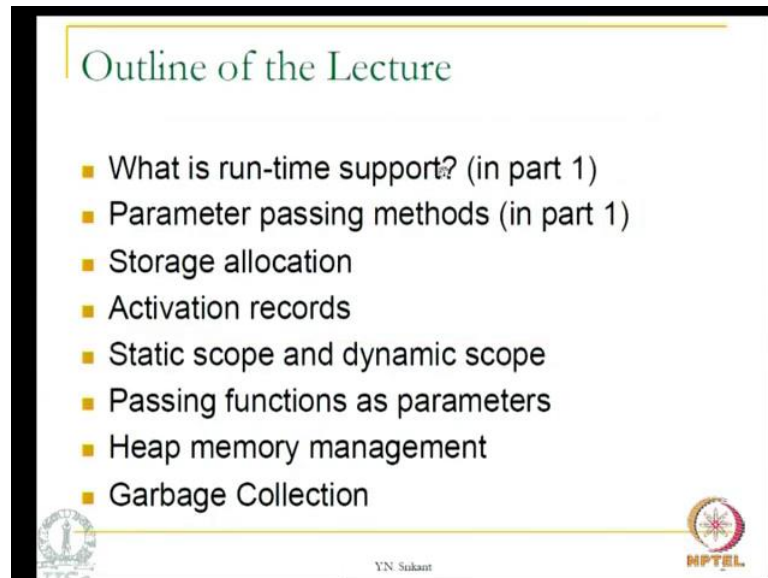


Principles of Compile R Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture - 21
Run-time environments Part – 2

(Refer Slide Time: 00:23)



Outline of the Lecture

- What is run-time support? (in part 1)
- Parameter passing methods (in part 1)
- Storage allocation
- Activation records
- Static scope and dynamic scope
- Passing functions as parameters
- Heap memory management
- Garbage Collection

YN Srikant

MPTEL

Welcome to part two of the lecture on runtime environments. So, we saw the need for runtime support and the parameter passing methods in the last part. Today we will continue with the rest of the topics on storage allocation, etcetera.

(Refer Slide Time: 00:34)

Code and Data Area in Memory

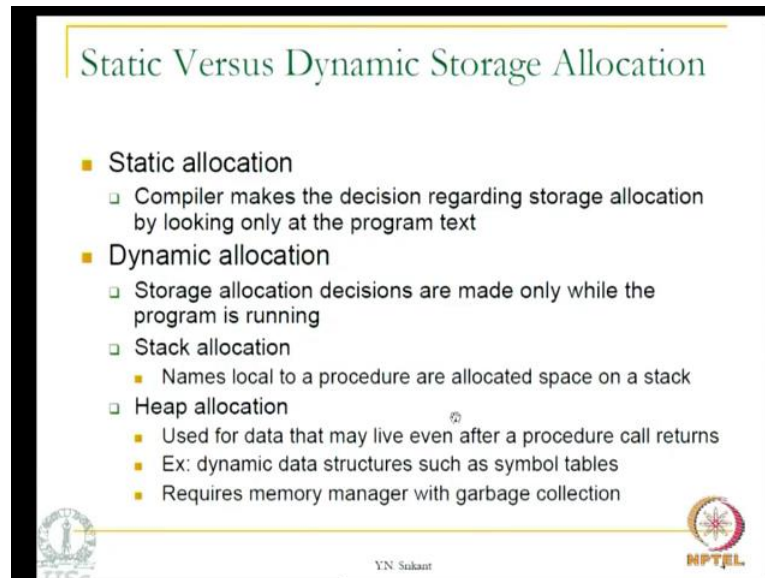
- Most programming languages distinguish between code and data
- Code consists of only machine instructions and normally does not have embedded data
 - Code area normally does not grow or shrink in size as execution proceeds
 - Unless code is loaded dynamically or code is produced dynamically
 - As in Java – dynamic loading of classes or producing classes and instantiating them dynamically through reflection
 - Memory area can be allocated to code statically
 - We will not consider Java further in this lecture
- Data area of a program may grow or shrink in size during execution

YN Srikant

So, programming languages the implementation usually distinguishes between code and data. And even in the language specification you know we have code which uses the variables, whereas the data which declares the variables; these are all different, and when we compile the program the code that is produced is actually just machine instructions and in 99.9 percent of the cases it does not have any embedded data in it. So, it is a good idea to keep both the data, and the code separate even when the program executes on a machine. The reasons for this are many actually. For example the code area does not grow or shrink in size as the execution proceeds. Whereas the data area can grow or shrink in size, but when can the code area change?

Well, if you consider java, which has dynamic loading of classes, and of course, it also has another facility to produce classes during runtime using reflection and then creating objects of that particular class. So these features of java actually increase the code size, because once you produce a class or load class which did not exist before. The methods of that class will also be loaded, and that means more code is added to the existing program. Memory area of course, can be allocated to code statically, and the reason is; in general if we do not use java or any other languages similar to that the code size does not change. And we can say now; the code will be placed in a particular area and it will never be shifted out of that area. And we will not consider java in this lecture.

(Refer Slide Time: 03:20)



Static Versus Dynamic Storage Allocation

- **Static allocation**
 - Compiler makes the decision regarding storage allocation by looking only at the program text
- **Dynamic allocation**
 - Storage allocation decisions are made only while the program is running
 - **Stack allocation**
 - Names local to a procedure are allocated space on a stack
 - **Heap allocation**
 - Used for data that may live even after a procedure call returns
 - Ex: dynamic data structures such as symbol tables
 - Requires memory manager with garbage collection

YN Sukant

MPTEL

We will consider java, and object oriented languages sometime later. Of course, as I already mentioned the data area of a program may grow or shrink in size during execution. And that is precisely what we are going to discuss in the rest of this lecture. We have 2 types of storage allocation; 1 is the static allocation, the other is dynamic allocation. So, what exactly is static allocation? The compiler says here is the data, and now it makes the decision regarding storage allocation of for this particular data by looking at the program text. It has no other consideration; I will give you an example of this you know in few minutes.

And, what about dynamic allocation; in this case, the storage allocation decisions are made only when the program is running. In the static case; the compiler itself makes the decision and nothing is left to the run time system. And there are 2 types 2 possible allocations. 1 is the stack allocation, other is the heap allocation. So, names local to a function or procedure are allocated space on a stack that would be stack allocation. Whereas, in the case heap allocation; usually this is not used for the names but this is used for dynamic data structures such as symbol table etcetera, etcetera. And this is used for data that may live sometimes even after a procedure call returns, of course; it can be used within a procedure also.

But the data structure being separate from the stack allocation given to variables etcetera, the data structure can live even after the procedure which created the data structure

actually terminates. For example, in the symbol table structure that is used by compilers; many stages, or phases of the compiler modify, or use the symbol table. And it is created at different places in the compiler, but it leaves as a global data structure. So, heap allocation requires a memory manager. And garbage collection is also required whether it is explicit or automatic that is up to the language and runtime system. So, we will study this a little later.

(Refer Slide Time: 05:46)

The slide is titled "Static Data Storage Allocation". It features a list of characteristics on the left and a memory map diagram on the right. The diagram shows a vertical stack of four light green boxes representing memory segments: "Main program variables", "Procedure P1 variables", "Procedure P2 variables", and "Procedure P4 variables". Below these boxes is the label "Main memory". The slide also includes logos for IIT Bombay and NPTEL, and the name "YN Srikant" at the bottom.

