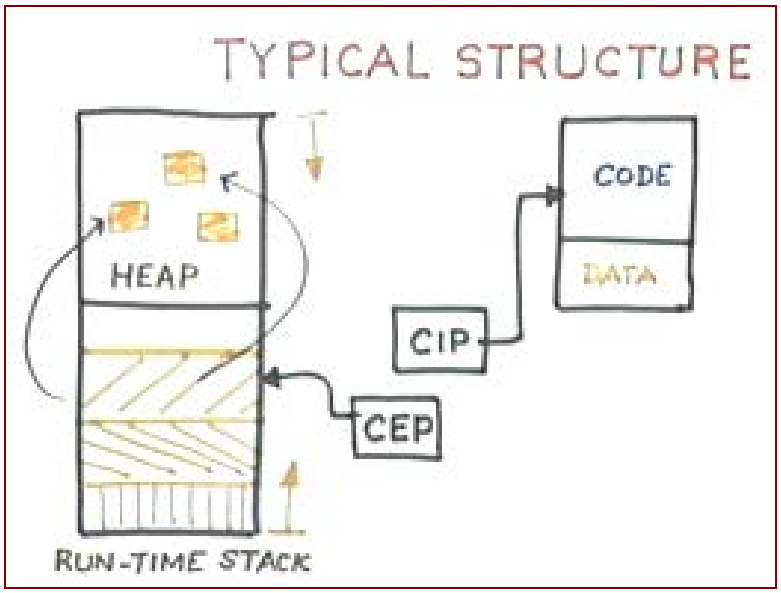
Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi

Lecture no 19
Lecture Title: Pragmatics

Welcome to lecture 19. We will talk about pragmatics. Let us start with a typical runtime structure. Implicit in the assumptions of this is that we are considering some form of block structure or statically scoped languages. In most of these languages a typical runtime environment consists of these entities. Firstly, there is this runtime stack which actually has data created and maintained at runtime and there is a current environment pointer usually stored in some register. This is the base address of the current block that has been entered at execution time. This is a purely runtime structure and you can consider this to be all the memory that you have available for the execution of the program. That memory is divided into two portions; one is known as the stack and the other is known as the heap.

The heap has got nothing to do with the heap data structure that you may have learnt in some data structure course. Here a heap just means a heap of data. That is it. It is like a junkyard. A normal sort of structure is that, given a certain amount of memory that you have available for the execution of a program, you make some arbitrary division which is such that a heap starts from here and grows probably down. The stack starts from one location and grows up. Essentially, if you have to partition a memory for optimal use you try to optimize it. Before compiling the program you really have no idea how much of the allocation can be done at compile time and how much of the allocation has to be done at run time. If the program has some runtime data structures like the typical Pascal new and dispose then you really have absolutely no idea how much space you might have to allocate dynamically at runtime for the program.

[Refer Slide Time: 03:11]



Instead of dividing it up really arbitrarily this division between the heap and the stack is really an artificial division. What usually happens is that you do not want to fix the stack size or the heap size. What you want to fix is only the total memory size and then you let the stack grow during execution from one end and you let the heap grow over another end. So, when you are out of memory a typical error message that you get is that stack overruns heap. If you have a large amount of data such that at any point during the execution of a program if the amount of data is such that it just cannot be accommodated either it could be just statically allocated data which is going to be allocated on the stack or dynamically created data such as lists, pointers, trees etc. There is just no more memory to allocate more data. This stack overruns heap means that there is only a fixed amount of total memory and if in your program you have declared all the data that you require then there are no pointers and the entire space could be used as a runtime stack.

If your program is such that it has very little actual statically allocated data but a large amount of dynamically created data, you are creating lists and trees and destroying them and creating new lists and trees and very little static data then your heap will grow downwards much closer. You would require a smaller stack size and a larger heap size and that is often not determinable at compile time. You just let them grow one towards the other till you have exhausted the memory and then you exit. Otherwise, it is alright. The heap is something that we will talk about later but it is important to know that it exists. It is just necessary to know that all the dynamically created data is stored in the heap. A typical Pascal new or disposed means that you are allocating data. Whenever there is a new allocation to a pointer data that data is going to be allocated in the heap.

