High Performance Computing

Prof. Matthew Jacob

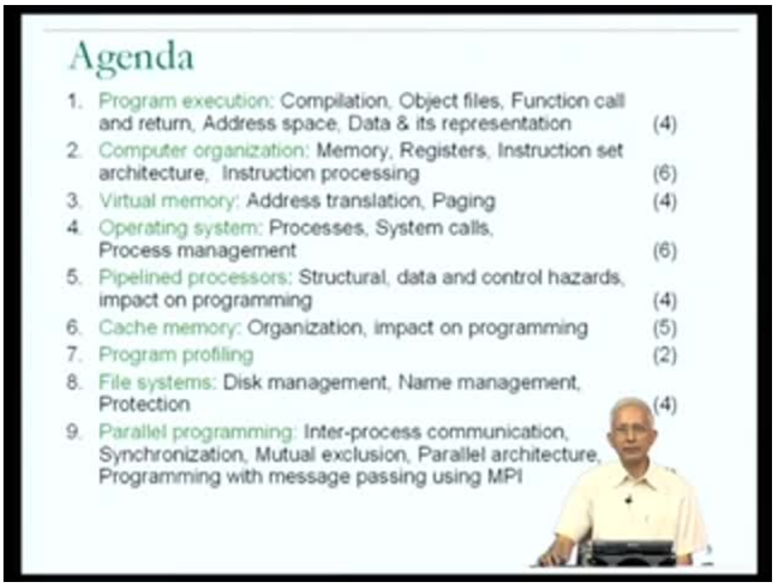
Department of Computer Science and Automation

Indian Institute of Science, Bangalore

Module No. # 01

Lecture No. # 01

(Refer Slide Time: 00:25)

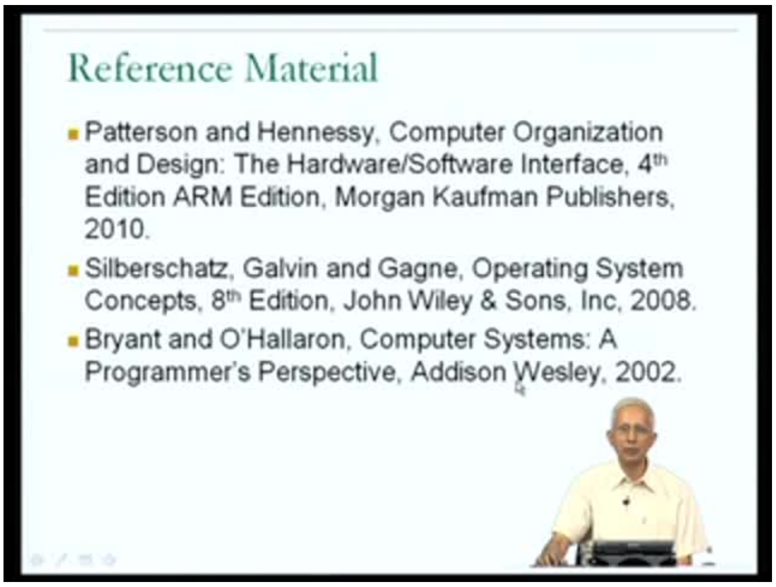


Agenda

1. Program execution: Compilation, Object files, Function call and return, Address space, Data & its representation (4)
2. Computer organization: Memory, Registers, Instruction set architecture, Instruction processing (6)
3. Virtual memory: Address translation, Paging (4)
4. Operating system: Processes, System calls, Process management (6)
5. Pipelined processors: Structural, data and control hazards, impact on programming (4)
6. Cache memory: Organization, impact on programming (5)
7. Program profiling (2)
8. File systems: Disk management, Name management, Protection (4)
9. Parallel programming: Inter-process communication, Synchronization, Mutual exclusion, Parallel architecture, Programming with message passing using MPI

Now, welcome to this first lecture on High performance computing. My name is Matthew Jacob. I am a Professor at the Indian Institute of Science. Let me start, by giving you a complete picture of what this course is going to cover. Basically, we have a nine point agenda and if you look at this list of topics I have put, this is the syllabus which you have also seen in a separate location. For each of the nine topics, I have put the number of lectures which I expect, will take to cover that material and you notice that, there are a fairly large number of topics, but if you looked at it carefully, you would observe that, half of, about half of these topics relate to normal computer systems, in which there is only one processor and the others relate to parallel computer systems, where there is more than one processor.

(Refer Slide Time: 01:05)



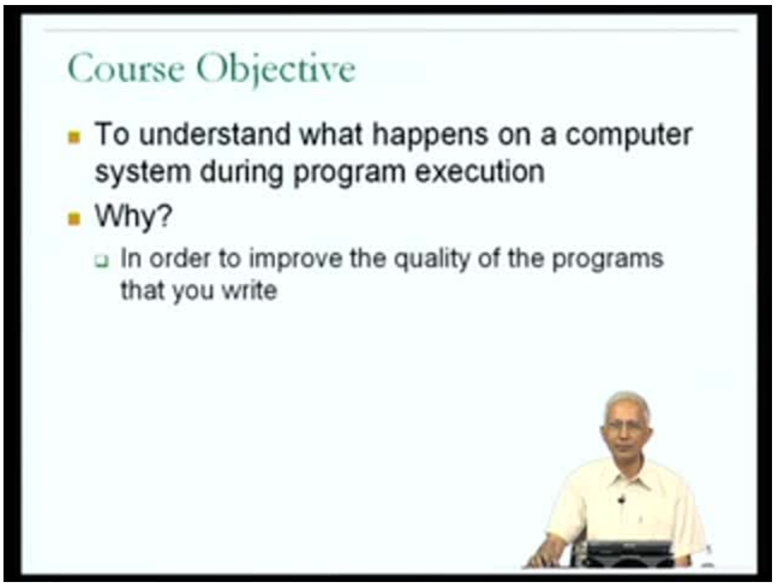
Reference Material

- Patterson and Hennessy, Computer Organization and Design: The Hardware/Software Interface, 4th Edition ARM Edition, Morgan Kaufman Publishers, 2010.
- Silberschatz, Galvin and Gagne, Operating System Concepts, 8th Edition, John Wiley & Sons, Inc, 2008.
- Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Addison Wesley, 2002.

So, this is in fact, the objective of this course, to learn a little bit about the different kinds of computer systems. Now, you will be thinking you we will be wondering what are, what are the different text books or reference material that might be important for this course? And here, I have mentioned three books which I think will be of great value. One is a book by Patterson and Hennessy. It is about Computer Organization. Second is a book on Operating Systems by Silberschatz, Galvin and Gagne and the third is a book on the Programmer's Perspective about Computer Systems.

So, all of these books are available in Indian edition or International edition at fairly low price and well worth buying.

(Refer Slide Time: 01:59)



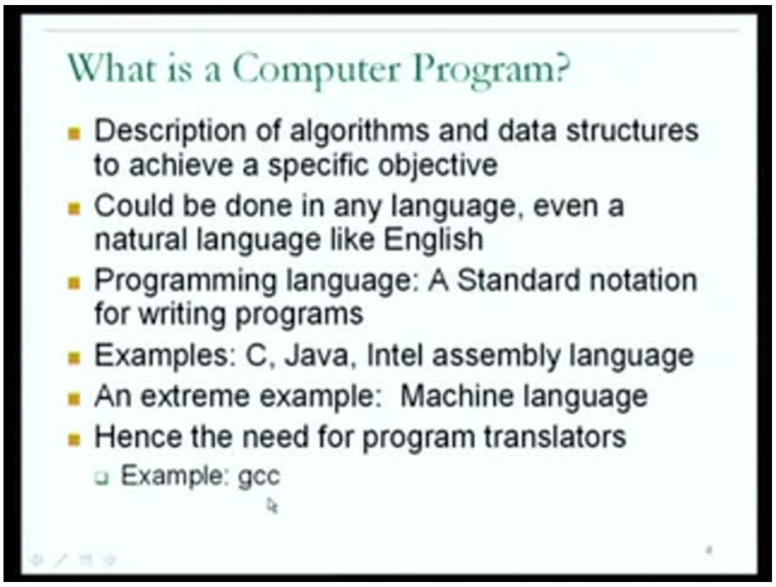
Course Objective

- To understand what happens on a computer system during program execution
- Why?
  - In order to improve the quality of the programs that you write

So, when you look at this reference material, you will get the impression that, this course is going to cover computer organization, followed by operating systems, followed by something about the programmer's perspective, which raises the question of what is the course objective? Let me state the course objective quite clearly. Our objective in this course is to understand what happens on a computer system, when a program is running on the computer system. We want to get a somewhat complete understanding of what is happening, not only in terms of the variables of the program, but also the interaction between the program and the operating system or the interaction between the program and the underlying hardware, which is why, we are trying to learn a little bit about computer organization, something about operating systems, as well as something about the programmer's perspective on the execution of a program.

Again, the question which may arise in your mind at this point is, why, why have, why go through a course in which this is the objective? And, the reason is basically that, if you know a lot about what is happening when your program is executing on a computer system, you can use that knowledge to improve the quality of the programs that you write. Possibly, you could make a program, change your program a little bit so that, it runs a little faster, or you could change a program little bit slow that, it runs using less memory or various other possible criteria that you might use to improve the quality of your program.

(Refer Slide Time: 03:17)



What is a Computer Program?

- Description of algorithms and data structures to achieve a specific objective
- Could be done in any language, even a natural language like English
- Programming language: A Standard notation for writing programs
- Examples: C, Java, Intel assembly language
- An extreme example: Machine language
- Hence the need for program translators
  - Example: gcc

So, this is our objective. We hope that, by the end of this course, we will be in a position where we are writing programs with better, better use of the system, on which the program is going to run.

Let me start with some basic, by trying to answer the some more basic questions, so, there is no doubt in anyone’s mind, what I mean by a computer program.

So, what is a computer program? In a course on programming or the first course in computer science that you take, you will be told that, a computer program is a description of algorithms and data structures to achieve a specific objective. So, you write a program with some objective in mind. For example, you might be writing a program to multiply two matrices. That could be the specific objective and the algorithm is the series of steps, the sequence of steps that you believe must be done, in order to achieve that objective and the data structures are the various species of data that are necessary in order to achieve the objective.

Now, a computer program could be written in any language. In fact, you could even write a computer program in a language like English or your, your mother tongue, because you know that, you can describe algorithms and data structures in that language. But there are certain standard ways, standard notations for writing programs and these are what are known as programming languages. The programming language is a standard notation for writing programs, and you would have heard of several programming languages. You would have studied some. You would have heard about C, Java, Intel assembly language.

