Principles of Programming Languages

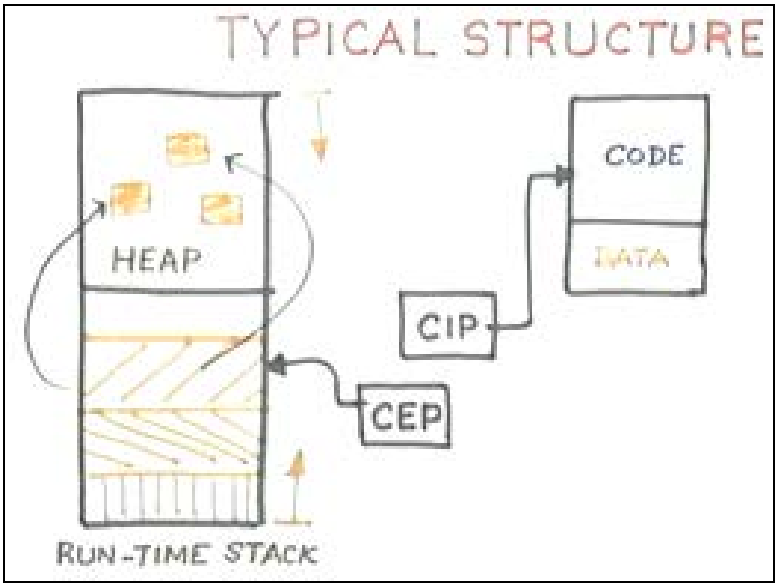
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi

Lecture no 20
Lecture Title: Data

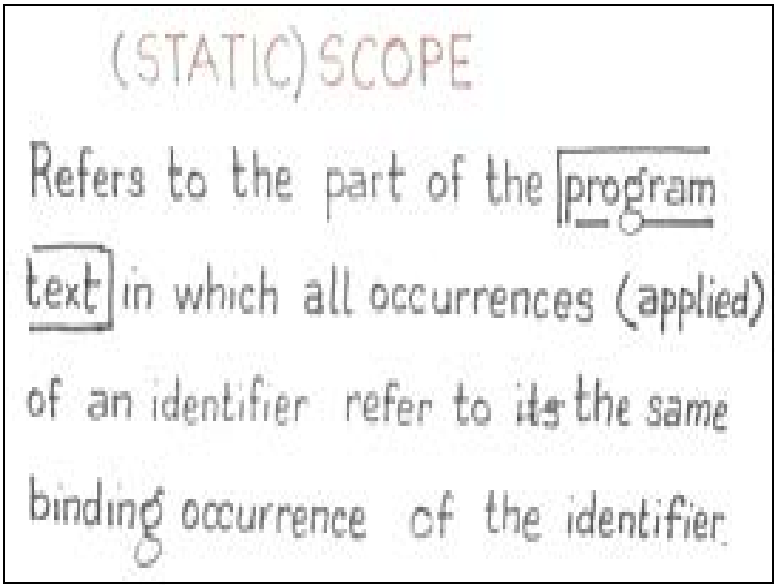
Welcome to lecture 20. Last time we dealt with pragmatic issues such as the large scale structure of the runtime system. Those structures are essentially basic to the rest of the course. Almost all languages would follow one or the other structure with mixtures. Let us quickly summarize all that and then we will get on to data.

A typical structure of the runtime environment of a block structured language is like this. There is a runtime stack with a heap where usually the heap and the stack grow towards each other; there is a code and some special kinds of data are stored along with the code and there is a current instruction pointer usually stored in a high speed register and a current environment pointer usually stored in another high speed register.

[Refer Slide Time: 01:11]

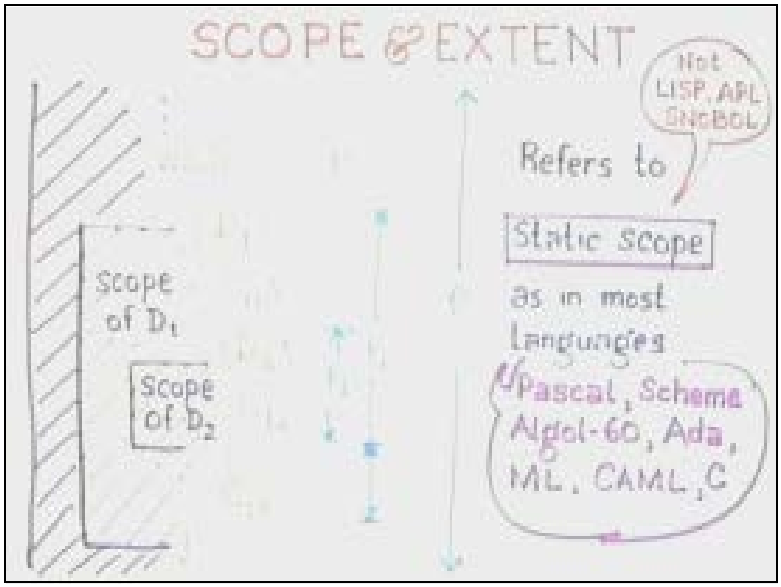


[Refer Slide Time: 01:35]

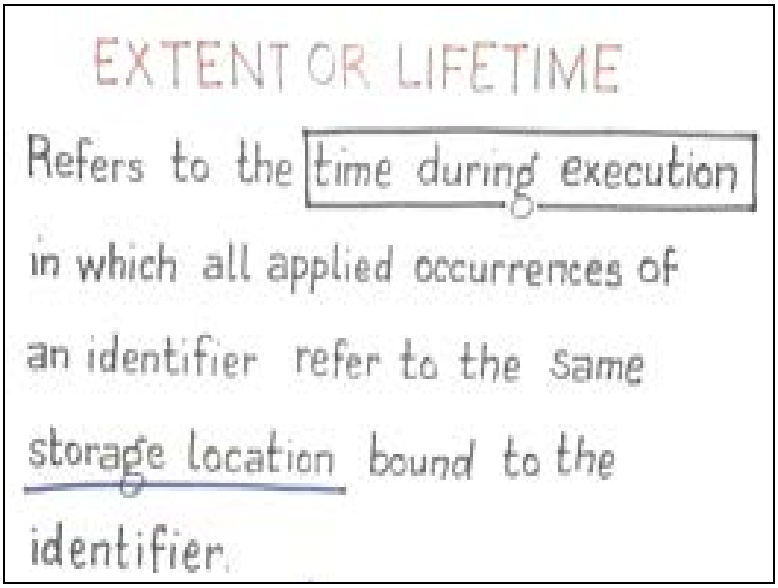


I also spoke about scope. At least static scope refers to that part of the program text in which all occurrences of an identifier refer to the same binding occurrence of the identifier. It is a region of program text and it was illustrated with this. Such scope rules are present in most of the modern block structured languages. However, there are languages like LISP, A P L and SNOBOL which have something known as dynamic scope. So, it is not possible to specify scope from the program text.