- Compiler allocates space for all variables (local and global) of all procedures at compile time
 - No stack/heap allocation; no overheads
 - Ex: Fortran IV and Fortran 77
 - Variable access is fast since addresses are known at compile time
 - No recursion

Main memory

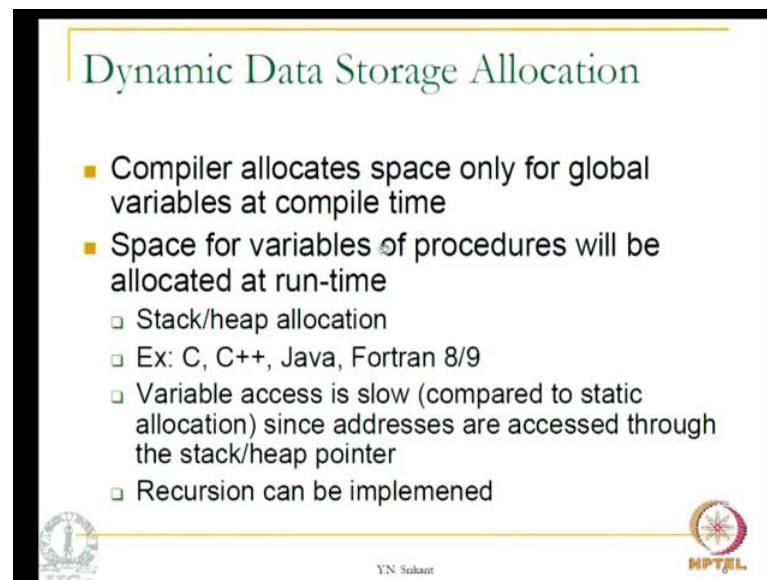
So, let us come back to static data storage allocation. Here, is a memory map of the of a particular program; there are main program variables then procedure P 1, and it is you know it is variables, procedure P 2 it is variables, procedure P 4 it is variables, and so on and so forth. As I said the code is separate; so these are just the data areas, and all this is in main memory. So, what is being shown here is that the addresses from this point to this point are occupied by the main program variables, from here to here it is occupied by the variables of P 1. From here to here it is occupied by the variables of P 2, and from here to here by the variables of P 4.

So, these addresses are fixed, they are not going to change at any time. The compiler allocates the space for all the variables both local and global of all the procedures at compiled time itself. The characteristics of this are; there is no stack or heap allocation. So, there is no overhead in accessing; the variables there is no need to create what is known as activation record as we will see later. So, the overheads are very less, the

languages which have such static data allocation are Fortran 4 and Fortran 77, which are very old programming languages. Then, the advantage is because the addresses are all known at compile time, accessing the variables is very fast. And the disadvantage of this scheme is you cannot implement recursion using static allocation. I will simply tell you why.

Say suppose, procedure P 1 calls itself now, the data area of P 1 is fixed. The same area will be used by the second instance of P 1 which being recursively is called the original instance of P 1. And the second instances are both alive at the same time; but the data area being simple single. The second instance of procedure P 1 will over write all the data which was probably created by the first instance. So, this implies that we really cannot use recursion profitably; like for example, in the case of factorial the values of the factorial for smaller numbers will be overwritten by the higher numbers. So, this is the problem that we have here. So, rather any other Fibonacci or any other number, you know when you write a recursive program; as the recursion goes deeper and deeper the old data would be destroyed. And the new data will be written over that and thereby no useful work gets done.

(Refer Slide Time: 09:00)



Dynamic Data Storage Allocation

- Compiler allocates space only for global variables at compile time
- Space for variables of procedures will be allocated at run-time
 - Stack/heap allocation
 - Ex: C, C++, Java, Fortran 8/9
 - Variable access is slow (compared to static allocation) since addresses are accessed through the stack/heap pointer
 - Recursion can be implemented

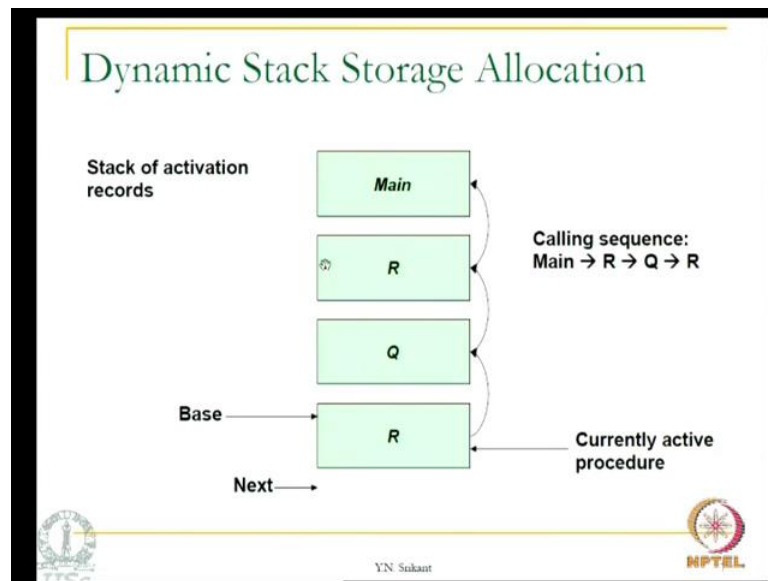
YN Srikant

MPTEL

So, what exactly is dynamic data storage allocation? The compiler allocates space only for the global variables at compile time. It does not allocate any space for the variables of procedures at compile time; they will all be allocated only at runtime. So, the implication

of this is that there is either a stack or heap necessary to create the space for all these variables. And the languages which use such data storage allocation are C, C plus plus, java, Fortran 8 or 9, etcetera, etcetera. The access to variables is a bit slow compared to the static allocation simply because the addresses are now accessed through a stack or heap pointer. And of course; the biggest advantage is that recursion can be implemented in this case.

(Refer Slide Time: 10:00)

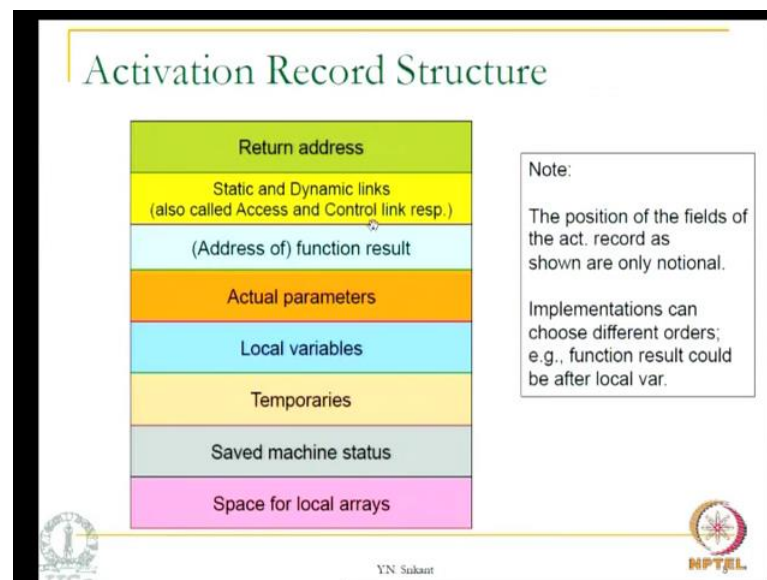


So, this is how the schema would be, in the case of dynamic storage allocation. Here, is the main program, which calls the function R, R calls Q and then Q call R again. They what is shown here are the, what are known as activation records, which are nothing but data areas for the various activations of the functions or R procedures. So, the variables of main R all stored here, variables of R all stored here. Similarly, for Q and this is the second instance of the function or R procedure R. And you can observe that the variable space for this second instance and the first instances are different, there by useful work can be done by both the instances.