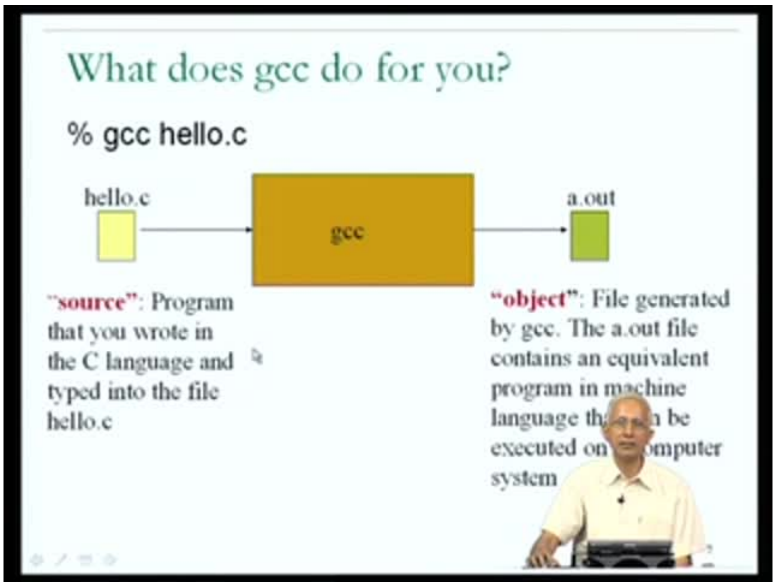
And when you look at these, if you have any familiarity with these languages, you will notice that, they are actually quite different. Because when you read a C program, or a Java program, it is somewhat easier to understand than, when you read an Intel assembly language program.

So, in, in the extreme case, you could think of a language, which is even harder to understand when you read it and that is, for example, the I-32 machine language. And the situation that we have here is, you have to bear in mind, that you write a program to achieve an objective and you write the program in a language that you understand, but ultimately the program must run on a hardware, on a piece of hardware on a computer and therefore, ultimately the program must be understood by the computer in a language that it understands.

So, there is the, there is this gap between what you would prefer as the language in which to write the program and what the computer would prefer, as a language in which to the execute program.

So, typically the, this gap between the languages, language of choice of the program and language of choice of the hardware makes it necessary for the programs that are written by human beings to be translated using program translators into a form that can be executed on a computer system. And one example of such a program translator is the compiler g c c. Some of you may have heard of g c c or even used it, because it is fairly widely available on most computer systems. g c c stands for the gnu c compiler.

(Refer Slide Time: 06:08)



What does gcc do for you?

% gcc hello.c

"source": Program that you wrote in the C language and typed into the file hello.c

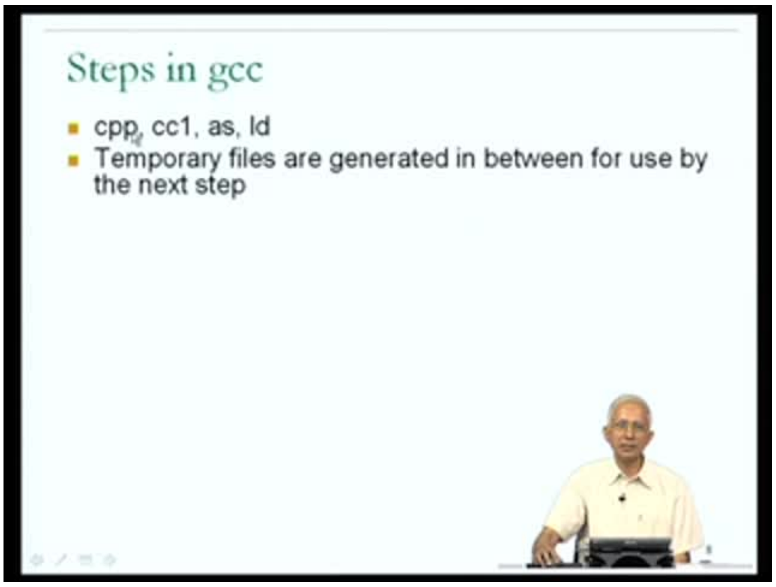
"object": File generated by gcc. The a.out file contains an equivalent program in machine language that can be executed on computer system

So, let us think a little bit about this. What does g c c do for you? Now, those of you who have used g c c on, let us say a Linux system or Unix system, will be familiar with the fact that, to the command prompt, which is this percent sign that you see over here, you execute the program g c c, providing hello dot c as the input. So, hello dot c is the file which contains the program that you have written in the C language.

And what g c c does for you, it takes your hello dot c program, which you have written and produces as output, a file called a dot out. The default name of the output produced by g c c is a dot out. And, terminology that is used is, that we refer to the input program to g c c or to any program translator as the source. So, it is the program that you have written in the C language and typed into the file called hello dot c. hello dot c is the name of a file.

So, g c c takes in this file, which contains a C program and generates a dot out, which is what is known as, an object file. So, g c c, a dot out is the file generated by g c c and as described in the previous slide, it contains an equivalent program to the one that you wrote in C, in the machine language, so, that it can be executed on a computer. So, this is essentially how a program translator works. It takes in a source program in a higher level language, such as C and translates it into an equivalent program in a lower level language, such as the machine language.

(Refer Slide Time: 07:44)



Steps in gcc

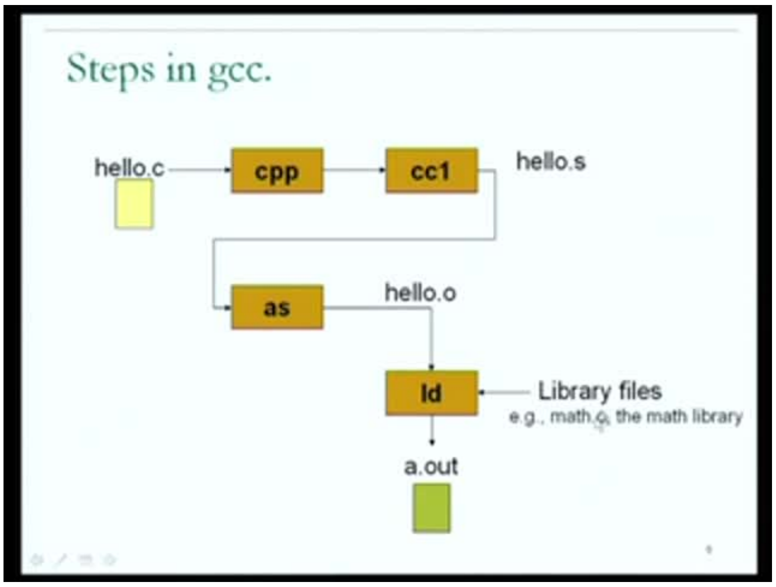
- cpp, cc1, as, ld
- Temporary files are generated in between for use by the next step

Now, in doing this, g c c actually does, does this in several, several steps, and it will be useful for us to understand a little about some of the steps the g c c goes through, in translating a source programm in C, let us say, into an object program in machine language. And, the names of some of these steps are c p p, c c one, a s and l d. These are just the names of the steps. They may not make too much sense to us at this point in time.

But the important thing is that, in the process of translating the source program hello dot c into an object program a dot out, g c c uses these steps and along the way it generates temporary files. For example, after c c one has finished its work, it generates an output file, a temporary file which is then, later on used by a s to do its work in the next step.

So, these temporary files, if they are available to us, may provide us with some idea about what is happening to our program, as it gets translated by g c c. It why I mention this.

(Refer Slide Time: 08:50)


Steps in gcc.

Ok, so, the way to look at this is that, your, your program hello dot c which was written in C is taken in as input by the first step in g c c, which is called c p p or the c pre processer. Some activity is done on your, some form of translation is done on your C program and the next step is, what is called c c one, which once again does some transformation of your program. It generates a temporary file called hello dot s, which is the input to the next step in the g c c translation, which is called a s. a s generates a temporary file called hello dot o, which is used as the input to the next step in the g c c translation, which is called l d.

l d may take in some additional files from, let us say, a library and do some merger of these hello dot o along with the library files, to generate the a dot out object executable file.

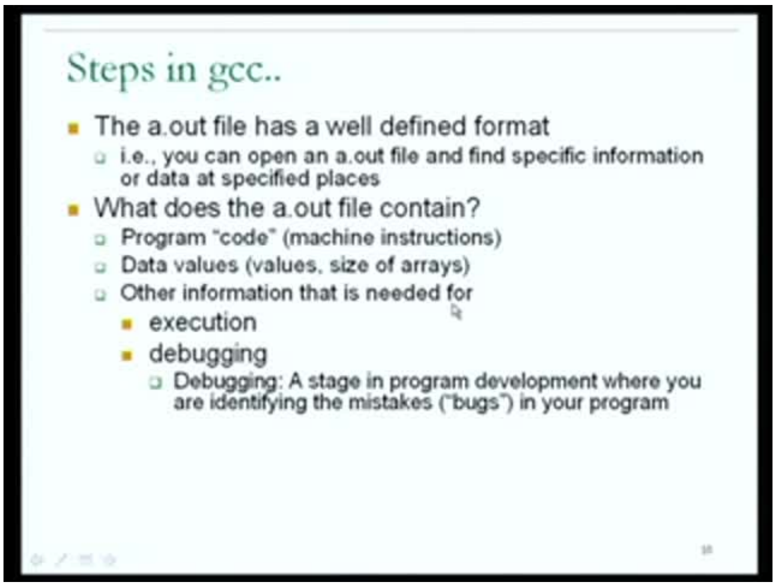
So, up to now, we were looking at this entire collection of four boxes as one box, which I had labelled g c c. In this slide, we have understood that, there are a few steps which take place in between and that, if we know the names of the temporary files, or if we ask g c c to show us the temporary files, we may be able to get a little bit more insight into what is happening, in terms of the translation of our program.

So, this is a useful piece of information, that may be relevant later in the course, ok.

Now, let me just say, a little bit about the library files. Many of you may be aware that, some of the programs that you write, for example, if you are writing a program doing some calculations, scientific calculation of some kind, you may be using certain library functions which might be available in a, a library file. For example, if you are using the sine or cosine or logarithm functions, you are not writing those functions yourself. They are written by somebody else and available in, what is called the library file.

So, if your program the hello dot c requires such a function, like log or sine or cosine, then it is pulled into your program, prior to the generation of the a dot out, so that, the final executable object file contains those library functions, such as log or sine, as needed by your program. So, just recapping.

(Refer Slide Time: 11:03)



Steps in gcc..