There are problems for example; if there is a large amount of allocation and de-allocation going on through let us say ‘new’ and ‘dispose’ then the heap might get totally fragmented and there is absolutely no way of accessing the heap because you do not know where the data and the heap are. When you create a new data you just have to take some allocation from the next available data. Even though you might contiguously allocate pointers initially, as you keep disposing off old structures and creating new structures then your heap space can get thoroughly fragmented. But all accesses to the heap will have to be from the stack because the only fixed addresses that you have come from the stack. The starting point of any list or the starting point of any tree can have a fixed location. Essentially, for dynamic data structures you require some way of statically allocating some place where you can store an address and addresses are of fixed size so you can statically allocate one address or a finite number of addresses that are pre-declared in the program. But you can really allocate them on the stack. All accesses into the heap will be through the stack. So, in the course of execution of some program which deals with a large amount of dynamic data and which continuously keeps allocating fresh space and deleting old space, the heap can get fragmented and when the heap gets fragmented you might have enough space all told to allocate a fresh structure in the heap but that space might be such that it cannot be allocated contiguously.

Supposing you have a large record which requires 10,000 bytes and you are going to allocate it in the heap there might be all told 10,000 places available in the heap but they might all be so fragmented that you cannot change the structure of the record because once a record structure has been fixed at compile time usually you have to allocate that much of contiguous memory also at runtime in order to be able to access the record components. Frequently the heap space is constantly coalesced so as to make it less fragmented. If you open the E MAX editor for example, which is based on LISP and languages like LISP believe much more in

allocating space in the heap than in the stack because all lists that are created are dynamically created. As a result fragmentation is periodically removed by moving data so that they form continuous chunks in the heap and thereby if you are trying to move data you cannot move data. This means that the base addresses in the stack also have to be modified.

In order to prevent fragmentation periodically or concurrently while the program execution is being delayed for some reason may be in I O or that it is just waiting for some response from the user they often have an automatic garbage collector which does a compaction of the heap and gets it less fragmented which means moving a lot of data around and moving a lot of pointers around. You must have learnt the actual algorithms for doing garbage collection in your data structures course. There are these standard reference count methods and the sure way is the mark and sweep algorithm which are used for first marking. You have to know what are valid data items in the heap and what are not data items being used any more.

For each data item you have to know whether there is a pointer in the current environment which actually points to the data item. If there is absolutely no pointer then that portion is garbage and can be collected or some other data can be moved into that. A heap is subject to a lot of variation which is not really visible to the user unless some explicit message comes. Very often the compaction of the heap is visible to the user in some particular way. It usually gives you a response like ‘please wait, garbage collection on’ or some such response and this is something that our E MAX editor on the Himalaya constantly does. It does garbage collection even before you have started the editor but it just periodically runs the garbage collector even if there is no garbage to be collected.

Let us look at the run time stack which is the only thing that is really accessible. That contains a current environment pointer. This current environment pointer is usually stored in some register. It is a base address of the current activation record and your actual code that is generated will reside in some other area of memory and there is a current instruction pointer also stored in a register. This current instruction pointer is really the same as a program counter. This current instruction pointer is usually in a high speed register and it points to the next instruction that is to be executed.