So, the currently active procedure is at the top of the stack here. So, if there is a sort to begin with we have allocation only for the global variables, and main. When main calls R the data space for R is created and then when it calls Q the space for Q gets created. And then when there is a recursive call to R another space for gets created and so on and so forth. And as the procedures terminate, so when R terminates the space for R will be

released, and the picture would be you know base will be here, and next will be here. Similarly, when Q returns the space of first space required by Q use by Q will be returned. And similarly, the space of R as well, and finally; when the main program terminates all the space will be released by the runtime system.

(Refer Slide Time: 11:50)



So, I kept mentioning the activation record more than once. Basically, an activation record has space for all the variables temporaries, you know the machine status, the function result, and the return address, apart from what is known as static and dynamic links. So, let me explain some of these now, and we will postpone a few others late R to other to a late R point in time. Another important thing is the position of the fields which are shown here, are only notional it is not necessary that the same layout is used by all compilers. For example; you know saved machine status, and you know the, could possibly be at the beginning return address would also be at the end.

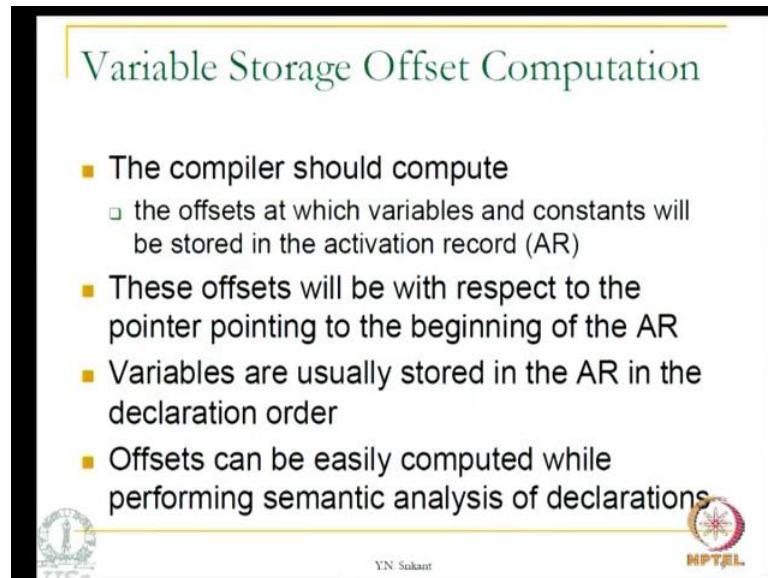
The function result could possibly be somewhere here. So, it is possible to change the location of these fields without affecting either the efficiency o R speed of the program itself. So, we know very well what exactly the return address is; it is required by the program to return to the caller. We will postpone the discussion of static and dynamic link, which is used to access global variables from the current procedure. The function result; so we already saw in the case of intermediate code generation that the address of the function result, the variable which is going to contain the function result. You know

the address of that variable will be passed as a parameter implicit parameter, and that is what it is put here.

So, the address of this variable actually belongs to the callers address space. These are the actual parameters then there is a lot of space for local variable and temporaries used by the function. And finally of course; saved machine status, and space for local arrays. The question that arises is why is the space for local arrays at the end, instead of being somewhere here? Suppose there are local variables which are arrays, why was the space not allocated among these variables, and why is it being push to the end. Well the same is true for parameters as well; if there are array parameters why are they not being allocated here, and why are they being allocated here. The answer is the local variables o R parameters which are non arrays; we will all require known amounts of space.

Therefore, the offsets for the various variables and parameters can be computed very quickly and very easily by the compiler. Whereas, the size of the arrays can possibly vary, you know if it is a parameter and the size of the array is not being supplied. Then, it is best left to the end of activation record, because we will know the space for this array only after we enter the procedure. You know the caller calls the procedure only then the size of that array will be known. Otherwise when we compile the procedure the size of the array will not be known. So, and of course; making it more uniform among these is the other reason. But technically speaking; including the unless the size of the array is unknown at compile time, including it within the space for local variables, o R parameters does not make any difference as far as the compile R is concerned.

(Refer Slide Time: 15:51)



The slide is titled "Variable Storage Offset Computation" in a green serif font. It contains a bulleted list of four items. The first item is a yellow square followed by "The compiler should compute", which is followed by a white square containing the text "the offsets at which variables and constants will be stored in the activation record (AR)". The second item is a yellow square followed by "These offsets will be with respect to the pointer pointing to the beginning of the AR". The third item is a yellow square followed by "Variables are usually stored in the AR in the declaration order". The fourth item is a yellow square followed by "Offsets can be easily computed while performing semantic analysis of declarations". At the bottom left is a circular logo with a book and a lamp. At the bottom center is the text "YN Sukant". At the bottom right is a circular logo with a gear and a star, with the text "NPTEL" below it.

- The compiler should compute
 - the offsets at which variables and constants will be stored in the activation record (AR)
- These offsets will be with respect to the pointer pointing to the beginning of the AR
- Variables are usually stored in the AR in the declaration order
- Offsets can be easily computed while performing semantic analysis of declarations

So, there is you know little more on the variable storage offset computation. We saw how this could be done during the, you know the other lectures. For example, the compile R should compute the offsets at which the variables and constants will be stored in the activation record. So, I showed you this picture; so here are the local variables and if there are several variables here, we should know at which position these variables will be placed on the stack. So, with respect to the beginning of the activation record, which is denoted as zero, the offsets of all the variables and temporaries will be computed. So, the variables are usually stored in the activation record in the declaration order. So, this is something I already mentioned, they can be computed during semantic analysis of the declarations.

(Refer Slide Time: 16:53)

Overlapped Variable Storage for Blocks in C

```
int example(int p1, int p2)
B1 { a,b,c; /* sizes - 10,10,10;
      offsets 0,10,20 */
...
B2 { d,e,f; /* sizes - 100, 180, 40;
      offsets 30, 130, 310 */
...
B3 { g,h,i; /* sizes - 20,20,10;
      offsets 30, 50, 70 */
...
B4 { j,k,l; /* sizes - 70, 150, 20;
      offsets 80, 150, 300 */
...
B5 { m,n,p; /* sizes - 20, 50, 30;
      offsets 80, 100, 150 */
...
}
```

Storage required =
 $B1 + \max(B2, (B3 + \max(B4, B5))) =$
 $30 + \max(320, (50 + \max(240, 100))) =$
 $30 + \max(320, (50 + 240)) =$
 $30 + \max(320, 290) = 350$

Overlapped storage

Overlapped storage

YN Srikant

MPTEL

I want to show you a little more than what we did in the semantic analysis stage, for the variable storage allocation for other offset computation. Here is a program procedure called `int example`; so it returns `int` as its result. It has 2 parameters then it has 3 you know variables `a b c`, each of size 10 so 10 10 and 10. Now, for this particular beginning it is very easy to compute the offset, because `a` begins at 0. Obviously `b` will have an offset of 10, and `c` will have an offset of 20. Let us assume that in the `c` style; there are blocks in between not procedures just you know the blocks compound blocks. So, the block `B 2` will have 3 variables `d e` and `f`. These are possibly arrays so they have sizes 100 180 and 40 respectively.