- The a.out file has a well defined format
  - i.e., you can open an a.out file and find specific information or data at specified places
- What does the a.out file contain?
  - Program "code" (machine instructions)
  - Data values (values, size of arrays)
  - Other information that is needed for
    - execution
    - debugging
      - Debugging: A stage in program development where you are identifying the mistakes ("bugs") in your program

The steps in g c c are, as we have seen, and the, the net result is that, you get file called a dot out. Now, the next thing which we will consider is, even as the temporary files that are generated by g c c and might be accessible to us during the compilation process, even if they are not of interest to us right now, we know for a fact that, the a dot out file is of interest to us. Because, that is the file that we will want to execute, to run our program. We wrote hello dot c with a particular objective in mind, such as multiplying matrices. If our, to actually do matrix multiplication, we will now execute the a dot out file. So, a dot out, a dot out file is of great interest to us.

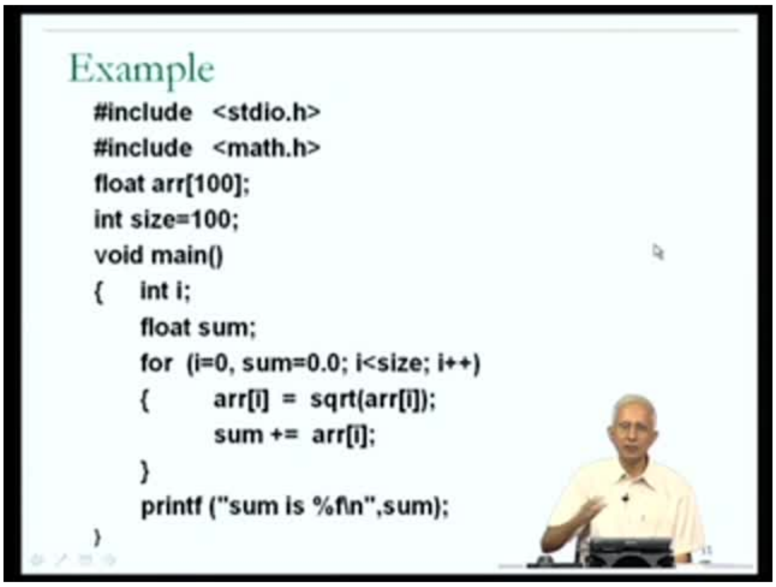
Now, it turns out that the a dot out file has a very well defined format. What I mean by this is, that if you could read the a dot out file and you knew what the format was, you would know where the different things of relevance to you are available in the a dot out file.

So, if you could actually open an a dot file, you would find specific information or data at specific places, that is what it means to have a well defined format. Which raises the question, what does the a dot out file contain? When you wrote the hello dot c C program, you knew exactly what it contained. It contained the statements, the variable declarations and so on.

So, you suspect that the a dot out file must contain equivalent things in the machine language. If you actually did study the format of the a dot out file, you would find that, the a dot out file contains machine instructions, which are the steps of the algorithm that you had used to describe the objective that you had in mind. This is referred to as the code of the program. Because it is machine instructions which is difficult to understand, and (()) like a code. In addition to this, the a dot out file will contain certain data values, possibly the sizes of arrays, possibly the constants which are declared in your program, probably not the different values which a program reads in as input, because, those would be available only when the program is executed, but every other piece of data would be available inside the a dot out file itself.

So, we suspect that, the a dot out file will contain the machine instructions, which describe the algorithm. It will contain the data values which are known prior to the execution of the program and it will also contain other information which might be needed for the execution of the program, as well as, for things like debugging, it is various other pieces of information, which we will learn more about, as the course progresses. You may have heard of this word debugging before. Some of you, who have written programs and had to fix the mistakes in them, will know that, debugging is the stage into program development, where you are identifying the mistakes or bugs in your program and in debugging you identify the mistakes and you correct them, so that, the net result is, after debugging , you have a correct version of the program.

(Refer Slide Time: 14:08)

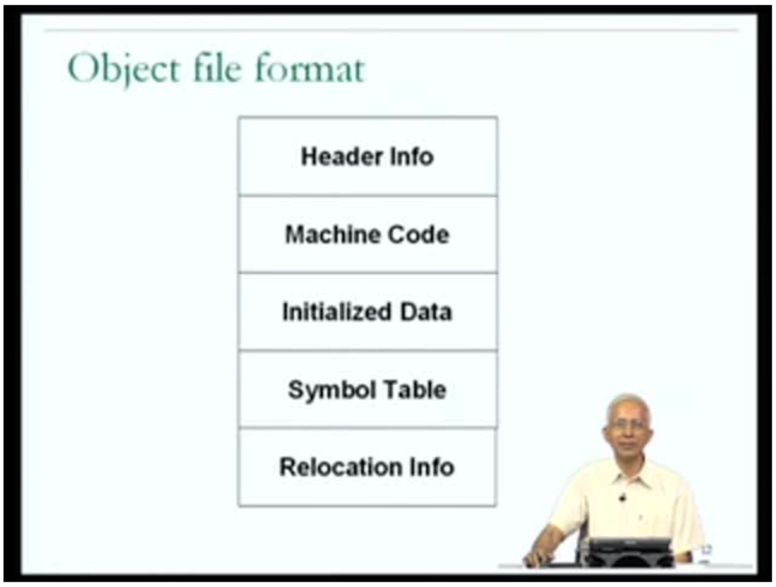


Example

```
#include   <stdio.h>
#include   <math.h>
float arr[100];
int size=100;
void main()
{     int i;
      float sum;
      for  (i=0, sum=0.0; i<size; i++)
      {        arr[i]  =  sqrt(arr[i]);
               sum +=  arr[i];
      }
      printf ("sum is %f\n",sum);
}
```

So, it is possible that the a dot out may, could be generated, so that, it contains some information which should be help of to you, when you are debugging a program. This is, look at a very simple example. What I have written over here, what you see in this slide, is a very simple C program. Many of you will be familiar with these include steps at the beginning, which request, which indicate that you are likely to be using standard i o functions, from a library, as well as math functions from a library. This program is dealing with a floating point array and you know something about computational real numbers. It is a dealing with an integer variable called size and then it does some computation using a for loop. Ultimately, it does a print f.

(Refer Slide Time: 14:54)

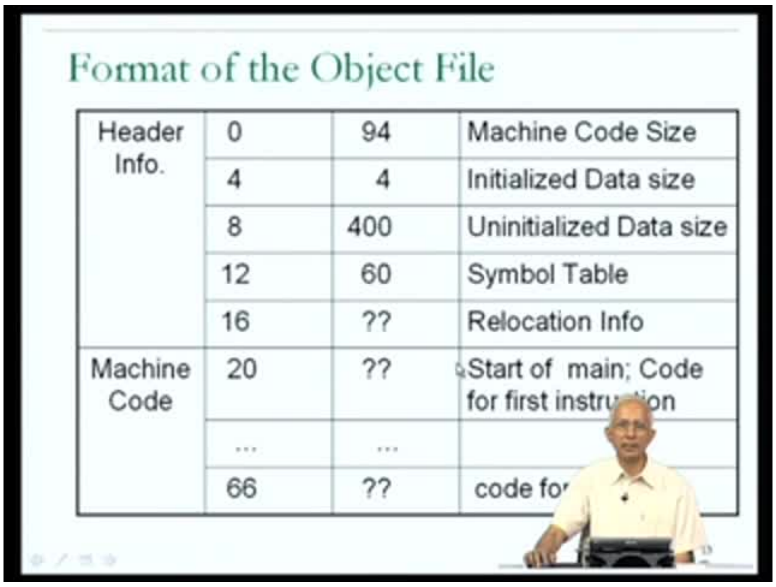


Object file format

Header Info
Machine Code
Initialized Data
Symbol Table
Relocation Info

So, this is a simple program, which, you may have seen programs like this before. As a question, which we are interested in now is, for such a simple program, what is likely to be present in the a dot out file? Now, as a step towards understanding this, I am showing you over here, a block, a diagram, which indicates, what are the different pieces of a such an a dot out file. So, if I was to somehow open the a dot out of that program that we just saw, apparently I would first see some header information. I would then see the machine code, which we now understand means the instructions corresponding to the steps of the algorithm, which, which you have encoded in your C program. After this, the a dot out file contains some of the data of your program. Then, there is something called the symbol table and something called relocation information, which at this point, we are not very clear about what they mean.

(Refer Slide Time: 15:36)



Format of the Object File

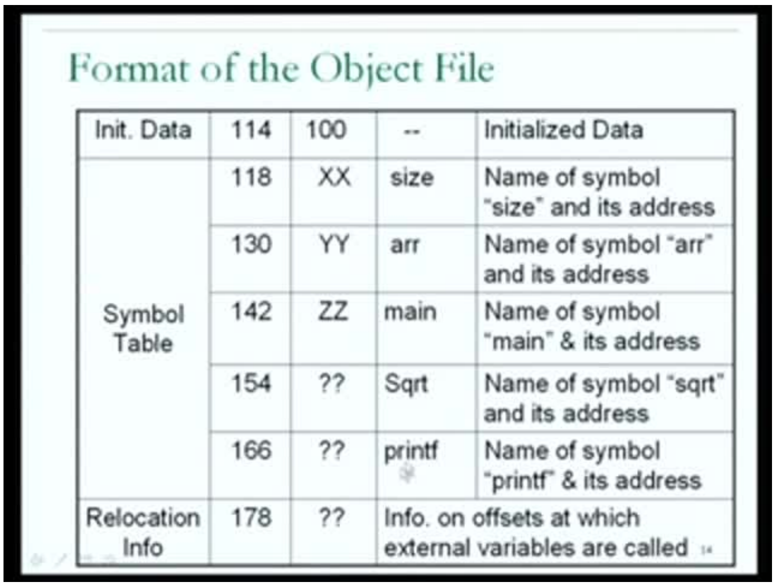
Header Info.	0	94	Machine Code Size
	4	4	Initialized Data size
	8	400	Uninitialized Data size
	12	60	Symbol Table
	16	??	Relocation Info
Machine Code	20	??	Start of main; Code for first instruction
	…	…	
	66	??	code for

Now, if I actually went through the exercise of opening the a dot out file, and over here, I am not showing you the contents of the a dot out file, but I am sort of giving you an idea about what different pieces of information might be available in those. Remember, there I have indicated that there are 4 or 5 different parts to an a dot out file.

So, first of all, you may be asking what information is present in the header. So, apparently in the header, if you look at the beginning of the header, in other words, offset 0 of the header, there is information about the size of the machine code. Apparently the size of the machine code, in this particular case is 94 bytes.