[Refer Slide Time: 01:55]

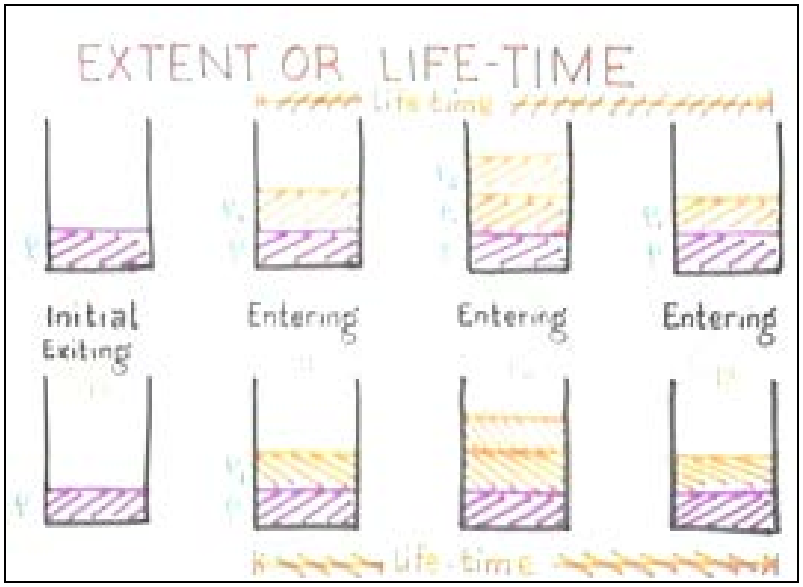


[Refer Slide Time: 02:25]

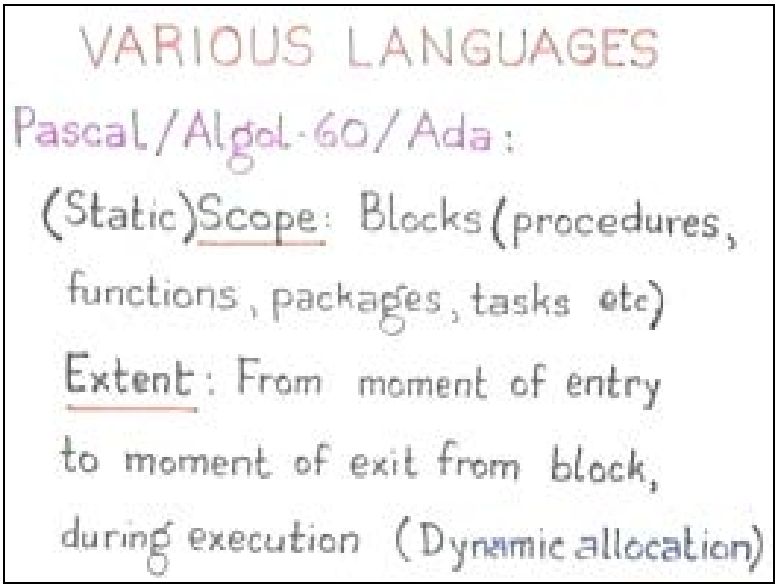


I also spoke of extent. The extent or lifetime of an identifier really refers to the time during execution in which all applied occurrences of an identifier refer to the same storage location that is bound to it. In a statically scoped block structured language as in the example shown before, you have lifetimes. For the same identifier in an inner block you have different lifetimes and the different lifetimes are not necessarily related.

[Refer Slide Time: 02:55]

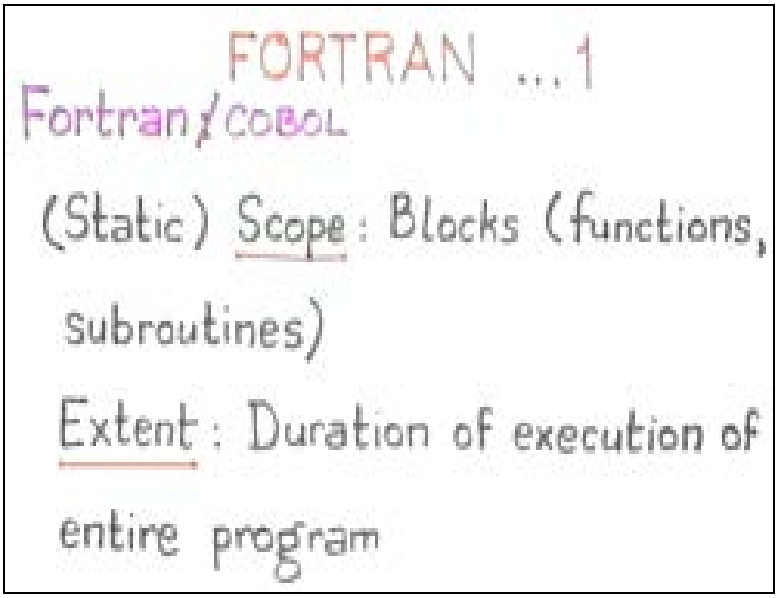


[Refer Slide Time: 03:11]



In most languages in the Algol 60 family like Pascal, Ada etc the notion of blocks is whatever we have done which is that we have taken the minimal subset that should define a block. Other languages have certain policies such as named procedures or functions, packages or tasks and even unnamed blocks and extent in such languages with a typical runtime structure is really from the moment of entry to the block to the moment of exit from the block because the allocation is dynamic at runtime.

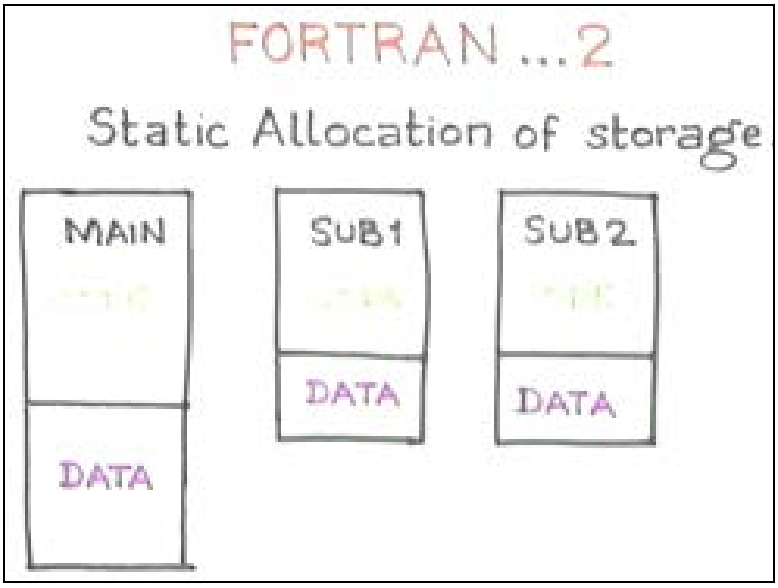
[Refer Slide Time: 03:48]



Then we saw that there are languages like FORTRAN and COBOL where the scope is again delimited by the program text and therefore they are statically scoped. However, the extent is the

duration of the execution of the entire program and that is because languages like FORTRAN and COBOL perform a static allocation of storage. At compile time every data item that occurs in a piece of code is actually allocated area. Besides this there is a common area for sharing locations between various program units which are known to have disjoint lifetimes but then it really depends upon the programmer to infer that they have disjoint lifetimes and so they can share a common area of storage.