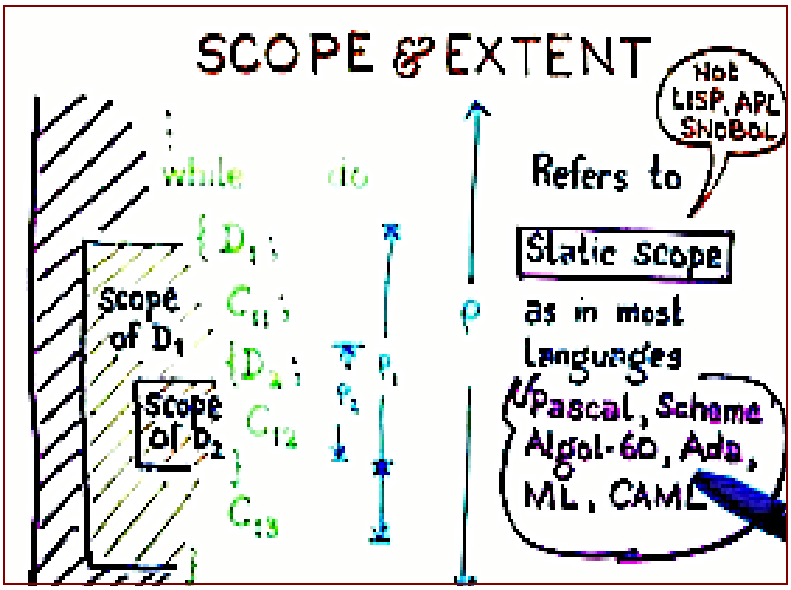
In many cases not all the data might be stored in the stack or the heap. Certain data which can be evaluated at compile time like constants in a particular language are often stored along with the code segment. It means that you can save on execution time by pre-computing the data values and at compile time of course you have no absolute addresses; you have no stack space, you have no idea how the runtime stack is going to look like. So, you allocate those constants immediately below the code segment and this is a standard practice. Most assembly languages have an allocation for data immediately after their code. Similarly, the constants would be stored in some area just immediately after the code for a particular block.

This is a typical structure that most languages with block structure would satisfy. Their implementations would satisfy this sort of constraint. Let us look at some certain issues. Firstly, when you are looking at block structured languages and their implementation we have to distinguish between two things. One is the scope of an identifier and the other is its extent. Take a typical Pascal like language or our ‘while’ language which allows declarations in blocks. You have a huge ‘while’ loop with a block being the body of the ‘while’ loop and this block of course contains a declaration D 1 followed by some code C 11 and this is followed by another unnamed block with declaration D 2 and a code C 12 and at the end of it there is another command C 13. All these commands could be compound commands. If you

look at the declarations in D 2 they have a scope that spans just this block and if you look at the declarations of D 1, it has a scope that spans this block and this is part of some larger program presumably and so, there is something that is global also.

Scope really refers to the program text over which declarations are usable. What we are looking at is the notion of static scope. This example shows you what static scope is about and this static scoping is followed in most languages like Pascal, Scheme, Algol 60, Ada, M L and CAML. The whole idea is that you can look at the program text that means you take a printout of the program and the scope of each identifier. The most standard static scoping structure is that any reference to an identifier refers to a binding created by the innermost enclosing block in which there is a binding occurrence of that identifier.

[Refer Slide Time: 18:44]

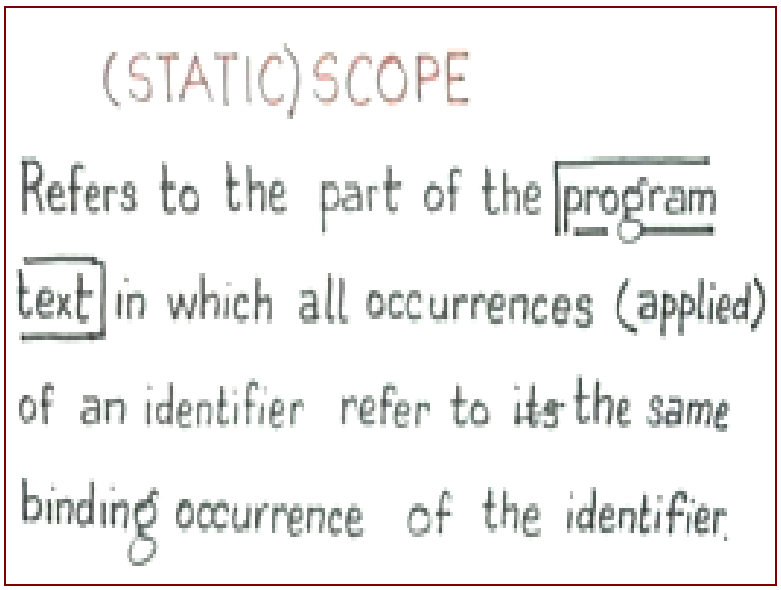


However, it is important that this static scoping is not the mechanism followed in many languages like LISP, A P L and SNOBOL. In fact although LISP is very close to Scheme there is one important difference that in LISP it follows a dynamic scoping as part of the language definition whereas SCHEME is statically scoped. At this point given a printout of a LISP program and given an arbitrary reference to an identifier in that program text, there are no guarantees as to where the binding for that identifier is. It is not possible to infer directly from that text where the binding occurrence for that identifier is. In particular in languages like LISP, A P L and SNOBOL the binding is dependant upon the calling structure. It really depends upon the runtime structure and its execution is not determined in a purely textual manner.