After there, that, there is some information about the size of the initialized data and that is available in the a dot out file at an off set of 4, whatever that means.

(Refer Slide Time: 16:31)



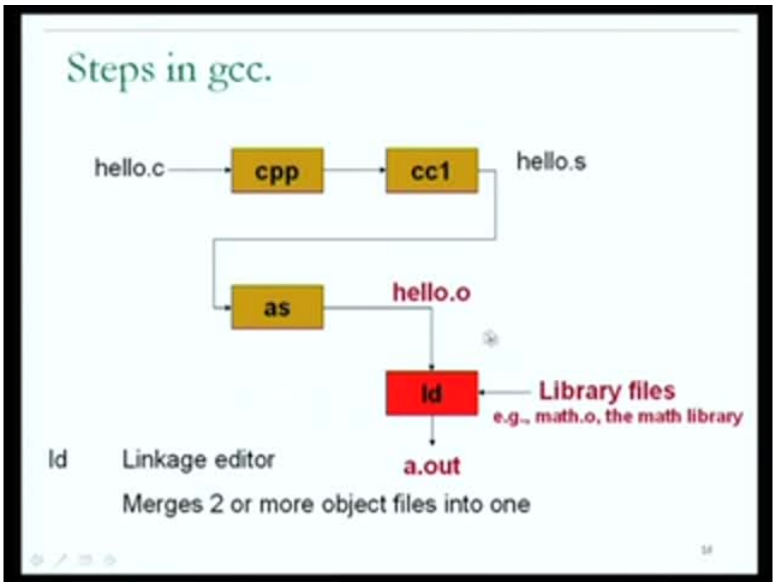
Format of the Object File

Init. Data	114	100	--	Initialized Data
Symbol Table	118	XX	size	Name of symbol "size" and its address
	130	YY	arr	Name of symbol "arr" and its address
	142	ZZ	main	Name of symbol "main" & its address
	154	??	Sqrt	Name of symbol "sqrt" and its address
	166	??	printf	Name of symbol "printf" & its address
Relocation Info	178	??	Info. on offsets at which external variables are called	

So, there are various pieces of information in the header. This is followed by the machine code, which is the machine instructions corresponding to the program, followed by the data, followed by this thing called the symbol table, which if you look at now, we begin to understand. The symbol table seems to contain information about the various symbols or identifiers that you had used in your program. If you recall the program that, the, we had talked about, there was a variable called size. There was an array called array a r r. There was a function called main, was also referring to function called s q r t and a function called print f. So, all of these are the symbols or identifiers of the program.

So, the symbol table contains one entry for each of those symbols and finally, there is the relocation information which contains the information on the offsets at which external variables are called. So, this clearly is some information which is going to be very important for some later step of g c c.

(Refer Slide Time: 17:24)



Steps in gcc.

$$\begin{tikzcd}
\text{hello.c} \arrow[r] & \text{cpp} \arrow[r] & \text{cc1} \arrow[dll, "\text{hello.s}"] & \\
 & \text{as} \arrow[dr, "\text{hello.o}"] & & \\
 & & \text{ld} \arrow[d] & \text{Library files} \arrow[l] \\
 & & \text{a.out} &
\end{tikzcd}$$

ld Linkage editor

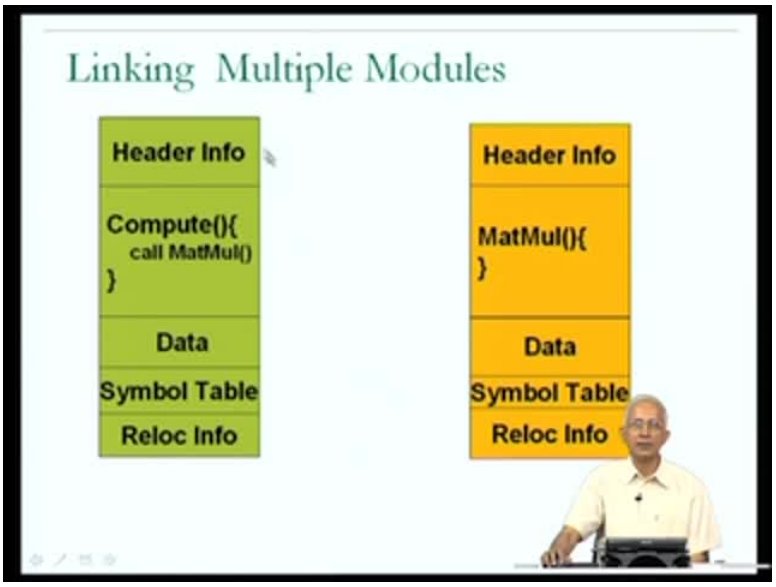
Merges 2 or more object files into one

Right now, going back to the slide that we saw, about the steps in g c c, the hello dot c program is first operated on by the, within g c c, first goes through c p p, then goes through c c one, then goes through a s, then goes through l d. Let me concentrate on what happens in the step called l d. So, I will recall that, in the step called l d, a temporary file generated by the previous step of g c c which was a s, is taken in along with library files, and the, what l d does is to produce a single output file called a dot out.

So, from this description, we would suspect that, what l d does, is to merge multiple files into a single executable object file. Now, l d itself, the word, the name l d stands for linkage editor and what it does, is exactly what I had just described. It is a program or a part of g c c, which merges two or more object files into one.

So, one of these object files, is a file that was generated by the previous steps of g c c, as an equivalent of the hello dot c program that you had written in C. That is being merged by l d, along with whatever library files are necessary to achieve the objective, that is described in hello dot o. And, they are merged into a single executable file, since what you want to do is, to execute a single executable file. So, let us just look into this operation of linking, a little bit more.

(Refer Slide Time: 18:50)

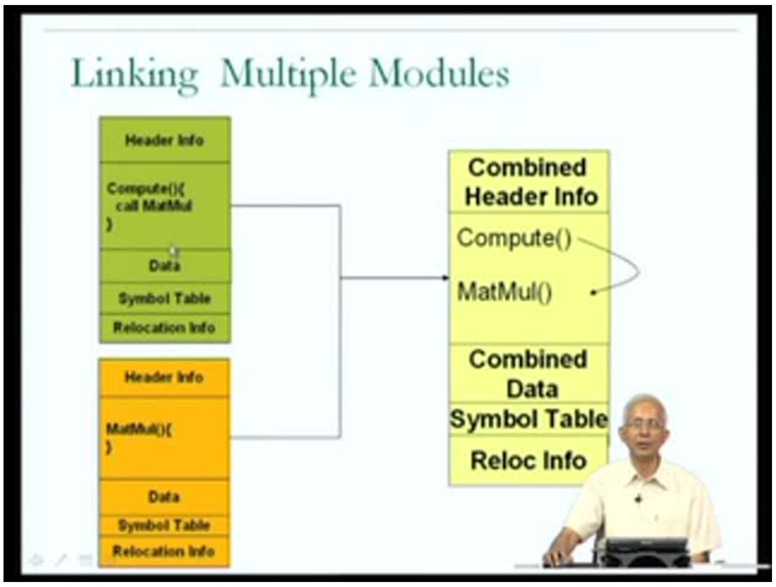


Linking Multiple Modules

So, the situation in linking is, that I have two object files. This might be the object file generated over here, hello dot o. So, this might be hello dot o and the one on the right might be math dot o, if my, if my program is using a math function; then I would need to link, join the hello dot o along with a math dot o, to generate a a dot out. So, the question, if you, if you recall, any one of these object files contains header information, code, data, symbol table, relocation information.

Now, in order to merge them into a single file, obviously, the different components of these separate files would have to be merged.

(Refer Slide Time: 19:35)



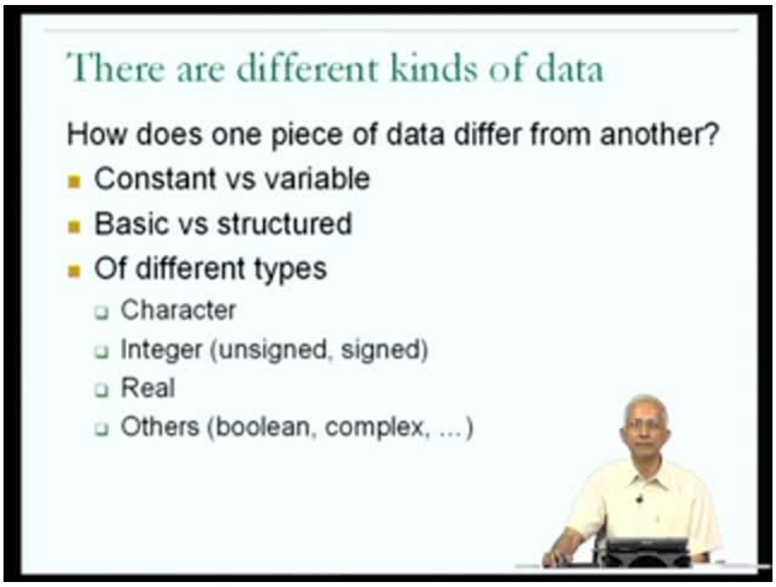
Linking Multiple Modules

So, what l d does is, it takes the different object files which have to be merged into one and combines their headers into one header. It combines their code into one code, so just, if you examine the, the green, the upper block, that is the object file corresponding to my math, to my hello dot o, generated from my hello dot c program and it contains a call to a function called MatMul. Now, the function MatMul itself, is apparently not available in my hello dot c program.

But it is available, in another file and therefore, comes from another object file. Therefore, the code for MatMul is available in this orange object file, down below. Therefore, when I look at the combined object file, I find out that there is a combined header, there is a single code segment, which contains both the green code as well as the orange code, combined data and combined symbol table, combined relocation information.

So, that is basically all that l d does. It is actually not a very complicated operation and is therefore, somewhat easy for us to understand. So, this is something about the different steps in g c c. The important thing to recall is, that some of the in steps produce intermediate files, which may be accessible to us, if we need them. Now, let us move on from here.

(Refer Slide Time: 21:02)


There are different kinds of data

How does one piece of data differ from another?

- Constant vs variable
- Basic vs structured
- Of different types
  - Character
  - Integer (unsigned, signed)
  - Real
  - Others (boolean, complex, ...)

Now, that we have understood what a program is - a program is a description of algorithms and data structures, which you have written to achieve some specific objective. So, from now on, we understand that a program will have associated with it, instructions or the steps describing the algorithm as well as data.

We will therefore, start by trying to learn a little bit about data. If you want to understand what happens when a program executes, we will have to understand a lot about what happens to data or what data looks like. It should be our first objective. Now, let me start out by saying, that as you are probably aware, there are different kinds of data. For the different kinds of programs that you write, you will find, that there are different kinds of data and we need to be more specific. Let me therefore, ask the question, can we, can we list out, how one piece of data differs from another piece of data?