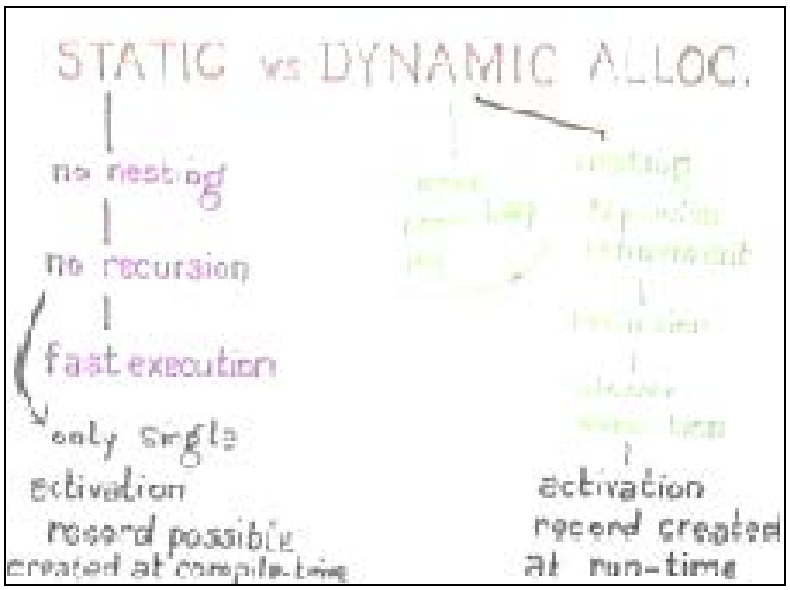
[Refer Slide Time: 05:11]



It is safe only when you can actually show that whatever is common in different blocks have disjoint lifetimes. Unless you can show that there is likely to be confusion there is likely to be an error also in a program. You have a static allocation of storage in which most of the data is allocated immediately below the code along with the code. Therefore the normal scope rules apply, if within any program unit all references to the data are either to the common area, which is separate or to the locally declared data to the local environment.

The advantages and disadvantages of doing pure static or pure dynamic allocation are that with a static allocation you cannot have nestings of blocks and you cannot have more than one activation record for the same unit. Therefore you cannot have recursion. However, since all the allocation is done at compile time the actual execution of programs is very fast. That is one good reason why scientific programs are still written in FORTRAN. Another good reason of course is that there is already a large library of FORTRAN routines that are available which are thoroughly tested and tried. So, the confidence level in those FORTRAN routines is very high.

[Refer Slide Time: 06:14]

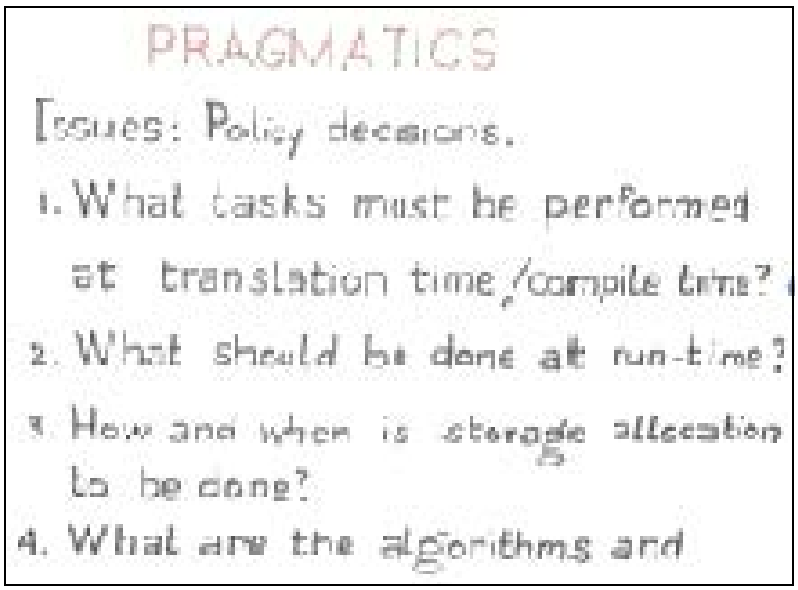


If you want to actually build more scientific routines it is a good idea to use the existing libraries but otherwise this fast execution is a very good reason why most scientific computation, which is highly computation intensive and with very little I O, is still done in FORTRAN. In structured versions of FORTRAN the original version of FORTRAN was FORTRAN 4 which is not structured but now you have these structured versions of FORTRAN the latest of which, I think is FORTRAN 90 where they also have highly optimized compilers so that the execution is really fast.

You can do a certain amount of stepwise refinement with these new structured statements of FORTRAN 77. FORTRAN 90 is really a parallelized version of FORTRAN 77 but the executions are fast and general purpose scientific routines are going to be developed once and are going to be run several millions of times. It is necessary to have fast execution even if compilation is slow but surprisingly FORTRAN compilers also compile very fast. But that is because the language is otherwise very close to the machine language and assembly language. In the case of dynamic allocation it actually allows you to do a systematic development of programs by allowing nesting so, you can actually apply a top down or a stepwise refinement method to inter-mix specifications and program code and derive the final code as a refinement of the intermediate steps of your program.

They allow recursion which is a very powerful specification facility and if it is directly implement-able then it simplifies a lot of matters because transforming recursion into iteration is not a particularly easy task. For large programs you would like an automatic transformation mechanism but that is very hard and most of the time the only transformations that work are very trivial sort of transformations. As a result there is more book-keeping at runtime in these dynamically allocated languages, in most block structured languages and because of the allocation and de-allocation that you have to do, the executions are also likely to be slow unless of course you optimize the compiler.

[Refer Slide Time: 09:44]

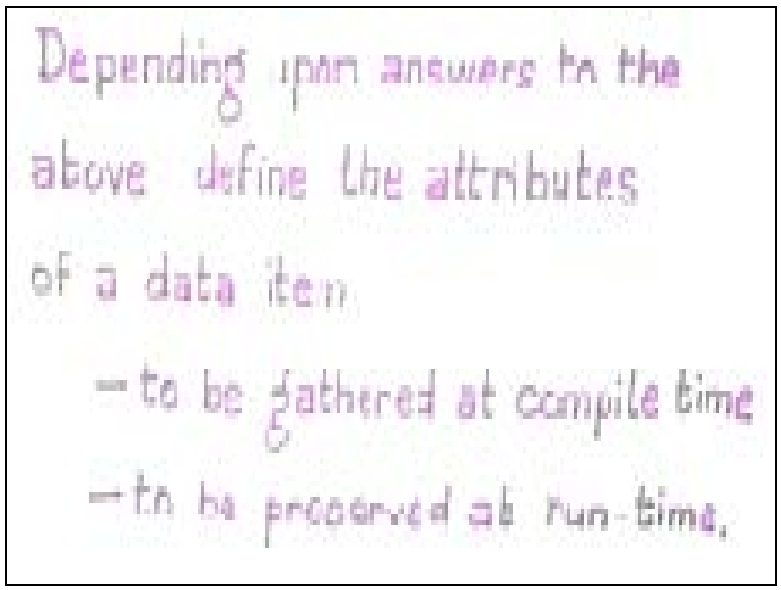


The main pragmatic issues that we have to deal with in a subject like programming languages are not at the level of very deep or complex algorithms. Most of the decisions are of a policy nature. What kind of allocation strategies should we have? The actual algorithms are never very complex. The main problem is integrating all the policy decisions into a consistent framework and making something actually work which is very hard. Mainly the questions are of policy decisions and algorithms are few and far between in the traditional sense of very deep and fancy algorithms. It is really a matter of optimizing resources, optimizing trade offs between whether you want to do something at compile time; you want to do something at run time, you want to do something dynamically at run time etc.