And, now the last variable was given offset of 20 here, its size was 10. So, it is only correct that the offset of the variable `d` here is 30 then 130, no 30 plus 100 and 130 plus 180 would be 310. So, this is the offset information as far as `B 2` is concerned. `B 2` ends here, and another block `B 3` begins at this point. So, the variables of this block and this block have nothing to do with each other. In fact I cannot access the variables of the block `B 2` from within the block `B 3`. Therefore, there is nothing wrong in reusing the space that was given to the variables of `B 2` for the variables of `B 3`. So, that is what we are trying to do.

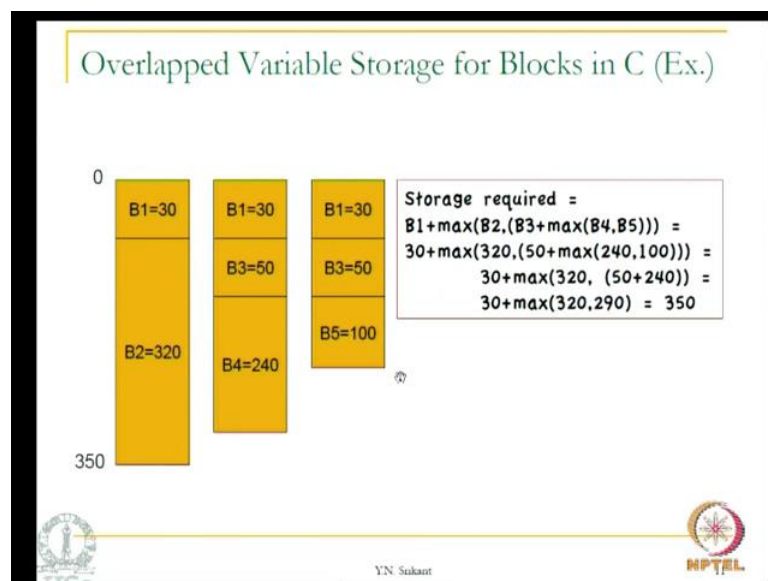
Now, how do we do that the offsets began here at 30. So, we will also begin the overlapping block offset at 30 here. The sizes are 20 20 and 10 for the variables of `B 3`.

So, the offsets are appropriately computed as 30 50 and 70. Within B 3 there are 2 more overlapping blocks B 4 and B 5 which again share the same storage area. So, for B 4 the offsets begin at 80, because this is 70 and plus 10 would give you 80. And the other 2 are 150 and 300, because 80 plus 70 and 150 plus 150. This B 5 also begins its offset at the same value 80 and then of course; 80 plus 20 is 100 100 and plus 50 is 150.

The storage is overlapped for these 2 blocks. And of course; the storage is overlapped for this block and this blocks these 2 inclusive of within B 3. So, what is the maximum storage that is required for the entire function and its variables? So, this is very easy to compute, the storage required would be the storage required for B 1 then the maximum of the storage required by this and this. So, B 2 and followed by you knows the B 3 within B 3 we have B4 and B 5. So, again we have a max of B 4 and B 5.

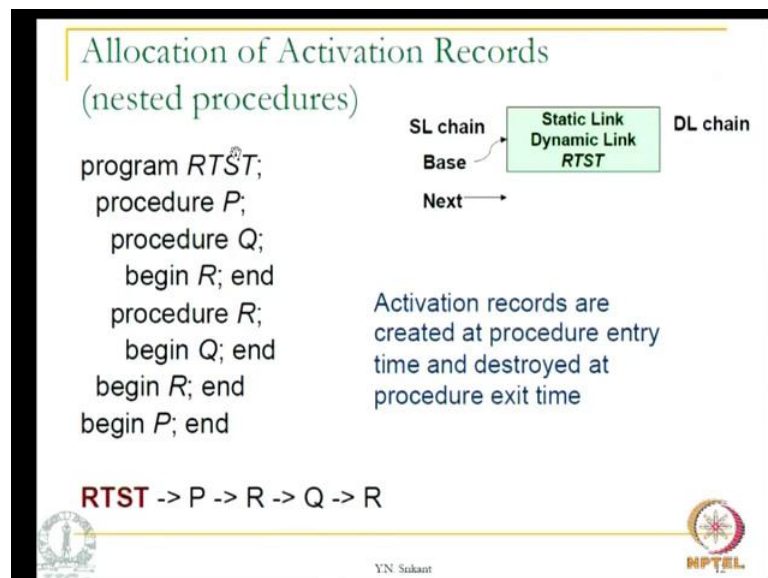
So, B 3 plus max of B 4, B 5 and that overlaps with B 2. So, if you replace the values as given here, we really get 300 and 50. This is much more than, the total space that is required by the program; if we simply add up all the sizes of all the variables. So, you can do that here you know this is 30. So, 320 plus 30 is 350 then plus 50 is 400 plus 240 is 640 and plus 100 would be 740. So, instead of that we are really using only 350 here, so this is the advantage of you know recognizing the fact that data blocks can you know overlap.

(Refer Slide Time: 21:45)



So, this is just another explanation of the same phenomenon. So, we have B 1 30 and B 2 320, the space here overlaps with B 1. So, B 1 remains the same within that instead of B 2 equal to 320 we have B 3 and B 4. Other possibilities we have B 1 B 3 and B 5, these actually happen at various points in time. So, for example; after B 1 begins you know its data area will always be present in memory. The area of B 2 comes into existence when B 2 executes, and it vanishes once B 2 terminates. The area for B 3 begins at the same offset here as B 2, and the same thing happens with B 4 and B 5. B 4 starts and then terminates so; the same area is used by B 5. And finally, when B 3 terminates all the area would be released and then B 1 terminates the program terminates.

(Refer Slide Time: 22:54)



So, now let us look at activation records with nested procedures. So, if there is no nesting of procedures for example; here let us assume that main R Q and R are all at the same level as in C. So, the allocation of activation records and deallocation of activation records is very simple I already explained it. As that calls sequence progresses the activation records created, and as the sequence strings and the procedures written the activation records are returned to the storage pool. Whereas, if we have nested procedures; 1 language which has such nested procedures is Pascal. It is necessary to know even though Pascal is not used in practice any more.

It is definitely necessary to know the implications of nesting procedures within other procedures. So, for example; here is a main program R T S T within that we have

declared a procedure P, within P we have declared another procedure Q. And this is the end of the procedure Q, Q and R are at the same level then we have the body for R here. And this procedure P has its body here, begin R end and finally, we have the body for the main program R T S T begin P end. Now, the program control begins at the main program so P will be called and when P begins execution, the body of P begins execution.

So, in this R will be called so when procedure R is called it calls Q in turn and then when Q is called it calls R in turn, and this recursion will go on for a while. And finally, you know this procedure R will terminate without calling Q and that would make Q also terminate. And finally, P will terminate and the program ends. So, this is the call sequence that we have assumed for our example. R T S T calls P, P calls R, Q calls R calls Q, Q calls R again and then the recursion ends. And the entire sequence drops down. In such a scenario it is not enough to simply link the activation records, and hope for the best. It is necessary to do something more than that.