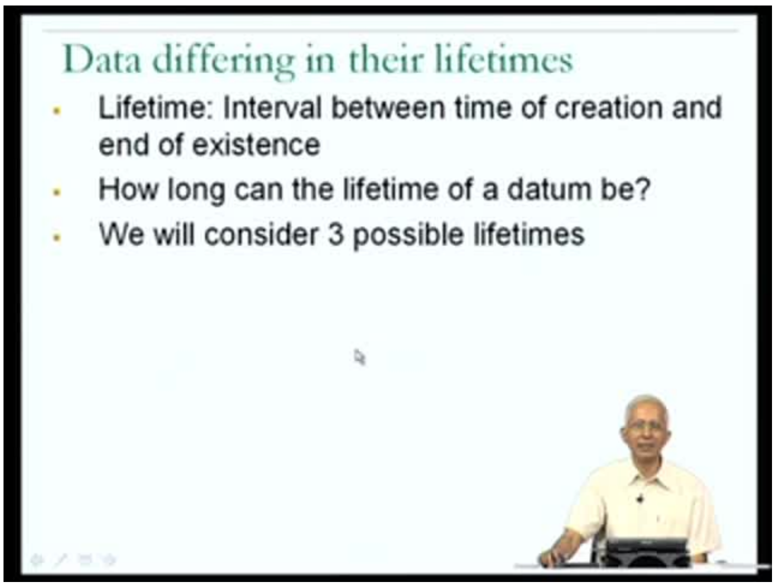
Now, from your experience with programming, you will be aware that, there are some data, some pieces of data, which are constant, in that, they do not change in value during the execution of a program. Whereas, there are other pieces of data which are variable. For example, if I have a program in which I have a variable called x, integer x, I could have a statement which changes the value of x and x is therefore, a variable. Whereas, I could have a program, which uses a constant, for example, the constant pi, which has a fixed value of 3.14159 etcetera. So, intrinsically the constant pi and the variable x are different from each other.

So, that is one way in which data can differ. Both of these are pieces of data. Another way in which data can differ is that, some data is basic, whereas, other basic, other data is structured. And what I mean by piece of data that is basic is that, it has a single value associated with it at any point in time.

For example, the integer variable x has only one value associated with it, at any point in time. On the other hand, an array of size 100 is called structured, because it has 100 different values associated with it. Similarly, you would be familiar with the c struct, in which there could be different fields, each of which is capable of having a different value associated with it. So, there is some data which is basic, single valued and other data which is structured, containing multiple values.

Ok, now, another way in which different data can differ, is that data could be of different types. For example, I could have a variable x which is of type integer. I could have another variable y, which is of type float, another variable z, which is of type character. These are different types and we understand more or less what it means to be of a different type and we know examples of the different types – character, integer, unsigned or signed and real. We understand that a piece of data which is of type character, can be operated on using certain operations, whereas, a piece of data which is of type real, may have different operations that are relevant to it.

(Refer Slide Time: 24:36)



Data differing in their lifetimes

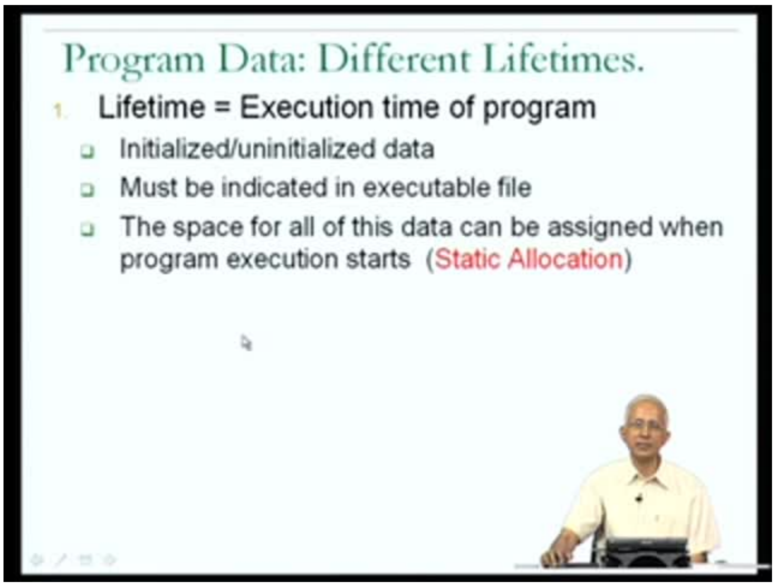
- Lifetime: Interval between time of creation and end of existence
- How long can the lifetime of a datum be?
- We will consider 3 possible lifetimes

For example, I may compute this, I, I may compute the logarithm of a real value, but I, it may not be that meaningful to talk about the logarithm of a character value, right? So, data of different types differ in the kinds of operations that can be done on them. So, these are at least three different ways in which data could be different. The different pieces of data can be different from each other.

But I am going to actually spend a little bit of time talking about another way in which data can differ and that is, that the different pieces of data in your program could differ from each other in terms of their lifetime.

Now, what I mean by lifetime is, very much what you probably understood by the, the word lifetime. The lifetime of a human being is the time between the death of the human being and the birth of the human being. In other words, the time interval between the time of creation and the end of existence. And we will use the same meaning for the word, as far as variables are concerned or as far as data is concerned. So, the lifetime of a piece of data is the interval between the end of its existence and the time of its creation. So, you have to start thinking about this question. What are the different possible lifetimes of data, as far as the program that I have written is concerned? How long can the life time of a piece of data be? I have used the word datum as a singular of the word data, ok.

(Refer Slide Time: 25:32)



Program Data: Different Lifetimes.

1. Lifetime = Execution time of program
   - Initialized/uninitialized data
   - Must be indicated in executable file
   - The space for all of this data can be assigned when program execution starts (Static Allocation)

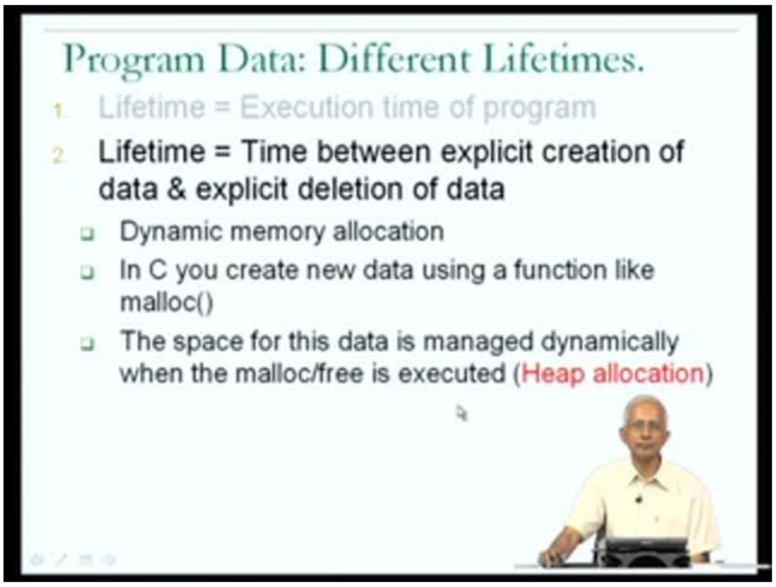
Now, we will consider three possible lifetimes for data. So, data could differ in their different lifetimes. Now, one possibility is that, I could have a piece of data whose lifetime is the complete execution time of the program, for which that data was created. So, in other words, the data came into existence when the program starts executing and the data only ceased to exist when the program finished executing.

So, the lifetime of that piece of data is the complete execution time of the program. Now, the data could be either initialized or uninitialized, right; could have in a, could have a value when the program starts executing or could be undefined at the beginning of the execution of the program, that is a separate issue.

But it is, what is quite clear to us is that, if there is a piece of data whose lifetime is the execution time of the program, then that piece of data must be provided for or mentioned in the executable file, in the a dot out, because, if it is to come into existence when the program starts executing, then it must be present in the program before it starts executing. In other words, in the a dot out file.

Next, right. So, basically we should now understand that, there must be space associated in the computer system, space in the memory of the computer system for the different pieces of data that are used when the program executes and therefore, if I have a piece of data which starts to exist, which comes into existence when the program starts executing and ceases to exist when the program stops executing, then the space for this piece of data must be assigned when the programs starts executing. And, this kind of allocation of space for data is what is called static allocation, since it happens, once when the program starts executing.

(Refer Slide Time: 27:15)



Program Data: Different Lifetimes.

1. Lifetime = Execution time of program
2. Lifetime = Time between explicit creation of data & explicit deletion of data
   - Dynamic memory allocation
   - In C you create new data using a function like malloc()
   - The space for this data is managed dynamically when the malloc/free is executed (Heap allocation)

So, that is the first kind of behavior as far as data is concerned, in terms of lifetime. Some pieces of data have lifetime which is the execution time of the program and space for those pieces of data is statically allocated when the program starts executing. Now, another possibility is, I could have a piece of data for which the lifetime is the time between an explicit creation of the data and the time at which it is explicitly deleted. In other words, if I read the program, I would find that, there is a point in the program at which the program has mentioned ‘here a new piece of data comes into existence’ and somewhere else in the program, a place where the program explicitly says ‘here that piece of data ceases to exist’.

So, that is the situation, where the lifetime of that piece of data is from the explicit point in time during its execution when its creation was requested, till the explicit point in time during execution when it is created, when its existence was deleted. And, for those of you who have done programming in C, with dynamic memory, dynamic variables, you will recognize this as the phenomenon which is called the dynamic memory allocation.

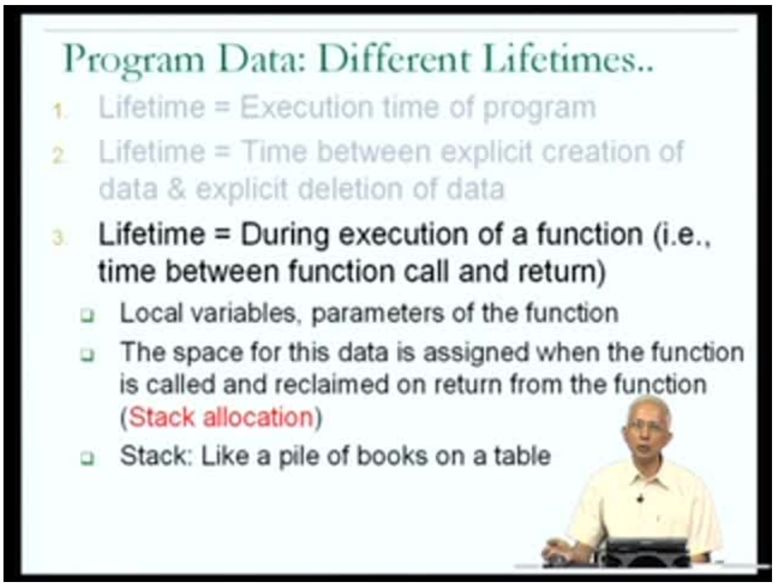
So, the way that you do this in C is, you can create new, new data using a function like malloc. There are other functions as well, but one of the functions you could use is the function called memory alloc or malloc.