The question is what tasks should be performed at translation or compile time? What should be done at runtime or what should be done immediately after compile time but just before runtime? Then how and when is storage allocation to be done? What kinds of data are required to have storage allocation done and when? How can you optimize memory usage not instantaneously but may be over an average? How can you optimize instantaneously when a program that requires a large amount of data at an instantaneous moment in the execution, runs out of memory and never works? So, it is a matter of taking decisions rather than actually designing fancy algorithms.

Then finally of course what are the algorithms and what are the kinds of data structures you have to use? Once you have decided on certain policies then data structures become more or less clear like the runtime stack or a heap. What is the relationship between the code segment and data? What is the structure of the underlying virtual machine if you are designing a compiler like P L 0? If that is the case then what should be the data which contains the code and how should that be executed? They are really not very central issues. So, depending upon answers to these questions you can define attributes of a data item and the kind of information.

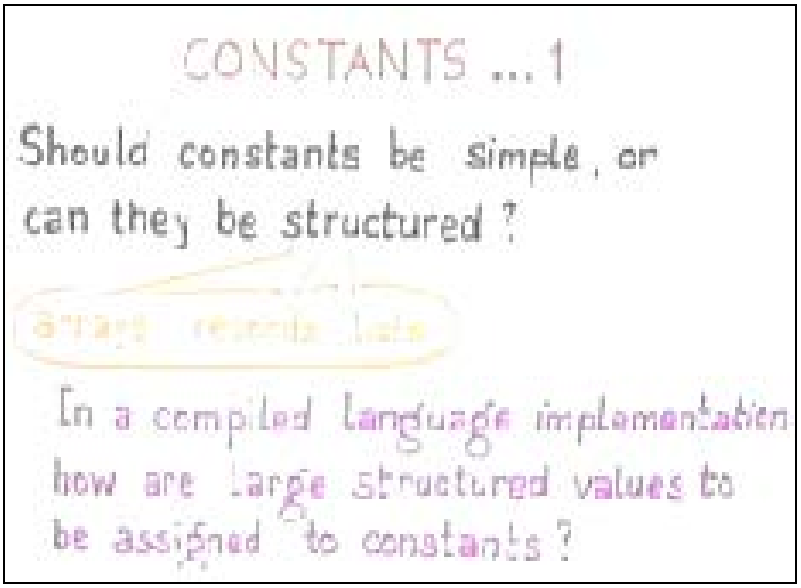
[Refer Slide Time: 12:44]



The answers to these questions will also determine what kinds of information you should gather at compile time, what kinds of information have to be preserved at run time in order to introduce checks or in order to do storage allocation itself at runtime. Let us look at the primitive forms of data that we already have in our language. Firstly, let us consider constants. We will for the present just assume an imperative language very much like the 'while' language that we have got with unnamed blocks and the local declarations could be either constants or variables. We will talk about complex data without being too specific about it.

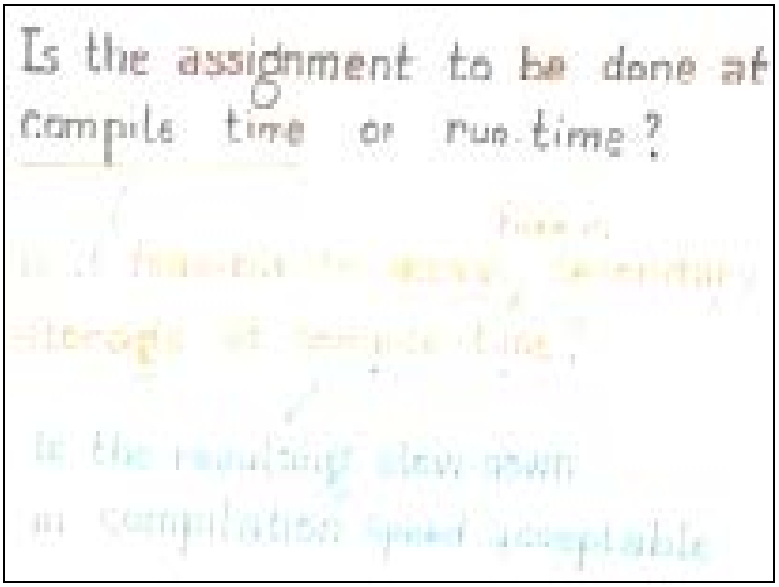
You will find that the decisions that we require even for constants are quite a few. It is not a question of deciding that we are going to have constants in this language and then we are going to set about implementing them according to this semantics. Very often you have to look into the future before deciding to allow constants of a particular type after allowing constants of another type. For example; the first question is should constants be simple or can they be structured? By structured constants I mean where you can have arrays, records and lists but if you allow for structured constants then you should also allow for structures that are very large. If structures are going to be very large then in a compiled language implementation how are you gong to assign these large structures to constants?

[Refer Slide Time: 14:55]



Assuming that we are going to have constants the first question really is: are these constants going to be given values at compile time or at translation time or runtime? Now based on that the question is that if you have large structured constants how are you going to assign values? Supposing you decide to do a compile time assignment of values to the constants then how are you going to do it for large constants?

[Refer Slide Time: 16:00]



If you are going to do it at compile time and if you have large constants then are you going to access files in the directory at compile time? Are you going to do compiling and access various files? You might have to access enough files in order to link and load the program. You will

have sufficient hassle just trying to compile the program. Now your compile time especially in a language which is suitable for development is going to be tremendously slow if you decide to take the policy that constants have to be allocated at compile time and you are also allowed large structured constants. Then you are going to also take in the value somehow from files.

Now if the constants are structured and large and if you allow for file input then a file typically is just a sequence which means a file has to be formatted in a certain way. So, you may have to parse the file before you actually perform the allocation to different components of your structured constant. If you just have an array maybe it is simple but if it is an array of records of an array of records, there has to be syntax in the file itself which specifies what components go where. This means that you have a certain grammar for the data in the file which means that you have to parse that input from the file before you actually do the assignment. The question is that are you willing to do that?

When you take a policy that constants are going to be assigned at compile time and large structured constants are going to be allowed it means that you have to be prepared for these constraints. Then is the resulting slow down in the compilation speed or translation speed really acceptable (if it is a compiled language in which you assume that the executable versions of the program are going to be run much more than the only repeated compilations that are required at development time of the program that later become part of the library)? Then may be a resulting slow down in compilation speed is acceptable. But in a language used to learn programming the number of runs are going to be much less than the number of compilations. So, what you require really are fast compilations and these decisions may not be acceptable in such a language.

These kinds of decisions have far reaching consequences and the actual algorithms themselves are not as important as the decisions that you take in language design in the implementation design. The next question of large structured constants automatically raised is; when should a constant be really assigned its value? Suppose you assign it at runtime and you can assume for practical purposes that you have to consider also extreme cases; suppose that constant is embedded in a block that is very deeply nested in the program and if you are going to do it at compile time then the value of the constant itself is going to be an attribute of the constant and it is an attribute that is going to be preserved. We have not spoken much about attributes but general attributes include various kinds of information that you gather at compile time.