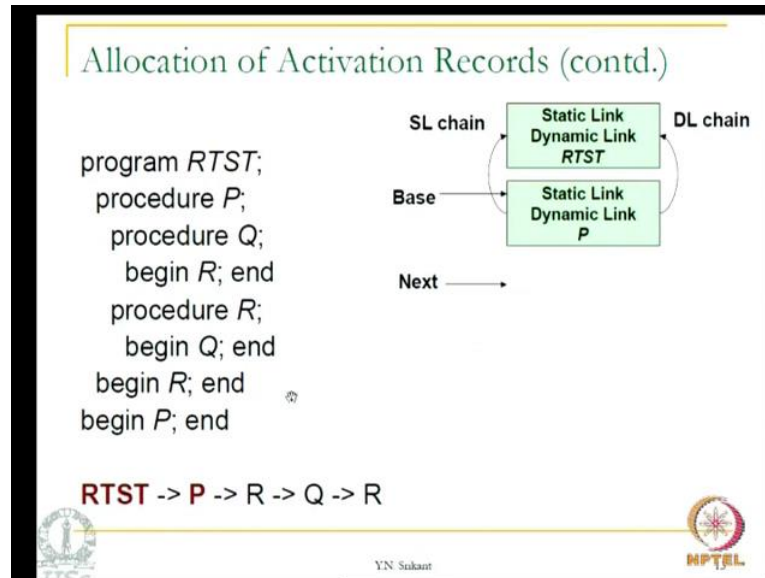
Activation records are created at procedure entry time and are destroyed at exit time as before. But the difficulty arises when we want to access the variables which are in various procedures, which have been invoked. So, let us understand the scope of the variables in such a scenario; for the main program the variables would all be declared just after the program statement. So, the main program can access all the variables declared within itself. But it cannot access any variables which are hidden within the procedure P or Q or R. So, these are actually insulated from the program R T S T.

So, we can only call the procedure P, and when procedure P starts executing; it will have its own variables just after the procedure statement. But again it cannot access any variables of procedure Q, or procedure R. The same thing holds as far as procedure Q and R it you know, Q cannot access the variables of R, and R cannot access the variables of Q. But when we are executing program the procedure P; we can access not only the variables of P, but also the variables of the main program. So, this is 1 level above it because P is nested within the program R T S T.

Similarly, when Q is executing it can access the variables within Q it can access the variables within P in which it is nested and of course, it can also access the main program variables in R T S T, because P is nested within R T S T. The same is true for R

as well variables of R variables of P and variables of program R T S T. But R cannot access the variables of Q, and Q cannot access the variables of R. This must be remembered when we construct the activation records as we go along.

(Refer Slide Time: 28:00)



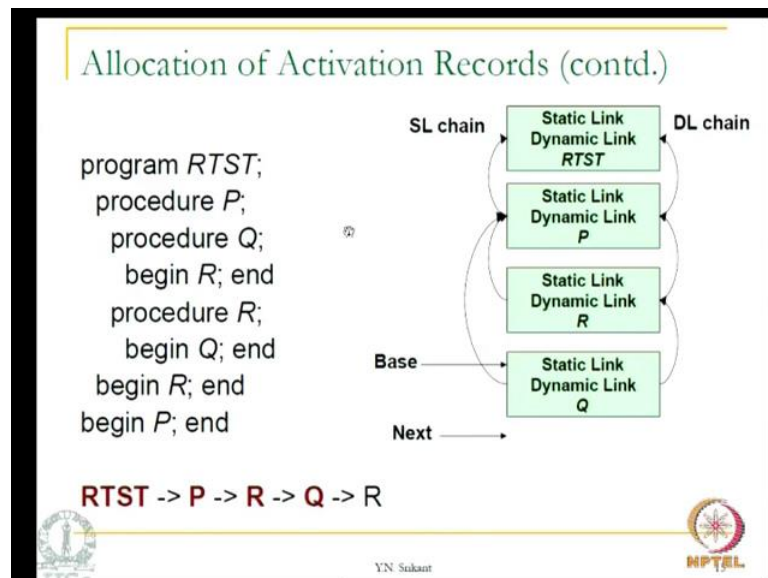
So, we call the procedure P; so an activation record for P is created. Now, let us understand how the variables are accessed from the procedure P. If the local variables of procedure P are being accessed then the top of the activation record which is available in the register base can be used to access the variables within the data area and activation record of P. We know the offsets of the various variables and parameters within P within the activation record. So, base plus that offset will give us the address at which the variable or parameter is situated. But what about the variables of the main program R T S T? It is not possible to do that using the base pointer, but we need to maintain extra information on the previous enclosing procedures activation record.

So, this is maintained in what is known as a static link chain S L chain. The D L chain simply you know chains all the activation records in order to maintain a stack structure nothing more than that really. So, when we want to access the variables of R T S T the S L you know field of the activation record has to be put into a register, and the contents of that activation of that register will now point to the beginning of the activation record for R T S T. So, we must now consider this particular value and then access the variables of R T S T using the offset.

So, in other words we have we need an indirect addressing mode here. So, rather double indirection; the first level of indirection we must move the contents of the static link into a register and then use the contents of that register as an address. So, we require 2 levels in order to access the variables of R T S T. whereas, to access the variables of P we already know the activation record address in base. So, only 1 level of indirection will be required. Now, P calls R so you can see that R is nested within the procedure P. So, we can access the variables of R the procedure that of the procedure P and also the program R T S T.

So, as usual the variables of R can be accessed very easily using the base pointer. And then using 1 level of indirection using the static link, we can access the variables of P, and using 2 level of indirection we can access the variables of R T S T. Every time even though the pointer points to the middle of this activation record; it is actually giving us the beginning of the activation record, the address of the beginning of the activation record from where all the offsets are measured. So, we must go from here to this activation record access the static link in that activation record, take that and go to this activation record and access the variables within R T S T. So, 2 level of indirection will be required. This is correct because I can access the variables of R and then P and also the R T S T.

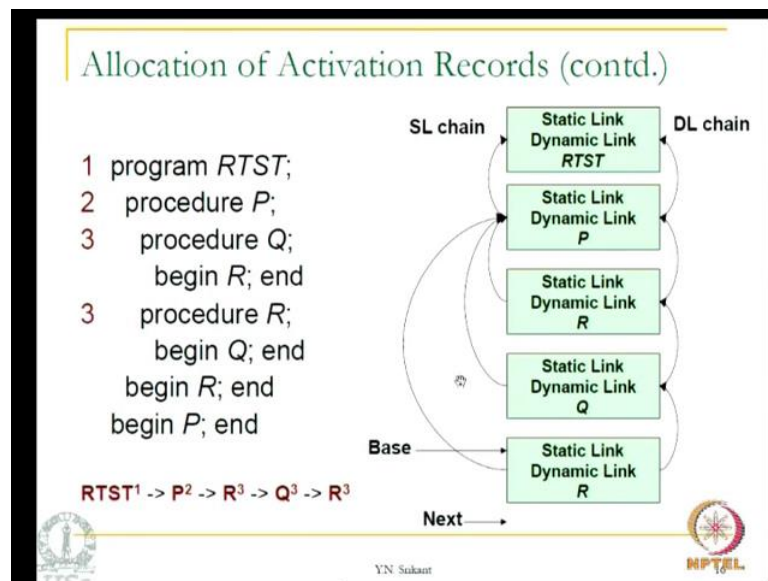
(Refer Slide Time: 31:58)



Now, Q calls R so as rather we call P then R then R calls Q. So, the activation record for a Q gets created but from within Q, I cannot access the variables of R. I can only access the variables of Q variables of P and variables of R T S T. So, making the static link point to the activation record of R would be a mistake. Because we can now, access the variables of R also using the static link available in Q to be correct the static link of Q should point to the activation record of P. Just like the static link of R pointed to the activation record of P.