So, there would be a, if you look at the program which does dynamic memory allocations, somewhere in the program there would be a call to the function malloc and that is the point in time at which that piece of data, when the program executes, that would be the point in time at which that piece of data comes into existence, right.

Now, the space for data of this kind must be allocated when the program is executing. Space for data of this kind cannot be allocated just when the program starts executing, as was the case with static allocation, because when the program starts executing, one cannot be too sure, how many different times or how many different instances of malloc are going to come, be executed by the program. Therefore, this is the situation where the space for those variables is going to be dynamically created, as and when the function malloc is executed or the the space for those variables could be reclaimed, when a function free is executed.

So, the name which is given for this kind of memory allocation for data is heap allocation. The reason is that, this kind of data is allocated out of run time data structure of the computer system, associated with that program called the heap. So, space, memory space is, is allocated for the dynamically created variable on the heap. So, we have a situation where, some of the variables that we have, some of the data that we have in a program may be statically allocated, because its life time is the complete execution time of the program and some of the other pieces of data that a program deals with might be heap allocated, because they come into existence when explicitly created by malloc and they cease to exist only when explicitly destroyed, may be by a function called free.

(Refer Slide Time: 30:24)



Program Data: Different Lifetimes..

1. Lifetime = Execution time of program
2. Lifetime = Time between explicit creation of data & explicit deletion of data
3. Lifetime = During execution of a function (i.e., time between function call and return)
   - Local variables, parameters of the function
   - The space for this data is assigned when the function is called and reclaimed on return from the function (Stack allocation)
   - Stack: Like a pile of books on a table

Ok, now, there is a third possibility, which I am going to discuss and that is the situation where, the lifetime of a particular piece of data is during the execution of particular function, that is, the time between the function call and the function return.

I am assuming that you are all familiar with what a function is, from your first course on programming. So, you know that, a function is called at a point in the program called the function call. The function is then executed and when the body of the function has been completely executed, control is returned to the point of call, which is that, that is what I mean by the return. So, what we are learning now is that, there could be some pieces of data, which come into existence when the program is executing, when a function is called and those pieces of data cease to exist, when control is returned from that function.

So, very clearly, these are going to be pieces of data, that have something to do with that function. And once again, your experience with programming, you will realize that, one possibility is that, these are what are called the local variables of the function.

In other words, variables which are declared in the function, for use inside the function only. Obviously, those variables need not come into existence until the function is called and have no meaning after the function returns and therefore, their lifetime might well be during the execution of the function. Similarly, the parameters of a function might have a similar property as far as their life time is concerned.

The parameters of a function are the different values which are passed to the function by the caller of the function. So, both local variables and parameters could be allocated in this way.

When the program runs, the local variables, and the space for the local variables and the parameters, could be allocated just before the function is called and they could be reclaimed, the space could be reclaimed when the function returns.

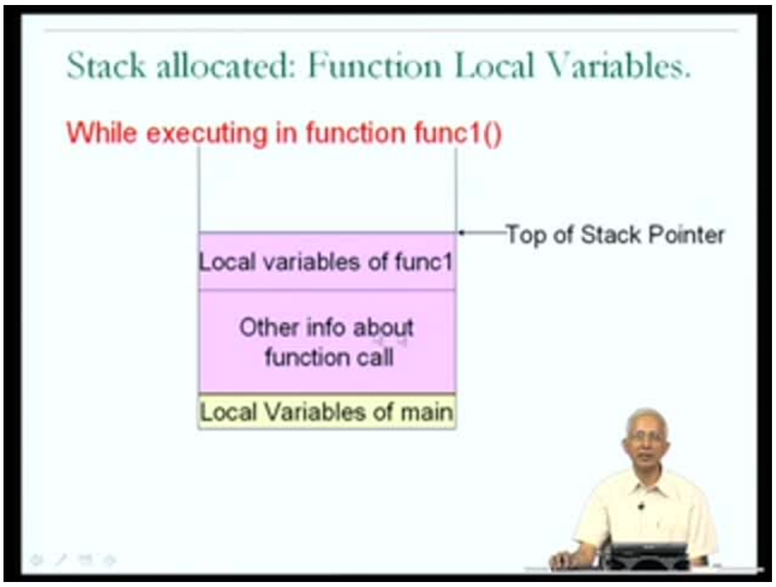
Now, space for this kind of data is assigned on something called the stack and therefore, this kind, allocation for this kind of data is what is known as stack, stack allocation.

Many of you would have heard about the stack before. The stack is a kind of a data structure. It is somewhat like a pile of books on a table. If you have a pile of books on a table, you and you looked at the pile of books as a whole, then, if you want to add a new book on to the pile you can only put it on to the top and if you want to remove a book from the pile you can safely remove only the book from the top and this is what is known as a last in first out or l i f o data structure. And a stack, is an implementation of such operations. Adding something onto the stack is called pushing a new book on to the pile or pushing on to the stack. Removing something from the stack is what is known as pop.

So, the two operations on a stack are push and pop. And, this begins to make a little bit of sense to us, if you have a function that is called and the variable, the space for its variables, local variables and parameters is allocated on a stack, then when the function is called, space is allocated on the stack, on top of the stack, for those local variables and parameters. If that function calls another function, then space for the local variables and parameters of that second function will be allocated on top of those for the first function.

And, the second function is going to return before the first function, therefore, its local variables and parameters can be popped off the stack and subsequently local variables and parameters of the first function can be popped off the stack.

(Refer Slide Time: 34:09)



Stack allocated: Function Local Variables.

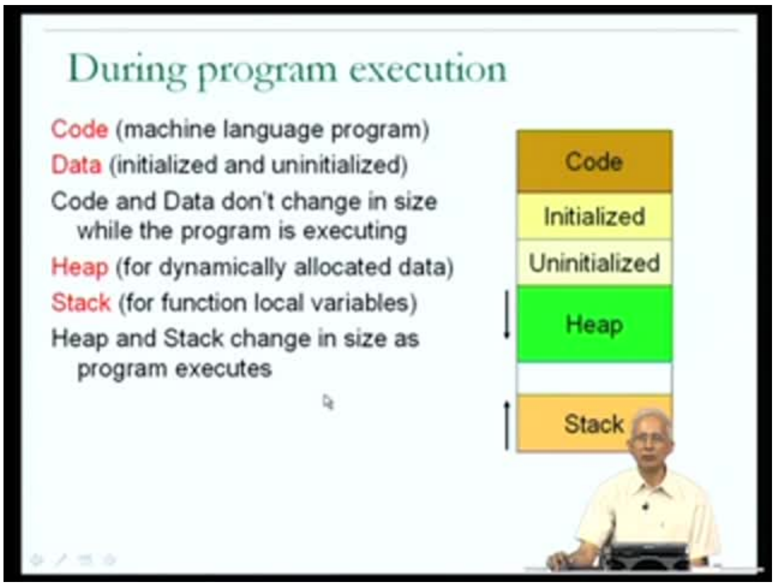
While executing in function func1()

So, this stack allocation seems to make sense for pieces of data, whose lifetime is the lifetime or the execution time of a function, that is, the time between call and return. Let us just look at each of these, a little bit more carefully. Since I just talked about stack allocation, let me start with stack allocated function local variables. So, the picture that you should have in mind is, associated with the execution of your program there is a stack. The stack is going to be used for things like local variables and parameters of the different functions that your program calls.

So, when your program starts executing, there could be some local variables on the stack, associated with the function that your program starts executing with. If you are writing a C program that function is the main function. So, at the bottom of the stack are the local variables of main.

Let us suppose that the main function calls a function called func one. So, when func one is called space will be allocated on the stack for the local variables and parameters of func one. I show those in pink. So, if I looked at the stack during the execution of func one, there will be local variables of func one as well as local variables of main, all available on the stack. What happens when func one returns, in other words, you come to the end of func one, then the information on the stack related to func one can be removed and we are back to the situation as it was in main, before func one was called. In other words, the variables of, local variables of main are once again freely accessible.

(Refer Slide Time: 35:45)



During program execution

Code (machine language program)
Data (initialized and uninitialized)
Code and Data don't change in size while the program is executing
Heap (for dynamically allocated data)
Stack (for function local variables)
Heap and Stack change in size as program executes

Code
Initialized
Uninitialized
Heap
Stack

So, this is a very, a very simple picture to illustrate the idea of stack allocation, which is what is used for function locals and parameters or any kinds of data whose lifetime is the execution of a function. So, the, let us just think a little bit more about what happens during program execution. So, the picture that we should have in mind is the following. We understand that our program, which we first wrote in C has been modified by the compiler into an a dot out, which contains machine language instructions corresponding to the steps of our program and that is called the code of our program.

In addition to this, the a dot out also contains data. It could be data which has initial values or it could be data which did not initially have values, but the values of those pieces of data were assigned by the different statements of my program. So, when my program executes, the code and the data must both be available, in order for the program to proceed, ok.

So, both code and data do not change in size when the program is executing. Remember, that the data is the statically allocated data of my program. As we have just learned, in addition to the code and the data, as my program executes, there are going to be certain pieces of data which I heap allocated and other pieces of data which are stack allocated and the heap allocated data is going to come into existence when it is explicitly created by malloc.

Whereas, stack allocated data, such as a function local variable will come into existence when a function is called. Therefore, we expect that during program execution, we also need to know something about what happens to the heap. This is the space in memory where dynamically allocated data is going to be assigned space, as well as the stack, which is the place in memory where function local variables, parameters etcetera are going to have space associated with them.

If our picture, which I could have in mind is that, the heap and the stack come into existence after the program starts executing, but that they could change in size as the program executes, because, every time there is function called, the stack is going to grow a little bit bigger. More and more items are put onto the stack as more and more functions get called from each other. Similarly, as more and more malloc functions, maollac is called, more and more times, more and more heap data will come into existence. Therefore, both heap and stack could grow and shrink in size, whereas our code and data are going to remain fixed in size.