In the case of constants it might be the value if such decisions are taken. In the case of variables and also the type of a data item it is also an attribute. What is the amount of storage that is required for a record or for a double precision floating point integer? The amount of storage that is actually required either contiguous- or non-contiguous is also an attribute that you are going to keep. What kinds of runtime checks are to be preserved or kept? It may be part of a sub ranged type or an enumerated data type. For example; if it is part of a sub-range type then the bounds are also part of the attributes. In the case of a constant if you decide to assign its value at compile time then its value becomes an attribute of the constant which has to be preserved through the runtime.

In general if you decide to assign the values before at compile time then the usually large structured constants are infeasible. This is in fact an excellent reason why Pascal decided not to

have large structured constants. It is a language for learning about programming, learning how to program and so, it makes sense not to have large structured constants. Arrays and records are all out. Supposing it is going to be allocated at runtime then there are two possibilities. Either it is allocated once before program execution begins or it is assigned each time the block in which it is declared is entered. The complexity of assigning it once before the program execution actually begins is just that at this point if you assign it at runtime then you can actually allow for large structured constants.

[Refer Slide Time: 21:18]

In some initialization procedure values are assigned just as you enter the block or just before program execution all the structured constants could be read in. But large structured constants become feasible. Supposing you do it either at compile time or once before the program execution begins that does not necessarily mean that you have actually reached the block in which that constant has been declared. But just before program execution begins, all the constants in all the nested blocks are assigned the values. This means that you are going to sweep through the entire program looking for all the constants which have to be assigned values regardless of scope rules or whatever it may be.

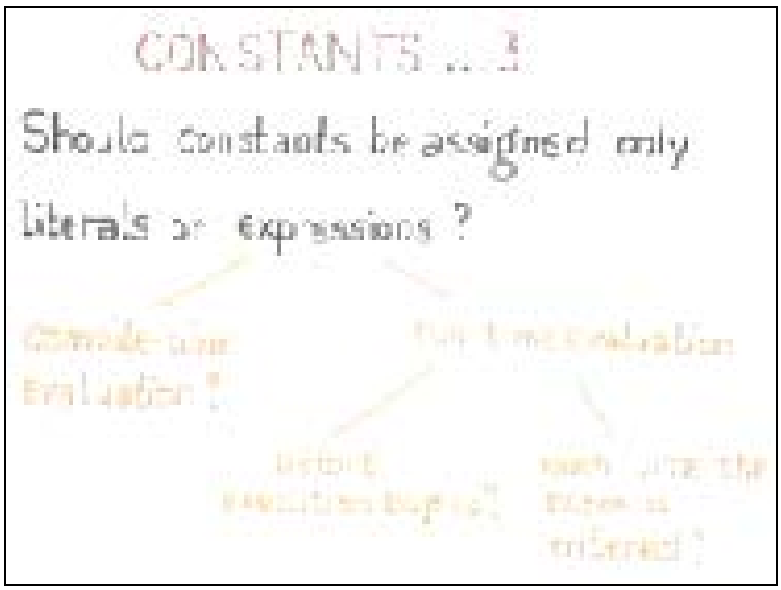
So, the constant cannot be part of a dynamically allocated storage. The storage has to be away from the activation record in some safe place. This is a decision you might take especially with large structured constants in order to save runtime. Once as part of the initialization you might just sweep through the entire program and perform an assignment of values to constants but then you have to be careful that that storage is separate from the activation record of the individual blocks because the activation record really signifies a dynamic allocation which has a very small extent that does not span multiple entries into that block. It has an extent which is only for one particular entry and exit into the block. Then the storage has to be away and this is one reason why you might have some data associated with the code segment. In my general structure where I spoke about data under the code segment and away from the activation record this is one possibility. You can actually store the constants along with the code, sweep through the code and

fill up all the constant data once before your actual main program execution begins. Those constants are then available. You do not have to redo those constants whereas if each time the block is entered if you have to assign a value to the constant then large structured constants will have to be reinitialized and reassigned values each time the block is entered, which is going to cut down on your execution speed firstly, because when you are talking of large structured constants, implicitly you are assuming that they will be stored in some external files or in some secondary form of storage. It is going to be much slower than memory. This means that there are going to be I O weights, interrupts and if in a multi- process system this program might get swapped out or it can become a background job and the actual elapse time will be much slower than even the lowering of this execution speed of the program.

In a stand alone mode only the slow down will be visible because of the access to external or secondary storage. But in a job environment of a large system the elapse time can be much more than the slow down in execution speed mainly because the job might get swapped out for I O. Then if it is going to be initialized each time then one possibility is that you have storage along with the other data items in the activation records but then you have to tag an attribute that is a constant and therefore it is of unchanging value that has to be carried through to runtime. It has to be an attribute of that storage location which has to be checked at run time to make sure that it is not updated. One benefit of having the constant stored along with the code segment is that there is a reasonable guarantee from the normal scope rules that that value is not going to be changed because only the ones in the activation record are going to be accessed.

However, if it is going to be stored along with other data items either in the activation record or in the heap area then there is a possibility that through aliasing and through various other means you might actually change its value unless you tag it sufficiently and put in code to prevent its implicit or explicit modification. Then the next question is: should constants be assigned only literals or can there be expressions? But then this also depends upon whether you are going to do a compile time evaluation of that if you allow expressions. If you allow literals the only question that arises is again of a large structured data. The problem reduces to whatever are the answers to the questions about large structured data or large structured constants.

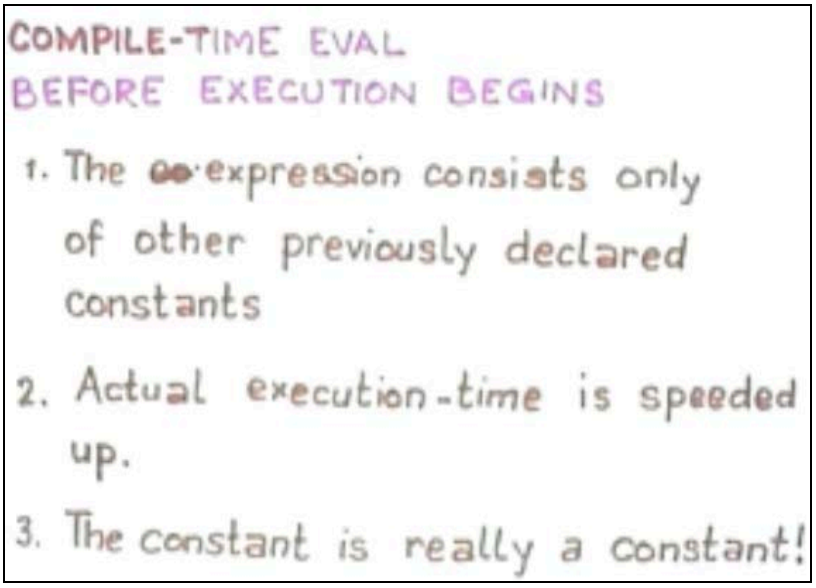
[Refer Slide Time: 29:02]



If you allow expressions then are you going to do a compile time evaluation or are you going to do a run time evaluation? If you are going to do a run time evaluation then is it going to be each time the block is entered or is it going to be before execution? If you are going to do a run time evaluation then you are always slowing down the execution of a program. If you are doing it each time the block is entered, you are slowing down the execution even more.