So, now I can access the variables of Q directly using base variables of P using 1 level of indirection using static link. And using 2 static link indirections I can access the variables of R T S T also. All this logic link traversal must also be translated to instructions in machine code. So, at runtime these instructions will be executed the static link fields will be moved into appropriate registers and then the variables will be accessed at runtime.

(Refer Slide Time: 33:26)



The last call in our chain Q calls R, so R is created again we cannot link the static link of R to the activation record of Q. And we cannot even link it to this R, because these 2 R s are very different. They are 2 instances of the recursive invocations; it should actually point to the activation record of the P. Exactly like this point it to the activation record of P. So, now we can access and then 1 traversal will give us the variables of P, and 2 traversals will give us the variables of R T S T. So, this is how the static link chain is

used in order to access the variables, which are global to this particular any 1 of the procedures.

So, we must allow observe rather we must also discuss how exactly we fill up this static link chains. So, as the call chain progresses; you know say it is fairly easy to understand that the creation of activation record takes place after the Cali assumes control, because the exact size of the activation record will be known to the Cali function. It will not be known to the caller. Cali functions can possibly be compiled even separately. So, the total area within the, for the variables of the function its temporaries will not be known to the caller. Callers will only the size of the parameter list and nothing else. So, the complete creation of the activation record really happens in the Cali code.

Now, something else must also happen during the callers, while the caller code executes. What exactly is that? The level or the nesting level of the caller that information is lost once the control to the Cali is actually handed over. You know so if we handover the control to the Cali from where did we come, and what was the nesting level of that particular procedure. This information will not be available anymore. The creation of the static link actually depends on both the caller and the Cali. So, let us understand how this can be done.

So, we have this call chain here; let us consider somewhere here in the middle, you know P 2 calls P calls R, the level of P 2 is indicated as 2 here, P is indicated as 2 here, level of R is indicated as 3 here. The formula which is used you know here is $L_1 - L_2 + 1$. So, what exactly does it show; it says we must skip $L_1 - L_2 + 1$ records starting from the callers activation record, and establish the static link to the activation record that is reached. So, let us understand what this is and then I will tell you what exactly the code that is going to be generated is. So, $2 - 3 + 1$ is 0. So, this is just the formula $L_1 - L_2 + 1$. So, we are really here P calls R. Now, we want to establish the static link of R, the static link of P has already been filled let us assume that.

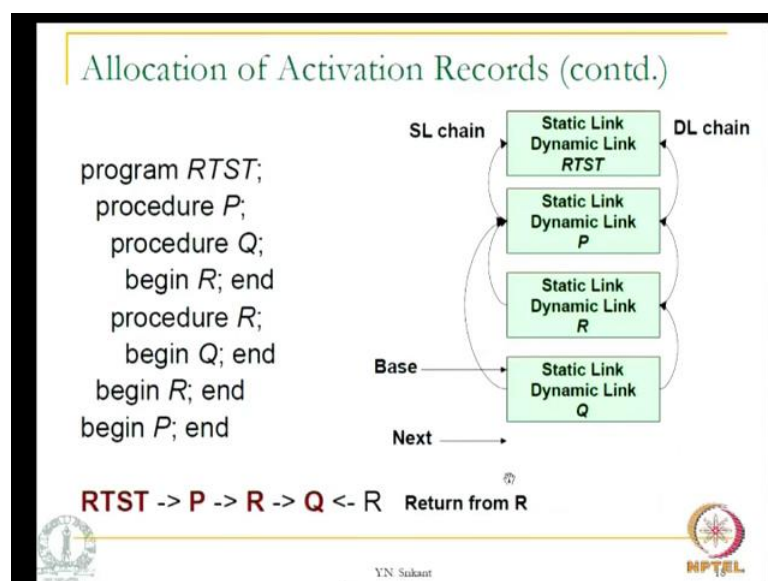
So, this is the caller, P is the caller and R is the Cali. Starting from the callers activation record; we must now skip 0 activation records, because $L_1 - L_2 + 1$ is 0; that means we stay at P. So, the activation record which is reached after we skip $L_1 - L_2$ records is P itself. Therefore, it is correct to establish the static link of R to point to P itself. Now, let us understand what happens when R calls Q. So, the next level R calls Q;

so this. So, the level of R is 3, the level of Q is also 3 so 3 minus 3 plus 1 is really 1. So, the formula tells us that starting from the caller which is R we must skip 1 activation record.

Now, we point to now we actually come to P, and it says the static link of Q must now be established so that it points to P. So, while the code generator is generating code the number of half's that must be done. You know must also be translated into instructions; so if we say we want to skip 1 link here, it implies that you know we read the activation. The static link of within the activation record of R and then takes the contents of that. So, that will give us the beginning of the activation record, and that is to be used as the activation rather the static link of this particular Q.

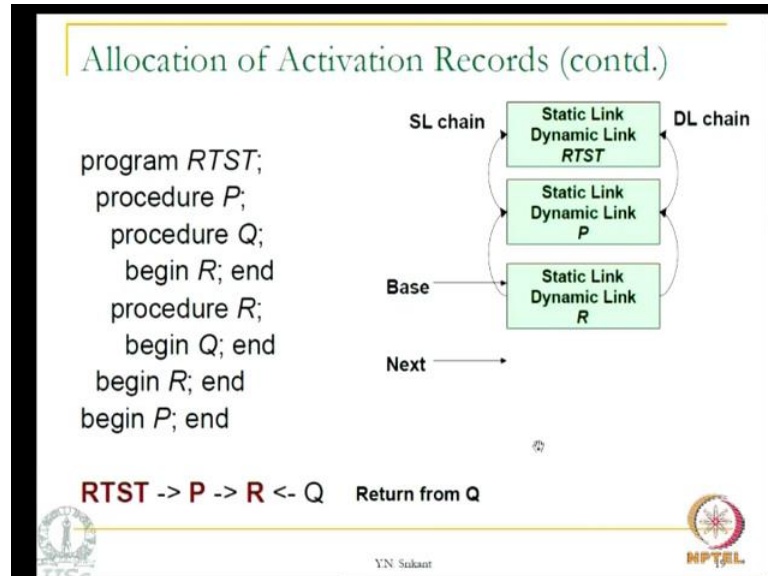
So, instructions must be generated to move this value into the static link field of the activation record. The offsets for all these fields are already known; so it is not they are all with respect to base pointer. So, the bases add base register will also be known, and therefore, generating these instructions is quite straight forward. So, let me you know state again so we really have to skip $L_1 - L_2 + 1$ records starting from the callers activation record, got to that particular record and then you know make that as the value of the static link for the Cali. So, from here we skip 1 and this is the static link value for Q. Similarly, from the caller we skip once, and that is the static link value for R etcetera.