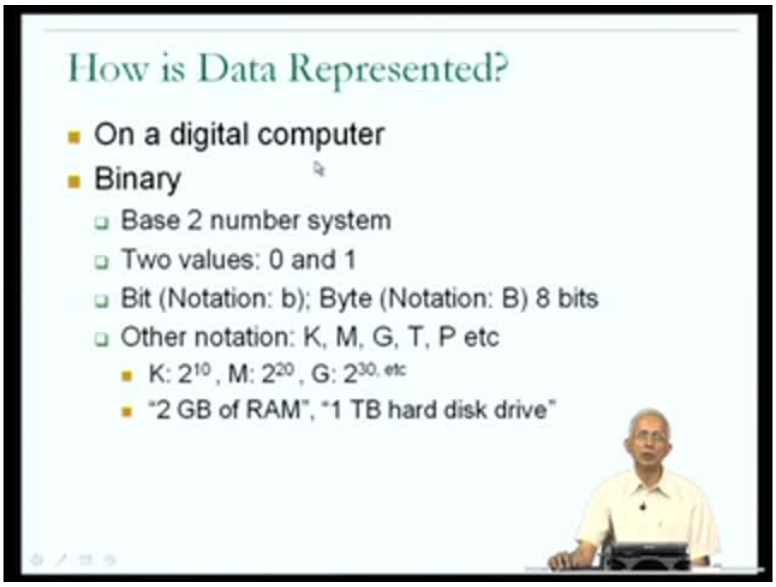
Therefore, these are the different components of my program as it executes and pictorially I might say the code is fixed in size, the initialized and uninitialized un-initialized data are fixed in size. They are all statically allocated. Whereas, the heap could grow and the stack could grow and therefore, there must be some free space in memory for the growth of the stack and the growth of the heap.

So, this is a notation which is often used to describe the different kinds of things in memory, associated with the execution of a program. The instructions of the program, statically allocated data, the dynamically allocated data and the stack allocated data.

So, we will be seeing pictures like this a few times during the remaining lectures of this course. Now, we are getting a slightly better understanding of data, but we do need to get a clear picture about the actual representation of data. So, we will cut to the quick and ask this question.

How is data represented? Now, until now, we were just talking about data in the abstract. We were talking about data. I used examples of integer data and real data and character data and talked about how data could differ in its type or in various other attributes.

(Refer Slide Time: 39:19)


How is Data Represented?

- On a digital computer
- Binary
  - Base 2 number system
  - Two values: 0 and 1
  - Bit (Notation: b); Byte (Notation: B) 8 bits
  - Other notation: K, M, G, T, P etc
    - K: 2^{10}, M: 2^{20}, G: 2^{30}, etc
    - "2 GB of RAM", "1 TB hard disk drive"

But now, we do need to understand more about what our data looks like inside a computer, because that is what we will need to manipulate, when, that is what is manipulated when our program executes.

Ok, now, in general, when we talk about programs today, we are talking about the programs executing on what are called digital computers. In the days gone by, or even today in fact, there are other computers which are not digital. They are called analog computers, but the property of the digital computer is that, for the most part both instructions and data are represented in binary. In other words, in a base 2 number system. This is for ease of implementation. It is much easier to build circuits, to build computers, in which the voltage or current values or charge values are either 0 or 1, right.

Therefore, it is because of the ease of building hardware, assuming that, things are in binary, most digital computers today use the binary or base 2 number system. What do I mean by the base 2 number system? Again this is something you may, you would probably have studied in school. The number system that we are used to dealing with in day to day life is the base 10 number system. The base 10 number system has values from 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 and we use the base 10 number system because, we have 10 fingers, right.

So, the computer uses for the most part, digital computers are built to use a base 2 number system, in which there are only 2 possible values. There is 0 and there is 1.

Right. So, the notation that we will use is, we will refer to any one binary value as a bit. Bit stands for binary digit. Right. Just as in the decimal world there are 10 possible digits, 0 to 9, in the binary world there are 2 possible digits, 0 and 1. And, we refer to any one of the digits as a bit because, digit technically refers to the base 10 number system.

The notations that I will use in this course is that, whenever I am referring to a bit, I will use lowercase b. So, lowercase b will be the short form for bit. Now, since typically, on computer systems there will be a large number of pieces of information stored, there will be a huge number of bits and therefore, we need notations to describe larger collections of bits. The next piece of notation which is often used is referred to as a byte, b y t e.

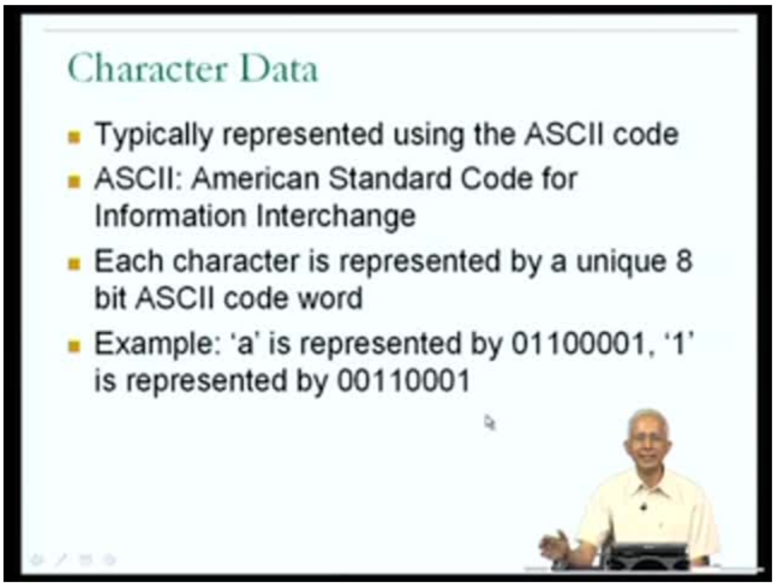
A byte is a collection of 8 bits. Convention which I will use is, whenever I want to refer a byte I will use upper case b. So, lower case b stands for bit, upper case b stands for byte and the byte is 8 bits. There is other notation that you will come across or that you would be familiar with. For example, you would have seen the notation, in addition to the notation lower case b and upper case b, you may of come across notation capital k, capital m, capital g, capital t, capital p etcetera. And these are all just short forms to ease the description of huge collections of bits.

For example, upper case k just stands for 2 to the power 10 in most situations. Therefore, if I use the notation upper case k, followed by b, in other words k b, I am actually referring to 2 to the power of 10 bytes. Similarly, upper case m stands for 2 to the power 20, upper case g stands for 2 to the power 30, upper case t stands for 2 power 40, etcetera, etcetera. Upper case, the just to give you some idea, k stands described as kilo, m as mega, g as giga, etcetera. You can find out the, the words that correspond to t, p and beyond this on your own. I will not, I leave that as a little bit of work for you. So, these are just notations by which we can describe huge collections of bits.

For example, if you are buying a computer, and you want to and you are, you are told that, the computer has 2 GB of RAM, RAM is something to do with the memory of your computer. So, you now understand that, what 2 GB means is, 2 multiplied by G which is 2 to the power of 30 bytes. So, this is 2 multiplied by 2 to the power of 30 bytes, right. That is a typical size of a memory, of a main memory today. 2 GB or 4 GB are sizes of main memory.

Similarly, you might, if you are buying a computer and you, you have a lot of hard disk, you may be told that the hard disk has 1 TB of space in it right, ok.

(Refer Slide Time: 44:08)



Character Data

- Typically represented using the ASCII code
- ASCII: American Standard Code for Information Interchange
- Each character is represented by a unique 8 bit ASCII code word
- Example: 'a' is represented by 01100001, '1' is represented by 00110001

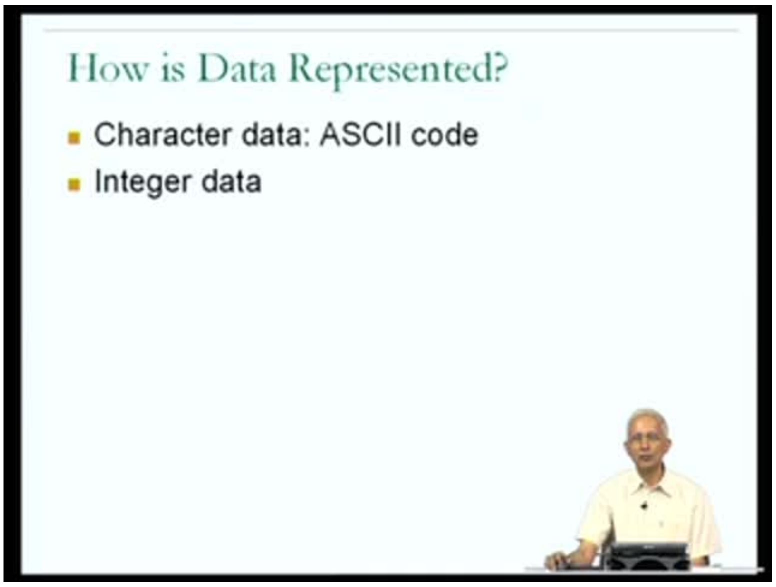
Now, our objective in talking about data at this point, is to understand more about how data is represented and I am going to immediately start by talking about character data. This is an easy thing to understand. You know that character data is used to represent characters, such as a character a or the character b or the character comma and so on. And typically, in digital computers today, character data is represented using something called the A S C I I code or the ASCII code. This is typically pronounced ASCII. ASCII stands for the American Standard Code for Information Interchange. This has been the standard represent, scheme used for representing characters for many years now. The idea here is that, each character is represented by a unique 8 bit code word, a unique sequence of 8 bits.

And you could find the complete ASCII code table in your text book or on the internet, but just to you give you some examples, the, the character lower case a is represented by the bit sequence 01100001. So, whenever a has to be represented, lower case a has to be represented in a digital computer, which is using the ASCII code for representing characters, it does so, using the bit sequence 01100001.

Similarly, the character 1, in other words, not the number 1, but the character 1 is represented by the ASCII code 00110001. Similarly, for any character that you may want to include as data or in a program, there is a corresponding ASCII code.

And therefore, the ASCII code table could be referred to, to find out about the different code words for the different characters are, ok.