That is in fact an important difference between Scheme and LISP even though they are otherwise almost identical. Static scoping makes program readability easy because you can determine the scopes of all identifiers in the program by using a static scoping method. Bindings are found as close as possible to where they should occur rather than elsewhere whereas in languages like LISP, scoping is really runtime dependent and not text dependent. That is as far as scope is concerned. A related issue is that of extent of an identifier or the lifetime of an identifier. To put it more formally, static scope just means that it refers to that

part of a program text in which all applied occurrences of an identifier refer to the same binding occurrence of the identifier.

[Refer Slide Time: 20:55]



Given an applied occurrence of an identifier you want to know which the binding occurrence is and it refers to that part of the program text in which all applied occurrences of this identifier refer to the same binding. The other concept that we have to look at is the notion of extent or lifetime. Take this typical program that we have got here. Pascal is a typically static scoped language at compile time (whenever we say static or dynamic we mean at compile time or run time). In the case of interpreted languages which are not compiled you would say translation time. The general term would be to say translation time rather than compile time. By scope we are saying that this is a region of the program text as it is actually written by hand.

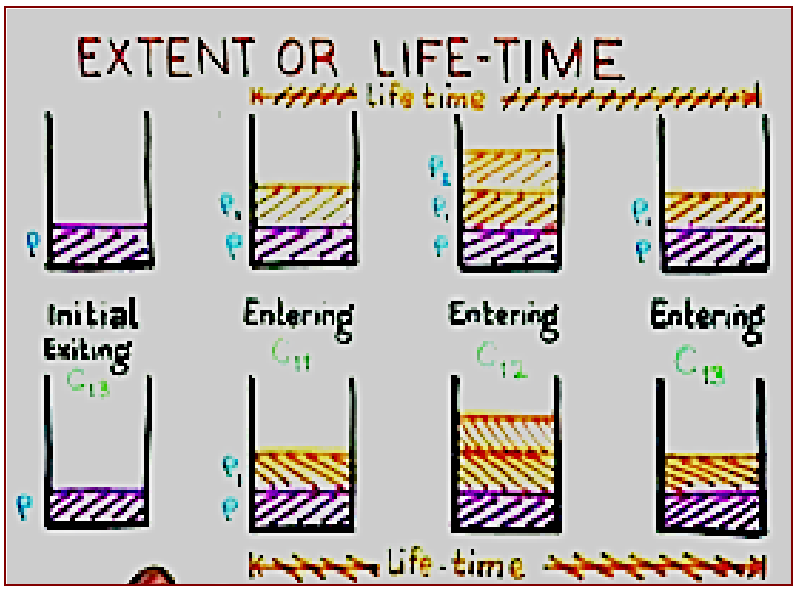
We are talking about this region in the program text that means for the declarations in D1 we are talking of all the text that appears between this left braces and this closing right braces. It is a purely textual matter but the fact that it is purely textual also means that it is only translation time dependent, the binding corresponding to an applied occurrence of an identifier. You cannot determine dynamic scoping at translation time. Given an arbitrary applied occurrence of an identifier it is only at runtime that you can determine what the corresponding binding occurrence of that identifier is. That is what dynamic scoping means.

Whereas static scoping means you can determine at translation time itself for a given applied occurrence of an identifier what its binding occurrence is and if you can do it at translation time then you can read that program text without a terminal and determine exactly the scopes of all identifiers. Whereas in a dynamically scoped language or in a program written in a dynamically scoped language you cannot move away from the runtime environment and determine what the binding occurrences of arbitrary applied occurrences of identifiers are. Let us look at this program. I have chosen a ‘while’ loop for a very specific reason because that illustrates the difference between scope and a related concept which is extent. At least in statically scoped languages scope refers to something that is compile-time determinable or

translational-time determinable whereas its extent actually refers to the binding occurrence that is applicable during runtime.