(Refer Slide Time: 40:00)



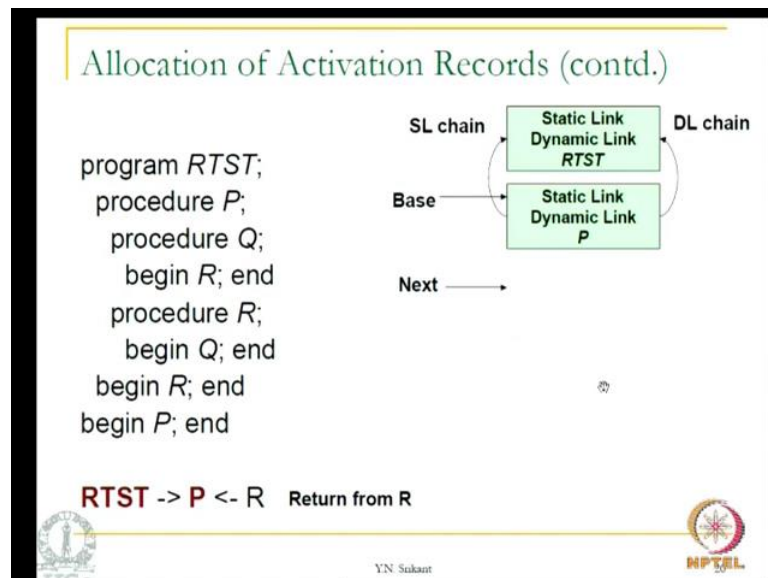
Now, the release of the activation records happens in the reverse order. So, we return from R so the activation record for R is released to the storage pool.

(Refer Slide Time: 40:17)



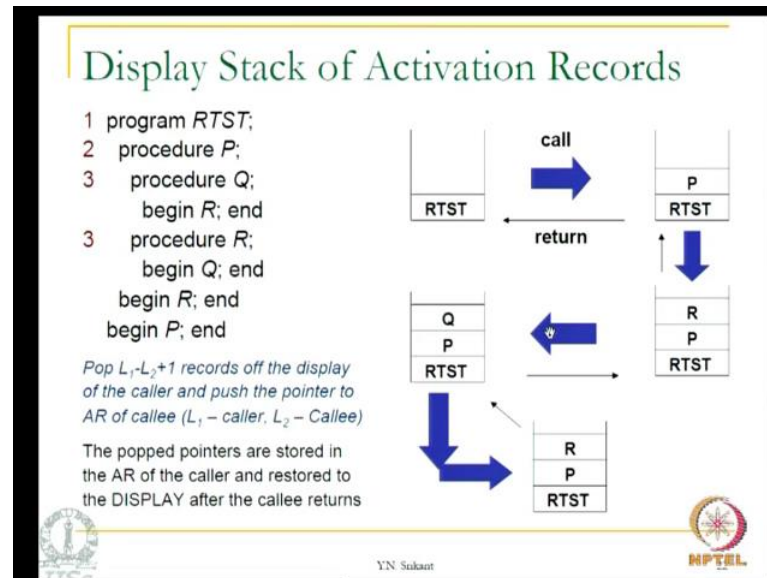
Then, we return from Q and the R of Q is returned.

(Refer Slide Time: 40:24)



Similarly, for R and P as well; once the main program terminates the storage required by the main program will also be returned to the storage pool.

(Refer Slide Time: 40:35)



So, what we have used so far, you know in our discussion was actually utilized the static link and the dynamic link. The static link was used to access the global variables in the various activation records. And the dynamic link was used to maintain the stack of activation records. This task can also be carried out using another data structure called as the display stack of activation records. So, instead of you know using a static link what we do here is to maintain a list of you know, a stack of pointers which point to the activation records of procedures which are right now executing.

For the same program that we saw here; the sequence is R T S T calls P, P calls R, you know R calls Q, Q calls R so this was the sequence we trace there. So, let us trace the same sequence here as well. So, to begin with the display stack contains a pointer to the activation record of R T S T alone. Now, there is no need for the static link, we have pointers of all the activation records of maintained on this stack. However, the dynamic link is definitely necessary to maintain the stack structure do the allocation and deallocation etcetera. R T S T calls P so the activation record pointers of R T S T and P are both on the stack; that of P is pushed on to this stack. P calls R. Now, we have 3 activation records which are present and R P and R T S T are also all 3 are on the stack. So, remember the most recent procedure which is activated its activation record pointer is on the top of the stack. Now, R calls Q so situation changes here; justifiably so because it is not correct to push the activation record of pointer Q on top of R here. The

implication of that would be the variables of all these would be accessed from all the other procedures as well.

So, the display stack structure must reflect the scope of the various functions and procedures appropriately. So, when we say R P and R T S T here the implication is the control within R can access the variables of R, the variables of P and also the variables of R T S T, and this reflects correctly the scope structure of the nesting of the program. So, you can see here R here, and P here, and R T S T here. So, when Q is called from R we must replace the pointer of R with the pointer of Q so we have only Q P and R T S T which again reflects in the nesting structure properly. So, Q P and R T S T so then what happened to the pointer R the pointer R was actually stored within the activation record of Q ok.

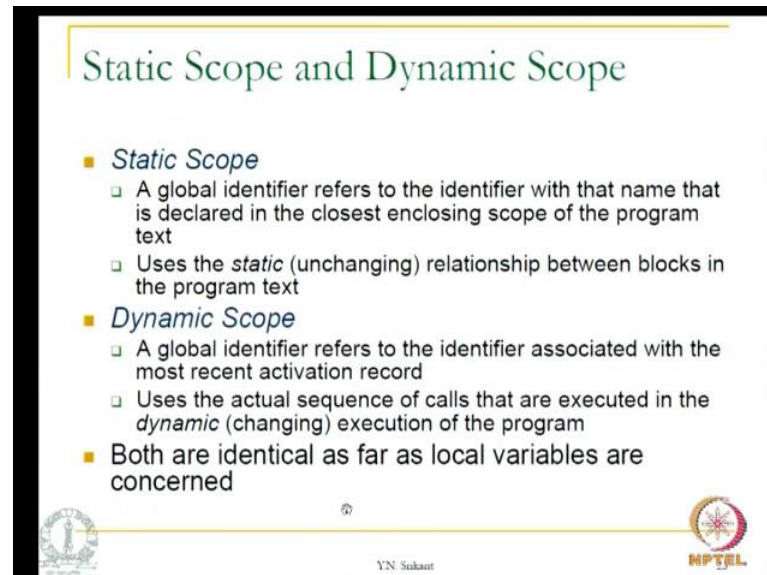
So, it would be unsaved once we return from Q. So, from Q we call R so again the activation record of Q is saved in the sorry; the pointer to the activation record of Q is popped from the stack. It is saved in the activation record of R, and the activation record pointer of R is pushed on to the stack so this is the stack structure which is very similar to the static link structure of the previous scheme. So, once we return the pointer of R is popped and the pointer of Q is unsaved from the activation record of R. The same thing happens when we return from Q, Q is popped and the pointer for R is unsaved, the same thing you know once we come here there is nothing to unsave the pointer of R will be thrown away.