Secondly, if you do it each time the block is entered you have to ensure somehow that there are other questions that arise. Supposing you decide to do a compile time evaluation or you do it just before execution begins again it means sweeping through the entire program and doing an evaluation of all the expressions. One question is: should the expression consist only of the previously declared constants? In this case the sweep will work and if you do a compile time evaluation then your actual execution time is speeded up or if you do before execution begins also the execution speed might be acceptable. But then if you ensure that the expression consists only of previously defined constants then you have got reasonable guarantee that every constant is in fact a constant.

[Refer Slide Time: 31:18]



But supposing you do this evaluation at runtime at block entry time then there is no reason why you should prevent the use of variables that are global to the block from being used. Since the lifetimes of different activation records are anyway different and the constants have different values then the constant is not really going to be a constant. At different extents they might have different values. If you allow expressions that use variables but with the guarantee that those variables are always going to be previously declared and initialized then you could use global variables that is variables that are global to that block. Then those variables are reasonably guaranteed to have values and they are not likely to be undefined but then it means that your constant is no longer just a constant. It is a constant only for a particular lifetime and that block has several lifetimes and so that constant might have different values at different lifetimes.

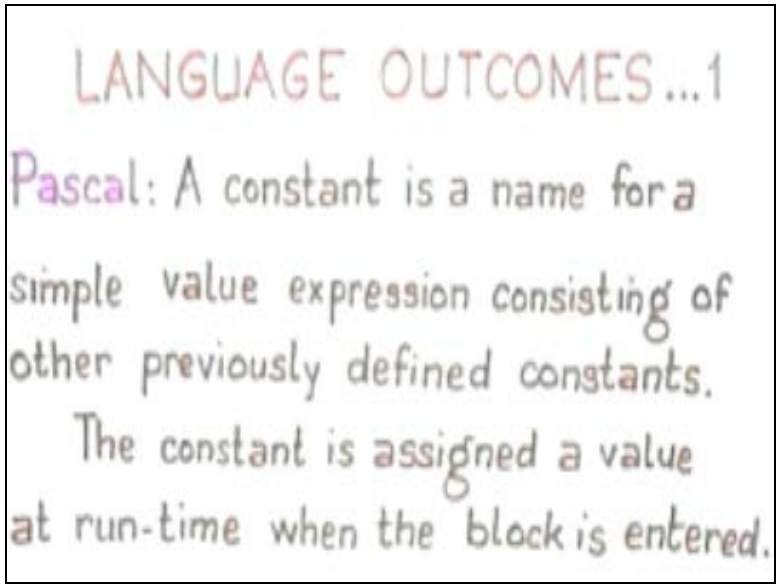
The actual execution speed is going to be slowed down with each evaluation. This means that you have a more flexible approach at the cost of execution speed and you also live with the fact that the constant really is not constant throughout the execution of the program but is only constant for each lifetime and at different lifetimes it could have different values. Another intermediate kind of decision that you might take is that your language might allow variable initialization and you could intersperse constant declarations with variable declarations and if your variable initializations are guaranteed then a constant declaration could follow a variable declaration and use that variable initialization in an expression.

But the fact that there are so many decisions to be taken at various points all this decision making can give the implementer difficulty because many of these issues are not there in the language definition. In the semantics these issues are not covered. So, what it means is that some kind of the language design at the semantics level should ensure that the implementer can take certain decisions which might be deemed right and not take arbitrary decisions. That is one excellent reason.

There is absolutely no semantic reason why Pascal does not allow the declaration of a variable to precede a constant. It does not allow declarations of constants and variables to be interspersed and one excellent reason is that it just did not want to get into so many policies. It decided to stick to some consistent policy. One is that a constant is a name for a simple value expression and it is just a name for some expression which might be very cumbersome or difficult to just type it out each time. So, you give it a value.

If you type out 3.14159265… repeatedly in your program you are likely to make a mistake. It is a convenient idea to just define a name called π and carefully type this out and use that π as the name for that value throughout. It avoided all the problems of trying to decide about large structured constants and what to do about them. One decision is that there are no large structured constants. All of them have to be variables and it is the programmer's responsibility to ensure that if he wants them to be constants then he keeps them as constants.

[Refer Slide Time: 36:38]



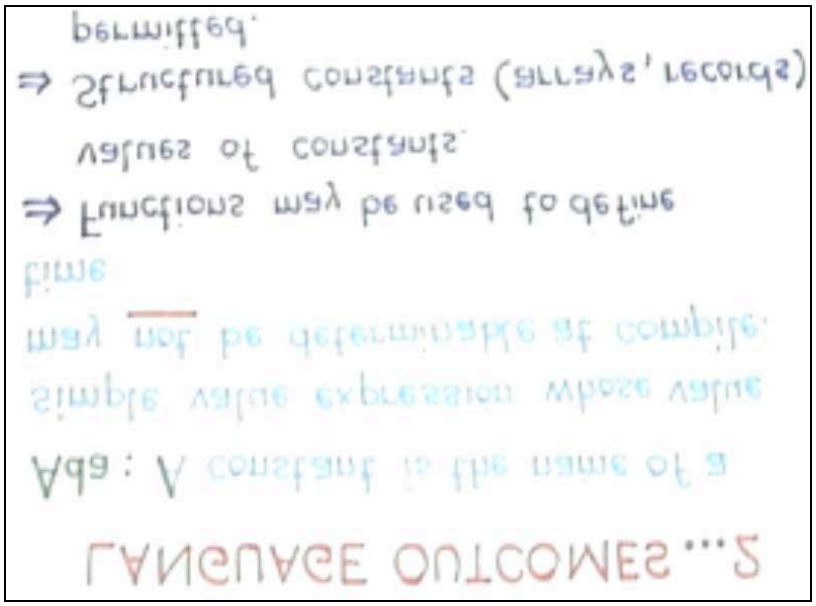
The point is that because of this policy actually the constants of a Pascal program are determinable at compile time. It is possible to do a compile time evaluation but somehow it was decided that a constant should be assigned a value only at runtime when the block is entered and since there was no variable initialization in Pascal there is no guarantee that if you allow variables in an expression assigning a value to a constant then you will actually get a meaningful result. They actually disallowed variable and constant declarations to be interspersed and since there was no variable declaration it was very clear that all constants can only use previously defined constants. So, a constant will actually be always constant. The only point is that they took this decision that the constant is assigned a value at runtime when the block is entered.

But that is perfectly alright. Since the constant is determinable at compile time you can evaluate the expression at compile time and store it as an attribute to be preserved at runtime and to be allocated storage in the activation record itself and maintain your scope rules. It is a simple

decision which actually side-steps all these other policy questions that arise. Since it is a language for learning about programming you might think that it is reasonable.

If you were to take some other language like Ada, which is not designed for learning about programming but is actually meant to send- satellites for various purposes then you should allow for large structured constants. But then the point is that they still stuck to the Pascal decision of having simple values and they assumed that it may not be determinable at compile time. At each lifetime the constant might actually have different values. Then what they also allowed was that you might use functions to define the values of constants which means you are going to use variables that are already available.