If you look at the typical runtime stack for the execution of this program then what happens is that you have an initial global environment on the runtime stack and as you get into the while loop you are entering a new block. Now a new environment row1 is created by the declaration D 1 which means all the variables that are declared in D 1 have a lifetime starting now at this point. The moment this new environment has been created those variables which have been declared in D 1 are born at this point. You enter the next nested block, create a new environment row 2 and as long as this environment row 1 is present those variables in the declaration D 1 continue to have their extent at this point and at this point the variables in the declaration D 2 are born.

[Refer Slide Time: 29:45]

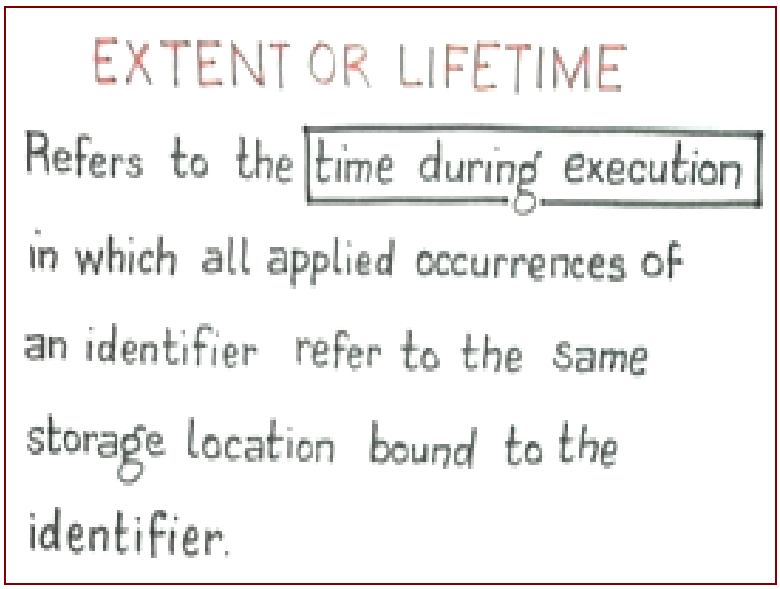


After you have exited C 1 2 this environment row 2 is lost and therefore all the variables that were declared in the declaration D 2 are lost. The variables in D 2 have a lifetime that spans only from the moment control enters the block D 2; C 1 2 and you execute C 1 2. Throughout the execution of C 1 2 these variables are alive so to speak and they die as soon as you exit C 1 2. When you are entering C 1 3, which is back to the original environment consisting of just this, these variables are no longer alive and so they are dead.

After you have exited C 1 3 since it is a wild loop you could get back into the original environment. Now those variables that were declared in D 1 have a lifetime spanning from this to this. When I am talking about variables I am talking about identifiers which are bound to locations. The locations of the variables in D 1 are available only up to this point entering the body of the while loop and exiting that last portion C 1 3. What happens is that when you get back into the body of the while loop again the environment row 1 is created and so these variables in D 1 are again born and so through the next iteration they have this lifetime but these are not the same as this. The extent is a runtime concept. Remember that our semantics always says that you should make a new location available to you.

Theoretically at runtime these storage locations need not necessarily be the same. It is just pragmatic that you might be reusing the same storage location otherwise think of it as follows. If the variables in D 1 had a lifetime which spanned up to here then the original values that the variables contained at the end of this execution should also be available here but they are not. They are freshly reinitialized and they have the same names. They have several lifetimes so theoretically speaking this is one lifetime of these variables, this is another lifetime of these variables and there is no relationship between these two lifetimes except that they come from a common scope. Except for the fact that they have the same names there is really no connection between these two lifetimes.