And, here we return to the main program the pointer of P will be thrown away. So, once R T S T completes the display stack becomes empty. Here also the formula which can be used to pop a certain number of pointers from the stack and then display stack and you know save them in the activation record, the formula is the same $\text{pop } L_1 \text{ minus } L_2 \text{ plus } 1$ records of the display of the caller and push the pointer to the activation record of the Cali. L_1 is the caller and L_2 is the Cali. So, let us do that here to here.

So, this R and this is Q so $3 \text{ minus } 3 \text{ plus } 1$ is 1. So, we have popped 1 pointer from the activation from the display stack of at this stage so R goes away and we push the activation record pointer of Q on the stack, this R will be saved. So, the same thing holds here, as well both are at the same level. So, 1 pointer is popped here and the other pointer

is pushed on to the stack. So, the popped pointers are stored in the activation record of the caller, and are restored to the display of after the Cali returns.

(Refer Slide Time: 46:37)



Static Scope and Dynamic Scope

- **Static Scope**
 - A global identifier refers to the identifier with that name that is declared in the closest enclosing scope of the program text
 - Uses the *static* (unchanging) relationship between blocks in the program text
- **Dynamic Scope**
 - A global identifier refers to the identifier associated with the most recent activation record
 - Uses the actual sequence of calls that are executed in the *dynamic* (changing) execution of the program
- Both are identical as far as local variables are concerned

YN Srikant

NPTEL

So, that brings us to the next concept in runtime environments. So, we have so far we have studied you know activation records, how the activation records are really you know allocated de allocated etcetera. So, in the just to add 1 extra point when we have a C like structure for the programming language; that is no nesting of any procedures within another. The situation is very simple, because no procedure can access the variables of another procedure. They are all similar to Q and R. So, there is no need for you know the static link structure at all. We access either the variables of the self procedure or the global variables. Global variables are all in a particular static area.

And, the local variables of the currently active procedure are in the activation record. So, I do not need any static link I only need 2 links; 1 to the beginning of the activation record, and another to the static area containing the global variables. The dynamic link structure of course, will be required to take care of the stack allocation and deallocation. So, far what we have seen is actually the static scope for a programming language. So, let us understand the terms static scope, the languages which we have seen far C, C plus plus, Pascal, java they all have what is known as static scope.

A global identifier refers to the identifier with the name that is declared in the closest enclosing scope of the program. And it uses the static or unchanging relationship

between the blocks in the program text. Static scope is also called as lexical scope. So, the nesting structure which is shown by the program like here. This is the 1 which is used by the static scope you know scheme. So, procedure Q is nested within P, procedure P is nested within R T S T. So, and as I already explained the variables are accessed from Q, the variables which can be accessed from Q R itself. And then those after procedure P and those after procedure program R T S T.

So, this nesting structure which is fixed and does not change you know dictates which variables can be accessed at various points in time. And that is information since it does not change can build into the code which is produced in the form of static links etcetera. There is another concept know as dynamic scope. So, functional programming languages actually use dynamic scope quite routinely. A global identifier refers to the identifier associated with the most recent activation record. And it uses the actual sequence of calls that are executed in the dynamic execution of the program. And both schemes are identical as far as the local variables are concerned. Now, it is time to take an example and understand what is dynamic scope? Because the text itself does not make it very clear.

(Refer Slide Time: 50:25)

Static Scope and Dynamic Scope :
An Example

```


int x = 1, y = 0;
int g(int z)
{ return x+z;}
int f(int y) {
  int x; x = y+1;
  return g(y*x);
}
y = f(3);

```

After the call to g ,
 Static scope: $x = 1$
 Dynamic scope: $x = 4$

x	1	outer block
y	0	
y	3	f(3)
x	4	
z	12	g(12)

Stack of activation records after the call to g



YN Srikant

So, here is a concocted program which looks like a C program. But it is definitely C, because C does not have dynamic scoping. So let us understand there are 2 variables x and y. Then, there is a function g which takes 1 parameter and returns x plus z. And

another function f which has a parameter y , another parameter another variable x , x is y plus 1. And it calls g y star x and returns that as value. So, here in the main program we call y equal to f 3. So, let us understand what happens if this were to be a simple C program, f is called so y becomes 3 and then here is the local variable x . So, x equal to y plus 1 will produce 4 y was 3. Now, we call g with y star x 4 into 3 is 12. So, we go to g , z is 12 and x which is here actually refers to the global variable x here. So, very this is 12 and this 1. So, we return 13 as the value, and 13 is returned to the main program as well and that is assigned to y .



This is the normal scheme as far as static scope is concerned in C. Dynamics so this is the activation record structure. Here, is the outer block main program x is 1 and y is 0 to begin with then we call f with 3 so the parameter y has 3 value and the integer variable x will have a value 4 at the time of calling g , because its assignment is already over. We call g with 12 so the variable z gets parameter z gets value 12. Now, if we had assumed C like structure; the x here always refers to the global variable, right. But if we assume a dynamic scope a non C like scheme the dynamic scope clearly says; the global identifiers clearly refers to the identifier associated with the most recent activation record. So, in other words; when we are in g here there is no problem, when we are in g the global variable x is 1 of the occurrences of x . And the local variable x within f is another occurrence of x , f has not terminated.

So, this value is very much alive you know when we have created this activation record; there is this instance of x within the activation record for f . And this instance of x within the activation record for the main program. Dynamic scope says start from the current block keep going upwards in the list of activation records; find the activation record which is very closest to z and contains an instance of the variable that we want. So, in this case as we go upwards; it is the local variable of f x which is the local of f that is relevant to us this is called dynamic scope. So, the value of x which is relevant at this point here is 4 instead of being 1, because it is the variable x within the function f . So, this beak this is 4 and this is 12, so we really return the value 16, this is dynamic scope.

(Refer Slide Time: 54:35)

Static Scope and Dynamic Scope: Another Example

<pre>float r = 0.25; void show() { printf("%f",r); } void small() { float r = 0.125; show(); } int main (){ show(); small(); printf("\n"); show(); small(); printf("\n"); }</pre>	<ul style="list-style-type: none">■ Under static scoping, the output is 0.25 0.25 0.25 0.25■ Under dynamic scoping, the output is 0.25 0.125 0.25 0.125
---	--

YN Srikant

Let us take another example; here is a variable R global variable 0.25 then there is a procedure void show which prints R then there is another function small which has a local variable R and assigned a value 0.25. And then show in the main program we have 2 sequences show small print f, show small print f. Let us see what is printed? Show is called so within show the value of R is obviously the global variable, because it has no global variables within itself, and there is no other function which is active. So, 0.25 is printed out and then we call small. So, there is a local variable R here and then show is called. But static scope says look at the static structure of the program so this variable R is not visible within show, it is again 0.25 which is printed out. Then you know we know we come the second line which is similar again 0.25 and 0.25 is printed out.

If we assume dynamic scoping for sure there is no difference because there is only 1 global variable which is visible, but once small is called this R is becomes visible. So this show now has a choice of either this R or this R this is the most recent activation record containing the variable R. So, it is 0.125 which is relevant to this show as well. So, this R is printed out as 0.125 the same is true for the second 1 as well. So, in dynamic scoping when we have a choice the most recent activation record is used and it is the value the variable which is present in that record is actually utilized. So, let me stop here, we will continue with the implementation of dynamic scope in the next lecture.

Thank you.