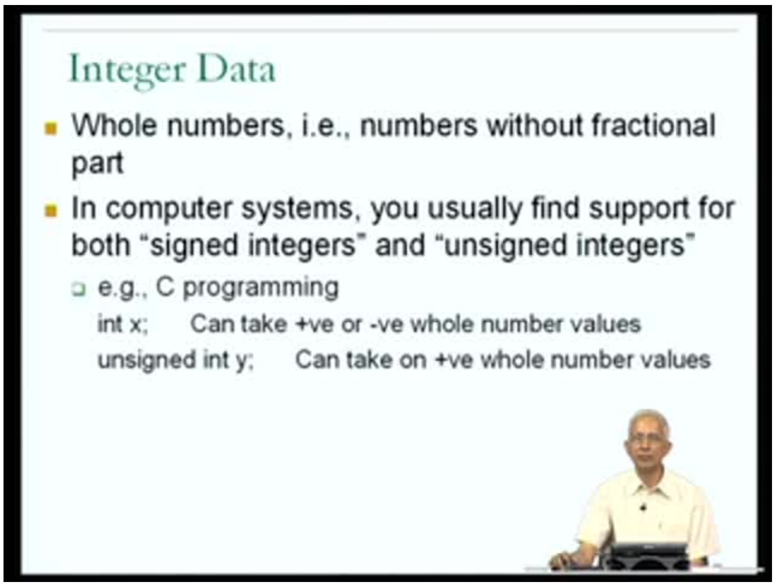
(Refer Slide Time: 45:54)



How is Data Represented?

- Character data: ASCII code
- Integer data

(Refer Slide Time: 46:06)



Integer Data

- Whole numbers, i.e., numbers without fractional part
- In computer systems, you usually find support for both "signed integers" and "unsigned integers"
  - e.g., C programming

```
int x;          Can take +ve or -ve whole number values
unsigned int y;     Can take on +ve whole number values
```

So, as we have seen, the character data is represented using the ASCII code. Now, next, let us move on to integer data. So, we all know what integer data is. Integers are whole numbers and what I mean by a whole number is, a number without a fractional part. So, examples of integers are 6, minus 3, etcetera, 0, etcetera. Now, in computer systems today, you usually find support for both signed integers as well as unsigned integers. And, therefore, you may be familiar with this idea. For example, those of you who have programmed in C may have come across this concept.

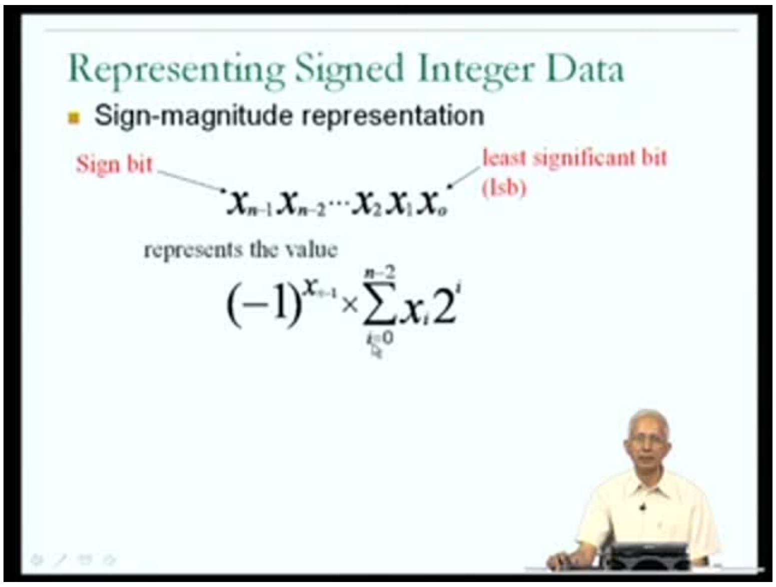
The idea that I could declare a variable called x to be an integer, I could declare a variable called y to be an unsigned integer, as in these two declarations, int x and unsigned int in unsigned int y.

The idea here is that, if I know that x may take on both positive and negative whole number values, I would have to declare it as an int or a signedint whereas, if I know that y never takes on negative values, that I could declare it as an unsigned int y, knowing that it takes on only positive whole number values.

So, this facility is provided in C and in general, we should understand that there are 2 kinds of integer data. They are the pieces of integer data which could be either, which could take on either negative or positive values and the pieces of integer data which, are, have no sign associated with them and by definition, on the positive, ok.

Now, it should be fairly clear that, to represent unsigned integer data, I can just use the binary number system. For example, if I had to represent the value 6, I could represent using 110. And, you would have learnt about the binary system in, even in high school these days, but the question of how to represent signed integers is not as, is not as obvious. Therefore, we probably need to spend a little bit of time talking about how signed integers are represented in a digital computer system. Which is why, we will do that next.

(Refer Slide Time: 48:10)



Representing Signed Integer Data

- Sign-magnitude representation

Sign bit — $x_{n-1}x_{n-2}\cdots x_2x_1x_0$ — least significant bit (lsb)

represents the value

$$(-1)^{x_{n-1}} \times \sum_{i=0}^{n-2} x_i 2^i$$

So, the problem is, I need to represent values which may be either positive or negative. So, one way to do this is, to use something called the sign-magnitude representation. So, this is used in some computer systems, and the idea here is that, if I have an n bit representation for a single integer, right, so, this is an n bit value, There are n bits. The n bits are x n minus 1, x n minus 2, right up to x 0. So, I have actually numbered these with x n minus 1 at the left end, x 0 at the right. That is just a convention, which I am using.

So, what, what does this mean, as far as a value is concerned? You should understand that, each of these xs' is a binary value. In other words, it is either 0 or 1. So, there are n binary values in n 0s or 1s and put together they represent a signed integer. So, the question is, what signed integer would the given sequence x n minus 1 through x 0 represent? And, the answer is x n minus 1, x n minus 2, etcetera upto x 0 represents the value minus 1 to the power x n minus 1 multiplied by the summation i equal to 0 to n minus 2 x sub i multiplied by 2 to the power i, right? So, what we have here is, i is the big position assuming that I number the bits from right 0 up through n minus 1, at extreme left.

So, the situation is that, as you can see from this expression, if the bit at the extreme left, in other words, at x n minus 1 has a value of 1, then this value as, as per this expression is going to be minus 1 to the power 1 which is going to be negative. In other words, the bit at the left determines whether the value that is represented is negative or positive.

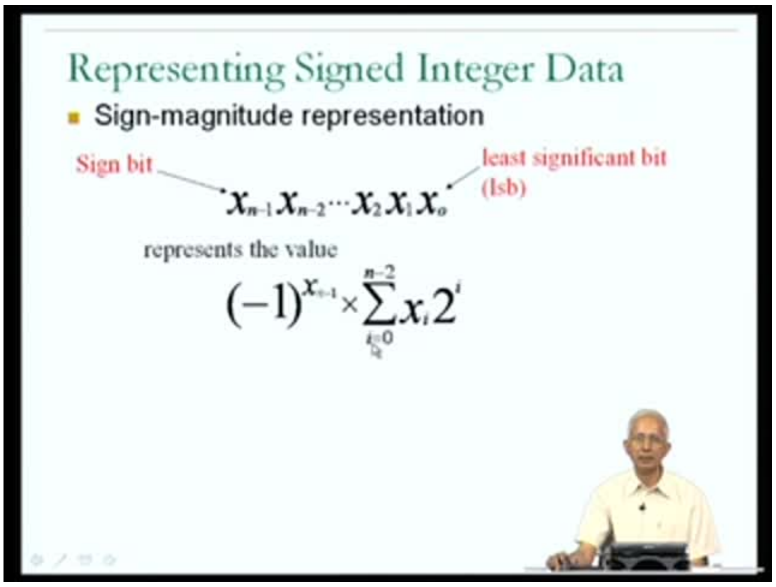
Which is why, the bit at the left is what you would call a sign bit. The sole purpose of that bit is, to distinguish between numbers that are negative and numbers which are positive. If the most significant bit, if the, if the bit at the left x n minus 1 is a 1, then under the sign magnitude representation, the signed integer value, which is being represented, is negative.

On the other hand, if the x n minus 1 is a zero, then the value is positive, because, the, whether I calculate the value is by using minus 1 to the power x n minus 1 and minus 1 to the power 0 is 1. The remaining bits define the binary value or magnitude associated with the signed integer, right.

So, this seems like a good idea, as far as the representation of signed integers is concerned, because, it makes it possible for me to represent not only positive values using x n minus 1 or 0, but also negative values with x n minus 1 equal to 1. And, the remaining n minus 1 bits from x 0 up through x n minus 2, can be used to represent the magnitude of the signed integer value.

Now, in terms of notation, since bit the bit at the right or x 0 carries the least weight in this expression, the bit at the right is typically referred to as the least significant bit or lowercase l s b, and the bit at the right is typically referred to as most significant bit, but in this case, it is the sign bit, since it determines the sign of the resultant value. So, the bit at the left is least significant in that, it contributes the least and is therefore, the least significant as far as the value of this, of this signed integer is concerned.

(Refer Slide Time: 48:10)


Representing Signed Integer Data

- Sign-magnitude representation

Sign bit

$x_{n-1}x_{n-2}\cdots x_2x_1x_0$

least significant bit (lsb)

represents the value

$$(-1)^{x_{n-1}} \times \sum_{i=0}^{n-2} x_i 2^i$$

Whereas, if I consider x n minus 2, it contributes a lot in this summation, because x n minus 2 is multiplied by 2 to the power n minus 2, which is much larger than any of the other powers of 2 in this summation. Therefore, x n minus 2 is a very significant bit. It contributes significantly to the resultant value. And so, this is what is known as the signed, the sign-magnitude representation for signed integer data.

Now, as it happens, the sign-magnitude representation is not used a lot in today's computers for the representation of signed integer data, for reasons that we will see shortly.

There are other representations which are more frequently used and the only reason that we are looking at this is, because there may be some instances where we see the sign-magnitude representation used in other contexts, but hardly in the context of the programs that we are going to write. Ok.

Now, let me just summarize what we have seen today, in today's lecture. Today, we have actually, we have started by trying to understand what a program is. We understand that, a program is a description of algorithms and data structures, which have been put together in order to achieve a specific objective. We have seen something about the representation of programs, that programs could be described in any language that you are interested in, English or your mother tongue, but would ultimately have to be translated into a form that, the computer can execute, which is why a step of translation, typically done by a program called a compiler comes into play, before you can execute a program.

We saw some of the steps involved in the translation of a program, such as by a complier like g c c and that, it is possible that, we may be able to see some of the intermediate files which are, which are created during compilation of a program. We then understood a little bit about how data values could differ from each other, in terms of their life time, type, etcetera.

Finally, we looked a little bit at the representation of data. We understood that, character data is represented using typically, a code called the ASCII code. We then moved our attention to integer data and looked at a particular scheme to represent signed integer values, which is the sign-magnitude representation. This is where we will continue in the next class.