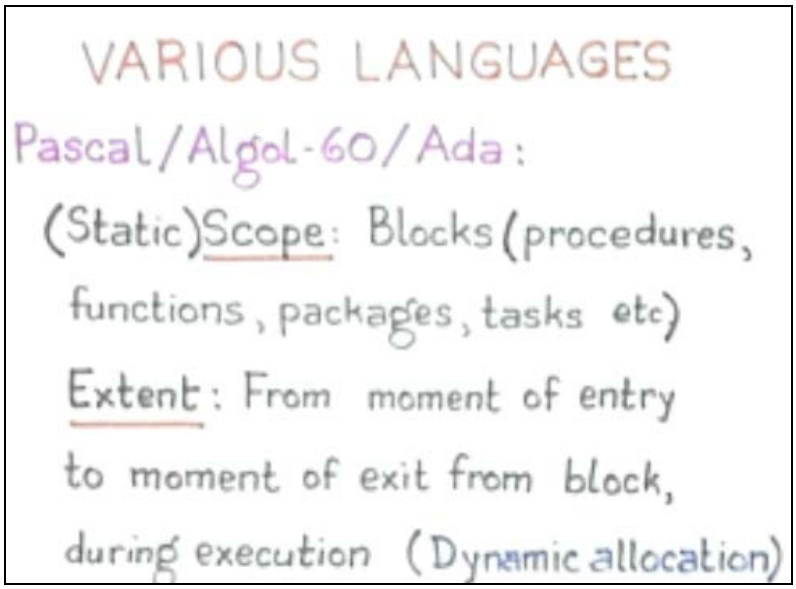
[Refer Slide Time: 30:44]



When we are talking about an extent or a lifetime it really refers to the time during execution in which all applied occurrences of an identifier refer to the same storage location that is bound to the identifier and in different lifetimes you have different storage locations at least theoretically. If I had disposable memory as soon as I have used it once I just throw it away and get new memory. It is just that we cannot afford disposable memory but essentially a lifetime really refers to that span in the execution of that user's program when an identifier-location binding is created and that identifier-location binding is destroyed. The identifier-binding location might be created again but then that is supposed to be a different location because there is no continuity between those two lifetimes as it is supposed to be a brand new location.

The extent or a lifetime refers only to the time during execution in which these applied occurrences of an identifier refer to the same storage location binding to that identifier. It is important to distinguish between these two because for example, the global variables of a Pascal program have a lifetime that spans the entire program execution. The notion of blocks that I have defined in my language or the notion of unnamed blocks which have declarations and commands really comes from the Algol 60 unnamed blocks. In Pascal for example, the notion of a block is just a procedure or a function. In Ada of course they carry things further. You might have packages, you might have tasks and packages are really modules. Tasks are for concurrent parallel execution of programs. The basic notion of a block in these languages determines the notion of scope in all statically scoped languages like this.

[Refer Slide Time: 33:55]

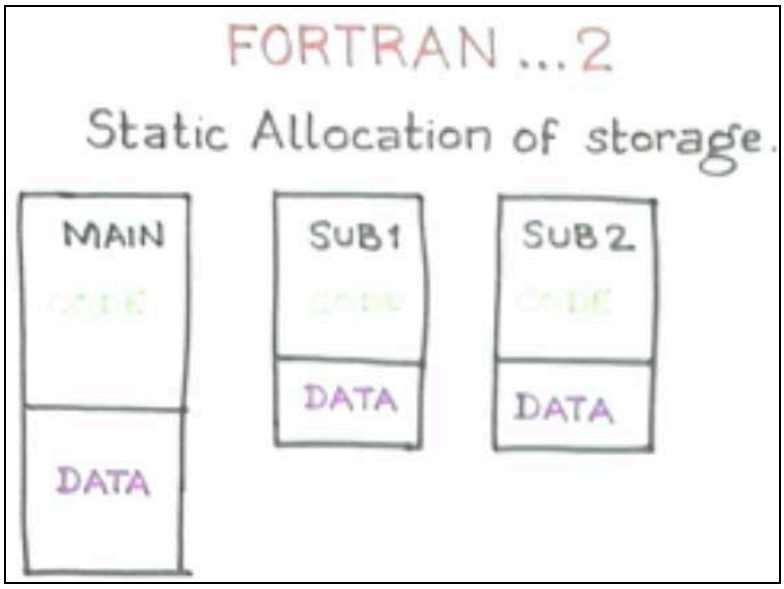


The scope is basically a block and what the notion of the block is might vary from language to language. In the case of Algol 60 you have both the notion of unnamed blocks such as procedures and functions; in the case of Ada you have procedures, functions, packages, tasks and unnamed blocks and in the case of PASCAL you have just procedures and functions. The notion of extent just comes from the fact that at runtime you are allocating storage to the identifiers or to the variables in a block. So, the extent or the lifetime is just from the moment of entry into a block to the exit from the block during execution time. If you enter and exit the block several times then they are all different lifetimes and no two lifetimes have any relationship with each other. For the same identifier no two lifetimes have any relationship and that is because of your dynamic allocation.