[Refer Slide Time: 39:18]

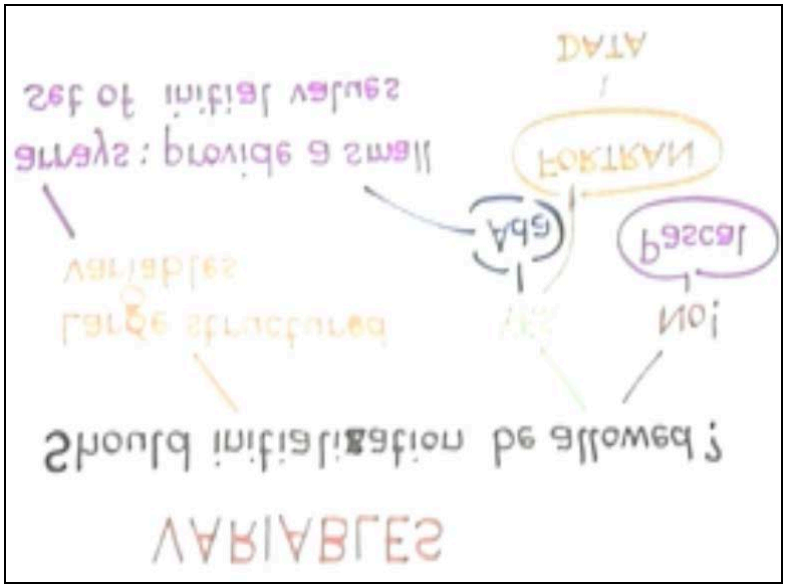


Since you can use functions it clearly means that nothing is going to be actually compile- time determinable unless explicitly as in the case of π where there is no use of explicit use of variable. It is of course easy to do a scan of the program. After the parsing phase it is possible to scan the program to find out about dependencies for each of the constants and clearly demarcate which constants can be evaluated at compile time and which constants can be evaluated only at execution time and then take a decision instantly. Make sure that you fill up those decisions at runtime during block entry time.

They also allowed structured constants like arrays and records to be permitted. After all remember that maybe a missile must know the exact longitude, latitude and altitude of a nuclear installation and so you require large structured constants.

Now let us get on to variables. In the case of variables actually the problems are not so severe. There are a few default decisions that we have to take but more or less the only other decisions are really that of storage.

[Refer Slide Time: 41:11]



Should initialization be allowed? Should initialization be mandatory? But not everybody likes initialization being mandatory because the whole idea of a variable is that it provides you a lot of flexibility. The only other question is should initialization be allowed? One possibility which is that in for example the language Pascal it does not allow any initialization at all. The syntax itself throws out the notion of an initialization. Languages like Ada actually allow you to provide a small set of initial values. So, even though it might be a large structured array you could have a declaration of this form.

Let me call this

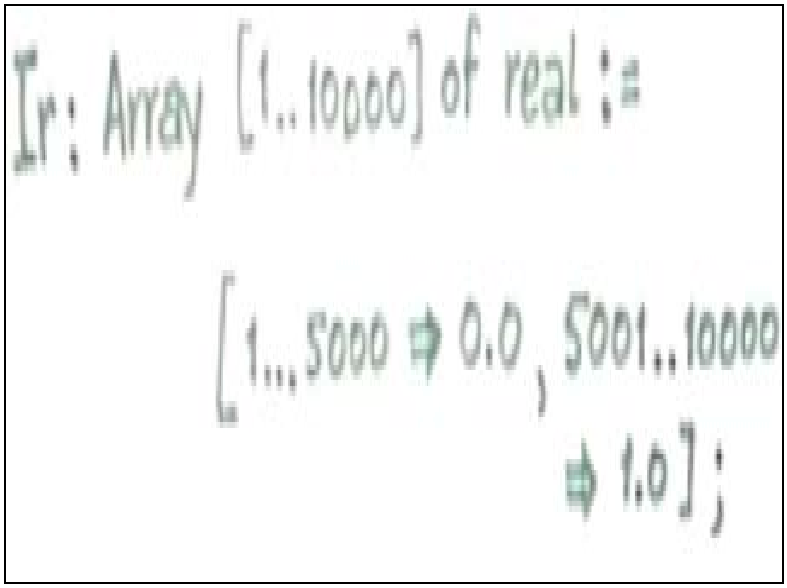
I r : Array [1….10000] of real : =

[1….5000 => 0.0, 5001…10000

=> 1.0];

So this is how the syntax of an initialization would look.

[Refer Slide Time: 42:57]



In large structured values like large arrays if you provide a small set of initial values then you could actually enumerate in such a fashion but if all the 10,000 values are going to be different then you have a severe problem. They do not answer this question as to what they are going to do in such a case. After all why should I be restricted to 2 or 3? Why can I not write 10000 different clauses?

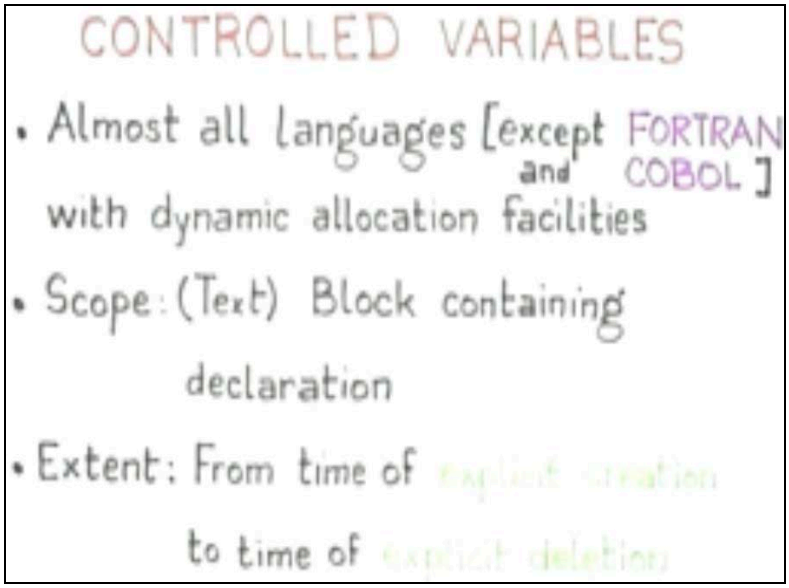
The other question is will a programmer write 10000 different clauses? He will take the easy way out and just write 1 or 2 values. That is it. He will read the rest of the values from file for instance but he is not likely to use that initialization unless he is absolutely sure that there are only a few initial values possible. The question of whether it is going to be mandatory is useless because you cannot prevent large structured variables. They have to be there. That is not a decision that is open to you. If you want your programming language to be actually used by people then you had better allow for large structured variables. But you just have something that is not mandatory but it provides for a small set of initial values whose size might be dependent.

In the case of languages like FORTRAN for example, there is an explicit data statement which allows for an initialization. Again in FORTRAN it is not mandatory and everything is initialized by default even if you did not provide an explicit initialization whether you like it or not. Whatever is not initialized by the data statement anyway has a default initialization in FORTRAN. If you are just looking for possibilities of allocating storage to variables then you could allocate them either here or here or here. There is absolutely no reason why you cannot allocate them in any of these three places if this is going to be your typical structure. There have to be good reasons why you want to allocate something here or here if your default allocation is going to be in the runtime stack.

As far as the heap is concerned it does not really matter. The heap is used for a dynamically allocated variable. So, the heap is really going to be like a garbage heap.

This means that for explicit allocation and de-allocation of variables you can use the heap. That is the policy most languages follow. The heap space is mostly used by all languages which allow for a heap space and where there is a dynamic or explicit allocation and de-allocation done by the user of the language.

