of “f printf”. So if I run this program, it will simply print this on the screen. We can see that here, if I run this program. So I have this program compiled and it runs.

So if I run this, when this program runs, it just prints it on the screen. This is the module to printing the variable 5 okay that is the x value, the square of that, and the cube of that, and that is what this part of the program module 2 is actually printing, right, it is printing the x variable which is 5, because it went up to 4 here and starts from here at 5, and then it prints out 5 and power 2. Now this is again, a function which is part of the math library. So it is power x square, it can compute x square power x 2. It is part of the math library, and it is again to show you that you do not need to declare something like this here and then put it into the print statement.

You can also put the assignment statement straight into the print statement if you want to if all you need to do is simply print it, and then here y is the cube of x. I could have simply put cube of x here to print it out also and then here is another program which is called add of x, which is again part of the sub function it calls starting from x equal to 0. Initialize x equal to 0 again here, and I pass it into this y to add of x and to return something, and that's printed as y, and I print that out x and y here. So you remember I am putting it here as x equal to 0, and then I pass it here and it returns something.

So, what does it return? Okay so if you look at this you will see that add of x, actually what it does is it takes the value x and it adds that x and y together. y is now initialized here to 0. It increments y by 2 and then adds that to x. Remember x is equal to 0 which is passed, so it is x plus y is what is going to be returned. So, y here is a static variable, so the next time when it is called it is not initialized. So this program is just to demonstrate the static variable. This variable y is only initialized once. Okay so we can see that again when we run this code here, this is part of the module 3.

You see we are passing x all the time as 0. x is always passed as 0, but what it returns is 2, 4, 6. If you look at module 3, you will find that x is always passed as 0 here, but what it returns is 2, and then 4 and then 6. So the reason for that is only in the first call, the y here is initialized okay the next time even though we have written y is equal to 0, that is initialized only the first time it is called to that function, and the next time it is called, it is incremented by 2, and z here goes from x plus y but x is always 0 because it is passed as 0 here. So z goes as 0, 2, 4,, etcetera. That is what it is demonstrating. So that shows us how to use floating point variable, sorry auto variables, and also external variables like this, and also how to use static variables and we will continue on this programming in the next class.