There are lots of languages which actually do not distinguish between scope and extent too much. For example; the notion of scope in languages like FORTRAN and COBOL is just any notion of a block which includes functions and subroutines. Subroutines are like Pascal procedures. They change the state and the extent is such that it is the duration of the entire execution of the program. Regardless of the scope in which an identifier is being declared its extent lasts the entire program execution.

I am not talking about just global variables; I am talking of even local variables. They have a lifetime from the start of execution of the entire program to the end of its execution regardless of whether you have entered a block or not. Every variable has a lifetime that spans the entire program execution time. This is due to the fact that in FORTRAN and COBOL actually allow a static allocation of storage and it in fact follows a different runtime model in the sense that you have data associated with every block and all the blocks are allocated separately.

[Refer Slide Time: 36:11]



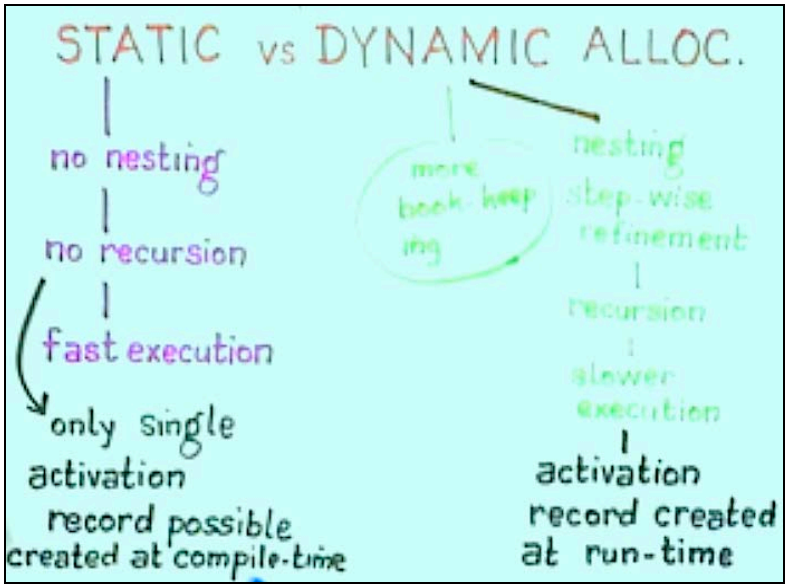
There are of course common areas. A simple FORTRAN program, which consists of functions and subroutines, would be such that all the data that has been declared in that block is also allocated along with the code for that block. You can enter a subroutine, exit from that, enter it again and the previous values will be preserved. They have an extent which is different from the scope. However, without entering a subroutine you cannot access its local variables. The scope rules disallow the accessing of variables that belong to a different scope. Those variables carry their value throughout the execution of the program regardless of whether you are inside that block or not.

Actually in some ways this is a very simplistic method of doing storage allocation. Based on your compilation of the code you also just directly allocate the storage that you require and the total amount of memory that is available to the code and the runtime structure is just divided up in terms of the blocks. This block requires a lot of memory and you need to allocate it that much. For each block you just allocate as much memory as it requires for both its code segment and its data segment at compile time. The result of this static allocation of storage means that you cannot have any dynamic creation of data. So, there is no notion of a heap space in an implementation of FORTRAN or COBOL. There is absolutely no way you can do pointer arithmetic except through arrays but then you are statically allocating all the heap space that you require within the data segment itself.

The only way you can build pointers and lists is by allocating a huge block of storage. I am going to use only that area which is allocated to that data. You have to pre-declare how much area you are likely to use in the heap. Most of us who are used to programming dynamic data structures in languages like Pascal really have no idea how much the heap can grow. But if you are going to do that in FORTRAN or COBOL you need to know how much memory you require and then declare it. There is no alternative to that. The maximum amount of memory that you will require at any instant during program execution would have to be declared in the program itself so that it can be allocated at compile time. At execution time there is no question of allocating that.