[Refer Slide Time: 46:54]



The scope of a variable is still the block containing the declaration of that variable which has to be dynamically allocated. The extent of that variable is also determined by the programmer. Therefore the extent is from the time the program explicitly creates data object to the time the program explicitly deletes it from the heap. There is an explicit creation and deletion. For example; a language like Pascal has this ‘new and dispose’ which are explicit creation and explicit deletion commands. Other languages like P L 1 followed FORTRAN. They had to clearly specify what is automatic, what is controlled and what is static.

The static was supposed to be FORTRAN, the automatic was supposed to be what is normally a variable in a block structured language and the control was explicit programmer creation and deletion. The other possibility is to actually use the own variables of Algol 60. As I said last time, the problem that the Algol 60 designers felt was how you are going to use random numbers unless you have a procedure which is somehow history-sensitive. It really depends on how many times it has been called before. There has to be some variable. The seed of a random number generator should be changing every time and normally you would take the last random number generated as the seed for the next invocation. This means that in such a procedure you require a seed whose value should be retained between different life times of that procedure. Most languages actually allow for this version of own variables.

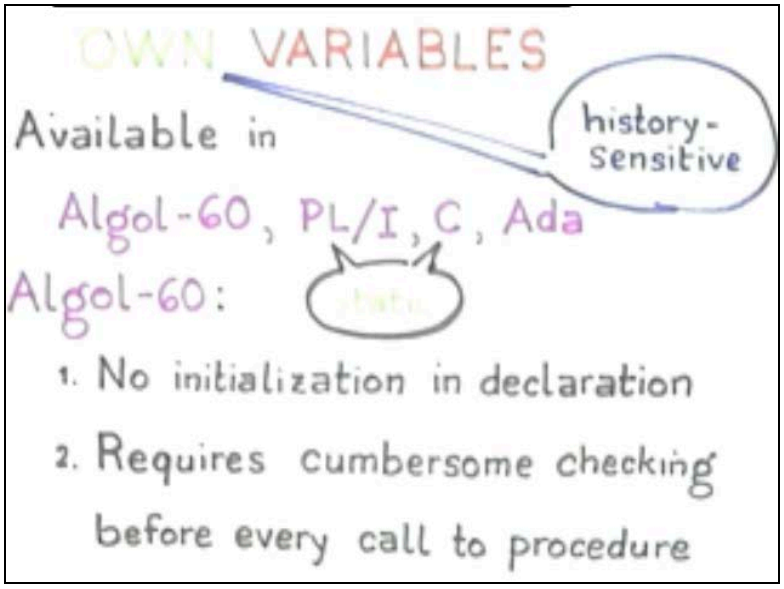
Normally, since the values of these own variables have to be preserved between two different lifetimes of that block they are stored in the data area under the code segment. That would be the normal place to store them so that they do not get washed out when you do block exits. Note that heap space is still accessible only through the runtime stack. So, if your activation record gets

washed out on block exit then you cannot store the own variable in the heap because the pointer to it has been lost.

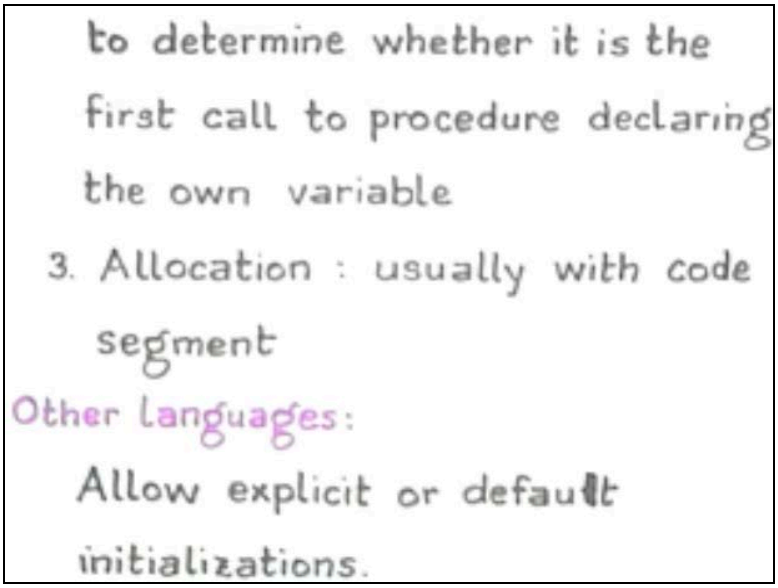
The third space is that reserved space allocated to each block in which history-sensitive variables may be stored. They have this reserved word called ‘own’ for a variable which for example in C and P L 1 they call static, which retains its value between successive invocations and you can use that. The only problem with these ‘own’ variables is that if your language does not make variable initialization mandatory then there may not be any initialization. This means that each time you call this procedure which has an ‘own’ variable, you have to introduce code in your calling procedure to check whether it is the first time that procedure is being called or not.

If it is the first time then you have to provide an initialization value for the own variable. If it is not the first time then you just call it. You need to introduce every call to that procedure which contains an ‘own’ variable if your language does not allow initializations and declarations.

[Refer Slide Time: 51:18]

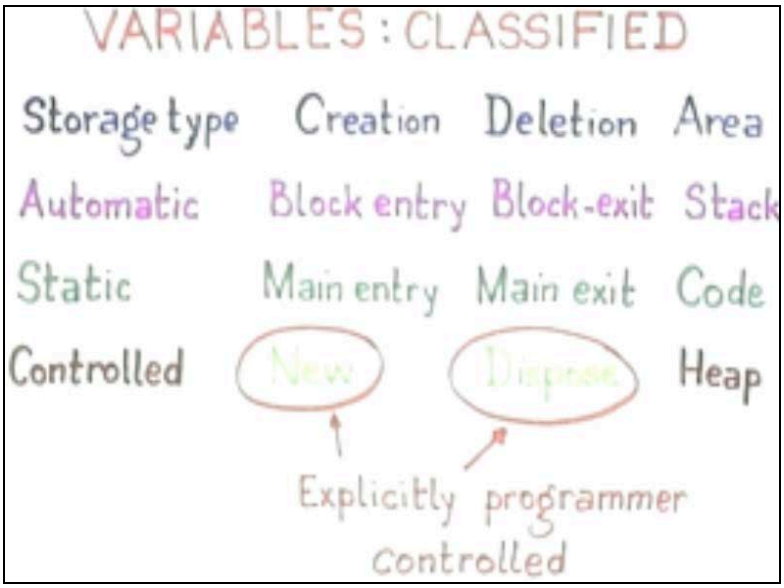


[Refer Slide Time: 51:38]



You have to write a cumbersome piece of code to find out whether this is the first time you are calling that procedure or not. Normally, this allocation is with the code segment if in languages with default initializations, there is no problem of checking whether it is the first time or not and most languages which allow it actually allow for explicit or default initializations.

[Refer Slide Time: 52:06]



We might finally classify variables in this form. You have variables which are automatic in the sense that they are created at block entry, they are deleted on block exit and the area where they are stored is a runtime stack. You have static variables which are created when you enter the program but may not be initialized depending on the other features of the language. They are

deleted when you exit the entire program. They have an extent spanning the entire execution of the program and the area is usually the code segment area and there are controlled variables which are explicitly programmer controlled, which are created by an explicit programmer command and which are deleted by an explicit programmer command. They are stored usually in the heap and they have a lifetime between their creation and their deletion.