When you have an extent that spans the entire program execution you are making suboptimal use of the memory. While you are in the main program the scope rules do not allow you to access the data of a subroutine but all that data are still there. So, you are making suboptimal use of memory but the result is that its FORTRAN programs execute among the fastest in the world because this extra overhead of dynamically allocating memory is absent.

[Refer Slide Time: 41:05]



The result of all this static allocation is also that you cannot nest structures and blocks in FORTRAN or COBOL. All blocks will be kept separate and since it is statically allocated you cannot have any recursion because when we are talking about recursive invocation you are dynamically allocating more and more activation blocks and records for that recursive invocation and you do not know before hand how many activations are going to be required for a condition to be satisfied and the recursion to terminate.

You cannot have any recursion and you cannot have any nesting but the executions are very fast. You have to determine the amount of memory you require initially but you are guaranteed very fast production style executions. On the other hand in the case of these statically scoped languages you are creating a fresh activation record at each time you enter a block and you are de-allocating that much of storage each time you exit a block. If you are going to do that there are the overheads of actually doing this allocation because allocation has to be done meaning you have to do a lot of book keeping. At any instant of the program execution you are using exactly as much memory as you really require but you are doing a lot more hard work in doing the allocation, initializing, storing and consequently you are going to get slower executions. But it allows for a nesting of blocks; it allows for a natural development of the program in a stepwise refinement fashion, it allows you to make clear what is going to be local to a block and it allows recursion which is a very powerful mechanism for most people who cannot analyze recursion enough to convert it into iteration.

A large number of mathematical problems are recursively defined so it is very simple to just encode that recursion in your program. Whereas if you were to write a FORTRAN program you would have to completely analyze that recursion and transform it into a pure form of iteration and also calculate exactly how much storage you require before you actually write

the program. It is because of these powerful methods of program development which are available with a dynamic allocation your executions will be slower. The methods imply extra overhead of having to do an allocation of memory each time a block is entered.

In the original definition of the Algol 60 language, its designers came up with a dynamic allocation strategy and the whole runtime stack structure and came up with Algol 60. But because of the fact that the extents were finite and did not last the entire program execution it was felt that if you are going to allocate and de-allocate every time at entry and exit you cannot program something like a random number generator which might be used by your program but is not a part of it. You are just taking a random number from a random number generator and you are using it. So you do not want any part of that code of that random number generator to be mixed up in your program especially if everybody is going to use that random number generator for stimulation purposes.

I do not want to declare global variables especially for example the seed of the random number generator after having generated some random numbers to generate the next. These random number generators are all pseudo random number generators. If you keep allocating and de-allocating each time at entry you will generate the same sequence of random numbers. The random number sequence at different lifetimes is going to be the same and you do not want that. If you wanted a true simulation behavior you would like the last random number that you generated in the last lifetime to be preserved but you do not want that to be a part of your global environment because you are using only the random number generator and yet it has to be preserved.

They allowed certain variables to be declared locally. Remember that the random number generator might be written by somebody else and you are just using that. In that code certain variables are declared as being statically allocated. So, they thought that would solve the problem. It was an adhoc solution and in fact this adhoc solution has been institutionalized in C. There are these declarations called static which explicitly tell you that these things have to be allocated not at runtime but at compile time itself and even though they have a scope which is local they should have an extent which lasts the entire life of the program but most of the other languages have similar means for looking at scope and extents but these are the board classifications.

There are parts of FORTRAN which allow for sharing of memory. In those days memory was very scarce. If you had only 32 K for the entire computer system with 200 users, you had to allow sharing for memory and each user got very little memory. You had to allow sharing in the sense that between different blocks you allow different names to be allocated the same area of memory and that area anyway had an extent that lasted the entire program's execution which means that there was an aliasing problem.

There were different identifiers referring to the same location and if an identifier was not initialized and FORTRAN had a default initialization for integers and if it is not explicitly initialized then it is 0. But if it has an extent that spans the entire program execution then if you had default initialization your program could often work wrong because as a programmer you assume that it would take a default initialization.

Since it has an extent that lasts the entire program execution, the initialization was in fact the last exit from the block and so, it led to a lot of insecurities in the language. We will continue with pragmatics in the next lecture.